# 40.319: Statistical and Machine Learning

Max Lee  Choo Yong Sheng

## Recurrent Neural Networks (RNN)

# Table of Contents

# Table of Contents

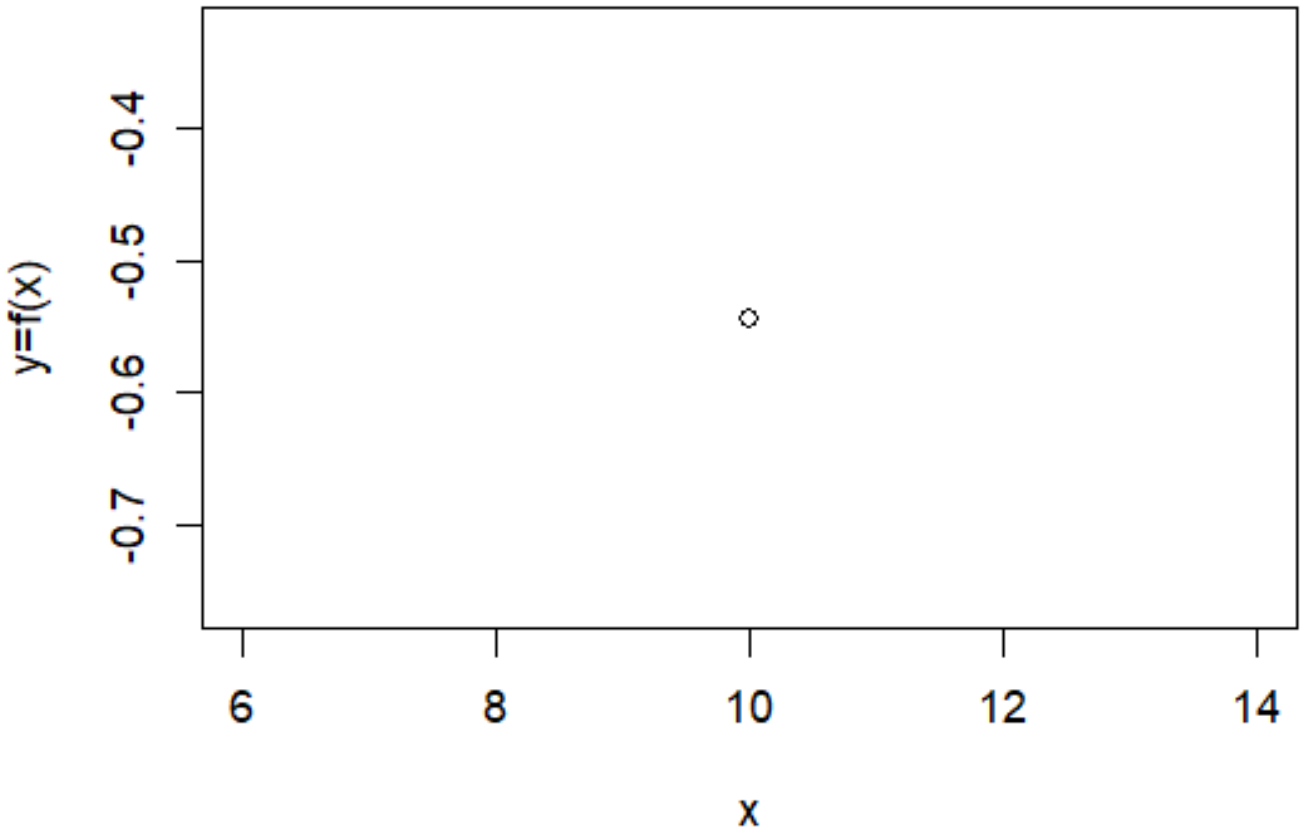
# What is a RNN?

Recurrent Neural Networks (RNN) is a type of artificial neural network which uses **sequential data** or **time series data**.

They are distinguished by their "memory" as they take information from prior inputs to influence the current input and output. While traditional deep neural networks assume that inputs and outputs are independent of each other, the output of recurrent neural networks depend on the prior elements within the sequence.

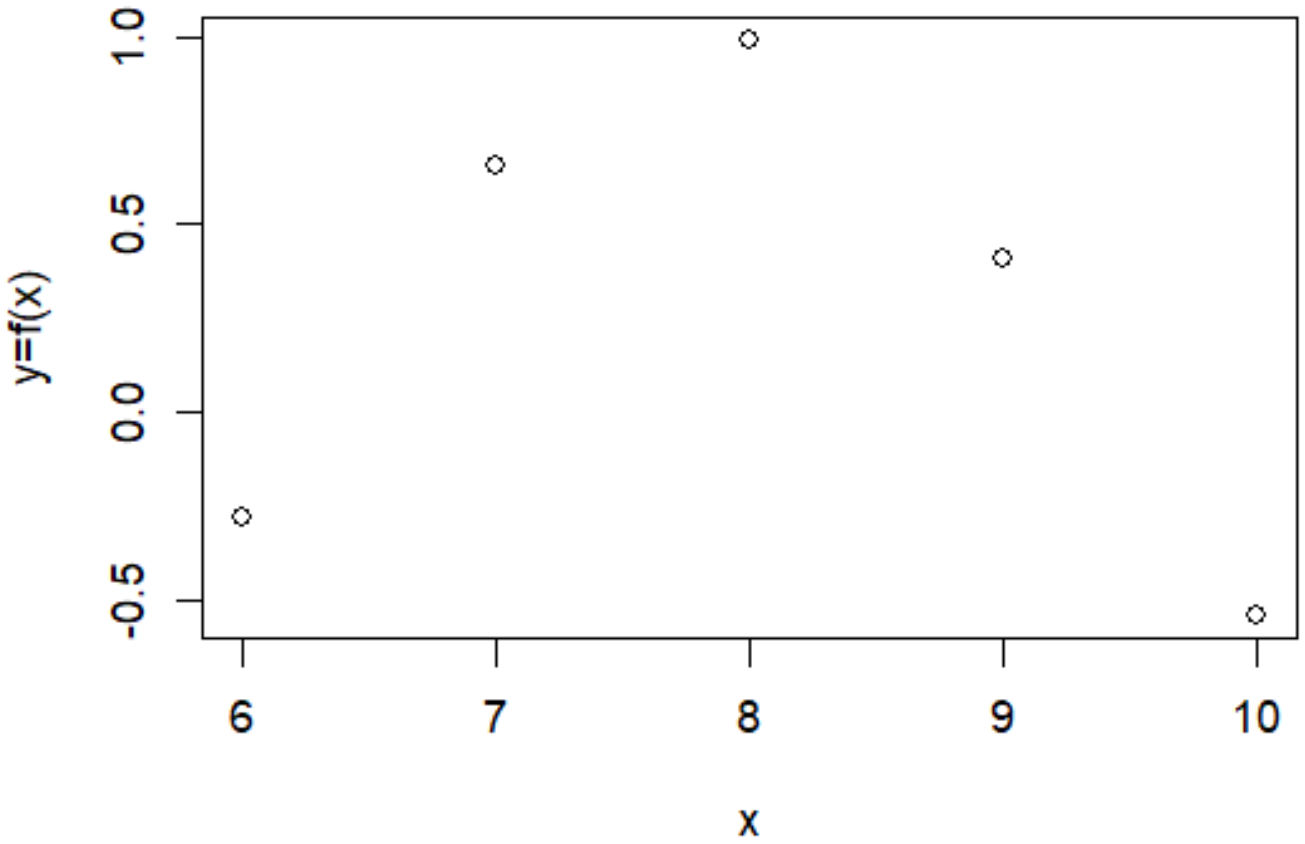
# Why are we teaching RNN?

Let us have experiment with a function, $y = f(x)$, where $f(x)$ is unknown. Then, let us say we have a point at $x = 10, y = -0.544$, where do you think $y$ will be, if $x = 11$?



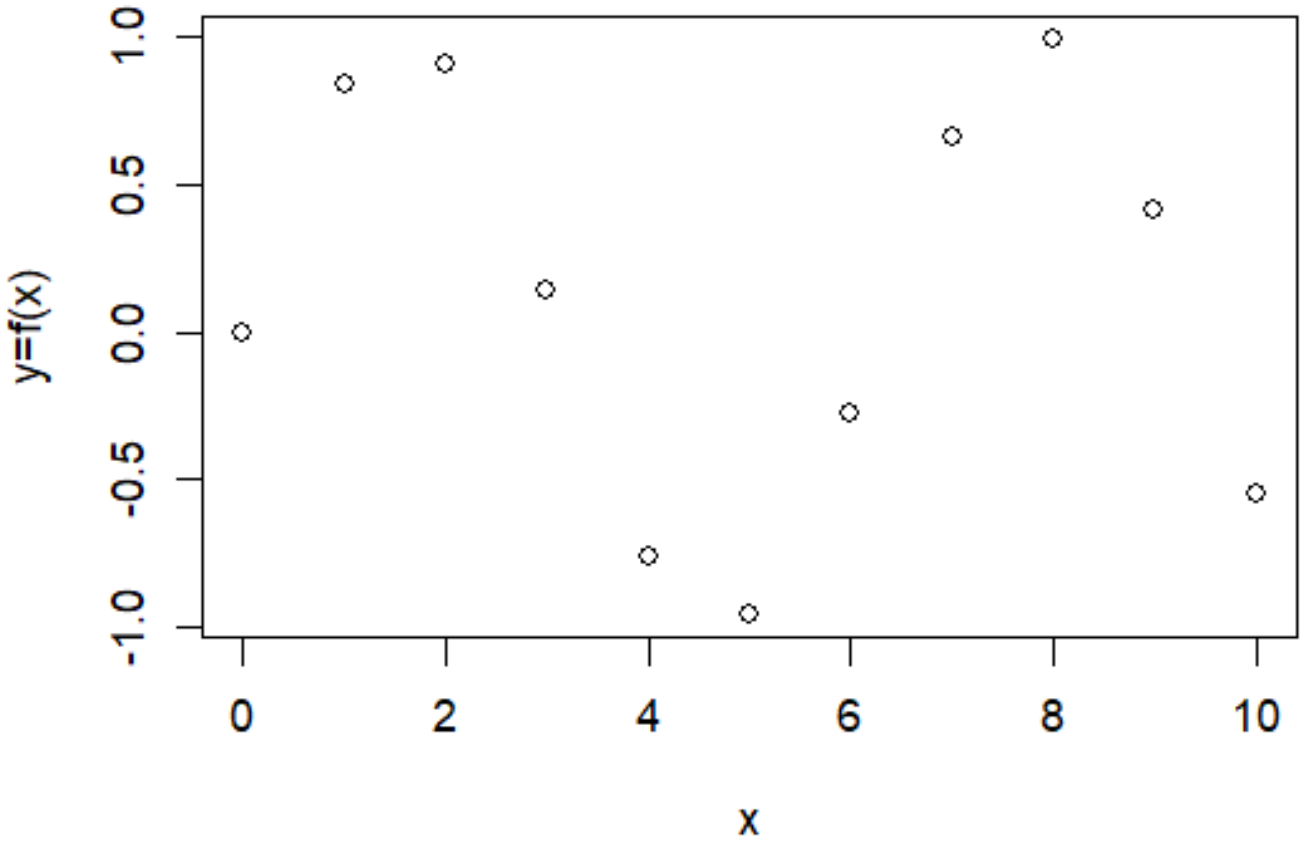
# Why are we teaching RNN?

What about now? We have the points for $x = 6, 7, 8, 9$ as well, with $y$ values being $-0.279, 0.657, 0.989, 0.412$.



