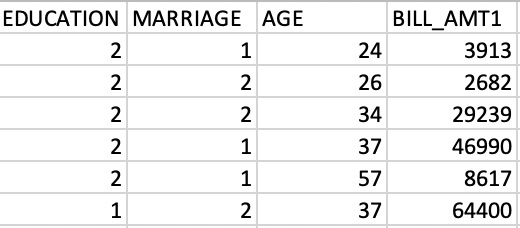
Tensor flow

**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 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)

**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**

**100 XP**

**Instructions**

**100 XP**

* Define a variable, A1, as the 1-dimensional tensor: [1, 2, 3, 4].
* 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)
* # Convert A1 to a numpy array and assign it to B1
* B1 = A1.numpy()
* # Print B1
* print('\n B1: ', B1)

**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:

[1221]⊙[3125]=[3245]

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**

**100 XP**

**Instructions**

**100 XP**

* 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()))

**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.

features=[224226257137], bill=[39132682861764400], params=[1000150]

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**

**100 XP**

**Instructions**

**100 XP**

* 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 = 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())

# 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 = [115072460302510]

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

**35 XP**

##### Instructions

**35 XP**

##### Possible Answers

* 

The individual in the first row has the highest total wealth (i.e. stocks + bonds).

* 

Combined, the 5 individuals hold $50,000 in stocks.

* 

Combined, the 5 individuals hold $50,000 in bonds. -ans

* 

The individual in the second row has the lowest total wealth (i.e. stocks + bonds).

**Submit Answer**

###### Hint

* reduce\_sum(wealth,1) will sum wealth over its rows.
* reduce\_sum(wealth,0) will sum wealth over its columns.
* Note bonds appear first and stocks appear second in reduce\_sum(wealth,0).
* You can append .numpy() to reduce\_sum() to convert the results to a numpy array.
* # 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())
* In [1]:
* reduce\_sum(wealth)
* Out[1]:
* <tf.Tensor: shape=(), dtype=int32, numpy=172>
* In [2]:
* reduce\_sum(wealth,1)
* Out[2]:
* <tf.Tensor: shape=(5,), dtype=int32, numpy=array([61, 9, 64, 3, 35], dtype=int32)>
* In [3]:
* reduce\_sum(wealth,0)
* Out[3]:
* <tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 50, 122], dtype=int32)>
* In [4]:
* reduce\_sum(wealth).numpy()
* Out[4]:
* 172

**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**

**100 XP**

**Instructions**

**100 XP**

* 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

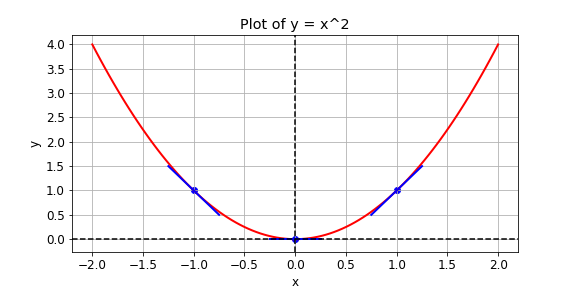
gray\_vector = reshape(gray\_tensor, (784, 1))

# Reshape the color image tensor into a vector

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

**Optimizing with gradients**

You are given a loss function, y=x2, 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**

**100 XP**

**Instructions**

**100 XP**

* 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 = Variable(x0)
* with GradientTape() as tape:
* tape.watch(x)
* # Define y using the multiply operation
* y = 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))

**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**

**100 XP**

* 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 = 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())

**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**

**100 XP**

* 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
* 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'])

**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**

**100 XP**

**Instructions**

**100 XP**

* 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
* 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)

**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 2/2**

**50 XP**

* 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
* from tensorflow import keras
* # Compute the mean absolute error (mae)
* #loss = keras.losses.mse(price, predictions)
* loss = keras.losses.mae(price, predictions)
* # Print the mean absolute error (mae)
* print(loss.numpy())

