Group 2: Stock Market Forecasting and Playing

Koustubh Rao
Onkar Borade
Ashutosh Pankaj
Naman Bhandari

November 24, 2022

Contents

1 Abstract

The analysis of financial data represents a challenge that researchers have to deal with. The rethinking of the basis of financial markets has led to an urgent demand for developing innovative models to understand financial assets. In the past few decades, researchers have proposed several systems based on traditional approaches, such as autoregressive integrated moving average (ARIMA) and the exponential smoothing model, in order to devise an accurate data representation. Despite their efficacy, the existing works face some drawbacks due to poor performance when managing a large amount of data with intrinsic complexity, high dimensionality and casual dynamicity. Furthermore, these approaches are not suitable for understanding hidden relationships (dependencies) between data. This paper proposes a review of some of the most significant works providing an exhaustive overview of recent machine learning (ML) techniques in the field of quantitative finance.

Project submission for the course CS 335: Artificial Intelligence and Machine Learning Lab by the collaborative efforts of Koustubh Rao(200100176), Onkar Borade(200050022), Ashutosh Pankaj(200050015) and Naman Bhandari(22v0012) of CSE 2024 under the guidance of Professor Abir De.

2 Introduction

In this project we look into the two most important fields in Market Prediction, Market Forecasting and Real Time Playing Agent. We discuss various cutting edge Deep Neural Network (DNN) models used for stock prediction. We dig into the field of Recurrent Neural Networks (RNN) for this task as it is best suited for sequenced data. We observe the efficacy of the LSTMs and Seq2Seq models for timed sequence prediction. We finally note how Convolutional Neural Networks (CNN) can be equally useful for predicting timed sequences despite not being its forte, hence showing the true prowess of CNNs.

Next we dig into the field of Reinforcement Learning (RL) and Bayesian Statistics to build some real time stock agent trained on data and updated real time as the market sways. We will look at well known algorithms like policy gradient, Q-Learning, evolution strategy and actor-critic agent.

We will compare how the various models perform on average and also compare the Reinforcement Learning approach with the Neural Network approach. We use stock data from NASDAQ of Google, Twitter, Tesla and many more.

3 Market Forecasting

In Market Forecasting we train agents based on several well known Deep Neural Network models and then let them **forecast**. We set aside the last 30 days of every dataset to achieve the market environment. Due considerations are taken to separate this data from the training data. Accuracy is measured in terms of how close the prediction is with the real data using a RMS measure. A better architecture would produce a higher accuracy after 30 days.

3.1 Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this report, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

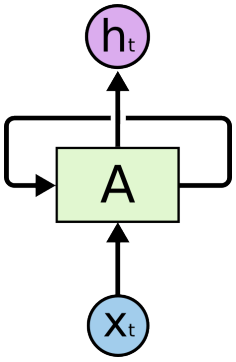

Figure 1: Recurrent Neuron

In the above diagram, a chunk of neural network, A, looks at some input xt and outputs a value ht. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

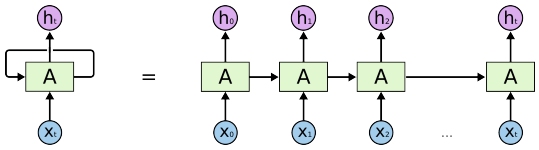

Figure 2: Recurrent Neuron Unrolled

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning... The list goes on. I'll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy's excellent blog post, The Unreasonable Effectiveness of Recurrent Neural Networks. But they really are pretty amazing.

3.1.1 The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.
Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the sky," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

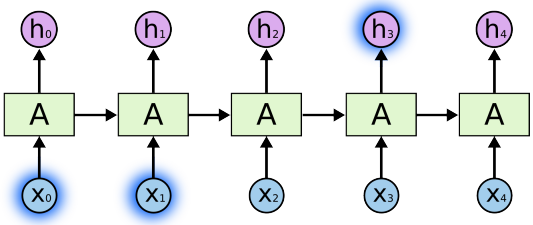

Figure 3: Short Term Dependency

But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

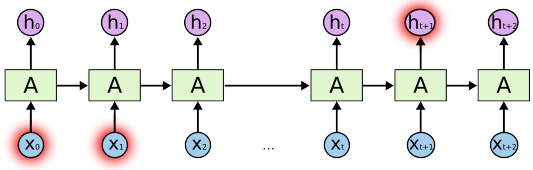

Figure 4: Long Term Dependency

In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

Thankfully, LSTMs don't have this problem!

