Neural Stack Writeup

February 4, 2018

Abstract

We present neural network based ensemble models for forecasting influenza using predictions in the form of probability distributions from a set of component models. The question we try to learn an answer for is 'can we exploit the flexibility of neural networks to create a better ensemble than possible by a weighted averaging ensemble?'. On the two models we built (mixture density and convolution based) the results are mostly negative for the original question and mixed as far as the absolute benefit of the ensemble is considered. We present some reasoning behind these results, and steps to take for a better neural model.

1 Introduction

[*What did you do? Why?*]

2 Methods

In this section, we describe the dataset used, experimental setup and details of models evaluated.

The models involved in this work (both the component and the ensemble ones) try to forecast influenza in the United States represented as the *weighted influenza like illness* (wili%) time-series publicly provided by the US Centers for Disease Control and Prevention (CDC). In the wili% time series, there are the following targets of interest:

1. Week ahead targets: 1, 2, 3 and 4 weeks ahead wili% values from the time of making the prediction.

2. Seasonal targets

(a) Peak week: The peak week for the current season.

(b) Peak value: wili% value at the peak week.

(c) Onset week: Based on a defined baseline for the season, the onset week for that season is defined as the first week of the first 3 consecutive weeks with wili% equaling or exceeding the baseline.

For each week in the season, a model predicts these 7 targets for each of the 10 Health and Human Services (HHS) regions and the whole nation (making a total of 11 regions). A season 20xx-20yy here comprises of a set of mmwr weeks ([**mmwr**]) starting from week 30 of year 20xx and ending at week 29 of year 20yy. The data we use for each season starts from week 40 (instead of 30) and covers 33 weeks (including week 40 and ending at week 19 or 20 of next year) since we have most of the component model predictions in this, relatively active, zone of the whole season.

2.1 Component model inputs / outputs

A component model can takes input depending on the way it chooses to model influenza. As the output, it provides probability distributions for the 7 targets of interest as described in previous section. From these 7 targets, 2 are week values and 5 are wili% values.

For week values, the output is a normalized vector of size 33, with each value representing the probability of a bin like $[week_x, week_{x+1})$. Since there can be *no onset* in a season, e.g. if the wili% values are all below the baseline for that season, we have 34 bins for onset week instead.

For wili% values, the output is a normalized vector of size 130 representing probabilities of wili% lying in bins of size 0.1 starting from 0.0, i.e. $[0.0, 0.1)$, $[0.1, 0.2)$, ... See figure **??** for an example.

2.2 Ensemble model inputs

Unlike previous works, we don't rely on the performance of component models for estimating a set of *weights*. Instead, we use the predictions (probability distributions) from the components as input to the ensemble model and predict directly the output for the wili% prediction problem, skipping any intermediate weight estimation. The ensemble models *merge* probability distributions from input component models and predict probability distributions, in the same space, as output. During training, the negative of log score (log of probability assigned to the true bin) is minimized which is equivalent to cross entropy loss minimization.

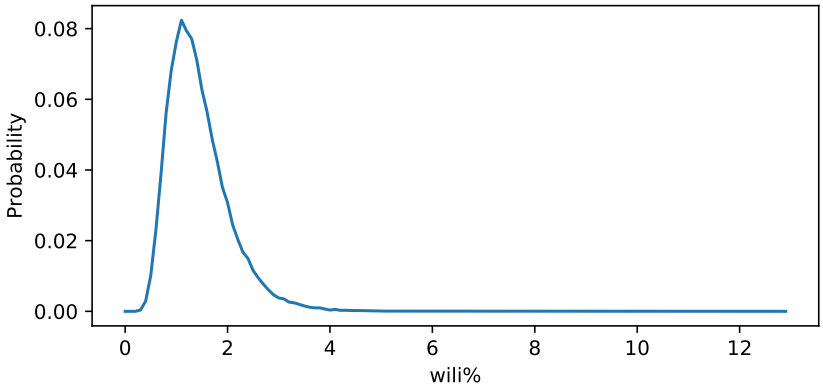


Figure 1: A sample discrete probability distribution for 1 week ahead wili% from one of the component models. x-axis is weighted ILI% split across 130 bins like [0.0, 0.1), [0.1, 0.2), etc.

