Table of Contents

[Exploding Gradients 1](#_Toc83665153)

[Norm Scaling 3](#_Toc83665154)

[Gradient Clipping 5](#_Toc83665155)

[Importance of Scaling 5](#_Toc83665156)

[Input and Output Scaling 7](#_Toc83665157)

[Greedy Layer-wise Pretraining 7](#_Toc83665158)

## Exploding Gradients

Exploding gradients can occur when overflowing gradients are back propagated. This situation is more common with LSTM’s 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.

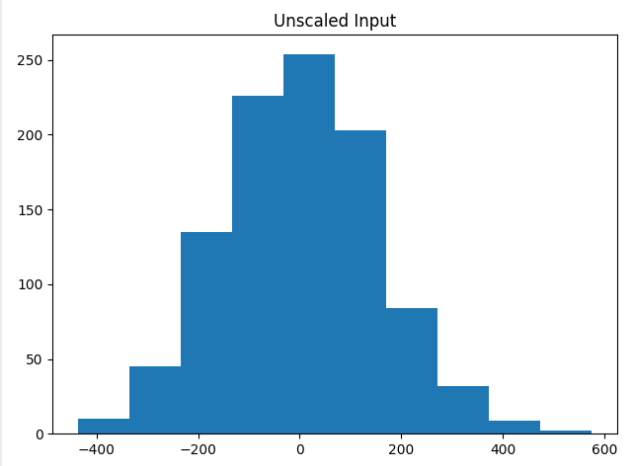
Gradient clipping can be used to fix this error. There are two ways to clip gradients:

* 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 1).

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 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)

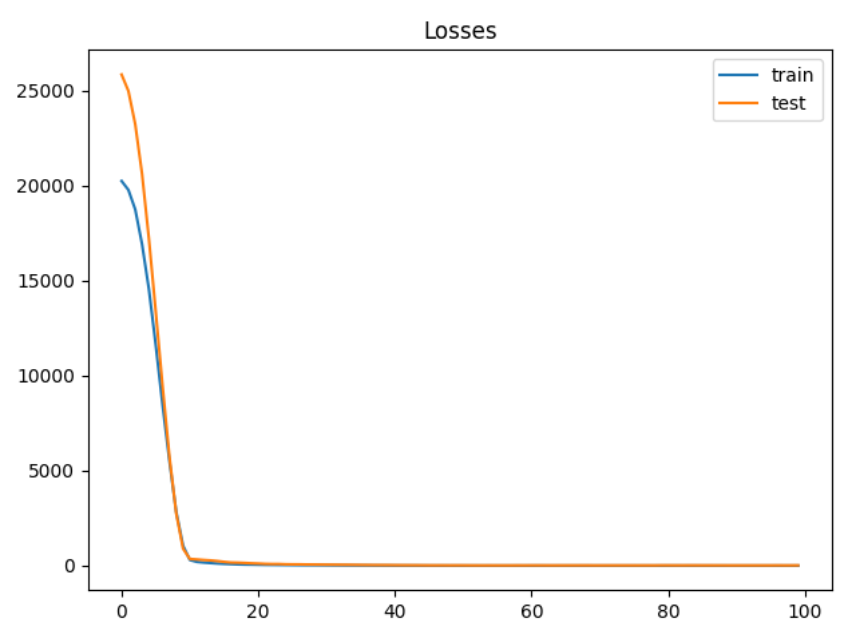
Example : Norm Scaling (See Chapter 8 Better Deep Learrning)

A way to fix the exploding gradient issue presented in Example 2 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 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.

|  |
| --- |
| {'train mse': 0.7066060900688171, 'test mse': 14.414047241210938, 'clip size': 0.7}  {'train mse': 0.45103055238723755, 'test mse': 16.370412826538086, 'clip size': 0.8}  {'train mse': 0.40858176350593567, 'test mse': 15.985372543334961, 'clip size': 0.9}  **{'train mse': 0.4082964062690735, 'test mse': 14.278223037719727, 'clip size': 0.95}**  {'train mse': 1.2649400234222412, 'test mse': 14.364845275878906, 'clip size': 1}  {'train mse': 0.9165008664131165, 'test mse': 16.532474517822266, 'clip size': 1.05}  {'train mse': 1.1797434091567993, 'test mse': 15.216545104980469, 'clip size': 1.1}  {'train mse': 0.9404534697532654, 'test mse': 17.028005599975586, 'clip size': 1.3}  {'train mse': 1.9586129188537598, 'test mse': 21.995697021484375, 'clip size': 2}  Optimal clip norm scale is 0.95 because it has the lowest MSE out of the clip sizes tested. |

### Gradient Clipping

Another solution involves clipping gradients if they become too large.

