Technological Institute of the Philippines	Quezon City - Computer Engineering
Course Code:	CPE 019
Code Title:	Emerging Technologies in CpE 2 - Fundamentals of Data Science
1st Semester	AY 2023-2024

**ACTIVITY NO 8.1 **	**Saving Models**
Name	William Laurence M. Ramos
Section	CPE32S1
Date Performed:	08/07/2024
Date Submitted:	08/07/2024
Instructor:	Engr. Roman M. Richard

```
import numpy as np
import pandas as pd
```

```
# Choose any dataset applicable to either a classification problem or a regression problem.
# Explain your datasets and the problem being addressed.
appleDF = pd.read_csv("/content/drive/MyDrive/CPE019/apple_quality.csv") #Classification type dataset
```

The dataset used is called the Apple Quality dataset. It focuses on binary classification, where the target variable Quality is categorized as Good or Bad. The goal is to build a model using the dataset's input variables for binary classification. It's important to save the model for future use and create checkpoints during training to ensure it completes successfully.

```
# Epoch verbose call to reduce visual clutter
# Reference: https://stackoverflow.com/questions/72660874/how-to-print-one-log-line-per-every-10-epochs-when-training-models-with-tensorfl
import tensorflow as tf
class epochs_Callback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
      if ((int(epoch) % 10) == 0 or (int(epoch) == t_epoch-1)):
          print(
              f"Epoch: {epoch:>3}"
              #+ f" | Loss: {logs['loss']}"
              + f" | Accuracy: {logs['accuracy']}"
              #+ f" | Validation loss: {logs['val_loss']}"
              #+ f" | Validation accuracy: {logs['val_accuracy']}"
          )
my_callbacks = [epochs_Callback()]
```

```
from scikeras.wrappers import KerasClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential, model_from_json
from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D
import matplotlib.pyplot as plt
```

```
columns = ["Size", "Weight", "Sweetness", "Crunchiness", "Juiciness", "Ripeness", "Acidity"]
apple_x = appleDF[list(columns)].values
apple_y = appleDF["Quality"].values

seed = 7
np.random.seed(seed)

# Pre-processing to convert categorical data to numerical data
encoder = LabelEncoder()
encoder.fit(apple_y)
apple_y = encoder.transform(apple_y)
```

```
# create model
t_epoch=150
model = Sequential()
model.add(Dense(60, input_shape=(len(columns),), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(apple_x, apple_y, batch_size=50, epochs=t_epoch, verbose = 0, callbacks = my_callbacks)
# evaluate the model
scores = model.evaluate(apple_x, apple_y, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argumen
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch:   0 | Accuracy: 0.6794999837875366
Epoch:  10 | Accuracy: 0.8830000162124634
Epoch:  20 | Accuracy: 0.902999997138977
Epoch:  30 | Accuracy: 0.921500027179718
Epoch:  40 | Accuracy: 0.9319999814033508
Epoch:  50 | Accuracy: 0.9427499771118164
Epoch:  60 | Accuracy: 0.9465000033378601
Epoch:  70 | Accuracy: 0.9514999985694885
Epoch:  80 | Accuracy: 0.9549999833106995
Epoch:  90 | Accuracy: 0.9580000042915344
Epoch: 100 | Accuracy: 0.9589999914169312
Epoch: 110 | Accuracy: 0.9645000100135803
Epoch: 120 | Accuracy: 0.968500018119812
Epoch: 130 | Accuracy: 0.9697499871253967
Epoch: 140 | Accuracy: 0.9714999794960022
Epoch: 149 | Accuracy: 0.9700000286102295
Accuracy: 97.72%
```

Our initial model serves as a standard against which all subsequent models are measured and our evaluation findings are derived. The basic model consists of a single binary output layer, one hidden layer with eight nodes, and a network of sixty nodes on the base layer. The accuracy after executing the baseline model is 97.72%.

1. Save a model in HDF5 format

```
# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("/content/drive/MyDrive/CPE019/A8.1/model.weights.h5")
print("Saved model to disk")

# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("/content/drive/MyDrive/CPE019/A8.1/model.weights.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
score = loaded_model.evaluate(apple_x, apple_y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

```
Saved model to disk
Loaded model from disk
compile_metrics: 97.72%
```

As we can see above, after loading the model's weights from the disc, we obtain the same 97.72% accuracy. This is because we just saved the model that we just created again before, thus loading our.weights.h5 file as the model's weights should give us the exact same evaluation results as our baseline.

2. Save a model and load the model in a JSON format

```
# serialize model to JSON
model_json = model.to_json()
with open("/content/drive/MyDrive/CPE019/A8.1/model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("/content/drive/MyDrive/CPE019/A8.1/model.weights.h5")
print("Saved model and weights to disk")

