# ShortTermStockPricePredictionA-CNN+

December 16, 2020

**Authors:** Ruizhe Li (rl3070), Ruochen Nan (rn2498)

### 0.1 Goal

The goal of this project is using order information and deep learning methods to do the **short-term stock price trend prediction**.

### 0.2 Script Contents

In this script we are going to show how to build the **ACNN, ACNN+** given in the paper, and improved ACNN+ **with inceptron kernels**. We also tested LSTM models **with or without average pooling**, and we named them **A-LSTM, A-LSTM+**.

The script is based on Tensorflow 2.x, its Keras API, and common machine learning libraries (pandas, numpy, etc.)

### 0.3 Model Quick View

#### 0.3.1 Input:

Sequence of equity trading records from time T-window_size, to time T-1.
i.e. $t \in [T - windowSize, T - 1]$, where windowSize is a hyper-parameter used to control length of the sequence.

#### 0.3.2 Output:

The probability that the stock price is going up at time T.

### 0.4 Workflow

1. Dataset preparing
2. Data processing
3. Model building
4. Training and testing
5. Plot
6. Conclusion

## 1 Dataset preparing

We used the TAQ NYSE Equities public dataset of Oct 7th, 2019. Click here to download the data.

The raw data contains 9 columns: * symbol (abbreviation of stock) * status (pre-opening, stock halted, early session, open ,late session, stock closed) * date * time * order type (buy/sell) * price * shares * order numbers * listing market

You can take a look at TAQ OPENBOOK CLIENT SPECIFICATION for details.

As the data is **aggregated**, we selected 20 stocks randomly and stored them as CSV files **separately**.

## 1.1 Get 20 stocks randomly

[ ]:
```
# Get 20 stocks from the huge dataset
import numpy as np
import pandas as pd

count = 0
symbols = []
data = []
with open('./data/EQY_US_NYSE_BOOK_AGGR_20191007/
↪EQY_US_NYSE_BOOK_AGGR_20191007', 'rb') as f:
    while True:
        line = f.readline()
        if not line:
            break

    entries = line.strip().decode('ascii').split('|')
    symbol = entries[0]
    bs = entries[4]
    lm = entries[8]
    if lm == 'N' and bs and (symbol not in symbols) and len(symbols) < 20:
        symbols.append(symbol)
    if symbol in symbols and lm == 'N':
        data.append(entries)
```

[ ]:
```
# Build a CSV file with headers
cols = [
    'Symbol',
    'Status',
    'Date',
    'Time',
    'Buy/Sell',
    'Price Point',
    'Shares',
    'Number of Orders',
    'Listing Market'
]
df = pd.DataFrame(data, columns=cols)
df.to_csv('./data/data.csv')
```

[ ]:
```
# Check na values
df.isnull().sum() # Results showed there is no na value
```

[ ]:
```
Symbol              0
Status              0
Date                0
Time                0
Buy/Sell            0
Price Point         0
Shares              0
Number of Orders    0
Listing Market      0
dtype: int64
```

[ ]:
```
# Order type is relatively balanced
df['Buy/Sell'].value_counts()
```

[ ]:
```
B    801485
S    769369
Name: Buy/Sell, dtype: int64
```

### 1.2 Separate the CSV file according to different stocks

[ ]:
```
import pandas as pd
import numpy as np
import os
```

[ ]:
```
path = './data'
file_name = 'data.csv'
df = pd.read_csv(os.path.join(path, file_name), index_col='Unnamed: 0')
```

[ ]:
```
symbols = df['Symbol'].unique()
for symbol in symbols:
  data_split = df[df['Symbol']==symbol]
  data_split.to_csv(os.path.join(path, f'{symbol}.csv'))
```

[ ]:
```
!ls "$path"
```

```
APD.csv  'BRK A.csv'   CHWY.csv   CVS.csv    EPAM.csv   HD.csv    RCL.csv
BDX.csv   BVN.csv      CL.csv     DAL.csv    FNB.csv    HPQ.csv   SBGL.csv
BMY.csv   CDE.csv      CSTM.csv   data.csv   GOLD.csv   NVS.csv   TEVA.csv
```

## 2 Data processing

Here we are going to do data processing as shown in the paper.

### 2.0.1 Processing Steps:

- Compute and categorize time difference.
- Compute and categorize relative price
- Get crossed features
- Process label
- Get sequence according to time
- Split training/validation/test dataset

[1]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
```

## 2.1 Function definitions

We first define the functions we need

[5]:
```
def to_time(time_col):
    ''' Convert the time column from string to datetime '''
    time_col = time_col.astype('str').transform(lambda x: '0'+x if len(x.
↪split('.')[0]) < 6 else x)
    time_col = time_col.transform(lambda x: x[:15])
    time_col = pd.to_datetime(time_col, format="%H%M%S.%f")
    return time_col
```

[6]:
```
def get_time_diff(time_col):
    ''' Get time difference '''
    diff = time_col.diff()
    diff[diff.isna()] = 0 # Assign 'na' values to 0

    assert diff.isna().sum() == 0 # sanity check, no 'na'

    diff = diff.values.astype(np.float32) # nano-seconds
    diff /= 1e6 # milli-seconds
    return diff
```

[8]:
```
def diff_to_cat(time_diff):
    ''' Transfer time difference to categorical values '''
    time_diff = time_diff.transform(lambda x: 'TD_is_0' if x == 0 else␣
↪'TD_not_0')
    return time_diff
```

[13]:
```
def price_to_cat(relative_price):
    ''' Convert relative price to be categorical '''

    def relative_price_trans(relative_price):
    ''' A helper function '''
```

```
    thresholds = [1, 2, 3, 5, 7, 10]
    thresholds = [-i for i in thresholds[::-1]] + thresholds
    if relative_price < thresholds[0]:
        return f'<{thresholds[0]}'
    for i in range(1, len(thresholds)):
        if relative_price < thresholds[i]:
            return f'[{thresholds[i-1]},{thresholds[i]})'
    return f'>{thresholds[-1]}'

    res = relative_price.transform(relative_price_trans)
    return res
```

[20]:
```
def get_sequence(x, window_size=50):
    ''' Convert training samples from single datapoints to sequences '''
    x = np.array([x[i-window_size:i]
            for i in range(window_size, len(x))])
    return x
```

## 2.2 Load data

[2]:
```
path = './data'
symbol = 'APD' # The selected doc
data = pd.read_csv(os.path.join(path, f'{symbol}.csv'), index_col='Unnamed: 0')

