Table of Contents

[References – Better Deep Learning 1](#_Toc189465068)

[ReLU – Vanishing Gradients 1](#_Toc189465069)

[Leaky ReLU – Avoiding Dying ReLU 2](#_Toc189465070)

[Overfitting-Early Stopping and Model Save 6](#_Toc189465071)

[Exploding Gradients 10](#_Toc189465072)

[Norm Scaling 12](#_Toc189465073)

[Gradient Clipping 14](#_Toc189465074)

[Importance of Scaling 14](#_Toc189465075)

[Input and Output Scaling 16](#_Toc189465076)

[Greedy Layer-wise Pretraining 16](#_Toc189465077)

## References – Better Deep Learning

ReLU and Vanishing Gradients – Chapter 7

Early Stopping – Chapter 18

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

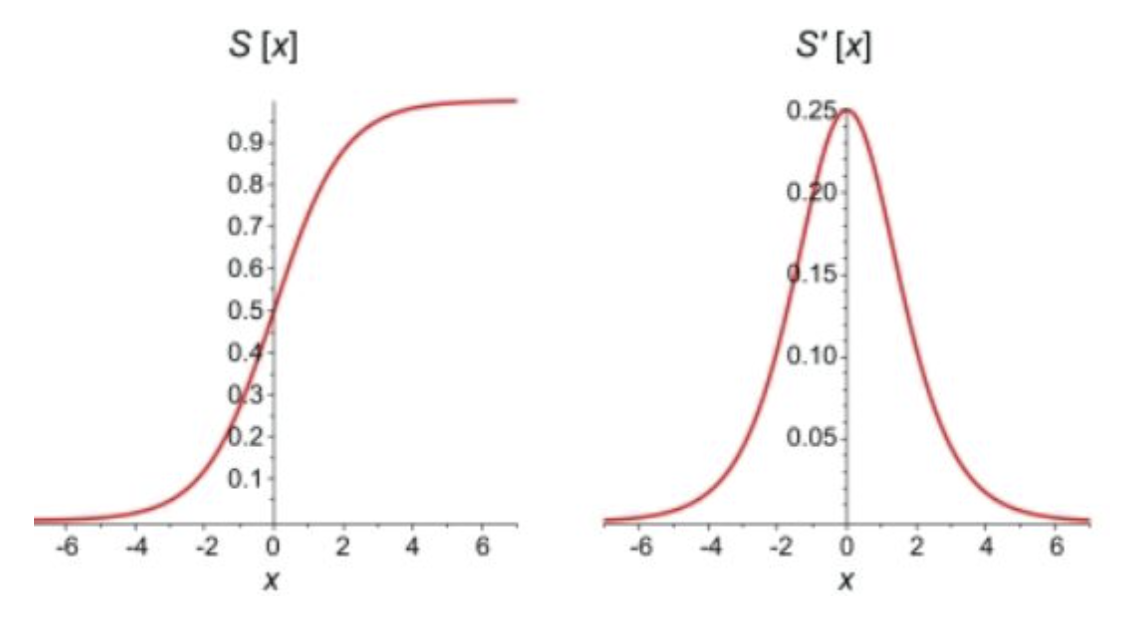
## ReLU – Vanishing Gradients

**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>

Deep networks were not possible until recently with the emergence of the ReLU activation function. Back propagation with activation functions like **sigmoid** led to vanishing gradients. For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25 (refer to Figure 1).

Figure : Sigmoid function and Derivative of Sigmoid Function



When there are more layers in the network, the **product of derivatives decreases** until at some point the partial derivative of the loss function approaches a value close to zero. When this happens, the partial derivative vanishes (0.25\*0.25\*0.25 leads to a small number). We call this the vanishing gradient problem. As a result, sigmoid and tanh activation functions (and others) often cause failures to update weights during back propagation so learning stops in deep networks.

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). Neurons with negative values of X will still die but enough neurons will survive to keep learning most of the time.

