Time Series Prediction

Objectives

1. Build a linear, DNN and CNN model in Keras.
2. Build a simple RNN model and a multi-layer RNN model in Keras.

In this lab we will with a linear, DNN and CNN model

Since the features of our model are sequential in nature, we'll next look at how to build various RNN models in Keras. We'll start with a simple RNN model and then see how to create a multi-layer RNN in Keras.

We will be exploring a lot of different model types in this notebook.

In [1]:

```
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [2]:

```
!pip install --user google-cloud-bigquery==1.25.0
```

```
Collecting google-cloud-bigquery==1.25.0
  Downloading google_cloud_bigquery-1.25.0-py2.py3-none-any.whl (169 kB)
     |████████████████████████████████| 169 kB 7.8 MB/s eta 0:00:01
Collecting google-cloud-core<2.0dev,>=1.1.0
  Downloading google_cloud_core-1.7.2-py2.py3-none-any.whl (28 kB)
Requirement already satisfied: protobuf>=3.6.0 in /opt/conda/lib/python3.7/site-
packages (from google-cloud-bigquery==1.25.0) (3.18.1)
Collecting google-api-core<2.0dev,>=1.15.0
  Downloading google_api_core-1.31.3-py2.py3-none-any.whl (93 kB)
     |████████████████████████████████| 93 kB 2.1 MB/s  eta 0:00:01
Collecting google-auth<2.0dev,>=1.9.0
  Downloading google_auth-1.35.0-py2.py3-none-any.whl (152 kB)
     |████████████████████████████████| 152 kB 28.1 MB/s eta 0:00:01
Requirement already satisfied: six<2.0.0dev,>=1.13.0 in /opt/conda/lib/python3.
7/site-packages (from google-cloud-bigquery==1.25.0) (1.16.0)
Collecting google-resumable-media<0.6dev,>=0.5.0
  Downloading google_resumable_media-0.5.1-py2.py3-none-any.whl (38 kB)
Requirement already satisfied: googleapis-common-protos<2.0dev,>=1.6.0 in /opt/c
onda/lib/python3.7/site-packages (from google-api-core<2.0dev,>=1.15.0->google-c
loud-bigquery==1.25.0) (1.53.0)
Requirement already satisfied: pytz in /opt/conda/lib/python3.7/site-packages (f
rom google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.0) (2021.3)
Collecting protobuf>=3.6.0
  Downloading protobuf-3.17.3-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.
whl (1.0 MB)
     |████████████████████████████████| 1.0 MB 42.2 MB/s eta 0:00:01
Requirement already satisfied: requests<3.0.0dev,>=2.18.0 in /opt/conda/lib/pyth
on3.7/site-packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery
==1.25.0) (2.25.1)
Requirement already satisfied: setuptools>=40.3.0 in /opt/conda/lib/python3.7/si
te-packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.
0) (58.2.0)
Requirement already satisfied: packaging>=14.3 in /opt/conda/lib/python3.7/site-
packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.0)
```