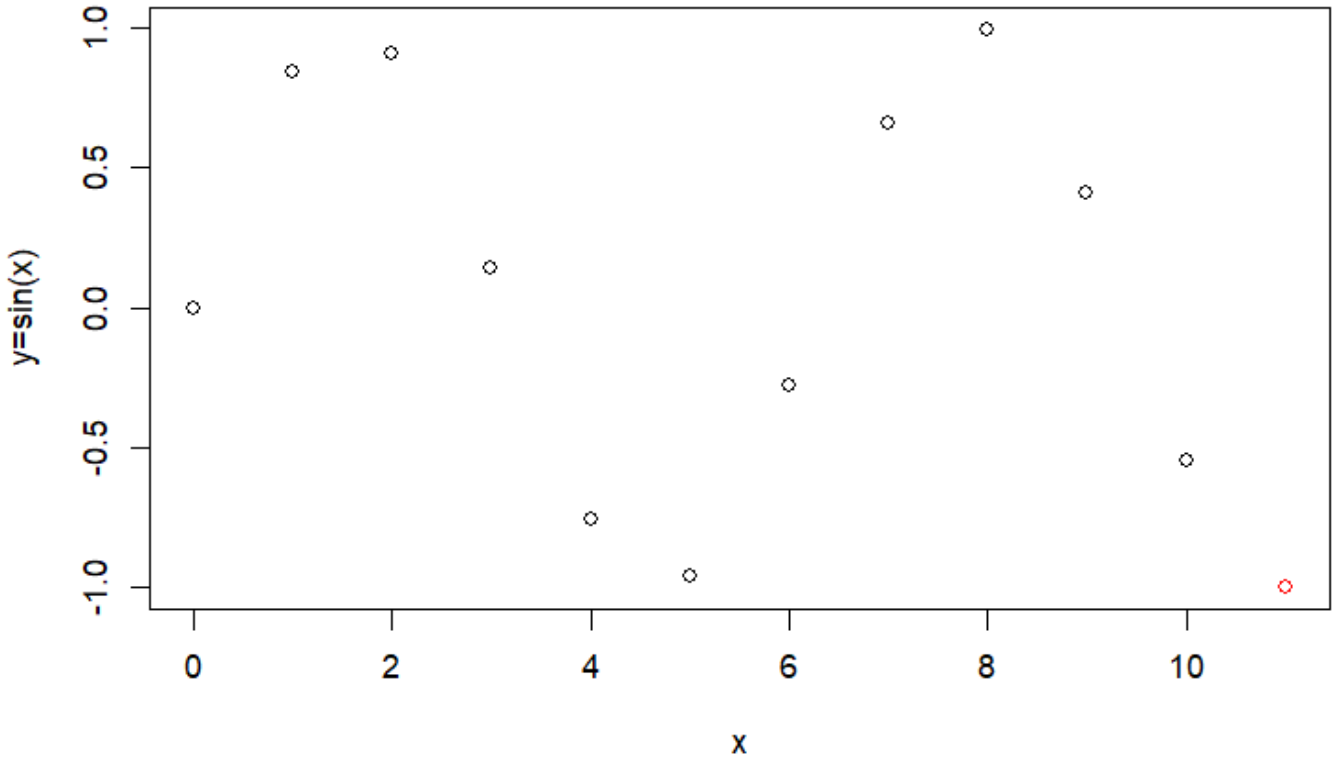
# Why are we teaching RNN?

What about now? We have the points for $x = 0, 1, 2, 3, 4, 5$ as well, with $y$ values being $0, 0.841, 0.909, 0.141, -0.757, -0.959$.



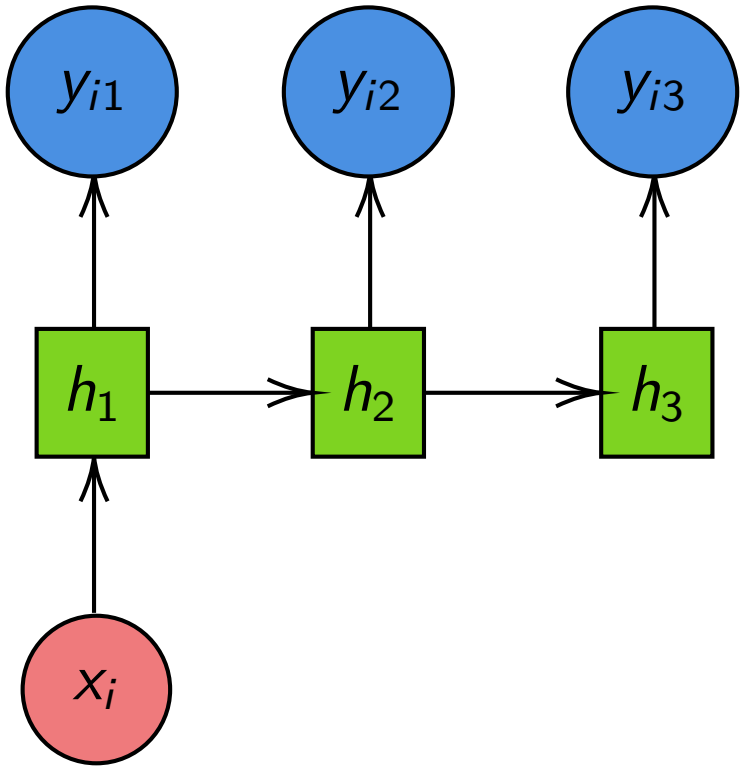
# Why are we teaching RNN?

As more and more points were revealed, a pattern emerged. Based on this pattern, we might guess that when $x = 11$, $y$ lies below $-0.5$, perhaps close to $-1$. Indeed, $y$ is actually $-1$ when $x = 11$. This is in fact, a sine wave.



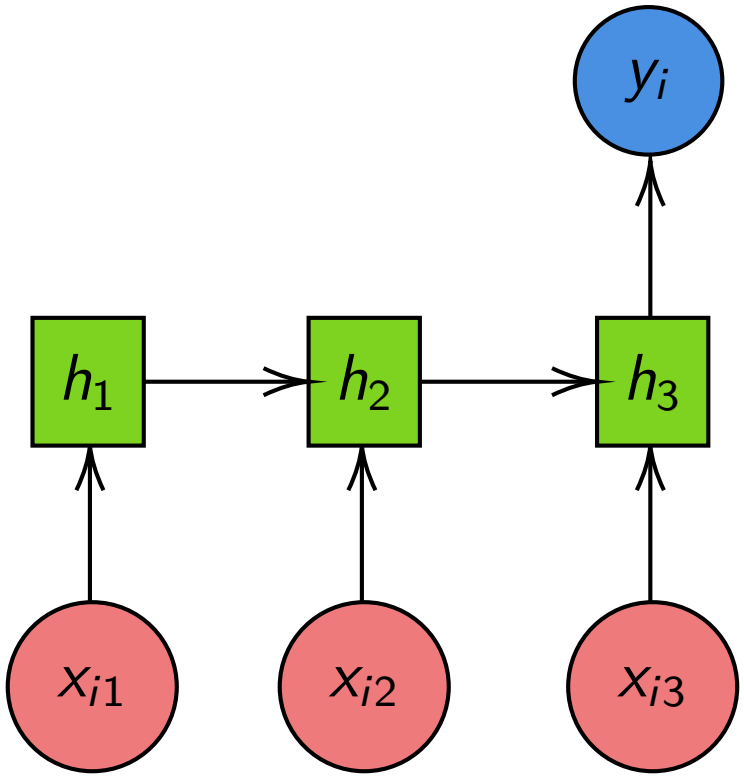
# Types of RNNs

RNNs have a few different types. First, one-to-many. A single input is passed in, and the RNN spits out a sequence as its output. One good example is Image Captioning. An image is passed in, and processed using passed into some CNN layers, before be passed into the RNN as a vector.



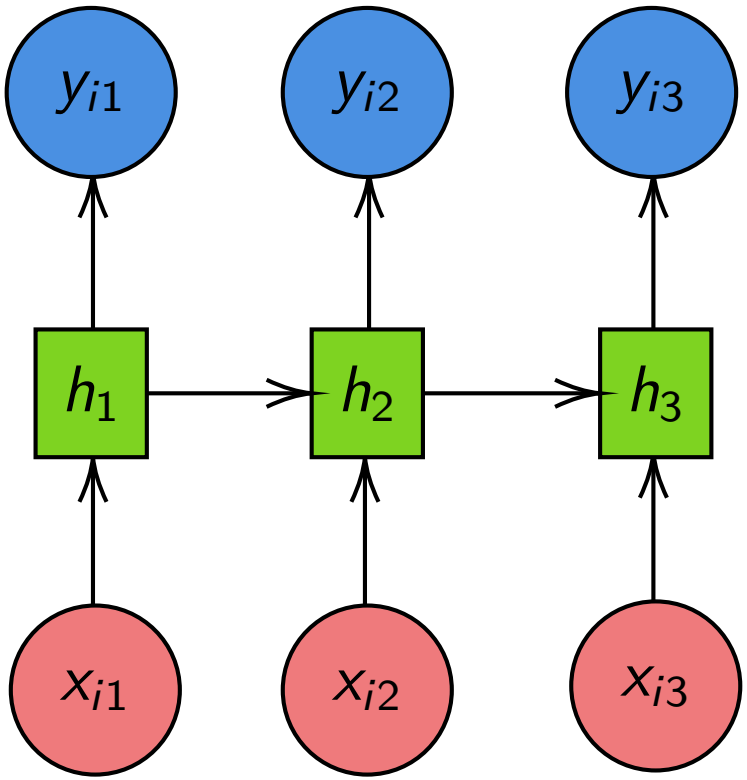
# Types of RNNs

Then, we also have many-to-one RNNs, where a sequence is passed in and a single output is obtained. One good example is Sentiment Analysis, where a sentence is passed in and the RNN determines whether it is a positive or negative sentiment.



# Types of RNNs

Finally, we have many-to-many RNNs, where a sequence is passed in and a different sequence is passed as output. One good example is Video Classification, which is simlar to Image Captioning, except it takes in a sequence of images (or frames) as input, and produces a sentence describing the video.



# Applications of RNN

Below is a list of more applications of RNNs:

- Prediction Problems
- Language Modelling and Generating Text
- Language Translation
- Speech Recognition

Today we will be looking at **Language Translation** in detail.

# Why Language Translation?

The ability to communicate with one another is a fundamental part of being human. There are nearly **7,000** different languages worldwide.

As our world becomes increasingly connected, language translation provides a **critical cultural and economic bridge** between people from different countries and ethnic groups.

# Importance of Language Translation


Correct Translation : Beware, this plant's prickles are very sharp.

# How has RNN impacted the field of Language Translation?

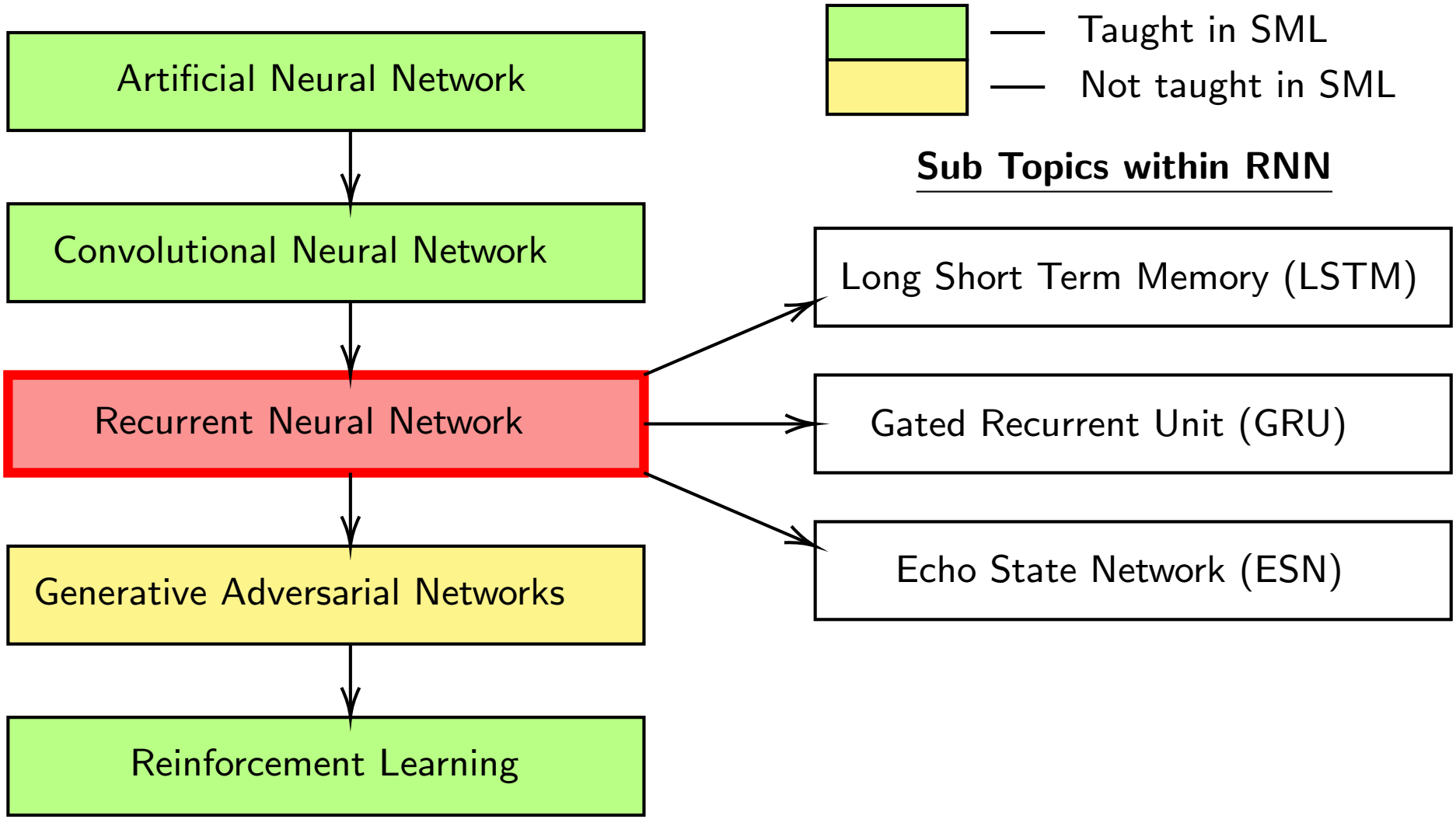
According to Google, switching to deep learning produced a **60% increase in translation accuracy** compared to the phrase-based approach used previously.

