**LECTURE**

**https://github.com/JNYH/DataCamp\_Introduction\_to\_Tensorflow\_in\_Python/blob/4daba9dd303c2ca3317ea906e8b4f059dd214ecd/Course\_notes\_solutions\_answers\_Introduction\_to\_Tensorflow\_in\_Python.pdf**

**1. Constants and variables**

00:00 - 00:18

Hi! My name is Isaiah Hull and this is a course on the fundamentals of the TensorFlow API in Python. In our first video, we will briefly introduce TensorFlow and then discuss its two basic objects of computation: constants and variables.

**2. What is TensorFlow?**

00:18 - 00:56

TensorFlow is an open-source library for graph-based numerical computation. It was developed by the Google Brain Team. It has both low and high level APIs. You can use TensorFlow to perform addition, multiplication, and differentiation. You can also use it to design and train machine learning models. TensorFlow two point zero brought with it substantial changes. Eager execution is now enabled by default, which allows users to write simpler and more intuitive code. Additionally, model building is now centered around the Keras and Estimators high-level APIs.

**3. What is a tensor?**

00:56 - 01:14

The TensorFlow documentation describes a tensor as "a generalization of vectors and matrices to potentially higher dimensions." Now, if you are not familiar with linear algebra, you can simply think of a tensor as a collection of numbers, which is arranged into a particular shape.

**4. What is a tensor?**

01:14 - 01:39

As an example, let's say you have a slice of bread and you cut it into 9 pieces. One of those 9 pieces is a 0-dimensional tensor. This corresponds to a single number. A collection of 3 pieces that form a row or column is a 1-dimensional tensor. All 9 pieces together are a 2-dimensional tensor. And the whole loaf, which contains many slices, is a 3-dimensional tensor.

**5. Defining tensors in TensorFlow**

01:39 - 01:56

Now that you know what a tensor is, let's define a few. We will start by importing tensorflow as tf. We will then define 0-, 1-, 2-, and 3-dimensional tensors. Note that each object will be a tf dot Tensor object.

**6. Defining tensors in TensorFlow**

01:56 - 02:06

If we want to print the array contained in that object, we can apply the dot numpy method and pass the resulting object to the print function.

**7. Defining constants in TensorFlow**

02:06 - 02:28

We next move on to constants, which are the simplest category of tensor in TensorFlow. A constant does not change and cannot be trained. It can, however, have any dimension. In the code block, we've defined two constants. The constant a is a 2x3 tensor of 3s. The constant b is a 2x2 tensor, which is constructed from the 1-dimensional tensor: 1, 2, 3, 4.

**8. Using convenience functions to define constants**

02:28 - 03:06

In the previous slide, we worked exclusively with the constant operation. However, in some cases, there are more convenient options for defining certain types of special tensors. You can use the zeros or ones operations to generate a tensor of arbitrary dimension that is populated entirely with zeros or ones. You can use the zeros\_like or ones\_like operations to populate tensors with zeros and ones, copying the dimension of some input tensor. Finally, you can use the fill operation to populate a tensor of arbitrary dimension with the same scalar value in each element.

**9. Defining and initializing variables**

03:06 - 03:52

Unlike a constant, a variable's value can change during computation. The value of a variable is shared, persistent, and modifiable. However, its data type and shape are fixed. Let's take a look at how variables are constructed and used in TensorFlow. In the code, we first define a variable, a0, which is a 1-dimensional tensor with 6 elements. We can set its datatype to a 32-bit float or something else, such as a 16-bit int, as we have for a1. We then define a constant, b. And define c0 as the product of a0 and b. Note that certain TensorFlow operations, such as tf.multiply are overloaded, which allows us to use the simpler a0\*b expression instead.

**10. Let's practice!**

03:52 - 03:57

It's now time to put what you've learned to use in some exercises.

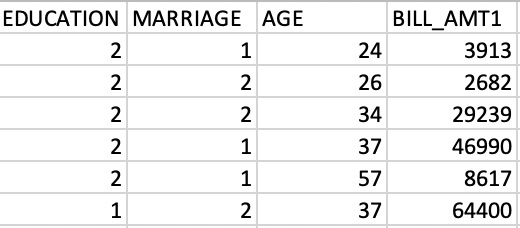
## Exercise

# Defining data as constants

Throughout this course, we will use tensorflow version 2.6.0 and will exclusively import the submodules needed to complete each exercise. This will usually be done for you, but you will do it in this exercise by importing constant from tensorflow.

After you have imported constant, you will use it to transform a numpy array, credit\_numpy, into a tensorflow constant, credit\_constant. This array contains feature columns from a dataset on credit card holders and is previewed in the image below. We will return to this dataset in later chapters.

Note that tensorflow 2 allows you to use data as either a numpy array or a tensorflow constant object. Using a constant will ensure that any operations performed with that object are done in tensorflow.



## Instructions

100 XP

* Import the constant submodule from the tensorflow module.
* Convert the credit\_numpy array into a constant object in tensorflow. Do not set the data type.

# Import constant from TensorFlow

from tensorflow import \_\_\_\_

# Convert the credit\_numpy array into a tensorflow constant

credit\_constant = constant(\_\_\_\_)

# Print constant datatype

print('\n The datatype is:', credit\_constant.dtype)

# Print constant shape

print('\n The shape is:', credit\_constant.shape)

# Import constant from TensorFlow

from tensorflow import constant

# Convert the credit\_numpy array into a tensorflow constant

credit\_constant = constant(credit\_numpy)

# Print constant datatype

print('\n The datatype is:', credit\_constant.dtype)

# Print constant shape

print('\n The shape is:', credit\_constant.shape)

**The datatype is: <dtype: 'float64'>**

**The shape is: (30000, 4)**

# Import constant from TensorFlow

from tensorflow import constant

# Convert the credit\_numpy array into a tensorflow constant

credit\_constant = constant(credit\_numpy)

# Print constant datatype

print('\n The datatype is:', credit\_constant.dtype)

# Print constant shape

print('\n The shape is:', credit\_constant.shape)

Excellent! You now understand how constants are used in tensorflow. In the following exercise, you'll practice defining variables.

## Exercise

# Defining variables

Unlike a constant, a variable's value can be modified. This will be useful when we want to train a model by updating its parameters.

Let's try defining and printing a variable. We'll then convert the variable to a numpy array, print again, and check for differences. Note that Variable(), which is used to create a variable tensor, has been imported from tensorflow and is available to use in the exercise.

## Instructions

* Define a variable, A1, as the 1-dimensional tensor: [1, 2, 3, 4].

# Convert A1 to a numpy array and assign it to B1

B1 = A1.numpy()

# Print B1

print('\n B1: ', B1)

Apply .numpy() to A1 and assign it to B1.

# Define the 1-dimensional variable A1

A1 = Variable([1, 2, 3, 4])

# Print the variable A1

print('\n A1: ', A1)

# Define the 1-dimensional variable A1

A1 = Variable([1, 2, 3, 4])

# Print the variable A1

print('\n A1: ', A1)

# Convert A1 to a numpy array and assign it to B1

B1 = A1.numpy()

# Print B1

print('\n B1: ', B1)

A1: <tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>

B1: [1 2 3 4]

Nice work! Did you notice any differences between the print statements for A1 and B1? In our next exercise, we'll review how to check the properties of a tensor after it is already defined.

**1. Basic operations**

00:00 - 00:05

In this video, we'll talk about basic operations in TensorFlow.

**2. What is a TensorFlow operation?**

00:05 - 00:17

TensorFlow has a model of computation that revolves around the use of graphs. A TensorFlow graph contains edges and nodes, where the edges are tensors and the nodes are operations.

**3. What is a TensorFlow operation?**

00:17 - 00:28

In the graph shown, which was drawn using TensorFlow, the const operations define 2 by 2 constant tensors. Two tensors are summed using the add operation.

**4. What is a TensorFlow operation?**

00:28 - 00:33

Another two tensors are then summed using the add operation.

**5. What is a TensorFlow operation?**

00:33 - 00:38

Finally, the resulting matrices are multiplied together with the matmul operation.

**6. Applying the addition operator**

00:38 - 00:53

Let's start with the addition operator. We will first import the constant and add operations. We may now use constant to define 0-dimensional, 1-dimensional, and 2-dimensional tensors.

**7. Applying the addition operator**

00:53 - 01:03

Finally, let's add them together using the operation for tensor addition. Note that we can perform scalar addition with A0 and B0, vector addition with A1 and B1, and matrix addition with A2 and B2.

**8. Performing tensor addition**

01:03 - 01:42

The add operation performs element-wise addition with two tensors. Each pair of tensors added must have the same shape. Element-wise addition of the scalars 1 and 2 yields the scalar 3. Element-wise addition of the vectors 1,2 and 3,4 yields the vector 4,6. Element-wise addition of the matrices 1,2,3,4 and 5,6,7,8 yields the matrix 6,8,10,12. Furthermore, the add operator is overloaded, which means that we can also perform addition using the plus symbol.

**9. How to perform multiplication in TensorFlow**

01:42 - 02:16

We will consider both element-wise and matrix multiplication. For element-wise multiplication, which is performed with the multiply operation, the tensors involved must have the same shape. For instance, you may want to multiply the vector 1,2,3 by 3,4,5 or 1,2 by 3,4. For matrix multiplication, you use the matmul operator. Note that performing matmul(A,B) requires that the number of columns of A equal the number of rows of B.

**10. Applying the multiplication operators**

02:16 - 03:01

Let's look at some examples of multiplication in TensorFlow. We'll import the ones operator, along with the two types of multiplication we will use. We will also define a scalar, A0, a 3 by 1 vector of ones, a 3 by 4 vector of ones, and a 4 by 3 vector of ones. What operations can be performed using these tensors of ones? We can perform element-wise multiplication of any element by itself, such as A0 by A0, A31 by A31, or A34 by A34. We can also perform matrix multiplication of A43 by A34, but not A43 by A43.

**11. Summing over tensor dimensions**

03:01 - 03:25

Finally, we end this lesson by discussing summation over tensors, which is performed using the reduce sum operator. This can be used to sum over all dimensions of a tensor or just one. Let's see how this works in practice. We will import ones and reduce sum from tensorflow. We will then define a 2 by 3 by 4 tensor that consists of ones.

**12. Summing over tensor dimensions**

03:25 - 03:52

If we sum over all elements of A, we get 24, since the tensor contains 24 elements, all of which are 1. If we sum over dimension 0, we get a 3 by 4 matrix of 2s. If we sum over 1, we get a 2 by 4 matrix of 3s. And if we sum over 2, we get a 2 by 3 matrix of 4s. In each case, we reduce the size of the tensor by summing over one of its dimensions.

**13. Let's practice!**

03:52 - 04:01

Now that you understand how to perform basic operations in TensorFlow, let's put this to work with some exercises.

## Exercise

# Performing element-wise multiplication

Element-wise multiplication in TensorFlow is performed using two tensors with identical shapes. This is because the operation multiplies elements in corresponding positions in the two tensors. An example of an element-wise multiplication, denoted by the

symbol, is shown below:

In this exercise, you will perform element-wise multiplication, paying careful attention to the shape of the tensors you multiply. Note that multiply(), constant(), and ones\_like() have been imported for you.

## Instructions

* Define the tensors A1 and A23 as constants.
* Set B1 to be a tensor of ones with the same shape as A1.
* Set B23 to be a tensor of ones with the same shape as A23.
* Set C1 and C23 equal to the element-wise products of A1 and B1, and A23 and B23, respectively.

**# Define tensors A1 and A23 as constants**

**A1 = constant([1, 2, 3, 4])**

**A23 = constant([[1, 2, 3], [1, 6, 4]])**

**# Define B1 and B23 to have the correct shape**

**B1 = ones\_like(A1)**

**B23 = ones\_like(A23)**

**# Perform element-wise multiplication**

**C1 = multiply(A1, B1)**

**C23 = multiply(A23, B23)**

**# Print the tensors C1 and C23**

**print('\n C1: {}'.format(C1.numpy()))**

**print('\n C23: {}'.format(C23.numpy()))**

**# Define tensors A1 and A23 as constants**

**A1 = constant([1, 2, 3, 4])**

**A23 = constant([[1, 2, 3], [1, 6, 4]])**

**# Define B1 and B23 to have the correct shape**

**B1 = ones\_like(A1)**

**B23 = ones\_like(A23)**

**# Perform element-wise multiplication**

**C1 = multiply(A1, B1)**

**C23 = multiply(A23, B23)**

**# Print the tensors C1 and C23**

**print('\n C1: {}'.format(C1.numpy()))**

**print('\n C23: {}'.format(C23.numpy()))**

**C1: [1 2 3 4]**

**C23: [[1 2 3]**

**[1 6 4]]**

**Excellent work! Notice how performing element-wise multiplication with tensors of ones leaves the original tensors unchanged.**

## Exercise

# Making predictions with matrix multiplication

In later chapters, you will learn to train linear regression models. This process will yield a vector of parameters that can be multiplied by the input data to generate predictions. In this exercise, you will use input data, features, and a target vector, bill, which are taken from a credit card dataset we will use later in the course.

, ,

The matrix of input data, features, contains two columns: education level and age. The target vector, bill, is the size of the credit card borrower's bill.

Since we have not trained the model, you will enter a guess for the values of the parameter vector, params. You will then use matmul() to perform matrix multiplication of features by params to generate predictions, billpred, which you will compare with bill. Note that we have imported matmul() and constant().

## Instructions

* Define features, params, and bill as constants.
* Compute the predicted value vector, billpred, by multiplying the input data, features, by the parameters, params. Use matrix multiplication, rather than the element-wise product.
* Define error as the targets, bill, minus the predicted values, billpred.

# Define features, params, and bill as constants

features = \_\_\_\_([[2, 24], [2, 26], [2, 57], [1, 37]])

params = \_\_\_\_([[1000], [150]])

bill = \_\_\_\_([[3913], [2682], [8617], [64400]])

# Compute billpred using features and params

billpred = \_\_\_\_

# Compute and print the error

error = \_\_\_\_ - \_\_\_\_

print(error.numpy())

# Define features, params, and bill as constants

features = constant([[2, 24], [2, 26], [2, 57], [1, 37]])

params = constant([[1000], [150]])

bill = constant([[3913], [2682], [8617], [64400]])

# Compute billpred using features and params

billpred = matmul(features, params)

# Compute and print the error

error = bill - billpred

print(error.numpy())

# Define features, params, and bill as constants

features = constant([[2, 24], [2, 26], [2, 57], [1, 37]])

params = constant([[1000], [150]])

bill = constant([[3913], [2682], [8617], [64400]])

# Compute billpred using features and params

billpred = matmul(features, params)

# Compute and print the error

error = bill - billpred

print(error.numpy())

[[-1687]

[-3218]

[-1933]

[57850]]

**Nice job! Understanding matrix multiplication will make things simpler when we start making predictions with linear models.**

## Exercise

# Summing over tensor dimensions

You've been given a matrix, wealth. This contains the value of bond and stock wealth for five individuals in thousands of dollars.

wealth =

The first column corresponds to bonds and the second corresponds to stocks. Each row gives the bond and stock wealth for a single individual. Use wealth, reduce\_sum(), and .numpy() to determine which statements are correct about wealth.

## Instructions

### Possible answers

The individual in the first row has the highest total weath (ie stocks +bonds)

Combined, the 5 individuals hold %50000 in stocks.

**Combined, the 5 individuals hold $50000 in bonds.**

The individual in the second row has the lowest total wealth (ie stocks + bonds)

**Excellent work! Understanding how to sum over tensor dimensions will be helpful when preparing datasets and training models**.

LECTURE 2

**1. Advanced operations**

00:00 - 00:11

In this video, we will cover a selection of advanced operations. Some will be used frequently in later chapters. Others will help you to gain intuition about complex machine learning routines.

**2. Overview of advanced operations**

00:11 - 00:27

We have already covered basic operations in TensorFlow, including add, multiply, matmul, and reduce sum. In this lesson, we will move on to more advanced operations, including gradient, reshape, and random.

**3. Overview of advanced operations**

00:27 - 00:43

The gradient operation, which we'll use in conjunction with gradient tape, computes the slope of a function at a point. The reshape operation changes the shape of a tensor. And the random module generates a tensor out of randomly-drawn values.

**4. Finding the optimum**

00:43 - 01:20

In many machine learning problems, you will need to find an optimum--that is, a minimum or maximum. You may, for instance, want to find the model parameters that minimize the loss function or maximize the objective function. Fortunately, we can do this by using the gradient operation, which tells us the slope of a function at a point. We start this process by passing points to the gradient operation until we find one where the gradient is zero. Next, we check if the gradient is increasing or decreasing at that point. If it is increasing, we have minimum. Otherwise, we have a maximum.

**5. Calculating the gradient**

01:20 - 01:34

The plot shows the function y equals x. Notice that the gradient--that is, the slope at a given point, is constant. If we increase x by 1 unit, y also increases by 1 unit.

**6. Calculating the gradient**

01:34 - 01:58

This is not true if we instead consider the function y equals x squared. When x is less than 0, y decreases when x increases. When x is greater than 0, y increases when x increases. Thus, the gradient is initially negative, but becomes positive for x larger than 0. This means that x equals 0 minimizes y.

**7. Gradients in TensorFlow**

01:58 - 03:01

Let's use TensorFlow to compute the gradient. We will start by defining a variable, x, which we initialize to minus one point zero. We will then define y to be x squared within an instance of gradient tape. Note that we apply the watch method to an instance of gradient tape and then pass the variable x. This will allow us to compute the rate of change of y with respect to x. Next, we compute the gradient of y with respect to x using the tape instance of gradient tape. Note that y is the first argument and x is the second. As written, the operation computes the slope of y at a point. Running the code and printing, we find that the slope is -2 at x equals -1, which means that y is initially decreasing in x, as we saw on the previous slide. Much of the differentiation you do in deep learning models will be handled by high level APIs; however, gradient tape remains an invaluable tool for building advanced and custom models.

**8. Images as tensors**

03:01 - 03:26

We'll next consider an operation that is particularly useful for image classification problems: reshaping. The grayscale image shown has a natural representation as a matrix with values between 0 and 255. While some algorithms exploit this shape, others require you to reshape matrices into vectors before using them as inputs, as shown in the diagram.