```
(21.0)
Requirement already satisfied: rsa<5,>=3.1.4 in /opt/conda/lib/python3.7/site-pa
ckages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (4.7.2)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /opt/conda/lib/python3.
7/site-packages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0)
(0.2.7)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /opt/conda/lib/python3.
7/site-packages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0)
(4.2.4)
Requirement already satisfied: pyparsing>=2.0.2 in /opt/conda/lib/python3.7/site
-packages (from packaging>=14.3->google-api-core<2.0dev,>=1.15.0->google-cloud-b
igquery==1.25.0) (2.4.7)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /opt/conda/lib/python3.7/
site-packages (from pyasn1-modules>=0.2.1->google-auth<2.0dev,>=1.9.0->google-cl
oud-bigquery==1.25.0) (0.4.8)
Requirement already satisfied: idna<3,>=2.5 in /opt/conda/lib/python3.7/site-pac
kages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->google-
cloud-bigquery==1.25.0) (2.10)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /opt/conda/lib/python3.
7/site-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.
0->google-cloud-bigquery==1.25.0) (1.26.7)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/lib/python3.7/si
te-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->g
oogle-cloud-bigquery==1.25.0) (2021.10.8)
Requirement already satisfied: chardet<5,>=3.0.2 in /opt/conda/lib/python3.7/sit
e-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->go
ogle-cloud-bigquery==1.25.0) (4.0.0)
Installing collected packages: protobuf, google-auth, google-api-core, google-re
sumable-media, google-cloud-core, google-cloud-bigquery
ERROR: pip's dependency resolver does not currently take into account all the pa
ckages that are installed. This behaviour is the source of the following depende
ncy conflicts.
tensorflow-io 0.18.0 requires tensorflow-io-gcs-filesystem==0.18.0, which is not
installed.
explainable-ai-sdk 1.3.2 requires xai-image-widget, which is not installed.
tfx-bsl 1.3.0 requires absl-py<0.13,>=0.9, but you have absl-py 0.14.1 which is
 incompatible.
tfx-bsl 1.3.0 requires google-api-python-client<2,>=1.7.11, but you have google-
api-python-client 2.24.0 which is incompatible.
tfx-bsl 1.3.0 requires pyarrow<3,>=1, but you have pyarrow 5.0.0 which is incomp
atible.
tensorflow 2.6.0 requires six~=1.15.0, but you have six 1.16.0 which is incompat
ible.
tensorflow 2.6.0 requires tensorboard~=2.6, but you have tensorboard 2.5.0 which
is incompatible.
tensorflow 2.6.0 requires typing-extensions~=3.7.4, but you have typing-extensio
ns 3.10.0.2 which is incompatible.
tensorflow 2.6.0 requires wrapt~=1.12.1, but you have wrapt 1.13.1 which is inco
mpatible.
tensorflow-transform 1.3.0 requires absl-py<0.13,>=0.9, but you have absl-py 0.1
4.1 which is incompatible.
tensorflow-transform 1.3.0 requires pyarrow<3,>=1, but you have pyarrow 5.0.0 wh
ich is incompatible.
tensorflow-metadata 1.2.0 requires absl-py<0.13,>=0.9, but you have absl-py 0.1
4.1 which is incompatible.
tensorflow-io 0.18.0 requires tensorflow<2.6.0,>=2.5.0, but you have tensorflow
 2.6.0 which is incompatible.
google-cloud-storage 1.42.3 requires google-resumable-media<3.0dev,>=1.3.0; pyth
on_version >= "3.6", but you have google-resumable-media 0.5.1 which is incompat
ible.
cloud-tpu-client 0.10 requires google-api-python-client==1.8.0, but you have goo
```

```
gle-api-python-client 2.24.0 which is incompatible.
apache-beam 2.33.0 requires dill<0.3.2,>=0.3.1.1, but you have dill 0.3.4 which
 is incompatible.
apache-beam 2.33.0 requires httplib2<0.20.0,>=0.8, but you have httplib2 0.20.1
 which is incompatible.
apache-beam 2.33.0 requires pyarrow<5.0.0,>=0.15.1, but you have pyarrow 5.0.0 w
hich is incompatible.
Successfully installed google-api-core-1.31.3 google-auth-1.35.0 google-cloud-bi
gquery-1.25.0 google-cloud-core-1.7.2 google-resumable-media-0.5.1 protobuf-3.1
7.3
```

Note: Restart your kernel to use updated packages.

Kindly ignore the deprecation warnings and incompatibility errors related to google-cloud-storage.

Load necessary libraries and set up environment variables

In [1]:

```
PROJECT = "qwiklabs-gcp-01-832ad75c1b64" # REPLACE WITH YOUR PROJECT NAME
BUCKET = "qwiklabs-gcp-01-832ad75c1b64" # REPLACE WITH YOUR BUCKET
REGION = "us-central1" # REPLACE WITH YOUR BUCKET REGION e.g. us-central1
```

In [2]:

```
%env
PROJECT = PROJECT
BUCKET = BUCKET
REGION = REGION
```

In [3]:

```
import os

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf

from google.cloud import bigquery
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Dense, DenseFeatures,
                                     Conv1D, MaxPool1D,
                                     Reshape, RNN,
                                     LSTM, GRU, Bidirectional)
from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint
from tensorflow.keras.optimizers import Adam

# To plot pretty figures
%matplotlib inline
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# For reproducible results.
from numpy.random import seed
seed(1)
tf.random.set_seed(2)
```

Explore time series data

We'll start by pulling a small sample of the time series data from Big Query and write some helper functions to clean up the data for modeling. We'll use the data from the `percent_change_sp500` table in BigQuery. The `close_values_prior_260` column contains the close values for any given stock for the previous 260 days.

In [4]:

```
%%time
bq = bigquery.Client(project=PROJECT)

bq_query = '''
#standardSQL
SELECT
  symbol,
  Date,
  direction,
  close_values_prior_260
FROM
  `stock_market.eps_percent_change_sp500`
LIMIT
  100
'''
```

```
CPU times: user 5.83 ms, sys: 234 µs, total: 6.06 ms
Wall time: 10.9 ms
```

The function `clean_data` below does three things:

1. First, we'll remove any inf or NA values
2. Next, we parse the `Date` field to read it as a string.
3. Lastly, we convert the label `direction` into a numeric quantity, mapping 'DOWN' to 0, 'STAY' to 1 and 'UP' to 2.

In [5]:

```
def clean_data(input_df):
    """Cleans data to prepare for training.

