ALTERNATIVE INVESTEMENTS
ASSET MANAGEMENT ANALYTICS

Panathea Capital Partners

White Paper

Some Basic Aspects of Deep Learning for Applications to Financial Trading

Clemens H. Glaffig
Panathea Capital Partners GmbH & Co. KG, Freiburg, Germany

July 18, 2018

Some Basic Aspects of Deep Learning for Applications to Financial Trading

Clemens H. Glaffig
Panathea Capital Partners GmbH & Co. KG, Freiburg, Germany

July 18, 2018

In this treatise, we describe some of the basic and common deep learning architectures amenable to objectives in finance and give heuristic arguments as to which of those seems preferable for applications in financial trading. A specific deep architecture, allowing information cascades and different feature weights for different market environments, is described. Applicable objectives and input feature engineering for this architecture are discussed.

1. Introduction

Financial trading has been an area highly amenable to data driven decision processes ranging from quantitative dashboards to algorithmic trading systems. There is a vast number of accessible data available in easy to use formats, including tic data for price processes and order book data, providing information on trade intensities and order flows. In addition, alternative data like port activities, degree of city illumination, etc. are proving their added value in economic forecasting, enriching the classical macro-economic data driven analysis. This makes financial trading a prime target for data driven analytics. Indeed, throughout the last decade, the number of algorithmic or data driven ("quantamental") approaches to trade markets has flourished.

Speed, lack of undesired emotions and computational ease make it attractive to have a fully automated trading process, or at least a quantitative information system, aggregating and processing data online. Typically, trading systems are constructed by deriving informative features out of the available data set and aggregating them into a rule set to determine and drive trading decisions. However, a lot of trading systems in use are still based on essentially linear aggregations of simple features.

Specifically in the highly competitive area of electronic market making, strategies based solely on average speed and linear aggregations of order book data are nowadays insufficient to be profitable. An edge in the form of predictive capabilities is required, which demands more advance features and interactions. Feature engineering and particularly figuring out which features to combine in what way and in which circumstances is a tedious task if not automated in some way.

Machine learning ("ML") is an ideal candidate for this task. However, after very sobering experiences with early ML approaches around the turn of the last century, ML applications in financial trading have only slowly moved back into attention. Over the last decade, it has been primarily the shallow models (gradient bosting machines, random forests, support vector machines etc.), that were employed for simple tasks. With the recent hype in deep learning, the focus has shifted; every quant shop feels the urge to at least consider deep learning architectures. While the deep learning approach to finance still meets a lot of sceptics, there is hope that this time optimism is better justified.

Deep Learning in general has seen a tremendous development in recent years with several widely reported successes. Ever increasing computational power, the better understanding of relevant aspects for improved generalization properties and new techniques have led to faster learning and more robust results. While most of the new techniques like stochastic gradient descent, batch normalization, dropout, random perturbations, better behaved activations and targets, gradient momentum, skip connections etc. are widely used in all applications, we are skeptical with respect to some of the tricks applied specifically to financial trading and will briefly note on these later.

However, if higher speed and more computational steps in neural networks by stacking up more hidden layers is all there is to it, we wouldn't expect more success in applications to financial trading than before. What does make a substantial difference, is that deep architectures are now viewed as more than just a single stack of feed forward network layers: The general deep architecture is that of a feed forward a-cyclical graph, with each node being a computational unit that can be a network layer, a complete network on its own or other computational structure like some classical, shallow architecture. With this, a computational exercise can be split up into a number of smaller computational tasks, which can add to the overall picture. The deep network becomes a graph, divided into a number of interacting subgraphs. Simple examples include the various forms of gated memory networks (Long Short Term Memory Networks ("LSTM"), Neural Turing Machines, Differential Neural Computer ("DCN"), see [8], [7] and [6] respectively) that have transformed standard recurrent neural networks, which were originally notoriously hard to train, into the primary contender in sequential modeling, outperforming Hidden Markov Models. Other, more complex examples are some of the widely reported successes of deep learning like Alpha Go from DeepMind.

With most of the research effort still concentrated on areas like visual pattern recognition, speech recognition etc., the success of new techniques in these areas has stimulated more interest in financial applications. One of the major differences of financial data to data of other areas of application is the lack of stationarity of price processes. A key to successful applications of deep learning to financial trading will be to find the balance in the interplay between stationarity and information content.

A number of approaches employing deep learning to financial trading applications have been reported. However, most don't seem to tap the full potential of deep architectures: A lot of them appear to be simple, off the shelf stacks of purely feed-forward network layers, without the use of the power of more widely distributed computation. Moreover, the input features used are often rather simplistic. Besides raw data, well known technical indicators, already in use for several decades by a large portion of the trading community, were employed. It is highly unlikely, that a simple stack of network layers will unlock a new mapping from such well researched indicators to standard targets, offering superior returns. We do not believe that approaches with simple off the shelf architectures, fed by standard textbook indicators will be able to produce successful trading systems.

With today's view of a model representing a computational graph, feature engineering has been expanded to include (or in some nonfinancial applications replaced by) architectural design. A lot of problem specific knowledge goes not only into crafting the input and objective of a model, but specifically the construction of the graphical structure and the decision on how the overall computational problem is best divided into sub-tasks. With all the advances being made, model construction still seems to be more of an art, not a science yet.

In part 2 of this paper, we will briefly list three common, basic deep network architectures: A fully connected network, a sequential (1-dim) convolution network and a recurrent network. All of these could serve as basic computational devices for financial trading applications that could be embedded into a more refined, distributed, deep architecture. We compare those three networks and based on preferences of what a network ought to represent, we formulate our preference for one of these networks as the basic building block of a distributed computational architecture.

In part 3, we propose a general deep architecture that reflects trading and modeling preferences, build out of several individual networks, each solving a subtask of the overall objective. Specifically, the model will allow differentiating between several environmental settings and enable for different weightings of higher order features under different market environments. The proposed architecture will also allow cascading information across several time scales. The presentation will only describe the input features and the different applications in broad terms and will thus remain abstract, but comments on input feature engineering and target construction will be made. The details of a specific application or objective are determined by the specific definitions of target and objective functions and user-defined input feature engineering. Part 4 will conclude.