# load json and create model
json_file = open('/content/drive/MyDrive/CPE019/A8.1/model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("/content/drive/MyDrive/CPE019/A8.1/model.weights.h5")
print("Loaded model and weights from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
score = loaded_model.evaluate(apple_x, apple_y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

```
Saved model and weights to disk
Loaded model and weights from disk
compile_metrics: 97.72%
```

```
pd.read_json('/content/drive/MyDrive/CPE019/A8.1/model.json', lines=True)
```

	module	class_name	config	registered_name	build_config	compile_config
0	keras	Sequential	{'name': 'sequential',	NaN	{'input_shape':	{'optimizer': 'adam', 'loss':

Similar to before, I saved the model weights as .weights.h5 files and the entire model as a .json file on disk. After loading both the model and weights, I confirmed that the evaluation results remained consistent with the baseline and H5 formats, still at 97.72% accuracy.

3. Save a model and load the model in a YAML format

```
# serialize model to YAML
model_yaml = model.to_json()
with open("/content/drive/MyDrive/CPE019/A8.1/model.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)
# serialize weights to HDF5
model.save_weights("/content/drive/MyDrive/CPE019/A8.1/model_yaml.weights.h5")
print("Saved model and weights to disk")

# later...

# load YAML and create model
yaml_file = open('/content/drive/MyDrive/CPE019/A8.1/model.yaml', 'r')
loaded_model_yaml = yaml_file.read()
yaml_file.close()
loaded_model = model_from_json(loaded_model_yaml)
# load weights into new model
loaded_model.load_weights("/content/drive/MyDrive/CPE019/A8.1/model_yaml.weights.h5")
print("Loaded model and weights from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
score = loaded_model.evaluate(apple_x, apple_y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

```
Saved model and weights to disk
Loaded model and weights from disk
compile_metrics: 97.72%
```

```
import yaml
from yaml.loader import SafeLoader
with open('/content/drive/MyDrive/CPE019/A8.1/model.yaml') as f:
    data = yaml.load(f, Loader=SafeLoader)
    print(data)
```


```
{'module': 'keras', 'class_name': 'Sequential', 'config': {'name': 'sequential', 'trainable': True, 'dtype': 'float32', 'layers': [{'mod
```


And again just like the previous one, we will save our model into our disk as a YAML file format instead of a .json, it can easily be done just by changing the format type into our open() parameters from .json to .yaml. And as we can see from the evaulation results, it still the same from the previous ones and is still 97.72%

4. Checkpoint Neural Network Model Improvements

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(apple_x, apple_y, test_size=0.33, random_state=11111)
```

```
from keras.callbacks import ModelCheckpoint
t_epoch=150
model = Sequential()
model.add(Dense(60, input_shape=(len(columns),), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# checkpoint
filepath="weights-improvement-{epoch:02d}-{val_accuracy:.2f}.keras"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]

# Fit the model
model.fit(apple_x, apple_y, validation_data=(x_test, y_test), batch_size=10, epochs=t_epoch, verbose = 0, callbacks = callbacks_list)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1: val_accuracy improved from -inf to 0.84773, saving model to weights-improvement-01-0.85.keras

Epoch 2: val_accuracy improved from 0.84773 to 0.87045, saving model to weights-improvement-02-0.87.keras

Epoch 3: val_accuracy improved from 0.87045 to 0.88106, saving model to weights-improvement-03-0.88.keras

Epoch 4: val_accuracy improved from 0.88106 to 0.88409, saving model to weights-improvement-04-0.88.keras

Epoch 5: val_accuracy improved from 0.88409 to 0.89848, saving model to weights-improvement-05-0.90.keras

Epoch 6: val_accuracy improved from 0.89848 to 0.90606, saving model to weights-improvement-06-0.91.keras

Epoch 7: val_accuracy did not improve from 0.90606

Epoch 8: val_accuracy did not improve from 0.90606

Epoch 9: val_accuracy did not improve from 0.90606

Epoch 10: val_accuracy improved from 0.90606 to 0.91136, saving model to weights-improvement-10-0.91.keras

Epoch 11: val_accuracy improved from 0.91136 to 0.91439, saving model to weights-improvement-11-0.91.keras

Epoch 12: val_accuracy improved from 0.91439 to 0.92576, saving model to weights-improvement-12-0.93.keras

Epoch 13: val_accuracy improved from 0.92576 to 0.92803, saving model to weights-improvement-13-0.93.keras

Epoch 14: val_accuracy improved from 0.92803 to 0.93409, saving model to weights-improvement-14-0.93.keras

Epoch 15: val_accuracy did not improve from 0.93409

Epoch 16: val_accuracy did not improve from 0.93409

Epoch 17: val_accuracy improved from 0.93409 to 0.93939, saving model to weights-improvement-17-0.94.keras

Epoch 18: val_accuracy did not improve from 0.93939
```

```
Epoch 19: val_accuracy did not improve from 0.93939

Epoch 20: val_accuracy did not improve from 0.93939

Epoch 21: val_accuracy improved from 0.93939 to 0.94318, saving model to weights-improvement-21-0.94.keras

Epoch 22: val_accuracy improved from 0.94318 to 0.94697, saving model to weights-improvement-22-0.95.keras

Epoch 23: val_accuracy did not improve from 0.94697

Epoch 24: val_accuracy did not improve from 0.94697

Epoch 25: val_accuracy did not improve from 0.94697

Epoch 26: val_accuracy improved from 0.94697 to 0.94773, saving model to weights-improvement-26-0.95.keras

Epoch 27: val_accuracy improved from 0.94773 to 0.95000, saving model to weights-improvement-27-0.95.keras
```

