TraderNet

TraderNet was created as a final project for CS6350 - big data. It attempts to predict future stock prices either through classification or regression. It incorporates big data technologies like Spark to preprocess a large number of historical stock price / volume records into training examples that are then fed to a deep neural network architecture implemented in Tensorflow.

Preprocessing

Dataset

The input dataset consists of approximately 8000 CSV files, each of which contains historical price records for a stock symbol from 1970 to 2018. Each record consists of the following columns:

```
date, high, low, open, close, volume, adjclose
```

The high, low, open, and close prices are historical prices as recorded on the day of the record. The `adjclose` column provides an adjusted close price that accounts for events like splits, which retroactively affect historical price data.

For example, if a company doubles its number of existing shares, the price per share will be reduced by half. The adjusted close price accounts for these changes, providing a shared scale on which to measure closing price over the history of the stock.

Filtering

Several filtering techniques can be applied to the input dataset in order to produce higher quality training examples. One such technique is the ability to remove records older than a parameterized year. Since the dataset contains records dating back to 1970, the ability to constrain to more recent records may prove useful in capturing market behavior under modern trading styles.

Another filtering technique is the penny stocks, which tend to exhibit greater volatility due to their low price per share. This is accomplished through the use of a parameterized closing price threshold. If the average closing price of a stock over the constrained date window is below the parameterized threshold, that stock will be excluded entirely from the training example set.

Positional Encoding

A positional encoding strategy was used to give the neural network information about what day of year a record. The underlying assumption is that stocks may behave differently at different times of year (eg quarterly earnings reports, end of year, etc). The following function was used to calculate a positional encoding based on the day of year:

$$f(x) = sin\left(\frac{\pi x}{365}\right)$$

This function transforms a day of year into a real number on a continuous interval from $0, 1$. Under this transformation, day 365 and day 1 will have similar values, capturing the idea that December 31st and January 1st are close to each other. It is worth noting that each point under this transformation will be produced by two points in the original "day of year" space. This is not ideal, but the network should be able to compensate based on whether or not the point lies on an increasing or decreasing region of the sin function.

Price Adjustment

The close and adjusted close price given in the original dataset were used to rescale the other price metrics (high, low, open). This was accomplished by apply the following transformation to each unscaled price in the record:

$$scaled = unscaled \cdot \frac{adjclose}{close}$$

After this point the unscaled prices were discarded.

Future Price Calculation

A future percent change in closing price was calculated for each record by looking x days into the future and calculating a pointwise or aggregate measure of closing price over those x days.

One such strategy is to calculate a pointwise percent change in price as follows, where i indicates an index of the record to be labeled:

$$\Delta = \frac{P_{i+x} - P_i}{P_i} * 100$$

Another strategy is to calculate an aggregate of some price metric over x days in the future and calculate percent change from present to that aggregate. For example, one could calculate an average closing price over the following x days and then calculate percent change in closing price from the present to that average.

$$a = \frac{1}{x}\sum_{i}^{i+x} P_i$$
$$\Delta = \frac{a - P_i}{P_i} * 100$$

The decision to use a pointwise or aggregate basis for percent change calculation is dependent on the choice of window size among other things. The choice of window size is primarily determined by the style of trading to be used, along with the interval between records (if this approach were to be applied to by the minute data for day trading).

Spark ML Pipeline

A parameterized Spark ML pipeline is used to extract feature vectors and produce discretized labels for each record.

1. `VectorAssembler` - Collects feature columns into a single vector for processing
2. Normalizer - A strategy to normalize each feature to a shared domain. Can be one of the following, or excluded entirely:
    - `StandardScaler` - Rescale each feature to zero mean unit variance
    - `MaxAbsScaler` - Rescale each feature to the range $0, 1$, **preserving sparsity**
3. Labeler - A strategy to assign examples a discretized label based on future percent change. Can be one of the following, or excluded entirely:
    - `Bucketizer` - Label based on fixed sized buckets of percent change
    - `QuantileDiscretizer` - Label based on variable sized buckets such that each each class contains an equal number of examples

The each record in the output of this pipeline will have a vector of (normalized) features for that day, a future percent change value, and an integer label if requested. By retaining both the future percent change and a discretized label, each record is suitable for use with classification or regression networks.

