Table of Contents

[References – Better Deep Learning 1](#_Toc114575847)

[Back Propogation Summary 1](#_Toc114575848)

[Hyperparameter Tuning Continued 2](#_Toc114575849)

[Adjusting the Decay Rate 2](#_Toc114575850)

[ReLU – Vanishing Gradients 6](#_Toc114575851)

[Leaky ReLU – Avoiding Dying ReLU 7](#_Toc114575852)

[Overfitting-Early Stopping and Model Save 11](#_Toc114575853)

## References – Better Deep Learning

Decay rates – Chapter 5

ReLU and Vanishing Gradients – Chapter 7

Early Stopping – Chapter 18

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

# Back Propogation Summary

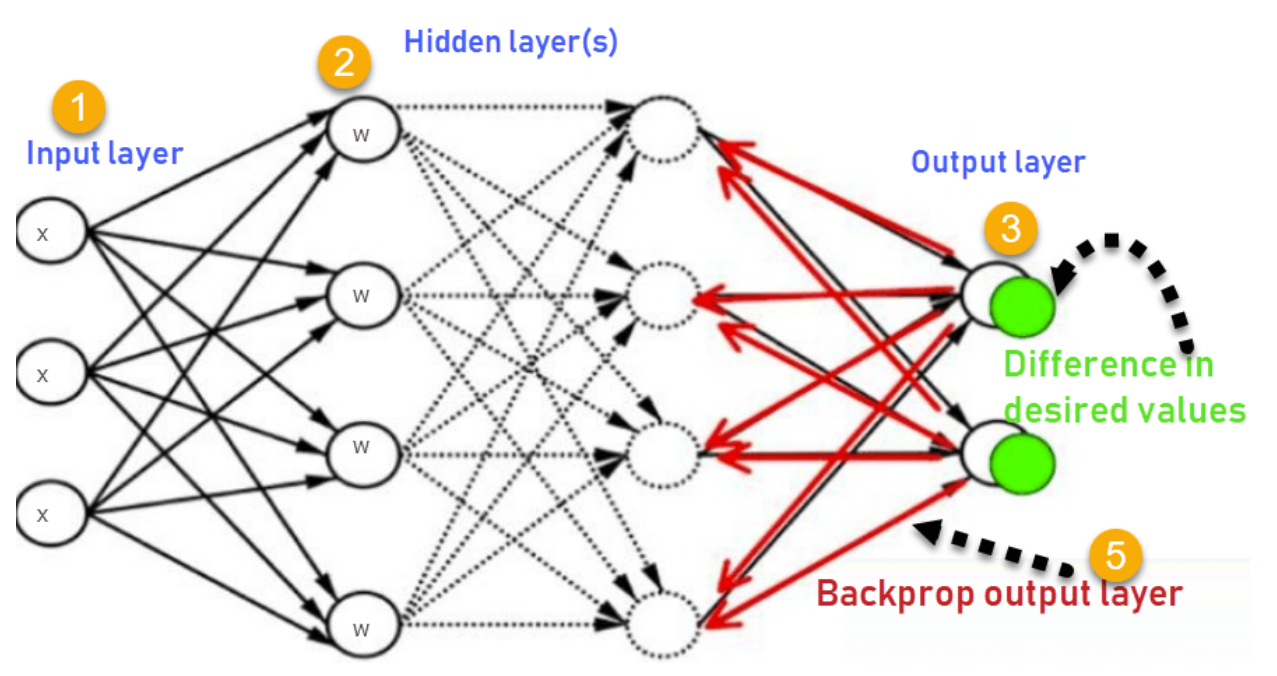
**Hidden Layer(s)**

* Each neuron applies unique weights to attributes and sums the results.
* The activation function formats the sum and sends the neuron output to next layer.

**Output layer**

* Applies loss function.
* Calculates gradients.
* Back propogates changes to hidden layers where weights are updated.

**Repeat**



# Hyperparameter Tuning Continued

This document discusses additional ways to improve your neural networks.

## Adjusting the Decay Rate

Ideally, learning rates should be reduced as the training approaches the solutions. A decay rate can help to gradually reduce the learning rate. The SGD optimizer includes a *decay* attribute to set the decay rate directly:

opitimizer = tf.keras.optimizers.SGD(

learning\_rate=0.0005, momentum=0.9, name="SGD", decay=decay\_rate

)

SGD’s decay attribute makes the following calculation:

lrate = initial\_lrate \* (1 / (1 + decay \* iteration))

Example : Adjusting the Decay Rate

The code in this example builds an MLP network to predict whether a person has diabetes. To establish a reliable conclusion, each decay rate is grid searched multiple times and the results are averaged. Table 1 Figure 1 and Table 1 show that some decay performs better than zero decay. A decay rate of **0.0010** wins with a 91.7% accuracy rate.

