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 11

Testing with Out-of-Sample Data

Up to this point, we've created a model that is seemingly 98% accurate at predicting the testing dataset that we've generated. These generated data are created based on a very clear set of rules outlined in the *spiral_data* function. The expectation is that a well-trained neural network can learn a representation of these rules and use this representation to predict classes of additional generated data.

Imagine that you've trained a neural network model to read license plates on vehicles. The expectation for a well-trained model, in this case, would be that it could see future examples of license plates and still accurately predict them (a prediction, in this case, would be correctly identifying the characters on the license plate).

The complexity of neural networks is their biggest issue and strength. By having a massive amount of tunable parameters, they are exceptional at "fitting" to data. This is a gift, a curse, and something that we must constantly try to balance. With enough neurons, a model can easily memorize a dataset, which will cause overfitment and poor performance with previously-unseen data; however, it can not generalize the data, which is our goal, with too few. This is one reason why we do not simply solve problems with neural networks by using the most neurons or biggest models possible.

At the moment, we're uncertain whether our latest neural network's 98% accuracy is due to learning to meaningfully represent the underlying data-generation function or instead **overfitting** the data. So far, we have only tuned hyper-parameters to achieve the highest possible accuracy on the training data, and have never tried to challenge the model with the previously unseen data.

Overfitting is effectively just memorizing the data without any understanding of it. An overfit model will do very well predicting the data that it has already seen, but often significantly worse on unseen data.

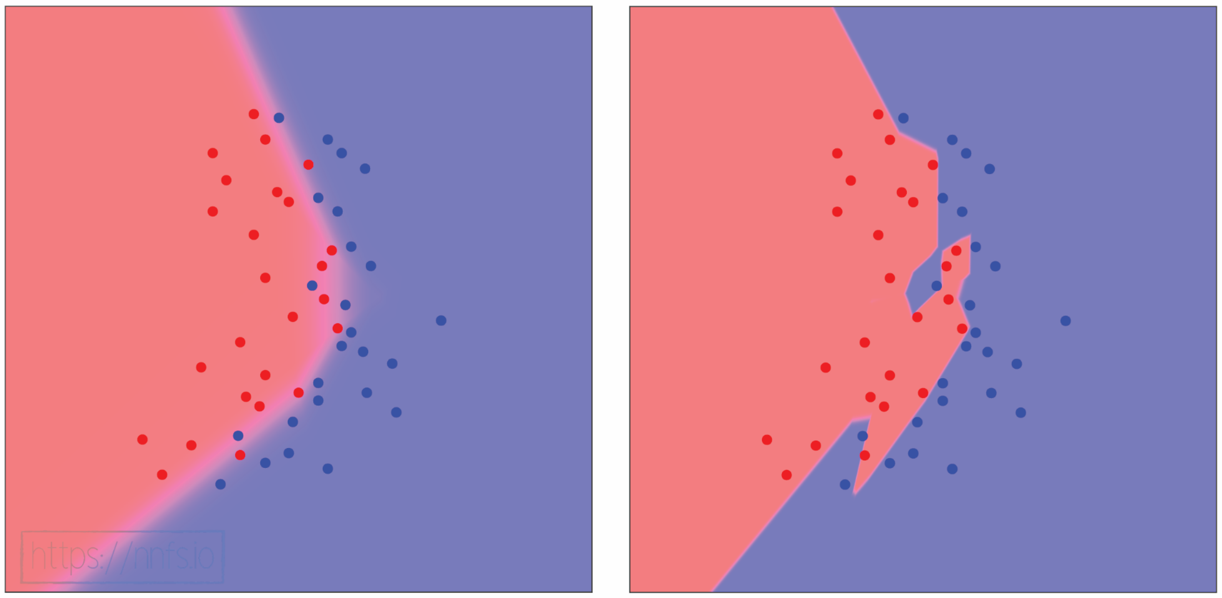

Fig 11.01: Good generalization (left) and overfitting (right) on the data

The left image shows an example of generalization. In this example, the model learned to separate red and blue data points, even if some of them will be predicted incorrectly. One reason for this might be the data that contains some "confusing" samples. When you look at the image, you can see that, for example, some of these blue dots might not be there, which would raise the data quality and make it easier to fit. A good dataset is one of the biggest challenges with neural networks. The image on the right shows the model that memorized the data, fitting them perfectly and ruining generalization.

Without knowing if a model overfits the training data, we cannot trust the model's results. For this reason, it's essential to have both **training** and **testing data** as separate sets for different purposes.

Training data should only be used to train a model. The **testing,** or **out-of-sample** data, should only be used to validate a model's performance after training (we are using the testing data during training later in this chapter for demonstration purposes only). The idea is that some data are reserved and withheld from the training data for testing the model's performance.

In many cases, one can take a random sampling of available data to train with and make the remaining data the testing dataset. You still need to be very careful about information leaking through. One common area where this can be problematic is in time-series data. Consider a scenario where you have data from sensors collected every second. You might have millions of observations collected, and randomly selecting your data for the **testing** data might result in samples in your **testing** dataset that are only a second in time apart from your **training** data, thus are very similar. This means overfitting can spill into your testing data, and the model can achieve good results on both the training and the testing data, which won't mean it generalized well. Randomly allocating time-series data as testing data may be very similar to training data. Both datasets must differ enough to prove the model's ability to generalize. In time-series data, a better approach is to take multiple slices of your data, entire blocks of time, and reserve those for testing.