2. Comparison of basic deep networks

We assume we have readily engineered and normalized (not "de-meaned") a number d of input features, represented by a *d*-dimensional input vector for each trading period for a given trading scale. We further assume that the data covers T trading periods or bins, which can be segregated into training and validation sets.

2.1 Fully Connected Deep Neural Nets ("DNN")

Motivated by dynamical systems theory to predict states of a dynamical system by a large enough embedding, we combine the input vectors for the last *w* trading periods into a (*d* times *w*)-dimensional matrix *X (i, s)* for *i* = 1 to *d*, denoting the different features and *s* = *t*-*w*+1 to *t*, denoting the trading periods leading up to time *t*, with '*w*' as a pre-defined, fixed embedding dimension. Given that in the standard setting the input to a DNN is a vector, we convert the matrix X to an input vector Z for each data point *t*:

$$Z_t = [X(1,t), \dots, X(1,t-w+1), X(2,t), \dots X(2,t-w+1), \dots . X(d,t, \dots ., X(d,t-w+1)] \in R^{dw}$$

s.th.

$$Z_t(k) = X(i, t-s+1), \qquad for\ k = w(i-1)+s, \qquad i = 1\ to\ d, \qquad s = 1\ to\ w$$

For the first hidden layer *l*=1, we have for fixed data point *t*

$$\mathrm{H}_{l=1}(j) = \sigma\left(\sum_{k=1}^{d_0 w} W_{l=1}(j,k) Z_t(k) + b_{l=1}(j)\right), j = 1\ to\ d_{l=1}$$

Where d_l is the number of hidden units, W_l are the transfer matrix and b_l the bias of the current layer that map features of the previous layer linearly into new aggregated features, before de-linearizing them by a nonlinear activation function like sigmoid, tanh or Rectified Linear.

For subsequent layers *l* > 1, including the final output layer, we have

$$\mathrm{H}_l(j) = \sigma\left(\sum_{i=1}^{d_{l-1}} W_l(j,i) \mathrm{H}_{l-1}(i) + b_l(j)\right), j = 1\ to\ d_l \qquad (1)$$

Where d_l denote the number of aggregated features in the *l*-th layer. We have, for ease of notation, used the same activation for each layer and continue doing so in the following.

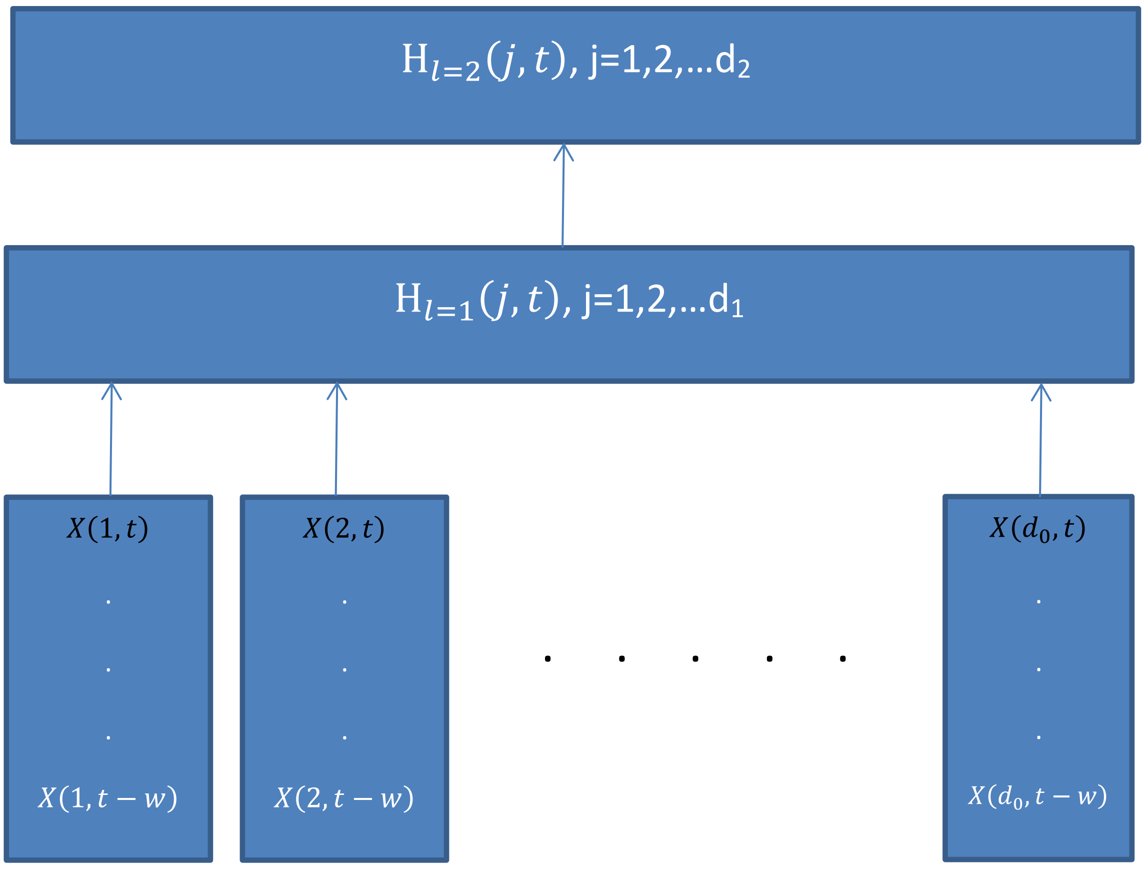

Figure 1
Fully connected network (DNN). Only the connection of the time embedding of each input feature and only the first two hidden layers are displayed.

