RecurrentNeuralNetworks

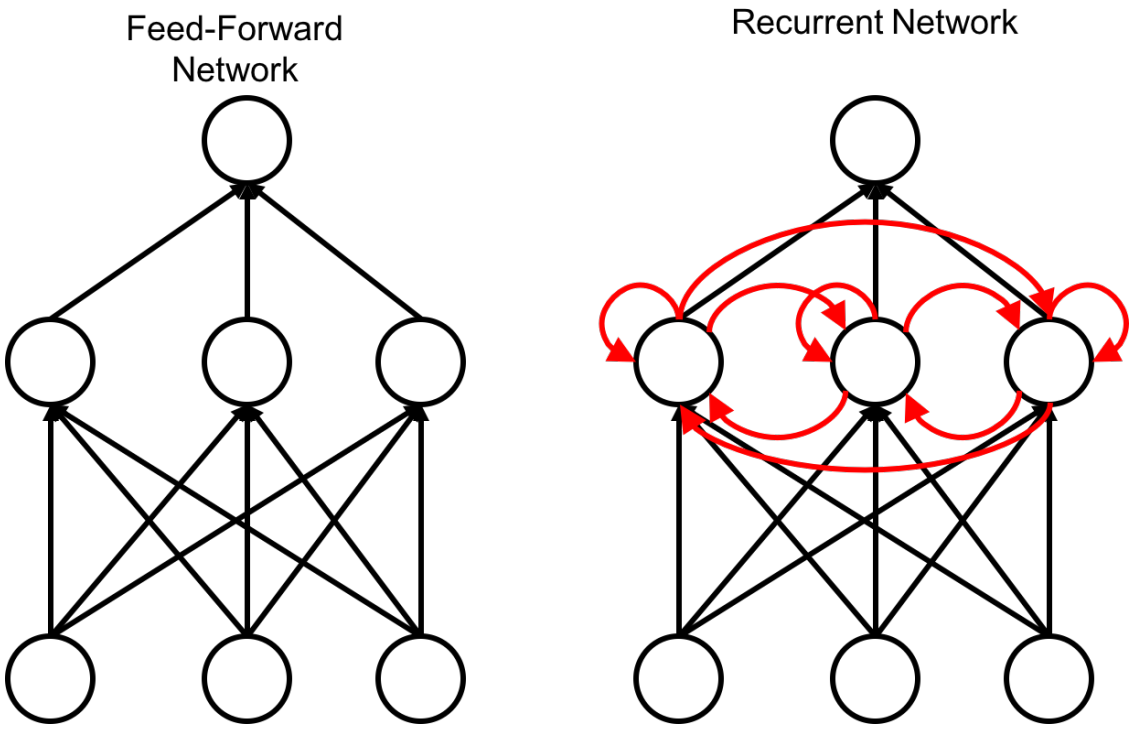
January 27, 2019

1 Feed-Forward Vs. Recurrent Neural Networks

So far we have only covered feed-forward neural networks i.e. networks where the computations flow from input to output and units are only connected to units in downstream layers. However, if we allow connections within a layer or to previous layers then we get a recurrent neural network. Below is a simple diagram comparing the architecture of a feedforward neural network to a recurrent neural network (the recurrent connections have been highlighted in red):

In [10]:
```
from IPython.display import Image
Image(filename='FFWVsRecurrent.png')
```

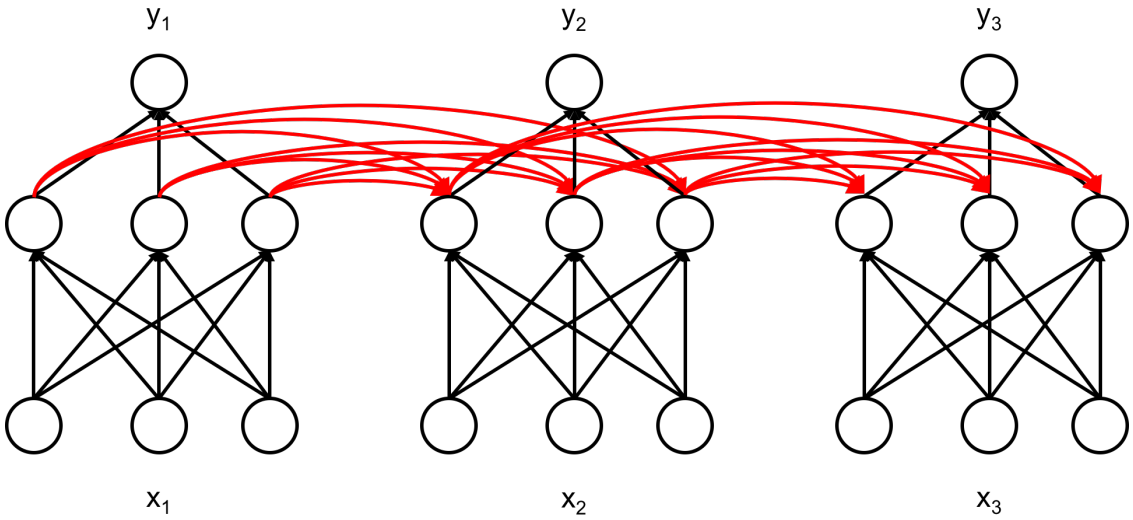
Out[10]:



Recurrent neural networks are fundamentally different to feedforward neural networks in that they allow for 'contextual processing', whereby the incoming input is processed within the 'context' of the network's internal state. A feedforward network can only use the information provided by the input to calculate the activation values of its units. It this ability to perform contextual processing that makes recurrent neural networks particularly suited to problems that involve sequences e.g. understanding written text or speech. For example, imagine we wanted to feed a sentence in English into a neural network and translate it into French. If we gave a feedforward neural network one word at a time then it would only be able to use information from the current word to output a word in french. In comparison, a recurrent neural network would have access to both information from the current word and information about previously presented words through the input and internal state respectively. We can illustrate this using a diagram:

```
In [11]: Image(filename='Unfolded.png')
```

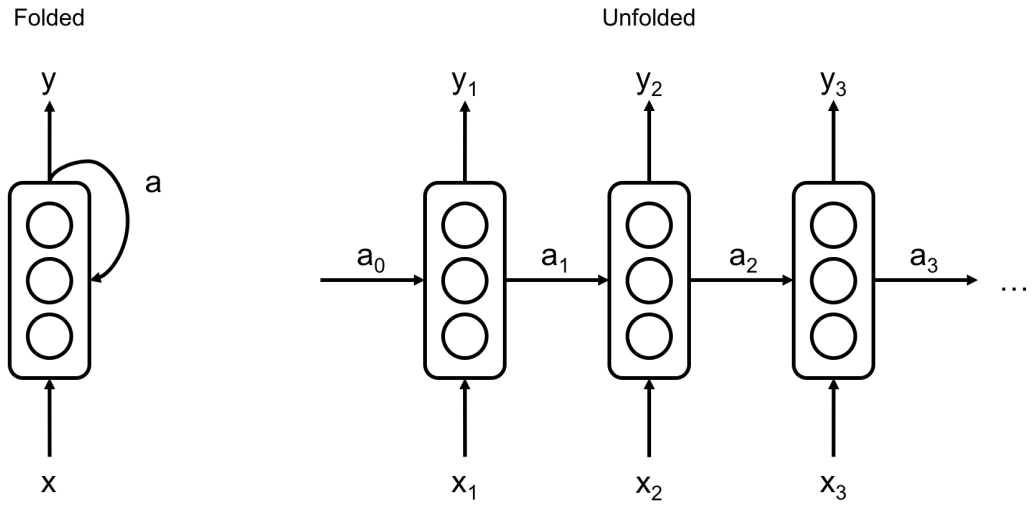
```
Out[11]:
```



When we draw a recurrent neural network like this as a series of steps we say it has been unfolded in time. Hopefully you can see how the connections between units in the hidden layer allow for the passing of information from previous inputs to the current input. It is this information that we refer to when we talk about the internal state of the network. In effect, this internal state gives the recurrent neural network a form of memory, whereby past input can have an effect on the current input. The above diagram is quite messy and you can imagine this problem would be much worse if we were to draw a larger network. It is therefore common to draw recurrent neural networks using 'box diagrams' as shown below:

```
In [12]: Image(filename='RecurrentBoxDiagram.png')
```

```
Out[12]:
```



In this diagram, $\mathbf{x}$, $\mathbf{a}$ and $\mathbf{y}$ are all vectors with each element of the vector containing a value for an individual unit. You will often hear the box for the hidden layer referred to as a 'cell', which really just means a collection of units, with their activations represented as a vector.

