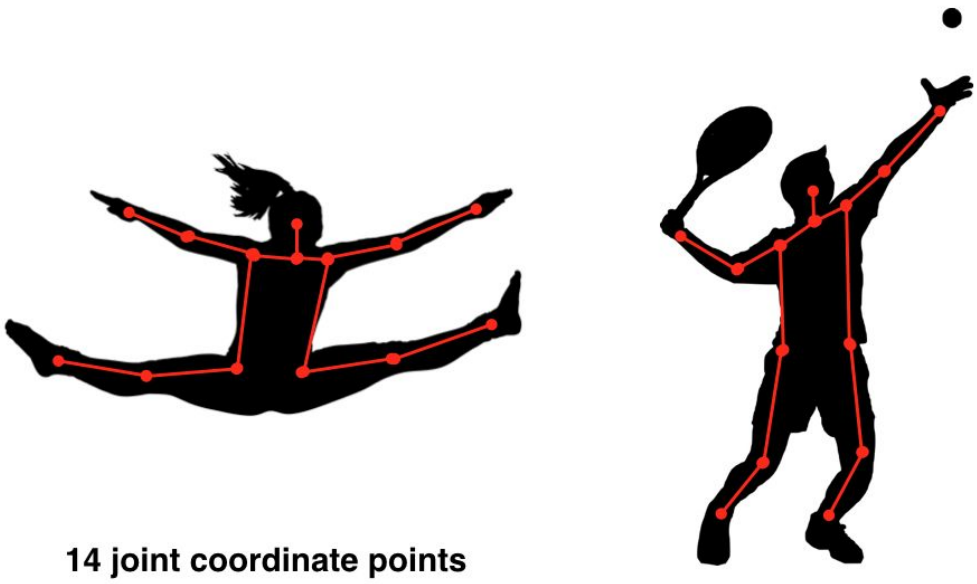
Advanced Computer Vision & Deep Learning

Distracted Driver Detection

As was mentioned in the video, detecting distracted drivers is a classification *and* localization challenge. As part of this task, some approaches localize cell phones in a image; if a cell phone was detected as close to a driver's face, then it assumes that the driver is on the phone and distracted, but if a phone is located on a car seat or far away from the driver, then the driver is more likely to be focused on the road. Check out an example of such detection code in Keras.

Beyond Bounding Boxes

To predict bounding boxes, we train a model to take an image as input and output coordinate values: (x, y, w, h). This kind of model can be extended to *any* problem that has coordinate values as outputs! One such example is **human pose estimation**.



Huan pose estimation points.

In the above example, we see that the pose of a human body can be estimated by tracking 14 points along the joints of a body.

Weighted Loss Functions

You may be wondering: how can we train a network with two different outputs (a class and a bounding box) and different losses for those outputs?

We know that, in this case, we use categorical cross entropy to calculate the loss for our predicted and true classes, and we use a regression loss (something like smooth L1 loss) to compare predicted and true bounding boxes. But, we have to train our whole network using one loss, so how can we combine these?

There are a couple of ways to train on multiple loss functions, and in practice, we often use a weighted sum of classification *and* regression losses (ex. `0.5*cross_entropy_loss + 0.5*L1_loss`); the result is a single error value with which we can do backpropagation. This does introduce a hyperparameter: the loss weights. We want to weight each loss so that these losses are balanced and combined effectively, and in research we see that another regularization term is often introduced to help decide on the weight values that best combine these losses.

MSE loss is calculated by 1. summing the squared differences between points: Ex: `(2.5-2)^2 + (3-5)^2 = 4.25` and then 2. averaging that sum, so for two values this is `4.25/2 = 2.125`.

Smooth L1 Loss : Ex: For a difference less than 1, in the case between 2.5 and 2, we use a squared function 0.5*(0.5)^2 = 0.125 and for a difference greater than one as is the case between 3 and 5, we use a linear function 2-0.5 = 1.5. We then sum the losses between these two points and take their average: (1/2)*(0.125+1.5) = 0.8125.

Consider the above image, how do you think you would select the best proposed regions; what criteria do good regions have?

The regions we want to analyze are those with complete objects in them. We want to get rid of regions that contain image background or only a portion of an object. So, two common approaches are suggested: 1. identify similar regions using feature extraction or a clustering algorithm like k-means, as you've already seen; these methods should identify any areas of interest. 2. Add another layer to our model that performs a binary classification on these regions and labels them: object or not-object; this gives us the ability to discard any non-object regions!

R-CNN Outputs

The R-CNN is the least sophisticated region-based architecture, but it is the basis for understanding how multiple object recognition algorithms work! It outputs a class score and bounding box coordinates for every input RoI.

An R-CNN feeds an image into a CNN with regions of interest (RoI's) already identified. Since these RoI's are of varying sizes, they often need to be **warped to be a standard size**, since CNN's typically expect a consistent, square image size as input. After RoI's are warped, the R-CNN architecture, processes these regions one by one and, for each image, produces 1. a class label and 2. a bounding box (that may act as a slight correction to the input region).

1. R-CNN produces bounding box coordinates to reduce localization errors; so a region comes in, but it may not perfectly surround a given object and the output coordinates `(x,y,w,h)` aim to *perfectly* localize an object in a given region.
2. R-CNN, unlike other models, does not explicitly produce a confidence score that indicates whether an object is in a region, instead it cleverly produces a set of class scores for which one class is "background". This ends up serving a similar purpose, for example, if the class score for a region is `Pbackground = 0.10`, it likely contains an object, but if it's `Pbackground = 0.90`, then the region probably doesn't contain an object.

RoI Pooling

To warp regions of interest into a consistent size for further analysis, some networks use RoI pooling. RoI pooling is an additional layer in our network that takes in a rectangular region of any size, performs a maxpooling operation on that region in pieces such that the output is a fixed shape. Below is an example of a region with some pixel values being broken up into pieces which pooling will be applied to; a section with the values:

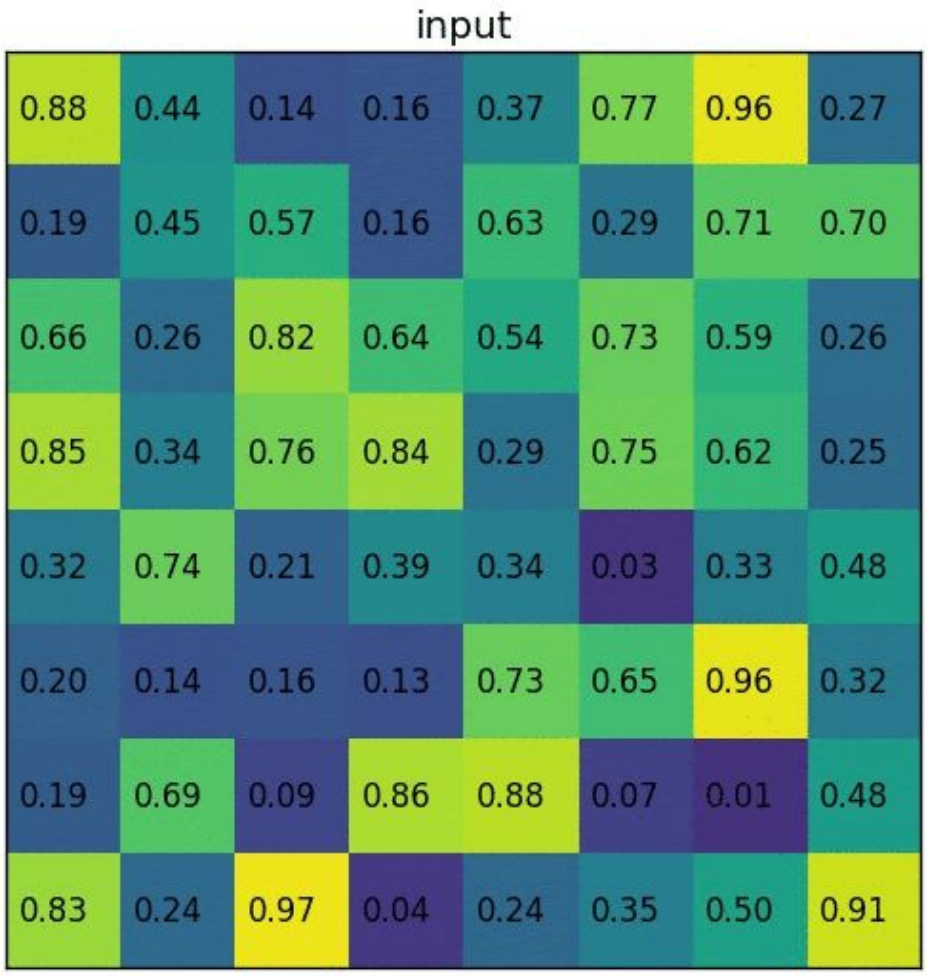
```
[[0.85, 0.34, 0.76],

 [0.32, 0.74, 0.21]]
```

Will become a single max value after pooling: `0.85`. After applying this to an image in these pieces, you can see how any rectangular region can be forced into a smaller, square representation.

An example of pooling sections, credit to

this informational resource on RoI pooling [by Tomasz Grel].

You can see the complete process from input image to region to reduced, maxpooled region, below.



Credit to

this informational resource on RoI pooling.

Speed

Fast R-CNN is about 10 times as fast to train as an R-CNN because it only creates convolutional layers once for a given image and then performs further analysis on the layer. Fast R-CNN also takes a shorter time to test on a new image! It's test time is dominated by the time it takes to create region proposals.

Why is speed important?

Multi-object identification consists of locating and classifying different objects in an image. Some models trade off accuracy for speed. What kinds of applications require speed in object recognition? Check all that apply.

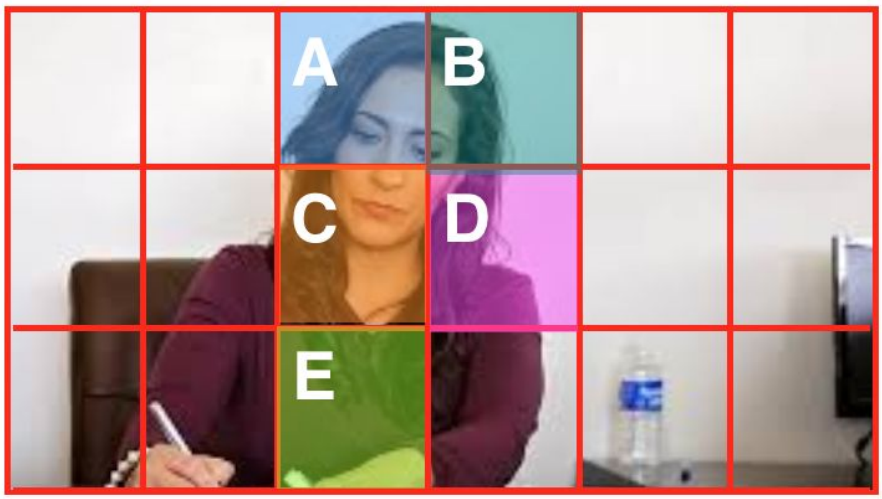
- Pedestrian detection for an autonomous vehicles.
- Tracking people's faces so that a camera can focus on them.

Faster R-CNN Implementation

If you'd like to look at an implementation of this network in code, you can find a peer-reviewed version, at this Github repo.

Seeing a Test Image

What do you think will happen once a network, like the CNN we've been talking about, sees a new, test image with an object in it? First, our network has to break this new, test image into a grid.



An image of a woman working broken into a grid of cells; 5 of which are labelled: A, B, C, D, and E.

QUIZ QUESTION

For the above image, which cell do you think will predict the bounding box for the person in that image? (You may select multiple cells if you think more than one will detect the person and produce a bounding box.)

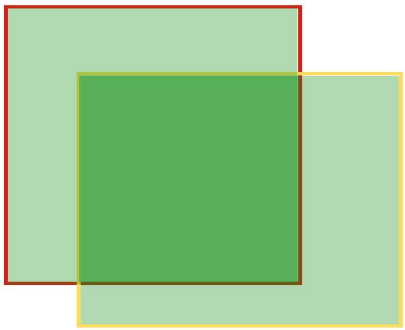
Ans : A

The truth is, one of the problems with this method is that the CNN will often output **multiple grid cells** that try to detect the *same* object.

So, any of these cells might try to detect the person and we might end up with multiple bounding boxes for the same person. Next, let's talk about how to prevent this from happening so that there is only one true bounding box for an object.

