Recurrent Neural Networks

So far, we've been looking at convolutional neural networks and models that allows us to analyze the spatial information in a given input image. CNN's excel in tasks that rely on finding spatial and visible patterns in training data.

In this and the next couple lessons, we'll be reviewing RNN's or recurrent neural networks. These networks give us a way to incorporate **memory** into our neural networks, and will be critical in analyzing sequential data. RNN's are most often associated with text processing and text generation because of the way sentences are structured as a sequence of words, but they are also useful in a number of computer vision applications, as well!

RNN's in Computer Vision

At the end of this lesson, you will be tasked with creating an automatic image captioning model that takes in an image as input and outputs a *sequence* of words, describing that image. Image captions are used to create accessible content and in a number of other cases where one may want to read about the contents of an image. This model will include a CNN component for finding spatial patterns in the input image *and* and RNN component that will be responsible for generative descriptive text!

RNN's are also sometimes used to analyze sequences of images; this can be useful in captioning video, as well as video classification, gesture recognition, and object tracking; all of these tasks see as input a *sequence* of image frames.

Sketch RNN

One of my favorite use cases for RNN's in computer vision tasks is in generating drawings. Sketch RNN (demo here) is a program that learns to complete a drawing, once you give it something (a line or circle, etc.) to start!

Sketch RNN example output. Left, Mona Lisa. Right, pineapple.

It's interesting to think of drawing as a sequential act, but it is! This network takes a starting line or squiggle and then, having trained on a number of types of sketches, does it's best to complete the drawing based on your input squiggle.

Next, you'll learn all about how RNN's are structured and how they can be trained! This section is taught by Ortal, who has a PhD in Computer Engineering and has been a professor and researcher in the fields of applied cryptography and embedded systems.

Supporting Materials

Video classification methods

RNN HISTORY

As mentioned in this video, RNNs have a key flaw, as capturing relationships that span more than 8 or 10 steps back is practically impossible. This flaw stems from the "**vanishing gradient**" problem in which the contribution of information decays geometrically over time.

What does this mean?

As you may recall, while training our network we use **backpropagation**. In the backpropagation process we adjust our weight matrices with the use of a **gradient**. In the process, gradients are calculated by continuous multiplications of derivatives. The value of these derivatives may be so small, that these continuous multiplications may cause the gradient to practically "vanish".

LSTM is one option to overcome the Vanishing Gradient problem in RNNs.

Please use these resources if you would like to read more about the Vanishing Gradient problem or understand further the concept of a Geometric Series and how its values may exponentially decrease.

If you are still curious, for more information on the important milestones mentioned here, please take a peek at the following links:

- TDNN
- Here is the original Elman Network publication from 1990. This link is provided here as it's a significant milestone in the world on RNNs. To simplify things a bit, you can take a look at the following additional info.
- In this LSTM link you will find the original paper written by Sepp Hochreiter and Jürgen Schmidhuber. Don't get into all the details just yet. We will cover all of this later!

As mentioned in the video, Long Short-Term Memory Cells (LSTMs) and Gated Recurrent Units (GRUs) give a solution to the vanishing gradient problem, by helping us apply networks that have temporal dependencies. In this lesson we will focus on RNNs and continue with LSTMs. We will not be focusing on GRUs. More information about GRUs can be found in the following blog. Focus on the overview titled: **GRUs**.

RNN APPLICATION

There are so many interesting applications, let's look at a few more!

- Are you into gaming and bots? Check out the DotA 2 bot by Open AI

- How about automatically adding sounds to silent movies?
- Here is a cool tool for automatic handwriting generation
- Amazon's voice to text using high quality speech recognition, Amazon Lex.
- Facebook uses RNN and LSTM technologies for building language models
- Netflix also uses RNN models - here is an interesting read

Feedforward

If you are not feeling confident with linear combinations and matrix multiplications, you can use the following links as a refresher:

- Linear Combination
- Matrix Multiplication

Vector h' of the hidden layer will be calculated by multiplying the input vector with the weight matrix

W^1 the following way: $\bar{h'} = (\bar{x}W^1)$

Using vector by matrix multiplication, we can look at this computation the following way:

$$\begin{bmatrix} h'_1 & h'_2 & h'_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & ..... & x_n \end{bmatrix} \cdot \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ \vdots & & \\ W_{n1} & W_{n2} & W_{n3} \end{bmatrix}$$