3.2 LSTM DNN

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter and Schmidhuber (1997), and were refined and popularized by many people. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

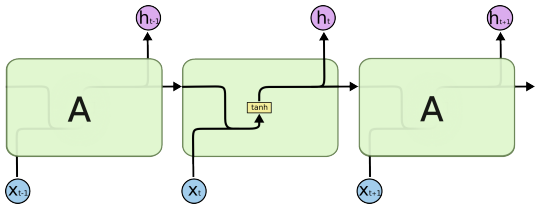

Figure 5: The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

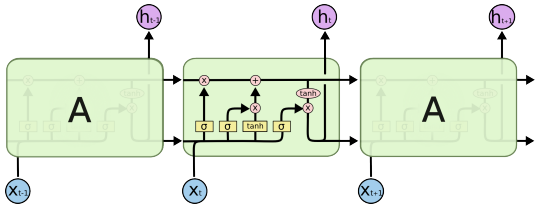

Figure 6: The repeating module in an LSTM contains four interacting layers.

Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using.

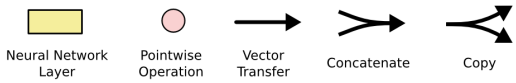

Figure 7: Notations.

In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!".

An LSTM has three of these gates, to protect and control the cell state.

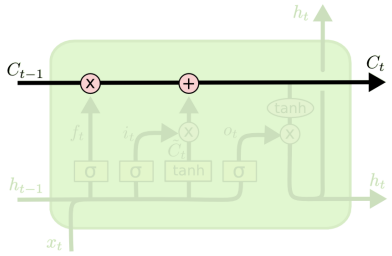

Figure 8: LSTM Working.

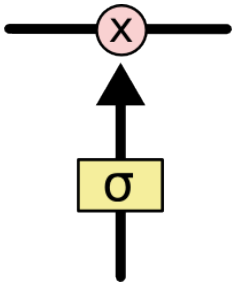

Figure 9: Gates.

3.2.1 Performance

Google:

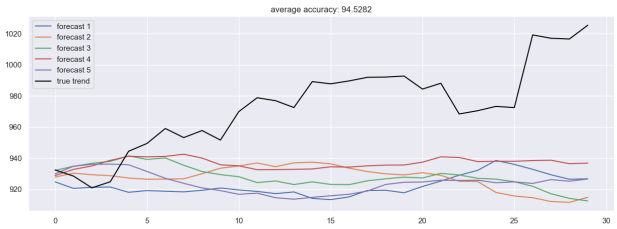

Figure 10: Google Performance on LSTM

TESLA:

Facebook:

3.3 Bidirectional LSTM

One shortcoming of conventional RNNs is that they are only able to make use of previous context. Bidirectional RNNs (BRNNs) do this by processing the data in both directions with two separate hidden layers, which are then fed forwards to the same output layer. Combining BRNNs with LSTM gives bidirectional LSTM, which can access long-range context in both input directions.

The basic idea of bidirectional recurrent neural nets is to present each training sequence forwards and backwards to two separate recurrent nets, both of which are connected to the same output layer. This means that for every point in a given sequence, the BRNN has complete, sequential information about all points

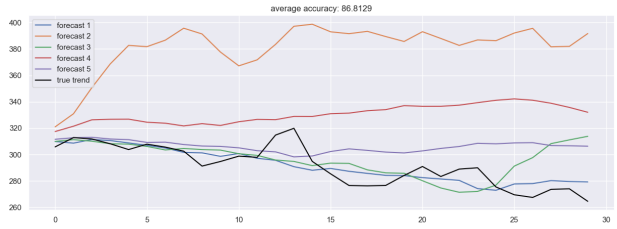

Figure 11: Tesla Performance on LSTM

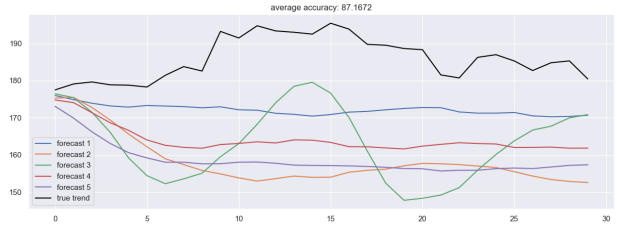

Figure 12: Facebook Performance on LSTM

before and after it. Also, because the net is free to use as much or as little of this context as necessary, there is no need to find a (task-dependent) time-window or target delay size.

