ME4 Machine Learning - Tutorial 6

Lecture 6 introduced neural networks, one of the big areas of machine learning, and one which is gaining huge interest at the moment. In this tutorial you will use Python and Keras to implement neural networks, putting the theory into practice. We have not used Keras so far on the course; it is a library specifically for generating neural networks in Python. The interface is very straightforward and there are many similarities to scikit-learn which we have been using up to this point. You will get to design different networks with different activation functions and parameters, as was discussed in the lecture.

1 1D function fitting

Neural networks (and in fact all of supervised machine learning) can be considered to be taking a dataset and fitting a function to it which aims to capture the underlying behaviour. This function should both interpolate – fill in any gaps between the data points – but also perform some smoothing to avoid overfitting errors/noise in the data. The resulting function has the ability to make predictions about unknown outputs from a given set of inputs, whether classification (discrete outputs) or regression (continuous outputs). We will, first of all, look at function fitting with a neural network, which is a form of regression – calculating a continuous output for a given input.

In the lecture we illustrated fitting with a simple 1D piecewise linear function, showing how using a single layer of ReLUs can capture the behaviour of this function. We did this in a 'manual' way, identifying how several of these functions could be combined together, which worked for the simple example. We will now try to fit a similar function using Keras's tools.

Firstly, define the function and plot using the following code

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 200)

#set up a function of x
y = x.copy()
#^ note that we don't actually end up using y = x
# anywhere, but it does set up the array
y[x < 0.1] = 0 #y = 0 for x < 0.1
y[x >= 0.1] = 0.5*(x[x >= 0.1]-0.1) #add a gradient of +0.5 if x >= 0.5
```

```
y[x >= 0.6] = 0.25-0.25/0.2*(x[x >= 0.6]-0.6)
#^ gradient downwards from 0.6 to 0.8 (gradient
#calculated so that line goes from
#(0.6, 0.25) -> (0.8, 0), i.e. joins up with the other segments)
y[x > 0.8] = 0 #y = 0 for x > 0.8

fig, ax = plt.subplots()
plt.plot(x,y)
```

Now we will use Keras to fit the function.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

#set up a sequential neural network
model = Sequential()

#add a layer of 3 nodes of ReLUs, taking a single parametric input
model.add(Dense(units=3, activation='relu', input_dim=1))
#add a linear node at the end to combine the nodes together
model.add(Dense(units=1, activation='linear'))

#compile the model, trying to minimise mean squared
#error and using the Adam algorithm to fit this
model.compile(loss="mean_squared_error",
              optimizer='adam')

#fit the data provided previously, using 200 epochs and a batch size of 32
model.fit(x, y, epochs=200, batch_size=32)

#obtain a predicted set of values from the fitted function along its length
y_pred = model.predict(x)
```

Then do another plot which compares these predicted values to the true functions. What do you think of this? Hopefully you can see that the fitting has not worked well. However, we know that this should be able to fit the function in theory. In fact, for this very simple example, we can manually set the weights to generate the correct output (add this at the end, then run model.predict() again and plot the output):

```
model.layers[0].set_weights([np.array([[0.5, -(-0.5-0.25/0.2), 0.25/0.2]],),
        np.array([-0.5*0.1, (-0.5-0.25/0.2)*0.6, -(0.25/0.2)*0.8],)])
model.layers[1].set_weights([np.array([[1],[-1],[1]],), np.array([0],)])
```

Note that you should not worry about the specific syntax used here because there should be no practical examples where you will need to worry about setting the weights manually – the whole point

of machine learning is to learn from a training dataset! The values obtained are using the theory from Sect. 6.1.1 of the course notes; for example, the first ReLU function has weightings $w^1_{11} = 0.5$ and $w^1_{10} = -0.5 \times 0.1$ assigned in the first layer (and just 1 in the second, linear layer, which we therefore don't need to worry about) which means that the corresponding output of $f(w^1_{10} + w^1_{11}x)$ will be 0 up to 0.1 and increasing at a gradient of 0.5 after this point. Try setting the other weightings to zero to see just this function. The other functions are defined in a similar way, but there is a bit more complexity needed to ensure that the gradients are correct.

So we can clearly see that in theory, it should be possible for these activation functions to capture the behaviour fully. However, it is almost impossible to train these; we suffer a lot from getting stuck in local minima. To address this and make a trainable model, we need to allow more complexity into the model. Try 150 nodes in the layer instead of three used above, and use 2000 epochs rather than 200. Does this work better?

This highlights a particular issue with training neural networks – while, in theory, they can be highly expressive of very complex arbitrary functions, actually fitting the necessary coefficients is not straightforward. This is because the optimisation process must navigate various challenges, such as local minima, in order to converge, and having more nodes and variables available helps with the fitting.

You should also recognise the fairly simple Keras syntax for setting up and using the neural network – each layer can be added with the model.add() function. Various parameters can be set at the compile stage, including the specific loss function (what we want to minimise) and also the optimiser, which will usually be some variant of gradient descent. The fit and predict functions are similar to what we have done with scikit-learn previously, although the fitting routines need to have arguments of epochs and batch size[1] to define how the data is presented to the routine.

Try playing around with this. Can you use a different function instead of ReLU? Some alternatives are:

- softplus - the smoothed version of ReLU
- tanh
- sigmoid

From this you should start to appreciate some of the complexities of designing and training neural networks; while very powerful, they can be fiddly and typically require significant effort to achieve a satisfactory result. You should also note that the training time is often quite significant[2].

