Under Pressure: Predicting pressure on micro CT-scans of archaeological soil samples using convolutional neural networks

Author: Bram Kreuger

Utrecht University

Supervisor: Dr. M. van Ommen

Second assessor: Dr. T.B. Klos

7,5 ECTS

Bachelor thesis for the degree in

Bachelor of Science

in Artificial Intelligence at the Faculty of Humanities

ABSTRACT

In order to protect buried archaeological remains from the pressure of build sites, it is important to assess how this pressure affects the remains. To aid this assessment we answer the question: Can we use convolutional neural networks (CNN) to predict the pressure that was applied to archaeological soil samples scanned by a micro CT-scanner? The dataset used in this study was created by repeatedly scanning a single, artifact rich soil sample. With each scan, an increasing amount of pressure is applied to the sample, damaging the artifacts. The soil sample was scanned by a micro CT-scanner. The amount pressure applied serves as the label, in the machine learning process while the images (slices from the 3D scan) from the various samples serve as input. A convolutional neural network making use of transfer learning, tries to predict the pressure belonging to the images when the image is fed as input. The test results on 261 unseen images after training the model show a 99.61% correct prediction rate. The results are very promising, but since the model was trained on a very specific dataset, they are not representative for a more general prediction of pressure applied to archaeological soil samples.

CONTENTS

1 INTRODUCTION

The use of artificial intelligence in the field of archaeology is relatively new. As is the use of micro CT-scanners within the field of archaeology. Our research question is: Can we use convolutional neural networks to predict the pressure that was applied to archaeological soil samples scanned by a micro CT-scanner?

1.1 Academic relevance

There have been some studies where machine learning has been used in the field of archaeology. Like in the following studies where it was used to classify vases (Van der Maaten, 2006) and coins (Van der Maaten & Postma, 2006). Machine learning techniques have been used to analyse remote sensing imagery (Lambers, Verschoof-van der Vaart, & Bourgeois, 2019). Machine learning has been applied on CT-scans within the field of medicine. (Wang et al., 2019) uses CNN's to predict the biological age of brains to predict dementia, a similar technique as we use in this thesis.

Since the Council of Europe ratified the 1992 Valletta treaty/Malta convention on European archaeological heritage (Council of Europe, 1992), parties involved in new construction works have to find out if there are any archaeological remains under the building site and whether this construction will damage these remains. There is currently a lot of ongoing research on ways to assess the damage being done. This thesis will present a method to aid researchers in their objective to assess these damages. The CNN will try to learn a pattern, which indicates the pressure applied to the sample. Whenever a new image is presented to the trained model, it will try to predict the pressure that was applied. For further reading about research on tools to assess the loss of archaeological value, we would recommend reading Ngan-Tillard et al. (2016). This thesis attempts to show that the use of CNN's is a worthwhile addition to these tools.

1.2 Structure of the thesis

In chapter two we discuss how the initial dataset is obtained, how we pre-process the images and form the dataset we will feed to the network. In chapter three we explain the basics of convolutional neural networks, the main tool we use in this thesis. In chapter four we discuss the implementation. In the last two chapters we analyse our results and we conclude the thesis. Finally, there is a bibliography and an appendix. In the appendix, links to our own code can be found. The code generates the CNN and other relevant functions. The appendix explains how to run this code.

2 THE DATA

The data we use comes from excavations at a neolithic site in the Swifterbant area (the Netherlands). The sites are dated between 4300 and 4000 B.C.. Since the soil is of a certain quality, the artifacts and ecofacts (a biological archaeological artifact not altered by humans) have been preserved very well. The specific data we use for this research comes from a refuse dump full of fish vertebrae. A single sample has been taken from this refuse dump and was scanned using a micro CT-scanner. The scanner creates a 3D representation of the sample and its contents. This 3D scan is made up by 750 2D images stacked on top of each other, meaning that each image shows a dissection of the scan. Here follows one of these images from the dataset (Huisman et al., 2014). The image shows bone fragments.

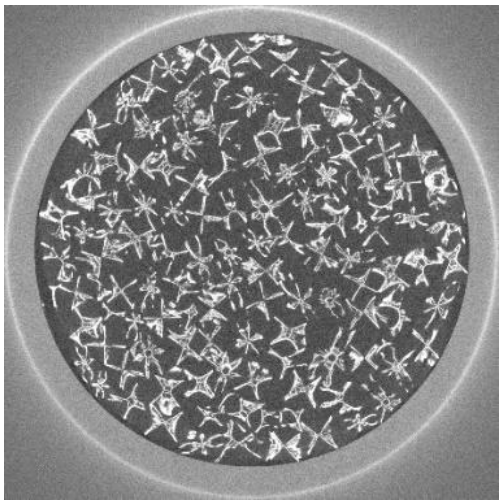

Figure 1: One of the 750 images which make up the 3D scans. Created by scanning the single sample.

