Table of Contents

[References 1](#_Toc126171328)

[Hyperparameter Tuning 1](#_Toc126171329)

[Regular Logistic Regression 2](#_Toc126171330)

[Configuring the Optimizer 3](#_Toc126171331)

[Stochastic Gradient Descent (SGD) 3](#_Toc126171332)

[Tuning the Number of Nodes 4](#_Toc126171333)

[Configuring the Number of Layers 6](#_Toc126171334)

[Configuring Batch Size 9](#_Toc126171335)

[Summary 11](#_Toc126171336)

[Other Tips 12](#_Toc126171337)

# References

“Better Deep Learning’ by Dr. Jason Brownlee <https://machinelearningmastery.com/better-deep-learning/>

# Hyperparameter Tuning

Hyperparameter tuning is the most challenging task in machine learning. Even for experts, hyperparameter tuning is difficult due to many reasons including:

* A complicated solution hyperplane that has many potential optima which are not necessarily the best optima in the solution space.
* Many settings to choose from that can lead to millions of tuning combinations that are not linearly related.
* Large amounts of training time needed.

This discussion presents a somewhat sequential approach to hyperparameter tuning. However, in practice you will likely find that you will need to iterate over different steps to fine tune the network. As you tune your models, sometimes you will have to settle on good-enough solutions at each step. Ideally you are able to iterate over and fine tune these configurations later to find a better solution. The use of loss and accuracy graphs should also be used where possible to visualize the success of the tuning effort. Accuracy graphs are not available for linear regression problems.

A generally recommended tuning order is:

* Performing a basic regression or classification to better understand the targets.
* Use stochastic gradient descent so you can adjust the learning rate and momentum. Pay attention to the validation loss and accuracy ratings.
* Tune the number of nodes in a single layer network. Tweak the learning rate, momentum and number of epochs if needed. Pay attention to the validation loss and accuracy ratings.
* Tune the number of layers. Tweak the learning rate, momentum and number of epochs if needed. Pay attention to the validation loss and accuracy ratings.
* Determine the optimal batch size. Tweak the learning rate, momentum and number of epochs if needed. Pay attention to the validation loss and accuracy ratings.

## Regular Logistic Regression

To better understand the tuning targets before configuring a network, a basic regression to identify reasonable targets is recommended.

Example : Basic Logistic Regression

In this example we will start the series by first perform basic logistic regression with the diabetes data set. Simple logistic regression can detect diabetes with accuracy score of 80%.

|  |
| --- |
| Accuracy: 0.8070866141732284  Confusion Matrix  Predicted 0 1  Actual  0 156 16  1 33 49 |

Here is the code:

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  PATH = "/Users/pm/Desktop/DayDocs/data/"  from sklearn.linear\_model import LogisticRegression  from sklearn import metrics  import numpy as np  # load the dataset  df = pd.read\_csv(PATH + 'diabetes.csv', sep=',')  # split into input (X) and output (y) variables  X = df[['Pregnancies','Glucose','BloodPressure','SkinThickness','Insulin','BMI',  'DiabetesPedigreeFunction', 'Age']]  y = df[['Outcome']]  # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.33)  # Perform logistic regression.  logisticModel = LogisticRegression(fit\_intercept=True, random\_state = 0,  solver='liblinear')  logisticModel.fit(X\_train,y\_train)  y\_pred=logisticModel.predict(X\_test)  # Show model coefficients and intercept.  print("\nModel Coefficients: ")  print("\nIntercept: ")  print(logisticModel.intercept\_)  print(logisticModel.coef\_)  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(np.array(y\_test['Outcome']), y\_pred,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix")  print(confusion\_matrix) |

## Configuring the Optimizer

Configuring the optimizer is usually done simultaneously while choosing the number of nodes in a single layer network.

### Stochastic Gradient Descent (SGD)

When faced with a large set of unknown parameters, use of the stochastic gradient descent optimizer is recommended for simplicity. With SGD you can tune the learning rate for speed of learning and the momentum for smoothing. The *learning\_rate* and *momentum* parameters can range between 0 and 1. You can grid search these values or you can experiment with different settings to find reasonably adequate settings.

optimizer = tf.keras.optimizers.SGD(

learning\_rate=0.1, momentum=0.9, name="SGD")

High **learning rates** can lead to volatile but quick learning at the risk of overshooting the optimum. Low learning rates lead to longer training times but they offer a smoother trend for losses and accuracy. Learning rates that are too low though may never converge on a solution. It is up to you to find a decent learning rate and there is no perfect solution. You may need to adjust this setting several times as you adjust other settings.

