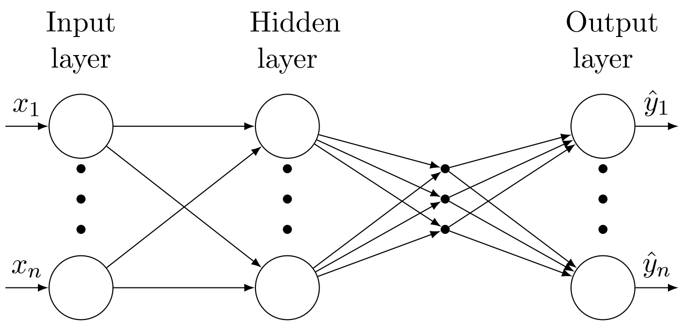
EECS 16ML Fall 2020 — Introduction to RNNs — Team Chairun — Note

1 Recurrent Neural Networks

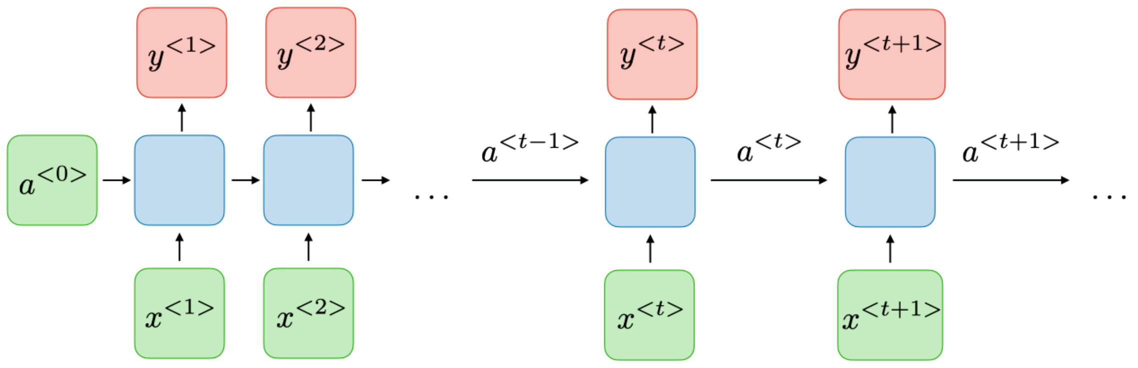
We have already seen how neural networks can be successfully applied to many different regression and classification problems. Recall that if we have a datapoint $\mathbf{x} \in \mathbb{R}^n$, we can represent the action of a neural network on it by a directed acyclic graph as follows:



This kind of neural network is typically referred to as a **feedforward neural network**. Notice that each datapoint is propagated independently from the rest by the network. That is, it is not possible to capture any sequential dependence between datapoints in our training dataset. This causes us to run into problems when we try to apply this model to learning tasks related to sequential data such as those dealing with language and time-series data, since each datapoint is strongly dependent on the preceding (and sometimes upcoming) datapoints. For example, consider the task of translating the English sentence "When are they coming?" to French. As you are probably aware, simply converting each word to its French equivalent does not typically yield a meaningful translation because words and grammar are inherently dependent on the surrounding context. In this example, without knowing what the subject of the sentence is, we run into ambiguities when trying to conjugate the verbs "are" and "coming".

To take into account surrounding context and sequentially process inputs in cases like this, we introduce a new architecture called a **recurrent neural network** (RNN). Suppose we have sequential data $\mathbf{x}^{\langle 1 \rangle}, \ldots, \mathbf{x}^{\langle t \rangle}$. For example, each $\mathbf{x}^{\langle i \rangle}$ could be a word in a sentence that we are trying to translate. The main idea behind an RNN is to compute an output $\hat{\mathbf{y}}^{\langle i \rangle}$ using not only the corresponding input $\mathbf{x}^{\langle i \rangle}$ but also the output from the previous element in the sequence, $\hat{\mathbf{y}}^{\langle i-1 \rangle}$. We can illustrate a traditional RNN architecture as follows [1]:

[1]Figure from https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks



Recall that in a feedforward neural network, the output of a layer ℓ can be expressed as $\mathbf{a}_\ell = \mathbf{g}_\ell(\mathbf{W}_\ell \mathbf{a}_{\ell-1})$, where $\mathbf{g}_\ell$ is the activation function for that layer and $\mathbf{W}_\ell$ is the weight matrix that we want to learn. From the above diagram then, we see that in a recurrent neural network, we now have

$$\mathbf{a}^{\langle t\rangle} = \mathbf{g}_1(\mathbf{W}_a \mathbf{a}^{\langle t-1\rangle} + \mathbf{W}_x \mathbf{x}^{\langle t\rangle})$$
$$\hat{\mathbf{y}}^{\langle t\rangle} = \mathbf{g}_2(\mathbf{W}_y \mathbf{a}^{\langle t\rangle})$$

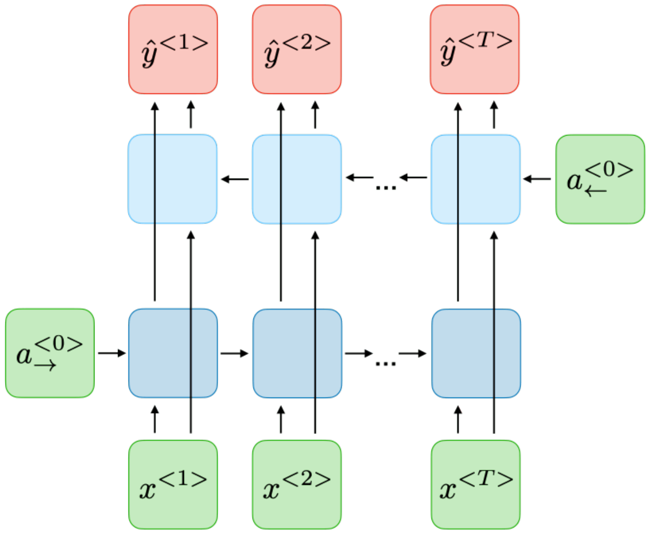
where we now have two activation functions $\mathbf{g}_1$ and $\mathbf{g}_2$ at each stage and three weight matrices $\mathbf{W}_a, \mathbf{W}_x, \mathbf{W}_y$ that are shared across temporal stages.

Depending on the application, we may want output $\mathbf{y}^{\langle t\rangle}$ for all t or just for one specific timestep (usually the last). For example, in a machine translation setting, we are interested in inputting a sequence and getting out an output sequence that is an accurate translation of it. That is, given $\mathbf{x}^{\langle 1\rangle}, \dots, \mathbf{x}^{\langle t\rangle}$, we want to obtain $\mathbf{y}^{\langle 1\rangle}, \dots, \mathbf{y}^{\langle t\rangle}$. This is sometimes called a **many to many RNN**. On the other hand, in a task like sentiment classification, we are interested in passing in a sequence and determining what its overall 'sentiment' (or tone) is. That is, given $\mathbf{x}^{\langle 1\rangle}, \dots, \mathbf{x}^{\langle t\rangle}$, we want to obtain a single output $\mathbf{y}^{\langle t\rangle}$. This is sometimes called a **many to one RNN**.