Equation 1

After finding , we need an activation function (Φ) to finalize the computation of the hidden layer's values. This activation function can be a Hyperbolic Tangent, a Sigmoid

or a ReLU function. We can use the following two equations to express the final hidden vector $\bar{h}$

$$\bar{h} = \Phi(\bar{x}W^1)$$

or

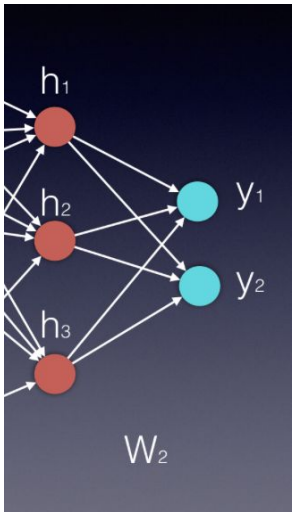
$$\bar{h} = \Phi(h')$$

Since W_{ij} represents the weight component in the weight matrix, connecting neuron **i** from the input to neuron **j** in the hidden layer, we can also write these calculations in the following way: (notice that in this example we have *n* inputs and only 3 hidden neurons)

$$h_1 = \Phi(x_1W_{11}+x_2W_{21}+....+x_nW_{n1})$$
$$h_2 = \Phi(x_1W_{12}+x_2W_{22}+....+x_nW_{n2})$$
$$h_3 = \Phi(x_1W_{13}+x_2W_{23}+....+x_nW_{n3})$$

More information on the activation functions and how to use them can be found here

The process of calculating the output vector is mathematically similar to that of calculating the vector of the hidden layer. We use, again, a vector by matrix multiplication, which can be followed by an activation function. The vector is the newly calculated hidden layer and the matrix is the one connecting the hidden layer to the output.



Essentially, each new layer in an neural network is calculated by a vector by matrix multiplication, where the vector represents the inputs to the new layer and the matrix is the one connecting these new inputs to the next layer.

In our example, the input vector is $\bar{h}$ and the matrix is W^2, therefore $\bar{y} = \bar{h}W^2$. In some applications it can be beneficial to use a softmax function (if we want all output values to be between zero and 1, and their sum to be 1).

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

The two error functions that are most commonly used are the Mean Squared Error (MSE) (usually used in regression problems) and the cross entropy (usually used in classification problems).

Backpropagation Theory

Since partial derivatives are the key mathematical concept used in backpropagation, it's important that you feel confident in your ability to calculate them. Once you know how to calculate basic derivatives, calculating partial derivatives is easy to understand.

For more information on partial derivatives use the following link

For calculation purposes, you can use the following link as a reference for common derivatives.

In the **backpropagation** process we minimize the network error slightly with each iteration, by adjusting the weights.

If we look at an arbitrary layer k, we can define the amount by which we change the weights from neuron i to neuron j stemming from layer k as: ΔW^k

The superscript (*k*) indicates that the weight connects layer *k* to layer *k+1*. Therefore, the weight update rule for that neuron can be expressed as:

$$W_{new} = W_{previous} + \Delta W^k$$

The updated value ΔW_{ij}^k is calculated through the use of the gradient calculation, in the following way:

$$\Delta W_{ij}^k = \alpha(-\frac{\partial E}{\partial W_{ij}^k})$$

α is a small positive number called the **Learning Rate**.

From these derivation we can easily see that the weight updates are calculated by the following equation:

$$W_{new} = W_{previous} + \alpha(-\frac{\partial E}{\partial W})$$

Since many weights determine the network's output, we can use a vector of the partial derivatives (defined by the Greek letter Nabla ∇) of the network error - each with respect to a different weight.

$$W_{new} = W_{previous} + \alpha \nabla_W(-E)$$

Here you can find other good resources for understanding and tuning the Learning Rate:

- resource 1
- resource 2

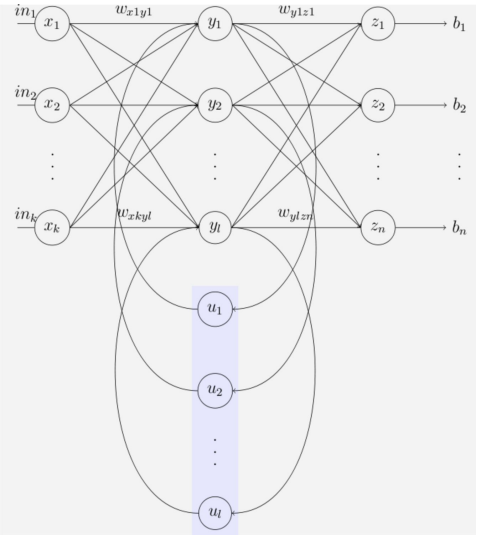
RNNs

RNNs are based on the same principles as those behind FFNNs, which is why we spent so much time reminding ourselves of the feedforward and backpropagation steps that are used in the training phase.

There are two main differences between FFNNs and RNNs. The Recurrent Neural Network uses:

- **sequences** as inputs in the training phase, and
- **memory** elements

Memory is defined as the output of hidden layer neurons, which will serve as additional input to the network during next training step.