For temporal problems like speech recognition, relying on knowledge of the future seems at first sight to violate causality. How can we base our understanding of what we've heard on something that hasn't been said yet? However, human listeners do exactly that. Sounds, words, and even whole sentences that at first mean nothing are found to make sense in the light of future context.

3.3.1 Performance

Google:

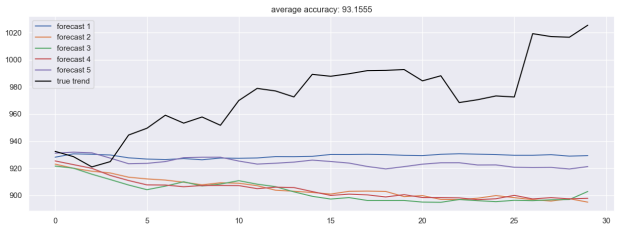

Figure 13: Google Performance on Bidirectional-LSTM

TESLA:

Facebook:

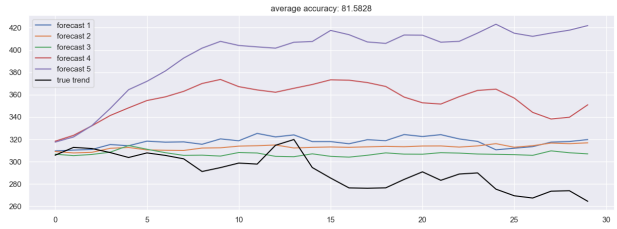

Figure 14: Tesla Performance on Bidirectional-LSTM

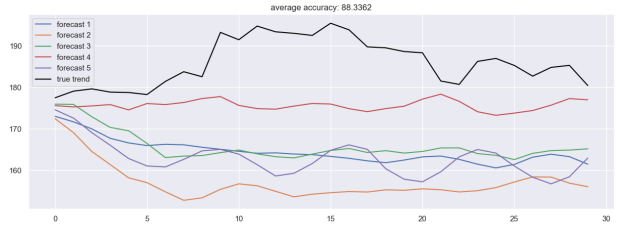

Figure 15: Facebook Performance on Bidirectional-LSTM

3.4 LSTM-Seq2Seq

The sequence-to-sequence LSTM, also called encoder-decoder LSTMs, are an application of LSTMs that are receiving a lot of attention given their impressive capability.

An "encoder" RNN reads the source sentence and transforms it into a rich fixed-length vector representation, which in turn in used as the initial hidden state of a "decoder" RNN that generates the target sentence. Here, we propose to follow this elegant recipe, replacing the encoder RNN by a deep convolution neural network (CNN). It is natural to use a CNN as an image "encoder", by first pre-training it for an image classification task and using the last hidden layer as an input to the RNN decoder that generates sentences.

The idea is to use one LSTM to read the input sequence, one timestep at a time, to obtain large fixed-dimensional vector representation, and then to use another LSTM to extract the output sequence from that vector. The second LSTM is essentially a recurrent neural network language model except that it is conditioned on the input sequence.

The LSTM's ability to successfully learn on data with long range temporal dependencies makes it a natural choice for this application due to the considerable time lag between the inputs and their corresponding outputs.

We were able to do well on long sentences because we reversed the order of words in the source sentence but not the target sentences in the training and test set. By doing so, we introduced many short term dependencies that made the optimization problem much simpler. The simple trick of reversing the words in the source sentence is one of the key technical contributions of this work.

