Table of Contents

[Neural Networks Summarized by Siraj in 9 Minutes 1](#_Toc82503446)

[The Learning Cycle of an Artificial Neural Network (ANN) 2](#_Toc82503447)

[Hyperparameter Terminology 3](#_Toc82503448)

[Epoch 3](#_Toc82503449)

[Batch Size 3](#_Toc82503450)

[Loss Functions 3](#_Toc82503451)

[1. Binary Crossentropy **'binary\_crossentropy'** 3](#_Toc82503452)

[2. Sparse Categorial Crossentropy **'sparse\_categorical\_crossentropy'** 3](#_Toc82503453)

[Activation Functions 3](#_Toc82503454)

[1. Sigmoid 'sigmoid' 4](#_Toc82503455)

[2. Tanh 'tanh' 4](#_Toc82503456)

[3. ReLU (Rectified Linear Unit) 'relu' 4](#_Toc82503457)

[4. Leaky ReLU 4](#_Toc82503458)

[5. Softmax 'softmax' 5](#_Toc82503459)

[Optimizers 5](#_Toc82503460)

[1. Adam 'adam' 5](#_Toc82503461)

[2. SGD 'sgd' 5](#_Toc82503462)

[Kernel Initializers 5](#_Toc82503463)

[Using a Grid Search Tuning the Neural Network Hyperparameters 6](#_Toc82503464)

[Least Squares Regression Baseline 6](#_Toc82503465)

[Neural Network Baseline 8](#_Toc82503466)

[Epoch and Batch Size 10](#_Toc82503467)

[Optimizer 12](#_Toc82503468)

[Learning Rate and Other Optimizer Parameters 13](#_Toc82503469)

[Kernel Initializer 14](#_Toc82503470)

[Number of Neurons 15](#_Toc82503471)

[Adding Another Layer 16](#_Toc82503472)

[Final Model 17](#_Toc82503473)

[Additional Tuning Tips 19](#_Toc82503474)

# Neural Networks Summarized by Siraj in 9 Minutes

This video summarizes the neural network structures at a lightning fast pace but the video is fun and it offers good re-enforcement for learning. You may want to revisit this video several times later in the course as you become more familiar with neural networks.

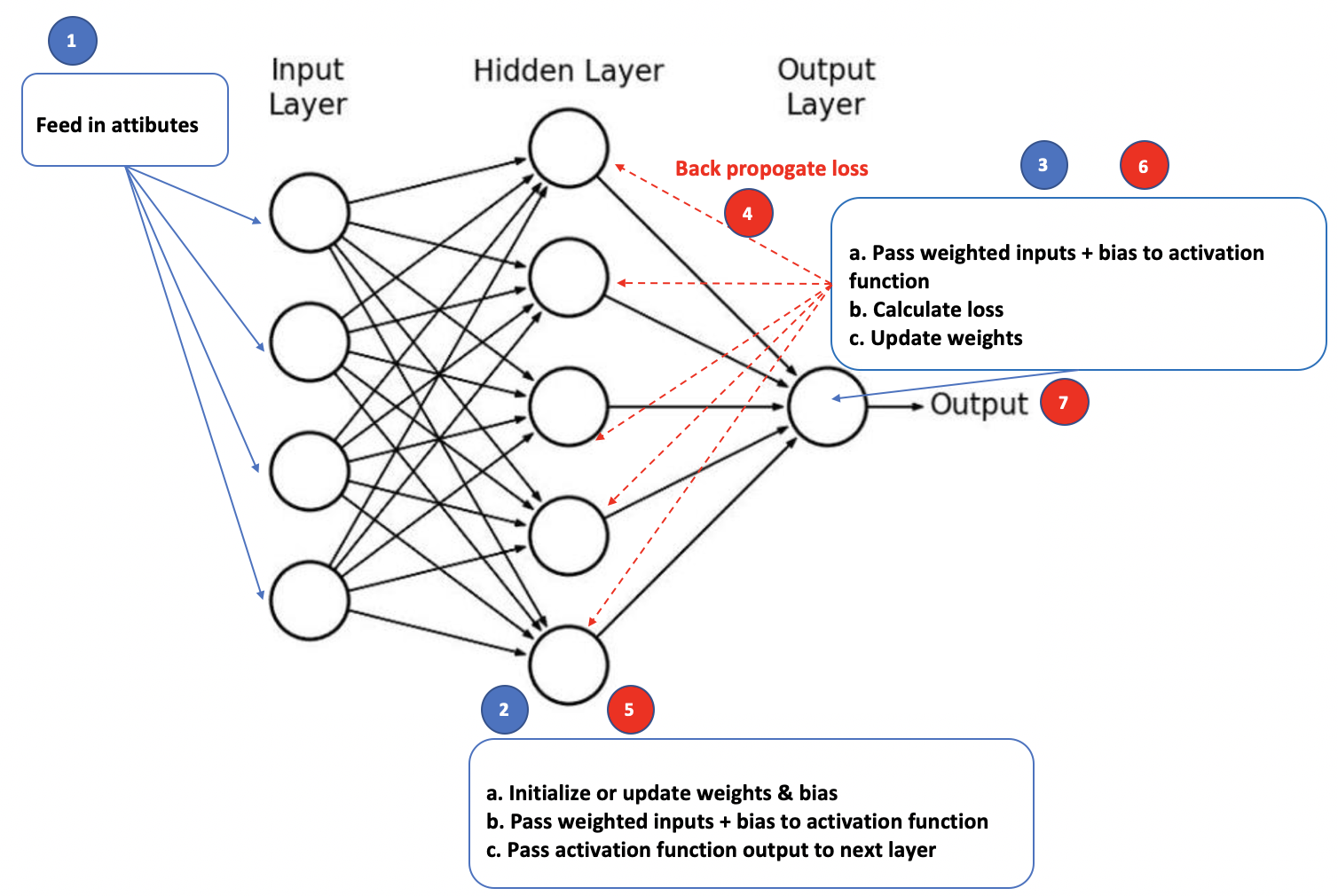
<https://www.youtube.com/watch?v=p69khggr1Jo>

