NEURAL NETWORKS FROM SCRATCH IN PYTHON

By Harrison Kinsley & Daniel Kukiela

Feel free to post comments/questions by highlighting the text and clicking the comment button to the right. Looks like this:

Do not resolve comments that are not yours.

Links to chapters:

Neural Networks from Scratch in Python

Harrison Kinsley & Daniel Kukieła

Copyright

License for Code

Readme

The objective of this book is to break down an extremely complex topic, neural networks, into small pieces, consumable by anyone wishing to embark on this journey. Beyond breaking down this topic, the hope is to dramatically demystify neural networks. As you will soon see, this subject, when explored from scratch, can be an educational and engaging experience. This book is for anyone willing to put in the time to sit down and work through it. In return, you will gain a far deeper understanding than most when it comes to neural networks and deep learning.

This book will be easier to understand if you already have an understanding of Python or another programming language. Python is one of the most clear and understandable programming languages; we have no real interest in padding page counts and exhausting an entire first chapter with a basics of Python tutorial. If you need one, we suggest you start here: *https://pythonprogrammingnet/python-fundamental-tutorials/* To cite this material:

Harrison Kinsley & Daniel Kukieła Neural Networks from Scratch (NNFS) https://nnfs.io

Chapter 13

Training Dataset

Since we are talking about datasets and testing, it's worth mentioning a few things about the training dataset and operations that we can perform on it; this technique is referred to as **preprocessing**. However, it's important to remember that any preprocessing we do to our training data also needs to be done to our validation and testing data and later done to the prediction data.

Neural networks usually perform best on data consisting of numbers in a range of 0 to 1 or -1 to 1, with the latter being preferable. Centering data on the value of 0 can help with model training as it attenuates weight biasing in some direction. Models can work fine with data in the range of 0 to 1 in most cases, but sometimes we're going to need to rescale them to a range of -1 to 1 to get training to behave or achieve better results.

Speaking of the data range, the values do not have to strictly be in the range of -1 and 1 — the model will perform well with data slightly outside of this range or with just some values being many times bigger. The case here is that when we multiply data by a weight and sum the results with a bias, we're usually passing the resulting output to an activation function. Many activation functions behave properly within this described range. For example, *softmax* outputs a vector of probabilities containing numbers in the range of 0 to 1; *sigmoid* also has an output range of 0 to 1,

but *tanh* outputs a range from -1 to 1.

Another reason why this scaling is ideal is a neural network's reliance on many multiplication operations. If we multiply by numbers above 1 or below -1, the resulting value is larger in scale than the original one. Within the -1 to 1 range, the result becomes a fraction, a smaller value. Multiplying big numbers from our training data with weights might cause floating-point overflow or instability — weights growing too fast. It's easier to control the training process with smaller numbers.

There are many terms related to data **preprocessing**: standardization, scaling, variance scaling, mean removal (as mentioned above), non-linear transformations, scaling to outliers, etc., but they are out of the scope of this book. We're only going to scale data to a range by simply dividing all of the numbers by the maximum of their absolute values. For the example of an image that consists of numbers in the range between *0* and *255*, we divide the whole dataset by *255* and return data in the range from *0* to *1*. We can also subtract *127.5* (to get a range from *-127.5* to *127.5*) and divide by 127.5, returning data in the range from -1 to 1.

We need to ensure identical scaling for all the datasets (same scale parameters). For example, we can find the maximum for training data and divide training, validation and testing data by this number. In general, we should prepare a scaler of our choice and use its instance on every dataset. It is important to remember that once we train our model and want to predict using new samples, we need to scale those new samples by using the same scaler instance we used on the training, validation, and testing data. In most cases, when we are working with data (e.g., sensor data), we will need to save the scaler object along with the model and use it during prediction as well; otherwise, results are likely to vary as the model might not effectively recognize these data without being scaled. It is usually fine to scale datasets that consist of larger numbers than the training data using a scaler prepared on the training data. If the resulting numbers are slightly outside of the *-1* to *1* range, it does not affect validation or testing negatively, since we do not train on these data. Additionally, for linear scaling, we can use different datasets to find the maximum as well, but be aware that non-linear scaling can leak the information from other datasets to the training dataset and, in this case, the scaler should be prepared on the training data only.

In cases where we do not have many training samples, we could use **data augmentation**. One easy way to understand augmentation is in the case of images. Let's imagine that our model's goal is to detect rotten fruits — apples, for example. We will take a photo of an apple from different angles and predict whether it's rotten. We should get more pictures in this case, but let's assume that we cannot. What we could do is to take photos that we have, rotate, crop, and save those as worthy data too. This way, we have added more samples to the dataset, which can help with model generalization. In general, if we use augmentation, then it's only useful if the augmentations that we make are similar to variations that we could see in reality. For example, we may refrain from using a rotation when creating a model to detect road signs as they are not being rotated in real-life scenarios (in most cases, anyway). The case of a rotated road sign, however, is one you better

consider if you're making a self-driving car. Just because a bolt came loose on a stop sign, flipping it over, doesn't mean you no longer need to stop there!

How many samples do we need to train the model? There is no single answer to this question — one model might require just a few per class, and another may require a few million or billion. Usually, a few thousand per class will be necessary, and a few tens of thousands should be preferable to start. The difference depends on the data complexity and model size. If the model has to predict sensor data with 2 simple classes, for example, if an image contains a dark area or does not, hundreds of samples per class might be enough. To train on data with many features and several classes, tens of thousands of samples are what you should start with. If you're attempting to train a chatbot the intricacies of written language, then you're going to likely want at least millions of samples.

Supplementary Material: https://nnfs.io/ch13
Chapter code, further resources, and errata for this chapter.