The archaeologists chose to analyse the sample with a micro CT-scanner for the following reasons. When scanned with a micro CT-scanner the bones stay intact, it is a very non-destructive way to analyse sensitive data. Furthermore, the spatial relations of the bones are kept intact. Finally, even the smallest fragments are observable this way. This is all compared to the results from the traditional way of sieving artifact rich soil.

To try to simulate pressure from build sites, pressure is applied to the sample to investigate how harmful it would be to the bone fragments. The use of the micro CT-scanner is useful here, since the contents of the sample can be viewed while the spatial relations are kept intact. Furthermore the sample can be used again.

First, the sample was scanned without any pressure applied to it. This is repeated three more times, but then with increasingly more pressure applied. Leaving us with the following four scans: the 0kPa scan, the 40kPa scan, the 80kPa scan and finally the 160kPa scan. These four values will be used as the classes we wish to predict in our Neural Network.

Before we can use the data, it needs to be cleaned up. One scan consists of 750 images. We have four scans, so we have 3.000 raw images created by scanning the same sample four times, with an increasing amount of pressure applied to it. We will use the images as input data. Therefore, they need to contain useful information. As seen in figure one, the images contain a border around the sample, we try to remove most of this border by cropping the image. We also downsize it, because of computational reasons, this leaves us with 200 by 200 pixel images. The beginning of the scan (meaning a certain number of images) is blank, this is because the way the scanner works. The same goes for the final part of the scan. The amount of these blank images varies per scan. This leaves us with 410, 410, 269 and 206 images respectively.

3 HOW CONVOLUTIONAL NEURAL NETWORKS WORK

A convolutional neural network (CNN) is a form of a neural network which works particularly well for image classification. Since CNN's are just neural networks with some extras, a short introduction to neural networks is required. Here follows a simplified explanation without too much mathematics. The focus lies on the elements where the designer can make decisions when constructing a CNN.

3.1 The link between neural networks and the brain

The famous experiment by Hubel & Wiesel (1963), where they anaesthetized a cat and inserted a microelectrode into the primary visual cortex, showed that some neurons fired when the visual stimulus had a certain angle. Others fired when the stimulus had changing edges. This experiment showed that the brain combined the working of all these neurons with simple tasks to achieve a complex visual task. Marr & others (1982), also show that the brain stacks multiple simple or lower level observations, like edges, colours and textures. These observations combined form a complex end result. This is similar to how a neural network works.

3.2 Artificial neural networks

An artificial neural network (ANN) is a network loosely based on the biological neural network. They can "learn" to approximate functions without giving them specific rules. Therefore, they are used for a wide variety of tasks, such as image classification, speech recognition and playing games. The neural network (NN) contains neurons which are linked together by edges. The neurons take an input, compute a result by a non-linear function. The values are transmitted to the connected neurons via edges. Finally, these values are returned in the output layer. The edges have a weight assigned to them which increases or decreases the

strength of the transmitted value. Through the process of backpropagation, the weights get adjusted so that to make sure the network gives a better result.

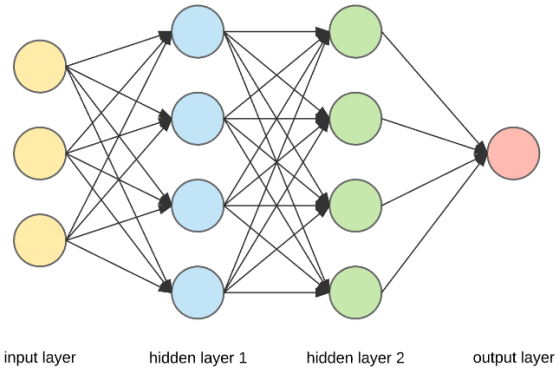

Figure 2: a basic example of an ANN. (Dertat, 2017)

Here follows a simplified step by step procedure of training a simple feed-forward, backpropagation neural network. The following sections will explain this process more clearly:

1. Initialize weights, and biases with random values.
    a. Take an input and pass it through the network.
    b. The hidden layers return a result based on their functions (as shown in formula 1) and the output layer makes a prediction.
    c. Backpropagation changes the weights and biases to minimize the error in the prediction of the output layer.
2. Repeat steps a, b and c until the desired result has been obtained. (often a certain minimal error, or a maximum number of epochs) When all training data has gone through the network, an epoch is finished.

