Question 1: Modifying the Neural Network Code

(a) Modify the code so that it writes the error for the training set and the error for the test set after each epoch into a file. Let the network run for at least 150 epochs (if you let it run overnight, you can as well perform more epochs) and plot training and test error as functions of epoch number like in Figure 9.9 (right panel) of Chapter 9 that I sent you. If your plot looks completely different from Figure 9.9, please let me know and we can check what went wrong.

OK, so it seems I caused some severe misunderstanding here, and it is not clear in the book chapter, either. The training error shown in Fig. 9.9 is measured during the epoch, as exemplars are “shown” to the network. This means that this error always “lags behind” the test error, which is measured after each epoch. Moreover, the training error is measured while dropout is used i.e., when half of the neurons are turned off. This is not the case for the test error. Both of these differences cause the training error to look greater than it actually is. In order to have a fair comparison, we must measure both training and test error after each epoch, without dropout. So if you did that, that is great, but if not, that is no problem, you will not lose any points.

In order to show you the actual training error and how it compares to the “on the fly” training error that we measured before, I created sample solutions for this question that show both types of error plus the test error, measured for the first 150 epochs. Here is the sample solutions for part (a):
As you can see, the actual training error (after each epoch) does get very close to zero (0.02%), while the error during the epoch (and including dropout) starts out greater than the test error but undershoots it at some point and reaches a minimum of approximately 0.67%. The test error once again gets close to 1%; to be exact, 1.07%.

(b) See what happens if you use fewer neurons in the two hidden layers. You can choose any numbers, but make at least one layer significantly smaller than in the original network. Again, plot the training and test errors across epochs. How do the results differ from the initial ones?

Here I am showing an example using 4 layers of 784, 250, 100, and 10 neurons (see next slide):
Compared to the original network (a), both training and testing performance are lower. The network is simply able to learn the underlying function very closely. The training error after each epoch only goes down to 0.51%. Consequently, the network is doing worse for the test data as well, never dropping below 1.68%. The training error during the epoch never catches up with the test error, demonstrating that dropout particularly reduces network performance when there are not enough neurons to begin with.

(c) Restore the original number of neurons, and now change the dropout rate. Originally, 25% of input units (see line 164 in the code) and 50% of hidden-layer neurons (see line 172) are randomly chosen to drop out, i.e., give no output. Note that when we run the network in production (non-training) mode without dropout, we need to increase the output of neurons accordingly to keep the overall activation at the same level (lines 180 and 182). Choose a different dropout rate for the hidden-layer and/or input layer units to whichever value you like, and run the network again for at least 150 epochs. Plot the results and compare them to the ones you got in (a).

Here I chose a lower dropout rate of 25% as opposed to the original (and typical) rate of 50%. This is the result (see next page):
The results are actually rather similar to the narrower network, with a final after-epoch training error of 0.51% and a minimum test error of 1.55%. Again, the during-epoch training error is never able to drop below the test error. This is a bit unexpected since the 25% dropout should affect the during-epoch training error less than the 25% dropout does. It thus seems that 25% dropout does simply not achieve the benefits of the usual 50% dropout, but at the same time it still reduces the information available for performing classification.

(d) Add noise to the data by randomly choosing a certain percentage of pixels in both the training and test images and flipping their intensity so that intensity $i$ will become $(255 - i)$. For example, if a pixel has the original value 10, then after this transformation it has the value 245. The easiest way to do this is to modify the readData() function. You can choose any percentage you like, and any of the three networks that you created above. Again, run it for at least 150 epochs and compare the results to the original network (whichever one you chose).

Here, I used a probability of 20% for each pixel to be flipped. This is a huge amount of noise; choosing 50% would completely remove any information from the images. Consequently, the resulting errors are rather large:
Interestingly, the learning of the training data works very well, with an after-epoch training error that actually reaches 0%. It seems that dropout at the input layer is able to create low training error for noisy input data, even when the error during dropout converges slowly and only reaches 4.91% after 150 epochs. The test error also remains large (7.57%), showing that the unpredictable noise in the test images impairs the generalizability of the learned information.

