# Chapter 1 Basics of deep learning and neural networks

In this chapter, you'll become familiar with the fundamental concepts and terminology used in deep learning, and understand why deep learning techniques are so powerful today. You'll build simple neural networks and generate predictions with them.

## Introduction to deep learning Video 5m

Comparing neural network models to classical regression models Quiz

## Forward propagation Video 4m

Coding the forward propagation algorithm Code

## Activation functions Video 3m

The Rectified Linear Activation Function Code

Applying the network to many observations/rows of data Code

## Deeper networks Video 5m

Forward propagation in a deeper network Quiz

Multi-layer neural networks Code

Representations are learned Quiz

Levels of representation Quiz

# Chapter 2 Optimizing a neural network with backward propagation

Learn how to optimize the predictions generated by your neural networks. You'll use a method called backward propagation, which is one of the most important techniques in deep learning. Understanding how it works will give you a strong foundation to build on in the second half of the course.

## The need for optimization Video 4m

Calculating model errors Quiz

Understanding how weights change model accuracy Quiz

Coding how weight changes affect accuracy Code

Scaling up to multiple data points Code

## Gradient descent Video 5m

Calculating slopes Code

Improving model weights Code

Making multiple updates to weights Code

## Backpropagation Video 4m

The relationship between forward and backward propagation Quiz

Thinking about backward propagation Quiz

## Backpropagation in practice Video 4m

A round of backpropagation Quiz

# Chapter 3 Building deep learning models with keras

In this chapter, you'll use the Keras library to build deep learning models for both regression and classification. You'll learn about the Specify-Compile-Fit workflow that you can use to make predictions, and by the end of the chapter, you'll have all the tools necessary to build deep neural networks.

## Creating a Keras model Video 3m20

Understanding your data Code

Specifying a model Code

## Compiling and fitting a model Video 2m30

Compiling the model Code

Fitting the model Code

## Classification models Video 3m30

Understanding your classification data Code

Last steps in classification models Code

## Using models Video 2m

Making predictions Code

# Chapter 4 Fine-tuning keras models

Learn how to optimize your deep learning models in Keras. Start by learning how to validate your models, then understand the concept of model capacity, and finally, experiment with wider and deeper networks.

## Understanding model optimization Video 3m30

Diagnosing optimization problems Quiz

Changing optimization parameters Code

## Model validation Video 4m

Evaluating model accuracy on validation dataset Code

Early stopping: Optimizing the optimization Code

Experimenting with wider networks Code

Adding layers to a network Code

## Thinking about model capacity Video 4m

Experimenting with model structures Quiz

## Stepping up to images Video 1m

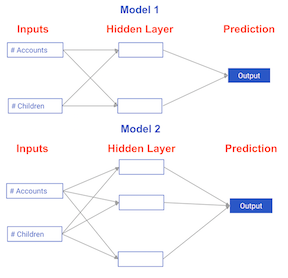
Building your own digit recognition model Code

## Final thoughts Video 2m

# Tutorial Chapter 1

# Quiz 1.1 Comparing neural network models to classical regression models

Which of the models in the diagrams has greater ability to account for interactions?



##### Answer the question

#### Possible Answers

Select one answer

* Model 1

**PRESS1**

* **Model 2**

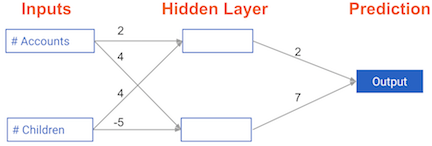
**PRESS2**

* They are both the same

**PRESS3**

# Code 1.2 Coding the forward propagation algorithm

In this exercise, you'll write code to do forward propagation (prediction) for your first neural network:



Each data point is a customer. The first input is how many accounts they have, and the second input is how many children they have. The model will predict how many transactions the user makes in the next year. You will use this data throughout the first 2 chapters of this course.

The input data has been pre-loaded as input\_data, and the weights are available in a dictionary called weights. The array of weights for the first node in the hidden layer are in weights['node\_0'], and the array of weights for the second node in the hidden layer are in weights['node\_1'].

The weights feeding into the output node are available in weights['output'].

NumPy will be pre-imported for you as np in all exercises.

## Instructions

* Calculate the value in node 0 by multiplying input\_data by its weights weights['node\_0']and computing their sum. This is the 1st node in the hidden layer.
* Calculate the value in node 1 using input\_data and weights['node\_1']. This is the 2nd node in the hidden layer.
* Put the hidden layer values into an array. This has been done for you.
* Generate the prediction by multiplying hidden\_layer\_outputs by weights['output'] and computing their sum.
* Hit 'Submit Answer' to print the output!

Sample code

import numpy as np

# Assign input data

input\_data = np.array([3, 5])

# Assign weights

weights = {'node\_0': np.array([\_\_\_\_, \_\_\_\_]),

'node\_1': np.array([\_\_\_\_, \_\_\_\_]),

'output': np.array([\_\_\_\_, \_\_\_\_])}

# Calculate node 0 value: node\_0\_value

node\_0\_value = (\_\_\_\_ \* \_\_\_\_).\_\_\_\_

# Calculate node 1 value: node\_1\_value

node\_1\_value = \_\_\_\_

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_value, node\_1\_value])

# Calculate output: output

output = \_\_\_\_

# Print output

print(output)

Answer code

import numpy as np

# Assign input data

input\_data = np.array([3, 5])

# Assign weights

weights = {'node\_0': np.array([2, 4]),

'node\_1': np.array([4, -5]),

'output': np.array([2, 7])}

# Calculate node 0 value: node\_0\_value

node\_0\_value = (input\_data \* weights['node\_0']).sum()

# Calculate node 1 value: node\_1\_value

node\_1\_value = (input\_data \* weights['node\_1']).sum()

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_value, node\_1\_value])

# Calculate output: output

output = (hidden\_layer\_outputs \* weights['output']).sum()

# Print output

print(output)

# Code 1.3 The Rectified Linear Activation Function

As Dan explained to you in the video, an "activation function" is a function applied at each node. It converts the node's input into some output.

The rectified linear activation function (called ReLU) has been shown to lead to very high-performance networks. This function takes a single number as an input, returning 0 if the input is negative, and the input if the input is positive.

Here are some examples:  
**relu(3) = 3**  
**relu(-3) = 0**

## Instructions

* Fill in the definition of the relu() function:
  + Use the max() function to calculate the value for the output of relu().
* Apply the relu() function to node\_0\_input to calculate node\_0\_output.
* Apply the relu() function to node\_1\_input to calculate node\_1\_output.

Sample code

def relu(input):

'''Define your relu activation function here'''

# Calculate the value for the output of the relu function: output

output = max(\_\_\_\_, \_\_\_\_)

# Return the value just calculated

return(output)

# Calculate node 0 value: node\_0\_output

node\_0\_input = (input\_data \* weights['node\_0']).sum()

node\_0\_output = \_\_\_\_

# Calculate node 1 value: node\_1\_output

node\_1\_input = (input\_data \* weights['node\_1']).sum()

node\_1\_output = \_\_\_\_

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output (do not apply relu)

model\_output = (hidden\_layer\_outputs \* weights['output']).sum()

# Print model output

print(model\_output)

Answer code

import numpy as np

# Assign input data

input\_data = np.array([3, 5])

# Assign weights

weights = {'node\_0': np.array([2, 4]),

'node\_1': np.array([4, -5]),

'output': np.array([2, 7])}

def relu(input):

'''Define your relu activation function here'''

# Calculate the value for the output of the relu function: output

output = max(input, 0)

# Return the value just calculated

return(output)

# Calculate node 0 value: node\_0\_output

node\_0\_input = (input\_data \* weights['node\_0']).sum()

node\_0\_output = relu(node\_0\_input)

# Calculate node 1 value: node\_1\_output

node\_1\_input = (input\_data \* weights['node\_1']).sum()

node\_1\_output = relu(node\_1\_input)

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output (do not apply relu)

model\_output = (hidden\_layer\_outputs \* weights['output']).sum()

# Print model output

print(model\_output)

# Code 1.4 Applying the network to many observations/rows of data

You'll now define a function called predict\_with\_network() which will generate predictions for multiple data observations, which are pre-loaded as input\_data. As before, weights are also pre-loaded. In addition, the relu() function you defined in the previous exercise has been pre-loaded.