**Momentum** is a decaying weighted average of past gradients which helps to smooth weight adjustments. Momentum can help to overcome barriers to current weight adjustments such as steep curves or flat regions. A momentum that is too high though can lead to weight changes that are not sensitive enough to current changes in the solution plane.

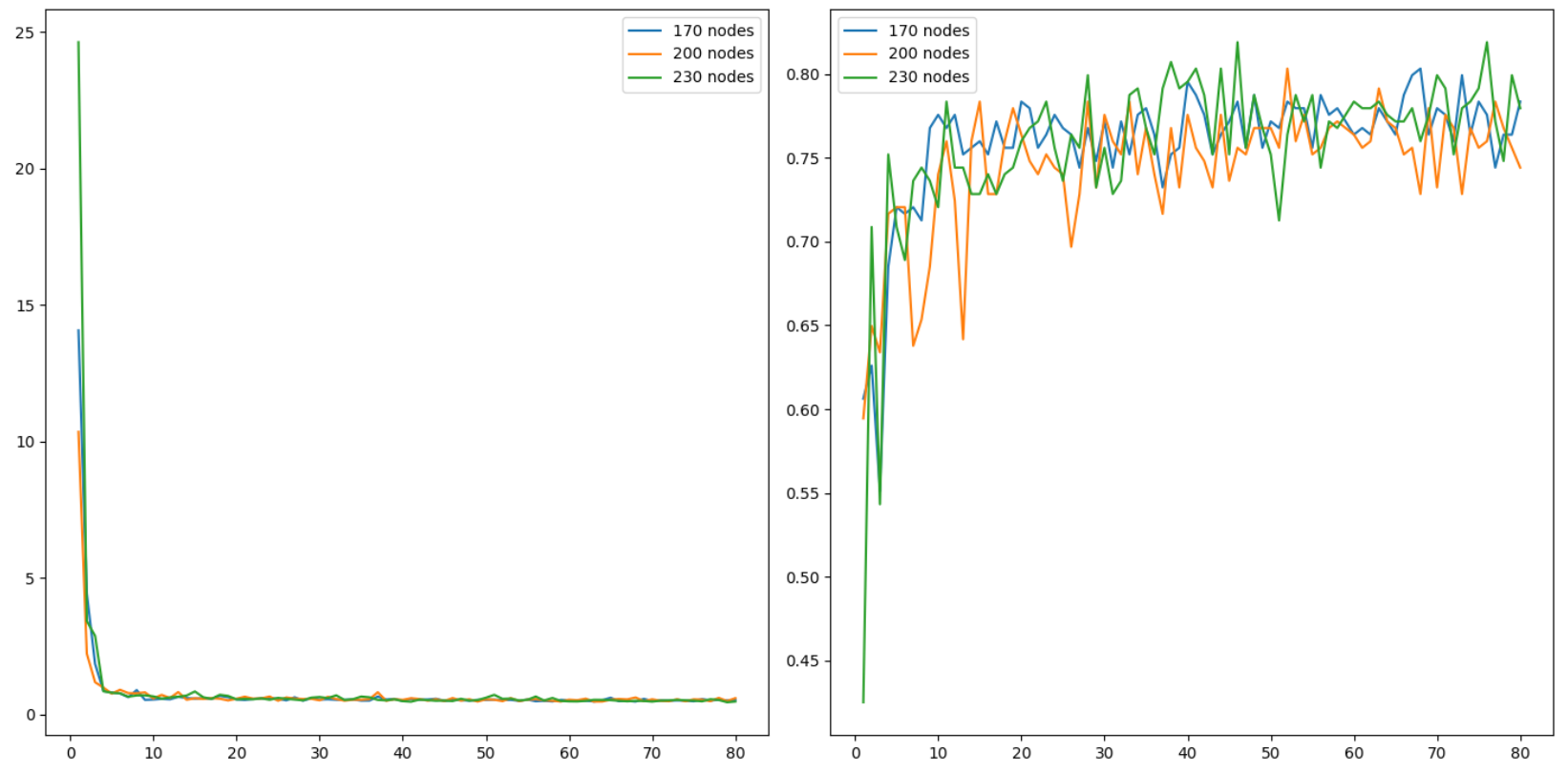
## Tuning the Number of Nodes

In his book, “Better Deep Learning’, Dr. Jason Brownlee suggests starting the tuning session by determining the optimal number of nodes in a single layer network. A single layer network is capable of solving most learning problems if it is made wide enough. In this case, **width** refers to the number of nodes in the layer.

Example : Determining the Total Nodes

This example compares the losses and accuracy when of a single layer network that has different numbers of nodes. I actually started by testing a set of node counts totalling 3, 6 and 9. 9 nodes showed the most promise so I tried a set of larger nodes. However, the highest number of nodes in all of my trials kept offering the best results. I ended up drastically increasing the numbers of nodes until the single layer network performance plateaued at 230 nodes (see Figure 1). Note also that at this time I had to keep tweaking the learning rate and momentum of the stochastic gradient descent optimizer to find a balance between volatility and validation accuracy.

Figure : Comparing Performance of a Single Layer Network with Different Numbers of Nodes



Here is the winning code so far at this point.

|  |
| --- |
| # first neural network with keras tutorial  from keras.models import Sequential  from keras.layers import Dense  import pandas as pd  from sklearn.model\_selection import train\_test\_split  import matplotlib.pyplot as plt  PATH = "/Users/pm/Desktop/DayDocs/data/"  import tensorflow as tf  # load the dataset  df = pd.read\_csv(PATH + 'diabetes.csv', sep=',')  # split into input (X) and output (y) variables  X = df[['Pregnancies','Glucose','BloodPressure','SkinThickness','Insulin','BMI',  'DiabetesPedigreeFunction', 