Open in Colab

T81-558: Applications of Deep Neural Networks

Module 9: Transfer Learning

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 9 Material

- **Part 9.1: Introduction to Keras Transfer Learning** [Video] [Notebook]
- Part 9.2: Keras Transfer Learning for Computer Vision [Video] [Notebook]
- Part 9.3: Transfer Learning for NLP with Keras [Video] [Notebook]
- Part 9.4: Transfer Learning for Facial Feature Recognition [Video] [Notebook]
- Part 9.5: Transfer Learning for Style Transfer [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [ ]:
```
# Start CoLab
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 9.1: Introduction to Keras Transfer Learning

Human beings learn new skills throughout their entire lives. However, this learning is rarely from scratch. No matter what task a human learns, they are most likely drawing

on experiences to learn this new skill early in life. In this way, humans learn much differently than most deep learning projects.

A human being learns to tell the difference between a cat and a dog at some point. To teach a neural network, you would obtain many cat pictures and dog pictures. The neural network would iterate over all of these pictures and train on the differences. The human child that learned to distinguish between the two animals would probably need to see a few examples when parents told them the name of each type of animal. The human child would use previous knowledge of looking at different living and non-living objects to help make this classification. The child would already know the physical appearance of sub-objects, such as fur, eyes, ears, noses, tails, and teeth.

Transfer learning attempts to teach a neural network by similar means. Rather than training your neural network from scratch, you begin training with a preloaded set of weights. Usually, you will remove the topmost layers of the pretrained neural network and retrain it with new uppermost layers. The layers from the previous neural network will be locked so that training does not change these weights. Only the newly added layers will be trained.

It can take much computing power to train a neural network for a large image dataset. Google, Facebook, Microsoft, and other tech companies have utilized GPU arrays for training high-quality neural networks for various applications. Transferring these weights into your neural network can save considerable effort and compute time. It is unlikely that a pretrained model will exactly fit the application that you seek to implement. Finding the closest pretrained model and using transfer learning is essential for a deep learning engineer.

Transfer Learning Example

Let's look at a simple example of using transfer learning to build upon an imagenet neural network. We will begin by training a neural network for Fisher's Iris Dataset. This network takes four measurements and classifies each observation into three iris species. However, what if later we received a data set that included the four measurements, plus a cost as the target? This dataset does not contain the species; as a result, it uses the same four inputs as the base model we just trained.

We can take our previously trained iris network and transfer the weights to a new neural network that will learn to predict the cost through transfer learning. Also of note, the original neural network was a classification network, yet we now use it to build a regression neural network. Such a transformation is common for transfer learning. As a reference point, I randomly created this iris cost dataset.

The first step is to train our neural network for the regular Iris Dataset. The code presented here is the same as we saw in Module 3.

In [ ]:
```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values


# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output

model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

```
Epoch 1/100
5/5 - 1s - loss: 1.7638 - 634ms/epoch - 127ms/step
Epoch 2/100
5/5 - 0s - loss: 1.2951 - 9ms/epoch - 2ms/step
Epoch 3/100
5/5 - 0s - loss: 1.0713 - 8ms/epoch - 2ms/step
Epoch 4/100
5/5 - 0s - loss: 1.0110 - 12ms/epoch - 2ms/step
Epoch 5/100
5/5 - 0s - loss: 0.9364 - 9ms/epoch - 2ms/step
Epoch 6/100
5/5 - 0s - loss: 0.8444 - 8ms/epoch - 2ms/step
Epoch 7/100
5/5 - 0s - loss: 0.7800 - 12ms/epoch - 2ms/step
Epoch 8/100
5/5 - 0s - loss: 0.7321 - 13ms/epoch - 3ms/step
Epoch 9/100
5/5 - 0s - loss: 0.6806 - 13ms/epoch - 3ms/step
Epoch 10/100
5/5 - 0s - loss: 0.6377 - 12ms/epoch - 2ms/step
Epoch 11/100
5/5 - 0s - loss: 0.6021 - 13ms/epoch - 3ms/step
Epoch 12/100
5/5 - 0s - loss: 0.5693 - 10ms/epoch - 2ms/step
Epoch 13/100
5/5 - 0s - loss: 0.5470 - 11ms/epoch - 2ms/step
Epoch 14/100
5/5 - 0s - loss: 0.5219 - 11ms/epoch - 2ms/step
Epoch 15/100
5/5 - 0s - loss: 0.4992 - 24ms/epoch - 5ms/step
Epoch 16/100
5/5 - 0s - loss: 0.4757 - 15ms/epoch - 3ms/step
Epoch 17/100
5/5 - 0s - loss: 0.4576 - 11ms/epoch - 2ms/step
Epoch 18/100
5/5 - 0s - loss: 0.4422 - 11ms/epoch - 2ms/step
Epoch 19/100
5/5 - 0s - loss: 0.4267 - 13ms/epoch - 3ms/step
Epoch 20/100
5/5 - 0s - loss: 0.4119 - 17ms/epoch - 3ms/step
Epoch 21/100
5/5 - 0s - loss: 0.3994 - 12ms/epoch - 2ms/step
Epoch 22/100
5/5 - 0s - loss: 0.3877 - 16ms/epoch - 3ms/step
Epoch 23/100
5/5 - 0s - loss: 0.3783 - 16ms/epoch - 3ms/step
Epoch 24/100
5/5 - 0s - loss: 0.3640 - 16ms/epoch - 3ms/step
Epoch 25/100
5/5 - 0s - loss: 0.3537 - 16ms/epoch - 3ms/step
Epoch 26/100
5/5 - 0s - loss: 0.3443 - 9ms/epoch - 2ms/step
Epoch 27/100
5/5 - 0s - loss: 0.3375 - 11ms/epoch - 2ms/step
Epoch 28/100
5/5 - 0s - loss: 0.3245 - 14ms/epoch - 3ms/step
```

```
Epoch 29/100
5/5 - 0s - loss: 0.3152 - 16ms/epoch - 3ms/step
Epoch 30/100
5/5 - 0s - loss: 0.3050 - 15ms/epoch - 3ms/step
Epoch 31/100
5/5 - 0s - loss: 0.2934 - 16ms/epoch - 3ms/step
Epoch 32/100
5/5 - 0s - loss: 0.2830 - 14ms/epoch - 3ms/step
Epoch 33/100
5/5 - 0s - loss: 0.2738 - 13ms/epoch - 3ms/step
Epoch 34/100
5/5 - 0s - loss: 0.2651 - 17ms/epoch - 3ms/step
Epoch 35/100
5/5 - 0s - loss: 0.2609 - 19ms/epoch - 4ms/step
Epoch 36/100
5/5 - 0s - loss: 0.2499 - 13ms/epoch - 3ms/step
Epoch 37/100
5/5 - 0s - loss: 0.2436 - 12ms/epoch - 2ms/step
Epoch 38/100
5/5 - 0s - loss: 0.2353 - 13ms/epoch - 3ms/step
Epoch 39/100
5/5 - 0s - loss: 0.2291 - 12ms/epoch - 2ms/step
Epoch 40/100
5/5 - 0s - loss: 0.2226 - 14ms/epoch - 3ms/step
Epoch 41/100
5/5 - 0s - loss: 0.2171 - 16ms/epoch - 3ms/step
Epoch 42/100
5/5 - 0s - loss: 0.2140 - 16ms/epoch - 3ms/step
Epoch 43/100
5/5 - 0s - loss: 0.2037 - 17ms/epoch - 3ms/step
Epoch 44/100
5/5 - 0s - loss: 0.2039 - 17ms/epoch - 3ms/step
Epoch 45/100
5/5 - 0s - loss: 0.1982 - 14ms/epoch - 3ms/step
Epoch 46/100
5/5 - 0s - loss: 0.1897 - 9ms/epoch - 2ms/step
Epoch 47/100
5/5 - 0s - loss: 0.1862 - 18ms/epoch - 4ms/step
Epoch 48/100
5/5 - 0s - loss: 0.1812 - 13ms/epoch - 3ms/step
Epoch 49/100
5/5 - 0s - loss: 0.1798 - 15ms/epoch - 3ms/step
Epoch 50/100