Weights are updated as usual by the backpropagation algorithm for neural networks, but with one small difference – the backpropagation is carried out independently for each time step. This is sometimes called **backpropagation through time**. Consequently, we also define the loss function independently at each time step, denoted by $\mathcal{L}(\hat{\mathbf{y}}^{\langle t\rangle}, \mathbf{y}^{\langle t\rangle})$. Notice that since the model is sequential, the total error at time T is then just given by $\mathcal{L}^{\langle T\rangle} = \sum_{t=1}^{T} \mathcal{L}(\hat{\mathbf{y}}^{\langle t\rangle}, \mathbf{y}^{\langle t\rangle})$. By the linearity of derivatives, we then see that $\frac{\partial \mathcal{L}^{\langle T\rangle}}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{\langle t\rangle}}{\partial \mathbf{W}}$ for one of the weight matrices $\mathbf{W}$. Thus for each time step T, the backpropagation through time algorithm iteratively minimizes this gradient for all $\mathbf{W}$ (and any bias terms).

Tasks like machine translation, sentiment classification, and speech recognition fall within the field of Natural Language Processing, where RNNs are abundantly used due to the contextual nature of language.

From the traditional RNN architecture above, we can easily see how this captures relevant information about the dependence of $\mathbf{y}^{\langle t\rangle}$ on the previous inputs in the sequence $\mathbf{x}^{\langle 1\rangle},\ldots,\mathbf{x}^{\langle t-1\rangle}$. But for tasks like machine translation, it is natural to wonder: what if an element in a sequence depends on elements that come after it? It turns out that we can easily modify our traditional architecture to accomodate this by adding another sequential hidden layer, with the sequence reversed as illustrated below [2]. Models with this architecture are referred to as **bidirectional recurrent neural networks** (Bi-RNNs).



2 Gated Recurrent Units

An issue that arises often when dealing with long sequences in RNN models is the **vanishing gradient problem**. The same issue can arise with deep feedforward networks but comes up more often in RNN models because they are typically much longer. Intuitively, you can think of "unrolling" an RNN into a very deep feedforward network. Since backpropagating errors through deep networks involves the multiplication of several partial derivatives (from the chain rule for partial derivatives), as we approach convergence, these terms become very small and so their product goes to 0. Once this happens, gradient descent can no longer update any weights and so no more learning can take place. Thus, using a traditional RNN architecture, it is hard to capture any long-term dependencies in the data sequence.

A **Gated Recurrent Unit** (GRU) model fixes this problem by introducing two "gate" vectors and a "candidate activation vector" that are able to retain long-term information without succumbing to the vanishing gradient problem. The first gate, termed the **update gate**, introduces new weights $\mathbf{U}_z$ and $\mathbf{W}_z$

[2]Figure from https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

and performs the computation

$$\mathbf{z}^{\langle t\rangle} = \sigma(\mathbf{W}_z\mathbf{x}^{\langle t\rangle} + \mathbf{U}_z\mathbf{h}^{\langle t-1\rangle})$$

where $\mathbf{h}_{t-1}$ is a vector that holds some information about the previous $t-1$ elements in the sequence and σ is the sigmoid activation function. The second gate, termed the **reset gate** introduces weights $\mathbf{U}_r$ and $\mathbf{W}_r$ to perform the computation

$$\mathbf{r}^{\langle t\rangle} = \sigma(\mathbf{W}_r\mathbf{x}^{\langle t\rangle} + \mathbf{U}_r\mathbf{h}^{\langle t-1\rangle})$$

Lastly, the **candidate activation** introduces weights $\mathbf{U}_{\hat{h}}$ and $\mathbf{W}_{\hat{h}}$ to compute

$$\hat{\mathbf{h}}^{\langle t\rangle} = \tanh(\mathbf{W}_{\hat{h}}\mathbf{x}^{\langle t\rangle} + \mathbf{r}^{\langle t\rangle} \odot \mathbf{U}_{\hat{h}}\mathbf{h}^{\langle t-1\rangle})$$

where $\odot$ represents element-wise multiplication (or the Hadamard product). Finally, the GRU outputs

$$\mathbf{h}^{\langle t\rangle} = \mathbf{z}^{\langle t\rangle} \odot \mathbf{h}^{\langle t-1\rangle} + (\mathbf{1} - \mathbf{z}^{\langle t\rangle}) \odot \hat{\mathbf{h}}^{\langle t\rangle}$$

Intuitively, you can interpret this as a weighted sum dictating how much information to forget from the previous timestep and how much to pass on.

