University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 22: November 8

Lecturer: Alexander Schwing Scribe: Kanika Narang

1 Autoregressive Methods

The models discussed till now can be broadly categorized into two categories:

1. Discriminative Models : Given the input data x, learn a model which maximizes the probability of it's associated output variable y, $\max_{\mathbf{y}} p(\mathbf{y}|x)$

2. Generative Models : Given the input data x, model the data using latent variables θ which maximize the log likelihood of the input, $\max_{\theta} p(x;\theta)$.

These class of models however are unable to model time dependence. There can be time dependence in input sequence, for instance machine translation, next word depends on the previous words in the output or output sequence, e.g. image captioning where words in the output caption depend on the previous words generated. Autoregressive methods model joint distribution over input/output variables y, i.e.

$$P(y_1 \dots y_d) = \prod_{t=1}^{d} p(y_t|y_{<t})$$

where $y_1, \dots, y_d$ is an input/output sequence of length d. Each input/output variable y_t is modeled as a conditional distribution over all the previous $y < t$ variables.

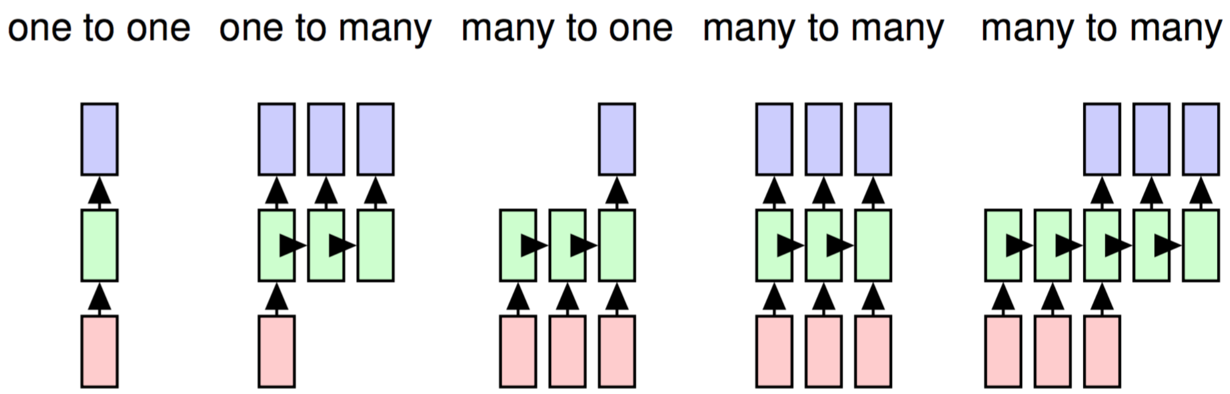

Figure 1: **Multiple architectures of Autoregressive models. Red blocks denote the input, green blocks denote the hidden states and blue blocks denote the output.**

Autoregressive models are thus flexible and allow for variable length of input and output. Thus these models can be further divided into five categories as shown in Figure 1;

- one-to-one : Length of input and output data is same. E.g. Recommender Systems where we recommend next item for each current item.

- one-to-many : Predict sequence of outputs for single input. E.g. Image Captioning systems where static image is used to generate sequence of words i.e. caption.
- many-to-one : Sequence of inputs is used to predict a single output. E.g. Sentiment prediction of reviews where input text is fed one by one into the model and final cell predict a sentiment score.
- many-to-many : Length of Input and Output sequences are different. E.g. Machine Translation models where a sentence in one language is translated into a sentence of another language. The length of sentence in both language may not be the same.

We now discuss in detail popular autoregessive models proposed in the literature.

1.1 Recurrent Neural Nets

The most popular autoregressive methods in deep learning is Recurrent Neural Networks (RNNs). They employ markovian assumption where current input depends on the previous output. Formally, hidden state at time t, h_t depends on previous hidden state h_{t-1}, current input x_t and model parameters $\mathbf{w}$. The hidden state is then used to predict the final output variable $y^{(t)}$.

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, \mathbf{w}) \tag{1}$$

$$y^{(t)} = g(h^{(t)}) \tag{2}$$

RNN models are used in applications where modeling sequence of input or output is important. They are thus extensively used in Natural Language Processing, Speech Recognition, Image Processing and Video Processing.