2.2 Basics Deep Convolutional Neural Network ("CNN")

The archetypical application of CNNs is image recognition, in which each input channel typically has 2-dimensional input, representing pixel- characteristics in a 2-dimensional grid. The difference to fully connected DNNs is that the input connections are localized by a 2-dimensional convolution, which restricts the input to each hidden node to a small bounded region, the localization being determined by the position of the hidden node. It is very similar to wavelet analysis for image recognition. As wavelet analysis, CNNs have also been applied to time series forecasting, where they have distinct advantages relative to Recurrent Neural Networks ("RNNs"), the natural choice for times series modeling in a network setting. CNNs for time series are reduced to a one dimensional, unidirectional convolution along the time axis only. For input matrix $X(i,s)$ as above, the convolution only applies to the s direction, not w.r.t. the feature or input channel i. With the understanding, that the input corresponds to the 0-th layer, we obtain for the nodes of the hidden layers:

$$H_l(j,t) = \sigma\left(\sum_{s,i} W_l(j,i,s)\, H_{l-1}(i,t-s+1) + b_l(j)\right),\ j = 1\ to\ d_l,\ i = 1\ to\ d_{l-1},\ s = 1\ to\ w_{l-1} \tag{2}$$

with d_l as before and w_l the kernel width or embedding dimension of the l-th layer.

A CNN thus takes a linear combination of a w-embedding in time of all the features of the previous layer and mixes those by applying a nonlinear activation function. This mixing does not differentiate between time and feature index, i.e. it mixes past or present feature values within the time window.

While for general CNNs of several dimensions, a CNN can be viewed as restricting a fully connected DNN to a locally connected network, this interpretation is not valid for the time series application as given by (1) and (2). Here, the convolution along the time axis is not a timewise-local restriction of connections, as a DNN does not consider time as a feature coordinate, but as an index of the data sample. With the embedding, the time dimension was artificially introduced in restricted form to the DNN.

Sometimes it might be preferable to first construct a new feature out of the time evolution of a previous feature before mixing those with other features or mix different features at a given time first, before mixing those new combinations along the time axis. This can be achieved by separable CNNs: Choosing

$$W_l(j,i,s) = W_{l,f}(j,i) W_{l,w}(j,s)$$

and splitting a given layer into two layers, where either the feature index or the time index is run first results in these two different mixing preferences:

$$\mathrm{H}_l(j,t) = \sigma_f\left(\sum_i W_{l,f}(j,i)\, \sigma_w\left(\sum_s W_{l,w}(j,s)\, \mathrm{H}_{l-1}(i,t-s+1) + b_l(j)\right)\right),\ j,i,s\ as\ before \qquad (2a)$$

Example: VWAP, signed volume or certain measures of imbalances arising from the order book would first mix one feature with a second feature and then aggregate the time evolution of that mixture further. Feature weighted moving averages like VWAP can also be represented by the first layer of a DNN as previously defined.

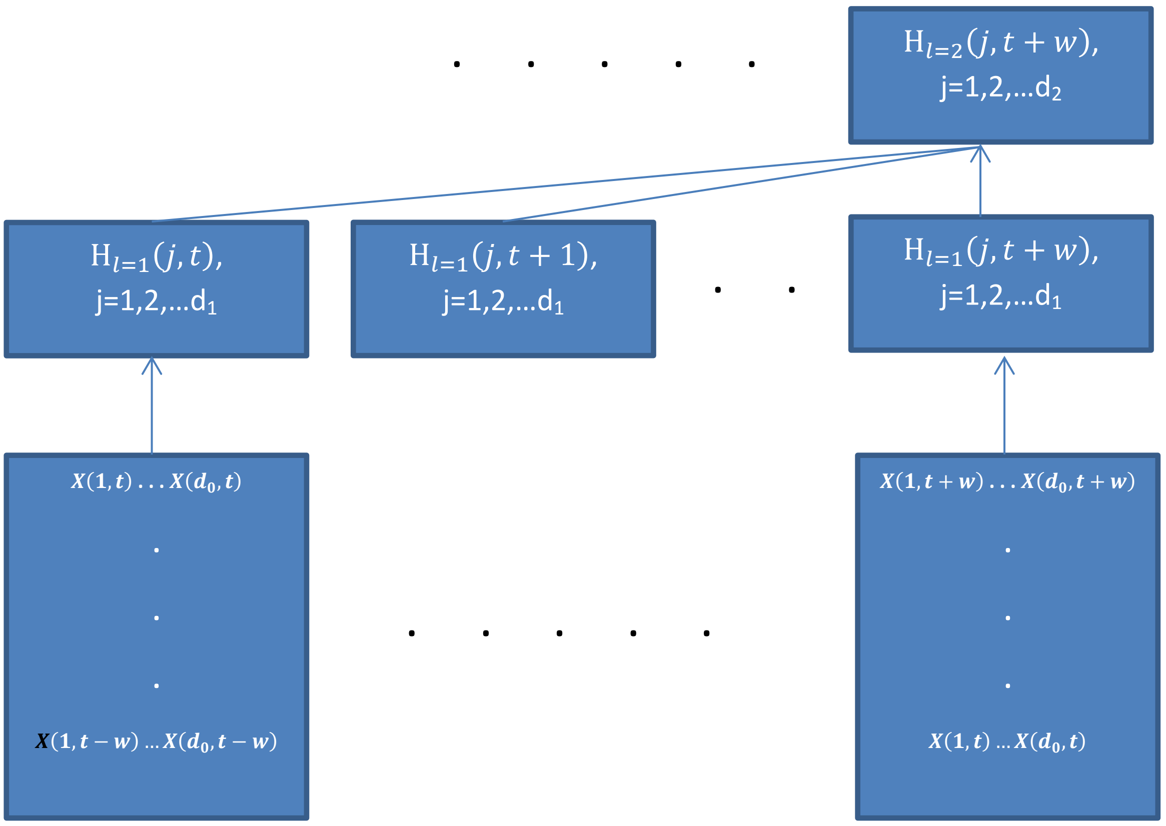
$H_{l=2}(j, t+w)$, j=1,2,…d₂
$H_{l=1}(j, t)$, j=1,2,…d₁
$H_{l=1}(j, t+1)$, j=1,2,…d₁
$H_{l=1}(j, t+w)$, j=1,2,…d₁
$X(1,t)\ldots X(d_0,t)$
$X(1,t-w)\ldots X(d_0,t-w)$
$X(1,t+w)\ldots X(d_0,t+w)$
$X(1,t)\ldots X(d_0,t)$

Figure 2
CNN. For each time step, the full matrix of time embedding (kernel width w) of each input feature is presented as input to all features of the first layer. For the second layer, only a specific point in time and no higher order layers are included in the figure.