Figure : Comparing Models with and without Decay

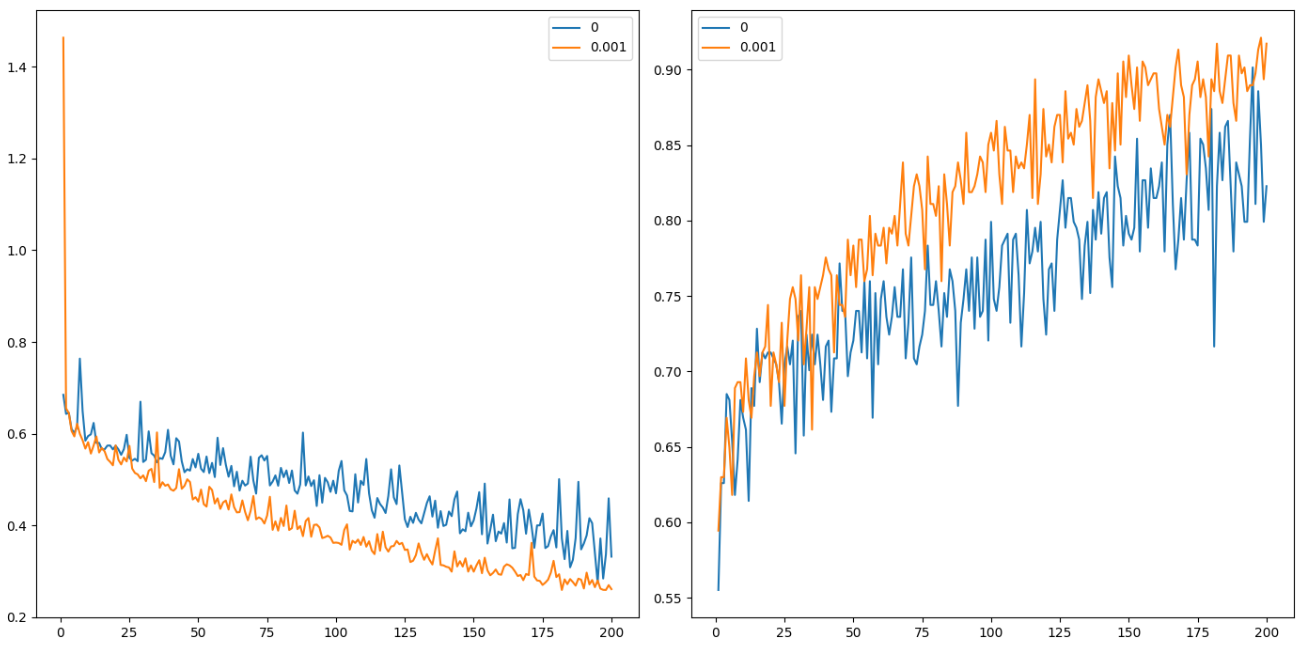


Table : Averaged Decay Rates

|  |
| --- |
| decay acc loss  0 0.000 0.822835 0.332191  1 0.001 0.917323 0.261379 |

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)  resultList = []  def buildModel(decayRate):  # define the keras model  model = Sequential()  model.add(Dense(230, input\_dim=8, activation='relu',  kernel\_initializer='he\_normal'))  NUM\_LAYERS = 7  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", decay=decayRate  )  # 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)  resultList.append({'loss':loss, 'acc':acc, 'decay':decayRate})  print('Test Accuracy: %.3f' % acc)  return history  def showLoss(history, rates):  # 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(rates)  plt.subplot(1, 2, 1)  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  def showAccuracy(history, rates):  # 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(rates)  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, label=actualLabel)  plt.legend()  decayRates = [0, 0.001]  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  for i in range(0, len(decayRates)):  history = buildModel(decayRates[i])  showLoss(history, decayRates[i])  showAccuracy(history, decayRates[i])  plt.show()  import pandas as pd  df = pd.DataFrame(columns=['decay', 'acc', 'loss'])  for result in resultList:  df = df.append(result, ignore\_index=True)  print(df) |

Exercise (2 marks)

Change the decayRates grid search list to the following.

decayRates = [0, 0.001, 0.0005, 0.0001]

What are the accuracy and loss values for the different decay rates when running the code sample?