**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**

**100 XP**

**Instructions**

**100 XP**

* 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 = 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())

**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**

**100 XP**

**Instructions**

**100 XP**

* 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 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())

**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**

**100 XP**

**Instructions**

**100 XP**

* 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.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)

Chart, line chart, scatter chart

Description automatically generated

**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**

**100 XP**

**Instructions**

**100 XP**

* 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\*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)

**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**

**100 XP**

* 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 =  Variable(10.0, float32)
* slope = Variable(0.5, float32)
* # Define the model
* def linear\_regression(intercept, slope, features):
* # Define the predicted values
* return intercept+features\*slope
* # 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)

**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.

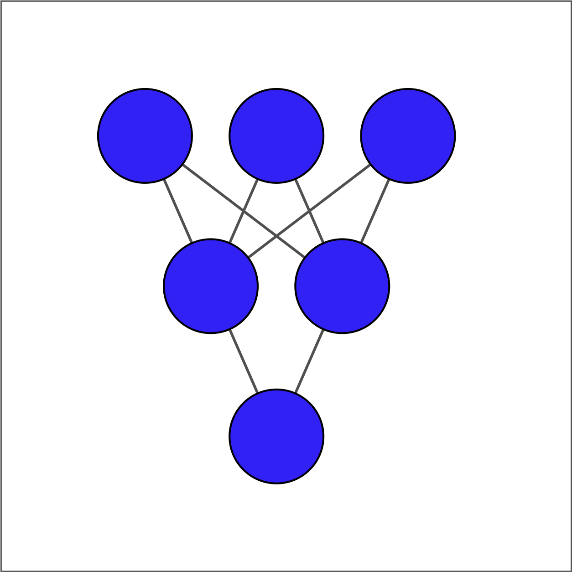
**Instructions**

**100 XP**

* 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.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())

**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**

**50 XP**

* [2](javascript:void(0))

**Instructions 1/2**

**50 XP**

* [2](javascript:void(0))
* 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 = 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 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 = 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')

**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.

products1=[33232124114911492129][−0.60.60.8−0.3−0.09−0.08]

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

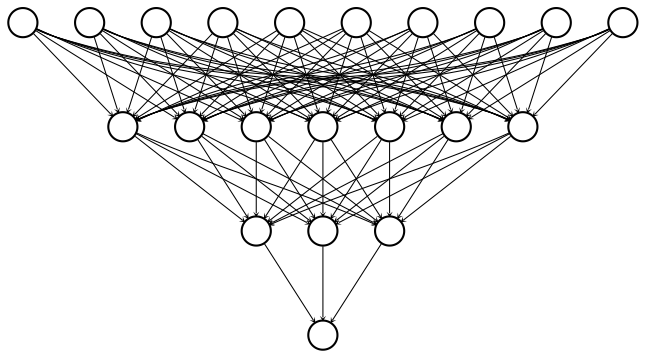
**Instructions**

**100 XP**

* 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 = 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)

**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**

**100 XP**

**Instructions**

**100 XP**

* 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(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)

**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**

**100 XP**

* 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 = 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)

**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.

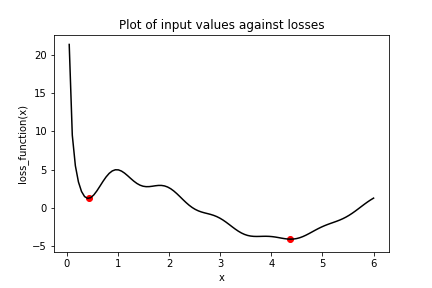
**Instructions**

**100 XP**

* 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 = 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])

**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**

**100 XP**

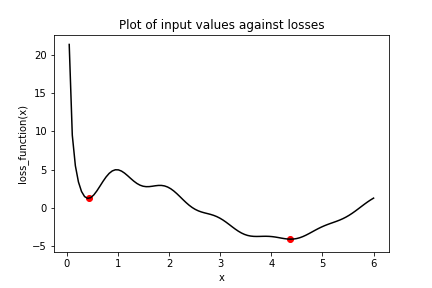
**Instructions**

**100 XP**

* 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.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())