2.3 Difference DNN to CNN

If we choose

$$W_{l=1}^{DNN}(j,k) = W_{l=1}^{CNN}(j,i,s),$$

$$for\ k = w_0(i-1)+s, \qquad i = 1\ to\ d_0, \qquad s = 1\ to\ w_0$$

we can see, that for the first layer, CNN and DNN are equivalent. However, for the second layer, the DNN does not consider a time embedding of the first layer features, i.e. a time evolution of higher order features is not taken into account, except for the input layer. If the time evolution of a higher order feature (i.e. one arrived at after several layers) is preferred, a DNN is not the appropriate approach. A DNN would correspond to a CNN with $w_l = 1, \forall l > 1$.

2.4 Basic Deep Recurrent Neural Networks (“RNN”):

RNNs are the quintessential network structure to represent time evolution and modeling time series. Their limitation thus far has been the difficulty to train them. Memory networks like LSTMs placing gated memory cells in place of the computational nodes have substantially improved that weakness.

New developments like Neural Turing Machines and DCNs have expanded this approach. A simple RNN propagates a feature node vector of a given layer at time *t*-1 to each feature node of the same layer at time *t*, jointly with the feature node vector of the previous layer at time *t*.

$$\mathrm{H}_l(j,t) = \sigma\left(\sum_i W_l^X(j,i)\,\mathrm{H}_{l-1}(i,t) + \sum_k W_l^H(j,k)\,\mathrm{H}_l(k,t-1) + b_l(j)\right), i = 1\ to\ d_{l-1}, k = 1\ to\ d_l \tag{3}$$

In this way, a specific feature at time *t* is mixed with its value at time *t*-1, but not directly, rather by passing the *t*-1 value to the next layer first, where it is activated and mixed with previous history, before it gets mixed with its *t*-time value. This can be better illustrated by expanding the above formula one step further into the past:

$$\mathrm{H}_l(:,t) = \sigma\left(W_l^X \mathrm{H}_{l-1}(:,t) + W_l^H \mathrm{H}_l(:,t-1) + b_l\right)$$

$$= \sigma\left(W_l^X \mathrm{H}_{l-1}(:,t) + W_l^H \sigma\left(W_l^X \mathrm{H}_{l-1}(:,t-1) + W_l^H \mathrm{H}_l(:,t-2) + b_l\right) + b_l\right) \tag{4}$$

(4) shows, how past data is transported to the new layer before the present data and interacts with the present only in “activated” and “pre-mixed” form.

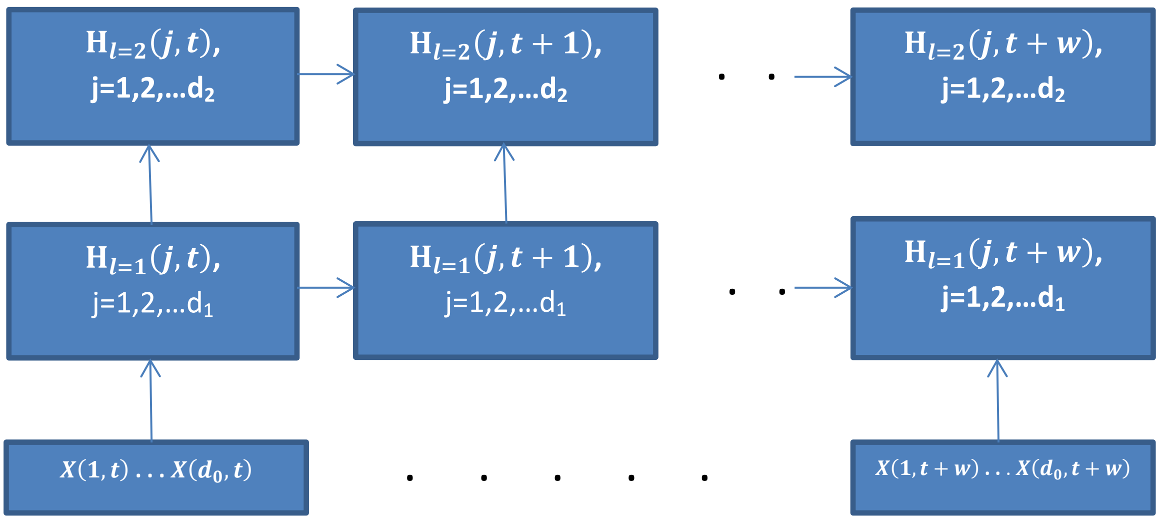
$\mathrm{H}_{l=2}(j,t)$, j=1,2,…d2
$\mathrm{H}_{l=2}(j,t+1)$, j=1,2,…d2
$\mathrm{H}_{l=2}(j,t+w)$, j=1,2,…d2
$\mathrm{H}_{l=1}(j,t)$, j=1,2,…d1
$\mathrm{H}_{l=1}(j,t+1)$, j=1,2,…d1
$\mathrm{H}_{l=1}(j,t+w)$, j=1,2,…d1
$X(1,t)\ldots X(d_0,t)$
$X(1,t+w)\ldots X(d_0,t+w)$

Figure 3
RNN. At each point of time, the full input feature vector, but only for that point in time is presented as input to all the features of the first layer at that specific time. Each first layer feature at that time is passed to the features of that layer at the next time step, along with the input feature vector of the next time step.