## Instructions

* Define a function called predict\_with\_network() that accepts two arguments - input\_data\_row and weights - and returns a prediction from the network as the output.
* Calculate the input and output values for each node, storing them as: node\_0\_input, node\_0\_output, node\_1\_input, and node\_1\_output.
  + To calculate the input value of a node, multiply the relevant arrays together and compute their sum.
  + To calculate the output value of a node, apply the relu() function to the input value of the node.
* Calculate the model output by calculating input\_to\_final\_layer and model\_output in the same way you calculated the input and output values for the nodes.
* Use a for loop to iterate over input\_data:
  + Use your predict\_with\_network() to generate predictions for each row of the input\_data - input\_data\_row. Append each prediction to results.

Sample code

# Define predict\_with\_network()

def predict\_with\_network(input\_data\_row, weights):

# Calculate node 0 value

node\_0\_input = \_\_\_\_

node\_0\_output = \_\_\_\_

# Calculate node 1 value

node\_1\_input = \_\_\_\_

node\_1\_output = \_\_\_\_

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output

input\_to\_final\_layer = \_\_\_\_

model\_output = \_\_\_\_

# Return model output

return(model\_output)

# Create empty list to store prediction results

results = []

for input\_data\_row in input\_data:

# Append prediction to results

results.append(\_\_\_\_)

# Print results

print(results)

Answer code

import numpy as np

# Assign input data

input\_data = [np.array([3, 5]), np.array([ 1, -1]),

np.array([0, 0]), np.array([8, 4])]

# Assign weights

weights = {'node\_0': np.array([2, 4]),

'node\_1': np.array([4, -5]),

'output': np.array([2, 7])}

# Define predict\_with\_network()

def predict\_with\_network(input\_data\_row, weights):

# Calculate node 0 value

node\_0\_input = (input\_data\_row \* weights['node\_0']).sum()

node\_0\_output = relu(node\_0\_input)

# Calculate node 1 value

node\_1\_input = (input\_data\_row \* weights['node\_1']).sum()

node\_1\_output = relu(node\_1\_input)

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output

input\_to\_final\_layer = (hidden\_layer\_outputs \* weights['output']).sum()

model\_output = relu(input\_to\_final\_layer)

# Return model output

return(model\_output)

# Create empty list to store prediction results

results = []

for input\_data\_row in input\_data:

# Append prediction to results

results.append(predict\_with\_network(input\_data\_row,weights))

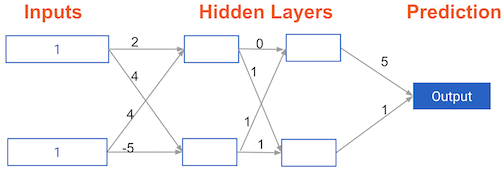
# Print results

print(results)

# Quiz 1.5 Forward propagation in a deeper network

You now have a model with 2 hidden layers. The values for an input data point are shown inside the input nodes. The weights are shown on the edges/lines. What prediction would this model make on this data point?

Assume the activation function at each node is the *identity function*. That is, each node's output will be the same as its input. So the value of the bottom node in the first hidden layer is -1, and not 0, as it would be if the ReLU activation function was used.



**Answer the question**

**Possible Answers**

Select one answer

* **0**

**PRESS1**

* 7

**PRESS2**

* 9

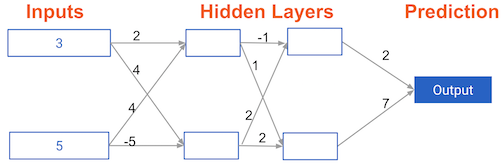
**PRESS3**

# Code 1.6 Multi-layer neural networks

In this exercise, you'll write code to do forward propagation for a neural network with 2 hidden layers. Each hidden layer has two nodes. The input data has been preloaded as input\_data. The nodes in the first hidden layer are called node\_0\_0 and node\_0\_1. Their weights are pre-loaded as weights['node\_0\_0'] and weights['node\_0\_1'] respectively.

The nodes in the second hidden layer are called node\_1\_0 and node\_1\_1. Their weights are pre-loaded as weights['node\_1\_0'] and weights['node\_1\_1'] respectively.

We then create a model output from the hidden nodes using weights pre-loaded as weights['output'].



## Instructions

* Calculate node\_0\_0\_input using its weights weights['node\_0\_0'] and the given input\_data. Then apply the relu() function to get node\_0\_0\_output.
* Do the same as above for node\_0\_1\_input to get node\_0\_1\_output.
* Calculate node\_1\_0\_input using its weights weights['node\_1\_0'] and the outputs from the first hidden layer - hidden\_0\_outputs. Then apply the relu() function to get node\_1\_0\_output.
* Do the same as above for node\_1\_1\_input to get node\_1\_1\_output.
* Calculate model\_output using its weights weights['output'] and the outputs from the second hidden layer hidden\_1\_outputs array. Do not apply the relu() function to this output.

Sample code

def predict\_with\_network(input\_data):

# Calculate node 0 in the first hidden layer

node\_0\_0\_input = (\_\_\_\_ \* \_\_\_\_).sum()

node\_0\_0\_output = relu(\_\_\_\_)

# Calculate node 1 in the first hidden layer

node\_0\_1\_input = \_\_\_\_

node\_0\_1\_output = \_\_\_\_

# Put node values into array: hidden\_0\_outputs

hidden\_0\_outputs = np.array([node\_0\_0\_output, node\_0\_1\_output])

# Calculate node 0 in the second hidden layer

node\_1\_0\_input = \_\_\_\_

node\_1\_0\_output = \_\_\_\_

# Calculate node 1 in the second hidden layer

node\_1\_1\_input = \_\_\_\_

node\_1\_1\_output = \_\_\_\_

# Put node values into array: hidden\_1\_outputs

hidden\_1\_outputs = np.array([node\_1\_0\_output, node\_1\_1\_output])

# Calculate model output: model\_output

model\_output = \_\_\_\_

# Return model\_output

return(model\_output)

output = predict\_with\_network(input\_data)

print(output)

Answer code

import numpy as np

# Assign input data

input\_data = np.array([3, 5])

# Assign weights

weights = {'node\_0\_0': array([2, 4]),

'node\_0\_1': array([ 4, -5]),

'node\_1\_0': array([-1, 2]),

'node\_1\_1': array([1, 2]),

'output': array([2, 7])}

def predict\_with\_network(input\_data):

# Calculate node 0 in the first hidden layer

node\_0\_0\_input = (input\_data \* weights['node\_0\_0']).sum()

node\_0\_0\_output = relu(node\_0\_0\_input)

# Calculate node 1 in the first hidden layer

node\_0\_1\_input = (input\_data \* weights['node\_0\_1']).sum()

node\_0\_1\_output = relu(node\_0\_1\_input)

# Put node values into array: hidden\_0\_outputs

hidden\_0\_outputs = np.array([node\_0\_0\_output, node\_0\_1\_output])

# Calculate node 0 in the second hidden layer

node\_1\_0\_input = (hidden\_0\_outputs \* weights['node\_1\_0']).sum()

node\_1\_0\_output = relu(node\_1\_0\_input)

# Calculate node 1 in the second hidden layer

node\_1\_1\_input = (hidden\_0\_outputs \* weights['node\_1\_1']).sum()

node\_1\_1\_output = relu(node\_1\_1\_input)

# Put node values into array: hidden\_1\_outputs

hidden\_1\_outputs = np.array([node\_1\_0\_output, node\_1\_1\_output])

# Calculate model output: model\_output

model\_output = (hidden\_1\_outputs \* weights['output']).sum()

# Return model\_output

return(model\_output)

output = predict\_with\_network(input\_data)

print(output)

# Quiz 1.7 Representations are learned

How are the weights that determine the features/interactions in Neural Networks created?

**Answer the question**

**Possible Answers**

Select one answer

* A user chooses them when creating the model.

**PRESS1**

* **The model training process sets them to optimize predictive accuracy.**

**PRESS2**

* The weights are random numbers.

**PRESS3**

# Quiz 1.8 Levels of representation

Which layers of a model capture more complex or "higher level" interactions?

##### Answer the question

#### Possible Answers

Select one answer

* The first layers capture the most complex interactions.

**PRESS1**

* **The last layers capture the most complex interactions.**

**PRESS2**

* All layers capture interactions of similar complexity.

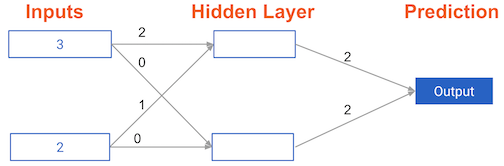
**PRESS3**

# Tutorial Chapter 2

# Quiz 2.1 Calculating model errors

For the exercises in this chapter, you'll continue working with the network to predict transactions for a bank.