|  |
| --- |
| decay acc loss  0 0.0000 0.889764 0.271136  1 0.0010 0.897638 0.266808  2 0.0005 0.897638 0.210469  3 0.0001 0.940945 0.189229 |

Which decay rate appears to have the best accuracy and loss rating?

|  |
| --- |
| The decay rate of 0.0001 has the best accuracy of 94.09% and loss rating of 0.1892. |

Show the accuracy and loss plots that appear when grid searching the values in this exercise.

|  |
| --- |
|  |

What do the accuracy and loss plots tell us about the number of epochs?

|  |
| --- |
| Higher epochs tend to result in lower loss and higher accuracy. |

Exercise (3 marks)

Can you find a decay rate that performs better with a different number of epochs?

Yes / No

What number of epochs did you use?

|  |
| --- |
| 300 |

What are the accuracy and loss values for the different decay rates when running the code sample with adjusted epochs?

|  |
| --- |
| decay acc loss  0 0.0000 0.846457 0.299924  1 0.0010 0.921260 0.233803  2 0.0005 0.972441 0.107956  3 0.0001 0.968504 0.106252 |

Show the accuracy and loss plots that appear when grid searching the values in this exercise with the new number of epochs.

|  |
| --- |
|  |

Exercise (2 marks0

Show the accuracy and loss plots and statistics that are displayed when running your optimized solution for Exercise 2.

|  |
| --- |
| decay acc loss  0 0.0000 0.866142 0.239151  1 0.0010 0.854331 0.325965  2 0.0005 0.850394 0.317502  3 0.0001 0.944882 0.124999 |

**Note:** I will not get into the math but a nice explanation of vanishing gradients. However, this article offers a nice summary of how they work.

<https://medium.com/analytics-vidhya/how-batch-normalization-and-relu-solve-vanishing-gradients-3f1a8ace1c88>

## ReLU – Vanishing Gradients

Deep networks were not possible until recently with the emergence of the ReLU activation function. Back propagation with activation functions like **sigmoid** and **tanh** led to vanishing gradients. The derivative of the tanh and sigmoid functions effectively solves to 0 for really large and really small values of X. As a result, sigmoid and tanh activation functions often cause failures to update weights during back propagation. When weights fail to update the neurons stopped learning.

The rectified linear unit (ReLU) helps to avoid the problem for vanishing gradients. The derivative of the activation function for positive values of X solves to 1 so gradients can back propagate (See Figure 2). Some neurons with negative values of X will still die but enough neurons will survive to keep learning most of the time.

