Table of Contents

[Overcoming Memory Issues 1](#_Toc125489513)

[Hyperparameter Terminology 2](#_Toc125489514)

[Epoch 2](#_Toc125489515)

[Batch Size 2](#_Toc125489516)

[Loss Functions 2](#_Toc125489517)

[For Classification 2](#_Toc125489518)

[1. *Binary Crossentropy* **'binary\_crossentropy'** 2](#_Toc125489519)

[2. *Sparse Categorial Crossentropy* **'sparse\_categorical\_crossentropy'** 2](#_Toc125489520)

[For Regression 3](#_Toc125489521)

[Activation Functions 3](#_Toc125489522)

[1. Sigmoid 'sigmoid' 3](#_Toc125489523)

[2. Tanh 'tanh' 3](#_Toc125489524)

[3. ReLU (Rectified Linear Unit) 'relu' 3](#_Toc125489525)

[4. Leaky ReLU 4](#_Toc125489526)

[5. Softmax 'softmax' 4](#_Toc125489527)

[Optimizers 4](#_Toc125489528)

[1. Adam 'adam' 4](#_Toc125489529)

[2. SGD 'sgd' 4](#_Toc125489530)

[Kernel Initializers 5](#_Toc125489531)

[Gridsearching the Neural Network Hyperparameters 5](#_Toc125489532)

[Least Squares Regression Baseline 5](#_Toc125489533)

[Neural Network Baseline 7](#_Toc125489534)

[Epoch and Batch Size 8](#_Toc125489535)

[Optimizer 11](#_Toc125489536)

[Learning Rate and Other Optimizer Parameters 12](#_Toc125489537)

[Kernel Initializer 13](#_Toc125489538)

[Number of Neurons 15](#_Toc125489539)

[Adding Another Layer 16](#_Toc125489540)

[Final Model 17](#_Toc125489541)

[Additional Tuning Tips 17](#_Toc125489542)

[Neural Network for Classification Tuning 18](#_Toc125489543)

[Grid Searching the Learning Rate for a Classifier 23](#_Toc125489544)

## Overcoming Memory Issues

When working with networks or any data set in this course, if you experience memory issues, some strategies to cope are:

* Cut the data set into a half or a third of the size.
* Use Google Colab or AWS Sagemaker Studio Lab.

## Hyperparameter Terminology

Artificial neural networks have many parameters that need to be configured. This section describes some of the basic hyperparameter terms. The process of configuring hyperparameters is called **tuning**. This section will discuss some of the initial hyperparameters that we will work adjust when tuning.

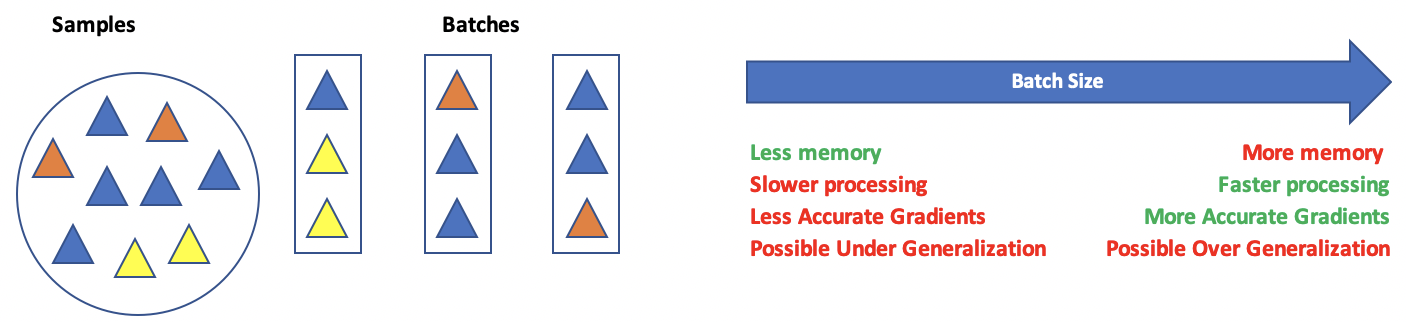
### Epoch

An epoch is an iteration of all sample data through the network.

### Batch Size

The batch size is the number of samples that are processed before the model is updated. Batch size can range from 1 up to the number of samples. Figure 1 compares the advantages and disadvantages of small versus full sample batch sizes.

Figure : Comparing Small and Large Batch Sizes



## Loss Functions

Loss functions help to measure the difference between actual and predicted results. There are many loss function algorithms. The selection relies on the type of output that is desired. For more information see <https://keras.io/api/losses/>

Here are some common loss function options:

### For Classification

### Binary Crossentropy **'binary\_crossentropy'**

This option is used for a single output of 0 or 1.

### Sparse Categorial Crossentropy **'sparse\_categorical\_crossentropy'**

This option is used when categories are expressed as a series of non-binary integers such as:

0, 1, 2, 3

1. Categorical Crossentropy **'categorical\_crossentropy'**

This option is less common but is helpful when output categories are one-hot encoded:

[1,0,0]

[0,1,0]

[0,0,1]

### For Regression

#### Mean squared error.

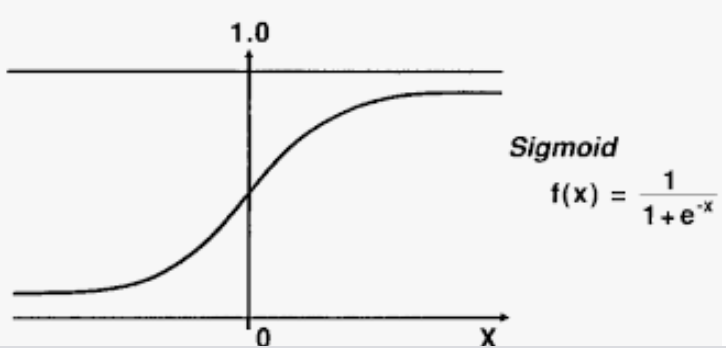
#### Mean absolute error.