Some of you may have noticed that we could actually give a feedforward neural network information about other inputs by including them as one single input. For instance, in the example of translating from English to French, we could give all the english words to a feedforward network at once and then train it to produce the full French sentence. While in principle this could work, there are two main reasons why recurrent neural networks are still favoured over this approach. Firstly, recurrent neural networks are able to deal with input sequences of varying lengths whereas a feedforward neural network typically requires inputs that are of the same dimension. Secondly, a recurrent neural network can learn feature repsentations that generalise to other parts of the sequence. For example, a recurrent neural network may learn that 'I am' should be translated to 'je suis' and can use this information to translate 'I am' regardless of whether it occurs at the start or end of a sentence. In comparison, a feedforward neural network would need to learn from examples that contained 'I am' in many different locations. Recurent neural networks therefore generalise over sequences in a manner that is similar to how convolutional neural networks generalize over space. It is also worth noting that because a recurrent neural network uses the same parameters for each step it will require significantly less parameters than a feedforward network if the sequences are long.

2 Forward Propagation

To be more concrete about how information flows through a simple recurrent neural network lets define the forward propagation process mathematically. Lets use the following notation for the activation values of the different layers:

$\mathbf{x_t} \rightarrow$ vector of input unit activations at time t.

$\mathbf{a_t} \rightarrow$ vector of hidden unit activations at time t.

$\mathbf{y_t} \rightarrow$ vector of output unit activations at time t.

Similarly lets define our notation for the matrices that will contain the weight values for the different connections between units:

$\mathbf{W_{ax}} \rightarrow$ matrix of weight values for the connections from the input units $\mathbf{x}$ to the hidden units $\mathbf{a}$.

$\mathbf{W_{aa}} \rightarrow$ matrix of weight values for the connections from the hidden units $\mathbf{a}$ to the hidden units $\mathbf{a}$.

$\mathbf{W_{ya}} \rightarrow$ matrix of weight values for the connections from the hidden units $\mathbf{a}$ to the output units $\mathbf{y}$.

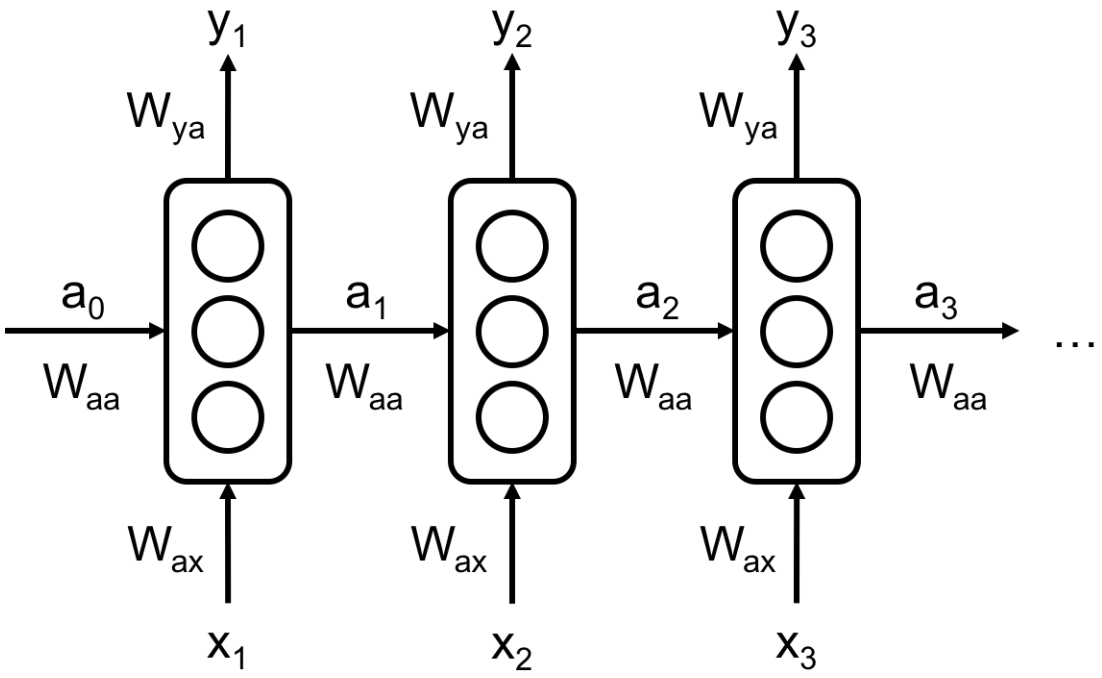
$\mathbf{b_a} \rightarrow$ vector of bias values for the hidden units $\mathbf{a}$.