Other than the probability distributions, the ensemble models also have information about the current week (the week they are merging probabilities for) of the season. Since week goes from 1 to 52/53, to preserve continuity, we encode each week (w) in a vector $\bar{w}$ of two elements in a sinusoidal representation as follows:

$$\bar{w} = [\sin(\frac{2\pi w}{n}), \cos(\frac{2\pi w}{n})]$$

Where n is the total number of weeks (52/53) in the season year. For each neural network model, we create a *with-week* variant which takes in $\bar{w}$ as one of its input.

2.3 Models

We evaluate two neural network models for the stacking task. The first model (mixture density network) works by approximating the input probability distributions using a gaussian and the output as a mixture of gaussians. The second model (convolutional neural network) works directly on the probability distributions (as vector of bin values) from components as input and returns a vector representing a probability distribution as output. Next subsection provides a brief introduction to general neural networks. Further sections explain the specific neural models used in our experiments:

2.3.1 Neural Networks

Neural Networks (or Artificial Neural Networks) are machine learning models based loosely on the way neurons are connected in animal brains. The high level aim is to learn a mapping from input to output which may be non-linear. In a general neural network, the *neurons* are arranged in some number of *hidden layers* along with an input and an output layer see figure **??**.

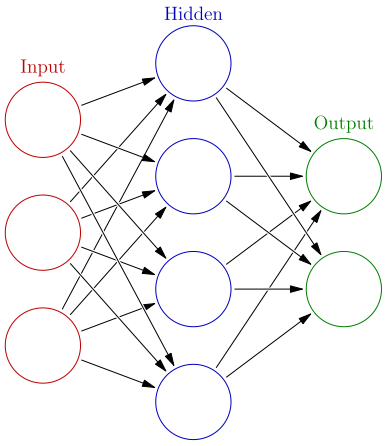


Figure 2: A feed forward neural network with one hidden layer. The input to this network is a vector of size 3 and the output is a vector of size 2. Image by Glosser.ca under CC BY-SA 3.0, source here.

In the most general case of a feedforward neural network, the neurons in i^{th} layer have incoming connections from all the neurons in $(i - 1)^{th}$ layer and outgoing connections to all the neurons in $(i + 1)^{th}$ layer. Each neuron in itself collects *its* input values (also called activations of the input neurons), uses its personal set of weights to find a weighted sum of them and passes the result through an activation function to produce its activation value. The whole pipeline effectively results in a mapping from input to output parametrized by the neuron connection weights.

To actually fit a model for the input and output, the network needs to *train* its weights so that it minimizes a certain loss function. The loss function is problem dependent and describes how poorly the output of the network matches with the actual output for the same input. This training is done using backpropagation which is a simple application of differentiation chain rule for propagating the gradient of loss function to all the neurons' weights. As an example, suppose we have a final loss function L in a neural

network with one scalar output as given below:

$$L = (\hat{y} - y)^2$$

Where y is the true value and $\hat{y}$ is the output from the neural network. If $\bar{w}_i(t)$ is the weight vector for the i^{th} layer at time t then a training iteration for simple gradient descent changes its value using the derivative $\frac{\partial L}{\partial \bar{w}_i}$ as:

$$\bar{w}_i(t+1) = \bar{w}_i(t) - \alpha \frac{\partial L}{\partial \bar{w}_i}\Big|_{\bar{w}_i = \bar{w}_i(t)}$$

There are many variations of the simple gradient update rule presented above which try to be avoid getting stuck in local optima and/or improve learning speed. Network training in this work uses rmsprop ([**rmsprop**]) which is an adaptive rate algorithm.

Neural networks have been successful on a variety of tasks. Recent advancements in the techniques and tooling have made it possible to train very *deep* networks capable of learning highly non-linear mappings with high generalization. A short review of these deep learning methods is presented in [**lecun2015deep**].

2.3.2 Mixture density network

A mixture density network [**bishop1994mixture**] is a simple feed forward neural network which outputs parameters for a mixture of distributions. The model we use *assumes* the output from the component models as normally distributed with certain mean and standard deviation. This translates to assuming a single gaussian peak in the output probability distribution from the inputs. It takes in these two inputs (mean and standard deviation of the distribution) from each of the component models and returns a mixture of n gaussians by outputting a set of means (μ_i), standard deviations (σ_i) and weights (w_i) for each distribution in the mixture. The final distribution for a network outputting n mixtures is then given by:

$$F(x) = \sum_{i=1}^{n} w_i f(x, \mu_i, \sigma_i^2) \tag{1}$$

Where $f(x, \mu_i, \sigma_i^2)$ represents a gaussian with mean μ_i and variance σ_i^2. Figure **??** shows the structure of a mixture density model (with weeks).