'Age']]  y = df[['Outcome']]  # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.33)  def buildModel(num\_nodes):  # define the keras model  model = Sequential()  model.add(Dense(num\_nodes, input\_dim=8, activation='relu',  kernel\_initializer='he\_normal'))  model.add(Dense(1, activation='sigmoid'))  opitimizer = tf.keras.optimizers.SGD(  learning\_rate=0.0005, momentum=0.9, name="SGD",  )  # compile the keras model  model.compile(loss='binary\_crossentropy', optimizer=opitimizer,  metrics=['accuracy'])  # fit the keras model on the dataset  history = model.fit(X, y, epochs=80, batch\_size=10, validation\_data=(X\_test,  y\_test))  # evaluate the keras model  # Evaluate the model.  loss, acc = model.evaluate(X\_test, y\_test, verbose=0)  print('Test Accuracy: ' + str(acc) + ' Num nodes: ' + str(num\_nodes))  return history  def showLoss(history, numNodes):  # Get training and test loss histories  training\_loss = history.history['loss']  validation\_loss = history.history['val\_loss']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  # Visualize loss history for training data.  actualLabel = str(numNodes) + " nodes"  plt.subplot(1, 2, 1)  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  def showAccuracy(history, numNodes):  # Get training and test loss histories  training\_loss = history.history['accuracy']  validation\_loss = history.history['val\_accuracy']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 2)  actualLabel = str(numNodes) + " nodes"  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  nodeCounts = [170,200,230]  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  for i in range(0, len(nodeCounts)):  history = buildModel(nodeCounts[i])  showLoss(history, nodeCounts[i])  showAccuracy(history, nodeCounts[i])  plt.show() |

## Configuring the Number of Layers

Deeper networks are typically more efficient than single layer networks. An optimal number of layers should be chosen with the same number of nodes that was optimized for a single layer in Example 2. Later in Example 3, we will use this code snippet to grid search the optimal number of layers.