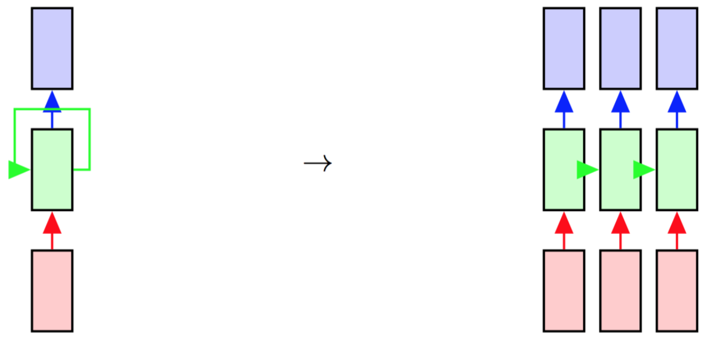

Figure 2: **Basic block of RNN architecture. The figure on left shows the folded architecture. The self loop in the middle emphasizes the parameter sharing aspect of RNN model. Weights and biases in RNN cell are shared across timesteps. The figure on right shows the unfolded architecture. As shown by the forward arrows, each hidden state in RNN cell depends on the hidden state in previous RNN cell.**

RNNs can be thought of as very deep neural networks where each timestep represents a hidden layer. As the input sequence can be very long, the hidden layers can be large. Thus such models can suffer from vanishing gradient problem and are also computationally expensive because of large number of learnable parameters causing overfitting. To alleviate these issues, RNN models **share parameter** across timesteps. At each timestep t, $\mathbf{w}$ in eq 2 remains the same leading to identical operations across time. Thus, the input still depends on the previous output but are more tractable to learn due to shared parameters. Also note that functions f and g remains constant across time.

RNN based methods use the general framework discussed in eq 2 and different versions differ in their specific implemetation of f, g functions and w weights. Note that the functions f and g can be any differentiable functions.

We will now discuss in detail most popular three reccurent neural nets; Original Recurrent Neural networks, Long-Short Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs).

1.2 Original RNNs

RNNs predict output sequence using affine transformation of the input and apply point-wise non linearity. Specifically, Original RNNs also referred to as Elman network [2] compute hidden state at time t, $h^{(t)}$ as,

$$h^{(t)} = \sigma_h(W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + w_{hb}) \tag{3}$$

$$y^{(t)} = \sigma_y(W_{yh}h^{(t)} + w_{yb}) \tag{4}$$

with W_* as the different learnable weight matrices and w as learnable bias parameter. σ_h and σ_y are activation functions used to enforce non-linearity. Typical choices of activation functions used are tanh and sigmoid.

The original RNN framework has a series of shortcomings,

1. Vanishing Gradients - As earlier stated, RNNs are very deep neural network where depth is equal to the number of timesteps in the sequence. Therefore, gradients with respect to lower layers goes to zero and it takes very long to train the lower layers or initial timesteps.

2. Long Term Dependency - Original RNNs do not model higher order relationships as they are just modeled as *affine* transformations of the input. Also as current output $h^{(t)}$ to the RNN just depends on previous timestep $h^{(t-1)}$, hidden state has to learn to represent longer past just than the previous step in case of longer dependencies. For instance, when predicting the next word in a sentence, modeling the sequence of words till the current word will result in more meaningful sentence than just the previous word.

1.3 Long-Short Term Memory Networks

Long Short Term Memory Networks (LSTM) were proposed to overcome the previously discussed issues [3]. They have been empirically proven to better capture long term dependencies and address the vanishing gradient problems seen in original RNNs. Formally, they compute the hidden state at timestep t, $h^{(t)}$ as,

$$\begin{aligned}
i^{(t)} &= \sigma_i(W_{ix}x^{(t)} + W_{ih}h^{(t-1)} + w_{bi}) && \text{Input gate} \\
f^{(t)} &= \sigma_f(W_{fx}x^{(t)} + W_{fh}h^{(t-1)} + w_{bf}) && \text{Forget gate} \\
o^{(t)} &= \sigma_o(W_{ox}x^{(t)} + W_{oh}h^{(t-1)} + w_{bo}) && \text{Output/Exposure gate} \\
\tilde{c}^{(t)} &= \sigma_c(W_{cx}x^{(t)} + W_{ch}h^{(t-1)} + w_{bc}) && \text{New Memory Cell} \\
c^{(t)} &= f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)} && \text{Final Memory Cell} \\
h^{(t)} &= o^{(t)} \circ \sigma_h(c^{(t)})
\end{aligned}$$