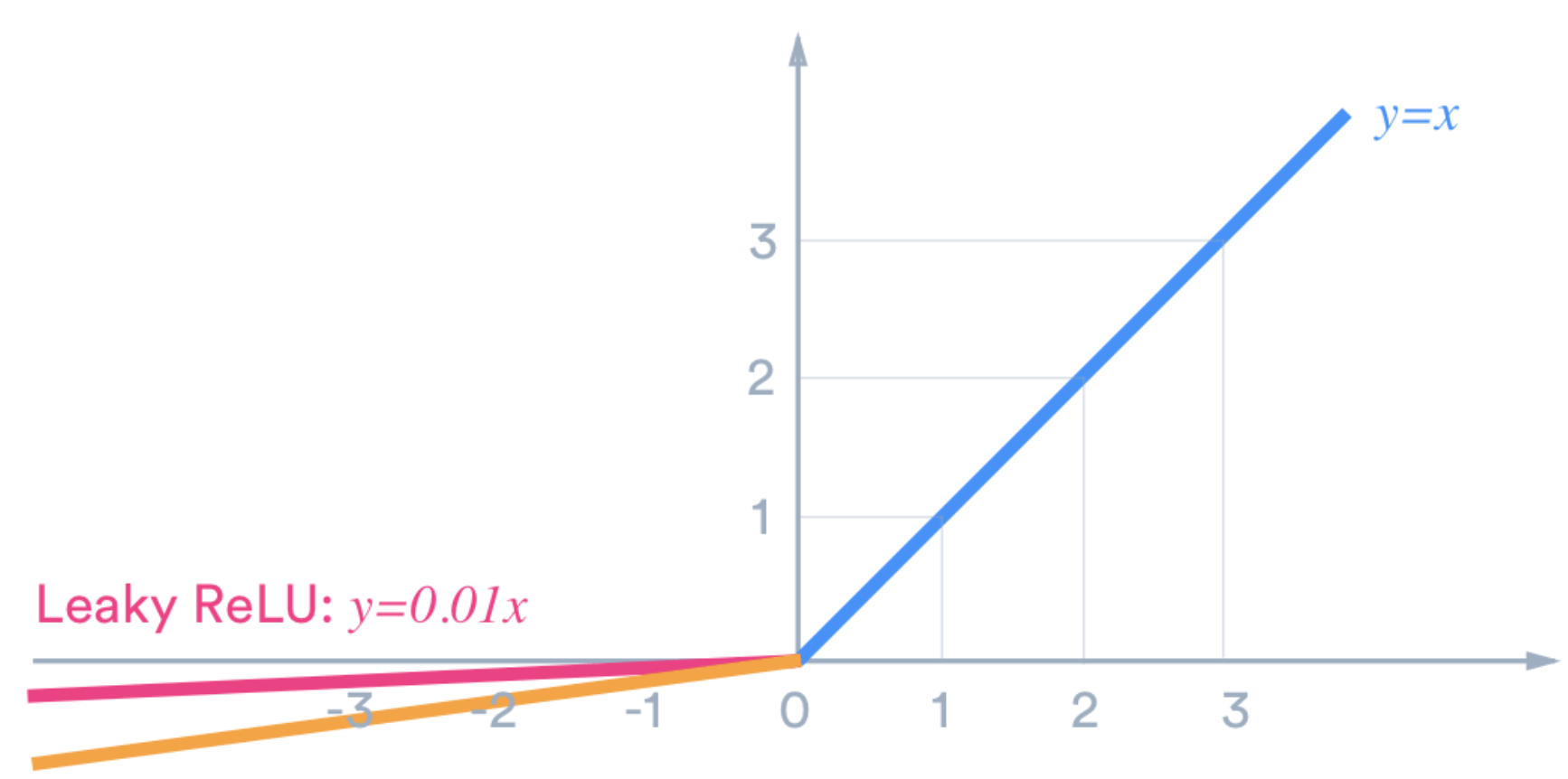
Figure : ReLU Function

|  |  |
| --- | --- |
|  | # 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



The blue lines in Figure 4 show the derivative of the ReLU function. Derivative values are equal to 1 when ReLU > 0 so weights will not vanish when multiplying derivatives during back propagation.

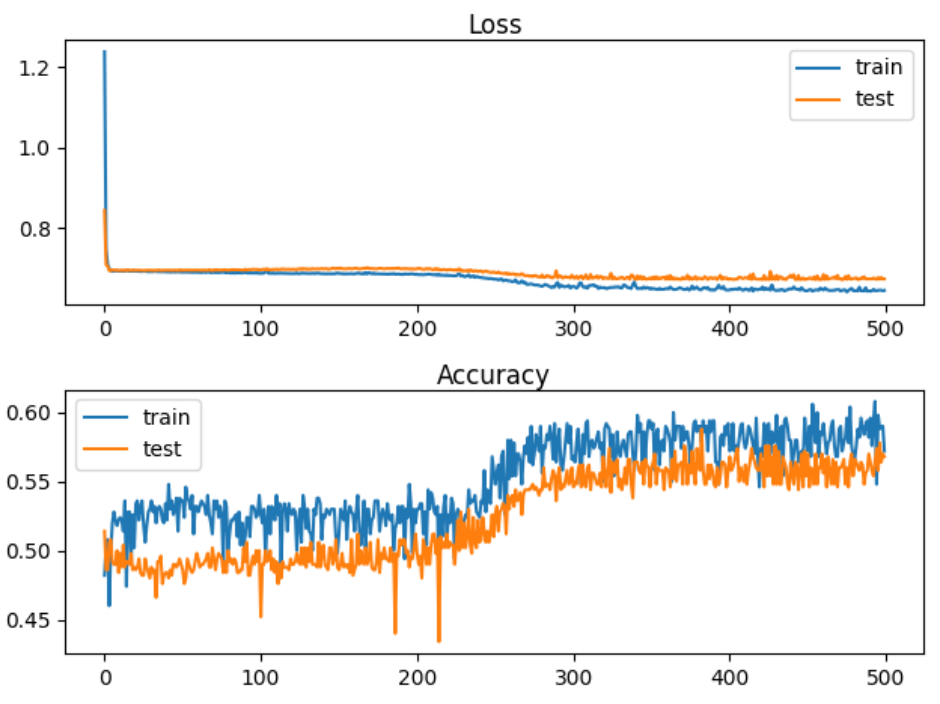
Figure : ReLU (red) and Derivative of ReLU (blue)



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 5 show that the network is unable to learn. To enable this failure a deep network has been built with the ‘sigmoid’ 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 ‘sigmoid’ activation function is used as the activation function.

