Chapter 8

Initial Experiments: Naive Machine Learning

Before using complex architectures it is valuable to use more simple models to study learning behaviour and to assess the performance of further models. The algorithms we used for this are XGBoost and linear regression, popular machine learning tools. We started here to assess the performance of a more naive approach and to justify our more complex models later.

8.1 Prediction and evaluation setup

Contrary to further models these algorithms require that the input data is of constant size. They also don't allow us the parameter sharing that will make further models able to aggregate in model. This forces us to train the models on a per instance basis, which is of course a pretty big limitation.

This means that our dataset D has the following structure:

$$D : X \times Y \text{ where } X : \mathbb{R}^{28},\ Y : \mathbb{R}$$

and its content is a number of samples of a target funtcion f:

$$D = (x, f(x))\ \forall x \in X \text{ where } f : X \to Y$$

so that our models will select a function f^*:

$$f^* : X \to Y$$

to minimize the following objective

$$\epsilon = \int_X MSE(f(x), f^*(x))\, dx$$

which we will practically calculate as follows:

$$\epsilon = \sum_{X_{val}} MSE(f(x), f^*(x))$$

To get a more accurate estimate of performance on the whole set we will use multiple runs with different datasets.Because normal cross validation would be a non realistic situation we will evaluate the model on the 5 last folds of a 10 fold walk forward cross validation.

8.2 Linear regression, and other linear models

Linear regeressions are some of the simplest machine learning algorithms. They have no parameters, and are deterministic. The validation metric was average Root Mean Squared Error over the last 5 folds of a walk forward validation.

To somewhat controll the overfitting behaviour seen we also used an elasticnet regression, this model has both l1 and l2 regularization. To tune this model we used a bayesian optimization over the α and $regularization\, fraction$ hyperparameters.

8.3 XGBoost

Gradient Tree Boosting is a more complex algorithm that tries to construct an ensemble of very weak learners to make accurate predicitons. It was allowed train for 200 rounds but early stopping stops training when the RMSE stopped decreasing for more than 10 rounds. We use the dart booster[15] which means that more than one tree will be trained every round, depending on the number or trees dropped in a round, this allows us to reduce the effect of overspecialisation, and let the later training stages conctribute more to predictions.

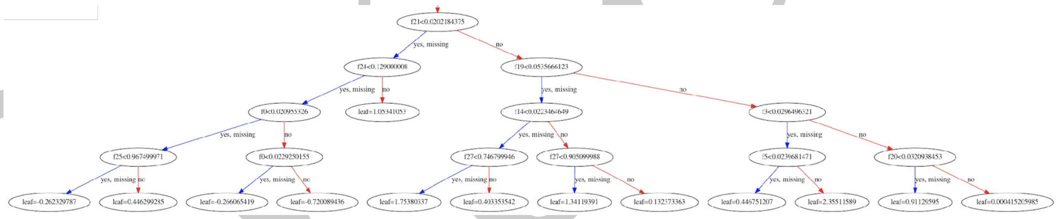

Figure 81: A part of a decision tree

Tuning We tuned the depth, eta, gamma and dropout rate hyperparameters. the large amount of feature engineering and dimensionality reduction was shown in the optimization favoring relatively low tree depths. It is here that the value of bayesian optimization becomes clear, we sampled +- 1500 configurations, but this means that is we performed a grid search less than 7 values per variable would have been possible. The acquisition functon of the optimization was expected improvement, with one random sample point every 10 evaluations to limit the greedyness of the search. After a 1000 iterations we limited the Tree depth, and Dropout rate to smaller intervals as the process seemed close to being converged for these values.

The validation metric was average Root Mean Squared Error over the last 5 folds of a walk forward validation. We can estimate the importances of the variables by plotting the F-score, which is the number of times each variable is used as a node in a tree, it's clear that our algorithm is still strongly driven by the textual sentiment.

8.4 Random forest

do we do this?

8.5 Limitations

Our relatively simple methods are unable to understand the variable amount of tweets in a day, this means we will have to use the mean of the predictions over the entire day to compare to the models that can handle the whole day.