# The Learning Cycle of an Artificial Neural Network (ANN)

Machine learning algorithms use the term **learning** when referring to training. The diagram in Figure 1 helps to illustrate the neural network learning cycle.

1. A sample’s features are fed forward to a neuron in the input layer. Each input is then fed forward to each **neuron** in the hidden layer. Each neuron initializes a weight and a bias.
2. The dot product of each weight vector times the sample attributes is then added to each corresponding bias. This result is then sent to the activation function which usually compresses the result to a range that is roughly between 0 to 1. The output from the activation function is then sent forward to the next layer.
3. The loss is calculated at the output layer.
4. The weights and bias in the output layer are calculated.
5. The losses are then propagated backwards to the preceding layers. Weights and biases in all preceding layers are updated until reaching the first hidden layer.
6. Steps 2 through 5 are repeated. Each full pass of all samples through the network is called an **epoch**.
7. The final model weights and biases are then stored when the **learning** (training) is complete.

Figure : Single Dense Layer Network Learning Cycle



## 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.

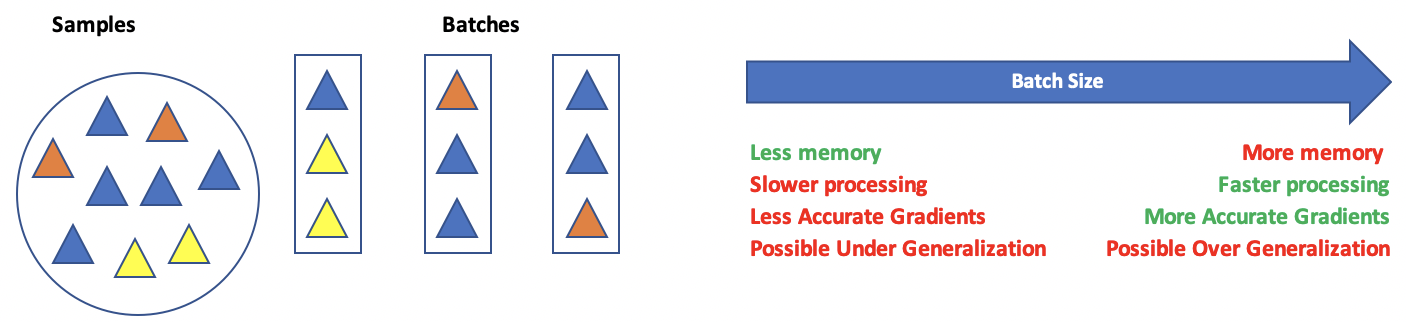
### Epoch

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 2 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.

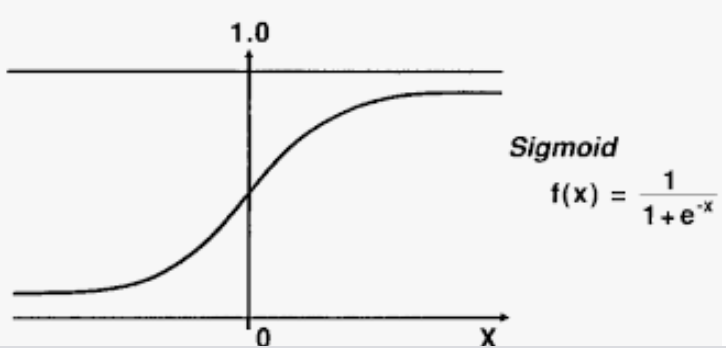
#### Mean absolute error.

<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:

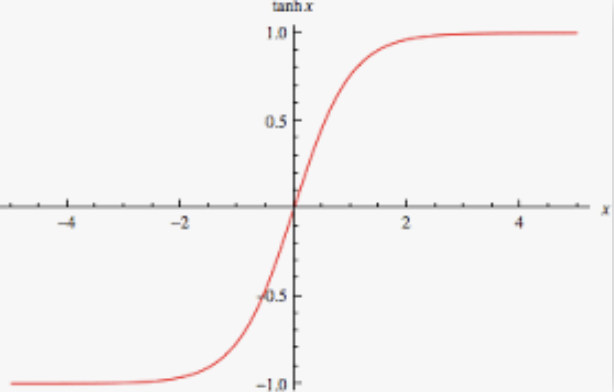
### Sigmoid 'sigmoid'