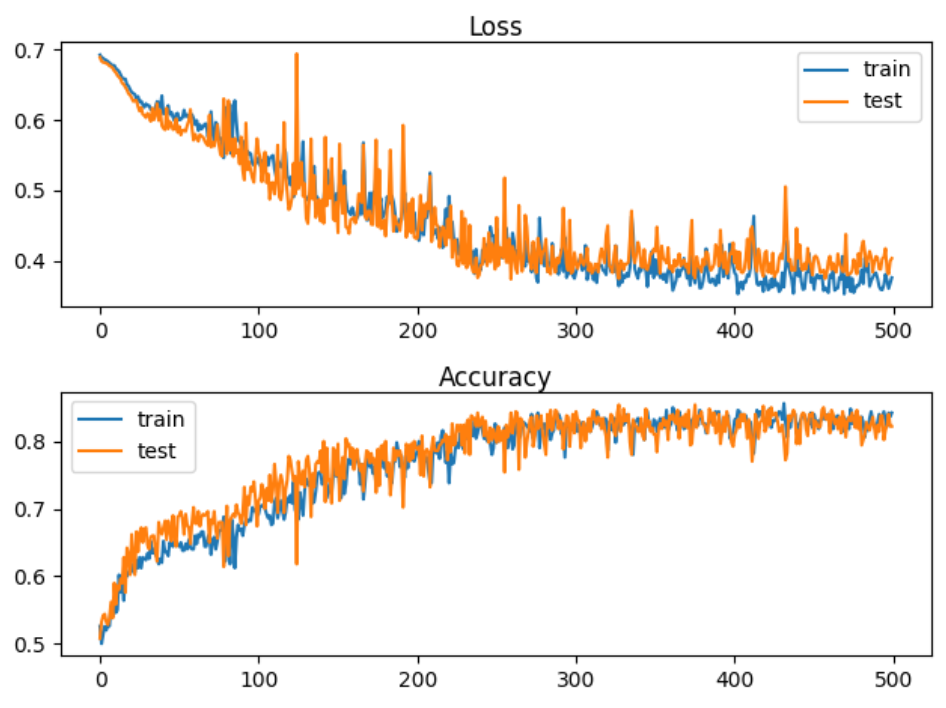
|  |
| --- |
| # deeper mlp with tanh for the two circles classification problem  from sklearn.datasets import make\_circles  from sklearn.preprocessing import MinMaxScaler  from keras.layers import Dense  from keras.models import Sequential  from tensorflow.keras.optimizers import SGD  from keras.initializers 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 = 'he\_uniform'  model.add(Dense(5, input\_dim=2, activation='sigmoid', kernel\_initializer=init))  model.add(Dense(5, activation='sigmoid', kernel\_initializer=init))  model.add(Dense(5, activation='sigmoid', kernel\_initializer=init))  model.add(Dense(5, activation='sigmoid', kernel\_initializer=init))  model.add(Dense(5, activation='sigmoid', kernel\_initializer=init))  model.add(Dense(1, activation='sigmoid'))  # 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 1. A ReLU activation function 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 1 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()  init = 'he\_uniform'  model.add(Dense(5, input\_dim=2, activation='relu', kernel\_initializer=init))  model.add(Dense(5, activation='relu', kernel\_initializer=init))  model.add(Dense(5, activation='relu', kernel\_initializer=init))  model.add(Dense(5, activation='relu', kernel\_initializer=init))  model.add(Dense(5, activation='relu', kernel\_initializer=init))  model.add(Dense(1, activation='sigmoid')) # output layer |

Example : Leaky ReLU

To implement a leaky ReLU replace the model definition code inside Example 2 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

)

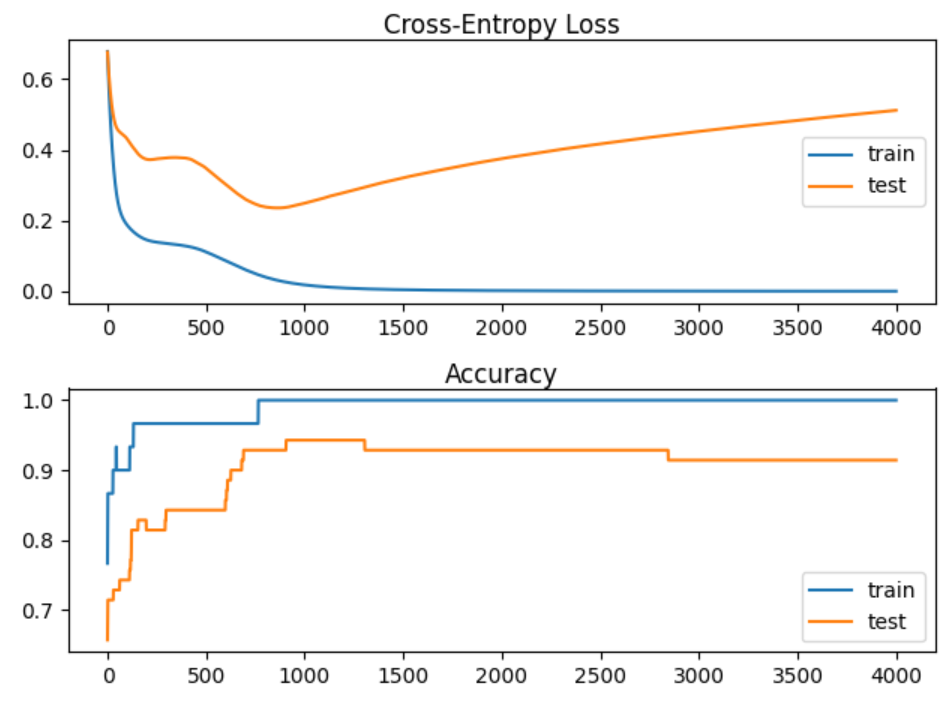
Example : Preventing overfitting with early-stopping and model save.

This example demonstrates a network that fails due to overfitting. Figure 7 shows a typical overfitting situation where a test loss curve which starts out by gradually declining and then rising.