Chapter 9

Experiment: Deep Learning Architectures

9.1 Basic Architectural Concepts

One very challenging part of the predictive task is that our input information and our output information are in a totally different order. The shape of our response variable is constant, one number for every day. Contrasting to this our input variables are in a non constant shape if looked at daily. One naive way of solving this would be to look at every part of the data as a sample and try to predict implied volatility on a per sample basis. This would however be sub optimal as our model doesn't get to consider the multiple samples in a day for making its prediction.

The question arises if it wouldn't be a good idea to just try to bin the data in a constant size to avoid these variable size problem. And while a form of binning will prove to be a good idea for the training of our network, keeping our model sample size agnostic is valuable because in a prediction context we will want to consider the whole day not just a subsection.

9.1.1 Embedding

The first part of all our model will perform an embedding. This means we will project data into a new representation, one learned by our model. This embedding learns a function over singular tweets, and will learn a lower level representation. This embedding is implemented by one or more one dimensional convolutional layers with the size of the input at that level as filter size, by adding more filters we can increase the dimension of the output embedding.

In our models we we apply at least one convolution layer over the data, this layer has a kernel size of 28 and n filters this will transform our data with the shape $(1, 28, sample\, size)$ to $(1, n, sample\, size)$. Further layers kernel size will (in this simple example) be equal to the number of filters in the previous layer.

A convolutional layer used in this way is isomorphic to the application of a fully connected layer over every single tweet.

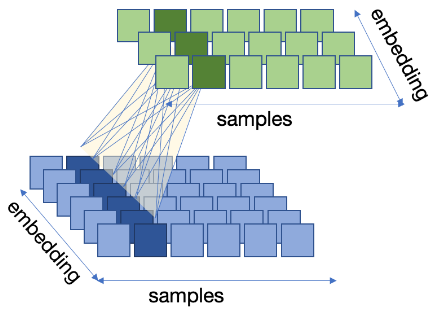

Figure 91: An example of how a convolutional layer performs an embedding

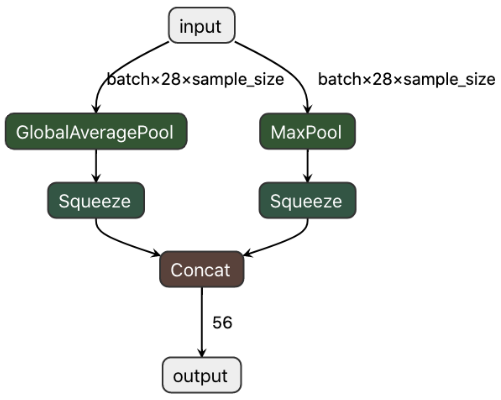

Figure 92: Example of an aggregation operation[1].

9.1.2 Aggregate

After the embedding stage we still have a varying dimension along our sample size, to make a prediction we will have to reduce this to a known dimension. For this we will use adaptive pooling, this is a special case of normal pooling. In our case an adaptive pooling layer will generally perform a transformation with the following shape $(1, embed\,size, samplesize) \rightarrow (1, embed\,size, 1)$ often squeezed to $embedsize$. In most cases we performed max and avg pooling and concatenated the result.

9.1.3 Post-processing

After the aggregation is applied we have a tensor of constant shape, this allows us to use a classic neural network to make our final prediction. In most cases we will use a hidden dense layer and a dense output layer here. These layers are the ones really

[1]Due to the varying size of our sample size our batch_size will always be one, this is why it is squeezed away.

making the prediction, this in contrast to the embedding layers that will re-embed the input to a more representation.

9.2 Architectural Approaches

9.2.1 PreNet

A first, naive approach could be to not post process the information and instead fully rely on the embedding to preform the predictions, and taking the mean of the predictions. This is a relatively simple methods that removes all interaction between samples. This approach closely mirrors the XGB approach, but it has a lot less parameters, still it performs largely similar.

9.2.2 PostNet

The opposite of the previous approach would be to skip the embedding part and embed the input data immediately, And then perform a prediction. This approach works slightly better than the pre processing approach but is still not really performant. This method basically is a deep unordered composition[10] or Deep Averaging Network (DAN). This is an approach to sentence learning that sacrifices the compositionality and thus syntactical information of their input for better learning performance and lower model size.

9.2.3 HybridNet

The logical next step is to combine the two approaches and make a so called Convolutional Deep Averaging Network, further called CDAN. These CDAN's will be the basis of all further models. These models perform a lot better than the previous ones, and will be considered as the base line model for our further deep learning experiments.