3.2.1 Activation functions

As mentioned above, the neurons emit a certain value calculated by their function. We call this function the activation function. A neuron receives all the outputs of the previous connected neurons multiplied by their weights and sums it together. It then adds the bias to this summation and finally it applies the activation function. Then it passes it along to the next neurons. Here follows the formula for calculating the output of a neuron:

$$f(\sum_i w_i x_i + b).$$

Formula 1: The output function for a neuron.

Here f is the activation function, w and x are the weight and output of the incoming $i'th$ neuron and b is the bias of the current neuron.

There are multiple different activation functions, but the most important aspect is that they should be non-linear. An artificial neural network is considered a Universal Function Approximator, which means that it should be able to approximate any function. Linear

functions are simply not powerful enough to approximate these complex functions. Two popular activation functions are the sigmoid function and the rectified linear unit (ReLU) function

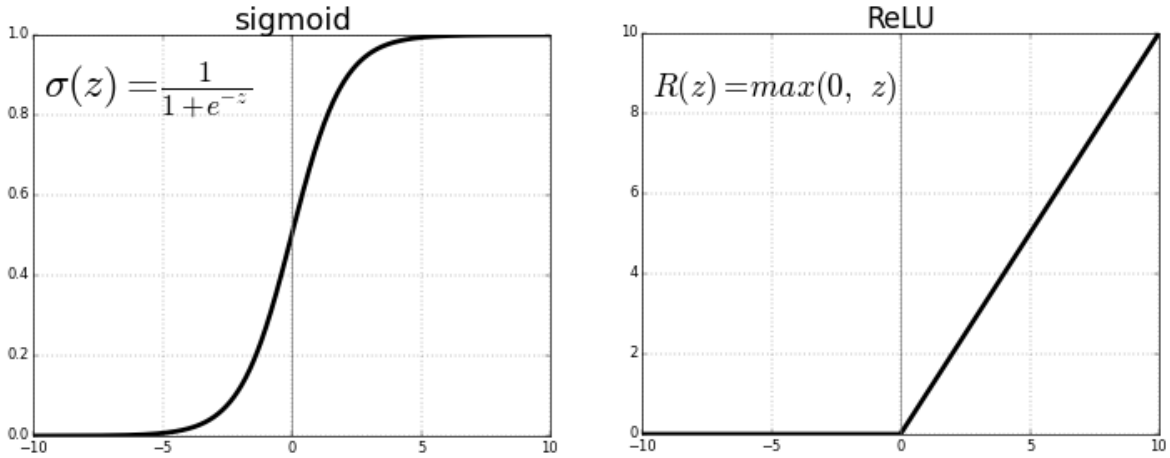

Figure 3: Two popular activation functions. The sigmoid and the ReLU (Sharma, 2017)

ReLU has become very popular these days due to it being very simple, it's either 0 or z. This reduces computing time. It also solves the problem which the sigmoid function suffers from, namely the vanishing gradient problem. An intuitive explanation of this problem is that a function like sigmoid squashes the input into a very small range of output values, namely between zero and one. This means that even if there is a massive increase in the input, the change in the output will be limited. This problem increases when these functions are stacked upon one another (which is what happens in a NN) a small increase becomes smaller every layer. The ReLU function fixes this by having a range of [0, ∞], therefore big changes in the input will result in equally big changes in the output.

3.2.2 Loss functions

The loss function is essential to the learning process of the neural network. After an input has passed through all the layers, there needs to be a value indicating how well the network predicted the outcome. This is where the loss function comes in. On of the most widely used loss functions, is the Mean Squared Error (MSE) function. The formula is as follows:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2.$$

Formula 2: Mean squared error.

Here N is the number of data points. We call $\left(y^{(i)} - \hat{y}^{(i)}\right)$ the residual since it is the difference between the actual value y and $\hat{y}$, the predicted value. This is an excellent example of a loss function. It calculates the squared error of each data point and then it takes the mean of all these errors.

Since in a multi-categorical image classification problem, we will not be calculating the error of every data point, but rather a probability a certain input corresponds with a certain label. Which means we will have an output of a vector of probabilities between zero and one, adding up to one. Therefore, we will need a different sort of loss function. A popular loss function is the Cross-Entropy loss function. The formula is as follows:

$$C_0(x) = -\sum_j y(x)_j \log \hat{y}(x)_j .$$

Formula 3: Cross-Entropy loss function.