Question 2: The Soccer Network

There are many ways of accomplishing this task. Here is one possible solution;

First you have to consider how to format the input to the network:

- The skill level represents a scale, and therefore we only need one neuron to represent each value. Just divide the level by 10 to get input values ranging from 0 to 1. If you only consider the mean level or the mean and its standard deviation, that is also OK. But if you consider each individual player’s level, you need a total of 40 neurons.
• The number of matches that each team has played during the last two weeks can be represented by two neurons, one for each team. As there are never more than seven matches in that period of time, just take each number and divide it by seven.

• The statistics of former matches between the same two teams within the past 10 years can be represented by three neurons representing the proportions of (1) wins for Team A, (2) ties, and (3) wins for Team B.

• The continent that each team comes from does not represent a scale, and therefore we have to use six neurons to represent it. Then we could use the following input patterns: North America: 100000, South America: 010000, Europe: 001000, Africa: 000100, Asia: 000010, and Australia: 000001).

• To represent where the match takes place (Team A’s stadium, Team B’s stadium, or neutral place), you could either use three neurons in the same way as for the continent, or you could use one neuron and consider the following scale (from best to worst for Team A): Team A’s stadium, neutral place, Team B’s stadium.

• The phase of the soccer season (early season vs. late season) requires one neuron with value 0 or 1, respectively.

Then the input vectors consist of \(40 + 2 + 3 + 6 + 3 + 1 = 55\) elements.

Regarding the output (predicted number of goals scored by each team), these values form scales and should be represented by one neuron each. Let us say that for high-level matches there will never be more than seven goals per team (remember Germany vs. Brazil at the 2014 World Cup!). Then we can define that we multiply the output by seven and round to the closest integer in order to get the goal prediction for each team.

So your network needs 55 input neurons, maybe 20 hidden-layer neurons (just a guess), and 2 output neurons. As usual, each hidden neuron and each output neuron also receive an extra “offset” input.

In the next step you have to access soccer databases to obtain exemplars for training and testing. For each match in the database, you collect all required data to form the 55-element input vector and the 2-element desired output vector (the final score). These vector pairs serve as your exemplars. Make sure that you cover the input space as best as possible. For example, you should cover all continents, all phases of the season, and all types of competitions, leagues, and skill levels. Since our network has \(56 \cdot 20 + 21 \cdot 2 = 1162\) weights, we should ideally have at least ten times as many training exemplars, and just as many test exemplars, so it would be best to collect data from at least 23,000 matches.

We now initialize the network with random weights and then pick the training exemplars in random order. The input part of each exemplar is fed into the input neurons, and the network’s output is compared to the output part of the exemplar. Then the weights are
adjusted according to the backpropagation rule. We repeat this procedure until the network error, i.e., the mean squared error between actual and desired outputs, falls below a threshold that we determine empirically. Then we freeze the weights and measure how well the network predicts the outcomes of the test exemplars.

If the performance is high, we have a way to make lots of money!

(In reality, this would not work well. There are too many factors influencing soccer results, including coincidences that cannot be predicted.)

**Question 3: Linear Neurons**

The following is a network of linear neurons - that is, neurons whose output is identical to their net input, $x \cdot w$ (in other words, their output function that translates net input into output is simply the identity function). These neurons do not receive any “dummy” inputs (biases or offsets). The numbers in the circles indicate the output of a neuron, and the labels of connections indicate the value of the corresponding weight.

(a) Just as a warm-up exercise, compute the output of the hidden-layer and the output-layer neurons for the given input $(2, 1)$.

Let us call the outputs of the hidden-layer units $h_1$, $h_2$, and $h_3$ (from left to right). Then we have:
\[ h_1 = (-3) \cdot 2 + 2 \cdot 1 = -4 \]
\[ h_2 = 2 \cdot 2 + 0 \cdot 1 = 4 \]
\[ h_3 = 3 \cdot 2 + 1 \cdot 1 = 7 \]

Then for the output layer neurons o1 and o2 we get:

\[ o_1 = (-4) \cdot (-4) + 3 \cdot 4 + 1 \cdot 7 = 35 \]
\[ o_2 = (-2) \cdot (-4) + 4 \cdot 4 + (-1) \cdot 7 = 17 \]

**(b) Only mandatory for CS670:** Show that a network of linear neurons, such as this one, always computes a linear function, regardless of its number of layers and neurons.