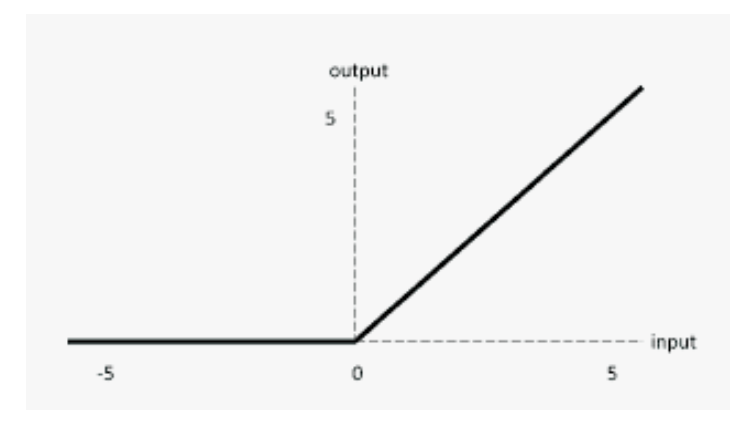
A sigmoid function transforms values to a value between 0 and 1. The sigmoid function can lead to a vanishing gradient problem in deeper networks because really large and really small values are not much different from each other. Computationally, the sigmoid is expensive.

### 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 adjust **weights and bias** for the regression equation and the **learning rate** 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'.

# Using a Grid Search Tuning the Neural Network Hyperparameters

A grid search is an exhaustive loop through a list of potential neural network hyperparameter combinations. The goal is to find the most effective hyperparameter combination. Grid searches can be conducted with the help of the **KerasClassifier** for classification problems or the **KerasRegressor** class for linear regression type problems.

This section is based on this post at <https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-pythond-keras/>

https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/

The example in the post provided demonstrates an example which how the **KerasClassifier** can help to tune a neural network for classification. Example 1, however, shows how to implement the **KerasRegressor** to tune a network that is performing regression for a continuous variable.

## 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 model can also help to determine a reasonable level of accuracy for your 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 0.773 indicates a strong correlation between the price and predictor variables. The root mean-square error, RMSE, value of 5.78 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:

|  |
| --- |
| Root Mean Squared Error: 5.7835  OLS Regression Results  ==============================================================================  Dep. Variable: y R-squared: 0.773  Model: OLS Adj. R-squared: 0.765  Method: Least Squares F-statistic: 102.2  Date: Sun, 13 Sep 2020 Prob (F-statistic): 9.64e-117  Time: 08:25:09 Log-Likelihood: -1171.5  No. Observations: 404 AIC: 2371.  Df Residuals: 390 BIC: 2427.  Df Model: 13  Covariance Type: nonrobust  ==============================================================================  coef std err t P>|t| [0.025 0.975]  ------------------------------------------------------------------------------  const 38.0917 5.522 6.898 0.000 27.234 48.949  x1 -0.1194 0.037 -3.257 0.001 -0.192 -0.047  x2 0.0448 0.014 3.102 0.002 0.016 0.073  x3 0.0055 0.063 0.087 0.931 -0.119 0.130  x4 2.3408 0.902 2.595 0.010 0.567 4.115  x5 -16.1236 4.212 -3.828 0.000 -24.404 -7.843  x6 3.7087 0.458 8.106 0.000 2.809 4.608  x7 -0.0031 0.014 -0.218 0.828 -0.031 0.025  x8 -1.3864 0.214 -6.480 0.000 -1.807 -0.966  x9 0.2442 0.070 3.481 0.001 0.106 0.382  x10 -0.0110 0.004 -2.819 0.005 -0.019 -0.003  x11 -1.0459 0.137 -7.636 0.000 -1.315 -0.777  x12 0.0081 0.003 2.749 0.006 0.002 0.014  x13 -0.4928 0.054 -9.086 0.000 -0.599 -0.386  ==============================================================================  Omnibus: 141.494 Durbin-Watson: 1.996  Prob(Omnibus): 0.000 Jarque-Bera (JB): 629.882  Skew: 1.470 Prob(JB): 1.67e-137  Kurtosis: 8.365 Cond. No. 1.55e+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)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, 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** |
| 5.7835 | 6.5880 |

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  from keras.wrappers.scikit\_learn import KerasRegressor  from sklearn.model\_selection import cross\_val\_score  from sklearn.model\_selection import KFold  # 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\_test, y\_train, y\_test = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.2, 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  # Since this is a linear regression use KerasRegressor.  estimator = KerasRegressor(build\_fn=create\_model, epochs=100,  batch\_size=5, verbose=1)  # Use kfold analysis for a more reliable estimate.  kfold = KFold(n\_splits=10)  results = cross\_val\_score(estimator, X\_train, y\_train, cv=kfold)  print("Baseline Mean (%.2f) MSE (%.2f) " % (results.mean(), results.std()))  print("Baseline RMSE: " + str(np.sqrt(results.std())))  # Build the model.  model = create\_model()  history = model.fit(X\_train, y\_train, epochs=100,  batch\_size=5, verbose=1,  validation\_data=(X\_test, y\_test))  # 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

The yellow highlighted code sets up the grid search for batch\_size and epochs.

|  |
| --- |
| ### Grid Building Section #######################  model = KerasRegressor(build\_fn=create\_model)  # define the grid search parameters  batch\_size = [10, 20, 40, 60, 80, 100]  epochs = [5, 10, 50, 100]  param\_grid = dict(batch\_size=batch\_size, epochs=epochs)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid,  n\_jobs=-1, cv=3, verbose=1)  ################################################# |

The output from the first scan suggests that 100 epochs and a batch size of 10 are optimal.

|  |
| --- |
| Best: -30.506042 using {'batch\_size': 10, 'epochs': 100}  -73.378028 (15.199006) with: {'batch\_size': 10, 'epochs': 5}  -59.623238 (17.230753) with: {'batch\_size': 10, 'epochs': 10} |

After observing a suggested batch size of 10, I ran the code again with a finer tuned set of numbers and determined that a batch size of 9 might be more optimal. I decided to carry the batch size of 9 into the remaining examples after this observation.