Intersection over Union (IoU)

We know the IoU is given by the area of: intersection/union. The next couple questions will test your intuition about what values IoU can take as it compares two bounding boxes.



Two bounding boxes. Intersection in dark green and union in light green.

QUESTION 1 OF 2

Imagine you're comparing a ground truth box to a predicted box. If you want your predicted box to be as close to this ground truth box as possible, what would you want the IoU to be?

Ans: 1(One)

QUESTION 2 OF 2

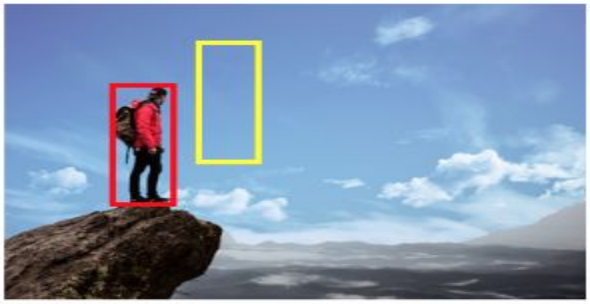
What is the lowest value IoU can have? Ans: 0(Zero)

IoU Values

The IoU between two bounding boxes will always have a value between 0 and 1 because, as two boxes drift apart, their intersection approaches 0, but if two bounding boxes overlap *perfectly* their IoU will be 1. So, the higher the IOU the more overlap there is between the two bounding boxes!

In the next video we will see how Non-Maximal suppression uses the IOU to only choose the best bounding box.

$$IOU = \frac{2000}{2000} = 1$$



$$IOU = \frac{0}{4000} = 0$$

Examples of maximum and minimum IoU values between two boxes.

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

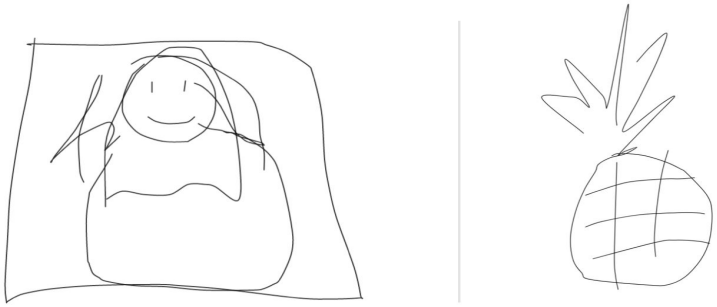
At the end of this lesson, you will be tasked with creating an automatic image captioning model that takes in an image as input and outputs a *sequence* of words, describing that image. Image captions are used to create accessible content and in a number of other cases where one may want to read about 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.

Recurrent Neural Networks (RNNs).

The neural network architectures you've seen so far were trained using the current inputs only. We did not consider previous inputs when generating the current output. In other words, our systems did not have any **memory** elements. RNNs address this very basic and important issue by using **memory** (i.e. past inputs to the network) when producing the current output.

How did the theory behind RNN evolve? Where were we a few years ago and where are we now?

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**.

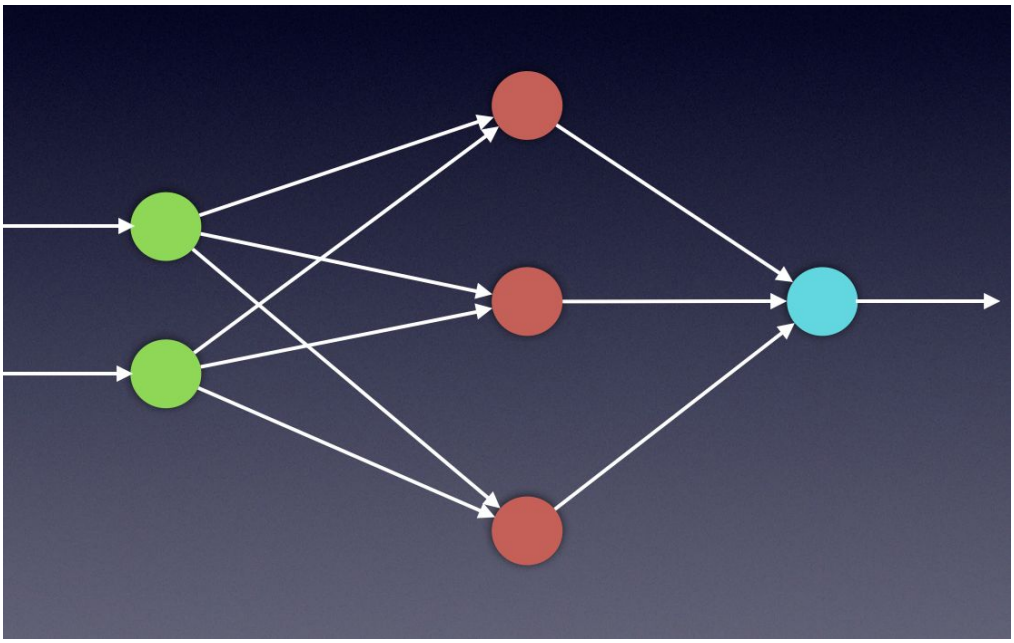
Applications

The world's leading tech companies are all using RNNs, particularly LSTMs, in their applications. Let's take a look at a few.

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 Neural Network - A Reminder



The mathematical calculations needed for training RNN systems are fascinating. To deeply understand the process, we first need to feel confident with the vanilla FFNN system. We need to thoroughly understand the feedforward process, as well as the backpropagation process used in the training phases of such systems. The next few videos will cover these topics, which you are already familiar with. We will address the feedforward process as well as backpropagation, using specific examples. These examples will serve as extra content to help further understand RNNs later in this lesson.

The following couple of videos will give you a brief overview of the **Feedforward Neural Network (FFNN)**.

As mentioned before, when working with neural networks we have 2 primary phases:

Training

and

Evaluation.

During the **training** phase, we take the data set (also called the *training set*), which includes many pairs of inputs and their corresponding targets (outputs). Our goal is to find a set of weights that would best map the inputs to the desired outputs. In the **evaluation** phase, we use the network that was created in the training phase, apply our new inputs and expect to obtain the desired outputs.

The training phase will include two steps:

Feedforward

and

Backpropagation

We will repeat these steps as many times as we need until we decide that our system has reached the best set of weights, giving us the best possible outputs.

The next two videos will focus on the feedforward process.

You will notice that in these videos I use subscripts as well as superscript as a numeric notation for the weight matrix.

As mentioned before, when working with neural networks we have 2 primary phases:

Training

and

Evaluation.

*During the **training** phase, we take the data set (also called the training set), which includes many pairs of inputs and their corresponding targets (outputs). Our goal is to find a set of weights that would best map the inputs to the desired outputs. In the **evaluation** phase, we use the network that was created in the training phase, apply our new inputs and expect to obtain the desired outputs.*

The training phase will include two steps:

Feedforward

and

Backpropagation

We will repeat these steps as many times as we need until we decide that our system has reached the best set of weights, giving us the best possible outputs.

The next two videos will focus on the feedforward process.

You will notice that in these videos I use subscripts as well as superscript as a numeric notation for the weight matrix.

For example:

- W_k is weight matrix k
- W_{ij}^k is the ij element of weight matrix k

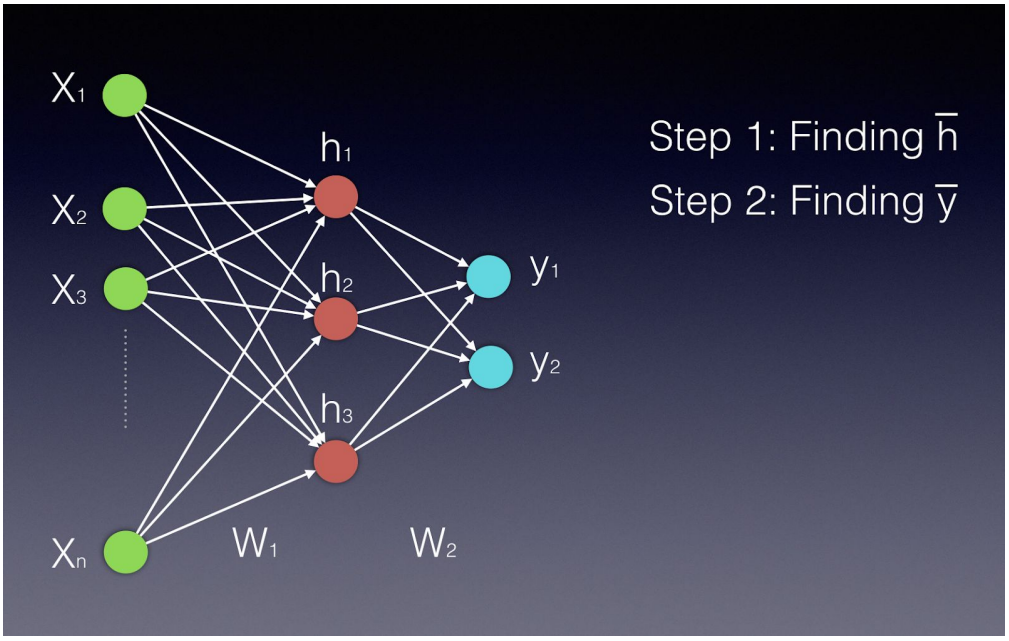
Feedforward

In this section we will look closely at the math behind the feedforward process. With the use of basic Linear Algebra tools, these calculations are pretty simple!

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*

Assuming that we have a single hidden layer, we will need two steps in our calculations. The first will be calculating the value of the hidden states and the latter will be calculating the value of the outputs.



Notice that both the hidden layer and the output layer are displayed as vectors, as they are both represented by more than a single neuron.

As you saw in the video above, 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:

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

Equation 1

After finding h' , 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)

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)

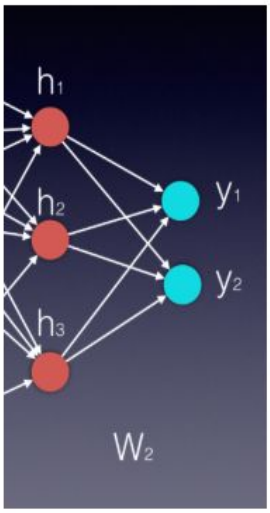
$$\begin{aligned} h_1 &= \Phi(x_1W_{11}+x_2W_{21}+....+x_nW_{n1}) \\ h_2 &= \Phi(x_1W_{12}+x_2W_{22}+....+x_nW_{n2}) \\ h_3 &= \Phi(x_1W_{13}+x_2W_{23}+....+x_nW_{n3}) \end{aligned}$$

Equation 2

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

https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions

As you've seen in the video above, 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).

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).

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

Equation 3

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).

In the above calculations we used a variation of the MSE.

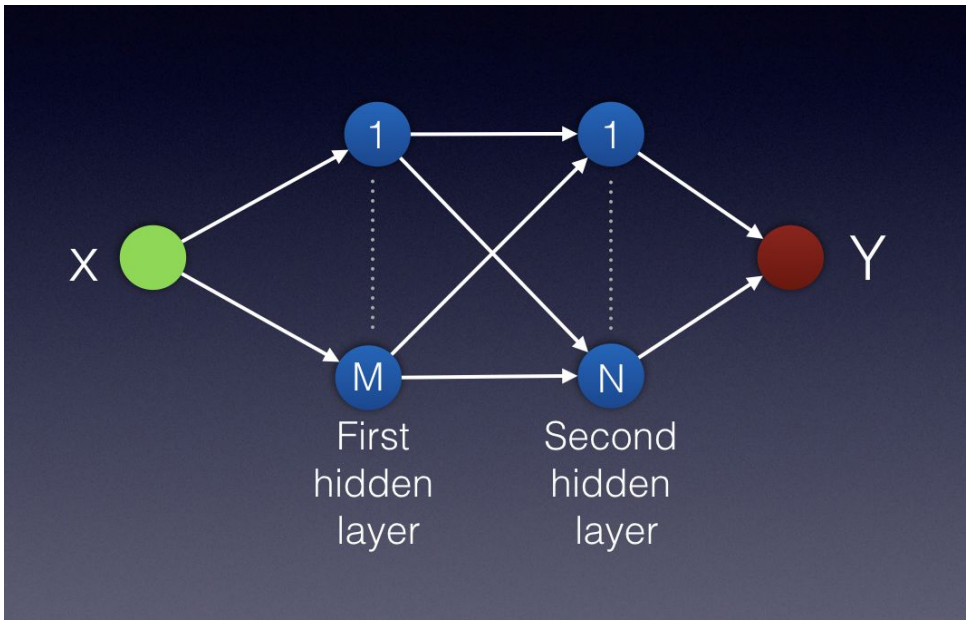
The next few videos will focus on the backpropagation process, or what we also call stochastic gradient decent with the use of the chain rule.