The basic three layer neural network with feedback that serve as memory inputs is called the **Elman Network** and is depicted in the following picture:

As mentioned in the *History* concept, here is the original Elman Network publication from 1990. This link is provided here as it's a significant milestone in the world on RNNs. To simplify things a bit, you can take a look at the following additional info.

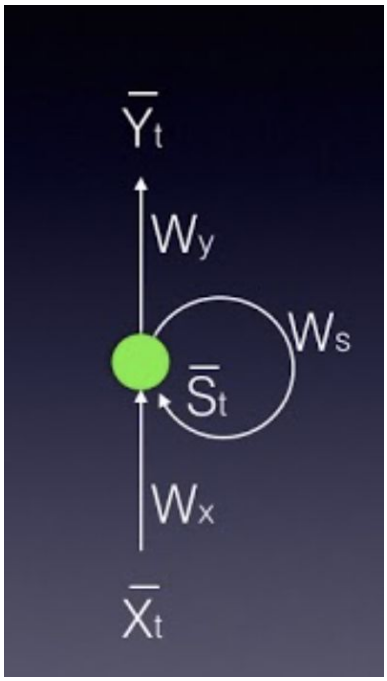
As we've see, in FFNN the output at any time *t*, is a function of the current input and the weights. This can be easily expressed using the following equation:

$$\bar{y}_t = F(\bar{x}_t, W)$$

In RNNs, our output at time t, depends not only on the current input and the weight, but also on previous inputs. In this case the output at time t will be defined as:

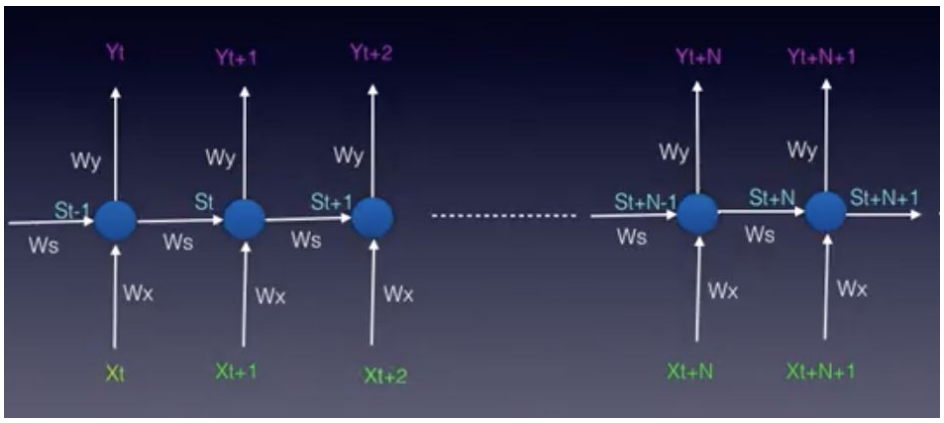
$$\bar{y}_t = F(\bar{x}_t, \bar{x}_{t-1}, \bar{x}_{t-2}, \cdots, \bar{x}_{t-t_0}, W)$$

This is the RNN **folded model**:



In this picture, $\bar{x}$ represents the input vector, $\bar{y}$ represents the output vector and $\bar{s}$ denotes the state vector. W_x is the weight matrix connecting the inputs to the state layer. W_y is the weight matrix connecting the state layer to the output layer. W_s represents the weight matrix connecting the state from the previous timestep to the state in the current timestep.

The model can also be "unfolded in time". The **unfolded model** is usually what we use when working with RNNs.



The RNN unfolded model

In both the folded and unfolded models shown above the following notation is used: $\bar{x}$ represents the input vector, $\bar{y}$ represents the output vector and $\bar{s}$ represents the state vector. W_x is the weight matrix connecting the inputs to the state layer. W_y is the

weight matrix connecting the state layer to the output layer. W_s represents the weight matrix connecting the state from the previous timestep to the state in the current timestep.

In FFNNs the hidden layer depended only on the current inputs and weights, as well as on an activation function Φ in the following way:

$$\bar{h} = \Phi(\bar{x}W)$$

In RNNs the state layer depended on the current inputs, their corresponding weights, the activation function and **also** on the previous state:

$$\bar{s}_t = \Phi(\bar{x}_t W_x + \bar{s}_{t-1} W_s)$$

The output vector is calculated exactly the same as in FFNNs. It can be a linear combination of the inputs to each output node with the corresponding weight matrix W_y, or a softmax function of the same linear combination.

$$\bar{y}_t = \bar{s}_t W_y$$

or

$$\bar{y}_t = \sigma(\bar{s}_t W_y)$$

BPTT

For the error calculations we will use the Loss Function, where E_t represents the output error at time t d_t represents the desired output at time t y_t represents the calculated output at time t

$$E_t = (\bar{d}_t - \bar{y}_t)^2$$

In **BPTT** we train the network at timestep t as well as take into account all of the previous timesteps.