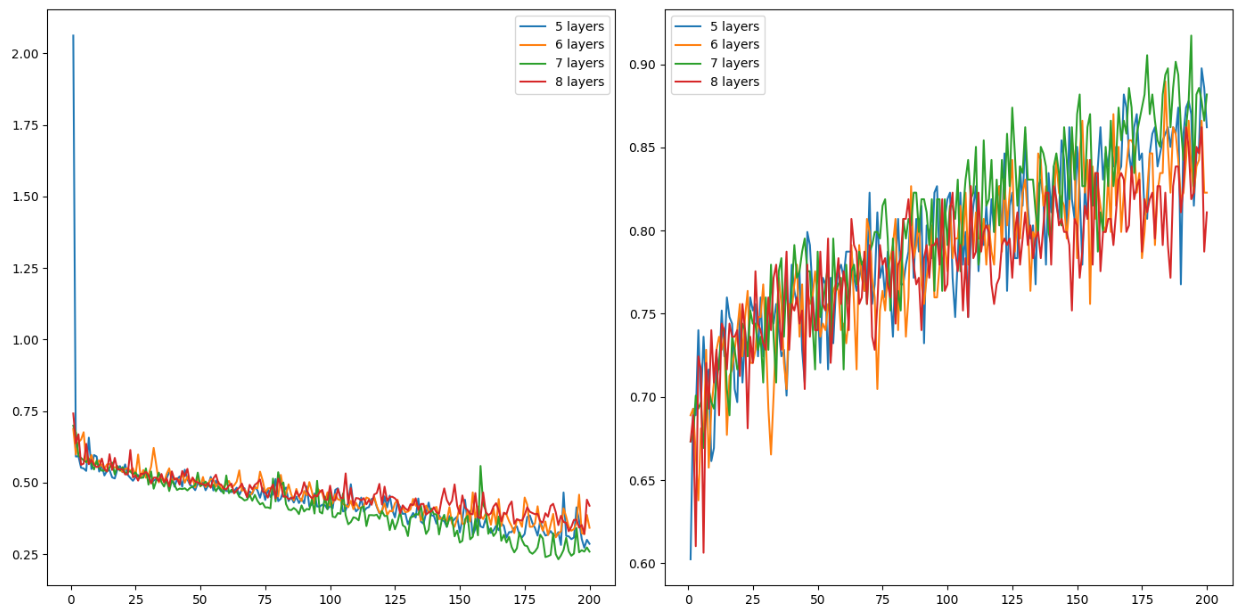
|  |
| --- |
| numLayers = [5,6,7,8]  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  for i in range(0, len(numLayers)):  history = buildModel(numLayers[i])  showLoss(history, numLayers[i])  showAccuracy(history, numLayers[i]) |

When we refer to numbers of layers we are really referring to network **depth**. Until recently, deeper networks were often not possible due to vanishing gradients. The **relu** activation function helps to overcome the vanishing gradient problem so use it when configuring the number of layers.

Example : Number Layers

When searching for the optimal number of layers, I started with network layers of 1, 3 and 4. I kept shifting the number of layers up and eventually found that 7 layers was optimal as shown in Figure 2. As you can see in Figure 2, the losses are still declining and the accuracy is still rising. It appears that additional gains can be obtained by increasing the number of epochs so I did later.

Figure : Comparing Network Depth

****

Here is the code:

|  |
| --- |
| # first neural network with keras tutorial  from keras.models import Sequential  from keras.layers import Dense  import pandas as pd  from sklearn.model\_selection import train\_test\_split  import matplotlib.pyplot as plt  PATH = "/Users/pm/Desktop/DayDocs/data/"  import tensorflow as tf  # load the dataset  df = pd.read\_csv(PATH + 'diabetes.csv', sep=',')  # split into input (X) and output (y) variables  X = df[['Pregnancies','Glucose','BloodPressure','SkinThickness','Insulin','BMI',  'DiabetesPedigreeFunction', 'Age']]  y = df[['Outcome']]  # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.33)  def buildModel(numLayers):  # define the keras model  model = Sequential()  model.add(Dense(230, input\_dim=8, activation='relu',  kernel\_initializer='he\_normal'))  for i in range(0, numLayers-1):  model.add(Dense(230, activation='relu',  kernel\_initializer='he\_normal'))  model.add(Dense(1, activation='sigmoid'))  opitimizer = tf.keras.optimizers.SGD(  learning\_rate=0.0005, momentum=0.9, name="SGD",  )  # Compile the keras model.  model.compile(loss='binary\_crossentropy', optimizer=opitimizer,  metrics=['accuracy'])  # Fit the keras model on the dataset.  history = model.fit(X, y, epochs=200, batch\_size=10,  validation\_data=(X\_test, y\_test))  # Evaluate the model.  loss, acc = model.evaluate(X\_test, y\_test, verbose=0)  print('Test Accuracy: %.3f' % acc)  return history  def showLoss(history, numNodes):  # Get training and test loss histories  training\_loss = history.history['loss']  validation\_loss = history.history['val\_loss']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  # Visualize loss history for training data.  actualLabel = str(numNodes) + " layers"  plt.subplot(1, 2, 1)  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  def showAccuracy(history, numNodes):  # Get training and test loss histories  training\_loss = history.history['accuracy']  validation\_loss = history.history['val\_accuracy']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 2)  actualLabel = str(numNodes) + " layers"  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  numLayers = [5,6,7,8]  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  for i in range(0, len(numLayers)):  history = buildModel(numLayers[i])  showLoss(history, numLayers[i])  showAccuracy(history, numLayers[i])  plt.show() |

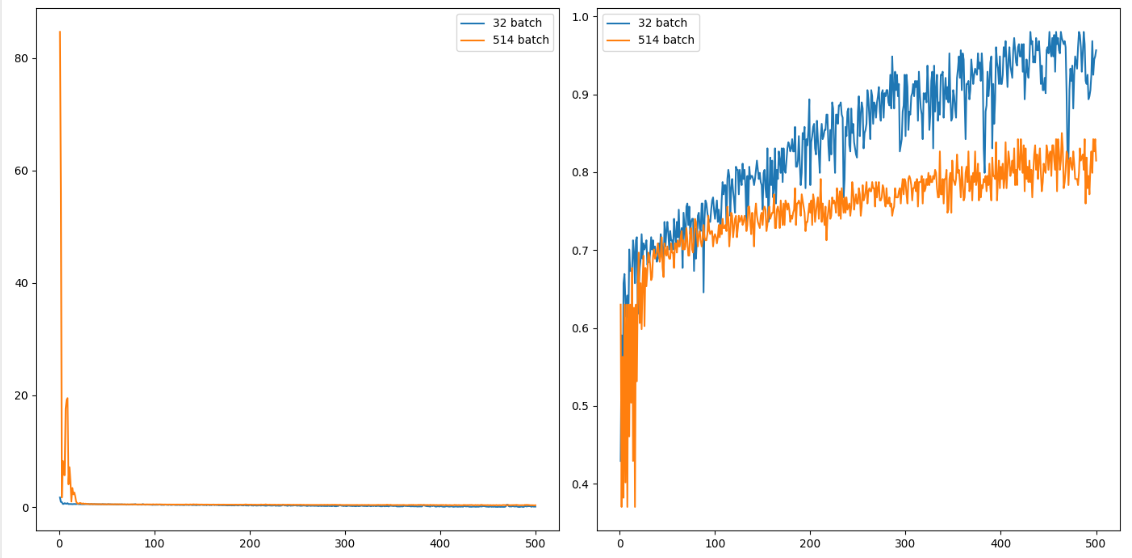
## Configuring Batch Size

Once a reasonable number of nodes and layers have been chosen, the next step involves configuring the batch size. Batch sizes can range between 1 and the size of the training set. You can actually set the batch size to larger than the training set but there is usually no gain from doing that.

Small batches will provide better generalization to help prevent overfitting but they will take longer to train and the results will be more volatile. A large batch size is faster but can lead to overfitting.

Example : Configuring Batch Size

This example builds on the observations that were found in Example 3. The smaller batch size of 32 did really well.



The validation accuracy reading is 0.957 for a batch size of 32 versus 0.815 for a batch size of 514. This is much better than logistic regression which had an accuracy rating of 80.7%.

