Dynamical Neural Network Models for Sequence Learning

Sharath Ramkumar
Faculty Advisor: Dr. Robert Kozma
University of Massachusetts Amherst
COMPSCI 496: Independent Study

Abstract

In this study, we explore various existing neural network models for learning sequences. We also propose alternative dynamic spiking neural architectures and compare accuracy of these network models on an artificial prediction task.

Introduction

Neural network models are able to achieve superhuman performance on static tasks due to recent breakthroughs in deep learning. However, the majority of real-world time-series data is often complex and noisy. As a result, we need dynamic models that can adapt to changing data streams to learn and predict in an robust, semi-supervised fashion. There have been some recent studies comparing various models for time-series prediction tasks [**cui2016continuous**] on discretized data. In this study, we will replicate their results on some of the models to establish a baseline and propose semi-surpervised spiking neural architectures for performing these tasks. Spiking neurons [**gerstner2002spiking**] are biologically-inspired models of neurons. Spiking neural networks (SNNs) [**maass1997networks**] are organized connections between these spiking neurons which have more energy efficient [**cruz2012energy**] implementations on neuromorphic hardware.

Methodology

Artificial Dataset

Cui et al. (2016) propose an artificial dataset of sequences of varying length (formed with characters) with overlapping subsequences. A sequence is sampled from the dataset and presented to the model at each time step. After the last character of the sequence is presented, a character is sampled from the noise distribution and shown to the model. This process is repeated until $10,000$

characters (counting both sequences and noise characters) are shown to the model. After the model sees $10,000$ characters, the last character of sequences with shared subsequences is swapped. The performance on this task is the accuracy of predicting the last character over a window of the last 100 sequences. A sample dataset is summarized in Table **??**. The code and datasets are open-source and available at https://github.com/sharath/sequence-learning.

Pre-$10,000$			Post-$10,000$		
Start	Subsequence	End	Start	Subsequence	End
6	$8,7,4,2,3$	0	6	$8,7,4,2,3$	5
1	$8,7,4,2,3$	5	1	$8,7,4,2,3$	0
6	$3,4,2,7,8$	5	6	$3,4,2,7,8$	0
1	$3,4,2,7,8$	0	1	$3,4,2,7,8$	5
0	$9,7,8,5,3,4$	1	0	$9,7,8,5,3,4$	6
2	$9,7,8,5,3,4$	6	2	$9,7,8,5,3,4$	1
0	$4,3,5,8,7,9$	6	0	$4,3,5,8,7,9$	1
2	$4,3,5,8,7,9$	1	2	$4,3,5,8,7,9$	6

Table 1: A sample artificial dataset for this task. The ending character can be predicted by learning the mapping from the start character and the first character of the shared subsequence to the end character.

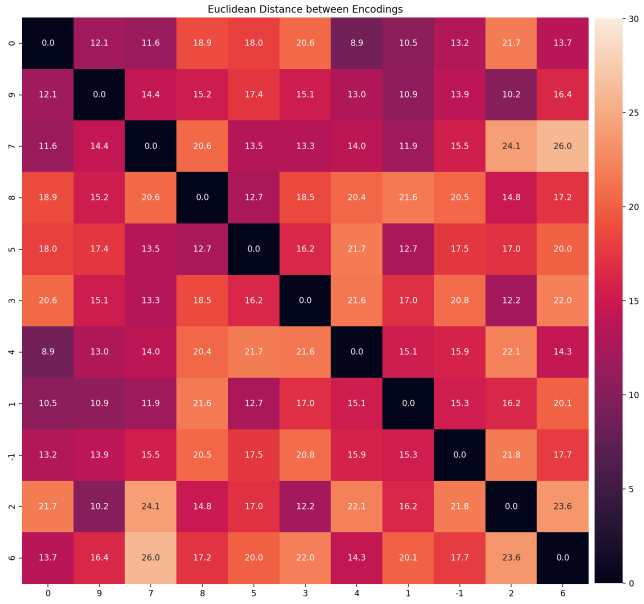

Figure 1: Heatmap of squared errors between distributed random encodings for the sequence characters, with -1 as the separating bit for ease of visualization.