The following picture is of a feedforward network with

- A single input x
- Two hidden layers
- A singe output

The first hidden layer has M neurons.

The second hidden layer has N neurons.



QUIZ QUESTION

What is the total number of multiplication operations needed for a single feedforward pass?

Ans: **M+N+MN**

Solution

To calculate the number of multiplications needed for a single feedforward pass, we can break down the network to three steps:

- Step 1: From the single input to the first hidden layer
- Step 2: From the first hidden layer to the second hidden layer
- Step 2: From the second hidden layer to the single output

Step 1

The single input is multiplied by a vector with M values. Each value in the vector will represent a weight connecting the input to the first hidden layer. Therefore, we will have **M** multiplication operations.

Step 2

Each value in the first hidden layer (M in total) will be multiplied by a vector with N values. Each value in the vector will represent a weight connecting the neurons in the first hidden layer to the neurons in the second hidden layer. Therefore, we will have here M times N calculations, or simply **MN** multiplication operations.

Step 3

Each value in the second hidden layer (N in total) will be multiplied once, by the weight element connecting it to the single output. Therefore, we will have **N** multiplication operations.

In total, we will add the number of operations we calculated in each step: **M+MN+N** .

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 in future quizzes of the lesson, 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. The following video will help you understand the mathematical process we use for computing these adjustments.

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: $\Delta W^k{}_{ij}$.

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:

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

Equation 4

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

$\Delta W_{ij}^k = \alpha(-\frac{\partial E}{\partial W})$, where α is a small positive number called the **Learning Rate.**

Equation 5

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

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

Equation 6

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.

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

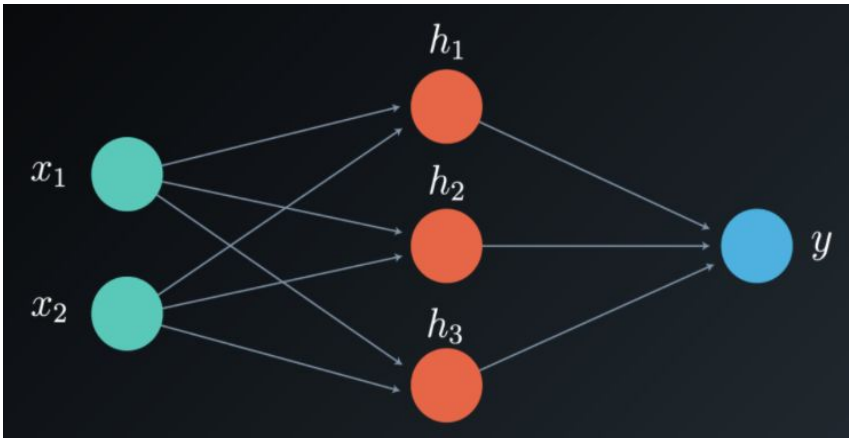
Equation 7

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

- resource 1
- resource 2

Backpropagation- Example (part a)

We will now continue with an example focusing on the backpropagation process, and consider a network having two inputs [x_1, x_2]three neurons in a single hidden layer [h_1, h_2, h_3] and a single output y.



The weight matrices to update are W^1 from the input to the hidden layer, and W^2 from the hidden layer to the output. Notice that in our case W^2 is a vector, not a matrix, as we only have one output.

The chain of thought in the weight updating process is as follows:

To update the weights, we need the network error. To find the network error, we need the network output, and to find the network output we need the value of the hidden layer, vector $\bar{h}$.

$$\bar{h} = [h_1, h_2, h_3]$$

Equation 8

Each element of vector $\bar{h}$ is calculated by a simple linear combination of the input vector with its corresponding weight matrix W^1, followed by an activation function.

$$h_1 = \Phi(\sum_i^2 (x_i W_{i1}))$$
$$h_2 = \Phi(\sum_i^2 (x_i W_{i2}))$$
$$h_3 = \Phi(\sum_i^2 (x_i W_{i3}))$$

Equation 9

We now need to find the network's output, y. y is calculated in a similar way by using a linear combination of the vector $\bar{h}$ with its corresponding elements of the weight vector W^2.

$$y = \sum_i^3 (h_i W_i)$$

After computing the output, we can finally find the network error.

As a reminder, the two Error functions most commonly used are the Mean Squared Error (MSE) (usually used in regression problems) and the cross entropy (often used in classification problems).

In this example, we use a variation of the MSE:

$E = \frac{(d-y)^2}{2}$,

where d is the desired output and y is the calculated one. Notice that **y** and **d** are not vectors in this case, as we have a single output.

The error is their squared difference, $E = (d-y)^2$, and is also called the network's **Loss Function**. We are dividing the error term by 2 to simplify notation, as will become clear soon.

The aim of the backpropagation process is to minimize the error, which in our case is the Loss Function. To do that we need to calculate its partial derivative with respect to all of the weights.

Since we just found the output y, we can now minimize the error by finding the updated values ΔW_{ij}^k. The superscript k indicates that we need to update each and every layer k.

As we noted before, the weight update value ΔW_{ij}^k is calculated with the use of the gradient the following way:

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

Therefore:

$$\Delta W_{ij}^k = \alpha(-\frac{\partial E}{\partial W}) = -\frac{\alpha}{2}\frac{\partial(d-y)^2}{\partial W_{ij}} = -2\frac{\alpha}{2}(d-y)\frac{\partial(d-y)}{\partial W_{ij}}$$

which can be simplified as:

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

Equation 11

(Notice that d is a constant value, so it's partial derivative is simply a zero)

This partial derivative of the output with respect to each weight, defines the gradient and is often denoted by the Greek letter δ.

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}}$$

Equation 12

We will find all the elements of the gradient using the chain rule.

If you are feeling confident with the **chain rule** and understand how to apply it, skip the next video and continue with our example. Otherwise, give Luis a few minutes of your time as he takes you through the process!

In our example we only have one hidden layer, so our backpropagation process will consist of two steps:

Step 1: Calculating the gradient with respect to the weight vector W^2 (from the output to the hidden layer).
Step 2: Calculating the gradient with respect to the weight matrix W^1 (from the hidden layer to the input).

Step 1 (Note that the weight vector referenced here will be W^2. All indices referring to W^2 have been omitted from the calculations to keep the notation simple).

$$y = \sum_i^3 (h_i W_i)$$
$$\Rightarrow \delta_i = \frac{\partial y}{\partial W_i} = \frac{\partial \sum_i^3 (h_i W_i)}{\partial W_i} = h_i$$

Equation 13

As you may recall:

$$\Delta W_{ij} = \alpha(d-y)\frac{\partial y}{\partial W_{ij}}$$

In this specific step, since the output is of only a single value, we can rewrite the equation the following way (in which we have a weights *vector*):

$$\Delta W_i = \alpha(d-y)\frac{\partial y}{\partial W_i}$$

Since we already calculated the gradient, we now know that the incremental value we need for step one

is:

$$\Delta W_i = \alpha(d-y)h_i$$

Equation 14

Having calculated the incremental value, we can update vector W^2 the following way:

$$W^2_{new} = W^2_{previous} + \Delta W^2_i$$

$$W^2_{new} = W^2_{previous} + \alpha(d-y)h_i$$

Equation 15

Step 2 (In this step, we will need to use both weight matrices. Therefore we will not be omitting the weight indices.)

In our second step we will update the weights of matrix W^1 by calculating the partial derivative of y with respect to the weight matrix W^1.

The chain rule will be used the following way:

obtain the partial derivative of y with respect to $\bar{h}$, and multiply it by the partial derivative of $\bar{h}$ with respect to the corresponding elements in W^1. Instead of referring to vector $\bar{h}$, we can observe each element and present the equation the following way:

$$\delta_{ij} = \frac{\partial y}{\partial W^1_{ij}} = \sum_{p=1}^{N}(\frac{\partial y}{\partial h_p}\frac{\partial h_p}{\partial W^1_{ij}})$$

Equation 16

In this example we have only 3 neurons the the single hidden layer, therefore this will be a linear combination of three elements:

$$\delta_{ij} = \frac{\partial y}{\partial W^1_{ij}} = \sum_{p=1}^{3}(\frac{\partial y}{\partial h_p}\frac{\partial h_p}{\partial W^1_{ij}})$$

Equation 17

We will calculate each derivative separately. $\frac{\partial y}{\partial h_j}$ will be calculated first, followed by $\frac{\partial h_j}{\partial W^1_{ij}}$.

$$\frac{\partial y}{\partial h_j} = \frac{\partial \sum_{i=1}^{3}(h_i W^2_i)}{\partial h_j} = W^2_j$$

Equation 18

Notice that most of the derivatives were zero, leaving us with the simple solution of $\frac{\partial y}{\partial h_j} = W^2_j$

To calculate $\frac{\partial h_j}{\partial W^1_{ij}}$ we need to remember first that

$$h_j = \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))$$

Equation 19

Therefore:

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1}$$

Equation 20

Since the function h_j is an activation function (Φ) of a linear combination, its partial derivative will be calculated the following way:

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1} = \frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))} \frac{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1}$$

Equation 21

Given that there are various activation functions, we will leave the partial derivative of Φ using a general notation. Each neuron j will have its own value for Φ and Φ', according to the activation function we choose to use.

$$\frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))} = \Phi_j'$$

Equation 22

The second calculation of equation 21 can be calculated the following way:

(Notice how simple the result is, as most of the components of this partial derivative are zero).

$$\frac{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1} = x_i$$

Equation 23

After understanding how to treat each multiplication of equation 21 separately, we can now summarize it the following way:

$$\frac{\partial h_j}{\partial W_{ij}^1} = \Phi_j' x_i$$

Equation 24

We are ready to finalize **step 2**, in which we update the weights of matrix W^1 by calculating the gradient shown in equation 17. From the above calculations, we can conclude that:

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{3} \frac{\partial y}{\partial h_p} \frac{\partial h_p}{\partial W_{ij}^1} = W_j^2 \Phi_j' x_i$$

Equation 25

Since $\Delta W_{ij}^1 = \alpha(d-y)\frac{\partial y}{\partial W_{ij}^1}$, when finalizing step 2, we have:

$$\Delta W_{ij}^1 = \alpha(d-y) W_j^2 \Phi_j' x_i$$

Equation 26

Having calculated the incremental value, we can update vector W^1 the following way:

$$W_{new}^1 = W_{previous}^1 + \Delta W_{ij}^1$$

$$W_{new}^1 = W_{previous}^1 + \alpha(d-y) W_j^2 \Phi_j' x_i$$

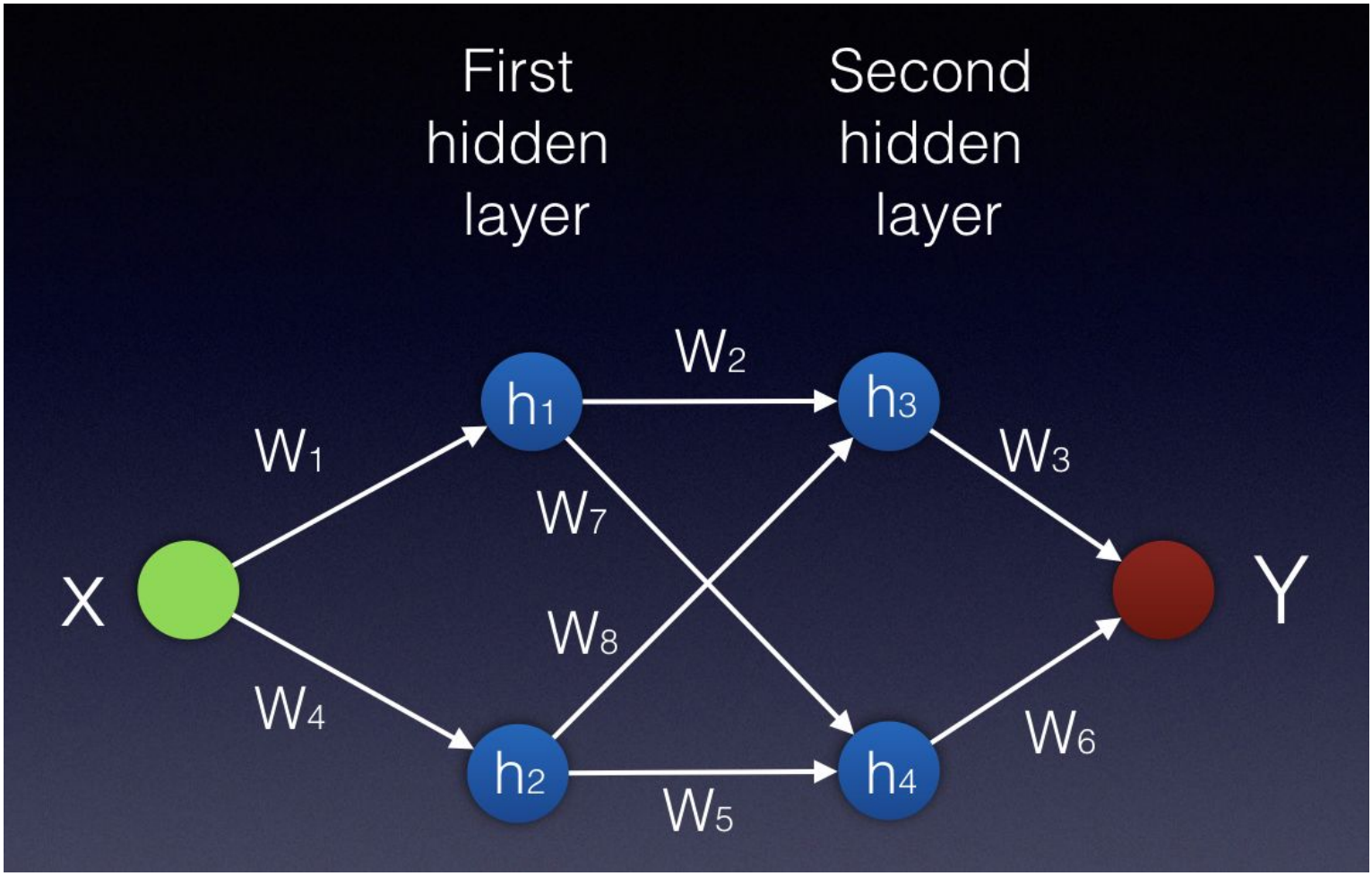
Equation 27

After updating the weight matrices we begin once again with the Feedforward pass, starting the process of updating the weights all over again.

This video touches on the subject of Mini Batch Training. We will further explain things in our **Hyperparameters** lesson coming up.

The following picture is of a feedforward network with

- A single input x
- Two hidden layers with two neurons in each layer
- A single output



QUIZ QUESTION

What is the update rule of weight matrix W1?

(In other words, what is the partial derivative of y with respect to W1?)

Hint: Use the chain rule

Ans : Equation A

Equation A $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

Equation B $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_2}\frac{\partial h_2}{\partial W_1}$$

Equation C $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

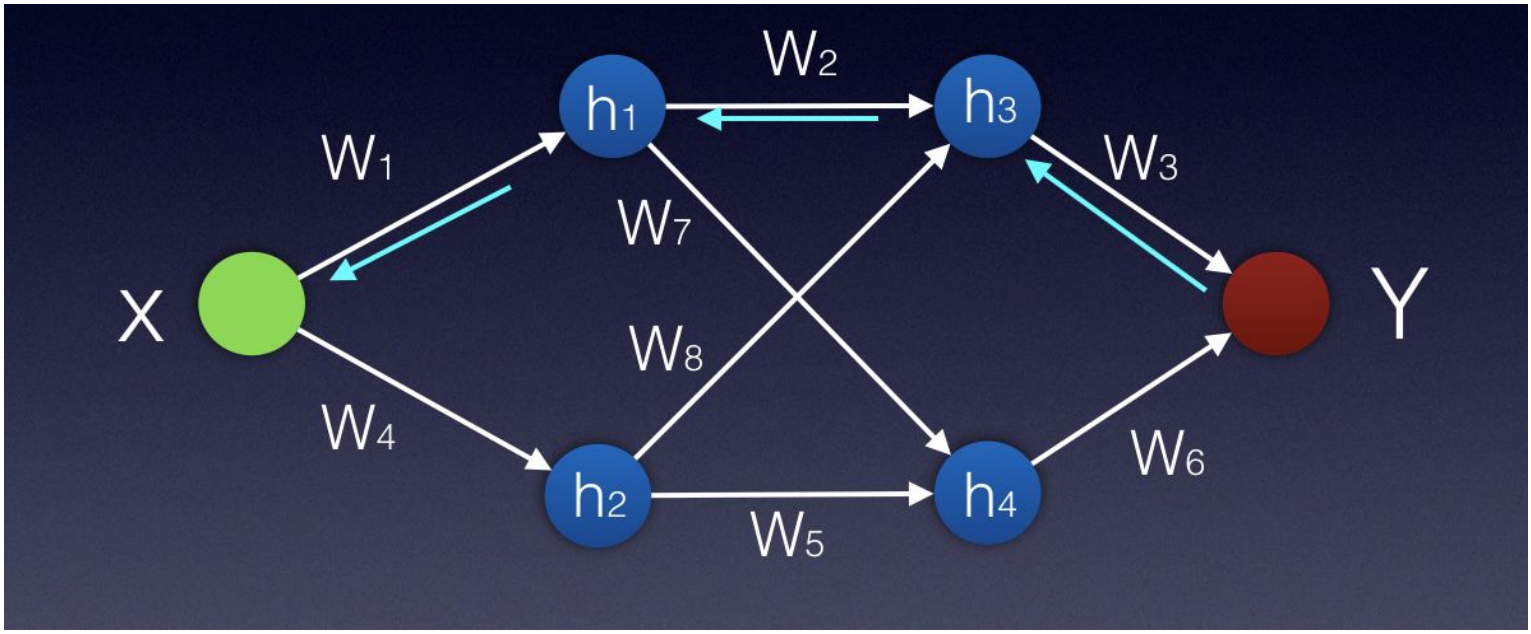
Equation D $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

Solution

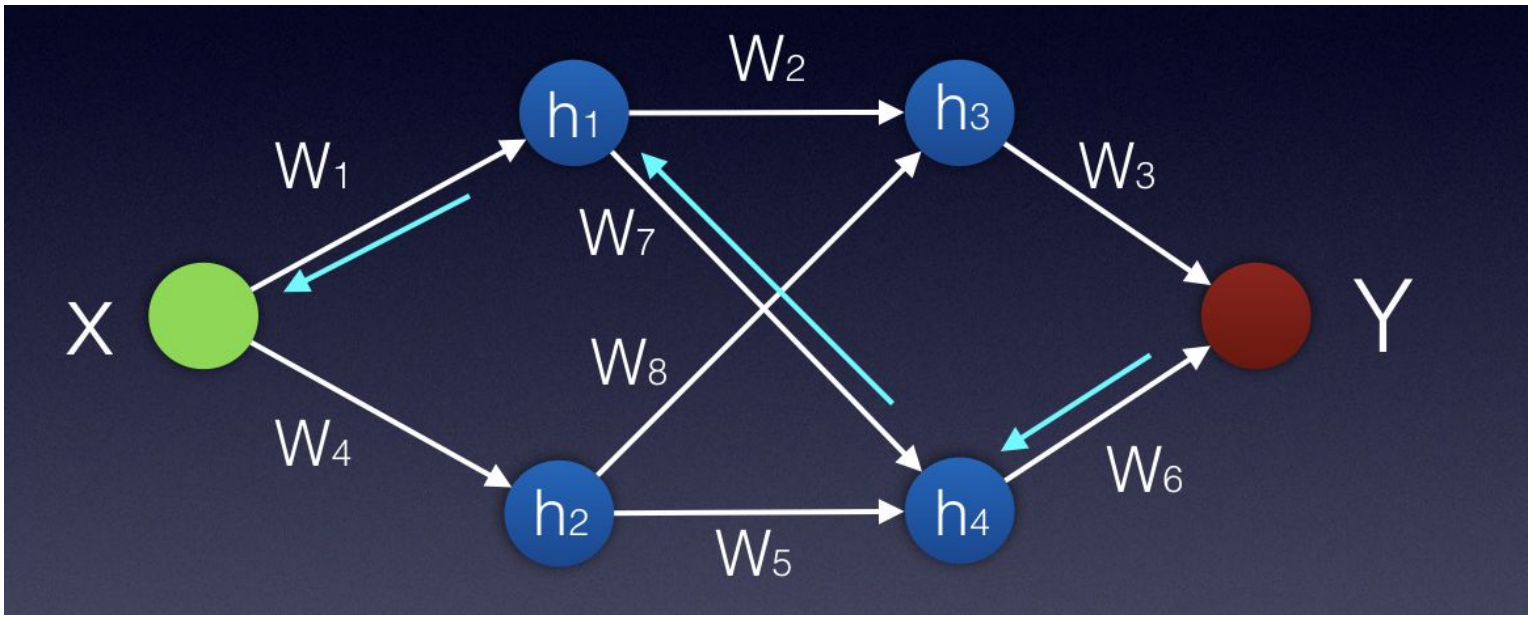
There are two separate paths which W_1 contributes to the output in:

- Path A
- Path B

(both displayed in the pictures below)



Path A



Path B

The mathematical derivations considering path A (while applying the chain rule) are:

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

The mathematical derivations considering path B (while applying the chain rule) are:

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

To finalize our calculations we need to consider all of the paths contributing to the calculation of y. In this case we have the two paths mentioned. Therefore, the final calculation will be the addition of the derivatives calculated in each path.

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

We are finally ready to talk about Recurrent Neural Networks (or RNNs), where we will be opening the doors to new content!

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.

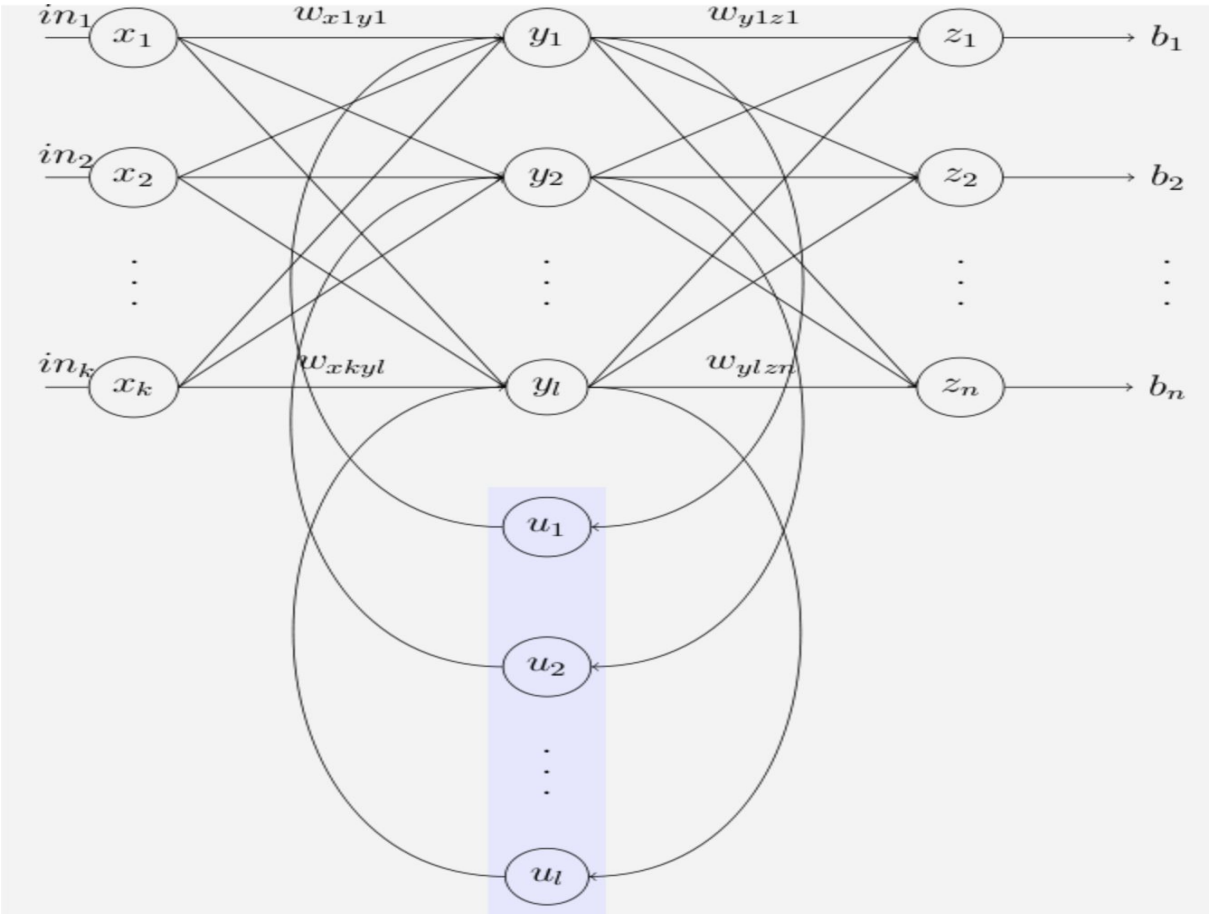
There are two main differences between FFNNs and RNNs. The Recurrent Neural Network uses:

- **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:



Elman Network, source: Wikipedia

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.