What is the error (predicted - actual) for the following network using the ReLU activation function when the input data is [3, 2] and the actual value of the target (what you are trying to predict) is 5? It may be helpful to get out a pen and piece of paper to calculate these values.



**Answer the question**

**Possible Answers**

Select one answer

* 5

**PRESS1**

* 6

**PRESS2**

* **11**

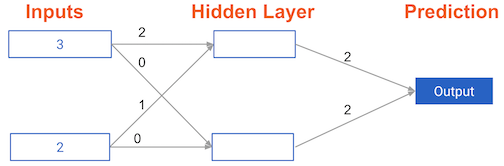
**PRESS3**

* 16

**PRESS4**

# Quiz 2.2 Understanding how weights change model accuracy

Imagine you have to make a prediction for a single data point. The actual value of the target is 7. The weight going from node\_0 to the output is 2, as shown below. If you increased it slightly, changing it to 2.01, would the predictions become more accurate, less accurate, or stay the same?



##### Answer the question

#### Possible Answers

Select one answer

* More accurate.

**PRESS1**

* **Less accurate.**

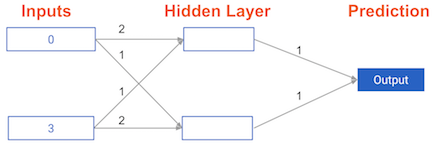
**PRESS2**

* Stay the same.

**PRESS3**

# Code 2.3 Coding how weight changes affect accuracy

Now you'll get to change weights in a real network and see how they affect model accuracy!

Have a look at the following neural network:

Its weights have been pre-loaded as weights\_0. Your task in this exercise is to update a **single** weight in weights\_0 to create weights\_1, which gives a perfect prediction (in which the predicted value is equal to target\_actual: 3).

Use a pen and paper if necessary to experiment with different combinations. You'll use the predict\_with\_network() function, which takes an array of data as the first argument, and weights as the second argument.

## Instructions

* Create a dictionary of weights called weights\_1 where you have changed **1** weight from weights\_0 (You only need to make 1 edit to weights\_0 to generate the perfect prediction).
* Obtain predictions with the new weights using the predict\_with\_network() function with input\_data and weights\_1.
* Calculate the error for the new weights by subtracting target\_actual from model\_output\_1.
* Hit 'Submit Answer' to see how the errors compare!

Sample code

import numpy as np

# The data point you will make a prediction for

input\_data = np.array([0, 3])

# Sample weights

weights\_0 = {'node\_0': [2, 1],

'node\_1': [1, 2],

'output': [1, 1]

}

# The actual target value, used to calculate the error

target\_actual = 3

# Make prediction using original weights

model\_output\_0 = predict\_with\_network(input\_data, weights\_0)

# Calculate error: error\_0

error\_0 = model\_output\_0 - target\_actual

# Create weights that cause the network to make perfect prediction (3): weights\_1

weights\_1 = {'node\_0': [\_\_\_\_, \_\_\_\_],

'node\_1': [\_\_\_\_, \_\_\_\_],

'output': [\_\_\_\_, \_\_\_\_]

}

# Make prediction using new weights: model\_output\_1

model\_output\_1 = \_\_\_\_

# Calculate error: error\_1

error\_1 = \_\_\_\_ - \_\_\_\_

# Print error\_0 and error\_1

print(error\_0)

print(error\_1)

Answer code

import numpy as np

# The data point you will make a prediction for

input\_data = np.array([0, 3])

# Sample weights

weights\_0 = {'node\_0': [2, 1],

'node\_1': [1, 2],

'output': [1, 1]

}

# The actual target value, used to calculate the error

target\_actual = 3

# Define predict\_with\_network()

def predict\_with\_network(input\_data\_row, weights):

# Calculate node 0 value

node\_0\_input = (input\_data\_row \* weights['node\_0']).sum()

node\_0\_output = relu(node\_0\_input)

# Calculate node 1 value

node\_1\_input = (input\_data\_row \* weights['node\_1']).sum()

node\_1\_output = relu(node\_1\_input)

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output

input\_to\_final\_layer = (hidden\_layer\_outputs \* weights['output']).sum()

model\_output = relu(input\_to\_final\_layer)

# Return model output

return(model\_output)

# Make prediction using original weights

model\_output\_0 = predict\_with\_network(input\_data, weights\_0)

# Calculate error: error\_0

error\_0 = model\_output\_0 - target\_actual

# Create weights that cause the network to make perfect prediction (3): weights\_1

weights\_1 = {'node\_0': [2, 1],

'node\_1': [1, 0],

'output': [1, 1]

}

# Make prediction using new weights: model\_output\_1

model\_output\_1 = predict\_with\_network(input\_data, weights\_1)

# Calculate error: error\_1

error\_1 = model\_output\_1 - target\_actual

# Print error\_0 and error\_1

print(error\_0)

print(error\_1)

# Code 2.4 Scaling up to multiple data points

You've seen how different weights will have different accuracies on a single prediction. But usually, you'll want to measure model accuracy on many points. You'll now write code to compare model accuracies for two different sets of weights, which have been stored as weights\_0 and weights\_1.

input\_data is a list of arrays. Each item in that list contains the data to make a single prediction.target\_actuals is a list of numbers. Each item in that list is the actual value we are trying to predict.

In this exercise, you'll use the mean\_squared\_error() function from sklearn.metrics. It takes the true values and the predicted values as arguments.

You'll also use the preloaded predict\_with\_network() function, which takes an array of data as the first argument, and weights as the second argument.

**Instructions**

* Import mean\_squared\_error from sklearn.metrics.
* Using a for loop to iterate over each row of input\_data:
  + Make predictions for each row with weights\_0 using the predict\_with\_network() function and append it to model\_output\_0.
  + Do the same for weights\_1, appending the predictions to model\_output\_1.
* Calculate the mean squared error of model\_output\_0 and then model\_output\_1 using the mean\_squared\_error() function. The first argument should be the actual values (target\_actuals), and the second argument should be the predicted values (model\_output\_0 or model\_output\_1).

Sample code

from sklearn.metrics import mean\_squared\_error

# Create model\_output\_0

model\_output\_0 = []

# Create model\_output\_1

model\_output\_1 = []

# Loop over input\_data

for row in input\_data:

# Append prediction to model\_output\_0

model\_output\_0.append(\_\_\_\_)

# Append prediction to model\_output\_1

model\_output\_1.append(\_\_\_\_)

# Calculate the mean squared error for model\_output\_0: mse\_0

mse\_0 = \_\_\_\_

# Calculate the mean squared error for model\_output\_1: mse\_1

mse\_1 = \_\_\_\_

# Print mse\_0 and mse\_1

print("Mean squared error with weights\_0: %f" %mse\_0)

print("Mean squared error with weights\_1: %f" %mse\_1)

Answer code

import numpy as np

from sklearn.metrics import mean\_squared\_error

# Assign input data and actual target values

input\_data = [np.array([0, 3]), np.array([ 1, 2]),

np.array([-1, -2]), np.array([4, 0])]

target\_actuals = [1, 3, 5, 7]

# Sample weights

weights\_0 = {'node\_0': [2, 1],

'node\_1': [1, 2],

'output': [1, 1]

}

weights\_1 = {'node\_0': [2, 1],

'node\_1': [1., 1.5],

'output': [1., 1.5]

}

# Define predict\_with\_network()

def predict\_with\_network(input\_data\_row, weights):

# Calculate node 0 value

node\_0\_input = (input\_data\_row \* weights['node\_0']).sum()

node\_0\_output = relu(node\_0\_input)

# Calculate node 1 value

node\_1\_input = (input\_data\_row \* weights['node\_1']).sum()

node\_1\_output = relu(node\_1\_input)

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output

input\_to\_final\_layer = (hidden\_layer\_outputs \* weights['output']).sum()

model\_output = relu(input\_to\_final\_layer)

# Return model output

return(model\_output)

# Create model\_output\_0

model\_output\_0 = []

# Create model\_output\_1

model\_output\_1 = []

# Loop over input\_data

for row in input\_data:

# Append prediction to model\_output\_0

model\_output\_0.append(predict\_with\_network(row, weights\_0))

# Append prediction to model\_output\_1

model\_output\_1.append(predict\_with\_network(row, weights\_1))

# Calculate the mean squared error for model\_output\_0: mse\_0

mse\_0 = mean\_squared\_error(target\_actuals, model\_output\_0)

# Calculate the mean squared error for model\_output\_1: mse\_1

mse\_1 = mean\_squared\_error(target\_actuals, model\_output\_1)

# Print mse\_0 and mse\_1

print("Mean squared error with weights\_0: %f" %mse\_0)

print("Mean squared error with weights\_1: %f" %mse\_1)

# Code 2.5 Calculating slopes

You're now going to practice calculating slopes. When plotting the mean-squared error loss function against predictions, the slope is 2 \* x \* (xb-y), or 2 \* input\_data \* error. Note that x and b may have multiple numbers (x is a vector for each data point, and b is a vector). In this case, the output will also be a vector, which is exactly what you want.

