Utrecht University

Master's Thesis

Towards Explaining Neural Networks

Yunlu Chen
5684218

Internal Supervisors

Dr. A.J. Feelders
Prof. Dr. A.P.J.M. Siebes

Daily Supervisor:

Dr. E.J.E.M. Pauwels
(Centrum Wiskunde & Informatica)

August 13, 2017

Abstract

In recent years, *Neural Networks* (NN) and *Deep Learning* (DL) have achieved exceptional performance in a number of applications such as computer vision, natural language processing, audio recognition and machine translation. However, NNs as predictors are usually not interpretable in practice and the learning mechanism is theoretically not well understood yet by researchers. Therefore, NNs are known as "black boxes". To explain how NNs work, we perform different types of empirical analysis on trained models for a simple supervised classification task on one-dimensional signals, including analysis of hidden layer activations, visualization by gradient ascent, experiments on learning noise labels and measuring distance in the high-dimensional feature space, etc. In practice, NN models surpass the traditional signal processing methods in terms of performance on the task. For explanations on how NNs work, first we observe that this specific task can be interpreted directly from weights with some certain NN structures with limited expressivity; second, empirically NNs learn a smoothed first derivative extractor in this task, from which we suggest that NN models learn "principal subpatterns"; third, with measuring the inner- and inter-class distance of the data samples, we suggest that the behaviour of the networks that learn from real or structured data is to shrink the layer activation representation to a certain range of encoding for data samples in the same class with internal hidden layers, which differs from the behaviour of the networks in the abnormal case to fit random noise with brute-force memorization. The difference in network behaviour also provides a reasonable answer to the question why the over-parameterized NNs are able to achieve generalization power.

Contents

Chapter 1

Problem Statement and Brief Introduction to Neural Networks

Contents

1.1 Introduction

History and recent achievements of Neural Networks. Neural Networks (NN) are a group of learning models inspired by the connection of neurons in human brains. The research of neural network (NN) models has a long history of more than 70 years. The first predecessor of NNs is the *Perceptron* [1, 2], which was proposed in the 1950s. Afterwards, in the 1980s, some computer scientists with a background in psychology and neuroscience proposed the *Multi-Layer Perceptrons* (MLP) [3]. MLPs are a deep structure with hidden layer(s), which consists of multiple layers of neurons, instead of the single neuron Perceptron. Around the same time, a new technique called backpropagation to train MLPs was introduced [4]. However, after initial success, researchers neglected NNs in the 1990s in favour of more traditional and statistical models, as the latter outperformed NNs.

In the last ten years we have witnessed a revival of NNs, as new techniques have been developed to increase the depth and the performance of the neural networks. It is widely believed that the this re-emergence of NNs in the 21st century started in 2006, when Hinton et al. proposed a new algorithm to train the deep belief network (DBN), which is later known as *greedy layer-wise unsupervised pre-training* [5]. This work resulted in a better performance than other statistical learning models such as Support Vector Machines (SVM), which were dominant at that time. The term of *deep learning* (DL) has become popular since then. The next milestone of deep learning was in 2012, when the first modern CNN structure AlexNet [6] achieved exceptional performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Since then, the research and applications of deep NN models have developed explosively. Deep NNs have been applied in the fields such as computer vision, natural

language processing, audio signal recognition, and have become the dominant method in quite a few of these domains.

The recent achievement of NNs and deep learning are due to three main factors [7]:

- Scaling up of computation with GPUs.
- Massive amounts of data being collected and processed.
- Huge models with a massive number of parameters being used.

The first two factors are all related to the the growth of computation power, and the third factor is a consequence of the first two. However, as a result, the parameters of NNs are too numerous so that the connections inside the network are too complicated to be easily interpreted or understood, which makes them known as "the black boxes".

The problem of explaining neural networks. The problem of understanding NNs can be expressed from two perspectives, which differ from each other in concept but also overlap a lot:

1. Interpreting neural predictions. It is hardly possible to explain the results from the neural models. The internal information on how the prediction is made is not human-interpretable.
2. Understanding the essence or the mechanism of NNs. Although the optimization works well in practice, the reason why deep NN models are able to achieve high performance effectively and efficiently is not yet completely clear to researchers.

In application, the users usually have a strong need for interpretability. Interpretation of the results from artificial intelligence (AI) approaches, or the understanding of the way inputs relate to an output in a model, is a desirable property when applying NNs to real-world problems.

For some approaches such as linear models or decision trees, one is able to trace human-comprehensible information and understand easily how the output results come from the model. Therefore, these approaches are good for interpretability. However, the predictions made by NNs are much more difficult. Tracing how final predictions are produced while passing the inner layers of the network is usually too complicated to be interpreted. This lack of interpretability also means that it is usually hard to give an explanation for erroneous decisions taken by NN models. A relevant concept in this context are *adversarial examples* which are inputs to the learning model that an attacker has intentionally designed to cause the model to make a mistake [8, 9], which is especially common in NN models. As a consequence, NNs are usually not trusted in the fields demanding more on security or reliability, although they are often able to achieve outstanding performance.

The second perspective of understanding is of great interest to deep learning researchers because they are interested in the essence of the model and how better performance can be achieved. Deep NN models are greatly over-parameterized. Intuitively, such over-parameterized learning models are neither expected to achieve high performance because of over-fitting, nor efficient to be trained in acceptable time because of the huge parameter space that needs to be searched. Nevertheless, one finds empirically that in practice optimization of deep NN models is very effective and efficient. In addition, a number of learning methods and tricks empirically help to enhance the performance or the learning speed. But likewise the theoretical explanations for them are usually insufficient. Some related research work of this sort of interpretation is introduced in Section 2.3.1 and 2.3.2.

Outline of thesis This thesis is about explaining Neural Networks based on an analysis which is mainly based on empirical observations. We aim at both interpretation and understanding because the two aspects are highly related to each other. Unlike other research work that attempts understanding or interpretation from complicated real world application tasks which they have foundations on, we choose a relatively simple task on one-dimensional signals as the entry point of explaining NN models, in order to make use of the existing theoretical solution and the clear statistical properties of the data type.

We continue this chapter with an introduction to the basics of neural networks including the structure of multi-layer perceptrons (MLP) and convolutional neural networks (CNN).

In Chapter 2, some existing work on understanding and interpreting neural network models is discussed. Rule extraction is the most popular interpretation approach in traditional NN research. While visualization techniques are more concerned for state of the art architectures. The rest of the chapter shows some more recent work of understanding NNs from other perspectives including the resemblance of deep NN architectures and physical laws, the memorization argument and the semantic neurons.

In Chapter 3, the jump detection task is performed with NN structures including fully connected MLPs and CNNs. The performance overwhelms the traditional signal processing approach significantly. The weights and the hidden layer activations are explored, by which some of the structures are directly interpretable.

In Chapter 4, we also demonstrated that the activation is regarded as the spatial information in the layer activation space, in which the continuity of data is preserved.

In Chapter 5, results from the Gradient Ascent visualization suggest that the NN model recognizes more kinds of concrete patterns than the pattern that it has been trained on. Specifically, they recognize the main feature of a maximum first derivative in our jump detection task.

In Chapter 6, the network is challenged with the task to memorize noise labels. In exploring hidden layer features of networks, we conclude that the behaviour of the NN in memorizing random data and learning reasonably structured data is different.

Chapter 7 is the conclusion and discussion. We connect the empirical observations from all above content, from which we draw the final conclusion that gives the explanation to the question "how NN models work" as they learn principal subpatterns from the training samples, by which they achieve good generalization ability.

Main contributions We contribute by using our empirical study to take one step along the road to explaining how NNs work.

The most significant contribution of our work is to understand the internal layer behaviour of NNs from measuring the inner- and inter-class distance of the data samples. With the view of network layers as the mapping functions between high-dimensional layer activation spaces, empirically we suggest the internal hidden layer behaviour of the network is to shrink the layer activation representation for data samples in the same class. Moreover, the generalization ability of the over-parameterized model can be explained by this network behaviour. This behaviour only applies to the case of learning real or structured data, which differs from the abnormal case of fitting random noise data with brute-force memorization, which leads to lack of generalization ability.

The second contribution of our works is from gradient ascent visualization. Empirically NNs learn a smoothed first derivative extractor in this task, from which we suggest that NN models tend learn "principal subpatterns". The previous work on gradient ascent visualization for image tasks did not lead to a similar conclusion because in natural images the subpattern is not as pronounced as in 1-dimensional signals with simpler and clearer statistical properties.

Besides, we also observe that the specific task is able to be interpreted directly from weights for certain NN structures with limited expressivity. But it is a less general result in that such a structure is not ensured to exist for all tasks.

1.2 Perceptron: Minimal Unit in NNs

The most fundamental computational unit in a NN is called a neuron. Fig. 1.1 shows a neuron with three inputs x_1, x_2 and x_3, and one output $h_{w,b}(\mathbf{x}) = f(\mathbf{W}^T\mathbf{x} + b)$, where $\mathbf{W}$ is the weight matrix, b the bias term, and $f(\cdot)$ the activation function.

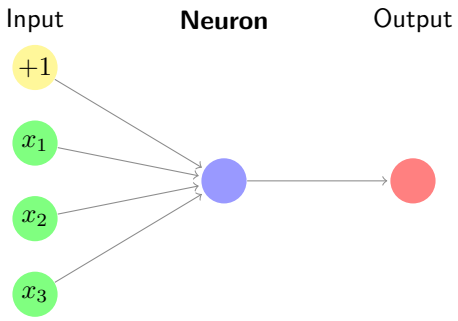


Figure 1.1: A single neuron (purple) with three inputs and a +1 intercept (or bias) term. The output is a single value $h_{W,b}(\mathbf{x}) = f(\mathbf{W}^T\mathbf{x} + b)$

The learning model with only one single neuron is called Perceptron[1], which was first proposed in 1950s. Originally, the very primitive Perceptron used a binary hard threshold step function as activation function:

$$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

as its activation function.

Traditionally, the tanh function $f(z) = \tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$ and sigmoid function $f(z) = 1/(1 + \exp(-z))$ were usually applied as the activation function in NNs. However, more recently, rectified linear unit (ReLU) activation [10]:

$$f(z) = \max(z, 0), \tag{1.1}$$

inspired by the action potential threshold in neuroscience, has become a dominant activation function in the application of NNs and deep learning. Compared to tanh and sigmoid activation, ReLU has two distinct advantages:

- less computation cost in back propagation.
- less sensitive to the gradient vanishing problem in deep networks.

Another useful activation function for classification tasks is the softmax activation which takes an n-tuple $\mathbf{z} = (z_1, z_2, \ldots, z_n)$ as input and produces a n-tupe $\mathbf{f}(\mathbf{z})$ as output for which the i-th component is given by:

$$f_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{1.2}$$

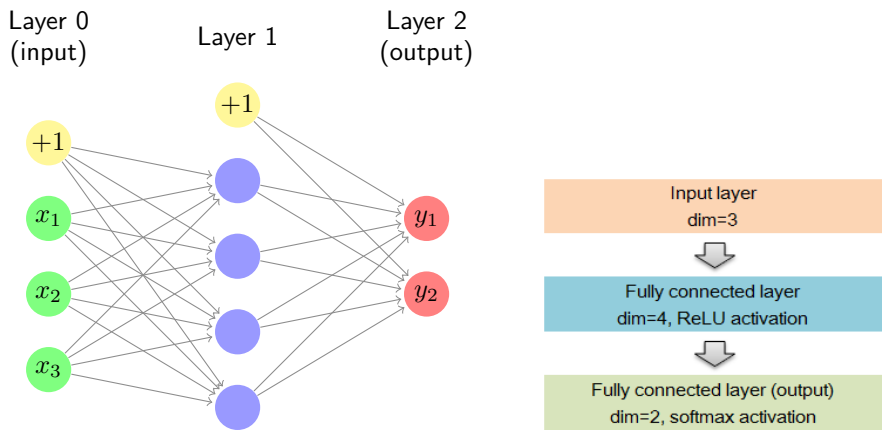
Compared to the other activation functions, the softmax function is different in that it takes all neurons in the layer into consideration, not only a single neuron. The result is therefore a multinoulli distribution with a sum of 1 to the output layer.

1.3 Multi-Layer Perceptrons (MLP)

1.3.1 Network Architecture

The Multi- layer perceptrons (MLPs) [3] are feedforward artificial neural network models consisting of layers of neurons. Each neuron in each layer contributes to all neurons in the next layer. Below in Fig. 1.2 is a simple case of a MLP with two layers. The input layer is regarded as the 0^{th} layer. Except for the input layer, the network has one hidden layer and one output layer, consisting of 3 input neurons, 4 hidden layer neurons and 2 output neurons. Fig. 1.2(a) is the representation in nodes. $+1$ nodes contribute to the bias term. The output neurons y_i are $h_{\mathbf{W},\mathbf{b}}(\mathbf{x})_i$. Fig. 1.2(b) is the structure

illustration. In the following chapters we use graphs like Fig. 1.2(b) to schematically represent the network structure.



(a) MLP illustrated with nodes

(b) Structure illustration by layers

Figure 1.2: A two-layer MLP with only one hidden layer, consisting of 3 input neurons, 4 hidden layer neurons and 2 output neurons.
(a) is the representation in nodes. $+1$ nodes contribute to the bias term. The output neurons y_i are $h_{\mathbf{W},\mathbf{b}}(\mathbf{x})_i$.
(b) is the structure illustration of layers. In the following chapters we use graphs like (b) to represent the network structure.

The activation $a_j^{(l)}$ of the j^{th} neuron in the l^{th} layer is

$$a_j^{(l)} = f(z_j^{(l)}) = f\left(\sum_k W_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right), \tag{1.3}$$

where $z_j^{(l)}$ is the weighted sum of inputs to the j^{th} neuron in the l^{th} layer, i.e., the value of the neuron before activation. Practically, this equation is usually written in vectorized representation (the activation function $f(\cdot)$ also in the vectorized form):

$$\mathbf{a}^{(l)} = \mathbf{f}(\mathbf{z}^{(l)}) = \mathbf{f}(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \tag{1.4}$$

The bias $\mathbf{b}^{(l)}$ is a vector and the weight $\mathbf{W}^{(l)}$ is a matrix. For example, in the figure shown case, in the first layer $\mathbf{W}^{(1)} \in \mathbb{R}^{3*3}$ and in the second layer, $\mathbf{W}^{(2)} \in \mathbb{R}^{1*3}$. The final output of the network is $\mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x})$ is the activation of the last layer:

$$\mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \mathbf{a}^{(L)}, \tag{1.5}$$

where L represents the last layer of the network.

1.3.2 Back Propagation Algorithm

The most commonly used method for training neural networks is the back propagation algorithm, which uses a gradient descent approach of optimization.

In order to understand the algorithm, it is necessary to first introduce the cost function, which is the target of the algorithm to optimize.

Denote the cost function of a single training example $(\mathbf{x}, y)$ as $J(\mathbf{W}, \mathbf{b}; \mathbf{x}, y)$. For example, the squared-error cost function is

$$J(\mathbf{W}, \mathbf{b}; \mathbf{x}, y) = ||y - \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x})||^2, \tag{1.6}$$

Similarly, the cross-entropy cost function for one input data instance $(\mathbf{x}, \mathbf{y})$ is defined by

$$J(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y}) = -\left[\mathbf{y} \ln \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}) + (1 - \mathbf{y}) \ln(1 - \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}))\right], \tag{1.7}$$

where $\mathbf{h}_{\mathbf{W},\mathbf{b}}$ is the output of the network, which is the same as the activation of the L^{th} and the last layer $\mathbf{a}^{(L)}$. In this case the activation function is the softmax function. $\mathbf{y}$ is the corresponding desired output.

For a training step with N as the total number of samples fed into the network, which is the same as the batch size, the overall (average) categorical cross-entropy error is

$$J(\mathbf{W}, \mathbf{b}) = -\frac{1}{N} \sum_{n=1}^{N} \left[\mathbf{y}_n \ln \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}_n) + (1 - \mathbf{y}_n) \ln(1 - \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}_n))\right]. \tag{1.8}$$

The optimization is through gradient descent. Within the vectorized representation, the processes of back propagation are:

1. From the input $\mathbf{x}$ (denoted as the 0^{th} layer activation $\mathbf{a}^{(0)}$), for each $l = 1, 2, ..., L$, calculate $\mathbf{a}^{(l)} = \mathbf{f}(\mathbf{z}^{(l)}) = \mathbf{f}(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})$
2. Calculate the output error for the last layer L: $\delta^{(L)} = \nabla_{(\mathbf{z}^{(L)})} J(\mathbf{W}, \mathbf{b}) = \nabla_{(\mathbf{a}^{(L)})} J(\mathbf{W}, \mathbf{b}) \odot f'(\mathbf{z}^{(L)})$. Where $\odot$ denotes the element-wise product or Hadamard product.
3. Back propagate the error: For each $l = L - 1, L - 2, ..., 1$, calculate $\delta^{(l)} = ((\mathbf{W}^{(l+1)})^T \delta^{(l+1)} \odot f'(\mathbf{z}^{(l)})$
4. Update the gradient with the partial derivatives $\nabla_{(\mathbf{W}^{(l)})} J(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y}) = \delta^{(l+1)}(\mathbf{a}^{(l)})^T$ and $\nabla_{(\mathbf{b}^{(l)})} J(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y}) = \delta^{(l+1)}$

Stochastic gradient descent (SGD) is a stochastic approximation of the gradient descent optimization method for minimizing an objective function. Instead of learning the whole training data set in one training iteration, SGD only takes a certain amount of the data instances each time, which is known as a batch. The amount is usually called batch size.

1.4 Convolutional Neural Networks (CNN)

Generally speaking, natural data such as 2-dimensional images or 1-dimensional signals or time series are represented as discrete variables with strong local correlations: that is, each element in the data sequence tends to be strongly related to its contiguous elements. However, as feedforward MLPs treat each elements in the data sequence separately, the spatial information is no longer preserved in learning. Therefore, new network structures are proposed to retain the spatial information, of which the most fundamental ones are the Convolutional Neural Networks (CNN) and the Recurrent Neural Networks (RNN). The CNN structure is introduced in this section.

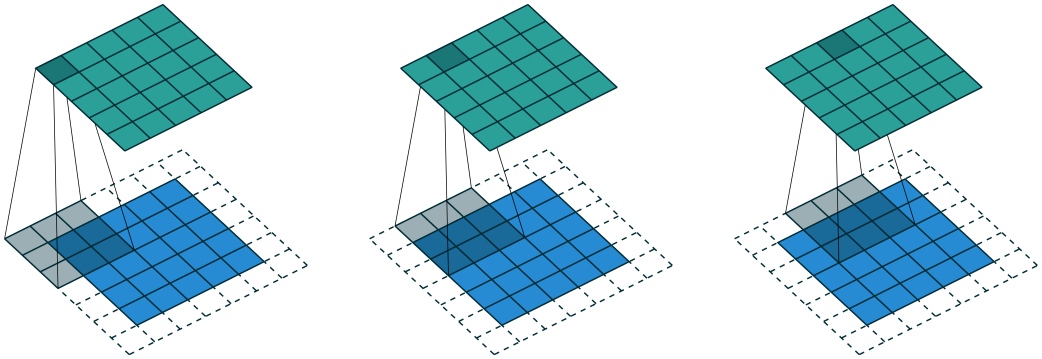
The *convolutional neural networks (CNN)* was first proposed in 1995[11]. LeNet[12] is an early CNN architecture that performs well in recognizing hand-written digits and characters. Modern CNN architectures, including AlexNet[6], VGG[13] and GoogLeNet[14], are much more capable to handle complicated tasks such as image classification on the huge ImageNet[15] data set.

The basic idea of convolutional networks is to select a local region of elements to be connected together in the next hidden layer. For images as inputs, the locality is a contiguous 2-dimensional region of pixels in the input. While for 1-dimensional signals or time series, the locality is the input elements in a clip of a certain time span.

A concrete example of 2-dim convolution manipulation is illustrated in Fig. 1.3 (with only the first three steps). The input size is 5×5 (blue area) and each grid element has a numerical value. The shaded area is the *convolution (conv) kernel* of size 3×3. The white grid elements are attached to the input of which the values are all 0. The technique is called *padding*, or more specifically this is a

case of *same padding* because after such a padding the size of output (green area) becomes the same as the size of input.

As illustrated in the figure, the convolution starts from the edge of the input (with the attached padding area as well). In each step of a convolution manipulation, the kernel is placed on the corresponding area of the input, and one element of the output is formed from the dot product of the conv kernel and the input area overlapped by the kernel. After one step, the conv kernel moves on to the next area of the input. The stride is distance between two consecutive positions of the kernel. A conv layer usually consists of multiple conv filters and each results in an output feature map.



(a) (b) (c)

Figure 1.3: Illustration of convolution. The conv kernel (shaded area) size is 3×3. Same padding is performed to keep the output dimension same as the input dimension (the area for padding is the white grids). The stride (distance between two consecutive positions of the kernel) is 1.
Extracted from https://github.com/vdumoulin/conv_arithmetic

Pooling is another useful technique in CNN structures, usually applied directly after (a sequence of) convolution manipulations. The main idea is to extract the most significant features and limit the feature space. Taking the most commonly used *max pooling* as an example, which in image tasks splits the image into non-overlapping 2-dim patches and chooses the maximum value of each 2-dim patch as output. Pooling, however will not be used for our task in the following chapters which requires an element-wise detection.

Below in Fig. 1.4 is the illustration of a fundamental CNN structure (LeNet-5[12]). After features are extracted from convolutions and poolings (noted in the figure as subsampling), the features are manipulated with fully connected layers to achieve the final outputs.

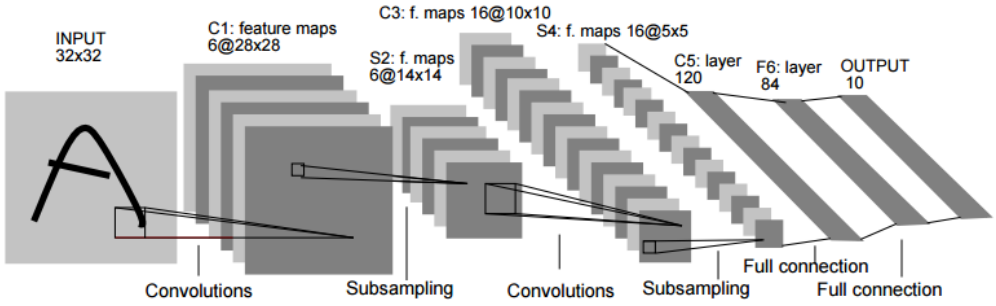


Figure 1.4: LeNet-5: a CNN structure [12].

Chapter 2

Related Work on Explaining Neural Networks

Contents

2.1 Traditional Approaches for Interpreting NNs

2.1.1 Overview

Attempts to interpret neural networks have been made since the 1990s. In the pre-deep-learning age, traditional approaches to NN interpretation included rule extraction, sensitivity analysis, and simulation. Rule extraction of the NNs takes the dominant position among all these approaches, since some algorithms in this class make it possible to inspect the network at the neuron-level, whereas other traditional approaches are only able to treat the network as a black box.

Rule extraction is the most common approach to realize interpretation in NNs with a long history. Therefore this approach is systematically reviewed in section 2.1.2 below.

Other traditional approaches, including sensitivity analysis [16, 17], and simulation [18], are also developed to interpret NNs by researchers in 1990s and early 2000s. These methods are similar, as they investigate the response of the output as the input varies. They are general interpretation tools for all kinds of machine learning algorithms, since they treat the networks as black-boxes and do not care about the inner structure or the dynamics. Moreover, as rule extractions, these traditional methods are insufficient for complicated tasks since they are unable to deal with high-dimensional inputs.

However, these old-style approaches have all been introduced prior to the rapid development of Neural Networks and Deep Learning in recent years. As a result, these approaches are usually limited and do not apply well to complicated tasks, e.g. the input and the output have to be low-dimensional data. In this chapter, rule extraction and visualization are introduced as two main categories of approaches to add interpretability and explanability to NNs. Some other approaches that bring insights to the NNs are also mentioned afterwards.

2.1.2 Rule extraction from NNs

Rule extraction approaches translate the prediction results from NNs to simple and human comprehensible rules. Most rule extraction algorithms deal with already trained NNs.

Categorization of the rule extraction algorithms

To have a review of the existing rule extraction approaches, it is helpful to know the classification of them. Generally a rule extraction algorithm can be categorized in terms of two different aspects: the forms of rules that the algorithms generates, and whether the inner structure of the network is considered.

The extracted rules usually take one of the three forms listed below (assuming the NN is designed for a simple supervised classification task).

- IF-THEN rules:

$$\text{IF } \mathbf{x} \in \mathbb{D}^{(i)} \text{ THEN } \hat{\mathbf{y}} = y_i \tag{2.1}$$

  where $\mathbf{x} \in \mathbb{D}$ is an instance (vector) of the input (or an intermediate neuron) of the network ($\mathbb{D}$ is the set of all possible x), and $\mathbb{D}^{(i)} \subseteq \mathbb{D}$ is a certain class of constraints that x is required to meet $\mathbf{x} \in \mathbb{D}^{(i)}$ is the condition, or the left part of the rule, and $\hat{\mathbf{y}} = y_i$ is the consequence, or the right part of the rule. If the condition is true, i.e., $\mathbf{x}$ is an instance in $\mathbb{D}^{(i)}$, then the output $\hat{\mathbf{y}}$ will be labeled as class y_i.

- M-of-N rules:

$$\text{IF } M \text{ of } N \text{ THEN } \hat{\mathbf{y}} = y_i \tag{2.2}$$

  which means if the condition that (at least) M of the N attributes are true is satisfied, then the rule predicts class y_i

- Sometimes rule extraction also generate decision trees as decision trees can also be represented with a sequence of (discrete or continuous) IF-THEN rules.

According to a widely used taxonomy [19], rule extraction approaches can be divided into two main categories. If the rule extraction algorithm considers the inner structure of the NN and works at the level of neurons, this approach is called **decompositional**. KT [20], one of the most famous rule extration algorithms, is a typical decompositional rule extraction algorithm that produces IF-THEN rules. It performs a layer-by-layer tree search for rules in the input and the intermediate neurons. FERNN [21] is a similar algorithm but it generates both IF-THEN rules and M-of-N rules. CRED [22] is a rather special case because it generates decision trees as rules. The decision tree is generated by the C4.5 algorithm [23]. Moreover, the author also develops a technique to simplify the rules. It reduces over-fitting, and as a result, the performance of the extracted decision tree is even better than the original Neural Network.

Alternatively, if the algorithm operates at the level of the whole network and irrespective of the NN's architecture, it is called **pedagogical**. RxREN [24] is a typical pedagogical algorithm that generates IF-THEN rules by reverse engineering (analyzing the output and tracing back components that cause the final result.). TREPAN [25] is another case that generates M-of-N rules by query and sampling. These pedagogical methods are faster than the decompositional methods, but they do not provide layer-by-layer investigation, thus the extracted rules are less transparent.

Remarks

According to Andrews et al. [19], there are two strong motivations for the use of rule extraction. Firstly, it provides insight on how the NNs use input variables to come to a decision. Secondly, it is mandatory to be used in safety-critical applications such as airlines and power stations.

However, these techniques come with serious limitations:

- Restrictions on network structure. Most rule extraction algorithms do not apply to deep Neural Networks, as they were developed in late 1990s or early 2000s, when no effective techniques to train deep and complicated networks existed. So they were designed for simple Multi-Layer Perceptrons (MLPs) with only one hidden layer. They do not apply to any other architectures such as Convolutional NNs or Recurrent NNs or NNs with more than two layers.
- Restrictions on the freedom of the input and output. Rule extraction is not applicable for complicated tasks with large input and output spaces. Since it is basically a symbolic approach which decomposes the NN prediction results into rules, a large input space (e.g. as is the case when processing collections of images), would result in enormous rules extracted from the network and consequently they would be uninterpretable.
- The extraction process often simplifies the model complexity, and as a consequence, it may lead to rules that do not accurately represent the original model [16].

These reasons make rule extraction ill suited for realistic applications with complicated settings. As a result, rule extraction is seldom applied in real world applications or state-of-the-art research on NNs or Deep Learning.

2.2 Understanding or Interpreting NNs through Visualization

2.2.1 Overview

In the recent research of NNs and DL, the networks are huge and deep, and the traditional symbolic methods are no longer able to interpret state-of-the-art neural models, since the generated rules are too numerous and too detailed for humans to understand. Instead, various visualization techniques are proposed as the most popular approaches to provide human comprehensible information about the networks. One obvious reason is that in recent years, most of the Neural Networks are designed for vision tasks, for which visualization is naturally a good solution. Another possible reason might be that visualization is the best way for humans to perceive and process massive and complicated data.

Most of the visualization techniques are therefore developed for vision tasks, although there also exist visualizations for natural language processing tasks[26, 27, 28], but these are much less well explored. Generally speaking, there can be three classes of visualization of NNs, which are discussed below.

Direct visualization of network architecture and neuron activations The first class of visualization techniques only visualizes the inner states of the NN, such as the connection of neurons or the activation value at the hidden neurons. Some of these tools have been turned into software or web applications to illustrate the network structure. For example, DrawNet [29] is a web application (see Fig. 2.1) to visualize object representations in CNNs, within an already trained network for image classification tasks. As one specific neuron is clicked, the software tool shows how this neuron is connected to other neurons in the upper or the lower layers, as well as, the receptive field of these neurons, which for CNNs in vision tasks usually means the area in the original input image that causes activation at this neuron. The most famous example in this class of tools is Google's TensorFlow Playground [30]. Unlike DrawNet which provides only static visualizations, TensorFlow Playground visualizes the dynamic of training, including how the weights and neural activations vary during the training process. Admittedly, due to the limitations of web applications, TensorFlow Playground only performs simple toy tasks. The works of visualization in natural language processing tasks [26, 27, 28]

are basically all in this class since they visualize dynamics or static states of neural activations in order to support human comprehension.

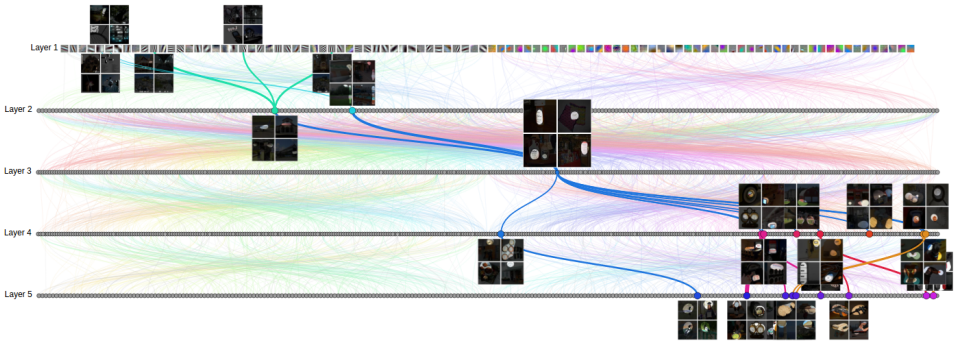


Figure 2.1: DrawNet[29]. After one neuron in Layer 3 is clicked, the visualization highlights the neurons that give the most contribution to this neuron in preceding the layers, and also the neurons that this neuron gives the most contribution to in the following layers.

Visualization from a specific given input The second class of visualization is based on a specific input, to find the response of the network either in the inner layers (Deconvolutional Networks [31], introduced in more detail in section 2.2.2) or in the input space [32, 33]. In [32] and [33], the basic idea is to measure the difference of the probability of prediction with and without a specific input pixel, as the measurement of how (positive or negative) this input pixel is related to the final prediction class. Fig. 2.2 provides an example: The red part in the input pixel space are the pixels that support the true prediction class "cockatoo", while the blue part of pixels work against this prediction since they refer to another confounding class.

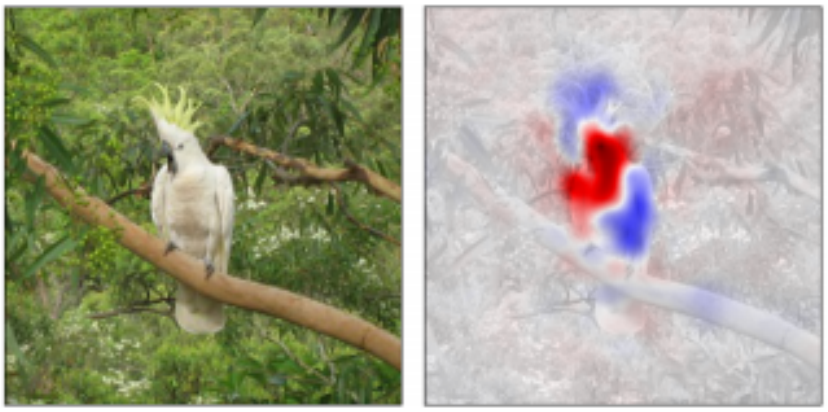


Figure 2.2: visualization by prediction different analysis[33]. The red part in the input pixel space are the pixels that support the true prediction class "cockatoo", while the blue part of pixels are against this prediction since they refer to another confounding class

Visualization through maximizing neuron activations The third class of visualization is with the idea to find an input image that maximally activates a specific neuron (in most cases it is an output neuron) for deep belief networks [34] and CNNs [35, 36](will be further introduced below in detail). They use some optimization technique, or more specifically, the gradient ascent, to obtain the maximum activation. In some other works [37, 38] similar results are obtained by generation models. Some intriguing findings are achieved through the latter kinds of visualization techniques. In the paper [38] it was first revealed how NNs are easily fooled, that is, some images which are non-sensical to humans, are nonetheless, confidently (with over 99.6% probability) but erroneously assigned to specific image classes.

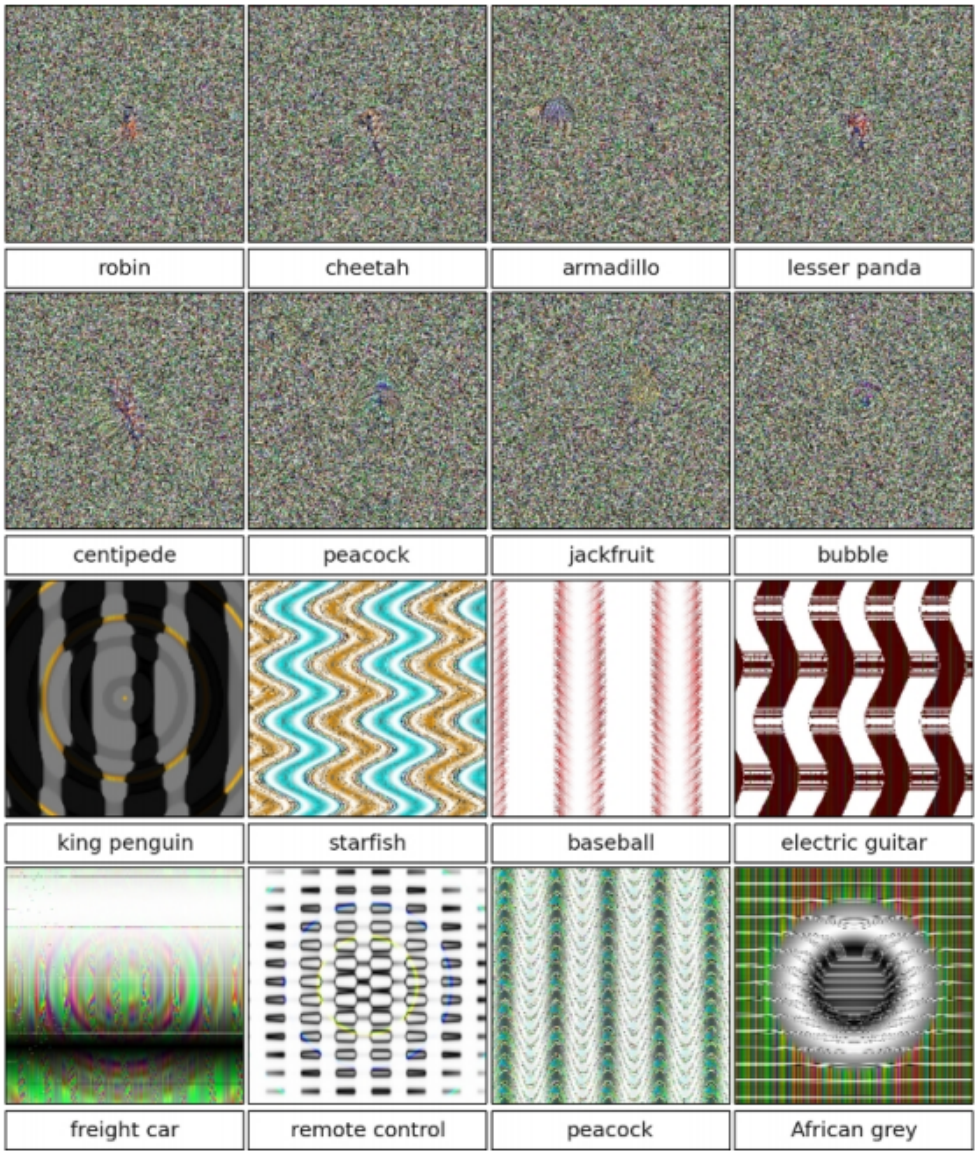


Figure 2.3: NNs are easily fooled.[38] These human-unrecognizable images are classified as various familiar objects with over 99.6% certainty.

In subsequent research, the kind of artificial inputs that cause the models to make wrong predictions are called "adversarial examples"[8]. Furthermore, from generative models the researchers can even generate adversarial examples that look like natural images but in the class other than the one that

NN model predicted [39], which indicates that the adversarial example can be used to attack deep learning models [9, 40]. As the NNs are black-box models, an attack by adversarial examples is difficult to defend against, which would probably cause severe security problems when deep learning is widely applied in everyday life.

2.2.2 Reconstructing input images with Deconvolutional Networks

Visualization with deconvolutional networks is one of the most famous visualization techniques [31].

Definition Deconvolutional networks are a powerful method with a wide range of applications. Initially, they were proposed for feature learning and convolutional sparse coding in unsupervised learning [41, 42], but the performance is not significant and it has the deficiency of large computational cost. The more widely known application of deconvnet is for CNN understanding and visualizing[31], which is also more related to our research theme to seek for interpretation in NNs. In recent years deconvnet is also applied in upsampling in FCN (Fully Connected Networks)[43] and generating images in DCGAN (Deep Convolutional Generative Adversarial Networks) [44].

The usual CNN procedure is to convolve a filter F on the input image I, in order to map the input image I into a hidden feature H (i.e. the feature map):

$$H = I \oplus F, \tag{2.3}$$

which is a discriminative model. In contrast, deconvnet reverses this process. Being more close to a generative thinking, the deconvnet obtains the original input by searching for a group of hidden features and then deconvolving a group of filters on them:

$$I = H \oplus F. \tag{2.4}$$

This is called deconvolution by the original authors since the convolution is from low dimension to high dimension. The deconvolution is later on also called transposed convolution, when a group of researchers noticed the ambiguity. In mathematics and signal processing, the word "deconvolution" actually denotes a different technique. But in this paper we still adopts the original term "deconvnet".

A single Deconvnet layer takes an image y^i as input, which contains K color channels and each color channel of the image is y_c^i.

$$y_c^i = \sum_{k=1}^{K_1} z_k^i \oplus f_{k,c} \tag{2.5}$$

where z_k^i are the latent feature maps and $f_{k,c}$ are the filters. There are K_1 latent feature maps in all and they are convolved with the filters. The cost function is defined as

$$C_1(y^i) = \frac{\lambda}{2} \sum_{c=1}^{K_0} || \sum_{k=1}^{K_1} z_k^i \oplus f_{k,c} - y_c^i||_2^2 + \sum_{k=1}^{K_1} |z_k^i|^p \tag{2.6}$$

in which the sparse norm $|w|^p = \sum_{i,j} |w(i,j)|^p$ and usually $p = 1$. Some techniques are applied to optimize this cost function but we omit introducing the concrete learning process.

Their later publication[42] adds new techniques such as unpooling to the deconvnet, which are useful in high level feature learning and visualization.

Visualization What we are interested in is how deconvnet visualizes CNNs. In this application the function of the deconvnet is only as a probe of an already trained CNN, and the deconvnet itself does not perform learning. In brief, deconvnet reconstruct features extracted from convolutional layers and pooling layers back to the pixel space of the input image. The network helps understanding the operation of a CNN by interpreting the feature activity in intermediate layers. More specifically, it takes the transpose of the conv kernels which are applicable in the feature maps learnt by CNNs. From such

a manipulation, the image features in the feature map space can be represented in the original pixel space. As a result, it becomes possible to discover which (combination of) pixels activated a certain feature map, which is helpful in order to analyze and understand CNNs.

The structure for CNN feature visualization is in Fig. 2.4, which consists only one convolutional layer and one pooling layer. In practice, usually multiple such layers are added together. A deconvnet is attached to each layer of a CNN. In the CNN (encoding) part, the input image turns into feature maps through a convolutional filter, and further feature maps are produced by adding rectified linear activation, and max pooling respectively. The corresponding deconv (decoding) process is basically a reverse. First the pooled feature maps reverse to unpooled feature maps by max unpooling. (In this unpooling process, a switch is necessary, which records the location of local max in the pooling (separate for each input image).) Then after a ReLU activation, the feature maps are projected to the original pixel space through the deconvolution. The deconvolution adopt information from the corresponding learned convolutional filter. If the convolution is processed through a matrix manipulation, the deconvolution is to multiply a transposed matrix. (So the deconvolution is sometimes called transposed convolution.)