    Args:
        input_df: Pandas dataframe.
    Returns:
        Pandas dataframe.
    """
    df = input_df.copy()

    # Remove inf/na values.
    real_valued_rows = ~(df == np.inf).max(axis=1)
    df = df[real_valued_rows].dropna()

    # TF doesn't accept datetimes in DataFrame.
    df['Date'] = pd.to_datetime(df['Date'], errors='coerce')
    df['Date'] = df['Date'].dt.strftime('%Y-%m-%d')

    # TF requires numeric label.
    df['direction_numeric'] = df['direction'].apply(lambda x: {'DOWN': 0,
                                                               'STAY': 1,
                                                               'UP': 2}[x])

    return df
```

Read data and preprocessing

Before we begin modeling, we'll preprocess our features by scaling to the z-score. This will ensure that the range of the feature values being fed to the model are comparable and should help with convergence during gradient descent.

In [6]:

```
STOCK_HISTORY_COLUMN = 'close_values_prior_260'
COL_NAMES = ['day_' + str(day) for day in range(0, 260)]
LABEL = 'direction_numeric'
```

In [7]:

```
def _scale_features(df):
    """z-scale feature columns of Pandas dataframe.

    Args:
        features: Pandas dataframe.
    Returns:
        Pandas dataframe with each column standardized according to the
        values in that column.
    """
    avg = df.mean()
    std = df.std()
    return (df - avg) / std


def create_features(df, label_name):
    """Create modeling features and label from Pandas dataframe.

    Args:
        df: Pandas dataframe.
        label_name: str, the column name of the label.
    Returns:
        Pandas dataframe
    """
    # Expand 1 column containing a list of close prices to 260 columns.
    time_series_features = df[STOCK_HISTORY_COLUMN].apply(pd.Series)

    # Rename columns.
    time_series_features.columns = COL_NAMES
    time_series_features = _scale_features(time_series_features)

    # Concat time series features with static features and label.
    label_column = df[LABEL]

    return pd.concat([time_series_features,
                      label_column], axis=1)
```

Make train-eval-test split

Next, we'll make repeatable splits for our train/validation/test datasets and save these datasets to local csv files. The query below will take a subsample of the entire dataset and then create a 70-15-15 split for the train/validation/test sets.

In [8]:

```
def _create_split(phase):
```

```
    """Create string to produce train/valid/test splits for a SQL query.

    Args:
        phase: str, either TRAIN, VALID, or TEST.
    Returns:
        String.
    """
    floor, ceiling = '2002-11-01', '2010-07-01'
    if phase == 'VALID':
        floor, ceiling = '2010-07-01', '2011-09-01'
    elif phase == 'TEST':
        floor, ceiling = '2011-09-01', '2012-11-30'
    return '''
    WHERE Date >= '{0}'
    AND Date < '{1}'
    '''.format(floor, ceiling)


def create_query(phase):
    """Create SQL query to create train/valid/test splits on subsample.

    Args:
        phase: str, either TRAIN, VALID, or TEST.
        sample_size: str, amount of data to take for subsample.
    Returns:
        String.
    """
    basequery = """
    #standardSQL
    SELECT
      symbol,
      Date,
      direction,
      close_values_prior_260
    FROM
      `stock_market.eps_percent_change_sp500`
    """

    return basequery + _create_split(phase)
```

Modeling

For experimentation purposes, we'll train various models using data we can fit in memory using the `.csv` files we created above.

In [9]:

```
N_TIME_STEPS = 260
N_LABELS = 3

Xtrain = pd.read_csv('stock-train.csv')
Xvalid = pd.read_csv('stock-valid.csv')

ytrain = Xtrain.pop(LABEL)
yvalid  = Xvalid.pop(LABEL)

ytrain_categorical = to_categorical(ytrain.values)
yvalid_categorical = to_categorical(yvalid.values)
```

To monitor training progress and compare evaluation metrics for different models, we'll use the function below to plot metrics captured from the training job such as training and validation loss or accuracy.

In [10]:

```
def plot_curves(train_data, val_data, label='Accuracy'):
    """Plot training and validation metrics on single axis.