Today, translation applications from Google and Microsoft can translate over 100 different languages and are approaching human-level accuracy for many of them.

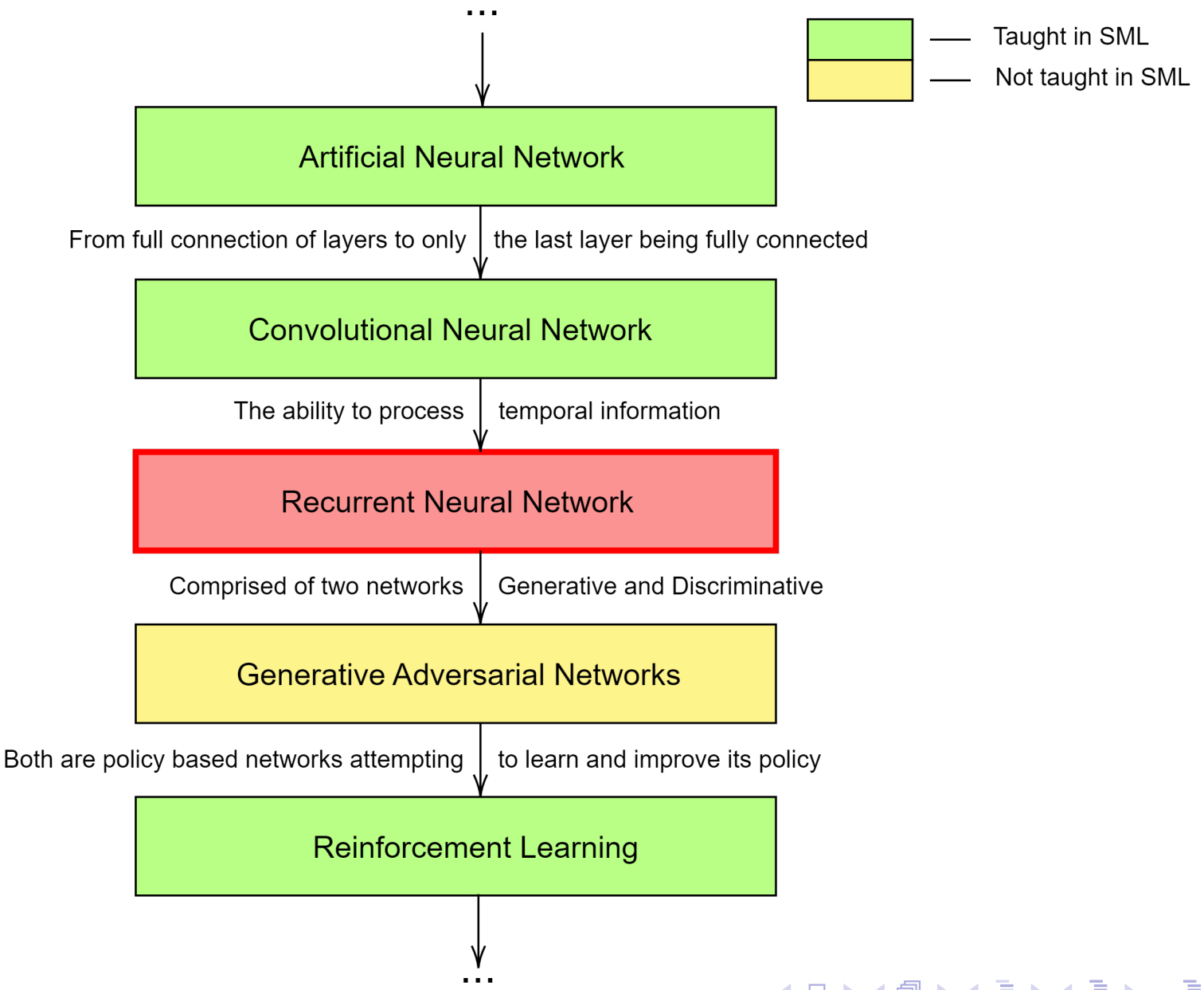
# Table of Contents

# Table of Contents

# Sequence of Topics for Deep Learning



# Difference between Topics



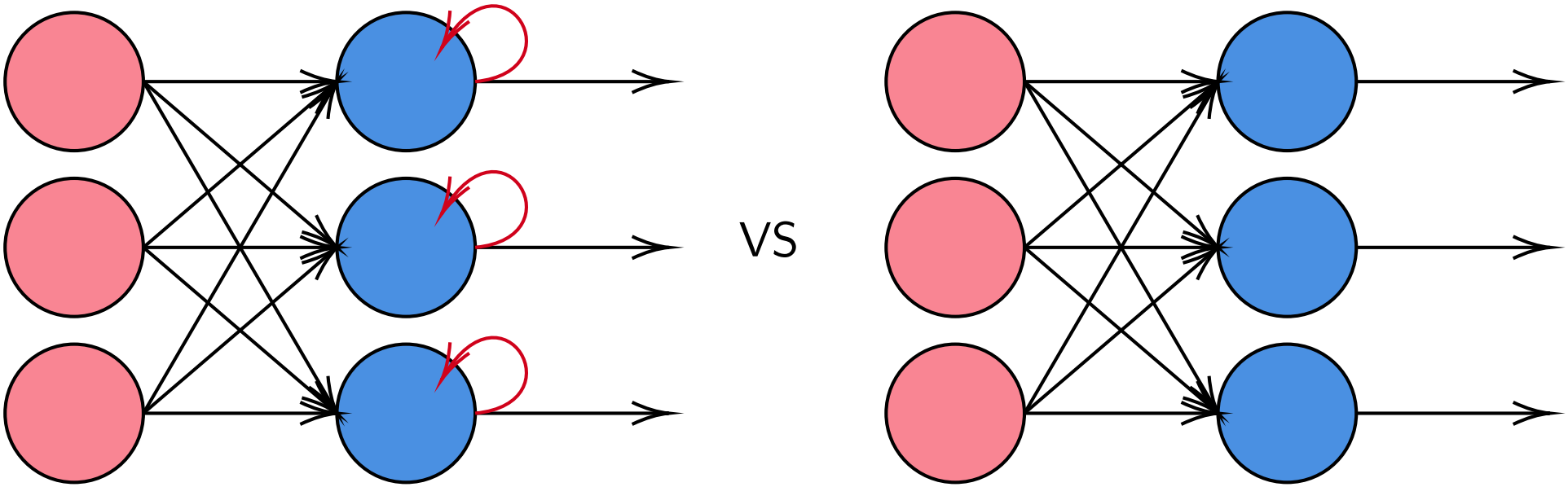
# Table of Contents

# Why do we use RNN instead of ANN?

Imagine you have a normal feed-forward neural network and give it the word “neuron” as an input and it processes the word character by character. By the time it reaches the character “r”, it has already forgotten about “n”, “e” and “u,” which makes it almost impossible for this type of neural network to predict which character would come next.

A recurrent neural network, however, is able to remember those characters because of its internal memory. It produces output, copies that output and loops it back into the network.

# Network Difference Between RNN and ANN




RNN Architecture

ANN Architecture

As seen from the red arrows, RNN considers the current input and also what it has learned from the inputs it received previously.

# Table of Contents

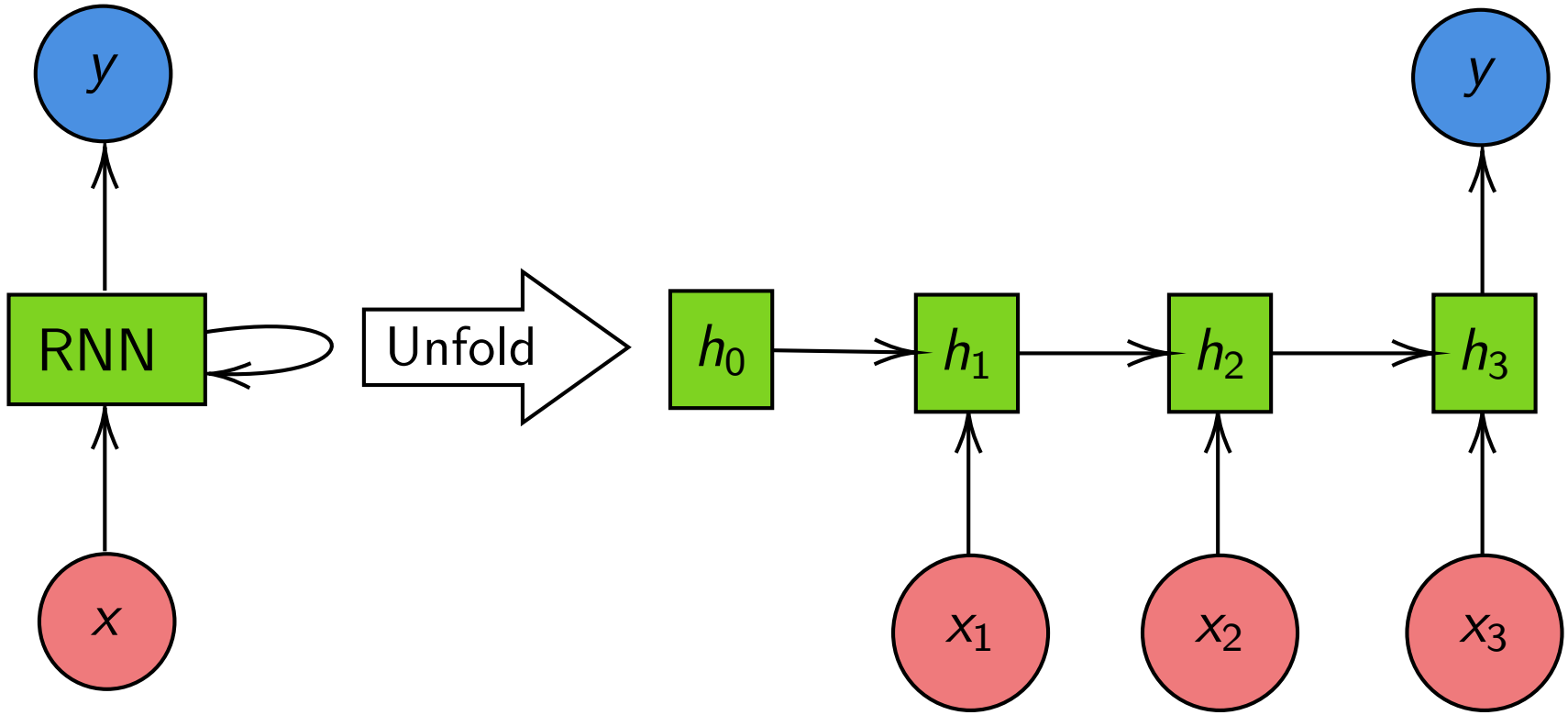
# Table of Contents

# Theory

- We will first run through the mathematics behind an extremely simple RNN.
- This many-to-one RNN will take in a fixed-length input and produce a single output.
- Input matrix $X$ will be 3x3, where each row will denote an entry of three sequential data points.
- Output matrix $Y$ will be a 3x1 column vector, where each row will denote the ground truth value that follows each row from $X$.
- After which, we will go through the code for writing an RNN in R for a similar problem.

# RNN Architecture

Each input, $x$, in the RNN architecture will make use of the previous hidden state, $h$, to compute the next hidden state. For our many-to-one case, a single value will be the output for each forward pass, serving as the prediction for the given input.



# Variables and Hyperparameters

Now is a good time to introduce some variables and hyperparameters essential to an RNN.

- Learning rate, $\alpha$. Similar to traditional neural networks, RNNs use backpropagation to converge into suitable weights, and the rate of convergence may be controlled by the learning rate.
- Number of epochs. How many times we plan to work through the training data set.
- Sequence length, the length of each input, $x$. For our case, this will be 3.
- Hidden dimension, which defines the shape of our hidden layer and output layer weight matrices. For simplicity, we will set this to 3.
- Output dimension, which, for our case, refers to the $y$ output we plan to predict. For our case, this will be 1.

# Weight Matrices

In our RNN, there are three weight matrices to take note of.

- $U$, with dimensions 3x3, derived from the hidden dimension x the sequence length.
- $W$, also with dimensions 3x3, but derived from the hidden dimension x the hidden dimension.
- $V$, with dimensions 1x3, derived from the output dimension x the hidden dimension.

# Activation Function

Each element in the input will be weighted against the weight matrix $U$, and added to the previous hidden state weighted against the weight matrix $W$. The result will be passed into an activation function, and that would be the resultant vector for the hidden layer.