2.5 Difference CNN to RNN

For $\sigma(x) = x$ i.e. a purely linear model without nonlinear activation and a restriction of the recurrence to time-depth (kernel width) w, we obtain:

$$\begin{aligned} \mathrm{H}_l(:,t) &= W_l^X \mathrm{H}_{l-1}(:,t) + W_l^H W_l^X \mathrm{H}_{l-1}(:,t-1) + (W_l^H)^2 W_l^X \mathrm{H}_{l-1}(:,t-2) + \cdots \\ &\quad + (W_l^H)^w W_l^X \mathrm{H}_{l-1}(:,t-w) + w b_l \\ &= \sum_s (W_l^H)^{s-1} W_l^X \mathrm{H}_{l-1}(:,t-s+1) + b_l, s = 1\ to\ w \end{aligned}$$

Setting $W_l^{CNN}(j,i,s) = (W_l^{RNN,H})^{s-1} W_l^{RNN,X}(j,i)$, we see that in the purely linear case, CNNs and (restricted) RNNs, as defined by (2) and (3), are similar to the extent that linear RNNs are special cases of linear CNNs: For the strictly linear case, RNNs of the above form are more restricted through their time lag-weighting, as the weighting at time lag s is dependent on the weighting of time lag s-1.

For general, nonlinear RNNs, $\mathrm{H}_{l-1}(k,t-1)$ enters $\mathrm{H}_l(k,t)$ only through $\mathrm{H}_l(k,t-1)$ and is nonlinearly combined with $\mathrm{H}_{l-1}(k,t)$.

2.6 Desired Feature Aggregation:

While it may seem appealing to let the model act on data unconstrained by human expert interference, too much freedom leads often to computational capacity exceeding what is necessary, and finally resulting in overfitting. One way to cope with overfitting is to incorporate as much pre- or subject knowledge and guidance as possible without lowering the capacity too much. Specifically for applications in financial trading, knowledge can be employed not only in the engineering of input features and sensual targets, but also via preferences on the general production process for higher order features.

DNNs, as introduced above, are only able to build computational nodes for the first degree of time evolution, time evolution of higher order features is not considered. In our view, the time evolution of for higher order features is essential for constructing indicators for momentum, dynamics and impact of an underlying price process. Specifically, path dependency and autocorrelation of derived individual features along the price path provide valuable information to predict future behavior, higher serial independence (timewise) of features (not independence of different features) lower the signal to noise ratio. Therefore, we favor the inclusion of time evolutions for higher order computational layers and thus CNNs over DNNs. However, DNNs can be used as a computational stack on top of a CNN, to mix final, time evolved features out of a previous CNN.

RNNs are typically hard to train and have thus mostly been used in forms like LSTMs (i.e. memory cells instead of hidden nodes). But those as well as the recent developments like Neural Turing machines and DCNs have the aspect, that each lagged feature value will first be activated (non-linearized) before it can interact with present feature values. In a lot of cases, this overshoots on nonlinearity and is a source for training problems like vanishing or exploding gradients, the accompanying large Hessian points to weaker generalization properties (see [11]). Specifically, before a feature value at time *t-w* can interact with a feature value at time *t*, it has to undergo *w* mixing and nonlinear activations. However, if not only raw data but also preprocessed data is used, the need for additional nonlinearity is reduced. It might be desired to first mix lagged feature values linearly before activation.

Moreover, we would like to guide the network to the extent, that it mixes features and history as noted before (2a), in the remarks about separable transfer matrices of CNNs. As a consequence, out of the three deep network structures as discussed above, a separable CNN architecture seems to be the most adequate architecture to promote higher order feature computation for financial trading, specifically if input feature engineering is applied.

Next we give an example of an architecture, which combines several networks to target an overall task by dividing the problem into interacting sub-tasks.

3. A deep architecture for financial trading:

The behavior of an asset's price process will vary with the state of the environment the market is in. Consequently, the relevance of features will vary with different market environments. One can partition the data into segments of different market environments and develop individual models for each environment, while using an independent classifier to identify the environment. We prefer an integrated model that reflects environmental dependency of features and targeted objectives for the price process by splitting the overall task into interacting sub-tasks to reflect this dependency. In [1], the authors introduce an autoregressive convolution network for financial time series, in which they develop two stacks of deep CNN networks using one to supply a significance weighting to the other network, before being fed into a final, fully connected network producing the final output. We adopt and extend this architecture (with different objective) in the following way: We develop two computational stacks, each containing parallel silos of deep CNNs.

The first stack, which we call main-stack, will be used to develop signaling features from specific input ("main-input"). The second stack is called environmental stack and contains several deep CNNs modeling the market environment.

The task of the main stack will be to develop higher order features derived from input features. The task of the environmental stack is the development of an assessment of the given environment according to predefined characteristics and will be combined with the main task in the form of importance weighting for each higher order main-stack-feature.

Each stack (main and environmental) will be further divided into networks representing additional subtasks:

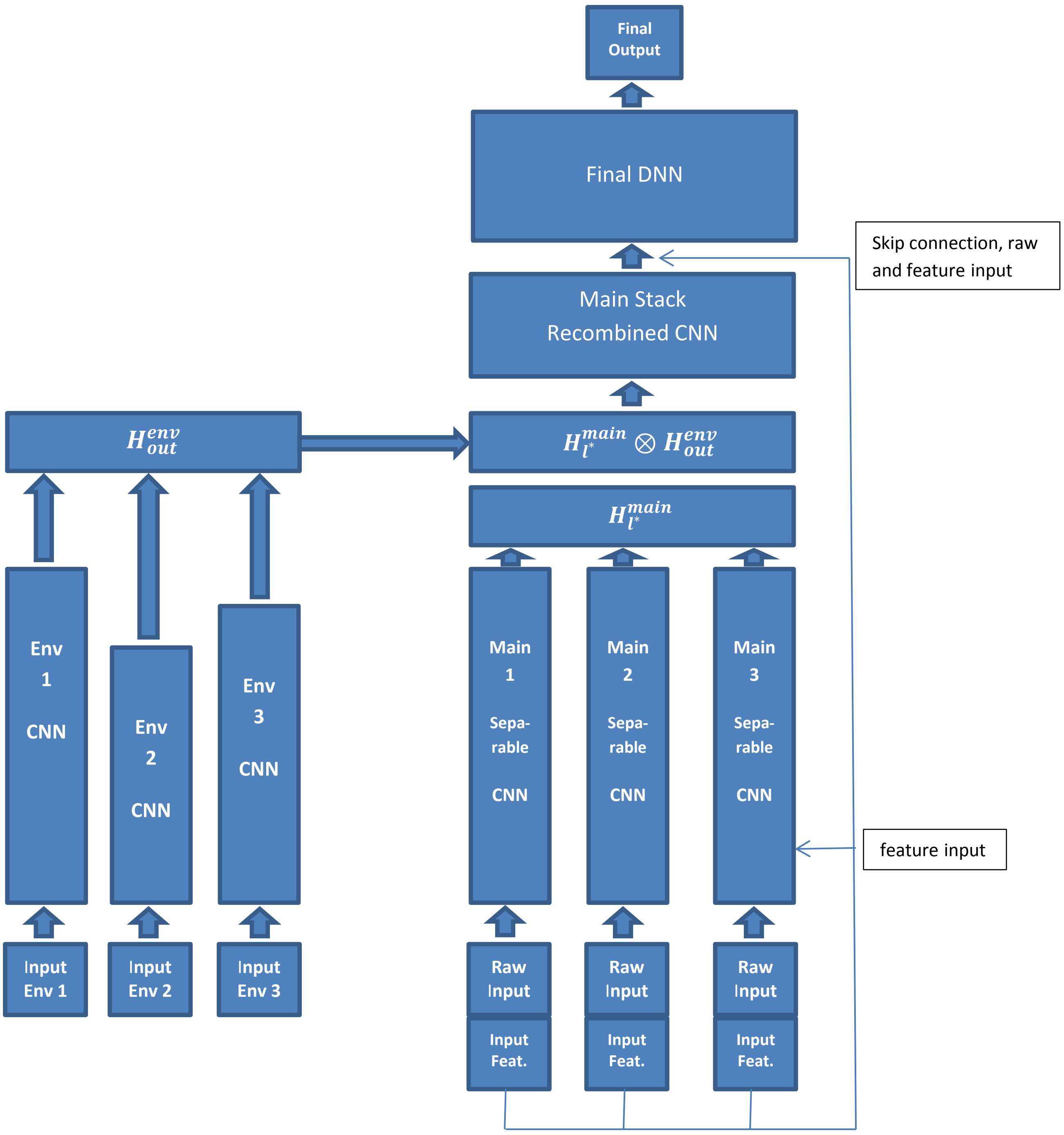

Figure 4:
Deep architecture with a stack of three environmental CNNs and a stack of three main CNNs. Input features to the main stack are fed into the respective CNNs at a later layer than raw input. Recombination of the three main stack CNNs take place at layer l^*, just before the importance multipliers of the environmental stack are applied. The development of path dependent features is continued, before a final DNN mixes without path dependence for the final output.

3.1 The environmental stack of the model architecture:

The environmental task will be further divided to develop importance weights for different characteristics of environment. Specifically, the environmental stack will consist of three non-interacting deep CNNs, with potentially different numbers of layers, each following its own computational path, developing its individual environmental characteristic's importance multiplier for the main stack. The three CNNs represent significance of a given data point, general market state and volatility regime respectively. The multipliers will be applied to the main stack at a specific layer ("weighting layer") by multiplying each feature of that layer with the maximum of the three importance weights, i.e. a feature is important if it is important with respect to at least one of the environment characteristics (see Figure 4). After that, the importance weighted features will be further developed within the main-stack-CNN (see section 3.2).

The input to the environmental stack is denoted by

$$X_{t-i}^{env}(k) = [X_{t-i}^{env,1}(1,2,\dots,k_1), X_{t-i}^{env,2}(1,2,\dots,k_2), X_{t-i}^{env,3}(1,2,\dots,k_3)]$$

Where „*env,z"* for *z*=1,2,3 denotes the *z*-th environment, k_z denotes the number of input features for the *z*-th environment and *i*=1,2,…,*w* the kernel width at the input level. We have used the same width of all three environments, which is not necessary. We also use regular spaced sample intervals, aligned with the main stack.

The individual input features for each CNN within the environmental stack, reflect a pre-chosen bias to the specific characteristic or task. E.g. the volatility regime characteristic will be fed by input-features like order cancellation rate, trade intensity, volume, true range, etc. This way, while a bias is fed at the outset, the system will develop its own understanding of how volatility regime determinants are to be used to form features resulting in a helpful environmental-relevance-multiplier for the main stack.

The features of the environmental layers would then be given according to equation (2) above, for $l \le l_{env,z}$, $l_{env,z}$ denoting the number of layers of the *z*-th characteristic of the environmental stack:

$for\ z = 1,2,3\ and\ j = 1,2, \dots d_{l,z}$:

$$H_l^{env,z}(j,t) = \sigma^{env,z}\left(\sum_{s,i} W_l^{env,z}(j,i,s)\, H_{l-1}^{env,z}(i,t-s+1) + b_l^{env,z}(j)\right), i = 1\ to\ d_{l-1,z}\,, \quad s = 1\ to\ w_{l-1}$$

The final output of the environmental stack will be:

$$H_{out}^{env}(:,t) = [H_{l_{env,1}}^{env,1}(1,2,\dots,d_{l,1},t), H_{l_{env,2}}^{env,2}(1,2,\dots,d_{l,2},t), H_{l_{env,3}}^{env,3}(1,2,\dots,d_{l,3},t)]$$

The final activation function for each environmental CNN will be the sigmoid function.