Input Encoding

To present each of these discrete categories to the model, we have to encode the categories to a vector representations. One possibility is to one-hot encode each character, but this representation scales poorly with the number of unique characters. For this reason, the characters are encoded as random vectors with real values in the interval $[-1, 1]$, similar to distributed representations used in natural language learning. [**mikolov2013distributed**] The euclidean distances between pairs of unique sequence characters using a 25 length random ending is shown in Figure **??**. The decoder for converting vectors back to character uses a nearest neighbor approach, therefore the precision in the output vector from the model is important as the number of noise characters increases.

Supervised Models

Long short-term memory networks (LSTM)

Long short-term memory [**hochreiter1997long**] networks are able to achieve state of the art results on various sequence learning tasks, so they are a good starting point to establish baseline performance on this task. We use the same LSTM architecture proposed by Cui et al. (2016) with 25 input neurons connected to a hidden layer of 20 LSTM cells and 25 output neurons. The network is trained using truncated backpropagation through time (TBPTT) [**mozer1995focused**, **sutskever2013training**] on the last 100 seen elements. The output is classified using the nearest neighbor decoder.

Long short-term memory (LSTM) Behavior

There are many different variants of LSTM implementations. In this work, the LSTM cell has a forget gate, but no peephole connections. The equations defining the forward pass of the hidden layer in the LSTM model are:

$$i_t = \sigma\left(W_{ii}x_t + W_{hi}h_{t-1} + b_i\right) \tag{1}$$
$$f_t = \sigma\left(W_{if}x_t + W_{hf}h_{t-1} + b_f\right) \tag{2}$$
$$g_t = \tanh\left(W_{ig}x_t + W_{hg}h_{t-1} + b_g\right) \tag{3}$$
$$o_t = \sigma\left(W_{io}x_t + W_{ho}h_{t-1} + b_o\right) \tag{4}$$
$$c_t = c_{t-1} * f_t + i_t * g_t \tag{5}$$
$$h_t = o_t * \tanh\left(c_t\right) \tag{6}$$

In the above equations, σ is the sigmoid activation function, i_t is the input gate, f_t is the forget gate, g_t is the cell gate, o_t is the output gate and the c_t and h_t are the cell and hidden states respectively. The $*$ operator is the element-wise product operator. The biases (b_i, b_f, b_g, b_o) and weights

$(W_{ii}, W_{hi}, W_{if}, W_{hf}, W_{ig}, W_{hg}, W_{io}, W_{ho})$ are initialized uniformly randomly from $\left[-\frac{1}{\sqrt{k}}, \frac{1}{\sqrt{k}}\right]$, where k is the number of hidden units. [**jozefowicz2015empirical**]

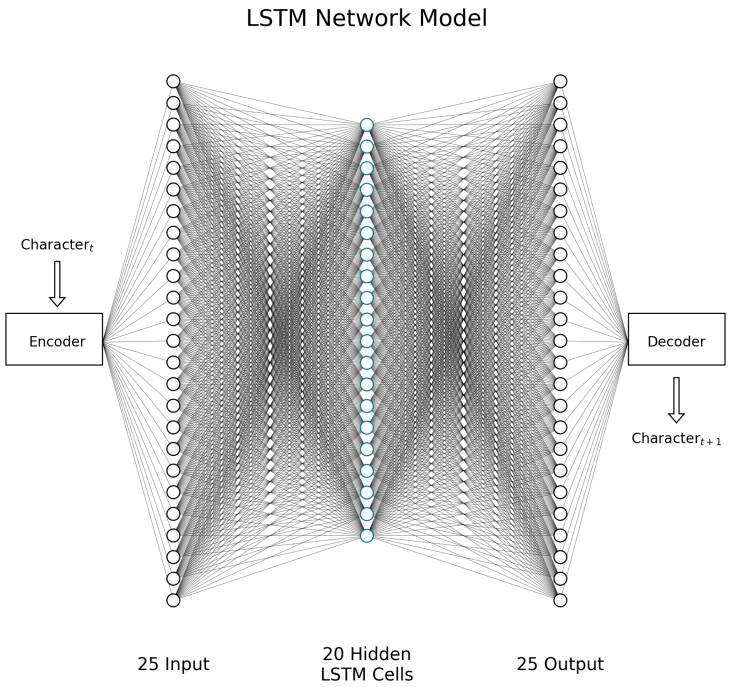

Figure 2: The network architecture for the LSTM-online model. The LSTM cells (teal) maintain an internal hidden state.