Figure : Failed Network due to Overfitting

Train accuracy: 1.000, Test accuracy: 0.914

Train loss: 0.000, Test loss: 0.497



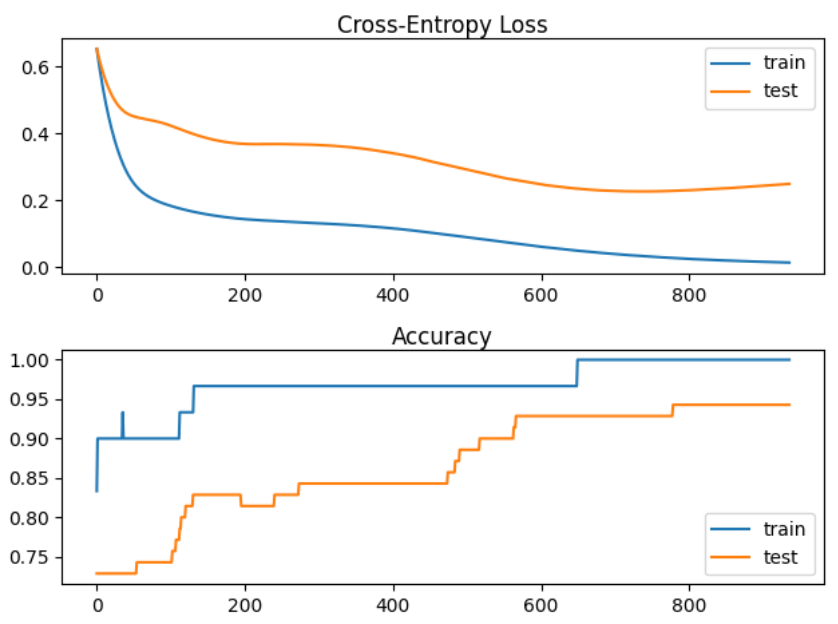
Here is the code to set up and evaluate the failed network:

|  |
| --- |
| # 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

This code implements a save of the best model and early stopping to avoid prevent the model from getting worse when it stops learning. To build this example, replace the green highlighted section in Example 4 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  # patience: # of epochs observed where no improvement before exiting.  # mode: Could be max, min, or auto.  # min\_delta: Amount of change needed to be considered an improvement.  es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, min\_delta=0.000001, patience=200)  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') |



The improvement is evident in the graph where no overfitting occurs. However, only modest gains were achieved in accuracy.

Train accuracy: 1.000, Test accuracy: 0.929

Exercise (2 marks)

Study the output after running Example 5. How many epochs executed? How many epochs would have executed without the patience attribute? How many epochs would have executed without early stopping?

|  |
| --- |
|  |

Exercise (2 marks)

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

|  |
| --- |
|  |

Exercise (3 marks)

Change the implementation of early stopping so it maximizes val\_accuracy.

es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, min\_delta=0.000001,

patience=200)

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

save\_best\_only=True)

The parameters that need changing are shown above. Note, in this case it will not improve results but it is important to know where to configure these settings. This is the final output that appears when running the code from Example 5. Notice that it is reporting changes for val\_loss.



Show a screenshot of the final output when making the changes in this exercise. Also be sure to include a section which shows that val\_accuracy did not improve.

|  |
| --- |
|  |

## Exploding Gradients

Exploding gradients occur when large error gradients accumulate during training, causing the model's weights to become excessively large. This often leads to unstable training, where the model fails to converge or produces NaN (not a number) values. Essentially, instead of gradually improving, the model's updates become so large that they disrupt learning. This situation is more common with LSTM’s (long short term memory networks) where the context from several past gradients are included with the back propagation.

Exploding gradients can occur due to:

* Improperly scaled inputs to the network.
* Poorly chosen learning rates that allow for large weight updates.
* Loss functions which allow the calculation of large error values.

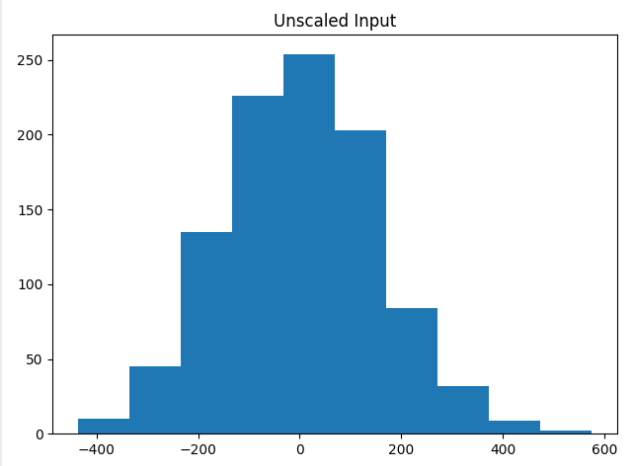
Gradient clipping can be used to fix this error. There are two ways to clip gradients:

* Norm scaling
* Norm clipping

Example : Bad Exploding Gradient (See Chapter 8 of Better Deep Learning)

This example causes exploding gradients because the inputs are not scaled. The sample data has 20 unscaled features. The y-value is unscaled as well (see Figure 8).

Figure : Unscaled y-values



The unscaled data caused the model weights to explode during training. Very large errors were generated while the data was fed through the network. The network was unable to learn anything. As a result, the predictions are NaN values which leads to MSE values that are also not numbers.

Train MSE: nan, Test MSE: nan

Here is the code which sets up the data and network that causes the exploding gradients.

|  |
| --- |
| # mlp with unscaled data for the regression problem  from sklearn.datasets import make\_regression  from keras.layers import Dense  from keras.models import Sequential  from tensorflow.keras.optimizers import SGD  import matplotlib.pyplot as plt  # Generate the regression dataset.  X, y = make\_regression(n\_samples=1000, n\_features=20, noise=0.1, random\_state=1)  plt.hist(y)  plt.title("Unscaled Input")  plt.show()  # 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()  model.add(Dense(25,input\_dim=20, activation='relu',kernel\_initializer='he\_uniform'))  model.add(Dense(1, activation='linear'))  # Compile the model.  model.compile(loss='mean\_squared\_error', optimizer=SGD(lr=0.01, momentum=0.9))  # Fit the model.  history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=100, verbose=1)  # Evaluate the model.  train\_mse = model.evaluate(trainX, trainy, verbose=0)  test\_mse = model.evaluate(testX, testy, verbose=0)  print('Train MSE: %.3f, Test MSE: %.3f' % (train\_mse, test\_mse))  # Plot losses during training.  plt.title('Losses')  plt.plot(history.history['loss'], label='train')  plt.plot(history.history['val\_loss'], label='test')  plt.legend()  plt.show() |

### Norm Scaling

One way to clip the gradients involves norm scaling. When weights are greater than a specific value, they are rescaled to a chosen value. Norm scaling is implemented with the *clipnorm* attribute. In this case clipnorm is set to 1 but other values could have been used.

opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)

A suitable range for clipnorm in stochastic gradient descent (SGD) is typically between 1.0 and 5.0. Clipping the norm of gradients helps prevent exploding gradients by limiting the size of the updates to the model's weights. The exact value may depend on the specific dataset and model architecture, so some experimentation is often necessary to find the best threshold.