The loss function here is the crossentropy loss between the mixture of distributions generated by the network and one-hot representation of the truth. This loss is equivalent (with a sign flip) to the log score which just

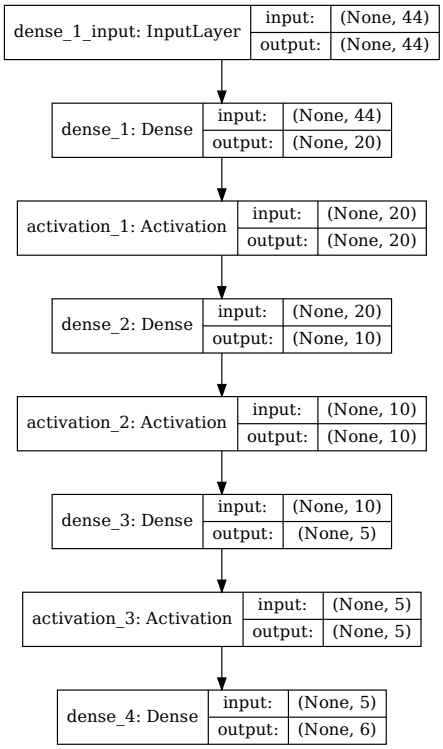


Figure 3: Graph of the mixture density network model. This specific network takes in means and standard deviations of 21 component models (42 inputs) and 2 inputs encoding week. It outputs 6 parameters to be interpreted as weights, means and standard deviations for a mixture of 2 gaussians.

tells the log of probability assigned to the true bin by the network. As an example suppose there are 100 discrete bins representing values from 0.01 to 1.00 and the true value (for an instance) is a single scalar 0.33. This truth can be represented in a *one-hot* representation as a vector of size 100 with just the 33^{rd} bin being 1 and the rest being 0. From the network's output of mixtures we can find the probability for this *true* bin and return its negative log as the final loss to minimize.

2.3.3 Convolutional neural network

Convolutional neural networks (CNNs) are neural networks characterized generally by presence of *convolutional layers*. First trained via backpropogation in [**lecun1989backpropagation**, **lecun1990handwritten**, **lecun1998gradient**] these layers differ from the regular fully connected layers in that the inputs to these layers and the weights themselves are arranged in a more general grid and each neuron is only connected to its *local* patch in the previous layers. A single convolutional layer has a set of such *locally responsive* filters. See figure **??** for a CNN working on an image (2D grid).

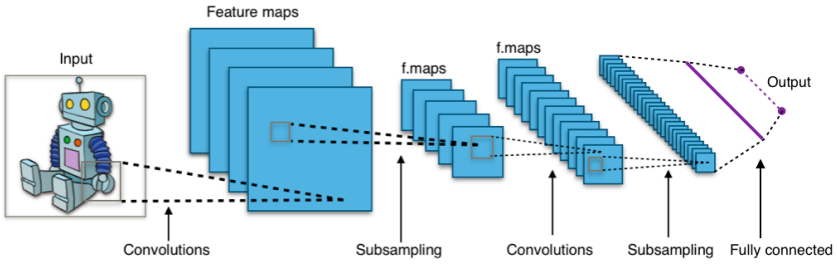


Figure 4: A general architecture of a CNN for image data. The image of a robot is a tensor with 2 dimensions specifying the pixel positions and the 3^{rd} dimension (not shown) specifying the color channel (R, G or B). A convolution layer has a certain number of filters working on local patches of input channels and creating a number of output channels shown as *feature maps* here. Intermediate subsampling layers reduce the grid dimension to bring spatial invariance. Finally, the result is calculated after flattening the outputs from the last subsampling layer and using simple feed forward layers. Image by Aphex34 under CC BY-SA 4.0, source here.

The CNN model in our work puts less assumptions on the input and output distributions and uses a set of 1-dimensional convolutional layers over the complete discrete input distributions. As the output, it returns

another discrete probability distribution vector. Figure **??** shows structure of the convolutional model which also takes in week encoding along with the inputs from the components.