Checkpoints serve the main purpose of saving the model's training progress as files and, more precisely, just the best ones. That is, it will only save it as a checkpoint during the epochs where advancements, such validation accuracy, have grown and reached the highest score. The only negative aspect of this is that there are numerous files because there are multiple checkpoints, which could cause some disc drive clutter.

5. Checkpoint Best Neural Network Model only

```
model = Sequential()
model.add(Dense(60, input_shape=(len(columns),), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# checkpoint
filepath="weights.best.keras"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]
# Fit the model
model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=10, epochs=t_epoch, verbose = 0, callbacks = callbacks_list)
```

```
Epoch 103: val_accuracy did not improve from 0.95303

Epoch 104: val_accuracy did not improve from 0.95303

Epoch 105: val_accuracy did not improve from 0.95303

Epoch 106: val_accuracy did not improve from 0.95303

Epoch 107: val_accuracy did not improve from 0.95303

Epoch 108: val_accuracy did not improve from 0.95303

Epoch 109: val_accuracy did not improve from 0.95303

Epoch 110: val_accuracy did not improve from 0.95303

Epoch 111: val_accuracy did not improve from 0.95303

Epoch 112: val_accuracy did not improve from 0.95303
```

This checkpoint technique differs from the previous one in that it saves multiple checkpoints as separate files, whereas the previous one saves multiple checkpoints as separate files. This means that the highest checkpoint is the one that is currently saved into that single file. Additionally, that file will be overwritten anytime a new checkpoint surpasses the old one in terms of progress, becoming the checkpoint that the model is now saving in that file.

6. Load a saved Neural Network model

```
model.load_weights("weights.best.keras")
# Compile model (required to make predictions)
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
print("Created model and loaded weights from file")

scores = model.evaluate(x_test, y_test, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
Created model and loaded weights from file
compile_metrics: 95.30%
```

By just loading the saved weights from the file we made from those checkpoints, we can easily load these checkpoints. And we can see that the final val_accuracy is 0.95303, or 95.303%, based on the last callback from the model's training. We obtain 95.30% after loading the model's weights and testing it, indicating that it is, in fact, the same model.

7. Visualize Model Training History in Keras

```
import tensorflow as tf
class epochs_Callback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
      if ((int(epoch) % 10) == 0 or (int(epoch) == t_epoch-1)):
          print(
              f"Epoch: {epoch:>3}"
              + f" | Loss: {logs['loss']}"
              + f" | Accuracy: {logs['accuracy']}"
              + f" | Validation loss: {logs['val_loss']}"
              + f" | Validation accuracy: {logs['val_accuracy']}"
          )
my_callbacks = [epochs_Callback()]
```

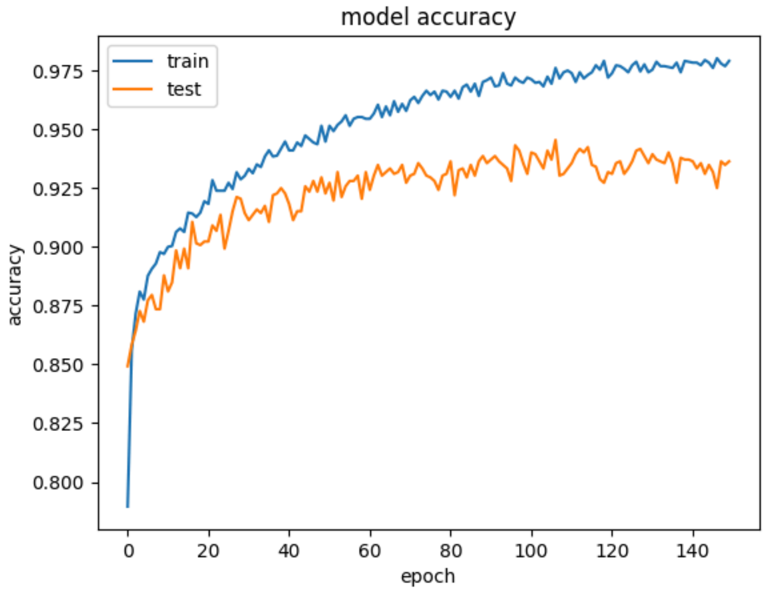
```
from keras.callbacks import ModelCheckpoint
t_epoch=150
model = Sequential()
model.add(Dense(60, input_shape=(len(columns),), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=10, epochs=t_epoch, verbose = 0, callbacks = my_callbacks
```