Here $y(x)$ represent all the values of the actual label. So, the value of $y(x)_j$ is one when it corresponds to the class j, otherwise it is zero. $\hat{y}(x)$ represents the predicted probabilities of the class. Since there is only one class per input, the probability of the input being that class gets returned as a positive logarithm. Here follows a short example:

Class	$\hat{y}(x)$	$y(x)$	$y(x)_j \log \hat{y}(x)_j$	$C_0(x)$
Dog	0.85	1	-0.163	0.163
Cat	0.10	0	0	0.163
Car	0.05	0	0	0.163

Table 1: A dissection of the Cross-Entropy loss function

The smaller the outcome of the loss function the better the network performs. To optimize the network and to minimize the loss we use backpropagation and stochastic gradient descent. To fully explain these algorithms is outside the scope of this thesis, but in short it goes like this: gradient descent "decides" which of the output neurons' values need to increase or decrease. Then the weights connecting to the output layers are updated so they modify the output values in a preferred way. We would also like to modify the output of the neurons connected to the output layer, but since these values are calculated by a predefined activation function, we cannot do this. What we can do is update the weights connecting them to the previous layer. This is continued until the first weights are updated. If the loss decreases, gradient descent will continue in the same direction. Otherwise, the algorithms will modify the weights in different ways to find the minimum loss.

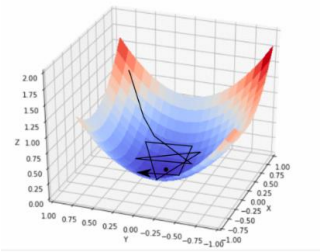

Figure 4: Example of Stochastic Gradient Descent. (Hello Paperspace, 2018)

3.3 Convolutional Neural Networks

A CNN is composed of different layers. There is an input layer, multiple hidden layers and an output layer. The most commonly used layers are convolutional layers, pooling layers, dropout layers and fully connected layers. The use of the convolutional layer really differentiates it from a normal neural network as we know it.

3.3.1 Convolutional layers

A CNN uses kernels to compare images to each other. The network learns a filter also called a kernel. It then tries to match it to a part of the input image and returns the amount of similarity. The CNN repeats this for every part of the image. This is where the convolution in convolutional neural networks comes from. The filters can be compared to the neurons in a normal neural network.

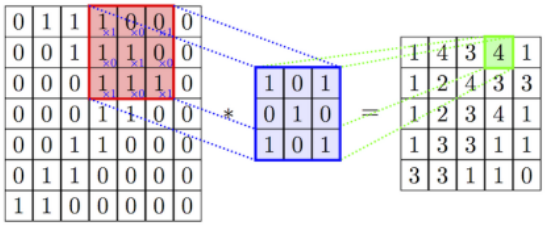

Figure 5: Parts of the image (red) are compared with a kernel (blue) and yields a convolved feature. (Laufer, 2019)

The CNN trains different filters on its own. Examples of these filters are edge detection, sharpen, blur, etc. Although the network learns this on its own the designer should still specify some parameters for the convolutional layer. Like the number of filters, the filter size, the stride and zero padding. These parameters are called hyperparameters.

3.3.1.1 Hyperparameters

1. Number of filters: This is also called the depth; the resulting convolved features of the convolutional layer are stacked on to each other. Since there are no clear rules for the number of filters needed for a CNN it is often decided after trial and error. Though it is good to keep in mind that when the number of filters goes up, a greater amount of details can be captured but the computational power required increases.
2. Filter Size: This is the number of pixels which the filter scans. The standard choice is 3 x3 which means every pixel looks around itself. When the filter size goes up, bigger details are captured.
3. Stride: This is the number of pixels the filter moves per step. Increasing the stride can be used for reducing the size of the convolved feature. This could increase

generalization. Increasing stride also reduces computing time (because you make fewer steps).

4. Zero padding: Do we add zero-valued pixels around the input image? This makes the convolved feature larger because the filter gets applied to every pixel in the input image. This can be beneficial when the filter size is large compared to the size of the input image.

After generating all these features, we need to scale down our data. The formula to calculate the number of pixels in all of the convolved features is as follows:

$$k * (r - a + 1) * (c - b + 1)$$

where k is the number of filters, $(r \times c)$ is the size of the input image and $(a \times b)$ the size of the filter.

Let's say we have an image of 64 x 64 (4.096 pixels in total) and we have learned 100 features over 3 x 3 inputs. We then end up with:

$$100 * (64 - 3 + 1) * (64 - 3 + 1) = 384.400$$

There is a massive increase in the number of pixels. It is necessary to bring this number down for computational reasons. That is where pooling comes in.