**9. How to reshape a grayscale image**

03:26 - 03:50

Now that you've seen how images can be represented as tensors, let's generate some input images and reshape them. We will create a random grayscale image by drawing numbers from the set of integers between 0 and 255. We will use these to populate a 2 by 2 matrix. We can then reshape this into a 4 by 1 vector, as shown in the diagram.

**10. How to reshape a color image**

03:50 - 04:00

For color images, we will generate 3 such matrices to form a 2 by 2 by 3 tensor. We could then reshape the image into a 4 by 3 tensor, as shown in the diagram.

**11. Let's practice!**

04:00 - 04:06

It's time to put what you've learned to work in some exercises.

## Exercise

# Reshaping tensors

Later in the course, you will classify images of sign language letters using a neural network. In some cases, the network will take 1-dimensional tensors as inputs, but your data will come in the form of images, which will either be either 2- or 3-dimensional tensors, depending on whether they are grayscale or color images.

The figure below shows grayscale and color images of the sign language letter A. The two images have been imported for you and converted to the numpy arrays gray\_tensor and color\_tensor. Reshape these arrays into 1-dimensional vectors using the reshape operation, which has been imported for you from tensorflow. Note that the shape of gray\_tensor is 28x28 and the shape of color\_tensor is 28x28x3.



## Instructions

* Reshape gray\_tensor from a 28x28 matrix into a 784x1 vector named gray\_vector.
* Reshape color\_tensor from a 28x28x3 tensor into a 2352x1 vector named color\_vector.

# Reshape the grayscale image tensor into a vector

gray\_vector = reshape(\_\_\_\_, (\_\_\_\_, 1))

# Reshape the color image tensor into a vector

color\_vector = reshape(\_\_\_\_, (\_\_\_\_, \_\_\_\_))

# Reshape the grayscale image tensor into a vector

gray\_vector = reshape(gray\_tensor, (784, 1))

# Reshape the color image tensor into a vector

color\_vector = reshape(color\_tensor, (2352, 1))

# Reshape the grayscale image tensor into a vector gray\_vector = reshape(gray\_tensor, (784, 1)) # Reshape the color image tensor into a vector color\_vector = reshape(color\_tensor, (2352, 1))

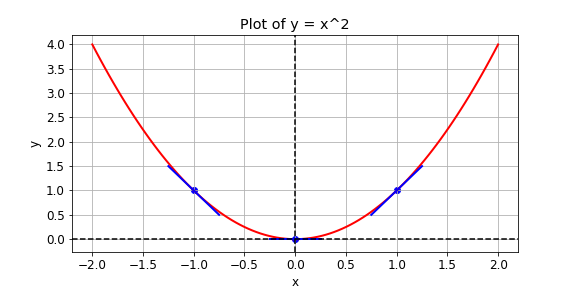
**Excellent work! Notice that there are 3 times as many elements in color\_vector as there are in gray\_vector, since color\_tensor has 3 color channels.**

## Exercise

# Optimizing with gradients

You are given a loss function,

, which you want to minimize. You can do this by computing the slope using the GradientTape() operation at different values of x. If the slope is positive, you can decrease the loss by lowering x. If it is negative, you can decrease it by increasing x. This is how gradient descent works.



In practice, you will use a high level tensorflow operation to perform gradient descent automatically. In this exercise, however, you will compute the slope at x values of -1, 1, and 0. The following operations are available: GradientTape(), multiply(), and Variable().

## Instructions

* Define x as a variable with the initial value x0.
* Set the loss function, y, equal to x multiplied by x. Do not make use of operator overloading.
* Set the function to return the gradient of y with respect to x.

def compute\_gradient(x0):

    # Define x as a variable with an initial value of x0

    x = \_\_\_\_(x0)

    with GradientTape() as tape:

        tape.watch(x)

        # Define y using the multiply operation

        y = \_\_\_\_

    # Return the gradient of y with respect to x

    return tape.gradient(\_\_\_\_, \_\_\_\_).numpy()

# Compute and print gradients at x = -1, 1, and 0

print(compute\_gradient(-1.0))

print(compute\_gradient(1.0))

print(compute\_gradient(0.0))

**def compute\_gradient(x0):**

**# Define x as a variable with an initial value of x0**

**x = Variable(x0)**

**with GradientTape() as tape:**

**tape.watch(x)**

**# Define y using the multiply operation**

**y = multiply (x, x)**

**# Return the gradient of y with respect to x**

**return tape.gradient(y, x).numpy()**

**# Compute and print gradients at x = -1, 1, and 0**

**print(compute\_gradient(-1.0))**

**print(compute\_gradient(1.0))**

**print(compute\_gradient(0.0))**

**def compute\_gradient(x0):**

**# Define x as a variable with an initial value of x0**

**x = Variable(x0)**

**with GradientTape() as tape:**

**tape.watch(x)**

**# Define y using the multiply operation**

**y = multiply (x, x)**

**# Return the gradient of y with respect to x**

**return tape.gradient(y, x).numpy()**

**# Compute and print gradients at x = -1, 1, and 0**

**print(compute\_gradient(-1.0))**

**print(compute\_gradient(1.0))**

**print(compute\_gradient(0.0))**

**-2.0**

**2.0**

**0.0**

**Excellent work! Notice that the slope is positive at x = 1, which means that we can lower the loss by reducing x. The slope is negative at x = -1, which means that we can lower the loss by increasing x. The slope at x = 0 is 0, which means that we cannot lower the loss by either increasing or decreasing x. This is because the loss is minimized at x = 0.**

## Exercise

# Working with image data

You are given a black-and-white image of a letter, which has been encoded as a tensor, letter. You want to determine whether the letter is an X or a K. You don't have a trained neural network, but you do have a simple model, model, which can be used to classify letter.

The 3x3 tensor, letter, and the 1x3 tensor, model, are available in the Python shell. You can determine whether letter is a K by multiplying letter by model, summing over the result, and then checking if it is equal to 1. As with more complicated models, such as neural networks, model is a collection of weights, arranged in a tensor.

Note that the functions reshape(), matmul(), and reduce\_sum() have been imported from tensorflow and are available for use.

## Instructions

* The model, model, is 1x3 tensor, but should be a 3x1. Reshape model.
* Perform a matrix multiplication of the 3x3 tensor, letter, by the 3x1 tensor, model.
* Sum over the resulting tensor, output, and assign this value to prediction.
* Print prediction using the .numpy() method to determine whether letter is K.

# Reshape model from a 1x3 to a 3x1 tensor

model = \_\_\_\_(model, (\_\_\_\_, \_\_\_\_))

# Multiply letter by model

output = \_\_\_\_(letter, model)

# Sum over output and print prediction using the numpy method

prediction = \_\_\_\_

print(prediction.\_\_\_\_)

# Reshape model from a 1x3 to a 3x1 tensor

model = reshape(model, (3, 1))

# Multiply letter by model

output = matmul(letter, model)

# Sum over output and print prediction using the numpy method

prediction = reduce\_sum(output)

print(prediction.numpy())

**# Reshape model from a 1x3 to a 3x1 tensor**

**model = reshape(model, (3, 1))**

**# Multiply letter by model**

**output = matmul(letter, model)**

**# Sum over output and print prediction using the numpy method**

**prediction = reduce\_sum(output)**

**print(prediction.numpy())**

**1.0**

**Excellent work! Your model found that prediction=1.0 and correctly classified the letter as a K. In the coming chapters, you will use data to train a model, model, and then combine this with matrix multiplication, matmul(letter, model), as we have done here, to make predictions about the classes of objects**.

**LECTURE**

**1. Input data**

00:00 - 00:10

In the previous chapter, we learned how to perform core TensorFlow operations. In this chapter, we will work towards training a linear model with TensorFlow.

**2. Using data in TensorFlow**

00:10 - 00:32

So far, we've only generated data using functions like ones and random uniform; however, when we train a machine learning model, we will want to import data from an external source. This may include numeric, image, or text data. Beyond simply importing the data, numeric data will need to be assigned a type, and text and image data will need to be converted to a usable format.

**3. Importing data for use in TensorFlow**

00:32 - 00:56

External datasets can be imported using TensorFlow. While this is useful for complex data pipelines, it will be unnecessarily complicated for what we do in this chapter. For that reason, we will use simpler options to import data. We will then convert the data into an NumPy array, which we can use without further modification in TensorFlow.

**4. How to import and convert data**

00:56 - 01:32

Let's start by importing numpy under the alias np and pandas under the alias pd. We will then read housing transaction data from kc\_housing.csv using the pandas method read csv and assign it to a dataframe called housing. When you are ready to train a model, you will want to convert the data into a numpy array by passing the pandas dataframe, housing, to np array. We will focus on loading data from csv files in this chapter, but you can also use pandas to load data from other formats, such as json, html, and excel.

**5. Parameters of read\_csv()**

01:32 - 02:18

Let's take a closer look at the read csv method of pandas, since you will use it frequently to import data. In the code block, we filled in the only required parameter, which was the filepath or buffer. Note that you could have instead supplied a URL, rather than a filepath to load your data. Another important parameter is sep, which is the delimiter that separates columns in your dataset. By default, this will be a comma; however, other common choices are semi-colons and tabs. Note that if you do use whitespace as a delimiter, you will need to set the delim whitespace parameter to true. Finally, if you are working with datasets that contain non-ASCII characters, you can specify the appropriate choice of encoding, so that your characters are correctly parsed.

**6. Using mixed type datasets**

02:18 - 02:48

Finally, we will end this lesson by talking about how to transform imported data for use in TensorFlow. We will use housing data from King County, Washington as an example. Notice how the dataset contains columns with different types. One column contains data on house prices in a floating point format. Another column is a boolean variable, which can either be true, 1, or false, 0. In this case, a 1 indicates that a property is located on the waterfront.

**7. Setting the data type**

02:48 - 03:12

Let's say we want to perform TensorFlow operations that require price to be a 32-bit floating point number and waterfront to be a boolean. We can do this in two ways. The first approach uses the array method from numpy. We select the relevant column in the DataFrame, provide it as the first argument to array, and then provide the datatype as the second argument.

**8. Setting the data type**

03:12 - 03:30

The second approach uses the cast operation from TensorFlow. Again, we supply the data first and the data type second. While either tf cast or np array will work, waterfront will be a tf dot Tensor type under the former option and a numpy array under the latter.

**9. Let's practice!**

03:30 - 03:37

You now know how to load data and set its data type. Let's put that to use in some exercises!

## Exercise

# Load data using pandas

Before you can train a machine learning model, you must first import data. There are several valid ways to do this, but for now, we will use a simple one-liner from pandas: pd.read\_csv(). Recall from the video that the first argument specifies the path or URL. All other arguments are optional.

In this exercise, you will import the King County housing dataset, which we will use to train a linear model later in the chapter.

## Instructions

* Import pandas under the alias pd.
* Assign the path to a string variable with the name data\_path.
* Load the dataset as a pandas dataframe named housing.
* Print the price column of housing.

# Import pandas under the alias pd

\_\_\_\_

# Assign the path to a string variable named data\_path

\_\_\_\_ = 'kc\_house\_data.csv'

# Load the dataset as a dataframe named housing

\_\_\_\_ = pd.read\_csv(\_\_\_\_)

# Print the price column of housing

print(\_\_\_\_)

# Import pandas under the alias pd

import pandas as pd

# Assign the path to a string variable named data\_path

data\_path = 'kc\_house\_data.csv'

# Load the dataset as a dataframe named housing

housing = pd.read\_csv(data\_path)

# Print the price column of housing

print(housing['price'])

**# Import pandas under the alias pd**

**import pandas as pd**

**# Assign the path to a string variable named data\_path**

**data\_path = 'kc\_house\_data.csv'**

**# Load the dataset as a dataframe named housing**

**housing = pd.read\_csv(data\_path)**

**# Print the price column of housing**

**print(housing['price'])**

**0 221900.0**

**1 538000.0**

**2 180000.0**

**3 604000.0**

**4 510000.0**

**...**

**21608 360000.0**

**21609 400000.0**

**21610 402101.0**

**21611 400000.0**

**21612 325000.0**

**Name: price, Length: 21613, dtype: float64**

**Excellent work! Notice that you did not have to specify a delimiter with the sep parameter, since the dataset was stored in the default, comma-separated format.**

## Exercise

# Setting the data type

In this exercise, you will both load data and set its type. Note that housing is available and pandas has been imported as pd. You will import numpy and tensorflow, and define tensors that are usable in tensorflow using columns in housing with a given data type. Recall that you can select the price column, for instance, from housing using housing['price'].

## Instructions

* Import numpy and tensorflow under their standard aliases.
* Use a numpy array to set the tensor price to have a data type of 32-bit floating point number
* Use the tensorflow function cast() to set the tensor waterfront to have a Boolean data type.
* Print price and then waterfront. Did you notice any important differences?

# Import numpy and tensorflow with their standard aliases

\_\_\_\_

\_\_\_\_

# Use a numpy array to define price as a 32-bit float

price = np.\_\_\_\_(housing['price'], np.\_\_\_\_)

# Define waterfront as a Boolean using cast

waterfront = tf.\_\_\_\_(housing['waterfront'], tf.\_\_\_\_)

# Print price and waterfront

print(\_\_\_\_)

print(\_\_\_\_)

# Import numpy and tensorflow with their standard aliases

import numpy as np

import tensorflow as tf

# Use a numpy array to define price as a 32-bit float

price = np.array(housing['price'], np.float32)

# Define waterfront as a Boolean using cast

waterfront = tf.cast(housing['waterfront'], tf.bool)

# Print price and waterfront

print(price)

print(waterfront)

**# Import numpy and tensorflow with their standard aliases**

**import numpy as np**

**import tensorflow as tf**

**# Use a numpy array to define price as a 32-bit float**

**price = np.array(housing['price'], np.float32)**

**# Define waterfront as a Boolean using cast**

**waterfront = tf.cast(housing['waterfront'], tf.bool)**

**# Print price and waterfront**

**print(price)**

**print(waterfront)**

**[221900. 538000. 180000. ... 402101. 400000. 325000.]**

**tf.Tensor([False False False ... False False False], shape=(21613,), dtype=bool)**

**Great job! Notice that printing price yielded a numpy array; whereas printing waterfront yielded a tf.Tensor().**

**LECTURE**

**1. Loss functions**

00:00 - 00:12

We now know how to import datasets and perform TensorFlow operations on them, but how can we use this knowledge to train models? In this video, we'll move closer to that goal by taking a look at loss functions.

**2. Introduction to loss functions**

00:12 - 00:51

Loss functions play a fundamental role in machine learning. We need loss functions to train models because they tell us how well our model explains the data. Without this feedback, it is unclear how to adjust model parameters during the training process. A high loss value indicates that the model fit is poor. Typically, we train the model by selecting parameter values that minimize the loss function. In some cases, we may want to maximize a function instead. Fortunately, we can always place a minus sign before the function we want to maximize and instead minimize it. For this reason, we will always talk about loss functions and minimization.

**3. Common loss functions in TensorFlow**

00:51 - 01:13

While it is possible to define a custom loss function, this is typically not necessary, since many common options are available in TensorFlow. Typical choices for training linear models include the mean squared error loss, the mean absolute error loss, and the Huber loss. All of these are accessible from tf dot keras dot losses.

**4. Why do we care about loss functions?**

01:13 - 01:52

Here, we plot the MSE, MAE, and Huber loss for error values between minus two and two. Note that the MSE strongly penalizes outliers and has high sensitivity near the minimum. The MAE scales linearly with the size of the error and has low sensitivity near the minimum. And the Huber loss is similar to the MSE near zero and similar to the MAE away from zero. For greater sensitivity near the minimum, you will want to use the MSE or Huber loss. To minimize the impact of outliers, you will want to use the MAE or Huber loss.

**5. Defining a loss function**

01:52 - 02:13

Let's say we decide to use the MSE loss. We'll need two tensors to compute it: the actual values or "targets" tensor and the predicted values or "predictions." Passing them to the MSE operation will return a single number: the average of the squared differences between the actual and predicted values.

**6. Defining a loss function**

02:13 - 03:08

In many cases, the training process will require us to supply a function that accepts our model's variables and data and returns a loss. Here, we'll first define a model, "linear\_regression," which takes the intercept, slope, and features as arguments and returns the model's predictions. We'll next define a loss function called "loss\_function" that accepts the slope and intercept of a linear model -- the variables -- and the input data, the targets and the features. It then makes a prediction and computes and returns the associated MSE loss. Note that we've defined both functions to use default argument values for features and targets. We will do this whenever we train on the full sample to simplify the code.

**7. Defining the loss function**

03:08 - 03:33

Notice that we've nested TensorFlow's MSE loss function within a function that first uses the model to make predictions and then uses those predictions as an input to the MSE loss function. We can then evaluate this function for a given set of parameter values and input data. Here, we've evaluated the loss function using a test dataset and it returned a loss value of ten point seven seven. If we had omitted the data arguments, test\_targets and test\_features, the loss function would have instead used the default targets and features arguments we set to evaluate model performance.

**8. Let's practice!**

03:33 - 03:39

It's now time to put what you've learned to work in some exercises.

## Exercise

# Loss functions in TensorFlow

In this exercise, you will compute the loss using data from the King County housing dataset. You are given a target, price, which is a tensor of house prices, and predictions, which is a tensor of predicted house prices. You will evaluate the loss function and print out the value of the loss.

## Instructions 1/2

Import the keras module from tensorflow. Then, use price and predictions to compute the mean squared error (mse).

Modify your code to compute the mean absolute error (mae), rather than the mean squared error (mse).