Time-delay neural networks (TDNN)

Time-delay neural networks (TDNN) are feed-forward neural networks trained using a lag on a window of previous data. [**waibel1995phoneme**] For this task, the TDNN is trained on the last 3000 samples every 1000 elements with a lag of 10. [**rojas1996backpropagation**] The model has 250 input neurons that are fully connected to a hidden layer of 200 neurons. The hidden layers utilize the ReLU non-linearity and are fully connected to 25 output neurons for the prediction. The model proposed by Cui et al. has a sigmoid non-linearity in the hidden layer. We replaced the non-linearity to allow for conversion into a spiking neural network. As with the LSTM model, the output is classified using the nearest neighbor decoder.

The network can be formulated as an equation in terms of the 250 length input vector x_t and the ReLU non-linearity function $R(z)$:

$$\text{TDNN}(x_t) = R\left(x_t W_{ih}^T + b_i\right) W_{ho}^T + b_h \tag{7}$$

The biases (b_i, b_h) and the weights (W_{ih}, W_{ho}) are initialized uniformly randomly from $\left[-\frac{1}{\sqrt{k_i}}, \frac{1}{\sqrt{k_i}}\right]$, where k_i is the number of input features to the respective layer. The weights are then learned through gradient descent.

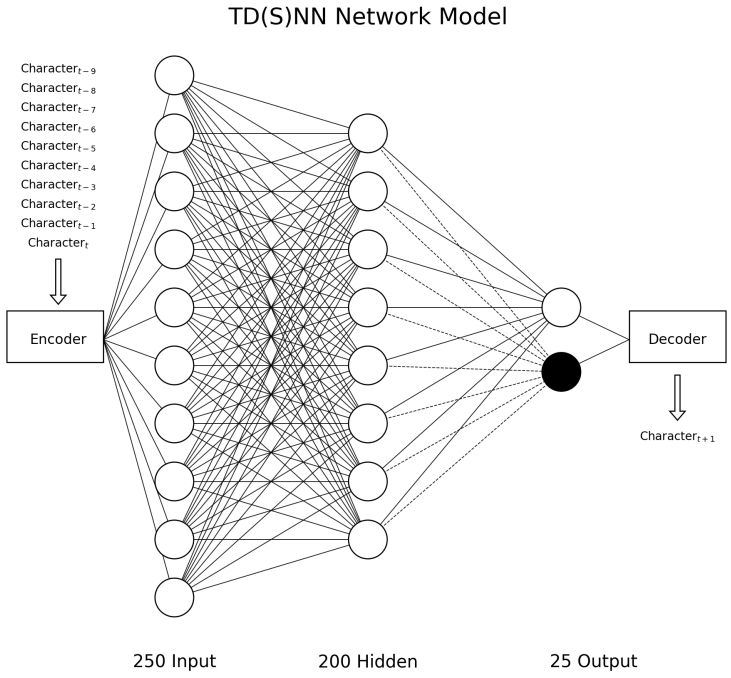

Figure 3: The network architecture for the TDNN/TDSNN model.

Time-delay spiking neural networks (TDSNN)

A common approach to training spiking neural networks is through the transfer of weights learned through backpropagation on an identical non-spiking neural network and then scaling the weights through data-based normalization. Using this approach, we can convert the TDNN model trained with backpropagtion into an equivalent spiking network. [**rueckauer2017conversion**]

Conversion

The converted network has neurons that output discrete spikes rather than continuous values. Each neuron accumulates voltage v from the previous layer until the voltage reaches a threshold. After the voltage surpasses the threshold, the neuron emits a discrete spike to the next layer and the voltage is reset by subtraction of the threshold. [**diehl2015fast**]