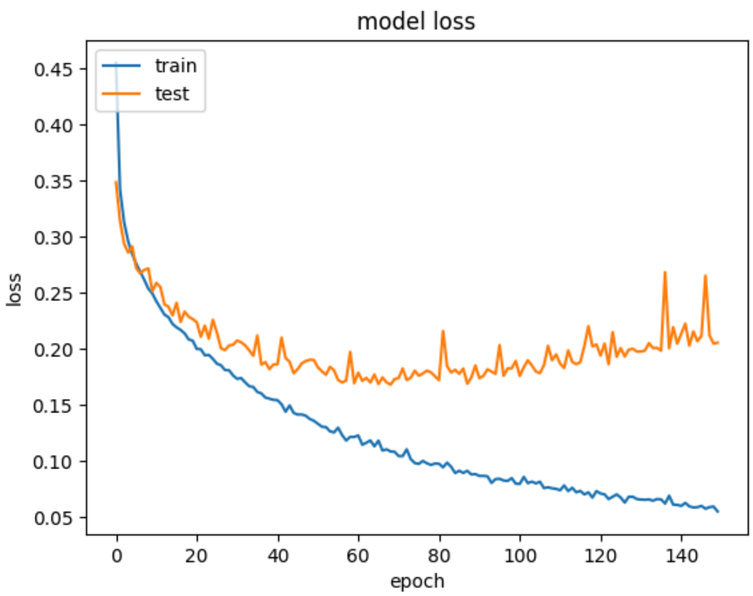
```
Epoch:   0 | Loss: 0.45513156056404114 | Accuracy: 0.7895522117614746 | Validation loss: 0.3486855626106262 | Validation accuracy: 0.849
Epoch:  10 | Loss: 0.24260397255420685 | Accuracy: 0.8999999761581421 | Validation loss: 0.2590392529964447 | Validation accuracy: 0.881
Epoch:  20 | Loss: 0.20021571218967438 | Accuracy: 0.9182835817337036 | Validation loss: 0.2240152359008789 | Validation accuracy: 0.902
```

```
Epoch:  30 | Loss: 0.1735236793756485 | Accuracy: 0.9332089424133301 | Validation loss: 0.2076452374458313 | Validation accuracy: 0.9113
Epoch:  40 | Loss: 0.1545400321483612 | Accuracy: 0.9410447478294373 | Validation loss: 0.18646346032619476 | Validation accuracy: 0.918
Epoch:  50 | Loss: 0.13348767161369324 | Accuracy: 0.9514925479888916 | Validation loss: 0.18382546305656433 | Validation accuracy: 0.92
Epoch:  60 | Loss: 0.12310835719108582 | Accuracy: 0.9544776082038879 | Validation loss: 0.17899169027805328 | Validation accuracy: 0.92
Epoch:  70 | Loss: 0.10474708676338196 | Accuracy: 0.9623134136199951 | Validation loss: 0.17418232560157776 | Validation accuracy: 0.93
Epoch:  80 | Loss: 0.09788330644369125 | Accuracy: 0.9638059735298157 | Validation loss: 0.17235006392002106 | Validation accuracy: 0.93
Epoch:  90 | Loss: 0.08704446256160736 | Accuracy: 0.9720149040222168 | Validation loss: 0.17413966357707977 | Validation accuracy: 0.93
Epoch: 100 | Loss: 0.07990701496601105 | Accuracy: 0.9712686538696289 | Validation loss: 0.17615431547164917 | Validation accuracy: 0.94
Epoch: 110 | Loss: 0.07424534857273102 | Accuracy: 0.9738805890083313 | Validation loss: 0.18715767562389374 | Validation accuracy: 0.93
Epoch: 120 | Loss: 0.07134021073579788 | Accuracy: 0.9738805890083313 | Validation loss: 0.19429874420166016 | Validation accuracy: 0.93
Epoch: 130 | Loss: 0.06600122898817062 | Accuracy: 0.9753731489181519 | Validation loss: 0.19782786071300507 | Validation accuracy: 0.93
Epoch: 140 | Loss: 0.060303766280412674 | Accuracy: 0.9783582091331482 | Validation loss: 0.21386979520320892 | Validation accuracy: 0.9
Epoch: 149 | Loss: 0.055384330451488495 | Accuracy: 0.9791044592857361 | Validation loss: 0.2057567536830902 | Validation accuracy: 0.93
```

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```





I plotted the model's training progress, focusing on accuracy and loss values over time. Initially, the model had high loss and low accuracy, but over time, the loss decreased and accuracy increased. However, around the 50-70% mark of training, the test loss began to slightly increase, indicating overfitting.