Let's continue now to the next video with more information about RNNs.

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:

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

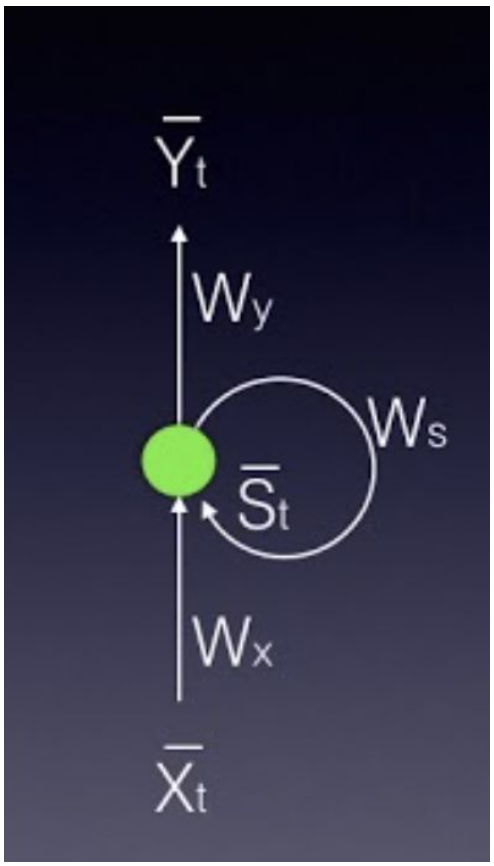
Equation 28

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:

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

Equation 29

This is the RNN **folded model**:



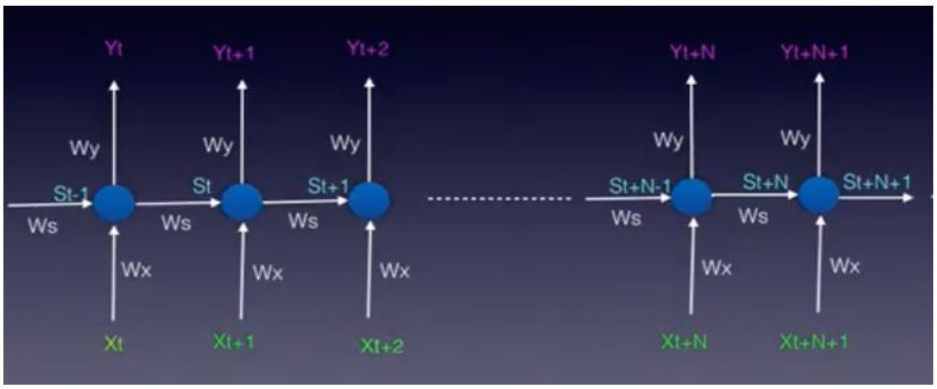
In this picture, $\bar{x}$ represents the input vector, $\bar{y}$ represents the output vector and $\bar{s}$ denotes the state vector.

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:

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

Equation 30

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

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

Equation 31

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

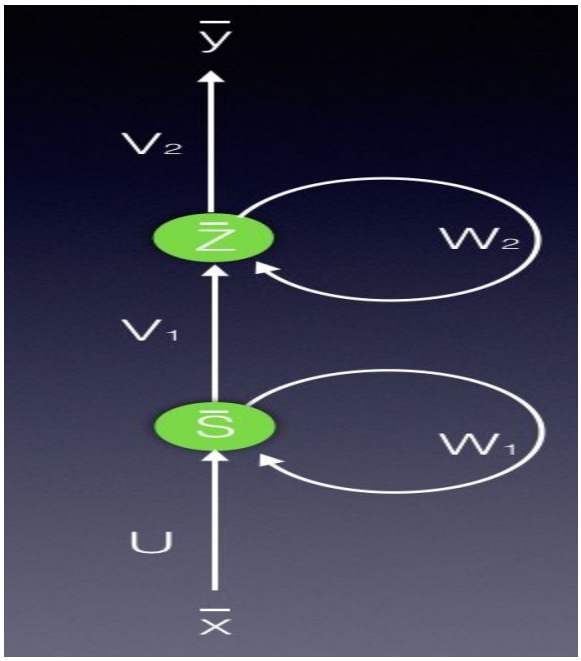
$$\bar{y}_t = \bar{s}_t W_y$$

or

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

Equation 32

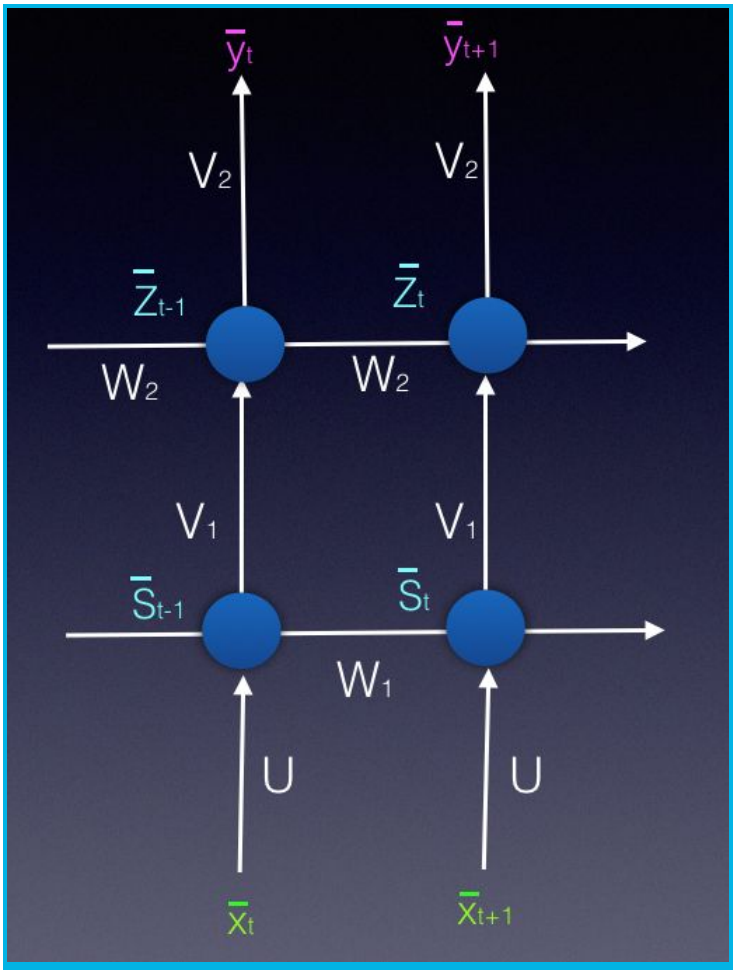
The next video will focus on the **unfolded model** as we try to further understand it.



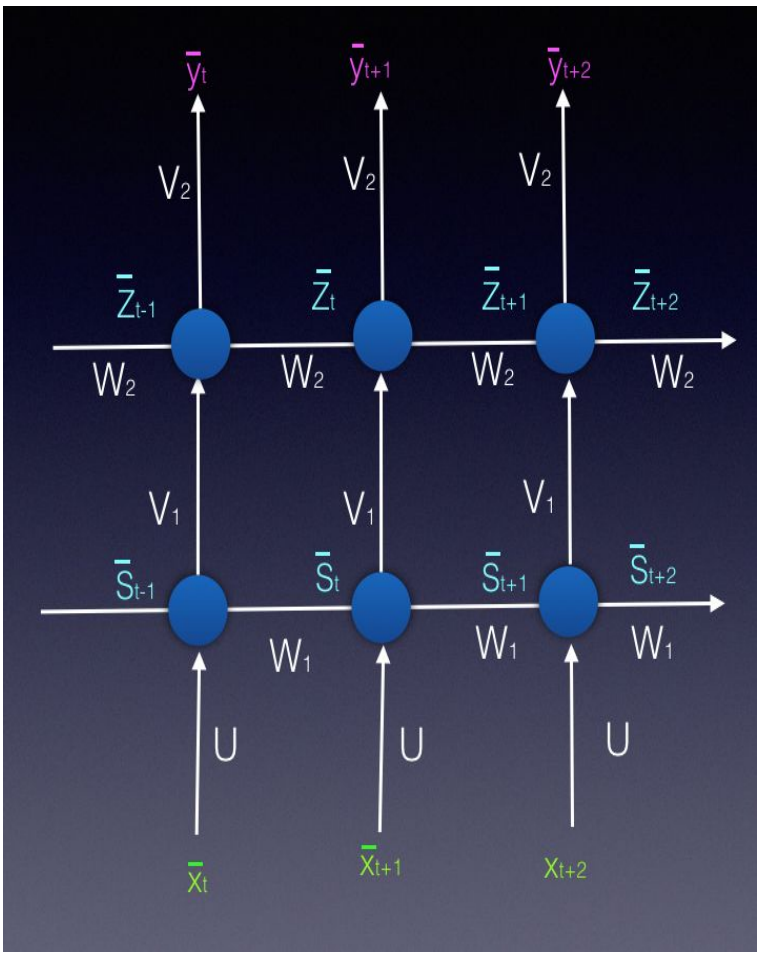
A folded model of a RNN

QUIZ QUESTION

Look at the above picture of a folded model of a RNN. Which of the pictures below represents the unfolded model of the same network? Ans : Both A and B are correct



Picture A



Picture B

In this example we will illustrate how RNNs can be helpful in detecting sequences. When detecting a sequence, the system has to remember what the previous inputs were, so it makes sense to use a recurrent network.

If you are unfamiliar with the term sequence detection, the idea is to see if a specific pattern of inputs has entered the system. In our example the pattern will be the word U,D,A,C,I,T,Y.

We are now ready to understand how to train the RNN.

When we train RNNs we also use backpropagation, but with a conceptual change. The process is similar to that in the FFNN, with the exception that we need to consider previous time steps, as the system has memory. This process is called **Backpropagation Through Time (BPTT)** and will be the topic of the next three videos.

- As always, don't forget to take notes.

In the following videos we will use the Loss Function for our error. The Loss Function is the square of the difference between the desired and the calculated outputs. There are variations to the Loss Function, for example, factoring it with a scalar. In the backpropagation example we used a factoring scalar of 1/2 for calculation convenience.

As described previously, the two most commonly used are the Mean Squared Error (MSE) (usually used in regression problems) and the cross entropy (usually used in classification problems).

Here, we are using a variation of the MSE.

Before diving into Backpropagation Through Time we need a few reminders.

The state vector $\bar{s}_t$ is calculated the following way:

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

Equation 33

The output vector $\bar{y}_t$ can be product of the state vector $\bar{s}_t$ and the corresponding weight elements of matrix W_y. As mentioned before, if the desired outputs are between 0 and 1, we can also use a softmax function. The following set of equations depicts these calculations:

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

Equation 34

As mentioned before, 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

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

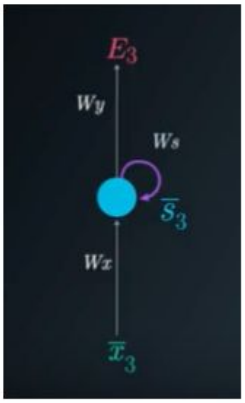
Equation 35

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



The Folded Model at Timestep 3

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.