**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**

**100 XP**

* 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=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())

**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**

**100 XP**

* 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 = 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)

**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 σ(layer1∗w2+b2), where σ 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**

**100 XP**

**Instructions**

**100 XP**

* 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.relu(matmul(features, w1) + b1)
* # Apply dropout rate of 0.25
* dropout = keras.layers.Dropout(0.25)(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)

**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**

**100 XP**

* 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.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)

Graphical user interface, chart

Description automatically generated

**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**

**100 XP**

* 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
* 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())

<script.py> output:

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 16) 12560

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 8) 136

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 4) 36

=================================================================

Total params: 12,732

Trainable params: 12,732

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

None

**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**

**100 XP**

* 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(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())

**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**

**100 XP**

**Instructions**

**100 XP**

* 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 (uses sigmoid)
* m1\_layer1 = keras.layers.Dense(12, activation='sigmoid')(m1\_inputs)
* m1\_layer2 = keras.layers.Dense(4, activation='softmax')(m1\_layer1)
* # For model 2, which uses relu,
* 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())

**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**

**100 XP**

**Instructions**

**100 XP**

* 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
* 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: 21s - loss: 1.4813

32/32 [==============================] - 1s 2ms/step - loss: 1.3655

Epoch 2/5

1/32 [..............................] - ETA: 0s - loss: 1.9584

26/32 [=======================>......] - ETA: 0s - loss: 1.1709

32/32 [==============================] - 0s 2ms/step - loss: 1.1445

Epoch 3/5

1/32 [..............................] - ETA: 0s - loss: 1.0988

24/32 [=====================>........] - ETA: 0s - loss: 1.0008

32/32 [==============================] - 0s 2ms/step - loss: 0.9762

Epoch 4/5

1/32 [..............................] - ETA: 0s - loss: 1.1345

32/32 [==============================] - 0s 1ms/step - loss: 0.8624

Epoch 5/5

1/32 [..............................] - ETA: 0s - loss: 0.7748

32/32 [==============================] - 0s 2ms/step - loss: 0.7443

<keras.callbacks.History at 0x7f80ae3c0c18>

**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**

**100 XP**

* 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(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)