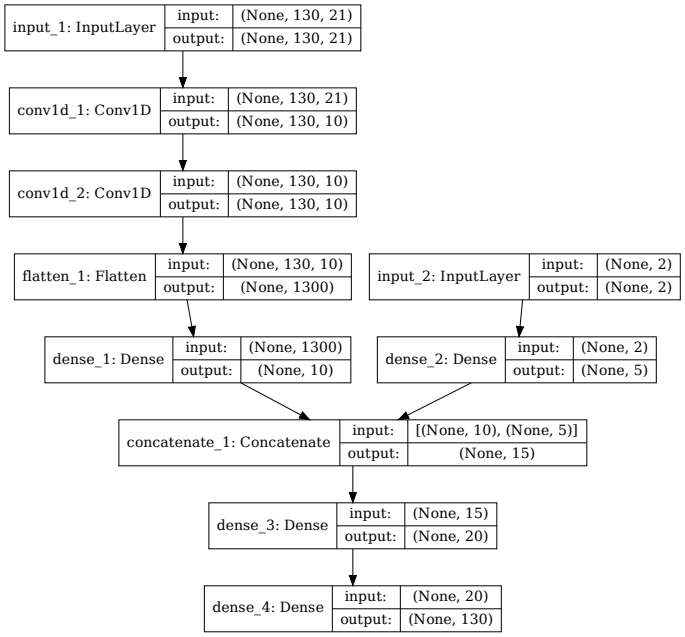


Figure 5: Graph of a convolutional neural model for wili target. The input on the left branch is a set of probability distributions (130 bins) representing wili values for 21 component models. The right branch takes in encoded weeks as vector of size 2. The model finally outputs a probability distribution using 130 bins (same as the component models).

2.4 Evaluation

We evaluate the two models in two different settings. Each setting has the same set of targets to predict but different number of component models and training seasons.

1. *Collaborative* ensemble setting: Here we use 21 component models from the FluSightNetwork collaboration [1] which provides us data for 4 train-

[1] `https://github.com/FluSightNetwork/cdc-flusight-ensemble`

ing and 3 test seasons.

2. *Lab* ensemble setting: This uses 3 component models from Reich lab and has data for 14 training and 5 test seasons.

We train separate models for each region and target. For both the settings, we use a leave one season out cross validation for tuning hyper-parameters (number of training epochs). Negative log score is used as the training loss function.

For comparison, we also train the following simpler ensemble models:

1. Five weighted averaging models based on the following weight learning approaches:
   (a) Equal weights: Assigns equal weight to each component. Equivalent to taking mean of the component bins.
   (b) Constant weights: Constant weight for each component learned using degenerate EM.
   (c) Target type weights: Different set of weights learned for *seasonal* and *weekly* targets using degenerate EM.
   (d) Target weights: Different weights learned for each 7 targets using degenerate EM.
   (e) Target region weights: Different weights learned for each 7 targets *and* 11 regions using degenerate EM.
2. Product ensemble: Takes geometric mean of the component bins.

Source code for reproducing our experiments is available on github at `https://github.com/reichlab/neural-stack`.

3 Results

We show here the mean log scores for the two settings grouped by target types (seasonal and weekly targets) on the test data. The ensemble models shown in the graphs are the following:

- `mdn`: Mixture density network
- `mdn-week`: Mixture density network with weeks
- `cnn`: Convolutional neural network

- `cnn-week`: Convolutional neural network with weeks
- `product`: Product model
- `dem-equal`: Equal weights model
- `dem-constant`: Constant weights model using degenerate EM
- `dem-target-type`: Target type weights model using degenerate EM
- `dem-target`: Target weights model using degenerate EM
- `dem-target-region`: Target and region weights model using degenerate EM

Results for the collaborative setting is in figure 3 and for the lab setting is in figure 3.

Although the results hint at neural ensemble performances being among the other ensembles, there is no concrete advantage visible [2]. We discuss the results in the next section.