The change in the voltages of each of the neurons in the hidden layer at time t can be written in terms of the input x_t, the scale for the connection between

the input and hidden layer found through the data-based normalization λ_{ih}, the transferred connection and bias weights from the TDNN (W_{ih}, b_i), and the neuron spike threshold τ as:

$$\frac{dV_t^{(h)}}{dt} = \lambda_{ih}\left(W_{ih}^T x_t + b_i\right) - S_{t-1}^{(h)}\tau \tag{8}$$

where $S_t^{(h)}$ are the spikes in the hidden layer at time t. For each neuron j in the hidden layer, the spike is defined as:

$$S_t^{(h,j)} = \begin{cases} 1 & V_t^{(h,j)} \geq \tau \\ 0 & \text{else.} \end{cases} \tag{9}$$

Similarly, the following equation models the change in the voltages of the neurons in the output layer of the TDSNN:

$$\frac{dV_t^{(o)}}{dt} = \lambda_{ho}\left(W_{ho}^T S_t^{(h)} + b_h\right) - S_{t-1}^{(o)}\tau \tag{10}$$

where λ_{ho} is the scale found through the data-based normalization for the connection between the hidden and output layer. $S_t^{(o)}$ is the spikes in the output layer at time t which is defined for each neuron j as:

$$S_t^{(o,j)} = \begin{cases} 1 & V_t^{(o,j)} \geq \tau \\ 0 & \text{else.} \end{cases} \tag{11}$$

Original Readout

The original readout method proposed by Rueckauer et al. looks at the voltage of the output layer rather than the spikes. This readout method combined with the the subtractive reset mechanism of the spiking neurons in the conversion method perform well on various tasks, [**rueckauer2017conversion**] but in some cases we may not want to use the voltages. A more biologically inspired alternative for approximating the output could be the average sum of spikes over the total runtime T in the output layer. This approximation of the TDNN output values can be written as:

$$\text{TDSNN}_{\text{readout1}} = \frac{1}{T}\sum_{t=1}^{T} S_t^{(o)} \tag{12}$$

Negative Readout

When the output values produced by the TDNN is negative, the average sum of spikes fails to account for the negative output values. To account for these negative spikes, the output layer was doubled in size and the negative weight matrix of the original connections was concatenated. The same was done for the bias. The positive sum of spikes was subtracted from the negative sum of

spikes and then divided by the runtime T to obtain an approximation of the original network's output.

For a TDNN that has l neurons in the output layer, the change in voltage for each neuron j in the output layer is changed to:

$$\frac{dV_t^{(o)}}{dt} = \begin{cases} \lambda_{ho}\left(W_{ho}^T S_t^{(h)} + b_h\right) - S_{t-1}^{(o,1:l)}\tau & 1 \leq j \leq l \\ \lambda_{ho}\left(-W_{ho}^T S_t^{(h)} - b_h\right) - S_{t-1}^{(o,l+1:2l)}\tau & l+1 \leq j \leq 2l \end{cases} \tag{13}$$

In Equation **??**, W_{ho} refers to the original TDNN's weight matrix and $S_t^{(o,j)}$ now has $j \in [1, 2l]$. The new readout is now:

$$\text{TDSNN}_{\text{readout2}} = \frac{1}{T}\sum_{t=1}^{T}\left[S_t^{(o,1:l)} - S_t^{(o,l+1:2l)}\right] \tag{14}$$

Conversion Loss

The drawback of using spikes instead of voltages to approximate real-values is the loss of precision based on the duration of the input. For this task, we show the input for 500 simulated milliseconds. The conversion loss, defined as the euclidean distance between the output of the TDNN and the TDSNN is shown in Figure **??**.