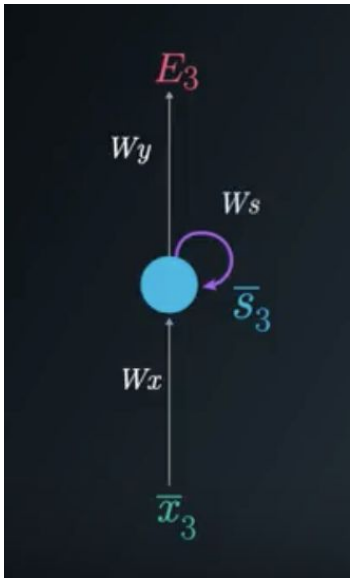
The easiest way to explain the idea is to simply jump into an example.

In this example we will focus on the **BPTT** process for time step t=3. You will see that in order to adjust all three weight matrices, W_x, W_s and W_y, we need to consider timestep 3 as well as timestep 2 and timestep 1.

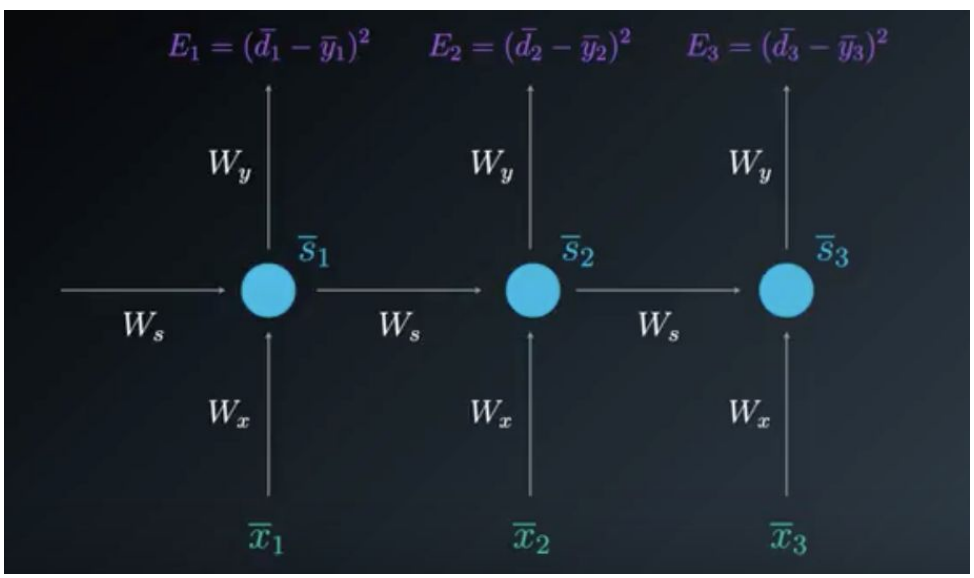
As we are focusing on timestep t=3, the Loss function will be:

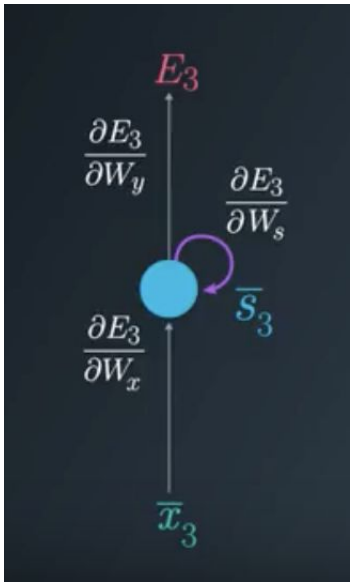
$$E_3 = (\bar{d}_3 - \bar{y}_3)^2$$

To update each weight matrix, we need to find the partial derivatives of the Loss Function at time 3, as a function of all of the weight matrices. We will modify each matrix using gradient descent while considering the previous timesteps.



The unfolded model can be very helpful in visualizing the BPTT process.





Gradient calculations needed to adjust W_y

The partial derivative of the Loss Function with respect to W_y is found by a simple one step chain rule: (Note that in this case we do not need to use BPTT.)