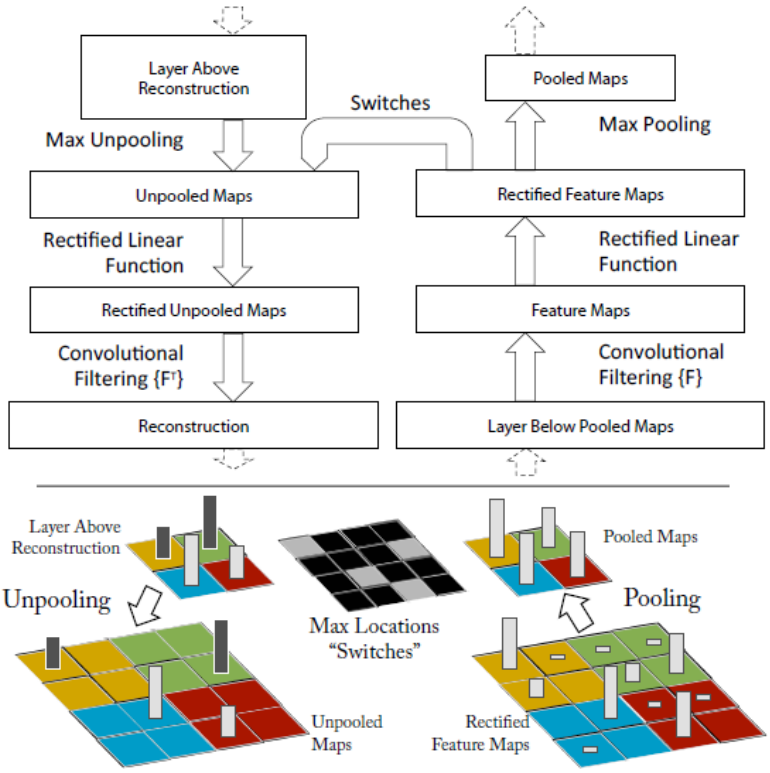


Figure 2.4: Structure of deconvnet for CNN feature visualization[31]. When pooling, the location of the maxima are recorded in the "switches" which can then be used to guide the unpooling.

In the experiment, five layers of the structure in Fig. 2.4 are represented together. In Fig. 2.5 9 strongest activations from the feature maps in the layers 2 to 5 are extracted and projected to the pixel space through the deconvnet approach (right to the reconstructed features are the corresponding original images). In a lower layer the receptive field of each activation in the pixel space is smaller than the activation in a higher layer. There exists hierarchical nature of the features in the network. Layer 2 shows corners or other edge/color conjunctions. Layer 3 is with more complicated textures. Layer 4 shows class-specific features such as dog faces and bird legs. Layer 5 shows the entire objects with significant pose variation.

Figure 2.5: visualization of single activations in deconvnet[31]

The deficiency of deconvnet as interpretation to NN is it can only project feature maps into pixel spaces in the convolution and pooling layers. However, the technique fails to give explanation to what happens to the neurons in the fully connected layers.

2.2.3 Gradient Ascent Visualization

This literature [35, 36] provides a different perspective to visualize deep NNs, which is in the third class from our categorization. The approach works for all kinds of network structures for classification tasks. In the literature the image classification task with deep CNN is studied as an example.

The main idea is to generate an artificial input. First, it starts with a random input image x. As the input to the network, it causes an activation $a_i(x)$ at some neuron i. The target is such an optimization problem to find an image x^* as a typical input that gives highest activation $a_i(x)$. Within a gradient descent framework, the x^* can be obtained from the original input x by iterations with

$$x \leftarrow x + \alpha \frac{\partial a_i(x)}{\partial x}, \tag{2.7}$$

where α is the learning rate.

Unfortunately in the experiment of image visualization, the approach failed to produce human-interpretable images (See e.g. Fig. 2.6). That's because these images were synthesized by optimization to maximally activate class neurons, but with no natural image prior (e.g. regularization).

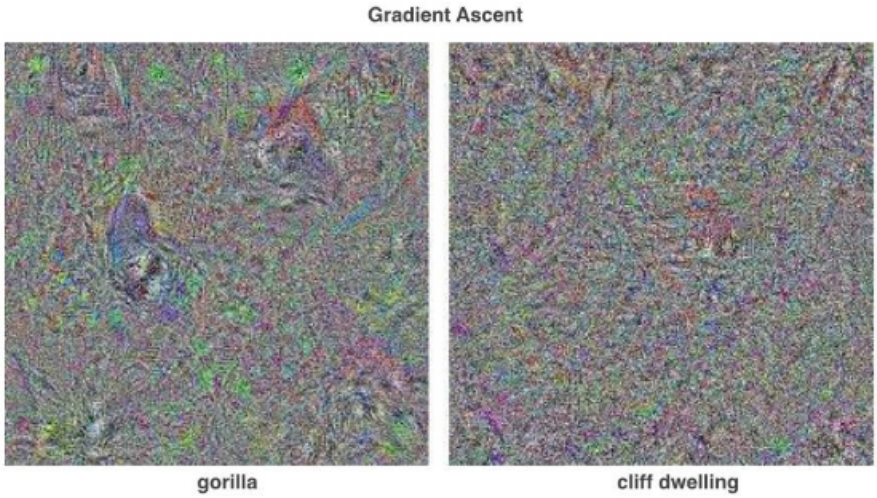


Figure 2.6: Deep visualization without regularization produces images that are not human-interpretable[35]

In order to make the result more interpretable and recognizable, various means of regularization can be performed. The regularization process in practice can be defined as an operator r_θ and the iteration formula needs to be modified to

$$x \leftarrow r_\theta(x + \alpha \frac{\partial a_i(x)}{\partial x}). \tag{2.8}$$

After introducing regularization (L_2 decay, which prevents the generated input influenced by extreme values of pixels), the images produced becomes more recognizable (e.g. Fig. 2.7).

In the literature [36] three more means of regularization are discussed, namely Gaussian blur, clipping pixels with small norm, and clipping pixels with small contribution. One conclusion is that the different regularization settings influence the representation of the visualization (in Fig. 2.8 four different regularization hyperparameter settings are illustrated). Some settings show the lower frequency patterns while some others are obviously more sensitive to higher frequency patterns.

2.3 Why Do NNs Work? Some Views and Insights on Understanding NNs

In the very recent two years, the problem of interpretability or explanability of NNs and DL has becoming popular among a growing number of people. Researchers are making attempts to achieve some insights of NNs from different perspectives.

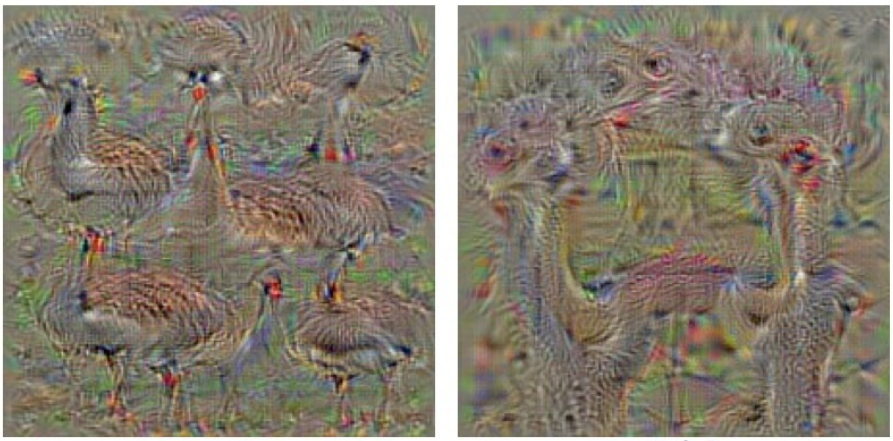


Figure 2.7: Deep visualization with L_2 regularization[35]

2.3.1 From the perspective of statistical physics: resemblance of physical world laws and NN structure

Some mathematicians and physicists are interested in NNs on higher level aspects as they attempt to reveal some essences or basic qualitative properties of NNs and deep learning. Lin & Tegmark [45] attempt to look for relationships between physics and deep learning. The authors claim that the outstanding performance of the deep learning at relatively low cost is due to some resemblances in the properties of the solution space of NNs and the physical laws, such as symmetry, low polynomial order and hierarchical processes. According to the authors, in NNs a "combinatorial swindle" is performed to bring expressability and efficiency to the NNs by limiting the exponentiation v^n parameters in theory to only the multiplication $v * n$ parameters, as n is the number of inputs with v values each. Generally the NN search for a probability $p(y|\mathbf{x})$ of class y given the input vector $\mathbf{x}$. With Bayes' theorem:

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{\sum_{y'} p(\mathbf{x}|y')p(y')}. \tag{2.9}$$

Then, with the representations $H_y(x) = -\ln p(\mathbf{x}|y)$ and $\mu_y = -\ln p(y)$ ($H_y(\mathbf{x})$ is referred to as **Hamiltonian** in statistical physics, which qualifies the energy of $\mathbf{x}$ given the parameter y), the equation is rewritten as

$$p(y|\mathbf{x}) = \frac{e^{-[H_y(\mathbf{x})+\mu_y]}}{N(\mathbf{x})}, \tag{2.10}$$

where $N(\mathbf{x}) = \sum_y e^{-[H_y(\mathbf{x})+\mu_y]}$. To take into consideration that usually multiple y's are learnt in the NNs, $p(y|\mathbf{x})$, H_y and μ_y are vectorized as $\mathbf{p}$, $\mathbf{H}$ and μ. Thus,

$$\mathbf{p}(\mathbf{x}) = \frac{e^{-[\mathbf{H}(\mathbf{x})+\mu_y]}}{N(\mathbf{x})}. \tag{2.11}$$

In classification problems, usually the softmax function is applied as the activation function of the last layer. As a result, the last equation can be represented in a simpler form:

$$\mathbf{p}(\mathbf{x}) = f[-\mathbf{H}(\mathbf{x}) - \mu], \tag{2.12}$$

which means that the ability of NNs are equivalent to computing a Hamiltonian vector $\mathbf{H}(\mathbf{x})$, and μ is the bias term.

Then, the authors explained why the Hamiltonians can be approximated by NNs. The Hamiltonians can be expanded as a power series $H_y(\mathbf{x}) = h + \sum_i h_i x_i + \sum_{i,j} h_{ij} x_i x_j + ...$ With a machinery using

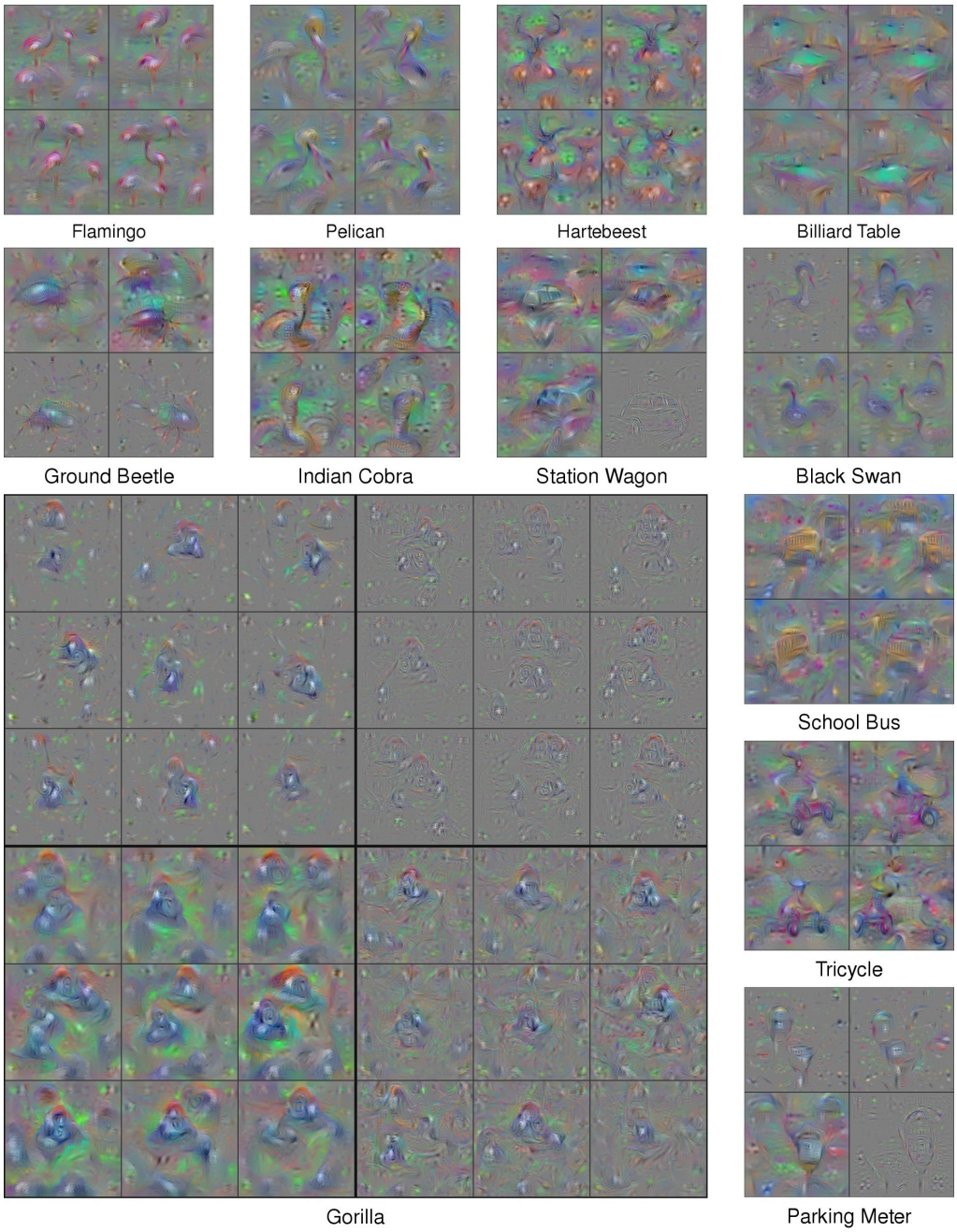


Figure 2.8: Deep visualization with regularization in eight layers [35]

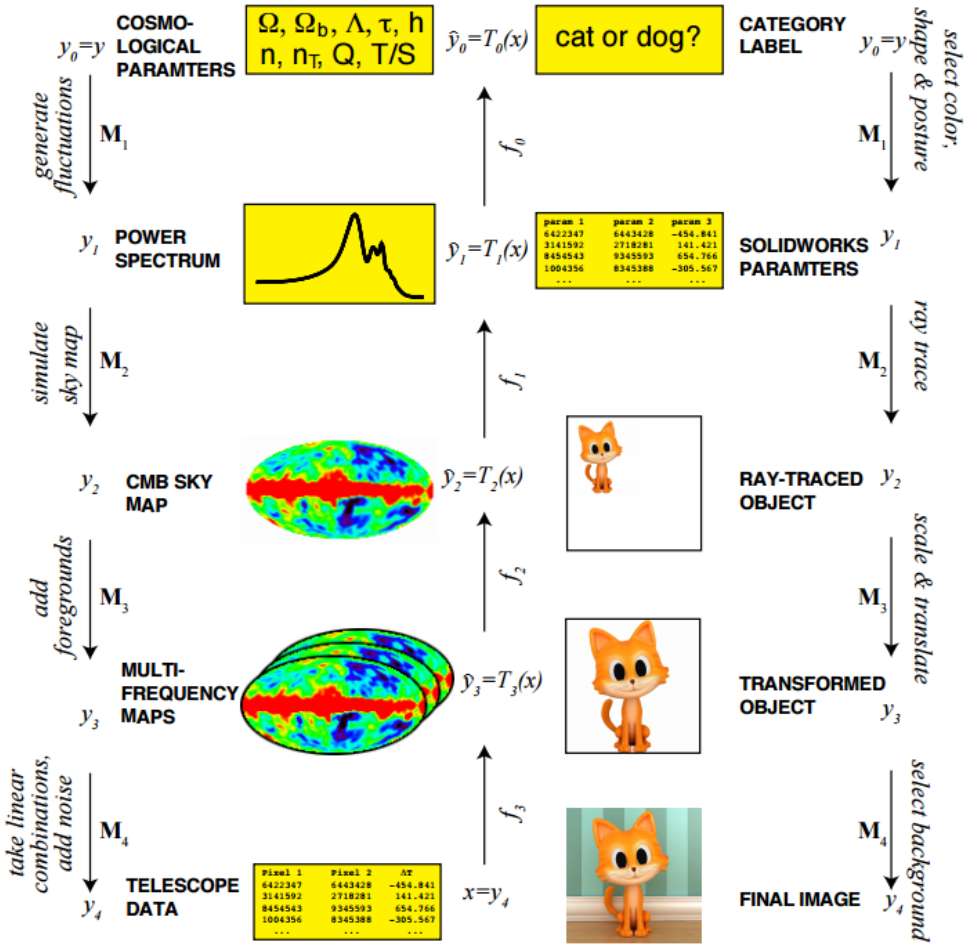


Figure 2.9: Examples of hierarchical structures. Left is a case in physics, and right is a case in image classification. The natural hierarchy goes through $y_0 \to y_1 \to ... \to y_n = y$. [45]

the logistic sigmoid activation function $f(x) = 1/(1+e^{-x})$, the multiplication can be achieved with a limited number of neurons. Furthermore, some other real world properties also limit the difficulties to approximate the Hamiltonian. First, the size of NN needed is bounded by the complexity of the very **low order of polynomials** (ranging from 2 to 4). Second, the **locality** is a principle in physics that objects only directly affect what is in their immediate vicinity. Third, the **symmetry** of the real world further limited the Hamiltonians that obey this rule. In conclusion, the authors claim that the number of continuous parameters in the Hamiltonian is reduced to only 32, with the constraints on locality, symmetry and polynomial order.

Afterwards, the authors argue that NNs being deep is both natural and efficient. For the discussion of deep networks being natural choices, in physical world, the very complex systems are composed of a hierarchical structure (such as particles, atoms, molecules, cells, organisms, planets, etc.). As illustrated in Fig. 2.9, the hierarchies of generative processes are created through a distinct sequence of simpler steps, where the probability distribution in one hierarchy is determined by its causal predecessor, which can be modelled by Markov chains. This discussion also indicates the meaningfulness of the **intermediate feature representations** in NNs. For the discussion of deep networks being efficient, the authors argues that if a flat network with one hidden layer needs 2^n neurons for multiplication of n variables. By contrast, a deep network needs only $4n$ neurons with $\log_2 n$ layers.

2.3.2 From the perspective of experimentation: "memorization hypothesis" of deep learning

Besides the research work from the theoretical perspective of physics, a widely accepted intuition shared by many deep learning researchers is that actual data (both training and test) that represent information in the natural world, actually lie on a very thin manifold in the high-dimensional (parameter) space. This argument applies for realistic or real-life data because they are under the constraint of some natural properties. But it should not necessarily hold for non-structured noise data. However, this intuition has been challenged by some recent experimental work. Indeed, recent work by Zhang et al.[46] which attempts to reveal some insights of NNs and DL from an experimentation perspective, has caused widespread concerns and brought great controversy within the deep learning community.

The main contribution of the work by Zhang et al.[46] is to show that deep NN models easily fit random noise data in practice. The authors perform some experiments of randomization, such as blurring the labels (shuffling the labels randomly) or blurring the samples (shuffling the pixels). Surprisingly the training still works until the model has completely memorized the noise data with 0 training error. As the data are more heavily blurred, the training takes more steps to converge, and the generalization error (which means the difference between the errors in the training set and the test set) grows in memorization with brute-force.

From another perspective, the traditional statistical learning suggests that the generalization ability is highly related to low-complexity of the model with the consideration to prevent over-fitting. (e.g. the cost complexity pruning in decision tree learning measures the model complexity with number of nodes as a penalty term in the cost function.) However, with the fact that the model complexity of deep NNs is extremely high, which is proved in the experiment of memorizing noise data, the low generalization error of deep NNs in learning natural or structured data (even without regularization techniques performed) becomes mysterious and cannot be explained by the statistical learning theory.

A widely accepted intuition is that regularization works to limit the hypothesis space in deep learning models so that a limited hypothesis space is expected to be unable to fit random labels. However, from empirical results with several implicit and explicit regularization techniques, the regularized models still fit random labels and the generalization error is not significantly reduced. Thus, the role of regularization cannot explain the paradox of memorization and generalization in deep NNs very well either.

As a conclusion, the authors argue that brute-force memorization could be also a strategy adopted by deep NNs in learning natural or structured data, which we called the "memorization hypothesis". In addition, the true cause of generalization power of deep NNs is yet to be explored.

Although the arguments are controversial for insufficient proofs and lack of solid and convincing reasoning, the phenomena of memorization of noise data inspires researchers to re-evaluate the essentials of deep learning. In Chapter 6, we return to this question and report on our own experiments related to this question, on how they shed light on this "memorization hypothesis". By exploring characteristics of the hidden layers in the network, we conclude that the behaviour of the NN in memorizing random data and learning structured data is different. Thus, we argue that NNs are not performing brute-force memorization when learning natural and structured data.

2.3.3 From the perspective of neuron-level observations: neurons are sensitive to certain semantic concepts

While in the above experiments NNs are considered black boxes, the paper [47] performs experiments at the neuron-level. The authors have developed a technique called *network dissection* to detect **semantic neurons**, i.e. neurons that are more sensitive to a specific concept. For this purpose the authors prepare a dataset with pixel-wise labelling. The labels for each pixel are not unique, but in six conceptual categories which are *object, part, colour, texture, scene* and *material*. The activation of each neuron is detected in performing semantic segmentation, in order to find if there exist some neurons that are specifically good at segmentation of any concepts in the six categories. Practically this is judged by a measurement $IoU_{k,c}$ which means a data-set-wide intersection over union score, as the score of each neuron k in detecting concept c.

The method has been tested in some popular CNN architectures such as AlexNet, VGG and GoogLeNet. The result is that such semantical neurons exist in all these networks. Fig. 2.12 illustrates some neurons in the networks are sensitive to the concepts of house, dog, train, plant and airplane. These neurons are marked as semantic detectors for these concepts. In Fig. 2.10 the authors make a comparative research to detect the number of semantic detector neurons in different network architectures. And in Fig. 2.11 they also perform this technique in training progress, which indicates that during the training process, more and more neurons develop certain semantic properties, i.e., the network becomes more interpretable.

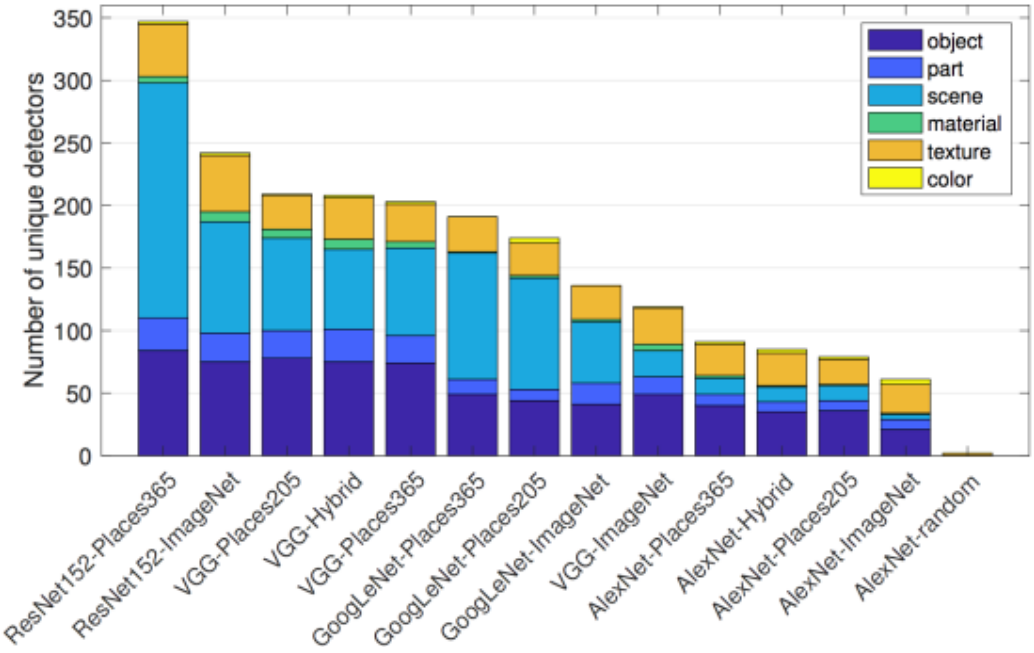


Figure 2.10: Number of semantic detector neurons in different architectures [47]

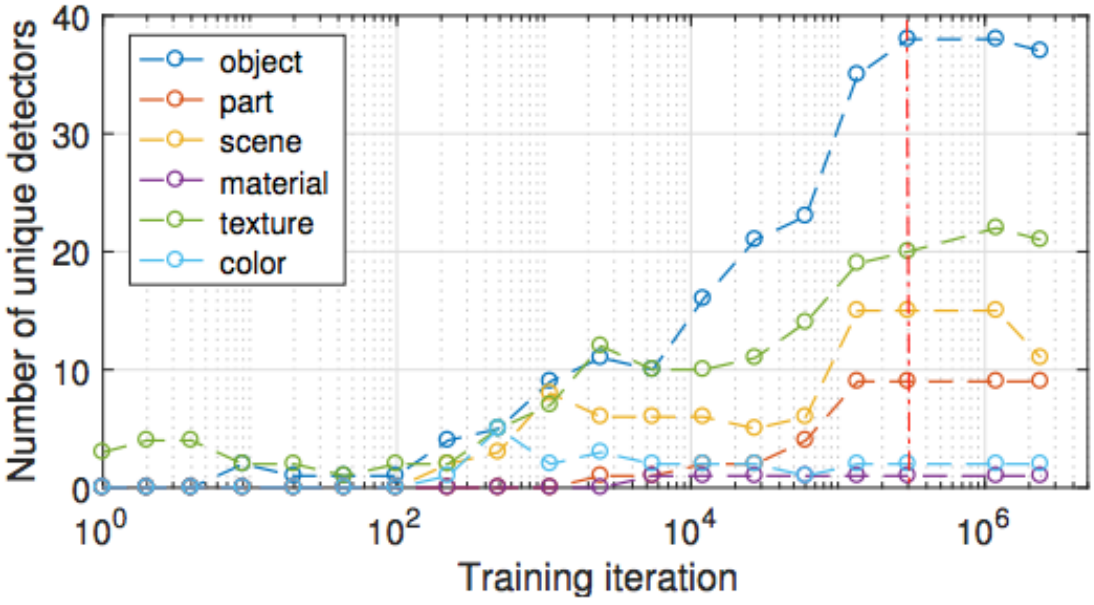


Figure 2.11: Growing number of semantic detector neurons in the training process [47]

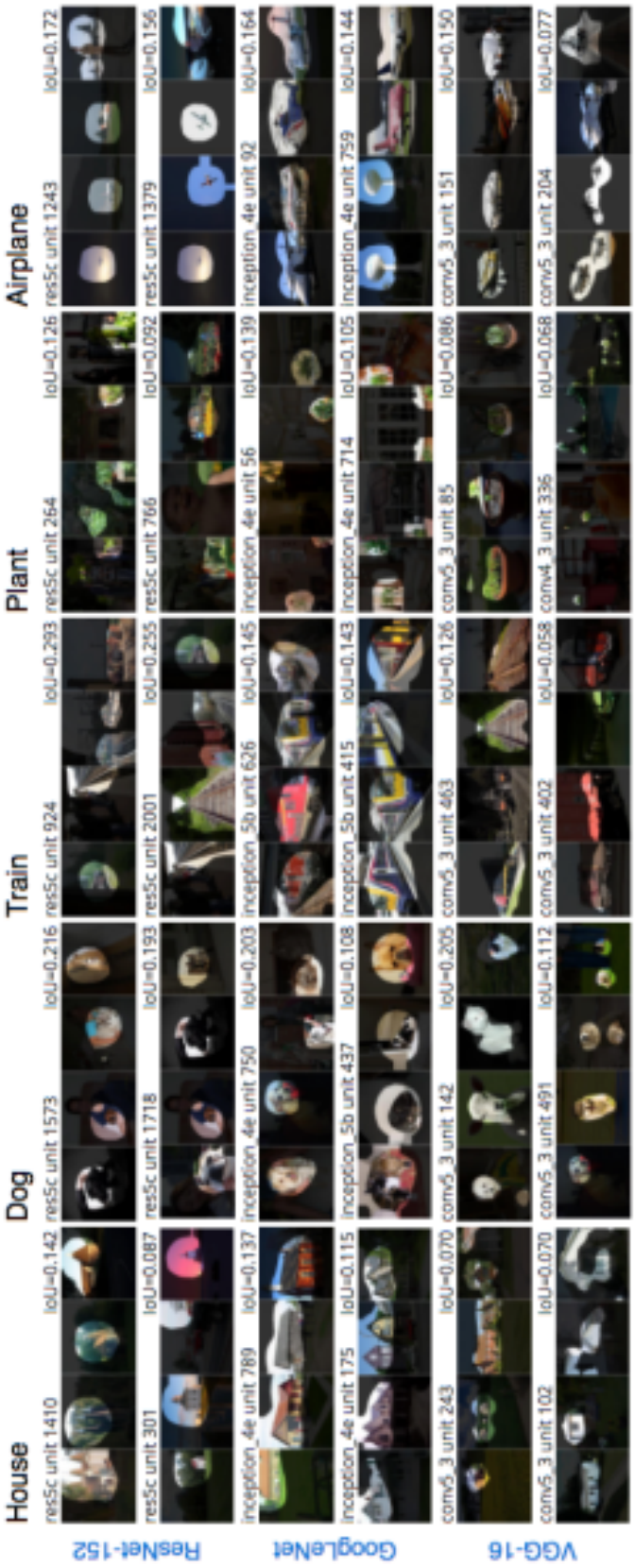


Figure 2.12: Semantic detectors for five different concepts: House, Dog, Train, Plant, Airplane [47]

Chapter 3

Detecting Discontinuities with Neural Networks

Contents

3.1 Introduction and Problem Description

Most recent work on deep learning and network visualization has been conducted on challenging image tasks. This work is highly non-trivial to reproduce as it usually requires access to huge collections of training data and computational power. We have therefore opted to restrict our attention to the simplest conceivable problem that has the same "image" flavour. We investigate how to locate a jump in a piece-wise constant but noisy 1-dim signal. This is basically a 1-dimensional version of the edge-detection problem in image processing, which is an important first step in computer vision.

First we will recast the problem into a precise mathematical formulation. Then we will spend some time on describing how classical signal processing would tackle this challenge. This solution will serve as a point of reference with respect to which we can interpret the NN-based solutions that will be the topic of the second part of this chapter.

General Formulation Assume that we have a noisy signal that jumps between two constant levels at a random time. More precisely, we have an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ for which there is an unknown $1 < t_0 < n$ such that:

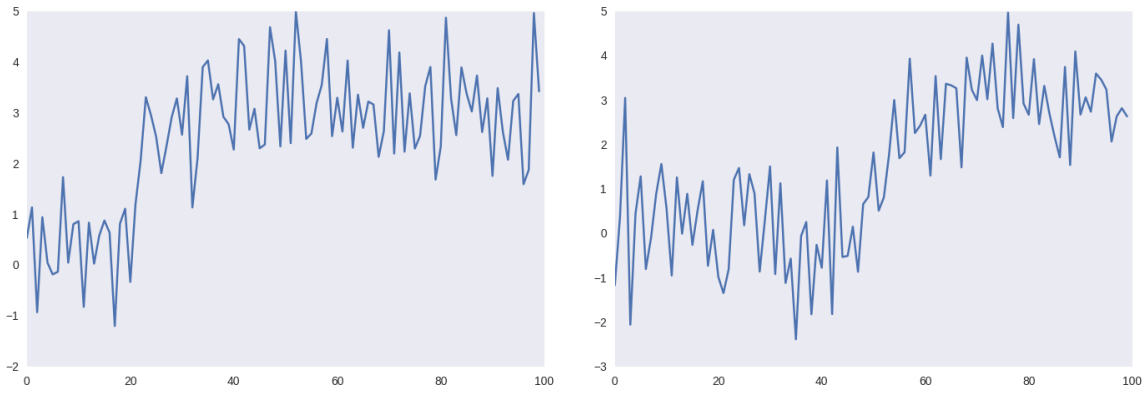
$$x_i \sim \begin{cases} N(0, \sigma_0^2) & \text{if } i < t_0 \\ N(\theta, \sigma_1^2) & \text{if } i \geq t_0 \end{cases} \tag{3.1}$$

In the above formulation, the difficulty resides in the fact that the parameters (jumping-time t_0, jump size θ, and (normal) noise levels σ_0, σ_1 are all unknown. Obviously, the problem becomes more challenging when the jumpsize becomes small with respect to the noise levels.

Simplified version To get started we focus on a simpler version of the problem. More specifically, we introduce the following simplifications:

- We assume the variance of the noise to be identical throughout the signal, hence: $\sigma_0 = \sigma_1 = \sigma$.
- Obviously, the difficulty of the problem is determined by the signal-to-noise ratio: the ratio between the jump size and the standard deviation of the noise (θ/σ). We can therefore fix the jump size (e.g. insisting on $\theta = 3$) and control the difficulty of the problem by varying σ (ranging from 0.5 to 2).
- As it is difficult (for both humans and algorithms) to spot a jump very close to the boundaries of the observed data window, we will restrict the possible jump-locations t_0 to points which are a minimum distance away from the endpoints. For instance, if the input signal has length $n = 100$ we will restrict the jump range as follows: $21 \leq t_0 \leq 80$.

The figure below shows two examples, in which the left Fig. 3.1(a) is a signal with a clear jump at the location $t_0 = 21$, There also exists some signals of which the jump locations are not easily detected. Fig. 3.1(b) is a case with a signal with a jump at the location $t_0 = 50$, but it is hard to tell this precise location.



(a) jump position 21 (b) jump position 50

Figure 3.1: Examples of signals.
For the left signal the jump location is easy to detect. For the right signal, it is hard to tell the exact location in a range between 45 and 55.

Experimental Setup We generate 12000 sample inputs (vectors of size $n = 100$) for the training data set, 6000 for the test data set, and 6000 for the validation data set. Hence, each input sample is a vector $\mathbf{x} = (x_1, x_2, \ldots, x_{100})$ where the individual components x_i are independent and adhere to the disribution specified in eq. 3.1. The same training and test data set serve for all experiments in this chapter.

Visualization

To have a first impression of our data, we performed visualization of our training dataset by t-SNE (t-distributed stochastic neighbour embedding) [48]. t-SNE is a machine learning algorithm for nonlinear dimensionality reduction. It first constructs a probability distribution over high-dimensional inputs to make similar objects have a higher probability of being picked. For a 2-dimensional visualization, t-SNE selects random 2-dimensional points and converts to a Student-t distribution. The KL (Kullback–Leibler) divergence of the two probability distribution is minimized through gradient descent, in order to force the random 2-dimensional points approaching the representation of the original high-dimensional data inputs. In the 2-dimensional map, the heavy-tailed Student-t distribution allows dissimilar 2-dimensional points far from each other. The visualization suggests that the input data are distributed along a low-dimensional manifold in the 100-dim input space.

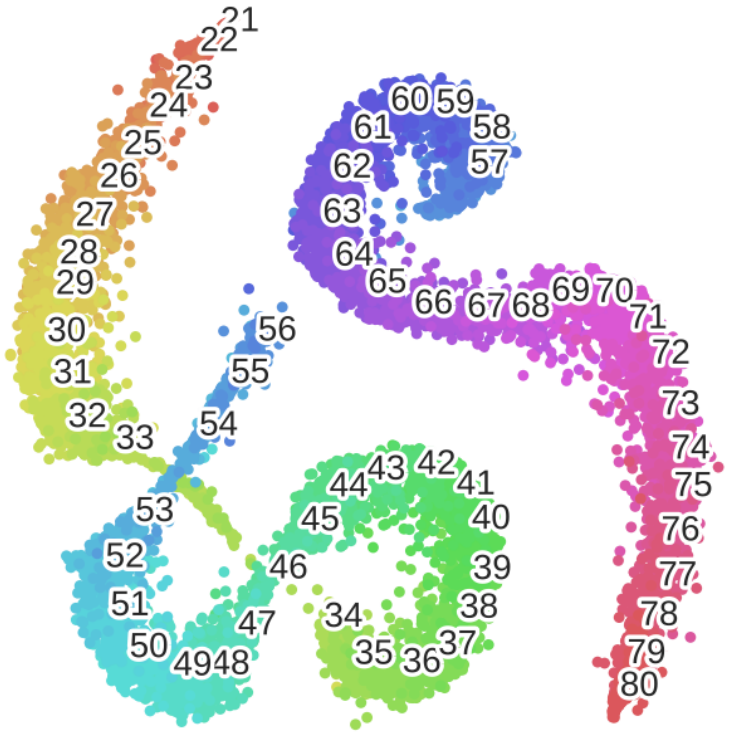


Figure 3.2: t-SNE visualization of training data.
The super-imposed numbers specify the jump-location within each sample (ranging between 21 and 80). Instances with each same jump-location are given the same colour in the plotting.
The visualization shows that the input data are grouped along a low-dimensional manifold in the 100-dimensional input space, which in turn suggests that the data are distributed continuously in the 100-dim input space the problem is learnable.

3.2 Signal Processing Approach

3.2.1 Introduction

The standard signal processing method to detect the jump point for a (noisy) but piece-wise constant function $f(x)$:

1. Reduce the noise in $f(x)$ by convolving with a (Gaussian) smoothing filter of characteristic width τ, e.g. g_τ. Note that width of the filter (τ) should not be confused with σ which quantifies the level of noise in the signal. The smoothed signal is therefore given by :

$$g_\tau * f(x),$$

where $*$ denotes the convolution manipulation, which is defined as the integral of the product of the two functions after one is reversed and shifted. More specifically, in this discrete convolution,

$$g_\tau * f(x) = \sum_{u=1}^{\tau} g_\tau(u) f(x-u), \tag{3.2}$$

where u is a dummy integer variable.

2. compute the partial derivative of the smoothed f:

$$\frac{\partial}{\partial x}(g_\tau * f)$$

3. the maximum of this derivative yields that most likely jump position.

Notice that due to standard properties of the convolution operation we can write

$$\frac{\partial}{\partial x}(g_\tau * f) = \frac{\partial g_\tau}{\partial x} * f$$

showing that the combined operation of smoothing and subsequent differentiation can be accomplished by a single convolution with an appropriate filter (the derivative of a gaussian filter). This will become relevant when we look into convolutional neural networks.

3.2.2 Theoretical Analysis

Consider the input signal specified in eq. 3.1. Notice that the fixed jump size θ can be fixed to 1, as it suffices to tune the noise parameter σ in order to vary the signal-to-noise ratio. Assuming that the filter is given by ϕ the result after convolution reads:

$$y(i) = \sum_k \phi(k) x(i-k)$$

We can now compute the expected value and the variance (recall that the x_i are independent):

$$E\left(y(i)\right) = \sum_k \phi(k) E\left(x(i-k)\right) = \sum_{k \leq i - t_0} \phi(k), \tag{3.3}$$

and similarly (since the noise variance is assumed to be constant and independent of the jump location):

$$Var\left(y(i)\right) = \sum_k \phi^2(k) Var\left(x(i-k)\right) = \sum_k \phi^2(k)\sigma^2 = \sigma^2 \|\phi\|_2^2 \tag{3.4}$$

where $\|\cdot\|_2$ represents the L_2 norm.

Smooting versus Differentiation Filters

For the interpretation of the network weights it will be helpful to make the distinction between two types of convolution filters.

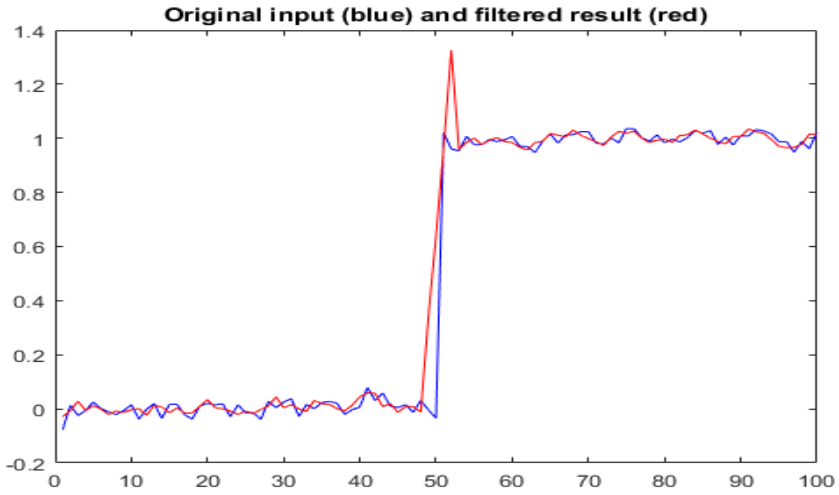


Figure 3.3: Result (red) of convolution of noisy step function (blue) by mixed filter $[1\,1\,1\,1\,-1]$. *This filter is the sum of a smoothing filter* $[1\,1\,1\,0\,0]$ *and a differentiation filter* $[0\,0\,0\,1\,-1]$.

Smoothing When all filter coefficients $\phi(k)$ are positive (or more generally: have the same sign), the filter ϕ is essentially a smoothing filter. This can be seen by observing that if we impose the normalization condition:

$$\sum_k \phi(k) = 1$$

(in order to ensure that the filter applied to a constant signal yields the same constant value), it then follows that (excluding trivial cases)

$$\|\phi\|_2^2 = \sum_k \phi^2(k) < 1$$

In combination with eq. 3.4 this means that

$$Var(y) < Var(x) \equiv \sigma^2,$$

which confirms that the filtered signal y is smoother (less noisy) than the original x.