data.shape
```

[2]:
```
(42213, 9)
```

[3]:
```
data.columns
```

[3]:
```
Index(['Symbol', 'Status', 'Date', 'Time', 'Buy/Sell', 'Price Point', 'Shares',
       'Number of Orders', 'Listing Market'],
      dtype='object')
```

[4]:
```
# Rename the column names
data = data[['Time', 'Buy/Sell', 'Price Point']]
data = data.rename(columns={'Time': 'time', 'Buy/Sell': 'order_type', 'Price␣
 ↪Point': 'price'})
```

## 2.3 Compute and categorize time difference

1. Calculate the time difference between the current data point and previous data point.

2. Categorize and one-hot encode the time difference.

The time difference could be a sign for trade participants, that means if the time difference between two data points are too small, then it is likely that the two records are made by the same person.

```
[ ]: data = data.sort_values(['time'])
     data['time'] = to_time(data['time'])
     data['time_diff'] = get_time_diff(data['time'])
```

Take a look at the time difference, we found over half of the time difference in our data is 0.

```
[7]: n = (data['time_diff'] == 0).sum()
     print(f'Number of data points whose time difference is 0: {n}')
     print(f'proportion: {n/len(data)}')
```

```
Number of data points whose time difference is 0: 31545
proportion: 0.7472816430957288
```

```
[ ]: data['time_diff_cat'] = diff_to_cat(data['time_diff'])
```

If we take a look at our target data, we would found that most of the prices are concentrated around 215.

### 2.4 Compute and categorize relative price

1. Compute the absolute price minus the median of all the price in that stock.
2. Categorize the relative price and turn them into one-hot encoding.

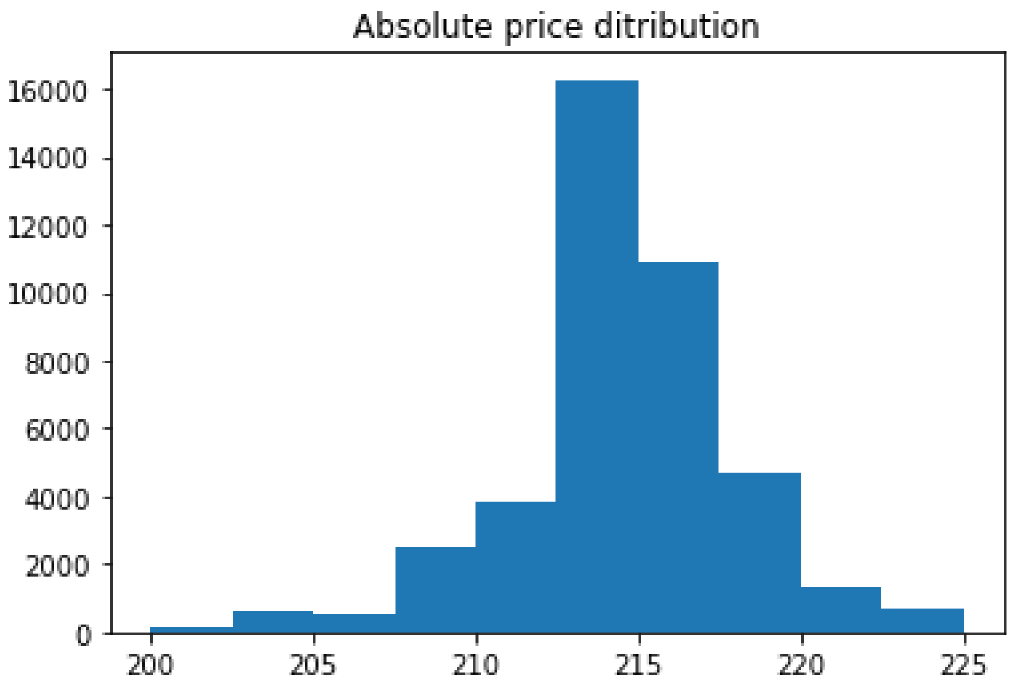
Because the relative price follows the power law, which means most of prices are concentrated around the median, it is hard to be normalized. Therefore, following the paper, we categorized them.

```
[9]: data.price.describe()
```

```
[9]: count    42213.000000
     mean       218.781125
     std        146.369588
     min          0.010000
     25%        213.240000
     50%        214.660000
     75%        215.440000
     max       4294.670000
     Name: price, dtype: float64
```

```
[10]: plt.title('Absolute price ditribution')
      plt.hist(data.price, range=(200, 225))
```

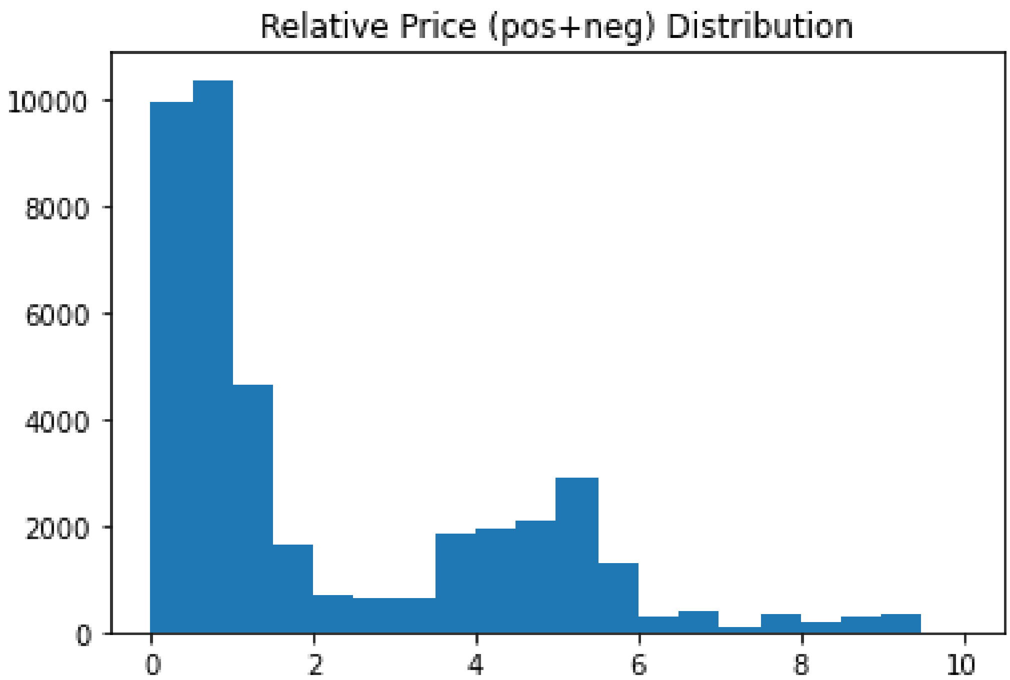
```
[10]: (array([  126.,   595.,   513.,  2481.,  3798., 16274., 10873.,  4708.,
          1303.,   664.]),
       array([200. , 202.5, 205. , 207.5, 210. , 212.5, 215. , 217.5, 220. ,
              222.5, 225. ]),
       <BarContainer object of 10 artists>)
```