It is important to consider the distribution of examples among the possible classes when using `Bucketizer` as a labeling strategy. It was observed that percent change tends to concentrate near zero, meaning that the bucket that overlaps with zero will have a potentially large number of examples. Conversely, when using `QuantileDiscretizer` a uniform distribution of examples over each class will be created by compressing or expanding bucket ranges. This will manifest as buckets that span only a few percentage points of percent change, taxing the network's ability to distinguish these classes.

Creating Training Examples

Having extracted numerically stable features and labels from the original dataset, steps must now be taken to construct complete training examples. Taking inspiration from human styles of trading, the decision was made to create training examples by aggregating chronologically ordered feature vectors over a sliding window into a feature matrix.

As a concrete example, consider training example i with features h, l, o, c, v, p and a window size of x. Example i would then consist of features over the x previous days for that stock as follows:

$$train_i = \begin{bmatrix} h_i & l_i & o_i & c_i & v_i & p_i \\ h_{i-1} & l_{i-1} & o_{i-1} & c_{i-1} & v_{i-1} & p_{i-1} \\ h_{i-2} & l_{i-2} & o_{i-2} & c_{i-2} & v_{i-2} & p_{i-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{i-x} & l_{i-x} & o_{i-x} & c_{i-x} & v_{i-x} & p_{i-x} \end{bmatrix}$$

The result of this process is an intuitively constructed training example, consisting of an ordered timeseries of features over the x previous days and a label indicating price movement in the near future. Implementation of this process is discussed in the following section.

We can take further steps to improve the quality of the resultant training examples. Consider the case where the process described above is applied to two adjacent records, i and $i+1$. The result will be two training examples with $x \times 6$ feature matrices which have a substantial overlap. Specifically, we will produce feature matrices as follows:

$$features_i \in \mathbb{R}^{x \times 6} = \begin{bmatrix} h_i & l_i & o_i & c_i & v_i & p_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{i-x} & l_{i-x} & o_{i-x} & c_{i-x} & v_{i-x} & p_{i-x} \end{bmatrix}$$

$$features_{i+1} \in \mathbb{R}^{x \times 6} = \begin{bmatrix} h_{i+1} & l_{i+1} & o_{i+1} & c_{i+1} & v_{i+1} & p_{i+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{i-x+1} & l_{i-x+1} & o_{i-x+1} & c_{i-x+1} & v_{i-x+1} & p_{i-x+1} \end{bmatrix}$$

Applying a stride greater than 1 to the historical window will reduce the amount of overlap between feature matrices, creating a more diverse training set.

Generating the feature matrices as described above is a computationally expensive process, especially as at small window strides and large window sizes. To put this in perspective, consider a training set produced from 1000 stock symbols from 2008 to 2018 with a window size of 128 days and a window stride of 1. Assume the stock only trades 200 days out of the year. The number of resultant examples is given by

$$\begin{aligned} N &= 1000 * 200 * (2018 - 2008) \\ &= 2,000,000 \end{aligned}$$

Similarly, if we consider this process applied the entirety of the dataset without constraint

$$
\begin{aligned}
N &= 8000 * 200 * (2018 - 1971) \\
&= 75,200,000
\end{aligned}
$$

We will have a 128×6 feature matrix per row. If we assume that each feature and the percent change is stored as a 32 bit float and the label is stored as an 8 bit integer, we can calculated the expected size of the resultant training set in

$$
\begin{aligned}
features &= N\,(128 * 6 * 4) \\
labels &= N\,(4 + 1) \\
S &= labels + features \\
&\approx 230GB
\end{aligned}
$$

Interfacing with Tensorflow

The above illustrates the need for careful consideration on how Tensorflow should interface with Spark in order to handle the data volume. Several strategies were considered:

~~Connect Tensorflow's computational graph to Spark~~

Some libraries do exist which promise to connect Tensorflow and Spark at the graph level, allowing for traning of a Tensorflow model in Spark. This approach was ultimately not used, as it would require sacrificing many of Tensorflow's high level features and presented compatibility issues with Tensorflow 2.0. It would however avoid the need to write the entire training set to disk prior to training and would expedite the process of iteratively refining preprocessing techniques based on model performance.

Write training examples to TFRecords

Tensorflow uses `TFRecords` as a format for storing training examples in a serialized form. A dataset can be split in to multiple TFRecord files, enabling distributed training. Through the use of spark-tensorflow-connector it is possible to generate TFRecords from a Spark dataframe, where the number of TFRecord files is determined by the partitioning of the dataframe. Initial efforts aimed to partition the records by stock symbol, order them by date, and write each partition to a separate TFRecord file. Under this strategy, Tensorflow would be responsible for constructing training examples using the feature matrix generation procedure described earlier.