# Import the keras module from tensorflow

\_\_\_\_

# Compute the mean squared error (mse)

loss = keras.losses.\_\_\_\_(price, predictions)

# Print the mean squared error (mse)

print(loss.numpy())

# Import the keras module from tensorflow

from tensorflow import keras

# Compute the mean squared error (mse)

loss = keras.losses.mse(price, predictions)

# Print the mean squared error (mse)

print(loss.numpy())

**# Compute the mean squared error (mse)**

**loss = keras.losses.mse(price, predictions)**

**# Print the mean squared error (mse)**

**print(loss.numpy())**

**141171604777.12717**

# Import the keras module from tensorflow

from tensorflow import keras

# Compute the mean absolute error (mae)

loss = keras.losses.mse(price, predictions)

# Print the mean absolute error (mae)

print(loss.numpy())

**# Import the keras module from tensorflow**

**from tensorflow import keras**

**# Compute the mean absolute error (mae)**

**loss = keras.losses.mae(price, predictions)**

**# Print the mean absolute error (mae)**

**print(loss.numpy())**

**# Import the keras module from tensorflow**

**from tensorflow import keras**

**# Compute the mean absolute error (mae)**

**loss = keras.losses.mae(price, predictions)**

**# Print the mean absolute error (mae)**

**print(loss.numpy())**

**268827.99302088**

Great work! You may have noticed that the MAE was much smaller than the MSE, even though price **and predictions were the same. This is because the different loss functions penalize deviations of predictions from price differently. MSE does not like large deviations and punishes them harshly.**

## Exercise

# Modifying the loss function

In the previous exercise, you defined a tensorflow loss function and then evaluated it once for a set of actual and predicted values. In this exercise, you will compute the loss within another function called loss\_function(), which first generates predicted values from the data and variables. The purpose of this is to construct a function of the trainable model variables that returns the loss. You can then repeatedly evaluate this function for different variable values until you find the minimum. In practice, you will pass this function to an optimizer in tensorflow. Note that features and targets have been defined and are available. Additionally, Variable, float32, and keras are available.

## Instructions

* Define a variable, scalar, with an initial value of 1.0 and a type of float32.
* Define a function called loss\_function(), which takes scalar, features, and targets as arguments in that order.
* Use a mean absolute error loss function.

# Initialize a variable named scalar

scalar = \_\_\_\_(1.0, \_\_\_\_)

# Define the model

def model(scalar, features = features):

    return scalar \* features

# Define a loss function

def loss\_function(\_\_\_\_, features = features, targets = targets):

    # Compute the predicted values

    predictions = model(scalar, features)

    # Return the mean absolute error loss

    return keras.losses.\_\_\_\_(targets, predictions)

# Evaluate the loss function and print the loss

print(loss\_function(scalar).numpy())

# Initialize a variable named scalar

scalar = Variable(1.0, float32)

# Define the model

def model(scalar, features = features):

    return scalar \* features

# Define a loss function

def loss\_function(scalar, features = features, targets = targets):

    # Compute the predicted values

    predictions = model(scalar, features)

    # Return the mean absolute error loss

    return keras.losses.mae(targets, predictions)

# Evaluate the loss function and print the loss

print(loss\_function(scalar).numpy())

**# Initialize a variable named scalar**

**scalar = Variable(1.0, float32)**

**# Define the model**

**def model(scalar, features = features):**

**return scalar \* features**

**# Define a loss function**

**def loss\_function(scalar, features = features, targets = targets):**

**# Compute the predicted values**

**predictions = model(scalar, features)**

**# Return the mean absolute error loss**

**return keras.losses.mae(targets, predictions)**

**# Evaluate the loss function and print the loss**

**print(loss\_function(scalar).numpy())**

**3.0**

**Great work! As you will see in the following lessons, this exercise was the equivalent of evaluating the loss function for a linear regression where the intercept is 0.**

**LECTURE**

**1. Linear regression**

00:00 - 00:10

Now that you understand how to construct loss functions, you're well-equipped to start training models. We'll do that for the first time in this video with a linear regression model.

**2. What is a linear regression?**

00:10 - 00:44

So what is a linear regression model? We can answer this with a simple illustration. Let's say we want to examine the relationship between house size and price in the King County housing dataset. We might start by plotting the size in square feet against the price in dollars. Note that we've actually plotted the relationship after taking the natural logarithm of each variable, which is useful when we suspect that the relationship is proportional. That is, we might expect an x% increase in size to be associated with a y% increase in price.

**3. What is a linear regression?**

00:44 - 00:57

A linear regression model assumes that the relationship between these variables can be captured by a line. That is, two parameters--the line's slope and intercept--fully characterize the relationship between size and price.

**4. The linear regression model**

00:57 - 01:34

In our case, we've assumed that the relationship is linear after taking natural logarithms. Training the model will involve recovering the slope of the line and the intercept, where the line intersects the vertical axis. Once we have trained the intercept and slope, we can take a house's size and predict its price. The difference between the predicted price and actual price is the error, which can be used to construct a loss function. The example we've shown is for a univariate regression, which has only one feature, size. A multiple regression has multiple features, such as size and location.

**5. Linear regression in TensorFlow**

01:34 - 02:21

Let's look at some code to see how this can be implemented. We will first define our target variable, price, and feature, size. We also initialize the intercept and slope as trainable variables. After that, we define the model, which we'll use to make predictions by multiplying size and slope and then adding the intercept. Again, remember that we can do this using the addition and multiplication symbols, since these are overloaded operators and intercept and slope are tensorflow operations. Our next step is to define a loss function. This function will take the model's parameters and the data as an input. It will first use the model to compute the predicted values. We then set the function to return the mean squared error loss. We, of course, could have selected a different loss.

**6. Linear regression in TensorFlow**

02:21 - 03:24

With the loss function defined, the next step is to define an optimization operation. We'll do this using the adam optimizer. For now, you can ignore the choice of optimization algorithm. We will discuss the selection of optimizers in greater detail later. For our purposes, it is sufficient to understand that executing this operation will change the slope and intercept in a direction that will lower the value of the loss. We will next perform minimization on the loss function using the optimizer. Notice that we've passed the loss function as a lambda function to the minimize operation. We also supplied a variable list, which contains intercept and slope, the two variables we defined earlier. We will execute our optimization step 1000 times. Printing the loss, we'll see that it tends to decline, moving closer to the minimum value with each step. Finally, we print the intercept and the slope. This is our linear model, which enables us to predict the value of a house given its size.

**7. Let's practice!**

03:24 - 03:32

You now have all of the tools you'll need to train a linear model in TensorFlow, so let's try that in an exercise!

## Exercise

# Set up a linear regression

A univariate linear regression identifies the relationship between a single feature and the target tensor. In this exercise, we will use a property's lot size and price. Just as we discussed in the video, we will take the natural logarithms of both tensors, which are available as price\_log and size\_log.

In this exercise, you will define the model and the loss function. You will then evaluate the loss function for two different values of intercept and slope. Remember that the predicted values are given by intercept + features\*slope. Additionally, note that keras.losses.mse() is available for you. Furthermore, slope and intercept have been defined as variables.

## Instructions

* Define a function that returns the predicted values for a linear regression using intercept, features, and slope, and without using add() or multiply().
* Complete the loss\_function() by adding the model's variables, intercept and slope, as arguments.
* Compute the mean squared error using targets and predictions.

# Define a linear regression model

def linear\_regression(intercept, slope, features = size\_log):

    return \_\_\_\_

# Set loss\_function() to take the variables as arguments

def loss\_function(\_\_\_\_, \_\_\_\_, features = size\_log, targets = price\_log):

    # Set the predicted values

    predictions = linear\_regression(intercept, slope, features)

    # Return the mean squared error loss

    return keras.losses.\_\_\_\_

# Compute the loss for different slope and intercept values

print(loss\_function(0.1, 0.1).numpy())

print(loss\_function(0.1, 0.5).numpy())

# Define a linear regression model

def linear\_regression(intercept, slope, features = size\_log):

    return intercept + features\*slope

# Set loss\_function() to take the variables as arguments

def loss\_function(intercept, slope, features = size\_log, targets = price\_log):

    # Set the predicted values

    predictions = linear\_regression(intercept, slope, features)

    # Return the mean squared error loss

    return keras.losses.mse(targets, predictions)

# Compute the loss for different slope and intercept values

print(loss\_function(0.1, 0.1).numpy())

print(loss\_function(0.1, 0.5).numpy())

**# Define a linear regression model**

**def linear\_regression(intercept, slope, features = size\_log):**

**return intercept + features\*slope**

**# Set loss\_function() to take the variables as arguments**

**def loss\_function(intercept, slope, features = size\_log, targets = price\_log):**

**# Set the predicted values**

**predictions = linear\_regression(intercept, slope, features)**

**# Return the mean squared error loss**

**return keras.losses.mse(targets, predictions)**

**# Compute the loss for different slope and intercept values**

**print(loss\_function(0.1, 0.1).numpy())**

**print(loss\_function(0.1, 0.5).numpy())**

**145.44653**

**71.866**

**Great work! In the next exercise, you will actually run the regression and train intercept and slope.**

## Exercise

# Train a linear model

In this exercise, we will pick up where the previous exercise ended. The intercept and slope, intercept and slope, have been defined and initialized. Additionally, a function has been defined, loss\_function(intercept, slope), which computes the loss using the data and model variables.

You will now define an optimization operation as opt. You will then train a univariate linear model by minimizing the loss to find the optimal values of intercept and slope. Note that the opt operation will try to move closer to the optimum with each step, but will require many steps to find it. Thus, you must repeatedly execute the operation.

## Instructions

* Initialize an Adam optimizer as opt with a learning rate of 0.5.
* Apply the .minimize() method to the optimizer.
* Pass loss\_function() with the appropriate arguments as a lambda function to .minimize().
* Supply the list of variables that need to be updated to var\_list

# Initialize an Adam optimizer

opt = keras.optimizers.\_\_\_\_(0.5)

for j in range(100):

    # Apply minimize, pass the loss function, and supply the variables

    opt.\_\_\_\_(lambda: \_\_\_\_(\_\_\_\_, \_\_\_\_), var\_list=[\_\_\_\_, \_\_\_\_])

    # Print every 10th value of the loss

    if j % 10 == 0:

        print(loss\_function(intercept, slope).numpy())

# Plot data and regression line

plot\_results(intercept, slope)

# Initialize an Adam optimizer

opt = keras.optimizers.Adam(0.5)

for j in range(100):

    # Apply minimize, pass the loss function, and supply the variables

    opt.minimize(lambda: loss\_function(intercept, slope), var\_list=[intercept, slope])

    # Print every 10th value of the loss

    if j % 10 == 0:

        print(loss\_function(intercept, slope).numpy())

# Plot data and regression line

plot\_results(intercept, slope)

**# Initialize an Adam optimizer**

**opt = keras.optimizers.Adam(0.5)**

**for j in range(100):**

**# Apply minimize, pass the loss function, and supply the variables**

**opt.minimize(lambda: loss\_function(intercept, slope), var\_list=[intercept, slope])**

**# Print every 10th value of the loss**

**if j % 10 == 0:**

**print(loss\_function(intercept, slope).numpy())**

**# Plot data and regression line**

**plot\_results(intercept, slope)**

**9.669482**

**11.726698**

**1.1193314**

**1.6605737**

**0.7982884**

**0.8017316**

**0.6106565**

**0.59997976**

**0.5811015**

**0.5576158**

**Excellent! Notice that we printed loss\_function(intercept, slope) every 10th execution for 100 executions. Each time, the loss got closer to the minimum as the optimizer moved the slope and intercept parameters closer to their optimal values.**

## Exercise

# Multiple linear regression

In most cases, performing a univariate linear regression will not yield a model that is useful for making accurate predictions. In this exercise, you will perform a multiple regression, which uses more than one feature.

You will use price\_log as your target and size\_log and bedrooms as your features. Each of these tensors has been defined and is available. You will also switch from using the the mean squared error loss to the mean absolute error loss: keras.losses.mae(). Finally, the predicted values are computed as follows: params[0] + feature1\*params[1] + feature2\*params[2]. Note that we've defined a vector of parameters, params, as a variable, rather than using three variables. Here, params[0] is the intercept and params[1] and params[2] are the slopes.

## Instructions

* Define a linear regression model that returns the predicted values.
* Set loss\_function() to take the parameter vector as an input.
* Use the mean absolute error loss.
* Complete the minimization operation.

# Define the linear regression model

def linear\_regression(params, feature1 = size\_log, feature2 = bedrooms):

    return params[0] + feature1\*\_\_\_\_ + feature2\*\_\_\_\_

# Define the loss function

def loss\_function(\_\_\_\_, targets = price\_log, feature1 = size\_log, feature2 = bedrooms):

    # Set the predicted values

    predictions = linear\_regression(params, feature1, feature2)

    # Use the mean absolute error loss

    return keras.losses.\_\_\_\_(targets, predictions)

# Define the optimize operation

opt = keras.optimizers.Adam()

# Perform minimization and print trainable variables

for j in range(10):

    opt.minimize(lambda: loss\_function(\_\_\_\_), var\_list=[\_\_\_\_])

    print\_results(params)

**# Define the linear regression model**

**def linear\_regression(params, feature1 = size\_log, feature2 = bedrooms):**

**return params[0] + feature1\*params[1] + feature2\*params[2]**

**# Define the loss function**

**def loss\_function(params, targets = price\_log, feature1 = size\_log, feature2 = bedrooms):**

**# Set the predicted values**

**predictions = linear\_regression(params, feature1, feature2)**

**# Use the mean absolute error loss**

**return keras.losses.mae(targets, predictions)**

**# Define the optimize operation**

**opt = keras.optimizers.Adam()**

**# Perform minimization and print trainable variables**

**for j in range(10):**

**opt.minimize(lambda: loss\_function(params), var\_list=[params])**

**print\_results(params)**

**# Define the linear regression model**

**def linear\_regression(params, feature1 = size\_log, feature2 = bedrooms):**

**return params[0] + feature1\*params[1] + feature2\*params[2]**

**# Define the loss function**

**def loss\_function(params, targets = price\_log, feature1 = size\_log, feature2 = bedrooms):**

**# Set the predicted values**

**predictions = linear\_regression(params, feature1, feature2)**

**# Use the mean absolute error loss**

**return keras.losses.mae(targets, predictions)**

**# Define the optimize operation**

**opt = keras.optimizers.Adam()**

**# Perform minimization and print trainable variables**

**for j in range(10):**

**opt.minimize(lambda: loss\_function(params), var\_list=[params])**

**print\_results(params)**

**loss: 12.418, intercept: 0.101, slope\_1: 0.051, slope\_2: 0.021**

**loss: 12.404, intercept: 0.102, slope\_1: 0.052, slope\_2: 0.022**

**loss: 12.391, intercept: 0.103, slope\_1: 0.053, slope\_2: 0.023**

**loss: 12.377, intercept: 0.104, slope\_1: 0.054, slope\_2: 0.024**

**loss: 12.364, intercept: 0.105, slope\_1: 0.055, slope\_2: 0.025**

**loss: 12.351, intercept: 0.106, slope\_1: 0.056, slope\_2: 0.026**

**loss: 12.337, intercept: 0.107, slope\_1: 0.057, slope\_2: 0.027**

**loss: 12.324, intercept: 0.108, slope\_1: 0.058, slope\_2: 0.028**

**loss: 12.311, intercept: 0.109, slope\_1: 0.059, slope\_2: 0.029**

**loss: 12.297, intercept: 0.110, slope\_1: 0.060, slope\_2: 0.030**

**Great job! Note that params[2] tells us how much the price will increase in percentage terms if we add one more bedroom. You could train params[2] and the other model parameters by increasing the number of times we iterate over opt.**

**LECTURE**

**1. Batch training**

00:00 - 00:09

Earlier in the chapter, we learned how to train a linear model to predict house prices. In this video, we will use batch training to handle large datasets.

**2. What is batch training?**

00:09 - 00:55

So what is batch training exactly? To answer this, let's return to the linear model you used to predict house prices earlier in the chapter. But this time, let's say the dataset is much larger and you want to perform the training on a GPU, which has only small amount of memory. Since you can't fit the entire dataset in memory, you will instead divide it into batches and train on those batches sequentially. A single pass over all of the batches is called an epoch and the process itself is called batch training. It will be quite useful when you work with large image datasets. Beyond alleviating memory constraints, batch training will also allow you to update model weights and optimizer parameters after each batch, rather than at the end of the epoch.

**3. The chunksize parameter**

00:55 - 01:37

Earlier, we discussed using pandas to load data with read csv. The same function can be used to load data in batches. If, for instance, we have a 100 gigabyte dataset, we might want to avoid loading it all at once. We can do this by using the chunksize parameter. The code block shows how this can be done. Let's first import pandas and numpy. Instead of loading the data in a single one-liner, we'll write a for loop that iterates through the data in steps of 100 examples. Each 100 will be available as batch, which we can use to extract columns, such as price and size in the housing dataset. We can then convert these to numpy arrays and use them to train.

**4. Training a linear model in batches**

01:37 - 02:06

We now know how to load data from csv files in fixed-size batches using pandas. This means that we can handle data sets of tens or even hundreds of gigabytes without exceeding the memory constraints of our system. Let's look at a minimal example with a linear model using the King County housing dataset. We will start by loading tensorflow, pandas, and numpy. Next, we'll define variables for the intercept and slope, along with the linear regression model.

**5. Training a linear model in batches**

02:06 - 02:23

We then define a loss function, which takes the slope and intercept, and two sources of data: the features and the targets. It then returns the mean squared error loss. After defining the loss function, we instantiate an adam optimizer, which we will use to perform minimization.

**6. Training a linear model in batches**

02:23 - 03:15

The next step is to train the model in batches. Again, we do this by using a for loop and supplying a chunksize to the read csv function. Note that we take each batch, separate it into features and a target, convert those into numpy arrays, and then pass them to the minimize operation. Within the minimize operation, we pass the loss function as a lambda function and we supply a variable list that contains only the trainable parameters, intercept and slope. This loop will continue until we have stepped through all of the examples in csv read. Importantly, we did not ever need to have more than 100 examples in memory during the entire process. Finally, we print our trained intercept and slope. Note that we did not use default argument values for input data. This is because our input data was generated in batches during the training process.

**7. Full sample versus batch training**

03:15 - 03:53

So what is the value of batch training? When we trained with the full sample, we updated the optimizer and model parameters once per training epoch and passed data to the loss function without modification, but were limited by memory constraints. With batch training, we updated the model weights and optimizer parameters multiple times each epoch and divided the data into batches, but no longer faced any memory constraints. In later chapters, you'll automate batch training by using high level APIs. Importantly, however, high level APIs will not typically load the sample in batches by default, as we have done here.