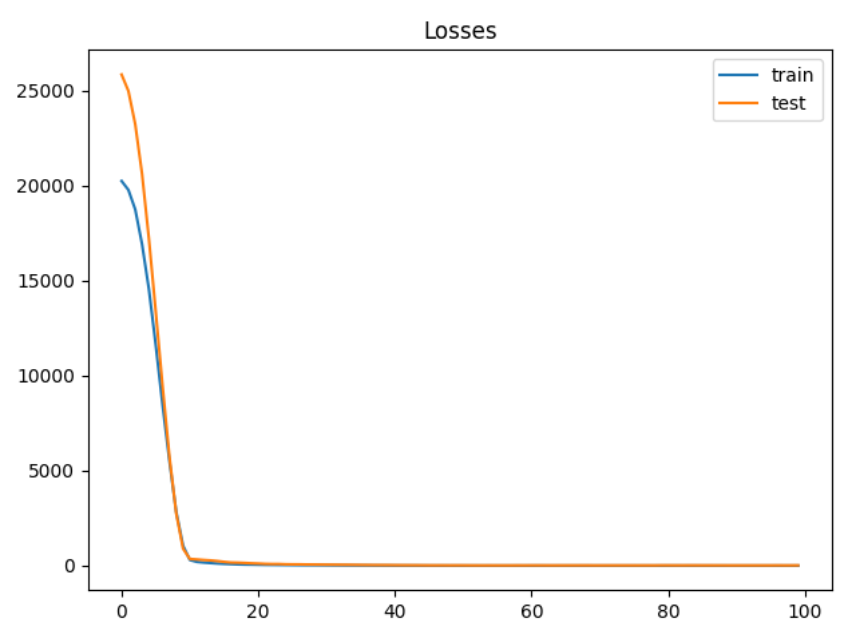
Example : Norm Scaling (See Chapter 8 Better Deep Learrning)

A way to fix the exploding gradient issue presented in Example 6 is to replace the compilation statement with one that includes a *clipnorm* attribute in the optimizer.

|  |
| --- |
| opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)  model.compile(loss='mean\_squared\_error', optimizer=opt) |

When running the example now, training is possible. An MSE value of 16.874 is generated.

Train MSE: 1.158, Test MSE: 16.874



Exercise (2 marks)

Starting with the following code, determine if you can find a better clip norm:

|  |
| --- |
| # mlp with unscaled data for the regression problem  from sklearn.datasets import make\_regression  from keras.layers import Dense  from keras.models import Sequential  from tensorflow.keras.optimizers import SGD  import matplotlib.pyplot as plt  # Generate the regression dataset.  X, y = make\_regression(n\_samples=1000, n\_features=20, noise=0.1, random\_state=1)  plt.hist(y)  plt.title("Unscaled Input")  plt.show()  # Split into train and test.  n\_train = 500  trainX, testX = X[:n\_train, :], X[n\_train:, :]  trainy, testy = y[:n\_train], y[n\_train:]  clipResults = []  def buildModel(clipSize):  # Define the model.  model = Sequential()  model.add(Dense(25,input\_dim=20, activation='relu',kernel\_initializer='he\_uniform'))  model.add(Dense(1, activation='linear'))  # Compile the model.  opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)  model.compile(loss='mean\_squared\_error', optimizer=opt)  # Fit the model.  history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=100, verbose=1)  # Evaluate the model.  train\_mse = model.evaluate(trainX, trainy, verbose=0)  test\_mse = model.evaluate(testX, testy, verbose=0)  print('Train MSE: %.3f, Test MSE: %.3f' % (train\_mse, test\_mse))  clipResults.append({'train mse': train\_mse, 'test mse':test\_mse,  'clip size':clipSize})  # Plot losses during training.  plt.title('Losses')  plt.plot(history.history['loss'], label='train')  plt.plot(history.history['val\_loss'], label='test')  plt.legend()  plt.show()  clipSizes = [0.9, 1, 1.1, 2]  for i in range(0, len(clipSizes)):  buildModel(clipSizes[i])  for clipResult in clipResults:  print(clipResult) |

Try to improve the clips scale. Show your optimal clip norm scale here and explain why you chose it.

|  |
| --- |
|  |

### Gradient Clipping

Another solution involves clipping gradients if they become too large. A suitable range for clipvalue in stochastic gradient descent (SGD) is typically between -0.5 and 0.5, but it can vary depending on the model and data. clipvalue works by limiting the maximum and minimum values of the gradients during training. If a gradient exceeds this threshold, it is set to the limit, preventing excessively large updates that can destabilize the training process.