3.4.1 Performance

Google:

TESLA:

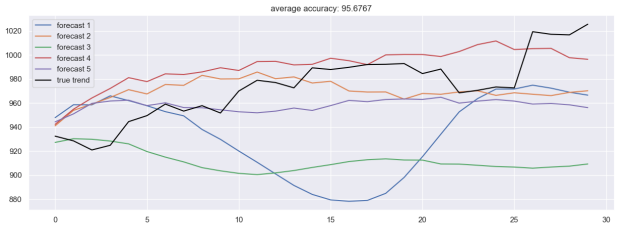

Figure 16: Google Performance on LSTM-Seq2Seq

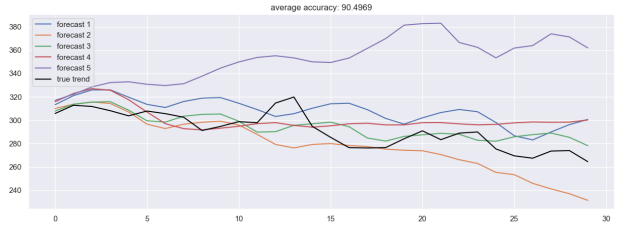

Figure 17: Tesla Performance on LSTM-Seq2Seq

Facebook:

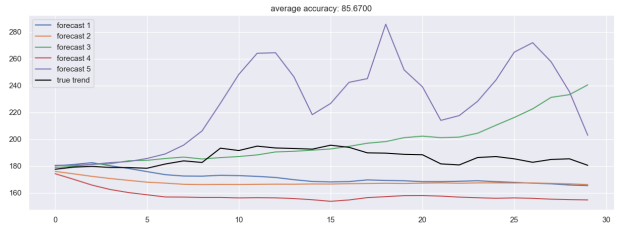

Figure 18: Facebook Performance on LSTM-Seq2Seq

3.5 CNN-Seq2Seq

Seq2seq model maps variable input sequence to variable length output sequence using encoder -decoder that is typically implemented as RNN/LSTM model. But this paper https://arxiv.org/pdf/1705.03122.pdf shows application of convolutional neural network for seq2seq learning which is state of the art for computer vision using deep learning.

There are many benefits that CNN has over RNN in terms of performance as number of non-linearity is fixed in CNN unlike RNN where it depends on input sequence. Also, during training computation can be parallelized in CNN to exploit GPU parallel computation whereas in RNN sequences are learned one input at a time in sequence. This paper introduces Gated Linear Units instead of Gated Recurrent Units which eases gradient propagation and equip each decoder layer with separate attention. As per author, they outperform in both WMT'14 English-German and English-French translation tasks of Wu et al (2016).

CNN vs RNN:

Seq2seq model implements stacked b-directional LSTM or GRU as encoder to generate a variable length output from encoder and feed into decoder which is also a stacked bi-directional RNN/LSTM. They interface using a soft attention mechanism. This architecture has long outperformed phrase-based models by large margin in many tasks such as machine translation and speech recognition.

In an advantage over RNN, CNN learns fixed length context representation and stacking multiple layers it can create larger context representation. This gives CNN control maximum length of dependencies to be modeled. In RNN this length will depend on input sequence and can get very large. Not being dependent on previous time step, CNN can parallelize entire learning process and can be very fast to train in contrast to RNN where current step depends on vector from previous step and need to learn in sequence.

CNN allows a hierarchical learning process where in lower layer nearby input sequence interacts and as we move downstream in the layers interactions of distant input sequences are learned.

It can be shown that CNN as hierarchical learning process takes O(n/k) operations where it has k layers for n input sequence whereas for RNN it would be O(n).

Inputs to CNN are fed through constant number of kernels and non-linearities, whereas RNN applies n operations and non-linearities to the first word and only a single set of operations to the last word. Fixed number of non-linearities in CNN expedites training process.

CNN Architecture for Seq2seq learning:

In the paper authors introduced complete CNN architecture for seq2seq learning. It applies Gated Linear Units (Dauphin et al., 2016) and residual connections (He et al., 2015a) which were already in place.

It also applies separate attentions to each decoder layer and demonstrate that each attention adds very little overhead.

The architecture is evaluated on several large datasets for translation tasks and compared with then current state of the art result. On WMT'16 English-Romanian translation we achieve a new state of the art, outperforming the previous best result by 1.9 BLEU. On WMT'14 English-German we outperform the strong LSTM setup of Wu et al. (2016) by 0.5 BLEU and on WMT'14 English-French we outperform the likelihood trained system of Wu et al. (2016) by 1.6 BLEU. Furthermore, our model can translate unseen sentences at an order of magnitude faster speed than Wu et al. (2016) on GPU and CPU hardware.