Best: -27.585736 using {'batch\_size': 9, 'epochs': 100}

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  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)  # Increase number of columns that display on one line.  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]  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,  y, test\_size=0.2, random\_state=0)  from keras.models import Sequential  from keras.layers import Dense  from keras.wrappers.scikit\_learn import KerasRegressor  from sklearn.model\_selection import GridSearchCV  # 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  ### Grid Building Section #######################  model = KerasRegressor(build\_fn=create\_model)  # define the grid search parameters  batch\_size = [10, 20, 40, 60, 80, 100]  epochs = [5, 10, 50, 100]  param\_grid = dict(batch\_size=batch\_size, epochs=epochs)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid,  n\_jobs=-1, cv=3, verbose=1)  #################################################  grid\_result = grid.fit(X\_train, y\_train)  # summarize results  print("Best: %f using %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))  means = grid\_result.cv\_results\_['mean\_test\_score']  stds = grid\_result.cv\_results\_['std\_test\_score']  params = grid\_result.cv\_results\_['params']  for mean, stdev, param in zip(means, stds, params):  print("%f (%f) with: %r" % (mean, stdev, param)) |

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. Show the RMSE that results after making these changes:

|  |
| --- |
| Neural network RMSE: 5.884266807 |

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

|  |
| --- |
| Neural network RMSE: 5.9427307 |

Show the section of code that you modified here.

|  |
| --- |
| *# Since this is a linear regression use KerasRegressor.*  estimator = KerasRegressor(*build\_fn*=create\_model, *epochs*=100,  *batch\_size*=10, *verbose*=1)  *# Use kfold analysis for a more reliable estimate.*  kfold   = KFold(*n\_splits*=10)  results = cross\_val\_score(estimator, X\_train, y\_train, *cv*=kfold)  print("Baseline Mean (%.2f) MSE (%.2f) " % (results.mean(), results.std()))  print("Baseline RMSE: " + str(np.sqrt(results.std())))  *# Build the model.*  model   = create\_model()  history = model.fit(X\_train, y\_train, *epochs*=100,  *batch\_size*=10, *verbose*=1,  *validation\_data*=(X\_test, y\_test)) |

Exercise (1 mark)

What change in behaviour do you notice if the verbose=1 attribute is missing from the GridSearchCV() construction?

|  |
| --- |
| There are a lot fewer messages printed to terminal when verbose=1 is missing from GridSearchCV(). |

Exercise (1 mark)

What does the cv parameter of the GridSearchCV() contructor do? (Please look this up online)

|  |
| --- |
| The cv parameter determines the cross-validation splitting strategy. |

## 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.

|  |
| --- |
| Best: -30.511445 using {'optimizer': 'Adam'}  -85.811811 (15.191973) with: {'optimizer': 'SGD'}  -35.981366 (11.191688) with: {'optimizer': 'RMSprop'}  -83.595862 (30.616171) with: {'optimizer': 'Adagrad'} |

Since we are building on the best outcome from the last example, starting with Example 3, replace the create\_model() code here from the previous example with this version to allow for dynamic adjustments of the optimizer function.

|  |
| --- |
| 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 |

Replace the \*Grid Buildinig Section \* from the previous example with this version.

|  |
| --- |
| ### Grid Building Section #######################  model = KerasRegressor(build\_fn=create\_model, epochs=100, batch\_size=10, verbose=1)  # Define the grid search parameters.  optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']  param\_grid = dict(optimizer=optimizer)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3)  ################################################# |

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 to update weights for each neuron**.
3. An optimizer sets an algorithm for activation.
4. An optimizer sets an algorithm for producing output.

## 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.

|  |
| --- |
| Best: -24.077209 using {'learningRate': 0.005}  -35.221776 (11.059594) with: {'learningRate': 0.001}  -24.077209 (7.600185) with: {'learningRate': 0.005} |

To build this example, replace the create\_model() function from the previous example with this code:

|  |
| --- |
| from tensorflow.keras.optimizers import Adam #for adam optimizer  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  opt = Adam(lr=learningRate)  model.compile(loss='mse', metrics=['accuracy'], optimizer=opt)  return model |

Also, replace the “Grid Building Section” from the previous example with this version:

|  |
| --- |
| ### Grid Building Section #######################  model = KerasRegressor(build\_fn=create\_model, epochs=100, batch\_size=10, verbose=1)  # Define the grid search parameters.  learningRates= [0.001, 0.005, 0.01, 0.015, 0.2]  param\_grid = dict(learningRate=learningRates)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3)  ################################################# |

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?

|  |
| --- |
| The new create\_model() in example 5 includes a learning rate (0.001) which is fed to the Adam optimizer. This fine tunes the optimizer just a bit more. Example 4 does not make use of the learning rate and uses the SGD optimizer. |