```
[11]: # Get relative price
      data['relative_price'] = data['price'] - data['price'].median()
```

```
[12]: plt.hist(data['relative_price'].abs(), bins=20, range=(0, 10))
      plt.title('Relative Price (pos+neg) Distribution')
```

```
[12]: Text(0.5, 1.0, 'Relative Price (pos+neg) Distribution')
```




[14]:
```
data['relative_price_cat'] = price_to_cat(data['relative_price'])
```

### 2.5 Get crossed features

To capture the information between features, we followed the paper to build cross features.

[15]:
```
# Cross feature time difference and order type
data['TD_OT'] = data['time_diff_cat']+data['order_type']
```

### 2.6 Process label

we set label equal to 1 if the current price is higher than previous price, , otherwise we have a label 0.

[ ]:
```
# DEBUG!! How to set target??
# Up is 1, down is 0
data['target'] = data['relative_price'].diff()
data['target'][0] = 0
data['target'] = (data['target'] > 0).astype(np.int32)
```

[18]:
```
# Categorical features
data['order_type'] = data['order_type'].transform(lambda x: 1 if x=='B' else 0)
X = data[['order_type', 'time_diff_cat', 'relative_price_cat', 'TD_OT']]
X = pd.get_dummies(X, columns=['time_diff_cat', 'relative_price_cat', 'TD_OT']).
↪values
```

```
# Target
y = data['target'].values
```

### 2.7 Split training/validation/test dataset

[19]:
```
# Data split
n_train = int(len(X)*0.8)
n_val = int(len(X)*0.1)
X_train = X[:n_train]
X_val = X[n_train:n_train+n_val]
X_test = X[n_train+n_val:]

y_train = y[:n_train]
y_val = y[n_train:n_train+n_val]
y_test = y[n_train+n_val:]

# Sequenth length
window_size = 50
```

### 2.8 Get sequence according to time

We combine the adjacent data points to form sequences. In the experiment we use window of length 50. Which means, to predict the price up or down for current time t, we would use a sequence from time t-50 to t-1.

[21]:
```
# Get sequence features
X_train = get_sequence(X_train, window_size)
X_val = get_sequence(X_val, window_size)
X_test = get_sequence(X_test, window_size)
```

[22]:
```
from tensorflow.keras.utils import to_categorical
# # Get the label for corresponding sequences
y_train = y_train[window_size:]
y_val = y_val[window_size:]
y_test = y_test[window_size:]

y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)
```

[23]:
```
# Sanity Check
X_train.shape, y_train.shape
```

[23]:
```
((33720, 50, 20), (33720, 2))
```

### 2.9 Summary of data processing

In order to run our code for multiple stocks, we combined the data processing process in one section

[24]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical
path = './data'
symbol = 'APD' # The selected doc
```

[25]:
```
def data_processing(path, symbol):
    print(f'Data processing {symbol}')
    data = pd.read_csv(os.path.join(path, f'{symbol}.csv'), index_col='Unnamed:␣
 ↪0')

    # Rename the column names
    data = data[['Time', 'Buy/Sell', 'Price Point']]
    data = data.rename(columns={'Time': 'time', 'Buy/Sell': 'order_type',␣
 ↪'Price Point': 'price'})
    data = data.sort_values(['time'])

    data['time'] = to_time(data['time'])
    data['time_diff'] = get_time_diff(data['time'])
    data['time_diff_cat'] = diff_to_cat(data['time_diff'])
    data['relative_price'] = data['price'] - data['price'].median()

    data['relative_price_cat'] = price_to_cat(data['relative_price'])
    data['TD_OT'] = data['time_diff_cat']+data['order_type']

    # Up is 1, down is 0
    data['target'] = data['relative_price'].diff()
    data['target'][0] = 0
    data['target'] = (data['target'] > 0).astype(np.int32)

    # Categorical features
    data['order_type'] = data['order_type'].transform(lambda x: 1 if x=='B'␣
 ↪else 0)
    X = data[['order_type', 'time_diff_cat', 'relative_price_cat', 'TD_OT']]
    X = pd.get_dummies(X, columns=['time_diff_cat', 'relative_price_cat',␣
 ↪'TD_OT']).values

    # Target
    y = data['target'].values

    # Data split
    n_train = int(len(X)*0.7)
```

```
    n_val = int(len(X)*0.15)
    X_train = X[:n_train]
    X_val = X[n_train:n_train+n_val]
    X_test = X[n_train+n_val:]

    y_train = y[:n_train]
    y_val = y[n_train:n_train+n_val]
    y_test = y[n_train+n_val:]

    # Sequenth length
    window_size = 50

    # Get sequence features
    X_train = get_sequence(X_train, window_size)
    X_val = get_sequence(X_val, window_size)
    X_test = get_sequence(X_test, window_size)

    # Get the label for corresponding sequences
    y_train = y_train[window_size:]
    y_val = y_val[window_size:]
    y_test = y_test[window_size:]

    y_train = to_categorical(y_train)
    y_val = to_categorical(y_val)
    y_test = to_categorical(y_test)

    print(X_train.shape)
    return X_train, X_val, X_test, y_train, y_val, y_test
```

[ ]:
```
# Sanity check
X_train, X_val, X_test, y_train, y_val, y_test = data_processing(path, symbol)
```

# 3 Model building

We are going to build several models including CNN and LSTM. ### CNN * CNN * ACNN * ACNN+ * CNN with inceptron module * ACNN with inceptron module * ACNN+ with inceptron module

### 3.0.1 LSTM

- LSTM with average pooling
- LSTM with various LSTM layers

## 3.1 CNN

We combine the model building process for all the CNN models in one function.