Could this result be improved? Possibly more epochs and maybe even a smaller batch size would lead to an even better solution. I am happy with the 15% increase in accuracy that I received by tuning though so I will leave it there for now.

|  |
| --- |
| # first neural network with keras tutorial  from keras.models import Sequential  from keras.layers import Dense  import pandas as pd  from sklearn.model\_selection import train\_test\_split  import matplotlib.pyplot as plt  PATH = "/Users/pm/Desktop/DayDocs/data/"  import tensorflow as tf  # load the dataset  df = pd.read\_csv(PATH + 'diabetes.csv', sep=',')  # split into input (X) and output (y) variables  X = df[['Pregnancies','Glucose','BloodPressure','SkinThickness','Insulin','BMI',  'DiabetesPedigreeFunction', 'Age']]  y = df[['Outcome']]  # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.33)  accuracy = []  def buildModel(batchSize):  NUM\_LAYERS = 7  # define the keras model  model = Sequential()  model.add(Dense(230, input\_dim=8, activation='relu', kernel\_initializer='he\_normal'))  for i in range(0, NUM\_LAYERS-1):  model.add(Dense(230, activation='relu', kernel\_initializer='he\_normal'))  model.add(Dense(1, activation='sigmoid'))  opitimizer = tf.keras.optimizers.SGD(  learning\_rate=0.0005, momentum=0.9, name="SGD",  )  # compile the keras model  model.compile(loss='binary\_crossentropy', optimizer=opitimizer, metrics=['accuracy'])  # fit the keras model on the dataset  history = model.fit(X, y, epochs=500, batch\_size=batchSize,  validation\_data=(X\_test, y\_test))  # evaluate the keras model  # Evaluate the model.  loss, acc = model.evaluate(X\_test, y\_test, verbose=0)  print('Test Accuracy: %.3f' % acc)  accuracy.append(acc)  print(accuracy)  return history  def showLoss(history, batchSize):  # Get training and test loss histories  training\_loss = history.history['loss']  validation\_loss = history.history['val\_loss']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  # Visualize loss history for training data.  actualLabel = str(batchSize) + " batch"  plt.subplot(1, 2, 1)  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  def showAccuracy(history, batchSize):  # Get training and test loss histories  training\_loss = history.history['accuracy']  validation\_loss = history.history['val\_accuracy']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 2)  actualLabel = str(batchSize) + " batch"  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  batchSizes = [32, len(y\_train)]  for i in range(0, len(batchSizes)):  history = buildModel(batchSizes[i])  showLoss(history, batchSizes[i])  showAccuracy(history, batchSizes[i])  plt.show() |

## Summary

To tune your parameters:

* Perform regular linear regression or logistic regression to figure out what will happen.
* Configure the optimizer. To keep it simple, use SGD and set the learning rate and momentum. Slower learning rate can help to make it easier to identify the loss trend but takes more time. More momentum can make the learning smoother but too much momentum can make the algorithm insensitive to changes in the solution plane.
* Determine the optimal number of nodes (width) for a single layer network.
* Configure the number of layers using a **relu** activation function.
* Configure the batch size.
* Tweak the learning rate, momentum and number of epochs as needed during the entire process.
* Remember to use loss visualizations and where possible accuracy visualizations to help understand the impact of the tuning parameter values.
* Pay attention to the validation loss and accuracy ratings.

## Other Tips

* Reduce the size of your data set to help speed up training until your model is more finely tuned.
* Emulate existing networks that solve similar problems.
* Use MLPRegressor or MLPClassifer to automate the process of generating a network quickly with a decent set of hyperparameters.

Exercise (2 marks)

Starting with the fluDiagnosis.csv file from the data sets folder, use the following code to perform logistic regression:

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  PATH = "/Users/pm/Desktop/DayDocs/data/"  from sklearn.linear\_model import LogisticRegression  from sklearn import metrics  import numpy as np  # load the dataset  df = pd.read\_csv(PATH + 'fluDiagnosis.csv')  # split into input (X) and output (y) variables  print(df)  X = df[['A','B']]  y = df[['Diagnosed']]  # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.33)  # Perform logistic regression.  logisticModel = LogisticRegression(fit\_intercept=True, random\_state = 0,  solver='liblinear')  logisticModel.fit(X\_train,y\_train)  y\_pred=logisticModel.predict(X\_test)  # Show model coefficients and intercept.  print("\nModel Coefficients: ")  print("\nIntercept: ")  print(logisticModel.intercept\_)  print(logisticModel.coef\_)  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(np.array(y\_test['Diagnosed']), y\_pred,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix")  print(confusion\_matrix) |