**8. Let's practice!**

03:53 - 04:00

That was a lot of information! Let's break this down into simple steps with some exercises.

## Exercise

# Preparing to batch train

Before we can train a linear model in batches, we must first define variables, a loss function, and an optimization operation. In this exercise, we will prepare to train a model that will predict price\_batch, a batch of house prices, using size\_batch, a batch of lot sizes in square feet. In contrast to the previous lesson, we will do this by loading batches of data using pandas, converting it to numpy arrays, and then using it to minimize the loss function in steps.

Variable(), keras(), and float32 have been imported for you. Note that you should not set default argument values for either the model or loss function, since we will generate the data in batches during the training process.

## Instructions

* Define intercept as having an initial value of 10.0 and a data type of 32-bit float.
* Define the model to return the predicted values using intercept, slope, and features.
* Define a function called loss\_function() that takes intercept, slope, targets, and features as arguments and in that order. Do not set default argument values.
* Define the mean squared error loss function using targets and predictions.

# Define the intercept and slope

intercept = \_\_\_

slope = Variable(0.5, float32)

# Define the model

def linear\_regression(intercept, slope, features):

    # Define the predicted values

    return \_\_\_\_

# Define the loss function

def \_\_\_\_:

    # Define the predicted values

    predictions = linear\_regression(\_\_\_\_, \_\_\_\_, features)

    # Define the MSE loss

    return keras.losses.\_\_\_\_(\_\_\_\_, \_\_\_\_)

# Define the intercept and slope

intercept = Variable(10.0, float32)

slope = Variable(0.5, float32)

# Define the model

def linear\_regression(intercept, slope, features):

    # Define the predicted values

    return intercept + slope\*features

# Define the loss function

def loss\_function(intercept, slope, targets, features):

    # Define the predicted values

    predictions = linear\_regression(intercept, slope, features)

    # Define the MSE loss

    return keras.losses.mse(targets, predictions)

**# Define the intercept and slope intercept = Variable(10.0, float32) slope = Variable(0.5, float32) # Define the model def linear\_regression(intercept, slope, features): # Define the predicted values return intercept + slope\*features # Define the loss function def loss\_function(intercept, slope, targets, features): # Define the predicted values predictions = linear\_regression(intercept, slope, features) # Define the MSE loss return keras.losses.mse(targets, predictions)**

**Excellent work! Notice that we did not use default argument values for the input data, features and targets. This is because the input data has not been defined in advance. Instead, with batch training, we will load it during the training process.**

## Exercise

# Training a linear model in batches

In this exercise, we will train a linear regression model in batches, starting where we left off in the previous exercise. We will do this by stepping through the dataset in batches and updating the model's variables, intercept and slope, after each step. This approach will allow us to train with datasets that are otherwise too large to hold in memory.

Note that the loss function,loss\_function(intercept, slope, targets, features), has been defined for you. Additionally, keras has been imported for you and numpy is available as np. The trainable variables should be entered into var\_list in the order in which they appear as loss function arguments.

## Instructions

* Use the .Adam() optimizer.
* Load in the data from 'kc\_house\_data.csv' in batches with a chunksize of 100.
* Extract the price column from batch, convert it to a numpy array of type 32-bit float, and assign it to price\_batch.
* Complete the loss function, fill in the list of trainable variables, and perform minimization.

# Initialize Adam optimizer

opt = keras.optimizers.\_\_\_\_

# Load data in batches

for batch in pd.read\_csv('\_\_\_\_', \_\_\_\_=\_\_\_\_):

    size\_batch = np.array(batch['sqft\_lot'], np.float32)

    # Extract the price values for the current batch

    price\_batch = np.array(batch['\_\_\_\_'], np.\_\_\_\_)

    # Complete the loss, fill in the variable list, and minimize

    opt.minimize(lambda: loss\_function(\_\_\_\_, slope, price\_batch, size\_batch), var\_list=[intercept, \_\_\_\_])

# Print trained parameters

print(intercept.numpy(), slope.numpy())

# Initialize Adam optimizer

opt = keras.optimizers.Adam()

# Load data in batches

for batch in pd.read\_csv('kc\_house\_data.csv', chunksize=100):

    size\_batch = np.array(batch['sqft\_lot'], np.float32)

    # Extract the price values for the current batch

    price\_batch = np.array(batch['price'], np.float32)

    # Complete the loss, fill in the variable list, and minimize

    opt.minimize(lambda: loss\_function(intercept, slope, price\_batch, size\_batch), var\_list=[intercept, slope])

# Print trained parameters

print(intercept.numpy(), slope.numpy())

**# Initialize Adam optimizer**

**opt = keras.optimizers.Adam()**

**# Load data in batches**

**for batch in pd.read\_csv('kc\_house\_data.csv', chunksize=100):**

**size\_batch = np.array(batch['sqft\_lot'], np.float32)**

**# Extract the price values for the current batch**

**price\_batch = np.array(batch['price'], np.float32)**

**# Complete the loss, fill in the variable list, and minimize**

**opt.minimize(lambda: loss\_function(intercept, slope, price\_batch, size\_batch), var\_list=[intercept, slope])**

**# Print trained parameters**

**print(intercept.numpy(), slope.numpy())**

**10.217888 0.7016**

**Great work! Batch training will be very useful when you train neural networks, which we will do next.**

**LECTURE**

**1. Dense layers**

00:00 - 00:11

In this chapter, we will focus on training neural networks in TensorFlow. We will start with an overview of a frequently used component of neural networks: the dense layer.

**2. The linear regression model**

00:11 - 00:37

Throughout this chapter, we'll make use of a dataset on credit card default. It contains features, such as marital status and payment amount, which we'll use to predict a target, default. Here, we have the familiar linear regression model. We take marital status, which is 1, and bill amount, which is 3. We then multiply the inputs by their respective weights, zero point one and minus zero point two five, and sum.

**3. What is a neural network?**

00:37 - 01:11

So how do we get from a linear regression to a neural network? By adding a hidden layer, which, in this case, consists of two nodes. Each hidden layer node takes our two inputs, multiplies them by their respective weights, and sums them together. We also typically pass the hidden layer output to an activation function, but we will come back to that later. Finally, we sum together the outputs of the two hidden layers to compute our prediction for default. This entire process of generating a prediction is referred to as forward propagation.

**4. What is a neural network?**

01:11 - 01:48

In this chapter, we will construct neural networks with three types of layers: an input layer, some number of hidden layers, and an output layer. The input layer consists of our features. The output layer contains our prediction. Each hidden layer takes inputs from the previous layer, applies numerical weights to them, sums them together, and then applies an activation function. In the neural network graph, we have applied a particular type of hidden layer called a dense layer. A dense layer applies weights to all nodes from the previous layer. We will use dense layers throughout this chapter to construct networks.

**5. A simple dense layer**

01:48 - 02:12

Let's look at a simple example of a dense layer. We'll first define a constant tensor that contains the marital status and age data as the input layer. We then initialize weights as a variable, since we will train those weights to predict the output from the inputs. We also define a bias, which will play a similar role to the intercept in the linear regression model.

**6. A simple dense layer**

02:12 - 02:49

Finally, we define a dense layer. Note that we first perform a matrix multiplication of the inputs by the weights and assign that to the tensor named product. We then add product to the bias and apply a non-linear transformation, in this case the sigmoid function. This is called the activation function and we will explore this in more depth in the next video, but do not worry about it for now. Furthermore, note that the bias is not associated with a feature and is analogous to the intercept in a linear regression. We will typically not draw it in neural network diagrams for simplicity.

**7. Defining a complete model**

02:49 - 03:23

Note that TensorFlow also comes with higher level operations, such as tf dot keras dot layers dot Dense, which allows us to skip the linear algebra. In this example, we take input data and convert it to a 32-bit float tensor. We then define a first hidden dense layer using keras layers dense. The first argument specifies the number of outgoing nodes. And the second argument is the activation function. By default, a bias will be included. Note that we've also passed inputs as an argument to the first dense layer.

**8. Defining a complete model**

03:23 - 03:34

We can easily define another dense layer, which takes the first dense layer as an argument and then reduces the number of nodes. The output layer reduces this again to one node.

**9. High-level versus low-level approach**

03:34 - 04:03

Finally, let's compare the high-level and low-level approaches. The high-level approach relies on complex operations in high-level APIs, such as Keras and Estimators, reducing the amount of code needed. The weights and the mathematical operations will typically be hidden by the layer constructor. The low-level approach uses linear algebra, which allows for the construction of any model. TensorFlow allows us to use either approach or even combine them.

**10. Let's practice!**

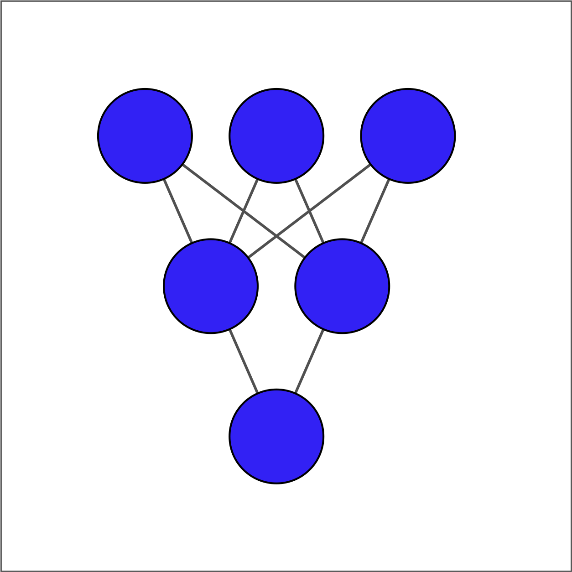
04:03 - 04:12

You now know how to construct dense layers using both the high and low-level approaches, so let's do that in some exercises!

## Exercise

# The linear algebra of dense layers

There are two ways to define a dense layer in tensorflow. The first involves the use of low-level, linear algebraic operations. The second makes use of high-level keras operations. In this exercise, we will use the first method to construct the network shown in the image below.



The input layer contains 3 features -- education, marital status, and age -- which are available as borrower\_features. The hidden layer contains 2 nodes and the output layer contains a single node.

For each layer, you will take the previous layer as an input, initialize a set of weights, compute the product of the inputs and weights, and then apply an activation function. Note that Variable(), ones(), matmul(), and keras() have been imported from tensorflow.

## Instructions 1/2

* Initialize weights1 as a variable using a 3x2 tensor of ones.
* Compute the product of borrower\_features by weights1 using matrix multiplication.
* Use a sigmoid activation function to transform product1 + bias1.

 Initialize bias1

bias1 = Variable(1.0)

# Initialize weights1 as 3x2 variable of ones

weights1 = \_\_\_\_(ones((\_\_\_\_, \_\_\_\_)))

# Perform matrix multiplication of borrower\_features and weights1

product1 = \_\_\_\_

# Apply sigmoid activation function to product1 + bias1

dense1 = keras.activations.\_\_\_\_(\_\_\_\_ + \_\_\_\_)

# Print shape of dense1

print("\n dense1's output shape: {}".format(dense1.shape))

# Initialize bias1

bias1 = Variable(1.0)

# Initialize weights1 as 3x2 variable of ones

weights1 = Variable(ones((3, 2)))

# Perform matrix multiplication of borrower\_features and weights1

product1 = matmul(borrower\_features, weights1)

# Apply sigmoid activation function to product1 + bias1

dense1 = keras.activations.sigmoid(product1 + bias1)

# Print shape of dense1

print("\n dense1's output shape: {}".format(dense1.shape))

**# Initialize bias1**

**bias1 = Variable(1.0)**

**# Initialize weights1 as 3x2 variable of ones**

**weights1 = Variable(ones((3, 2)))**

**# Perform matrix multiplication of borrower\_features and weights1**

**product1 = matmul(borrower\_features, weights1)**

**# Apply sigmoid activation function to product1 + bias1**

**dense1 = keras.activations.sigmoid(product1 + bias1)**

**# Print shape of dense1**

**print("\n dense1's output shape: {}".format(dense1.shape))**

**dense1's output shape: (1, 2)**

**Exercise 2/3**

* Initialize weights2 as a variable using a 2x1 tensor of ones.
* Compute the product of dense1 by weights2 using matrix multiplication.
* Use a sigmoid activation function to transform product2 + bias2.

# From previous step

bias1 = Variable(1.0)

weights1 = Variable(ones((3, 2)))

product1 = matmul(borrower\_features, weights1)

dense1 = keras.activations.sigmoid(product1 + bias1)

# Initialize bias2 and weights2

bias2 = Variable(1.0)

weights2 = \_\_\_\_(ones((\_\_\_\_, \_\_\_\_)))

# Perform matrix multiplication of dense1 and weights2

product2 = \_\_\_\_

# Apply activation to product2 + bias2 and print the prediction

prediction = keras.activations.\_\_\_\_(\_\_\_\_ + \_\_\_\_)

print('\n prediction: {}'.format(prediction.numpy()[0,0]))

print('\n actual: 1')

# From previous step

bias1 = Variable(1.0)

weights1 = Variable(ones((3, 2)))

product1 = matmul(borrower\_features, weights1)

dense1 = keras.activations.sigmoid(product1 + bias1)

# Initialize bias2 and weights2

bias2 = Variable(1.0)

weights2 = Variable(ones((2,1)))

# Perform matrix multiplication of dense1 and weights2

product2 = matmul(dense1, weights2)

# Apply activation to product2 + bias2 and print the prediction

prediction = keras.activations.sigmoid(product2 + bias2)

print('\n prediction: {}'.format(prediction.numpy()[0,0]))

print('\n actual: 1')

# Perform matrix multiplication of dense1 and weights2

product2 = matmul(dense1, weights2)

# Apply activation to product2 + bias2 and print the prediction

prediction = keras.activations.sigmoid(product2 + bias2)

print('\n prediction: {}'.format(prediction.numpy()[0,0]))

print('\n actual: 1')

prediction: 0.9525741338729858

actual: 1

**Excellent work! Our model produces predicted values in the interval between 0 and 1. For the example we considered, the actual value was 1 and the predicted value was a probability between 0 and 1. This, of course, is not meaningful, since we have not yet trained our model's parameters.**

## Exercise

# The low-level approach with multiple examples

In this exercise, we'll build further intuition for the low-level approach by constructing the first dense hidden layer for the case where we have multiple examples. We'll assume the model is trained and the first layer weights, weights1, and bias, bias1, are available. We'll then perform matrix multiplication of the borrower\_features tensor by the weights1 variable. Recall that the borrower\_features tensor includes education, marital status, and age. Finally, we'll apply the sigmoid function to the elements of products1 + bias1, yielding dense1.

Note that matmul() and keras() have been imported from tensorflow.

## Instructions

* Compute products1 by matrix multiplying the features tensor by the weights.
* Use a sigmoid activation function to transform products1 + bias1.
* Print the shapes of borrower\_features, weights1, bias1, and dense1.

# Compute the product of borrower\_features and weights1

products1 = \_\_\_\_

# Apply a sigmoid activation function to products1 + bias1

dense1 = \_\_\_\_

# Print the shapes of borrower\_features, weights1, bias1, and dense1

print('\n shape of borrower\_features: ', borrower\_features.shape)

print('\n shape of weights1: ', \_\_\_\_.shape)

print('\n shape of bias1: ', \_\_\_\_.shape)

print('\n shape of dense1: ', \_\_\_\_.shape)

# Compute the product of borrower\_features and weights1

products1 = matmul(borrower\_features, weights1)

# Apply a sigmoid activation function to products1 + bias1

dense1 = keras.activations.sigmoid(products1+bias1)

# Print the shapes of borrower\_features, weights1, bias1, and dense1

print('\n shape of borrower\_features: ', borrower\_features.shape)

print('\n shape of weights1: ', weights1.shape)

print('\n shape of bias1: ', bias1.shape)

print('\n shape of dense1: ', dense1.shape)

**# Compute the product of borrower\_features and weights1**

**products1 = matmul(borrower\_features, weights1)**

**# Apply a sigmoid activation function to products1 + bias1**

**dense1 = keras.activations.sigmoid(products1+bias1)**

**# Print the shapes of borrower\_features, weights1, bias1, and dense1**

**print('\n shape of borrower\_features: ', borrower\_features.shape)**

**print('\n shape of weights1: ', weights1.shape)**

**print('\n shape of bias1: ', bias1.shape)**

**print('\n shape of dense1: ', dense1.shape)**

**shape of borrower\_features: (5, 3)**

**shape of weights1: (3, 2)**

**shape of bias1: (1,)**

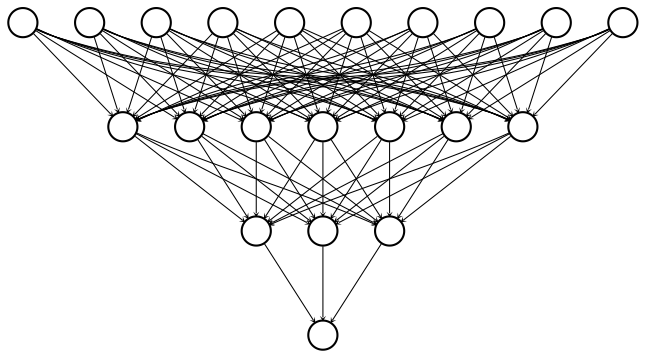
**shape of dense1: (5, 2)**

**Good job! Note that our input data**, borrower\_features, is 5x3 **because it consists of 5 examples for 3 features. The shape of** weights1 is 3x2**, as it was in the previous exercise, since it does not depend on the number of examples. Additionally**, bias1 is a scalar**. Finally,** dense1 is 5x2**, which means that we can multiply it by the following set of weights,** weights2**, which we defined to be** 2x1 **in the previous exercise.**

## Exercise

# Using the dense layer operation

We've now seen how to define dense layers in tensorflow using linear algebra. In this exercise, we'll skip the linear algebra and let keras work out the details. This will allow us to construct the network below, which has 2 hidden layers and 10 features, using less code than we needed for the network with 1 hidden layer and 3 features.



To construct this network, we'll need to define three dense layers, each of which takes the previous layer as an input, multiplies it by weights, and applies an activation function. Note that input data has been defined and is available as a 100x10 tensor: borrower\_features. Additionally, the keras.layers module is available.