**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**

**100 XP**

**Instructions**

**100 XP**

* 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
* 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=0.5)
* <script.py> output:
* Epoch 1/50
* 1/1 [==============================] - ETA: 0s - loss: 1.3461 - accuracy: 0.3077
* 1/1 [==============================] - 1s 1s/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 29ms/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 20ms/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 20ms/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 20ms/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 21ms/step - loss: 2.3775 - accuracy: 0.6154 - val\_loss: 2.8193 - val\_accuracy: 0.0769
* Epoch 7/50
* 1/1 [==============================] - ETA: 0s - loss: 1.5247 - accuracy: 0.6154
* 1/1 [==============================] - 0s 20ms/step - loss: 1.5247 - accuracy: 0.6154 - val\_loss: 1.0500 - val\_accuracy: 0.5385
* Epoch 8/50
* 1/1 [==============================] - ETA: 0s - loss: 0.6675 - accuracy: 0.7692
* 1/1 [==============================] - 0s 21ms/step - loss: 0.6675 - accuracy: 0.7692 - val\_loss: 0.9406 - val\_accuracy: 0.6154
* Epoch 9/50
* 1/1 [==============================] - ETA: 0s - loss: 1.2254 - accuracy: 0.5385
* 1/1 [==============================] - 0s 20ms/step - loss: 1.2254 - accuracy: 0.5385 - val\_loss: 1.1072 - val\_accuracy: 0.4615
* Epoch 10/50
* 1/1 [==============================] - ETA: 0s - loss: 1.6209 - accuracy: 0.5385
* 1/1 [==============================] - 0s 21ms/step - loss: 1.6209 - accuracy: 0.5385 - val\_loss: 0.9477 - val\_accuracy: 0.6923
* Epoch 11/50
* 1/1 [==============================] - ETA: 0s - loss: 1.3329 - accuracy: 0.5385
* 1/1 [==============================] - 0s 24ms/step - loss: 1.3329 - accuracy: 0.5385 - val\_loss: 0.8459 - val\_accuracy: 0.6923
* Epoch 12/50
* 1/1 [==============================] - ETA: 0s - loss: 0.8220 - accuracy: 0.6154
* 1/1 [==============================] - 0s 22ms/step - loss: 0.8220 - accuracy: 0.6154 - val\_loss: 1.0359 - val\_accuracy: 0.5385
* Epoch 13/50
* 1/1 [==============================] - ETA: 0s - loss: 0.5389 - accuracy: 0.8462
* 1/1 [==============================] - 0s 26ms/step - loss: 0.5389 - accuracy: 0.8462 - val\_loss: 1.5257 - val\_accuracy: 0.3846
* Epoch 14/50
* 1/1 [==============================] - ETA: 0s - loss: 0.6948 - accuracy: 0.6923
* 1/1 [==============================] - 0s 22ms/step - loss: 0.6948 - accuracy: 0.6923 - val\_loss: 2.0301 - val\_accuracy: 0.3846
* Epoch 15/50
* 1/1 [==============================] - ETA: 0s - loss: 0.9550 - accuracy: 0.6923
* 1/1 [==============================] - 0s 22ms/step - loss: 0.9550 - accuracy: 0.6923 - val\_loss: 2.0480 - val\_accuracy: 0.3846
* Epoch 16/50
* 1/1 [==============================] - ETA: 0s - loss: 0.9436 - accuracy: 0.6923
* 1/1 [==============================] - 0s 24ms/step - loss: 0.9436 - accuracy: 0.6923 - val\_loss: 1.6047 - val\_accuracy: 0.3846
* Epoch 17/50
* 1/1 [==============================] - ETA: 0s - loss: 0.6777 - accuracy: 0.6923
* 1/1 [==============================] - 0s 25ms/step - loss: 0.6777 - accuracy: 0.6923 - val\_loss: 1.1002 - val\_accuracy: 0.4615
* Epoch 18/50
* 1/1 [==============================] - ETA: 0s - loss: 0.4211 - accuracy: 0.8462
* 1/1 [==============================] - 0s 24ms/step - loss: 0.4211 - accuracy: 0.8462 - val\_loss: 0.8775 - val\_accuracy: 0.6923
* Epoch 19/50
* 1/1 [==============================] - ETA: 0s - loss: 0.4486 - accuracy: 0.8462
* 1/1 [==============================] - 0s 57ms/step - loss: 0.4486 - accuracy: 0.8462 - val\_loss: 0.8258 - val\_accuracy: 0.6923
* Epoch 20/50
* 1/1 [==============================] - ETA: 0s - loss: 0.6064 - accuracy: 0.6154
* 1/1 [==============================] - 0s 24ms/step - loss: 0.6064 - accuracy: 0.6154 - val\_loss: 0.7554 - val\_accuracy: 0.6923
* Epoch 21/50