    Args:
        train_data: list, metrics obtrained from training data.
        val_data: list, metrics obtained from validation data.
        label: str, title and label for plot.
    Returns:
        Matplotlib plot.
    """
    plt.plot(np.arange(len(train_data)) + 0.5,
             train_data,
             "b.-", label="Training " + label)
    plt.plot(np.arange(len(val_data)) + 1,
             val_data, "r.-",
             label="Validation " + label)
    plt.gca().xaxis.set_major_locator(mpl.ticker.MaxNLocator(integer=True))
    plt.legend(fontsize=14)
    plt.xlabel("Epochs")
    plt.ylabel(label)
    plt.grid(True)
```

Baseline

Before we begin modeling in Keras, let's create a benchmark using a simple heuristic. Let's see what kind of accuracy we would get on the validation set if we predict the majority class of the training set.

In [11]:

```
sum(yvalid == ytrain.value_counts().idxmax()) / yvalid.shape[0]
```

Out[11]:

```
0.29490392648287383
```

Ok. So just naively guessing the most common outcome `UP` will give about 29.5% accuracy on the validation set.

Linear model

We'll start with a simple linear model, mapping our sequential input to a single fully dense layer.

In [12]:

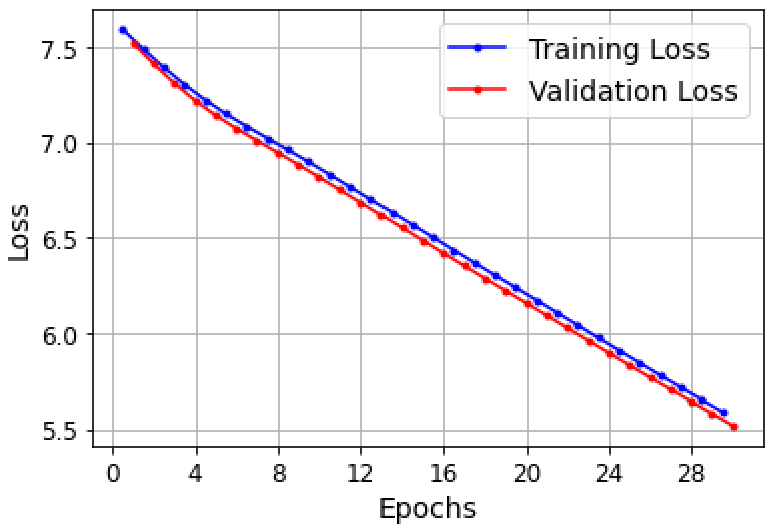
```
model = Sequential()
model.add(Dense(units=N_LABELS,
                activation='softmax',
                kernel_regularizer=tf.keras.regularizers.l1(l=0.1)))

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x=Xtrain.values,
                    y=ytrain_categorical,
```

```
                    batch_size=Xtrain.shape[0],
                    validation_data=(Xvalid.values, yvalid_categorical),
                    epochs=30,
                    verbose=0)
```

```
2021-10-19 13:07:18.582421: I tensorflow/core/common_runtime/process_util.cc:14
6] Creating new thread pool with default inter op setting: 2. Tune using inter_o
p_parallelism_threads for best performance.
2021-10-19 13:07:18.713861: I tensorflow/compiler/mlir/mlir_graph_optimization_p
ass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
```

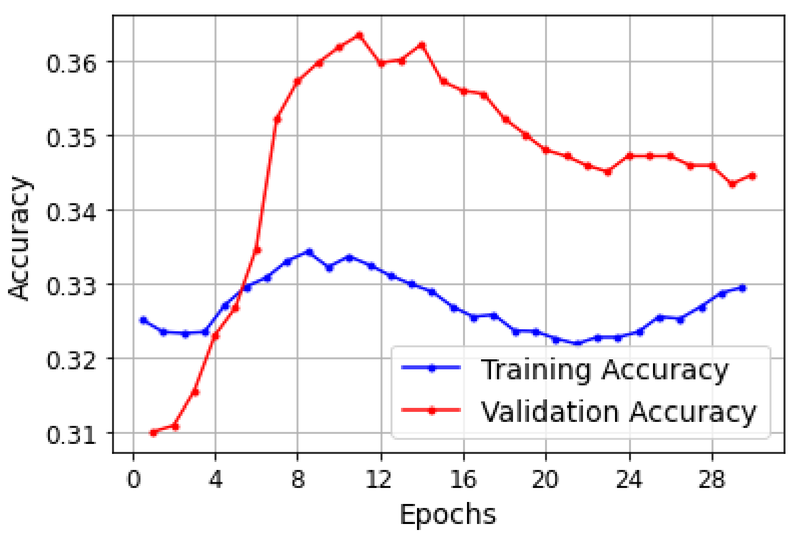
In [13]:

```
plot_curves(history.history['loss'],
            history.history['val_loss'],
            label='Loss')
```



In [14]:

```
plot_curves(history.history['accuracy'],
            history.history['val_accuracy'],
            label='Accuracy')
```



The accuracy seems to level out pretty quickly. To report the accuracy, we'll average the accuracy on the validation set across the last few epochs of training.

In [15]:

```
np.mean(history.history['val_accuracy'][-5:])
```

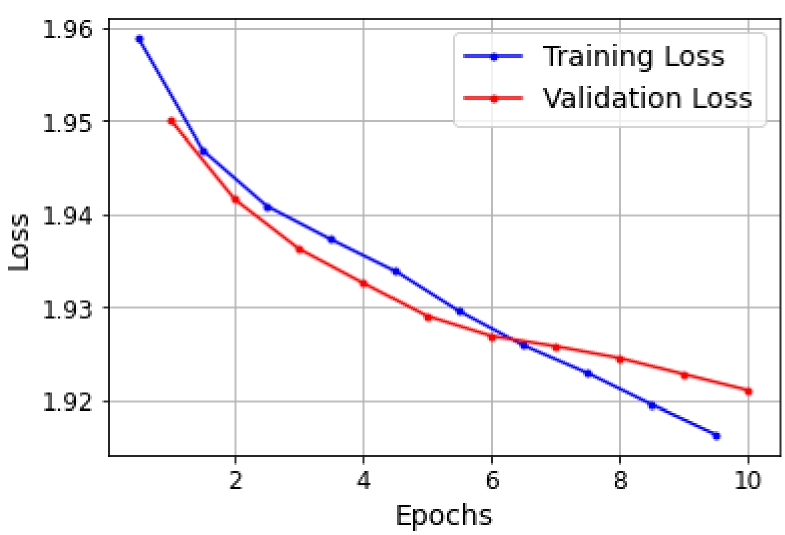
Out[15]: 0.3453634023666382

Deep Neural Network

The linear model is an improvement on our naive benchmark. Perhaps we can do better with a more complicated model. Next, we'll create a deep neural network with Keras. We'll experiment with a two layer DNN here but feel free to try a more complex model or add any other additional techniques to try an improve your performance.

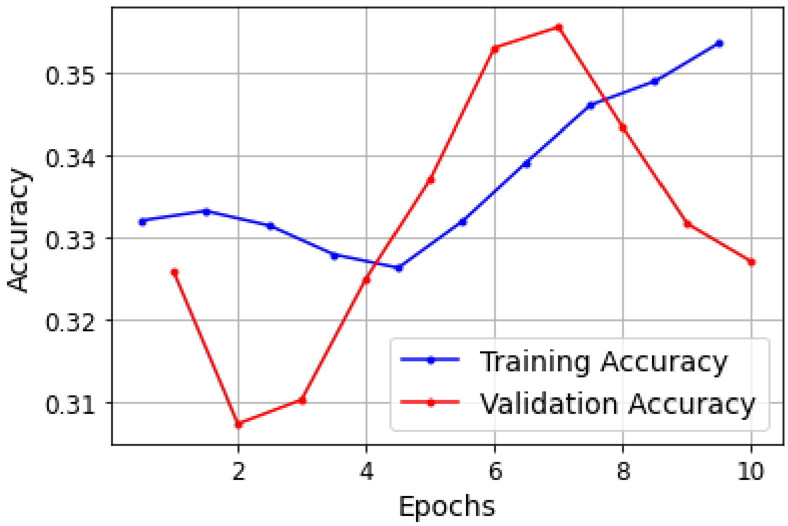
In [16]:
```
dnn_hidden_units = [16, 8]

model = Sequential()
for layer in dnn_hidden_units:
    model.add(Dense(units=layer,
                    activation="relu"))

model.add(Dense(units=N_LABELS,
                activation="softmax",
                kernel_regularizer=tf.keras.regularizers.l1(l=0.1)))

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x=Xtrain.values,
                    y=ytrain_categorical,
                    batch_size=Xtrain.shape[0],
                    validation_data=(Xvalid.values, yvalid_categorical),
                    epochs=10,
                    verbose=0)
```

In [17]:
```
plot_curves(history.history['loss'],
            history.history['val_loss'],
            label='Loss')
```



In [18]:
```
plot_curves(history.history['accuracy'],
```