By allowing Tensorflow to handle feature generation the size of the TFRecord dataset will be reduced by a factor of the historical window size. Unfortunately, this technique has the following requirements:

1. Strict partitioning by symbol to separate TFRecord files

2. Strict ordering by record date within TFRecord files
3. A complicated process to interleave examples from each file

According to this issue, spark-tensorflow-connector does not currently respect the requested partitioning by column, making requirement 1 unsatisfiable. This necessitated the use of spark for feature matrix generation.

The training examples written to TFRecords by spark are complete examples, containing a feature matrix, percent change value, and discretized label (if requested). Tensorflow is able to deserialize these examples with little additional processing.

The following steps are performed in Tensorflow to finalize the dataset for training:

1. Create `(feature, label)` tuples for each example where the label is chosen based on the current mode (regression or classification)
2. Split the dataset into training and validation sets based on a parameterized validation set size.
3. Batch training and validation sets into a parameterized batch size. A batch size of 128 was used in an effort to improve tiling efficiency when training on a GPU
4. Final modifications like `repeat()` and `prefetch()`

Model

TraderNet is based on a dominant object detection network I have been implementing here. TraderNet reduces the original 2D convolutional encoder to a 1D residual encoder, with the addition of a multi-head attention layer at the head to better distinguish features based on their sequential relationship.

Model Architecture

A summary of the model generated by Tensorflow is shown below. By default there are four levels of downsampling with $3, 3, 5, 3$ bottleneck block repeats respectively. During experimentation, it was found that a $4, 6, 10, 6$ level repeat structure was more robust, and is shown below:

```
Model: "trader_net"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
tail (Tail)                  (None, 128, 32)           672
_________________________________________________________________
bottleneck (Bottleneck)      (None, 128, 32)           696
_________________________________________________________________
bottleneck_1 (Bottleneck)    (None, 128, 32)           696
_________________________________________________________________
bottleneck_2 (Bottleneck)    (None, 128, 32)           696
```

bottleneck_3 (Bottleneck)	(None, 128, 32)	696
downsample (Downsample)	(None, 64, 64)	8048
bottleneck_4 (Bottleneck)	(None, 64, 64)	2416
bottleneck_5 (Bottleneck)	(None, 64, 64)	2416
bottleneck_6 (Bottleneck)	(None, 64, 64)	2416
bottleneck_7 (Bottleneck)	(None, 64, 64)	2416
bottleneck_8 (Bottleneck)	(None, 64, 64)	2416
bottleneck_9 (Bottleneck)	(None, 64, 64)	2416
bottleneck_10 (Bottleneck)	(None, 64, 64)	2416
bottleneck_11 (Bottleneck)	(None, 64, 64)	2416
bottleneck_12 (Bottleneck)	(None, 64, 64)	2416
bottleneck_13 (Bottleneck)	(None, 64, 64)	2416
downsample_1 (Downsample)	(None, 32, 128)	31456
bottleneck_14 (Bottleneck)	(None, 32, 128)	8928
bottleneck_15 (Bottleneck)	(None, 32, 128)	8928
bottleneck_16 (Bottleneck)	(None, 32, 128)	8928
bottleneck_17 (Bottleneck)	(None, 32, 128)	8928
bottleneck_18 (Bottleneck)	(None, 32, 128)	8928
bottleneck_19 (Bottleneck)	(None, 32, 128)	8928
downsample_2 (Downsample)	(None, 16, 256)	124352
bottleneck_20 (Bottleneck)	(None, 16, 256)	34240
bottleneck_21 (Bottleneck)	(None, 16, 256)	34240
bottleneck_22 (Bottleneck)	(None, 16, 256)	34240

```
_________________________________________________________________
bottleneck_23 (Bottleneck)   (None, 16, 256)           34240
_________________________________________________________________
downsample_3 (Downsample)    (None, 8, 512)            494464
_________________________________________________________________
bn (BatchNormalization)      (None, 8, 512)            2048
_________________________________________________________________
relu (ReLU)                  (None, 8, 512)            0
_________________________________________________________________
attn (MultiHeadAttention)    (None, None, 512)         1051649
_________________________________________________________________
head (ClassificationHead)    (None, 5)                 2565
=================================================================
Total params: 1,932,726
Trainable params: 1,922,902
Non-trainable params: 9,824
_________________________________________________________________
```

Loss / Metrics

When operating in classification mode, the model uses a sparse softmax cross-entropy loss. Sparse categorical accuracy is continuously tracked for the training set and is reported once per epoch for the validation set. Depending on the number of discretized classes it may be helpful to add a top k categorical accuracy in the case of closely related classes. Precision and recall are also tracked, but should only be emphasized for benchmarking when there is a non-uniform distribution of examples across the classes.