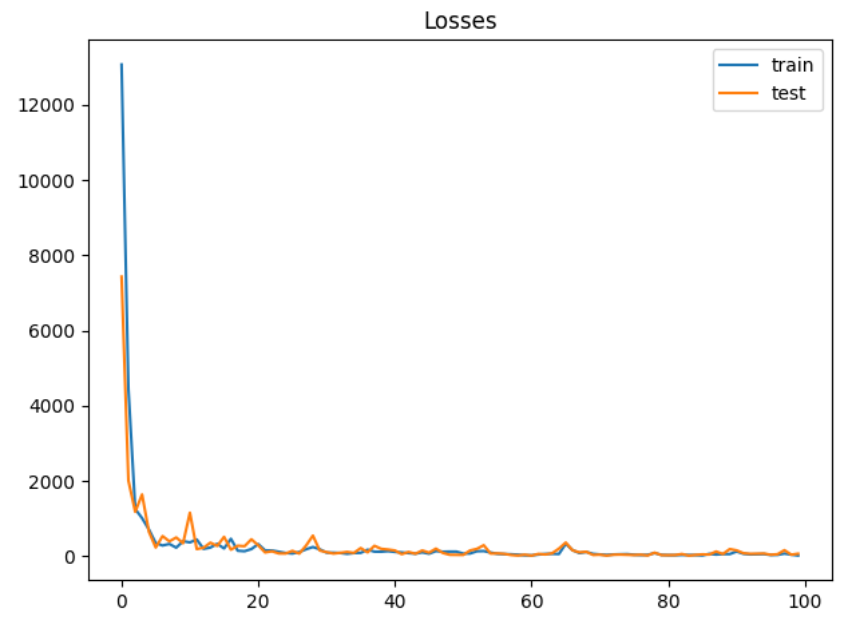
Example : Norm Clipping (See Chapter 8 Better Deep Learning)

To build this example, replace the compile statement in Example 1 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 1 to so it implements a grid search on clip value. Show your revised program here:

|  |
| --- |
| # 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 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(clipValueSize):  # 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, clipvalue=clipValueSize)  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 value':clipValueSize})  # 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()  clipValueSizes = [2,3,4,5,6,7] for i in range(0, len(clipValueSizes)):  buildModel(clipValueSizes[i])  for clipResult in clipResults:  print(clipResult) |

Show an optimized value here and show the MSE.

|  |
| --- |
| {'train mse': 52.36796951293945, 'test mse': 89.49208068847656, 'clip value': 2}  {'train mse': 80.23286437988281, 'test mse': 101.4061279296875, 'clip value': 3}  {'train mse': 49.562679290771484, 'test mse': 61.44961929321289, 'clip value': 4}  {'train mse': 81.38309478759766, 'test mse': 100.7852554321289, 'clip value': 5}  **{'train mse': 36.556434631347656, 'test mse': 37.80767059326172, 'clip value': 6}**  {'train mse': 69.10971069335938, 'test mse': 71.18988800048828, 'clip value': 7}  Optimized at a clipvalue of 6. |

## 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 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 4 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