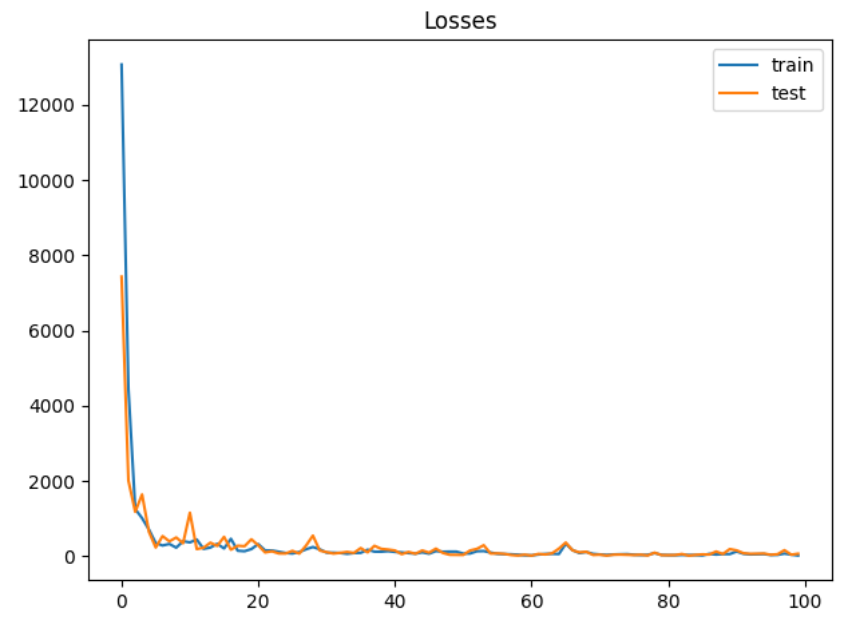
Example : Norm Clipping (See Chapter 8 Better Deep Learning)

To build this example, replace the compile statement in Example 6 with this code.

|  |
| --- |
| opt = SGD(lr=0.01, momentum=0.9, clipvalue=5.0)  model.compile(loss='mean\_squared\_error', optimizer=opt) |

Train MSE: 50.182, Test MSE: 71.380

Figure : Loss Functions with Gradient Clipping



Exercise (2 marks)

Modify the code in Exercise 4 to so it implements a grid search on clip value. Show your revised program here:

|  |
| --- |
|  |

Show an optimized value here and show the MSE.

|  |
| --- |
|  |

## Importance of Scaling

Since neural networks rely heavily on properly sized weights for the model, scaling the data often matters.

Example : Unscaled Data

For comparison, this example generates mse values from regression without scaling.

|  |
| --- |
| Train loss: 10.055, Test loss: 13.413 |

Here is the code.

|  |
| --- |
| from sklearn.datasets import make\_regression  from keras.layers import Dense  from keras.models import Sequential  from tensorflow.keras.optimizers import SGD  from matplotlib import pyplot  # Generate regression set.  X, y = make\_regression(n\_samples=1000, n\_features=20,  noise=0.1, random\_state=1)  # Split data into train and test.  n\_train = 500  trainX, testX = X[:n\_train, :], X[n\_train:, :]  trainy, testy = y[:n\_train], y[n\_train:]  normSizeEvaluations = []  # Define the model.  model = Sequential()  model.add(Dense(25, input\_dim=20, activation='relu',  kernel\_initializer='he\_uniform'))  model.add(Dense(1, activation='linear'))  # Compile the model.  model.compile(loss='mean\_squared\_error',  optimizer=SGD(lr=0.01, momentum=0.9, clipvalue=1.05))  # Fit the model.  history = model.fit(trainX, trainy,  validation\_data=(testX, testy),  epochs=200, verbose=1)  # Evaluate the model.  train\_mse = model.evaluate(trainX, trainy, verbose=0)  test\_mse = model.evaluate(testX, testy, verbose=0)  print('Train loss: %.3f, Test loss: %.3f' % (train\_mse, test\_mse))  normSizeEvaluations.append({'train mse':train\_mse,  'test mse':test\_mse,  'size':1})  # Plot the loss during training.  pyplot.title('Mean Squared Error - norm size: ')  pyplot.plot(history.history['loss'], label='train')  pyplot.plot(history.history['val\_loss'], label='test')  pyplot.legend()  pyplot.show() |

## Input and Output Scaling

As you may have guessed, gains can be achieved by scaling the X and Y values.

Example : Input and Output Scaling

For this case we can achieve better gains by scaling the y-value. The output is:

|  |
| --- |
| Train loss: 0.002, Test loss: 0.004 |

To implement this solution, replace the compile instruction in Example 9 with this version.

|  |
| --- |
| # Compile the model.  model.compile(loss='mean\_squared\_error',  optimizer=SGD(lr=0.01, momentum=0.9))  from sklearn.preprocessing import StandardScaler  # reshape 1d arrays to 2d arrays  trainy = trainy.reshape(len(trainy), 1)  testy = testy.reshape(len(trainy), 1)  # Scale y  scaler = StandardScaler()  scaler.fit(trainy)  trainy = scaler.transform(trainy)  testy = scaler.transform(testy)  # Scale x  xscaler = StandardScaler()  xscaler.fit(trainX)  trainX = xscaler.transform(trainX)  testX = xscaler.transform(testX) |