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:

|  |
| --- |
| Neural network RMSE: 5.4898663014758196 |

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

|  |  |
| --- | --- |
| Neural network RMSE: 5.9427307 |  |

Show the section of code that you modified here.

|  |
| --- |
| def create\_model(*learningRate* = 0.005):      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*      opt = Adam(*lr*=learningRate)      model.compile(*loss*='mse', *metrics*=['accuracy'], *optimizer*=opt)  *return* model |

## 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.

|  |
| --- |
| Best: -21.859127 using {'init\_mode': 'uniform'}  -21.859127 (4.547774) with: {'init\_mode': 'uniform'}  -33.742336 (13.388874) with: {'init\_mode': 'lecun\_uniform'}  -26.358027 (9.466685) with: {'init\_mode': 'normal'}  -182.339040 (28.125249) with: {'init\_mode': 'zero'} |

To build this example, replace the create\_model() function from the last example with this version.

|  |
| --- |
| n\_features = X\_train.shape[1]  # define base model  def create\_model(init\_mode='uniform'):  model = Sequential()  model.add(Dense(13, kernel\_initializer=init\_mode,  input\_dim=n\_features, activation='relu'))  model.add(Dense(1, kernel\_initializer=init\_mode))    opt = Adam(lr=0.005)  # Compile model  model.compile(loss='mse', metrics=['accuracy'], optimizer=opt)  return model |

Next, replace the “Grid Building Section” from the previous example with this version.

|  |
| --- |
| ### Grid Building Section #######################  model = KerasRegressor(build\_fn=create\_model, epochs=100, batch\_size=10, verbose=1)  # Define the grid search parameters.  init\_mode = ['uniform', 'lecun\_uniform', 'normal', 'zero', 'glorot\_normal',  'glorot\_uniform', 'he\_normal', 'he\_uniform']  param\_grid = dict(init\_mode=init\_mode)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3)  ################################################# |

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:

|  |
| --- |
| Neural network RMSE: 5.884411462409584 |

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

|  |
| --- |
| Neural network RMSE: 5.9427307 |

Show the section of code that you modified here.

|  |
| --- |
| def create\_model(*learningRate* = 0.005, *init\_mode*='uniform'):      model = Sequential()      model.add(Dense(13, *input\_dim*=13, *kernel\_initializer*=init\_mode,  *activation*='relu'))      model.add(Dense(1, *kernel\_initializer*=init\_mode))  *# Use Adam optimizer with the given learning rate*      opt = Adam(*lr*=learningRate)      model.compile(*loss*='mse', *metrics*=['accuracy'], *optimizer*=opt)  *return* model |

## 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.

|  |
| --- |
| Best: -18.883469 using {'neurons': 25}  -75.784907 (57.613429) with: {'neurons': 1}  -34.282229 (3.353030) with: {'neurons': 5}  -24.196648 (4.968971) with: {'neurons': 10} |

To build this example, replace the create\_model() function from the last example with this version:

|  |
| --- |
| def create\_model(neurons=1):  model = Sequential()  model.add(Dense(neurons, kernel\_initializer='uniform',  input\_dim=n\_features, activation='softplus'))  model.add(Dense(1, kernel\_initializer='he\_normal', activation='softplus'))  opt = Adam(lr=0.005)  model.compile(loss='mse', metrics=['accuracy'], optimizer=opt)  return model |

Next, replace the “grid building section” with this version:

|  |
| --- |
| ### Grid Building Section #######################  model = KerasRegressor(build\_fn=create\_model, epochs=100, batch\_size=9, verbose=1)  neurons = [1, 5, 10, 15, 20, 25, 30]  param\_grid = dict(neurons=neurons)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3)  ################################################# |

## 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.

**Warning:** This code takes a really really long time to run. However, the outcome suggests that a new layer could improve results. The optimal parameters are highlighted:

|  |
| --- |
| Best: -16.677304 using {'activation': 'softplus', 'initializer': 'lecun\_uniform', 'numNeurons': 10}  -89.553500 (18.304827) with: {'activation': 'softmax', 'initializer': 'uniform', 'numNeurons': 10}  -89.235888 (17.908884) with: {'activation': 'softmax', 'initializer': 'uniform', 'numNeurons': 15}  -89.686081 (18.243811) with: {'activation': 'softmax', 'initializer': 'uniform', 'numNeurons': 20} |

To build this function, replace the create\_model() function with this version to scan for optimal neurons, the initializer and the activation function in the new layer.

|  |
| --- |
| def create\_model(numNeurons=5, initializer='uniform', activation='softplus'):  # create model  model = Sequential()  model.add(Dense(25, kernel\_initializer='uniform',  input\_dim=n\_features, activation='softplus'))  model.add(Dense(numNeurons, kernel\_initializer=initializer,  activation=activation))  model.add(Dense(1, kernel\_initializer='he\_normal', activation='softplus'))  opt = Adam(lr=0.005)  # Compile model  model.compile(loss='mse', metrics=['accuracy'], optimizer=opt)  return model |