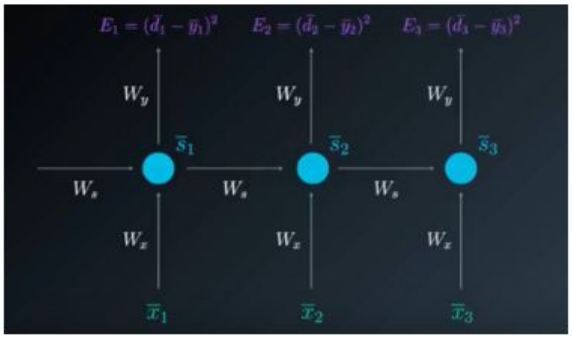
Gradient Considerations in the Folded Model

We will now unfold the model. You will see that unfolding the model in time is very helpful in visualizing the number of steps (translated into multiplication) needed in the Backpropagation Through Time process. These multiplications stem from the chain rule and are easily visualized using this model.

In this video we will understand how to use Backpropagation Through Time (BPTT) when adjusting two weight matrices:

- W_y - the weight matrix connecting the state the output
- W_s - the weight matrix connecting one state to the next state

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



The Unfolded Model at timestep 3

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. Visualization of the calculations path can be found in the video).

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

Equation 36

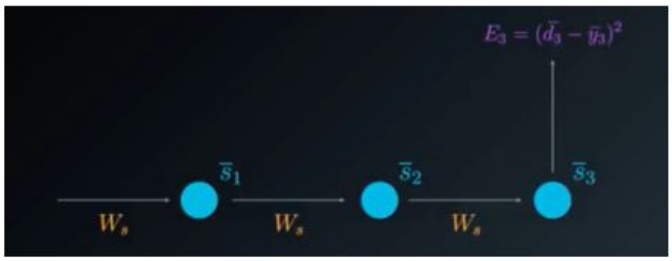
Generally speaking, we can consider multiple timesteps back, and not only 3 as in this example. For an arbitrary timestep N, the gradient calculation needed for adjusting W_y, is:

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

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 : (Notice the use of the chain rule here. If you need, go back to the video to visualize the calculation path).

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

- 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 $\bar{s_2}$ to $\bar{s_3}$. If you need, go back to the video to visualize the calculation path).

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

Equation 39

- 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 $\bar{s_3}$. If you need, go back to the video to visualize the calculation path).

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

Equation 40

After considering the contributions from all three states: $\bar{s_3}$,$\bar{s_2}$ and $\bar{s_1}$, 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**:

$$\begin{aligned}\frac{\partial E_3}{\partial W_s} = &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial W_s} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial W_s} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial \bar{s_1}}\frac{\partial \bar{s_1}}{\partial W_s}\end{aligned}$$

Equation 41

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 41, 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**:

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

Equation 42

Notice that Equation 6 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.

We still need to adjust W_x, the weight matrix connecting the input to the state.

Let's take a small break. You can use this time to go over the **BPTT** process we've seen so far. Try to get yourself comfortable with the math.

Let's take a small break. You can use this time to go over the **BPTT** process we've seen so far. Try to get yourself comfortable with the math.

Once you are feeling confident with the content of the video you just viewed, try to derive the calculations for adjusting the last matrix, W_x by yourself. This is by no means a must, but if you feel that you are up for the challenge, go for it! It will be interesting to compare your notes with ours.

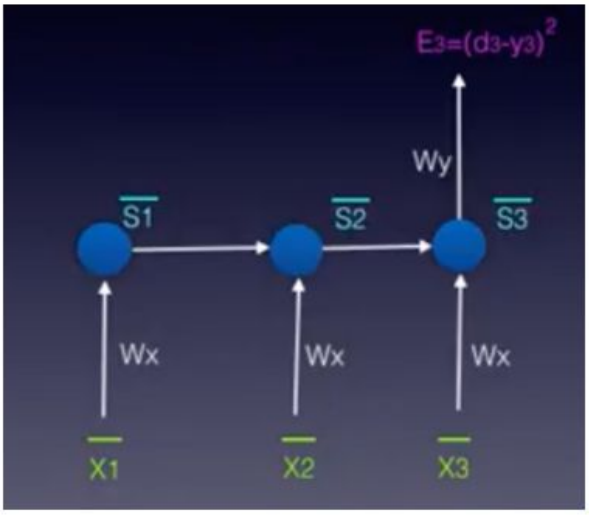
If you chose to take on the challenge, focus on simplifying the unfolded model, leaving only what you need for the calculations. Sketch the backpropagation "path", and step by step think of how the chain rule helps with the derivations here. Don't forget to **accumulate!**.

Last step! Adjusting W_x, the weight matrix connecting the input to the state.

If you took on the previous challenge of deriving the math by yourself first, sit back, fasten your seat belts and compare our notes to yours! Don't worry if you made mistakes, we all do. Your mistakes will help you learn what to avoid next time.

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:



Simplified Unfolded model for Adjusting Wx

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 : (Notice the use of the chain rule here. If you need, go back to the video to visualize the calculation path).

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

Equation 43

- 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 $\bar{s_3}$. If you need, go back to the video to visualize the calculation path).

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

Equation 44

- 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 $\bar{s_3}$. If you need, go back to the video to visualize the calculation path).

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

Equation 45

After considering the contributions from all three states: $\bar{s_3}$,$\bar{s_2}$ and $\bar{s_1}$, we will **accumulate** them to find the final gradient calculation.

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

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

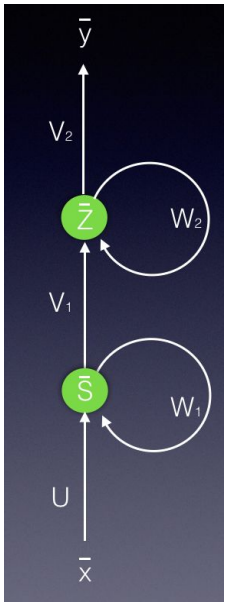
Equation 46

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**:

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

Equation 47

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.



Consider the above folded RNN Model. Both states **S** and **Z** have multiple neurons in each layer. The mathematical derivation of state **Z** at time t is: **Ans: Eqn D**

Equation A $$z_t = \phi(s_t v_1 + z_{t-1} w_2)$$

Equation B $$\bar{z}_t = \phi(\bar{s}_t v_1 + \bar{z}_t w_2)$$

Equation C $$z_t = \phi(s_t v_1 + z_t w_2)$$

Equation D $$\bar{z}_t = \phi(\bar{s}_t v_1 + \bar{z}_{t-1} w_2)$$

Solution

$\bar{z}$ and $\bar{s}$ are vectors, as we indicate that they have multiple neurons in each layer. Using this logic we can understand that equations A and C are incorrect. Since w_2 connects the hidden state $\bar{z}$ to itself, we know that we need to consider the previous timestep here. Therefore only equation D is the correct one.

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 V1 at time t, over a single timestep ? **Ans: Eqn B**

Equation A $$\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v_1}}$$

Equation B $$\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{Z}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v_1}}$$

Equation C $$\Delta v_1 = \frac{\partial E_t}{\partial v_1} = \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{Z}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v_1}}$$

Equation D $$\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{v_1}}$$

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. **Ans : Eqn 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

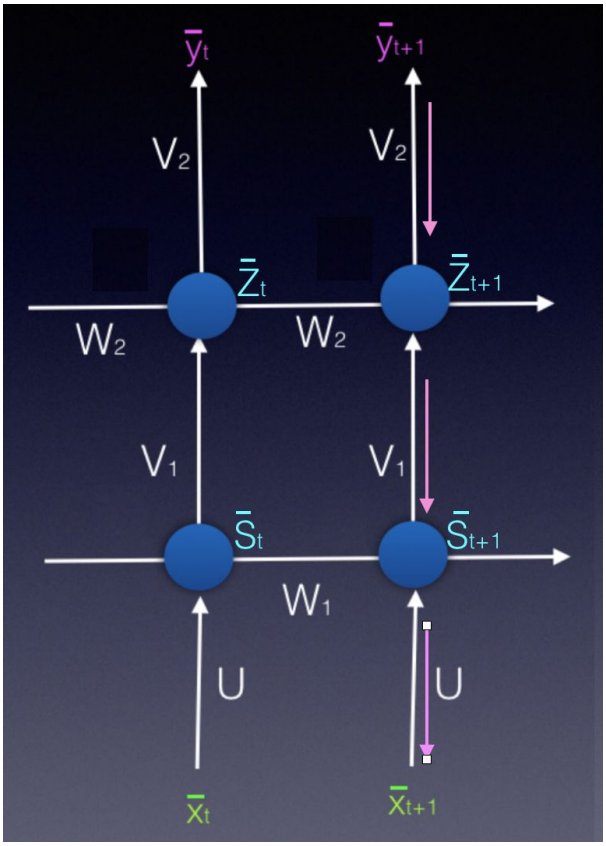
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U} + \frac{\partial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

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

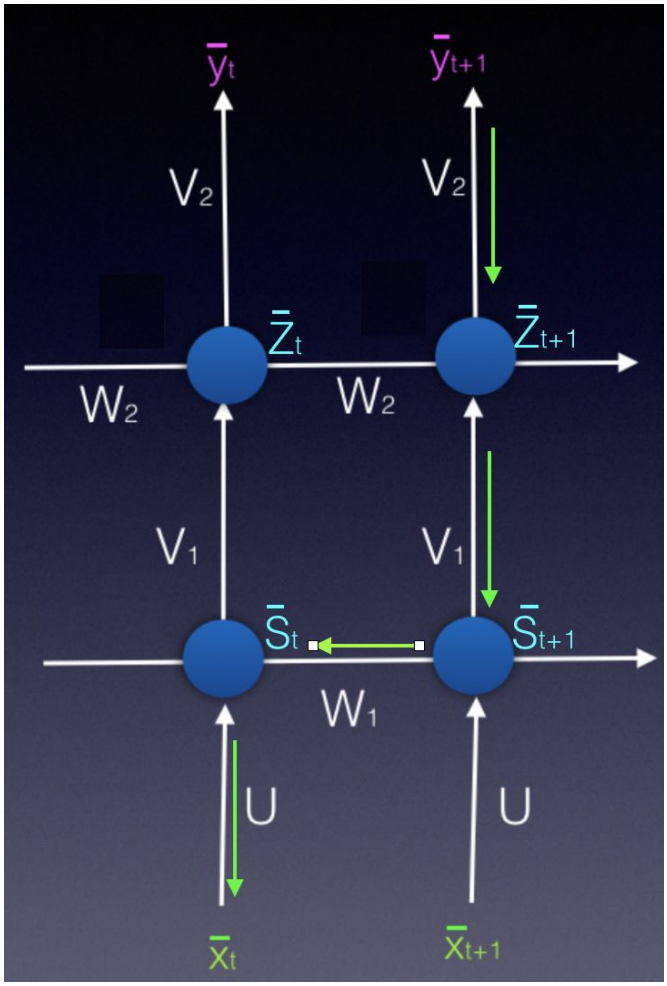
To understand how to update weight matrix U, we will need to unfold the model in time. We will unfold the model over two time steps, as we need to look only time t and time t+1. The following three pictures will help you understand the **three** paths we need to consider. Notice that we have two hidden 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:

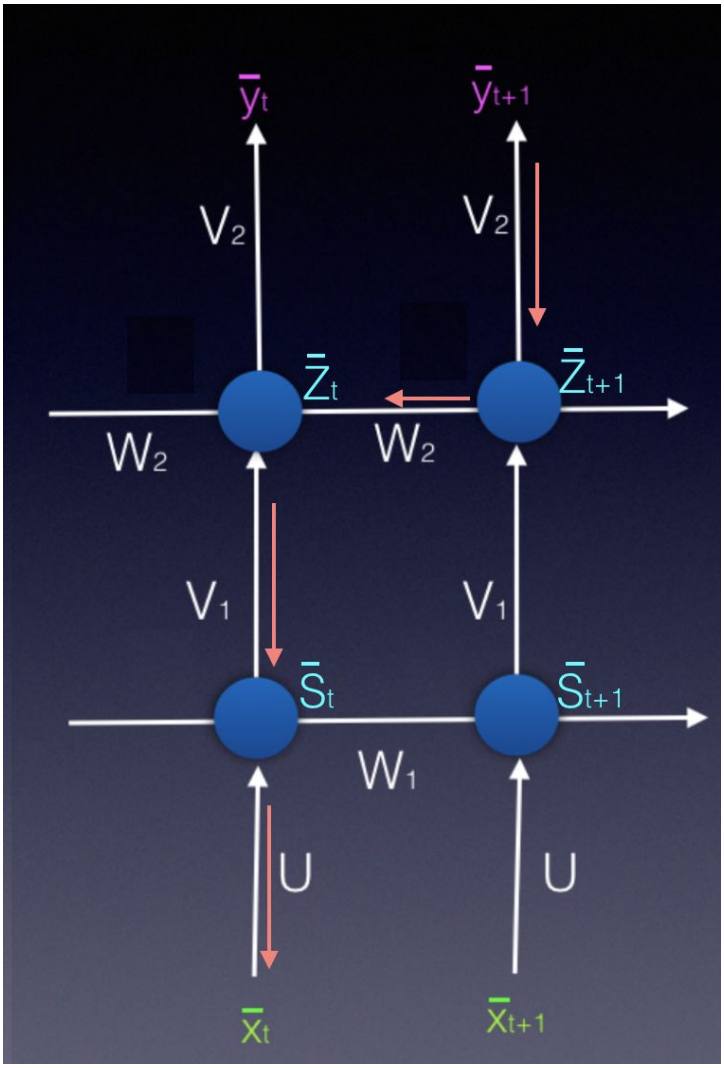
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$



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:

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

$$+$$

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

$$+$$

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

This section is given as bonus material and is not mandatory. If you are curious how we derived the final accumulative equation for BPTT, this section will help you out.