```
             history.history['val_accuracy'],
             label='Accuracy')
```



In [19]:

```
np.mean(history.history['val_accuracy'][-5:])
```

Out[19]:

```
0.3421052634716034
```

Convolutional Neural Network

The DNN does slightly better. Let's see how a convolutional neural network performs.

A 1-dimensional convolutional can be useful for extracting features from sequential data or deriving features from shorter, fixed-length segments of the data set. Check out the documentation for how to implement a Conv1d in Tensorflow. Max pooling is a downsampling strategy commonly used in conjunction with convolutional neural networks. Next, we'll build a CNN model in Keras using the `Conv1D` to create convolution layers and `MaxPool1D` to perform max pooling before passing to a fully connected dense layer.

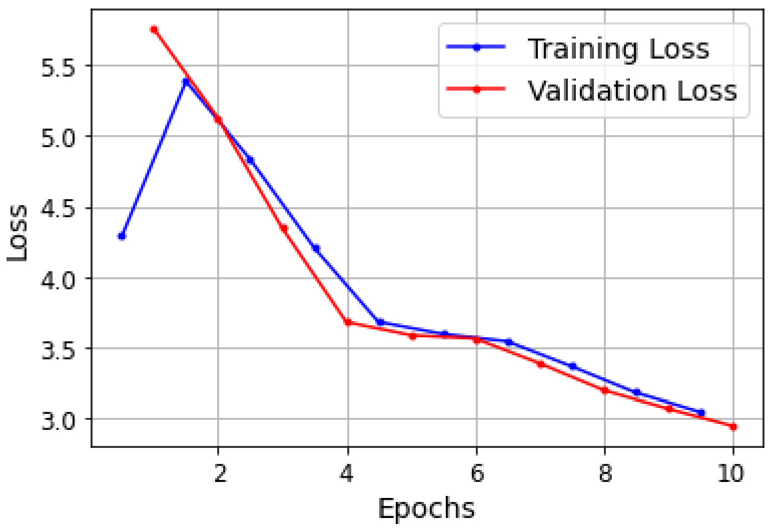
In [20]:

```
model = Sequential()

# Convolutional layer
model.add(Reshape(target_shape=[N_TIME_STEPS, 1]))
model.add(Conv1D(filters=5,
                 kernel_size=5,
                 strides=2,
                 padding="valid",
                 input_shape=[None, 1]))
model.add(MaxPool1D(pool_size=2,
                    strides=None,
                    padding='valid'))


# Flatten the result and pass through DNN.
model.add(tf.keras.layers.Flatten())
model.add(Dense(units=N_TIME_STEPS//4,
                activation="relu"))

model.add(Dense(units=N_LABELS,
```

```
                    activation="softmax",
                    kernel_regularizer=tf.keras.regularizers.l1(l=0.1)))

model.compile(optimizer=Adam(learning_rate=0.01),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x=Xtrain.values,
                    y=ytrain_categorical,
                    batch_size=Xtrain.shape[0],
                    validation_data=(Xvalid.values, yvalid_categorical),
                    epochs=10,
                    verbose=0)
```

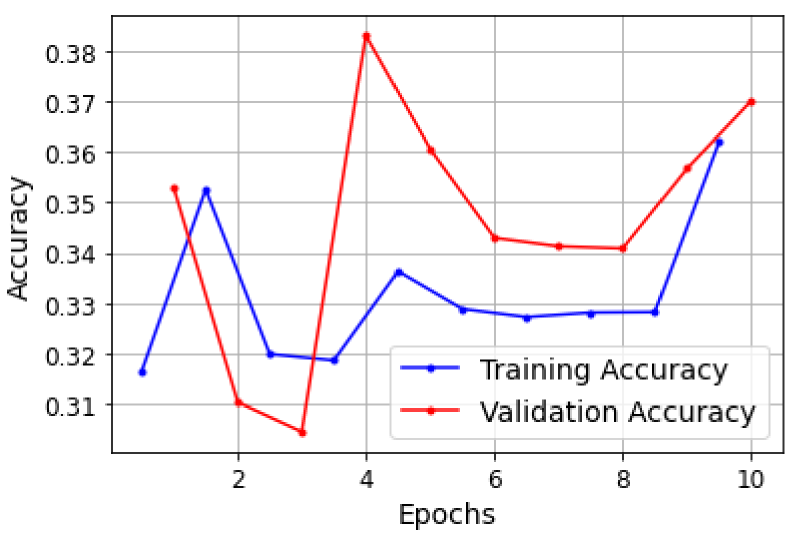
In [21]:

```
plot_curves(history.history['loss'],
            history.history['val_loss'],
            label='Loss')
```



In [22]:

```
plot_curves(history.history['accuracy'],
            history.history['val_accuracy'],
            label='Accuracy')
```



In [23]:
```
np.mean(history.history['val_accuracy'][-5:])
```

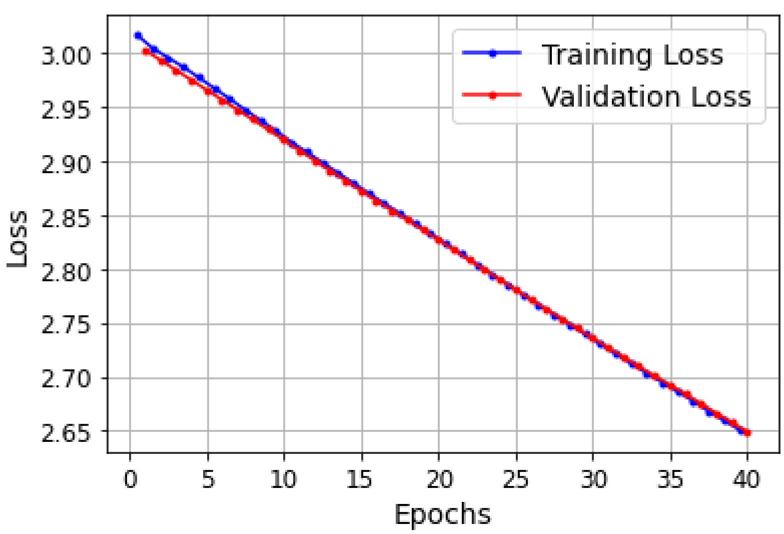
Out[23]:
```
0.35037594437599184
```

Recurrent Neural Network

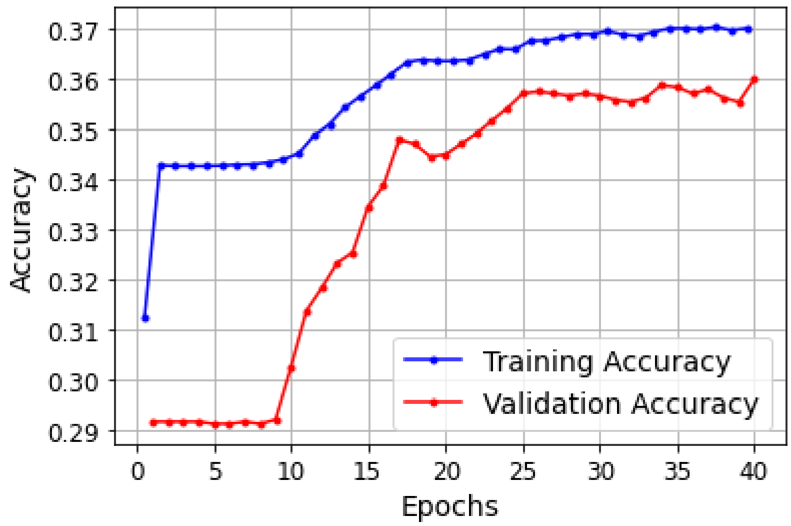
RNNs are particularly well-suited for learning sequential data. They retain state information from one iteration to the next by feeding the output from one cell as input for the next step. In the cell below, we'll build a RNN model in Keras. The final state of the RNN is captured and then passed through a fully connected layer to produce a prediction.

In [24]:
```
model = Sequential()

# Reshape inputs to pass through RNN layer.
model.add(Reshape(target_shape=[N_TIME_STEPS, 1]))
model.add(LSTM(N_TIME_STEPS // 8,
               activation='relu',
               return_sequences=False))

model.add(Dense(units=N_LABELS,
                activation='softmax',
                kernel_regularizer=tf.keras.regularizers.l1(l=0.1)))