Next, replace the grid building section with this version to scan feed options for activation function, number of neurons and the initializer function:

|  |
| --- |
| ### Grid Building Section #######################  # Define the parameters to try out  params = { 'activation' : ['softmax', 'softplus', 'softsign', 'relu', 'tanh',  'sigmoid', 'hard\_sigmoid', 'linear'],  'numNeurons':[10, 15, 20, 25, 30, 35],  'initializer': ['uniform', 'lecun\_uniform', 'normal', 'zero',  'glorot\_normal', 'glorot\_uniform', 'he\_normal', 'he\_uniform']  }  model = KerasRegressor(build\_fn=create\_model, epochs=100, batch\_size=9, verbose=1)  grid = GridSearchCV(estimator=model, param\_grid=params, n\_jobs=-1, cv=3)  ################################################# |

## 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.

|  |  |
| --- | --- |
| Model | RMSE |
| OLS Model | 5.78 |
| Baseline Neural Network | 6.59 |
| Optimized Neural Network | 5.40 |

It is likely that further improvements can be made. Taking additional steps may be worth it depending on the payload. Perhaps additional layers could be added. Perhaps additional iterations of fine-tuning parameters could be made. To keep this manageable I will stop here with this reasonable success in hand. Here is the final network code which is based on the grid search:

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn import metrics  from sklearn.model\_selection import train\_test\_split  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)  # Increase number of columns that display on one line.  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]  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,  test\_size=0.2, random\_state=0)  from keras.models import Sequential  from keras.layers import Dense  from keras.wrappers.scikit\_learn import KerasRegressor  from sklearn.model\_selection import cross\_val\_score  from sklearn.model\_selection import KFold  # define base model  from keras.optimizers import Adam #for adam optimizer  def baseline\_model():  model = Sequential()  model.add(Dense(25, input\_dim=13, kernel\_initializer='uniform',  activation='softplus'))  model.add(Dense(10, kernel\_initializer='lecun\_uniform', activation='softplus'))  model.add(Dense(1, kernel\_initializer='uniform'))  # Use Adam optimizer with the given learning rate  opt = Adam(lr=0.005)  model.compile(loss='mean\_squared\_error', optimizer=opt)  return model  # evaluate model  estimator = KerasRegressor(build\_fn=baseline\_model, epochs=100,  batch\_size=9, verbose=1)  kfold = KFold(n\_splits=10)  results = cross\_val\_score(estimator, X\_train, y\_train, cv=kfold)  print("Baseline Mean (%.2f) MSE (%.2f) " % (results.mean(), results.std()))  print("Baseline RMSE: " + str(np.sqrt(results.std())))  # So then we build the model.  model = baseline\_model()  history = model.fit(X\_train, y\_train, epochs=100,  batch\_size=9, verbose=1,  validation\_data=(X\_test, y\_test))  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))) |

## 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**. You can see that the results from the examples in this post show some variance. A default cross-validation of 3 was used, but perhaps k=5 or k=10 would be more stable. Carefully choose your cross-validation configuration to ensure your results are stable.
* **Review the Whole Grid**. Do not just focus on the best result, review the whole grid of results and look for trends to support configuration decisions.
* **Parallelize**. Use all your cores if you can, neural networks are slow to train and we often want to try a lot of different parameters. Consider spinning up a lot of [AWS instances](http://machinelearningmastery.com/develop-evaluate-large-deep-learning-models-keras-amazon-web-services/).
* **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.
* **Start with Coarse Grids**. Start with coarse-grained grids and zoom into finer grained grids once you can narrow the scope.
* **Do not Transfer Results**. Results are generally problem specific. Try to avoid favorite configurations on each new problem that you see. It is unlikely that optimal results you discover on one problem will transfer to your next project. Instead look for broader trends like number of layers or relationships between parameters.
* **Reproducibility is a Problem**. Although we set the seed for the random number generator in NumPy, the results are not 100% reproducible. There is more to reproducibility when grid searching wrapped Keras models than is presented in this post.

Example : Grid Search

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn import metrics  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "heart\_disease.csv"  data = pd.read\_csv(PATH + FILE)  x\_data = data.drop("target", axis=1)  y\_values = data["target"]  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(data.head())  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  x\_data, y\_values, test\_size=0.3, 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)  clf = LogisticRegression(max\_iter=1000)  clf.fit(X\_trainScaled, y\_train)  lr\_pred = clf.predict(X\_testScaled)  print("Accuracy:{} ".format(clf.score(X\_testScaled, y\_test) \* 100))  print("Error Rate:{} ".format((1 - clf.score(X\_testScaled, y\_test)) \* 100))  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(y\_test, lr\_pred,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, lr\_pred))  print("\nConfusion Matrix")  print(confusion\_matrix)  COLUMN\_DIMENSION = 1  #######################################################################  # Part 2  from keras.models import Sequential  from keras.layers import Dense  from keras.wrappers.scikit\_learn import KerasClassifier  from sklearn.model\_selection import GridSearchCV  # shape() obtains rows (dim=0) and columns (dim=1)  n\_features = X\_trainScaled.shape[COLUMN\_DIMENSION]  #######################################################################  # Model tuning 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  model = KerasClassifier(build\_fn=create\_model, verbose=1)  batch\_size = [10, 20, 40, 60, 80, 100, 150, 200]  epochs = [10, 50, 100]  param\_grid = dict(batch\_size=batch\_size, epochs=epochs)  grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3)  #######################################################################  grid\_result = grid.fit(X\_trainScaled, y\_train)  # summarize results  print("Best: %f using %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))  means = grid\_result.cv\_results\_['mean\_test\_score']  stds = grid\_result.cv\_results\_['std\_test\_score']  params = grid\_result.cv\_results\_['params']  for mean, stdev, param in zip(means, stds, params):  print("%f (%f) with: %r" % (mean, stdev, param)) |