5/5 - 0s - loss: 0.1726 - 19ms/epoch - 4ms/step
Epoch 51/100
5/5 - 0s - loss: 0.1685 - 14ms/epoch - 3ms/step
Epoch 52/100
5/5 - 0s - loss: 0.1674 - 15ms/epoch - 3ms/step
Epoch 53/100
5/5 - 0s - loss: 0.1651 - 16ms/epoch - 3ms/step
Epoch 54/100
5/5 - 0s - loss: 0.1576 - 10ms/epoch - 2ms/step
Epoch 55/100
5/5 - 0s - loss: 0.1550 - 14ms/epoch - 3ms/step
Epoch 56/100
5/5 - 0s - loss: 0.1515 - 16ms/epoch - 3ms/step
```

```
Epoch 57/100
5/5 - 0s - loss: 0.1526 - 16ms/epoch - 3ms/step
Epoch 58/100
5/5 - 0s - loss: 0.1487 - 14ms/epoch - 3ms/step
Epoch 59/100
5/5 - 0s - loss: 0.1458 - 17ms/epoch - 3ms/step
Epoch 60/100
5/5 - 0s - loss: 0.1411 - 14ms/epoch - 3ms/step
Epoch 61/100
5/5 - 0s - loss: 0.1401 - 9ms/epoch - 2ms/step
Epoch 62/100
5/5 - 0s - loss: 0.1355 - 14ms/epoch - 3ms/step
Epoch 63/100
5/5 - 0s - loss: 0.1344 - 18ms/epoch - 4ms/step
Epoch 64/100
5/5 - 0s - loss: 0.1321 - 15ms/epoch - 3ms/step
Epoch 65/100
5/5 - 0s - loss: 0.1295 - 12ms/epoch - 2ms/step
Epoch 66/100
5/5 - 0s - loss: 0.1278 - 12ms/epoch - 2ms/step
Epoch 67/100
5/5 - 0s - loss: 0.1261 - 10ms/epoch - 2ms/step
Epoch 68/100
5/5 - 0s - loss: 0.1255 - 15ms/epoch - 3ms/step
Epoch 69/100
5/5 - 0s - loss: 0.1237 - 9ms/epoch - 2ms/step
Epoch 70/100
5/5 - 0s - loss: 0.1210 - 16ms/epoch - 3ms/step
Epoch 71/100
5/5 - 0s - loss: 0.1184 - 15ms/epoch - 3ms/step
Epoch 72/100
5/5 - 0s - loss: 0.1168 - 15ms/epoch - 3ms/step
Epoch 73/100
5/5 - 0s - loss: 0.1146 - 12ms/epoch - 2ms/step
Epoch 74/100
5/5 - 0s - loss: 0.1169 - 15ms/epoch - 3ms/step
Epoch 75/100
5/5 - 0s - loss: 0.1123 - 10ms/epoch - 2ms/step
Epoch 76/100
5/5 - 0s - loss: 0.1109 - 15ms/epoch - 3ms/step
Epoch 77/100
5/5 - 0s - loss: 0.1089 - 15ms/epoch - 3ms/step
Epoch 78/100
5/5 - 0s - loss: 0.1079 - 15ms/epoch - 3ms/step
Epoch 79/100
5/5 - 0s - loss: 0.1074 - 12ms/epoch - 2ms/step
Epoch 80/100
5/5 - 0s - loss: 0.1060 - 10ms/epoch - 2ms/step
Epoch 81/100
5/5 - 0s - loss: 0.1054 - 10ms/epoch - 2ms/step
Epoch 82/100
5/5 - 0s - loss: 0.1030 - 18ms/epoch - 4ms/step
Epoch 83/100
5/5 - 0s - loss: 0.1037 - 15ms/epoch - 3ms/step
Epoch 84/100
5/5 - 0s - loss: 0.1014 - 15ms/epoch - 3ms/step
```

```
Epoch 85/100
5/5 - 0s - loss: 0.0989 - 17ms/epoch - 3ms/step
Epoch 86/100
5/5 - 0s - loss: 0.0989 - 9ms/epoch - 2ms/step
Epoch 87/100
5/5 - 0s - loss: 0.1000 - 13ms/epoch - 3ms/step
Epoch 88/100
5/5 - 0s - loss: 0.0953 - 8ms/epoch - 2ms/step
Epoch 89/100
5/5 - 0s - loss: 0.0994 - 11ms/epoch - 2ms/step
Epoch 90/100
5/5 - 0s - loss: 0.0941 - 12ms/epoch - 2ms/step
Epoch 91/100
5/5 - 0s - loss: 0.0947 - 15ms/epoch - 3ms/step
Epoch 92/100
5/5 - 0s - loss: 0.0963 - 18ms/epoch - 4ms/step
Epoch 93/100
5/5 - 0s - loss: 0.0913 - 14ms/epoch - 3ms/step
Epoch 94/100
5/5 - 0s - loss: 0.0922 - 20ms/epoch - 4ms/step
Epoch 95/100
5/5 - 0s - loss: 0.0916 - 13ms/epoch - 3ms/step
Epoch 96/100
5/5 - 0s - loss: 0.0905 - 12ms/epoch - 2ms/step
Epoch 97/100
5/5 - 0s - loss: 0.0886 - 9ms/epoch - 2ms/step
Epoch 98/100
5/5 - 0s - loss: 0.0900 - 15ms/epoch - 3ms/step
Epoch 99/100
5/5 - 0s - loss: 0.0868 - 15ms/epoch - 3ms/step
Epoch 100/100
5/5 - 0s - loss: 0.0892 - 8ms/epoch - 2ms/step
```