Other biases like these can sneak into your testing dataset, and this is something you must be vigilant about, carefully considering if data leakage has occurred and how to truly isolate **out-of-sample** data.

In our case, we can use our data-generating function to create new data that will serve as out-of-sample/testing data:

```python
# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)
```

Given what was just said about overfitting, it may look wrong to only generate more data, as the testing data could look similar to the training data. Intuition and experience are both important to spot potential issues with out-of-sample data. By looking at the image representation of the data, we can see that another set of data generated by the same function will be adequate. This is just about as safe as it gets for out-of-sample data as the classes are partially mixing at the edges (also, we're quite literally using the "underlying function" to make more data).

With these data, we evaluate the model's performance by doing a forward pass and calculating loss and accuracy the same as before:

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)

# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y_test)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
    y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions==y_test)

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
>>>
...
epoch: 9800, acc: 0.983, loss: 0.075, lr: 0.04975621940303483
epoch: 9900, acc: 0.983, loss: 0.074, lr: 0.049753743844839965
epoch: 10000, acc: 0.983, loss: 0.074, lr: 0.04975126853296942
validation, acc: 0.803, loss: 0.858
```

While 80.3% accuracy and a loss of 0.858 is not terrible, this contrasts with our training data that achieved 98% accuracy and a loss of 0.074. This is evidence of over-fitting. In the following image, the training data is dimmed, and validation data points are shown on top of it at the same positions for both the well-generalized (on the left) and overfitted (on the right) models.

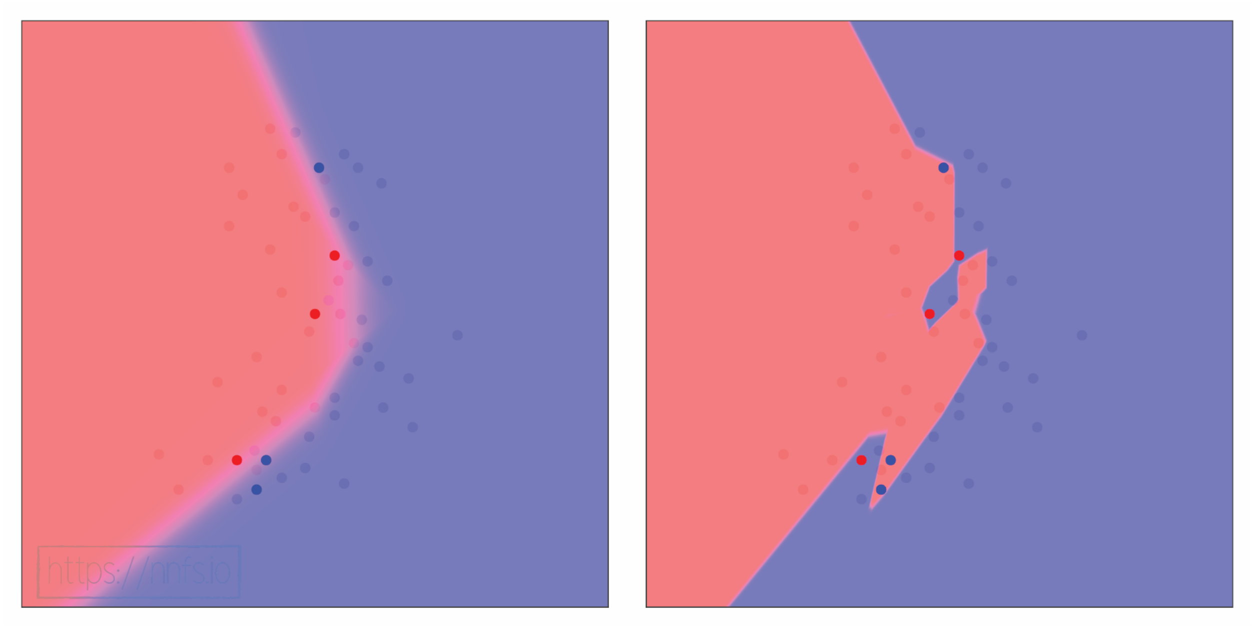

Fig 11.02: Left - prediction with well-generalized model; right - prediction mistakes with an overfit model.

We can recognize overfitting when testing data results begin to diverge in trend from training data. It will usually be the case that performance against your training data is better, but having training loss differ from test performance by over 10% approximately is a common sign of serious overfitting from our anecdotal experience. Ideally, both datasets would have identical performance. Even a small difference means that the model did not correctly predict some testing samples, implying slight overfitting of training data. In most cases, modest overfitting is not a serious problem, but something we hope to minimize.

Let's see the training process of this model once again, but with the training data, training accuracy, and loss plots dimmed. We add the test data and its loss and accuracy plotted on top of the training counterparts to show this model overfitting:

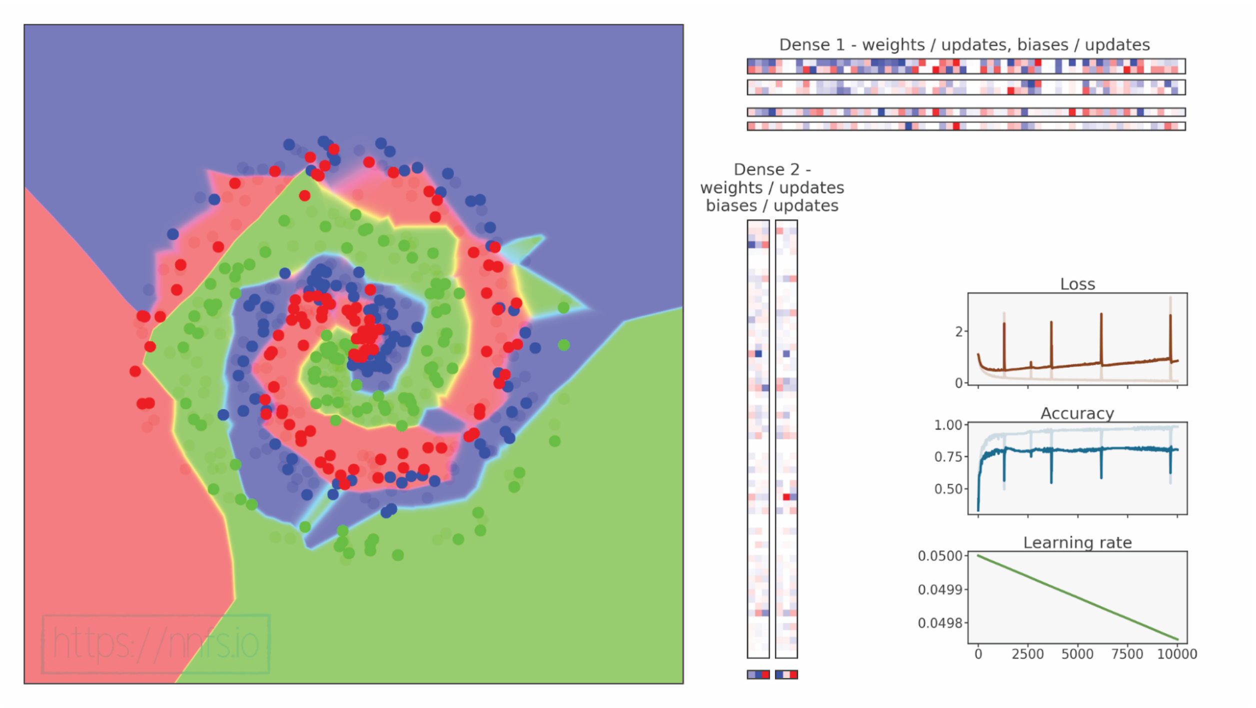