Exercise (1 mark)

What loss function is used in Example 9? Why?

|  |
| --- |
| In example 9 binary crossentropy is used because we are predicting a binary event. |

Exercise (1 mark)

Does Example 9 use KerasClassifier or KerasRegressor? Why?

|  |
| --- |
| Example 9 uses KerasClassifier because example 9 is a classification type problem: the variable in consideration is not continuous. |

Does Example 3 use KerasClassifier or KerasRegressor? Why?

|  |
| --- |
| Example 3 uses KerasRegressor because we want to perform linear regression on a continuous variable. |

Exercise (1 mark)

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

|  |
| --- |
| Since we just have 1 column, the input\_dim = n\_features = 1 |

Exercise (1 mark)

What batch and epoch size are recommended when running Example 9?

|  |
| --- |
| Best: 0.836508 using {'batch\_size': 40, 'epochs': 100} |

Exercise (1 mark).

The values for batch or epoch may be out of range. A more suitable range of epochs and batch size might be found. Can you find a total epochs and batch size pairing that offers better performance than in Exercise 11? If so, what is the improved pairing. You might run the code several times and record each answer to decide.

|  |
| --- |
| Best: 0.837084 using {'batch\_size': 60, 'epochs': 100} |

Exercise (3 marks)

Next replace the code in the model tuning section of Example 9 which is highlighted in yellow with code to tune the **optimizer**. Be sure to include the optimal batch size and number of epochs in your model which you found in Exercise 12. See <https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/> for a reference. What optimizer is recommended? (You may need to run the code several times and the results may not be conclusive).

|  |
| --- |
| The SGD optimizer is recommended at batch size = 60, epochs = 100. |

Show your revised model tuning section code here and be sure to include your optimized parameters from previous exercises.

|  |
| --- |
| *# Part 2*  *from* keras.models *import* Sequential  *from* keras.layers *import* Dense  *from* keras.wrappers.scikit\_learn *import* KerasClassifier  *from* keras.wrappers.scikit\_learn *import* KerasRegressor  *from* sklearn.model\_selection *import* GridSearchCV  *# shape() obtains rows (dim=0) and columns (dim=1)*  n\_features = X\_trainScaled.shape[COLUMN\_DIMENSION]  *#######################################################################*  *# Model tuning section.*  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  *### Grid Building Section #######################*  model = KerasRegressor(*build\_fn*=create\_model, *epochs*=100, *batch\_size*=10, *verbose*=1)  *# Define the grid search parameters.*  optimizer   = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']  param\_grid  = dict(*optimizer*=optimizer)  grid        = GridSearchCV(*estimator*=model, *param\_grid*=param\_grid, *n\_jobs*=-1, *cv*=3)  *#################################################*    grid\_result = grid.fit(X\_trainScaled, y\_train)  *# summarize results*  print("Best: %f using %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))  means  = grid\_result.cv\_results\_['mean\_test\_score']  stds   = grid\_result.cv\_results\_['std\_test\_score']  params = grid\_result.cv\_results\_['params']  *for* mean, stdev, param *in* zip(means, stds, params): |

What does an optimizer do and how is it different than a kernel initializer?

|  |
| --- |
| Kernel initializers are used to initialize the weights in the model. An optimizer sets and updates those values. |

Exercise (3 marks)

Try tuning the **learning rate**. Documentation about common optimizer parameters can be found at <https://keras.io/api/optimizers/>

What learning rate is recommended? (You may need to run the code several times and the results may not be conclusive).

|  |
| --- |
| A learning rate of 0.001 is recommended. |

Show your revised model tuning section code here and be sure to include your optimized parameters from previous exercises.

|  |
| --- |
| *#######################################################################*  *# Part 2*  *from* keras.models *import* Sequential  *from* keras.layers *import* Dense  *from* keras.wrappers.scikit\_learn *import* KerasClassifier  *from* keras.wrappers.scikit\_learn *import* KerasRegressor  *from* sklearn.model\_selection *import* GridSearchCV  *# shape() obtains rows (dim=0) and columns (dim=1)*  n\_features = X\_trainScaled.shape[COLUMN\_DIMENSION]  *#######################################################################*  *# Model tuning section.*  *from* tensorflow.keras.optimizers *import* Adam *#for adam optimizer*  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*      opt = Adam(*lr*=learningRate)      model.compile(*loss*='mse', *metrics*=['accuracy'], *optimizer*=opt)  *return* model    *### Grid Building Section #######################*  model = KerasRegressor(*build\_fn*=create\_model, *epochs*=100, *batch\_size*=60, *verbose*=1)  *# Define the grid search parameters.*  learningRates= [0.001, 0.005, 0.01, 0.015, 0.2]  param\_grid   = dict(*learningRate*=learningRates)  grid         = GridSearchCV(*estimator*=model, *param\_grid*=param\_grid, *n\_jobs*=-1, *cv*=3)  *#################################################* |

Exercise (4 marks)

Try tuning the **kernel initializer.**

What initializer is recommended? (You may need to run the code several times and the results may not be conclusive).