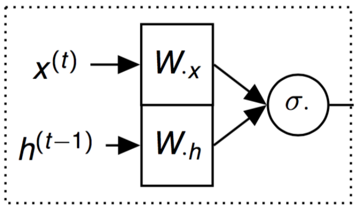

Figure 3: **Basic structure of Input gate $i^{(t)}$, Forget gate $f^{(t)}$ and Output gate $o^{(t)}$. They all are functions of current input $x^{(t)}$ and previous hidden state $h^{(t-1)}$ parametrized by their respective weight matrices $W_{.x}$ and $W_{.h}$ followed by σ RelU non-linearity.**

where $\circ$ denotes the Hadamard product and σ is the activation function.

All the three gates, Input gate $i^{(t)}$, Forget gate $f^{(t)}$ and Output gate $o^{(t)}$ are functions of current input $x^{(t)}$ and previous hidden state $h^{(t-1)}$(similar to Eq 3) as shown in Figure 3. Input Gate computes how much attention to pay to the input while Forget gate decides how much to forget from the past. New memory cell $\tilde{c}^{(t)}$ denotes memory cell of current timestep and is computed in similar way to Eq 3 in Original RNN. The final memory cell $c^{(t)}$ is linear combination of present memory cell $\tilde{c}^{(t)}$ and previous final memory cell $c^{(t-1)}$. The weights for current memory cell and previous memory cell are defined by the input gate and forget gate respectively. The final hidden state h^t is a hadmard product of output gate and the computed new memory cell c^t. Output gate acts as a further restriction on the final memory cell $c^{(t)}$ to determine how much it should effect in the final hidden state. The basic LSTM cell structure is shown as a logic circuit diagram in Figure 4.

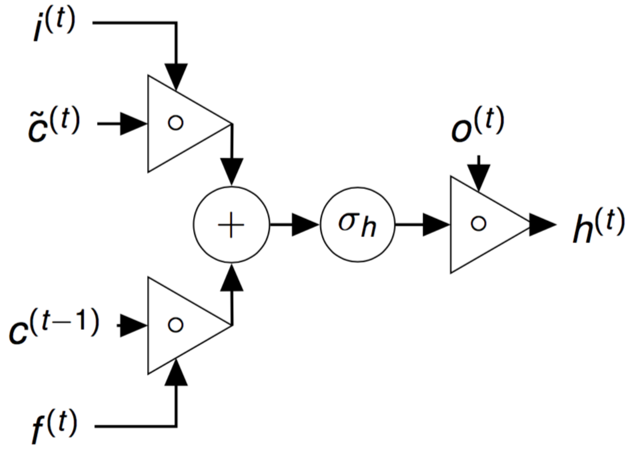

Figure 4: **Logic circuit diagram showing computation of final hidden state $h^{(t)}$ in a LSTM cell. $\circ$ denotes the Hadmard product while σ^h denotes the activation function.**

Although there are multiple variables inside a basic cell of LSTM, they are indepenedent of the other timesteps and can be denoted as a separate block in neural network as shown in Figure 5. Thus each LSTM cell can be trained parallely and independently of the other timesteps. This is similar to the original RNN framework.

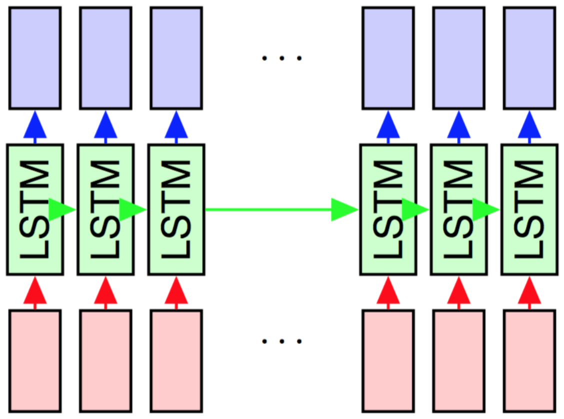

Figure 5: **Each LSTM cell can be represented as a block in the neural network.**