The activation functions of choice are similar to that of traditional neural networks. For our case, we will stick with the sigmoid function, and denote it as $\phi$.

## Sigmoid function

The sigmoid function is given as:

$$\phi(x) = \frac{1}{1 + exp(-x)}$$

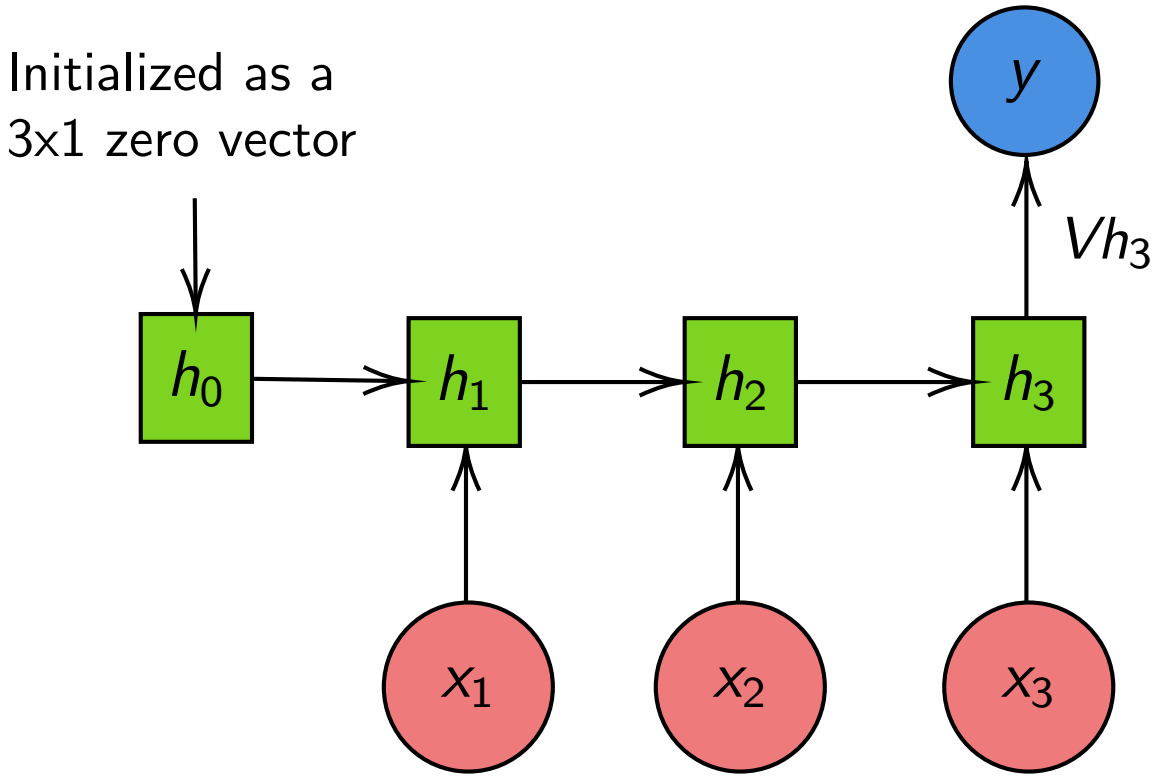
# Additional Hyperparameters

There are also some hyperparameters to take note of, though we shall explain them in more detail later on.

- Backpropagate through time truncate, set to 5.
- Minimum and maximum clip values, set to -10 and 10 respectively.

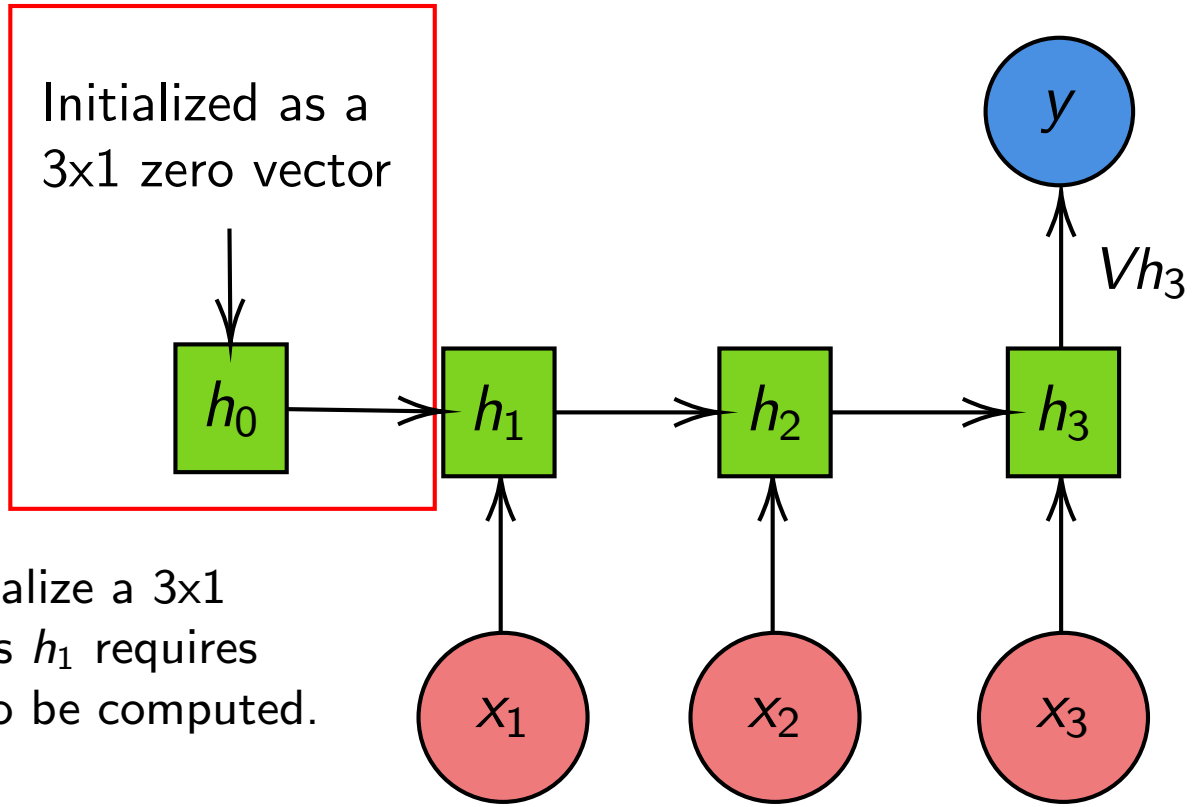
# Forward Pass for Predictions

Now that we have defined the necessary variables and hyperparameters, let us look at how a forward pass occurs.




$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Forward Pass for Predictions

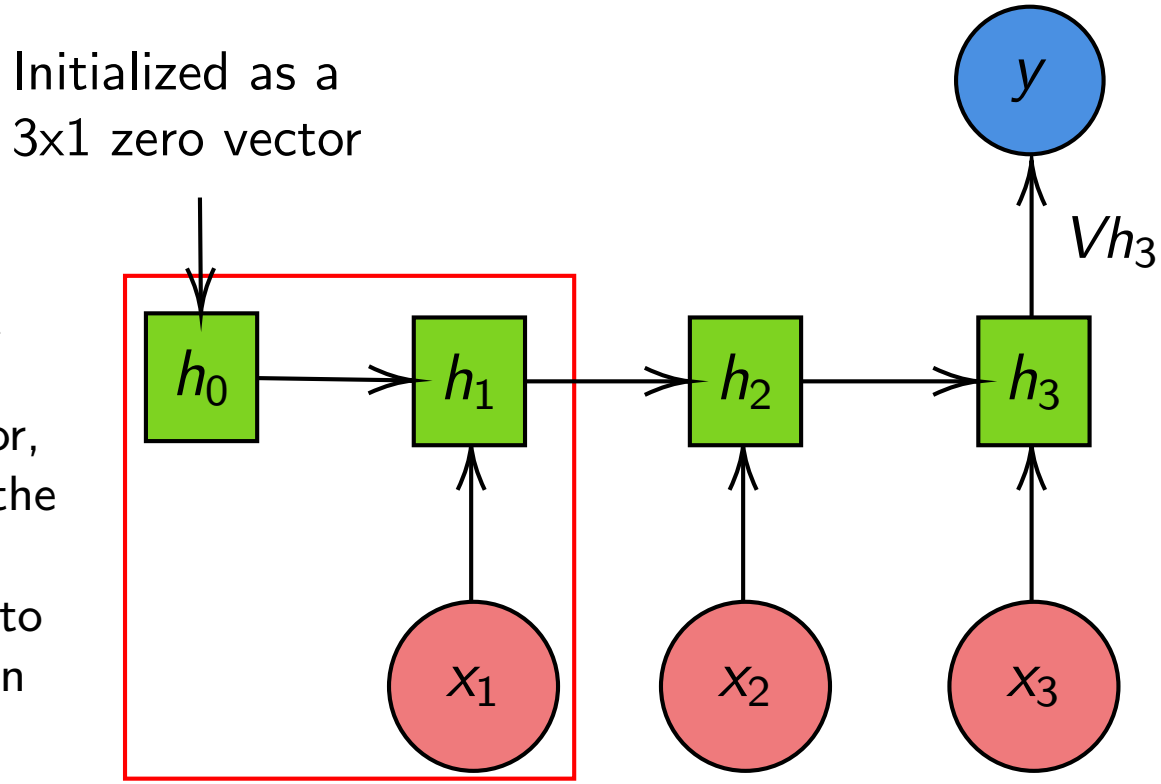



First, we initialize a 3x1 zero vector as $h_1$ requires a vector $h_0$ to be computed.

$$h_t = \phi(Wh_{t-1} + Ux_t)$$

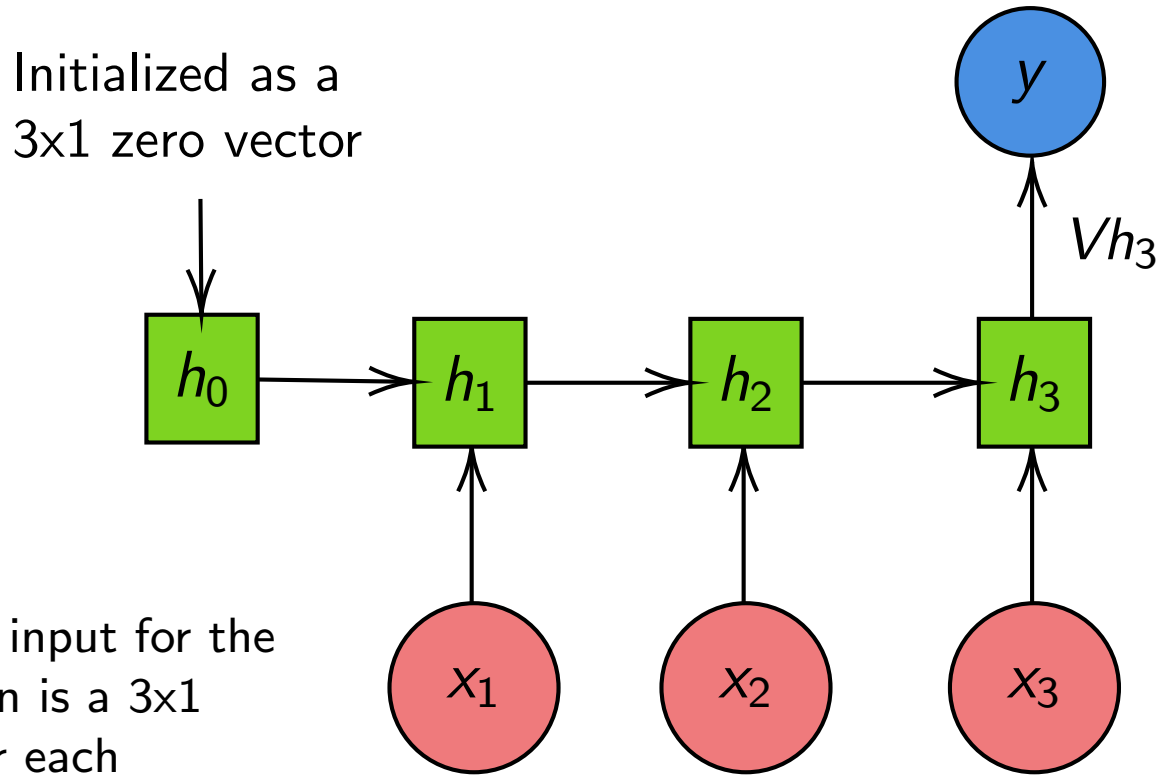
# Forward Pass for Predictions

Next, we pass in the input $x_1$ for the computation of $h_1$. $x_1$ is passed in as a 3x1 vector, with the first element being the value of $x_1$, and the other elements being zero. This is to facilitate matrix multiplication with the weight matrix $U$.