[27]:
```python
import tensorflow as tf
from datetime import datetime

from tensorflow.keras.models import Sequential
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import AveragePooling1D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Reshape
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import MaxPooling1D
from tensorflow.keras.layers import concatenate
from tensorflow.keras.layers import Embedding

from tensorflow.keras.utils import plot_model
```

[41]:
```python
def build_CNN_model(filter_fk, avg_pool_size):
    ''' ACNN building function

    Arguments:
        filter_fk (List[Tuple]):
            list of (kernel size, number of filters)
            It is used to define convolutional neural networks
        avg_pool_size (List[int]):
            list of average pooling sizes

    Return:
        A CNN model
            without/with inceptron module if filter_fk contains only
                one/multiple pair(s) of (kernel size, number of filters)
            with/without average pooling if avg_pool_size isn't/is empty
    '''

    def build_sub_model(inputs, pool_size=None):
        ''' Submodel for different pool sizes '''
        if pool_size:
            x = AveragePooling1D(pool_size=pool_size,
                    strides=1)(inputs)
        else:
            x = inputs

        x = Reshape((x.shape[1], x.shape[2], 1))(x)

        feature_maps = [conv_pool(x, f, k) for k, f in filter_fk]
        if len(feature_maps) > 1:
            x = concatenate(feature_maps)
```

```
        else:
            x = feature_maps[0]
        x = Flatten()(x)
        return x

    def conv_pool(x, filters, kernel_size):
        ''' Conv-layer + Max-pooling-layer '''
        x = Conv2D(
            filters=filters,
            kernel_size=(kernel_size, x.shape[2]),
            activation='relu')(x)
        x = Reshape((x.shape[1], x.shape[3]))(x)
        x = MaxPooling1D(pool_size=x.shape[1])(x)
        return x

    raw_inputs = Input((seq_length, n_features), name='raw_input') # Other␣
↪inputs
    order_embeddings = Embedding(2, 5)(raw_inputs[:, :, 0]) # Order type␣
↪embedding
    inputs = concatenate([order_embeddings, raw_inputs[:, :, 1:]])

    if avg_pool_size:
        sub_models = [build_sub_model(inputs, s) for s in avg_pool_size]
    else:
        sub_models = [build_sub_model(inputs)]

    if len(sub_models) > 1:
        x = concatenate([build_sub_model(inputs, s) for s in avg_pool_size])
    else:
        x = sub_models[0]

    y = Dense(2, activation='softmax')(x) # output probabilities

    model = Model(inputs=raw_inputs, outputs=y)
    return model
```

[29]:

```
def test_cnn(symbol):
    ''' Train and do grid search on CNN models

    Arguments:
        symbol (str):
            Abbreviation of stock
    Returns:
        None
    '''
    # (kernelSize, numOfFilters) for Conv2d
    filter_fk_lst = [
```

```
    [(3, 20)],
    [(5, 20)],
    [(7, 20)],
    [(3, 20), (5, 20)],
    [(5, 20), (7, 20)],
    [(3, 20), (5, 20), (7, 20)],
  ]

  # For multiple avg pools
  avg_pool_size_lst = [
    [],
    [5],
    [10],
    [5, 10],
  ]
  cnn_accs, cnn_val_accs = [], []
  cnn_test_accs, cnn_durs = [], []
  for filter_fk in filter_fk_lst:
      for avg_pool_size in avg_pool_size_lst:
          print(f'{symbol} params | filter_fk:', filter_fk, 'p_size:',␣
↪avg_pool_size)
          cnn_model = build_CNN_model(filter_fk, avg_pool_size)
          cnn_model.compile(optimizer='adam',␣
↪loss='categorical_crossentropy', metrics=['accuracy'])
          start = datetime.now()
          history = cnn_model.fit(X_train, y_train, epochs=20,␣
↪validation_data=(X_val, y_val), verbose=0)
          dur = (datetime.now()-start).seconds
          loss, acc = cnn_model.evaluate(X_test, y_test, verbose=0)
          cnn_test_accs.append(acc)
          cnn_accs.append(history.history['accuracy'])
          cnn_val_accs.append(history.history['val_accuracy'])
          print(f'dur: {dur}s')
          cnn_durs.append(dur)

  # Save the train/val accuracy and training time
  np.array(cnn_accs).dump(os.path.join(path, 'result', f'{symbol}_cnn_accs.
↪npy'))
  np.array(cnn_val_accs).dump(os.path.join(path, 'result',␣
↪f'{symbol}_cnn_val_accs.npy'))
  np.array(cnn_test_accs).dump(os.path.join(path, 'result',␣
↪f'{symbol}_cnn_test_accs.npy'))
  np.array(cnn_durs).dump(os.path.join(path, 'result', f'{symbol}_cnn_durs.
↪npy'))
```

## 3.2 LSTM

Same as in CNN, we combine the model building process for all the LSTM models in one function.

[30]:
```
import tensorflow as tf
from datetime import datetime

from tensorflow.keras.models import Sequential
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import AveragePooling1D
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import concatenate
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import LSTM

from tensorflow.keras.utils import plot_model
```

[31]:
```
def build_LSTM_model(avg_pool_size=None, n_lstm=1):
    ''' LSTM models building function

    Arguments:
        avg_pool_size (int):
            average pooling size

        n_lstm (int):
            number of LSTM layers

    Return:
        A LSTM model
            without/with average pooling if avg_pool_size is/isn't None
            with n_lstm LSTM layers
    '''

    raw_inputs = Input((seq_length, n_features)) # Other inputs
    order_embeddings = Embedding(2, 5)(raw_inputs[:, :, 0]) # Order type␣
↪embedding
    inputs = concatenate([order_embeddings, raw_inputs[:, :, 1:]])

    if avg_pool_size:
        x = AveragePooling1D(pool_size=5, strides=1)(inputs)
    else:
        x = inputs

    for i in range(n_lstm-1):
        # multiple LSTM layers
```

```
        x = LSTM(units=64, return_sequences=True)(x)
        x = Dropout(0.2)(x)
        x = BatchNormalization()(x)

    x = LSTM(units=64)(x)
    x = Dropout(0.1)(x)
    x = BatchNormalization()(x)
    y = Dense(units=2, activation='softmax')(x)

    model = Model(inputs=raw_inputs, outputs=y)
    return model
```