In the previous videos, we talked about **Backpropagation Through Time**. We used a lot of partial derivatives, accumulating the contributions to the change in the error from each state. Remember? When we needed a general scheme for the BPTT, I simply displayed the equation without giving you further explanations.

As a reminder, the following two equations were derived when adjusting the weights of matrix W_s and matrix W_x:

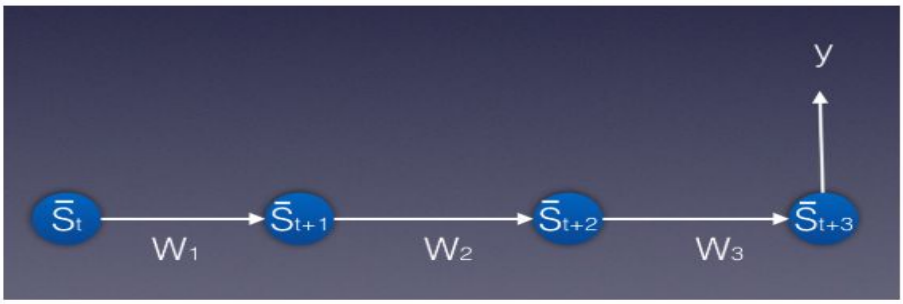
$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_s}$$

Equation 48: BPTT calculations for the purpose of adjusting Ws

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

Equation 49: BPTT calculations for the purpose of adjusting Wx

To generalize the case, we will avoid proving equation 48 or 49, and will focus on a general framework. Let's look at the following sketch, presenting a portion of a network:



In the picture above, we have four states, starting with s_t. We will initially consider the three weight matrices W_1,W_2 and W_3 as three different matrices.

Using the chain rule we can derive the following three equations:

$$\frac{\partial y}{\partial W_3} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W_3}$$

$$\frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W_2}$$

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W_1}$$

Equation 50 (Equation set)

In **Backpropagation Through Time** we accumulate the contributions, therefore:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial W_1} + \frac{\partial y}{\partial W_2} + \frac{\partial y}{\partial W_3}$$

Equation 51

Since this network is displayed as *unfolded in time*, we understand that the weight matrices connecting each of the states are identical. Therefore:

$$W_1 = W_2 = W_3$$

Lets simply call it weight matrix W. Therefore:

$$W_1 = W_2 = W_3 = W$$

Equation 52

From *equation 52*, *equation 51* and the *set of equations 50* we derive that:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

$$+ \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 53

Equation 53 summarizes the mathematical procedure of BPTT and can be simply written as:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+3} \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 54

Notice that for $i = t + 1$, we derive the following:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 55

With the use of the chain rule we can derive the following equation (displayed in *set of equations 50*).

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 56

A general derivation of the BPTT calculation can be displayed the following way:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial \bar{s}_{t+N}} \frac{\partial \bar{s}_{t+N}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 57

As you have seen, in RNNs the current state depends on the input as well as the previous states, with the use of an activation function.

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

Equation 56

The current output is a simple linear combination of the current state elements with the corresponding weight matrix.

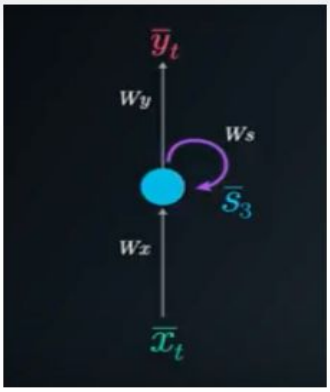
$\bar{y}_t = \bar{s}_t W_y$ (without the use of an activation function)

or

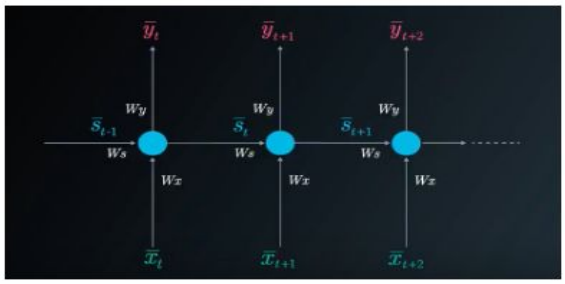
$\bar{y}_t = \sigma(\bar{s}_t W_y)$ (with the use of an activation function)

Equation 57

We can represent the recurrent network with the use of a folded model or an unfolded model:



The RNN Folded Model



The RNN Unfolded Model

In the case of a single hidden (state) layer, we will have three weight matrices to consider. Here we use the following notations:

W_x - represents the weight matrix connecting the inputs to the state layer.

W_y - represents the weight matrix connecting the state to the output.

W_s - represents the weight matrix connecting the state from the previous timestep to the state in the following timestep.

The gradient calculations for the purpose of adjusting the weight matrices are the following:

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

Equation 58

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

Equation 59

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

Equation 60

In equations _51_ and _52_ we used **Backpropagation Through Time (BPTT)** where we accumulate all of the contributions from previous timesteps.

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 δ_{ij} represents the gradient calculated once every inputs sample and M represents the number of gradients we accumulate in the process).

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

Equation 61

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.

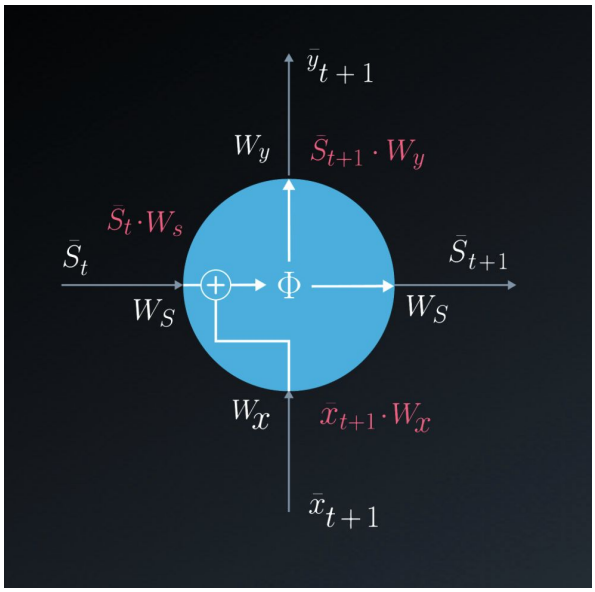
In RNNs we can also have the opposite problem, called the **exploding gradient** problem, in which the value of the gradient grows uncontrollably. A simple solution for the exploding gradient problem is **Gradient Clipping**.

You can concentrate on Algorithm 1 which describes the gradient clipping idea in simplicity.

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

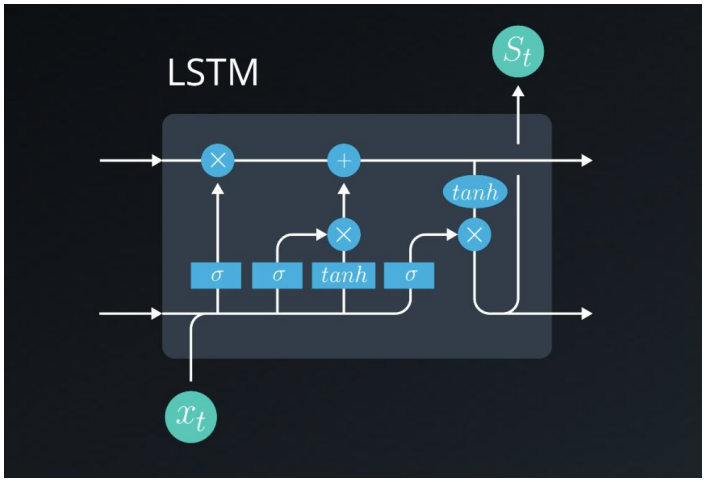
If we take a close look at the RNN neuron, we can see that we have simple linear combinations (with or without the use of an activation function). We can also see that we have a single addition.

Zooming in on the neuron, we can graphically see this in the following configuration:



Closeup Of The RNN Neuron

The **LSTM** cell is a bit more complicated. If we zoom in on the cell, we can see that the mathematical configuration is the following:



Closeup Of the LSTM Cell

The LSTM cell allows a recurrent system to learn over many time steps without the fear of losing information due to the vanishing gradient problem. It is fully differentiable, therefore gives us the option of easily using backpropagation when updating the weights.

Now that you've gone through the **Recurrent Neural Network lesson**, I'll be teaching you what an **LSTM** is. This stands for **Long Short Term Memory Networks**, and are quite useful when our neural network needs to switch between remembering recent things, and things from long time ago. But first, I want to give you some great references to study this further. There are many posts out there about LSTMs, here are a few of my favorites:

- Chris Olah's LSTM post
- Edwin Chen's LSTM post
- Andrej Karpathy's lecture on RNNs and LSTMs from CS231n

The output of the *Learn Gate* is $N_t i_t$ where:

$$N_t = \tanh(W_n[STM_{t-1}, E_t] + b_n)$$

$$i_t = \sigma(W_i[STM_{t-1}, E_t] + b_i)$$

Equation 1

The output of the *Forget Gate* is $LTM_{t-1} f_t$ where:

$$f_t = \sigma(W_f[STM_{t-1}, E_t] + b_f)$$

Equation 2

The output of the *Remember Gate* is:

$LTM_{t-1} f_t + N_t i_t$

Equation 3

(N_t, i_t and f_t are calculated in *equations 1* and _2_)

At 00:27 : Luis refers to obtaining New Short Term Memory instead it's **New Long Term Memory**.

The output of the *Use Gate* is $U_t V_t$ where:

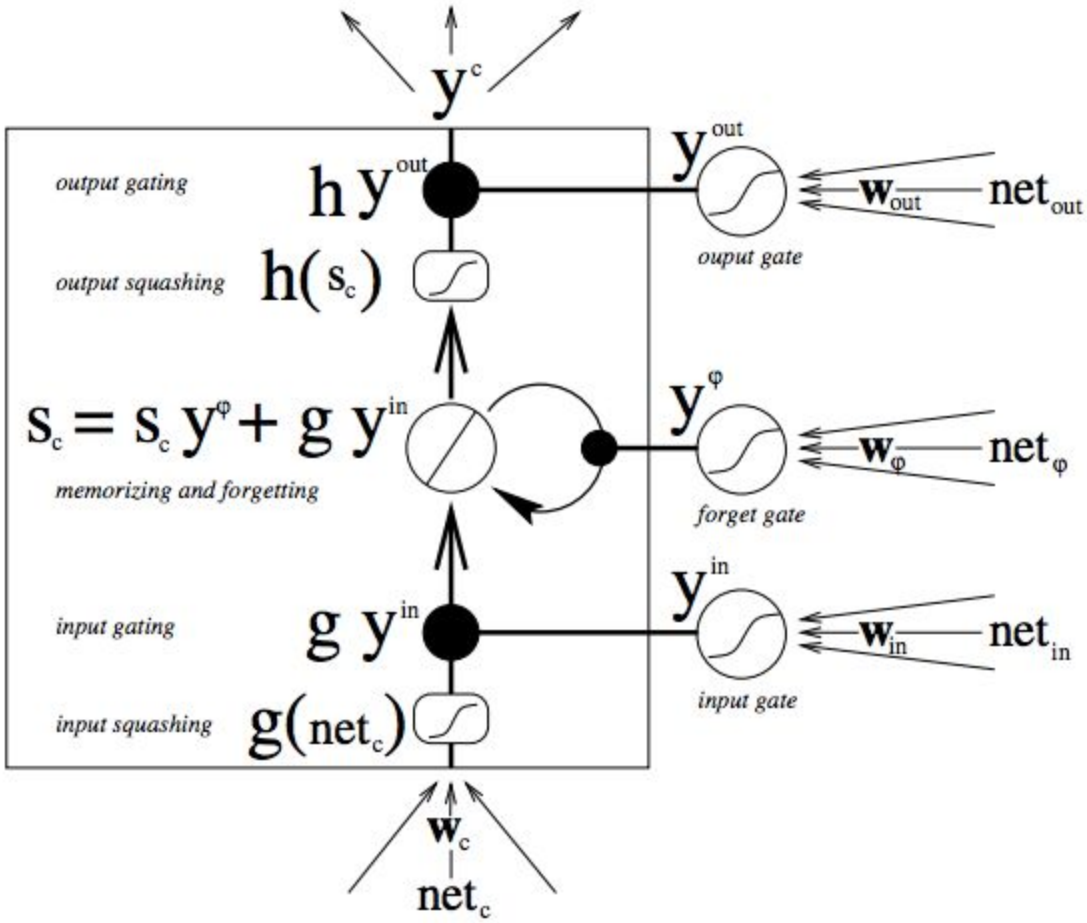
$$U_t = \tanh(W_u LTM_{t-1} f_t + b_u)$$

$$V_t = \sigma(W_v[STM_{t-1}, E_t] + b_v)$$

Equation 4

If you would like to deepen your knowledge even more, go over the following tutorial. Focus on the overview titled: **Long Short-Term Memory Units (LSTMs)**.