<https://keras.io/api/losses/>

## Activation Functions

Activation functions are mathematical equations that determine the output of a neuron. Activation functions of the hidden layer are usually different than the activation function of an output layer. Here are some common activation functions:

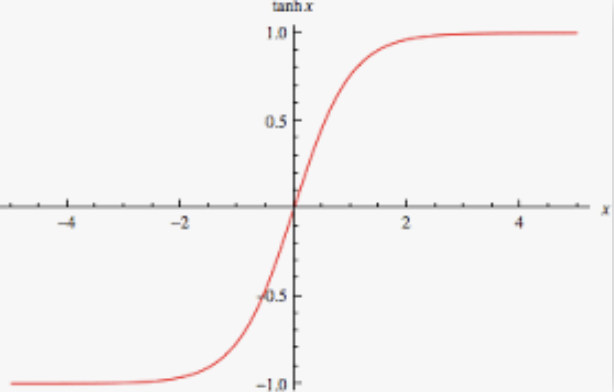
### Sigmoid 'sigmoid'



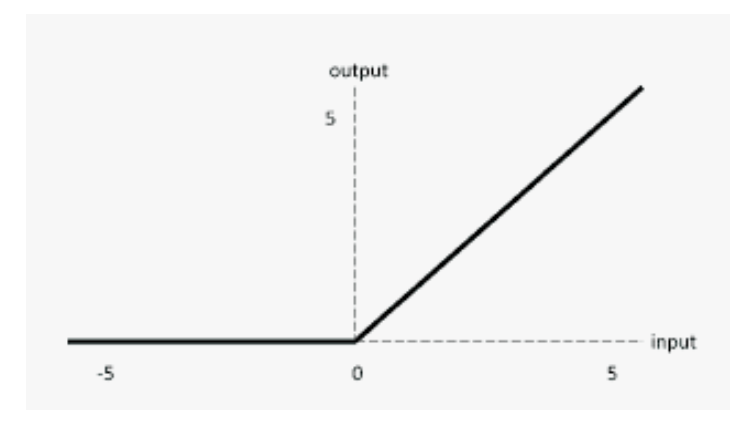
A sigmoid function transforms values to a value between 0 and 1. The sigmoid function can lead to a vanishing gradient problem in deeper networks because really large and really small values are not much different from each other. Computationally, the sigmoid is expensive.

### Tanh 'tanh'

With a tanh function, the outputs are centered about zero. Otherwise the advantages and disadvantages are similar to the sigmoid function.



### ReLU (Rectified Linear Unit) 'relu'



The ReLU is computationally efficient—it allows the network to converge very quickly. The ReLU is non-linear—but it looks like a linear function, ReLU has a derivative function and allows for backpropagation

**The Dying ReLU problem**—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn when this happens.

### Leaky ReLU

|  |  |
| --- | --- |
|  | import tensorflow.keras as tf  lrelu = tf.keras.layers.LeakyReLU(  alpha=0.3  )  LeakyReLU allows a small gradient when neuron not active.  f(x) = alpha \* x if x < 0   f(x) = x if x >= 0 |

The Leaky ReLU (LReLU or LReL) modifies the function to allow small negative values when the input is less than zero. This prevents dying the ReLU problem—this variation of ReLU has a small positive slope in the negative area, so it enables backpropagation, even for negative input values. Results are not consistent though since the leaky ReLU does not provide consistent predictions for negative input values.

### Softmax 'softmax'

Softmax is often used with **output neurons**. Softmax gives the probability of the input value being in a specific class. Softmax is good for classifying outputs in many categories and continuous values.

## Optimizers

Optimizers provide an algorithm to adjust **weights and bias** to reduce the losses in the network. Common optimizers are: Adadelta, Adagrad, Adam, Adamax, Ftrl, Nadam, RMSprop, and SGD. Often, optimizers can be implemented by name but often they can be further customized with additional parameters for items such as learning rates. For example, here are custom configurations for the adam and sgd optimizers:

### Adam 'adam'

tf.keras.optimizers.Adam(  
    learning\_rate=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-07,

amsgrad=False, name='Adam', \*\*kwargs  
)

### SGD 'sgd'

Updates with SGD can fluctuate with significant volatility. Momentum is a time-based weighting that helps to smooth the updates. Nesterov’s Accelerated Gradient Descent is an additional option that can sometimes eliminate volatility.

tf.keras.optimizers.SGD(  
    learning\_rate=0.01, momentum=0.0, nesterov=False, name='SGD', \*\*kwargs  
)

## Kernel Initializers

Kernel initializers set the algorithm which initializes weight values in each neuron. The aim of weight initialization is to prevent neuron gradients from exploding or vanishing. In other words, neurons cannot be revived if their gradients become too large or if the gradient is 0. The network may never reach an optimum if too many neuron gradients explode or vanish. Some initializer options include 'uniform', 'lecun\_uniform', 'normal', 'zero', 'glorot\_normal', 'glorot\_uniform', 'he\_normal', 'he\_uniform'.

# Gridsearching the Neural Network Hyperparameters

KerasClassifier and KerasRegressor libraries appear to be deprecated for the grid search pipeline so I will perform manual grid searches instead.

## Least Squares Regression Baseline

When tuning your network hyperparameters, I recommend that you first build a basic logistic regression model or a simple least squares regression model. This model can help to identify significant predictor variables. The simple model can also help to suggest a reasonable level of metrics which can be used as a baseline for your artificial neural network.