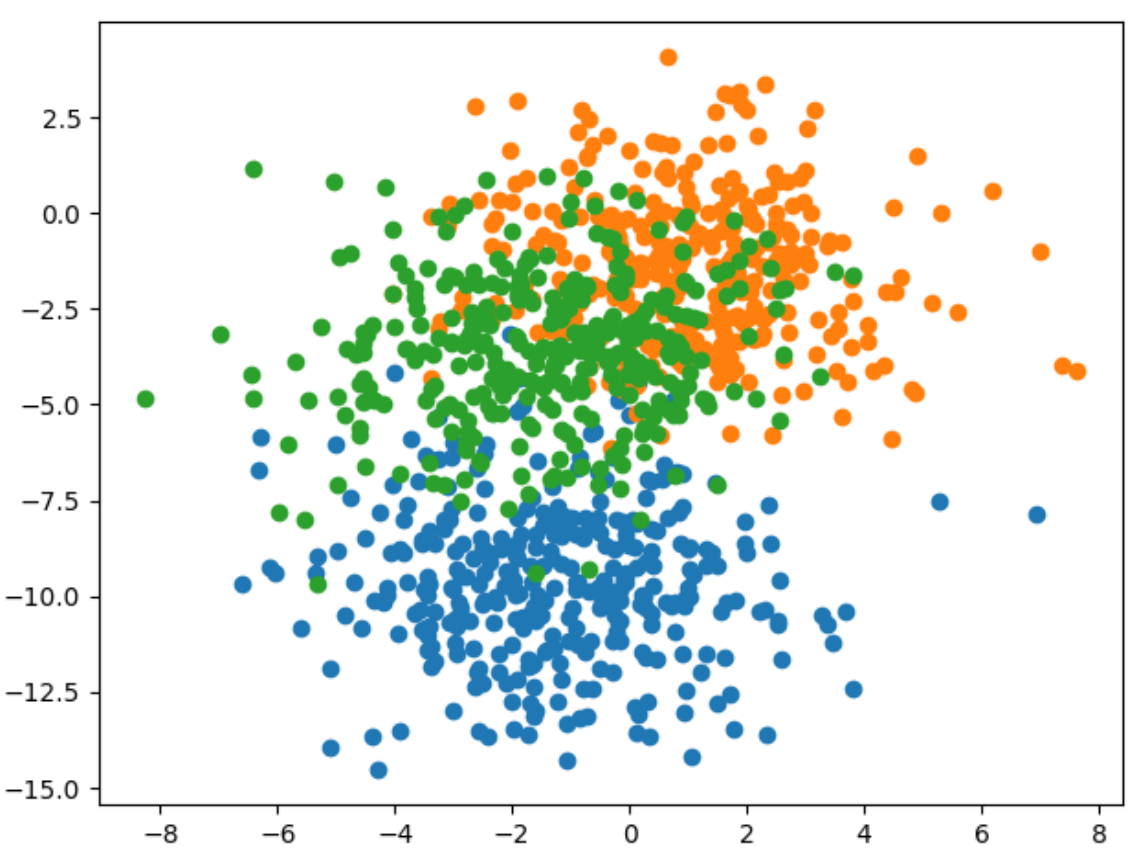
Greedy layer-wise pretraining involves successively adding a new hidden layer to a model and refitting, allowing the newly added model to learn the inputs from the existing hidden layer, often while keeping the weights for the existing hidden layers. Sometimes, but not always, being able to build layers 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 (See Chapter 20 of Better Deep Learning)

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 keras.optimizers import SGD  from 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 keras.optimizers import SGD  from 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:

|  |
| --- |
| # layers train\_acc test\_acc train\_loss test\_loss  0 -1.0 0.813230 0.673228 0.460823 1.275653  1 0.0 0.838521 0.712598 0.334927 1.142595  2 1.0 0.852140 0.728346 0.349439 1.315949  3 2.0 0.867704 0.720472 0.307426 1.065675  4 3.0 0.865759 0.736220 0.321653 0.940315  5 4.0 0.863813 0.688976 0.321372 1.031350  6 5.0 0.846304 0.716535 0.326061 0.956837  **7 6.0 0.852140 0.751969 0.321356 0.738948**  8 7.0 0.793774 0.665354 0.490491 0.861082  9 8.0 0.867704 0.732283 0.303628 0.773556  10 9.0 0.844358 0.688976 0.354361 0.779690  11 10.0 0.852140 0.720472 0.346199 0.749457  12 11.0 0.852140 0.708661 0.325519 0.793386  13 12.0 0.822957 0.700787 0.376356 0.871078  14 13.0 0.799611 0.669291 0.369450 0.908168  6 layers yields the highest test accuracy. |

Exercise (1 mark)

Which line of code in Exercise 3 saves the output layer for use later?

|  |
| --- |
| Line 58:  output\_layer = model.layers[-1] |

Exercise (1 mark)

Which line of code in Exercise 3 adds the output layer that is stored in memory back onto the network.

|  |
| --- |
| Line 71:  model.add(output\_layer) |

Exercise (1 mark)

Which line of code in Exercise 3 removes the output layer so a new layer can be added?

|  |
| --- |
| Line 61:  model.pop() |

Exercise (1 mark)

Which lines of code in Exercise 3 ensures that the previously added hidden layers cannot be adjusted?

|  |
| --- |
| Line 64/65:  for layer in model.layers:  layer.trainable = False |

Exercise (1 mark)

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

|  |
| --- |
| Greedy layer-wise pretraining lets individual layers build/fit without the previous layers affecting it by storing previous output layers then adding it to the list of output layers. |