1.4 Gated Recurrent Units

Although LSTMs are shown to perform better in practice to capture long term dependencies in sequential data, they are computationally expensive due to large number of parameters. Gated Recurrent Units (GRU) proposed by [1] perform similarly to LSTM but have fewer parameters than LSTM. Specifically, they do not have an output gate. Following are the update equations of GRU cell,

$$
\begin{aligned}
z^{(t)} &= \sigma_z(W_{zx}x^{(t)} + W_{zh}h^{(t-1)} + w_{bz}) && \text{Update gate} \\
r^{(t)} &= \sigma_r(W_{rx}x^{(t)} + W_{rh}h^{(t-1)} + w_{br}) && \text{Reset gate} \\
\tilde{h}^{(t)} &= \sigma_h(W_{hx}x^{(t)} + W_{rwh}(r^{(t)} \circ h^{(t-1)}) + w_{bh}) && \text{New Memory Cell} \\
h^{(t)} &= (1 - z^{(t)}) \circ \tilde{h}^{(t)} + z^{(t)} \circ h^{(t-1)} && \text{Hidden State}
\end{aligned}
$$

where $\circ$ denotes the Hadamard product and σ is the activation function.

Similar to LSTM, there is an Update gate $z^{(t)}$ and Reset gate $r^{(t)}$ which are linear function of current input $x^{(t)}$ and previous hidden state $h^{(t-1)}$ similar to Figure 3. New Memory Cell $\tilde{h}^{(t)}$ is computed in a similar way to the gates but Reset gate decides how much of previous hidden state $h^{(t-1)}$ to be included in the new memory cell. Final hidden state $h^{(t)}$ is linear combination of new memory cell $\tilde{h}^{(t)}$ and previous hidden state $h^{(t-1)}$ weighed by the update gate $z^{(t)}$. Note that it is similar in spirit to LSTM where gates decide the contribution of previous memory and current input. Also similar to LSTM, GRU cell can be interpreted as a separate block in the neural model.

There are other variants proposed of recurrent networks in the literature. Notable ones are bi-directional LSTM [] which stacks two LSTMs on top of one another with opposite directions (forward and backward) of dependencies. This is shown to be useful specially in Natural Language Processing and allows the model to see future and past words. Continuous Time RNN [] is another pouplar variant that does not use discrete timesteps for modeling sequential data.

1.5 Training RNN

RNNs model a sequence of inputs that depends on the past inputs. Assume the sequence can be defined as $y_1, \ldots, y_T$, then,

$$p(y_1, \ldots, y_T) = \prod_{i=1}^{T} p(y_i | y_1, \ldots y_{i-1})$$

$$\log(p(y_1, \ldots, y_T)) = \sum_{i=1}^{T} \log(p(y_i | y_1, \ldots y_{i-1}))$$

RNNs training use loss function that optimizes the maximum log likelihood of obtaining the sequence. RNNs are trained using backpropagation through time (BPTT). Specifically, backpropagation is performed in the backward direction, from end of the sequence to the start as hidden state in RNN at each timestep $h^{(t)}$ is dependent on the last timestep's hidden state $h^{(t-1)}$. Each intermediate layer stores the hidden state $h^{(t)}$ of it's timestep to compute the final loss of the whole sequence. This can lead to large memory requirements if the sequences are particularly large. Therefore, RNNs can't have very complex relationships. Although, their current formulation that involves affine transformations with point wise non linearity has shown to work well in practice for sequence modeling.