3.3.2 Pooling layers

Pooling layers often follow convolutional layers with the main incentive to bring down the size of the convolved features so that the computational power needed goes down.

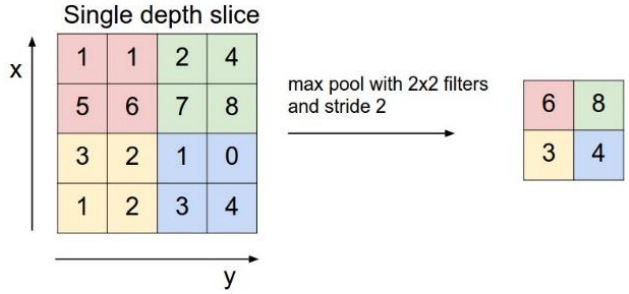

Figure 6: Pooling operation example. (Britz, 2015)

As seen in image 4 pooling applies a filter on parts of the input image. In this case, the filter takes the maximum value of the pixels in the 2×2 area. Because the maximum is taken, this form of pooling is called max-pooling. There is also a form where the average is taken. The number of pixels in the output after the pooling layer is calculated as follows:

$$\left(\frac{r - a}{S} + 1\right) * \left(\frac{c - b}{S} + 1\right) \text{(Asawa, n. d.)}$$

$(r \times c)$ is the size of the input. $(a \times b)$ is the size of the filter. S is the value of the stride.

The next two layers are found outside of convolutional neural networks as well. But since they are very often used within CNN's and since they are used for our implementation they will be discussed here.

3.3.3 Dropout layers

Dropout is a method to fight overfitting. It is a very simple technique which drops out nodes in the network based on a chosen probability. The network then continues training with reset weights and biases of these dropped out nodes. It helps against overfitting because of its randomness, it makes sure that the network doesn't rely as much on certain nodes and very specific patterns. Therefore, it increases generalization.

3.3.4 Fully connected layers

The last layer of a CNN is often a fully connected (FC) layer, also called a dense layer. A fully connected layer is simply a neural network as we know it. The input will be all the convolved features of the previous layer and the output will be a vector of N output neurons. In the case of the FC being the last layer these N neurons correspond to the number of classes which are to be predicted. The values of these N neurons together will be one. Therefore, every neuron will return a probability indicating the chance of the input value being of a certain class. To achieve these probabilities in the FC we need to alter the values. This is because our inputs probably don't have a value between zero and one, adding up to one. The solution will be to use a soft-max activation function in the FC. The formula is as follows:

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Formula 4: soft-max activation function.

where $\boldsymbol{z}$ are the inputs and k denotes the amount of inputs which is equal to the number of outputs. Here follows a short example. $\boldsymbol{z} = [1, 2, 5]$, by applying e^z on each of the elements in the vector we get this result: $[2.718, 7.389, 148.413]$. Now we add these together to get 158.52. Finally, all elements from the vector are divided by this value, resulting in: $[0.017, 0.046, 0.936]$. The rounded total of this vector is 1. This function is very useful to calculate the final probabilities.

4 IMPLEMENTATION

The implementation consists of a few essential elements and was based on an example by François Chollet, the creator of Keras. A link to the example can be found in the bibliography**.** All the code is written in Python using the Tensorflow and Keras packages since they provide a high-level API to create CNN's. The code was run in Google Colab, which is a service from Google where they offer 12 hours of continuous use of their powerful GPU. A link to our code can be found in the appendix.

We will first explain data augmentation, then transfer learning and the addition of these techniques to our problem. Then we will discuss our final implementation. In the next chapter we will discuss the results and its shortcomings.

4.1 Data augmentation

The biggest issue of this project is the amount of data. We have 1,295 images available. A portion of this data needs to be reserved for validation and for testing. This leaves us with a very small dataset. For example, a standard classification challenge provides 150,000 photos. When a CNN has little training data to learn from, it quickly overfits. This means that the CNN will learn the specific features of the input data, instead of learning features which generally describe the type of data. Therefore, there is always a need for more data, to learn more general features. The same goes for the other way. When you increase the complexity of the model, it tends to set the weights in such a way that it captures the smallest details. This again causes overfitting. As a rule of thumb, a simpler model is preferred over a more complex model which produces the same results. We call this principle Occam's razor: "when you have two competing theories that make exactly the same predictions, the simpler one is the better." (Thorburn, 1915)

Data augmentation can help with increasing the available data so to increase generalization. The idea is that data which is very similar to the original training data can be used as training data. Therefore, we apply a horizontal and a vertical flip to the images. We could have used other transformations, but then the images could have ended up being too different from the real-world data. In this implementation we use online data augmentation where for each round of training a batch of images is generated. This increases computation time but saves disk space.