$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Forward Pass for Predictions



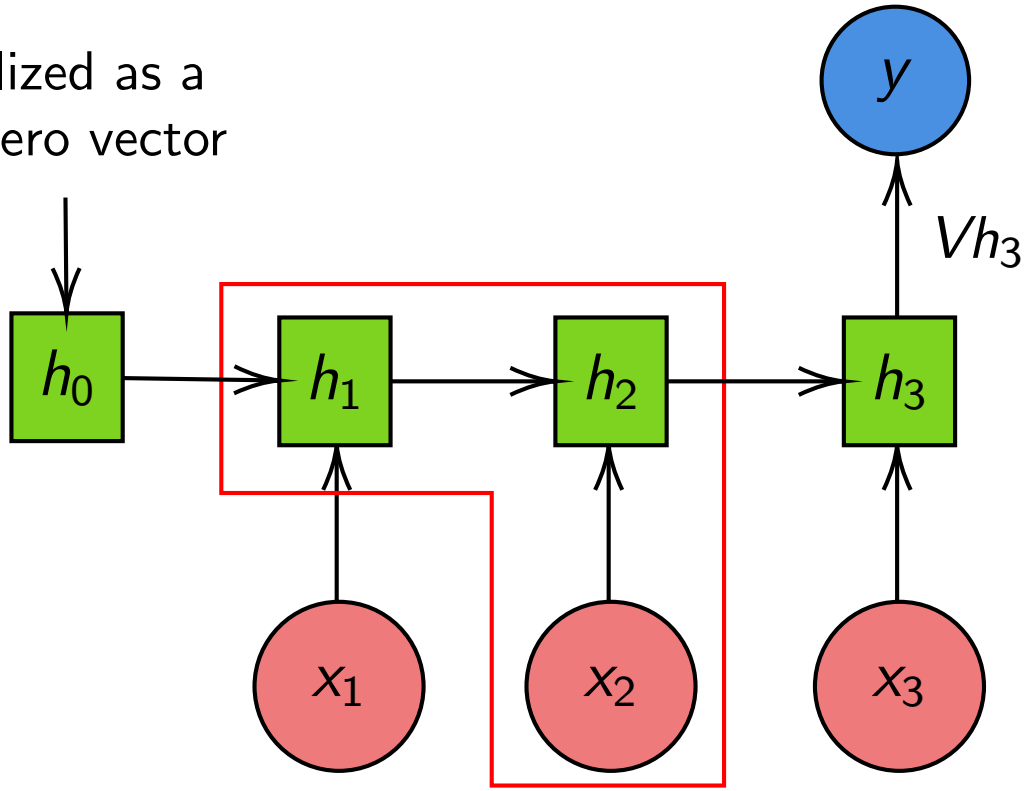

We note that the input for the activation function is a 3x1 vector. Hence, for each computation of $h_t$, the result is a 3x1 vector, with each element being the output of the activation function on that element.

$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Forward Pass for Predictions

Initialized as a
3x1 zero vector

We then repeat similar computation for $h_2$.

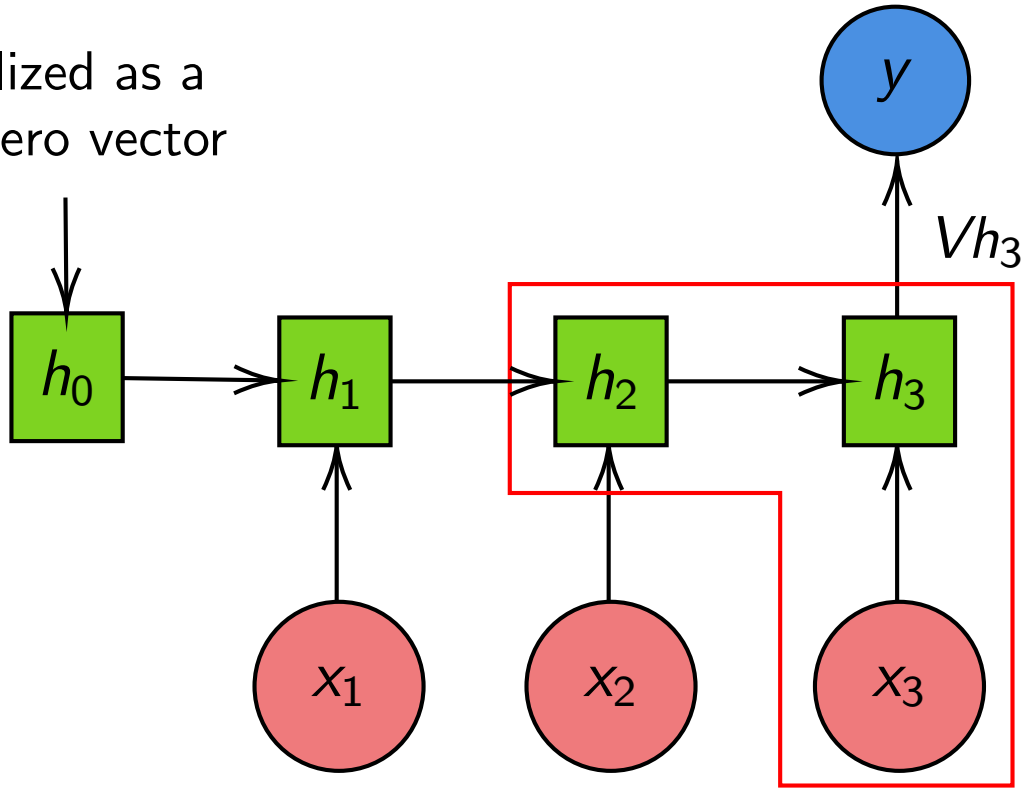



$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Forward Pass for Predictions

Initialized as a
3x1 zero vector

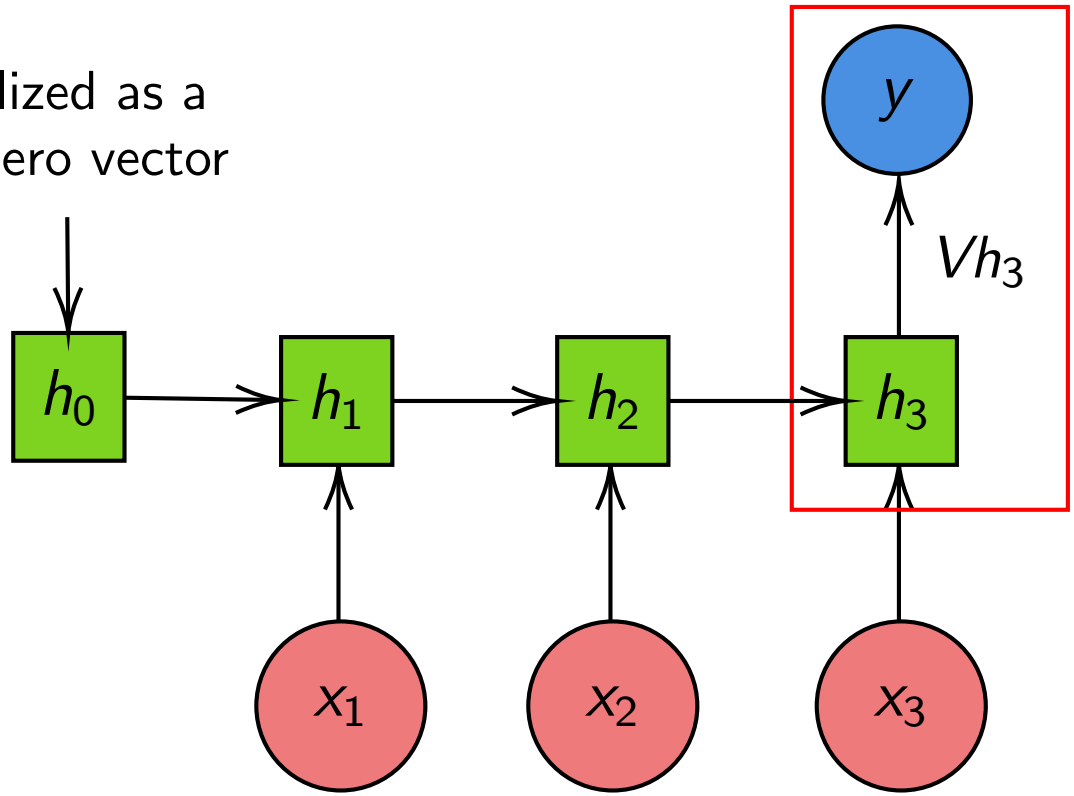
And then for $h_3$.




$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Forward Pass for Predictions

Initialized as a
3x1 zero vector




And we get the prediction for $y_i$, which is a scalar from the multiplication between a 1x3 and a 3x1 vector.

$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Loss Calculation and Second Forward Pass

- After passing through all our input vectors, we will have a prediction for each output. We then calculate the loss for that epoch, and proceed to the second forward pass.

## Loss Calculation

The loss function is given as:

$$L = \sum_{i=1}^{3} \frac{(y_i - V(h_3)_i)^2}{2} \tag{1}$$

This allows for easy computation for the derivative of the loss.

- For this second forward pass, we will also be storing the layers at each timestep, as well as the previous layer at that timestep. Then, the truncated backpropagation through time will follow.

# Loss Minimization

Similar to ANNs, we want to perform backpropagation to minimize the loss function. We will update the weights at each epoch using the following, where $\alpha$ is the learning rate:

$$W = W - \alpha \frac{\partial L}{\partial W}$$
$$U = U - \alpha \frac{\partial L}{\partial U}$$
$$V = V - \alpha \frac{\partial L}{\partial V}$$

Gathering all the equations from the forward pass, we have:

$$h_1 = \phi(Wh_0 + Ux_1) \tag{2}$$
$$h_2 = \phi(Wh_1 + Ux_2) \tag{3}$$
$$h_3 = \phi(Wh_2 + Ux_3) \tag{4}$$
$$y = Vh_3 \tag{5}$$

# Partial Differentials

Let us begin with an easy one, the partial differential for $V$. By the chain rule,

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial V}$$

By carrying out the respective partial on (1), with the summation omitted, and (5),

$$\frac{\partial L}{\partial y} = y - Vh_3$$
$$\frac{\partial y}{\partial V} = h_3$$

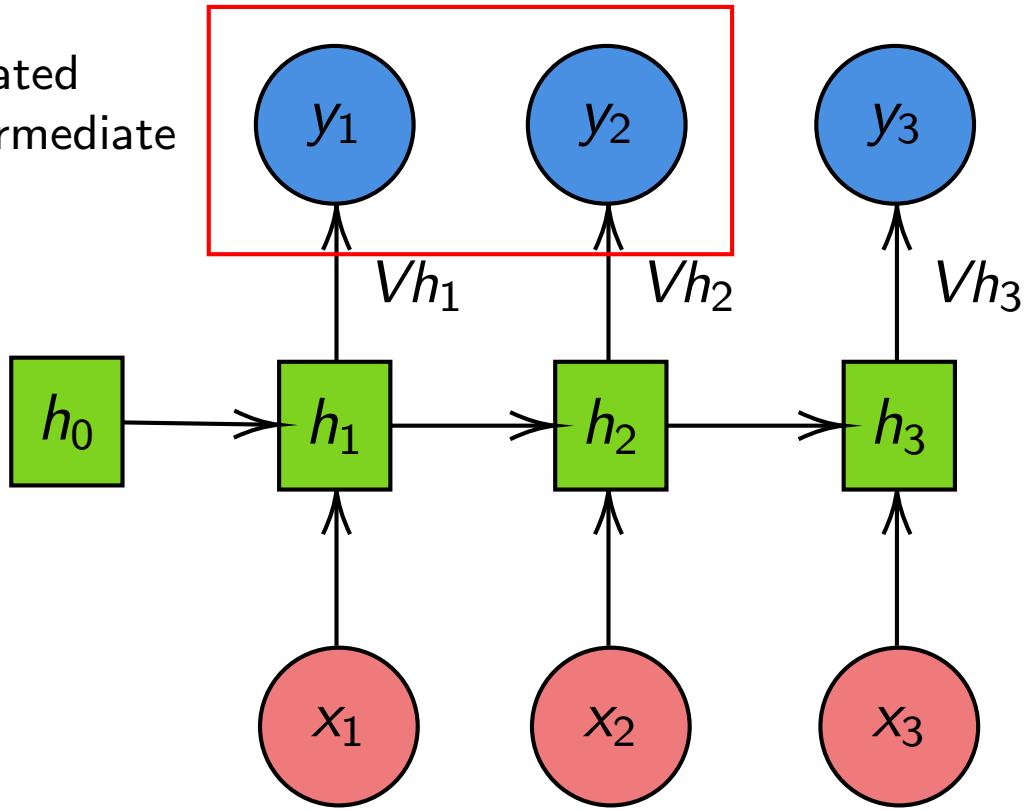
Which leaves us with:

$$\frac{\partial L}{\partial V} = (y - Vh_3)h_3$$

# A Note on Many-to-One Application

It is possible to compute intermediate losses for a many-to-one application to further train the weights for $V$, though we will not go into detail on the specifics here.