$\mathbf{b_y} \rightarrow$ vector of bias values for the output units $\mathbf{y}$.

Again note how we only need to define one matrix for each set of connections because it will be re-used for each step of the recurrent neural network. With this notation in mind we can draw our simple recurrent neural network as follows:

```
In [13]: Image(filename='ForwardPropDiagram.png')
```

```
Out[13]:
```



Hopefully this diagram makes it clear what our equations should be to perform forward propagation. Firstly, we calculate our hidden layer activations $\mathbf{a_t}$ by adding the weighted sum of the previous steps activation values $\mathbf{a_{t-1}}$ to the weighted sum of the current steps input values $\mathbf{x_t}$ and applying a non-linear activation function $g_1()$:

$$\mathbf{a_t} = g_1(\mathbf{W_{aa}a_{t-1}} + \mathbf{W_{ax}x_t} + \mathbf{b_a})$$

On the very first step of the recurrent neural network the hidden layer activations $\mathbf{a_0}$ are typically initialised to a vector of zeros or small random values. Once the hidden layer activation values have been calculated they are then used to calculate the current time steps output values $\mathbf{y_t}$ and are also passed on to the next time step:

$$\mathbf{y_t} = g_2(\mathbf{W_{ya}a_t} + \mathbf{b_y})$$

These two equations are just repeated for each step of the recurrent neural network. Hopefully you can see that the only real difference compared to feedforward neural networks is the additional weighted sum of the previous steps activation values.

As an implementational note it is common to use a single matrix $\mathbf{W_a}$ to represent all of the incoming weights for the hidden units. This matrix is formed by just concatenating $\mathbf{W_{aa}}$ and $\mathbf{W_{ax}}$ horizontally as follows:

$$\mathbf{W_a} = \begin{bmatrix} \mathbf{W_{aa}} & \mathbf{W_{ax}} \end{bmatrix}$$

The previous steps activations and the current steps inputs are then also concatenated so that the equation for calculating the hidden activations can be written using a single matrix multiplication:

$$\mathbf{a_t} = g_1(\mathbf{W_a} \begin{bmatrix} \mathbf{a_{t-1}} \\ \mathbf{x_t} \end{bmatrix} + \mathbf{b_a})$$

3 Backpropagation Through Time

The unfolded diagrams above should give you a fairly good sense of how we perform backpropagation in a recurrent neural network. In effect we propagate the partial derivatives in the opposite direction to the arrows for forward propagation. As a result backpropagation in recurrent neural networks is often called backpropagation through time because the partial derivatives are passed backwards to previous time steps via the hidden activations.

In terms of defining an objective / cost function, we generally set it as the sum of the individual object / cost functions for each $\mathbf{y_t}$ in the sequence. For example if we assumed that $p_{model}(y \mid \mathbf{x})$ has a gaussian distribution and we wanted to predict the mean of the distribution then our cost function for calculating the maximum likelihood solution would be:

$$J(\theta) = \sum_{t=0}^{T} J^t(\theta)$$
$$J^t(\theta) = (y^t - f(\mathbf{x^t}, \mathbf{x^{t-1}}, ..., \mathbf{x^1}, \theta))^2$$

It is then relatively straightforward to calculate the partial derivatives required to update our parameters because we are dealing with sums. Of course we don't have to output a prediction for every step of the recurrent neural network. Some sequence problems just require us to make a prediction at the end of a sequence or to predict a sequence only after the whole input sequence has been presented. Indeed, there are some problems where the input is just a single step and the network has to then predict a whole sequence. These different types of problems don't represent a problem for backpropagation through time because we can just include the neccessary cost terms and backpropagate through the hidden layer activations accordingly.