8. Show the application of Dropout Regularization

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import SGD
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from tensorflow.keras.constraints import MaxNorm


# baseline
def create_baseline():
  # create model
  model = Sequential()
  model.add(Dense(60, input_shape=(len(columns),), activation='relu'))
```

```
  model.add(Dense(30,  activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # Compile model
  sgd = SGD(learning_rate=0.01, momentum=0.8)
  model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
  return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_baseline, epochs=20, batch_size=16, verbose=1)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, apple_x, apple_y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
Epoch 1/20
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
225/225 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.6951 - loss: 0.5851
Epoch 2/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8206 - loss: 0.3881
Epoch 3/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8594 - loss: 0.3195
Epoch 4/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8713 - loss: 0.2874
Epoch 5/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8800 - loss: 0.2728
Epoch 6/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8951 - loss: 0.2448
Epoch 7/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8995 - loss: 0.2395
Epoch 8/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8970 - loss: 0.2369
Epoch 9/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8926 - loss: 0.2508
Epoch 10/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8995 - loss: 0.2318
Epoch 11/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9077 - loss: 0.2144
Epoch 12/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9062 - loss: 0.2281
Epoch 13/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9221 - loss: 0.2116
Epoch 14/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9215 - loss: 0.1930
Epoch 15/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9136 - loss: 0.2036
Epoch 16/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9169 - loss: 0.2102
Epoch 17/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9296 - loss: 0.1791
Epoch 18/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9257 - loss: 0.1825
Epoch 19/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9296 - loss: 0.1642
Epoch 20/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9305 - loss: 0.1767
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Epoch 1/20
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.6614 - loss: 0.6156
Epoch 2/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8174 - loss: 0.3924
Epoch 3/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8525 - loss: 0.3274
Epoch 4/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8661 - loss: 0.2952
Epoch 5/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8782 - loss: 0.2744
Epoch 6/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8780 - loss: 0.2757
```

We can see that we started by developing a baseline model featuring 60 nodes in the base layer, a hidden layer with 30 nodes, and a single binary output layer, without incorporating any dropouts. Using stratified KFold cross-validation for evaluation, the model underwent 10 folds with 50 epochs each. The results showed a baseline accuracy of 91.45% and a standard deviation of 1.58%.|

9. Show the application of Dropout on the visible layer

```
# dropout in the input layer with weight constraint
def create_model():
    # create model
    model = Sequential()
    model.add(Dropout(0.2, input_shape=(len(columns),)))
    model.add(Dense(60, activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dense(30, activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    sgd = SGD(learning_rate=0.1, momentum=0.9)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_model, epochs=50, batch_size=16, verbose=1)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, apple_x, apple_y, cv=kfold)
print("Visible: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8170 - loss: 0.4071
Epoch 22/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8192 - loss: 0.4032
Epoch 23/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8076 - loss: 0.4376
Epoch 24/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8165 - loss: 0.4114
Epoch 25/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8094 - loss: 0.4102
Epoch 26/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8154 - loss: 0.4138
Epoch 27/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8174 - loss: 0.3932
```

```
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8124 - loss: 0.4049
```

We start our application of the Dropout into our visible layers, which are the input layers. We initialize our input layers and then succeed it with a Dropout layer, the 0.2 indicates that 20% of the layers will be "dropped" randomly whilst the model is training for each cycle. The evaluation results show a 88.67% accuracy on our first dropout use on the visible layer, which is lower than our baseline which is 91.45%

10. Show the application of Dropout on the hidden layer

```
# dropout in hidden layers with weight constraint
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(len(columns),), activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(30, activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    sgd = SGD(learning_rate=0.1, momentum=0.9)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_model, epochs=50, batch_size=16, verbose=1)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, apple_x, apple_y, cv=kfold)
print("Hidden: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.7606 - loss: 0.4676
Epoch 2/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8521 - loss: 0.3603
Epoch 3/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8532 - loss: 0.3182
Epoch 4/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8542 - loss: 0.3269
Epoch 5/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8641 - loss: 0.3354
Epoch 6/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8817 - loss: 0.2995
Epoch 7/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8790 - loss: 0.2943
Epoch 8/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8921 - loss: 0.2999
Epoch 9/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9017 - loss: 0.2491
Epoch 10/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8872 - loss: 0.2465
Epoch 11/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8969 - loss: 0.2538
Epoch 12/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9056 - loss: 0.2440
Epoch 13/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9118 - loss: 0.2419
Epoch 14/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9084 - loss: 0.2382
Epoch 15/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9190 - loss: 0.2186
Epoch 16/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9181 - loss: 0.2183
Epoch 17/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9114 - loss: 0.2422
Epoch 18/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9143 - loss: 0.2153
Epoch 19/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9082 - loss: 0.2111
Epoch 20/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9050 - loss: 0.2539
Epoch 21/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9090 - loss: 0.2327
Epoch 22/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9103 - loss: 0.2177
```