4.2 Transfer learning

A common attribute among CNN's is that the first few layers create basic features, detecting edges, corners, etc. We call these features general. Since these first few layers are so general the objective of the CNN's or the intended use of the input images do not differ that much from each other. The last few layers are different. They are specific to the type of input and the objective of the CNN. We call them specific layers (Krizhevsky, et al., 2012). In transfer learning, we often take a pretrained model. In our case, we take the VGG16 model proposed by Simonyan & Zisserman (2014), as a base. VGG16 is a very deep CNN and the researchers trained it for weeks with high end GPU's on imageNet. ImageNet consists of over 14 million images with 1000 classes. Therefore, VGG16 generalizes very well. We feed our images to the pretrained VGG16 model and it creates the basic features. Since we only feed our dataset once to the pretrained model and we don't train the model, computation is very cheap on GPU's. We call these features bottleneck features. After generating these bottleneck features, we feed them to a network on top of this pretrained model, which then classifies it to our liking. In summary: The bottleneck features are generated by the VGG16. The VGG16 is not trained by us, it is merely being used to generate the features. Then we feed these bottleneck features to our own shallow model, which is being trained by us.

The usage of this technique has a few advantages with small datasets. First of all, since we tend to only have a limited number of trainable parameters, we don't have a massive difference between trainable parameters and number of images. This reduces the chance that the model will overfit. Secondly, since the bottleneck features are generated by a model trained on a wide variety of images, the features will be very general. Again, this reduces the chance of overfitting. Finally, since the VGG16 model is already trained, we only use it to generate the bottleneck features, which doesn't take long.

4.3 Our model

As explained in the previous paragraph we use the VGG16 model as a pretrained model to generate bottleneck features. The VGG16 has the following structure:

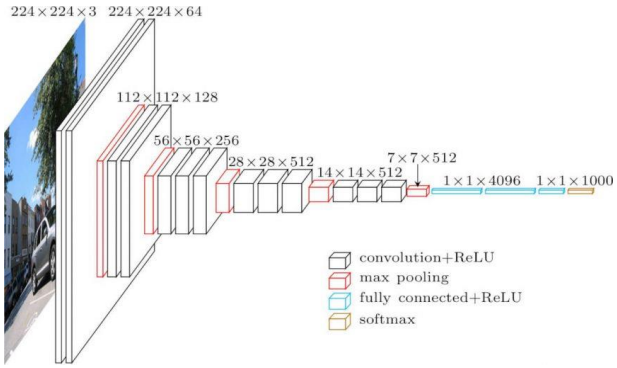

Figure 7: The architecture of the VGG16 model. (ul Hassan, 2018)

After feeding the augmented training images and the non-augmented validation images to the VGG16 model once, we feed the bottleneck features to the following network:

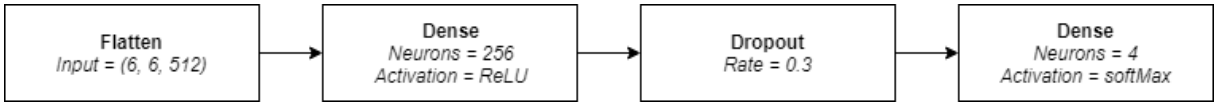

Figure 8: Our top model.

The VGG16 creates 512 features of size 6 by 6 pixels, this is the standard output of the VGG16 model. It then flattens the input from a (32, 6, 6, 512) tensor to 18432 neurons. These neurons are connected to 256 neurons in the next layer. The value of 256 is the same as in Chollet's example. When trying different values for the dense layer, we noticed that the validation accuracy would be lower with a lower amount of neurons in the dense layer. On the other hand, the accuracy would not improve with a higher amount of neurons and according to Occam's razor, we should choose the less complex method. A dropout layer with 256 neurons then 'drops' some of the neurons out, resetting their weights. A final dense layer with four output neurons and SoftMax activation gives a prediction. The model has a total trainable parameters of: $18{,}432 \times 256 + 256 + 4 \times 256 + 4 = 4{,}719{,}876$. For the hyperparameters, we chose a stochastic gradient descent (SGD) optimizer with a learning rate of 0.0001 and a momentum value of 0.9. This optimizer and the values were the same as in the aforementioned example.

5 ANALYSIS

Considering the size of our data, the model performed very well. We ended up using 768 images for training, 256 for validation and we held back 261 for a final test. Below are the

graphs picturing training metrics. Even on 500 epochs training took a few minutes.