1.6 Pixel RNN

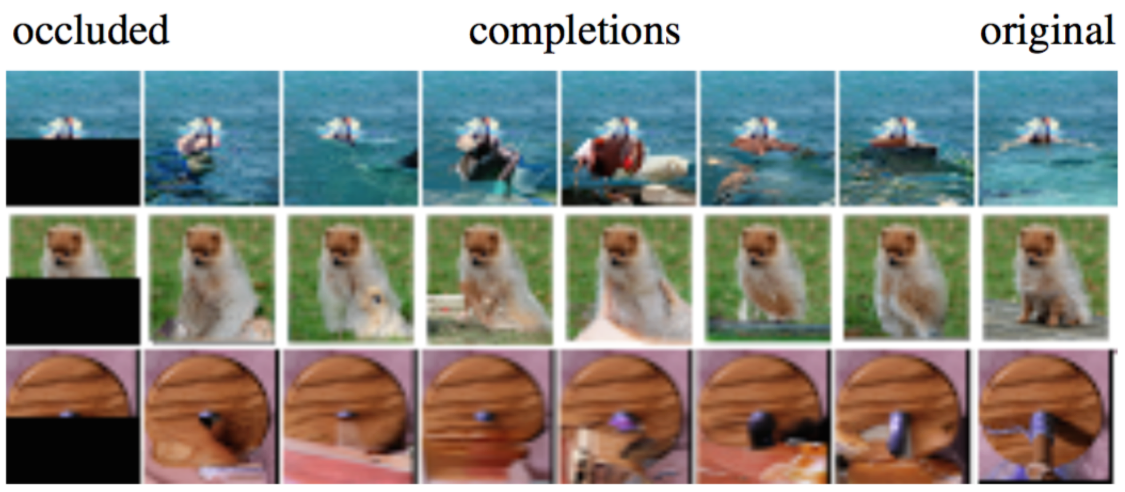

Figure 6: **Output of the Autoregressive PixelRNN model which predicts missing pixels in the image shown as black in the left most figure. Pixel RNN models conditions the current pixel value on all previously seen pixels. Middle figure shows the multiple completion sampled from the model while the right figure the original image. The completions are not necessarily similar to the original image but are valid completions. Therefore, PixelRNN is a powerful generative model.**

Autoregressive models like RNNs are used as generative models.Figure 6 shows output of an autoregressive PixelRNN model [5] that models available pixels as timeseries and predicts future or missing pixels in the image.This approach is powerful as they are able to generate various plausible completions of the image.They are not necessarily same as the original image but though are plausible completions.

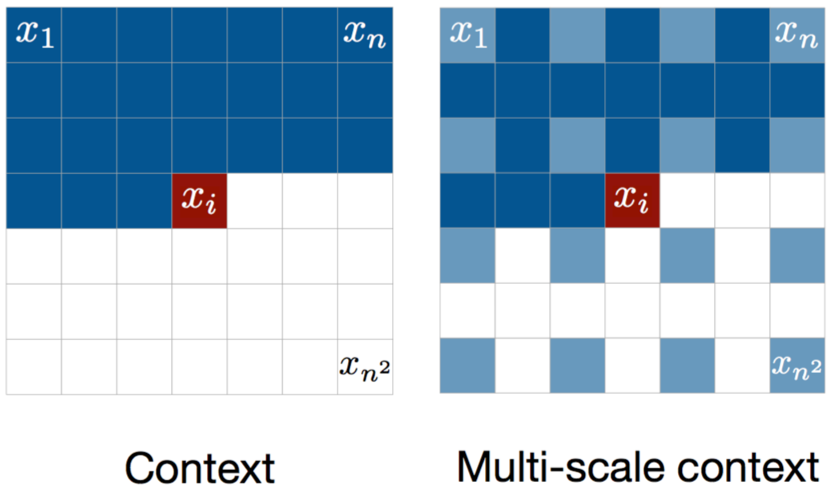

Figure 7: **Pixel RNN model conditions probability of the current pixel x_i on all the previous pixels to the left and top of x_i. Multi-Scale Pixel RNN also provides as input a subsampled image (shown in light blue) along with preceding pixel values to improve prediction.**

Pixel RNN conditions probability of the current pixel x_i on all the previous pixels to the left and top of x_i as shown in Figure 7. To predict the conditional distribution over the possible pixel values for x_i, the model scans the image one row one pixel at a time in each row before x_i. Like in RNNs formulation, the final probability is product of such conditional distributions and these values are shared across all the pixels of the image. Therefore probability of all pixels $\mathbf{x}$ can be defined as,

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i|x_1, \ldots, x_{i-1})$$

where n^2 is the number of pixels in a $n \times n$ image.

Another variant, Multi-Scale Pixel RNN [5] also provides as input a subsampled image (shown in light blue) in Figure 7 along with preceding pixel values to improve prediction.