GRU models turn out to be extremely robust against the vanishing gradient problem and generalize to another widely used model called **Long Short-Term Memory** (LSTM) but we won't get into that now. Additionally, just like with traditional RNNs, we can add a sequentially reversed hidden layer on top of the original one to construct a **bidirectional GRU**, or a Bi-GRU.

3 Coding Assignment: Learning stock embeddings for price movement classification using bidirectional RNNs

Predicting stock prices is one of the biggest challenges in quantitative finance, and with the recent explosion of ML, several investors have begun to incorporate deep computational models like neural networks to try and 'beat' the market – a problem that is notoriously hard. However, these attempts have mostly obtained mixed results (link some paper here). Instead of looking at it from the prediction point of view, what if we treated the problem from the classification point of view instead?

This assignment is based on a paper by Xin Du and Kumiko Tanaka-Ishii at the University of Tokyo [3] about learning a relationship between news reported about a company over a short timespan and changes in its stock price in the same time period. Du and Tanaka-Ishii explore the possibility of training a Bi-GRU model to classify stock movement on each day by using news articles from the previous 5 days as input data for the model.

[3]Stock Embeddings Acquired from News Articles and Price History, and an Application to Portfolio Optimization https://www.aclweb.org/anthology/2020.acl-main.307.pdf

For the purposes of this assignment, we will focus on training a classifier for the "AAPL" stock from the S&P 500. The goal of our classifier is as follows:
We are interested in training a Bi-GRU model that learns a relationship between news taglines related to the stock that we have selected and the prices of those stocks. Define $p^{(t)}$ to be the price of the stock on day t. Then, we can formally define our objective as follows:
Let $y^{(t)} = \begin{cases} 1 & \log(p^{(t)}) \geq \log(p^{(t-1)}) \\ 0 & \log(p^{(t)}) < \log(p^{(t-1)}) \end{cases}$. We use log returns to define price movement because it is a good approximator for percentage change. Suppose our dataset $D = \{N^{(t)}\}_{t_{in} \leq t \leq t_f}$, where $N^{(t)}$ is a collection of all the articles from day t and t_{in} and t_f represent the dates of the earliest and latest articles in our dataset respectively. Then, we want to learn a mapping $\hat{y}_i^{(t)} = f(N^{(t-\mu)} \cup \ldots \cup N^{(t)})$ such that $\hat{y}_i^{(t)}$ accurately predicts $y_i^{(t)}$. More specifically, as is often the case with classification problems, we want to minimize the loss function given by the mean cross-entropy loss:

$$\mathcal{L} = \frac{-1}{t_f - t_{in}} \sum_{t=t_{in}}^{t_f} \left(y_i^{(t)} \log \hat{y}_i^{(t)} + (1 - y_i^{(t)}) \log(1 - \hat{y}_i^{(t)}) \right)$$

Here, we choose to use $\mu = 4$, so we aim to classify the price movement of the AAPL on day t, given by $p_i^{(t)}$, using news information from days $[t-4, t]$, i.e., articles $\{N^{(t-4)}, N^{(t-3)}, N^{(t-2)}, N^{(t-1)}, N^{(t)}\}$. Notice that we are including information from day t, so we are not *predicting* the price movement but rather identifying a relationship between the stock price movement and the information contained in the news taglines from day t and the previous 4 days.

To convert the news article taglines in our dataset into meaningful quantitative data that we can pass into our model, we have generated word embeddings for them. You will learn more about word embeddings later in the class but essentially, we associate a vector with every word in our dataset such that the vector captures information about how the word relates to other words in the dataset. For the assignment, we have generated a key vector K for each word, which captures contextual information about the word within our dataset.