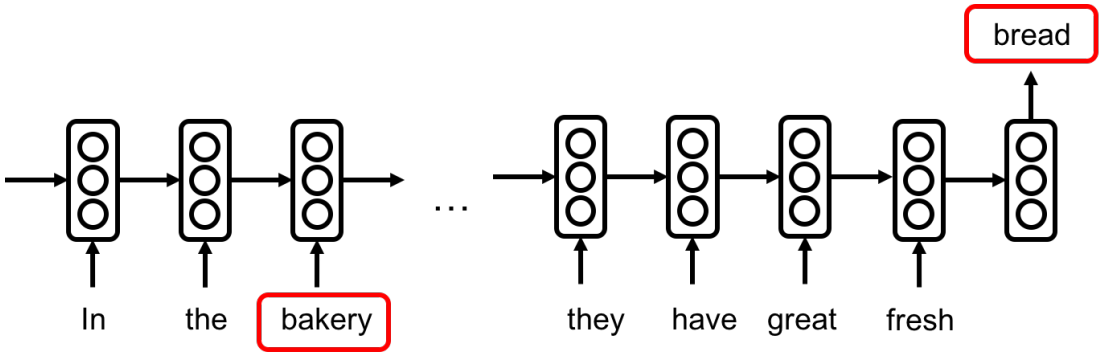
4 Vanishing Gradients

As mentioned in the notebook on feedforward neural networks, the 'vanishing graident' problem is one of the main culprits for poor neural network training. This problem is compounded when you add more layers to a feed forward neural network because the partial derivitives get

smaller and smaller as you move back through the layers. Unfortunately recurrent neural networks are particularly susceptible to the vanishing gradient problem because of the backpropagation through time. It is often the case in sequence problems that the information from inputs many steps ago may be needed to make a prediction at the current step. This phenomenon is commonly called a long-term dependency and it requires the partial derivatives from the current step to be backpropagated several steps backward in time. This is a similar problem to backpropagating the partial derivatives through many layers of a deep feed-forward neural network and so vanishing gradients can become extrememly common. Below is an example of a long-term dependency when trying to predict the next word in a sentence:

```
In [14]: Image(filename='LongRangeDependency.png')
```

```
Out[14]:
```



In order to predict the correct word 'bread', the recurrent neural network needs to be able to use information from the word 'bakery' from several steps ago. Without this information the network could predict several other suitable, yet incorrect, words e.g. 'fruit'. To be able to learn parameters that use this information the recurrent neural network needs to be able to backpropagate the partial derivatives over several steps, however this often results in vanishing gradients.

In summary, vanishing gradients prevent simple recurrent neural networks from being able to solve sequence problems that require any kind of long-term dependencies. As a result various modifications to recurrent neural networks have been proposed to alleviate the vanshing gradient problem. The most common modifications to simple recurrent neural networks are known as Gated Recurrent Units (GRUs) and Long Short-Term Memory units (LSTMs), both of which will be the subject of the next notebook. In general, these modifications rely upon gating mechanisms within the units themselves to store information for long-range dependencies.

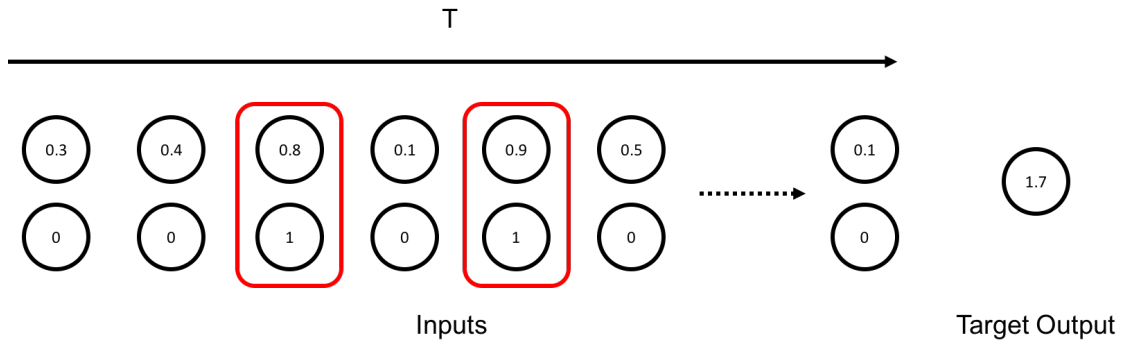
5 Example of a Simple Recurrent Neural Network - The Adding Problem

To demonstrate how a simple recurrent neural network works in practice we shall train one on a well known problem known as the 'adding problem'. In this problem the network is a fed a

sequence of two-dimensional vectors and it has to output a single real-valued number at the end of the sequence. The first entry of the input vector consists of a random decimal number between 0 and 1. The second entry of the input vector is binary and indicates which two numbers from the first dimension should be added together in the sequence. The correct output of the network is then the sum of the two first dimension numbers indicated by the binary mask of the second dimension. Below is a diagram illustrating the adding problem:

In [15]:
```
Image(filename='AddingProblem.png')
```

Out[15]:



This problem is ideal for recurrent neural networks because it involves a sequence of inputs and the network must 'remember' past inputs in order to predict the correct output value. To start with lets generate our training and test sets for the adding problem using the code below:

In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 100

def GenerateAddingProblemData(num_examples, sequence_length):

    numbers = np.random.uniform(low=0, high=1, size=(num_examples,
                                                     sequence_length,
                                                     1))
    mask = np.zeros((num_examples, sequence_length, 1))
    Y = np.zeros((num_examples, 1))

    for i in range(num_examples):
        numbers_to_add = np.random.choice(sequence_length, size=2,
                                          replace=False)
        mask[i, numbers_to_add] = 1
        Y[i, 0] = np.sum(numbers[i, numbers_to_add])
```

```
            X = np.append(numbers, mask, axis=2)

            return X, Y

        X_train, Y_train = GenerateAddingProblemData(10000, 6)
        X_test, Y_test = GenerateAddingProblemData(1000, 6)
```

Now we have our training and test sets we need to define our recurrent neural network class. As with feedforward networks, the Train() method of our recurrent neural network will alternate between forward propagation and backpropagation on randomly sampled minibatches. Importantly however, the recurrent neural network backpropagates through time, starting at the last time step and propagating the gradients back to the first time step. The gradients calculated at each time step are then summed in order to get the overall gradient required for the parameter update.

```
In [2]: class RNN(object):

            def __init__(self, sequence_length, num_hidden, learning_rate):

                self.sequence_length = sequence_length
                self.learning_rate = learning_rate
                self.minibatch_size = 32

                self.num_units = {'Input': 2, 'Hidden': num_hidden,
                                  'Output': 1}
                self.weights = {'Wa': np.random.randn(
                    self.num_units['Hidden'], self.num_units['Hidden'] +
                    self.num_units['Input']) * np.sqrt(
                    1/self.num_units['Hidden'] + self.num_units['Input']),
                                'Wya': np.random.randn(
                                    self.num_units['Output'],
                                    self.num_units['Hidden']) * np.sqrt(
                                    1/self.num_units['Hidden'])}

                self.biases = {'ba': np.random.randn(
                    self.num_units['Hidden'], 1) / 100,
                               'by': np.random.randn(
                                   self.num_units['Output'], 1) / 100}

                return

            def ForwardPropagation(self, X_batch):

                num_examples = X_batch.shape[0]
                self.activations = {'t0': np.zeros((
                    self.num_units['Hidden'], num_examples))}
```

```
        X_batch = X_batch.transpose()
        self.X_batch = X_batch

        for t in range(self.sequence_length):
            net = np.matmul(self.weights['Wa'], np.concatenate(
                [self.activations['t' + str(t)],
                 X_batch[:, t, :]], axis=0)) + np.tile(
                self.biases['ba'], (1, num_examples))

            self.activations['t' + str(t + 1)] = np.tanh(net)

        y = np.matmul(self.weights['Wya'],
                      self.activations['t' + str(
                          self.sequence_length)]) + np.tile(
            self.biases['by'], (1, num_examples))

        return y.transpose()

    def CalculateCost(self, y_pred, y):
        return np.mean(np.squeeze(np.square(y_pred - y)),
                       axis=-1)


    def PropagateOneStepBack(self, da_next, t):

        Waa = self.weights['Wa'][:,
                                 :self.num_units['Hidden']]
        Wax = self.weights['Wa'][:,
                                 self.num_units['Hidden']:]

        a_next = self.activations['t' + str(t)]
        a_prev = self.activations['t' + str(t-1)]
        xt = self.X_batch[:, t-1, :]
        dtanh = (1 - a_next ** 2) * da_next

        dWax = np.dot(dtanh, xt.T)
        da_prev = np.dot(Waa.T, dtanh)
        dWaa = np.dot(dtanh, a_prev.T)
        dba = np.sum(dtanh, axis=1, keepdims=1)

        gradients = {"da_prev": da_prev, "dWax": dWax,
                     "dWaa": dWaa, "dba": dba}