Fig 11.03: Prediction issues on the testing data — overfitted model.

Anim 11.03: https://nnfs.io/zog

This is a classic example of overfitting — the validation loss falls down, then starts rising once the model starts overfitting. The dots representing classes in the validation data can be spotted over areas of effect of other classes. Previously, we weren't aware that this was happening; we were just seeing very good training results. That's why we usually should use the testing data to test the model after training. The model is currently tuned to achieve the best possible score on the training data, and most likely the learning rate is too high, there are too many training epochs, or the model is too big. There are other possible causes and ways to fix this, but this is the topic of the following chapters. In general, the goal is to have the testing loss identical to the training loss, even if that means higher loss and lower accuracy on the training data. Similar performance on both datasets means that model generalized instead of overfitting on the training data.

As mentioned, one option to prevent overfitting is to change the model's size. If a model is not learning at all, one solution might be to try a larger model. If your model is learning, but there's a divergence between the training and testing data, it could mean that you should try a smaller model. One general rule to follow when selecting initial model hyperparameters is to find the smallest model possible that still learns. Other possible ways to avoid overfitting are regularization techniques we'll discuss in chapter 14, and the *Dropout* layer explained in chapter 15. Often the divergence of the training and testing data can take a long time to occur. The process of trying different model settings is called hyperparameter searching. Initially, you can very quickly (usually within minutes) try different settings (e.g., layer sizes) to see if the models are learning *something*. If they are, train the models fully — or at least significantly longer — and compare results to pick the best set of hyperparameters. Another possibility is to create a list of different hyperparameter sets and train the model in a loop using each of those sets at a time to pick the best set at the end. The reasoning here is that the fewer neurons you have, the less chance you have that the model is memorizing the data. Fewer neurons can mean it's easier for a neural network to generalize (actually learn the meaning of the data) compared to memorizing the data. With enough neurons, it's easier for a neural network to memorize the data. Remember that the neural network wants to decrease training loss and follows the path of least resistance to meet that objective. Our job as the programmer is to make the path to generalization the easiest path. This can often mean our job is actually to make the path to lowering loss for the model more challenging!

Supplementary Material: https://nnfs.io/ch11
Chapter code, further resources, and errata for this chapter.

Chapter 12 - Validation Data