Out[ ]: <keras.callbacks.History at 0x7fea3fb1ef50>

To keep this example simple, we are not setting aside a validation set. The goal of this example is to show how to create a multi-layer neural network, where we transfer the weights to another network. We begin by evaluating the accuracy of the network on the training set.

In [ ]:
```
from sklearn.metrics import accuracy_score
pred = model.predict(x)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Training Accuracy: {correct}")
```

```
Training Accuracy: 0.9866666666666667
```

Viewing the model summary is as expected; we can see the three layers previously defined.

In [ ]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 50)                250

 dense_1 (Dense)             (None, 25)                1275

 dense_2 (Dense)             (None, 3)                 78

=================================================================
Total params: 1,603
Trainable params: 1,603
Non-trainable params: 0
_________________________________________________________________
```

Create a New Iris Network

Now that we've trained a neural network on the iris dataset, we can transfer the knowledge of this neural network to other neural networks. It is possible to create a new neural network from some or all of the layers of this neural network. We will create a new neural network that is essentially a clone of the first neural network to demonstrate the technique. We now transfer all of the layers from the original neural network into the new one.

In [ ]:
```
model2 = Sequential()
for layer in model.layers:
    model2.add(layer)
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 50)                250

 dense_1 (Dense)             (None, 25)                1275

 dense_2 (Dense)             (None, 3)                 78

=================================================================
Total params: 1,603
Trainable params: 1,603
Non-trainable params: 0
_________________________________________________________________
```

As a sanity check, we would like to calculate the accuracy of the newly created model. The in-sample accuracy should be the same as the previous model that the new model transferred.

In [ ]:
```
from sklearn.metrics import accuracy_score
pred = model2.predict(x)
```

```
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Training Accuracy: {correct}")
```

```
Training Accuracy: 0.9866666666666667
```

The in-sample accuracy of the newly created neural network is the same as the first neural network. We've successfully transferred all of the layers from the original neural network.

Transfering to a Regression Network

The Iris Cost Dataset has measurements for samples of these flowers that conform to the predictors contained in the original iris dataset: sepal width, sepal length, petal width, and petal length. We present the cost dataset here.

In [ ]:
```
df_cost = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris_cost.csv",
    na_values=['NA', '?'])

df_cost
```

Out[ ]:

	sepal_l	sepal_w	petal_l	petal_w	cost
0	7.8	3.0	6.2	2.0	10.740
1	5.0	2.2	1.7	1.5	2.710
2	6.9	2.6	3.7	1.4	4.624
3	5.9	2.2	3.7	2.4	6.558
4	5.1	3.9	6.8	0.7	7.395
...	...	...	...	...	...
245	4.7	2.1	4.0	2.3	5.721
246	7.2	3.0	4.3	1.1	5.266
247	6.6	3.4	4.6	1.4	5.776
248	5.7	3.7	3.1	0.4	2.233
249	7.6	4.0	5.1	1.4	7.508

250 rows × 5 columns

For transfer learning to be effective, the input for the newly trained neural network most closely conforms to the first neural network we transfer.

We will strip away the last output layer that contains the softmax activation function that performs this final classification. We will create a new output layer that will output the cost prediction. We will only train the weights in this new layer. We will mark the first two layers as non-trainable. The hope is that the first few layers have learned to abstract the raw input data in a way that is also helpful to the new neural network. This process is accomplished by looping over the first few layers and copying them to the new neural network. We output a summary of the new neural network to verify that Keras stripped the previous output layer.

In [ ]:
```
model3 = Sequential()
for i in range(2):
    layer = model.layers[i]
    layer.trainable = False
    model3.add(layer)
model3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 50)                250