## Greedy Layer-wise Pretraining

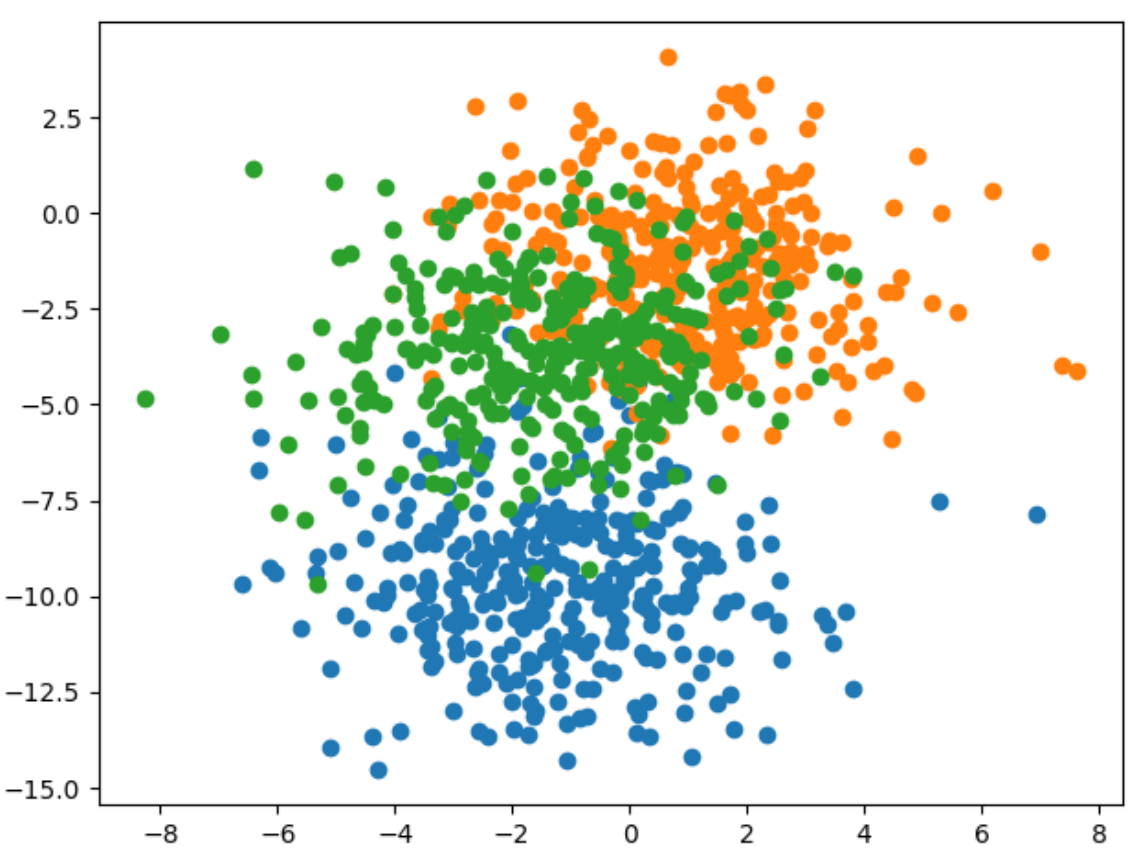
Greedy layer-wise pretraining involves successively adding a new hidden layer to a model and refitting, allowing the newly added model to learn the inputs from the existing hidden layer, often while keeping the weights for the existing hidden layers. Sometimes, but not always, being able to build layers in isolation can lead to a better model.

The algorithm begins with a base model just like we have created before. However, when adding new layers, the output layer is stored and removed, existing layers are set so they cannot be modified, a new layer is added and then the output layer is added back.

|  |
| --- |
| # Add one new layer and re-train only the new layer.  def add\_layer(model, trainX, trainy):  # Store the output layer.  output\_layer = model.layers[-1]  # Remove the output layer.  model.pop()  # Mark all remaining layers as non-trainable.  for layer in model.layers:  layer.trainable = False  # Add a new hidden layer.  model.add(Dense(10, activation='relu', kernel\_initializer='he\_uniform'))  # Add the output layer back.  model.add(output\_layer)  # fit model  model.fit(trainX, trainy, epochs=100, verbose=1)  return model |

Example : Data Setup (See Chapter 20 of Better Deep Learning)

This example sets up the data for greedy layer-wise pretraining classification problem by generating three separate clusters.



Here is the code that generates and displays the data for the example

|  |
| --- |
| # scatter plot of blobs dataset  from sklearn.datasets import make\_blobs  from matplotlib import pyplot  from numpy import where  # generate 2d classification dataset  X, y = make\_blobs(n\_samples=1000, centers=3, n\_features=2, cluster\_std=2, random\_state=2)  # scatter plot for each class value  for class\_value in range(3):  # select indices of points with the class label  row\_ix = where(y == class\_value)  # scatter plot for points with a different color  pyplot.scatter(X[row\_ix, 0], X[row\_ix, 1])  # show plot  pyplot.show() |

Example : Greedy Layer-wise Pretraining

Here is the entire code needed to implement greedy layer-wise pretraining.

|  |
| --- |
| from sklearn.datasets import make\_blobs  from keras.layers import Dense  from keras.models import Sequential  from tensorflow.keras.optimizers import SGD  from tensorflow.keras.utils import to\_categorical  import matplotlib.pyplot as plt  # Generate the data.  def prepare\_data():  X, y = make\_blobs(n\_samples=1000, centers=3,  n\_features=2, cluster\_std=2, random\_state=2)  y = to\_categorical(y)  n\_train = 500  trainX, testX = X[:n\_train, :], X[n\_train:, :]  trainy, testy = y[:n\_train], y[n\_train:]  return trainX, testX, trainy, testy  # Build the base model.  def get\_base\_model(trainX, trainy):  # Define the model.  model = Sequential()  model.add(Dense(10, input\_dim=2, activation='relu',  kernel\_initializer='he\_uniform'))  model.add(Dense(3, activation='softmax'))  # Compile the model.  opt = SGD(lr=0.01, momentum=0.9)  model.compile(loss='categorical\_crossentropy', optimizer=opt,  metrics=['accuracy'])  # Fit the model.  model.fit(trainX, trainy, epochs=100, verbose=0)  return model  stats = []  # Evaluate the model.  def evaluate\_model(numLayers, model, trainX, testX, trainy, testy):  train\_loss, train\_acc = model.evaluate(trainX, trainy, verbose=1)  test\_loss, test\_acc = model.evaluate(testX, testy, verbose=1)  stats.append({ '# layers':numLayers, 'train\_acc':train\_acc, 'test\_acc':test\_acc,  'train\_loss':train\_loss, 'test\_loss':test\_loss })  # Add one new layer and re-train only the new layer.  def add\_layer(model, trainX, trainy):  # Store the output layer.  output\_layer = model.layers[-1]  # Remove the output layer.  model.pop()  # Mark all remaining layers as non-trainable.  for layer in model.layers:  layer.trainable = False  # Add a new hidden layer.  model.add(Dense(10, activation='relu', kernel\_initializer='he\_uniform'))  # Add the output layer back.  model.add(output\_layer)  # fit model  model.fit(trainX, trainy, epochs=100, verbose=1)  return model  # Get the data and build the base model.  trainX, testX, trainy, testy = prepare\_data()  model = get\_base\_model(trainX, trainy)  # Evaluate the base model  scores = dict()  evaluate\_model(-1, model, trainX, testX, trainy, testy)  # add layers and evaluate the updated model  n\_layers = 10  for i in range(n\_layers):  model = add\_layer(model, trainX, trainy)  evaluate\_model(i, model, trainX, testX, trainy, testy)  import pandas as pd  columns = ['# layers', 'train\_acc', 'test\_acc', 'train\_loss', 'test\_loss']  df = pd.DataFrame(columns=columns)  for i in range(0, len(stats)):  df = df.\_append(stats[i], ignore\_index=True)  print(df) |