In regression mode the model uses mean squared error as a loss function and is tracked with the same frequency as the classification pipeline.

Hyperparameters

Key tunable hyperparameters include the following:

1. **Dropout** - A dropout layer can be added after the attention layer to make the model more robust. This layer was excluded from the final project submission, as it would be impractical to train the model to convergence in the time alloted to complete the assignment.
2. **Batch size** - Tune this as needed for your hardware. It is probably wise to use a power of two batch size in order to improve tiling efficiency.
3. **Learning rate** - Learning rate can be tuned directly, with values around 0.005 tending to work well. A learning rate greater than 0.1 often hindered the training process, likely due to the lack of a robust weight initialization strategy. Learning rate is reduced in a stepwise manner using a

parameterized Tensorflow callback based on the change in loss between epochs.

4. **Regularization** - Some experiments were conducted with regularization of the dense layers at the network head, but there appeared to be no discernible impact on model performance. This may be more significant when training on a small subset of the original dataset, such as a single stock or a narrow range of dates.

Monitoring

Tensorboard callbacks are used to provide an interface for monitoring the training process and hyperparameter tuning. Use of Tensorboard is discussed in the usage section below. Model weights are checkpointed once per epoch and can be restored when entering a new training session.

Usage

There are two separate usage procedures, one for the Scala JAR file that handles TFRecord production, and one for the Python Tensorflow training pipeline.

Spark

The spark directory contains a `sbt` project that can be used to package a fat JAR file. The fat JAR file contains all dependencies (like spark-tensorflow-connector) but can become large. Build the fat JAR locally using `sbt assembly`.

The program provides usage instructions via the `--help` flag as follows:

```
trader 1.0
Usage: trader [options] <shell_glob>

  -o, --out <path>         output file path
  -n, --norm std, maxabs   Normalization strategy, StandardScaler or MaxAbsScaler
  -d, --date <year>        limit to records newer than <year>
  -p, --past <int>         aggregate past <int> records into an example
  --stride <int>           stride training example window by <int> days
  -f, --future <int>       calculate percent change over <int> following days. if >1, use av
  --max-change <float>     drop examples with absolute percent change > <float>
  -s, --shards <int>       split the TFRecord set into <int> shards
  --quantize <int>         run QuantileDiscretizer to <int> buckets
  --bucketize <float>,[float,...]
                           run Bucketizer with the given buckets
  --penny-stocks           if set, allow penny stocks. default false
  --help                   prints this usage text
Note: TFRecords will be written if output file path is specified

Writing many TFRecords has a high RAM requirement.
```

```
  <shell_glob>              input file shell glob pattern
```

If you run the program without specifying an output file path, no TFRecords will be produced, but dataframes will still be printed that describe the processed output. **Note* that the process of creating TFRecords can have significant CPU, memory, and disk requirements depending on the combination of flags given. The flags typically used would use approximately 40GB of memory and the resultant TFRecords would occupy 10G of disk.

The `<shell_glob>` argument should be path or wildcard containing path (eg `path/to/dir/*.csv`). This string will be fed directly to Spark's `DataFrameReader.csv()`.

For simple testing the most effective strategy is to choose a shell glob pattern that will select only a small number of CSV files, eg `AAPL.csv` or `AAP*.csv`. Please see the output directory for sample commands and output.

Tensorflow

TraderNet is packaged as a Docker container to effectively manage dependencies on Tensorflow 2.0 and Tensorboard. A docker-compose file is provided to simplify the build process, and allows for selection of the Tensorflow base image tag (gpu or no gpu). Edit the docker-compose `upstream` arg to pick the desired Tensorflow container tag. You can download Tensorflow 2.0 and run the model outside of Docker if desired.

A `.env` file is provided that sets environment variables needed when starting the container with docker-compose. These variables include paths to the TFRecord source directory and log / checkpoint output directory:

```
SRC_DIR=/path/to/tfrecords
ARTIFACT_DIR=/path/to/logs
```

Set these variables as desired or override them directly in the docker-compose file.