## Instructions

* Set dense1 to be a dense layer with 7 output nodes and a sigmoid activation function.
* Define dense2 to be dense layer with 3 output nodes and a sigmoid activation function.
* Define predictions to be a dense layer with 1 output node and a sigmoid activation function.
* Print the shapes of dense1, dense2, and predictions in that order using the .shape method. Why does each of these tensors have 100 rows?

# Define the first dense layer

dense1 = keras.layers.Dense(\_\_\_\_, activation='\_\_\_\_')(borrower\_features)

# Define a dense layer with 3 output nodes

dense2 = \_\_\_\_

# Define a dense layer with 1 output node

predictions = \_\_\_\_

# Print the shapes of dense1, dense2, and predictions

print('\n shape of dense1: ', dense1.shape)

print('\n shape of dense2: ', \_\_\_\_.shape)

print('\n shape of predictions: ', \_\_\_\_.shape)

# Define the first dense layer

dense1 = keras.layers.Dense(7, activation='sigmoid')(borrower\_features)

# Define a dense layer with 3 output nodes

dense2 = keras.layers.Dense(3, activation='sigmoid')(dense1)

# Define a dense layer with 1 output node

predictions = keras.layers.Dense(1, activation='sigmoid')(dense2)

# Print the shapes of dense1, dense2, and predictions

print('\n shape of dense1: ', dense1.shape)

print('\n shape of dense2: ', dense2.shape)

print('\n shape of predictions: ', predictions.shape)

**# Define the first dense layer**

**dense1 = keras.layers.Dense(7, activation='sigmoid')(borrower\_features)**

**# Define a dense layer with 3 output nodes**

**dense2 = keras.layers.Dense(3, activation='sigmoid')(dense1)**

**# Define a dense layer with 1 output node**

**predictions = keras.layers.Dense(1, activation='sigmoid')(dense2)**

**# Print the shapes of dense1, dense2, and predictions**

**print('\n shape of dense1: ', dense1.shape)**

**print('\n shape of dense2: ', dense2.shape)**

**print('\n shape of predictions: ', predictions.shape)**

**shape of dense1: (100, 7)**

**shape of dense2: (100, 3)**

**shape of predictions: (100, 1)**

**Great work! With just 8 lines of code, you were able to define 2 dense hidden layers and an output layer. This is the advantage of using high-level operations in tensorflow. Note that each layer has 100 rows because the input data contains 100 examples.**

**LECTURE**

**1. Activation functions**

00:00 - 00:12

In the previous video, we discussed dense layers. We also briefly introduced the concept of an activation function through the sigmoid function. We will now return to activation functions.

**2. What is an activation function?**

00:12 - 00:24

A typical hidden layer consists of two operations. The first performs matrix multiplication, which is a linear operation, and the second applies an activation function, which is nonlinear operation.

**3. Why nonlinearities are important**

00:24 - 00:37

Why do we need this nonlinear component? Consider a simple model using the credit card data. The features are borrower age and credit card bill amount. The target variable is default.

**4. Why nonlinearities are important**

00:37 - 01:01

Let's say we create a scatterplot of age and bill amount. We can see that bill amount usually increases early in life and decreases later in life. This suggests that a high bill for young and older borrowers may mean something different for default. If we want our model to capture this, it can't be linear. It must allow the impact of the bill amount to depend on the borrower's age. This is what an activation function does.

**5. A simple example**

01:01 - 01:24

Let's look at a simple example, where we assume that the weight on age is 1 and the weight on the bill amount is 2. Note that ages are divided by 100 and the bill's amount is divided by 10000. We then perform the matrix multiplication step for all combinations of features: young with a high bill, young with a low bill, old with a high bill, and old with a low bill.

**6. A simple example**

01:24 - 01:50

If we don't apply an activation function and we assume the bias is zero, we find that the impact of bill size on default does not depend on age. In both cases, we predict a value of 0 point 8. Note that our target is a binary variable that is equal to 1 when the borrower defaults; however, predictions will be real numbers between 0 and 1, where values over 0 point 5 will be treated as predicting default.

**7. A simple example**

01:50 - 02:05

But what if we apply a sigmoid activation function? The impact of bill amount on default now depends on the borrower's age. In particular, we can see that the change in the predicted value for default is larger for young borrowers than it is for old borrowers.

**8. The sigmoid activation function**

02:05 - 02:33

In this course, we'll use the three most common activation functions: sigmoid, relu, and softmax. The sigmoid activation function is used primarily in the output layer of binary classification problems. When we use the low-level approach, we'll pass the sum of the product of weights and inputs into tf dot keras dot activations dot sigmoid. When we use the high-level approach, we'll simply pass sigmoid as a parameter to a keras dense layer.

**9. The relu activation function**

02:33 - 02:48

We'll typically use the rectified linear unit or relu activation in all layers other than the output layer. This activation simply takes the maximum of the value passed to it and 0.

**10. The softmax activation function**

02:48 - 03:05

Finally, the softmax activation function is used in the output layer in classification problems with more than two classes. The outputs from a softmax activation function can be interpreted as predicted class probabilities in multiclass classification problems.

**11. Activation functions in neural networks**

03:05 - 03:33

Let's wrap up by applying some activation functions in a neural network. We'll do this using the high-level approach, starting with an input layer. We'll pass this to our first dense layer, which has 16 output nodes and a relu activation. Dense layer 2 then reduces the number of nodes from 16 to 8 and applies a sigmoid activation. Finally, we apply a softmax activation function in the output layer, since there are more than 2 outputs.

**12. Let's practice!**

03:33 - 03:45

We've now seen what an activation function is and how to use the most common activation functions. Let's put that knowledge to work in some exercises!

## Exercise

# Binary classification problems

In this exercise, you will again make use of credit card data. The target variable, default, indicates whether a credit card holder defaults on his or her payment in the following period. Since there are only two options--default or not--this is a binary classification problem. While the dataset has many features, you will focus on just three: the size of the three latest credit card bills. Finally, you will compute predictions from your untrained network, outputs, and compare those the target variable, default.

The tensor of features has been loaded and is available as bill\_amounts. Additionally, the constant(), float32, and keras.layers.Dense() operations are available.

## Instructions

* Define inputs as a 32-bit floating point constant tensor using bill\_amounts.
* Set dense1 to be a dense layer with 3 output nodes and a relu activation function.
* Set dense2 to be a dense layer with 2 output nodes and a relu activation function.
* Set the output layer to be a dense layer with a single output node and a sigmoid activation function.

# Construct input layer from features

inputs = \_\_\_\_

# Define first dense layer

dense1 = keras.layers.Dense(\_\_\_\_, activation='\_\_\_\_')(inputs)

# Define second dense layer

dense2 = \_\_\_\_

# Define output layer

outputs = \_\_\_\_

# Print error for first five examples

error = default[:5] - outputs.numpy()[:5]

print(error)

# Construct input layer from features

inputs = constant(bill\_amounts, float32)

# Define first dense layer

dense1 = keras.layers.Dense(3, activation='relu')(inputs)

# Define second dense layer

dense2 = keras.layers.Dense(2, activation='relu')(dense1)

# Define output layer

outputs = keras.layers.Dense(1,activation='sigmoid')(dense2)

# Print error for first five examples

error = default[:5] - outputs.numpy()[:5]

print(error)

**# Construct input layer from features inputs = constant(bill\_amounts, float32) # Define first dense layer dense1 = keras.layers.Dense(3, activation='relu')(inputs) # Define second dense layer dense2 = keras.layers.Dense(2, activation='relu')(dense1) # Define output layer outputs = keras.layers.Dense(1,activation='sigmoid')(dense2) # Print error for first five examples error = default[:5] - outputs.numpy()[:5] print(error) [[-0.5] [-0.5] [-0.5] [-0.5] [-0.5]]**

**Excellent work! If you run the code several times, you'll notice that the errors change each time. This is because you're using an untrained model with randomly initialized parameters. Furthermore, the errors fall on the interval between -1 and 1 because default is a binary variable that takes on values of 0 and 1 and outputs is a probability between** 0 and 1

## Exercise

# Multiclass classification problems

In this exercise, we expand beyond binary classification to cover multiclass problems. A multiclass problem has targets that can take on three or more values. In the credit card dataset, the education variable can take on 6 different values, each corresponding to a different level of education. We will use that as our target in this exercise and will also expand the feature set from 3 to 10 columns.

As in the previous problem, you will define an input layer, dense layers, and an output layer. You will also print the untrained model's predictions, which are probabilities assigned to the classes. The tensor of features has been loaded and is available as borrower\_features. Additionally, the constant(), float32, and keras.layers.Dense() operations are available.

## Instructions

* Define the input layer as a 32-bit constant tensor using borrower\_features.
* Set the first dense layer to have 10 output nodes and a sigmoid activation function.
* Set the second dense layer to have 8 output nodes and a rectified linear unit activation function.
* Set the output layer to have 6 output nodes and the appropriate activation function.

**# Construct input layer from borrower features**

**inputs = \_\_\_\_**

**# Define first dense layer**

**dense1 = keras.layers.Dense(\_\_\_\_, activation='\_\_\_\_')(inputs)**

**# Define second dense layer**

**dense2 = \_\_\_\_**

**# Define output layer**

**outputs = \_\_\_\_**

**# Print first five predictions**

**print(outputs.numpy()[:5])**

# Construct input layer from borrower features

inputs = constant(borrower\_features, float32)

# Define first dense layer

dense1 = keras.layers.Dense(10, activation='sigmoid')(inputs)

# Define second dense layer

dense2 = keras.layers.Dense(8, activation='relu')(dense1)

# Define output layer

outputs = keras.layers.Dense(6, activation='softmax')(dense2)

# Print first five predictions

print(outputs.numpy()[:5])

**# Construct input layer from borrower features**

**inputs = constant(borrower\_features, float32)**

**# Define first dense layer**

**dense1 = keras.layers.Dense(10, activation='sigmoid')(inputs)**

**# Define second dense layer**

**dense2 = keras.layers.Dense(8, activation='relu')(dense1)**

**# Define output layer**

**outputs = keras.layers.Dense(6, activation='softmax')(dense2)**

**# Print first five predictions**

**print(outputs.numpy()[:5])**

**[[0.24982327 0.13220716 0.15474503 0.14913292 0.23624825 0.07784334]**

**[0.24982327 0.13220716 0.15474503 0.14913292 0.23624825 0.07784334]**

**[0.26214162 0.09721127 0.10593247 0.118999 0.3498708 0.06584484]**

**[0.24690026 0.23019755 0.22535135 0.11616834 0.14780991 0.03357264]**

**[0.2601174 0.1514326 0.17059962 0.13312687 0.21709381 0.06762964]]**

**Great work! Notice that each row of outputs sums to one. This is because a row contains the predicted class probabilities for one example. As with the previous exercise, our predictions are not yet informative, since we are using an untrained model with randomly initialized parameters. This is why the model tends to assign similar probabilities to each class.**

**LECTURE**

**1. Optimizers**

00:00 - 00:13

In chapter 2, you minimized a loss function with an optimizer. We'll revisit that here in the context of training neural networks. This entails finding the set of weights that corresponds to the minimum value of the loss.

**2. How to find a minimum**

00:13 - 00:44

So what is a minimization problem? And what can go wrong when we try to solve one? Let's start with a simple thought experiment: you want to find the lowest point in the Grand Canyon, but all you can do is pick a point, measure the elevation, and then repeat the same to nearby points. This is what you do when you train a neural network: you pick a starting point, measure the loss, and then try to move to a lower loss. We will see how a common optimization algorithm, gradient descent, solves this problem.

1. 1 Source: U.S. National Park Service

**3. How to find a minimum**

00:44 - 01:11

Let's start by picking a point and measuring the elevation. From that point, we'll move along the slope until we arrive on a flat surface. To understand what's going on, imagine you dropped a ball into the canyon from the point you selected. If you drop the ball on a slope above a plateau, the ball will stop when it reaches the plateau. If this happens, the gradient descent algorithm will fail. It will stop on a local minimum and will progress no further.

1. 1 Source: U.S. National Park Service

**4. How to find a minimum**

01:11 - 01:30

Let's say you pick a different spot. This time, the ball lands on a slope with an unobstructed path to the lowest point in the canyon. Here, the gradient descent algorithm works and ball reaches the global minimum. Notice that gravity performs the role of the gradient descent optimizer.

1. 1 Source: U.S. National Park Service

**5. Stochastic gradient descent**

01:30 - 02:01

Stochastic gradient descent or SGD is an improved version of gradient descent that is less likely to get stuck in local minima. For simple problems, the SGD algorithm performs well. Here, the SGD loss function value quickly falls below the losses for the more recently developed RMS Prop and the Adam optimizers on a simple minimization task. Adam and RMS require 10 times as many iterations to achieve a similar loss.

**6. The gradient descent optimizer**

02:01 - 02:43

Let's move on to the TensorFlow implementation for these optimizers, starting with SGD, which you can instantiate using the keras optimizers module. You can then supply a learning rate, typically between zero point five and zero point zero zero one, which will determine how quickly the model parameters adjust during training. Think of a higher learning rate as exerting more force on the ball than gravity alone. The ball will move faster and skip over some plateaus, but it may miss the global minimum, too. The main advantage of SGD is that it is simpler and easier to interpret than more modern optimization algorithms.

**7. The RMS prop optimizer**

02:43 - 03:08

Next, we'll consider the RMS propagation optimizer, which has two advantages over SGD. First, it applies different learning rates to each feature, which can be useful for high dimensional problems. And second, it allows you to both build momentum and also allow it to decay. Setting a low value for the decay parameter will prevent momentum from accumulating over long periods during the training process.

**8. The adam optimizer**

03:08 - 03:32

Finally, the adaptive moment or "adam" optimizer provides further improvements and is generally a good first choice. Similar to RMS prop, you can set the momentum to decay faster by lowering the beta1 parameter. Relative to RMS prop, the adam optimizer will tend to perform better with the default parameter values, which we will typically use.

**9. A complete example**

03:32 - 04:01

Let's return to our credit card default prediction problem and assume that features have been imported and weights have been initialized. We'll then define a model that computes the predictions and a loss function that computes the binary\_crossentropy loss, which is the standard for binary classification problems. Finally, we define an RMS prop optimizer with a learning rate of zero point one and a momentum parameter of zero point nine, and then perform minimization.

**10. Let's practice!**

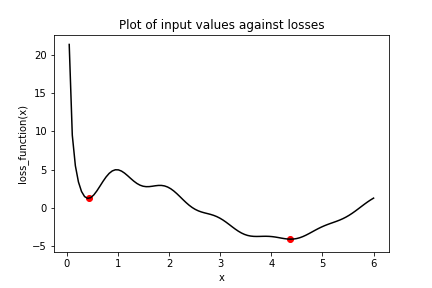
04:01 - 04:07

We now know how to find a minimum, so that's do that in some exercises.

## Exercise

# The dangers of local minima

Consider the plot of the following loss function, loss\_function(), which contains a global minimum, marked by the dot on the right, and several local minima, including the one marked by the dot on the left.



In this exercise, you will try to find the global minimum of loss\_function() using keras.optimizers.SGD(). You will do this twice, each time with a different initial value of the input to loss\_function(). First, you will use x\_1, which is a variable with an initial value of 6.0. Second, you will use x\_2, which is a variable with an initial value of 0.3. Note that loss\_function() has been defined and is available.

## Instructions

* Set opt to use the stochastic gradient descent optimizer (SGD) with a learning rate of 0.01.
* Perform minimization using the loss function, loss\_function(), and the variable with an initial value of 6.0, x\_1.
* Perform minimization using the loss function, loss\_function(), and the variable with an initial value of 0.3, x\_2.
* Print x\_1 and x\_2 as numpy arrays and check whether the values differ. These are the minima that the algorithm identified.

# Initialize x\_1 and x\_2

x\_1 = Variable(6.0,float32)

x\_2 = Variable(0.3,float32)

# Define the optimization operation

opt = keras.optimizers.\_\_\_\_(learning\_rate=\_\_\_\_)

for j in range(100):

    # Perform minimization using the loss function and x\_1

    opt.minimize(lambda: loss\_function(\_\_\_\_), var\_list=[\_\_\_\_])

    # Perform minimization using the loss function and x\_2

    opt.minimize(lambda: \_\_\_\_, var\_list=[\_\_\_\_])

# Print x\_1 and x\_2 as numpy arrays

print(\_\_\_\_.numpy(), \_\_\_\_.numpy())

# Initialize x\_1 and x\_2

x\_1 = Variable(6.0,float32)

x\_2 = Variable(0.3,float32)

# Define the optimization operation

opt = keras.optimizers.SGD(learning\_rate=0.01)

for j in range(100):

    # Perform minimization using the loss function and x\_1

    opt.minimize(lambda: loss\_function(x\_1), var\_list=[x\_1])

    # Perform minimization using the loss function and x\_2

    opt.minimize(lambda: loss\_function(x\_2), var\_list=[x\_2])

# Print x\_1 and x\_2 as numpy arrays

print(x\_1.numpy(), x\_2.numpy())

**# Initialize x\_1 and x\_2**

**x\_1 = Variable(6.0,float32)**

**x\_2 = Variable(0.3,float32)**

**# Define the optimization operation**

**opt = keras.optimizers.SGD(learning\_rate=0.01)**

**for j in range(100):**

**# Perform minimization using the loss function and x\_1**

**opt.minimize(lambda: loss\_function(x\_1), var\_list=[x\_1])**

**# Perform minimization using the loss function and x\_2**

**opt.minimize(lambda: loss\_function(x\_2), var\_list=[x\_2])**

**# Print x\_1 and x\_2 as numpy arrays**

**print(x\_1.numpy(), x\_2.numpy())**

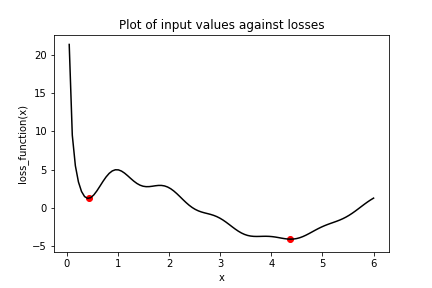
**4.3801394 0.42052683**

**Great work! Notice that we used the same optimizer and loss function, but two different initial values. When we started at 6.0 with x\_1, we found the global minimum at 4.38, marked by the dot on the right. When we started at 0.3, we stopped around 0.42 with x\_2, the local minimum marked by a dot on the far left.**

## Exercise

# Avoiding local minima

The previous problem showed how easy it is to get stuck in local minima. We had a simple optimization problem in one variable and gradient descent still failed to deliver the global minimum when we had to travel through local minima first. One way to avoid this problem is to use momentum, which allows the optimizer to break through local minima. We will again use the loss function from the previous problem, which has been defined and is available for you as loss\_function().