It is of course possible to use the full set of input features and only one network to learn about the environment and which importance multiplier best applied to the main stack. In theory, a single environmental CNN with the combined input ought to be able to learn everything the three individual networks are able to come up with. In practice however, it won't. Different and important environmental characteristics are more likely to be distinguished by different thematic input features, developing their own weighting each and are thus also more likely to partition the environment into information bearing settings for the overall task.

3.2 The main stack of the model architecture

The main stack will consist of several parallel deep, residual (skip connection), separable CNNs, each with the task to develop higher order features for its individual time scale, reaching from high frequency to minutes, hours and potentially days.

The use of multi time scales is common in technical trading systems, where coinciding signals on several time scales are used for entry or as confirmation of signals. It also allows to model cascading information from microstructure data and order book dynamics to short term price formation to longer term (hours to days) trends or anticyclical behavior. To make sure, that the different time scales regularly meet at specific points in time (achieved via strides for the layer if non-overlapping sampling periods are used), we use regular spaced sampling. We could also use irregular spaced data, subordinated to, e.g. trading time. Different scales would then meet at specific points of accumulated volume. The input features of the different CNNs will vary according to the specific scale and will include raw data as well as proprietary, aggregated features. The proprietary input features can be introduced to a later layer instead of the first layer. This way, we don't mix proprietary features with raw data immediately, but allow the system to develop its own features out of raw data for several layers. Further comments on the input will be given in section 3.6.

The developed features for the different CNNs of the main stack will be combined just before the weightings of the environmental stack will be applied. More specifically, the different CNNs of the main stack will all end after a certain number of layers with a layer of features combined from all of those CNNs, see Figure 4. At this stage the importance weights for each feature of the environmental stack will be applied before the main stack is further evolving along a single CNN, developing the combined and importance weighted feature set of the main stack.

Let l^* denote the weighting layer, at which the environmental stack will be combined with the main stack. The features at the l-th layer for $l \le l^*$ of each CNN representing time scale a=1,2,3 of the main stack are developed according to:

$$H_l^{main,a}(j,t) = \sigma_f^{main,a}\left(\sum_s W_{l,w}^{main,a}(j,s)\,\sigma_w^{main,a}\left(\sum_i W_{l,f}^{main,a}(j,i)\,H_{l-1}^{main,a}(i,t-s+1) + b_l^{main,a}(j)\right)\right),$$

$$i = 1\ to\ d_{l-1}\,,\ s = 1\ to\ w_{l-1}\,,\ a = 1,2,3 \qquad (5)$$

After the weighting layer l^* is calculated via (5), the different CNNs of the main stack end and their features are combined to one set of features at each data point of the highest time scale, achieved via the use of corresponding different strides for the different CNNs at layer l^*, if non-overlapping sampling periods are used. As an example, assume that non-overlapping sampling periods are used and the scales are 5M, 30M and 120M. The lowest scale will meet the highest scale every 24 steps and the second scale will meet the highest scale every 4 steps. It corresponds to including strides for layer l^* for the CNNs of the main stack of 24, 4 and 1 respectively for the three CNNs. The opposite extreme w.r. to overlapping sampling periods is to sample all data every 5M, i.e. according to the lowest scale. In this case, no striding is required.

$$H_{l^*}^{main} = [\ H_{l^*}^{main,1},\ \ H_{l^*}^{main,2},\ \ H_{l^*}^{main,3}]$$

and we set $d_{l^*}^{main} = d_{l^*}^{main,1} + d_{l^*}^{main,2} + d_{l^*}^{main,3}$

A note on data points for different time scales and non-overlapping sampling periods for all scales:

Staying with the previous scale example of non-overlapping 5M, 30M and 120M bins and assuming the data points of the 120M scale are the signaling points. For T such data points, the other scales have 24T and 4T data points respectively. One way to develop the features for the different scales is to use the full data set for all scales and use strides of 24, 4 and 1 respectively for the last, recombination layer (aligning with the signaling times). In this case, for e.g. the 5M scale we would, at a certain signaling point "*t*" use w times *l* of historic 5M bin information, with *l* being the current layer. A second way would be to use strides summing up to 24 and 4 respectively for the smaller scales along the way, i.e. before recombination. In this case, the same data as in the first case is used, just with fewer computational nodes. A third possibility is to use only the 120M scale data points and take the last w 5M bins, 30M bins and 120M bins respectively, i.e. in the case of the 5M scale, only the last w 5M bins of data will be used for all layers.

3.3 Combining environmental and main stacks:

The environmental stack would be recombined with the main stack directly after layer l^*, with $d_{l_{env,z}}^{env,z} = d_{l^*}^{main}, \forall z$, according to:

$$H_{l^*}^{main,recombined}(j,t) = \; H_{l^*}^{main}(j,t) \otimes H_{out}^{env}(j,t)$$

$$= \; H_{l^*}^{main}(j,t) \otimes \max(H_{l_{env,1}}^{env,1}(j,t), H_{l_{env,2}}^{env,2}(j,t), H_{l_{env,3}}^{env,3}(j,t)), \qquad j = 1,2,\dots,d_{l^*}^{main}$$

As mentioned before, multiplying each feature of the main stack with the maximum of the different environmental characteristic's weightings, we make sure that a feature, being important in one environmental setting keeps its importance also in cases of unimportance with respect to the other characteristics.

After the importance weightings are applied, we continue the main CNN for several layers to further develop time evolutions importance weighted higher order features that now combine all time scales.

For all layers $l > l^*$, the main features of that layer are calculated as in equation (3).

$$H_l^{main,recombined}(j,t) = \sigma^{main}\left(\sum_{s,i} W_l^{main}(j,i,s)\, H_{l-1}^{main,recombined}(i,t-s+1) + b_l^{main}(j)\right),$$
$$i = 1 \; to \; d_{l-1}, \qquad s = 1 \; to \; w_{l-1}$$

The final layer of the main stack can be joined by a strided skip connection, not only to better learn the identity function, but also, in case the original input features provide added value.