        return gradients


    def BackPropagation(self, y_pred, y):
```

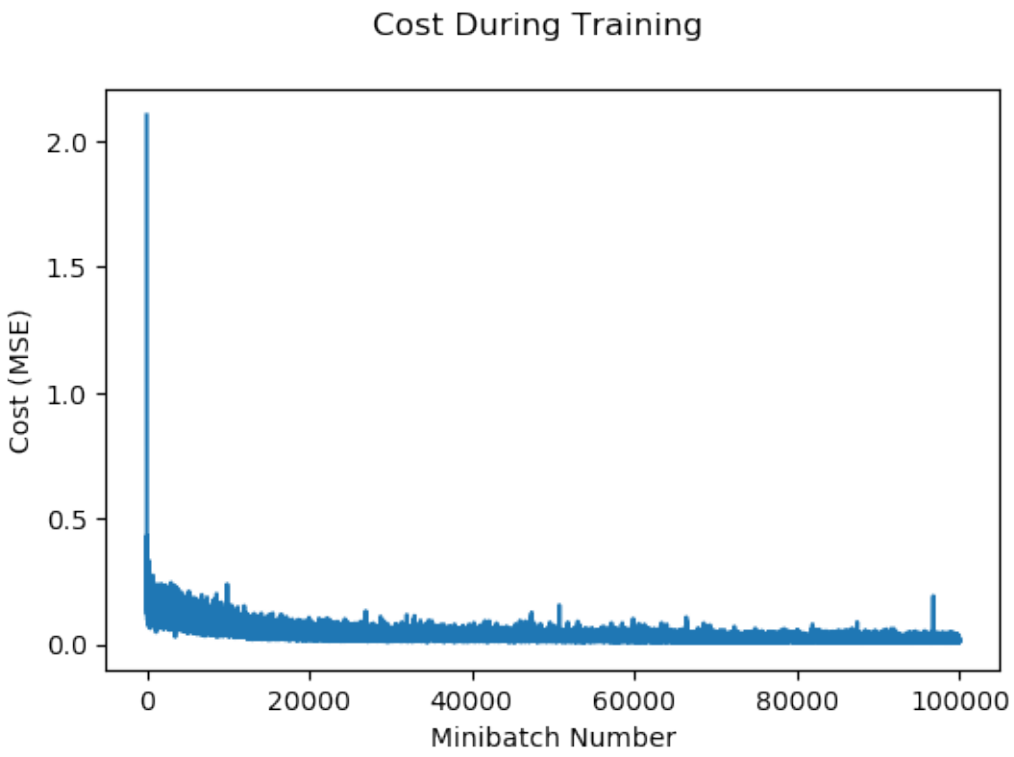
```
        dWax = np.zeros((self.num_units['Hidden'],
                         self.num_units['Input']))
        dWaa = np.zeros((self.num_units['Hidden'],
                         self.num_units['Hidden']))
        dba = np.zeros((self.num_units['Hidden'], 1))

        delta = y_pred - y

        da_prevt = np.matmul(
            delta, self.weights['Wya']).transpose()
        dWya = np.matmul(self.activations['t' + str(
            self.sequence_length)], delta).transpose()
        dby = np.sum(delta)


        for t in reversed(range(self.sequence_length)):

            gradients = self.PropagateOneStepBack(da_prevt, t+1)

            da_prevt, dWaxt, dWaat, dbat = gradients[
                "da_prev"], gradients["dWax"], gradients[
                "dWaa"], gradients["dba"]

            dWax += dWaxt
            dWaa += dWaat
            dba += dbat

        gradients = {"dWya": dWya, "dby": dby, "dWax": dWax,
                     "dWaa": dWaa, "dba": dba}

        return gradients



    def Train(self, X_train, y_train, num_iters):

        costs = []

        for i in range(num_iters):

            inds = np.random.randint(0, X_train.shape[0],
                                     self.minibatch_size)

            y_pred = self.ForwardPropagation(X_train[inds, :])
            cost = self.CalculateCost(y_pred, y_train[inds, :])
            costs.append(cost)

            gradients = self.BackPropagation(
```

```
                    y_pred, y_train[inds, :])

                self.weights['Wya'] -= self.learning_rate * gradients[
                    'dWya']
                self.weights['Wa'][:, :self.num_units['Hidden']
                                   ] -= self.learning_rate * gradients[
                    'dWaa']
                self.weights['Wa'][:, self.num_units['Hidden']:
                                   ] -= self.learning_rate * gradients[
                    'dWax']

                self.biases['by'] -= self.learning_rate * gradients[
                    'dby']
                self.biases['ba'] -= self.learning_rate * gradients[
                    'dba']


                if (i % 10000 == 0):
                    print('iter: ' + str(i) + ' Cost: ' + str(cost))