Here is an outline of how you will build your model:

- Our input dataset consists of rows of articles, where each article $n_i^{(t)}$ has an associated date t, stock ticker, and key vector $K_i^{(t)} \in \mathbb{R}^{64}$. First, filter out the dataset so that we are only dealing with articles for the ticker AAPL.
- Let $N^{(t)}$ represent the set of all articles for the AAPL stock on day t. Then, on each day t, our dataset has $|N^{(t)}|$ key vectors $\in \mathbb{R}^{64}$ (which can be concatenated together to yield a vector $\in \mathbb{R}^{64|N^{(t)}|}$). Since we want to be able to pass in our entire input as a tensor, it would help if the vector associated with each day had a fixed length. We can achieve this by letting $\kappa = \max_t(|N^{(t)}|)$ and zero-padding appropriately so that for each day, concatenating the key vectors for the articles on that day and zero-padding gives us a vector $\in \mathbb{R}^{64\kappa}$.

- Now, we need to generate sequences of length 5 for our input (key vectors articles from day $t-4$ to day t). Suppose we have articles for m days $t_1, \ldots, t_m$. Then, we have $k = m-4$ sequences of length 5: $t_1, \ldots, t_5$, $t_2, \ldots, t_6$, …, $t_{m-4}, \ldots t_m$. So our input is a tensor of shape $(k, 5, 64\kappa)$ – it has k sequences of length 5, where each element of a sequence is a vector $\in \mathbb{R}^{64\kappa}$.

- Now with our input ready, we also need to generate binary output labels since we are performing binary classification for the price movement. Recall that for day t, we define the label $y^{(t)} = \begin{cases} 1 & \log(p^{(t)}) \geq \log(p^{(t-1)}) \\ 0 & \log(p^{(t)}) < \log(p^{(t-1)}) \end{cases}$. Note that due to the nature of the dataset, we do not always have a perfectly contiguous sequence of days. In this case, we use the next best approximation (i.e. $t-2$, if it is available, in place of $t-1$).

- With the input ready, the first layer of our neural network computes *score* values for each article. The *score* for an article i on day t is defined as $score_i^{(t)} = K_i^{(t)} \cdot s$, where s is the stock embedding weight that we are trying to learn. So the output of the first layer has shape $(k, 5, \kappa)$.

- Then, we convert the *score* values into probabilistic weights $\alpha_i^{(t)}$ representing the importance of article i on day t relative to the other articles from day t using a softmax activation. That is, $\alpha_i^{(t)} = \dfrac{\exp(score_i^{(t)})}{\sum_{j\in[\kappa], j\neq i} exp(score_j^{(t)})}$.

- Finally, we pass these sequences of α values into a Bi-GRU with r recurrent units and obtain a single output vector $\hat{h}^{(t)} \in \mathbb{R}^r$ for each sequence. That is, for each sample $[\alpha_1^{(t-4)}, \ldots, \alpha_\kappa^{(t-4)}], \ldots, [\alpha_1^{(t)}, \ldots, \alpha_\kappa^{(t)}]$ of shape $(1, 5, \kappa)$, we obtain a single output $\hat{h}^{(t)}$ (this is a many to one RNN).

- Since we want to get a single binary output 0 or 1 for $\hat{y}^{(t)}$ for each sequence, we apply a final single-neuron layer with a sigmoid activation to transform $\hat{h}^{(t)}$ to $\hat{y}^{(t)}$

References

[1] Afshine Amidi and Shervine Amidi. Recurrent neural networks cheatsheet, 2019.

[2] Xin Du and Kumiko Tanaka-Ishii. Stock embeddings acquired from news articles and price history, and an application to portfolio optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3353–3363, Online, July 2020. Association for Computational Linguistics.

[3] Simeon Kostadinov. Understanding gru networks, Nov 2019.

[4] SuperDataScience. Recurrent neural networks (rnn) - the vanishing gradient problem, Aug 2018.