9.2.4 PoolingNet

One thing that could be valuable is for the embedding to be able to access the values of other samples. This would allow us mix the information without the drastic reduction in dimension we have at our aggregation part. One method for this could be to do the aggregation stage by stage, but this is hard to do without making more assumptions about the amount of samples we have. Therefore we use an approach where, for every sample, we use pooling layers to get information about other samples in his area, we do this by performing a max and average pool with a limited kernel size over the data. To make sure we have the correct shape we use circular padding. This means we extend both sides of the input along the sample axis with values on the other side.

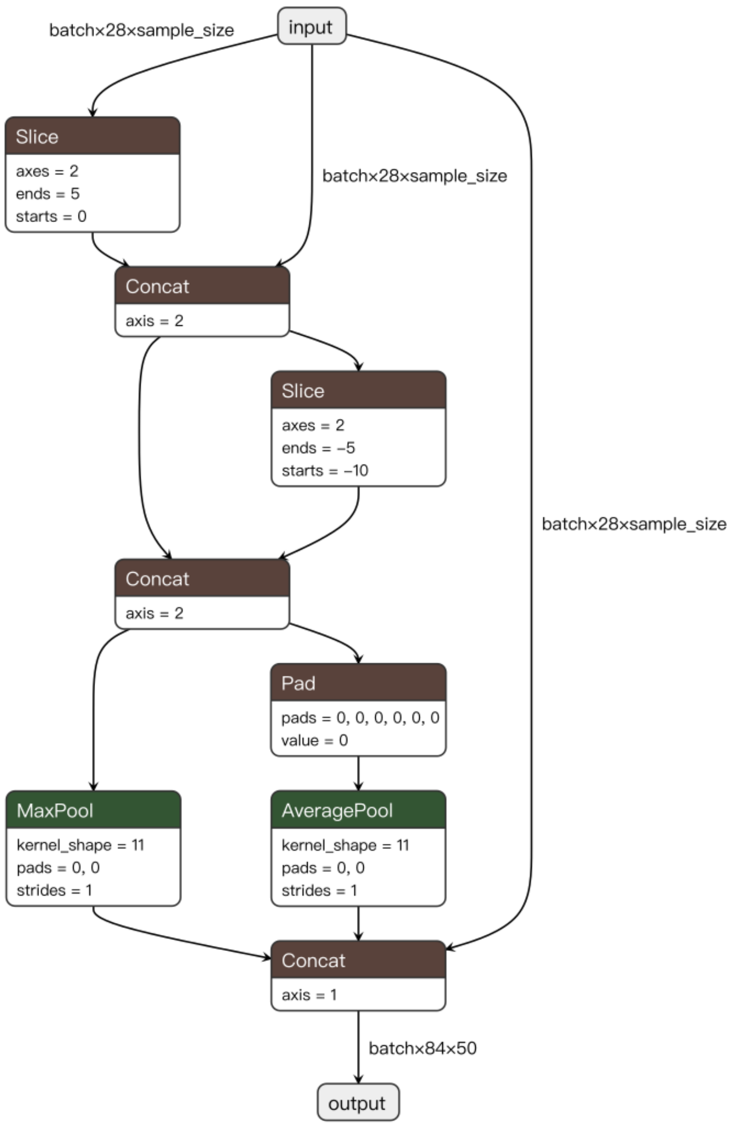

Figure 93: Example of a pooling operation

9.2.5 More Complex Data

Until now we have always used only the tweets that were accompanied by an image. This however is only a subset of the total dataset, therefore we would like to use that information in our architecture without losing our multimodal focus.

Parallel embedding

The first approach used is to perform a similar embedding and then aggregate, we then combine it with the result of the multimodal aggregate. This lets our post processing part look at the two different parts for making its prediction.

Distributive Pooling

In our PoolingNet model we let let the model inspect the state of other samples while embedding, it would of course be attractive to also be able to inspect some of the state of the tweets. This is however a lot more complicated. The amount of tweets without images is quite a lot bigger than the amount of tweets with images. This forces us to distribute the text embeddings among the tweets with text. For this we again will use a distributive pooling strategy.

9.3 Learning

The networks were trained on cpu with double precision, mini-batch size was one because our input had variable size. We used Huber loss, which is a variant of MSE loss