[32]:
```
def test_lstm(symbol):
    ''' Train and do grid search on LSTM models

    Arguments:
        symbol (str):
            Abbreviation of stock
    Returns:
        None
    '''

    avg_pool_size_lst = [0, 5, 10]
    n_lstm_lst = [1, 2]

    lstm_accs, lstm_val_accs = [], []
    lstm_test_accs, lstm_durs = [], []
    for n_lstm in n_lstm_lst:
        for avg_pool_size in avg_pool_size_lst:
            print(f'{symbol} params | nlstm:', n_lstm, ', p_size:',␣
↪avg_pool_size)

            # For multiple avg pools
            lstm_model = build_LSTM_model(avg_pool_size=avg_pool_size,␣
↪n_lstm=n_lstm)
            lstm_model.compile(optimizer='adam',␣
↪loss='categorical_crossentropy', metrics=['accuracy'])

            start = datetime.now()
            history = lstm_model.fit(X_train, y_train, epochs=20,␣
↪validation_data=(X_val, y_val), verbose=0)
            dur = (datetime.now() - start).seconds
            lstm_durs.append(dur)
            loss, acc = lstm_model.evaluate(X_test, y_test, verbose=0)
            lstm_test_accs.append(acc)
            lstm_accs.append(history.history['accuracy'])
            lstm_val_accs.append(history.history['val_accuracy'])
```

```
            print(f'dur: {dur}s')

    # Save the train/val accuracy and training time
    np.array(lstm_accs).dump(os.path.join(path, 'result', f'{symbol}_lstm_accs.
↪npy'))
    np.array(lstm_val_accs).dump(os.path.join(path, 'result',␣
↪f'{symbol}_lstm_val_accs.npy'))
    np.array(lstm_test_accs).dump(os.path.join(path, 'result',␣
↪f'{symbol}_lstm_test_accs.npy'))
    np.array(lstm_durs).dump(os.path.join(path, 'result', f'{symbol}_lstm_durs.
↪npy'))
```

# 4 Training and Testing

We selected 4 stocks to train and test our models.

[ ]:
```
symbols = ['APD', 'CDE', 'FNB', 'BDX']

path = './data'
for symbol in symbols:
    X_train, X_val, X_test, y_train, y_val, y_test = data_processing(path,␣
↪symbol)
    _, seq_length, n_features = X_train.shape
    test_cnn(symbol)
    test_lstm(symbol)
```

# 5 Plot

## 5.1 Plot of our most complicated model CNN

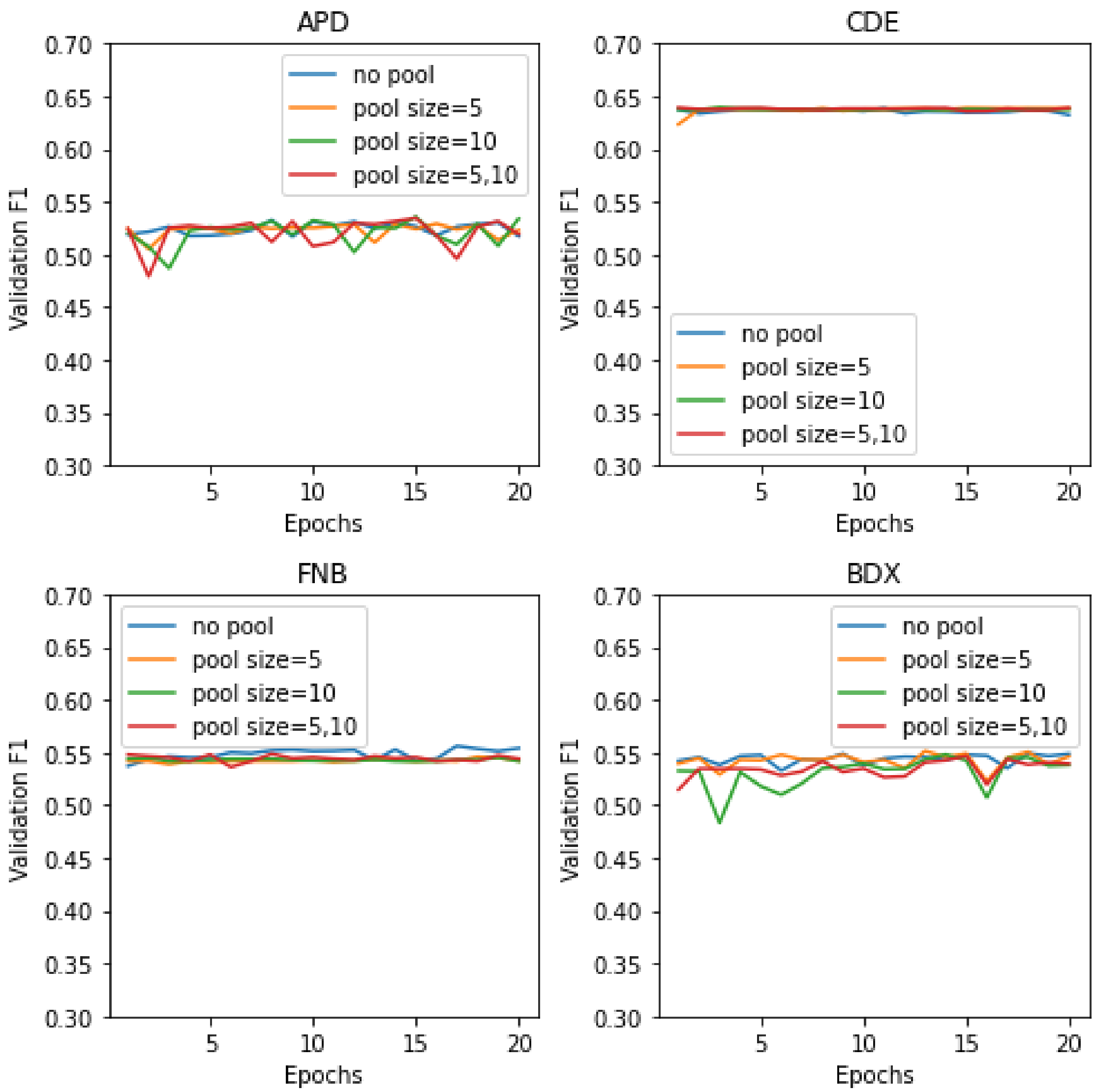
[43]:
```
filter_fk = [(3, 20), (5, 20), (7, 20)]

# For multiple avg pools
avg_pool_size = [5, 10]

model = build_CNN_model(filter_fk, avg_pool_size)
plot_model(model)
```

[43]:


## 5.2 Plot of CNN validation accuracies with different pool size

Here we show the accuracies of models with different pool size.