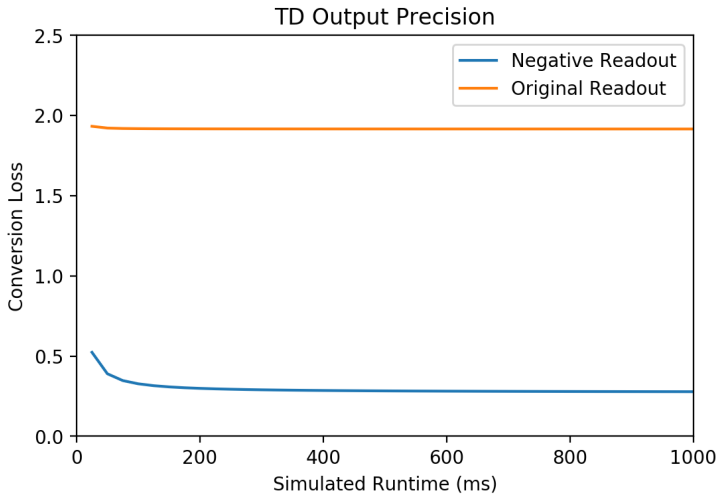

Figure 4: Average distance between the output of the TDNN and the TDSNN for different runtimes over the $20,000$ elements in the artificial dataset.

Semi-supervised Models

K-Nearest Neighbors (KNN)

The K-Nearest Neighbors (KNN) model makes a prediction $\hat{d}_{t+1}$ on an input x_t at time t, where x_t is the concatenation of the encodings of the characters d_i for $i \in [t-9, t]$. At time $t+1$, the KNN stores the tuple (x_t, d_{t+1}). To make a prediction, the KNN computes the Euclidean distance from x_t to all other samples. The majority vote of the top k samples is taken as the prediction for d_{t+1}.

Columnar Spiking Neural Network (CSNN)

While conversion is one way to train a spiking neural network, it is a supervised method that is still not energy efficient on hardware implementations. We propose a spiking neural network architecture, with k inputs, each fully connected to 1 of k columns of L Leaky Integrate-and-Fire (LIF) neurons with weights initialized uniformly randomly from $[0, 0.3]$. [**gerstner2002spiking**] Each neuron in each column has a zero-weight connection to each neuron in each of the other columns. The inter-column connections have a Hebbian-inspired update rule. To map the spikes from this network, we use the K-Nearest Neighbors model with the sum of spikes of each neuron in each column over the runtime T. [**beliaev2007time**] The model diagram is shown in Figure **??**.

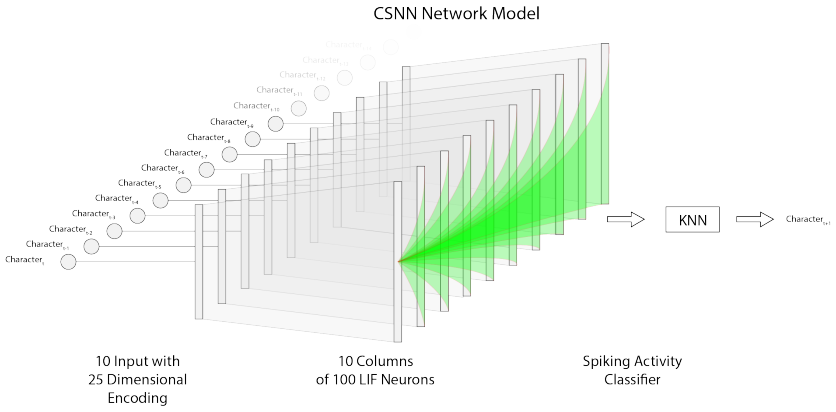

Figure 5: The network architecture for the CSNN model.

Hebbian Learning Rule

The Hebbian learning rule for the CSNN model is defined in terms of the spiking activity of its k columns over the runtime T. The sum of spikes in a column c over the runtime is defined as:

$$\gamma_c = \sum_{t=1}^{T} s_t^{(c)} \tag{15}$$

where $s_t^{(c)}$ is a binary vector that represents the spiking neurons at time t in column c. The mean spiking activity for each neuron $\tilde{s}$ is then defined as:

$$\tilde{s} = \frac{1}{k} \sum_{c=1}^{k} \gamma_c \tag{16}$$

The weight change on spike in connections from column a to column b for learning rate η is defined as follows:

$$\Delta w_{a,b} = \eta * (\gamma_a - \tilde{s}) \otimes (\gamma_b - \tilde{s}) \tag{17}$$

The weight updates for this model are computed at the end of each input rather than continuously.