CNN is less commonly used for NLP problem in deep learning. Intuitively convolutional architecture is learning spatial features from raw data as 2-D or 3-D images are typically not the way natural language is perceived to be learned or make meaning of. For example, when we see a part of the sentence we may not make better inference for the rest of the sentence and arbitrary learning of parts of sentence may not result an accurate interpretation. Even CS244D lecture on NLP from Stanford has just one lecture on the use of CNN for NLP problems. CNN uses a fixed size filters over sentence vector (concatenated word vectors of the sentence) and learned n-gram feature from the sentence. Depending on the filter size chosen CNN can learn many different combination of n-grams and able to learns context features of the sentence. Normally multiple layers of CNN are stacked using different filter sizes to learn multiple n-grams. In this way, CNN are also learning in parallel and unlike RNN it doesn't depend on the previous time step. In RNN it needs to maintain hidden state from previous step and current input to compute current hidden state.

Multi layered CNN able to learn hierarchical feature with lower layer learning close range dependencies where higher layers learn long range interactions.

RNN Architecture:

Sequence2sequence learning is analogous to encoder-decoder model (Sutskever et al. 2014; Bahdanau et

al. 2014). The encoder process an input sequence X = (x1, x2, ...,xm) of m elements and returns state representation Z = (z1, z2, ...,zm). The decoder RNN takes Z as input and generates output sequence Y = (y1, y2, ...ym) left to right one element at a time. To generate output $y_i + 1$ it takes input as previous step hidden state h_i, an embedding g_i as well as conditional input c_i derived from the encoder output Z. Hence RNN is basically a conditional language model which is predicting next output in the sequence based on the hidden state from the encoder and previous.

Encoder-Decoder model:

Encoder- decoder architecture without attention uses a fixed vector as output from encoder Z as single input whereas encoder-decoder with attention generates a weighted sum of all hidden state output from encoder $(z_1, z_2, \ldots, z_m)$ and compute c_i at each i-th step. These weights are also jointly learned in attention model along with hidden states. Attention scores are computed by essentially comparing each encoder state z_i to a combination of the previous decoder state h_i and the last prediction y_i; the result to be distribution over input elements. Popular choices for both encoder and decoder has been stacked bi-directional LSTM or GRU in some occasions.

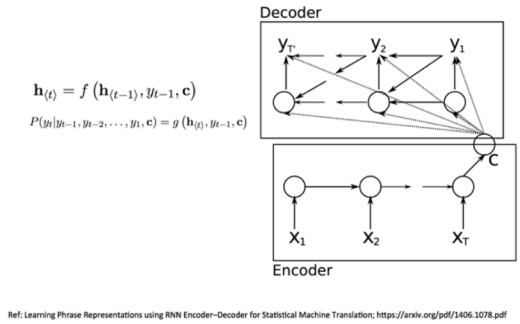

Figure 19: Encoder and Decoder

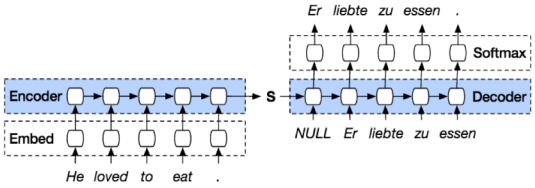

Figure 20: Encoder Decoder

3.5.1 Performance

Google:

TESLA:

Facebook:

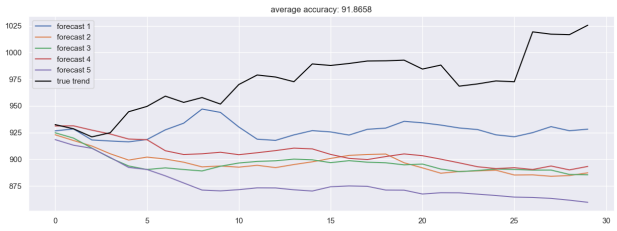

Figure 21: Google Performance on CNN-Seq2Seq

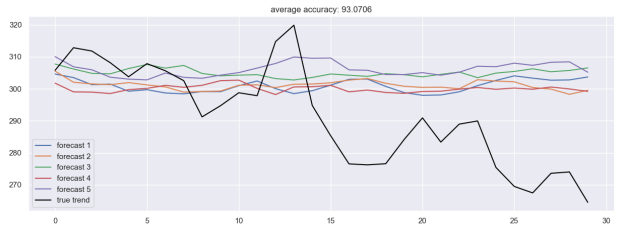

Figure 22: Tesla Performance on CNN-Seq2Seq

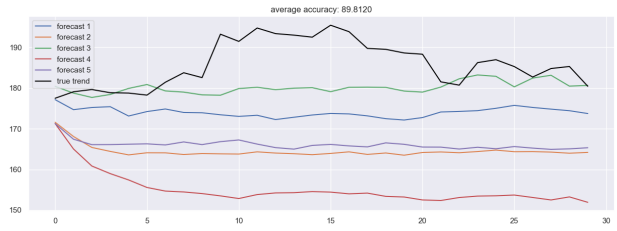

Figure 23: Facebook Performance on CNN-Seq2Seq

4 Market Playing

In Market Playing we train agents based on several well known learning algorithms and then let them **play** in real time. We set aside the last 30 days of every dataset to achieve the real time market environment. Due considerations are taken to separate this data from the training data. A better algorithm would produce a higher return after 30 days.

4.1 Policy Gradient

The objective of a Reinforcement Learning agent is to maximize the "expected" reward when following a policy π. Like any Machine Learning setup, we define a set of parameters θ (e.g. the coefficients of a complex polynomial or the weights and biases of units in a neural network) to parametrize this policy — π_θ(also written a π for brevity). If we represent the total reward for a given trajectory τ as r(τ), we arrive at the following definition.

Reinforcement Learning Objective: Maximize the "expected" reward following a parametrized policy.

$$J(\theta) = E_\pi[r(\tau)] \tag{1}$$

All finite MDPs have at least one optimal policy (which can give the maximum reward) and among all the optimal policies at least one is stationary and deterministic.
Like any other Machine Learning problem, if we can find the parameters θ which maximize J, we will have solved the task. A standard approach to solving this maximization problem in Machine Learning Literature is to use Gradient Ascent (or Descent). In gradient ascent, we keep stepping through the parameters using the following update rule:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2}$$

The Policy Gradient Theorem: The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ.

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[r(\theta) \nabla log \pi_\theta(\tau)] \tag{3}$$

Using the fact that $\pi_\theta(\tau) = P(s_0) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}, r_{t+1}|s_t, a_t)$, we can derive the following:

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[r(\tau)(\sum_{t=1}^{T} \nabla log \pi_\theta(a_t|s_t))] \tag{4}$$

This result is beautiful in its own right because this tells us, that we don't really need to know about the ergodic distribution of states P nor the environment dynamics p. This is crucial because for most practical purposes, it hard to model both these variables. Getting rid of them, is certainly good progress. As a result, all algorithms that use this result are known as "Model-Free Algorithms" because we don't "model" the environment.

The "expectation" (or equivalently an integral term) still lingers around. A simple but effective approach is to sample a large number of trajectories (I really mean LARGE!) and average them out. This is an approximation but an unbiased one, similar to approximating an integral over continuous space with a discrete set of points in the domain. This technique is formally known as Markov Chain Monte-Carlo (MCMC), widely used in Probabilistic Graphical Models and Bayesian Networks to approximate parametric probability distributions.

One term that remains untouched in our treatment above is the reward of the trajectory r(τ). Even though the gradient of the parametrized policy does not depend on the reward, this term adds a lot of variance in the MCMC sampling. Effectively, there are T sources of variance with each R_t contributing. However, we

can instead make use of the returns G_t because from the standpoint of optimizing the RL objective, rewards of the past don't contribute anything. Hence, if we replace r(τ) by the discounted return G_t, we arrive at the classic algorithm Policy Gradient algorithm called REINFORCE. This doesn't totally alleviate the problem as we discuss further.

REINFORCE Gradient:

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[(\sum_{t=1}^{T} G_t \nabla log \pi_\theta(a_t|s_t))] \tag{5}$$

We still have not solved the problem of variance in the sampled trajectories. One way to realize the problem is to reimagine the RL objective defined above as Likelihood Maximization (Maximum Likelihood Estimate). In an MLE setting, it is well known that data overwhelms the prior — in simpler words, no matter how bad initial estimates are, in the limit of data, the model will converge to the true parameters. However, in a setting where the data samples are of high variance, stabilizing the model parameters can be notoriously hard. In our context, any erratic trajectory can cause a sub-optimal shift in the policy distribution. This problem is aggravated by the scale of rewards.