Differentiation When some of the filter coefficients (but not all!) are negative, the filter incorporates some aspect of differentiation. In that case, the conditions

$$\sum_k \phi(k) = 1 \text{ (i.e. no amplification)} \qquad \text{or} \qquad \sum_k \phi(k) = 0 \text{ (i.e. no bias)}$$

no longer guarantee that $\sum \phi^2(k) \leq 1$ and the filtered signal might actually be noisier than the original. The prototype differentiation filters are of course:

- First derivative: $[1\ -1]$;
- Second derivative: $[1\ -2\ 1]$

Mixed filters More complicated filters can often be interpreted as a combination of smoothing and differentiation. This can be seen in Figs 3.3 and 3.4. The effect of the smoothing part on the slope of the jump is clearly discernible as well as the spike which is due to the differentiation part. We will encounter similar effects in Fig. 3.23.

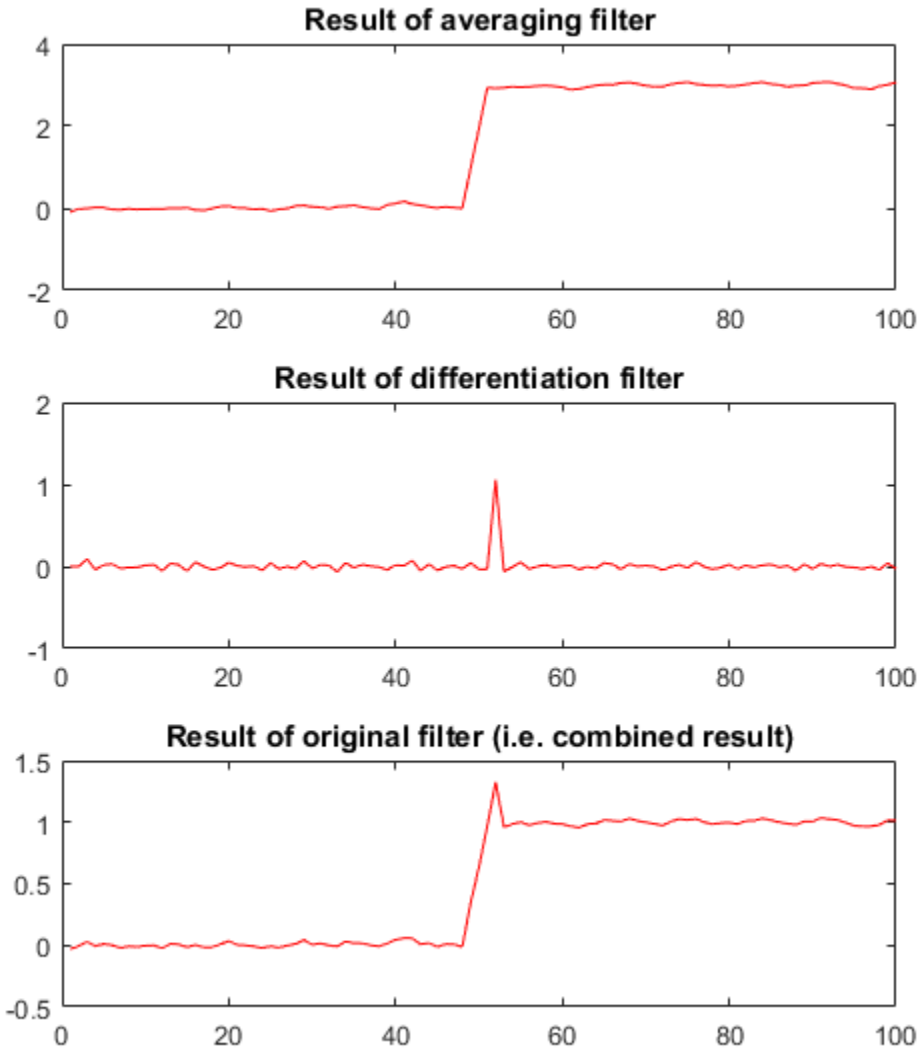


Figure 3.4: Since convolution is a linear operation, it is possible to separate the effect of the smoothing $[1\,1\,1\,0\,0]$ *(top) and the differentiation* $[0\,0\,0\,1\,-1]$ *(middle panel). Adding these two contributions yields the final result (bottom panel).*

Construction of optimal filter

Assuming that the filter ϕ is based on a Gaussian with standard deviation τ we can compute the above quantities and determine the optimal ratio of signal (Ey) to noise ($\sqrt{Var\, y}$). It is clear that two opposing effects are at work. If there are high levels of noise (i.e. σ is large), it pays to have a broad filter (i.e. larger value for τ) as this will result in more smoothing and a clearer filtered signal to determine the jump location. However, there is a price to pay as a broader filter makes accurate localisation more difficult. Hence, if the noise levels are low, it is better to choose a filter that is more narrow (i.e. smaller value for τ). Hence the optimal choice of the filter width τ is proportional to the noise level σ in the signal.

To illustrate this we computed the probability for accurate jump location estimation (absolute error at most 1) for different filter widths τ as a function of the signal noise levels $\sigma = 0.1, 0.2, \ldots, 0.6$ (see Fig 3.5). As expected, low signal noise means that it is advantageous to choose a narrow filter, as localization is then most accurate (with high probability). Increasing the noise has two effects: the optimal detection probability drops and the width of the optimal filter increases.

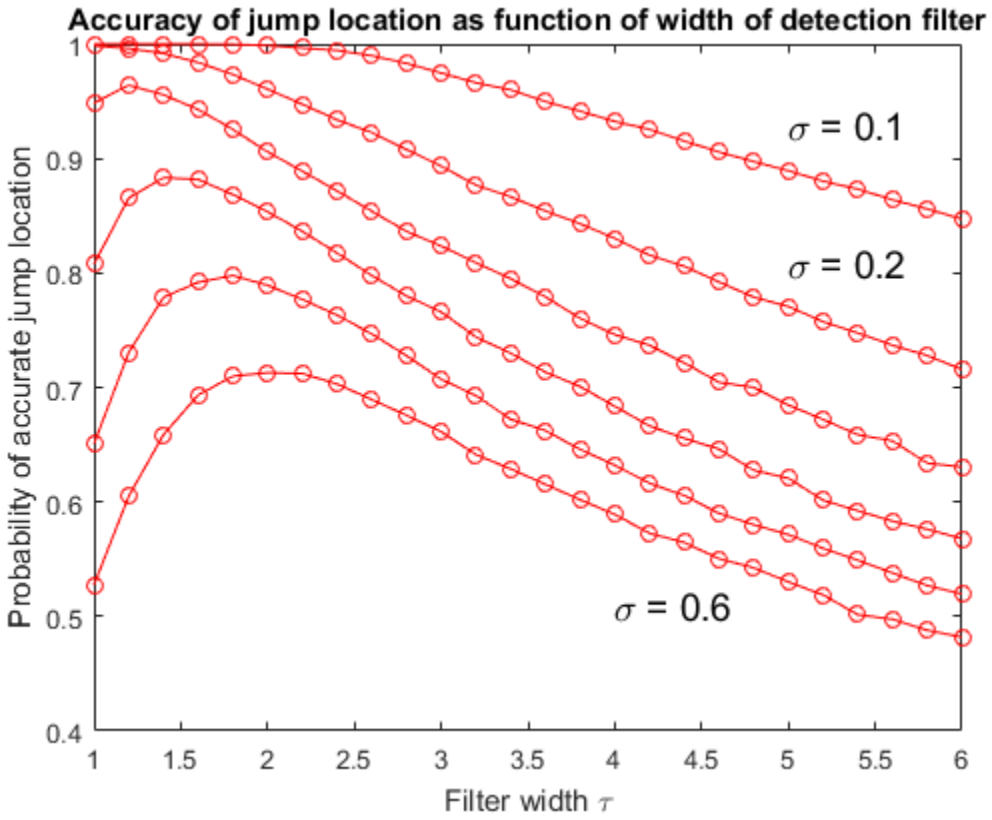


Figure 3.5: Optimal choice of filter width τ for accurate jump location (jump-size $= 1$) as function of signal noise level σ. The six curves represent noise levels $\sigma = 0.1$ (top curve) through $\sigma = 0.6$ (bottom curve) with increments of 0.1.

Notice that in the standard signalling approach we need to determine the filter width upfront. However, in a NN-approach the filter width is automatically determined by the training procedure, which therefore could be seen as a distinct advantage of a NN approach.

3.2.3 Evaluation on the test data set

Size of smoothing filter For the Gaussian smoother g_τ, we set window size $w = 19$, and $\tau = 2.9$ for which empirically we found the best performance in the training data set. (But also see Section 3.2.2.) The shape of this Gaussian smoother is shown in Fig. 3.6.

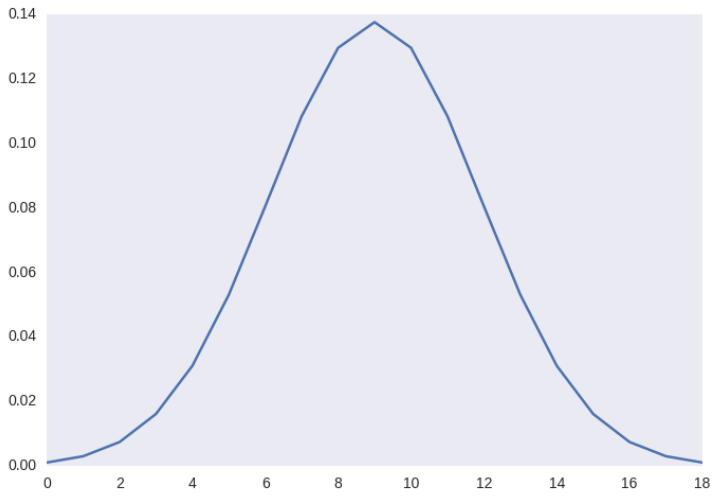


Figure 3.6: Gaussian smoother g_τ, with window size $w = 19$, and $\tau = 2.9$

In order to evaluate the performance, we define the first error rate for each signal as the absolute value of the difference between the predicted and actual jump position:

$$err_{L_1} = |t_{predict} - t_{true}|,$$

which is also known as the absolute error or the L_1 error. The histograms of the L_1 errors on the test data of 6000 data samples is illustrated in Fig. 3.7(a). Notice that most of the error values are 0 and 1, but there also exist (a few) high error values (the maximum is 58). These high error values, however, contribute too much to the overall error rate. Thus, we consider to count only the 99% data samples with lowest error, as the 99 percentile data. The error of the 99 percentile data is in Fig. 3.7(b), where all error values are at most 2.

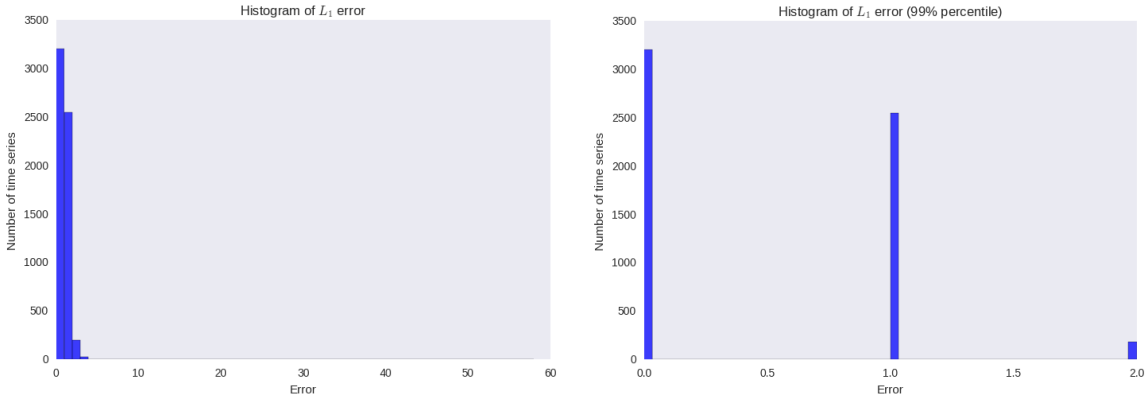
For the performance in the whole data set we look at the following two error measures:

- L_1 error: $\overline{err_{L_1}} = \frac{1}{N}\sum_t |t_{predict} - t_{true}|$ over all N data samples.
- L_2 error: $err_{L_2} = ||t_{predict} - t_{true}||_2 = \sqrt{\frac{1}{N}\sum_t (t_{predict} - t_{true})^2}$.

The four error rates are listed in Table 3.1, including the error rates in the 99 percentile of the data. The L_1 error slightly decreases and the L_2 error significantly drops after taking the 99 percentile of the data set.

	Complete data		99 percentile	
	L_1 error	L_2 error	L_1 error	L_2 error
Signal processing approach	0.5532	1.4946	0.4917	0.7443

Table 3.1: Performance of the signal processing approach



(a) Full data

(b) 99 percentile

Figure 3.7: The histograms of the L_1 error performed by the signal processing approach on the test data. On the left is the result with the all 6000 test data samples and on the right is the 99 percentile (the 99% data samples with lowest error).

3.3 MLPs for Detection of Jump Location

The problem described in section 3.1 can be modelled as either a regression problem or a classification problem. In a regression setting, the output is only one single neuron whose value represents the jump position. In contradistinction, in a classification problem, the output layer is the one-hot encoded vector for the jump position. For example, if there exists three classes in all, the second class represented by a one-hot encoded vector is $(0, 1, 0)$. In our case, there are 60 possibilities of jump positions in all, ranging from 21 through 80. The output layer has 60 neurons, and each neuron, with a value in the range $(0, 1)$, stands for the probability that the sequence has a jump at this position. Considering that the supervised classification tasks are most typical for NNs, and data sets designed for classification tasks, such as MNIST, CIFAR and ImageNet, are popular and widely investigated in NN and DL research, we choose to model the problem as a classification task.

3.3.1 MLP with one hidden layer

First, we consider the most simple network structure which is a two-layer MLP with one hidden layer and one output layer (Fig. 3.8). The input layer (regarded as the 0^{th} layer) has 100 neurons to represent a signal of length of 100. The first layer is the hidden layer. The number of the hidden layer neuron are set to be also 100, of which the consideration in design is that it might be helpful for interpretation to be consistent with the size of the input. The rectified linear activation $f(z) = \max(z, 0)$ is adopted in the hidden layer. The output layer, also the second and the last layer, has 60 neurons. Softmax function $f(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ is the activation function for this layer, since the task is modelled as a classification problem with one-hot encoding.

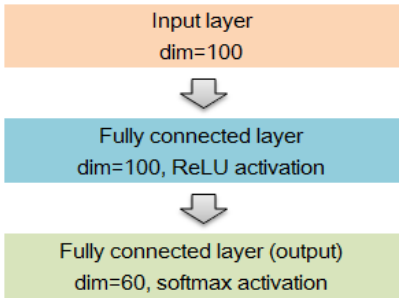
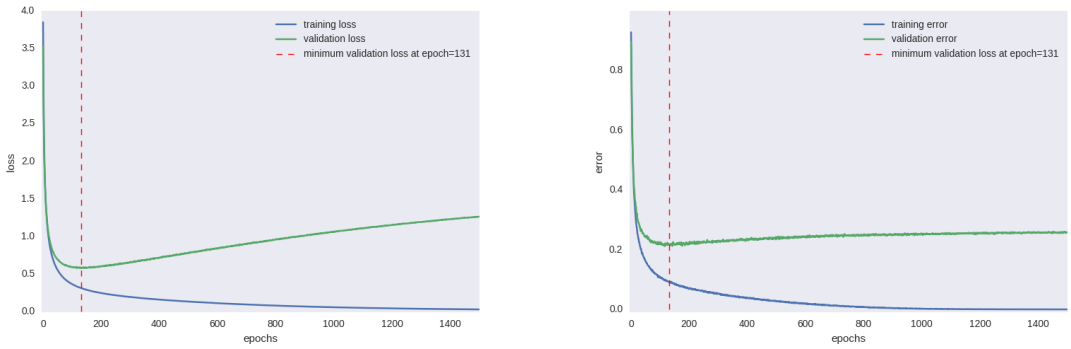


Figure 3.8: The network structure: MLP with two layers (the input layer is regarded as the zeroth layer).

As for details and hyperparameters in the training session, the optimization is through stochastic gradient descent (SGD) with no momentum. The learning rate is 0.01. The batch size for each step of training is 100. The cost function is categorical cross-entropy cost. For a classification problem, the categorical cross entropy error is usually defined as a classification because in comparison with classification error or mean squared error, cross entropy error brings convenience in gradient calculation[49]. The cross-entropy cost function is defined by

$$J(\mathbf{W},\mathbf{b}) = -\frac{1}{N}\sum_{n=1}^{N}\left[\mathbf{y}_n \ln \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}_n) + (1-\mathbf{y}_n)\ln(1-\mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}_n))\right], \tag{3.5}$$

(see section 2.2) where N is the total number of samples fed into the network in one step of training, which is the same as the batch size. $\mathbf{h}_{\mathbf{W},\mathbf{b}}$ is the output of the network, which is the same as the activation of the L^{th} and the last layer $\mathbf{a}^{(L)}$. In this case the activation function is the softmax function. The summation is over all N training inputs $\mathbf{x}$, and $\mathbf{y}$ is the corresponding desired output.

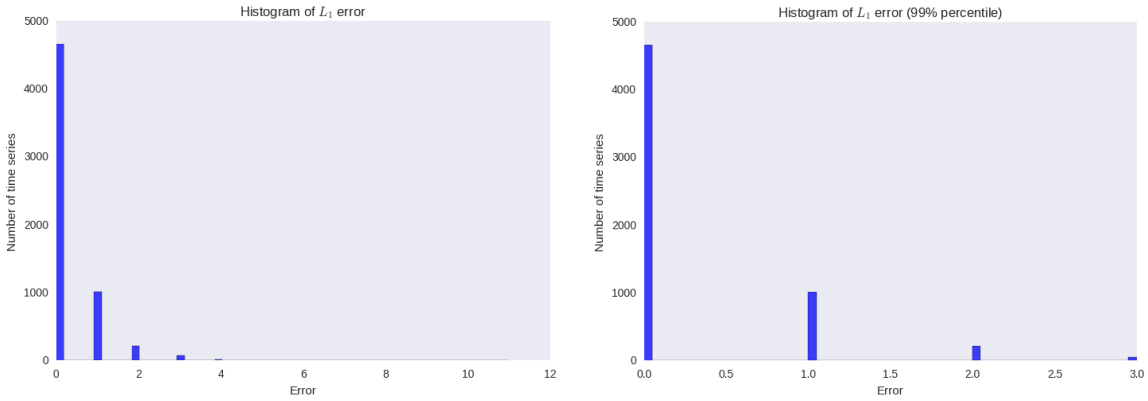


(a) Loss (b) Classification error

Figure 3.9: The training session: loss (left) and classification error (right). The metrics are recorded within both the training data set (blue) and the validation data set (green). The unit in the x-axis is not the iteration in one training step, but one epoch which means one iteration over all of the training data.

Fig. 3.9 illustrates how the loss and the classification error varies in the training process. During the iteration, the training loss keeps decreasing and the classification error drops to zero after 1400 epochs (one epoch means one iteration over all of the training data, not a training step). However, in a separate validation data set with 6000 data samples, the loss and classification error reaches their minimum values at epoch=131. Afterwards, the loss and classification error gradually increase again. Thus, seeking for the best performance on the unknown test data, in all later experiments, we adopt the model when the validation loss reaches the minimum. This technique is also known as *early stopping*.

After early stopping is adopted, the histograms of the L_1 error are shown in Fig. 3.10. The maximum absolute error is 12. Compared to the result from the signal processing approach (Section 3.2.3), the NN does not suggest outlying predictions with exceptionally high L_1 error.



(a) Full data

(b) 99 percentile

Figure 3.10: The histograms of the L_1 error performed by the 2-layer MLP on the test data. On the left is the result with the all 6000 test data samples and on the right is the 99 percentile (the 99% data samples with lowest error).

The performance (error rates) is listed in Table 3.2.

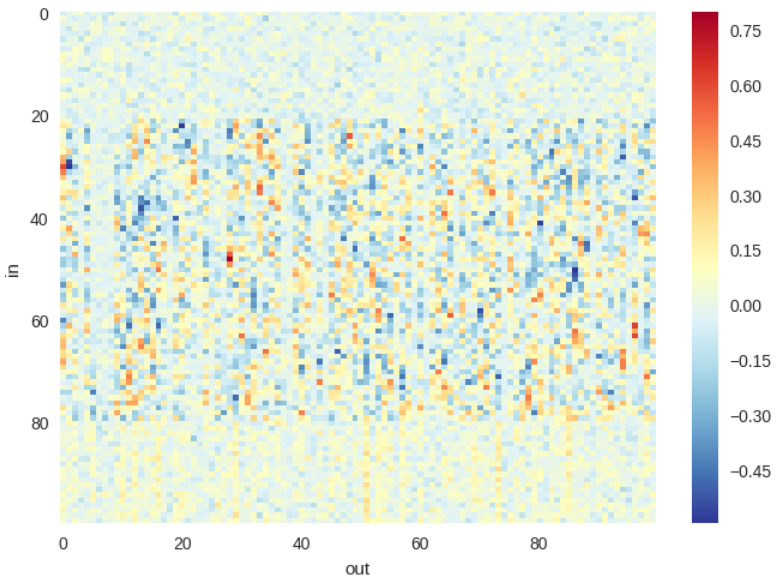
	Complete data		99 percentile	
	L_1 error	L_2 error	L_1 error	L_2 error
Signal processing approach	0.5532	1.4946	0.4917	0.7443
2-layer MLP	0.3083	0.7722	0.2675	0.6222

Table 3.2: Performance of 2-layer MLP, compared to the signal processing approach.
Both models are tested on the same test data set consisting of 6000 signals. The parameter of the signal processing approach are chosen by searching for the best performance in the training data set for the NN models.

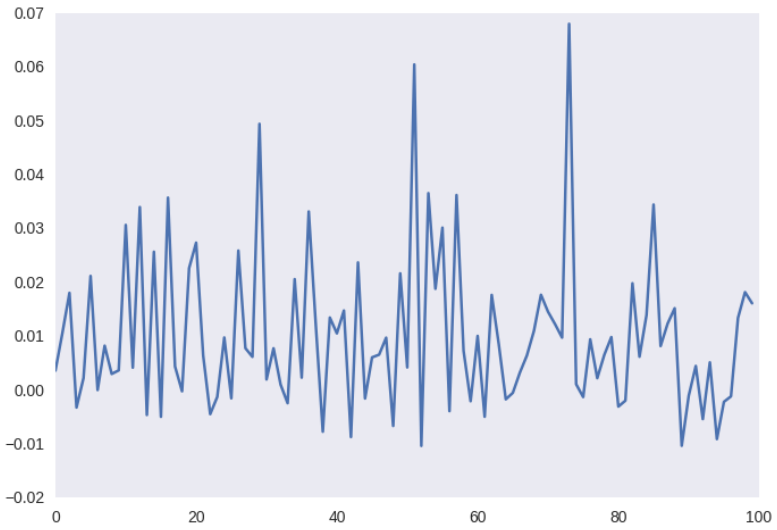
3.3.2 Weights and biases

In order to improve our understanding of the NN, we inspect weights from the trained model. For example, in the case of the classification NN referred to above, the first layer weights connecting the input neurons and hidden layer neurons has shape 100×100. The second layer weights that connect the hidden layer neurons and the output layer neurons give rise to a weight matrix of shape 100×60. We plot the weight matrix (as a 2D heat map) and the bias for each layer of the network.

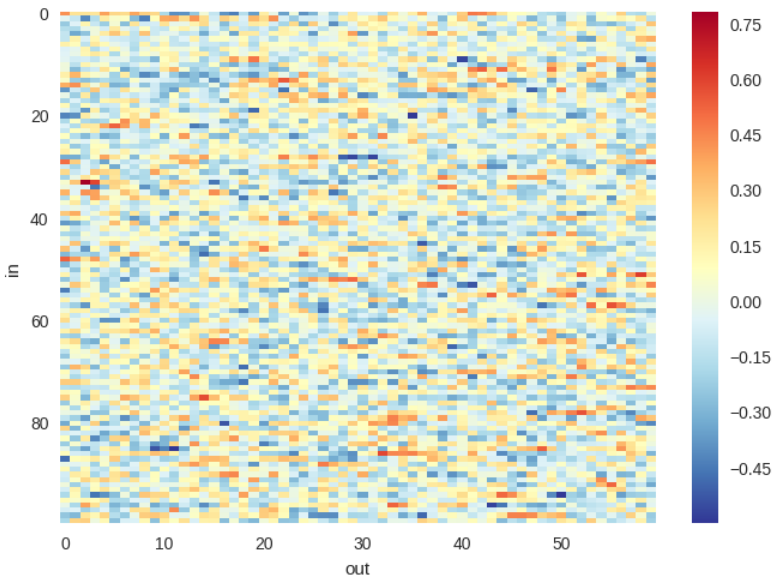
Generally speaking, the plots aren't very informative and not much insight can be gained from the weights and biases. Only one observation might be notable that in Fig. 3.11(a), the weights from input neurons outside the location range 20-80 are significantly smaller. This result can be naturally explained by the fact that the jump occurs only between positions 21 and 80.



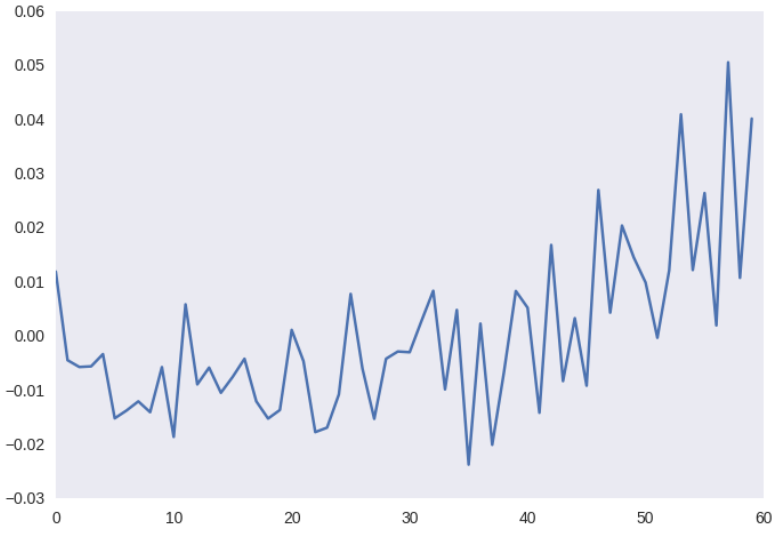
(a) 1st layer weights



(b) 1st layer biases



(c) 2nd layer weights



(d) 2nd layer biases

Figure 3.11: 2-layer MLP: weights and biases. (a) and (b) represent for the first hidden layer, the weights and bias respectively. (c) and (d) do the same for the output layer.

3.3.3 Visualizing layer-wise class representation by averaging activations

Besides the weights and biases, we also investigate the output of the network, not only of the last layer, but also of the hidden layer. Intuitively, we suppose that there would be some regular relationship between the output in the hidden layers and the jump positions. As one instance of data sample brings considerable noise into the network neuron activations, we average neuron activations from a large number of input data instances of the same class, as the representation of the class in the layer. This manipulation is reasonable in this specific task because averaging the original input in the same class results in a typical and ideal jump signal of the class and even with very little noise.

We propose a visualization of averaged layer-wise activations. For this purpose, we generate a new larger data set of 60000 signals, The labels in the networks are from 0 to 59, which actually represent the jump positions from 21 to 80. There are 60 different labels in all. In this new data set, there are exactly $N = 1000$ instances of signals designed in each jump position or class (c). We average the output of each layer from the 1000 instances.

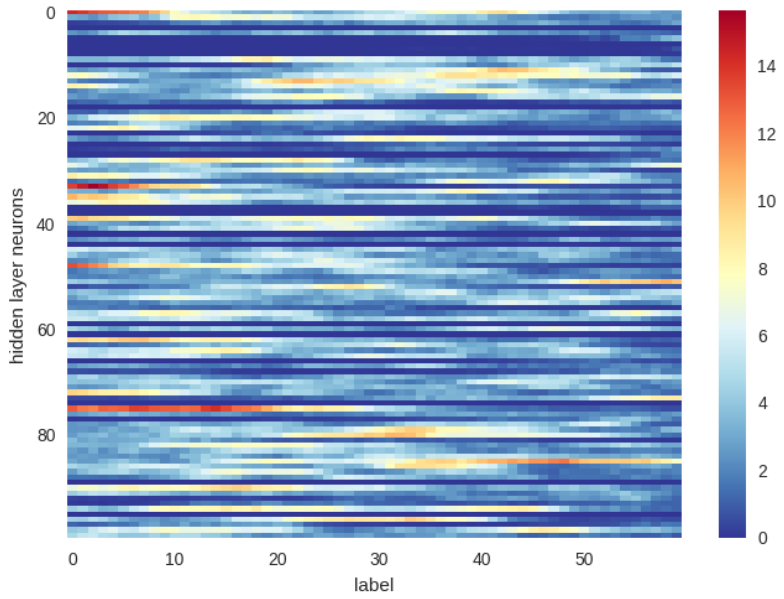
The averaged activation $\mathbf{A}^{(l)}_{(c)}$ of layer l for the jump position (c) is a vector:

$$\mathbf{A}^{(l)}_{(c)} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{a}^{(l)}_{(c),n}, \tag{3.6}$$

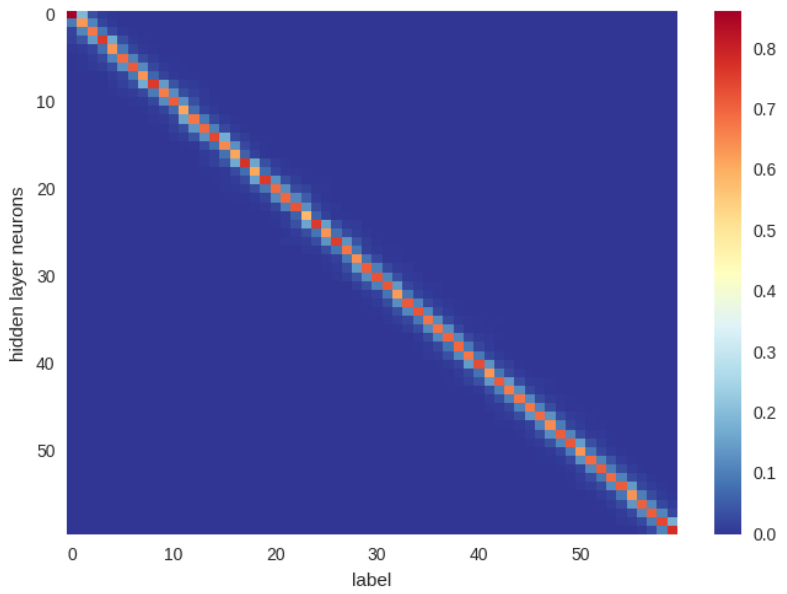
where $\mathbf{a}^{(l)}_{(c),n}$ is the activation vector of the l^{th} layer which is from the n^{th} of 1000 data instance in class (c). $\mathbf{A}^{(l)}_{(c)}$ is considered as a class representation of class (c) in the layer l. The averaged activation of layer l is a matrix:

$$\mathbf{A}^{(l)} = \begin{bmatrix} \mathbf{A}^{(l)}_{(21)}{}^{T} & \mathbf{A}^{(l)}_{(22)}{}^{T} & \dots & \mathbf{A}^{(l)}_{(80)}{}^{T} \end{bmatrix} \tag{3.7}$$

In each layer of the model, the matrix $\mathbf{A}^{(l)}$ is plotted as a heatmap in order to investigate the behaviour of the (intermediate) output of each layer with regard to the different classes (jump positions). The results are illustrated in Fig. 3.12. The two sub-figures are $\mathbf{A}^{(1)}$ (i.e. hidden layer) and $\mathbf{A}^{(2)}$ (i.e. output layer) respectively.



(a) $\mathbf{A}^{(1)}$: 1st (hidden) layer averaged activation



(b) $\mathbf{A}^{(2)}$: 2nd (output) layer averaged activation

Figure 3.12: 2-layer MLP: layer-wise output (averaged over 1000 samples for each jump position). The labels on the x-axis range from 1 through 60 representing actual (i.e. true) jump positions at locations 21 through 80. The values in each column specify the (averaged) activation of the neurons (in that layer) for signals with that jump position.

Figures 3.12(a) and Fig. 3.12(b) illustrate the output from layers 1 and 2. The labels on the x-axis range from 1 through 60 representing actual (i.e. true) jump positions at locations 21 through 80. The values in each column specify the (averaged) activation of the neurons (in that layer) for signals with that jump position.

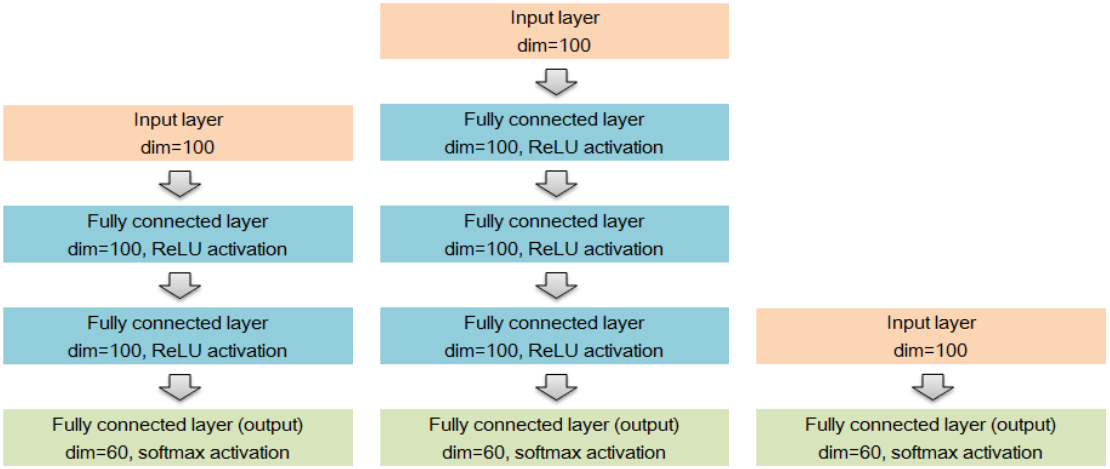
The visualization of the output layer $\mathbf{A}^{(2)}$ in Fig. 3.12(b) does not offer any surprises. From the overall error measures we already knew that the predicition is quite accurate, and this is reflected in the (nearly perfect) diagonal structure of the activation matrix. As for the hidden layer $\mathbf{A}^{(1)}$ we can make the following observations:

- In the intermediate layer, the value of activations changes more significantly with neurons rather than jump positions. In each neuron, the output usually varies gradually within a wide range of (contiguous) jump positions. This results in the stripe-like figures in the visualization.
- The stripe-like hidden features could be some way to encode the information of the jump position. For instance, activation of the neurons around position 75 are indicative of an actual jump location between 1 and 20 (i.e. 21 and 40 — taking the offset into account).
- There are some neurons which are almost never activated at all.
- The figures may also indicate that each single neuron is not informative enough for the final result (the jump positions). It is obvious that the network has too many redundant neurons to process such a problem.

The stripe-like figure in the hidden activation is yet difficulty to understand. For a more concrete understanding of encoding with hidden layer averaged activations, see Section 4.2.3, where we show that the stripes actually indicate that the continuous distribution of data is still preserved in hidden layer activation spaces.

MLPs with different numbers of layers

In addition to the most straightforwad two-layer MLP, other network structures are also considered (Fig. 3.13), with more intermediate hidden layers. And surprisingly, the one-layer fully connected NN also successfully serves for our jump location detection task.



(a) Three-layer MLP (b) Four-layer MLP (c) One layer fully connected NN

Figure 3.13: MLP structures

As usual, the input layer (regarded as the 0^{th} layer) has 100 neurons to represent an input signal of length of 100. The intermediate hidden layers are all with 100 neurons to be consistent with the size of the input. The rectified linear activation $f(z) = \max(z, 0)$ is adopted in the intermediate hidden layers. The output layer has 60 neurons. Softmax function is the activation function of the output layer.

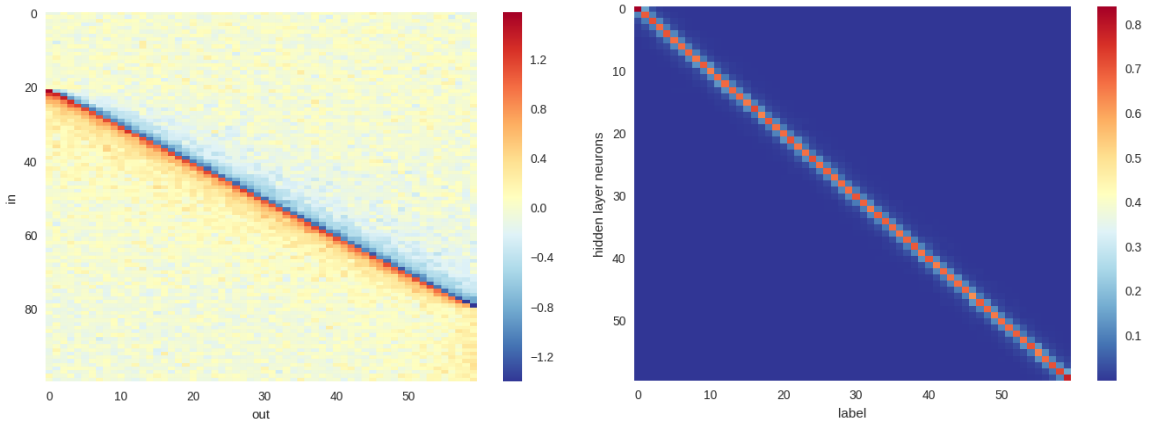
The performance of the different structures are listed in Table 3.3. All models outperform the signal processing approach in terms of accuracy. The error of the networks with one or two layers are relatively better. Adding a third layer seems to have negative effect on performance.

	Complete data		99 percentile	
	L_1 error	L_2 error	L_1 error	L_2 error
Signal processing approach	0.5532	1.4946	0.4917	0.7443
1-layer NN	0.3065	0.8043	0.2606	0.6304
2-layer MLP	0.3083	0.7722	0.2675	0.6222
3-layer MLP	0.3583	0.8132	0.3169	0.6667
4-layer MLP	0.3913	0.8428	0.3531	0.7075

Table 3.3: Performance MLP

We also plot the heatmap of the weight matrix and visualize the layer-wise averaged activation for these network structures. The results and the observations are generally the same as for the 2-layer MLP. (Therefore they are not listed.) It is still impossible to interpret much useful information from these visualizations.

However, the results of the one-layer fully connected network is much more interesting. In Fig. 3.14(a), the weight matrix is close to a diagonal matrix (regarding the range of y-axis between 21 and 80, where the jump happens). Only the elements around the first diagonal have significant values, while the values of other elements are almost zero. It is easy to understand that only the weights close to jump positions could help detecting the jump and picking up the location. Moreover, the pattern of the weights along the first diagonal is regular and implements a clear functionality.



(a) 1st (output) layer weight matrix

(b) 1st (output) layer averaged activation

Figure 3.14: 1-layer NN: weight matrix(left) and layer-wise averaged activation (right).
In the visualization of outputs, the labels in the x-axis is from 1 to 60, which actually represents the jump positions from 21 to 80.

To see this, the thirtieth column vector of the weight matrix is taken as an example, which stands for all the weights contributing to the output neuron of the jump location at 50: $\begin{bmatrix} w_{30,1}^{(1)} & w_{30,2}^{(1)} & \dots & w_{30,100}^{(1)} \end{bmatrix}^{\mathbf{T}}$, where $w_{j,i}^{(l)}$ is the weight from the i^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in l^{th} layer. The weight matrix can be regarded as a 1-dimensional convolutional filter moving along the the signal. Basically, the weight matrix represents a convolution with a smoothed first derivative implemented as a matrix multiplication.

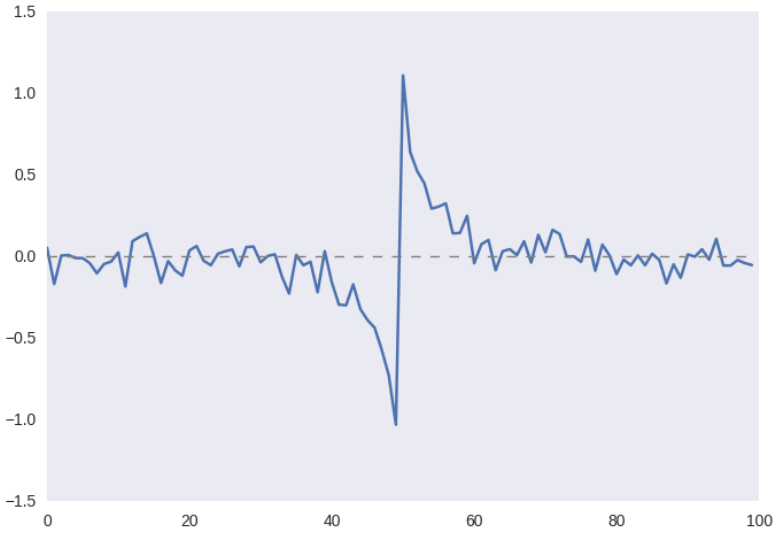


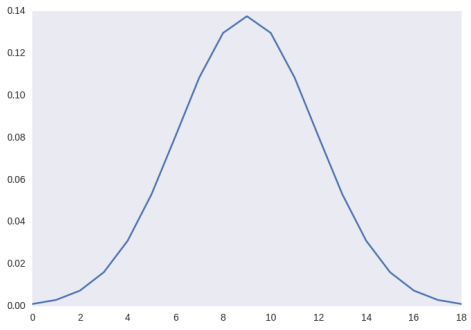
Figure 3.15: The thirtieth column vector of the weight matrix contributing to the output neuron of the jump location at 50.