Show the accuracy rating and confusion matrix here:

|  |
| --- |
|  |

Exercise (6 marks)

Next, use the following code to configure the optimal learning rate, momentum value and total number of nodes in a single layer network:

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  PATH = "/Users/pm/Desktop/DayDocs/data/"  from keras.models import Sequential  from keras.layers import Dense  import matplotlib.pyplot as plt  import tensorflow as tf  # load the dataset  df = pd.read\_csv(PATH + 'fluDiagnosis.csv')  # split into input (X) and output (y) variables  print(df)  X = df[['A','B']]  y = df[['Diagnosed']]  # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.33)  def buildModel(num\_nodes):  # define the keras model  model = Sequential()  model.add(Dense(num\_nodes, input\_dim=2, activation='relu',  kernel\_initializer='he\_normal'))  model.add(Dense(1, activation='sigmoid'))  opitimizer = tf.keras.optimizers.SGD(  learning\_rate=0.1, momentum=0.9, name="SGD",  )  # compile the keras model  model.compile(loss='binary\_crossentropy', optimizer=opitimizer,  metrics=['accuracy'])  # fit the keras model on the dataset  history = model.fit(X, y, epochs=80, batch\_size=10, validation\_data=(X\_test,  y\_test))  # evaluate the keras model  # Evaluate the model.  loss, acc = model.evaluate(X\_test, y\_test, verbose=0)  print('Test Accuracy: ' + str(acc) + ' Num nodes: ' + str(num\_nodes))  return history  def showLoss(history, numNodes):  # Get training and test loss histories  training\_loss = history.history['loss']  validation\_loss = history.history['val\_loss']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  # Visualize loss history for training data.  actualLabel = str(numNodes) + " nodes"  plt.subplot(1, 2, 1)  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  def showAccuracy(history, numNodes):  # Get training and test loss histories  training\_loss = history.history['accuracy']  validation\_loss = history.history['val\_accuracy']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 2)  actualLabel = str(numNodes) + " nodes"  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  nodeCounts = [170,200,230]  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  for i in range(0, len(nodeCounts)):  history = buildModel(nodeCounts[i])  showLoss(history, nodeCounts[i])  showAccuracy(history, nodeCounts[i])  plt.show() |

After finding decent values for your learning rate, momentum and number of nodes, show a screenshot of the graphs for the loss and accuracy functions here:

|  |
| --- |
|  |

Show the optimal learning rate so far here:

|  |
| --- |
|  |

Show the optimal momentum value so far here:

|  |
| --- |
|  |

Show the optimal number of nodes in a single layer here:

|  |
| --- |
|  |

Exercise (4 marks)

Starting with the code from Exercise 2, configure the optimal number of layers. Examine Example 3 for code hints. Show your full program which implements code to search for the optimal number of layers here. You may or may not need to adjust the learning rate, momentum and number of epochs:

|  |
| --- |
|  |

Exercise (1 mark)

Show the optimal number of layers here:

|  |
| --- |
|  |

Exercise (4 marks)

Show screenshots of the accuracy and loss plots after adjusting the number of layers here:

|  |
| --- |
|  |

Exercise (4 marks)

Starting with the code from Exercise 3, configure the optimal batch size. Examine Example 4 for code hints. Show your full program which implements code to search for the optimal batch size here. You may or may not need to adjust the learning rate, momentum and number of epochs:

|  |
| --- |
|  |

Exercise (1 mark)

Show the optimal batch size here:

|  |
| --- |
|  |

Exercise (4 marks)

Show screenshots of the accuracy and loss plots after adjusting the batch size here:

|  |
| --- |
|  |

Exercise (1 mark)

What type of activation function is used and why?

|  |
| --- |
|  |

Exercise (2 marks)

Compare accuracy ratings for the original logistic regression model and your MLP network model here:

|  |
| --- |
|  |

Exercise (2 marks)

While a multiparameter grid search is a good idea when configuring a network, summarize in your own words an alternate order for configuring hyperparameters as outlined in these notes based on Dr. Brownlee’s recommendations.

|  |
| --- |
|  |