The test accuracy appears to improve. The first row shows the full model with all layers when no greedy layer-wise pretraining is used. However, three rows with greedy layer-wise pretraining shows some slight improvement in accuracy.

|  |
| --- |
| # layers train\_acc test\_acc train\_loss test\_loss  0 -1.0 0.836 0.818 0.398433 0.413762  1 0.0 0.836 0.824 0.388216 0.407569  2 1.0 0.838 0.822 0.385031 0.411448  3 2.0 0.828 0.830 0.396971 0.411076  4 3.0 0.822 0.820 0.390866 0.409562  5 4.0 0.844 0.822 0.383811 0.418827  6 5.0 0.828 0.802 0.396735 0.437487  7 6.0 0.840 0.824 0.392394 0.433040  8 7.0 0.840 0.816 0.380537 0.412962  9 8.0 0.838 0.826 0.384223 0.426899  10 9.0 0.818 0.836 0.405867 0.413323 |

Exercise (4 marks)

This code uses greedy layer-wise pretraining to add layers to a model which performs diabetes diagnosis classification.

|  |
| --- |
| from sklearn.datasets import make\_blobs  from keras.layers import Dense  from keras.models import Sequential  from tensorflow.keras.optimizers import SGD  from tensorflow.keras.utils import to\_categorical  import matplotlib.pyplot as plt  # Generate the data.  import tensorflow as tf  from sklearn.model\_selection import train\_test\_split  import pandas as pd  def prepare\_data():  PATH = "/Users/pm/Desktop/DayDocs/data/"  # 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)  return X\_train, X\_test, y\_train, y\_test  # Build the base model.  def get\_base\_model(trainX, trainy):  # define the keras model  model = Sequential()  model.add(Dense(230, 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.  model.fit(trainX, trainy, epochs=200, batch\_size=10)  return model  stats = []  # Evaluate the model.  def evaluate\_model(numLayers, model, trainX, testX, trainy, testy):  train\_loss, train\_acc = model.evaluate(trainX, trainy, verbose=1)  test\_loss, test\_acc = model.evaluate(testX, testy, verbose=1)  stats.append({ '# layers':numLayers, 'train\_acc':train\_acc, 'test\_acc':test\_acc,  'train\_loss':train\_loss, 'test\_loss':test\_loss })  # Add one new layer and re-train only the new layer.  def add\_layer(model, trainX, trainy):  # Store the output layer.  output\_layer = model.layers[-1]  # Remove the output layer.  model.pop()  # Mark all remaining layers as non-trainable.  for layer in model.layers:  layer.trainable = False  # Add a new hidden layer.  model.add(Dense(230, activation='relu', kernel\_initializer='he\_uniform'))  # Add the output layer back.  model.add(output\_layer)  # fit model  model.fit(trainX, trainy, epochs=300, verbose=1)  return model  # Get the data and build the base model.  trainX, testX, trainy, testy = prepare\_data()  model = get\_base\_model(trainX, trainy)  # Evaluate the base model  scores = dict()  evaluate\_model(-1, model, trainX, testX, trainy, testy)  # add layers and evaluate the updated model  n\_layers = 14  for i in range(n\_layers):  model = add\_layer(model, trainX, trainy)  evaluate\_model(i, model, trainX, testX, trainy, testy)  columns = ['# layers', 'train\_acc', 'test\_acc', 'train\_loss', 'test\_loss']  df = pd.DataFrame(columns=columns)  for i in range(0, len(stats)):  df = df.\_append(stats[i], ignore\_index=True)  print(df) |

Run the code and find the optimal number of layers. Show the output here and highlight the optimal number of layers based on test accuracy:

|  |
| --- |
|  |

Exercise (1 mark)

Which line of code in Exercise 6 saves the output layer for use later?

|  |
| --- |
|  |

Exercise (1 mark)

Which line of code in Exercise 6 adds the output layer that is stored in memory back onto the network.

|  |
| --- |
|  |

Exercise (1 mark)

Which line of code in Exercise 6 removes the output layer so a new layer can be added?

|  |
| --- |
|  |

Exercise (1 mark)

Which lines of code in Exercise 6 ensures that the previously added hidden layers cannot be adjusted?

|  |
| --- |
|  |

Exercise (1 mark)

Explain in your own words how greedy layer-wise pretraining works.

|  |
| --- |
|  |