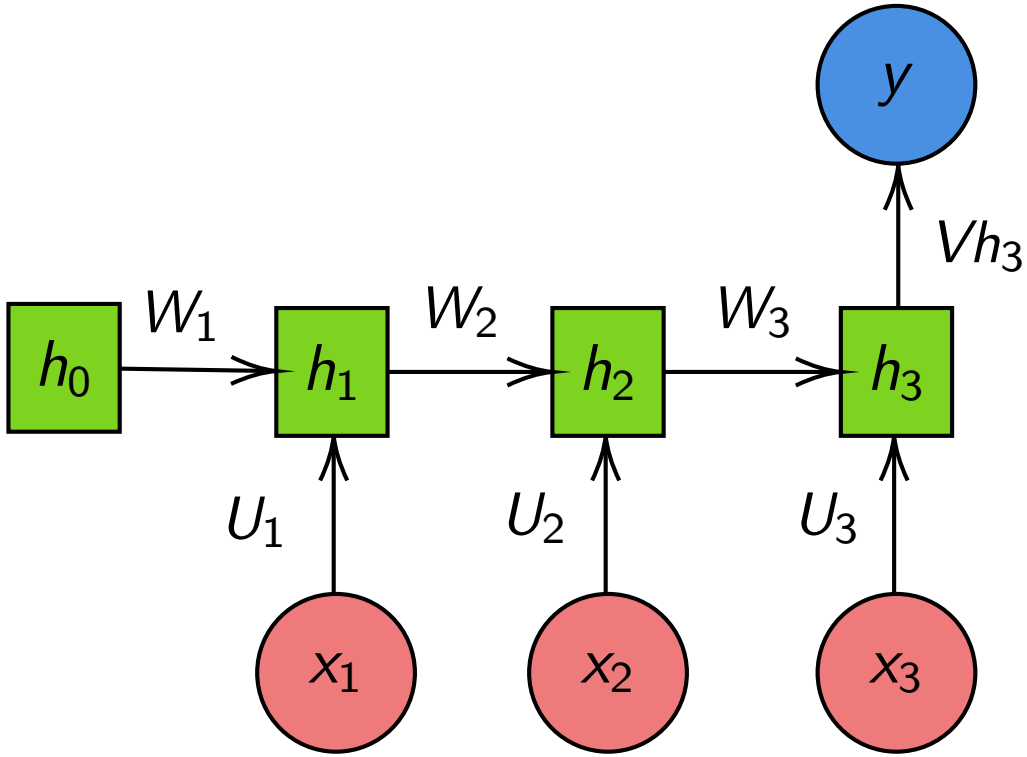
Can be incorporated to calculate intermediate loss.




$$h_t = \phi(Wh_{t-1} + Ux_t)$$

# Notation Update

The computation for $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial U}$ is quite similar to each other. First, let us update the RNN architecture graph for more clarity. We introduce $W_1, W_2, W_3, U_1, U_2, U_3$, where $W = W_1 = W_2 = W_3, U = U_1 = U_2 = U_3$.



# Partial Differentials

Then, the partial differentials are given by

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial W_1} + \frac{\partial L}{\partial W_2} + \frac{\partial L}{\partial W_3}$$
$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial U_1} + \frac{\partial L}{\partial U_2} + \frac{\partial L}{\partial U_3}$$

Then, by the chain rule,

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W_2} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial W_3}$$
$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial U_1} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial U_2} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial U_3}$$

# Truncated Backpropagation Through Time

- Clearly, the amount of computation increases very quickly as we work with longer sequences. When we compute the partial derivatives, the layers nearing the end of the output are naturally given more weight to the output.
  - An intuitive way to think about this is to ask: how much bearing would the value of a certain stock price have from five years ago compared to one day ago to predict its value today?
- As such, we introduce the idea of truncated backpropagation through time, where we only look back a certain number of states to calculate the partial derivatives.
- We will not see this in action yet, as we currently only have three timesteps, but we will later on when we look at the sine wave example.

# Gradient Clipping

A very well-known problem with backpropagation methods is the exploding and vanishing gradient problems.

- Vanishing gradient refers to how the calculated gradients are so low that it has no effect on the weights, and no convergence occurs.
- Exploding gradient refers to how calculated gradients are so large that they cause divergence.

There are techniques to combat exploding and vanishing gradients which are built into the many different deep learning frameworks. For simplicity, we will only combat exploding gradients using gradient clipping to impose a minimum and maximum for our gradients.

# Rinse and Repeat Until Convergence

We will then loop through the process for our predefined number of epochs, and hopefully by the end of the process, we would have a good set of weights to make the required predictions.

# Table of Contents

# Sine Wave Application

We will now proceed to look at an RNN to solve the relatively simple problem of predicting the next value of the sine wave created from scratch on R.

# Libraries Used

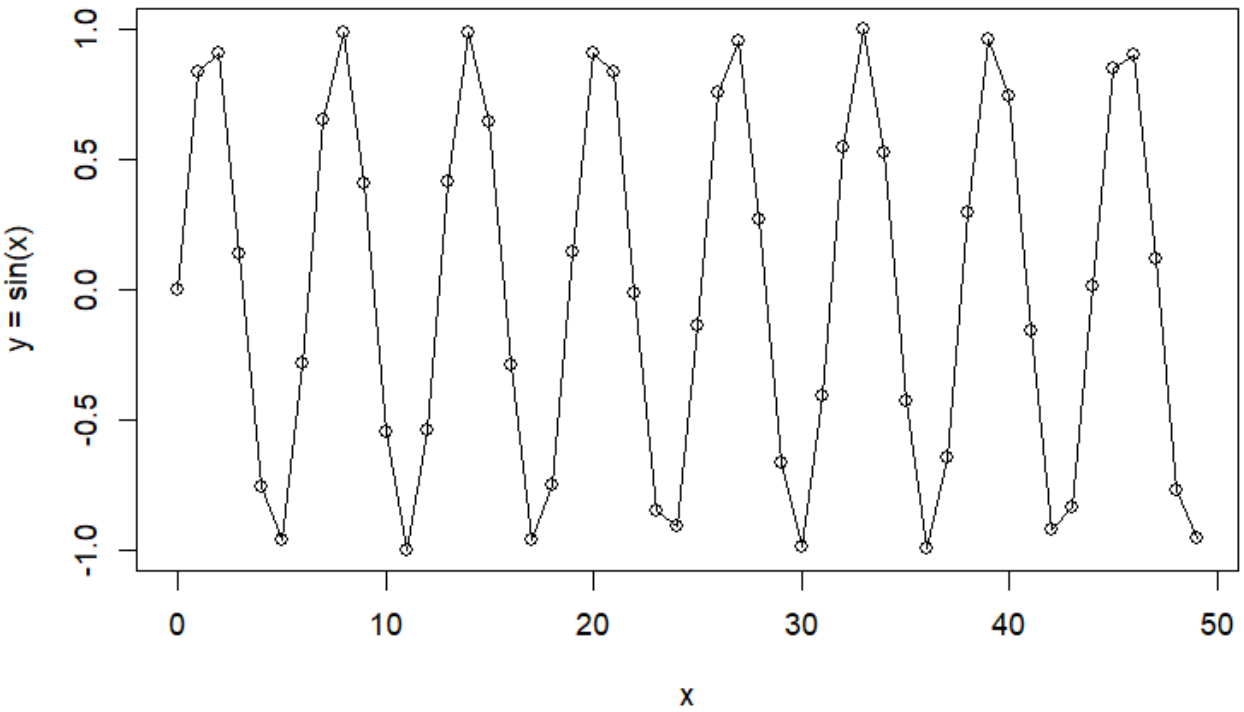
Since we are building the RNN from scratch, we will only be using some quality-of-life libraries, ie. dplyr and tidyverse.

```
library(dplyr)
library(tidyverse)
```

# The Sine Wave

We first generate the $\sin(x)$ for $x \in \mathbb{Z} : x \in [0, 199]$, and show what it looks like up to $x = 49$:

```
sine <- as.matrix(sin(0:199))
plot(0:49, sine[1:50],xlab="x",ylab="y = sin(x)")
lines(0:49, sine[1:50])
```



# Train/Validation Split

- We set the sequence length to 50.
- This would mean that we have 150 total entries to carry out, from predicting sin(50) to sin(199).
- We then create a train/validation split, where predicting sin(50) to sin(149) constitutes the training set, and predicting sin(150) to sin(199) constitutes the validation set.

```
# Training set
X = NULL
Y = NULL
sequence_length = 50
num_records = length(sine) - sequence_length
for (i in 1:(num_records-50)){
    X = rbind(X, sine[i:(i+sequence_length-1)])
    Y = rbind(Y, sine[i+sequence_length])
}
```

# Train/Validation Split

```
# Validation set
X_val = NULL
Y_val = NULL
for (i in (num_records-49):(num_records)){
    X_val = rbind(X_val, sine[i:(i+sequence_length-1)])
    Y_val = rbind(Y_val, sine[i+sequence_length])
}
```

# Hyperparameters

Setting up the necessary hyperparameters.

```
learning_rate = 0.0001
epochs = 15
hidden_dim = 100
output_dim = 1

bptt_truncate = 5
min_clip_value = -10
max_clip_value = 10
```

# Weights Initialization

To demonstrate convergence, we will load in a pre-initialized set of weights. Usually, we would randomize the initial set of weights. Recall that the dimensions are:

- U: hidden dimension x sequence length
- W: hidden dimension x hidden dimension
- V: output dimension x hidden dimension

```
U <- read.csv("U.csv", header=F) # 100 x 50
W <- read.csv("W.csv", header=F) # 100 x 100
V <- read.csv("V.csv", header=F) # 1 x 100

U <- as.matrix(U)
V <- as.matrix(V)
W <- as.matrix(W)
```

# Sigmoid Function

We also will be needing the sigmoid function

```
sigmoid <- function(x){
    1 / (1 + exp(-x))
}
```

# Training Loss Calculation

We will now proceed with the meat of the process, looping through the number of epochs, we first calculate the loss for the training set.

```
for (epoch in 1:epochs){
  loss = 0
  for (i in 1:dim(Y)[1]){
    x <- X[i,]
    y <- Y[i]
    prev_s <- rep(0, hidden_dim)
    for (t in 1:sequence_length){
      new_input <- rep(0, length(x))
      new_input[t] <- x[t]
      mulu <- U %*% new_input
      mulw <- W %*% prev_s
      add <- mulw + mulu
      s = sigmoid(add)
      mulv <- V %*% s
      prev_s <- s
    }
    loss_per_record <- ((y - mulv)^2)/2
    loss = loss + loss_per_record
  }
  loss = loss/length(y)
```

# Validation Loss Calculation

Then for the validation set, and then printing out the losses.

```
val_loss = 0
for (i in 1:dim(Y_val)[1]){
    x <- X_val[i,]
    y <- Y_val[i]
    prev_s <- rep(0, hidden_dim)
    for (t in 1:sequence_length){
    new_input <- rep(0, length(x))
    new_input[t] <- x[t]
    mulu <- U %*% new_input
    mulw <- W %*% prev_s
    add <- mulw + mulu
    s = sigmoid(add)
    mulv <- V %*% s
    prev_s <- s
}

loss_per_record <- ((y - mulv)^2)/2
    val_loss = val_loss + loss_per_record
}
val_loss <- val_loss/length(y)
    print(paste0("Epoch: ", epoch, ", Loss: ", loss, ", Val Loss: ", val_loss))
```

# Forward pass

Now, we proceed with the forward pass, once for each training observation.

```
for (i in 1:dim(Y)[1]){
    x <- X[i,]
    y <- Y[i]

    layers <- array(rep(NaN, sequence_length * hidden_dim * 2), c(sequence_length, 2, hidden_dim))
    prev_s <- rep(0, hidden_dim)
    dU <- 0 * U
    dV <- 0 * V
    dW <- 0 * W
    dU_t <- 0 * U
    dV_t <- 0 * V
    dW_t <- 0 * W
    dU_i <- 0 * U
    dW_i <- 0 * W
    for (t in 1:sequence_length){
      new_input <- rep(0, length(x))
      new_input[t] <- x[t]
      mulu <- U %*% new_input
      mulw <- W %*% prev_s
      add <- mulw + mulu
      s = sigmoid(add)
      mulv <- V %*% s
      layers[t,1,] <- s
      layers[t,2,] <- prev_s
      prev_s <- s
    }
```