Next, start the container with

```
docker-compose up -d trader
```

You should be able to visit `localhost:6006` to access Tensorboard. Once the container is running, start the training pipeline using

```
docker exec -it trader python /app/train.py [options]
```

The `train.py` script accepts several command line flags that parameterize the training process and can be accessed using the `--helpfull` flag.

```
docker exec -it trader python /app/train.py --helpfull
       USAGE: /app/train.py [flags]
```

```
  ...

  --artifacts_dir: Destination directory for checkpoints / Tensorboard logs
  (default: '/artifacts')
--[no]attention: If true, include a multi-head attention layer
  (default: 'true')
--batch_size: Batch size for training
  (default: '256')
  (an integer)
--checkpoint_fmt: Format to use when writing checkpoints
  (default: 'trader_{epoch:02d}.hdf5')
--checkpoint_freq: Checkpoint frequency passed to tf.keras.callbacks.ModelCheckpoint
  (default: 'epoch')
--classes: Number of output classes if running in classification mode.
  (default: '3')
  (an integer)
--[no]dry: If true, dont write Tensorboard logs or checkpoint files
  (default: 'false')
--epochs: Number of training epochs
  (default: '100')
  (an integer)
--features: Features to use in the training pipeline
  (default: 'position,volume,close')
  (a comma separated list)
--glob: Shell glob pattern for TFRecord file matching
  (default: 'part-r-*')
--label: TFRecord element to treat as training label
  (default: 'label')
--levels: Levels to use in the TraderNet encoder architecture.
  (default: '3,3,5,2')
  (a comma separated list)
--lr: Learning rate
  (default: '0.001')
  (a number)
--mode: Set to classification or regression
  (default: 'classification')
--past: Size of the historical window
  (default: '128')
  (an integer)
--[no]prefetch: Whether to prefetch TF dataset
  (default: 'true')
--[no]repeat: Repeat the input dataset
  (default: 'true')
--resume: Resume from the specified model checkpoint filepath
--[no]resume_last: Attempt to resume from the most recent checkpoint
--shuffle_size: Size of the shuffle buffer. If 0, do not shuffle input data.
```

```
  (default: '1024')
  (an integer)
--[no]speedrun: If true, run a small epoch and evaluate the model
  (default: 'false')
--src: Dataset source directory
  (default: '/data/tfrecords')
--steps_per_epoch: Number of batches to include per epoch
  (default: '4000')
  (an integer)
--[no]summary: Print a model summary and exit
  (default: 'false')
--tb_update_freq: Update frequency passed to tf.keras.callbacks.TensorBoard
  (default: 'epoch')
--[no]tune: Run one epoch for each hyperparam setting and exit
  (default: 'false')
--validation_size: Number of examples to include in the validation set
  (default: '10000')
  (an integer)
--weighted: A list of weights to use for each class.must have len(weighted) == num classes
  (a comma separated list)
```

Notable flags include:

1. `--classes` - **Must** be set to the number of labels in the TFRecord dataset
2. `--summary` - Prints the network summary and exits
3. `--src, --dest` - Defaults to Docker volume mounts from docker compose
4. `--mode` - Toggle from default `classification` mode to `regression` mode
5. `--speedrun` - Runs a quick demo to show that the pipeline is operating correctly. Runs through very small epochs, printing model output for a small number of training examples. This can be useful when combined with the `--resume` flag to evaluate a trained model given a saved model weights file.
6. `--artifacts_dir` - Directory where Tensorboard logs, model checkpoints, etc will be written. Defaults to a Docker volume
7. `--resume` - Filepath for a model weights file. The model will be initialized with these weights and will resume training.
8. `--resume_last` - Attempts to automatically detect the last model checkpoint file and resume training from the checkpointed weights.

Results

The model's classification performance was ultimately benchmarked using both categorical accuracy and top 2 categorical accuracy. Due to the model's com-

plexity, training to convergence was impractical in the timeframe alloted for the assignment, but the model still performed reasonably well. The plots below show sparse categorical and top k accuracy for several (interrupted) runs. Validation top 1 categorical accuracy reached as high as 51%, while top 2 categorical accuracy reached 74%.