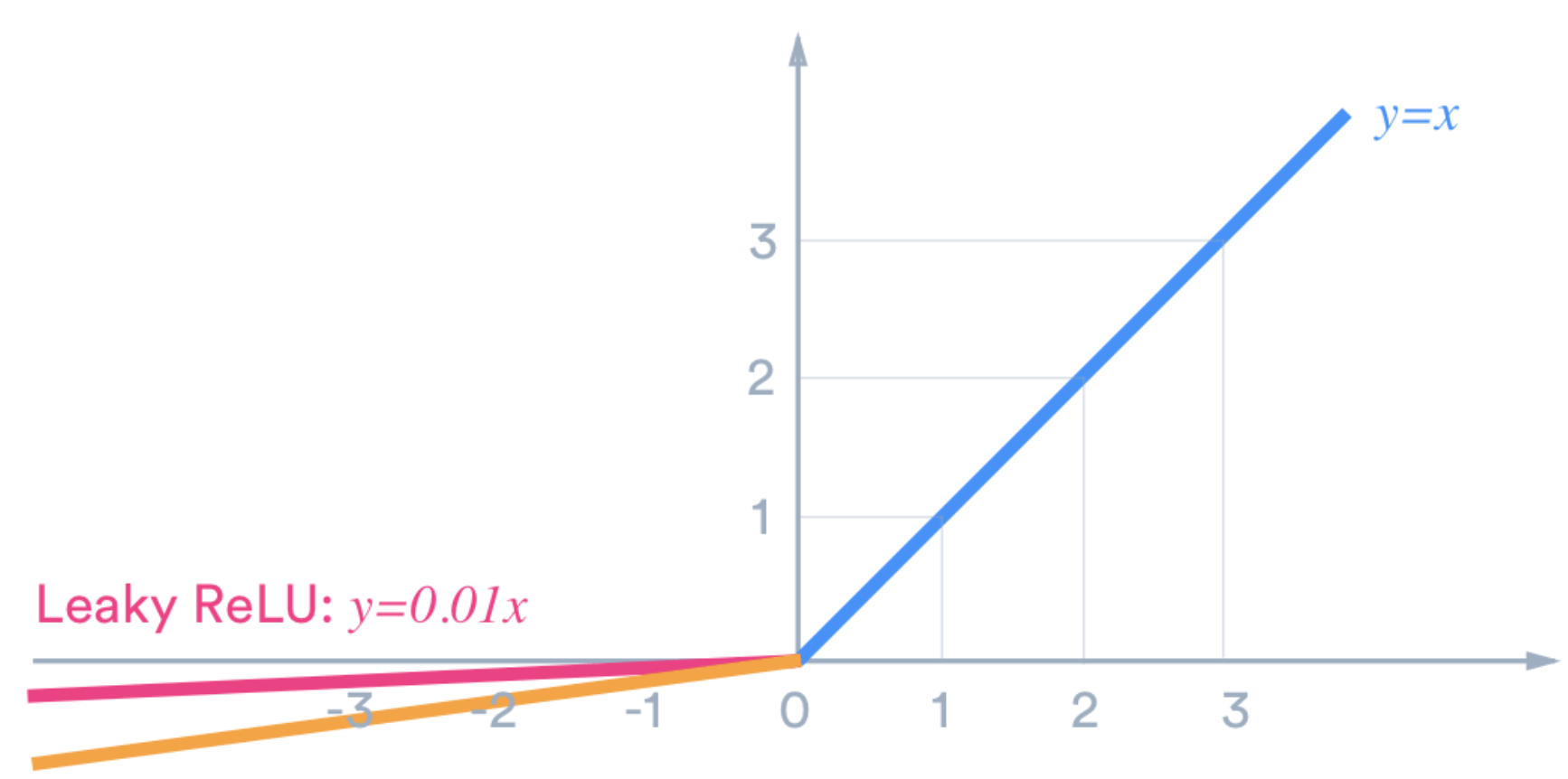
Figure : ReLU Function

|  |  |
| --- | --- |
|  | # ReLu function  def relu(X):  return np.maximum(0,X)  # Example with matrix input of:  # [[1,2,3],[4,5,6]]  relu(mmatrix)  # Output:  array([[1, 2, 3], [4, 5, 6]]) |

## Leaky ReLU – Avoiding Dying ReLU

Activation functions like the Leaky ReLU have been created to avoid the vanishing gradient altogether (see Figure 3). It turns out though that ReLU often performs just as well as the Leaky ReLU so a standard ReLU is more common.

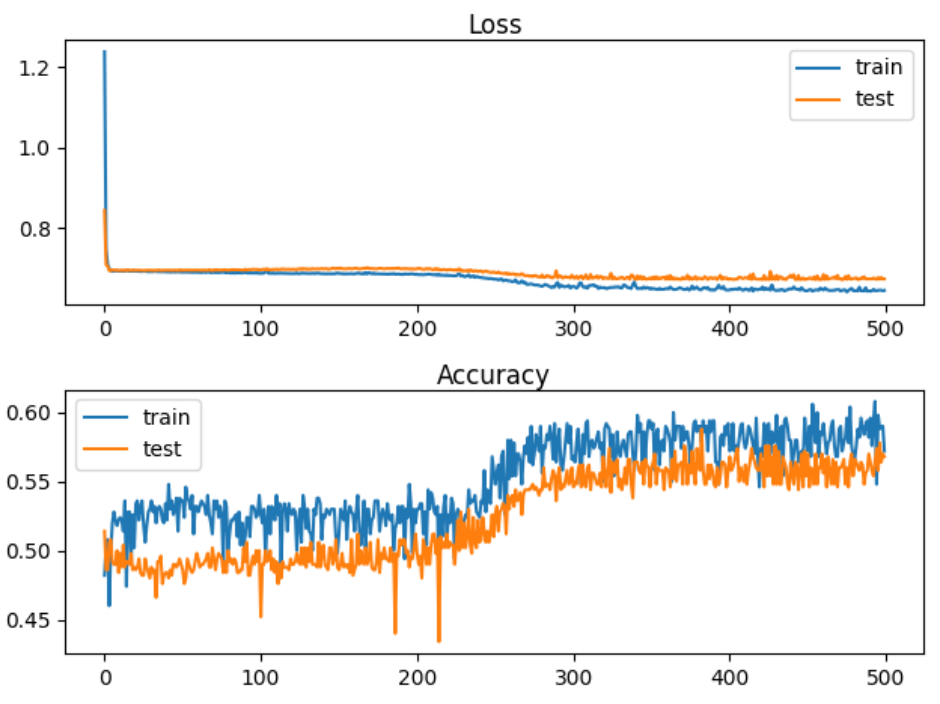
Figure : Leaky ReLU



Example : Under-performing deep network with Tanh function (See chapter 7 of Better Deep Learning)

This example demonstrates a situation where vanishing gradients occurs. For this case, the loss and accuracy curves that are displayed in Figure 4 show that the network is unable to learn. To enable this failure a deep network has been built with the ‘tanh’ activation function.