You're ready to write the code to calculate this slope while using a single data point. You'll use pre-defined weights called weights as well as data for a single point called input\_data. The actual value of the target you want to predict is stored in target.

**Instructions**

* Calculate the predictions, preds, by multiplying weights by the input\_data and computing their sum.
* Calculate the error, which is preds minus target. Notice that this error corresponds to xb-yin the gradient expression.
* Calculate the slope of the loss function with respect to the prediction. To do this, you need to take the product of input\_data and error and multiply that by 2.

Sample code

# Calculate the predictions: preds

preds = \_\_\_\_

# Calculate the error: error

error = \_\_\_\_ - \_\_\_\_

# Calculate the slope: slope

slope = \_\_\_\_ \* \_\_\_\_ \* \_\_\_\_

# Print the slope

print(slope)

Answer code

import numpy as np

# Assign target, input data and weights

target = 0

input\_data = np.array([1, 2, 3])

weights = np.array([0, 2, 1])

# Calculate the predictions: preds

preds = (weights \* input\_data).sum()

# Calculate the error: error

error = preds - target

# Calculate the slope: slope

slope = input\_data \* error \* 2

# Print the slope

print(slope)

# Code 2.6 Improving model weights

Hurray! You've just calculated the slopes you need. Now it's time to use those slopes to improve your model. If you add the slopes to your weights, you will move in the right direction. However, it's possible to move too far in that direction. So you will want to take a small step in that direction first, using a lower learning rate, and verify that the model is improving.

The weights have been pre-loaded as weights, the actual value of the target as target, and the input data as input\_data. The predictions from the initial weights are stored as preds.

## Instructions

* Set the learning rate to be 0.01 and calculate the error from the original predictions. This has been done for you.
* Calculate the updated weights by subtracting the product of learning\_rate and slope from weights.
* Calculate the updated predictions by multiplying weights\_updated with input\_data and computing their sum.
* Calculate the error for the new predictions. Store the result as error\_updated.
* Hit 'Submit Answer' to compare the updated error to the original!

Sample code

# Set the learning rate: learning\_rate

learning\_rate = 0.01

# Calculate the predictions: preds

preds = (weights \* input\_data).sum()

# Calculate the error: error

error = preds - target

# Calculate the slope: slope

slope = 2 \* input\_data \* error

# Update the weights: weights\_updated

weights\_updated = \_\_\_\_

# Get updated predictions: preds\_updated

preds\_updated = \_\_\_\_

# Calculate updated error: error\_updated

error\_updated = \_\_\_\_

# Print the original error

print(error)

# Print the updated error

print(error\_updated)

Answer code

import numpy as np

# Assign target, input data and weights

target = 0

input\_data = np.array([1, 2, 3])

weights = np.array([0, 2, 1])

# Set the learning rate: learning\_rate

learning\_rate = 0.01

# Calculate the predictions: preds

preds = (weights \* input\_data).sum()

# Calculate the error: error

error = preds - target

# Calculate the slope: slope

slope = 2 \* input\_data \* error

# Update the weights: weights\_updated

weights\_updated = weights - learning\_rate \* slope

# Get updated predictions: preds\_updated

preds\_updated = (weights\_updated \* input\_data).sum()

# Calculate updated error: error\_updated

error\_updated = preds\_updated - target

# Print the original error

print(error)

# Print the updated error

print(error\_updated)

# Code 2.7 Making multiple updates to weights

You're now going to make multiple updates so you can dramatically improve your model weights, and see how the predictions improve with each update.

To keep your code clean, there is a pre-loaded get\_slope() function that takes input\_data, target, and weights as arguments. There is also a get\_mse() function that takes the same arguments. The input\_data, target, and weights have been pre-loaded.

This network does not have any hidden layers, and it goes directly from the input (with 3 nodes) to an output node. Note that weights is a single array.

We have also pre-loaded matplotlib.pyplot, and the error history will be plotted after you have done your gradient descent steps.

**Instructions**

* Using a for loop to iteratively update weights:
  + Calculate the slope using the get\_slope() function.
  + Update the weights using a learning rate of 0.01.
  + Calculate the mean squared error (mse) with the updated weights using the get\_mse()function.
  + Append mse to mse\_hist.
* Hit 'Submit Answer' to visualize mse\_hist. What trend do you notice?

Sample code

n\_updates = 20

mse\_hist = []

# Iterate over the number of updates

for i in range(n\_updates):

# Calculate the slope: slope

slope = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_)

# Update the weights: weights

weights = \_\_\_\_ - \_\_\_\_ \* \_\_\_\_

# Calculate mse with new weights: mse

mse = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_)

# Append the mse to mse\_hist

\_\_\_\_

# Plot the mse history

plt.plot(mse\_hist)

plt.xlabel('Iterations')

plt.ylabel('Mean Squared Error')

plt.show()

Answer code

import numpy as np

# Assign target, input data and weights

target = 0

input\_data = np.array([1, 2, 3])

weights = np.array([0, 2, 1])

n\_updates = 20

mse\_hist = []

# Iterate over the number of updates

for i in range(n\_updates):

# Calculate the slope: slope

slope = get\_slope(input\_data, target, weights)

# Update the weights: weights

weights = weights - 0.01 \* slope

# Calculate mse with new weights: mse

mse = get\_mse(input\_data, target, weights)

# Append the mse to mse\_hist

mse\_hist.append(mse)

# Plot the mse history

plt.plot(mse\_hist)

plt.xlabel('Iterations')

plt.ylabel('Mean Squared Error')

plt.show()

# Quiz 2.8 The relationship between forward and backward propagation

If you have gone through 4 iterations of calculating slopes (using backward propagation) and then updated weights, how many times must you have done forward propagation?

**Answer the question**

**Possible Answers**

Select one answer

* 0

**PRESS1**

* 1

**PRESS2**

* **4**

**PRESS3**

* 8

**PRESS4**

# Quiz 2.9 Thinking about backward propagation

If your predictions were all exactly right, and your errors were all exactly 0, the slope of the loss function with respect to your predictions would also be 0. In that circumstance, which of the following statements would be correct?

##### Answer the question

#### Possible Answers

Select one answer

* **The updates to all weights in the network would also be 0.**

**PRESS1**

* The updates to all weights in the network would be dependent on the activation functions.

**PRESS2**

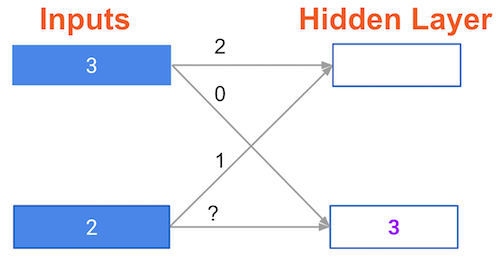
* The updates to all weights in the network would be proportional to values from the input data.

**PRESS3**

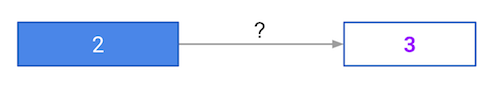
# Quiz 2.10 A round of backpropagation

In the network shown below, we have done forward propagation, and node values calculated as part of forward propagation are shown in white. The weights are shown in black. Layers after the question mark show the slopes calculated as part of back-prop, rather than the forward-prop values. Those slope values are shown in purple.

This network again uses the ReLU activation function, so the slope of the activation function is 1 for any node receiving a positive value as input. Assume the node being examined had a positive value (so the activation function's slope is 1).



What is the slope needed to update the weight with the question mark?



##### Answer the question

**50XP**

#### Possible Answers

Select one answer

* 0.

**PRESS1**

* 2.

**PRESS2**

* **6.**

**PRESS3**

* Not enough information.

**PRESS4**

gradient = input(white) \* slope(purple) \* ReLU\_slope(=1 here)   
= 2\*3\*1 = 6

# Tutorial Chapter 3

# Code 3.1 Understanding your data

You will soon start building models in Keras to predict wages based on various professional and demographic factors. Before you start building a model, it's good to understand your data by performing some exploratory analysis.