# Truncated Backpropagation Through Time

And now, the backpropagation. Note that intermediate loss is used in this case to adjust the weights for $V$ as well. We also see the truncated backpropagation in action here.

```
dmulv <- mulv - y
for (t in 1:sequence_length){
  dV_t <- dmulv %*% t(layers[t,1,])
  dsv <- t(V) %*% dmulv
  ds = dsv
  dadd = add * (1 - add) * ds
  dmulw = dadd * rep(1, length(mulw))
  dprev_s = t(W) %*% dmulw
  for (i in (t-1):(max(-1, (t-bptt_truncate-1))+1)){
    ds = dsv + dprev_s
    dadd = add * (1 - add) * ds
    dmulw = dadd * rep(1, length(mulw))
    dmulu = dadd * rep(1, length(mulu))
    dW_i = W %*% layers[t,2,]
    dprev_s = t(W) %*% dmulw
    new_input = rep(0, length(x))
    new_input[t] = x[t]
    dU_i = U %*% new_input
    dx = t(U) %*% dmulu
    dU_t = apply(dU_t, 2, function(x)x+dU_i)
    dW_t = apply(dW_t, 2, function(x)x+dW_i)
  }
```

# Gradient Clipping and Weights Update

Finally, we update the weights after performing necessary gradient clips.

```
    dU = dU + dU_t
    dW = dW + dW_t
    dV = dV + dV_t
    if (max(dU) > max_clip_value){
    dU[dU > max_clip_value] = max_clip_value
    }
    if (max(dV) > max_clip_value){
    dV[dV > max_clip_value] = max_clip_value
    }
    if (max(dW) > max_clip_value){
    dW[dW > max_clip_value] = max_clip_value
    }
    if (min(dU) < min_clip_value){
    dU[dU < min_clip_value] = min_clip_value
    }
    if (min(dV) < min_clip_value){
    dV[dV < min_clip_value] = min_clip_value
    }
    if (min(dW) < min_clip_value){
    dW[dW < min_clip_value] = min_clip_value
    }
    }
    U = U - learning_rate * dU
    V = V - learning_rate * dV
    W = W - learning_rate * dW
    }
}
```

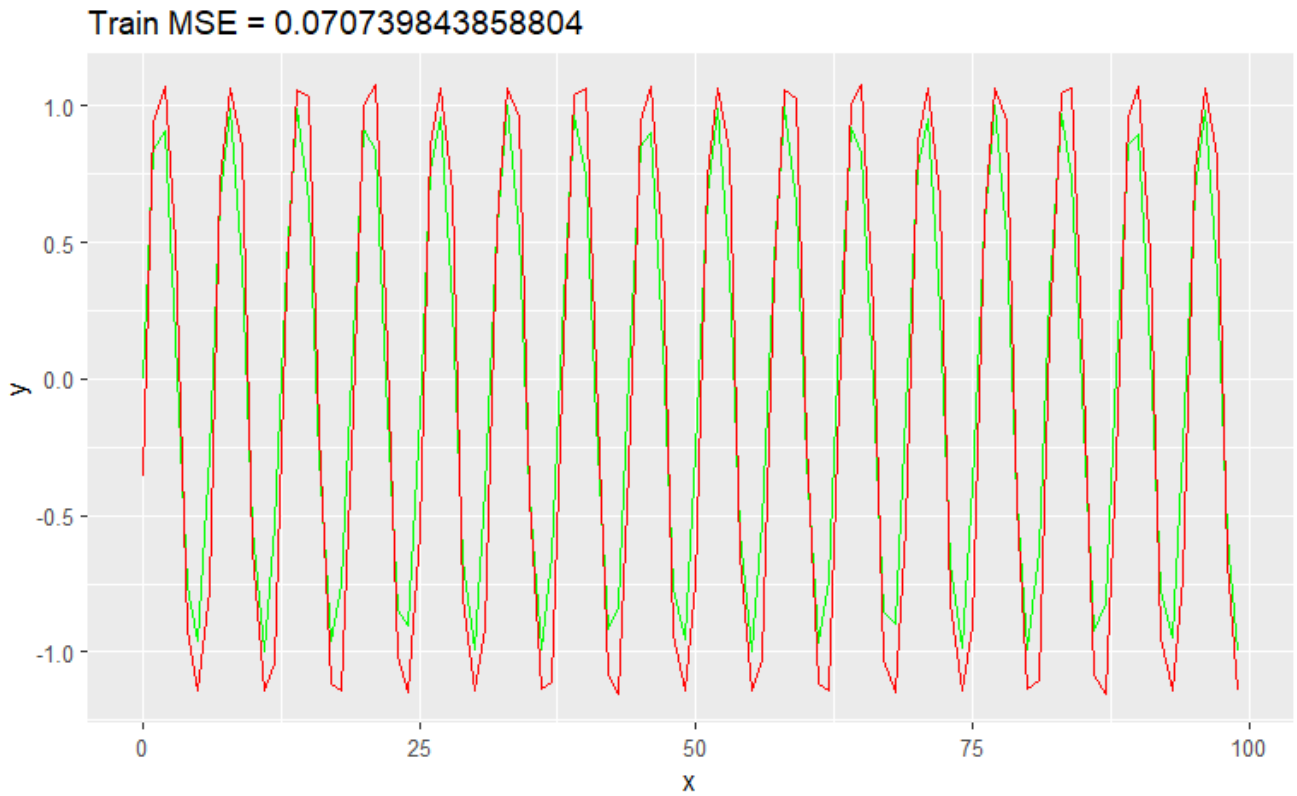
# Training MSE

Let us now see how well our model performs on the training set.

```
preds <- NULL
for (i in 1:dim(Y)[1]){
    x <- X[i,]
    y <- Y[i]
    prev_s <- rep(0, hidden_dim)
    for (t in 1:T){
    mulu <- U %*% x
    mulw <- W %*% prev_s
    add <- mulw + mulu
    s <- sigmoid(add)
    mulv <- V %*% s
    prev_s <- s
    preds <- rbind(preds, mulv)
    }
}

actual_df <- data.frame(x = 0:99, y = sine[1:100])
predicted_df <- data.frame(x = 0:99, y = preds)

train_mse <- sum((actual_df[y] - predicted_df[y])^2)/dim(actual_df[y])[1]
train_mse

ggplot(NULL, aes(x, y)) +
geom_line(data = actual_df, color = "green") +
geom_line(data = predicted_df, color = "red") +
ggtitle(paste0("Train MSE = ", train_mse))
```

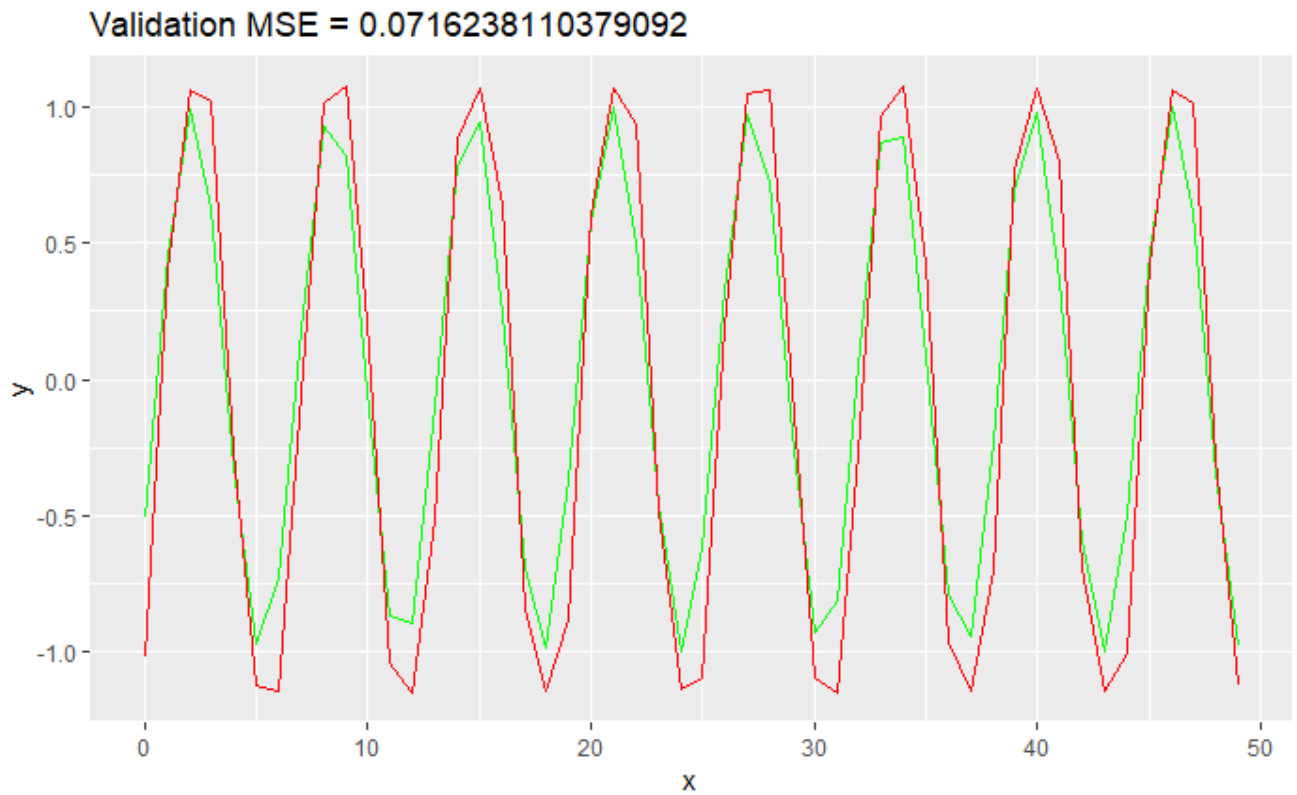
# Training MSE

With green being the actual value and red being the predicted value, we get an MSE of 0.07074.



# Validation MSE

To check for overfitting on the training set, we shall also check the validation MSE, which ends up being 0.07162.



# Table of Contents

# Application

Now that we have learnt the fundamentals of RNN, let us look at one of its possible application : **Language Translation**

**Goal:** To train an RNN such that it is able to take in English Text as an input and returning a French translated text as an output.

# Approach

Before we start looking at the code required for this application, let us take a look at how we are going to approach this problem:

1. **Preprocessing:** Load and Examine data, Tokenization, Padding
2. **Modeling:** Build, Train, and Test the model
3. **Prediction:** Generate specific translations of English to French, and compare the output translations to the ground truth translations

# Table of Contents

# Packages Used

Here are the packages required.

```
library(keras)
library(tensorflow)
library(tokenizers)
library(dplyr)
library(reticulate)
library(ramify)
library(stringr)
library(deepviz)
```

# Preprocessing (Examining the Dataset)

Importing the English and French Data-set, we take a look at the first 10 rows of the data-set. Based on the source of the data-set it has already been "cleaned", texts were converted to lowercase and it is ensured that there would be spaces between all words and punctuation.

| | English | French |
|---|---|---|
| 1 | new jersey is sometimes quiet during autumn , and it is snowy in april . | new jersey est parfois calme pendant l' automne , et il est neigeux en avril . |
| 2 | the united states is usually chilly during july , and it is usually freezing in november . | les états-unis est généralement froid en juillet , et il gèle habituellement en novembre . |
| 3 | california is usually quiet during march , and it is usually hot in june . | california est généralement calme en mars , et il est généralement chaud en juin . |
| 4 | the united states is sometimes mild during june , and it is cold in september . | les états-unis est parfois légère en juin , et il fait froid en septembre . |
| 5 | your least liked fruit is the grape , but my least liked is the apple . | votre moins aimé fruit est le raisin , mais mon moins aimé est la pomme . |
| 6 | his favorite fruit is the orange , but my favorite is the grape . | son fruit préféré est l'orange , mais mon préféré est le raisin . |
| 7 | paris is relaxing during december , but it is usually chilly in july . | paris est relaxant en décembre , mais il est généralement froid en juillet . |
| 8 | new jersey is busy during spring , and it is never hot in march . | new jersey est occupé au printemps , et il est jamais chaude en mars . |
| 9 | our least liked fruit is the lemon , but my least liked is the grape . | notre fruit est moins aimé le citron , mais mon moins aimé est le raisin . |
| 10 | the united states is sometimes busy during january , and it is sometimes warm in november . | les états-unis est parfois occupé en janvier , et il est parfois chaud en novembre . |

# Preprocessing (Tokenisation)

Tokenisation is a method which converts texts to numerical values. This would allow for the neural network to perform mathematical operations on the input data. Through tokenisation, each unique word and punctuation will be assigned a unique number to represent them.

Upon running the tokeniser function, it returns a word index (linking the words and their unique number) and the text in their numeric form.

```
[1] "Sequence in Text 1:"
[1] "Input: The quick brown fox jumps over the lazy dog ."
[1] "Output: c(1, 2, 4, 5, 6, 7, 1, 8, 9)"

[1] "Sequence in Text 2:"
[1] "Input: By Jove , my quick study of lexicography won a prize ."
[1] "Output: c(10, 11, 12, 2, 13, 14, 15, 16, 3, 17)"

[1] "Sequence in Text 3:"
[1] "Input: This is a short sentence ."
[1] "Output: c(18, 19, 3, 20, 21)"
```

# Preprocessing (Padding)

To feed our sequences of word IDs into the model, each one of them is required to have the same length (both english and french text).