* 1/1 [==============================] - ETA: 0s - loss: 0.6040 - accuracy: 0.6154
* 1/1 [==============================] - 0s 24ms/step - loss: 0.6040 - accuracy: 0.6154 - val\_loss: 0.6710 - val\_accuracy: 0.7692
* Epoch 22/50
* 1/1 [==============================] - ETA: 0s - loss: 0.4510 - accuracy: 0.7692
* 1/1 [==============================] - 0s 27ms/step - loss: 0.4510 - accuracy: 0.7692 - val\_loss: 0.7021 - val\_accuracy: 0.7692
* Epoch 23/50
* 1/1 [==============================] - ETA: 0s - loss: 0.3448 - accuracy: 1.0000
* 1/1 [==============================] - 0s 26ms/step - loss: 0.3448 - accuracy: 1.0000 - val\_loss: 0.8841 - val\_accuracy: 0.6154
* Epoch 24/50
* 1/1 [==============================] - ETA: 0s - loss: 0.3646 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.3646 - accuracy: 1.0000 - val\_loss: 1.0934 - val\_accuracy: 0.4615
* Epoch 25/50
* 1/1 [==============================] - ETA: 0s - loss: 0.4279 - accuracy: 0.7692
* 1/1 [==============================] - 0s 21ms/step - loss: 0.4279 - accuracy: 0.7692 - val\_loss: 1.1553 - val\_accuracy: 0.4615
* Epoch 26/50
* 1/1 [==============================] - ETA: 0s - loss: 0.4289 - accuracy: 0.6923
* 1/1 [==============================] - 0s 26ms/step - loss: 0.4289 - accuracy: 0.6923 - val\_loss: 1.0227 - val\_accuracy: 0.5385
* Epoch 27/50
* 1/1 [==============================] - ETA: 0s - loss: 0.3458 - accuracy: 0.6923
* 1/1 [==============================] - 0s 24ms/step - loss: 0.3458 - accuracy: 0.6923 - val\_loss: 0.8419 - val\_accuracy: 0.6923
* Epoch 28/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2693 - accuracy: 0.9231
* 1/1 [==============================] - 0s 22ms/step - loss: 0.2693 - accuracy: 0.9231 - val\_loss: 0.7599 - val\_accuracy: 0.6923
* Epoch 29/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2741 - accuracy: 0.9231
* 1/1 [==============================] - 0s 22ms/step - loss: 0.2741 - accuracy: 0.9231 - val\_loss: 0.7528 - val\_accuracy: 0.6923
* Epoch 30/50
* 1/1 [==============================] - ETA: 0s - loss: 0.3189 - accuracy: 0.8462
* 1/1 [==============================] - 0s 27ms/step - loss: 0.3189 - accuracy: 0.8462 - val\_loss: 0.7428 - val\_accuracy: 0.7692
* Epoch 31/50
* 1/1 [==============================] - ETA: 0s - loss: 0.3236 - accuracy: 0.8462
* 1/1 [==============================] - 0s 49ms/step - loss: 0.3236 - accuracy: 0.8462 - val\_loss: 0.7158 - val\_accuracy: 0.6923
* Epoch 32/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2771 - accuracy: 0.9231
* 1/1 [==============================] - 0s 31ms/step - loss: 0.2771 - accuracy: 0.9231 - val\_loss: 0.7075 - val\_accuracy: 0.6923
* Epoch 33/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2321 - accuracy: 1.0000
* 1/1 [==============================] - 0s 53ms/step - loss: 0.2321 - accuracy: 1.0000 - val\_loss: 0.7444 - val\_accuracy: 0.6923
* Epoch 34/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2246 - accuracy: 1.0000
* 1/1 [==============================] - 0s 25ms/step - loss: 0.2246 - accuracy: 1.0000 - val\_loss: 0.8045 - val\_accuracy: 0.6154
* Epoch 35/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2386 - accuracy: 1.0000
* 1/1 [==============================] - 0s 22ms/step - loss: 0.2386 - accuracy: 1.0000 - val\_loss: 0.8353 - val\_accuracy: 0.5385
* Epoch 36/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2408 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.2408 - accuracy: 1.0000 - val\_loss: 0.8052 - val\_accuracy: 0.5385
* Epoch 37/50
* 1/1 [==============================] - ETA: 0s - loss: 0.2196 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.2196 - accuracy: 1.0000 - val\_loss: 0.7360 - val\_accuracy: 0.6923
* Epoch 38/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1944 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.1944 - accuracy: 1.0000 - val\_loss: 0.6752 - val\_accuracy: 0.6923