The data is pre-loaded into a pandas DataFrame called df. Use the .head() and .describe() methods in the IPython Shell for a quick overview of the DataFrame.

The target variable you'll be predicting is wage\_per\_hour. Some of the predictor variables are binary indicators, where a value of 1 represents True, and 0 represents False.

Of the 9 predictor variables in the DataFrame, how many are binary indicators? The min and max values as shown by .describe() will be informative here. How many binary indicator predictors are there?

**Instructions**

**Possible answers**

0

5

**6**

# Code 3.2 Specifying a model

hourly\_wages.csv

Now you'll get to work with your first model in Keras, and will immediately be able to run more complex neural network models on larger datasets compared to the first two chapters.

To start, you'll take the skeleton of a neural network and add a hidden layer and an output layer. You'll then fit that model and see Keras do the optimization so your model continually gets better.

As a start, you'll predict workers wages based on characteristics like their industry, education and level of experience. You can find the dataset in a pandas DataFrame called df. For convenience, everything in df except for the target has been converted to a NumPy array called predictors. The target, wage\_per\_hour, is available as a NumPy array called target.

For all exercises in this chapter, we've imported the Sequential model constructor, the Denselayer constructor, and pandas.

## Instructions

* Store the number of columns in the predictors data to n\_cols. This has been done for you.
* Start by creating a Sequential model called model.
* Use the .add() method on model to add a Dense layer.
  + Add 50 units, specify activation='relu', and the input\_shape parameter to be the tuple (n\_cols,) which means it has n\_cols items in each row of data, and any number of rows of data are acceptable as inputs.
* Add another Dense layer. This should have 32 units and a 'relu' activation.
* Finally, add an output layer, which is a Dense layer with a single node. Don't use any activation function here.

Sample code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Save the number of columns in predictors: n\_cols

n\_cols = predictors.shape[1]

# Set up the model: model

model = \_\_\_\_

# Add the first layer

\_\_\_\_.\_\_\_\_(\_\_\_\_(\_\_\_\_, \_\_\_\_=\_\_\_\_, \_\_\_\_=(\_\_\_\_)))

# Add the second layer

\_\_\_\_

# Add the output layer

\_\_\_\_

Answer code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Save the number of columns in predictors: n\_cols

n\_cols = predictors.shape[1]

# Set up the model: model

model = Sequential()

# Add the first layer

model.add(Dense(50, activation='relu', input\_shape=(n\_cols,)))

# Add the second layer

model.add(Dense(32, activation='relu'))

# Add the output layer

model.add(Dense(1))

# Code 3.3 Compiling the model

You're now going to compile the model you specified earlier. To compile the model, you need to specify the optimizer and loss function to use. In the video, Dan mentioned that the Adam optimizer is an excellent choice. You can read more about it as well as other Keras optimizers [**here**](https://keras.io/optimizers/#adam), and if you are really curious to learn more, you can read the [**original paper**](https://arxiv.org/abs/1412.6980v8) that introduced the Adam optimizer.

In this exercise, you'll use the Adam optimizer and the mean squared error loss function. Go for it!

## Instructions

* Compile the model using model.compile(). Your optimizer should be 'adam' and the lossshould be 'mean\_squared\_error'.

Sample code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Specify the model

n\_cols = predictors.shape[1]

model = Sequential()

model.add(Dense(50, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(32, activation='relu'))

model.add(Dense(1))

# Compile the model

\_\_\_\_

# Verify that model contains information from compiling

print("Loss function: " + model.loss)

Answer code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Specify the model

n\_cols = predictors.shape[1]

model = Sequential()

model.add(Dense(50, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(32, activation='relu'))

model.add(Dense(1))

# Compile the model

model.compile(optimizer='adam', loss='mean\_squared\_error')

# Verify that model contains information from compiling

print("Loss function: " + model.loss)

# Code 3.4 Fitting the model

You're at the most fun part. You'll now fit the model. Recall that the data to be used as predictive features is loaded in a NumPy array called predictors and the data to be predicted is stored in a NumPy array called target. Your model is pre-written and it has been compiled with the code from the previous exercise.

**Instructions**

* Fit the model. Remember that the first argument is the predictive features (predictors), and the data to be predicted (target) is the second argument.

Sample code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Specify the model

n\_cols = predictors.shape[1]

model = Sequential()

model.add(Dense(50, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(32, activation='relu'))

model.add(Dense(1))

# Compile the model

model.compile(optimizer='adam', loss='mean\_squared\_error')

# Fit the model

\_\_\_\_

Answer code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Specify the model

n\_cols = predictors.shape[1]

model = Sequential()

model.add(Dense(50, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(32, activation='relu'))

model.add(Dense(1))

# Compile the model

model.compile(optimizer='adam', loss='mean\_squared\_error')

# Fit the model

model.fit(predictors, target)

# Code 3.5 Understanding your classification data

titanic\_all\_numeric.csv

Now you will start modeling with a new dataset for a classification problem. This data includes information about passengers on the Titanic. You will use predictors such as age, fare and where each passenger embarked from to predict who will survive. This data is from [**a tutorial on data science competitions**](https://www.kaggle.com/c/titanic). Look [**here**](https://www.kaggle.com/c/titanic/data) for descriptions of the features.

The data is pre-loaded in a pandas DataFrame called df.

It's smart to review the maximum and minimum values of each variable to ensure the data isn't misformatted or corrupted. What was the maximum age of passengers on the Titanic? Use the .describe() method in the IPython Shell to answer this question.

## Instructions

### Possible answers

29.699.

**80.**

891.

It is not listed.

# Code 3.6 Last steps in classification models

You'll now create a classification model using the titanic dataset, which has been pre-loaded into a DataFrame called df. You'll take information about the passengers and predict which ones survived.

The predictive variables are stored in a NumPy array predictors. The target to predict is in df.survived, though you'll have to manipulate it for Keras. The number of predictive features is stored in n\_cols.

Here, you'll use the 'sgd' optimizer, which stands for [**Stochastic Gradient Descent**](https://en.wikipedia.org/wiki/Stochastic_gradient_descent). You'll learn more about this in the next chapter!

## Instructions

* Convert df.survived to a categorical variable using the to\_categorical() function.
* Specify a Sequential model called model.
* Add a Dense layer with 32 nodes. Use 'relu' as the activation and (n\_cols,) as the input\_shape.
* Add the Dense output layer. Because there are two outcomes, it should have 2 units, and because it is a classification model, the activation should be 'softmax'.
* Compile the model, using 'sgd' as the optimizer, 'categorical\_crossentropy' as the loss function, and metrics=['accuracy'] to see the accuracy (what fraction of predictions were correct) at the end of each epoch.
* Fit the model using the predictors and the target.

Sample code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

from tensorflow.keras.utils import to\_categorical

# Convert the target to categorical: target

target = \_\_\_\_

# Set up the model

model = \_\_\_\_

# Add the first layer

\_\_\_\_

# Add the output layer

\_\_\_\_

# Compile the model

\_\_\_\_

# Fit the model

\_\_\_\_

Answer code

# Import necessary modules

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

# Specify the model

n\_cols = predictors.shape[1]

model = Sequential()

model.add(Dense(50, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(32, activation='relu'))

model.add(Dense(1))

# Compile the model

model.compile(optimizer='adam', loss='mean\_squared\_error')

# Fit the model

model.fit(predictors, target)

# Code 3.7 Making predictions

The trained network from your previous coding exercise is now stored as model. New data to make predictions is stored in a NumPy array as pred\_data. Use model to make predictions on your new data.

In this exercise, your predictions will be probabilities, which is the most common way for data scientists to communicate their predictions to colleagues.

**Instructions**

* Create your predictions using the model's .predict() method on pred\_data.
* Use NumPy indexing to find the column corresponding to predicted probabilities of survival being True. This is the second *column* (index 1) of predictions. Store the result in predicted\_prob\_true and print it.