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial W_y}$$

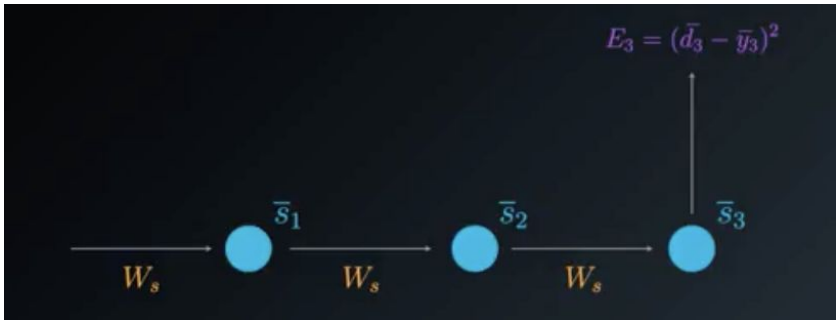
Generally speaking, we can consider multiple timesteps back, and not only 3 as in this example. For an arbitrary timestep N, the gradient calculation needed for adjusting W_y, is:

$$\frac{\partial E_N}{\partial W_y} = \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial W_y}$$

Gradient calculations needed to adjust W_s

We still need to adjust W_s ,the weight matrix connecting one state to the next and W_x the weight matrix connecting the input to the state. We will arbitrarily start with W_s

To understand the **BPTT** process, we can simplify the unfolded model. We will focus on the contributions of W_s to the output, the following way:



Simplified Unfolded model for Adjusting Ws

When calculating the partial derivative of the Loss Function with respect to W_s, we need to consider all of the states contributing to the output. In the case of this example it will be states $\bar{s_3}$ which depends on its predecessor $\bar{s_2}$ which depends on its predecessor $\bar{s_1}$, the first state.

In **BPTT** we will take into account every gradient stemming from each state, **accumulating** all of these contributions.

- At timestep t=3, the contribution to the gradient stemming from $\bar{s_3}$ is the following :

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \frac{\partial \bar{s}_3}{\partial W_s}$$

- At timestep t=3, the contribution to the gradient stemming from $\bar{s}_2$ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of $\bar{s}_2$ $\bar{s}_3$.

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \frac{\partial \bar{s}_2}{\partial W_s}$$

- At timestep t=3, the contribution to the gradient stemming from $\bar{s}_1$ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of $\bar{s}_1$ $\bar{s}_2$ and $\bar{s}_3$

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \frac{\partial \bar{s}_1}{\partial W_s}$$

After considering the contributions from all three states: $\bar{s}_3$ $\bar{s}_2$ $\bar{s}_1$, we will **accumulate** them to find the final gradient calculation.The following equation is the gradient contributing to the adjustment of W_s using **Backpropagation Through Time**:

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial W_s} + \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial W_s} + \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial \bar{s}_1}\frac{\partial \bar{s}_1}{\partial W_s}$$

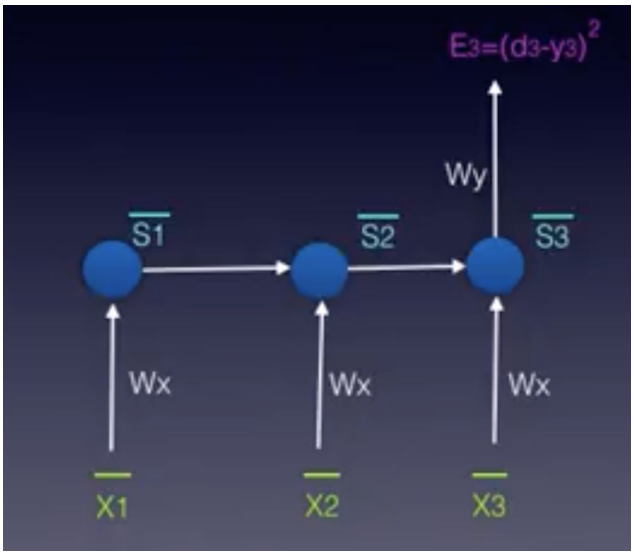
In this example we had 3 time steps to consider, therefore we accumulated three partial derivative calculations. Generally speaking, we can consider multiple timesteps back. If you look closely at the three components of equation , you will notice a pattern. You will find that as we propagate a step back, we have an additional partial derivatives to consider in the chain rule. Mathematically this can be easily written in the following general equation for adjusting W_s using **BPTT**:

$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N}\frac{\partial \bar{y}_N}{\partial \bar{s}_i}\frac{\partial \bar{s}_i}{\partial W_s}$$

Notice that it considers a general setting of N steps back. As mentioned in this lesson, capturing relationships that span more than 8 to 10 steps back is practically impossible due to the vanishing gradient problem. We will talk about a solution to this problem in our LSTM section coming up soon.

Gradient calculations needed to adjust W_x

To further understand the **BPTT** process, we will simplify the unfolded model again. This time the focus will be on the contributions of W_x to the output, the following way:



When calculating the partial derivative of the Loss Function with respect to to W_x we need to consider, again, all of the states contributing to the output. As we saw before, in the case of this example it will be states $\bar{s_3}$ which depend on its predecessor $\bar{s_2}$ which depends on its predecessor $\bar{s_1}$, the first state.