Example : Building a Baseline Least Squares Regression Model

This example generates prediction for housing price which is located in last column on the right.

|  |
| --- |
| 0 1 2 3 4 5 6 7 8 9 10 11 12 13  0 0.00632 18.0 2.31 0 0.538 6.575 65.2 4.0900 1 296.0 15.3 396.90 4.98 24.0  1 0.02731 0.0 7.07 0 0.469 6.421 78.9 4.9671 2 242.0 17.8 396.90 9.14 21.6  2 0.02729 0.0 7.07 0 0.469 7.185 61.1 4.9671 2 242.0 17.8 392.83 4.03 34.7  3 0.03237 0.0 2.18 0 0.458 6.998 45.8 6.0622 3 222.0 18.7 394.63 2.94 33.4  4 0.06905 0.0 2.18 0 0.458 7.147 54.2 6.0622 3 222.0 18.7 396.90 5.33 36.2 |

In this initial OLS example, coefficients 3 and 7 are insignificant. The value of 0.765 indicates a strong correlation between the price and predictor variables. The root mean-square error, RMSE, value of 5.21 provides a decent understanding of the deviation about the mean which can be used as a comparison for other models. Here is the output when running the code for this example:

|  |
| --- |
| OLS Regression Results  OLS Regression Results  ==============================================================================  Dep. Variable: y R-squared: 0.765  Model: OLS Adj. R-squared: 0.756  Method: Least Squares F-statistic: 84.92  Date: Tue, 24 Jan 2023 Prob (F-statistic): 2.76e-98  Time: 17:26:56 Log-Likelihood: -1032.2  No. Observations: 354 AIC: 2092.  Df Residuals: 340 BIC: 2147.  Df Model: 13  Covariance Type: nonrobust  ==============================================================================  coef std err t P>|t| [0.025 0.975]  ------------------------------------------------------------------------------  const 37.9371 5.876 6.457 0.000 26.380 49.494  x1 -0.1213 0.038 -3.185 0.002 -0.196 -0.046  x2 0.0445 0.016 2.834 0.005 0.014 0.075  x3 0.0113 0.068 0.166 0.868 -0.123 0.146  x4 2.5112 1.011 2.483 0.014 0.522 4.501  x5 -16.2313 4.487 -3.618 0.000 -25.056 -7.406  x6 3.8591 0.491 7.867 0.000 2.894 4.824  x7 -0.0100 0.016 -0.634 0.527 -0.041 0.021  x8 -1.5003 0.234 -6.402 0.000 -1.961 -1.039  x9 0.2421 0.077 3.152 0.002 0.091 0.393  x10 -0.0111 0.004 -2.586 0.010 -0.019 -0.003  x11 -1.0178 0.150 -6.803 0.000 -1.312 -0.723  x12 0.0068 0.003 2.099 0.037 0.000 0.013  x13 -0.4867 0.060 -8.068 0.000 -0.605 -0.368  ==============================================================================  Omnibus: 133.612 Durbin-Watson: 2.019  Prob(Omnibus): 0.000 Jarque-Bera (JB): 634.086  Skew: 1.547 Prob(JB): 2.04e-138  Kurtosis: 8.781 Cond. No. 1.52e+04  ============================================================================== |

Here is the code for generating the least squares regression:

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn import metrics  from sklearn.model\_selection import train\_test\_split  import statsmodels.api as sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(PATH + CSV\_DATA, header=None)  # Show all columns on one line.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head())  print(df.tail())  print(df.describe())  dataset = df.values  # Split data into input (X) and output (Y) variables.  X = dataset[:,0:13]  y = dataset[:,13]  # Adding an intercept \*\*\* This is required \*\*\*. Don't forget this step.  # The intercept centers the error residuals around zero  # which helps to avoid over-fitting.  X = sm.add\_constant(X)  # Split the data.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X,  y, test\_size=0.3, random\_state=0)  X\_test, X\_val, y\_test, y\_val = train\_test\_split(X\_temp,  y\_temp, test\_size=0.5, random\_state=0)  # Make predictions and evaluate with the RMSE.  model = sm.OLS(y\_train, X\_train).fit()  predictions = model.predict(X\_test)  print(model.summary())  print('Root Mean Squared Error:',  np.sqrt(metrics.mean\_squared\_error(y\_test, predictions))) |

## Neural Network Baseline

The next step involves creating a simple baseline neural network. I recommend that you begin by building a 1-layer network. You may decide later to start with more layers. For now, at least understand the grid search technique before designing more complicated solutions.

Example : Neural Network Baseline Housing Data

Right now, our crude baseline neural network model does not perform as well as the linear regression model. With some tuning it is very possible that we can generate a better fitting model than OLS.

|  |  |
| --- | --- |
| **OLS RMSE from Example 1** | **RMSE from Neural Network** |
| 5.2133376653973365 | 4.936674345497767 |

Here is the code.

**Note:** Remember to reshape your X and Y data into vertical columns.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn import metrics  from sklearn.model\_selection import train\_test\_split  from keras.models import Sequential  from keras.layers import Dense  # Read the data.  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(PATH + CSV\_DATA, header=None)  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head())  print(df.tail())  print(df.describe())  # Convert DataFrame columns to vertical columns so they can be used by the NN.  dataset = df.values  X = dataset[:, 0:13] # Columns 0 to 12  y = dataset[:, 13] # Columns 13  ROW\_DIM = 0  COL\_DIM = 1  x\_arrayReshaped = X.reshape(X.shape[ROW\_DIM], X.shape[COL\_DIM])  y\_arrayReshaped = y.reshape(y.shape[ROW\_DIM],1)  # Split the data.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.3, random\_state=0)  X\_test, X\_val, y\_test, y\_val = train\_test\_split(X\_temp,  y\_temp, test\_size=0.5, random\_state=0)  # Define the model.  def create\_model():  model = Sequential()  model.add(Dense(13, input\_dim=13, kernel\_initializer='normal',  activation='relu'))  model.add(Dense(1, kernel\_initializer='normal'))  model.compile(loss='mean\_squared\_error', optimizer='adam')  return model  model = create\_model()  # Build the model.  model = create\_model()  history = model.fit(X\_train, y\_train, epochs=100,  batch\_size=5, verbose=1,  validation\_data=(X\_val, y\_val))  # Evaluate the model.  predictions = model.predict(X\_test)  mse = metrics.mean\_squared\_error(y\_test, predictions)  print("Neural network MSE: " + str(mse))  print("Neural network RMSE: " + str(np.sqrt(mse))) |

## Epoch and Batch Size

Now we can start grid searching our hyperparameters ideally to minimize the RMSE the house price predictions more.