            return costs
```

With our RNN class defined lets train the network on the training data and visualize the cost during training to make sure it is decreasing:

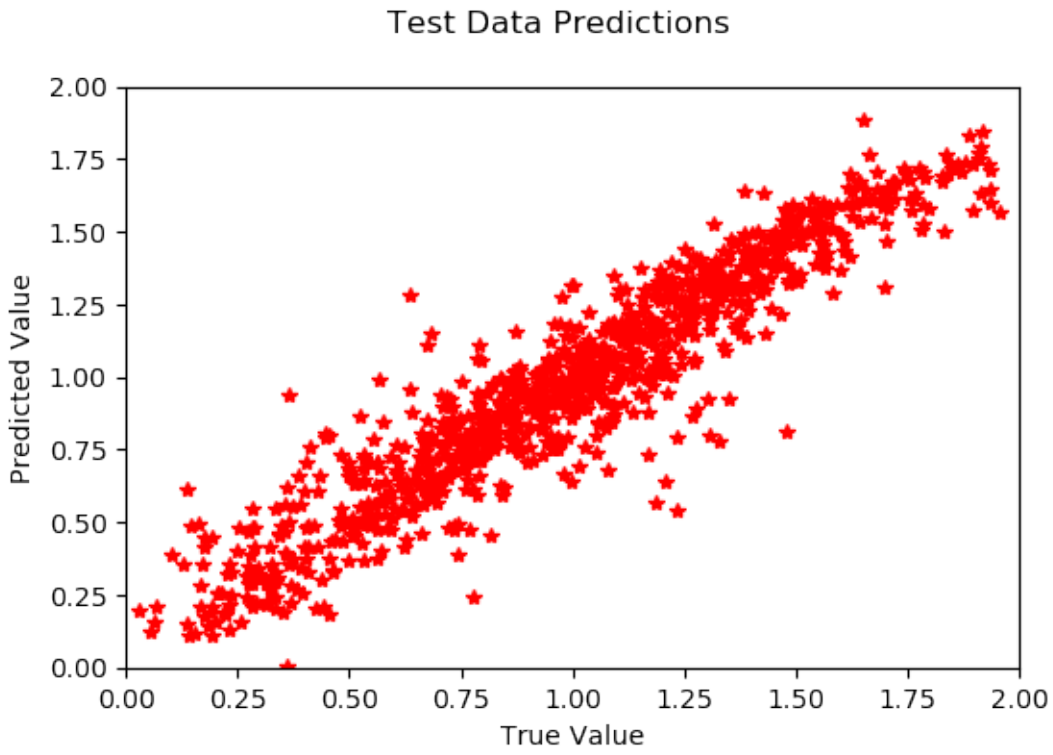
```
In [3]: rnn = RNN(6, 64, .001)
        costs = rnn.Train(X_train, Y_train, 100000)

        plt.figure()
        plt.suptitle('Cost During Training')
        plt.plot(costs)
        plt.xlabel('Minibatch Number')
        plt.ylabel('Cost (MSE)')
        plt.show()
```

```
iter: 0 Cost: 2.10309402889
iter: 10000 Cost: 0.0904310920005
iter: 20000 Cost: 0.0272517259151
iter: 30000 Cost: 0.0298038930579
iter: 40000 Cost: 0.0123323699142
iter: 50000 Cost: 0.0294346754789
iter: 60000 Cost: 0.0178820374561
iter: 70000 Cost: 0.0167823392853
iter: 80000 Cost: 0.0249010660264
iter: 90000 Cost: 0.0139247944983
```



Finally lets get the predictions of the trained network on the test data and plot them against the actual test data to visualize how well the network generalizes to unseen data:

```
In [4]: y_pred = rnn.ForwardPropagation(X_test)

        plt.figure()
        plt.suptitle('Test Data Predictions')
        plt.plot(Y_test, y_pred, 'r*')
        plt.xlim((0,2))
        plt.ylim((0,2))
        plt.xlabel('True Value')
        plt.ylabel('Predicted Value')
        plt.show()
```



The plot above shows that our RNN is able to generate some sensible predictions on the test data. With that being said there is still a fair amount of deviation from the true values. We shall see in the next notebook how we can improve the performance of our network by using different kinds of units. As mentioned previously, these modifications (GRUs and LSTMs) to standard recurrent units are specifically designed to tackle the vanishing gradient problem and are therefore much better at handling long range dependencies.