[70]:
```python
symbols = ['APD', 'CDE', 'FNB', 'BDX']
plt.figure(figsize=(7, 7))
for i, symbol in enumerate(symbols):
    plt.subplot(2, 2, i+1)
    x = np.arange(1, 21)
    accs = np.load(os.path.join(path, 'result', f'{symbol}_cnn_val_accs.npy'),
 ↪allow_pickle=True)
    plt.plot(x, accs[0], label='no pool')
    plt.plot(x, accs[1], label='pool size=5')
    plt.plot(x, accs[2], label='pool size=10')
    plt.plot(x, accs[3], label='pool size=5,10')
    plt.title(f'{symbol}')
    plt.ylim(0.3, 0.7)
    plt.xlabel('Epochs')
    plt.ylabel('Validation F1')
    plt.legend()
plt.tight_layout()
```



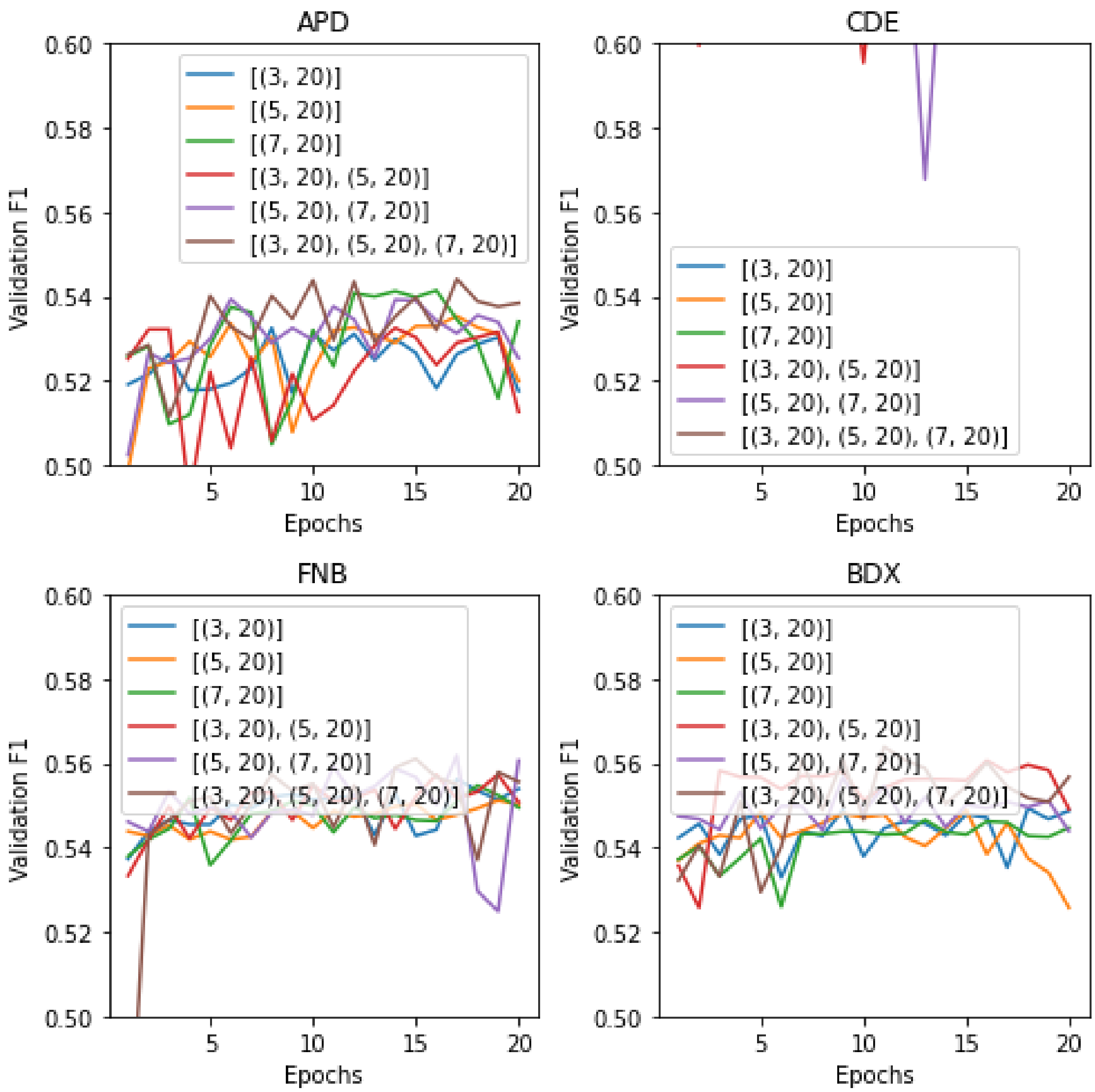

### 5.2.1 Conclusion

The pooling does not give our model a good improvement. Perhaps because in our dataset, the order market data has only high frequency. Pooling operation, which lower the data resolution, is not good at dealing with high resolution data.