Several optimizers in tensorflow have a momentum parameter, including SGD and RMSprop. You will make use of RMSprop in this exercise. Note that x\_1 and x\_2 have been initialized to the same value this time. Furthermore, keras.optimizers.RMSprop() has also been imported for you from tensorflow.

## Instructions

* Set the opt\_1 operation to use a learning rate of 0.01 and a momentum of 0.99.
* Set opt\_2 to use the root mean square propagation (RMS) optimizer with a learning rate of 0.01 and a momentum of 0.00.
* Define the minimization operation for opt\_2.
* Print x\_1 and x\_2 as numpy arrays.

# Initialize x\_1 and x\_2

x\_1 = Variable(0.05,float32)

x\_2 = Variable(0.05,float32)

# Define the optimization operation for opt\_1 and opt\_2

opt\_1 = keras.optimizers.RMSprop(learning\_rate=\_\_\_\_, momentum=\_\_\_\_)

opt\_2 = \_\_\_\_

for j in range(100):

    opt\_1.minimize(lambda: loss\_function(x\_1), var\_list=[x\_1])

    # Define the minimization operation for opt\_2

    \_\_\_\_

# Print x\_1 and x\_2 as numpy arrays

print(\_\_\_\_, \_\_\_\_)

# Initialize x\_1 and x\_2

x\_1 = Variable(0.05,float32)

x\_2 = Variable(0.05,float32)

# Define the optimization operation for opt\_1 and opt\_2

opt\_1 = keras.optimizers.RMSprop(learning\_rate=0.01, momentum=0.99)

opt\_2 = keras.optimizers.RMSprop(learning\_rate=0.01, momentum=0.00)

for j in range(100):

    opt\_1.minimize(lambda: loss\_function(x\_1), var\_list=[x\_1])

    # Define the minimization operation for opt\_2

    opt\_2.minimize(lambda: loss\_function(x\_2), var\_list=[x\_2])

# Print x\_1 and x\_2 as numpy arrays

print(x\_1.numpy(), x\_2.numpy())

**# Initialize x\_1 and x\_2**

**x\_1 = Variable(0.05,float32)**

**x\_2 = Variable(0.05,float32)**

**# Define the optimization operation for opt\_1 and opt\_2**

**opt\_1 = keras.optimizers.RMSprop(learning\_rate=0.01, momentum=0.99)**

**opt\_2 = keras.optimizers.RMSprop(learning\_rate=0.01, momentum=0.00)**

**for j in range(100):**

**opt\_1.minimize(lambda: loss\_function(x\_1), var\_list=[x\_1])**

**# Define the minimization operation for opt\_2**

**opt\_2.minimize(lambda: loss\_function(x\_2), var\_list=[x\_2])**

**# Print x\_1 and x\_2 as numpy arrays**

**print(x\_1.numpy(), x\_2.numpy())**

**4.3150263 0.4205261**

**Good work! Recall that the global minimum is approximately 4.38. Notice that opt\_1 built momentum, bringing x\_1 closer to the global minimum. To the contrary, opt\_2, which had a momentum parameter of 0.0, got stuck in the local minimum on the left.**

**LECTURE**

**Training a network in TensorFlow**

00:00 - 00:08

In the final video in this chapter, we'll wrap-up by discussing important topics related to training neural networks in TensorFlow.

**2. Initializing variables**

00:08 - 00:44

We saw that finding the global minimum can be difficult, even when we're minimizing a simple loss function. We also saw that we could improve our chances by selecting better initial values for variables. But what can we do for more challenging problems with many variables? Take the eggholder function, for example, which has many local minima. It is difficult to see a global minimum on the plot, but it has one. How can we select initial values for x and y, the two inputs to the eggholder function? Even worse, what if we have a loss function that depends on hundreds of variables?

**3. Random initializers**

00:44 - 01:16

We often need to initialize hundreds or thousands of variables. Simply using ones will not work. And selecting initial values individually is tedious and infeasible in many cases. A natural alternative to this is to use random or algorithmic generation of initial values. We can, for instance, draw them from a probability distribution, such as the normal or uniform distributions. There are also specialized options, such as the Glorot initializers, which are designed for ML algorithms.

**4. Initializing variables in TensorFlow**

01:16 - 01:42

Let's start by using the low-level approach to initialize a 500x500 variable. We can do this using draws from a random normal distribution by passing the shape 500, 500 to tf dot random dot normal and passing the result to tf dot Variable. Alternatively, we could use the truncated random normal distribution, which discards very large and very small draws.

**5. Initializing variables in TensorFlow**

01:42 - 02:01

We can also use the high-level approach by initializing a dense layer using the default keras option, currently the glorot uniform initializer, as we've done in all exercises thus far. If we instead wish to initialize values to zero, we can do this using the kernel initializer parameter.

**6. Neural networks and overfitting**

02:01 - 02:36

Overfitting is another important issue you'll encounter when training neural networks. Let's say you have a linear relationship between two variables. You decide to represent this relationship with a linear model, shown in red, and a more complex model, shown in blue. The complex model perfectly predicts the values in the training set, but performs worse in the test set. The complex model performed poorly because it overfit. It simply memorized examples, rather than learning general patterns. Overfitting is especially problematic for neural networks, which contain many parameters and are quite good at memorization.

**7. Applying dropout**

02:36 - 03:01

A simple solution to the overfitting problem is to use dropout, an operation that will randomly drop the weights connected to certain nodes in a layer during the training process, as shown on the right. This will force your network to develop more robust rules for classification, since it cannot rely on any particular nodes being passed to an activation function. This will tend to improve out-of-sample performance.

**8. Implementing dropout in a network**

03:01 - 03:15

Let's look at how dropout works. We first define an input layer using the borrower features from our credit card dataset as an input. We then pass the input layer to a dense layer, which has 32 nodes and uses a relu activation function.

**9. Implementing dropout in a network**

03:15 - 03:41

We'll next pass the first dense layer to a second layer, which reduces the number of output nodes to 16. Before passing those nodes to the output layer, we'll apply a dropout layer. The only argument specifies that we want to drop the weights connected to 25% of nodes randomly. We'll then pass this to the output layer, which reduces the 16 nodes to 1 and applies a sigmoid activation function.

**10. Let's practice!**

03:41 - 03:49

You now know everything you need to construct and train a neural network, so let's do that with some exercises.

## Exercise

# Initialization in TensorFlow

A good initialization can reduce the amount of time needed to find the global minimum. In this exercise, we will initialize weights and biases for a neural network that will be used to predict credit card default decisions. To build intuition, we will use the low-level, linear algebraic approach, rather than making use of convenience functions and high-level keras operations. We will also expand the set of input features from 3 to 23. Several operations have been imported from tensorflow: Variable(), random(), and ones().

## Instructions

* Initialize the layer 1 weights, w1, as a Variable() with shape [23, 7], drawn from a normal distribution.
* Initialize the layer 1 bias using ones.
* Use a draw from the normal distribution to initialize w2 as a Variable() with shape [7, 1].
* Define b2 as a Variable() and set its initial value to 0.0.

# Define the layer 1 weights

w1 = \_\_\_\_(random.normal([\_\_\_\_, \_\_\_\_]))

# Initialize the layer 1 bias

b1 = Variable(\_\_\_\_([7]))

# Define the layer 2 weights

w2 = \_\_\_\_

# Define the layer 2 bias

b2 = \_\_\_\_

 Define the layer 1 weights

w1 = Variable(random.normal([23,7]))

# Initialize the layer 1 bias

b1 = Variable(ones ([7]))

# Define the layer 2 weights

w2 = Variable(random.normal([7,1]))

# Define the layer 2 bias

b2 = Variable(0.0)

**# Define the layer 1 weights w1 = Variable(random.normal([23,7])) # Initialize the layer 1 bias b1 = Variable(ones ([7])) # Define the layer 2 weights w2 = Variable(random.normal([7,1])) # Define the layer 2 bias b2 = Variable(0.0)**

Excellent work! In the next exercise, you will start where we've ended and will finish constructing the neural network.

## Exercise

# Defining the model and loss function

In this exercise, you will train a neural network to predict whether a credit card holder will default. The features and targets you will use to train your network are available in the Python shell as borrower\_features and default. You defined the weights and biases in the previous exercise.

Note that the predictions layer is defined as

, where

is the sigmoid activation, layer1 is a tensor of nodes for the first hidden dense layer, w2 is a tensor of weights, and b2 is the bias tensor.

The trainable variables are w1, b1, w2, and b2. Additionally, the following operations have been imported for you: keras.activations.relu() and keras.layers.Dropout().

## Instructions

* Apply a rectified linear unit activation function to the first layer.
* Apply 25% dropout to layer1.
* Pass the target, targets, and the predicted values, predictions, to the cross entropy loss function.

# Define the model

def model(w1, b1, w2, b2, features = borrower\_features):

    # Apply relu activation functions to layer 1

    layer1 = keras.activations.\_\_\_\_(matmul(features, w1) + b1)

    # Apply dropout rate of 0.25

    dropout = keras.layers.Dropout(\_\_\_\_)(\_\_\_\_)

    return keras.activations.sigmoid(matmul(dropout, w2) + b2)

# Define the loss function

def loss\_function(w1, b1, w2, b2, features = borrower\_features, targets = default):

    predictions = model(w1, b1, w2, b2)

    # Pass targets and predictions to the cross entropy loss

    return keras.losses.binary\_crossentropy(\_\_\_\_, \_\_\_\_)

**# Define the model**

**def model(w1, b1, w2, b2, features = borrower\_features):**

**# Apply relu activation functions to layer 1**

**layer1 = keras.activations.relu(matmul(features, w1) + b1)**

**# Apply dropout rate of 0.25**

**dropout = keras.layers.Dropout(.25)(layer1)**

**return keras.activations.sigmoid(matmul(dropout, w2) + b2)**

**# Define the loss function**

**def loss\_function(w1, b1, w2, b2, features = borrower\_features, targets = default):**

**predictions = model(w1, b1, w2, b2)**

**# Pass targets and predictions to the cross entropy loss**

**return keras.losses.binary\_crossentropy(targets, predictions)**

**Nice work! One of the benefits of using tensorflow is that you have the option to customize models down to the linear algebraic-level, as we've shown in the last two exercises. If you print w1, you can see that the objects we're working with are simply tensors.**

## Exercise

# Training neural networks with TensorFlow

In the previous exercise, you defined a model, model(w1, b1, w2, b2, features), and a loss function, loss\_function(w1, b1, w2, b2, features, targets), both of which are available to you in this exercise. You will now train the model and then evaluate its performance by predicting default outcomes in a test set, which consists of test\_features and test\_targets and is available to you. The trainable variables are w1, b1, w2, and b2. Additionally, the following operations have been imported for you: keras.activations.relu() and keras.layers.Dropout().

## Instructions

* Set the optimizer to perform minimization.
* Add the four trainable variables to var\_list in the order in which they appear as arguments to loss\_function().
* Use the model and test\_features to predict the values for test\_targets.

# Train the model

for j in range(100):

    # Complete the optimizer

    opt.\_\_\_\_(lambda: loss\_function(w1, b1, w2, b2),

                 var\_list=[\_\_\_\_, \_\_\_\_, \_\_\_\_, \_\_\_\_])

# Make predictions with model using test features

model\_predictions = model(w1, b1, w2, b2, \_\_\_\_)

# Construct the confusion matrix

confusion\_matrix(test\_targets, model\_predictions)

# Train the model

for j in range(100):

    # Complete the optimizer

    opt.minimize(lambda: loss\_function(w1, b1, w2, b2),

                 var\_list=[w1, b1, w2, b2])

# Make predictions with model using test features

model\_predictions = model(w1, b1, w2, b2, test\_features)

# Construct the confusion matrix

confusion\_matrix(test\_targets, model\_predictions)

**# Train the model for j in range(100): # Complete the optimizer opt.minimize(lambda: loss\_function(w1, b1, w2, b2), var\_list=[w1, b1, w2, b2]) # Make predictions with model using test features model\_predictions = model(w1, b1, w2, b2, test\_features) # Construct the confusion matrix confusion\_matrix(test\_targets, model\_predictions)**

**Nice work! The diagram shown is called a ``confusion matrix.'' The diagonal elements show the number of correct predictions. The off-diagonal elements show the number of incorrect predictions. We can see that the model performs reasonably-well, but does so by overpredicting non-default. This suggests that we may need to train longer, tune the model's hyperparameters, or change the model's architecture.**

**LECTURE**

**1. Defining neural networks with Keras**

00:00 - 00:17

In chapter 3, we saw how to define neural networks in TensorFlow, both using linear algebra and higher level Keras operations. In this lesson, we will introduce the Keras sequential API, and expand on our brief and informal introduction of the Keras functional API.

**2. Classifying sign language letters**

00:17 - 00:36

Throughout this chapter, we'll focus on using Keras to classify four letters from the Sign Language MNIST dataset: a, b, c, and d. Note that the images appear to be low resolution because each is represented by a 28x28 matrix.

**3. The sequential API**

00:36 - 00:53

Now, let's say we experiment with several different architectures and select the one that makes the most accurate predictions. It has an input layer, a first hidden layer with 16 nodes, and a second hidden layer with 8 nodes. We'll have 4 output nodes, since there are 4 letters in the dataset.

**4. The sequential API**

00:53 - 01:12

A good way to construct this model in Keras is to use the sequential API. This API is simpler and makes strong assumptions about how you will construct your model. It assumes that you have an input layer, some number of hidden layers, and an output layer. All of these layers are ordered one after the other in a sequence.

**5. Building a sequential model**

01:12 - 01:53

We'll start by importing tensorflow. We can then define a sequential model, which we'll name model. Once we have defined this object, we can simply stack layers on top of it sequentially using the add method. Let's start by adding the first hidden layer, which is a dense layer with 16 nodes. We'll select a relu activation function and supply an input\_shape, which Keras requires for the first layer. This input shape is simply a tuple that contains the dimensions of our data. Since we'll be using 28 by 28 pixel images, reshaped into vector, we'll supply 28\*28 comma as the input shape.

**6. Building a sequential model**

01:53 - 02:32

Next, we'll define a second hidden layer according to the desired model architecture. Finally, we specify that the model has 4 output nodes and uses a softmax activation function. If we want to check our model's architecture, we can use the dot summarize method, which we'll return to in the upcoming exercises. The model has now been defined, but it is not yet ready to be trained. We must first perform a compilation step, where we specify the optimizer and loss function. Here, we've selected the adam optimizer and the categorical crossentropy loss function, which we'll use for classification problems with more than 2 classes.

**7. The functional API**

02:32 - 02:39

But what if you want to train two models jointly to predict the same target? The functional API is for that.

**8. Using the functional API**

02:39 - 03:15

As an example, let's say we have a set of 28x28 images and a set of 10 features of metadata. We want to use both to predict the image's class, but restrict how they interact in our model. We'll start by using the Keras inputs operation to define the input shapes for model 1 and model 2. Next, we define layer 1 and layer 2 as dense layers for model 1. Note that we have to pass the previous layer as an argument if we use the functional API, but did not with the sequential. You may remember that we did this in chapter 3. We were also using the functional API then.

**9. Using the functional API**

03:15 - 03:36

We now define layers 1 and 2 for model 2 and then use the add layer in keras to combine the outputs in a layer that merges the two models. Finally, we define a functional model. As inputs, it takes both the model 1 and model 2 inputs. As outputs, it takes the merged layer. The only thing left to do is compile it and train.

**10. Let's practice!**

03:36 - 03:43

It's now time to put what you've learned to work in some exercises.

## Exercise

# The sequential model in Keras

In chapter 3, we used components of the keras API in tensorflow to define a neural network, but we stopped short of using its full capabilities to streamline model definition and training. In this exercise, you will use the keras sequential model API to define a neural network that can be used to classify images of sign language letters. You will also use the .summary() method to print the model's architecture, including the shape and number of parameters associated with each layer.

Note that the images were reshaped from (28, 28) to (784,), so that they could be used as inputs to a dense layer. Additionally, note that keras has been imported from tensorflow for you.

## Instructions

* Define a keras sequential model named model.
* Set the first layer to be Dense() and to have 16 nodes and a relu activation.
* Define the second layer to be Dense() and to have 8 nodes and a relu activation.
* Set the output layer to have 4 nodes and use a softmax activation function.

# Define a Keras sequential model

\_\_\_\_

# Define the first dense layer

model.add(keras.layers.\_\_\_\_(\_\_\_\_, activation='\_\_\_\_', input\_shape=(784,)))

# Define the second dense layer

\_\_\_\_

# Define the output layer

model.add(keras.layers.Dense(\_\_\_\_))

# Print the model architecture

print(model.summary())

# Define a Keras sequential model

model = keras.Sequential()

# Define the first dense layer

model.add(keras.layers.Dense(16, activation='relu', input\_shape=(784,)))

# Define the second dense layer

model.add(keras.layers.Dense(8, activation='relu'))

# Define the output layer

model.add(keras.layers.Dense(4,activation='softmax'))

# Print the model architecture

print(model.summary())

**# Define a Keras sequential model**

**model = keras.Sequential()**

**# Define the first dense layer**

**model.add(keras.layers.Dense(16, activation='relu', input\_shape=(784,)))**

**# Define the second dense layer**

**model.add(keras.layers.Dense(8, activation='relu'))**

**# Define the output layer**

**model.add(keras.layers.Dense(4,activation='softmax'))**

**# Print the model architecture**

**print(model.summary())**

**Model: "sequential\_1"**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Layer (type) Output Shape Param #**

**=================================================================**

**dense\_1 (Dense) (None, 16) 12560**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_2 (Dense) (None, 8) 136**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_3 (Dense) (None, 4) 36**

**=================================================================**

**Total params: 12,732**

**Trainable params: 12,732**

**Non-trainable params: 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**None**

**Excellent work! Notice that we've defined a model, but we haven't compiled it. The compilation step in keras allows us to set the optimizer, loss function, and other useful training parameters in a single line of code. Furthermore, the .summary() method allows us to view the model's architecture.**

## Exercise

# Compiling a sequential model

In this exercise, you will work towards classifying letters from the Sign Language MNIST dataset; however, you will adopt a different network architecture than what you used in the previous exercise. There will be fewer layers, but more nodes. You will also apply dropout to prevent overfitting. Finally, you will compile the model to use the adam optimizer and the categorical\_crossentropy loss. You will also use a method in keras to summarize your model's architecture. Note that keras has been imported from tensorflow for you and a sequential keras model has been defined as model.