1.7 Generative Model comparison

We now compare the three types of Generative models discssed in the past few lectures.
Variational Auto-Encoder (VAEs)

- Pros : VAEs model can be interpreted as a probabilistic graphical model where posterior and model likelihood are parametrized by neural nets. It also learns a latent variable z for each datapoint separately. They model explicit distributions of the data.
- Cons : VAEs however are prone to produce slightly blurry images.

Generative Adversial Nets (GANs)

- Pros: GANs generate sharp images as they maximize the probability of generated images with respect to the discriminator. In contrast to VAEs, GANs model implicit distributuions of the data.
- Cons: However, they are difficult to optimize as the model parameters may oscillate and will not converge.

Autoregressive Models (RNNs)

- Pros: The training of RNN is relatively stable and it maximizes log likelihood of the training data.
- Cons: RNNs do not project the data into low-dimensional space as VAE so it is difficult to analyze intermediate representations and infer what the model learns. When generating output from RNNs, most likely output given the conditional distribution is chosen which is an inefficient sampling strategy as it produces only limited set of samples.

All the three approaches are very active research areas currently. And many modifications to the original formulation have been proposed to alleviate these discussed issues.

We now move to very recent area of research, Graph Convolutional Networks.

1.8 Graph Neural Networks

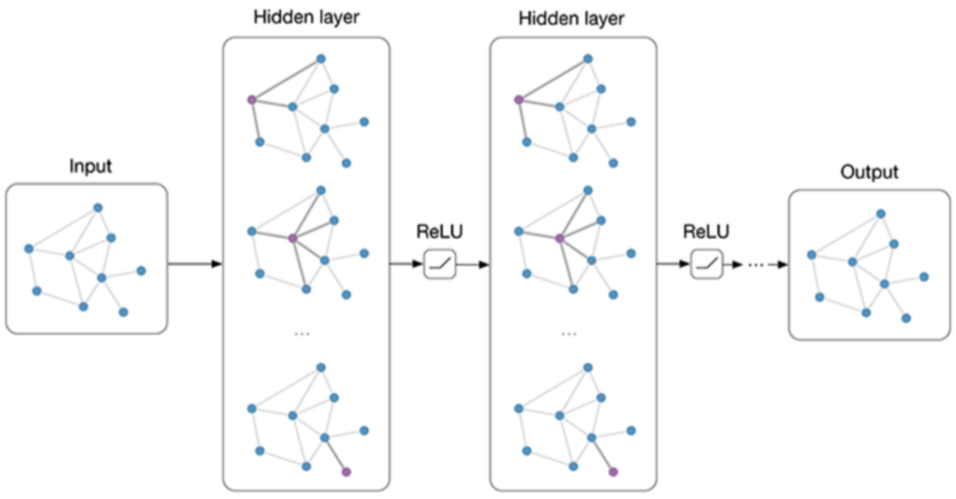

Figure 8: **Graph Convolution Networks proposed to compute node representations. Each hidden layer aggregates information from it's neighbors and is followed by ReLU non-linearity. The final node representation is computed by stacking such hidden layers.**

Graph Convolution Neural networks are a recent class of convolution approaches proposed to use for irregular graphs. Graph Convolution Network (GCN) proposed by [4] stacks series of convolution layers followed by non-linearity to compute convolved representations of a node. Each convolution layer computes representation of a node by aggregating current hidden representation from all it's neighboring nodes. Figure 8 represents the architecture of Graph Convolution Neural network.

Formally, output of $l + 1$th layer of Graph Convolution Network can be defined as,

$$H^{(l+1)} = f(H^{(l)}, A) \tag{5}$$

where $H^{(l)}$ represents the hidden representation of nodes in the graph at lth iteration. For the first layer, hidden matrix just represents the node features X itself, i.e. $H^{(0)} = X$. A is the adjacency graph and $f(.,.)$ represents non-linearity.

The original Graph Convolution Network approach uses the following form,

$$H^{(l+1)} = \sigma(AH^{(l)}W) \tag{6}$$

Therefore, it multiplies the hidden matrix $H^{(l)}$ with adjacency graph A and project it into some lower dimension space using weight matrice W. They employ ReLU non-linearity on top of it. More complex formulations of basic Equation 5 have also been proposed.

References

[1] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[2] J. L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

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

[4] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[5] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel recurrent neural networks. In *ICML*, 2016.