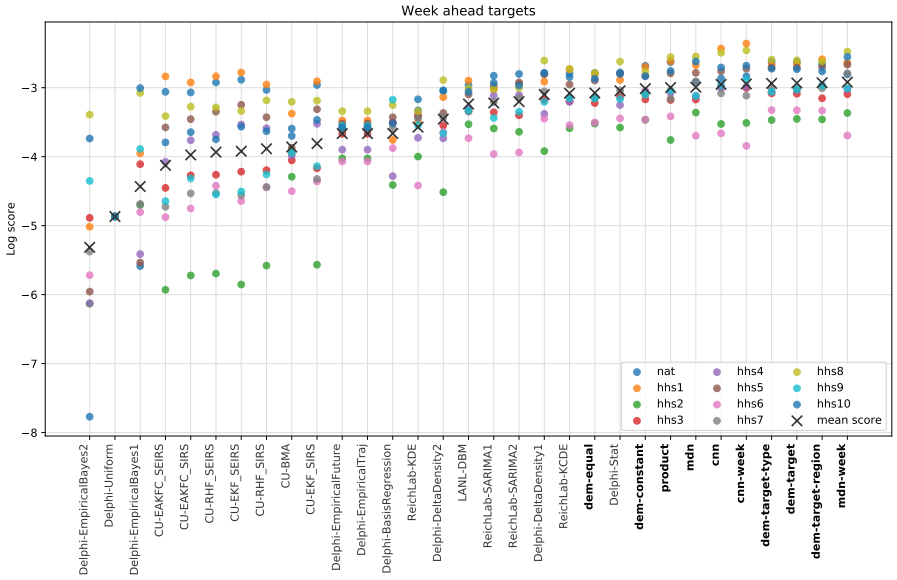
4 Discussion

In all the experiments, simple weighing based ensembles consistently performed good while the neural models had mixed results. Even though one reasonable line of reasoning is lack of lots of training data (we have *just* 52/53 weeks in a year), a more useful argument is lack of proper tuning and analysis. Neural networks based models are flexible enough to degenerate into simpler models like the weighted averaging ensembles without going very *deep* (which might need more instances to fit).

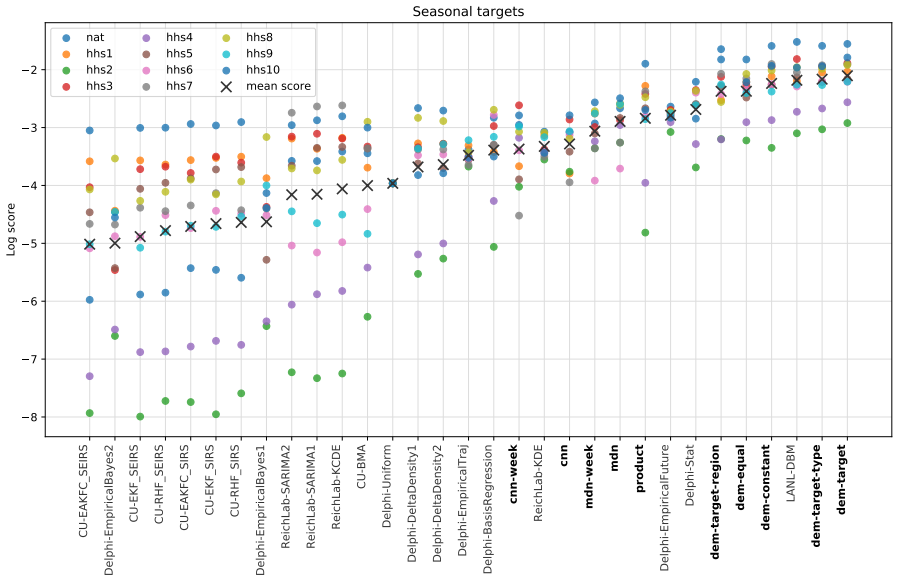
The original aim of exploring ensembles beyond simple weighing using neural models can be made less *black-boxy* by analyzing how exactly the components are *lacking* in modeling the truth. As an example consider that one of the component model's predictions are always correct but with low confidence, i.e. the peaks of its output distributions are always at the truth, but the probability is more spread out. In this case, it makes sense to have a model which has transformation capabilities for changing the variance instead of making a general purpose network using vectors as input/output.