 dense_1 (Dense)              (None, 25)                1275

=================================================================
Total params: 1,525
Trainable params: 0
Non-trainable params: 1,525
_________________________________________________________________
```

We add a final regression output layer to complete the new neural network.

In [ ]:
```
model3.add(Dense(1)) # Output

model3.compile(loss='mean_squared_error', optimizer='adam')
model3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 50)                250

 dense_1 (Dense)              (None, 25)                1275

 dense_3 (Dense)              (None, 1)                 26

=================================================================
Total params: 1,551
Trainable params: 26
Non-trainable params: 1,525
_________________________________________________________________
```

Now we train just the output layer to predict the cost. The cost in the made-up dataset is dependent on the species, so the previous learning should be helpful.

In [ ]:
```
# Convert to numpy - Classification
x = df_cost[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
y = df_cost.cost.values

# Train the last layer of the network
model3.fit(x,y,verbose=2,epochs=100)
```

```
Epoch 1/100
8/8 - 0s - loss: 14.0400 - 379ms/epoch - 47ms/step
Epoch 2/100
8/8 - 0s - loss: 12.6133 - 10ms/epoch - 1ms/step
Epoch 3/100
8/8 - 0s - loss: 11.3224 - 12ms/epoch - 1ms/step
Epoch 4/100
8/8 - 0s - loss: 10.1006 - 19ms/epoch - 2ms/step
Epoch 5/100
8/8 - 0s - loss: 9.0898 - 19ms/epoch - 2ms/step
Epoch 6/100
8/8 - 0s - loss: 8.1514 - 13ms/epoch - 2ms/step
Epoch 7/100
8/8 - 0s - loss: 7.3497 - 11ms/epoch - 1ms/step
Epoch 8/100
8/8 - 0s - loss: 6.6789 - 14ms/epoch - 2ms/step
Epoch 9/100
8/8 - 0s - loss: 6.0785 - 11ms/epoch - 1ms/step
Epoch 10/100
8/8 - 0s - loss: 5.5620 - 11ms/epoch - 1ms/step
Epoch 11/100
8/8 - 0s - loss: 5.1035 - 11ms/epoch - 1ms/step
Epoch 12/100
8/8 - 0s - loss: 4.7415 - 12ms/epoch - 2ms/step
Epoch 13/100
8/8 - 0s - loss: 4.4169 - 13ms/epoch - 2ms/step
Epoch 14/100
8/8 - 0s - loss: 4.1181 - 19ms/epoch - 2ms/step
Epoch 15/100
8/8 - 0s - loss: 3.8847 - 20ms/epoch - 3ms/step
Epoch 16/100
8/8 - 0s - loss: 3.6586 - 13ms/epoch - 2ms/step
Epoch 17/100
8/8 - 0s - loss: 3.4690 - 17ms/epoch - 2ms/step
Epoch 18/100
8/8 - 0s - loss: 3.3085 - 16ms/epoch - 2ms/step
Epoch 19/100
8/8 - 0s - loss: 3.1461 - 17ms/epoch - 2ms/step
Epoch 20/100
8/8 - 0s - loss: 3.0206 - 20ms/epoch - 2ms/step
Epoch 21/100
8/8 - 0s - loss: 2.8936 - 14ms/epoch - 2ms/step
Epoch 22/100
8/8 - 0s - loss: 2.7855 - 11ms/epoch - 1ms/step
Epoch 23/100
8/8 - 0s - loss: 2.6949 - 10ms/epoch - 1ms/step
Epoch 24/100
8/8 - 0s - loss: 2.6056 - 13ms/epoch - 2ms/step
Epoch 25/100
8/8 - 0s - loss: 2.5355 - 24ms/epoch - 3ms/step
Epoch 26/100
8/8 - 0s - loss: 2.4600 - 10ms/epoch - 1ms/step
Epoch 27/100
8/8 - 0s - loss: 2.4041 - 13ms/epoch - 2ms/step
Epoch 28/100
8/8 - 0s - loss: 2.3449 - 14ms/epoch - 2ms/step
```