Hence we do this by adding "<PAD >" (unique number 0 to represent it) at the end of a sentence till the sentence reaches the maximum length of a sentence within the data set

# Preprocessing (Result)

Output obtained from the french text of the data set after tokenisation and padding:

```
[1] "Sequence in Text 1:"
[1] "Input: new jersey est parfois calme pendant l' automne , et il est neigeux en avril ."
[1] "Output: c(15, 16, 1, 6, 7, 17, 18, 19, 3, 4, 1, 20, 2, 21)"
[1] "Output (Padded): c(15, 16, 1, 6, 7, 17, 18, 19, 3, 4, 1, 20, 2, 21)"

[1] "Sequence in Text 2:"
[1] "Input: les états-unis est généralement froid en juillet , et il gèle habituellement en novembre ."
[1] "Output: c(8, 9, 10, 1, 5, 11, 2, 22, 3, 4, 23, 24, 2, 25)"
[1] "Output (Padded): c(8, 9, 10, 1, 5, 11, 2, 22, 3, 4, 23, 24, 2, 25)"

[1] "Sequence in Text 3:"
[1] "Input: california est généralement calme en mars , et il est généralement chaud en juin ."
[1] "Output: c(26, 1, 5, 7, 2, 27, 3, 4, 1, 5, 28, 2, 12)"
[1] "Output (Padded): c(26, 1, 5, 7, 2, 27, 3, 4, 1, 5, 28, 2, 12, 0)"

[1] "Sequence in Text 4:"
[1] "Input: les états-unis est parfois légère en juin , et il fait froid en septembre ."
[1] "Output: c(8, 9, 10, 1, 6, 29, 2, 12, 3, 4, 30, 11, 2, 31)"
[1] "Output (Padded): c(8, 9, 10, 1, 6, 29, 2, 12, 3, 4, 30, 11, 2, 31)"

[1] "Sequence in Text 5:"
[1] "Input: votre moins aimé fruit est le raisin , mais mon moins aimé est la pomme ."
[1] "Output: c(32, 13, 14, 33, 1, 34, 35, 36, 37, 13, 14, 1, 38, 39)"
[1] "Output (Padded): c(32, 13, 14, 33, 1, 34, 35, 36, 37, 13, 14, 1, 38, 39)"
```

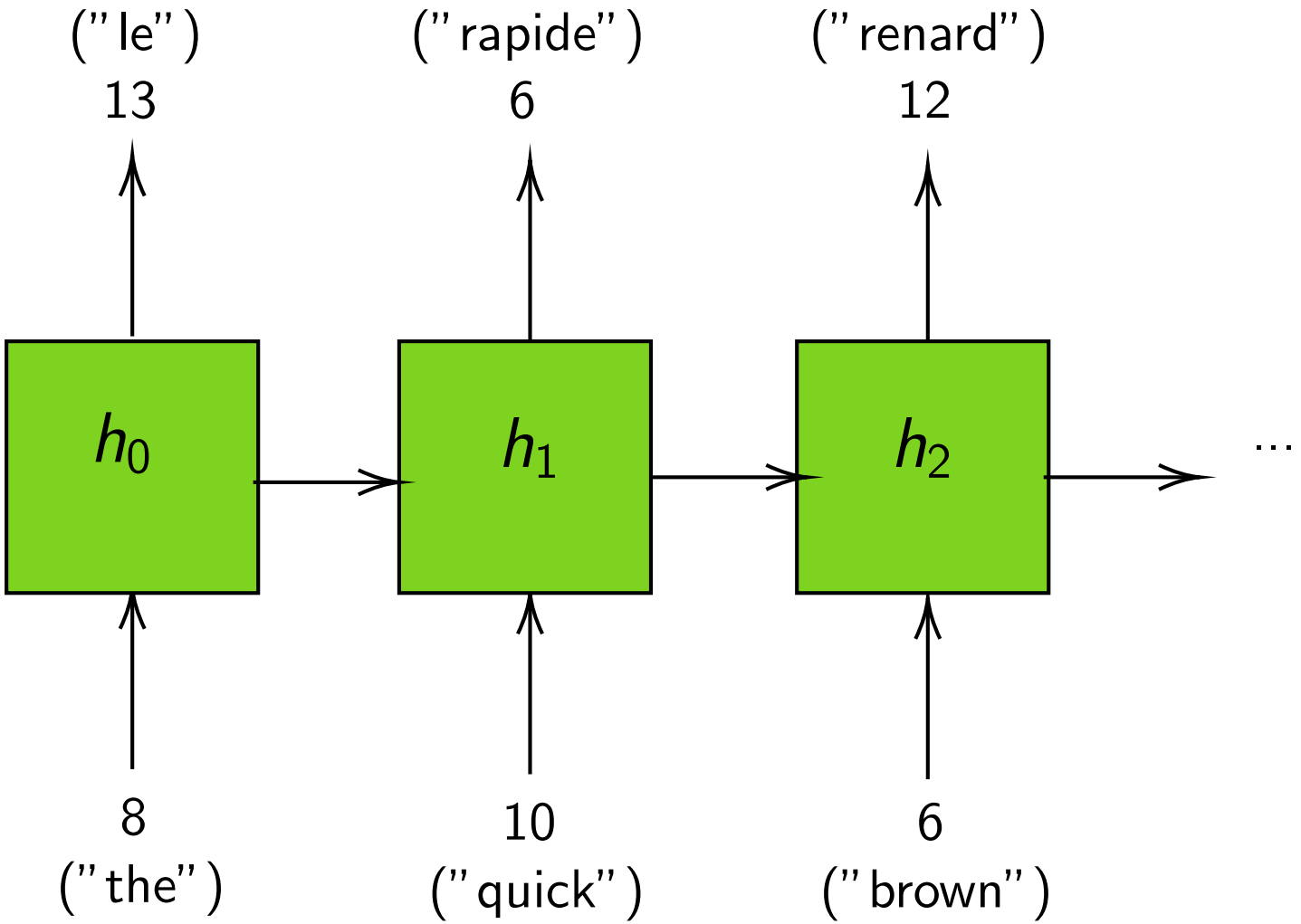
# Table of Contents

# Conversion to Tensor

Due to RNN only accepting 3D Tensors as input, we would convert the 2D matrix into 3D by giving its third dimension a value of 1.

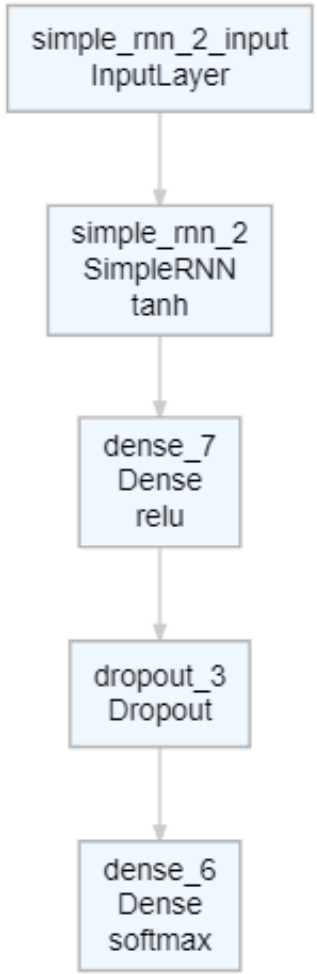
# RNN Model

In language translation we will be looking at a simple RNN model:



# RNN Model

RNN model layers (deepviz):

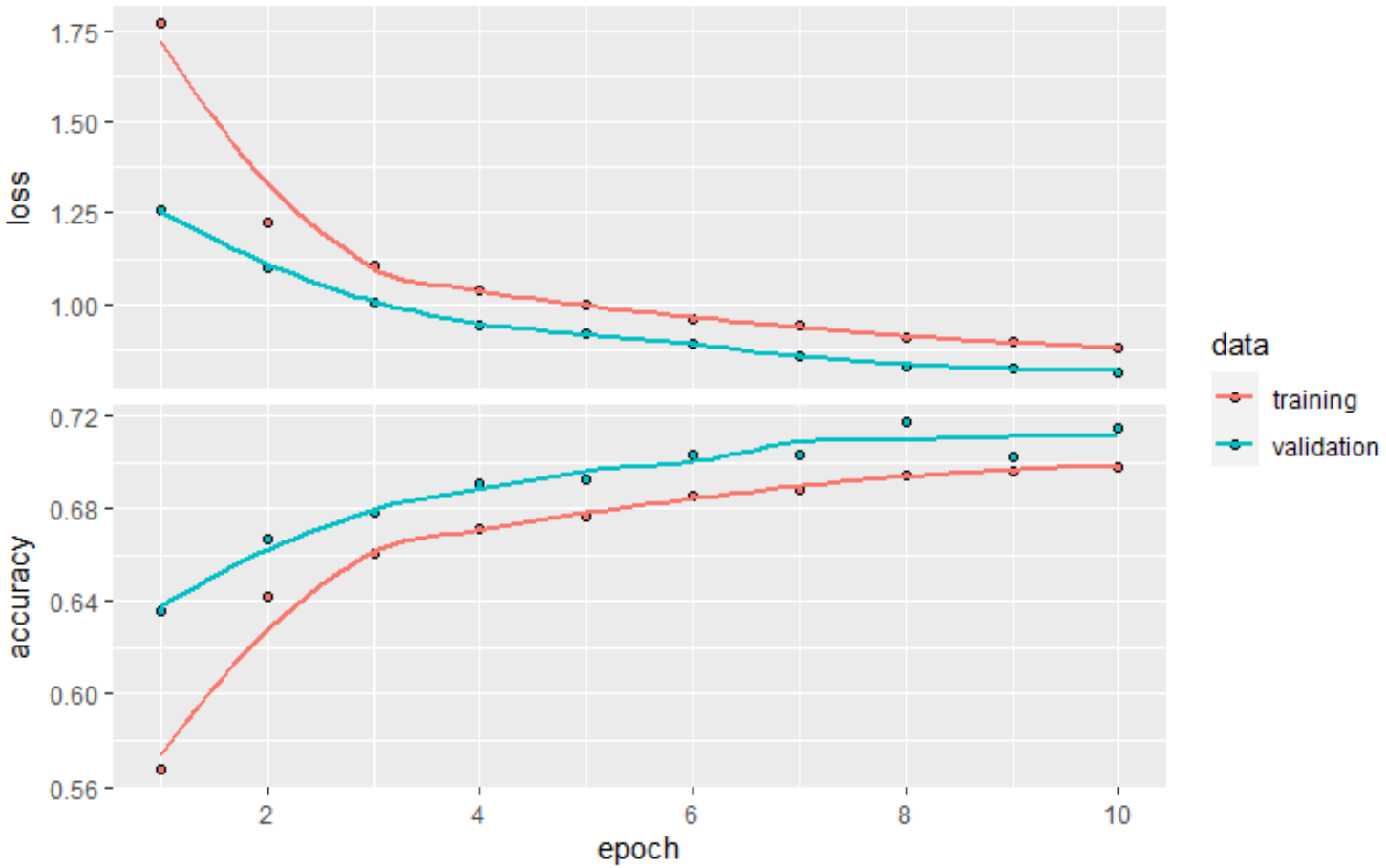



**Link to reason for Dropouts can be found under the References

# RNN Model Parameters

- Loss: Sparse Categorical Crossentropy
- Optimiser: Adam (Learning Rate = 0.005)
- Metrics: Accuracy

# Training Results




Completing the trained model gives us approximately a final accuracy of **69.83%** and validation accuracy of **71.49%**

# Table of Contents

# Decoding Logit to Text

From the output of the prediction, logits have all shifted up by a value of one. Hence when attempting to decode it, we balance the shift by subtracting one from all the logits obtained via argmax on the prediction output.

Then we use the word index to convert these logits back to text.

# Prediction Attempt

As we can see the prediction, although not perfect, is very similar to the actual translation of the english text.

```
[1] "Input sentence: your least liked fruit is the grape , but my least liked is the apple ."
[1] "Intended Output Sentence: votre moins aimé fruit est le raisin , mais mon moins aimé est la pomme ."
[1] "Predicted Output Sentence: votre fruit est moins aimÃ© la mais mais moins moins aimÃ© est la"
```

# Table of Contents

# References


- Week 10 slides from 50.038: Computational Data Science by Prof. Soujanya Poria
- Poorly Translated French to English image: My French Life™ - Ma Vie Française®
- The deep learning-simple problem image: Reddit
- Sine wave example: Analytics Vidhya
- Mathcha to create tikz diagrams
- Tracey, T. (29 August, 2018). Language Translation with RNNs
- Adrian, G.(22 June, 2018). A review of Dropout as applied to RNNs
- Niklas Donges (29 July, 2021) A Guide to RNN: Understanding Recurrent Neural Networks and LSTM Networks