```
Epoch 23/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9190 - loss: 0.2051
Epoch 24/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9220 - loss: 0.2148
Epoch 25/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9051 - loss: 0.2174
Epoch 26/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9306 - loss: 0.1975
Epoch 27/50
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9241 - loss: 0.1963
```

Next, we applied dropout to the hidden layers instead of the input layers. After initializing the base and hidden layers, we added a dropout layer, resulting in two dropout layers. This model achieved a 91.22% accuracy, higher than the model with only input dropout but slightly less than the baseline model.

10.2. Dropout on Larger Networks and both Visible and Hidden Layers

```
# dropout in hidden layers with weight constraint
def create_model():
  # create model
  model = Sequential()
  model.add(Dropout(0.2, input_shape=(len(columns),)))
  model.add(Dense(60, activation='relu', kernel_constraint=MaxNorm(3)))
  model.add(Dropout(0.2))
  model.add(Dense(120, activation='relu', kernel_constraint=MaxNorm(3)))
  model.add(Dropout(0.2))
  model.add(Dense(60, activation='relu', kernel_constraint=MaxNorm(3)))
  model.add(Dropout(0.2))
  model.add(Dense(1, activation='sigmoid'))
  # Compile model
  sgd = SGD(learning_rate=0.1, momentum=0.9)
  model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
  return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_model, epochs=20, batch_size=16, verbose=1)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, apple_x, apple_y, cv=kfold)
print("Hidden: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
Epoch 1/20
/usr/local/lib/python3.10/dist-packages/keras/src/layers/regularization/dropout.py:42: UserWarning: Do not pass an `input_shape`/`inp
  super().__init__(**kwargs)
225/225 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.6736 - loss: 0.5810
Epoch 2/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7680 - loss: 0.5064
Epoch 3/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7762 - loss: 0.4848
Epoch 4/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7765 - loss: 0.4914
Epoch 5/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7574 - loss: 0.4941
Epoch 6/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7833 - loss: 0.4705
Epoch 7/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7873 - loss: 0.4721
Epoch 8/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7988 - loss: 0.4535
Epoch 9/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7948 - loss: 0.4674
Epoch 10/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7701 - loss: 0.4730
Epoch 11/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7897 - loss: 0.4758
Epoch 12/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7654 - loss: 0.4979
Epoch 13/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7647 - loss: 0.4981
Epoch 14/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7926 - loss: 0.4712
Epoch 15/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7746 - loss: 0.4975
Epoch 16/20
```

```
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7754 - loss: 0.4803
Epoch 17/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7840 - loss: 0.4778
Epoch 18/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7719 - loss: 0.4937
Epoch 19/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7665 - loss: 0.4936
Epoch 20/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7626 - loss: 0.5148
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Epoch 1/20
/usr/local/lib/python3.10/dist-packages/keras/src/layers/regularization/dropout.py:42: UserWarning: Do not pass an `input_shape`/`inp
  super().__init__(**kwargs)
225/225 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7124 - loss: 0.5607
Epoch 2/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7754 - loss: 0.4941
Epoch 3/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7714 - loss: 0.4874
Epoch 4/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7591 - loss: 0.5170
Epoch 5/20
225/225 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7833 - loss: 0.4933
Epoch 6/20
```

We used dropouts in a larger network with 3 hidden layers (60 nodes in the first and third, 120 in the second) and applied dropouts to both visible and hidden layers. This model had the worst performance with 84.08% accuracy. The baseline model had the best performance at 91.54%, followed by the model with only hidden dropouts, and then the model with only visible dropouts.

11. Show the application of a time-based learning rate schedule

```
# create model
model = Sequential()
model.add(Dense(len(columns), input_shape=(len(columns),), activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Configure the rate schedule
epochs = 50
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
sgd = SGD(learning_rate=learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)

# Compile model
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
# Fit the model
model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=epochs, batch_size=28, verbose=2)
```