Figure : Loss and Accuracy Curves for Vanishing Gradients



The quantitative output shows that the accuracy stagnates at around 57% for training and for testing.

Train Accuracy: 0.578, Test Accuracy: 0.568

Here is the code. Note that the ‘tanh’ activation function is used as the activation function with a random uniform kernel initializer.

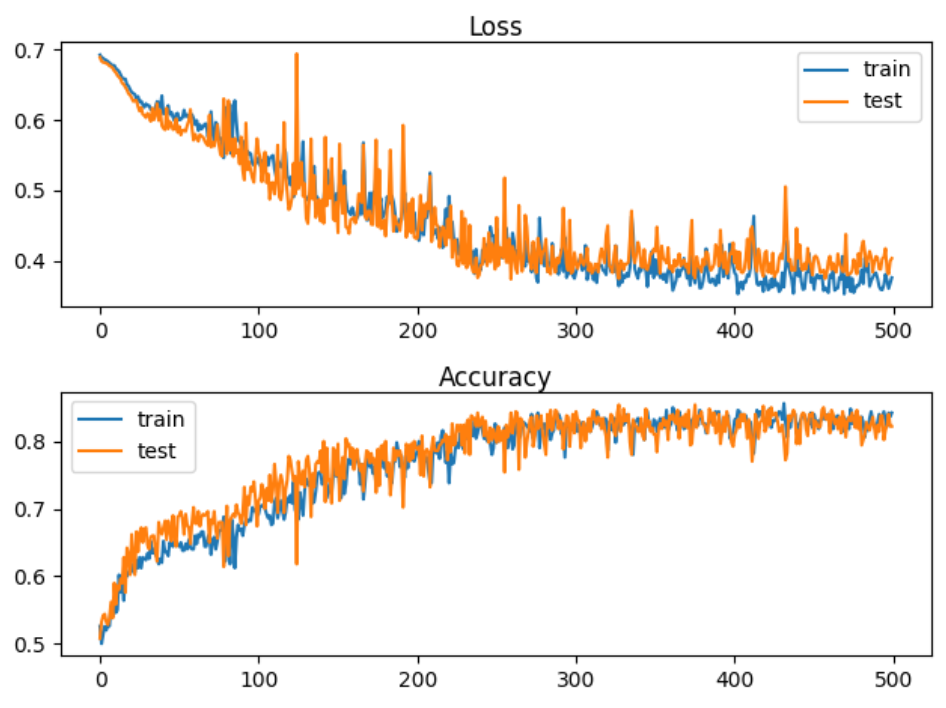
|  |
| --- |
| # deeper mlp with tanh for the two circles classification problem  from sklearn.datasets import make\_circles  from sklearn.preprocessing import MinMaxScaler  from keras.layers import Dense  from keras.models import Sequential  from keras.optimizers import SGD  from keras.initializers import RandomUniform  import matplotlib.pyplot as plt  # Generate 2d classification dataset.  X, y = make\_circles(n\_samples=1000, noise=0.1, random\_state=1)  scaler = MinMaxScaler(feature\_range=(-1, 1))  X = scaler.fit\_transform(X)  # split into train and test  n\_train = 500  trainX, testX = X[:n\_train, :], X[n\_train:, :]  trainy, testy = y[:n\_train], y[n\_train:]  # Define the model.  model = Sequential()  init = RandomUniform(minval=0, maxval=1)  model.add(Dense(5, input\_dim=2, activation='tanh', kernel\_initializer=init))  model.add(Dense(5, activation='tanh', kernel\_initializer=init))  model.add(Dense(5, activation='tanh', kernel\_initializer=init))  model.add(Dense(5, activation='tanh', kernel\_initializer=init))  model.add(Dense(5, activation='tanh', kernel\_initializer=init))  model.add(Dense(1, activation='sigmoid', kernel\_initializer=init))  # Compile model.  opt = SGD(lr=0.01, momentum=0.9)  model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=['accuracy'])  # fit model  history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=500, verbose=1)  # evaluate the model  \_, train\_acc = model.evaluate(trainX, trainy, verbose=0)  \_, test\_acc = model.evaluate(testX, testy, verbose=0)  print('Train Accuracy: %.3f, Test Accuracy: %.3f' % (train\_acc, test\_acc))  # Plot loss learning curves.  plt.subplot(211)  plt.title('Loss', pad=-40)  plt.plot(history.history['loss'], label='train')  plt.plot(history.history['val\_loss'], label='test')  plt.legend()  # Plot accuracy learning curves.  plt.subplot(212)  plt.title('Accuracy', pad=-40)  plt.plot(history.history['accuracy'], label='train')  plt.plot(history.history['val\_accuracy'], label='test')  plt.legend()  plt.show() plt.show() |