To put this into context, recall how the standard signal processing approach and the relevant filters (Fig. 3.16) achieves the localization:

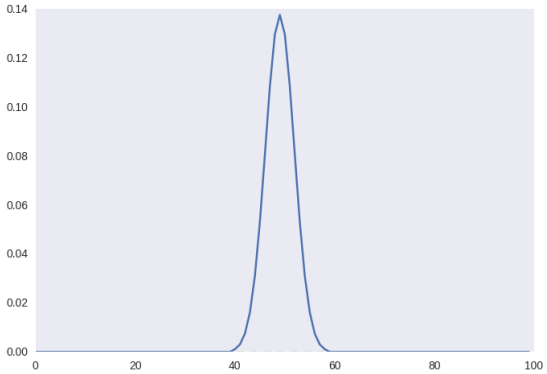
1. First, perform a convolution with Gaussian filter to smooth the signal.

2. Second, take the first derivative by convolving a filter as in Fig. 3.16(c).

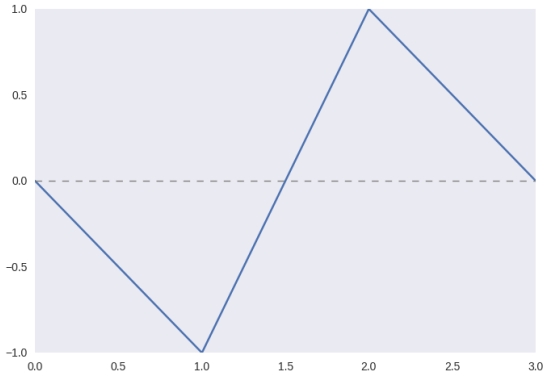
Notice that the two processes above are performed sequentially to the whole signal. The resulting bahaviour of the signal processing approach is a convolution filter as in Fig. 3.16(d), which is a convolution of the Gaussian smoother in Fig. 3.16(a) and the first derivative filter in Fig. 3.16(c).



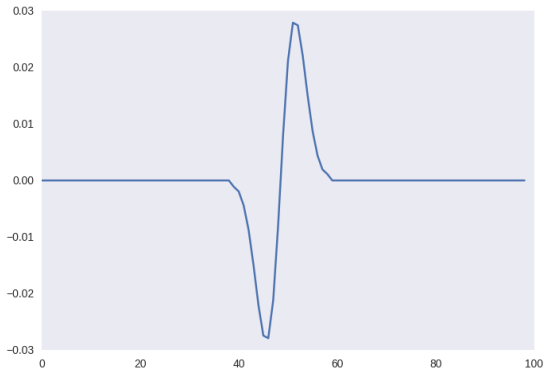
(a) filter as a Gaussian smoother



(b) Gaussian smoother: local at 50



(c) filter for the 1st derivative



(d) convolving (c) and (b): the actual filter of signal processing approach at location 50

Figure 3.16: Signal processing approach: (1) First, perform a convolution with Gaussian filter ((a), or (b) as the local illustration at location 50.) to smooth the signal (2)Second, take the first derivative by convolving a filter as (c), which is equivalent to (d) that convolves (b) and (c).
(d) is the actual filter of the signal processing approach at location 50

Compare Fig. 3.15 learnt from the network and Fig. 3.16(d) from the signal processing approach. They have similar shapes and also similar functions as a smoothed first derivative extractor. Besides, the range of the weights close to the exact jump location 50 perform some work of smoothing. Thus, the filter illustrated in Fig. 3.15 performs both the job of Gaussian smoother and a first derivative solver in the signal processing approach simultaneously. According to the error discussed in Table 3.3, the new filter even has a higher performance.

In conclusion, the weights in the 1-layer NN are explainable with the knowledge of signal processing and one-dimensional convolutional filters. Moreover, the resulting weights from the high performance of the NN is valuable as a reference to improve and optimize the traditional methodology.

3.4 CNNs for Detection of Jump Location

In addition to the feedforward networks, we also perform the discontinuity detection task with convolutional neural networks (CNNs). CNNs are powerful deep learning models that achieve significant performance in various tasks including vision, signal processing and natural language processing. In our task, we expect that CNNs would be a natural solution and the convolutional layers in CNN would mimic the function of Gaussian smoother windows in the signal processing approach. Furthermore, the experiments with a 1-layer MLP in section 3.3.3 also strongly suggest that CNNs would provide a viable solution. In fact, the weight matrix effectively is a convolution operation implemented as a matrix multiplication.

In this section, the consideration of the structure design of CNNs is introduced. The weights and averaged activations are visualised, in order to analyze the interpretability of the network. Through the analysis, CNNs with certain well designed structures become interpretable, by which we mean that the conv weights are corresponding to the function of filters in signal processing approaches. The comparative analysis of the performance of different CNN structures is at the end of this section.

3.4.1 One conv layer: structure, weights and hidden feature visualization

Firstly, we consider the simplest possible network structure (Fig. 3.17). Right after the input, the conv part of the network is only one conv layer with only one conv filter. Afterwards, the number of fully connected layers is kept to only one, for the reason that the fully connected layers are too powerful and difficult to interpret. Besides, since our task is to detect a specific position (i.e., an element-wise detection), no pooling layers are expected in the network design.

One-dimensional convolution is performed in conv layers. With unit stride and same padding, the convolution of a conv filter g_τ of length τ on an input signal $f(x)$ is

$$g_\tau * f(x) = \sum_{u=1}^{\tau} g_\tau(u) f(x-u), \tag{3.8}$$

where u is a dummy integer variable. This definition is exactly the same as in the signal processing approach. See eq. 3.2.

The network structure is illustrated in Fig. 3.17. In implementation, the length of the conv filter is set as 7 and same padding is performed in order to obtain the output with the same length as the input. The hyper-parameters in training remain the same as in fully connected networks. The batch size is 100 and the optimizer is SGD.

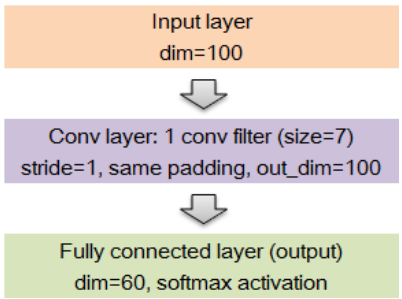
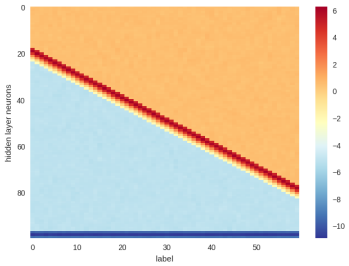


Figure 3.17: The network structure: CNN with one conv layer and one fully connected layer.

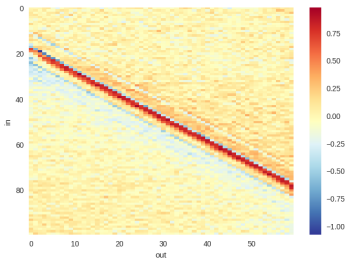
The weights and the averaged activation visualizations are in the Fig. 3.23. The weights are illustrated in the left column. The biases are omitted since they do not provide useful information. In the middle column is the averaged hidden activations (except in the last row is the activations of the output neurons). In order to clarify the hidden activations, in the right column the hidden activations at jump position 50 are shown as a representive case.

(a) 1st (conv) layer weights

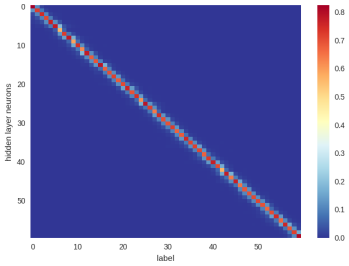


(b) 1st (conv) layer averaged hidden activation

(c) 1st (conv) layer averaged hidden activation at position 50



(d) fully connected layer weights

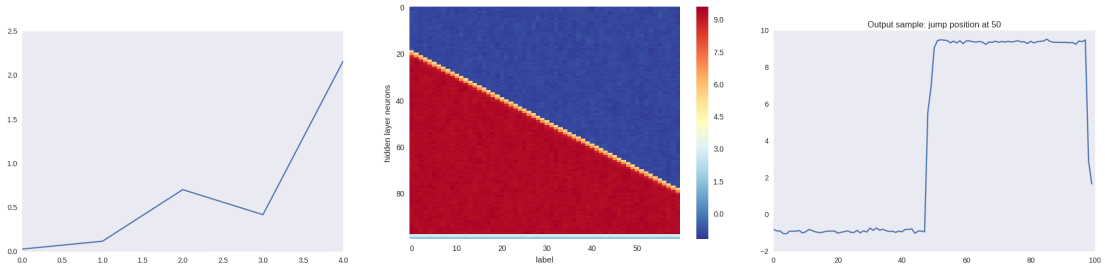


(e) fully connected layer averaged activation

Figure 3.18: The weights and the averaged activation visualizations of CNN with one conv layer and one fully connected layer. (conv filter length 7)
Weights are plotted in the left column. The averaged (hidden) activations are illustrated in the middle column, in which the labels from 0 to 59 stand for jump positions from 21 to 80. In the right column is the averaged (hidden) activation at position 50 (i.e., a slice from the matrix plotting in the middle column)
The results from the same layer are listed in the same row.

In the conv layer, the observation from the hidden activations shows that the conv filter produces a peak value near the jump position. This is a consequence of the fact that the convolution filter (cf. Fig 3.18(a)) is actually the linear combination of an averaging filter (left part) and a first derivative (right part). The result is therefore similar to the convolution results for the mixed filter in Section 3.2.2: the inputs are smoothed but discontinuities give rise to a sharp peak (due to the derivative). Afterwards, the weights of the fully connected layer are basically in the first diagonal. The function of this weight matrix is to pick up the position of the peak value.

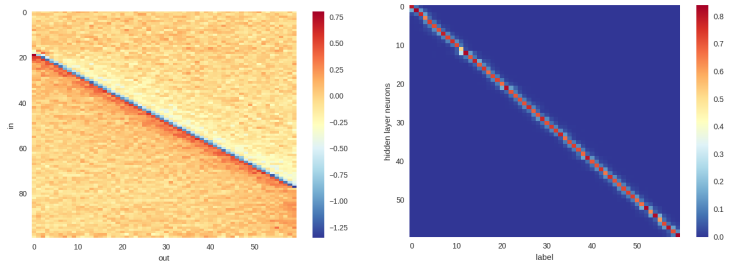
However, such a network structure does not always behave the same. Empirically the behaviour of the network is influenced by the length of the conv filter (7 in Fig. 3.23). Although we expected that a conv filter with less neurons can also suffice to extract the first derivative, the experimental results show that when the conv filter length is less than 7, it tend to only perform smoothing and gap-increasing instead of finding a first derivative (Fig. 3.19). Although the shape of the conv filter weights are similar, the sign of the weights is very important in the interpretation: in Fig. 3.19(a) all weights have the same sign and so it is a smoothing filter; otherwise, in Fig. 3.18(a) the weights are in different signs and so it is a derivative-type contrast enhancer.



(a) 1st (conv) layer weights

(b) 1st (conv) layer averaged hidden activation

(c) 1st (conv) layer averaged hidden activation at position 50



(d) fully connected layer weights

(e) fully connected averaged activation

Figure 3.19: An alternative situation of CNN with one conv layer and one fully connected layer. (conv filter length 5)
When the conv filter has an insufficient number of neurons, it can only smooth the signal and increase the gap, rather than extract a first derivative.

3.4.2 With two conv layers

The simplest CNN structure has been proven to be well interpretable in the jump detection task. Pushing beyond this result, we are also interested in more complicated (i.e. deeper) structures. For this purpose, we add one more conv layer to the network and investigate the properties and behaviour of the two conv layers.

Keeping all other conditions the same as in the experiment above, we attached two convolutional layers instead of one to the network, both with only one convolution filter of length 5 and 9 respectively. The network structure is illustrated in Fig. 3.20.

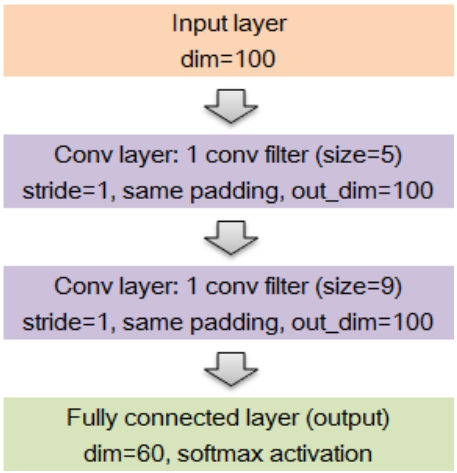
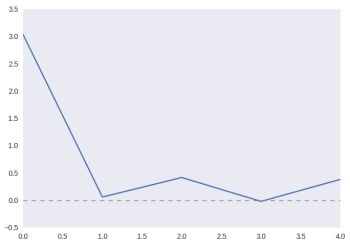
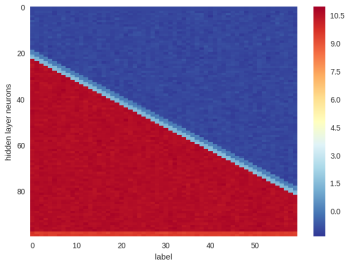


Figure 3.20: The network structure: CNN with two conv layers and one fully connected layer.

In this architecture, the behaviour of the two conv layers indicates hierarchical structure in the network. The two convolutional layers are functionally different. The first conv layer with a conv filter of length 5 performs a smoothing and gap increasing. The second conv layer with a conv filter of length 9 extracts the first derivative

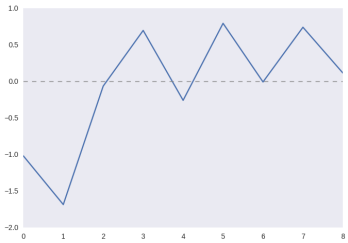


(a) 1st conv layer weights

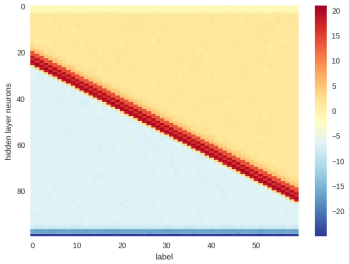


(b) 1st conv layer averaged hidden activation

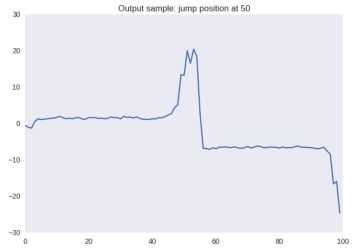
(c) 1st conv layer averaged hidden activation at position 50



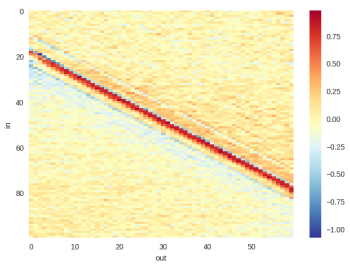
(d) 2nd conv layer weights



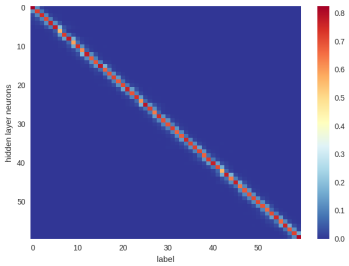
(e) 2nd conv layer averaged hidden activation



(f) 2nd conv layer averaged hidden activation at position 50



(g) fully connected layer weights



(h) fully connected layer averaged activation

Figure 3.21: The weights and the averaged activation visualizations of CNN with two conv layers and one fully connected layer. (conv filter length 5 and 9 respectively)
The first conv layer with a conv filter of length 5 performs a smoothing and gap increasing.
The second conv layer with a conv filter of length 9 extracts the first derivative.

3.4.3 With three conv layers

Furthermore, if we attach one more convolutional layer, what would be the function of this third convolutional layer? We attach three conv layers and each with only one conv filter of size 5, 7 and 9 respectively.

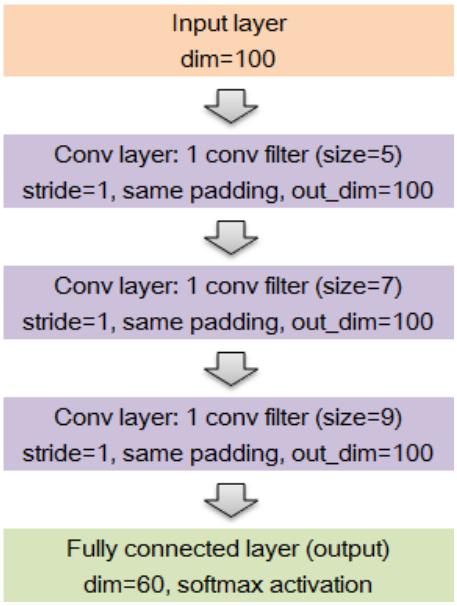
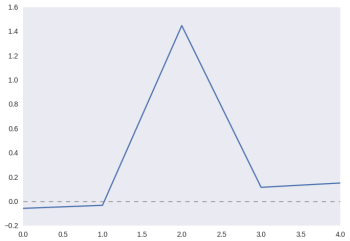
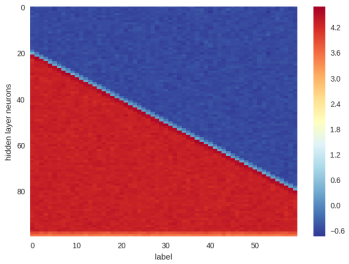


Figure 3.22: The network structure: CNN with three conv layers and one fully connected layer.

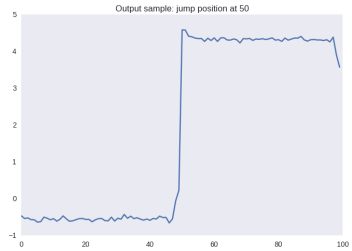
Similarly, the averaged hidden layer activations are illustrated. As well, the row of the figures indicates the results are from different layers in sequence. The first column is the hidden outputs with 1000 time series averaged in each jump position. The second column is a slice of charts in the first column where the jump position is 50. The third column takes a single signal as an example.



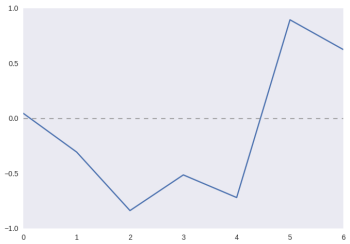
(a) 1st conv layer weights



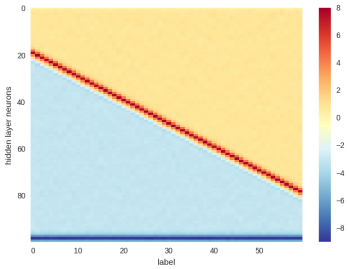
(b) 1st conv layer averaged hidden activation



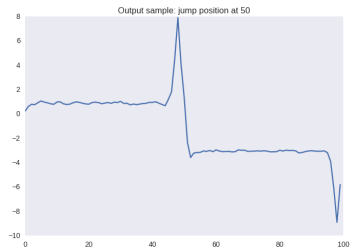
(c) 1st conv layer averaged hidden activation at position 50



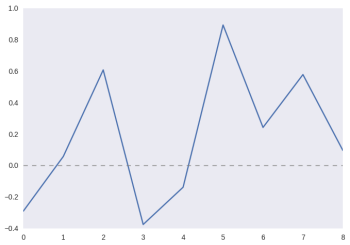
(d) 2nd conv layer weights



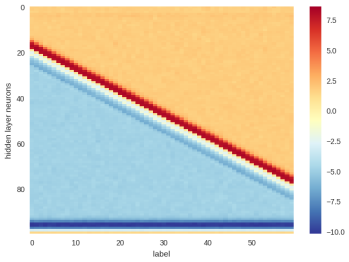
(e) 2nd conv layer averaged hidden activation



(f) 2nd conv layer averaged hidden activation at position 50

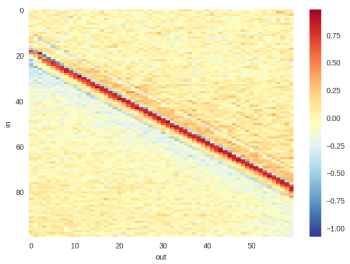


(g) 3rd conv layer weights

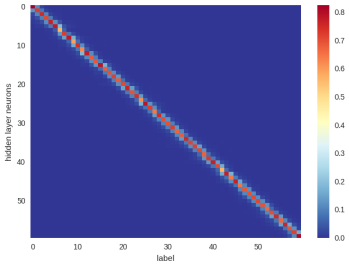


(h) 2nd conv layer averaged hidden activation

(i) 2nd conv layer averaged hidden activation at position 50



(j) fully connected layer weights



(k) fully connected layer averaged activation

Figure 3.23: Analysis of CNN with two conv layers and one fully connected layer. (conv filter length 5 and 9 respectively.)
The first conv layer with a conv filter of length 5 performs a smoothing and gap increasing.
The second conv layer with a conv filter of length 9 extracts the first derivative.

The first two convolutional layers are functionally similar to the CNN model with only two convolutional layers. The hidden output after the third convolutional layer shows no significant difference compared to the second convolutional layer, which indicates that this third conv layer is redundant to some extent.

As compared to the CNN with two conv layers, adding one more conv layer does not introduce new hierarchical properties into the model.

3.4.4 Performance

The error rates for CNNs are listed in Table 3.4. All CNN structures surpass that of the signal processing approach. Different CNN structures are close to each other and also very close to the MLPs in performance. The 1-layer MLP is noted as a 0-conv-layer CNN since the structure is related to our CNN models (all with only one fully connected layer). From the error scores, the overall performance of the 1-layer-MLP with no conv layers is a bit better than CNNs, while the large L_2 error may indicate that the 1-layer MLP has more exceptionally high errors. The performance of CNNs shows more consistency.

	Complete data		99 percentile	
	L_1 error	L_2 error	L_1 error	L_2 error
Signal processing approach	0.5532	1.4946	0.4917	0.7443
1-layer MLP (0-conv-layer CNN)	0.3065	0.8043	0.2606	0.6304
2-layer MLP	0.3083	0.7722	0.2675	0.6222
3-layer MLP	0.3583	0.8132	0.3169	0.6667
4-layer MLP	0.3913	0.8428	0.3531	0.7075
1-conv-layer CNN	0.3248	0.7862	0.2854	0.6452
2-conv-layer CNN	0.3213	0.7592	0.2839	0.6309
3-conv-layer CNN	0.3458	0.7946	0.3089	0.6721

Table 3.4: Performance CNN

The 2-conv-layer CNN is the best among CNNs. As discussed above, it is also the one with better interpretability and hierarchical structures. However, the performance is still no better than the 2-layer MLP. The 3-conv-layer CNN, with a redundant conv layer, produces highest errors.

3.4.5 Summary of CNNs

Empirically, the conv networks with simple structures (i.e. only one filter in each conv layer and no more than one fully connected layer attached to the network) can be interpreted directly from weights and hidden activations. The different layers perform certain functions sequentially. E.g. some layers perform smoothing and some layers perform derivative extraction. The resulting conv filters usually resemble one of the cases in Fig. 3.3 and Fig. 3.4. The final fully connected layer weights probably perform the function as a targeted smoothing with regard to the intermediate output after all conv layers. However, the conv filters are usually not compact. E.g. the simplest first derivative filter has only a length of two $[-1\ 1]$, while in NNs, the length of the conv filter should be at least 7 to learn such a behaviour, and there are inevitable redundant weights.

Chapter 4

Understanding Hidden Representations in Bottleneck Networks

Contents

4.1 Introduction

In this chapter, we will tackle the problem of jump location using "bottleneck" networks, which are a special type of MLP inspired by autoencoders. More specifically, in bottleneck networks, the size of one or more hidden layers is significantly less than the size of the in- or output layers. In this sense they are similar to auto-encoders, but the latter try to create an output which matches the input exactly. In our task, the output and input are different in that the output is a one-hot encoding of the location of the jump. So we are basically going from an analog input to a symbolic output. For that reason we have opted to use a different term, to forestall any possible confusion. The analysis of the results from the "bottleneck" network aims at better understanding the hidden layer features of NNs.

Autoencoder The idea of an autoencoder is to learn a function for which the output is the input itself $f(x) = x$. To make this task non-trivial, there is the restriction that the number of neurons in

the hidden layer is (significantly) less than input or output layers. The rationale for this set-up is that a successfully trained autoencoder network would extract a low dimensional representation of the original data. With the hidden layer of smaller size, the autoencoder is able (or forced) to extract principal features of the data. We would like to investigate how the fully connected feedforward NNs encode the information, with the first intuition that the low-dimensional representation would probably be easier to interpret.

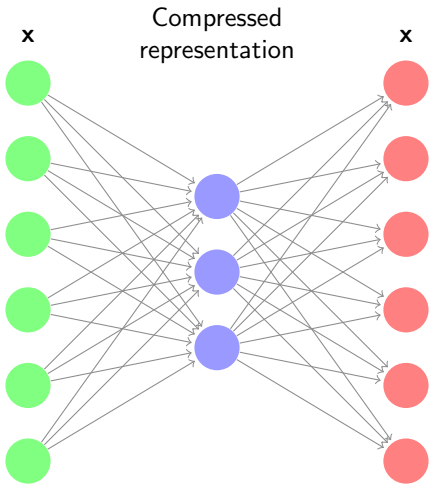


Figure 4.1: An autoencoder network. The input and the output is the same vector $\mathbf{x}$ *of length 6. The hidden layer is the compressed representation of the data.*

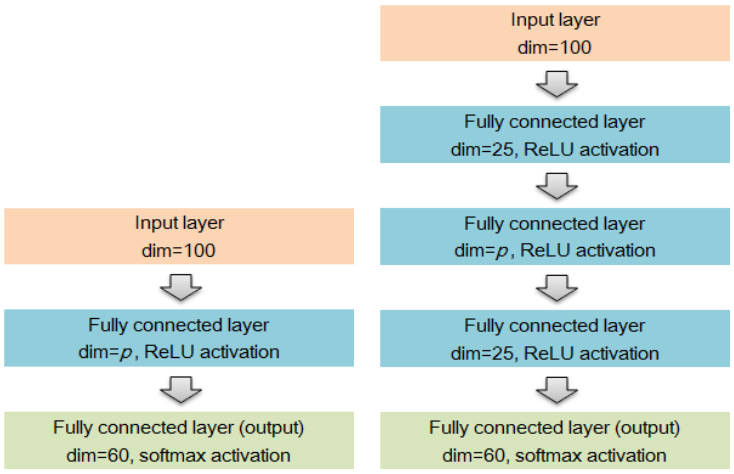
Bottleneck networks In our task, we do not actually learn the $f(x) = x$ function, but we set the output the same as before, that is, one-hot encoded jump positions. Because the Gaussian noise is totally random and the jump is the only informative feature in the signal we expect the low-dimensional representation in the bottleneck to only encode the information of the jump position, rather than the random noise in the original signal. The goal in this chapter is to find out how the network encodes the information when it needs to pass through a layer with very few neurons and is therefore (severely) restricted in the information it can transmit to the next layer(s).

Suppose we aim at a p-dim representation of the original input signal. Therefore, the number of neurons in the narrowest hidden layer is p.

4.2 Empirical Analysis

4.2.1 Network structures

Two kinds of structures are considered. The first one is a shallow bottleneck network with only one hidden layer, i.e., the number of neurons from input to output is 100-p-60. The second kind is the deep bottleneck network with three hidden layers, of which the structure is 100-25-p-25-60 (see Fig. 4.2). All the activation or hyperparameter settings are kept the same as in Section 3.3. In particular, the activation is ReLU for all hidden layer neurons and softmax for the output layer. The training, test and validation data consist of 12000, 6000, 6000 data samples respectively. In training, the batch size is 100 and SGD is used to optimize the categorical cross-entropy error. The learning rate is 0.01. Early-stopping is performed when the cross-entropy loss of the validation data starts to increase.



(a) shallow "bottleneck" network (100-4-60)

(b) deep "bottleneck" network (100-25-4-25-60)

Figure 4.2: "Bottleneck" network structures.

The weights and biases are hard to directly interpret like in ordinary MLPs. Generally speaking, the conclusions of the hidden layer activations in MLP applies to the "bottleneck" networks as well. (So the results are not listed here.)

4.2.2 Performance

We investigate the performance in "bottleneck" networks in Table 4.1. The results of ordinary MLPs with two and four layers are also listed as benchmark. As expected, the error rates are higher than a general MLP with the same number of layers. The performance is even significantly worse than the signal processing approach. Networks with fewer hidden neurons have worse error rates. The deeper alternative "bottleneck" network structures are with better performance than the shallow ones, although a general 4-layer MLP is worse than a 2-layer MLP. One understanding from the comparative observation of the performance is that a more complicated network structure provides more flexibility, either with deeper structure or with more intermediate hidden neurons. Usually in standard MLPs, the performance of a 4-layer network is not as good as a 2-layer network, because the gradients are not preserved well in spread between layers. However, the deep "bottleneck" network performs better, because as the parameters are not sufficient in the "bottleneck" network case, the deeper network with more parameters has the advantage.

	Complete data		99 percentile	
	L_1 error	L_2 error	L_1 error	L_2 error
Signal processing approach	0.5532	1.4946	0.4917	0.7443
2-layer MLP	0.3083	0.7722	0.2675	0.6222
4-layer MLP	0.3913	0.8428	0.3531	0.7075
"Shallow" "bottleneck" network 100-10-60	0.7980	1.4658	0.7370	1.0820
"Shallow" "bottleneck" network 100-4-60	1.1626	1.6809	1.0993	1.4755
Deep "bottleneck" network 100-25-10-25-60	0.6478	1.3121	0.5742	0.9255
Deep "bottleneck" network 100-25-4-25-60	0.7000	1.1795	0.6450	0.9661

Table 4.1: Error of "bottleneck" networks. Usually in standard MLPs, the performance of a 4-layer network is not as good as a 2-layer network, because the gradients are not preserved well in spread between layers. However, the deep "bottleneck" network performs better, because as the parameters are not sufficient in the "bottleneck" network case, the deeper network with more parameters has the advantage.

To have a more precise quantitative study on the relationship of the performance and the number of hidden neurons, networks with number of hidden neurons from 2 to 10 are tested. The results for shallow 2-layer networks are illustrated in Fig. 4.3. Generally speaking, the error rates monotonically decrease as more neurons are in the hidden layer. The only outlier is the L_2 error rate, to which the meaningless extremely large errors contributes more. This result indicates that the L_2 error rate is less reliable than the other three metrics.

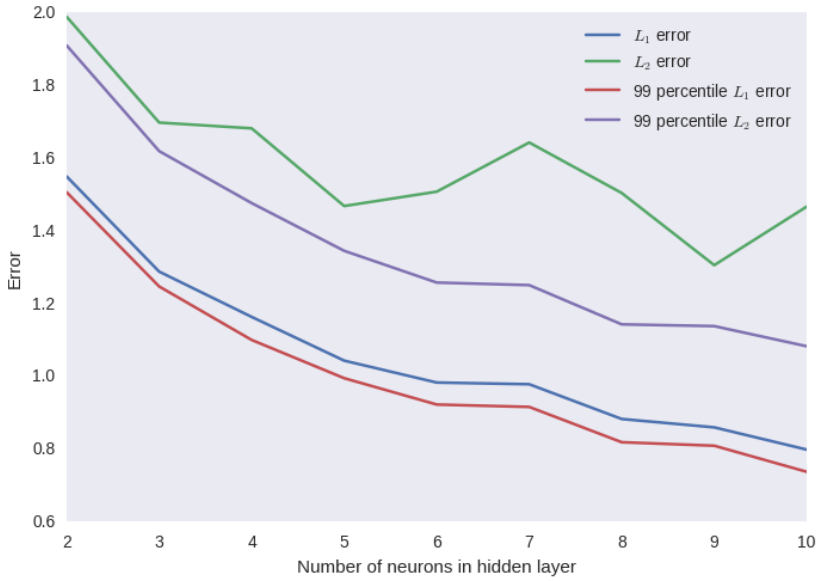


Figure 4.3: Shallow 2-layer "bottleneck" networks. Error rates against the number of hidden neurons.

The results for deep 4-layer networks are illustrated in Fig. 4.4. The general observations are similar. However, the three reliable error measures do not necessarily monotonically decrease with number of neurons in the bottleneck layer. Especially, from $p = 5$ to $p = 6$ and from $p = 9$ to $p = 10$, the errors even show a slight rise although it is not significant.

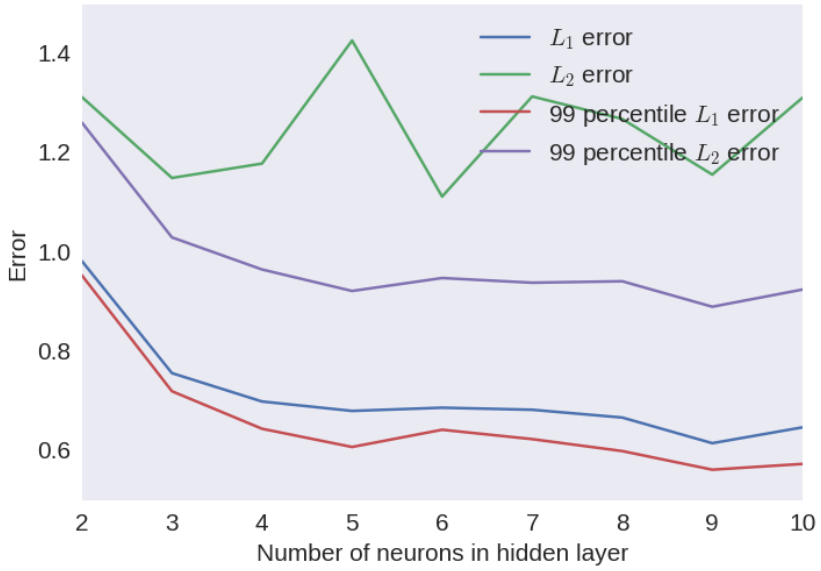
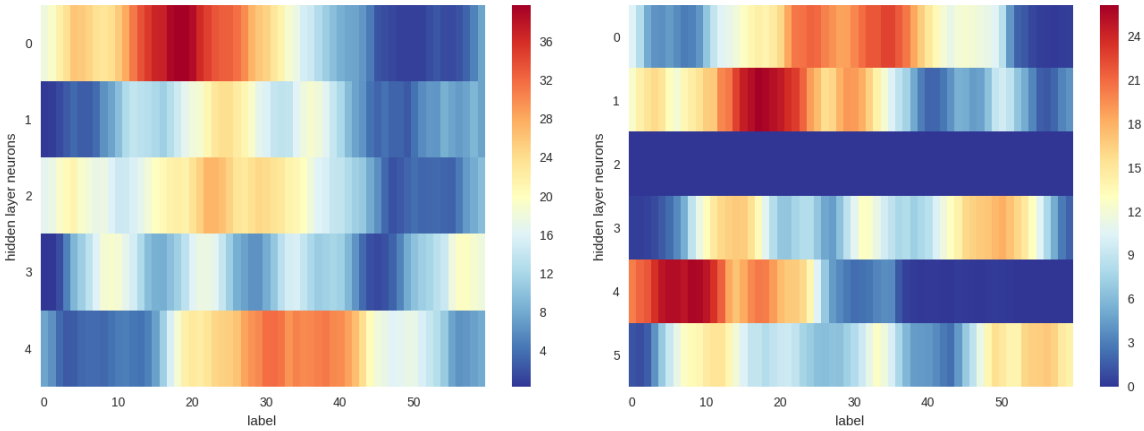


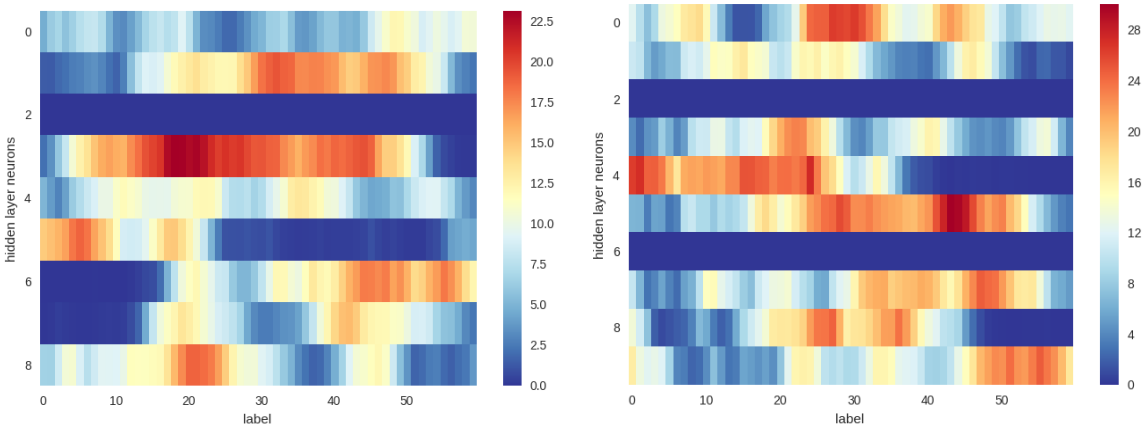
Figure 4.4: Deep 4-layer "bottleneck" networks. Error rates against the number of hidden neurons.

The abnormal change of errors is caused by de-activated neurons in the deep layer (see Fig. 4.5). When $p = 6$, the bottleneck layer has one neuron de-activated. The number of activated neurons is the same as in the case $p = 5$. Therefore, it is reasonable that the performance of $p = 6$ is no better than $p = 5$. Similarly, in the models $p = 9$ and $p = 10$, both have only 8 neurons activated in the bottleneck layer, with one and two neurons deactivated respectively. As a result, the error rate of the network with $p = 10$ is even slightly higher.



(a) 100-25-5-25-60: bottleneck layer activation

(b) 100-25-6-25-60: bottleneck layer activation



(c) 100-25-9-25-60: bottleneck layer activation

(d) 100-25-10-25-60: bottleneck layer activation

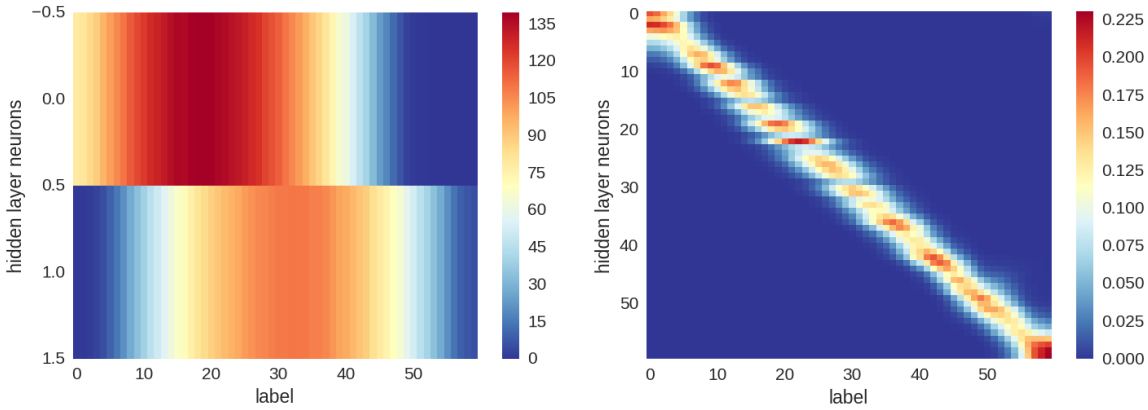
Figure 4.5: De-activated neurons in deep "bottleneck" networks in the bottleneck layer. When $p = 6$, the bottleneck layer has one neuron de-activated. The number of activated neurons is the same as in the case $p = 5$. Therefore, it makes sense that the performance of $p = 6$ is no better than $p = 5$. Similarly, in the models $p = 9$ and $p = 10$, both have only 8 neurons activated in the bottleneck layer. Thus, the error even increases a bit.

Empirically, the deeper 4-layer "bottleneck" networks produce non-activated neurons in all intermediate hidden layers including the bottleneck layer. While "shallow" 2-layer networks do not. One possible explanation could be that in a deeper network the gradient passing through are easier to get lost during the training session. As the activation value falls into the negative half in training, the rectified linear activation function forces the gradient to be zero, and so the neuron is effectively de-activated. The two more intermediate hidden layers before and after the "bottleneck" layer with the fewest neurons provide flexibility, which allows some deactivated neurons to be tolerated.

4.2.3 Understanding hidden layer activation with example of 2-dim representation

Autoencoders are designed to look for low-dimensional representations of the original data. With the same idea, in the "bottleneck" networks, it is feasible to obtain the low-dimensional representations of the original 100-dimensional data inputs that encodes the jump locations.

For convenience, only the networks with 2 neurons in the bottleneck layer are considered, because representations in the 2-dim space are easy to visualize. First we consider the shallow network 100-2-60. As in MLP, we visualize the layer-wise averaged activations as well in Fig. 4.9. Fig. 4.6(a) is the 2-dim representation of the original data.