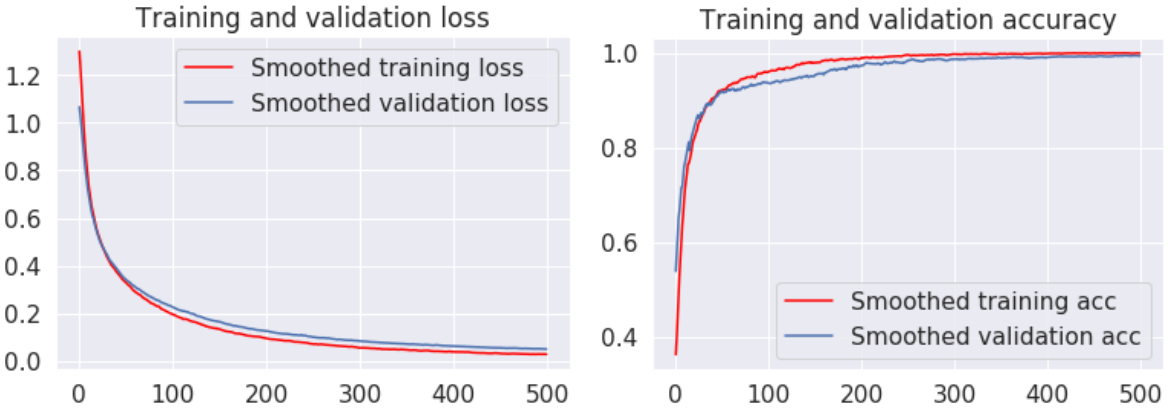

Figure 9: Training metrics.

After around 200 epochs the accuracy starts to converge, with training accuracy reaching 100% on some epochs. This could indicate overfitting, but we tried to optimize for validation. Hence, we used a model checkpoint callback, which saves the model at the first epoch with the lowest validation loss. After training we ran a test, with the 261 images we held back. We reached an accuracy of 99.61% and a loss of 0.0625. This clearly shows that the model can predict unseen data almost perfectly.

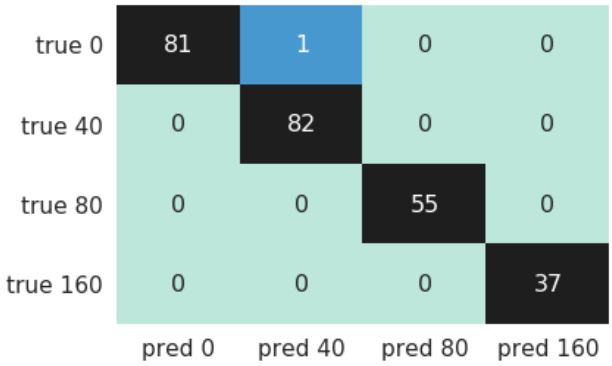

Figure 10: Confusion matrix for the final test.

These results are very promising, but there are arguments against and in favour of calling this a good model. Since the damage to the vertebrae in the scan increases gradually, it would be quite challenging for a human to see the difference between the adjacent classes. This pleads in favour of the model. On the other hand, the training, validation and test sets are randomly picked. Therefore, the model will not have a bias in favour of a certain class or a part of the scan. But due to the nature of the CT-scans slices adjacent to each other are extremely similar. This results in a very high test result because the test set barely differs from the training and validation sets. If there would have been a similar dataset from a different scan available, that would have been a more powerful way to prove the generalization of the model.

6 CONCLUSION

The main question of this thesis was: “Can we use CNN’s to predict the pressure that was applied to archaeological soil samples scanned by a micro CT-scanner?” If we just look at the results of the model, the answer would be yes. The model predicts the pressure nearly perfect on unseen data. But if we put the results in perspective with the data on which the model has been trained, it weakens the result of the overall research. The model is trained on a very specific type of soil scan, namely a prehistoric deposit, filled with fish vertebrae. Feeding it different types of samples will most likely not return the same results. Furthermore, the model only predicts the four ‘classes’ of pressure applied. When feeding it a similar sample with unknown pressure applied to it, the model would classify it nearest to one of those amounts. Hence a more sophisticated method should be developed where the prediction is a (semi) continuous number. Or at the very least interpolate the result between classes. Therefore, we can’t say that the model predicts any amount of pressure applied to any type of scanned archaeological soil sample. The model is clearly too limited for that. In conclusion, this research shows that CNN’s are capable of making predictions on this kind of data, even though the dataset is very small in size.

CNN’s are very powerful tools and models like this could be an asset to archaeologists trying to protect remains, buried under constructions. Since this research is only preliminary, further research should be done to make a more general model to make it useful for different types of soil. Or a range of models for different types of soil.