Example : Epoch and Batch Tuning

This example manually grid searches batch size and epochs.

My trial showed that a batch of 10 and 100 epochs were most optimal. However, results will vary. It would be much better to use cross fold validation to evaluate the model over multiple runs.

|  |
| --- |
| rmse epochs batch  1 5.088442 100 10  2 5.286852 200 10  5 5.627772 200 60  0 5.853373 50 10  8 6.675932 200 100  4 7.190926 100 60  7 7.759760 100 100  6 8.367856 50 100  3 8.792805 50 60 |

Here is the code:

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error  from keras.models import Sequential  from keras.layers import Dense  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(PATH + CSV\_DATA, header=None)  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head())  print(df.tail())  print(df.describe())  dataset = df.values  # split into input (X) and output (Y) variables  X = dataset[:,0:13]  y = dataset[:,13]  # Split the data.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X,  y, test\_size=0.3, random\_state=0)  X\_test, X\_val, y\_test, y\_val = train\_test\_split(X\_temp,  y\_temp, test\_size=0.5, random\_state=0)  def evaluateModel(model, X\_test, y\_test):  predictions = model.predict(X\_test)  rmse = np.sqrt(mean\_squared\_error(y\_test, predictions))  print("RMSE: " + str(rmse))  return rmse  def showResults(networkStats):  dfStats = pd.DataFrame.from\_records(networkStats)  dfStats = dfStats.sort\_values(by=['rmse'])  print(dfStats)  networkStats = []  ### Model parameters ############################  batch\_sizes = [10, 60, 100]  epochList = [50, 100, 200]  #################################################  #################################################  # Build model  def create\_model():  model = Sequential()  model.add(Dense(13, input\_dim=13, kernel\_initializer='normal',  activation='relu'))  model.add(Dense(1, kernel\_initializer='normal'))  model.compile(loss='mean\_squared\_error', optimizer='adam')  return model  for batch\_size in batch\_sizes:  for epochs in epochList:  model = create\_model()  history = model.fit(X\_train, y\_train, epochs=epochs,  batch\_size=batch\_size, verbose=1,  validation\_data=(X\_val, y\_val))  rmse = evaluateModel(model, X\_test, y\_test)  networkStats.append({"rmse":rmse, "epochs":epochs, "batch":batch\_size})  showResults(networkStats)  ################################################# |

## Optimizer

This section shows a strategy for grid searching the general optimizer. A more advanced version of each optimizer likely exists. It is usually possible to tweak the learning rate and other parameters that are specific to each optimizer which once an optimizer candidate is found.

Example : Searching for the most efficient Optimizer

Now that we have established a batch size of 10 and 100 epochs is a suitable estimate, we can look for an optimizer. The output suggests that the adam optimizer is best. The ‘adam’ optimizer will be used as a result for the model in all future examples.

|  |
| --- |
| rmse optimizer  1 5.036573 Adam  6 5.272861 Nadam  4 5.346332 RMSprop  5 7.874382 Adamax  2 9.605278 Adagrad  0 9.794729 SGD  3 18.393395 Adadelta |

To build this example, swap out the “Model parameters” and “Build model” sections of Example 3 with the following code:

|  |
| --- |
| ### Model parameters ############################  optimizers = ['SGD', 'RMSprop', 'Adagrad',  'Adadelta', 'Adam', 'Adamax', 'Nadam']  #################################################  #################################################  # Build model  def create\_model(optimizer="SGD"):  model = Sequential()  model.add(Dense(13, input\_dim=13, kernel\_initializer='normal',  activation='relu'))  model.add(Dense(1, kernel\_initializer='normal'))  model.compile(loss='mean\_squared\_error', optimizer=optimizer)  return model  for optimizer in optimizers:  BATCH\_SIZE = 10  EPOCHS = 100  model = create\_model(optimizer)  history = model.fit(X\_train, y\_train, epochs=EPOCHS,  batch\_size=BATCH\_SIZE, verbose=1,  validation\_data=(X\_val, y\_val))  rmse = evaluateModel(model, X\_test, y\_test)  networkStats.append({"rmse":rmse, "optimizer":optimizer})  showResults(networkStats)  ################################################# |

Exercise (1 mark)

Choose the best answer.

1. An optimizer sets an algorithm to initialize weights for each neuron.
2. An optimizer sets an algorithm for scaling data in each neuron.
3. An optimizer sets an algorithm for activation.
4. An optimizer sets an algorithm for minimizing losses.