(a) 2-dim representation of the data (averaged within each class): averaged hidden activation A^l in the bottleneck layer

(b) final output

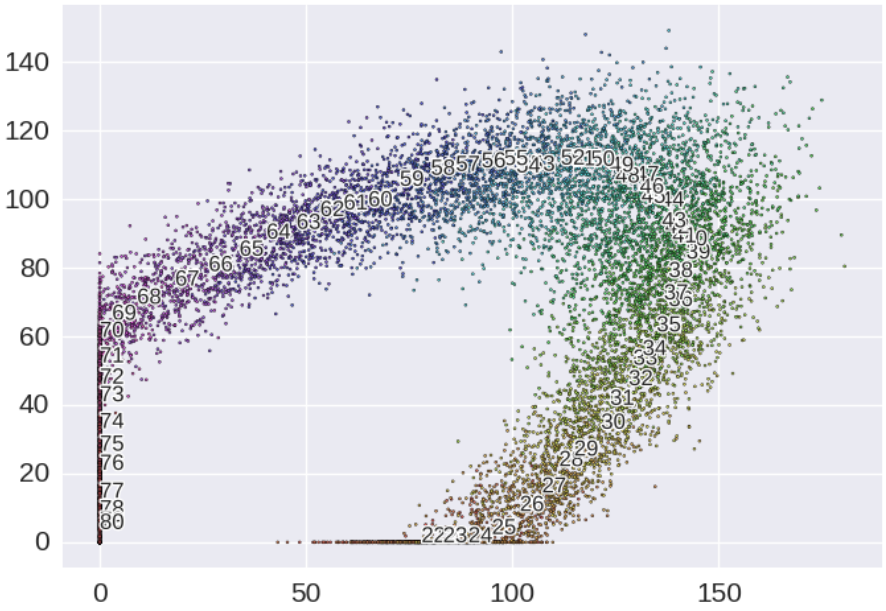
Figure 4.6: Layer activations in 100-2-60.
Left is the bottleneck layer, in which the intermediate activation is regarded as a 2-dim representation.
Right is the final output. The performance is relatively poor. Some adjacent classes are not well distinguished by the model.

For a more illustrative visualization of hidden layer activations, in Fig. 4.7(a), all 12000 data instances in the training set are represented by data points, by means that the 2-dim hidden activations in the bottleneck layer are adopted as the coordinates. Fig. 4.7(a) averages positions of the data points in the same class in Fig. 4.7(a). That is, Fig. 4.7(a) visualizes the averaged hidden activations as illustrated in Fig. 4.6(a) in 2-dim space.

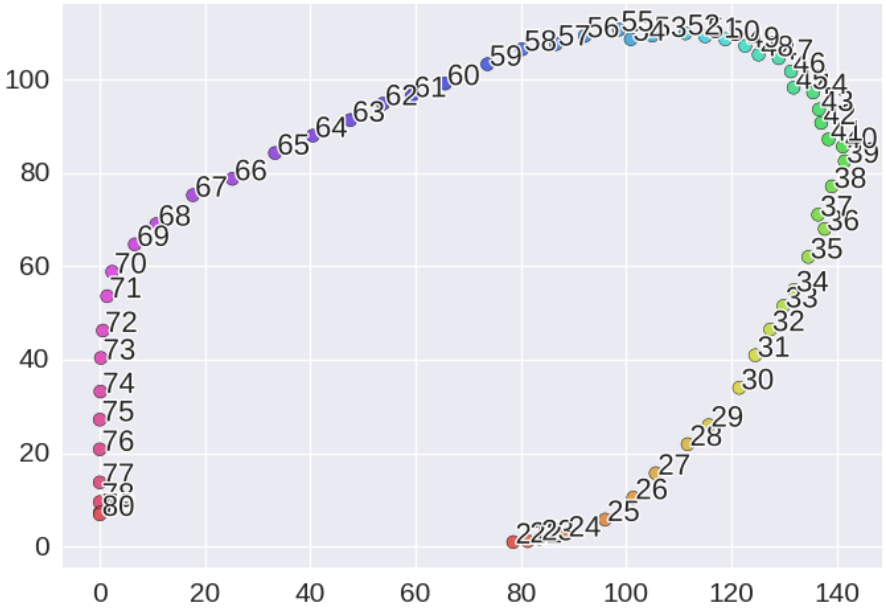
Recall that in the analysis of hidden activation visualizations in MLPs, the hidden activations are regarded as a way of encoding. Fig. 4.6(a) (illustrated in 2-dim space in Fig. 4.7) is a simplified case of such encoding with a vector of only two elements. Data points are distributed in the 2-dim space. Each class is located in a certain range of area and adjacent classes are located closer. The first neuron activation is zero in the last about 10 classes (jump location from 71 to 80). Therefore, the data points in these classes are all on the y axis in the figure because the x coordinate is zero. Similarly, the second neuron is deactivated at classes 21 to 23. The y coordinate of these points are zero. From the final output Fig. 4.6(b), the classes from 78 to 80 and the classes from 21 to 23 are the most poorly distinguished by the network. In each of the two cases, one coordinate of the class representations is zero, and the other coordinate of the classes is not far away from each other.

We also perform the same visualization of the 2-dim representations from the deep "bottleneck" network 100-25-2-25-60. The observations are similar.

Summary To summarize, the hidden layer activations are representations of the original data, i.e. each layer of the network maps the data from the input space or the activation space in the previous layer into the activation space of the current layer. The encoding of hidden activation is interpreted as the

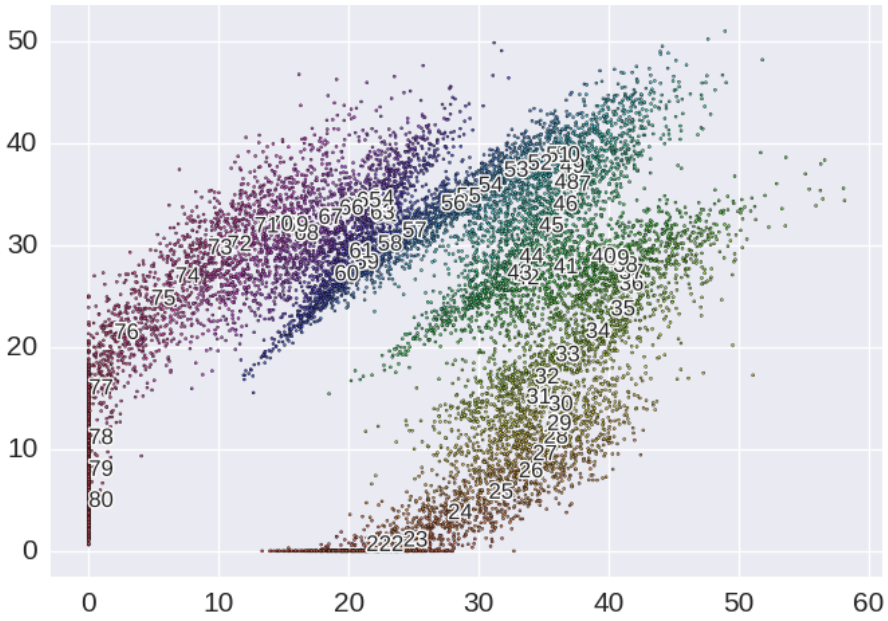


(a) single activations a of each data sample

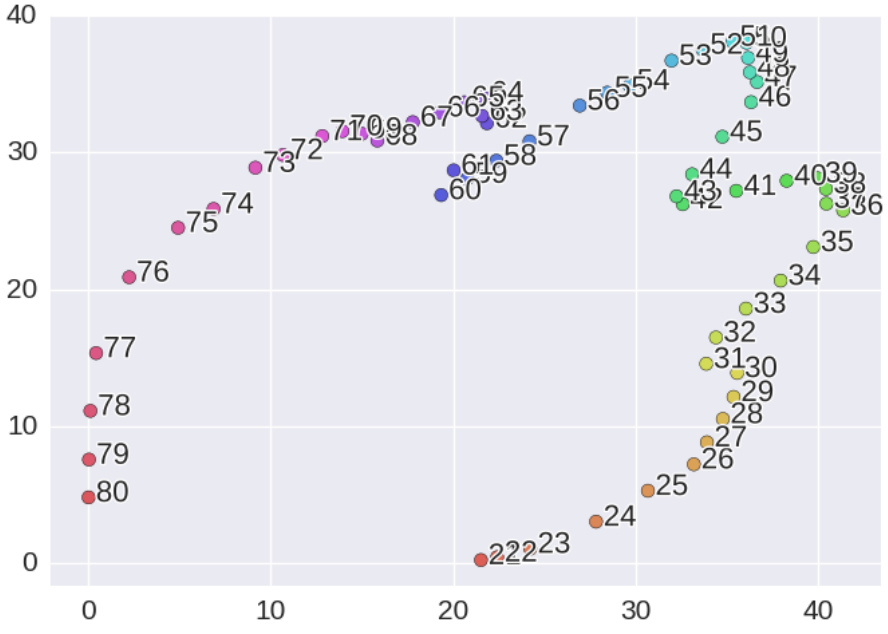


(b) averaged activations $A_{(c)}$ as class representations

Figure 4.7: Visualization of 2-dim representations from shallow "bottleneck" network structures 100-2-60.
(a): single activations a of each data sample;
(b): averaged activation A as class representations, i.e. each point is from the column vector in the matrix of Fig. 4.6(a).
The text in the graph denotes the classes (jump locations). The data points belongs to the same (similar) class are marked with the same (similar) colour(s).

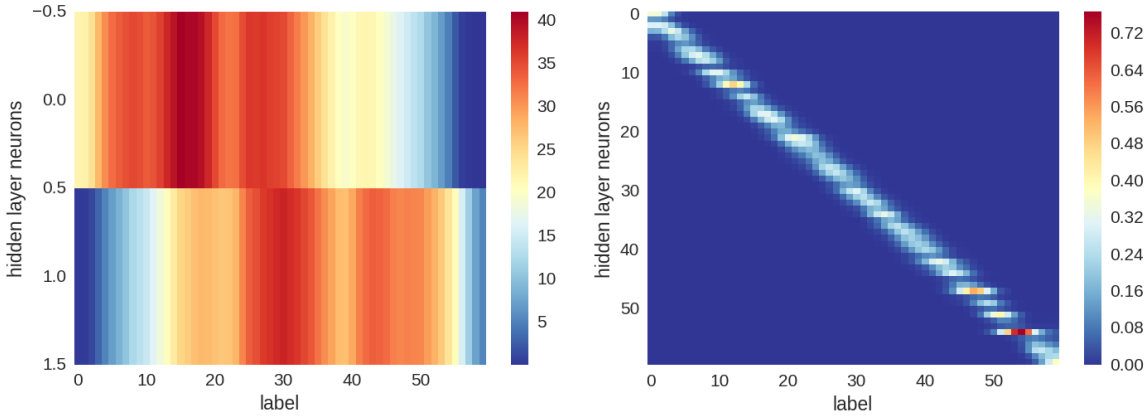


(a) single activations a of each data sample



(b) averaged activations $A_{(c)}$ as class representations

Figure 4.8: Visualization of 2-dim representations from deep "bottleneck" network structures 100-25-2-25-60. (a): single activations a of each data sample; (b): averaged activation A as class representations. The text in the graph denotes the classes (jump locations). The data points belongs to the same (similar) class are marked with the same (similar) colour(s).



(a) 2-dim representation of the data (averaged within each class): averaged hidden activation A^l in the bottleneck layer

(b) final output

Figure 4.9: Layer activations in 100-25-2-25-60. Only the bottleneck layer and the output layer are illustrated. Left is the bottleneck layer, in which the intermediate activation is regarded as a 2-dim representation. Right is the final output. The performance is relatively poor. Some adjacent classes are not well distinguished by the model.

coordinates in the layer activation space. The stripe-like figures in the averaged hidden representation indicate the fact that if the input data instances or classes are close to each other in the original input space, they usually remain close in the mapped layer activation space by most NN layers.

4.3 Theoretical Analysis

Empirically, the performance of the shallow "bottleneck" networks is acceptable even with a two-dimensional feature in the hidden layer, as shown in Fig. 4.1, which is surprising to us. Because the last layer of a shallow "bottleneck" network is a mapping from a low-dimensional representation to a high-dimensional one, and intuitively such a process would lose a lot of information. The following analysis in this section provides a theoretical view on the mapping function of the last layer of these networks.

4.3.1 Problem statement

The problem of locating a noisy unit-jump in a signal of length n can be solved (with surprising accuracy) using a "bottleneck" network of the following dimensions (see Fig. 4.10) :

1. Input layer of size (number of nodes) n;
2. hidden layer of size p with ReLU activation;
3. output layer of size n with softmax.

The subsequent analysis investigates how this can be accomplished within the given framework. We are assuming that the network has been trained as indicated in the previous chapters. In most of the examples below we will assume that the bottleneck is really tight and put $p = 2$, but this is simply for ease of graphical representation.

Formalisation To understand how the observed (remarkably good) performance is achieved we feed the trained network with noise-less input signals, one for each location. We denote by $\mathbf{u}_k$ the noise-less signal that has a (upward) jump at location k:

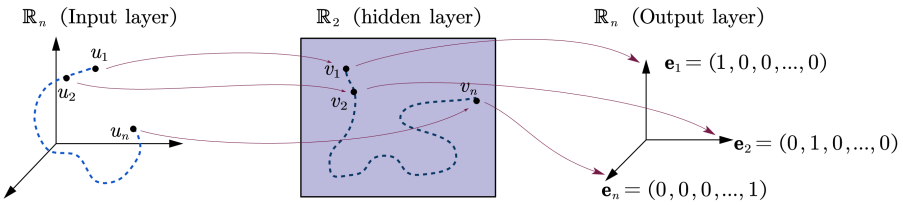


Figure 4.10: Abstract view of network-based classification.

$$\mathbf{u}_k(i) = \begin{cases} 0 & \text{if } i < k \\ 1 & \text{if } i \geq k. \end{cases}$$

If we dispense with the mild non-linearities in the activation of the nodes, the mapping from the input layer (of size n) to the first hidden layer (of size 2) basically amounts to a linear (or affine) mapping $f_1 : \mathbb{R}^n \longrightarrow \mathbb{R}^2$.

From input to hidden layer A network that has been successfully trained will map these inputs to well-separated points in (a compact square of) $\mathbb{R}^2$. In fact, if we order the inputs according to the jump location, we can think of them as points on a curve in $\mathbb{R}^n$ which is mapped to a self-avoiding curve in $\mathbb{R}^2$ (see Fig. 4.10). Numerical experiments have shown (c.f. section 4.3.4) that the performance of the network depends crucially on the way in which the mapping from the input to the hidden layer is able to map exemplars of the different classes to distinct and well-separated points in $f_1(\mathbf{u}_k) = \mathbf{v}_k \in \mathbb{R}^2$.

From hidden layer to output Since the output is encoded in one-hot fashion, the mapping from hidden to output should map $\mathbf{v}_k \equiv f_1(\mathbf{u}_k)$ to a point which is close to the standard unit vector $\mathbf{e}_k = (0,\ldots,1,\ldots,0)$, i.e. the k-th entry is close to 1, while all other entries are relatively small. That this can indeed be achieved using a linear (or affine) transformation follows from the following consideration.

Rather than looking at the map $f_2 : \mathbb{R}^2 \longrightarrow \mathbb{R}^n$, we first focus on the inverse mapping. It is straightforward to define a linear transformation that maps each unit vector $\mathbf{e}_k$ into the appropriate 2-dim point $\mathbf{v}_k$. Since this mapping is surjective, it cannot be uniquely inverted, but we can use the generalized Moore-Penrose inverse to get a first version of the mapping f_2. In the section below we explore how this initial mapping can be improved upon to arrive at the final solution. It will become clear that in this last step the soft-max non-linearity plays a crucial role.

Taking stock Summarizing, we can say that it looks like the auto-encoder (or bottleneck) structure achieves its performance by

1. constructing a linear mapping $f_1 : \mathbb{R}^n \longrightarrow \mathbb{R}^2$ which maps the different examplars into well-separated 2-dim points. The construction of this mapping is straightforward, as a linear mapping is completely and uniquely determined by specifying the result on a basis;

2. constructing a mapping $f_2 : \mathbb{R}^2 \longrightarrow \mathbb{R}^n$ which maps the points in the plane back to unit-vectors in $\mathbb{R}^n$ (one-hot encoding for output). To construct this mapping it is easier to start from the inverse for which we can apply the same techniques as in the first case.

Auto-encoder as feature extractor? Although this is a simple example it does provide us with some insights that are useful when thinking about networks.

- From the above analysis it transpires that there are many possible solutions for a given problem. It suffices to specify a well-behaved curve in the 2-dim space (representing the hidden layer) to obtain a well-performing neural net. Ideally, the curve should not be self-intersecting and self-avoiding. Any such solution (of which there are uncountably many) would function well as a starting point.

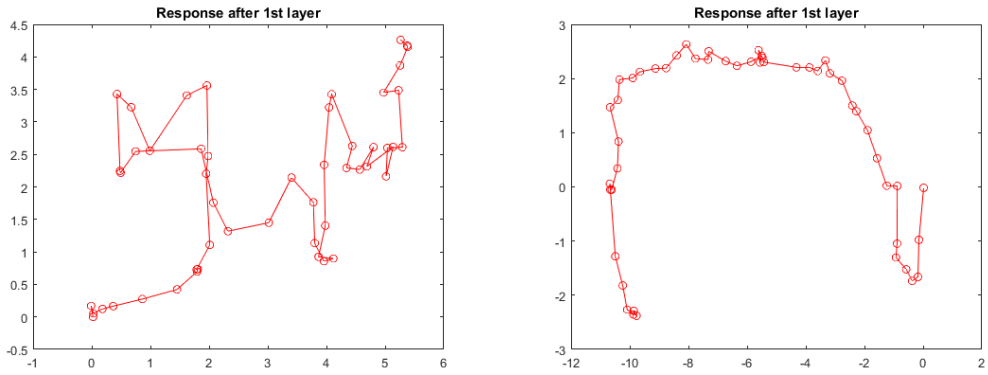


Figure 4.11: Left: Poor embedding in 2-dimensional space. The self-intersections and scrunching of points are clearly visible. Right: better embedding without self-intersections.

- This also explains why it is relatively difficult to get good performance with an auto-encoder of size 2, but it becomes easier when the number of nodes in the hidden layer is increased. This amounts to constructing such a self-avoiding curve in 3-dim (or higher) space, which is easier to do. It also explains why having more than 3-5 nodes in the hidden layer does not necessarily improve performance. Indeed, a 3-5 dimensional space already has plenty of room to accommodate such an encoding curve.

- It is often stated that auto-encoders extract underlying hidden features. This might be misleading and over-stated. Looking at the problem at hand the encoding of the jump in the 2-dim space is totally arbitrary, and does not really tell us anything useful about the underlying feature.

4.3.2 Explicit construction of map from hidden layer to output: Toy problem

When mapping from the hidden layer to the output we have to use a linear (or affine) map from a low dimensional space to a high dimensional one. In this paragraph we describe this process in more detail.

To keep things simple we will focus on a mapping from two to three- dimensional space $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^3$. We pick 3 arbitrary points in the 2-dim unit square and need to determine a transformation which maps these points close to the corresponding unit vectors in 3-dim space.

Suppose we have three random points in $\mathbf{v}_k \in \mathbb{R}^2\,(k = 1 \ldots 3)$ for which we want to construct an affine function

$$A : \mathbb{R}^2 \longrightarrow \mathbb{R}^3 : \mathbf{x} \longmapsto W\mathbf{x} + \mathbf{b}$$

such that each of the points is mapped *close* to its corresponding unit vector $\mathbf{e}_k$ (also see Fig. 4.12). In this context, *close* means that a one-hot representation based on (soft)max selection would yield the unit-vector.

To accomplish this we proceed as follows:

- First, compute the mean of the three points:

$$\mathbf{m} = \frac{1}{3}(\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3)$$

  and subtract this mean from the original 2-dimensional points: $\mathbf{w}_k = \mathbf{v}_k - \mathbf{m}$. Hence, without loss of generality we can assume that the three points are centered about the origin.

- Next, construct the unique linear mapping $F : \mathbb{R}^3 \longrightarrow \mathbb{R}^2$ that maps the unit vectors to these centred points, i.e. $F(\mathbf{e}_k) = \mathbf{w}_k$. The matrix of this linear transformation (which, with slight

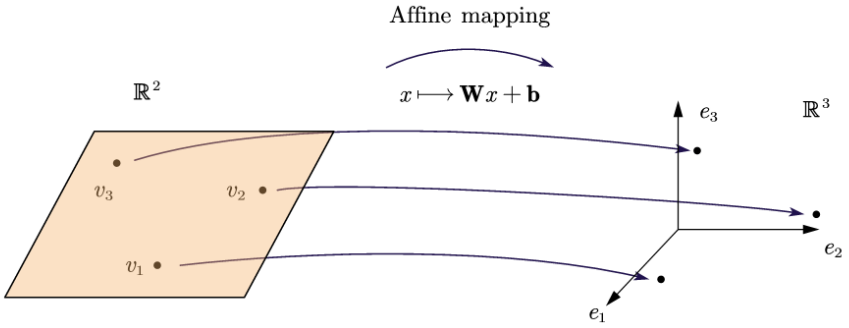
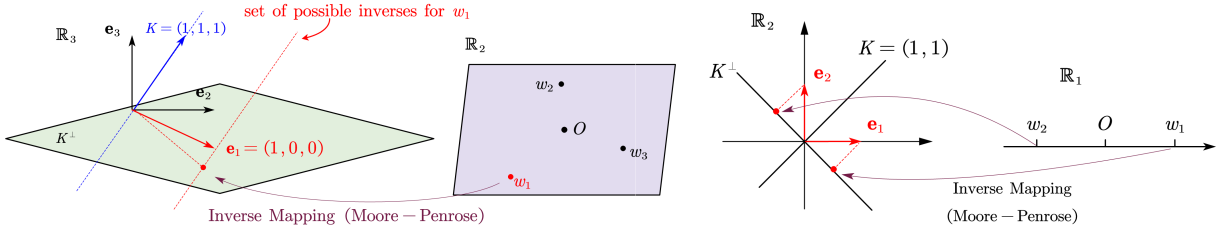


Figure 4.12: Problem: creating an affine embedding



(a) 2-dim to 3-dim

(b) 1-dim to 2-dim

Figure 4.13: (a): Construction of the inverse mapping using the Moore-Penrose inverse.
(b): The same construction for the simpler case of computing the MP-inverse for a linear mapping from 2-dim to 1-dim.

abuse of notation, we will also denote by F) is obtained by plugging each $\mathbf{w}_k$ into the k-th column. In shorthand:

$$F = (\mathbf{w}_1 \quad \mathbf{w}_2 \quad \mathbf{w}_3) \in \mathbb{R}^{2\times 3}$$

Notice that the fact that $\mathbf{w}_1 + \mathbf{w}_2 + \mathbf{w}_3 = \mathbf{0}$ implies that the (1-dim) kernel (null space) of F is spanned by the principal diagonal vector $(1,\ 1,\ 1)$.

- Construct the Moore-Penrose inverse F^+ for F which maps each $\mathbf{w}_k$ back into $\mathbb{R}^3$ in such a way that among all possible inverses the minimum norm is selected. Geometrically, this amounts to selecting as inverse image of $\mathbf{w}_k$ the projection of $\mathbf{e}_k$ on the plane $K^\perp$ orthogonal on K (see Fig. 4.13). The result of this Moore-Penrose inversion will indeed map each point $\mathbf{w}_k$ into the vicinity of the corresponding unit-vector. This can be seen by observing that $F^+(\mathbf{w}_1)$ is obtained by projecting $\mathbf{e}_1$ orthogonally on the plane $K^\perp$. This can be done by subtracting the component parallel to K. Recall that K is generated by the unit vector $\mathbf{k} := (\mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3)/\sqrt{3}$, which means that we can write:

$$F^+(\mathbf{w}_1) = \mathbf{e}_1 - <\mathbf{e}_1, \mathbf{k}>\ \mathbf{k} = (1 - 1/\sqrt{3})\mathbf{e}_1 - (\mathbf{e}_2 + \mathbf{e}_3)/\sqrt{3}.$$

  Clearly, only the first coefficient is positive, all the other ones are negative, which means that the first coefficient is automatically the maximum, or again, has a one-hot representation equal to $(1,0,0)$ as claimed.

- Putting all of the above together we see that the affine mapping takes on the following form:

$$A(x) = F^+(\mathbf{x} - \mathbf{m}) = W\mathbf{x} + \mathbf{b} \qquad \text{where} \quad W = F^+ \quad \text{and} \quad \mathbf{b} = -F^+\mathbf{m}.$$

4.3.3 Explicit construction of map from hidden layer to output: General case

Problem Framework

We consider the general case in which we need to construct an affine mapping from a low-dimensional space $\mathbb{R}^p$ to a high-dimensional space $\mathbb{R}^n$:

$$A : \mathbb{R}^p \longrightarrow \mathbb{R}^n : \mathbf{x} \longmapsto W\mathbf{x} + \mathbf{b},$$

where W is a $n \times p$ matrix, while $\mathbf{b}$ is $n \times 1$. Notice that this affine transformation can be recast as a linear one by applying the standard embedding:

$$A(\mathbf{x}) = \begin{pmatrix} W & \mathbf{b} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}.$$

As stated above, we are interested in finding a mapping that maps the n points $\mathbf{v}_k$ (where $k = 1, \ldots, n$) to the n unit vectors $\mathbf{e}_k$ in $\mathbb{R}^n$:

$$\begin{pmatrix} W & \mathbf{b} \end{pmatrix} \begin{pmatrix} \mathbf{v}_k \\ 1 \end{pmatrix} = \mathbf{e}_k$$

Recall that in this case the entries of W and $\mathbf{b}$ are the unknowns, whereas the $\mathbf{v}_k$ are given. Hence, to write this in the standard form, we transpose the above equation:

$$\begin{pmatrix} \mathbf{v}_k^T & 1 \end{pmatrix} \begin{pmatrix} W^T \\ \mathbf{b}^T \end{pmatrix} = \mathbf{e}_k^T$$

Since this equation has to hold (at least approximately) for every point $\mathbf{v}_k$, we can stack the different equations on top of each other to obtain:

$$\begin{pmatrix} \mathbf{v}_1^T & 1 \\ \mathbf{v}_2^T & 1 \\ \vdots & \vdots \\ \mathbf{v}_n^T & 1 \end{pmatrix} \begin{pmatrix} W^T \\ \mathbf{b}^T \end{pmatrix} = \begin{pmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_n^T \end{pmatrix} \equiv I_n. \tag{4.1}$$

Denoting the coefficient matrix in left hand side by V, we see that it has dimensions $n \times (p+1)$. The matrix of unknowns (the coefficients of the affine transformation) has dimensions that the product has indeed a size equal to $n \times n$. Notice that since the matrix of unknowns has n columns, eq. 4.1 actually represents n systems of equations of the form:

$$\begin{pmatrix} \mathbf{v}_1^T & 1 \\ \mathbf{v}_2^T & 1 \\ \vdots & \vdots \\ \mathbf{v}_n^T & 1 \end{pmatrix} \begin{pmatrix} W_i^T \\ b_i \end{pmatrix} = \begin{pmatrix} \mathbf{e}_i^T \end{pmatrix} \equiv (I_n)_i.$$

where the subscript i refers to the $i-th$ column. As a consequence we see that eq. 4.1 actually represents a total of n^2 equations for $n \times (p+1)$ unknown affine parameters. So unless $n = p+1$ (as in the case above) this system is overdetermined and we can only get an approximate solution.

Least squares solution By slight abuse of notation we collect the parameters for the affine transformation into the matrix $A = (W\ \mathbf{b})$, whereupon the above equation can be written as:

$$VA^T = I_n.$$

The least squares solution for this over-determined system is obtained by the standard procedure:

$$VA^T = I_n \quad \Longrightarrow \quad V^T V A^T = V^T \quad \Longrightarrow \quad A^T = (V^T V)^{-1} V^T.$$

or again, after transposition:

$$\begin{pmatrix} W & \mathbf{b} \end{pmatrix} = V(V^T V)^{-1}$$

Notice that this implies that original points $\mathbf{v}_k$ are mapped into the columns of the matrix

$$AV^T = V(V^T V)^{-1} V^T. \tag{4.2}$$

Positioning of points in low-dim space Eq. 4.2 specifies how the images of the $\mathbf{v}_k$ deviate from the ideal solution $\mathbf{e}_k$ as in general

$$V(V^T V)^{-1} V^T \neq I_n.$$

Notice that this does not mean that the classification is necessarily incorrect as the soft-max will map it to the correct unit vector as long as the dimension of maximum value corresponds to that of the correct unit vector.

4.3.4 Constrained embedding into higher dimensional space: Experimental exploration

In the previous section we have seen how an NN with a low-dimensional bottleneck is forced to map high-dimensional data points into a low-dimensional space (see Figs 4.8). However, in itself this is not a difficult problem. Indeed, it is straightforward to construct a linear (or affine) mapping that maps a basis in a high (n-)dimensional space onto n points in a lower dimensional space.

However, the mapping from the (last) hidden layer to the output actually amounts to solving the the reverse problem which is much more challenging. In fact, it requires mapping n pre-determined points in a low-dimensional space into a high-dimensional space in such a way that the points end up near the corresponding unit-vectors.

Experiments

To better understand how difficult this mapping problem is we performed the following experiment. We start by assuming that we have n random points $\mathbf{v}_k$ in a space of low dimension p which we need to map into high-dimensional space of dimension n (hence $p < n$). Moreover, the mapping is the composition of a linear mapping L followed by a hard-max non-linearity $h : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, i.e. h is the indicator function for the position of the max in the n-tuple $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. As an example:

$$h(3, -2, 1.2, 5, -2) = (0, 0, 0, 1, 0).$$

Given the n points $\mathbf{v}_k \in \mathbb{R}^p$, we need to construct a linear transformation

$$L : \mathbb{R}^p \longrightarrow \mathbb{R}^n$$

such that $L(\mathbf{v}_k)$ is closer to the k-th unit vector $\mathbf{e}_k \in \mathbb{R}^n$ than to any other unit vector. More formally:

$$h(L(\mathbf{v}_k)) = \mathbf{e}_k \qquad (\forall k = 1, \ldots, n).$$

To appreciate the difficulty of this problem, recall that the original points all live in a p-dimensional space and are therefore mapped by the linear (or even affine) transformation into a p-dimensional hyperplane within the n-dimensional target space. Put differently, the affine transformation has to position the p-dimensional hyperplane into the higher dimensional space in such a way that the images of the random points $\mathbf{v}_k$ are now located near the corresponding unit vectors.

Experimental Procedure

- Generate n random points $\mathbf{v}_k$ in $\mathbb{R}^p$, drawn from a random distribution. For the precise definition of the distribution used, see below.

- Next construct the unique linear map $F : \mathbb{R}^n \longrightarrow \mathbb{R}^p$ that maps the unit vectors $\mathbf{e}_k$ onto the points $\mathbf{v}_k$:

$$F(\mathbf{e}_k) = \mathbf{v}_k.$$

  As explained above, this is a trivial application of linear algebra, but it goes in the opposite direction of what we want to accomplish. However, we use F as a starting point go guide the inversion.

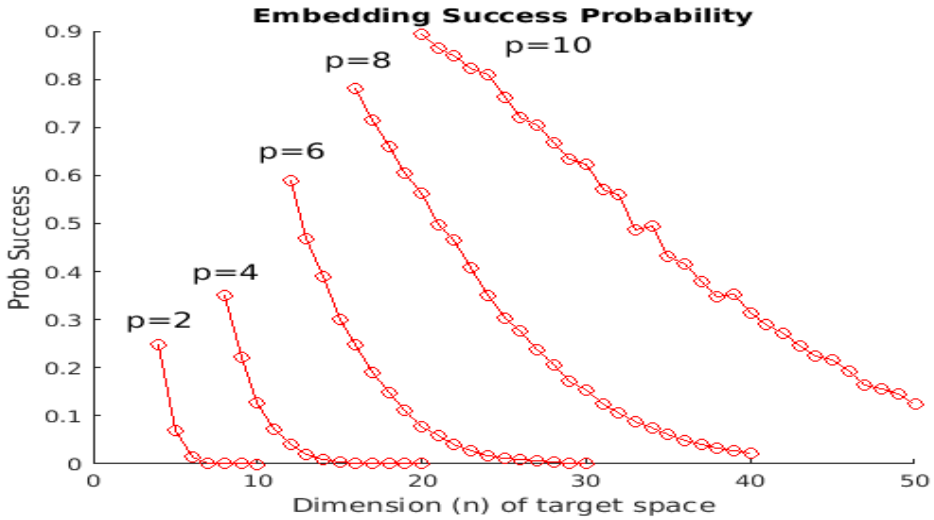
- Next, (pseudo-)invert the mapping F by computing the Moore-Penrose (pseudo)-inverse L of F. Recall that this is a linear transformation that in some precise sense, provides the best generalisation of an inverse for the non-invertible linear mapping F.

- Subsequently, check whether the hard max yields the required result, i.e. $h(L(\mathbf{v}_k)) = \mathbf{e}_k$ for every $k = 1 : n$. If so, count this as a success, if not count it as a failure.

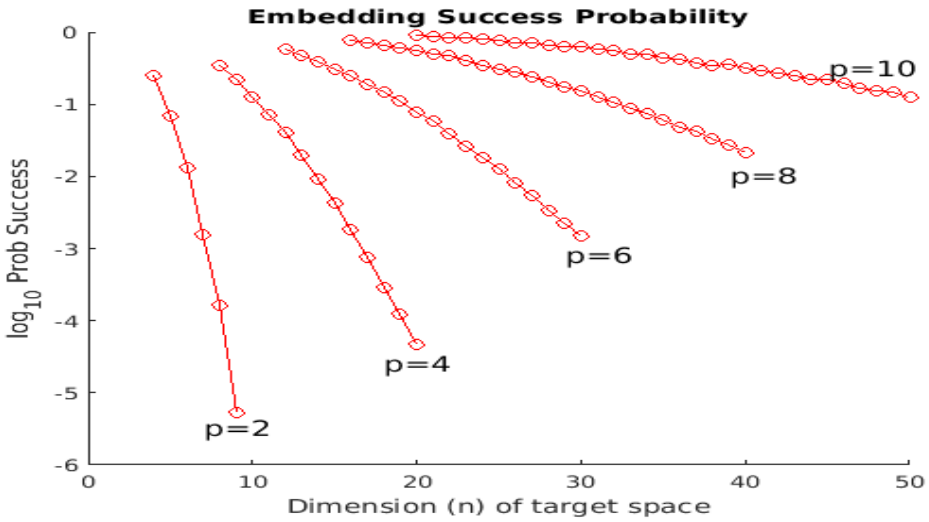
- After a sufficient number of repetitions we get an estimate for the success rate with which the Penrose inverse accomplishes the required goal of mapping each point to the correct one-hot classification. This is interpreted as an indication of the difficulty of the mapping problem.

Experimental Results

In a first experiment we explored how the probability of a successful inversion (or embedding) depends on the dimension p of the bottleneck space. The results are shown in Fig. 4.15 which shows the result for 5 different values of $p = 2 : 2 : 10$. As is to be expected, the tighter the bottleneck (small values for p) the more difficult it becomes to create a linear (using the PM-inverse) mapping that maps n randomly chosen exemplars to the correct one-hot classification. For instance, if the bottleneck dimension $p = 2$ then the probability of creating a linear map that assigns the correct class to $n = 10$ randomly positioned 2-dim points is less than 10^{-5}, indicating how difficult his problem is for a NN to solve. In contrast, for $p = 10$ even classifying $n = 50$ points is fairly straightforward and the PM-inverse has probability exceeding 0.1 of getting it right.



(a) probability of successful embedding



(b) log-prob of successful embedding

Figure 4.14: Tight bottlenecks make successful classification more difficult. The graphs above show the probability (a) and log-prob (b) of successful embedding in an output space which is 2 to 5 times larger than the bottleneck space (i.e. $n = 2p, \ldots, 5p$). Clearly, for each of the curves, the probability decreases as n increases which is to be expected. However, scaling up (between bottleneck and output) by a factor of 5 is much easier for the wider bottlenecks ($p = 8, 10$) than for the tighter ones ($p = 2, 4$).

In a second experiment we tested the intuition that the embedding or inversion problem becomes easier if the exemplar points are sufficiently well separated in the low-dimensional bottleneck space. The results are shown in Fig 4.15. In this experiment, we fixed the bottleneck dimension $p = 2$, i.e. all input is mapped to a 2-dim space, from which it is mapped to the output space of dimension n. We investigated $n = 5, \ldots, 8$ as for larger values the probabilities are very small. We used three sampling schemes to draw the n sample points:

1. Standard uniform on the square $[0, 1] \times [0, 1]$
2. Same as in 1, but points have a minimum mutual distance of $r = 0.1$;
3. Same as in 1, but points have a minimum mutual distance of $r = 0.2$;

The intuition is that by insisting on a minimum separation r between points, it becomes easier to correctly embed the points in the higher dimensional output space. This intuition is borne out by the experiments that show how the probabilities improve by ensuring better separation between the exemplars in the low-dimensional bottleneck space.

4.3.5 Summary

Based on the experimental and theoretical exploration in this section we draw the following intermediate conclusions:

- Although a tight bottleneck reduces that number of parameters that need to be trained, it should not be too tight as it then becomes very difficult to construct the correct expansion to the (high-dimensional) output space. That being said, it should not be expanded unnecessarily as the additional neurons do not significantly contribute to the capacity of the network. In fact, some of them might simply switch off (as can be seen in e.g. Fig 4.5).
- Mapping the (high-dimensional) input exemplars into the low-dimensional bottleneck space is straightforward. The challenge is to re-expand this low-dimensional space to the correct one-hot

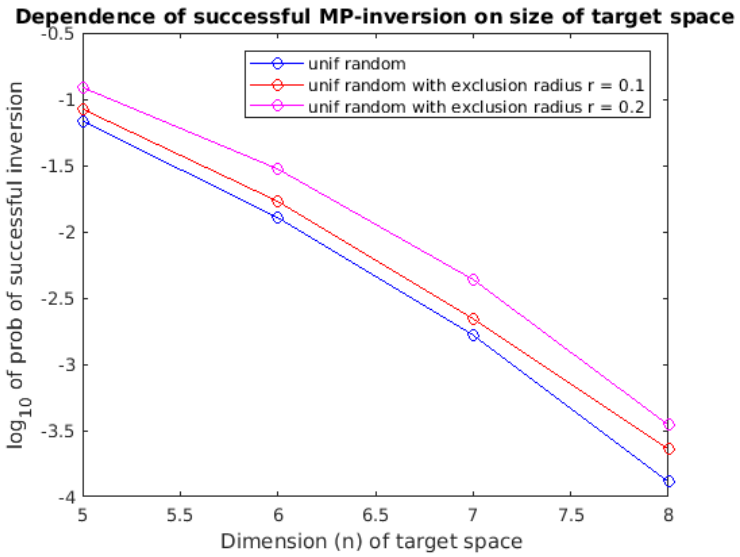


Figure 4.15: How does the degree of separation of the exemplar points in the low-dimensional (bottle-neck) space (dimension $p = 2$) influence the probability of successful embedding in the high-dimensional output space (dimension $n = 5, \ldots, 8$)? We show the results of three sampling schemes: standard uniform (blue), uniform but with guaranteed minimal distance of $r = 0.1$ (red) or $r = 0.2$ (magenta). Clearly, ensuring better separation between the points in the low-dimensional bottleneck space increases the probability of successful embedding.

label (i.e. unit vectors) in the (high-dimensional) output space. The limited experiments reported in Fig. 4.15 suggest that it helps if the initial projection to the bottleneck layer ensures that the exemplars are well-separated. We will encounter a similar conclusion in section 6.4 where we will show that during learning the NN tends to contract inputs with the same label.

- The graphs in Fig. 4.14 can also be seen as suggesting an appropriate architecture for the NN. For instance, if we have a problem for which we need to distinguish 40 or 50 classes, the graph show that a hidden layer of size $p = 10$ would provide sufficient capacity for easy training and generalisation. Conversely, a NN with a bottleneck of size $p = 2$ would be difficult to train.
- Finally, although we have not confirmed this experimentally, the theoretical analysis also suggests that the Moore-Penrose inverse might provide good initial estimates for the weight matrix. This might therefore reduce the computational burden and speed up training.

Cautionary note Most of the arguments and experiments in this section have been built on the idea of using the Moore-Penrose inverse to map the low-dimensional representation to the high-dimensional output. However we cannot claim that this is the only or indeed the best option for doing so, as this point needs further research. So the results in this section are tentative and should be seen as a first step towards more thorough investigations.

Chapter 5

Visualization by Gradient Ascent

Contents

The visualization approach by Gradient Ascent [35, 36] is applicable for interpretation of NNs in the jump location detection task.

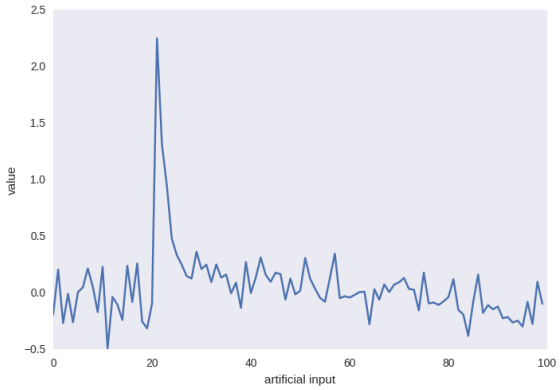
We use this technique to generate a typical signal input for a certain class (jump position). First, an artificial input vector x is generated, and each element in the artificial input is initialized randomly, uniformly in range $[-0.05 : 0.05]$. As the input to the network, it causes an activation $a_i(x)$ at an output neuron i. We now regard this as an optimization problem that aims to find an input x^* that gives highest activation $a_i(x)$. Within a gradient ascent framework, the x^* can be obtained from the original input x by iterations with

$$x \leftarrow x + \alpha \frac{\partial a_i(x)}{\partial x},$$

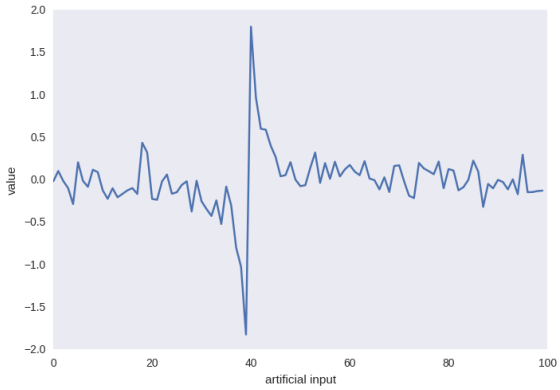
where α is the learning rate. In the experiments, we set α equal to 10. The iteration stops when the activation level of an output neuron is over 0.999, i.e., by the network, the artificial input is predicted as in the class of the output neuron with a probability over 99.9%.