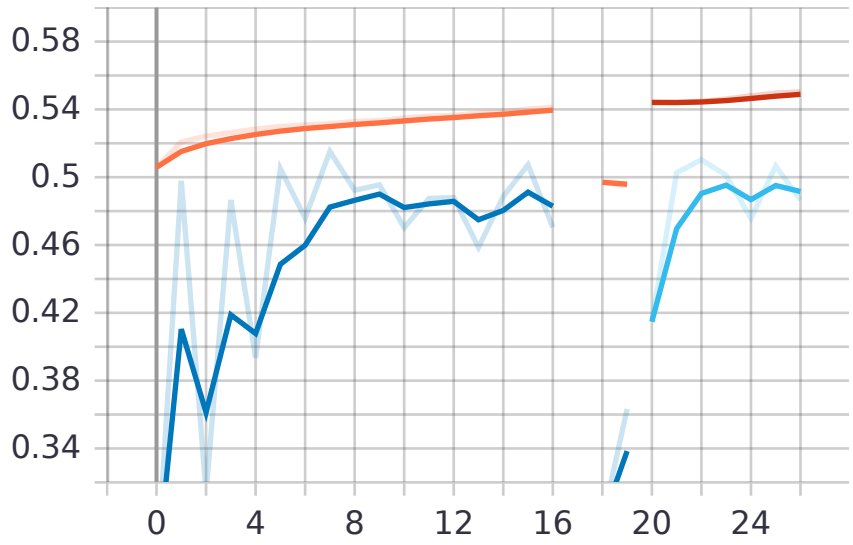

Figure 1: Top 1 Accuracy

In the near future I hope to repeat this experiment by training until convergence, perhaps using a cloud GPU provider. Though the accuracy reached by the model in the available training window is acceptable given the difficulty that humans have in predicting stock movement, this model would likely perform much better given adequate training time. Given that the number of potential training examples nears that of ImageNet (which may take month to train on with state of the art models), I suspect that the few hours of training alloted to this model across various hyperparameter turning runs was simply not sufficient.

A model weights file is provided in the root directory.

Future Work

There are several areas where the preprocessing and training pipeline can be improved. These include:

1. Add steps to produce metrics like exponential moving average to the Spark pipeline. This may allow the model to train faster by not forcing it to possibly learn a similar metric on its own.
2. Use Spark to enforce an equal distribution of examples over fixed bucket sizes. While the `QuantileDiscretizer` does produce an equal number of

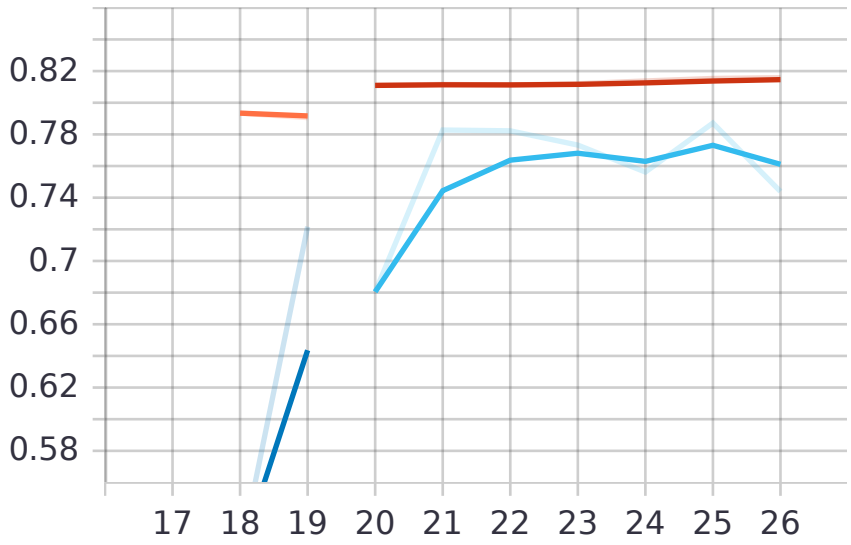

Figure 2: Top 2 Accuracy

labels per class, the bucket sizes become unacceptably close near zero percent change. The most promising resolution to unequal class distributions is to apply `Bucketizer` over fixed buckets and drop examples from highly represented classes.

3. Implement metrics that are more indicative of the model's performance. While categorical and top k categorical accuracy can provide some idea of the model's theoretical performance, these metrics are proxies for the most relevant metric: profitability. The model's performance in a real trading environment will be determined primarily by how well high-confidence predictions match future trends, as these predictions will form the basis for trading decisions. Future work could incorporate this model architecture into a reinforcement learning architecture, where network performance can be directly assessed on it's profitability