Sample code

# Specify, compile, and fit the model

model = Sequential()

model.add(Dense(32, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(2, activation='softmax'))

model.compile(optimizer='sgd',

loss='categorical\_crossentropy',

metrics=['accuracy'])

model.fit(predictors, target)

# Calculate predictions: predictions

predictions = \_\_\_\_

# Calculate predicted probability of survival: predicted\_prob\_true

predicted\_prob\_true = \_\_\_\_

# Print predicted\_prob\_true

print(predicted\_prob\_true)

Answer code

# Specify, compile, and fit the model

model = Sequential()

model.add(Dense(32, activation='relu', input\_shape = (n\_cols,)))

model.add(Dense(2, activation='softmax'))

model.compile(optimizer='sgd',

loss='categorical\_crossentropy',

metrics=['accuracy'])

model.fit(predictors, target)

# Calculate predictions: predictions

predictions = model.predict(pred\_data)

# Calculate predicted probability of survival: predicted\_prob\_true

predicted\_prob\_true = predictions[:,1]

# Print predicted\_prob\_true

print(predicted\_prob\_true)

# Tutorial Chapter 4

# Quiz 4.1 Diagnosing optimization problems

Which of the following could prevent a model from showing an improved loss in its first few epochs?

**Answer the question**

**Possible Answers**

Select one answer

* Learning rate too low.

**PRESS1**

* Learning rate too high.

**PRESS2**

* Poor choice of activation function.

**PRESS3**

* **All of the above.**

**PRESS4**

# Code 4.2 Changing optimization parameters

It's time to get your hands dirty with optimization. You'll now try optimizing a model at a very low learning rate, a very high learning rate, and a "just right" learning rate. You'll want to look at the results after running this exercise, remembering that a low value for the loss function is good.

For these exercises, we've pre-loaded the predictors and target values from your previous classification models (predicting who would survive on the Titanic). You'll want the optimization to start from scratch every time you change the learning rate, to give a fair comparison of how each learning rate did in your results. So we have created a function get\_new\_model() that creates an unoptimized model to optimize.

## Instructions

* Import SGD from tensorflow.keras.optimizers.
* Create a list of learning rates to try optimizing with called lr\_to\_test. The learning rates in it should be .000001, 0.01, and 1.
* Using a for loop to iterate over lr\_to\_test:
  + Use the get\_new\_model() function to build a new, unoptimized model.
  + Create an optimizer called my\_optimizer using the SGD() constructor with keyword argument lr=lr.
  + Compile your model. Set the optimizer parameter to be the SGD object you created above, and because this is a classification problem, use 'categorical\_crossentropy' for the lossparameter.
  + Fit your model using the predictors and target.

Sample code

# Import the SGD optimizer

\_\_\_\_

# Create list of learning rates: lr\_to\_test

lr\_to\_test = \_\_\_\_

# Loop over learning rates

for lr in lr\_to\_test:

print('\n\nTesting model with learning rate: %f\n'%lr )

# Build new model to test, unaffected by previous models

model = \_\_\_\_

# Create SGD optimizer with specified learning rate: my\_optimizer

my\_optimizer = \_\_\_\_

# Compile the model

\_\_\_\_

# Fit the model

\_\_\_\_

Answer code

# Import the SGD optimizer

from tensorflow.keras.optimizers import SGD

# Create list of learning rates: lr\_to\_test

lr\_to\_test = [0.000001, 0.01, 1]

# Loop over learning rates

for lr in lr\_to\_test:

print('\n\nTesting model with learning rate: %f\n'%lr )

# Build new model to test, unaffected by previous models

model = get\_new\_model()

# Create SGD optimizer with specified learning rate: my\_optimizer

my\_optimizer = SGD(lr=lr)

# Compile the model

model.compile(optimizer=my\_optimizer, loss='categorical\_crossentropy')

# Fit the model

model.fit(predictors, target)

# Code 4.3 Evaluating model accuracy on validation dataset

Now it's your turn to monitor model accuracy with a validation data set. A model definition has been provided as model. Your job is to add the code to compile it and then fit it. You'll check the validation score in each epoch.

**Instructions**

* Compile your model using 'adam' as the optimizer and 'categorical\_crossentropy' for the loss. To see what fraction of predictions are correct (the accuracy) in each epoch, specify the additional keyword argument metrics=['accuracy'] in model.compile().
* Fit the model using the predictors and target. Create a validation split of 30% (or 0.3). This will be reported in each epoch.

Sample code

# Save the number of columns in predictors: n\_cols

n\_cols = predictors.shape[1]

input\_shape = (n\_cols,)

# Specify the model

model = Sequential()

model.add(Dense(100, activation='relu', input\_shape = input\_shape))

model.add(Dense(100, activation='relu'))

model.add(Dense(2, activation='softmax'))

# Compile the model

\_\_\_\_

# Fit the model

hist = \_\_\_\_

Answer code

# Save the number of columns in predictors: n\_cols

n\_cols = predictors.shape[1]

input\_shape = (n\_cols,)

# Specify the model

model = Sequential()

model.add(Dense(100, activation='relu', input\_shape = input\_shape))

model.add(Dense(100, activation='relu'))

model.add(Dense(2, activation='softmax'))

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Fit the model

hist = model.fit(predictors, target, validation\_split=0.3)

# Code 4.4 Early stopping: Optimizing the optimization

Now that you know how to monitor your model performance throughout optimization, you can use early stopping to stop optimization when it isn't helping any more. Since the optimization stops automatically when it isn't helping, you can also set a high value for epochs in your call to .fit(), as Dan showed in the video.

The model you'll optimize has been specified as model. As before, the data is pre-loaded as predictors and target.

**Instructions**

* Import EarlyStopping from tensorflow.keras.callbacks.
* Compile the model, once again using 'adam' as the optimizer, 'categorical\_crossentropy'as the loss function, and metrics=['accuracy'] to see the accuracy at each epoch.
* Create an EarlyStopping object called early\_stopping\_monitor. Stop optimization when the validation loss hasn't improved for 2 epochs by specifying the patience parameter of EarlyStopping() to be 2.
* Fit the model using the predictors and target. Specify the number of epochs to be 30 and use a validation split of 0.3. In addition, pass [early\_stopping\_monitor] to the callbacksparameter.

Sample code

# Import EarlyStopping

\_\_\_\_

# Save the number of columns in predictors: n\_cols

n\_cols = predictors.shape[1]

input\_shape = (n\_cols,)

# Specify the model

model = Sequential()

model.add(Dense(100, activation='relu', input\_shape = input\_shape))

model.add(Dense(100, activation='relu'))

model.add(Dense(2, activation='softmax'))

# Compile the model

\_\_\_\_

# Define early\_stopping\_monitor

early\_stopping\_monitor = \_\_\_\_

# Fit the model

\_\_\_\_

Answer code

# Import EarlyStopping

from tensorflow.keras.callbacks import EarlyStopping

# Save the number of columns in predictors: n\_cols

n\_cols = predictors.shape[1]

input\_shape = (n\_cols,)

# Specify the model

model = Sequential()

model.add(Dense(100, activation='relu', input\_shape = input\_shape))

model.add(Dense(100, activation='relu'))

model.add(Dense(2, activation='softmax'))

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Define early\_stopping\_monitor

early\_stopping\_monitor = EarlyStopping(patience=2)

# Fit the model

model.fit(predictors, target, epochs=30, validation\_split=0.3, callbacks=[early\_stopping\_monitor])

# Code 4.5 Experimenting with wider networks

Now you know everything you need to begin experimenting with different models!

A model called model\_1 has been pre-loaded. You can see a summary of this model printed in the IPython Shell. This is a relatively small network, with only 10 units in each hidden layer.

In this exercise you'll create a new model called model\_2 which is similar to model\_1, except it has 100 units in each hidden layer.

After you create model\_2, both models will be fitted, and a graph showing both models loss score at each epoch will be shown. We added the argument verbose=False in the fitting commands to print out fewer updates, since you will look at these graphically instead of as text.

Because you are fitting two models, it will take a moment to see the outputs after you hit run, so be patient.

**Instructions**

* Create model\_2 to replicate model\_1, but use 100 nodes instead of 10 for the first two Dense layers you add with the 'relu' activation. Use 2 nodes for the Dense output layer with 'softmax' as the activation.
* Compile model\_2 as you have done with previous models: Using 'adam' as the optimizer, 'categorical\_crossentropy' for the loss, and metrics=['accuracy'].
* Hit 'Submit Answer' to fit both the models and visualize which one gives better results! Notice the keyword argument verbose=False in model.fit(): This prints out fewer updates, since you'll be evaluating the models graphically instead of through text.