5.1 For MLPs

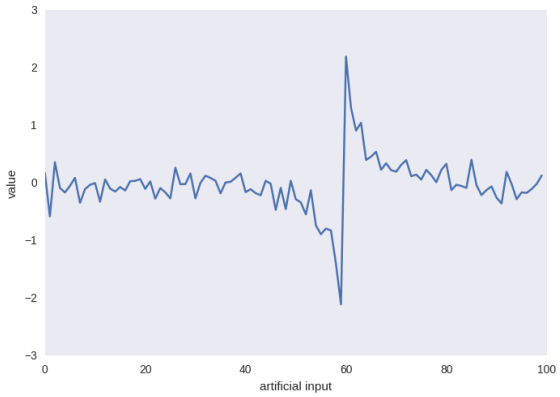
We start observing the results from the simplest network, with only one fully connected layer (Fig. 5.1).



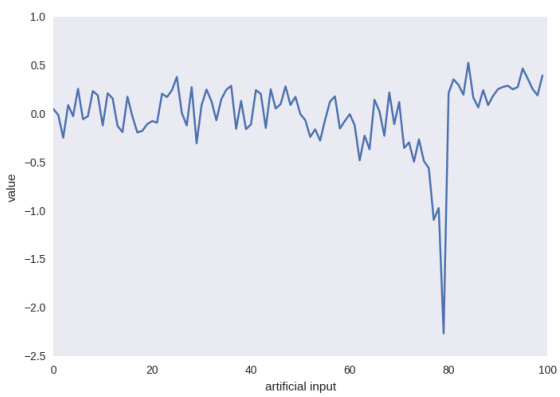
(a) 1-layer NN: Jump location 21



(b) 1-layer NN: Jump location 40



(c) 1-layer NN: Jump location 60

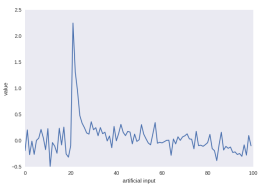


(d) 1-layer NN: Jump location 80

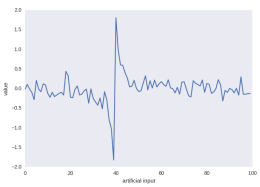
Figure 5.1: 1-layer NN: Gradient ascent from random artificial input vector.
Output neuron activations of jump positions at 21, 40, 60 and 80 are maximized.

Although the resulting inputs are not exactly the same as the original input time series fed to the network, there obviously appear discontinuities at the corresponding positions. For jump positions at 40 and 60, the signal drops first, then followed with a sharp sudden jump, finally it drops to a level slightly higher than before. There are two peak values in each signal. While for jump position at 21, there is no drop before the signal jumps, and symmetrically, for jump position at 80, there is no drop after the jump. In these cases there is only one peak in each signal.

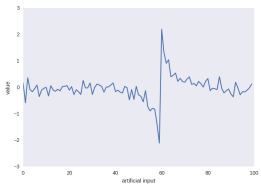
We further investigate deeper networks, with 2, 3 and 4 layers respectively. The comparative illustration is in Fig. 5.2



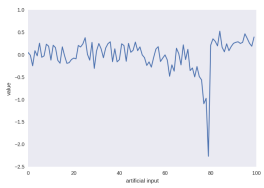
(a) 1-layer NN: Jump location 21



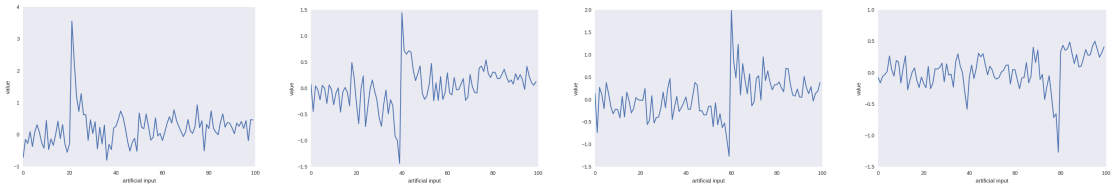
(b) 1-layer NN: Jump location 40



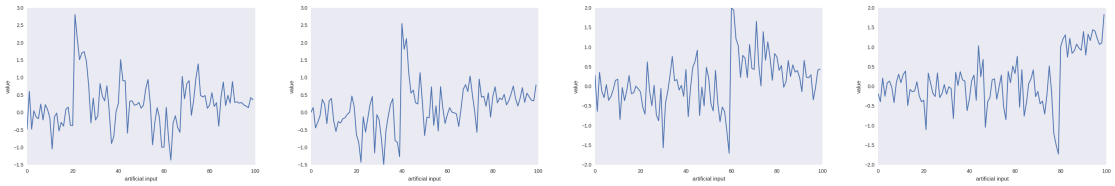
(c) 1-layer NN: Jump location 60



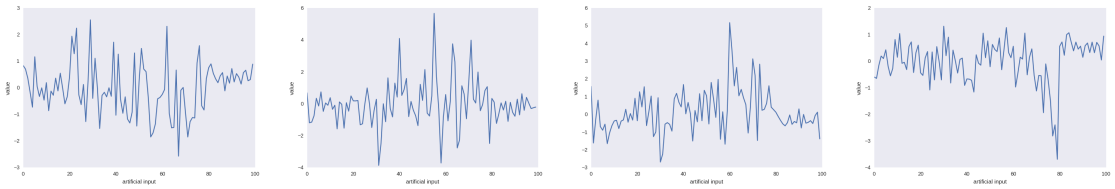
(d) 1-layer NN: Jump location 80



(e) 2-layer MLP: Jump location 21 (f) 2-layer MLP: Jump location 40 (g) 2-layer MLP: Jump location 60 (h) 2-layer MLP: Jump location 80



(i) 3-layer MLP: Jump location 21 (j) 3-layer MLP: Jump location 40 (k) 3-layer MLP: Jump location 60 (l) 3-layer MLP: Jump location 80



(m) 4-layer MLP: Jump location 21 (n) 4-layer MLP: Jump location 40 (o) 4-layer MLP: Jump location 60 (p) 4-layer MLP: Jump location 80

Figure 5.2: Gradient ascent from random artificial input vector.
Output neuron activations of jump positions at 21, 40, 60 and 80 are maximized.
Figures in the same row are from the same model.

Despite the similar phenomena, as the network goes deeper, the resulting artificial inputs are more blurred with noise. In a 4-layer network, in some cases (jump positions at 21 and 40), the discontinuity is not even recognizable and locate-able from the artificial input. However, these signals are still recognized by the network with 99.9% confidence as with a sudden jump at corresponding positions. For our methodology, this phenomenon could be explained by gradient vanishing. It may also lead to an empirical conclusion that a deeper network is easier to be attacked.

Also, the performance is related to the clarity of the generated artificial input. As in Table 3.3, models with less layers are with both lower error rates and clearer generated artificial signals.

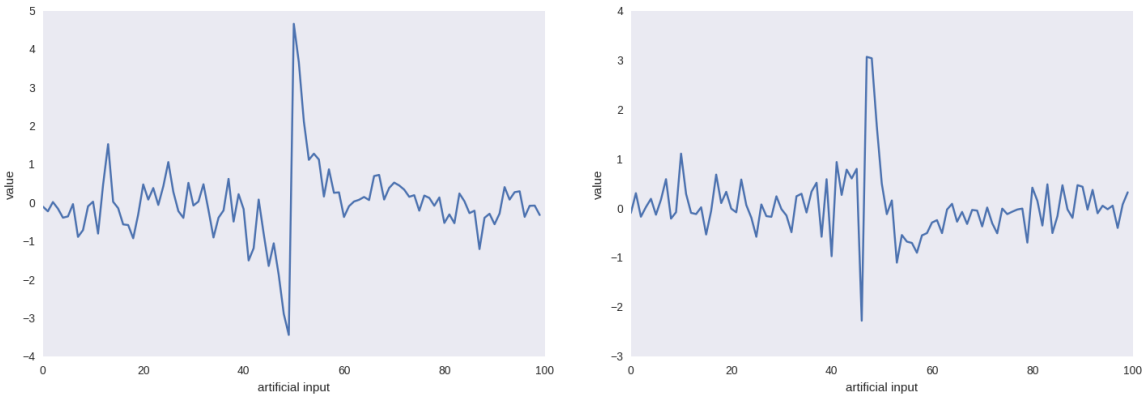
5.2 For CNNs

We also perform visualization with gradient ascent for CNN models. As the output of the convolutional layers keeps the spatial information from the input of the layer, it is also reasonable to visualize the intermediate signals after the convolutional layers.

Empirically, in CNN models the jump-position-wise comparative phenomena are similar to that in a feedforward network. Thus, in this section, we provide layer-wise comparative observations. We take a specific jump position 50 which is near the center of the signal which has generality.

5.2.1 With one conv layer

The visualization as in Fig. 5.1 is performed on the CNN model with one conv layer. See Fig. 3.23 for more details of the model. We only keep the resulting graphs for one specific jump position at 50 as a representative, since the main purpose is to compare the visualization from different (intermediate layers). Fig. 5.3(a) is the artificially generated input that maximizes the output neuron which stands for jump position at 50. While Fig. 5.3(b) is the signal as the output of the conv layer and the input of the fully connected layer.

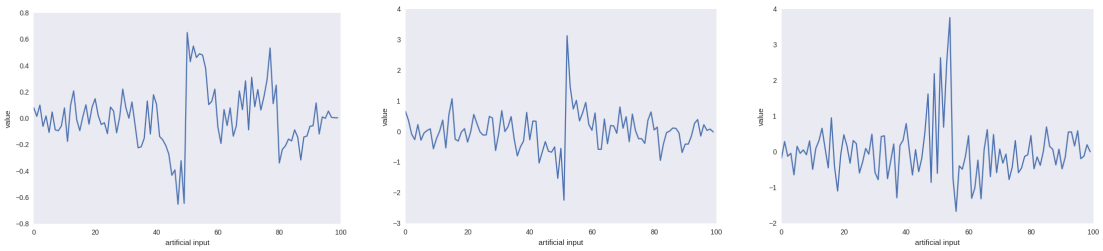


(a) Original input (b) After convolutional layer

Figure 5.3: The generated artificial input and the intermediate signal after convolutional layer for a CNN model with one convolutional layer. Activation maximized for jump position at 50.

Empirically, in the right figure (after the conv layer), after the high peak, the signal drops down to a bit under the base level, then raises back to that normal level. Recall the visualization of hidden layer activation in the same model, same layer (Fig. 3.18(c)), the intermediate activation is also with a drop after the peak, compared to the original input. Thus, the generated artificial signal is coherent to the hidden layer activation.

5.2.2 With two conv layers



(a) Original input (b) After first convolutional layer (c) After second convolutional layer

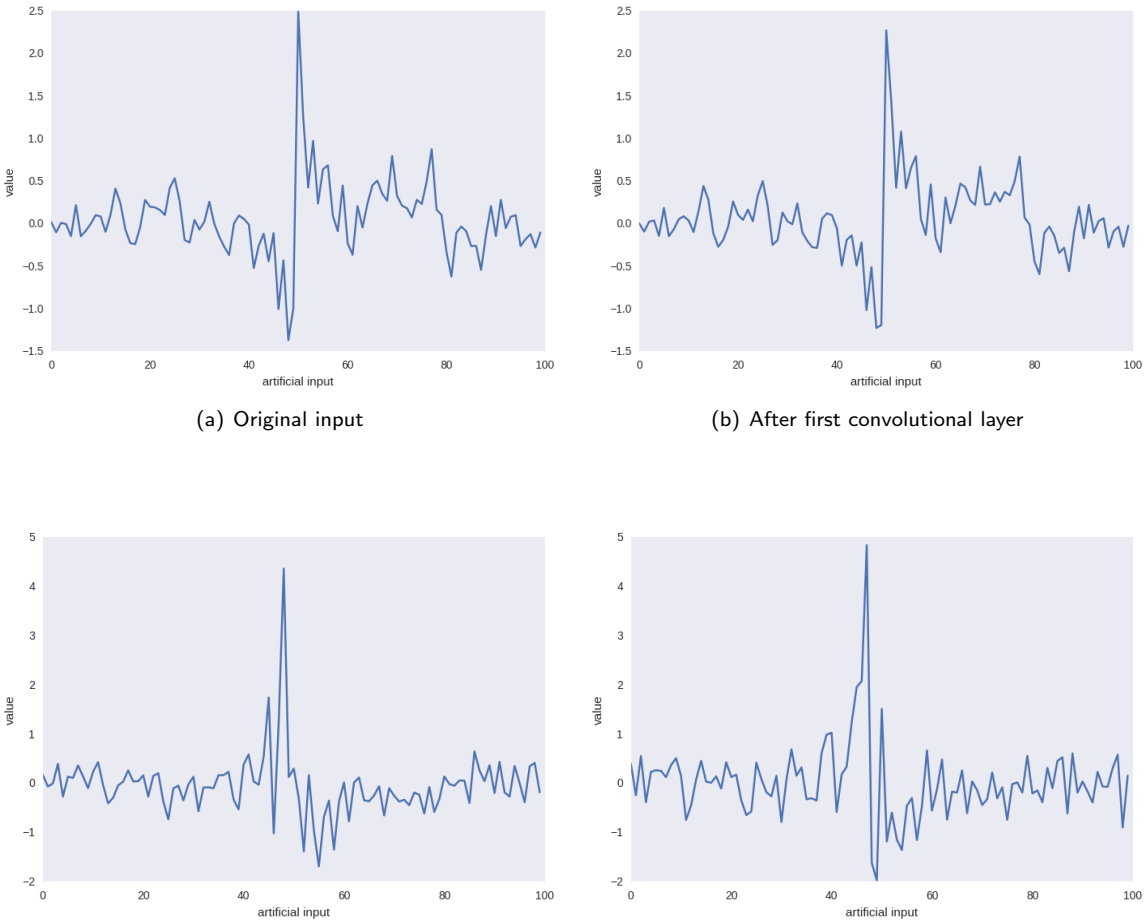
Figure 5.4: The generated artificial input and intermediate signals after all convolutional layers for a CNN model with two convolutional layers. Activation maximized for jump position at 50.

The detailed information of the corresponding trained model is illustrated in Fig. 3.21. The observation is almost the same as in the model with one conv layer.

1. The generated signals has sharper peaks in deeper layers that are closer to the final output.

2. Signals in the first and the second graph almost has the same shape. Recall that in this model, the first conv layer only increases the gap of the jump (Fig. 3.21(c)), which is a natural explanation for the similar shape and thinner peak in the generated signal. The signal in the third figure has a clear serrated part before the peak, which is also relevant to the shape of the hidden layer activation (Fig. 3.21(f)).

5.2.3 With three conv layers



(a) Original input

(b) After first convolutional layer

(c) After second convolutional layer

(d) After third convolutional layer

Figure 5.5: The generated artificial input and intermediate signals after all convolutional layers for a CNN model with three convolutional layers. Activation maximized for jump position at 50.

Generally, rules are the same as in the models with less layers, with some details being slightly different:

1. Generally speaking, the closer the layer is to the output, the sharper peak the generated signal has. While there is no significant difference between the last two signals (before and after the third layer). Recall that in the visualization of hidden layers, the third layer seems to have no obvious effect on the signal. The additional conv layer is not that necessary.

2. Signals in the first and the second graph almost has the same shape. Recall that in this model, the first conv layer only increases the gap of the jump (Fig. 3.23(c)), which is a natural explanation for the similar shape and thinner peak in the generated signal. The signal in the third figure

has a clear serrated part before the peak, which is also relevant to the shape of the hidden layer activation (Fig. 3.23(i)).

5.3 Summary: What are the Networks Actually Learning?

From gradient ascent visualization, the results can be concluded as the following three observations:

1. The generated artificial inputs shows discontinuity at the location at which the output neuron is supposed to peak. However, they do not have the same shape as the data fed to the network in the training session. The networks are trained on signals that are steplike, i.e. they have an average value of 0 before jump and keep areound the average level of 3 after jump. However, the signals obtained by gradient ascent typically show a jump, but not the plateau afterwards. This is probably caused by the fact that the gradient ascent acts primarily at or around the jump position, but leaves the other values as is (i.e. around zero).

2. In deeper networks with more layers, the generated artificial input is not as pronounced as in the shallower networks. Empirically, sometimes a generated signal in the four-layer MLP is not even recognizable (Fig. 5.2(m)).

3. The visualization can also be performed on intermediate convolutional layer activations. For the output in the layers in the same network, the generated signals illustrate more identical figures and sharper peaks in the layers that are closer to the final output of the network. Also, the signal from visualization with gradient ascent is coherent to the corresponding (hidden layer) activations.

The last two observations are strongly related. In our task, we maximize the activation of neurons in the output layer. The Gradient Ascent approach calculates the derivative of the gradient and the process passes through layer by layer. Thus, as the (intermediate layer) input we aim to generate has more layers to be separated by from the output layer, probably the gradient loss would be more serious as it propagate backward through the layers. This result is related to the recent finding that deep networks are easily fooled (See Section .2.2.1 and Fig. 2.3). In fact, the further argument that "deeper networks are more easily fooled" probably holds as well. This conclusion is also supported by the work of Szegedy et al.[50], in which they suggest that adversarial instances are more significant to higher layers of networks.

Comparatively, the first observation is more related to our research topic to interpret NNs. As empirically the network is able to recognize the signal pattern that is not fed into training, what is the NN actually learning? Firstly, we consider the 1-layer NN. From Section 3.3.3, we conclude from observation of its weights and hidden activations that the one-layer fully connected network learns a smoothed first derivative. In the Gradient Ascent visualization in Fig. 5.1, the most significant jump of an immediate increase also results in a positive first derivative. There exist drops before and after the jump, but the drops are less sharp and take more time steps (i.e. occur more gradually). Thus, the network in fact detects the unique most significant positive first derivative. Secondly, we further explore MLPs with more than one layer. In Fig. 5.2, as long as it remains recognizable, the result of the Gradient Ascent visualizations is similar as the 1-layer NN case, with a sharp increase and some relatively gentle drops. Therefore, although the MLPs are not interpretable from the weights and the hidden layer activations, the Gradient Ascent visualization results suggest that they also learn a first derivative detector, which is the same as the 1-layer NN.

Chapter 6

Learning Structured Data vs. Memorizing Noise Data

Contents

6.1 Introduction

As introduced in Section 2.3.2, Zhang et al. [46] show that deep NNs are able to memorize random labels with brute-force. They suggest that deep NNs to some extent use the strategy of memorization in learning natural or structured data as a conclusion of their empirical results. In this chapter, we also performed experiments along similar ideas for our own task and address this "memorization hypothesis". By exploring the network's hidden layer features, we conclude that the behaviour of the inner-layers of NNs in memorizing random data differs from that in learning natural or structured data.

An analogy drawn from the education of children might be helpful: rote learning vs. intelligent learning. In rote learning, pupils are repeatedly confronted with a list of cases they need to remember. After some training time they have memorized these cases and can reproduce them (but only those). In contrast, intelligent learning assumes that the training examples share some common structure that can be discovered and used to produce the required results.

The only way in which rote learning/memorization can lead to generaliation is through association: if a new example is sufficiently similar to a memorized one, then we can draw the same conclusion. Case-based learning is an example of this.

6.2 Memorizing Randomized Noise labels

6.2.1 Input randomization

Empirically, NNs are able to perform learning to some degree even after randomization of the input data. We attempt the same idea on our jump detection task. The original paper suggests various randomizations such as blurring the labels (shuffling the labels randomly) or blurring the samples (shuffling the pixels, or elements). However, the spatial information is no longer kept in training a deep MLP. As a consequence, it would be meaningless to shuffle the elements in the signal. Thus, we only test the network's ability to learn from random labels. More specificaly, the labels (referring to the jump location) of the training data are shuffled randomly before fed to the network. This effectively removes all the structure that was present in the training data, essentially leaving nothing to be learned.

6.2.2 Network structure

The network structure adopted in the experiment in this section is a 4-layer deep MLP. Among all the structures we have tested previously, deep MLPs are with significantly much more parameters than CNNs (with less than 20 parameters in each convolutional layers) and Autoencoder (or bottleneck) networks (with only a few neurons in the intermediate layer(s)), which is necessary for remembering the whole training set. The network structure and training details are the same as in section 4.3.4. To recall, the structure is illustrated in Fig. 6.1. The activation is ReLU for all hidden layer neurons and softmax for the output layer. The training, test and validation data sets are kept the same. In training, the batch size is 100 and SGD is used to optimize the categorical cross-entropy error. The learning rate is 0.01. Early-stopping is performed when the cross-entropy loss of the validation data starts to increase.

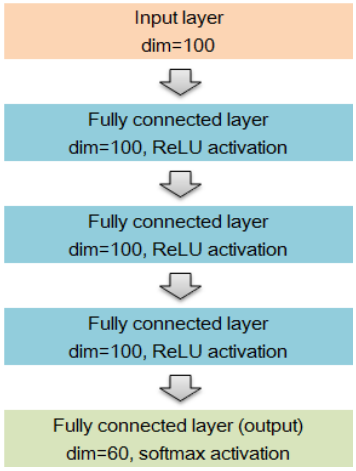
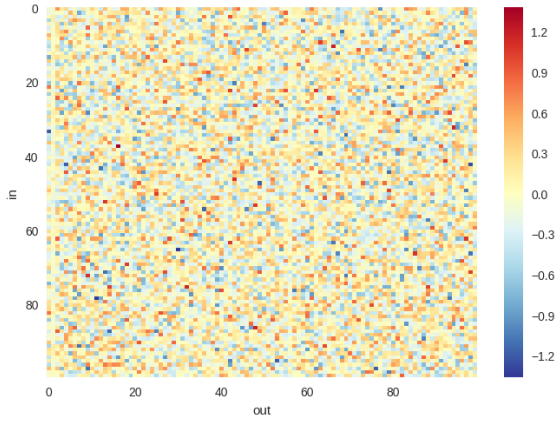


Figure 6.1: Network structure of the "memorizing network": MLP with three hidden layers

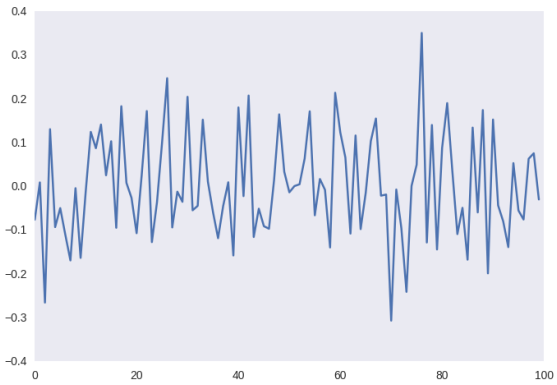
The training process is relatively slow for the randomly labeled data, but it is possible to reach 100% accuracy in training data. As a result, we confirm that with a large network and huge amount of parameters, the network is able to memorize the falsely labeled training data. The resulting model is called a **memorizing network** in order to be distinguishable from the normal *learning* models.

6.2.3 Weights and biases

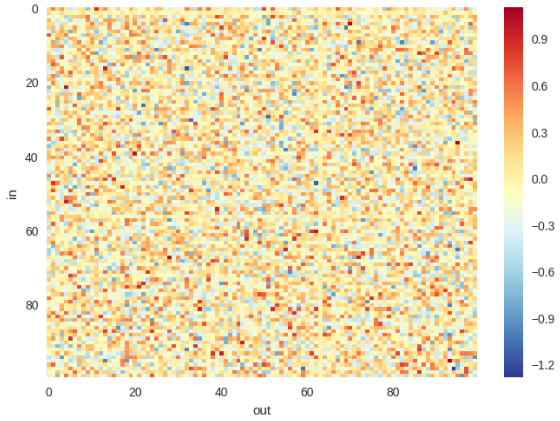
The heat maps of weight matrices are illustrated below. As usual, the figures do not provide much insight into the workings of the NN.



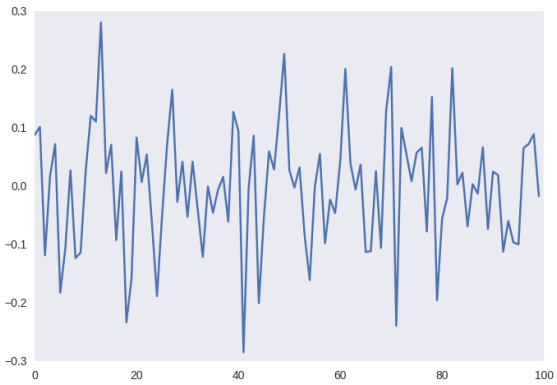
(a) 1st layer weights



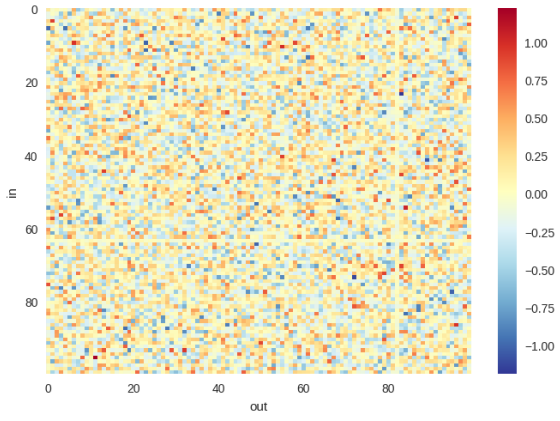
(b) 1st layer biases



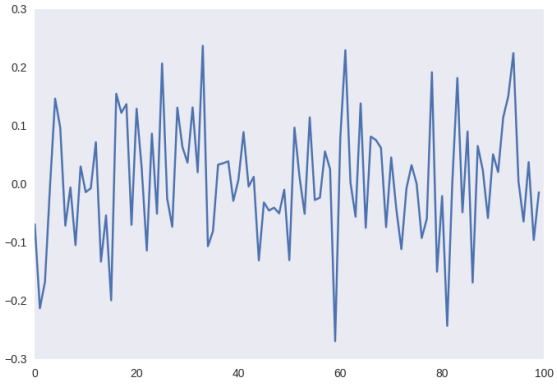
(c) 2nd layer weights



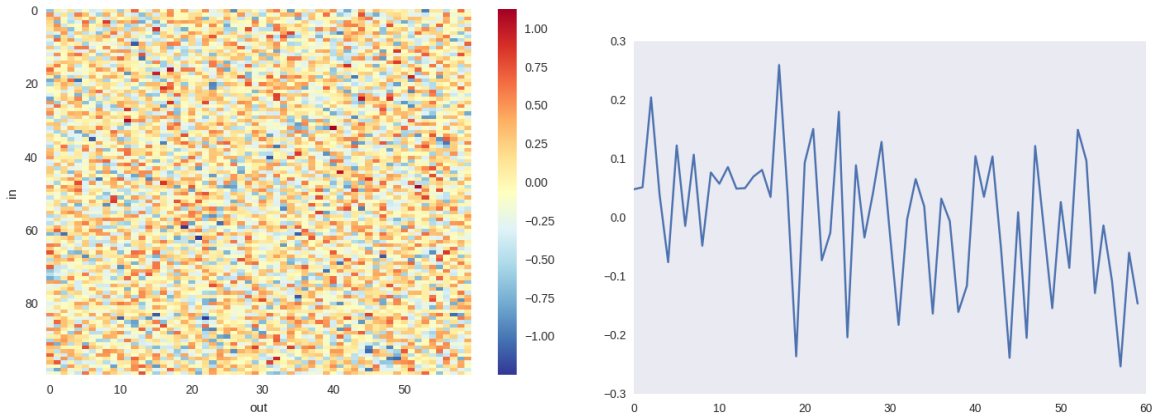
(d) 2nd layer biases



(e) 3rd layer weights



(f) 3rd layer biases



(g) 4th layer weights

(h) 4th layer biases

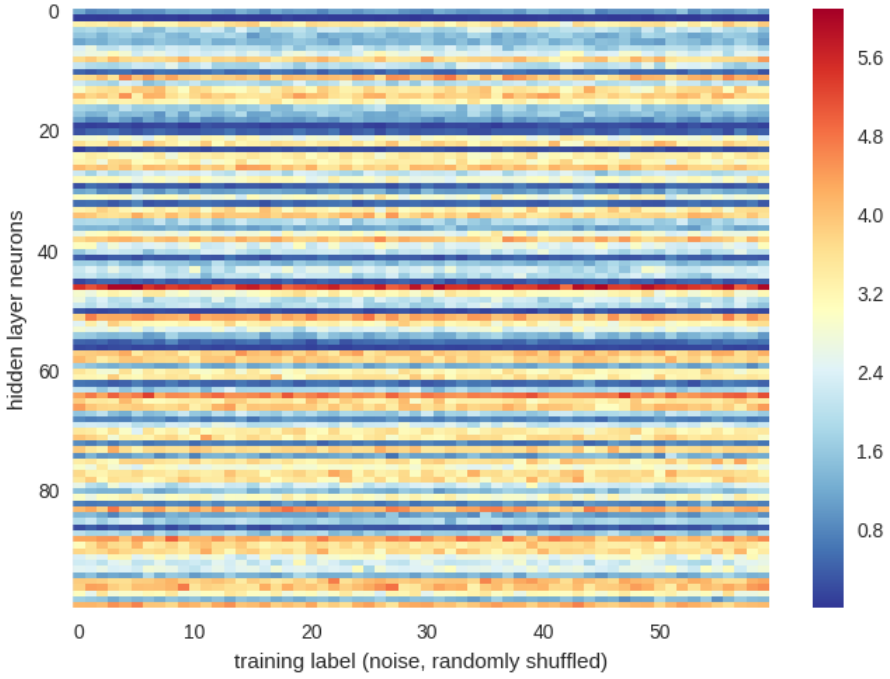
Figure 6.2: Weights and biases of the "memorizing network": MLP trained with random noise labels. Weights are in the left column, and biases are in the right column. Results from different layers are plotted in rows sequentially.

6.2.4 Averaged hidden layer activation

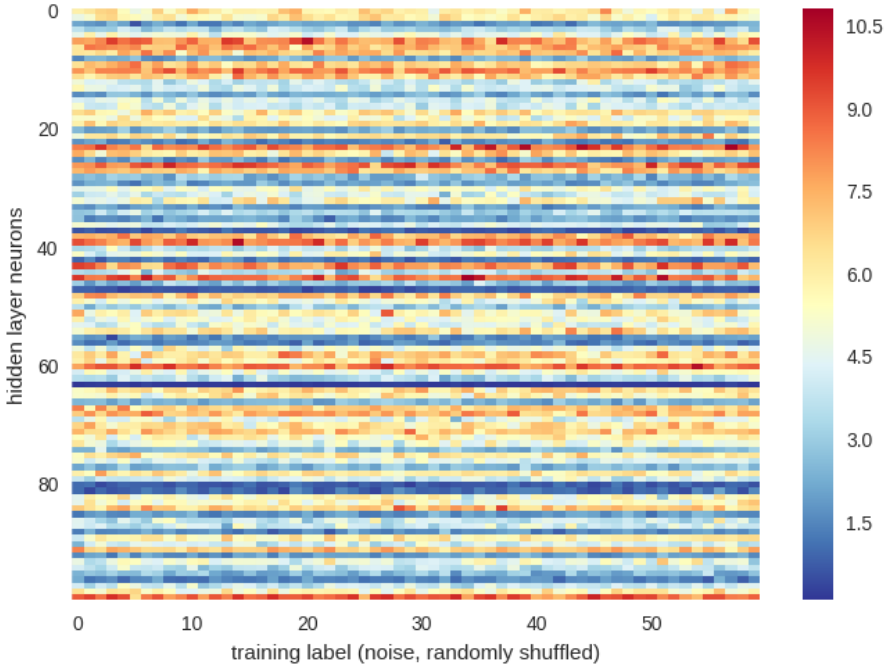
For visualization of averaged hidden layer activation, in this specific task we performed the visulization on either falsely (i.e. randomly) labeled data set (which is indeed the data set for training, of size 12000 or 200 signals for each label) and also the test data set with true labels (of size 60000 or 1000 signals for each label).

On data with false labels

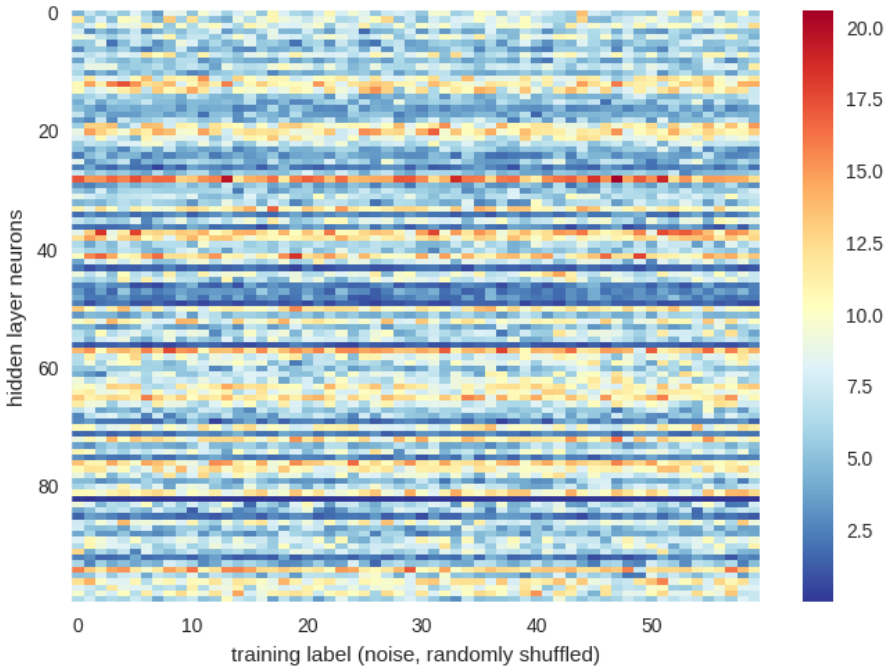
Below in the visualization of hidden layer average activation from falsely labelled data set, there comes no strip-like, patchy figures as before (e.g. see Figs 3.12(a)). Instead, each neuron is activated quite evenly across the range of (randomly) training labels through averaging. But looking to the degree of activation between neurons, it is clear that some neurons are more activated than others, and there are still neurons with almost no activation at all.



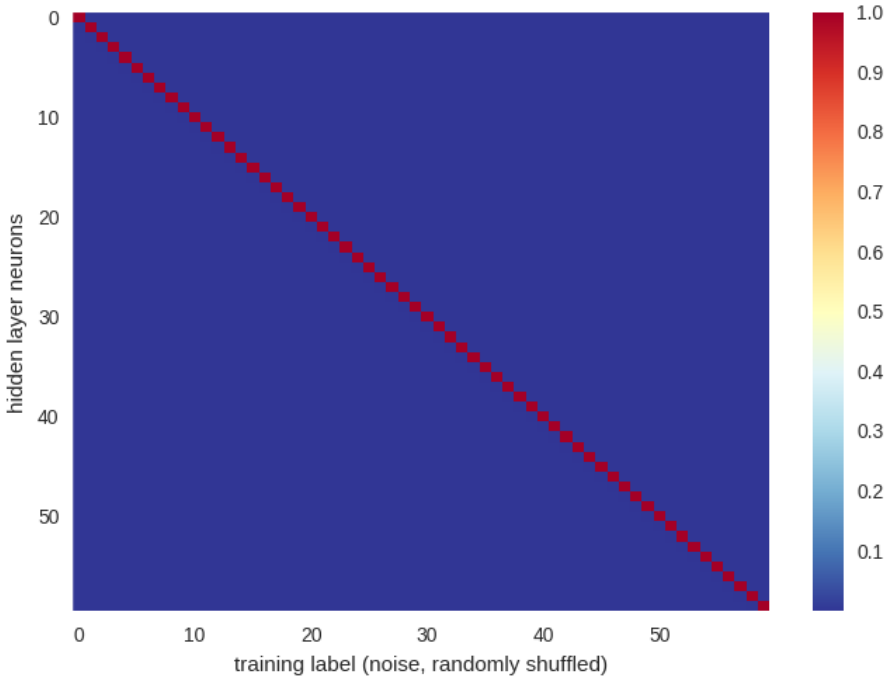
(a) 1st layer averaged activation



(b) 2nd layer averaged activation



(c) 3rd layer averaged activation



(d) 4th layer averaged activation

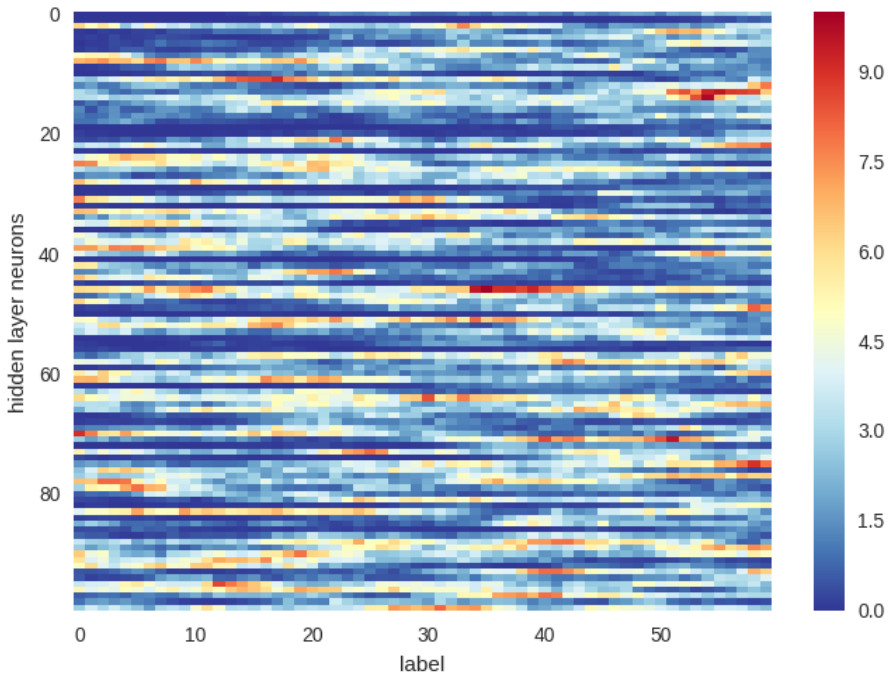
Figure 6.3: "Memorizing network": averaged hidden layer activations from randomized labelled data as input (the data used for training)
In hidden layers, the stripe-like figures disappears compared to normal network. All the class representations are similar.
The result in the output layer Fig. 6.3(d) indicates that the prediction accuracy is 100%, i.e., the network memorizes everything noise label from the training data.

On data with correct labels

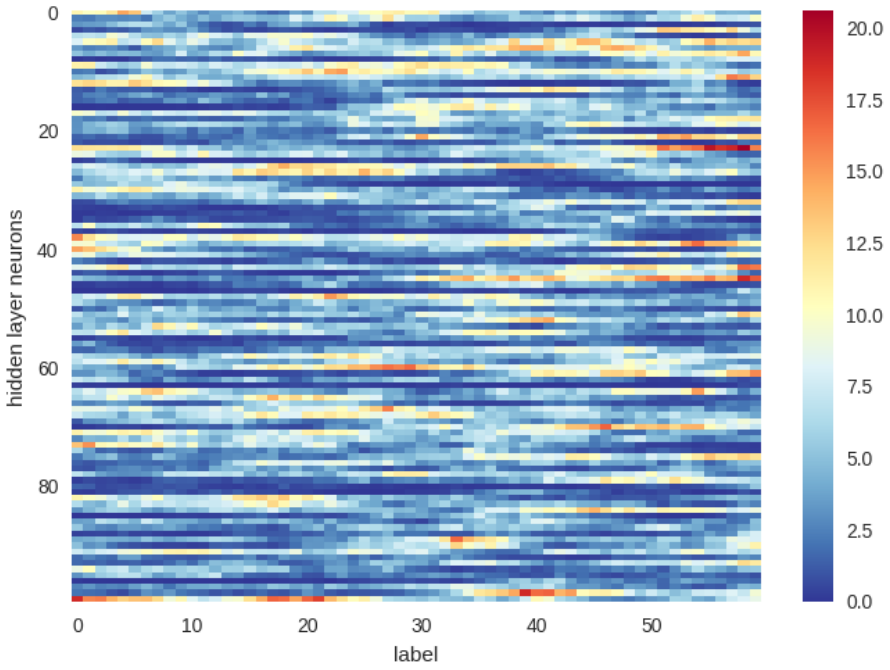
However, in the visualization of averaged hidden layer activation from correctly labeled data set, in Fig. 6.4(a),6.4(b) and 6.4(c), it shows similar stripe-like figures as in the normally trained MLP networks. It is reasonable that the plotting illustrates similar encoding patterns, because the inner- and inter-relationship of the signals in their actual classes still exist.