## Instructions

* In the first dense layer, set the number of nodes to 16, the activation to sigmoid, and the input\_shape to (784,).
* Apply dropout at a rate of 25% to the first layer's output.
* Set the output layer to be dense, have 4 nodes, and use a softmax activation function.
* Compile the model using an adam optimizer and categorical\_crossentropy loss function.

# Define the first dense layer

model.add(keras.layers.Dense(\_\_\_\_, \_\_\_\_, \_\_\_\_))

# Apply dropout to the first layer's output

model.add(keras.layers.\_\_\_\_(0.25))

# Define the output layer

\_\_\_\_

# Compile the model

model.compile('\_\_\_\_', loss='\_\_\_\_')

# Print a model summary

print(model.summary())

# Define the first dense layer

model.add(keras.layers.Dense(16, activation='sigmoid', input\_shape=(784,)))

# Apply dropout to the first layer's output

model.add(keras.layers.Dropout(0.25))

# Define the output layer

model.add(keras.layers.Dense(4, activation='softmax'))

# Compile the model

model.compile('adam', loss='categorical\_crossentropy')

# Print a model summary

print(model.summary())

**# Define the first dense layer**

**model.add(keras.layers.Dense(16, activation='sigmoid', input\_shape=(784,)))**

**# Apply dropout to the first layer's output**

**model.add(keras.layers.Dropout(0.25))**

**# Define the output layer**

**model.add(keras.layers.Dense(4, activation='softmax'))**

**# Compile the model**

**model.compile('adam', loss='categorical\_crossentropy')**

**# Print a model summary**

**print(model.summary())**

**Model: "sequential"**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Layer (type) Output Shape Param #**

**=================================================================**

**dense\_2 (Dense) (None, 16) 12560**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_3 (Dense) (None, 16) 272**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dropout (Dropout) (None, 16) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_4 (Dense) (None, 4) 68**

**=================================================================**

**Total params: 12,900**

**Trainable params: 12,900**

**Non-trainable params: 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**None**

**Great work! You've now defined and compiled a neural network using the keras sequential model. Notice that printing the .summary() method shows the layer type, output shape, and number of parameters of each layer.**

## Exercise

# Defining a multiple input model

In some cases, the sequential API will not be sufficiently flexible to accommodate your desired model architecture and you will need to use the functional API instead. If, for instance, you want to train two models with different architectures jointly, you will need to use the functional API to do this. In this exercise, we will see how to do this. We will also use the .summary() method to examine the joint model's architecture.

Note that keras has been imported from tensorflow for you. Additionally, the input layers of the first and second models have been defined as m1\_inputs and m2\_inputs, respectively. Note that the two models have the same architecture, but one of them uses a sigmoid activation in the first layer and the other uses a relu.

## Instructions

* Pass model 1's input layer to its first layer and model 1's first layer to its second layer.
* Pass model 2's input layer to its first layer and model 2's first layer to its second layer.
* Use the add() operation to combine the second layers of model 1 and model 2.
* Complete the functional model definition.

# For model 1, pass the input layer to layer 1 and layer 1 to layer 2

m1\_layer1 = keras.layers.Dense(12, activation='sigmoid')(\_\_\_\_)

m1\_layer2 = keras.layers.Dense(4, activation='softmax')(\_\_\_\_)

# For model 2, pass the input layer to layer 1 and layer 1 to layer 2

m2\_layer1 = keras.layers.Dense(12, activation='relu')(\_\_\_\_)

m2\_layer2 = keras.layers.Dense(4, activation='softmax')(\_\_\_\_)

# Merge model outputs and define a functional model

merged = keras.layers.add([m1\_layer2, \_\_\_\_])

model = keras.Model(inputs=[\_\_\_\_, m2\_inputs], outputs=\_\_\_\_)

# Print a model summary

print(model.summary())

# For model 1, pass the input layer to layer 1 and layer 1 to layer 2

m1\_layer1 = keras.layers.Dense(12, activation='sigmoid')(m1\_inputs)

m1\_layer2 = keras.layers.Dense(4, activation='softmax')(m1\_layer1)

# For model 2, pass the input layer to layer 1 and layer 1 to layer 2

m2\_layer1 = keras.layers.Dense(12, activation='relu')(m2\_inputs)

m2\_layer2 = keras.layers.Dense(4, activation='softmax')(m2\_layer1)

# Merge model outputs and define a functional model

merged = keras.layers.add([m1\_layer2, m2\_layer2])

model = keras.Model(inputs=[m1\_inputs, m2\_inputs], outputs=merged)

# Print a model summary

print(model.summary())

**# For model 1, pass the input layer to layer 1 and layer 1 to layer 2**

**m1\_layer1 = keras.layers.Dense(12, activation='sigmoid')(m1\_inputs)**

**m1\_layer2 = keras.layers.Dense(4, activation='softmax')(m1\_layer1)**

**# For model 2, pass the input layer to layer 1 and layer 1 to layer 2**

**m2\_layer1 = keras.layers.Dense(12, activation='relu')(m2\_inputs)**

**m2\_layer2 = keras.layers.Dense(4, activation='softmax')(m2\_layer1)**

**# Merge model outputs and define a functional model**

**merged = keras.layers.add([m1\_layer2, m2\_layer2])**

**model = keras.Model(inputs=[m1\_inputs, m2\_inputs], outputs=merged)**

**# Print a model summary**

**print(model.summary())**

**Model: "model"**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Layer (type) Output Shape Param # Connected to**

**==================================================================================================**

**input\_1 (InputLayer) [(None, 784)] 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**input\_2 (InputLayer) [(None, 784)] 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_4 (Dense) (None, 12) 9420 input\_1[0][0]**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_6 (Dense) (None, 12) 9420 input\_2[0][0]**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_5 (Dense) (None, 4) 52 dense\_4[0][0]**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dense\_7 (Dense) (None, 4) 52 dense\_6[0][0]**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**add (Add) (None, 4) 0 dense\_5[0][0]**

**dense\_7[0][0]**

**==================================================================================================**

**Total params: 18,944**

**Trainable params: 18,944**

**Non-trainable params: 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**None**

**Nice work! Notice that the .summary() method yields a new column: connected to. This column tells you how layers connect to each other within the network. We can see that dense\_2, for instance, is connected to the input\_2 layer. We can also see that the add layer, which merged the two models, connected to both dense\_1 and dense\_3.**

**LECTURE**

**1. Training with Keras**

00:00 - 00:09

Earlier in the chapter, we defined neural networks in Keras. In this video, we will discuss how to train and evaluate them.

**2. Overview of training and evaluation**

00:09 - 00:26

Whenever we train and evaluate a model in tensorflow, we typically use the same set of steps. First, we'll load and clean the data. Second, we'll define a model, specifying an architecture. Third, we'll train and validate the model. And fourth, we perform evaluation.

**3. How to train a model**

00:26 - 00:53

Let's see an example of how this works. We'll start by importing tensorflow and defining a keras sequential model. We'll then add a dense layer to the model with 16 nodes and a relu activation function. Note that our input shape is (784,), since our dataset consists of 28x28 images, reshaped into vectors. We next define the output layer, which has 4 nodes and a softmax activation function.

**4. How to train a model**

00:53 - 01:05

We next compile the model, using the adam optimizer and the categorical cross entropy loss. Finally, we train the model using the fit operation.

**5. The fit() operation**

01:05 - 01:21

Notice that we only supplied two arguments to fit: features and labels. These are the only two required arguments; however, there are also many optional arguments, including batch\_size, epochs, and validation\_split. We will cover each of these.

**6. Batch size and epochs**

01:21 - 01:51

Let's start with the difference between the batch size and epochs parameters. The number of examples in each batch is the batch size, which is 32 by default. The number of times you train on the full set of batches is called the number of epochs. Here, the batch size is 5 and the number of epochs is 2. Using multiple epochs allows the model to revisit the same batches, but with different model weights and possibly optimizer parameters, since they are updated after each batch.

**7. Performing validation**

01:51 - 02:03

So what does the validation\_split parameter do? It divides the dataset into two parts. The first part is the train set and the second part is the validation set.

**8. Performing validation**

02:03 - 02:10

Selecting a value of zero point two will put 20% of the data in the validation set.

**9. Performing validation**

02:10 - 02:42

The benefit of using a validation split is that you can see how your model performs on both the data it was trained on, the training set, and a separate dataset it was not trained on, the validation set. Here, we can see the first 10 epochs of training. Notice that we can see the training loss and validation loss separately. If the training loss becomes substantially lower than the validation loss, this is an indication that we're overfitting. We should either terminate the training process before that point or add regularization or dropout.

**10. Changing the metric**

02:42 - 03:04

Another benefit of the high level keras API is that we can swap less informative metrics, such as the loss, for ones that are easily interpretable, such as the share of accurately classified examples. We can do this by supplying accuracy to the metrics parameter of compile. We then apply fit to the model again with the same settings.

**11. Changing the metric**

03:04 - 03:21

Using the accuracy metric, we can see that the model performs quite well. In just 10 epochs, it goes from an accuracy of 42% to over 99%. Notice that the model performs equally well in the validation set, which means that we're unlikely to be overfitting.

**12. The evaluation() operation**

03:21 - 03:44

Finally, it is good idea to split off a test set before you begin to train and validate. You can use the evaluate operation to check performance on the test set at the end of the training process. Since you may tune model parameters in response to validation set performance, using a separate test set will provide you with further assurance that you have not overfitted.

**13. Let's practice!**

03:44 - 03:52

You now know how to streamline model training and validation in keras, so let's practice doing it with a few exercises.

## Exercise

# Training with Keras

In this exercise, we return to our sign language letter classification problem. We have 2000 images of four letters--A, B, C, and D--and we want to classify them with a high level of accuracy. We will complete all parts of the problem, including the model definition, compilation, and training.

Note that keras has been imported from tensorflow for you. Additionally, the features are available as sign\_language\_features and the targets are available as sign\_language\_labels.

## Instructions

* Define a sequential model named model.
* Set the output layer to be dense, have 4 nodes, and use a softmax activation function.
* Compile the model with the SGD optimizer and categorical\_crossentropy loss.
* Complete the fitting operation and set the number of epochs to 5.

# Define a sequential model

\_\_\_\_

# Define a hidden layer

model.add(keras.layers.Dense(16, activation='relu', input\_shape=(784,)))

# Define the output layer

\_\_\_\_

# Compile the model

model.compile('\_\_\_\_', loss='\_\_\_\_')

# Complete the fitting operation

model.fit(\_\_\_\_, \_\_\_\_, epochs=\_\_\_\_)

# Define a sequential model

model = keras.Sequential()

# Define a hidden layer

model.add(keras.layers.Dense(16, activation='relu', input\_shape=(784,)))

# Define the output layer

model.add(keras.layers.Dense(4, activation='softmax'))

# Compile the model

model.compile('SGD', loss='categorical\_crossentropy')

# Complete the fitting operation

model.fit(sign\_language\_features, sign\_language\_labels, epochs=5)

**# Define a sequential model**

**model = keras.Sequential()**

**# Define a hidden layer**

**model.add(keras.layers.Dense(16, activation='relu', input\_shape=(784,)))**

**# Define the output layer**

**model.add(keras.layers.Dense(4, activation='softmax'))**

**# Compile the model**

**model.compile('SGD', loss='categorical\_crossentropy')**

**# Complete the fitting operation**

**model.fit(sign\_language\_features, sign\_language\_labels, epochs=5)**

**Epoch 1/5**

**1/32 [..............................] - ETA: 7s - loss: 1.5992**

**32/32 [==============================] - 0s 652us/step - loss: 1.3770**

**Epoch 2/5**

**1/32 [..............................] - ETA: 0s - loss: 1.1744**

**32/32 [==============================] - 0s 1ms/step - loss: 1.1042**

**Epoch 3/5**

**1/32 [..............................] - ETA: 0s - loss: 1.0461**

**32/32 [==============================] - 0s 616us/step - loss: 0.9345**

**Epoch 4/5**

**1/32 [..............................] - ETA: 0s - loss: 1.1026**

**32/32 [==============================] - 0s 619us/step - loss: 0.8203**

**Epoch 5/5**

**1/32 [..............................] - ETA: 0s - loss: 0.6915**

**32/32 [==============================] - 0s 721us/step - loss: 0.7053**

**<keras.callbacks.History at 0x7fb5f86c7730>**

**Great work! You probably noticed that your only measure of performance improvement was the value of the loss function in the training sample, which is not particularly informative. You will improve on this in the next exercise.**

## Exercise

# Metrics and validation with Keras

We trained a model to predict sign language letters in the previous exercise, but it is unclear how successful we were in doing so. In this exercise, we will try to improve upon the interpretability of our results. Since we did not use a validation split, we only observed performance improvements within the training set; however, it is unclear how much of that was due to overfitting. Furthermore, since we did not supply a metric, we only saw decreases in the loss function, which do not have any clear interpretation.

Note that keras has been imported for you from tensorflow.

## Instructions

* Set the first dense layer to have 32 nodes, use a sigmoid activation function, and have an input shape of (784,).
* Use the root mean square propagation optimizer, a categorical crossentropy loss, and the accuracy metric.
* Set the number of epochs to 10 and use 10% of the dataset for validation.

# Define sequential model

model = keras.Sequential()

# Define the first layer

model.add(keras.layers.Dense(\_\_\_\_, \_\_\_\_, \_\_\_\_))

# Add activation function to classifier

model.add(keras.layers.Dense(4, activation='softmax'))

# Set the optimizer, loss function, and metrics

model.compile(optimizer='\_\_\_\_', loss='\_\_\_\_', metrics=['\_\_\_\_'])

# Add the number of epochs and the validation split

model.fit(sign\_language\_features, sign\_language\_labels, epochs=\_\_\_\_, validation\_split=\_\_\_\_)

# Define sequential model

model = keras.Sequential()

# Define the first layer

model.add(keras.layers.Dense(32, activation='sigmoid', input\_shape=(784,)))

# Add activation function to classifier

model.add(keras.layers.Dense(4, activation='softmax'))

# Set the optimizer, loss function, and metrics

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

# Add the number of epochs and the validation split

model.fit(sign\_language\_features, sign\_language\_labels, epochs=10, validation\_split=0.10)

# Define sequential model

model = keras.Sequential()

# Define the first layer

model.add(keras.layers.Dense(32, activation='sigmoid', input\_shape=(784,)))

# Add activation function to classifier

model.add(keras.layers.Dense(4, activation='softmax'))

# Set the optimizer, loss function, and metrics

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

# Add the number of epochs and the validation split

model.fit(sign\_language\_features, sign\_language\_labels, epochs=10, validation\_split=0.10)

Epoch 1/10

1/29 [>.............................] - ETA: 7s - loss: 1.6242 - accuracy: 0.2500

29/29 [==============================] - 0s 6ms/step - loss: 1.3073 - accuracy: 0.3904 - val\_loss: 1.2338 - val\_accuracy: 0.2700

Epoch 2/10

1/29 [>.............................] - ETA: 0s - loss: 1.3149 - accuracy: 0.2500

29/29 [==============================] - 0s 2ms/step - loss: 1.0383 - accuracy: 0.6674 - val\_loss: 1.0973 - val\_accuracy: 0.4900

Epoch 3/10

1/29 [>.............................] - ETA: 0s - loss: 0.9843 - accuracy: 0.6250

29/29 [==============================] - 0s 1ms/step - loss: 0.8826 - accuracy: 0.7386 - val\_loss: 0.8402 - val\_accuracy: 0.8000

Epoch 4/10

1/29 [>.............................] - ETA: 0s - loss: 0.8916 - accuracy: 0.7500

29/29 [==============================] - 0s 1ms/step - loss: 0.7344 - accuracy: 0.8087 - val\_loss: 0.8624 - val\_accuracy: 0.7000

Epoch 5/10

1/29 [>.............................] - ETA: 0s - loss: 0.7999 - accuracy: 0.7812

29/29 [==============================] - 0s 1ms/step - loss: 0.6422 - accuracy: 0.8576 - val\_loss: 0.6663 - val\_accuracy: 0.7600

Epoch 6/10

1/29 [>.............................] - ETA: 0s - loss: 0.7542 - accuracy: 0.7188

29/29 [==============================] - 0s 1ms/step - loss: 0.5631 - accuracy: 0.8743 - val\_loss: 0.7867 - val\_accuracy: 0.6200

Epoch 7/10

1/29 [>.............................] - ETA: 0s - loss: 0.8428 - accuracy: 0.6250

29/29 [==============================] - 0s 2ms/step - loss: 0.4988 - accuracy: 0.9121 - val\_loss: 0.4803 - val\_accuracy: 0.9700

Epoch 8/10

1/29 [>.............................] - ETA: 0s - loss: 0.5093 - accuracy: 0.9375

29/29 [==============================] - 0s 1ms/step - loss: 0.4456 - accuracy: 0.9155 - val\_loss: 0.5915 - val\_accuracy: 0.7100

Epoch 9/10

1/29 [>.............................] - ETA: 0s - loss: 0.4325 - accuracy: 0.8125

29/29 [==============================] - 0s 1ms/step - loss: 0.3928 - accuracy: 0.9399 - val\_loss: 0.3918 - val\_accuracy: 0.9700

Epoch 10/10

1/29 [>.............................] - ETA: 0s - loss: 0.3360 - accuracy: 1.0000

29/29 [==============================] - 0s 1ms/step - loss: 0.3482 - accuracy: 0.9410 - val\_loss: 0.3829 - val\_accuracy: 0.8500

<keras.callbacks.History at 0x7fb5c4260c10>

**Nice work! With the keras API, you only needed 14 lines of code to define, compile, train, and validate a model. You may have noticed that your model performed quite well. In just 10 epochs, we achieved a classification accuracy of over 90% in the validation sample!**

## Exercise

# Overfitting detection

In this exercise, we'll work with a small subset of the examples from the original sign language letters dataset. A small sample, coupled with a heavily-parameterized model, will generally lead to overfitting. This means that your model will simply memorize the class of each example, rather than identifying features that generalize to many examples.