Sample code

# Define early\_stopping\_monitor

early\_stopping\_monitor = EarlyStopping(patience=2)

# Create the new model: model\_2

model\_2 = \_\_\_\_

# Add the first and second layers

\_\_\_\_.\_\_\_\_(\_\_\_\_(\_\_\_\_, \_\_\_\_=\_\_\_\_, input\_shape=input\_shape))

\_\_\_\_

# Add the output layer

\_\_\_\_

# Compile model\_2

\_\_\_\_

# Fit model\_1

model\_1\_training = model\_1.fit(predictors, target, epochs=15, validation\_split=0.2, callbacks=[early\_stopping\_monitor], verbose=False)

# Fit model\_2

model\_2\_training = model\_2.fit(predictors, target, epochs=15, validation\_split=0.2, callbacks=[early\_stopping\_monitor], verbose=False)

# Create the plot

plt.plot(model\_1\_training.history['val\_loss'], 'r', model\_2\_training.history['val\_loss'], 'b')

plt.xlabel('Epochs')

plt.ylabel('Validation score')

plt.show()

Answer code

# Define early\_stopping\_monitor

early\_stopping\_monitor = EarlyStopping(patience=2)

# Create the new model: model\_2

model\_2 = Sequential()

# Add the first and second layers

model\_2.add(Dense(100, activation='relu', input\_shape=input\_shape))

model\_2.add(Dense(100, activation='relu'))

# Add the output layer

model\_2.add(Dense(2, activation='softmax'))

# Compile model\_2

model\_2.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Fit model\_1

model\_1\_training = model\_1.fit(predictors, target, epochs=15, validation\_split=0.2, callbacks=[early\_stopping\_monitor], verbose=False)

# Fit model\_2

model\_2\_training = model\_2.fit(predictors, target, epochs=15, validation\_split=0.2, callbacks=[early\_stopping\_monitor], verbose=False)

# Create the plot

plt.plot(model\_1\_training.history['val\_loss'], 'r', model\_2\_training.history['val\_loss'], 'b')

plt.xlabel('Epochs')

plt.ylabel('Validation score')

plt.show()

# Code 4.6 Adding layers to a network

You've seen how to experiment with wider networks. In this exercise, you'll try a deeper network (more hidden layers).

Once again, you have a baseline model called model\_1 as a starting point. It has 1 hidden layer, with 10 units. You can see a summary of that model's structure printed out. You will create a similar network with 3 hidden layers (still keeping 10 units in each layer).

This will again take a moment to fit both models, so you'll need to wait a few seconds to see the results after you run your code.

**Instructions**

* Specify a model called model\_2 that is like model\_1, but which has 3 hidden layers of 10 units instead of only 1 hidden layer.
  + Use input\_shape to specify the input shape in the first hidden layer.
  + Use 'relu' activation for the 3 hidden layers and 'softmax' for the output layer, which should have 2 units.
* Compile model\_2 as you have done with previous models: Using 'adam' as the optimizer, 'categorical\_crossentropy' for the loss, and metrics=['accuracy'].
* Hit 'Submit Answer' to fit both the models and visualize which one gives better results!

Sample code

# The input shape to use in the first hidden layer

input\_shape = (n\_cols,)

# Create the new model: model\_2

model\_2 = \_\_\_\_

# Add the first, second, and third hidden layers

\_\_\_\_

\_\_\_\_

\_\_\_\_

# Add the output layer

\_\_\_\_

# Compile model\_2

\_\_\_\_

# Fit model 1

model\_1\_training = model\_1.fit(predictors, target, epochs=15, validation\_split=0.4, verbose=False)

# Fit model 2

model\_2\_training = model\_2.fit(predictors, target, epochs=15, validation\_split=0.4, verbose=False)

# Create the plot

plt.plot(model\_1\_training.history['val\_loss'], 'r', model\_2\_training.history['val\_loss'], 'b')

plt.xlabel('Epochs')

plt.ylabel('Validation score')

plt.show()

Answer code

# The input shape to use in the first hidden layer

input\_shape = (n\_cols,)

# Create the new model: model\_2

model\_2 = Sequential()

# Add the first, second, and third hidden layers

model\_2.add(Dense(10, activation='relu', input\_shape=input\_shape))

model\_2.add(Dense(10, activation='relu'))

model\_2.add(Dense(10, activation='relu'))

# Add the output layer

model\_2.add(Dense(2, activation='softmax'))

# Compile model\_2

model\_2.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Fit model 1

model\_1\_training = model\_1.fit(predictors, target, epochs=15, validation\_split=0.4, verbose=False)

# Fit model 2

model\_2\_training = model\_2.fit(predictors, target, epochs=15, validation\_split=0.4, verbose=False)

# Create the plot

plt.plot(model\_1\_training.history['val\_loss'], 'r', model\_2\_training.history['val\_loss'], 'b')

plt.xlabel('Epochs')

plt.ylabel('Validation score')

plt.show()

# Quiz 4.7 Experimenting with model structures

You've just run an experiment where you compared two networks that were identical except that the 2nd network had an extra hidden layer. You see that this 2nd network (the deeper network) had better performance. Given that, which of the following would be a good experiment to run next for even better performance?

**Answer the question**

**Possible Answers**

Select one answer

* Try a new network with fewer layers than anything you have tried yet.

**PRESS1**

* **Use more units in each hidden layer.**

**PRESS2**

* Use fewer units in each hidden layer.

**PRESS3**

# Code 4.8 Building your own digit recognition model

mnist.csv

You've reached the final exercise of the course - you now know everything you need to build an accurate model to recognize handwritten digits!

We've already done the basic manipulation of the MNIST dataset shown in the video, so you have X and y loaded and ready to model with. Sequential and Dense from tensorflow.kerasare also pre-imported.

To add an extra challenge, we've loaded only 2500 images, rather than 60000 which you will see in some published results. Deep learning models perform better with more data, however, they also take longer to train, especially when they start becoming more complex.

If you have a computer with a CUDA compatible GPU, you can take advantage of it to improve computation time. If you don't have a GPU, no problem! You can set up a deep learning environment in the cloud that can run your models on a GPU. Here is a [**blog post**](https://www.datacamp.com/community/tutorials/deep-learning-jupyter-aws) by Dan that explains how to do this - check it out after completing this exercise! It is a great next step as you continue your deep learning journey.

Ready to take your deep learning to the next level? Check out [**Advanced Deep Learning with Keras**](https://www.datacamp.com/courses/advanced-deep-learning-with-keras) to see how the Keras functional API lets you build domain knowledge to solve new types of problems. Once you know how to use the functional API, take a look at [**Image Processing with Keras in Python**](https://www.datacamp.com/courses/image-processing-with-keras-in-python) to learn image-specific applications of Keras.

## Instructions

* Create a Sequential object to start your model. Call this model.
* Add the first Dense hidden layer of 50 units to your model with 'relu' activation. For this data, the input\_shape is (784,).
* Add a second Dense hidden layer with 50 units and a 'relu' activation function.
* Add the output layer. Your activation function should be 'softmax', and the number of nodes in this layer should be the same as the number of possible outputs in this case: 10.
* Compile model as you have done with previous models: Using 'adam' as the optimizer, 'categorical\_crossentropy' for the loss, and metrics=['accuracy'].
* Fit the model using X and y using a validation\_split of 0.3 and 10 epochs.

Sample code

# Create the model: model

model = \_\_\_\_

# Add the first hidden layer

\_\_\_\_

# Add the second hidden layer

\_\_\_\_

# Add the output layer

\_\_\_\_

# Compile the model

\_\_\_\_

# Fit the model

\_\_\_\_

Answer code

# Create the model: model

model = Sequential()

# Add the first hidden layer

model.add(Dense(50, activation='relu', input\_shape=(X.shape[1],)))

# Add the second hidden layer

model.add(Dense(50, activation='relu'))

# Add the output layer

model.add(Dense(y.shape[1], activation='softmax'))

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Fit the model

model.fit(X, y, validation\_split=0.3, epochs=10)

# Chapter 5 Structured data classification

This example demonstrates how to do structured data classification, starting from a raw CSV file. Our data includes both numerical and categorical features. We will use Keras preprocessing layers to normalize the numerical features and vectorize the categorical ones. Note that this example should be run with TensorFlow 2.5 or higher.

<https://www.kaggle.com/code/ryanholbrook/binary-classification>

## Tutorial

Introduction

In this exercise, you'll build a model to predict hotel cancellations with a binary classifier.

Use a neural network to predict cancellations in hotel reservations with the Hotel Cancellations dataset.

<https://www.kaggle.com/kernels/fork/11887335>

## Assignment 1

Our dataset is provided by the Cleveland Clinic Foundation for Heart Disease. It's a CSV file with 303 rows. Each row contains information about a patient (a sample), and each column describes an attribute of the patient (a feature). We use the features to predict whether a patient has a heart disease (binary classification).