# Create the model.
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x=Xtrain.values,
                    y=ytrain_categorical,
                    batch_size=Xtrain.shape[0],
                    validation_data=(Xvalid.values, yvalid_categorical),
                    epochs=40,
                    verbose=0)
```

In [25]:
```
plot_curves(history.history['loss'],
            history.history['val_loss'],
            label='Loss')
```



In [26]:
```
plot_curves(history.history['accuracy'],
            history.history['val_accuracy'],
            label='Accuracy')
```



In [27]:
```
np.mean(history.history['val_accuracy'][-5:])
```

Out[27]: 0.35739349126815795

Multi-layer RNN

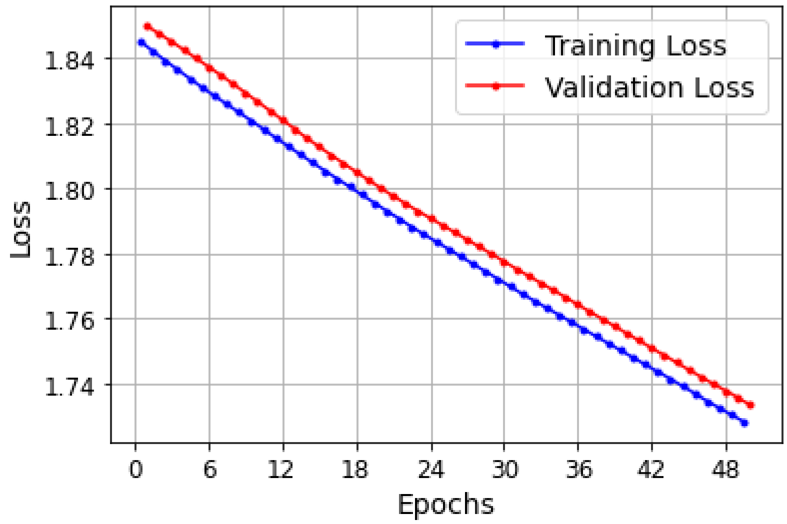
Next, we'll build multi-layer RNN. Just as multiple layers of a deep neural network allow for more complicated features to be learned during training, additional RNN layers can potentially learn complex features in sequential data. For a multi-layer RNN the output of the first RNN layer is fed as the input into the next RNN layer.

In [28]:
```
rnn_hidden_units = [N_TIME_STEPS // 16,
                    N_TIME_STEPS // 32]

model = Sequential()

# Reshape inputs to pass through RNN layer.
model.add(Reshape(target_shape=[N_TIME_STEPS, 1]))

for layer in rnn_hidden_units[:-1]:
    model.add(GRU(units=layer,
                  activation='relu',
                  return_sequences=True))

model.add(GRU(units=rnn_hidden_units[-1],
              return_sequences=False))
model.add(Dense(units=N_LABELS,
                activation="softmax",
                kernel_regularizer=tf.keras.regularizers.l1(l=0.1)))

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

```
history = model.fit(x=Xtrain.values,
                    y=ytrain_categorical,
                    batch_size=Xtrain.shape[0],
                    validation_data=(Xvalid.values, yvalid_categorical),
                    epochs=50,
                    verbose=0)
```

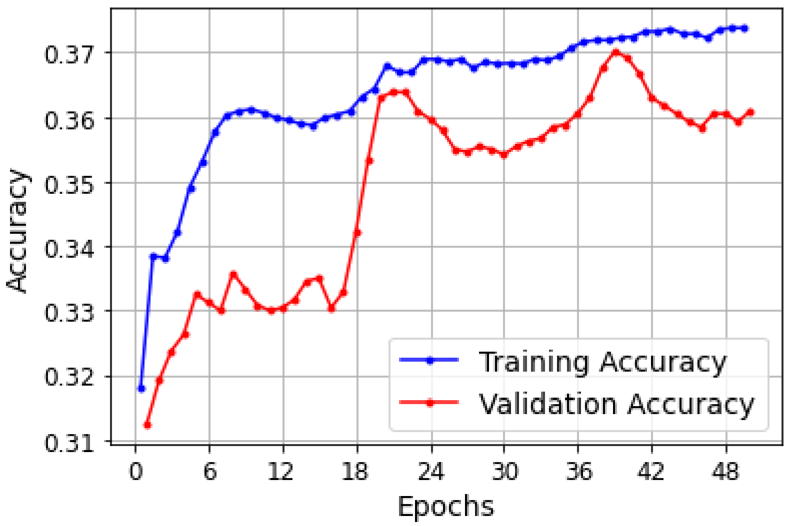
In [29]:

```
plot_curves(history.history['loss'],
            history.history['val_loss'],
            label='Loss')
```



In [30]:

```
plot_curves(history.history['accuracy'],
            history.history['val_accuracy'],
            label='Accuracy')
```



In [31]:

```
np.mean(history.history['val_accuracy'][-5:])
```

Out[31]: 0.3598997473716736