Example : Using RELU to Enable Deeper Networks (See Chapter 7 of Better Deep Learning)

This example shows how to solve the vanishing gradient problem that occurs in Example 2. A ReLU activation function with an ‘he\_uniform’ kernel\_initializer performs much better. The train and test accuracy scores jump by approximately 25% after implementing the ReLU activation functions.

Train Accuracy: 0.832, Test Accuracy: 0.824

Figure : Accuracy and Loss Curves when using the ReLU activation function with ‘he\_normal’.



To build this model, replace the model definition code in Example 2 with this version. The difference with this new code is in activation and kernel\_initializer function which enable the deep network:

|  |
| --- |
| # Define the model.  model = Sequential()  model.add(Dense(5, input\_dim=2, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dense(1, activation='sigmoid')) |

Exercise (3 marks)

The code that was presented in Example 1 is used to build a network for diagnosing heart disease. Starting with the code that is used in Exercise 1, replace the input and hidden layer definitions with the following definitions. These definitions use a tanh activation function with a RandomUniform kernel initializer.

|  |
| --- |
| init = RandomUniform(minval=0, maxval=1)  model.add(Dense(230, input\_dim=8, activation='tanh',  kernel\_initializer=init))  NUM\_LAYERS = 7  for i in range(0, NUM\_LAYERS-1):  model.add(Dense(230, activation='tanh', kernel\_initializer=init)) |

Remember to include the RandomUniform import:

|  |
| --- |
| from keras.initializers import RandomUniform |

How do the accuracy and loss levels from the output in this exercise compare with the accuracy and loss levels that are generated in Example 1Exercise 1? Explain why this occurred.

|  |
| --- |
| decay acc loss  0 0.0000 0.665354 0.641172  1 0.0010 0.665354 0.637703  2 0.0005 0.665354 0.638860  3 0.0001 0.665354 0.638983  The use of tanh activation function for back propagation led to a vanishing gradient. The derivative of the tanh functions effectively solves to 0 for really large and really small values of X. As a result, the tanh activation function caused a failure to update weights during back propagation. When weights fail to update the neurons stopped learning. |

Exercise (3 marks)

Show the loss and accuracy plots which are displayed when running the code in Exercise 4. Explain what happened.

|  |
| --- |
| The tanh activation function is failing to update the weights as you can see on the loss plot. The accuracy goes down then spikes back up after failing to update. The accuracy plot shows a flat line, also showing the failure of updating the weights. |

Example : Leaky ReLU

To implement a leaky ReLU replace the model definition code inside example 3 with the following:

|  |
| --- |
| import tensorflow as tf  leakyReLU = tf.keras.layers.LeakyReLU(alpha=0.3)  # Define the model.  model = Sequential()  model.add(Dense(5, input\_dim=2, activation=leakyReLU, kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation=leakyReLU, kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation=leakyReLU, kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation=leakyReLU, kernel\_initializer='he\_uniform'))  model.add(Dense(5, activation=leakyReLU, kernel\_initializer='he\_uniform'))  model.add(Dense(1, activation='sigmoid')) |

## Overfitting-Early Stopping and Model Save

Early stopping is a technique to help prevent over-fitting. Early stopping exits training when either losses increase or accuracy begins to diminish. A patience property can be included to set an additional number of epochs that must occur before the routine ends training.

An additional helpful technique involves saving the very best model during training. EarlyStopping and ModelCheckpoint work together to save you time and to provide you with the best model. Both features are set up with a callback.

Two important parameters of the EarlyStopping() function are:

**min\_delta**: Minimum change in the monitored quantity to qualify as an improvement.

An absolute change of less than min\_delta, will count as no improvement.

**patience**: Number of epochs with no improvement after which training will be

stopped.