```
96/96 - 0s - 2ms/step - accuracy: 0.8698 - loss: 0.3018 - val_accuracy: 0.8712 - val_loss: 0.2874
Epoch 34/50
96/96 - 0s - 2ms/step - accuracy: 0.8672 - loss: 0.3000 - val_accuracy: 0.8606 - val_loss: 0.3001
Epoch 35/50
96/96 - 0s - 2ms/step - accuracy: 0.8761 - loss: 0.3014 - val_accuracy: 0.8341 - val_loss: 0.3365
Epoch 36/50
96/96 - 0s - 2ms/step - accuracy: 0.8765 - loss: 0.2966 - val_accuracy: 0.8409 - val_loss: 0.3377
Epoch 37/50
96/96 - 0s - 2ms/step - accuracy: 0.8672 - loss: 0.3088 - val_accuracy: 0.8803 - val_loss: 0.2894
Epoch 38/50
96/96 - 0s - 3ms/step - accuracy: 0.8672 - loss: 0.2984 - val_accuracy: 0.8712 - val_loss: 0.2921
Epoch 39/50
96/96 - 0s - 4ms/step - accuracy: 0.8646 - loss: 0.3026 - val_accuracy: 0.8697 - val_loss: 0.2936
Epoch 40/50
96/96 - 1s - 6ms/step - accuracy: 0.8750 - loss: 0.2935 - val_accuracy: 0.8598 - val_loss: 0.3070
Epoch 41/50
96/96 - 1s - 7ms/step - accuracy: 0.8694 - loss: 0.2988 - val_accuracy: 0.8765 - val_loss: 0.2798
Epoch 42/50
96/96 - 1s - 7ms/step - accuracy: 0.8735 - loss: 0.2988 - val_accuracy: 0.8742 - val_loss: 0.2898
Epoch 43/50
96/96 - 1s - 6ms/step - accuracy: 0.8716 - loss: 0.3000 - val_accuracy: 0.8682 - val_loss: 0.2915
Epoch 44/50
96/96 - 1s - 5ms/step - accuracy: 0.8765 - loss: 0.2992 - val_accuracy: 0.8697 - val_loss: 0.2917
Epoch 45/50
96/96 - 0s - 3ms/step - accuracy: 0.8672 - loss: 0.2944 - val_accuracy: 0.8727 - val_loss: 0.2953
Epoch 46/50
96/96 - 0s - 3ms/step - accuracy: 0.8784 - loss: 0.2915 - val_accuracy: 0.8689 - val_loss: 0.2930
Epoch 47/50
96/96 - 0s - 2ms/step - accuracy: 0.8735 - loss: 0.3002 - val accuracy: 0.8606 - val loss: 0.2986
```

```
scores = model.evaluate(x_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Accuracy: 85.53%
```

We can see here that we used a time based learning rate schedule, that shows how decaying comes into place in learning rate on our SGD optimizer, means that our learning rate decays and gets smaller overtime.

12. Show the application of a drop-based learning rate schedule

```
import math
from tensorflow.keras.callbacks import LearningRateScheduler
# learning rate schedule
def step_decay(epoch):
    initial_lrate = 0.1
    drop = 0.5
    epochs_drop = 10.0
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lrate


# create model
model = Sequential()
model.add(Dense(len(columns), input_shape=(len(columns),), activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
sgd = SGD(learning_rate=0.0, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
# learning schedule callback
lrate = LearningRateScheduler(step_decay)
callbacks_list = [lrate]
# Fit the model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=50, batch_size=28, callbacks=callbacks_list, verbose=2)
```

```
Epoch 1/50
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
96/96 - 2s - 20ms/step - accuracy: 0.7515 - loss: 0.4993 - val_accuracy: 0.7909 - val_loss: 0.4172 - learning_rate: 0.1000
Epoch 2/50
96/96 - 0s - 2ms/step - accuracy: 0.8007 - loss: 0.4194 - val_accuracy: 0.8167 - val_loss: 0.3980 - learning_rate: 0.1000
Epoch 3/50
96/96 - 0s - 2ms/step - accuracy: 0.8235 - loss: 0.3899 - val_accuracy: 0.8379 - val_loss: 0.3316 - learning_rate: 0.1000
Epoch 4/50
96/96 - 0s - 3ms/step - accuracy: 0.8433 - loss: 0.3579 - val_accuracy: 0.8447 - val_loss: 0.3372 - learning_rate: 0.1000
Epoch 5/50
96/96 - 0s - 3ms/step - accuracy: 0.8437 - loss: 0.3590 - val_accuracy: 0.8500 - val_loss: 0.3286 - learning_rate: 0.1000
Epoch 6/50
```