From Fig. 6.4(d) we know that the model fails in predicting truly labeled data, which means that such a model does not have learning ability. However, since we still find the stripe-like figures as before, we suspect that the stripes are not ascribed to learning such a problem, but they are an artifact produced by the data in which the signals of the same or close jump positions shows similarity.

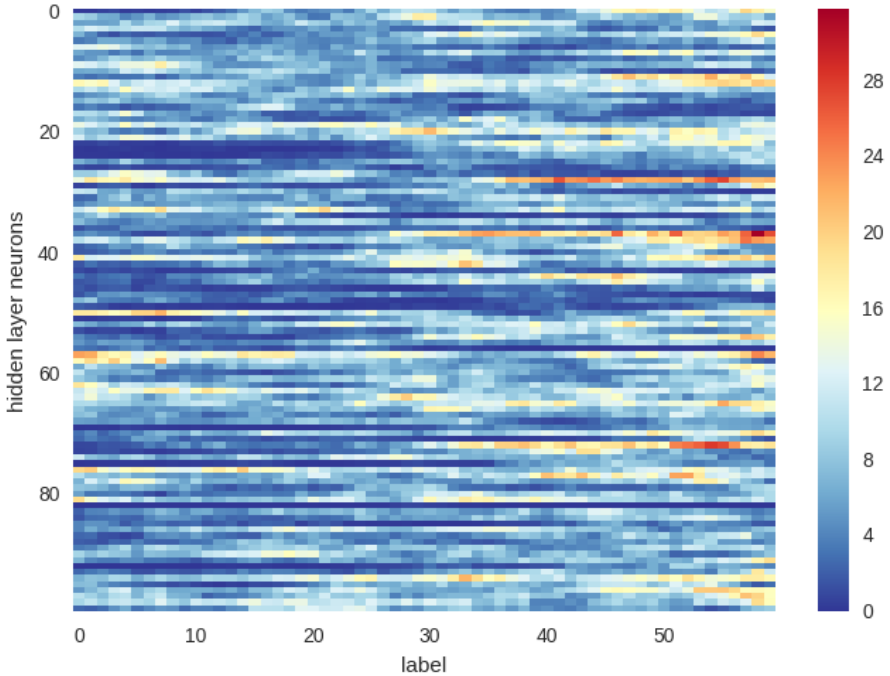
From the figure of hidden features, it is hard to distinguish the difference of the class representation here in the "memorizing network" and in the normal "learning network" with the similar MLP structures (see Fig. 3.11(a)). However, the difference, which is mainly about the distance between the class representations through layers, can be deduced from a measurement-based analysis which is the topic of Section 6.4.3.



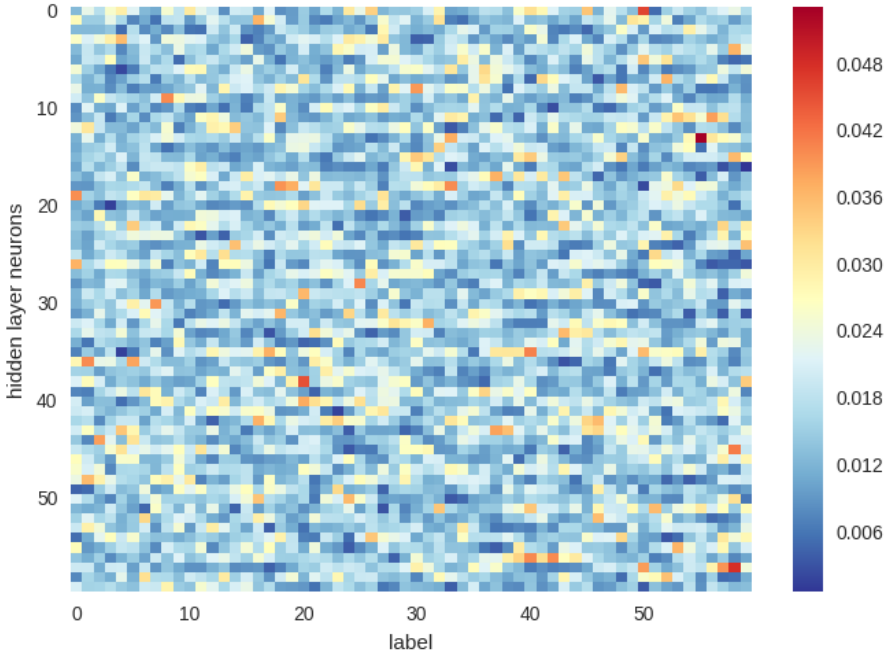
(a) 1st layer averaged activation



(b) 2nd layer averaged activation



(c) 3rd layer averaged activation



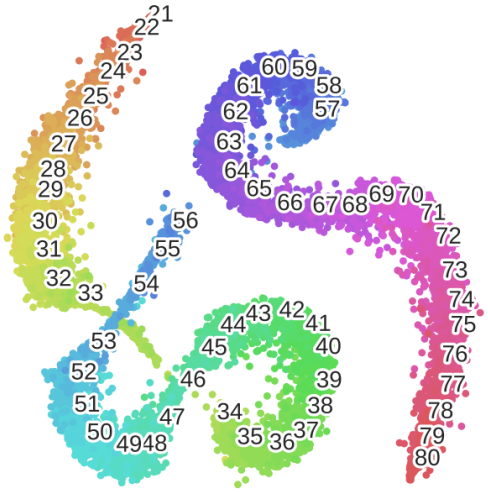
(d) 4th layer (output) averaged activation

Figure 6.4: "Memorizing network": averaged hidden layer activations. **Same network as in Fig. 6.3, but activations are from data with true labels (i.e. structured data).**
Stripe-like figures disappears compared to normal network. All the class representations are similar.

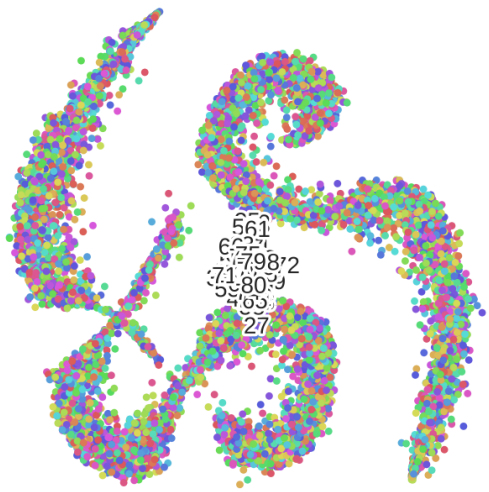
6.3 Visualization of Hidden Features with t-SNE

It is also interesting to use t-SNE visualization to inspect the hidden layer features of the "memorizing network", as illustrated in Fig. 6.5. Unlike the previous visualizations where activations belonging to the same class are averaged, each data point in the figures represent the activation from a single data instance.

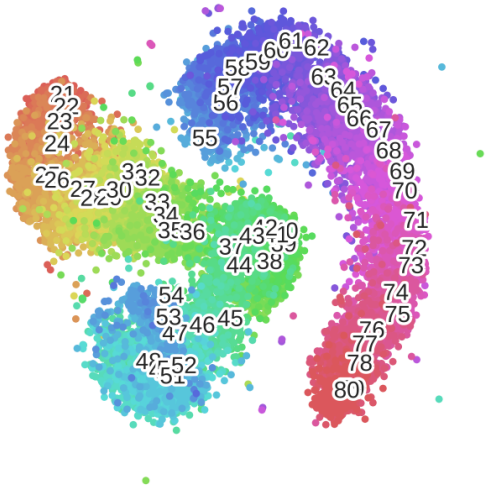
Left and right columns in this figure represent the same data cloud (i.e. the t-SNE representation of resulting activations for each layer) but left is labelled with the actual labels, and right is labelled with the labels after randomization, which are exactly the labels that the "memorizing network" is trained to learn.



(a) Input with true labelling

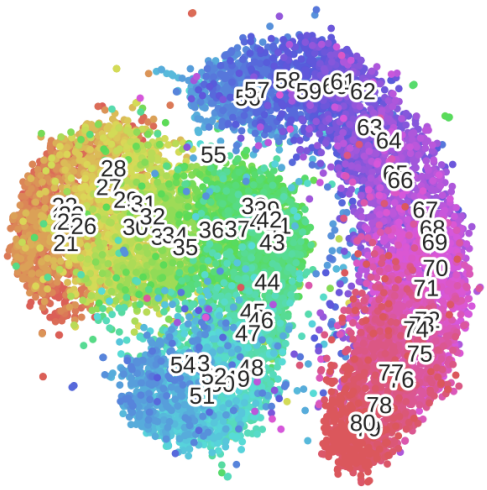


(b) Input with randomized labelling

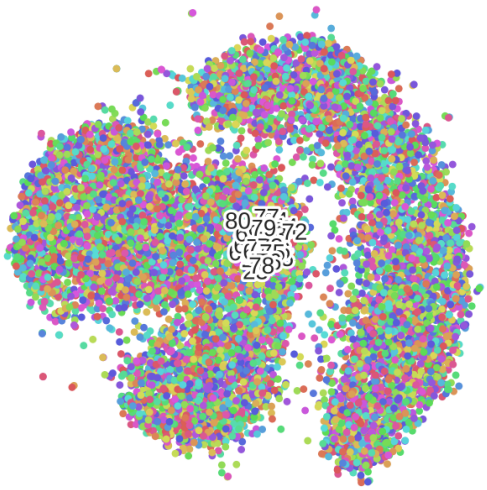


(c) 1st layer with true labelling

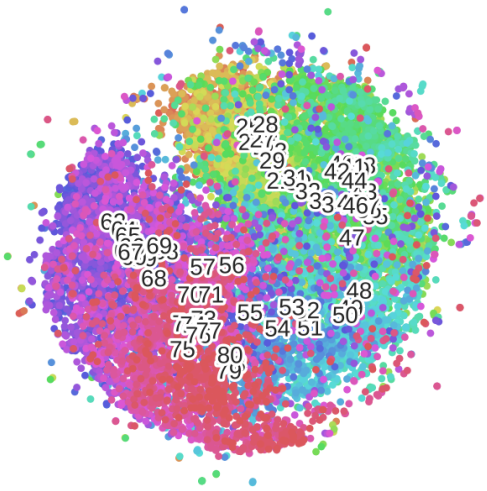
(d) 1st layer with randomized labelling



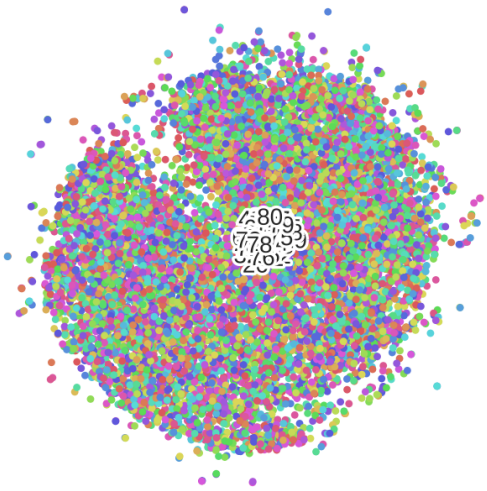
(e) 2nd layer with true labelling



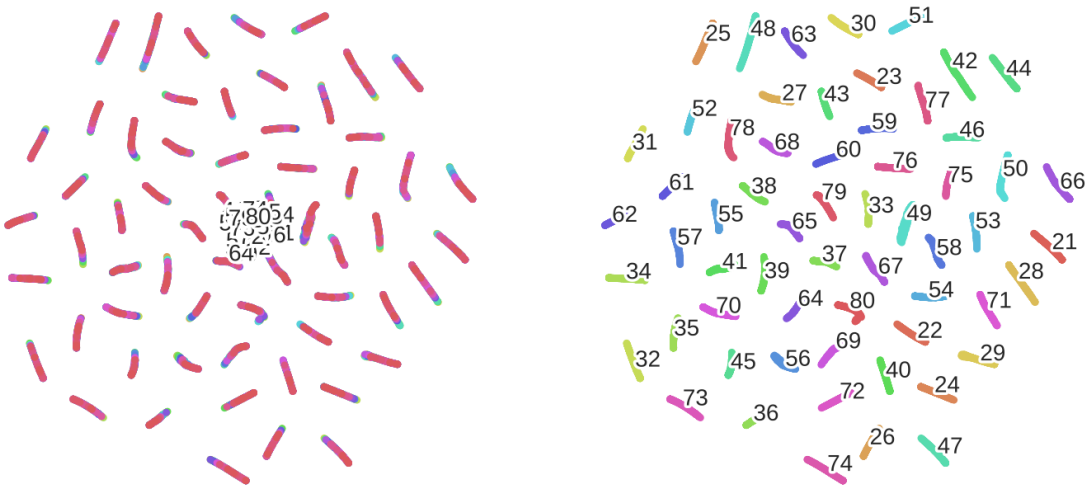
(f) 2nd layer with randomized labelling



(g) 3rd layer with true labelling



(h) 3rd layer with randomized labelling



(i) 4th layer with true labelling

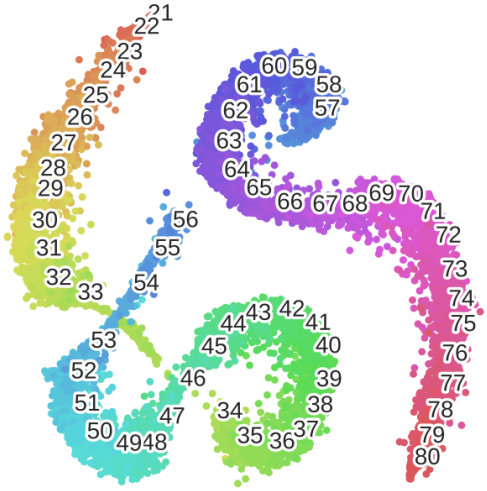
(j) 4th layer with randomized labelling

Figure 6.5: t-SNE visualization of hidden activations in the "memorizing network".
Each sub-figure consists of 12000 points represent 12000 instances in the training data.
Unlike the previous visualizations where the activations belong to the same class are averaged, each data point in the figures represent the activation from a single data instance.
Left and right are from the same data cloud, but left is labelled with the actual labels, and right is labelled with the labels after randomization (the labels for training).

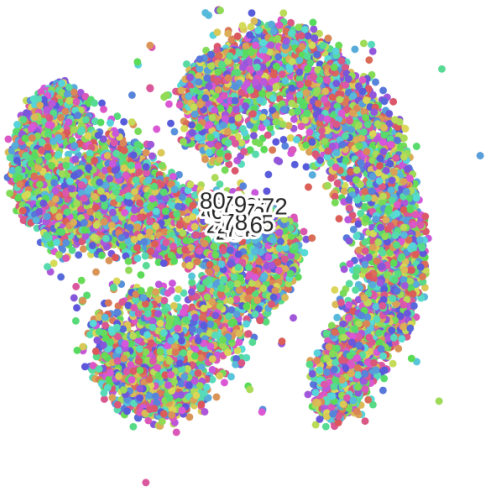
The first row in the figure represents the original signals in the training set. As expected, the input data are grouped along a low-dimensional manifold in the 100-dimensional input space. Data points with same and similar (true!) jump location are grouped together. As the layer goes deeper, the hidden features, which are also 100-dimensional, are grouped more and more loosely. The distance between the data points in the same class becomes greater. Especially in the third layer, which is also the last hidden layer, the data points are distributed quite evenly and no longer along a traceable thin manifold in the 100-dimensional space. Through the behaviour in the hidden layers, the network manages to confuse the original data from its actual class. In the final layer, each data point is designated to its new labelling which is false and randomized.

We are also interested in comparing the "memorizing network" and the "learning network" from the view of t-SNE visualization, as illustrated in Fig. 6.6.

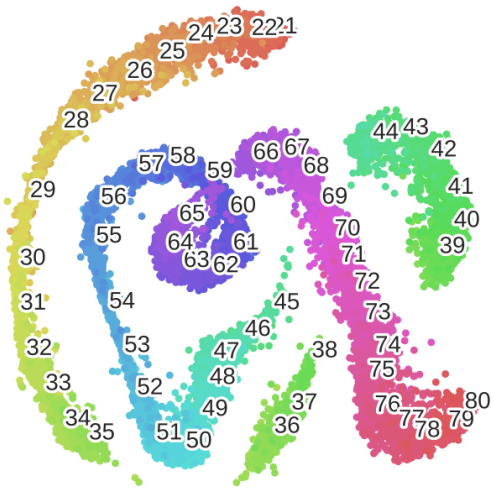
(a) "Memorizing network": Input



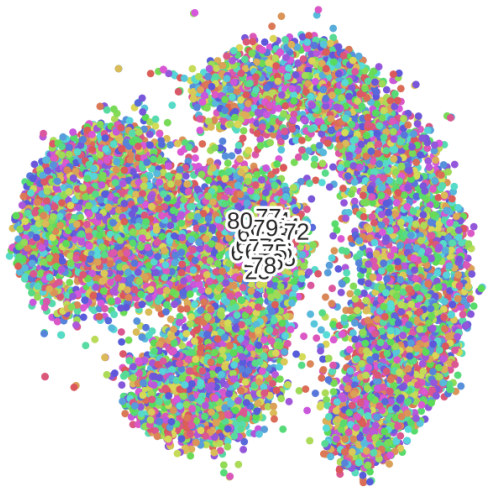
(b) "Learning network": Input



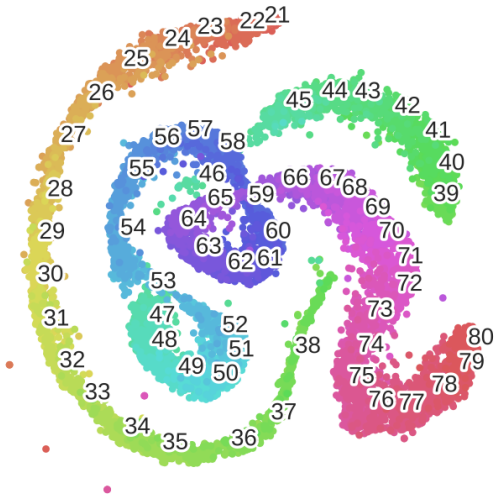
(c) "Memorizing network": 1st layer



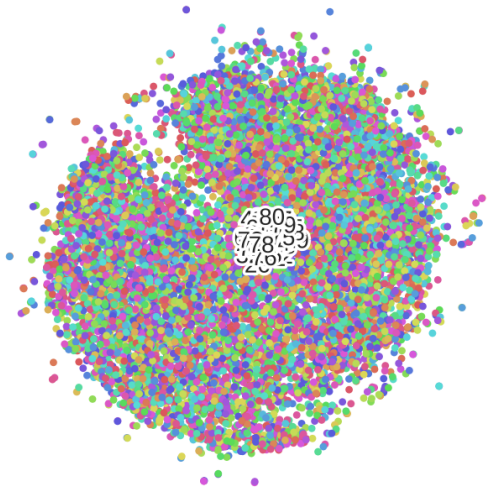
(d) "Learning network": 1st layer



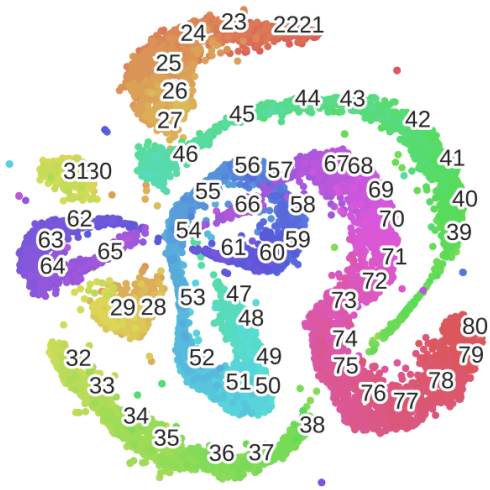
(e) "Memorizing network": 2nd layer



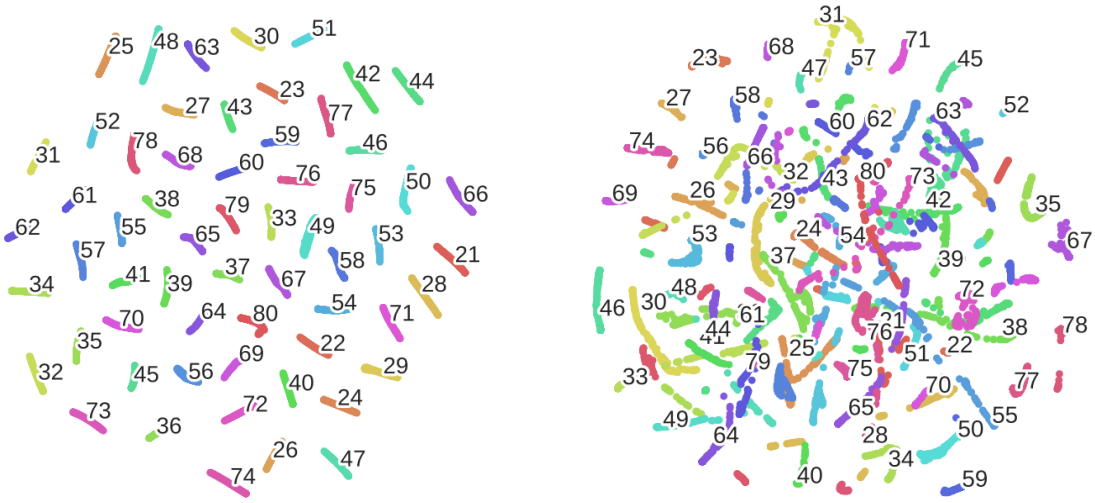
(f) "Learning network": 2nd layer



(g) "Memorizing network": 3rd layer



(h) "Learning network": 3rd layer



(i) "Memorizing network": 4th layer

(j) "Learning network": 4th layer

Figure 6.6: t-SNE visualization of hidden activations in the "memorizing network" (left) and the normally trained 4-layer MLP "learning network" (right).
Each sub-figure consists of 12000 points represent 12000 instances in the training data.
Both are labelled with the target labels for training (randomized labels for "memorizing network" and actual labels for the "learning network").

Obviously, the "memorizing network" (left) and the "learning network" (right) are significantly different in terms of the behaviour in the hidden layers. From the visualization of the "learning network", the distribution of the data representations in the hidden layers is not really different from the original input. It is hard to tell whether data points in the same class are grouped more tightly or more loosely as the layer goes deeper. At least, when performing learning natural or structured data normally, the network would not attempt to confuse the actual class of the network by expanding the distribution of the data points evenly in the layer activation space.

Also, notice that in the final output layer, the output of the "learning network" is not as solid as the "memorizing network". One difference is that the "memorizing network" aimed for an extreme over-fitting of 100% accuracy in the randomized training labels; while the normal learning model does not optimize that far for the training data in order to prevent over-fitting. The second difference is that the normal learning model does not necessarily make prediction of only one possible result. Consider a signal with random noise that can be recognised as a second jump. In this case, the one-hot vectorized prediction would probably give two possible choices of jump locations, with the value as the possibility. Such an output is known as a "soft target"[51] which is informative and believed to be a better indicator than the original only-one "hard" result. The "soft target" produced by the network which performs normal learning also contributes to the generalization ability of the network.

The paradox of memorizing and learning is to be further explored. From the t-SNE visualization of the the "memorizing network" and the "learning network", we conclude that the network behaves differently when learning correctly labeled, structured data as opposed to memorizing noise data. In the next section, a measurement-based analysis is introduced to provide some more insight in how the hidden layer features are distributed in both inner-class and inter-class cases, in order to better

interpret the difference of the network behaviour in memorizing noise data and actual learning with generalization ability.

6.4 Measuring Deviation of Hidden Features in MLPs

Inspired by the t-SNE visualization of the hidden features in the randomly shuffled model, we are interested in how the behaviour of the network changes in hidden layers. More specifically, we focus on how the representations of the data instances in each layer deviate from each other, either with the same jump position or with different jump positions. For such a purpose, we introduce a measurement-based analysis in this section.

6.4.1 Methodology and testing on jump detection task

First, we search for solution on the normally trained 4-layer MLP for the jump detection task. (See Section 3.3.3.) In order to further explore this question, it is necessary to first introduce some measures of deviation.

The first measure of deviation is the Euclidean distance of two n-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$ is

$$d_E(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2, \tag{6.1}$$

which is the usual L_2 norm of the difference of the two vectors. Using this measure of deviation, we are able to define the inner-class deviation and the inter-class deviation of the hidden features for each layer of the network. The purpose is to find if there is any interesting behaviour in the hidden layers of the network. For more detail on notations used in the text below we refer the reader to Section 3.3.3.

For the inner-class deviation, we consider all data instances in the same class. The hidden activation (as a vector) in the layer of each data instance is adopted as the representation of this data point. The (mean) inner-class deviation of a layer l is the average of the distance of the vector representation (activation) of every two (say the n_1^{th} and the n_2^{th}) data instances $\mathbf{a}^{(l)}_{(c),n_1}$ and $\mathbf{a}^{(l)}_{(c),n_2}$ from all $N_{(c)}$ data instances in the same class (jump position) c in the l^{th} layer. For convenience of use, we take the average over all 60 classes (c) from the jump position (21) to (80).

$$\begin{aligned} InnerDev^{(l)}_{Euc} &= \overline{d_E(\mathbf{a}^{(l)}_{c,n_1}, \mathbf{a}^{(l)}_{c,n_2})} \\ &= \frac{1}{N_{class}} \sum_{(c)=(21)}^{(80)} \frac{2}{N_{(c)}(N_{(c)}-1)} \sum_{\substack{n_1,\, n_2 \,\in\, N_{(c)} \\ n_1 \neq n_2}} d_E(\mathbf{a}^{(l)}_{c,n_1}, \mathbf{a}^{(l)}_{c,n_2}), \end{aligned} \tag{6.2}$$

where $N_{class} = 60$ is the total number of classes (i.e. one for each jump position).

For the inter-class deviation, we only consider the distance between the two adjacent classes (c) and $(c+1)$. We use the averaged activation $\mathbf{A}^{(l)}_{(c)}$ of layer l to stand for the class representation. (See Section 3.3.3.) The (mean) inner-class deviation of a layer (l) is the average of the distance of the class representation of any two adjacent classes (c) and $(c+1)$.

$$\begin{aligned} InterDev^{(l)}_{Euc} &= \overline{d_E(\mathbf{A}^{(l)}_{(c)}, \mathbf{A}^{(l)}_{(c+1)})} \\ &= \frac{1}{N_{class}} \sum_{(c)=(21)}^{(79)} d_E(\mathbf{A}^{(l)}_{(c)}, \mathbf{A}^{(l)}_{(c+1)}). \end{aligned} \tag{6.3}$$

The inner-class deviation is the measure of the inner-class deviation of all data points in the class, while the inter-class deviation is the measure of the difference from class to class.

We perform the layer-by-layer analysis on the inner- or inter-class deviation with Euclidean distance. The method is first tested on the normally trained 4-layer network with three hidden layers. (See Section 3.3.3.) The results are illustrated in Fig. 6.7.

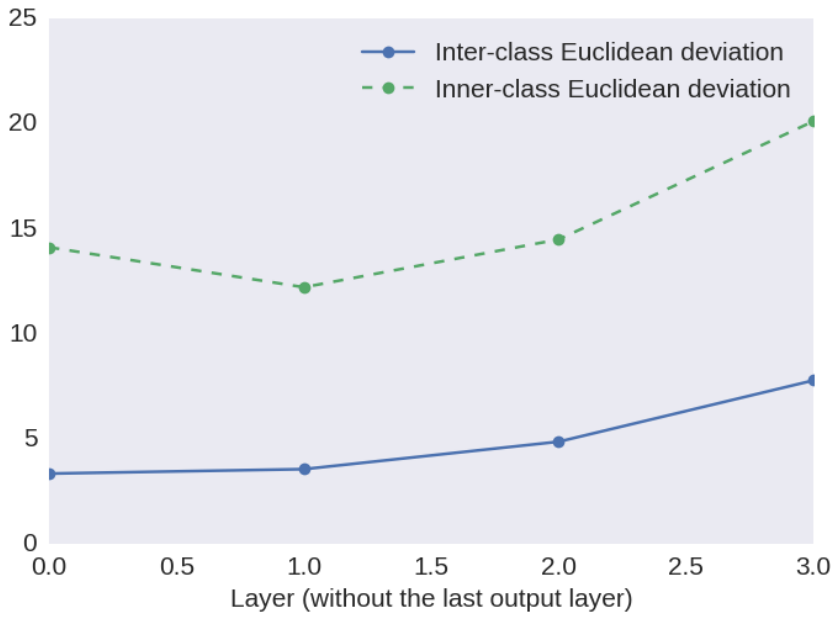


Figure 6.7: The inner- or inter-class deviation with Euclidean distance on a 4-layer MLP with three hidden layers.
The inner-class deviation is marked with dotted line.
The 0th layer (original input) is also considered as the original representation.
The final output is not included in this graph since the representation is one-hot vectors which is much different from the previous layers.

From Fig. 6.7, the deviation generally increases, except that there is a slight drop of the inner-class deviation from the input layer to the first hidden layer. This does not necessarily mean that both inner- and inter-class deviation increases. We notice that the overall activation level increases in deeper layers (Fig. 6.8(a)). The overall activation level is defined by the averaged summation of activation level in each layer

$$\overline{\| \mathbf{a}^{(l)}_{(c),n} \|_1} = (\sum_{(c)} \sum_{n} \| \mathbf{a}^{(l)}_{(c),n} \|_1)/(N_{(c)}N), \tag{6.4}$$

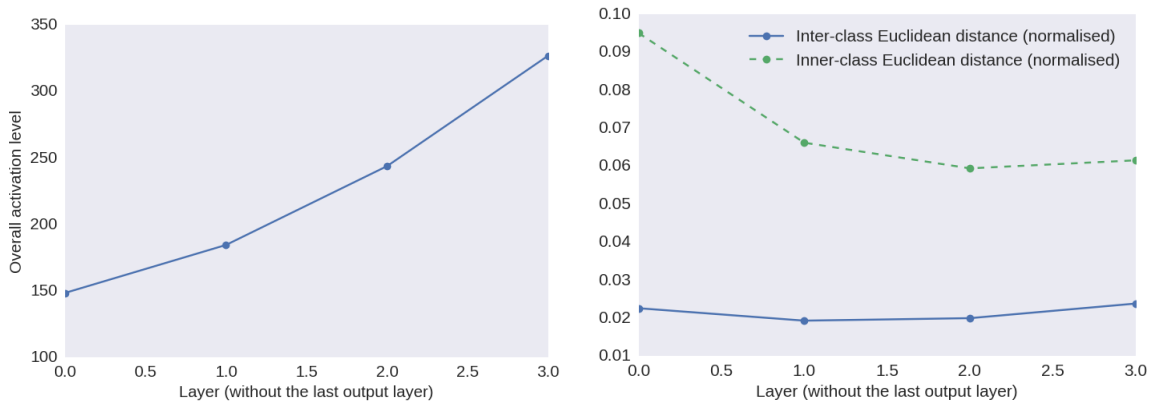
where $\| \cdot \|_1$ denotes the L_1 norm. As the Euclidean distance more concerns about the absolute value, the resulting deviation measure is interfered by the increasing overall activation level.

Compensating for global changes in activation levels In order to eliminate the influence of the increasing activation, the measurement is normalised by dividing by the average activation level of the layer:

$$\begin{aligned} InnerDev^{(l)}_{normEuc} &= \frac{InnerDev^{(l)}_{Euc}}{\overline{\| \mathbf{a}^{(l)}_{(c),n} \|_1}} \\ &= \frac{InnerDev^{(l)}_{Euc}}{\frac{1}{N_{class}} \sum_{(c)=(21)}^{(79)} \frac{1}{N_{(c)}} \sum_{n=1}^{N_{(c)}} \| \mathbf{a}^{(l)}_{(c),n} \|_1} \end{aligned} \tag{6.5}$$

$$
\begin{aligned}
InterDev^{(l)}_{normEuc} &= \frac{InterDev^{(l)}_{Euc}}{\overline{\| \mathbf{a}^{(l)}_{(c),n} \|_1}} \\
&= \frac{InterDev^{(l)}_{Euc}}{\frac{1}{N_{class}} \sum_{(c)=(21)}^{(79)} \frac{1}{N_{(c)}} \sum_{n=1}^{N_{(c)}} \| \mathbf{a}^{(l)}_{(c),n} \|_1}
\end{aligned}
\tag{6.6}
$$

(See Fig. 6.8(b)). From the normalised Euclidean deviation, there is a significant decreasing tendency in the inner deviation, which indicates that the network forces the representation of the data points in the same class closer to each other. While, there is no significant sign on inter-class deviation.



(a) Overall activation level in each layer

(b) Normalised Euclidean deviation

Figure 6.8: Left: the overall activation level in each layer. Calculated by averaged summation of activation level in each layers $\overline{\| \mathbf{a}^{(l)}_{(c),n} \|_1}$ *, where* $\| \cdot \|_1$ *denotes the* L_1 *norm.*
Right: Euclidean deviations divided by the activation level, as the normalised Euclidean deviation which is less interfered by the overall activation level.

Alternative distances In addition to the normalised Euclidean deviation, we further define two kinds deviation measures that are non-sensitive to the overall activation level. These are based on the similarity measure of the cosine similarity and correlation.

The cosine distance of two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined by

$$
d_{cos}(\mathbf{x}, \mathbf{y}) = 1 - \frac{\mathbf{x} \cdot \mathbf{y}}{\| \mathbf{x} \| \| \mathbf{y} \|}. \tag{6.7}
$$

The second term of the right part of the equation $(\mathbf{x} \cdot \mathbf{y}) / \| \mathbf{x} \| \| \mathbf{y} \|$ is also known as the cosine similarity of the vectors $\mathbf{x}$ and $\mathbf{y}$. The range of the cosine distance $[0, 2]$.

The correlation distance of two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$
d_{corr}(\mathbf{x}, \mathbf{y}) = 1 - \rho_{\mathbf{x},\mathbf{y}}, \tag{6.8}
$$

where $\rho_{\mathbf{x},\mathbf{x}}$ is the Pearson's correlation coefficient of $\mathbf{x}$ and $\mathbf{y}$

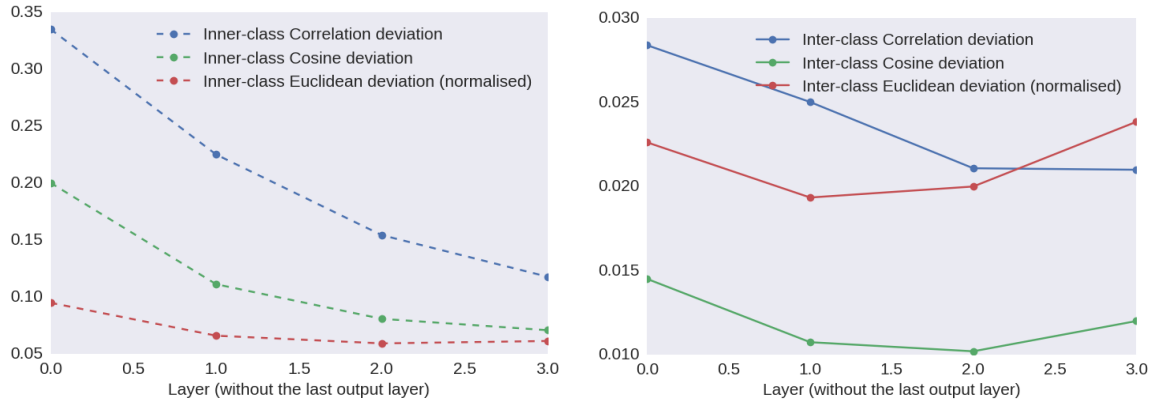
$$
\begin{aligned}
\rho_{\mathbf{x},\mathbf{y}} &= \frac{cov(\mathbf{x}, \mathbf{y})}{\sigma_{\mathbf{x}} \sigma_{\mathbf{y}}} \\
&= \frac{\mathbb{E}((\mathbf{x} - \mathbb{E}(\mathbf{x}))(\mathbf{y} - \mathbb{E}(\mathbf{y})))}{\sqrt{\mathbb{E}(\mathbf{x}^2) - (\mathbb{E}(\mathbf{x}))^2} \sqrt{\mathbb{E}(\mathbf{y}^2) - (\mathbb{E}(\mathbf{y}))^2}}.
\end{aligned}
\tag{6.9}
$$

$\mathbb{E}(\cdot)$ is the expectation, $cov(\cdot)$ denotes the covariance and σ is the standard deviation of the vector. Actually, correlation distance can be regarded as a normalised version of cosine distance: $d_{Corr}(\mathbf{x}, \mathbf{y}) =$

$d_{cos}(\mathbf{x}-\overline{\mathbf{x}}, \mathbf{y}-\overline{\mathbf{y}})$. The range of the correlation distance is also $[0, 2]$. For all the distances introduced in this section, higher values indicate that the two vectors are farther apart (more dissimilar).

As is to be expected, the average inner- and inter-class deviation of cosine and correlation distance are defined similarly as in Eq. 6.2 and Eq. 6.3.

The cosine deviation, correlation deviation and the normalised Euclidean deviation are considered together, as they are all non-sensitive to the overall activation level of the layer. As is apparent in Fig. 6.9, unlike the Euclidean distance case which is absolute value sensitive, the inner-class deviation drops significantly with layers. This stands in opposition to the inter-class deviation which shows an overall drop but with no significant tendency. The result is generally reminiscent of the normalised Euclidean deviation case (Fig. 6.8(b)). There is a significant decreasing tendency in the inner deviation, which indicates that the network forces the representation of the data points in the same class closer to each other. While, there is no significant sign on inter-class deviation. We expect that the network would force inter-class representations depart from each other, but it is not supported by the measure normalised with the overall layer activation.



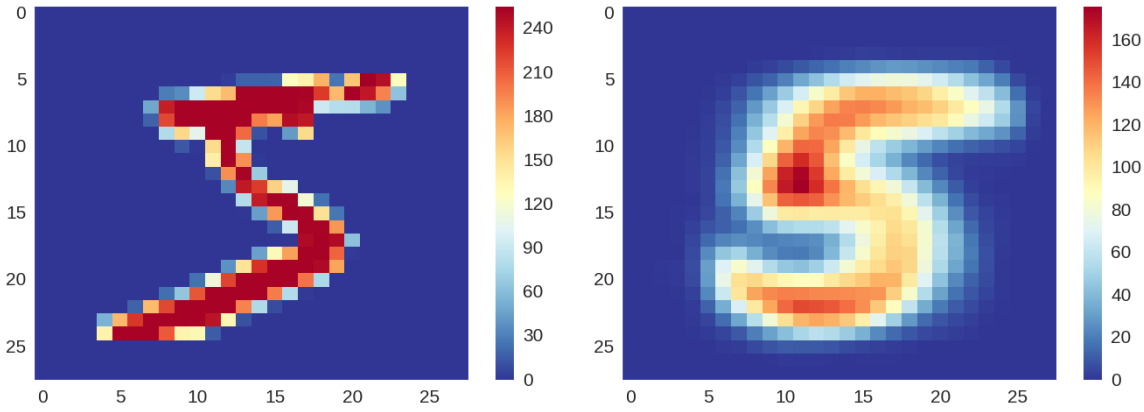
(a) Inner-class deviation measures that are not sensitive to the activation level

(b) Inter-class deviation measures that are not sensitive to the activation level

Figure 6.9: Inner- and inter-class deviation measures that are not sensitive to the activation level
Left is the inner-class measures and right is the inter-class measures. Illustrated separately because the difference of scale is larger than the Euclidean distance case.
The inner-class deviation drops significantly with layers. While the inter-class deviation shows an overall drop but with no significant tendency.

6.4.2 Test on MNIST data

MNIST data consists of 28 by 28 grey-scale images of handwritten digits. Each digit is represented by a vector of length 784. There are 60000 training samples and 10000 test samples. Fig. 6.10(a) visualises a single instances of digit "5". Fig. 6.10(b) averages 1000 images in the class "5", which is still human recognizable as a digit "5". The averaged figure can also be regarded as the class representation of class "5" $\mathbf{A}^{(0)}_{(5)}$ in the zeroth and the input layer. Thus, MNIST data set is applicable with our methodology. However, not all of the tasks are able the obtain class representation by averaging activations. Consider a task to recognize a cat. In this case, averaging 1000 cat images is usually meaningless.



(a) A single data instance in class "5"

(b) Averaging 1000 data instances in class "5"

Figure 6.10: Visualising MNIST image of class "5". Left is a single instance. Right is 1000 instances averaged. The averaged image of class "5" is still recognizable, which indicates that the averaged class representation is reasonable.

A 5-layer MLP is trained to perform MNIST handwritten digits recognition. The methodology is similar to our previous experiments on jump detection task. The network structure is 784-784-784-784-784-10. (See Fig. 6.11) The size of the hidden layers are set as the same as the input, which is convenient for analyzing the hidden features. The output is the one-hot vectors to indicate which of the ten digits the data instance belongs to. The activation is ReLU for all hidden layer neurons and softmax for the output layer. We separate the original training data of 60000 samples, taking 48000 for training and 12000 for validation. In training, the batch size is 1000 and SGD is used to optimize the categorical cross-entropy error.The learning rate is 0.01. Early-stopping is performed when the cross-entropy loss of the validation data starts to increase.