```
Epoch 29/100
8/8 - 0s - loss: 2.2991 - 17ms/epoch - 2ms/step
Epoch 30/100
8/8 - 0s - loss: 2.2528 - 15ms/epoch - 2ms/step
Epoch 31/100
8/8 - 0s - loss: 2.2143 - 15ms/epoch - 2ms/step
Epoch 32/100
8/8 - 0s - loss: 2.1818 - 17ms/epoch - 2ms/step
Epoch 33/100
8/8 - 0s - loss: 2.1527 - 15ms/epoch - 2ms/step
Epoch 34/100
8/8 - 0s - loss: 2.1262 - 17ms/epoch - 2ms/step
Epoch 35/100
8/8 - 0s - loss: 2.1046 - 13ms/epoch - 2ms/step
Epoch 36/100
8/8 - 0s - loss: 2.0811 - 13ms/epoch - 2ms/step
Epoch 37/100
8/8 - 0s - loss: 2.0657 - 10ms/epoch - 1ms/step
Epoch 38/100
8/8 - 0s - loss: 2.0487 - 15ms/epoch - 2ms/step
Epoch 39/100
8/8 - 0s - loss: 2.0368 - 17ms/epoch - 2ms/step
Epoch 40/100
8/8 - 0s - loss: 2.0257 - 21ms/epoch - 3ms/step
Epoch 41/100
8/8 - 0s - loss: 2.0131 - 15ms/epoch - 2ms/step
Epoch 42/100
8/8 - 0s - loss: 2.0053 - 24ms/epoch - 3ms/step
Epoch 43/100
8/8 - 0s - loss: 1.9972 - 22ms/epoch - 3ms/step
Epoch 44/100
8/8 - 0s - loss: 1.9898 - 17ms/epoch - 2ms/step
Epoch 45/100
8/8 - 0s - loss: 1.9838 - 14ms/epoch - 2ms/step
Epoch 46/100
8/8 - 0s - loss: 1.9788 - 12ms/epoch - 1ms/step
Epoch 47/100
8/8 - 0s - loss: 1.9743 - 14ms/epoch - 2ms/step
Epoch 48/100
8/8 - 0s - loss: 1.9702 - 19ms/epoch - 2ms/step
Epoch 49/100
8/8 - 0s - loss: 1.9662 - 15ms/epoch - 2ms/step
Epoch 50/100
8/8 - 0s - loss: 1.9646 - 14ms/epoch - 2ms/step
Epoch 51/100
8/8 - 0s - loss: 1.9604 - 15ms/epoch - 2ms/step
Epoch 52/100
8/8 - 0s - loss: 1.9578 - 16ms/epoch - 2ms/step
Epoch 53/100
8/8 - 0s - loss: 1.9552 - 14ms/epoch - 2ms/step
Epoch 54/100
8/8 - 0s - loss: 1.9528 - 11ms/epoch - 1ms/step
Epoch 55/100
8/8 - 0s - loss: 1.9511 - 19ms/epoch - 2ms/step
Epoch 56/100
8/8 - 0s - loss: 1.9489 - 18ms/epoch - 2ms/step
```