## Learning Rate and Other Optimizer Parameters

Most optimizers have more advanced customizable settings. Usually you can adjust the learning rate and other parameters. Documentation for several common optimizers can be found at <https://keras.io/api/optimizers/>

Example : Optimizing the Learning Rate

Since the ‘adam’ optimizer was identified as a strong performer in the previous example, we can fine tune it more by adjusting its learning rate. The ‘adam’ optimizer actually has many more parameters but we will focus on only the learning rate for this example:

|  |
| --- |
| tf.keras.optimizers.Adam(  learning\_rate=0.001  ) |

The output when running the grid search for the best learning rate suggest a learning rate of 0.005 for our optimizer.

|  |
| --- |
| rmse learningRate  2 4.992756 0.005  1 5.041506 0.010  0 5.465462 0.001  3 5.551813 0.015  4 9.798956 0.200 |

To build this example replace the “Model parameters” and “Build model” sections of Example 4 with the following:

|  |
| --- |
| ### Model parameters ############################  learningRates = [0.001, 0.005, 0.01, 0.015, 0.2]  #################################################  #################################################  # Build model  import tensorflow as tf  def create\_model(learningRate=0.001):  model = Sequential()  model.add(Dense(13, input\_dim=13, kernel\_initializer='normal',  activation='relu'))  model.add(Dense(1, kernel\_initializer='normal'))  # Use Adam optimizer with the given learning rate  optimizer = tf.keras.optimizers.Adam(lr=learningRate)  model.compile(loss='mean\_squared\_error', optimizer=optimizer)  return model  for learningRate in learningRates:  BATCH\_SIZE = 10  EPOCHS = 100  model = create\_model(learningRate)  history = model.fit(X\_train, y\_train, epochs=EPOCHS,  batch\_size=BATCH\_SIZE, verbose=1,  validation\_data=(X\_val, y\_val))  rmse = evaluateModel(model, X\_test, y\_test)  networkStats.append({"rmse": rmse, "learningRate": learningRate})  showResults(networkStats) |

Exercise (1 mark)

How is the parameter of the create\_model() function in Example 5 different than the parameter in the create\_model() function of Example 4?

|  |
| --- |
| Example 4s parameter took in an optimizer. This time we are taking in a specific floating value of learning rate. |

Exercise (3 marks)

Manually plug-in a batch size of 10 into the code within Example 2. Ensure that the model is tuned with 100 epochs. Inside create\_model() initialize an adam optimizer with a learning rate of 0.005. Assign this optimizer object to the optimizer parameter in create\_model(). Show the RMSE that results after making these changes:

|  |
| --- |
|  |

What is the RMSE of the original configuration from Example 2?

|  |
| --- |
|  |

Show the section of code that you modified here.

|  |
| --- |
|  |

## Kernel Initializer

Kernel initializers use a distribution to optimize the initial model weights. The different algorithms for initializing weight can sometimes help to prevent vanishing and exploding gradients which destroy a neuron’s ability to back-propagate data.

Example : Grid Searching the Kernel Initializer

This section shows that some initializers perform much better than others. ‘uniform’ offers the best score and it is noticeably smaller than the ‘zero’ initializer.

|  |
| --- |
| rmse initializer  2 5.141459 uniform  7 5.147688 he\_uniform  5 5.500346 glorot\_uniform  0 5.506580 normal  6 5.556400 he\_normal  1 5.840333 lecun\_uniform  4 6.555245 glorot\_normal  3 11.799726 zero |

To build this example replace the “Model parameters” and “Build model” sections of the last example with this version.

|  |
| --- |
| ### Model parameters ############################  init\_modes = ['uniform', 'lecun\_uniform', 'normal', 'zero',  'glorot\_normal',  'glorot\_uniform', 'he\_normal', 'he\_uniform']  #################################################  #################################################  # Build model  import tensorflow as tf  def create\_model(inialtizer = 'normal'):  model = Sequential()  model.add(Dense(13, input\_dim=13, kernel\_initializer=inialtizer,  activation='relu'))  model.add(Dense(1, kernel\_initializer=inialtizer))  # Use Adam optimizer with the given learning rate  LEARNING\_RATE = 0.005  optimizer = tf.keras.optimizers.Adam(lr=LEARNING\_RATE)  model.compile(loss='mean\_squared\_error', optimizer=optimizer)  return model  for initializer in init\_modes:  BATCH\_SIZE = 10  EPOCHS = 100  model = create\_model(initializer)  history = model.fit(X\_train, y\_train, epochs=EPOCHS,  batch\_size=BATCH\_SIZE, verbose=1,  validation\_data=(X\_val, y\_val))  rmse = evaluateModel(model, X\_test, y\_test)  networkStats.append({"rmse":rmse, "initializer":initializer})  showResults(networkStats)  ################################################# |

Exercise (3 marks)

Manually plug-in a batch size of 10 into the code within Example 2. Ensure that the model is tuned with 100 epochs. Inside create\_model() initialize an adam optimizer with a learning rate of 0.005. Assign this optimizer object to the optimizer parameter in create\_model(). Set the kernel initializer to ‘uniform’. Show the RMSE that results after making these changes:

|  |
| --- |
|  |

What is the RMSE of the original configuration from Example 2?

|  |
| --- |
|  |

Show the section of code that you modified here.

|  |
| --- |
|  |

## Number of Neurons

Now we can examine the optimal number of neurons for the initial hidden layer.

Example : Number of Neurons Tuning

The output suggests that 25 neurons is optimal.

|  |
| --- |
| rmse # neurons  2 4.928856 25  1 4.959842 50  3 5.103180 100  4 5.173724 150  0 6.875537 5 |

To build this example replace the “Model parameters” and “Build model” sections of the last example with this version.

|  |
| --- |
| ### Model parameters ############################  neuronList = [5, 25, 50, 100, 150]  #################################################  #################################################  # Build model  import tensorflow as tf  def create\_model(numNeurons):  model = Sequential()  model.add(Dense(numNeurons,  input\_dim=13, kernel\_initializer='uniform',  activation='relu'))  model.add(Dense(1, kernel\_initializer='uniform'))  # Use Adam optimizer with the given learning rate  LEARNING\_RATE = 0.005  optimizer = tf.keras.optimizers.Adam(lr=LEARNING\_RATE)  model.compile(loss='mean\_squared\_error', optimizer=optimizer)  return model  for numNeurons in neuronList:  BATCH\_SIZE = 10  EPOCHS = 100  model = create\_model(numNeurons)  history = model.fit(X\_train, y\_train, epochs=EPOCHS,  batch\_size=BATCH\_SIZE, verbose=1,  validation\_data=(X\_val, y\_val))  rmse = evaluateModel(model, X\_test, y\_test)  networkStats.append({"rmse":rmse, "# neurons":numNeurons})  showResults(networkStats)  ################################################# |

## Adding Another Layer

By now, you may realize just how many possible hyperparameter combinations exist. We could tinker with the algorithm further by adding another layer. You actually have the ability to include as many hyperparameters as you want in a grid search but a significant constraint is time so be practical about how many options that you tune at once.