A connected weak point of the experiments that we have presented here is that the models are not restricted to just learn *transformations* of the

[2] See results per target in the code repository at `https://github.com/reichlab/neural-stack/tree/master/notebooks`

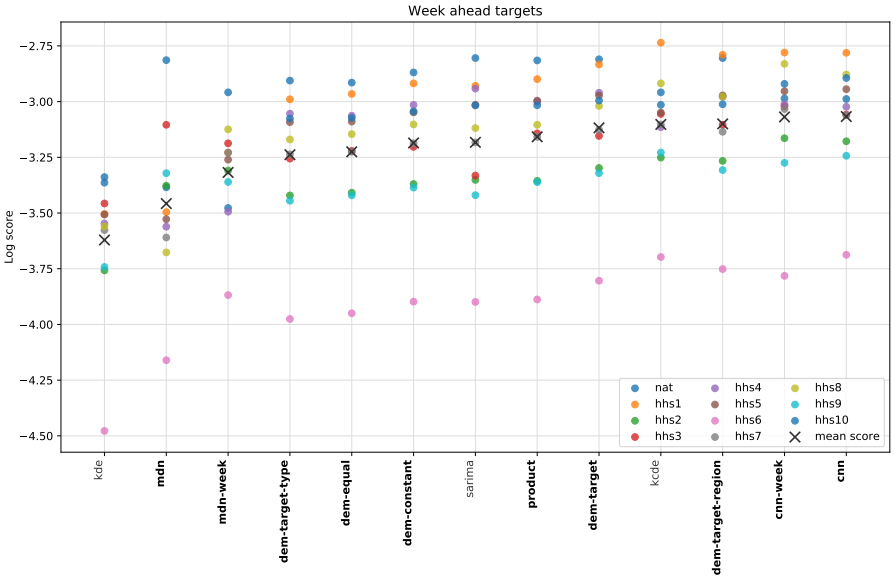


(a)

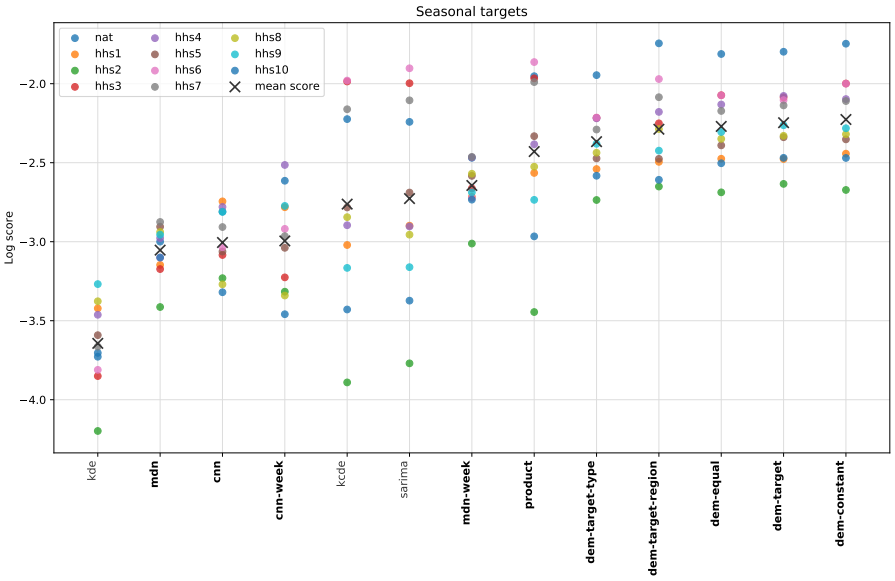


(b)

Figure 6: Test data log scores in collaborative setting sorted by increasing mean score (over all the regions). Higher score is better. **Bold** models are ensembles.



(a)



(b)

Figure 7: Test data log scores in lab setting sorted by increasing mean score (over all the regions). Higher score is better. **Bold** models are ensembles.

input distributions and can go about learning the times series itself (which is a separate problem). This generality in the network design hurts their interpretability which, in effect, makes it harder to debug and improve them in a reasonable way.

Going back to the original question, there are few things to do to learn more about the suitability of a specific ensemble model and the next steps to take from here:

- Analyze the main weaknesses in the weight based ensemble. For example, there is no way to get right predictions if all the models are disagreeing *around* the truth. If we use a weight based model which only considers the peak point of the probability, then this problem is partially solved, but in the probability weighting model, regions with low probability assigned by *all* components can't be reached.

- Start with a simple neural model which tries to patch the issues raised by the point above. For the given example, one fix is to have a network which outputs the component weights (opposed to what we do) and peak shifts.

- Proceed with generalizing the network design as much as possible under the data constraints we have.

Another unexplored possibility in the network design is of using recurrent components/layers which are helpful for modeling time based dependencies by maintaining states inside the network.

5 Conclusions