# Simple early stopping.

es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, min\_delta=0.1, patience=200)

mc = ModelCheckpoint('best\_model.h5', monitor='val\_accuracy', mode='max', verbose=1,

save\_best\_only=True)

# Fit model.

history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=4000, verbose=0,

callbacks=[es, mc])

The model can then be loaded later.

# Load the saved model.

saved\_model = load\_model('best\_model.h5')

The following code snippet above allows you to save the model when the best validation accuracy. However, you can also trigger a model save based on minimum validation loss.

tf.keras.callbacks.ModelCheckpoint(

filepath, monitor='val\_loss', verbose=0, save\_best\_only=False,

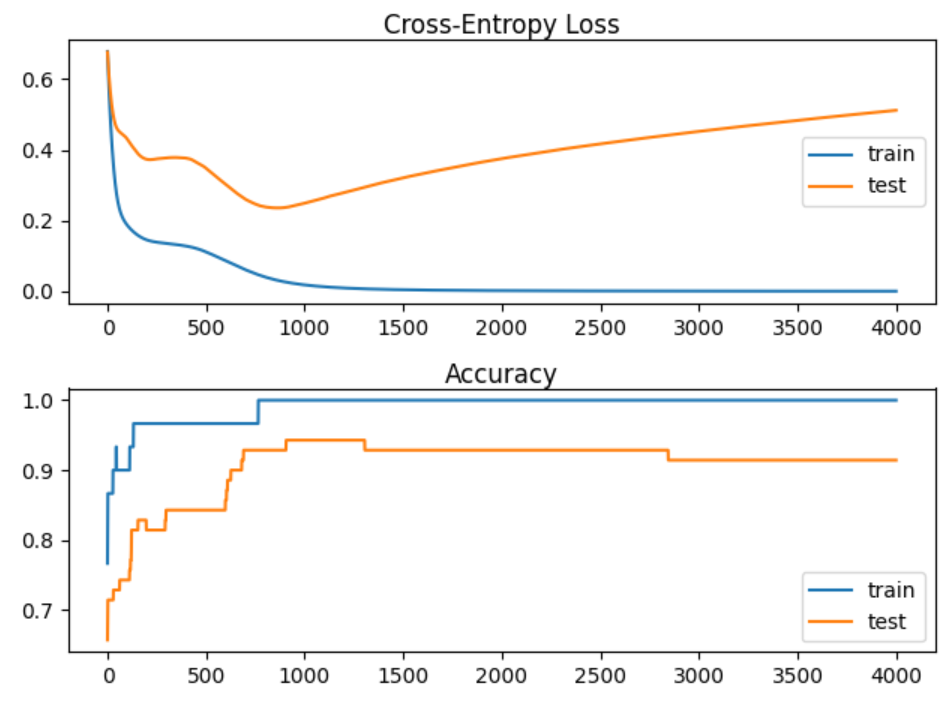
save\_weights\_only=False, mode='auto', save\_freq='epoch', options=None, \*\*kwargs

)

Example : Preventing overfitting with early-stopping and model save.

This example demonstrates a network that fails due to overfitting. Figure 6 shows a typical overfitting situation where a loss curve which starts out by gradually declining and then rising.

Figure : Failed Network due to Overfitting



Here is the code to set up and evaluate the failed network:

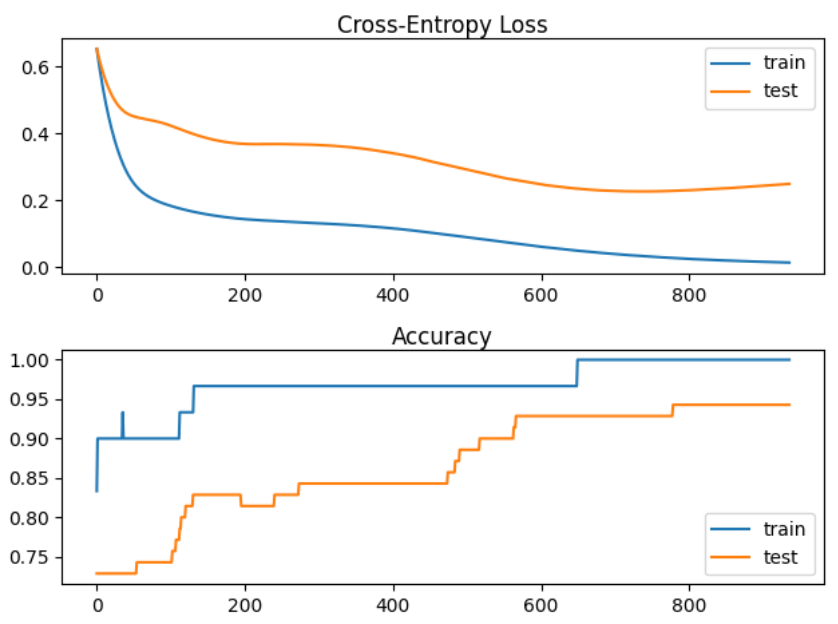
|  |
| --- |
| # Creating an overfit situation with the moons data set.  from sklearn.datasets import make\_moons  from keras.layers import Dense  from keras.models import Sequential  import matplotlib.pyplot as plt  # Generate 2d classification dataset.  X, y = make\_moons(n\_samples=100, noise=0.2, random\_state=1)  # Split data into train and test.  n\_train = 30  trainX, testX = X[:n\_train, :], X[n\_train:, :]  trainy, testy = y[:n\_train], y[n\_train:]  # Define the model.  model = Sequential()  model.add(Dense(500, input\_dim=2, activation='relu'))  model.add(Dense(1, activation='sigmoid'))  model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])  # Fit the model.  history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=4000, verbose=1)  # Evaluate the model.  train\_loss, train\_acc = model.evaluate(trainX, trainy, verbose=0)  test\_loss, test\_acc = model.evaluate(testX, testy, verbose=0)  print('Train accuracy: %.3f, Test accuracy: %.3f' % (train\_acc, test\_acc))  print('Train loss: %.3f, Test loss: %.3f' % (train\_loss, test\_loss))  # Plot loss learning curves.  plt.subplot(211)  plt.title('Cross-Entropy Loss', pad=-40)  plt.plot(history.history['loss'], label='train')  plt.plot(history.history['val\_loss'], label='test')  plt.legend()  # Plot accuracy learning curves.  plt.subplot(212)  plt.title('Accuracy', pad=-40)  plt.plot(history.history['accuracy'], label='train')  plt.plot(history.history['val\_accuracy'], label='test')  plt.legend()  plt.show() |

Example : Early Stopping

The code needed to save the best model and to implement early stopping w

To build this example, replace the green highlighted section in Example 5 with this version.

|  |
| --- |
| # Fit the model.  from keras.callbacks import EarlyStopping  from keras.callbacks import ModelCheckpoint  from keras.models import load\_model  # simple early stopping  # Apparently we must set patience to at least 1 for it to work on certain machines.  es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1,  min\_delta=0.1, patience=1)  mc = ModelCheckpoint('best\_model.h5', monitor='val\_loss', mode='min', verbose=1,  save\_best\_only=True)  # fit model  history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=4000, verbose=0,  callbacks=[es, mc])  # load the saved model  model = load\_model('best\_model.h5') |



Train accuracy: 1.000, Test accuracy: 0.943

Train loss: 0.026, Test loss: 0.235

Exercise (3 marks)

Study the output after running Example 6. How many epochs executed? How many epochs would have executed without the patience attribute? How many epochs would have executed without early stopping?

|  |
| --- |
| 2 epochs were executed, 4000 would have been executed without the early stopping. 0 epochs may have executed without patience=1 attribute. |

Exercise (2 marks)

Show a screenshot of the model that is saved in your project folder.

|  |
| --- |
|  |

Exercise (3 marks)

Replace the model checkpoint code in Example 6 with this version which saves the best model based on the lowest validation loss.

|  |
| --- |
| mc = ModelCheckpoint('best\_model.h5', monitor='val\_loss', mode='auto', verbose=1,  save\_best\_only=True) |

Run the code and show the output. Compare the output here with the output in Example 6. How is the output different?

|  |
| --- |
| Original:  Epoch 1: val\_loss improved from inf to 0.67217, saving model to best\_model.h5  Epoch 2: val\_loss improved from 0.67217 to 0.66171, saving model to best\_model.h5  Epoch 2: early stopping  Train accuracy: 0.833, Test accuracy: 0.671  Train loss: 0.635, Test loss: 0.662  Replaced:  Epoch 1: val\_loss improved from inf to 0.67791, saving model to best\_model.h5  Epoch 2: val\_loss improved from 0.67791 to 0.66746, saving model to best\_model.h5  Epoch 2: early stopping  Train accuracy: 0.833, Test accuracy: 0.614  Train loss: 0.660, Test loss: 0.667  The output is very similar. The replaced code uses mode=’auto’ instead of mode=’min’. This is the same because mode=’auto’ uses a value of ‘min’ for ‘val\_loss’ to minimize loss. |