The outcome suggests that two additional layers could improve results.

|  |
| --- |
| rmse # additional layers  2 4.761330 2  0 4.762825 0  5 5.143485 5  4 5.292722 4  1 5.308366 1  3 5.422089 3 |

To build this example replace the “Model parameters” and “Build model” sections of the last example with this version.

|  |
| --- |
| ### Model parameters ############################  additionalLayers = [0, 1, 2, 3, 4, 5]  #################################################  #################################################  # Build model  import tensorflow as tf  def create\_model(numExtraLayers):  NUM\_NEURONS = 25  model = Sequential()  model.add(Dense(NUM\_NEURONS,  input\_dim=13, kernel\_initializer='uniform',  activation='relu'))  for i in range(0, numExtraLayers):  # You could further grid search initializer, num\_neurons  # and activation function for each layer if desired.  model.add(Dense(NUM\_NEURONS,kernel\_initializer='uniform',  activation='relu'))  model.add(Dense(1, kernel\_initializer='uniform'))  # Use Adam optimizer with the given learning rate  LEARNING\_RATE = 0.005  optimizer = tf.keras.optimizers.Adam(learning\_rate=LEARNING\_RATE)  model.compile(loss='mean\_squared\_error', optimizer=optimizer)  return model  for numLayers in additionalLayers:  BATCH\_SIZE = 10  EPOCHS = 100  model = create\_model(numLayers)  history = model.fit(X\_train, y\_train, epochs=EPOCHS,  batch\_size=BATCH\_SIZE, verbose=1,  validation\_data=(X\_val, y\_val))  rmse = evaluateModel(model, X\_test, y\_test)  networkStats.append({"rmse":rmse, "# additional layers":numLayers})  showResults(networkStats)  ################################################# |

## Final Model

Example : Building the Final Model

The final model appears to be the best yet. Results will vary though since the data is randomized. For best results always evaluate models over multiple runs such as with cross fold validation.

|  |  |
| --- | --- |
| Model | RMSE |
| OLS Model | 5.21 |
| Baseline Neural Network | 4.94 |
| Optimized Neural Network | 4.76 |

## Additional Tuning Tips

This section lists some helpful tips to consider when tuning hyperparameters of your neural network. Pay attention to set size.

* **k-fold Cross Validation**. The network is trained with random subsets of data at each epoch so results will vary. Evaluating your models with cross fold validation over multiple runs will help to provide more stable evaluations.
* **Fine Tune**. Do not just focus on the best result, review the whole grid of results and look for trends to support configuration fine-tuning decisions.
* **Use a Sample of Your Dataset**. Because networks are slow to train, try training them on a smaller sample of your training dataset, just to get an idea of general directions of parameters rather than optimal configurations.

## Neural Network for Classification Tuning

Example : Grid Searching Batch Size and Number of Epochs

This example shows a way to grid search batch size and number of epochs for classification.

The logistic regression model shows the following statistics:

|  |
| --- |
| * **Precision**: 0.94 **Recall**: 0.70. **F1**: 0.80 |

The network does quite a bit better. Since 200 epochs produces the same results at 300 epochs I will go with 200 epochs to save time. Of course, there is room for fine-tuning and more reliable evaluation over multiple runs.

|  |
| --- |
| precision recall f1 epochs batch  6 0.941176 0.695652 0.800000 100 100  7 0.944444 0.739130 0.829268 200 100  4 1.000000 0.826087 0.904762 200 60  0 0.952381 0.869565 0.909091 100 10  2 0.952381 0.869565 0.909091 300 10  8 1.000000 0.869565 0.930233 300 100  3 0.954545 0.913043 0.933333 100 10  1 1.000000 0.913043 0.954545 200 60  5 1.000000 0.913043 0.954545 300 60 |

Here is the code:

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn.metrics import precision\_score, recall\_score, f1\_score,\  accuracy\_score, classification\_report  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "Social\_Network\_Ads.csv"  data = pd.read\_csv(PATH + FILE)  y = data["Purchased"]  X = data.copy()  del X['User ID']  del X['Purchased']  X['Gender'] = X['Gender'].map({'Male': 0, 'Female': 1})  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(data.head())  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(  X, y, test\_size=0.3, random\_state=42  )  X\_test, X\_val, y\_test, y\_val = train\_test\_split(  X\_temp, y\_temp, test\_size=0.5, random\_state=42  )  # Stochastic gradient descent models are sensitive to differences  from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  scaler.fit(X\_train)  X\_trainScaled = scaler.transform(X\_train)  X\_testScaled = scaler.transform(X\_test)  X\_valScaled = scaler.transform(X\_val)  def showResults(networkStats):  dfStats = pd.DataFrame.from\_records(networkStats)  dfStats = dfStats.sort\_values(by=['f1'])  print(dfStats)  def evaluate\_model(predictions, y\_test):  precision = precision\_score(y\_test, predictions)  recall = recall\_score(y\_test, predictions)  f1 = f1\_score(y\_test, predictions)  print("Precision: " + str(precision) + " " +\  "Recall: " + str(recall) + " " +\  "F1: " + str(f1))  return precision, recall, f1  clf = LogisticRegression(max\_iter=1000)  clf.fit(X\_trainScaled, y\_train)  predictions = clf.predict(X\_testScaled)  evaluate\_model(predictions, y\_test)  COLUMN\_DIMENSION = 1  #######################################################################  # Part 2  from keras.models import Sequential  from keras.layers import Dense  # shape() obtains rows (dim=0) and columns (dim=1)  n\_features = X\_trainScaled.shape[COLUMN\_DIMENSION]  def getPredictions(model, X\_test):  probabilities = model.predict(X\_test)  predictions = []  for i in range(len(probabilities)):  if (probabilities[i][0] > 0.5):  predictions.append(1)  else:  predictions.append(0)  return predictions  ### Model parameters ############################  batch\_sizes = [10, 60, 100]  epochList = [100, 200, 300]  #################################################  #######################################################################  # Model building section.  def create\_model():  model = Sequential()  model.add(Dense(12, input\_dim=n\_features, activation='relu'))  model.add(Dense(1, activation='sigmoid'))  model.compile(loss='binary\_crossentropy', optimizer='adam',  metrics=['accuracy'])  return model  networkStats = []  for batch\_size in batch\_sizes:  for epochs in epochList:  model = create\_model()  history = model.fit(X\_trainScaled, y\_train, epochs=epochs,  batch\_size=batch\_size, verbose=1,  validation\_data=(X\_valScaled, y\_val))  predictions = getPredictions(model, X\_testScaled)  precision, recall, f1 = evaluate\_model(predictions, y\_test)  networkStats.append({"precision":precision, "recall":recall,  "f1":f1, "epochs":epochs,  "batch":batch\_size})  showResults(networkStats)  ####################################################################### |

Exercise (1 mark)

How many input dimensions are used for the input layer in Example 9?

|  |
| --- |
| 3 based on this input    User ID Gender Age EstimatedSalary Purchased  0 15624510 Male 19 19000 0  1 15810944 Male 35 20000 0  2 15668575 Female 26 43000 0  3 15603246 Female 27 57000 0  4 15804002 Male 19 76000 0  and because USER ID and Purchased are removed meaning we are only working 3 features |

Exercise (1 mark)

The values for batch or epochs may be fined tuned. Suggest a better pairing for number of epochs and batch size for Example 9. Show the model parameters section after you adjust the range of options here:

|  |
| --- |
| Updated Model parameters |

Show the final results and highlight the optimal combination that is revealed here:

|  |
| --- |
| precision recall f1 epochs batch  18 0.909091 0.434783 0.588235 50 90  24 0.666667 0.608696 0.636364 50 100  31 0.933333 0.608696 0.736842 100 120  14 0.941176 0.695652 0.800000 150 60  19 0.941176 0.695652 0.800000 100 90  20 0.941176 0.695652 0.800000 150 90  30 0.941176 0.695652 0.800000 50 120  25 0.941176 0.695652 0.800000 100 100  32 0.941176 0.695652 0.800000 150 120  13 0.944444 0.739130 0.829268 100 60  7 0.944444 0.739130 0.829268 100 30  21 1.000000 0.739130 0.850000 200 90  26 0.947368 0.782609 0.857143 150 100  12 0.947368 0.782609 0.857143 50 60  33 0.947368 0.782609 0.857143 200 120  6 1.000000 0.782609 0.878049 50 30  0 0.950000 0.826087 0.883721 50 10  23 0.952381 0.869565 0.909091 300 90  3 0.952381 0.869565 0.909091 200 10  4 0.952381 0.869565 0.909091 250 10  11 0.952381 0.869565 0.909091 300 30  34 0.952381 0.869565 0.909091 250 120  5 0.952381 0.869565 0.909091 300 10  1 1.000000 0.869565 0.930233 100 10  35 1.000000 0.869565 0.930233 300 120  2 0.954545 0.913043 0.933333 150 10  28 0.954545 0.913043 0.933333 250 100  15 0.954545 0.913043 0.933333 200 60  9 1.000000 0.913043 0.954545 200 30  8 1.000000 0.913043 0.954545 150 30  10 1.000000 0.913043 0.954545 250 30  17 1.000000 0.913043 0.954545 300 60  29 1.000000 0.913043 0.954545 300 100  27 1.000000 0.913043 0.954545 200 100  16 1.000000 0.913043 0.954545 250 60  22 1.000000 0.913043 0.954545 250 90  150 epochs, 30 batchs are the best pair as it performs just as good with the higher end epochs and batches but its faster and has high scores |

Example : Grid Searching the Classifier for an Optimizer

This code grid searches an optimizer for the classification model. Here are the rankings. RMSprop appears to be the winner.

|  |
| --- |
| precision recall f1 optimizer  3 0.294118 0.434783 0.350877 Adadelta  5 0.941176 0.695652 0.800000 Adamax  2 0.800000 0.869565 0.833333 Adagrad  0 0.947368 0.782609 0.857143 SGD  4 0.947368 0.782609 0.857143 Adam  6 0.954545 0.913043 0.933333 Nadam  1 1.000000 0.913043 0.954545 RMSprop |

This is the code:

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn.metrics import precision\_score, recall\_score, f1\_score,\  accuracy\_score, classification\_report  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "Social\_Network\_Ads.csv"  data = pd.read\_csv(PATH + FILE)  y = data["Purchased"]  X = data.copy()  del X['User ID']  del X['Purchased']  X['Gender'] = X['Gender'].map({'Male': 0, 'Female': 1})  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(data.head())  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(  X, y, test\_size=0.3, random\_state=42  )  X\_test, X\_val, y\_test, y\_val = train\_test\_split(  X\_temp, y\_temp, test\_size=0.5, random\_state=42  )  # Stochastic gradient descent models are sensitive to differences  from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  scaler.fit(X\_train)  X\_trainScaled = scaler.transform(X\_train)  X\_testScaled = scaler.transform(X\_test)  X\_valScaled = scaler.transform(X\_val)  def showResults(networkStats):  dfStats = pd.DataFrame.from\_records(networkStats)  dfStats = dfStats.sort\_values(by=['f1'])  print(dfStats)  def evaluate\_model(predictions, y\_test):  precision = precision\_score(y\_test, predictions)  recall = recall\_score(y\_test, predictions)  f1 = f1\_score(y\_test, predictions)  print("Precision: " + str(precision) + " " +\  "Recall: " + str(recall) + " " +\  "F1: " + str(f1))  return precision, recall, f1  clf = LogisticRegression(max\_iter=1000)  clf.fit(X\_trainScaled, y\_train)  predictions = clf.predict(X\_testScaled)  evaluate\_model(predictions, y\_test)  COLUMN\_DIMENSION = 1  #######################################################################  # Part 2  from keras.models import Sequential  from keras.layers import Dense  # shape() obtains rows (dim=0) and columns (dim=1)  n\_features = X\_trainScaled.shape[COLUMN\_DIMENSION]  def getPredictions(model, X\_test):  probabilities = model.predict(X\_test)  predictions = []  for i in range(len(probabilities)):  if (probabilities[i][0] > 0.5):  predictions.append(1)  else:  predictions.append(0)  return predictions  ### Model parameters ############################  optimizers = ['SGD', 'RMSprop', 'Adagrad',  'Adadelta', 'Adam', 'Adamax', 'Nadam']  #################################################  #######################################################################  # Model building section.  def create\_model(optimizer):  model = Sequential()  model.add(Dense(12, input\_dim=n\_features, activation='relu'))  model.add(Dense(1, activation='sigmoid'))  model.compile(loss='binary\_crossentropy', optimizer=optimizer,  metrics=['accuracy'])  return model  networkStats = []  EPOCHS = 200  NUM\_BATCHES = 60  for optimizer in optimizers:  model = create\_model(optimizer)  history = model.fit(X\_trainScaled, y\_train, epochs=EPOCHS,  batch\_size=NUM\_BATCHES, verbose=1,  validation\_data=(X\_valScaled, y\_val))  predictions = getPredictions(model, X\_testScaled)  precision, recall, f1 = evaluate\_model(predictions, y\_test)  networkStats.append({"precision":precision, "recall":recall,  "f1":f1, "optimizer":optimizer})  showResults(networkStats)  ####################################################################### |

## Grid Searching the Learning Rate for a Classifier

I am not an expert on RMSprop optimizers. However, when searching the Keras documentation at <https://keras.io/api/optimizers/>

I can see many parameter options. I will only focus on the learning rate for now.

|  |
| --- |
| tf.keras.optimizers.RMSprop(  learning\_rate=0.001,  rho=0.9,  momentum=0.0,  epsilon=1e-07,  centered=False,  weight\_decay=None,  clipnorm=None,  clipvalue=None,  global\_clipnorm=None,  use\_ema=False,  ema\_momentum=0.99,  ema\_overwrite\_frequency=100,  jit\_compile=True,  name="RMSprop",  \*\*kwargs  ) |

Example : Grid Searching the Learning Rate

To build this example replace the “Model parameters” and “Build model” sections of the last example with this version.

|  |
| --- |
| ### Model parameters ############################  learningRates = [0.0001, 0.001, 0.005, 0.01]  #################################################  #######################################################################  # Model building section.  import tensorflow as tf  def create\_model(learningRate):  model = Sequential()  model.add(Dense(12, input\_dim=n\_features, activation='relu'))  model.add(Dense(1, activation='sigmoid'))  optimizer = tf.keras.optimizers.RMSprop(learning\_rate=learningRate)  model.compile(loss='binary\_crossentropy', optimizer=optimizer,  metrics=['accuracy'])  return model  networkStats = []  EPOCHS = 200  NUM\_BATCHES = 60  for learningRate in learningRates:  model = create\_model(learningRate)  history = model.fit(X\_trainScaled, y\_train, epochs=EPOCHS,  batch\_size=NUM\_BATCHES, verbose=1,  validation\_data=(X\_valScaled, y\_val))  predictions = getPredictions(model, X\_testScaled)  precision, recall, f1 = evaluate\_model(predictions, y\_test)  networkStats.append({"precision":precision, "recall":recall,  "f1":f1, "learningRate":learningRate})  showResults(networkStats)  ####################################################################### |

The output shows that a learning rate of 0.001 achieves optimal results. Efforts to fine tune results might include [0.0005, 0.0008, 0.0012, 0.0014, 0.002] and so on in case further improvements can be made.

|  |
| --- |
| precision recall f1 learningRate  0 0.916667 0.478261 0.628571 0.0001  1 0.954545 0.913043 0.933333 0.0010  2 0.954545 0.913043 0.933333 0.0050  3 0.954545 0.913043 0.933333 0.0100 |

Exercise (3 marks)

Starting with the code from Example 11, set the optimal learning rate and modify the code to enable grid searching for the optimal number of neurons in the hidden layer.

Show the changes that you make to the model parameters and model building sections here:

|  |
| --- |
|  |

Show a screenshot of the results that are obtained when grid searching the number of neurons in the hidden layer.

|  |
| --- |
|  |

How many neurons did you choose and explain why?

|  |
| --- |
| My Neuron Choice is 5. As you can see the resulting precision, recall and f1 have the highest values in comparison to other neuron values. |

|  |
| --- |
| What are the metrics of your logistic regression model? |

What are the metrics of your final neural network model?

|  |
| --- |
| here is the segment |