```
Epoch 57/100
8/8 - 0s - loss: 1.9468 - 17ms/epoch - 2ms/step
Epoch 58/100
8/8 - 0s - loss: 1.9462 - 19ms/epoch - 2ms/step
Epoch 59/100
8/8 - 0s - loss: 1.9435 - 14ms/epoch - 2ms/step
Epoch 60/100
8/8 - 0s - loss: 1.9421 - 11ms/epoch - 1ms/step
Epoch 61/100
8/8 - 0s - loss: 1.9406 - 17ms/epoch - 2ms/step
Epoch 62/100
8/8 - 0s - loss: 1.9389 - 14ms/epoch - 2ms/step
Epoch 63/100
8/8 - 0s - loss: 1.9387 - 15ms/epoch - 2ms/step
Epoch 64/100
8/8 - 0s - loss: 1.9358 - 19ms/epoch - 2ms/step
Epoch 65/100
8/8 - 0s - loss: 1.9347 - 17ms/epoch - 2ms/step
Epoch 66/100
8/8 - 0s - loss: 1.9332 - 20ms/epoch - 3ms/step
Epoch 67/100
8/8 - 0s - loss: 1.9329 - 16ms/epoch - 2ms/step
Epoch 68/100
8/8 - 0s - loss: 1.9300 - 17ms/epoch - 2ms/step
Epoch 69/100
8/8 - 0s - loss: 1.9289 - 22ms/epoch - 3ms/step
Epoch 70/100
8/8 - 0s - loss: 1.9286 - 17ms/epoch - 2ms/step
Epoch 71/100
8/8 - 0s - loss: 1.9259 - 13ms/epoch - 2ms/step
Epoch 72/100
8/8 - 0s - loss: 1.9250 - 16ms/epoch - 2ms/step
Epoch 73/100
8/8 - 0s - loss: 1.9231 - 13ms/epoch - 2ms/step
Epoch 74/100
8/8 - 0s - loss: 1.9217 - 14ms/epoch - 2ms/step
Epoch 75/100
8/8 - 0s - loss: 1.9199 - 17ms/epoch - 2ms/step
Epoch 76/100
8/8 - 0s - loss: 1.9189 - 21ms/epoch - 3ms/step
Epoch 77/100
8/8 - 0s - loss: 1.9176 - 15ms/epoch - 2ms/step
Epoch 78/100
8/8 - 0s - loss: 1.9163 - 16ms/epoch - 2ms/step
Epoch 79/100
8/8 - 0s - loss: 1.9146 - 14ms/epoch - 2ms/step
Epoch 80/100
8/8 - 0s - loss: 1.9132 - 15ms/epoch - 2ms/step
Epoch 81/100
8/8 - 0s - loss: 1.9118 - 14ms/epoch - 2ms/step
Epoch 82/100
8/8 - 0s - loss: 1.9100 - 15ms/epoch - 2ms/step
Epoch 83/100
8/8 - 0s - loss: 1.9083 - 15ms/epoch - 2ms/step
Epoch 84/100
8/8 - 0s - loss: 1.9070 - 25ms/epoch - 3ms/step
```

```
Epoch 85/100
8/8 - 0s - loss: 1.9074 - 12ms/epoch - 2ms/step
Epoch 86/100
8/8 - 0s - loss: 1.9039 - 18ms/epoch - 2ms/step
Epoch 87/100
8/8 - 0s - loss: 1.9027 - 13ms/epoch - 2ms/step
Epoch 88/100
8/8 - 0s - loss: 1.9010 - 15ms/epoch - 2ms/step
Epoch 89/100
8/8 - 0s - loss: 1.8994 - 16ms/epoch - 2ms/step
Epoch 90/100
8/8 - 0s - loss: 1.8978 - 15ms/epoch - 2ms/step
Epoch 91/100
8/8 - 0s - loss: 1.8968 - 14ms/epoch - 2ms/step
Epoch 92/100
8/8 - 0s - loss: 1.8952 - 15ms/epoch - 2ms/step
Epoch 93/100
8/8 - 0s - loss: 1.8956 - 20ms/epoch - 2ms/step
Epoch 94/100
8/8 - 0s - loss: 1.8925 - 16ms/epoch - 2ms/step
Epoch 95/100
8/8 - 0s - loss: 1.8906 - 16ms/epoch - 2ms/step
Epoch 96/100
8/8 - 0s - loss: 1.8891 - 13ms/epoch - 2ms/step
Epoch 97/100
8/8 - 0s - loss: 1.8877 - 11ms/epoch - 1ms/step
Epoch 98/100
8/8 - 0s - loss: 1.8860 - 14ms/epoch - 2ms/step
Epoch 99/100
8/8 - 0s - loss: 1.8851 - 17ms/epoch - 2ms/step
Epoch 100/100
8/8 - 0s - loss: 1.8838 - 9ms/epoch - 1ms/step
```

Out[ ]: <keras.callbacks.History at 0x7fea3f9bc890>

We can evaluate the in-sample RMSE for the new model containing transferred layers from the previous model.

In [ ]:
```
from sklearn.metrics import accuracy_score
pred = model3.predict(x)
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Final score (RMSE): {score}")
```

```
Final score (RMSE): 1.3716589625823072
```

Module 9 Assignment

You can find the first assignment here: assignment 9

In [ ]:
```

```