<https://keras.io/examples/structured_data/structured_data_classification_from_scratch/>

## Final Project 1

[Influencers in Social Networks](https://www.kaggle.com/competitions/predict-who-is-more-influential-in-a-social-network/overview)

Data Science London and the UK Windows Azure Users Group in partnership with Microsoft and Peerindex, announce the Influencers in Social Networks competition as part of The Big Data Hackathon. This competition asks you to predict human judgements about who is more influential on social media.

The dataset, provided by Peerindex, comprises a standard, pair-wise preference learning task. Each datapoint describes two individuals, A and B. For each person, 11 pre-computed, non-negative numeric features based on twitter activity (such as volume of interactions, number of followers, etc) are provided.

The binary label represents a human judgement about which one of the two individuals is more influential. A label '1' means A is more influential than B. 0 means B is more influential than A. The goal of the challenge is to train a machine learning model which, for pairs of individuals, predicts the human judgement on who is more influential with high accuracy. Labels for the dataset have been collected by PeerIndex using an application similar to the one described in this post.

A python script computing a sample benchmark solution is available here: https://gist.github.com/fhuszar/5372873

<https://github.com/bigrewal/Influencers-in-Social-Networks/blob/master/influencers.ipynb>

# Chapter 6 Embedding and Recommendations System

Introduction to embedding

<https://developers.google.com/machine-learning/crash-course/embeddings>

Introduction to Recommendation system

Recommender systems are widely employed in industry and are ubiquitous in our daily lives. These systems are utilized in a number of areas such as online shopping sites (e.g., amazon.com), music/movie services site (e.g., Netflix and Spotify), mobile application stores (e.g., IOS app store and google play), online advertising, just to name a few.

The major goal of recommender systems is to help users discover relevant items such as movies to watch, text to read or products to buy, so as to create a delightful user experience. Moreover, recommender systems are among the most powerful machine learning systems that online retailers implement in order to drive incremental revenue. Recommender systems are replacements of search engines by reducing the efforts in proactive searches and surprising users with offers they never searched for. Many companies managed to position themselves ahead of their competitors with the help of more effective recommender systems. As such, recommender systems are central to not only our everyday lives but also highly indispensable in some industries.

In this chapter, we will cover the fundamentals and advancements of recommender systems, along with exploring some common fundamental techniques for building recommender systems with different data sources available and their implementations. Specifically, you will learn how to predict the rating a user might give to a prospective item, how to generate a recommendation list of items and how to predict the click-through rate from abundant features. These tasks are commonplace in real-world applications. By studying this chapter, you will get hands-on experience pertaining to solving real world recommendation problems with not only classical methods but the more advanced deep learning based models as well.

<https://www.d2l.ai/chapter_recommender-systems/index.html>

<https://www.kaggle.com/code/ibtesama/getting-started-with-a-movie-recommendation-system>

## Tutorial

[Building a book Recommendation System using Keras](https://gilberttanner.com/blog/building-a-book-recommendation-system-usingkeras/)

<https://github.com/TannerGilbert/Tutorials/blob/master/Recommendation%20System/Recommendation%20System.ipynb?ref=gilberttanner.com>

## Assignment 2

[Collaborative Filtering for Movie Recommendations](https://keras.io/examples/structured_data/collaborative_filtering_movielens/)

https://keras.io/examples/structured\_data/collaborative\_filtering\_movielens/

## Final Project 2

[Food Recommendation System](https://www.kaggle.com/datasets/schemersays/food-recommendation-system/data)

This dataset represents the data related to food recommender system. Two datasets are included in this dataset file. First includes the dataset related to the foods, ingredients, cuisines involved. Second, includes the dataset of the rating system for the recommendation system.

# Chapter 7 Convolutional Neural Network

<https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns>

There’s been a lot of buzz about Convolution Neural Networks (CNNs) in the past few years, especially because of how they’ve revolutionized the field of Computer Vision. In this post, we’ll build on a basic background knowledge of neural networks and explore what CNNs are, understand how they work, and build a real one from scratch (using only numpy) in Python.

[Introduction to Convolutional Neural Networks](https://victorzhou.com/blog/intro-to-cnns-part-1/)

## Tutorial

[The Problem: MNIST digit classification](https://victorzhou.com/blog/keras-cnn-tutorial/)

## Assignment 3

[CIFAR-10 dataset](https://www.datacamp.com/tutorial/cnn-tensorflow-python)

## Final project 3

[Image classification with flowers photo dataset](https://www.tensorflow.org/tutorials/images/classification)

## Final project 4

[Vietnamese Currencies Classification](https://www.kaggle.com/code/boyle01/vietnamese-currencies-classification-96-76)

# Chapter 8 Recurrent Neural Network

[Introduction to Recurrent Neural Networks](https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/)

When it comes to sequential or time series data, traditional feedforward networks cannot be used for learning and prediction. A mechanism is required to retain past or historical information to forecast future values. Recurrent neural networks, or RNNs for short, are a variant of the conventional feedforward artificial neural networks that can deal with sequential data and can be trained to hold knowledge about the past.

[Understanding Simple Recurrent Neural Networks in Keras](https://machinelearningmastery.com/understanding-simple-recurrent-neural-networks-in-keras/)

This tutorial is designed for anyone looking for an understanding of how recurrent neural networks (RNN) work and how to use them via the Keras deep learning library. While the Keras library provides all the methods required for solving problems and building applications, it is also important to gain an insight into how everything works. In this article, the computations taking place in the RNN model are shown step by step. Next, a complete end-to-end system for time series prediction is developed.

## Tutorial

[Running the RNN on Sunspots Dataset](https://machinelearningmastery.com/understanding-simple-recurrent-neural-networks-in-keras/)

## Assignment 4

[MasterCard Stock Price Prediction](https://www.datacamp.com/tutorial/tutorial-for-recurrent-neural-network)

## Final project 5

[Shampoo Sales Dataset with LTMS](https://machinelearningmastery.com/time-series-forecasting-long-short-term-memory-network-python/)

# Chapter 9 Natural Language Processing

Natural Language Processing (NLP) is one of the hottest areas of artificial intelligence (AI) thanks to applications like text generators that compose coherent essays, chatbots that fool people into thinking they’re sentient, and text-to-image programs that produce photorealistic images of anything you can describe. Recent years have brought a revolution in the ability of computers to understand human languages, programming languages, and even biological and chemical sequences, such as DNA and protein structures, that resemble language. The latest AI models are unlocking these areas to analyze the meanings of input text and generate meaningful, expressive output.

[A COMPLETE GUIDE TO Natural Language Processing](https://www.deeplearning.ai/resources/natural-language-processing/)

## Tutorial

[Sentiment Analysis](https://github.com/skillcate/sentiment-analysis-with-deep-neural-networks/tree/main)

Training three separate Sentiment Classification Models, namely: Simple Neural Net, CNN & LSTM, on the popular IMDb Movie Reviews dataset.

[Using BERT](https://www.tensorflow.org/text/tutorials/classify_text_with_bert)

## Assignment 5

[Implement an image captioning model using a CNN and a Transformer](https://keras.io/examples/vision/image_captioning/)

## Final project 6

[Semantic Similarity with BERT](https://keras.io/examples/nlp/semantic_similarity_with_bert/)

## Final project 7

[Vietnamese Sentiment Analyst](https://www.kaggle.com/datasets/linhlpv/vietnamese-sentiment-analyst/data)

## Final project 8

[English to vietnamese translation](https://www.kaggle.com/datasets/hungnm/englishvietnamese-translation/code)

[Machine Translation(en to vi) with attention](https://www.kaggle.com/code/minhhngchong/machine-translation-en-to-vi-with-attention)

# Chapter 10 Generative Adversarial Networks

[Generative Adversarial Networks](https://www.datacamp.com/tutorial/generative-adversarial-networks)

[Llm Vs Gan: Understanding Adversarial Networks](https://www.restack.io/p/adversarial-networks-answer-llm-vs-gan-cat-ai)

## Tutorial

[A Simple Generative Adversarial Network with Keras](https://www.datacamp.com/tutorial/generative-adversarial-networks)

## Assignment 5

[Implement a Deep Convolutional Generative Adversarial Networks](https://pyimagesearch.com/2020/11/16/gans-with-keras-and-tensorflow/)

## Final project 9

[Develop a 1D Generative Adversarial Network](https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-1-dimensional-function-from-scratch-in-keras/)

## Final project 10

[Generate Cifar Images with a DCGAN](https://github.com/peremartra/GANs/blob/main/C2_GAN_CIFAR.ipynb)