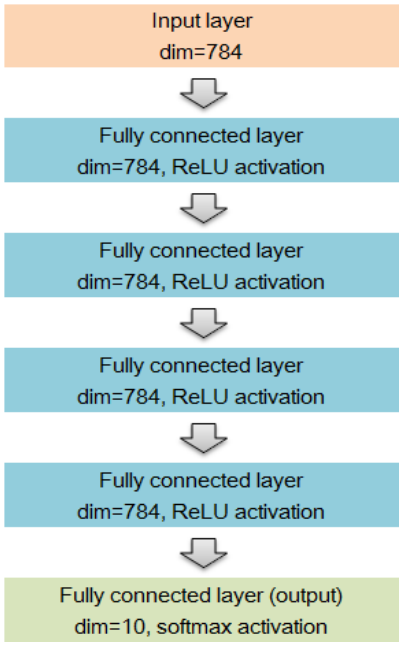
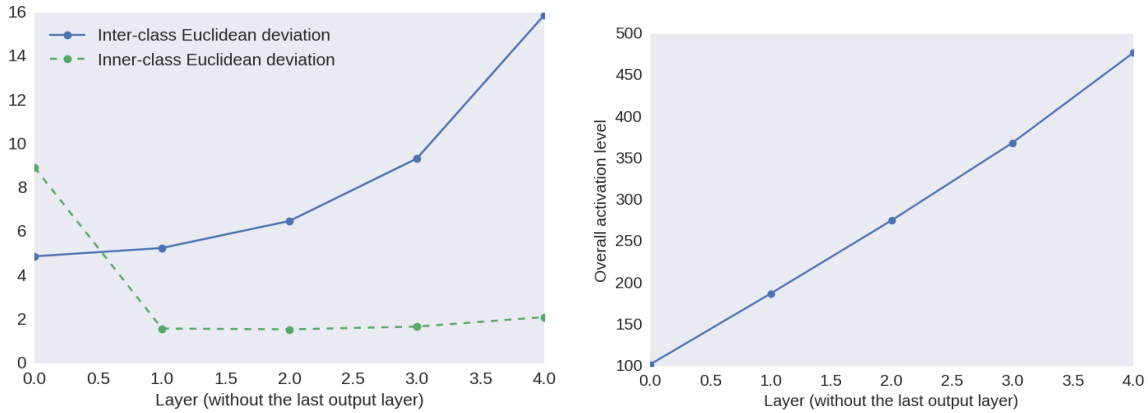


Figure 6.11: The network structure: 5-layer MLP for MNIST classification

The accuracy on the test data is 98.07%, which is acceptable. Our purpose is not seeking for the

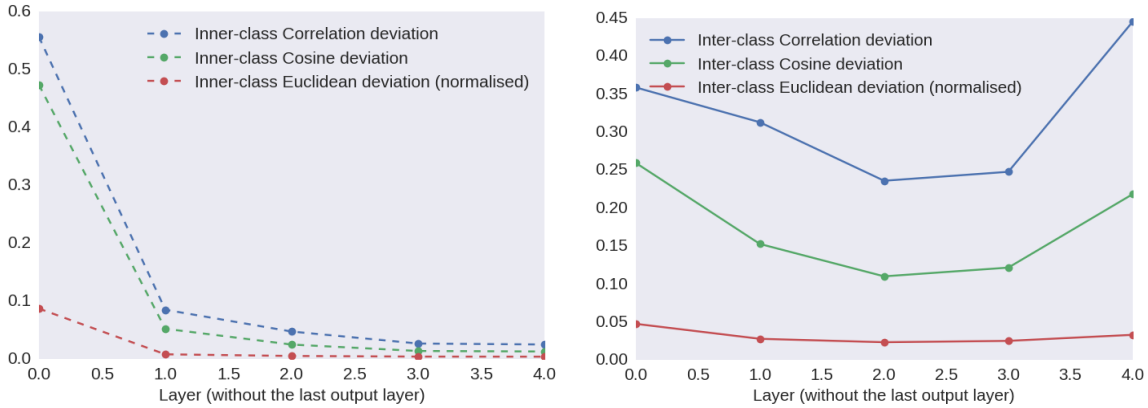
best accuracy, so there is still large space for improvement by fine-tuning the parameters.

Notice that in MNIST task, the definition of the inter-class deviation is a bit different from the jump detection task. Indeed, in the jump detection task, there is some natural concept of continuity as the class of with jump-location (40) and the class of (41) are adjacent to each other. While in handwritten digits, the class '3' and '4' are not really closer to each other than to other categories of digits. Thus, we change the definition of the inter-class deviation as the average of the distance of all the pairs of two class representations (45 pairs in all). The measures on the resulting model are illustrated in the graphs in Fig. 6.12.



(a) Euclidean deviation

(b) Overall activation level in each layer



(c) Inner-class deviation measures that are not sensitive to the activation level

(d) Inter-class deviation measures that are not sensitive to the activation level

Figure 6.12: Measurement-based analysis on a 5-layer MLP for MNIST classification.

From this exploration, the basic tendency of the measures in the hidden layers is similar as in the jump detection task. It is also similar that as illustrated in Fig. 6.12(b), from the original input to all hidden layers, the overall activation level increases with layers.

However, there also exist some differences that are worth noting:

- Firstly, the inner-class deviation is very high at the original input representation. As a result, there is a sharp drop from the input to the first hidden layer feature, even in the non-normalised Euclidean deviation. The network is so powerful that it is able to make the representations significantly closer to each other within only one layer.
- Secondly, in Fig. 6.12(d), the inter-class deviation shows a tendency that it first drops gradually and then increases until higher deviation than the original input. This tendency is much clearer

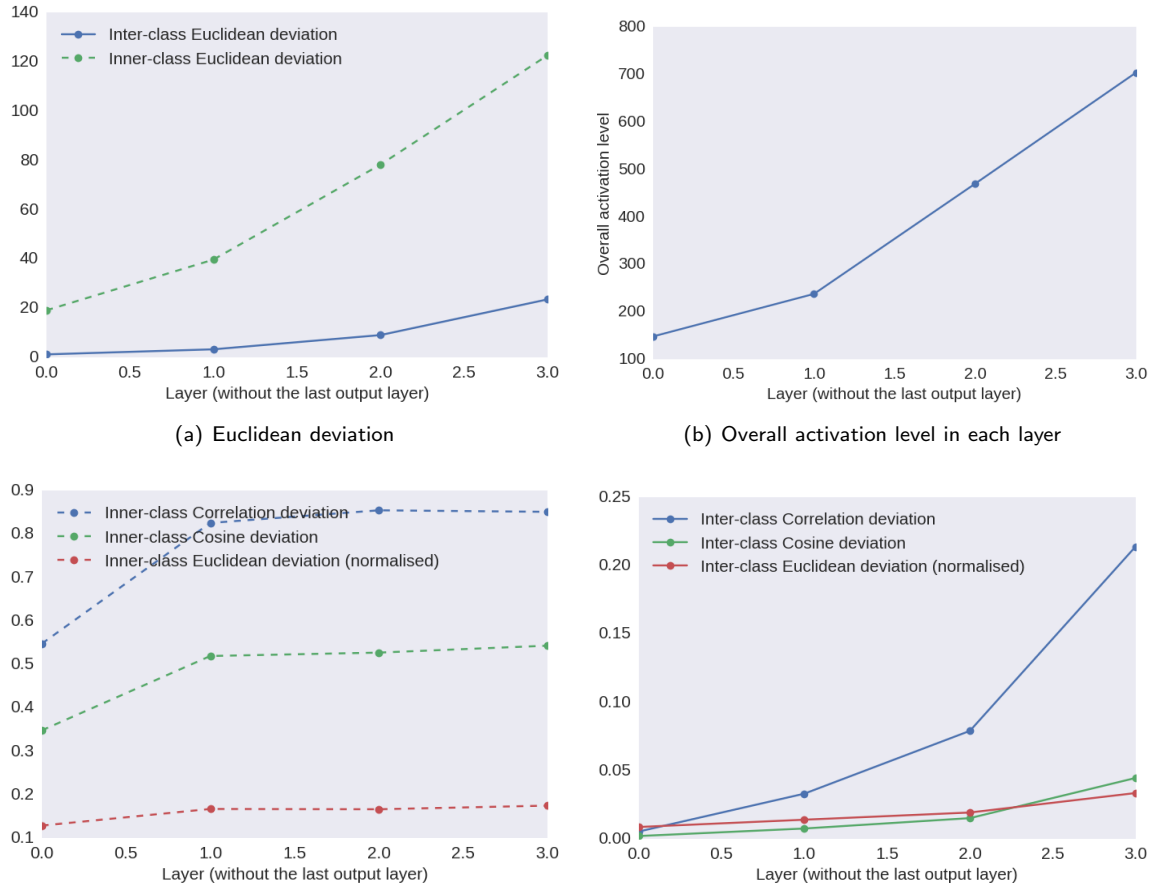
than in 4-layer networks for jump detection task. The reason might that this property requires either the depth of network, or the complexity of the task.

6.4.3 Insights on "memorizing network" with measurement-based analysis

In order to have better insight into the memorization ability of the network, we apply the measure-based analysis on the network trained with data with randomly shuffled labels. The details of

Measures on "memorizing network" with randomized labelled data.

The measurement is tested with random-labelled data (false labels), by which the model is trained (see Fig. 6.13).



(a) Euclidean deviation

(b) Overall activation level in each layer

(c) Inner-class deviation measures that are not sensitive to the activation level

(d) Inter-class deviation measures that are not sensitive to the activation level

Figure 6.13: Measures with the "memorizing network" on the randomized labelled data. (The class representation are from Fig. 6.3.)

Comparing the behaviour of the "memorizing network" and the normal network, there exists similarities as well as differences.

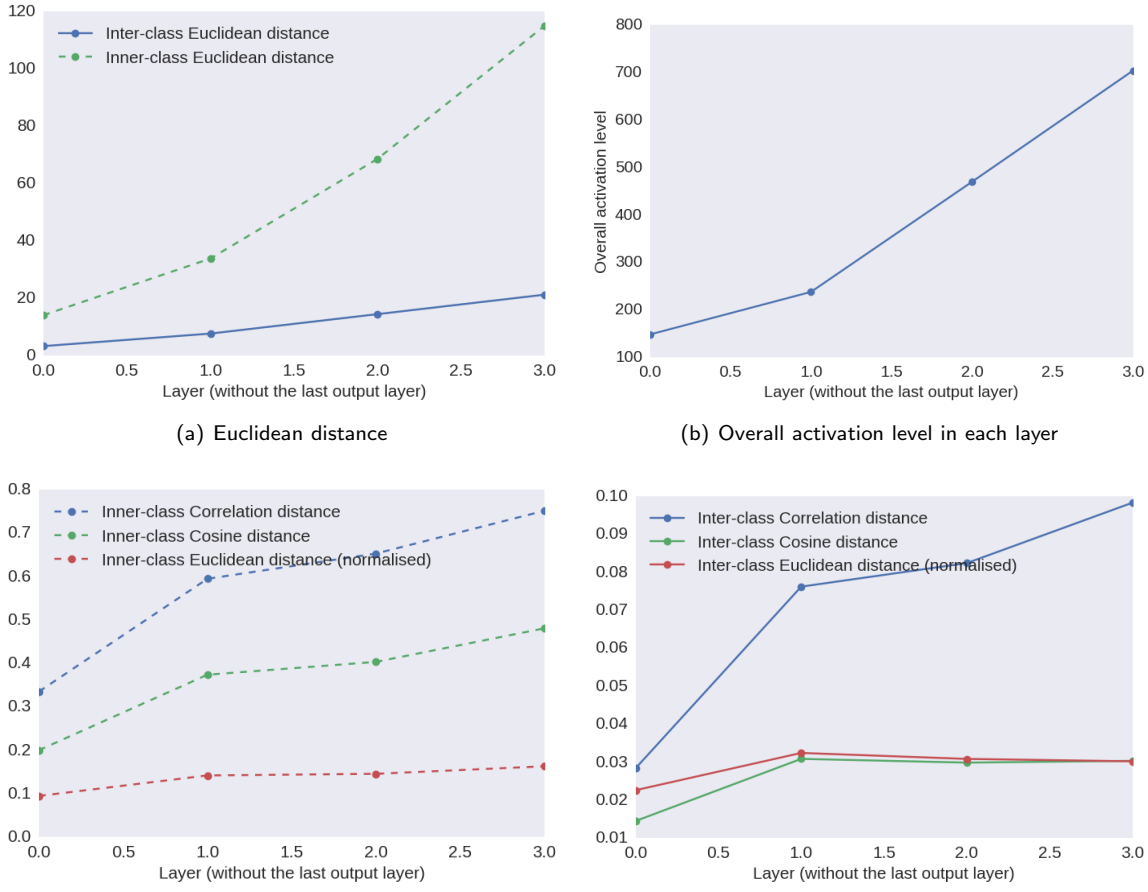
The similarities mostly exist in Figs. 6.13(a) and 6.13(b). The Euclidean deviation and the overall activation level shows the same tendency as in the normally trained network. As for the differences, one is also about the activation level in Fig. 6.13(b). The initial activation level in the input layer is the same, since the change of label does not interfere with the averaged activations over all classes. However, the activation in the following layers are significantly higher than in the normal model (Fig. 6.8(a)). The

higher overall activation probably indicates that the network requires more space to produce gradient to represent the randomized data. Some other differences are also worth mentioning:

- Regarding the inner-class deviation measures that are not sensitive to the activation level (Fig. 6.13(c) and Fig. 6.9(a)): whereas in the normal model the inner-class deviation gradually decreases, in the "memorizing network", the deviation climbs to a high level from the input to the first layer representation, and then remains constant.
- Regarding the inter-class deviation measures that are not sensitive to the activation level (Fig. 6.13(d) and Fig. 6.9(b)): The normal model shows a (gentle) drop followed by a (slight) increase. In the "memorizing network" in contrast, the inter-class deviation significantly increases with layers.

Measures on "memorizing network" with correctly labelled data.

In order to further understand the difference in behaviour of the "memorizing network" and the normal "learning network", the measurement analysis is also tested on the data with correct labels (see Fig. 6.13). Recall that in Section 6.2.4, it is mentioned that the visualization of the class representations in hidden layers are similar for the "memorizing network" and the normal learning MLP models. While some differences of network behaviour can be deduced from the measures.



(a) Euclidean distance

(b) Overall activation level in each layer

(c) Inner-class cosine and correlation distance

(d) Inter-class cosine and correlation distance

Figure 6.14: Measures with the "memorizing network" on the truly labelled data.
(The class representation are from Fig. 6.3.)
Same model as in Fig. 6.13 but different labelling.

The observation is very similar to Fig. 6.13 from the same "memorizing network" with different

labelling. But it is totally different from Fig. 6.9 from the different model with same labelling. The t-SNE visualization (Fig. 6.5) provides the same condition. While the condition is opposite from the hidden activation visualization, which is different in the former case, and similar in the latter case. More specifically,

- In inner-class case, the "memorizing network" also greatly expands the gap of the data instances in the same true class. This behaviour is totally different from the behaviour seen in the normal "learning network" with same labelling. This unusual behaviour can be understood by realising that the network aims to separate the data points in the same actual class and re-design to the false class. The t-SNE visualization also provides the same view (Fig. 6.5).
- In inter-class case, the network is not trying to increase the gap between different classes, although the deviation still increases slightly. This result is natural because the network is not aiming at learning the actual labels.

Some insights

We try to give an explanation for the observed differences in the inner- and inter-class deviation measures (specifically, the ones that are not sensitive to the activation level). This might be helpful to obtain some more insights into the memorization ability and how the generalization power is obtained by the NNs.

The totally different behaviour of the "memorizing network" and normal model indicates that even for the same network structure, memorizing random noise data is different from learning through genuine generalization. In a classification problem, we expect the network behaviour to either make the data representations in the same class more similar, or make the data representations of the separate classes more different. The former is equivalent to decreasing the inner-class deviation in our measurement-based analysis, while the latter results in increasing the inter-class deviation.

Observations based on our experiments, suggest that normal learning is performed by decreasing the inner-class deviation, which is probably the more efficient means in the learning (optimization) process of the network. Although the differences between classes is not (significantly) enhanced, the network forces all data in the same class into the same representation or encoding pattern. This produces a relatively thin manifold in the real hidden feature space. Thus, the generalization power is from the network behaviour to enhance the representation by delineating the various cases more clearly in representation space.

In contrast, the memorizing is performed by increasing the inter-class deviation, but the inner-class deviation also increases to a high level. The memorization is by expanding the encoding space to search for difference between different classes. The noise labelled data forces the network to perform the much less efficient behaviour. Thus, the memorization model can hardly acquire generalization power. Put differently, there exist no such a thin manifold in the expanded real hidden feature space, and the test data instances which are fed into training cannot converge to a certain encoding pattern for a certain class.

6.4.4 Summary

From exploring the inner- and inter-class deviation, the following conclusions are drawn from the experiments.

1. From the original input to all hidden layers, the overall activation level increases with layers. One possible explanation is that the representation with higher value might be helpful for the network to distinguish the data instances.
2. The network that performs learning in the normal sense will make data points in the same class converge to a small group of certain encodings. Put differently, by passing through successive hidden layers, the representations of the data instances in the same class are gradually forced closer together. Moreover, if the original representation of data in the same class are not close

to each other, the powerful network is able to make them significantly closer to each other in one layer. Evidently, such behaviour of the network helps classification. That is how learning is performed and how the network provides generalization power.

However, the network does not necessarily enhance the gap of different classes by increasing the distance. In inter-class case, there might be a tendency that the deviation first drops down and then increases. However, this drop is not as significant as in the inner-class case.

3. The memorization network trained on the randomized data shows behaviour that differs from normal models. The measurement-based analysis is able to provide a view on the difference of learning and simply memorizing by an NN with the same structure. The deviation of the representations of the data is extremely enhanced, even inside the same class. After the hidden layers the final layer designates each data point in the class-confused expanded distribution to a given class.

Chapter 7

Conclusion

Contents

7.1 About the Jump Detection Task

This work mainly focuses on the interpretation of NN models by performing a simple supervised learning task that aims to detect the jump locations in signals. We have chosen this task based on the three considerations:

- the data are very simple one-dimensional signals. NN and deep learning models are in fact more well-known for the ability and excellent performance in complex data such as rule-less images. Usually researchers in the deep learning community are more likely to attempt interpreting deep NNs for these complicated tasks. However, we expect to have some discoveries in a different direction, with the 1-dim signal of which the mathematical properties are well studied.
- there exist meaningful theoretical approaches to solve the problem. By comparing the NN solution and the theoretical solution, we expect it to be helpful in understanding the behaviour of the network better.
- the different classes in the task are not isolated. They are continuous in input space (and in fact this property applies to the layer activation spaces as well), although the one-hot representation of the jump position is discrete. This property helps in understanding the relation of data in input space and layer activation spaces and evaluation of the distance between adjacent classes in Section 6.4.3.

In fact, all of the three factors help in analysis of experimental results to some extent.

Performance of all tested learning models for the jump detection task is listed in Table 7.1 The reliability of the four measures are tested with the "bottleneck" network (see Fig. 4.3). Three of the four measures (L_1 error, 99 percentile L_1 error and 99 percentile L_2 error) precisely reflect the performance of a learning model.

	Complete data		99 percentile	
	L_1 error	L_2 error	L_1 error	L_2 error
Signal processing approach	0.5532	1.4946	0.4917	0.7443
1-layer NN	$\boxed{0.3065}$	0.8043	$\boxed{0.2606}$	0.6304
2-layer MLP	0.3083	$\boxed{0.7722}$	0.2675	$\boxed{0.6222}$
3-layer MLP	0.3583	0.8132	0.3169	0.6667
4-layer MLP	0.3913	0.8428	0.3531	0.7075
1-conv-layer CNN	0.3248	0.7862	0.2854	0.6452
2-conv-layer CNN	0.3213	0.7592	0.2839	0.6309
3-conv-layer CNN	0.3458	0.7946	0.3089	0.6721
"Bottleneck" network 100-10-60	0.7980	1.4658	0.7370	1.0820
"Bottleneck" network 100-2-60	1.5486	1.9874	1.5053	1.9086
"Bottleneck" network 100-25-10-25-60	0.6478	1.3121	0.5742	0.9255
"Bottleneck" network 100-25-2-25-60	0.9830	1.3136	0.9542	0.2621

Table 7.1: Performance of all tested learning models for the jump detection task. Boxed numbers indicate the model achieves the lowest error with the measure.

The performance of all NN models significantly improves on the traditional signal processing approach (excluding the bottleneck networks). Among all NN models, the shallow fully connected networks (1-layer NN and 2-layer MLP) have the best performance. In fact, there is still much room for improvement of the performance, especially in the deeper networks. In deeper networks, some tricks such as dropout [52], batch normalization [53] and more state-of-the-art activation functions [54, 55] are expected to be helpful in preventing the gradient vanishing problem and enhancing the performance. Some simple preprocessing can be beneficial to lower the error as well. However, all in all, the slight difference in the performance of the simple task on artificial data is by no means important. We only aim at interpretation of learnt networks, and we focused on simple problems because we expected that the learning process of NNs being primitive would simplify the problem.

7.2 Towards a Final Conclusion: Principal Subpatterns and Generalization

In Chapters 3, 4, 5 and 6, we explore experiments based on the jump detection task including direct visualization of weights and hidden layer activation, Gradient Ascent visualization, memorizing randomized noise labels and measuring distance of data in the same or different classes. We draw four main conclusions from the empirical analysis, which contribute to interpretation or explanation of neural networks and deep learning.

Direct interpretation via weights and activations might be feasible for certain tasks and certain network structures with limited expressivity. The first main conclusion (from Chapter 3) is that in some cases and for some tasks, with a well designed structure, the trained NN model can be human-interpretable directly from the weights and hidden activations. The most obvious case we have is the 1-layer fully connected NN, in which the weights can be regarded as a convolution of a smoothed first derivative extractor. Remarkably, the NN solution achieves a better performance than the filter in the proposed signal processing approach. However, this conclusion is not general because to design such a structure is task specific. The neural network approximation for most functions and especially almost all nonlinear functions can hardly be human interpretable.

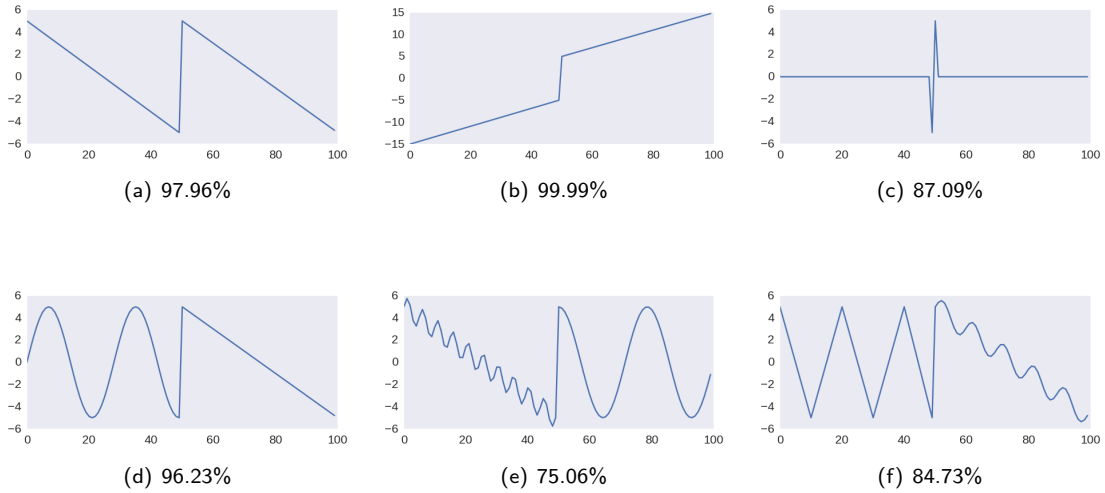
Hidden layer activations encode spatial information in the feature space The second main conclusion is from Chapter 4. With the "bottleneck" MLP networks, we demonstrate that the hidden layer activations are a sort of encoding format that can be regarded as the coordinates in the layer activation space. Thus,the hidden layers are in fact performing a mapping from the neuron activation

space of the last layer to that of the current layer. Our data are continuous in the initial input space (as demonstrated with t-SNE visualization in Fig. 3.2), and that continuous spatial information is still preserved in the intermediate hidden layers. In fact, this conclusion is not a new discovery but just a way of understanding of hidden layer activations. Nevertheless, such an understanding inspires the idea to inspect the spatial relation of classes in the subsequent analysis, which plays an important role in our final argument.

Gradient Ascent visualization suggests NNs learn first derivative positions in jump detection task The third main conclusion is from the Gradient Ascent visualization in Chapter 5. The learnt NN is capable of recognizing some concrete patterns that are not previously learnt by the network. In our jump detection task, the networks are trained on signals with step-function patterns, i.e. they have an average value of 0 before jump and keep an average level of 3 after jump. Through the use of gradient ascent visualization, it is shown that a signal that drops gradually after the jump is also recognized with over 99% confidence by the network prediction, as long as the discontinuity occurs at the appropriate position.

Empirically, the multi-layer networks learn the position of the maximum first derivative equally well as the 1-layer NN, although the former are not interpretable directly from weights. In previous research works, Gradient Ascent visualization has only been applied to 2-dim natural images, which is a more complex data type than the artificially generated 1-dim signals in our task. Obviously, the natural images have less clear statistical properties than the 1-dim signals. As a consequence, it has only been possible to find some shapes that roughly look like the ones in the training class. Therefore, Gradient Ascent visualization has usually not been able to draw clear and specific answers regarding what the network is actually learning such as e.g. a first derivative extractor in our case.

This result also applies to the problem of generalization in NNs. The network learns only the principal subpattern characteristic of a jump (i.e. the maximum first derivative). But away from the jump-location, as demonstrated in Fig. 7.1, the network does not care about what happens (e.g. whether there is a gradual drop somewhere), unless a larger first derivative value is caused. In fact, the network has not learnt to check whether there is an additional drop or not. But empirically, the network suffices to generalize the learnt samples to all data with the principal jump subpattern.



(a) 97.96% (b) 99.99% (c) 87.09%

(d) 96.23% (e) 75.06% (f) 84.73%

Figure 7.1: The network recognizes these signals as well.
All are predicted as jump location 50 correctly by a 2-layer MLP with confidence listed below each graph.
Although they do not resemble the data instances used for training, these signals all have in common the principal subpattern that they all have a maximum (positive) first derivative at position 50.

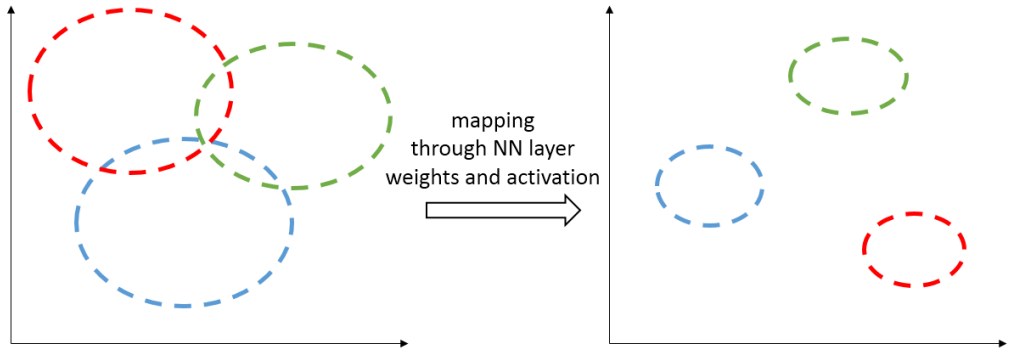
Network learning structured data differs from memorizing noise data in layer behaviours The fourth main conclusion is from Chapter 6, which is also the most significant contribution of this

thesis. In exploring hidden layer features of networks, we conclude that the behaviour of the NN while memorizing random data or learning reasonably structured data is different.

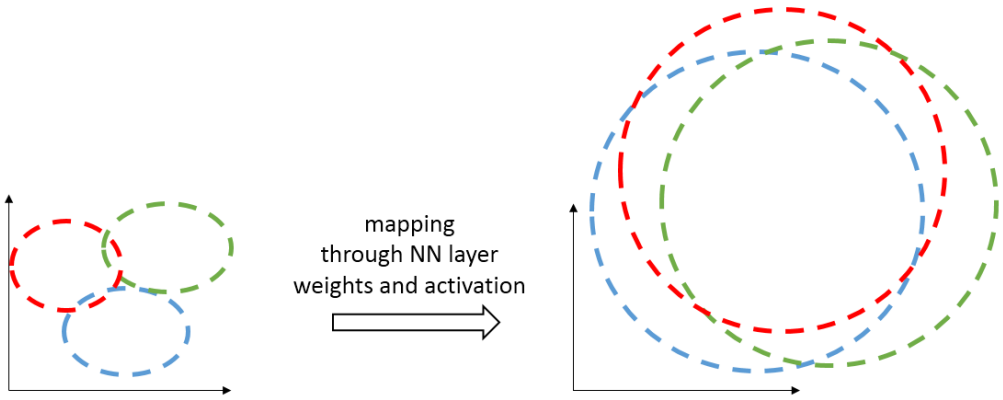
With this observation that *deep NNs do not perform learning through memorization* we address the so-called "memorization hypothesis" which recently emerged as a widespread concern. Proposed by Zhang et al.[46], the "memorization hypothesis" claims that deep NNs perform learning via memorization, based on the empirical observation that even optimization on random labels is still possible with deep NNs. According to the authors in the above paper, this shows that the capacity of deep NNs (in terms of adjustable parameters) is so huge that even large collections of random data can be memorized. The NN's ability of memorizing noise data contradicts the current mainstream intuition of why Deep Learning works so well. In essence, the rationale is that for data representing natural phenomena, there exists some "thin" manifold in the high-dimensional parameter space that can be discovered during training. However, the capability of memorizing noise data does not necessarily mean that NNs always learn via memorization. In our experiments we observed two different learning behaviours which we illustrate using the figures below.

Fig. 7.2(a) is the case of networks that learn reasonably structured data (i.e. there is a genuine learning task). As mentioned before, the layer's function is mapping data representations in the activation space of the last layer to that of the current layer. Empirically, the layers in networks that perform learning with normal data in fact shrink the range of distribution of data of each class in the activation space. Therefore, the classes are more distinguishable through multiple hidden layers. In terms of generalization in this case, an instance in or close to the class is pulled back through layer mapping to the shrunken small range of representation in the activation space. Such a relatively reasonable means of generalization ensures good generalization ability.

Fig. 7.2(b) is the network for memorizing randomized noise labels. The layer behaviour is different that it expands the range and the scale of the distributions of data in all classes in the high-dimensional layer activation space. Empirically, it takes some layers to completely merge the data from different classes and confuse the labels. After that, as illustrated in Fig. 7.3, the prediction has to be performed through over-fitting from the capacity of the over-parameterized network. Generalization performance in such a case is expected to be poor because the prediction is basically case by case. The distribution of the structured data can hardly be captured by such models.



(a) layer behaviour in networks that learn structured data



(b) layer behaviour in networks that memorize noise labels

Figure 7.2: Layer mapping in hidden activation space.
The dashed ellipses represent the range of the red, blue and green three classes (genuine labelling before randomization) in the activation space of each layer.
(a) In networks that aim at learning structured data, the layer behaviour is to shrink the range of data in each class in the activation space of the new layer, so that the data in the same class converge to a similar representation.
(b) In networks trained to memorize random noise labels, data in the original input space are distributed by same true class grouped together. However, the mapping function of the layer force the range of each class to expand greatly in the next layer activation space, so that the ranges of classes are merged, and the labels are confused.

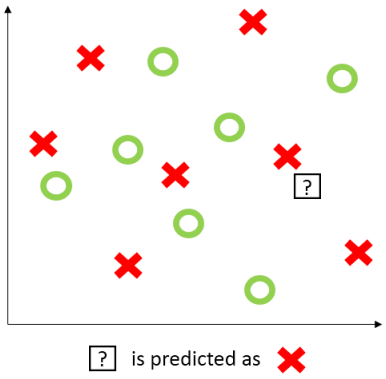


Figure 7.3: Demonstration of the last hidden layer activation space in a "memorizing network": memorizing noise labels is like case-based reasoning, which makes prediction by designating the class of the nearest data sample.
Suppose the network is trained to classify two categories (red cross and green circle) but the labels are completely blurred with randomization. In such a high-dimensional layer activation space illustrated in the figure, even after mapping functions from previous layers, there exists no clear identical hypersurface to segment the range of the two classes. Right after this layer, the noise labels are memorized case by case with the capacity of the over-parameterized network.
In terms of generalization in such a "memorizing network", the black square with a question mark inside would probably be predicted as a red cross. Because the area close to another red cross label is probably also learnt to be mapped into a representation for the red cross class by the NN's last layer. **Generalization is poor in such cases because they take no care of the distribution of the data**.

In addition to the empirical observation of the difference in layer behaviours, another possible indication that a NN's learning is not via memorization is that memorizing noise labels takes much more steps to converge. As suggested by Lin & Tegmark [45] (see Section 2.3.1), the structured data

from natural world are under constraints of physical laws such as symmetry, low polynomial order and hierarchical processes, which applies to deep NNs as well. As a result, deep NNs learning structured data is of more efficiency than memorizing random data. Thus, the network would naturally choose the more efficient behaviour in learning rather than the memorization behaviour.

Final conclusion: NNs achieve generalization power by learning principal subpatterns from structured data To summarize the conclusions from each separate chapter, as the final conclusion of this paper, we hypothesize that (deep) NNs learn principal subpatterns. As long as the data is structured or "natural" in that they exist in a narrow manifold in the high-dimensional space, the NN layers pull back high-dimensional representations to a shrunken small range for more reliable encoding. The data instances with the similar principal subpatterns are also converged to the range of encoding. For a specific example in our jump detection task, all kinds of signal are recognized as long as it has a significant maximum first derivative (as demonstrated in Fig. 7.1).

For some more insights into the problem of generalization proposed by Zhang et al.[46] (Section 2.3.2) explained from our way of understanding on how NN works, first we would like to define generalization ability as *to achieve good performance on structured or natural but unknown distribution of data*, rather than the narrow definition of *producing low difference between training error and test error*. Consider the range of the encoding of subpatterns being projected back to the input space. The resulting range in input data space is a set of all data with the resulting subpattern, which covers much more than barely the data with exact same patterns in training. NNs acquire good generalization ability through such a behaviour when learning structured data, which is totally different from NN's behaviour in memorizing noise data.

7.3 Future Work

Our work is an empirical study inspired by recently published research that aims to contribute to the understanding and interpretation of neural networks and deep learning. However, NNs have been known as black-box models for decades and the problem of understanding and interpreting NNs is by no means solved. Some directions for possible future study based on the results in this paper are discussed below.

Behaviour of the output layer One of the foremost unexplored questions is the function of the final output layer. Our measurement-based analysis of hidden layer activations only focus on the gradual change from the input layer to the hidden layers in MLP networks. However, the empirical observation in t-SNE visualization suggests that our methodology to compare the distance measures in layer activation spaces does not apply to the final output.

In the task to memorize random labels, we expect that the data with genuine labels first gradually break the structure of the narrow manifold in the high dimensional space, and second they gradually regroup into the classes indicated by the new labels. However, only the first half is observed from the input space to the activation space of the hidden layers in the t-SNE visualization (Fig. 6.5). The latter process, however, seems to be done abruptly only in the final softmax output layer.

Similarly, in learning normal structured data, we expect that there might be a gradual unfolding with consideration of the fact that the outputs are discrete one-hot vectors where the spatial relationship of the adjacent classes are not preserved anymore. However, from the input space to the last hidden layer activation space, the continuous distribution of the data set is maintained until the last hidden layer as well. One possible reason might be that our task can be tackled within one layer in practice. But such an excuse cannot provide an explanation for the same situation in the memorization model.

All this testifies to the fact that the mechanism of the final softmax output layer is still poorly understood and how each instance is mapped to the unique class is yet to be explored.

Real data set and state-of-the-art network structures Our work mainly focuses on, and indeed benefits from, the simple 1-dim signal task (except for a small experiment on MNIST data in Section 6.4.2). However, some interesting higher level behaviour of the network can only emerge from

more complex tasks. An example is that deep CNNs for vision tasks usually encode hierarchical properties in learning. The first layer usually learns low-level edge detectors, and deeper layers learn higher-level features [31]. In our experiments we have also observed the emergence of some simple hierarchical properties, e.g. in the CNN structure with two convolutional layers, the first conv layer performs smoothing and the second layer performs derivative extraction. But in this simple task the hierarchical processing is neither obvious in observation nor necessary for the networks to solve the task.

In fact, deep NN structures are well-known for the ability and high performance in tackling complicated real world tasks. Our artificial data are too structured in terms of distribution. The traditional signal processing approach suggests that the task can be tackled with even linear models. Therefore, the expressivity and the effective capacity of the network have not yet been much challenged (except for the task of memorizing random labels).

As a more specific direction, some of our methodologies are expected to be applicable in pixel-wise image tasks such as segmentation [43] or contour detection [56], since our task is a similar element-wise detection task. The interpretation from the perspective of semantic neurons are indeed from the pixel-wise semantic segmentation task (as introduced in Section 2.3.3).

Training dynamics, regularization We focus on interpretation of static networks which are already trained, with no regard to the difference within the optimization process. In fact, deep learning researchers usually are more interested in dynamic than static properties because it is more related to achieving higher performance. Similarly, some widely applied regularization approaches such as dropout and batch normalization, which play important roles in performance, have not been considered either.

When close to finishing this thesis, we come across a very recently published research by Arpit et al.[57] working on the similar direction that can be an inspiration for our future work. These authors also reject the memorization hypothesis based on observations that deep NN models behave differently when learning real data as opposed to memorizing random noise. But unlike our methodology, they take no consideration of the internal layer behaviours, but focus mainly on the behaviours in the training dynamics. More specifically, the model's behaviour for learning real data vs. memorizing random data are different in terms of loss-sensitivity (effect of each sample on average loss) varying with training steps, density of decision boundaries varying with training steps, and time-to-convergence varying with model size. These results inspire that our measures of inner- and inter-class distance in each layer are also applicable to the training dynamic, i.e. to explore how the measures vary with training steps, etc. Our work contributes to understanding the internal layer behaviour. As training dynamic is considered, the distance measures are expected to provide more insights on the learning mechanism of NNs.

Bibliography

[1] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[2] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961.

[3] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[4] Paul John Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974.

[5] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Max Welling. Are ml and statistics complementary? In *IMS-ISBA Meeting on 'Data Science in the Next 50 Years*, 2015.

[8] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[9] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697*, 2016.

[10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[11] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[13] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[16] Paulo Cortez and Mark J Embrechts. Using sensitivity analysis and visualization techniques to open black box data mining models. *Information Sciences*, 225:1–17, 2013.

[17] Julian D Olden and Donald A Jackson. Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological modelling*, 154(1):135–150, 2002.

[18] Orna Intrator and Nathan Intrator. Interpreting neural-network results: a simulation study. *Computational statistics & data analysis*, 37(3):373–393, 2001.

[19] Robert Andrews, Joachim Diederich, and Alan B Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-based systems*, 8(6):373–389, 1995.

[20] LiMin Fu. Rule generation from neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(8):1114–1124, 1994.

[21] Rudy Setiono and Wee Kheng Leow. Fernn: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12(1):15–25, 2000.

[22] Makoto Sato and Hiroshi Tsukimoto. Rule extraction from neural networks via decision tree induction. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3, pages 1870–1875. IEEE, 2001.

[23] J Ross Quinlan. *C4. 5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[24] M Gethsiyal Augasta and T Kathirvalavakumar. Reverse engineering the neural networks for rule extraction in classification problems. *Neural processing letters*, 35(2):131–150, 2012.

[25] Mark Craven and Jude W Shavlik. Using sampling and queries to extract rules from trained neural networks. In *ICML*, pages 37–45, 1994.

[26] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[27] Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. Visualizing and understanding neural models in nlp. *arXiv preprint arXiv:1506.01066*, 2015.

[28] Hendrik Strobelt, Sebastian Gehrmann, Bernd Huber, Hanspeter Pfister, and Alexander M Rush. Visual analysis of hidden state dynamics in recurrent neural networks. *arXiv preprint arXiv:1606.07461*, 2016.

[29] Antonio Torralba. Drawnet. http://people.csail.mit.edu/torralba/research/drawCNN/drawNet.html. Accessed June 20, 2017.

[30] Google. Tensorflow playground. http://playground.tensorflow.org/. Accessed June 20, 2017.

[31] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[32] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.

[33] Luisa M Zintgraf, Taco S Cohen, and Max Welling. A new method to visualize deep neural networks. *arXiv preprint arXiv:1603.02518*, 2016.

[34] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *University of Montreal*, 1341:3, 2009.

[35] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[36] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[37] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[38] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

[39] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[40] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

[41] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2528–2535. IEEE, 2010.

[42] Matthew D Zeiler, Graham W Taylor, and Rob Fergus. Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2018–2025. IEEE, 2011.

[43] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[44] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[45] Henry W Lin and Max Tegmark. Why does deep and cheap learning work so well? *arXiv preprint arXiv:1608.08225*, 2016.

[46] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[47] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. Network dissection: Quantifying interpretability of deep visual representations. *arXiv preprint arXiv:1704.05796*, 2017.

[48] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[49] Michael A Nielsen. Neural networks and deep learning, 2015.

[50] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[51] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[52] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[53] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[54] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[55] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017.

[56] Wei Shen, Xinggang Wang, Yan Wang, Xiang Bai, and Zhijiang Zhang. Deepcontour: A deep convolutional feature learned by positive-sharing loss for contour detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3982–3991, 2015.

[57] Devansh Arpit, Stanislaw K. Jastrzebski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron C. Courville, Yoshua Bengio, and Simon Lacoste-Julien. A closer look at memorization in deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 233–242, 2017.