As we mentioned previously, in **BPTT** we will take into account each gradient stemming from each state, **accumulating** all of the contributions.

- At timestep t=3, the contribution to the gradient stemming from $\bar{s_3}$ is the following :

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y_3}} \frac{\partial \bar{y_3}}{\partial \bar{s_3}} \frac{\partial \bar{s_3}}{\partial W_x}$$

- At timestep t=3, the contribution to the gradient stemming from $\bar{s_2}$ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of $\bar{s_2}$ to $\bar{s_3}$.)

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial W_x}$$

- At timestep t=3, the contribution to the gradient stemming from $\bar{s_1}$ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of $\bar{s_1}$ to $\bar{s_2}$ and $\bar{s_3}$.

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial \bar{s_1}}\frac{\partial \bar{s_1}}{\partial W_x}$$

After considering the contributions from all three states, we will **accumulate** them to find the final gradient calculation.

The following equation is the gradient contributing to the adjustment of W_x using **Backpropagation Through Time**:

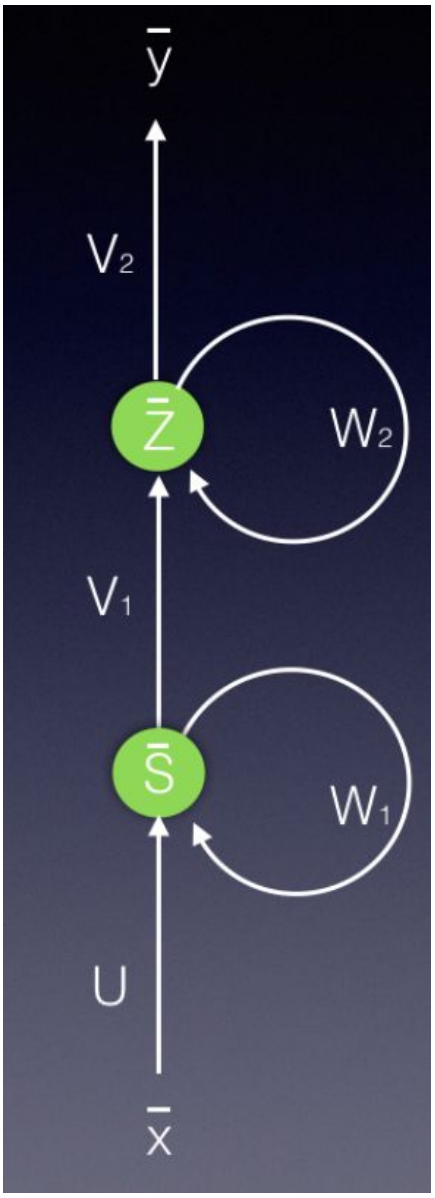
$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial W_x} + \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial W_x} + \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial \bar{s}_1}\frac{\partial \bar{s}_1}{\partial W_x}$$

As mentioned before, in this example we had 3 time steps to consider, therefore we accumulated three partial derivative calculations. Generally speaking, we can consider multiple timesteps back. If you look closely at equations 1, 2 and 3, you will notice a pattern again. You will find that as we propagate a step back, we have an additional partial derivatives to consider in the chain rule. Mathematically this can be easily written in the following general equation for adjusting W_x using **BPTT**:

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N}\frac{\partial \bar{y}_N}{\partial \bar{s}_i}\frac{\partial \bar{s}_i}{\partial W_x}$$

Notice the similarities between the calculations of $\frac{\partial E_3}{\partial W_s}$ and $\frac{\partial E_3}{\partial W_x}$. Hopefully after understanding the calculation process of $\frac{\partial E_3}{\partial W_s}$, understanding that of $\frac{\partial E_3}{\partial W_x}$ was straight forward.

QUIZ QUESTION



Lets look at the same folded model again (displayed above). Assume that the error is noted by the symbol **E**. What is the update rule of weight matrix U at time t+1 (over 2 timesteps) ? Hint: Use the unfolded model for a better visualization.