BIBLIOGRAPHY

Asawa, C. (n.d.). CS231n Convolutional Neural Networks for Visual Recognition. Retrieved 26 June 2019, from http://cs231n.github.io/convolutional-networks/#pool

Britz, D. (2015, November 7). Understanding Convolutional Neural Networks for NLP. Retrieved 19 June 2019, from WildML website: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

Council of Europe. (1992). European Convention on the Protection of the Archaeological Heritage (Revised). Retrieved 7 June 2019, from Treaty Office website: https://www.coe.int/en/web/conventions/full-list

Dertat, A. (2017, August 8). Applied Deep Learning - Part 1: Artificial Neural Networks. Retrieved 19 June 2019, from Towards Data Science website: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

Hello Paperspace. (2018, June 2). Intro to optimization in deep learning: Gradient Descent. Retrieved 19 June 2019, from Hello Paperspace website: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Hubel, D. H., & Wiesel, T. (1963). Shape and arrangement of columns in cat's striate cortex. *The Journal of Physiology*, *165*(3), 559–568.

Huisman, D. J., Ngan-Tillard, D., Tensen, M. A., Laarman, F. J., & Raemaekers, D. C. M. (2014). A question of scales: Studying Neolithic subsistence using micro CT scanning of midden deposits. *Journal of Archaeological Science*, *49*, 585–594. https://doi.org/10.1016/j.jas.2014.05.006

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25* (pp. 1097–1105). Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Lambers, K., Verschoof-van der Vaart, W. B., & Bourgeois, Q. P. J. (2019). Integrating Remote Sensing, Machine Learning, and Citizen Science in Dutch Archaeological Prospection. *Remote Sensing*, *11*(7), 794. https://doi.org/10.3390/rs11070794

Laufer, J. (2019, March 20). Image aesthetics quantification with a convolutional neural network (CNN). Retrieved 19 June 2019, from Jens Laufer website: https://jenslaufer.com/machine/learning/image-aesthetics-quantification-with-a-convolutional-neural-network.html

L.J.P. van der Maaten. (2006). *Computer Vision and Machine Learning for Archaeology*. Computer Vision and Machine Learning for Archaeology.

L.J.P. van der Maaten, & E.O. Postma. (2006). *Towards automatic coin classification*. Retrieved from https://lvdmaaten.github.io/publications/papers/EVA_2006.pdf

Marr, D., & others. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. San Francisco: WH Freeman.

Ngan-Tillard, D., Brinkgreve, R., Huisman, H. (D J.), Meerten, H. van, Müller, A., & Kappel, K. van. (2016). Tools for Predicting Damage to Archaeological Sites Caused by One-Dimensional Loading. *Conservation and Management of Archaeological Sites*, *18*(1–3), 70–85. https://doi.org/10.1080/13505033.2016.1181934

SHARMA, S. (2017, September 6). Activation Functions in Neural Networks. Retrieved 19 June 2019, from Towards Data Science website: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv:1409.1556 [Cs]*. Retrieved from http://arxiv.org/abs/1409.1556

Thorburn, W. M. (1915). Occam's razor. *Mind*, *24*(2), 287–288.

ul Hassan, M. (2018, November 20). VGG16 - Convolutional Network for Classification and Detection. Retrieved 19 June 2019, from https://neurohive.io/en/popular-networks/vgg16/

Wang, J., Knol, M., Tiulpin, A., Dubost, F., de Bruijne, M., Vernooij, M., … Roshchupkin, G. (2019). Grey Matter Age Prediction as a Biomarker for Risk of Dementia: A Population-based Study. *BioRxiv*, 518506. https://doi.org/10.1101/518506

François Chollet (2017) Using a pre-trained covnet. IPython notebook tutorial. https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.3-using-a-pretrained-convnet.ipynb

APPENDIX

The code for the thesis can be found here: https://colab.research.google.com/drive/1dLaCIYjFAyG0BhanvFjXwaRGSZF7G9ek.
To run it, you need a google account. You can copy the file to your drive and run it from there. To run the code, run the cells one by one by pressing the empty brackets in the left of the cell. If the cell has been run before the run button might show up as a triangle pointing right. For the first cell, you will be asked to grant google colab access to you drive, so that you can use the available files as data. Since the zip files containing the data can be viewed by anyone, no further permission requests are necessary. The datafiles can be downloaded from here: https://drive.google.com/open?id=1scvij2jNDJVrRrvAg0X-EVuc5Z8Tp2mf
Results may vary since the training of a model has some amount of randomness in it.