```
96/96 - 0s - 3ms/step - accuracy: 0.8418 - loss: 0.3546 - val_accuracy: 0.8470 - val_loss: 0.3294 - learning_rate: 0.1000
Epoch 7/50
96/96 - 0s - 3ms/step - accuracy: 0.8448 - loss: 0.3465 - val_accuracy: 0.8553 - val_loss: 0.2975 - learning_rate: 0.1000
Epoch 8/50
96/96 - 0s - 3ms/step - accuracy: 0.8466 - loss: 0.3392 - val_accuracy: 0.8508 - val_loss: 0.3341 - learning_rate: 0.1000
Epoch 9/50
96/96 - 0s - 5ms/step - accuracy: 0.8511 - loss: 0.3433 - val_accuracy: 0.8364 - val_loss: 0.3158 - learning_rate: 0.1000
Epoch 10/50
96/96 - 1s - 5ms/step - accuracy: 0.8638 - loss: 0.3143 - val_accuracy: 0.8712 - val_loss: 0.2917 - learning_rate: 0.0500
Epoch 11/50
96/96 - 1s - 6ms/step - accuracy: 0.8675 - loss: 0.3115 - val_accuracy: 0.8485 - val_loss: 0.3192 - learning_rate: 0.0500
Epoch 12/50
96/96 - 1s - 7ms/step - accuracy: 0.8799 - loss: 0.3038 - val_accuracy: 0.8652 - val_loss: 0.3008 - learning_rate: 0.0500
Epoch 13/50
96/96 - 0s - 4ms/step - accuracy: 0.8683 - loss: 0.3120 - val_accuracy: 0.8667 - val_loss: 0.2907 - learning_rate: 0.0500
Epoch 14/50
96/96 - 0s - 5ms/step - accuracy: 0.8750 - loss: 0.3011 - val_accuracy: 0.8652 - val_loss: 0.2947 - learning_rate: 0.0500
Epoch 15/50
96/96 - 0s - 3ms/step - accuracy: 0.8701 - loss: 0.3075 - val_accuracy: 0.8402 - val_loss: 0.3334 - learning_rate: 0.0500
Epoch 16/50
96/96 - 0s - 3ms/step - accuracy: 0.8679 - loss: 0.3156 - val_accuracy: 0.8621 - val_loss: 0.2860 - learning_rate: 0.0500
Epoch 17/50
96/96 - 0s - 3ms/step - accuracy: 0.8672 - loss: 0.3026 - val_accuracy: 0.8439 - val_loss: 0.3111 - learning_rate: 0.0500
Epoch 18/50
96/96 - 0s - 3ms/step - accuracy: 0.8698 - loss: 0.3051 - val_accuracy: 0.8689 - val_loss: 0.2899 - learning_rate: 0.0500
Epoch 19/50
96/96 - 0s - 2ms/step - accuracy: 0.8735 - loss: 0.2998 - val_accuracy: 0.8598 - val_loss: 0.2953 - learning_rate: 0.0500
Epoch 20/50
96/96 - 0s - 3ms/step - accuracy: 0.8769 - loss: 0.2907 - val_accuracy: 0.8705 - val_loss: 0.2791 - learning_rate: 0.0250
Epoch 21/50
96/96 - 0s - 3ms/step - accuracy: 0.8787 - loss: 0.2913 - val_accuracy: 0.8485 - val_loss: 0.3094 - learning_rate: 0.0250
Epoch 22/50
96/96 - 0s - 3ms/step - accuracy: 0.8780 - loss: 0.2910 - val_accuracy: 0.8636 - val_loss: 0.2851 - learning_rate: 0.0250
Epoch 23/50
96/96 - 0s - 2ms/step - accuracy: 0.8832 - loss: 0.2859 - val_accuracy: 0.8614 - val_loss: 0.3024 - learning_rate: 0.0250
Epoch 24/50
96/96 - 0s - 3ms/step - accuracy: 0.8765 - loss: 0.2895 - val_accuracy: 0.8644 - val_loss: 0.2872 - learning_rate: 0.0250
Epoch 25/50
96/96 - 0s - 3ms/step - accuracy: 0.8825 - loss: 0.2878 - val_accuracy: 0.8689 - val_loss: 0.2893 - learning_rate: 0.0250
Epoch 26/50
96/96 - 0s - 3ms/step - accuracy: 0.8757 - loss: 0.2876 - val_accuracy: 0.8697 - val_loss: 0.2797 - learning_rate: 0.0250
Epoch 27/50
96/96 - 0s - 3ms/step - accuracy: 0.8784 - loss: 0.2872 - val_accuracy: 0.8712 - val_loss: 0.2803 - learning_rate: 0.0250
Epoch 28/50
```

```
scores = model.evaluate(x_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Accuracy: 87.65%
```

```
# list all data in history
print(history.history.keys())
# summarize history for learning rate
plt.plot(history.history['learning_rate'])
plt.title('Model Learning Rate')
plt.ylabel('learning rate')
plt.xlabel('epoch')
plt.legend(['learning rate'], loc='upper right')
plt.show()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss', 'learning_rate'])
```