**Hint:** A function \( y = f(x) \) is linear if and only if it can be expressed as \( y = Ax \) for some matrix \( A \).

For a linear neuron, its output \( y \) is simply the scalar product of the weight vector and the input vector: \( y = wx \). This means that computing the output of all neurons in a given layer \( n \) can be computed as the outputs from the layer \( (n - 1) \) below it, multiplied by a matrix whose rows are the weights of the neurons in layer \( n \). For example, we can compute the outputs of our hidden-layer units as follows:

\[
\begin{bmatrix}
-3 \\
2 \\
3
\end{bmatrix}
\cdot
\begin{bmatrix}
2 \\
1
\end{bmatrix} =
\begin{bmatrix}
-4 \\
4 \\
7
\end{bmatrix}
\]

The resulting output vector becomes the input vector to the next layer \( (n + 1) \), so another matrix multiplication is performed, and so one, until we get the output of the last layer – the output layer – which represents the output of the network.

Therefore, if we have an input layer plus \( N \) computational layers with weight matrices \( W_1, W_2, \ldots, W_n \), the network’s output \( o \) for a given input \( x \) can be computed as:

\[ o(x) = W_N \cdot W_{N-1} \cdot \ldots \cdot W_1 \cdot x \]

Now the product of several matrices is simply a matrix itself, and therefore we can refer to the product of the weight matrices as \( W_{\text{combined}} \) and get:

\[ o(x) = W_{\text{combined}} \cdot x \]

Clearly, by definition, the output of the network is a linear function of \( x \). This is the reason why we require non-linear activation functions in neural networks. If we use linear ones, the entire network computation is limited to linear functions, and having multiple layers does not give us any benefit.

**(c) Only mandatory for CS670:** Given that our three-layer network computes a linear function, we suddenly notice that our network is wastefully large. It must be possible...
to compute exactly the same function with a two-layer network. Draw such a network, including all of its weights, that only consists of an input layer and an output layer and computes the same function as the network shown above. **Hint:** In the network above, determine how the output of each output-layer neuron depends on the two network inputs, and then you should be able to find the correct weights for the two-layer network. There is a more elegant way of deriving the solution that is related to (b), but any correct solution gets full points, regardless of your approach.

As we have seen in (b), we simply need to multiply the weight matrices of all layers to obtain a single, combined matrix. Let us do this for the given network:

\[
W_{\text{combined}} = \begin{bmatrix}
-4 & 3 & 1 \\
-2 & 4 & 1
\end{bmatrix} \cdot \begin{bmatrix}
-3 & 2 \\
2 & 0 \\
3 & 1
\end{bmatrix} = \begin{bmatrix}
21 & -7 \\
11 & -5
\end{bmatrix}
\]

Let us put this matrix to the test! For the input \((2, 1)\) in the example we get:

\[
o = \begin{bmatrix}
21 \\
11
\end{bmatrix} \cdot \begin{bmatrix}
2 \\
1
\end{bmatrix} = \begin{bmatrix}
35 \\
17
\end{bmatrix}
\]

This result looks great and matches our answer in (a). Now let us build a network with only input and output layers that uses this weight matrix as the output layer weights:

![Network Diagram](image)

This much simpler network computes exactly the same function as the larger one above.
Question 4 (Bonus): From Gradient Descent to Backpropagation

Explain in your own words how the concepts of gradient descent and backpropagation are related to each other.

Gradient descent is a technique for efficiently finding the input $x_0$ to a function $f$ that (ideally) produces the absolute minimum of $f$. It starts with an estimate of $x_0$ and iteratively refines it by moving against the function’s gradient. This is exactly what backpropagation does. In backpropagation, the function to be minimized is the mean squared error of the network, and the inputs to this function (i.e., the variables that the error depends on) are the network weights. In other words, we have to find those weights for which the network error is minimal, and we do this using gradient descent in a typically very high-dimensional function.