## 5.3 Plot of CNN validation accuracies with/without inception module

[78]:
```
symbols = ['APD', 'CDE', 'FNB', 'BDX']

plt.figure(figsize=(7, 7))
filter_fk_lst = [
        [(3, 20)],
```

```
      [(5, 20)],
      [(7, 20)],
      [(3, 20), (5, 20)],
      [(5, 20), (7, 20)],
      [(3, 20), (5, 20), (7, 20)],
]
for i, symbol in enumerate(symbols):
    plt.subplot(2, 2, i+1)
    x = np.arange(1, 21)
    accs = np.load(os.path.join(path, 'result', f'{symbol}_cnn_val_accs.npy'),␣
 ↪allow_pickle=True)
    for j in range(6):
        plt.plot(x, accs[j*4], label=str(filter_fk_lst[j]))
    plt.title(f'{symbol}')
    plt.ylim(0.5, 0.6)
    plt.xlabel('Epochs')
    plt.ylabel('Validation F1')
    plt.legend()
plt.tight_layout()
```



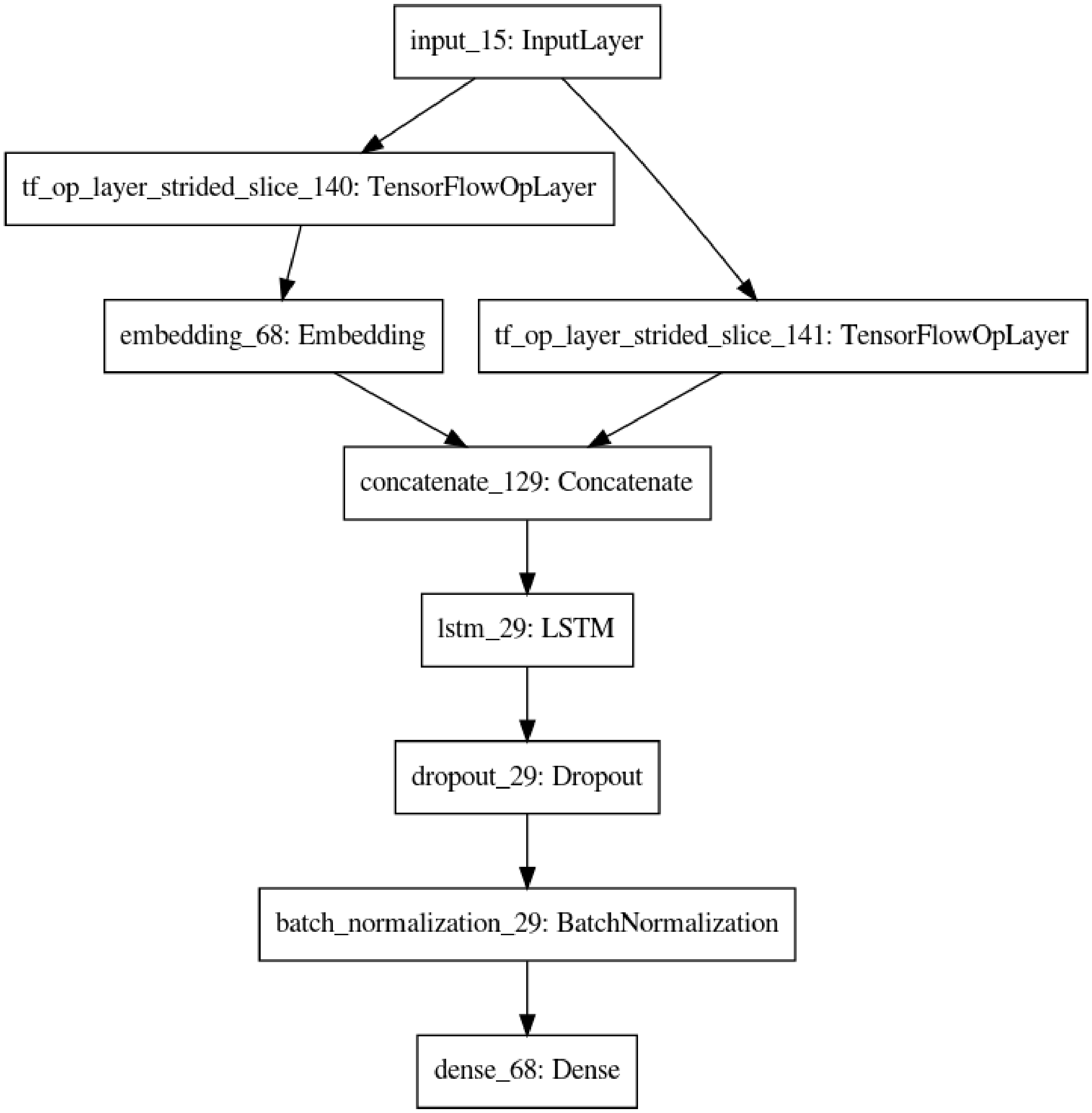

#### 5.3.1 Conclusion

Inception module does not help because our dataset only contains high accuracy data, which prefers smaller kernels. As you might have noticed that the model with smaller kernels are better than the ones with larger kernels.

### 5.4 Plot of LSTM model

[75]:
```
avg_pool_size_lst = [0, 5, 10]
n_lstm_lst = [1, 2]

model = build_LSTM_model(avg_pool_size_lst[0], 1)
plot_model(model)
```

[75]:



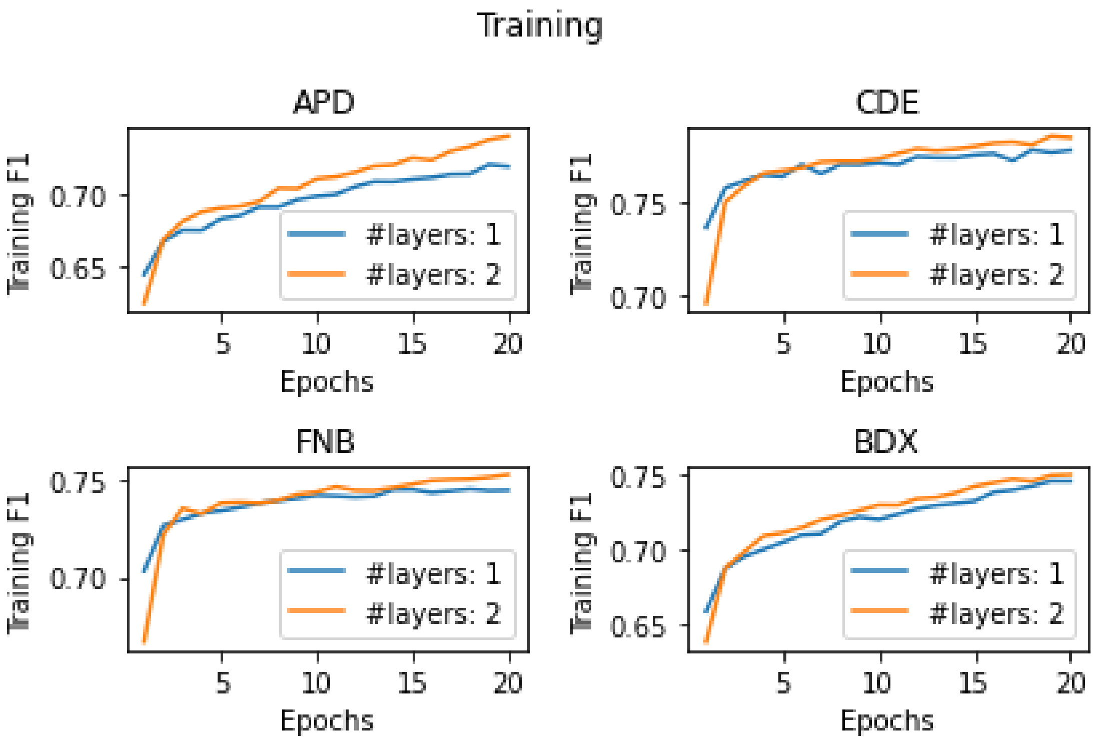

## 5.5 Plot of accuracies of LSTM models

Here we plot the LSTM model accuracies with different layers

### 5.5.1 Training accuracy

[85]:
```
symbols = ['APD', 'CDE', 'FNB', 'BDX']

plt.suptitle('Training')
for i, symbol in enumerate(symbols):
    plt.subplot(2, 2, i+1)
    x = np.arange(1, 21)
```

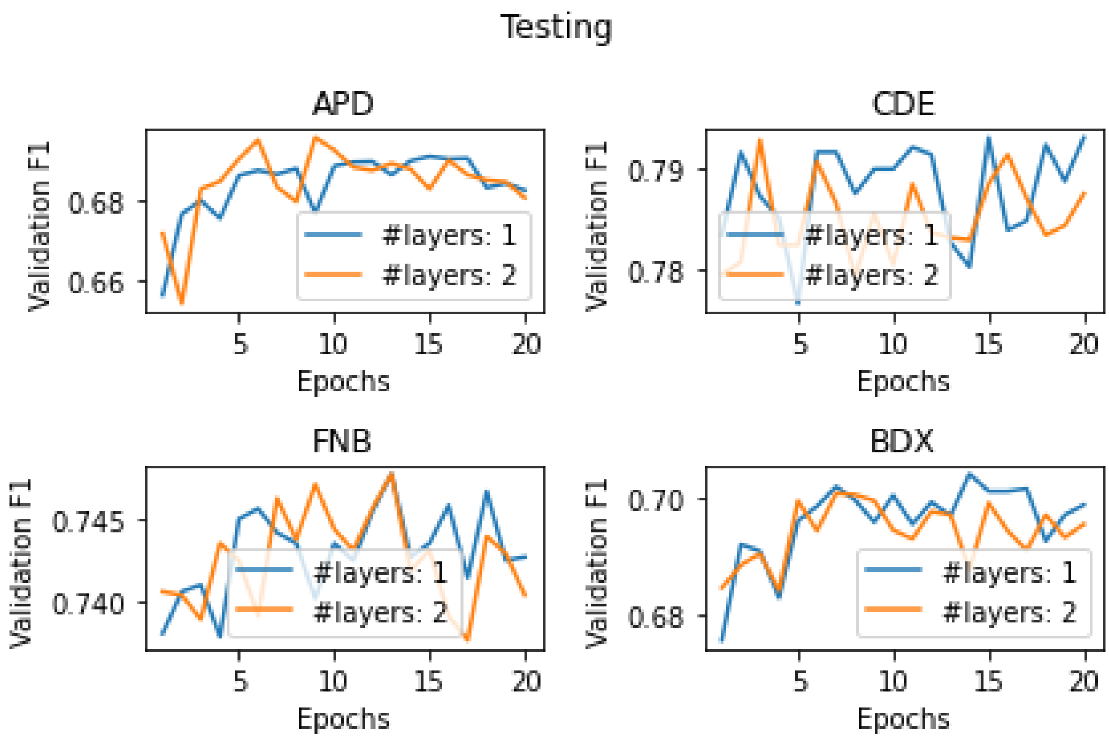
```
    accs = np.load(os.path.join(path, 'result', f'{symbol}_lstm_accs.npy'),␣
↪allow_pickle=True)
    for j in range(2):
        plt.plot(x, accs[j*3], label=f'#layers: {n_lstm_lst[j]}')
    plt.title(f'{symbol}')
#     plt.ylim(0.3, 0.7)
    plt.xlabel('Epochs')
    plt.ylabel('Training F1')
    plt.legend()
plt.tight_layout()
```




### 5.5.2 Testing accuracy

[84]:
```
symbols = ['APD', 'CDE', 'FNB', 'BDX']

plt.suptitle('Testing')
for i, symbol in enumerate(symbols):
    plt.subplot(2, 2, i+1)
    x = np.arange(1, 21)
    accs = np.load(os.path.join(path, 'result', f'{symbol}_lstm_val_accs.npy'),␣
↪allow_pickle=True)
    for j in range(2):
```

```
        plt.plot(x, accs[j*3], label=f'#layers: {n_lstm_lst[j]}')
    plt.title(f'{symbol}')
#     plt.ylim(0.3, 0.7)
    plt.xlabel('Epochs')
    plt.ylabel('Validation F1')
    plt.legend()
plt.tight_layout()
```




### 5.5.3 Conclusion

LSTM models are much better than CNN models. During training, it shows an elegant training F1 score curve, in which the model with 2 layers is more accurate than the model with a single layer. It is clearly because the model with 2 layers has a higher capacity so it can better fit the training data. If we look at the validation F1 score, it looks like the LSTM model with more layers are easier to be overfitted. Its validation F1 score keeps oscillating around a certain number without any improvement.

The 1-layer LSTM model is better because it achieves similar accuracy as 2-layer model and it is trained 30-40 seconds faster in 20 epochs on average.

# 6 Conclusion

By conducting experiment on our dataset, we got the following conclusion: * CNN model: * Achieves around 60% F1 score * Average pooling does not work * Small-kernel models are better * Parallel-kernel models gives performance comparable to the small-kernel kernels, but is overly complicated * LSTM model: * Achieves 70-80% F1 score, much better than CNN models * Average pooling does not work * 2-layer model is overfitted and slower than 1-layer model * LSTM model is slower than CNN models

```
[ ]:
```