|  |
| --- |
| The uniform initializer is recommended |

Show your revised model tuning section code here and be sure to include your optimized parameters from previous exercises.

|  |
| --- |
| def create\_model(*numNeurons*=5, *initializer*='uniform', *activation*='softplus'):  *# create model*      model = Sequential()      model.add(Dense(25, *kernel\_initializer*='uniform',  *input\_dim*=n\_features, *activation*='softplus'))      model.add(Dense(numNeurons, *kernel\_initializer*=initializer,  *activation*=activation))      model.add(Dense(1, *kernel\_initializer*='he\_normal',  *activation*='softplus'))      opt = SGD(*lr*=0.001)  *# Compile model*      model.compile(*loss*='mse', *metrics*=['accuracy'], *optimizer*=opt)  *return* model    *### Grid Building Section #######################*  *# Define the parameters to try out*  params = { 'activation' : ['softmax', 'softplus', 'softsign', 'relu', 'tanh',                             'sigmoid', 'hard\_sigmoid', 'linear'],            'numNeurons':[10, 15, 20, 25, 30, 35],            'initializer': ['uniform', 'lecun\_uniform', 'normal', 'zero',                         'glorot\_normal', 'glorot\_uniform', 'he\_normal', 'he\_uniform']            }  model      = KerasRegressor(*build\_fn*=create\_model, *epochs*=100, *batch\_size*=60, *verbose*=1)  grid = GridSearchCV(*estimator*=model, *param\_grid*=params, *n\_jobs*=-1, *cv*=3) |

Exercise (3 marks)

Try tuning the neuron **activation function.**

What activation function is recommended? (You may need to run the code several times and the results may not be conclusive).

|  |
| --- |
| The softmax activation function is recommended. |

Show your revised model tuning section code here and be sure to include your optimized parameters from previous exercises.

|  |
| --- |
| def create\_model(*numNeurons*=5, *initializer*='uniform', *activation*='softmax'):  *# create model*      model = Sequential()      model.add(Dense(25, *kernel\_initializer*='uniform',  *input\_dim*=n\_features, *activation*='softmax'))      model.add(Dense(numNeurons, *kernel\_initializer*=initializer,  *activation*=activation))      model.add(Dense(1, *kernel\_initializer*='uniform',  *activation*='softmax'))      opt = SGD(*lr*=0.001)  *# Compile model*      model.compile(*loss*='mse', *metrics*=['accuracy'], *optimizer*=opt)  *return* model    *### Grid Building Section #######################*  *# Define the parameters to try out*  params = { 'activation' : ['softmax', 'softplus', 'softsign', 'relu', 'tanh',                             'sigmoid', 'hard\_sigmoid', 'linear'],            'numNeurons':[10, 15, 20, 25, 30, 35],            'initializer': ['uniform', 'lecun\_uniform', 'normal', 'zero',                         'glorot\_normal', 'glorot\_uniform', 'he\_normal', 'he\_uniform']            }  model      = KerasRegressor(*build\_fn*=create\_model, *epochs*=100, *batch\_size*=60, *verbose*=1)  grid = GridSearchCV(*estimator*=model, *param\_grid*=params, *n\_jobs*=-1, *cv*=3) |

Exercise (3 marks)

Try tuning the **number of neurons** in the hidden layer.

What activation function is recommended? (You may need to run the code several times and the results may not be conclusive).

|  |
| --- |
| The softmax activation function is recommended. The recommended # of neurons is 35. |

Show your revised model tuning section code here and be sure to include your optimized parameters from previous exercises.

|  |
| --- |
| def create\_model(*numNeurons*=35, *initializer*='uniform', *activation*='softmax'):  *# create model*      model = Sequential()      model.add(Dense(25, *kernel\_initializer*='uniform',  *input\_dim*=n\_features, *activation*='softmax'))      model.add(Dense(numNeurons, *kernel\_initializer*=initializer,  *activation*=activation))      model.add(Dense(1, *kernel\_initializer*='uniform',  *activation*='softmax'))      opt = SGD(*lr*=0.001)  *# Compile model*      model.compile(*loss*='mse', *metrics*=['accuracy'], *optimizer*=opt)  *return* model    *### Grid Building Section #######################*  *# Define the parameters to try out*  params = { 'activation' : ['softmax', 'softplus', 'softsign', 'relu', 'tanh',                             'sigmoid', 'hard\_sigmoid', 'linear'],            'numNeurons':[10, 15, 20, 25, 30, 35],            'initializer': ['uniform', 'lecun\_uniform', 'normal', 'zero',                         'glorot\_normal', 'glorot\_uniform', 'he\_normal', 'he\_uniform']            }  model      = KerasRegressor(*build\_fn*=create\_model, *epochs*=100, *batch\_size*=60, *verbose*=1)  grid = GridSearchCV(*estimator*=model, *param\_grid*=params, *n\_jobs*=-1, *cv*=3) |

Exercise (5 marks)

Show your revised model after performing the requested optimizations before this exercise. Be sure to evaluate the model with the test data. Show the entire program here:

|  |
| --- |
|  |

What is the accuracy rate when running the test data though the neural network?

|  |
| --- |
|  |

Does the model have a higher accuracy rate when compared to the basic logistic regression model?

|  |
| --- |
|  |