- Equation A
- Equation B
- Equation C

Equation A

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$

Equation B

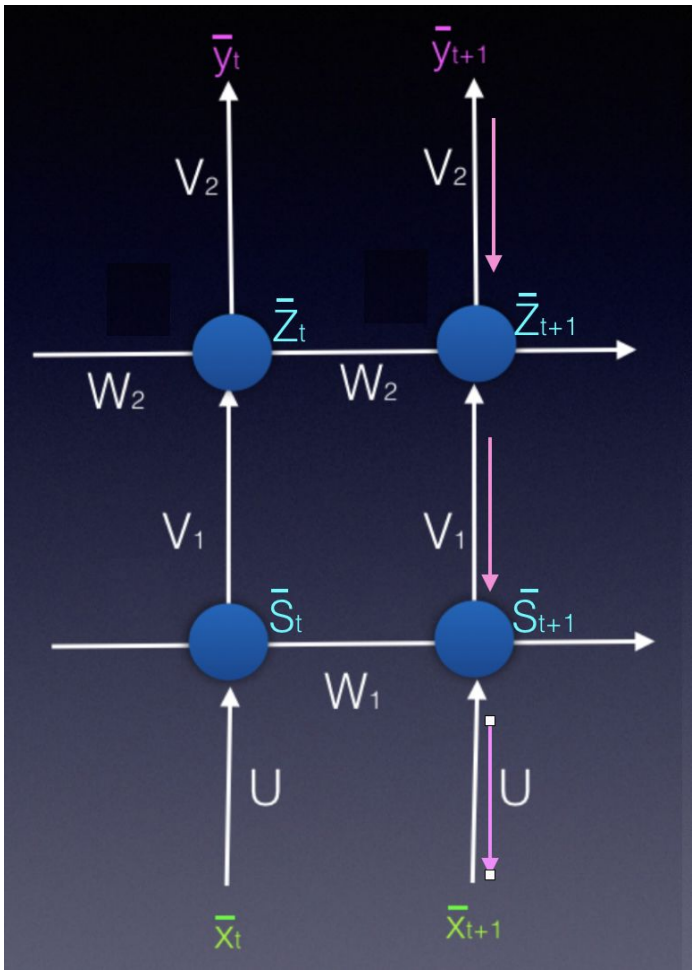
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Equation C

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Solution

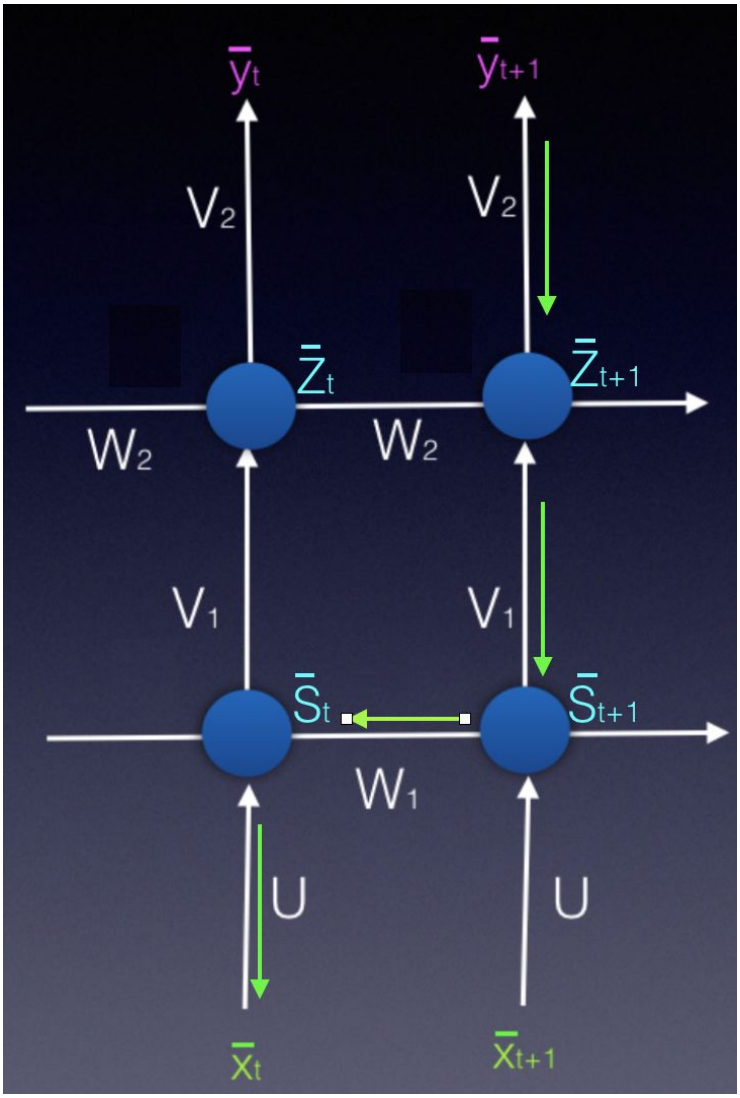
To understand how to update weight matrix U, we will need to unfold the model in time. We will unfold the model over two time steps, as we need to look only time t and time t+1. The following three pictures will help you understand the **three** paths we need to consider. Notice that we have two hidden layers that serve as memory elements, so this case will be different than the one we saw in the video, but the idea is the same. We will use **BPTT** while applying the chain rule.