* Epoch 39/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1877 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.1877 - accuracy: 1.0000 - val\_loss: 0.6398 - val\_accuracy: 0.7692
* Epoch 40/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1952 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.1952 - accuracy: 1.0000 - val\_loss: 0.6147 - val\_accuracy: 0.7692
* Epoch 41/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1948 - accuracy: 1.0000
* 1/1 [==============================] - 0s 20ms/step - loss: 0.1948 - accuracy: 1.0000 - val\_loss: 0.5950 - val\_accuracy: 0.7692
* Epoch 42/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1787 - accuracy: 1.0000
* 1/1 [==============================] - 0s 20ms/step - loss: 0.1787 - accuracy: 1.0000 - val\_loss: 0.5959 - val\_accuracy: 0.7692
* Epoch 43/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1610 - accuracy: 1.0000
* 1/1 [==============================] - 0s 20ms/step - loss: 0.1610 - accuracy: 1.0000 - val\_loss: 0.6252 - val\_accuracy: 0.7692
* Epoch 44/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1552 - accuracy: 1.0000
* 1/1 [==============================] - 0s 22ms/step - loss: 0.1552 - accuracy: 1.0000 - val\_loss: 0.6674 - val\_accuracy: 0.7692
* Epoch 45/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1578 - accuracy: 1.0000
* 1/1 [==============================] - 0s 27ms/step - loss: 0.1578 - accuracy: 1.0000 - val\_loss: 0.6989 - val\_accuracy: 0.7692
* Epoch 46/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1573 - accuracy: 1.0000
* 1/1 [==============================] - 0s 21ms/step - loss: 0.1573 - accuracy: 1.0000 - val\_loss: 0.7078 - val\_accuracy: 0.6923
* Epoch 47/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1495 - accuracy: 1.0000
* 1/1 [==============================] - 0s 19ms/step - loss: 0.1495 - accuracy: 1.0000 - val\_loss: 0.6976 - val\_accuracy: 0.6923
* Epoch 48/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1394 - accuracy: 1.0000
* 1/1 [==============================] - 0s 18ms/step - loss: 0.1394 - accuracy: 1.0000 - val\_loss: 0.6791 - val\_accuracy: 0.7692
* Epoch 49/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1334 - accuracy: 1.0000
* 1/1 [==============================] - 0s 19ms/step - loss: 0.1334 - accuracy: 1.0000 - val\_loss: 0.6602 - val\_accuracy: 0.6923
* Epoch 50/50
* 1/1 [==============================] - ETA: 0s - loss: 0.1323 - accuracy: 1.0000
* 1/1 [==============================] - 0s 19ms/step - loss: 0.1323 - accuracy: 1.0000 - val\_loss: 0.6412 - val\_accuracy: 0.6923

**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**

**100 XP**

* 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(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))
* <script.py> output:
* 1/4 [======>.......................] - ETA: 0s - loss: 0.1738
* 4/4 [==============================] - 0s 2ms/step - loss: 0.1698
* 1/4 [======>.......................] - ETA: 0s - loss: 0.3251
* 4/4 [==============================] - 0s 2ms/step - loss: 0.2849
* 1/4 [======>.......................] - ETA: 0s - loss: 0.0425
* 4/4 [==============================] - 0s 2ms/step - loss: 0.0396
* 1/4 [======>.......................] - ETA: 0s - loss: 0.1414
* 4/4 [==============================] - 0s 2ms/step - loss: 0.1454
* Small - Train: 0.16981548070907593,
* Test: 0.28487256169319153
* Large - Train: 0.03957207500934601, Test: 0.14543524384498596

**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**

**100 XP**

* 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("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

# 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

**100 XP**

* [1](javascript:void(0))

Use a deep neural network regressor with 2 nodes in both the first and second hidden layers and 1 training step.

# 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)

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.LinearRegressor(feature\_columns=feature\_list)

model.train(input\_fn, steps=2)