You will detect overfitting by checking whether the validation sample loss is substantially higher than the training sample loss and whether it increases with further training. With a small sample and a high learning rate, the model will struggle to converge on an optimum. You will set a low learning rate for the optimizer, which will make it easier to identify overfitting.

Note that keras has been imported from tensorflow.

## Instructions

* Define a sequential model in keras named model.
* Add a first dense layer with 1024 nodes, a relu activation, and an input shape of (784,).
* Set the learning rate to 0.001.
* Set the fit() operation to iterate over the full sample 50 times and use 50% of the sample for validation purposes.

# Define sequential model

\_\_\_\_

# Define the first layer

\_\_\_\_

# Add activation function to classifier

model.add(keras.layers.Dense(4, activation='softmax'))

# Finish the model compilation

model.compile(optimizer=keras.optimizers.Adam(lr=\_\_\_\_),

              loss='categorical\_crossentropy', metrics=['accuracy'])

# Complete the model fit operation

model.fit(sign\_language\_features, sign\_language\_labels, epochs=\_\_\_\_, validation\_split=\_\_\_\_)

# Define sequential model

model = keras.Sequential()

# Define the first layer

model.add(keras.layers.Dense(1024, activation='relu',input\_shape=(784,)))

# Add activation function to classifier

model.add(keras.layers.Dense(4, activation='softmax'))

# Finish the model compilation

model.compile(optimizer=keras.optimizers.Adam(lr=0.001),

              loss='categorical\_crossentropy', metrics=['accuracy'])

# Complete the model fit operation

model.fit(sign\_language\_features, sign\_language\_labels, epochs=50, validation\_split=0.50)

**# Define sequential model**

**model = keras.Sequential()**

**# Define the first layer**

**model.add(keras.layers.Dense(1024, activation='relu',input\_shape=(784,)))**

**# Add activation function to classifier**

**model.add(keras.layers.Dense(4, activation='softmax'))**

**# Finish the model compilation**

**model.compile(optimizer=keras.optimizers.Adam(lr=0.001),**

**loss='categorical\_crossentropy', metrics=['accuracy'])**

**# Complete the model fit operation**

**model.fit(sign\_language\_features, sign\_language\_labels, epochs=50, validation\_split=0.50)**

**Epoch 1/50**

**1/1 [==============================] - ETA: 0s - loss: 1.3461 - accuracy: 0.3077**

**1/1 [==============================] - 0s 391ms/step - loss: 1.3461 - accuracy: 0.3077 - val\_loss: 3.0842 - val\_accuracy: 0.3846**

**Epoch 2/50**

**1/1 [==============================] - ETA: 0s - loss: 2.4663 - accuracy: 0.2308**

**1/1 [==============================] - 0s 14ms/step - loss: 2.4663 - accuracy: 0.2308 - val\_loss: 4.2784 - val\_accuracy: 0.3846**

**Epoch 3/50**

**1/1 [==============================] - ETA: 0s - loss: 2.1878 - accuracy: 0.6154**

**1/1 [==============================] - 0s 14ms/step - loss: 2.1878 - accuracy: 0.6154 - val\_loss: 5.6272 - val\_accuracy: 0.3077**

**Epoch 4/50**

**1/1 [==============================] - ETA: 0s - loss: 3.6705 - accuracy: 0.3846**

**1/1 [==============================] - 0s 13ms/step - loss: 3.6705 - accuracy: 0.3846 - val\_loss: 4.5968 - val\_accuracy: 0.3077**

**Epoch 5/50**

**1/1 [==============================] - ETA: 0s - loss: 2.5616 - accuracy: 0.6923**

**1/1 [==============================] - 0s 14ms/step - loss: 2.5616 - accuracy: 0.6923 - val\_loss: 4.3052 - val\_accuracy: 0.0769**

**Epoch 6/50**

**1/1 [==============================] - ETA: 0s - loss: 2.3775 - accuracy: 0.6154**

**1/1 [==============================] - 0s 14ms/step - loss: 2.3775 - accuracy: 0.6154 - val\_loss: 2.8193 - val\_accuracy: 0.0769**

**Excellent work! You may have noticed that the validation loss, val\_loss, was substantially higher than the training loss, loss. Furthermore, if val\_loss started to increase before the training process was terminated, then we may have overfitted. When this happens, you will want to try decreasing the number of epochs.**

## Exercise

# Evaluating models

Two models have been trained and are available: large\_model, which has many parameters; and small\_model, which has fewer parameters. Both models have been trained using train\_features and train\_labels, which are available to you. A separate test set, which consists of test\_features and test\_labels, is also available.

Your goal is to evaluate relative model performance and also determine whether either model exhibits signs of overfitting. You will do this by evaluating large\_model and small\_model on both the train and test sets. For each model, you can do this by applying the .evaluate(x, y) method to compute the loss for features x and labels y. You will then compare the four losses generated.

## Instructions

* Evaluate the small model using the train data.
* Evaluate the small model using the test data.
* Evaluate the large model using the train data.
* Evaluate the large model using the test data.

# Evaluate the small model using the train data

small\_train = small\_model.evaluate(\_\_\_\_, \_\_\_\_)

# Evaluate the small model using the test data

small\_test = \_\_\_\_

# Evaluate the large model using the train data

large\_train = large\_model.evaluate(\_\_\_\_, \_\_\_\_)

# Evaluate the large model using the test data

large\_test = \_\_\_\_

# Print losses

print('\n Small - Train: {}, Test: {}'.format(small\_train, small\_test))

print('Large - Train: {}, Test: {}'.format(large\_train, large\_test))

# Evaluate the small model using the train data

small\_train = small\_model.evaluate(train\_features, train\_labels)

# Evaluate the small model using the test data

small\_test = small\_model.evaluate(test\_features, test\_labels)

# Evaluate the large model using the train data

large\_train = large\_model.evaluate(train\_features, train\_labels)

# Evaluate the large model using the test data

large\_test = large\_model.evaluate(test\_features, test\_labels)

# Print losses

print('\n Small - Train: {}, Test: {}'.format(small\_train, small\_test))

print('Large - Train: {}, Test: {}'.format(large\_train, large\_test))

**# Evaluate the small model using the train data**

**small\_train = small\_model.evaluate(train\_features, train\_labels)**

**# Evaluate the small model using the test data**

**small\_test = small\_model.evaluate(test\_features, test\_labels)**

**# Evaluate the large model using the train data**

**large\_train = large\_model.evaluate(train\_features, train\_labels)**

**# Evaluate the large model using the test data**

**large\_test = large\_model.evaluate(test\_features, test\_labels)**

**# Print losses**

**print('\n Small - Train: {}, Test: {}'.format(small\_train, small\_test))**

**print('Large - Train: {}, Test: {}'.format(large\_train, large\_test))**

**1/4 [======>.......................] - ETA: 0s - loss: 0.1738**

**4/4 [==============================] - 0s 588us/step - loss: 0.1698**

**1/4 [======>.......................] - ETA: 0s - loss: 0.3251**

**4/4 [==============================] - 0s 552us/step - loss: 0.2849**

**1/4 [======>.......................] - ETA: 0s - loss: 0.0425**

**4/4 [==============================] - 0s 633us/step - loss: 0.0396**

**1/4 [======>.......................] - ETA: 0s - loss: 0.1414**

**4/4 [==============================] - 0s 649us/step - loss: 0.1454**

**Small - Train: 0.1698155403137207, Test: 0.2848726212978363**

**Large - Train: 0.03957207500934601, Test: 0.14543527364730835**

**Great job! Notice that the gap between the test and train set losses is high for large\_model, suggesting that overfitting may be an issue. Furthermore, both test and train set performance is better for large\_model. This suggests that we may want to use large\_model, but reduce the number of training epochs.**

**LECTURE**

**1. Training models with the Estimators API**

00:00 - 00:08

In this video, we'll take a look at the high level Estimators API, which was elevated in importance in TensorFlow two point zero.

**2. What is the Estimators API?**

00:08 - 00:50

The Estimators API is a high level TensorFlow submodule. Relative to the core, lower-level TensorFlow APIs and the high-level Keras API, model building in the Estimator API is less flexible. This is because it enforces a set of best practices by placing restrictions on model architecture and training. The upside of using the Estimators API is that it allows for faster deployment. Models can be specified, trained, evaluated, and deployed with less code. Furthermore, there are many premade models that can be instantiated by setting a handful of model parameters.

1. 1 Image taken from https://www.tensorflow.org/guide/premade\_estimators

**3. Model specification and training**

00:50 - 01:34

So what does the typical model specification and training process look like in the Estimators API? Well, it starts with the definition of feature columns, which specify the shape and type of your data. Next, you load and transform your data within a function. The output of this function will be a dictionary object of features and your labels. The next step is to define an estimator. In this video, we'll use premade estimators, but you can also define custom estimators with different architectures. Finally, you will train the model you defined. Note that all model objects created through the Estimators API have train, evaluate, and predict operations.

**4. Defining feature columns**

01:34 - 02:08

Let's step through this procedure to get a sense of how it works. We'll first define the feature columns. If we were working with the housing dataset from chapter 2, we might define a numeric feature column for size using feature\_column.numeric\_column. Note that we supplied the dictionary key, "size," to the operation. We will do this for each feature column we create. We may also want a categorical feature column for the number of rooms using feature\_column.categorical\_column\_with\_vocabulary\_list.

**5. Defining feature columns**

02:08 - 02:20

We can then merge these into a list of features columns. Alternatively, if we were using the sign language MNIST dataset, we'd define a list containing a single vector of features.

**6. Loading and transforming data**

02:20 - 02:51

We next need to define a function that transforms our data, puts the features in a dictionary, and returns both the features and labels. Note that we've simply taken three examples from the housing dataset for the sake of illustration. Using them, we've defined a dictionary with the keys "size" and "rooms," which maps to the feature columns we defined. Next, we define a list or array of labels, which give the price of the house in this case, and then return the features and labels.

**7. Define and train a regression estimator**

02:51 - 03:24

We can now define and train the estimator. But before we do that, we have to define what estimator we actually want to train. If we're predicting house prices, we may want to use a deep neural network with a regression head using estimator.DNNRegressor. This allows us to predict a continuous target. Note that all we had to supply was the list of feature columns and the number of nodes in each hidden layer. The rest is handled automatically. We then apply the train function, supply our input function, and train for 20 steps.

**8. Define and train a deep neural network**

03:24 - 03:48

Alternatively, if we want to instead perform a classification task with a deep neural network, we just need to change the estimator to estimator.DNNClassifier, add the number of classes, and then train again. You can also use linear classifiers, boosted trees, and other common options. Just check the TensorFlow Estimators documentation for a complete list.

**9. Let's practice!**

03:48 - 03:57

Estimators might seem confusing initially, but they're very useful once you master them, so let's practice with a few exercises.

## Exercise

# Preparing to train with Estimators

For this exercise, we'll return to the King County housing transaction dataset from chapter 2. We will again develop and train a machine learning model to predict house prices; however, this time, we'll do it using the estimator API.

Rather than completing everything in one step, we'll break this procedure down into parts. We'll begin by defining the feature columns and loading the data. In the next exercise, we'll define and train a premade estimator. Note that feature\_column has been imported for you from tensorflow. Additionally, numpy has been imported as np, and the Kings County housing dataset is available as a pandas DataFrame: housing.

## Instructions

* Complete the feature column for bedrooms and add another numeric feature column for bathrooms. Use bedrooms and bathrooms as the keys.
* Create a list of the feature columns, feature\_list, in the order in which they were defined.
* Set labels to be equal to the price column in housing.
* Complete the bedrooms entry of the features dictionary and add another entry for bathrooms.

# Define feature columns for bedrooms and bathrooms

bedrooms = feature\_column.numeric\_column("\_\_\_\_")

bathrooms = \_\_\_\_

# Define the list of feature columns

feature\_list = [\_\_\_\_, \_\_\_\_]

def input\_fn():

    # Define the labels

    labels = np.array(\_\_\_\_)

    # Define the features

    features = {'bedrooms':np.array(housing['\_\_\_\_']),

                'bathrooms':\_\_\_\_}

    return features, labels

**# Define feature columns for bedrooms and bathrooms**

**bedrooms = feature\_column.numeric\_column("bedrooms")**

**bathrooms = feature\_column.numeric\_column("bathrooms")**

**# Define the list of feature columns**

**feature\_list = [bedrooms, bathrooms]**

**def input\_fn():**

**# Define the labels**

**labels = np.array(housing['price'])**

**# Define the features**

**features = {'bedrooms':np.array(housing['bedrooms']),**

**'bathrooms':np.array(housing['bathrooms'])}**

**return features, labels**

# Define feature columns for bedrooms and bathrooms bedrooms = feature\_column.numeric\_column("bedrooms") bathrooms = feature\_column.numeric\_column("bathrooms") # Define the list of feature columns feature\_list = [bedrooms, bathrooms] def input\_fn(): # Define the labels labels = np.array(housing['price']) # Define the features features = {'bedrooms':np.array(housing['bedrooms']), 'bathrooms':np.array(housing['bathrooms'])} return features, labels

**Excellent work! In the next exercise, we'll use the feature columns and data input function to define and train an estimator.**

## Exercise

# Defining Estimators

In the previous exercise, you defined a list of feature columns, feature\_list, and a data input function, input\_fn(). In this exercise, you will build on that work by defining an estimator that makes use of input data.

## Instructions 1/2

**Use a deep neural network regressor with 2 nodes in both the first and second hidden layers and 1 training step.**

**Modify the code to use a LinearRegressor(), remove the hidden\_units, and set the number of steps to 2.**

# Define the model and set the number of steps

model = estimator.\_\_\_\_(feature\_columns=feature\_list, hidden\_units=[\_\_\_\_,\_\_\_\_])

model.train(input\_fn, steps=\_\_\_\_)

# Define the model and set the number of steps model = estimator.DNNRegressor(feature\_columns=feature\_list, hidden\_units=[2,2]) model.train(input\_fn, steps=1)

# Define the model and set the number of steps

model = estimator.DNNRegressor(feature\_columns=feature\_list, hidden\_units=[2,2])

model.train(input\_fn, steps=1)

**# Define the model and set the number of steps**

**model = estimator.LinearRegressor(feature\_columns=feature\_list)**

**model.train(input\_fn, steps=2)**

**# Define the model and set the number of steps model = estimator.LinearRegressor(feature\_columns=feature\_list) model.train(input\_fn, steps=2)**

**Great work! Note that you have other premade estimator options, such as BoostedTreesRegressor(), and can also create your own custom estimators.**

**LECTURE**

**1. Congratulations!**

00:00 - 00:17

Congratulations! You've now completed this course on the fundamentals of the TensorFlow API in Python. In this final video, we'll review what you've learned, talk about two useful TensorFlow extensions, and then wrap-up with a discussion of the transition to TensorFlow two point zero.

**2. What you learned**

00:17 - 00:56

In chapter 1, you learned low-level, basic, and advanced operations in TensorFlow. You learned how to define and manipulate variables and constants. You also learned the graph-based computational model that underlies TensorFlow and how it can be used to compute gradients and solve arbitrary optimization problems. In chapter 2, you learned how to load and transform data for use in your TensorFlow projects. You also saw how to use predefined and custom loss functions. We ended with a discussion of how to train models, and when and how to divide the training into batches.

**3. What you learned**

00:56 - 01:35

In chapter 3, we moved on to training neural networks. You learned how to define neural network architecture in TensorFlow, both using low-level linear algebra operations and high-level Keras API operations. We talked about how to select activation functions and optimizers, and, ultimately, how to train models. In chapter 4, you learned how to make full use of the Keras API to train models in TensorFlow. We discussed the training and validation process and also introduced the high-level Estimators API, which can be used to streamline the production process.

**4. TensorFlow extensions**

01:35 - 02:44

In addition to what we covered, there are also a two important TensorFlow extensions that did not fit into the course, but may be worthwhile to explore on your own. The first is TensorFlow Hub, which allows users to import pretrained models that can then be used to perform transfer learning. This will be particularly useful when you want to train an image classifier with a small number of images, but want to make use of a feature-extractor trained on a much larger set of different images. TensorFlow Probability is another exciting extension, which is also currently available as a standalone module. One benefit of using TensorFlow Probability is that it provides additional statistical distributions that can be used for random number generation. It also enables you to incorporate trainable statistical distributions into your models. Finally, TensorFlow Probability provides an extended set of optimizers that are commonly used in statistical research. This gives you additional tools beyond what the core TensorFlow module provides.

1. 1 Screenshot from https://tfhub.dev.

**5. TensorFlow 2.0**

02:44 - 03:29

Finally, I will say a few words about the difference between TensorFlow 2 and TensorFlow 1. If you primarily develop in 1, you may have noticed that you do not need to define static graphs or enable eager execution. This is done automatically in 2. Furthermore, TensorFlow 2 has substantially tighter integration with Keras. In fact, the core functionality of the TensorFlow 1 train module is handled by tf.Keras operations in 2. In addition to the centrality of Keras, the Estimators API also plays a more important role in TensorFlow 2. Finally, TensorFlow 2 also allows you to use static graphs, but they are available through the tf.function operation.

1. 1 Screenshot taken from https://www.tensorflow.org/guide/premade\_estimators

**6. Congratulations!**

03:29 - 03:38

Congratulations! You've now completed the course and are ready to begin training your own models in TensorFlow.