The first path to consider

The following is the equation we derive using the first path:

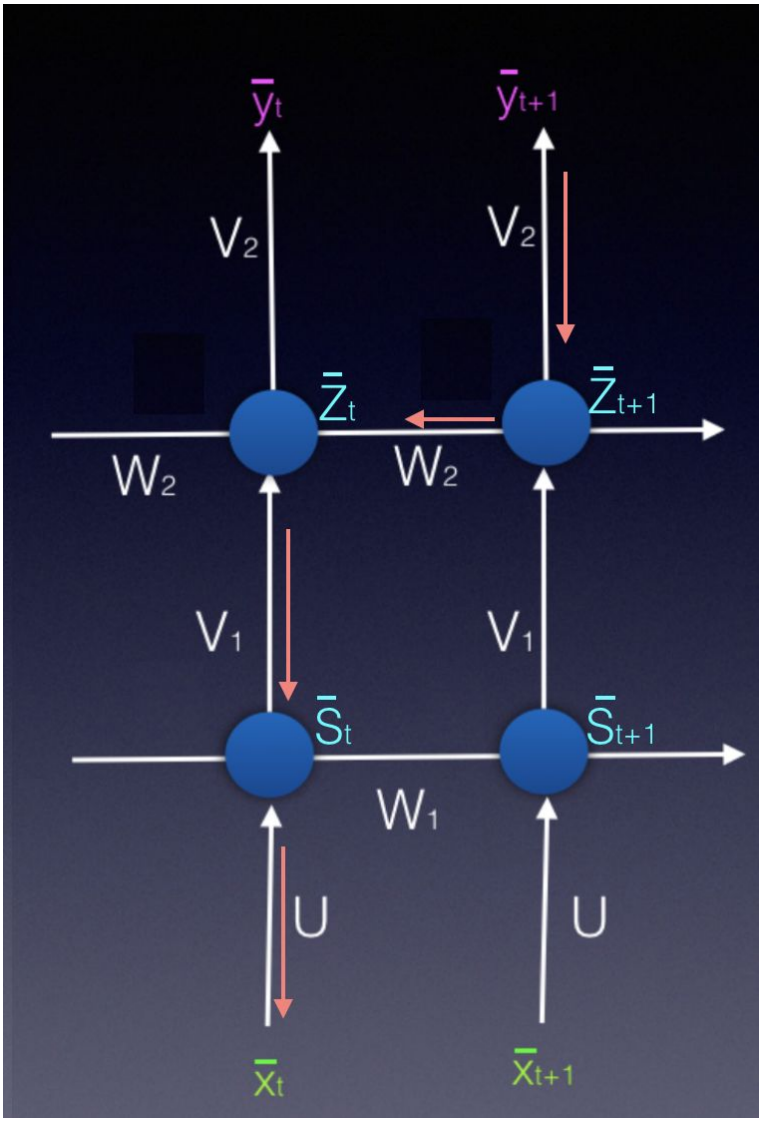
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$



The second path to consider

The following is the equation we derive using the second path:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$



The third path to consider

The following is the equation we derive using the third path:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Finally, after considering all three paths, we can derive the correct equation for the purposes of updating weight matrix U, using BPTT:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$
$$+$$
$$\frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$
$$+$$
$$\frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

The final answer for BPTT Quiz 3

When training RNNs using BPTT, we can choose to use mini-batches, where we update the weights in batches periodically (as opposed to once every inputs sample). We calculate the gradient for each step but do not update the weights right away. Instead, we update the weights once every fixed number of steps. This helps reduce the complexity of the training process and helps remove noise from the weight updates.

The following is the equation used for **Mini-Batch Training Using Gradient Descent**: (where $$\delta_{ij}$$ represents the gradient calculated once every inputs sample and M represents the number of gradients we accumulate in the process).

$$\delta_{ij} = \frac{1}{M} \sum_{k=1}^{M} \delta_{ij_k}$$

If we backpropagate more than ~10 timesteps, the gradient will become too small. This phenomena is known as the **vanishing gradient problem** where the contribution of information decays geometrically over time. Therefore temporal dependencies that span many time steps will effectively be discarded by the network. **Long Short-Term Memory (LSTM)** cells were designed to specifically solve this problem.

Long Short-Term Memory Cells, (LSTM) give a solution to the vanishing gradient problem, by helping us apply networks that have temporal dependencies. They were proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber

In RNNs we can also have the opposite problem, called the **exploding gradient** problem, in which the value of the gradient grows uncontrollably. A simple solution for the exploding gradient problem is **Gradient Clipping**.

More information about Gradient Clipping can be found here.