After the final layer $l = l_{main}$, the time-evolutionary aspect of the features of that layer are considered to be fully developed and will now serve as input to a final, fully connected deep network, which concentrates on mixing the final main-stack-features for the targeted output, according to equation (1).

$$\mathrm{H}_l(j,t) = \sigma^{DNN}\left(\sum_i W_l(j,i)\, \mathrm{H}_{l-1}(i,t) + b_l(j)\right)$$

with $\mathrm{H}_0 = \quad H_{l_{main}}^{main,recombined}$.

The final one-dimensional output, y, is then compared against the target via the chosen error functions.

$$y(t) = \sigma^{output}\left(\sum_i W_{l_{DNN}}(1,i)\,\mathrm{H}_{l_{DNN}-1}(i,t) + b_{l^{DNN}}(1)\right)$$

3.4 Output and target:

As final output target and training loss, we prefer to use handcrafted targets and error functions rather than the most common, standard functions. In the case of directional predictions, we would, e.g., use a utility based allocation/position output, which is then contrasted against a utility based performance evaluation for error calculations.

Other potential output targets for the proposed model include multi-asset allocation coefficients, regime switching points (change-point detection), and future input feature classification.

Specifically for short term applications the proposed type of architecture can also target the parameters of a model/process fit like drift or volatility of a diffusion process, or parameters of Point process to represent various trade- or order intensities. A classification of the parameters will prove more robust than a level regression.

Outputs that target the return, risk adjusted return, make predictions on the exact level of a market (regression) or simply include stop-loss and take-profit levels may sound like ideal objectives for trading strategies, but are in most cases too noisy, leading to overfitting.

3.5 Training:

The training of the model is done by standard Backpropagation, where we use some, but not all of the standard tricks and techniques to improve learning. In general we use techniques that speed up learning and lead to more robustness (like small Hessians around the optimal parameters set). We avoid tricks that merely circumvent problems without addressing their causes. Among those, we use:

- Stochastic Gradient descent
- Momentum
- Normalization of input scale
- Dropout

We don't use:

- Batch normalization. All layer outputs were brought to equal scale already which was in our experiments sufficient.
- De-meaning within normalization. The mean carries information content we would like to keep.
- Gradient Clipping: The necessity for gradient clipping points to rapid changes of the gradient, i.e. large Hessians, which again point to low robustness and overfitting. We would rather change the architecture.

3.6 Comments on input features

It has been reported in other areas of application that deep learning models can more easily digest raw data than more shallow architectures and are able to learn robust representations from them. However, our experience with respect to the applications to financial trading suggests that a solid understanding of the trading problem at hand, the underlying trading environment and process are still of distinct advantage in constructing informative input features. Input features, resulting from formalizing aged trading experience still considerably enhances performance and is in most cases key to differentiating between generalizing ability and overfitting. On the other hand, the approach to throw everything available in the feature universe at the model as input will provide the model with a lot of useless features for the task, the model will use them to better explain the training data, leading to more noise and overfitting. A PCA of the input features is a standard technique to reduce the dimension of the input space and achieve independence of feature space components. However, redundancies have sometimes advantages too. See [10] for more advanced methods on input feature preparation.

Without going into details at this stage, we use a feature selection process assessing the information content of input features for the objective before feeding the model. While this optimizes the input, some of the hyper parameters used in deep networks such as the number of hidden nodes/features per layer give rise to similar problems: Too many hidden nodes per layer provide the model potentially with too many higher order features for the task at hand and could again lead to overfitting. There are recipes for learning hyper parameters, which we will not consider here. The evolved importance multiplier applied to each higher order feature employed in the model above helps to reduce overfitting.

Moreover, we view the involvement of several time scales as very beneficial. Specifically, if microstructure data like order book information is used on very small scale, aggregated and combined with price data on larger scales, cascading effects can be more easily captured. Technical trading systems often use multiple time scales, but then consider the same price driven features, just on different scale. While this will still improve signal robustness and provide for better entry levels, it does not capture cascading effects the way different data and different features for different scales do.

4. Conclusion

ML and specifically deep learning models offer an automated method to extract higher order features together with their interaction from data. In particular, viewing a model as a computational graph, it offers an integrated approach to subdivide a task into interacting sub-tasks. Input feature engineering and architectural designs are key aspects to the success of these approaches when applied to financial trading. The construction of appropriate deep learning models reflect trading and market experiences as well as preferences arising out of experimental experience with different approaches. In this monograph, we have given reasons as to why serial CNNs are appropriate building blocks for a deep architecture application to financial trading and a specific example of how to distribute different aspects important to the overall task. Specifically, we have given an integrated approach reflecting changing relevancies of features in different market environments and informational cascades.

References

[1] Binkowski, M., Mart, G., Donnat, P. (2017). Autoregressive Convolutional Neural Network for Asynchronous Time Series. *arXiv: 1703.04122v2.*

[2] Borovykh, A., Bohte, S., Oosterlee, C. (2017). Conditional Time Series Forecasting with Convolutional Neural Networks. *arXiv: 1703.04691v3*

[3] Chollet, F. (2018). *Deep Learning with Python.* Manning

[4] Geron, A. (2017). *Hands–On Machine Learning with Scikit-Learn & TensorFlow.* O'Reilly.

[5] Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning.* MIT Press.

[6] Graves, A. et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature.*

[7] Graves, A., Wayne, G., Danihelka, I. (2014). Neural turing machines. *CORR, abs/1410.5401*

[8] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. *Neural Comput. 9(8):1735-1780, Nov. 199.7*

[9] Hsu, D. (2017). Time Series Forecasting Based on Augmented Long Short-Term Memory. *arXiv: 1707.00666v2*

[10] Lopez de Prado, M. (2018). *Advances in Financial Machine Learning.* Wiley.

[11] Smith, S., Le, Q. (2017). Understanding Generalization and Stochastic Gradient Descent. *arXiv: 1710.06451v1.*