Next, try adding multiple layers. This will exploit the 'function of a function' concept discussed in the lecture. Set the first hidden layer to 10 nodes (from 150 before), then add two more layers of 10 nodes, all with ReLUs. You can do this by calling model.add() again, each time you need to add a layer, although note that you only need 'input_dim' for the first layer. With 2000 epochs, again you should see good fitting.

[1] As discussed in the lectures, an epoch is one complete run of data through the training algorithm, and the batch size dictates how many training points to perform a training step with at a time.

[2] You may find you get more speed by using a GPU for training – in Colab go to 'Runtime', then 'Change runtime type'

Then try a different function – tanh – throughout the network. What do you think of the fitting with this – is it better or worse? And what do we mean by 'better'/'worse'? Also think about how tanh performs when in this multi-layer approach, compared to how it performed before. This improved behaviour will give some explanation as to why deep learning, which relies on the use of hundreds or possibly thousands of layers, is becoming so powerful.

2 Two-parameter example

At this point you are advised to start a fresh notebook if you are using Colab. This is because the output from Keras can often slow the web browser down. You should use the 'blank' notebook provided for this, which contains some starting functions. If using an offline IDE you can just use [tools.py] as in tutorial 5.

We will now consider a 2D classification example. We will take a simple circular dataset, similar to the ring example from before, but this time with just a single central cluster from one category, and the second category surrounding this. This can be generated using the gen_simple_circular_distribution() function. Do this with n=200 points, and generate a scatter plot so you understand what the distribution looks like.

We will use a dense sequential neural network, so the previous imports will be sufficient. We will have a four layer model, with two input parameters, two ReLU layers with four units each and finally a softmax layer with two units. This design can be placed into Python/Keras with

```
model = Sequential()

model.add(Dense(units=4, activation='relu', input_dim=2))
model.add(Dense(units=4, activation='relu'))
model.add(Dense(units=2, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='sgd')
```

You may notice that the final layer has two outputs, whereas our training data is a single value, indicating the category. We have changed the loss term to categorical_crossentropy which should better match up with this form of output. The two outputs correspond to each category, e.g. ω_1 and ω_2. If the output on the first is 1 and the second 0, this indicates ω_1 is the correct one, but if the second is 1 and the first 0, then this indicates ω_2. It's often not too clear at first why we would want to do this, so think about the situation where we have more than two classes. If we have three, then we can have three outputs from our neural network and have these represent probabilities – what is the probability, given the input, that a particular class is the correct one, and clearly this needs three probability values. As discussed in the lecture, the use of the softmax function ensures that the outputs sum to one, and because they are allowed to be continuous variables, the values can be treated as probabilities as discussed.

To deal with the two outputs, you will need to convert the single y vector into the two-output form; fortunately Keras provides a useful tool to achieve this:

```
from tensorflow.keras.utils import *
...
y_binary = to_categorical(y)
```

The model can then be fitted to this data as before:

```
model.fit(X, y_binary, epochs=250, batch_size=32)
```

Add these two sections of code to your script/notebook.

Then define a suitable grid to sample the space from -10 to 10 in each dimension. Use this to sample the space (the model.predict() function works in the same way we used with scikit-learn) then output the resulting image. You will need to select one of the two outputs and reshape this as given in the example at the end of tutorial 2. Plot the training data on the same axes, using marker='x', s=2 in the scatter command. Hint: if debugging, it can take a long time if you have to run the full training routine each time – set the epochs to e.g. 5 while you make sure everything else works, then put it back to 250 again at the end.

As with the previous example, have a play around with this on your own. Try the following variations:

- Increase and decrease the number of training points
- Try the xor dataset
- Increase and decrease the number of epochs used in training
- Change the neural network itself - reduce and increase the number of layers, change the numbers of nodes. Try different functions as before; options include
  - relu
  - softmax
  - tanh
  - softplus
  - sigmoid

Having tested out these options, try to do some overfitting. Generate a training dataset with fewer points (e.g n = 30 in total) where the distributions are overlapped quite significantly, by increasing the scale parameter (scale = 2 works well). Then add three 6 unit layers after the input layer, to increase the complexity of the model. Finally you want to train your model on this data with a large number of epochs, e.g. 1000. All these factors are likely to increase overfitting:

- Having fewer training samples means we do not get a general picture of the underlying behaviour
- Increasing the overlap between the distributions means that we are more susceptible to the patterns associated with one particular dataset
- Complexity of the model is increased by adding additional layers with large numbers of units

- Using large numbers of epochs means that the model is strongly fitted to the training dataset.

Generate a validation dataset from the same function you used to generate the training dataset, although with more points to thoroughly assess the model. Show how the performance is good with the training dataset by poor with the validation dataset. How would you describe this large change in performance from one dataset to another?

3 Saving a model

Pick an example from before where you trained a neural network on a particular dataset. Save the resulting model using the model.save() function from Keras – do this in the .h5 format with:

```
model.save('model.h5')
```

The ability to save models like this will be important for the coursework.

As shown in tutorial 4, if using Colab, you can download the file with:

```
from google.colab import files
files.download(...)
```

You can upload files to Colab with files.upload() too.

4 Ring distribution (optional)

You can also try the more complex circular distribution gen_circular_distribution(). Generate 500 points. Use the original ANN with three layers of six ReLUs and one layer of two softmax units for the output. Use as few epochs as you can get away with while still generating a good result. How do you know that you have got a good model?