Results

Supervised Models

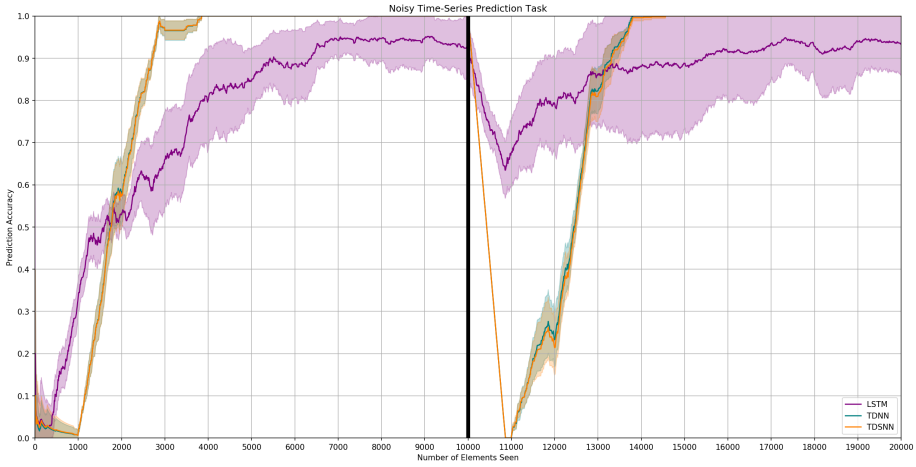

Figure 6: The TDNN (teal) and TDSNN (orange) models are able to perfectly predict the sequence endings after $4,000$ elements, but their performance drops dramatically after the sequence endings are swapped. The LSTM (purple) is able to reach a reasonable performance on this task before the task and the drop after the sequences endings are swapped is only about 30%.

Semi-supervised Models

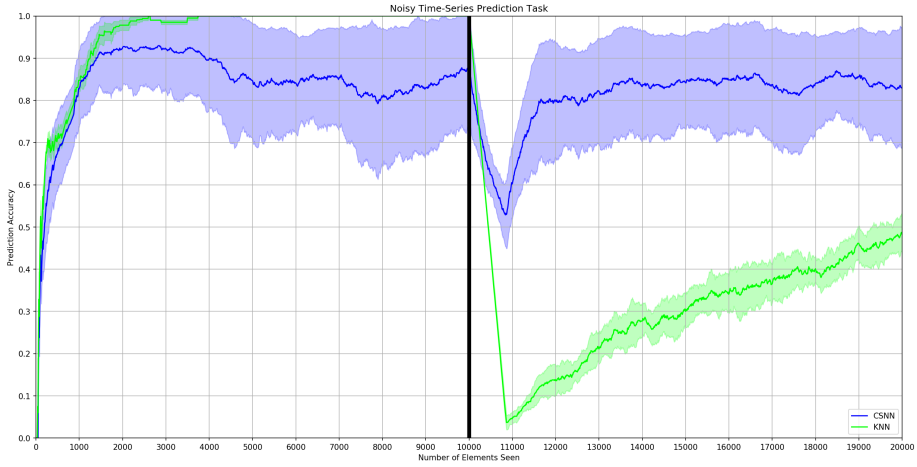

Figure 7: The KNN (green) reaches perfect accuracy before the dataset is endings are swapped, but fails to relearn the data and only reaches 50% accuracy afterward. The CSNN never manages to reach perfect accuracy on the task, but it is able to re-learn the sequence endings after they are swapped after experiencing a drop in performance similar to the LSTM in Figure **??**.

Conclusion

Model	Average Accuracy
LSTM	0.8049 ± 0.088
TDNN	0.8112 ± 0.012
TDSNN	0.8097 ± 0.013
KNN	0.6391 ± 0.021
CSNN	$\mathbf{0.8250} \pm 0.119$

Table 2: The average accuracy on the discrete sequence learning task over 10 randomized runs.

We have shown two possible spiking architectures for learning changing patterns in time-series data. We tested each of the models (LSTM, TDNN, TDSNN, KNN, CSNN) on the discrete sequence learning task. The TDSNN (orange) is able to match the TDNN (teal) Surprisingly, the CSNN (blue) performance over 10 runs has the highest average accuracy over the sequence learning task, but it also has the average standard deviation over the 10 runs.