4.1.1 Performance

Google:

Figure 24: Google on Policy Gradient

TESLA:

Figure 25: Tesla on Policy Gradient

Facebook:

4.2 Q Learning

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with

Figure 26: Facebook on Policy Gradient

stochastic transitions and rewards without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" refers to the function that the algorithm computes – the expected rewards for an action taken in a given state. Q is a function from the set of states and actions to the set of real numbers, that is $Q : S \times A \to R$. The main equation being:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{current value}} \right)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Figure 27: Q-Learning Update Equation

4.2.1 Performance

Google:

Figure 28: Google on Q Learning Policy

TESLA:

Facebook:

4.3 Actor Critic agent

Finding a good baseline is another challenge in itself and computing it another. Instead, let us make approximate that as well using parameters ω to make $V^{\omega}(s)$. All algorithms where we bootstrap the gradient using learnable $V^{\omega}(s)$ are known as Actor-Critic Algorithms because this value function estimate behaves

Figure 29: Tesla on Q Learning Policy

Figure 30: Facebook on Q Learning Policy

like a "critic" (good v/s bad values) to the "actor" (agent's policy). However this time, we have to compute gradients of both the actor and the critic.

One-Step Bootstrapped Return: A single step bootstrapped return takes the immediate reward and estimates the return by using a bootstrapped value-estimate of the next state in the trajectory.

$$G_t \simeq R_{t+1} + \gamma V^{\omega}(s_{t+1}) \tag{6}$$

Actor-Critic Policy Gradient:

It goes without being said that we also need to update the parameters of the critic. The objective there is generally taken to be the Mean Squared Loss (or a less harsh Huber Loss) and the parameters updated using Stochastic Gradient Descent.

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[(\sum_{t=1}^{T}(R_{t+1} + \gamma V^{\omega}(s_{t+1}) - V^{\omega}(s_t))\nabla log \pi_\theta(a_t|s_t)] \tag{7}$$

Critic's Objective:

$$J(\theta) = \frac{1}{2}(R_{t+1} + \gamma V^{\omega}(s_{t+1}) - V^{\omega}(s_t))^2 \tag{8}$$

$$\nabla J(\theta) = R_{t+1} + \gamma V^{\omega}(s_{t+1}) - V^{\omega}(s_t) \tag{9}$$

4.3.1 Performance

Google:

TESLA:

Facebook:

Figure 31: Google on Actor Critic Policy

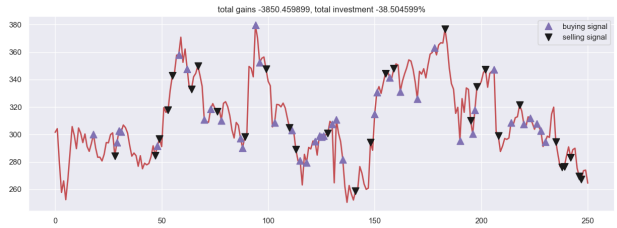

Figure 32: Tesla on Actor Critic Policy

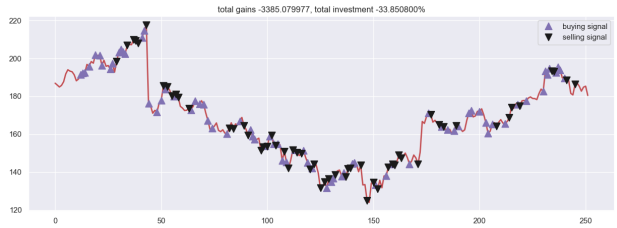

Figure 33: Facebook on Actor Critic Policy

5 Bibliography

- https://medium.com/@gautam.karmakar/summary-seq2seq-model-using-convolutional-neural-network-b1eb100fb4c4
- https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://arxiv.org/pdf/1705.03122.pdf
- https://arxiv.org/abs/1409.3215
- https://www.wikipedia.org/
- https://www.cse.iitb.ac.in/ shivaram/teaching/cs747-a2022/index.html
- https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d
- https://en.wikipedia.org/wiki/Q-learning
- https://oeis.org/wiki/ListofLaTeXmathematicalsymbols