If you are feeling confident enough, skip the overview and jump right into our next question:



The LSTM cell- taken form the Deep Learning tutorial

The illustration above is of a LSTM cell. What would be the values of the three gates in situations where the cell retains information for a long period, without accepting a new input or producing an output?

Ans: The Inputs Gate : close to 0; The Forget Gate : close to 1; The Output Gate : close to 0

Character-Level RNN

We will be building and training a basic character-level RNN, specifically an LSTM which reads words as a series of characters and outputs a prediction and “hidden state” at each time step, feeding its previous hidden state into each next step. Next, we'll have our team product lead, Mat, talk about what a character-level RNN can be expected to do.

Sequences of Data

The core reason that recurrent neural networks are exciting is that they allow us to operate over sequences of vectors: sequences in the input, the output, or in some cases, both!

Most RNN's expect to see a sequence of data in a fixed batch size, much like we've seen images processed in fixed batch sizes. The batches of data affect how the hidden state of an RNN train, so next you'll learn more about batching data.

Hyperparameters

For this section, we're introducing a new Udacity instructor, Jay Alammar. Jay has done some great work in interactive explorations of neural networks, check out his blog.

Jay will be reviewing some of the material you saw in the Deep Neural Networks section on hyperparameters, and he will also introduce the hyperparameters used in Recurrent Neural Networks Exponential Decay in TensorFlow.

Adaptive Learning Optimizers

- AdamOptimizer
- AdagradOptimizer

QUESTION 1 OF 2

Say you're training a model. If the output from the training process looks as shown below, what action would you take on the learning rate to improve the training?

```
Epoch 1, Batch 1, Training Error: 8.4181
Epoch 1, Batch 2, Training Error: 8.4177
Epoch 1, Batch 3, Training Error: 8.4177
Epoch 1, Batch 4, Training Error: 8.4173
Epoch 1, Batch 5, Training Error: 8.4169
```

Ans: Increase the learning rate

QUESTION 2 OF 2

Say you're training a model. If the output from the training process looks as shown below, what action would you take on the learning rate to improve the training?

```
Epoch 1, Batch 1, Training Error: 8.71
Epoch 1, Batch 2, Training Error: 3.25
Epoch 1, Batch 3, Training Error: 4.93
Epoch 1, Batch 4, Training Error: 3.30
Epoch 1, Batch 5, Training Error: 4.82
```

Ans:

- Decrease the learning rate
- Use an adaptive learning rate

Systematic evaluation of CNN advances on the ImageNet by Dmytro Mishkin, Nikolay Sergievskiy, Jiri Matas

The number of training iterations is a hyperparameter we can optimize automatically using a technique called early stopping (also "early termination").

ValidationMonitor (Deprecated)

In tensorflow, we can use a ValidationMonitor with tf.contrib.learn to not only monitor the progress of training, but to also stop the training when certain conditions are met.

The following example from the ValidationMonitor documentation shows how to set it up. Note that the last three parameters indicate which metric we're optimizing.

```
validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
  test_set.data,
  test_set.target,
  every_n_steps=50,
  metrics=validation_metrics,
  early_stopping_metric="loss",
  early_stopping_metric_minimize=True,
  early_stopping_rounds=200)
```

The last parameter indicates to ValidationMonitor that it should stop the training process if the loss did not decrease in 200 steps (rounds) of training.

The validation_monitor is then passed to tf.contrib.learn's "fit" method which runs the training process:

```
classifier = tf.contrib.learn.DNNClassifier(
  feature_columns=feature_columns,
  hidden_units=[10, 20, 10],
  n_classes=3,
  model_dir="/tmp/iris_model",
  config=tf.contrib.learn.RunConfig(save_checkpoints_secs=1))

classifier.fit(x=training_set.data,
           y=training_set.target,
           steps=2000,
           monitors=[validation_monitor])
```

SessionRunHook

More recent versions of TensorFlow deprecated monitors in favor of SessionRunHooks. SessionRunHooks are an evolving part of tf.train, and going forward appear to be the proper place where you'd implement early stopping.

At the time of writing, two pre-defined stopping monitors exist as a part of tf.train's training hooks:

- StopAtStepHook: A monitor to request the training stop after a certain number of steps
- NanTensorHook: a monitor that monitor's loss and stops training if it encounters a NaN loss

"in practice it is often the case that 3-layer neural networks will outperform 2-layer nets, but going even deeper (4,5,6-layer) rarely helps much more. This is in stark contrast to Convolutional Networks, where depth has been found to be an extremely important component for a good recognition system (e.g. on order of 10 learnable layers)." ~ Andrej Karpathy in https://cs231n.github.io/neural-networks-1/

More on Capacity

A more detailed discussion on a model's capacity appears in the Deep Learning book, chapter 5.2 (pages 110-120).

LSTM Vs GRU

"These results clearly indicate the advantages of the gating units over the more traditional recurrent units. Convergence is often faster, and the final solutions tend to be better. However, our results are not conclusive in comparing the LSTM and the GRU, which suggests that the choice of the type of gated recurrent unit may depend heavily on the dataset and corresponding task."

Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling by Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio

"The GRU outperformed the LSTM on all tasks with the exception of language modelling"

An Empirical Exploration of Recurrent Network Architectures by Rafal Jozefowicz, Wojciech Zaremba, Ilya Sutskever

"Our consistent finding is that depth of at least two is beneficial. However, between two and three layers our results are mixed. Additionally, the results are mixed between the LSTM and the GRU, but both significantly outperform the RNN."

Visualizing and Understanding Recurrent Networks by Andrej Karpathy, Justin Johnson, Li Fei-Fei

"Which of these variants is best? Do the differences matter? Greff, et al. (2015) do a nice comparison of popular variants, finding that they're all about the same. Jozefowicz, et al. (2015) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks."

Understanding LSTM Networks by Chris Olah

"In our [Neural Machine Translation] experiments, LSTM cells consistently outperformed GRU cells. Since the computational bottleneck in our architecture is the softmax operation we did not observe large difference in training speed between LSTM and GRU cells. Somewhat to our surprise, we found that the vanilla decoder is unable to learn nearly as well as the gated variant."

Massive Exploration of Neural Machine Translation Architectures by Denny Britz, Anna Goldie, Minh-Thang Luong, Quoc Le

Example RNN Architectures

Application	Cell	Layers	Size	Vocabulary	Embedding Size	Learning Rate	
Speech Recognition (large vocabulary)	LSTM	5, 7	600, 1000	82K, 500K	--	--	paper
Speech Recognition	LSTM	1, 3, 5	250	--	--	0.001	paper
Machine Translation (seq2seq)	LSTM	4	1000	Source: 160K, Target: 80K	1,000	--	paper
Image Captioning	LSTM	--	512	--	512	(fixed)	paper
Image Generation	LSTM	--	256, 400, 800	--	--	--	paper
Question Answering	LSTM	2	500	--	300	--	pdf

Text Summarization	GRU		200	Source: 119K, Target: 68K	100	0.001	pdf

How do Long Short Term Memory (LSTM) cells and Gated Recurrent Unit (GRU) cells compare?

Ans: It depends.. It's probably worth it to compare the two on any task and dataset.

Which embedding size looks more reasonable for the majority of cases?

Ans: 500, While some tasks show reasonable performance with embedding sizes between 50-200, it's not unusual to see it go up 500 or even 1000.

If you want to learn more about hyperparameters, these are some great resources on the topic:

- Practical recommendations for gradient-based training of deep architectures by Yoshua Bengio
- Deep Learning book - chapter 11.4: Selecting Hyperparameters by Ian Goodfellow, Yoshua Bengio, Aaron Courville
- Neural Networks and Deep Learning book - Chapter 3: How to choose a neural network's hyper-parameters? by Michael Nielsen
- Efficient BackProp (pdf) by Yann LeCun

More specialized sources:

- How to Generate a Good Word Embedding? by Siwei Lai, Kang Liu, Liheng Xu, Jun Zhao
- Systematic evaluation of CNN advances on the ImageNet by Dmytro Mishkin, Nikolay Sergievskiy, Jiri Matas
- Visualizing and Understanding Recurrent Networks by Andrej Karpathy, Justin Johnson, Li Fei-Fei

ATTENTION

Sequence to Sequence Models

Before we jump into learning about attention models, let's recap what you've learned about sequence to sequence models. We know that RNNs excel at using and generating sequential data, and sequence to sequence models can be used in a variety of applications!

Encoders and Decoders

The encoder and decoder do not have to be RNNs; they can be CNNs too!

In the example above, an LSTM is used to generate a sequence of words; LSTMs "remember" by keeping track of the input words that they see and their own hidden state.

In computer vision, we can use this kind of encoder-decoder model to generate words or captions for an input image or even to generate an image from a sequence of input words. We'll focus on the first case: generating captions for images, and you'll learn more about caption generation in the next lesson. For now know that we can input an image into a CNN (encoder) and generate a descriptive caption for that image using an LSTM (decoder).

Elective: Text Sentiment Analysis

If you would like more practice with analyzing sequences of words with a simple network, now would be a great time to check out the elective section: Text Sentiment Analysis. In this section, Andrew Trask teaches you how to convert words into vectors and then analyze the sentiment of these vectors. He goes through constructing and tuning a model *and* addresses some common errors in text analysis. This section does not contain material that is required to complete this program or the project in this section, but it is interesting and you may find it useful!

1. True or False: A sequence-to-sequence model processes the input sequence all in one step

Ans: False - a seq2seq model works by feeding one element of the input sequence at a time to the encoder

2. Which of the following is a limitation of seq2seq models which can be solved using attention methods?

Ans: 1.The fixed size of the context matrix passed from the encoder to the decoder is a bottleneck

2.Difficulty of encoding long sequences and recalling long-term dependencies

3. How large is the context matrix in an attention seq2seq model?

Ans: Depends on the length of the input sequence

4. In machine translation applications, the encoder and decoder are typically

Ans: Recurrent Neural Networks(Typically vanilla RNN, LSTM,GRUs)

5. What's a more reasonable embedding size for a real-world application?

Ans: 200

6. What are the steps that require calculating an attention vector in a seq2seq model with attention?

Ans: Every time step in the decoder only

7. Which of the following are valid scoring methods for attention?

Ans: A. Concat/additive

B. Dot product

C. General

8. What's the intuition behind using dot product as a scoring method?

Ans: The dot product of two vectors in word-embedding system space is a measure of similarity between them

Super interesting computer vision applications using attention:

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention [pdf]

Bottom-Up and Top-Down Attention for Image Captioning and Visual Question Answering [pdf]

Video Paragraph Captioning Using Hierarchical Recurrent Neural Networks [pdf]

Every Moment Counts: Dense Detailed Labeling of Actions in Complex Videos [pdf]

Tips and Tricks for Visual Question Answering: Learnings from the 2017 Challenge [pdf]

Visual Question Answering: A Survey of Methods and Datasets [pdf]

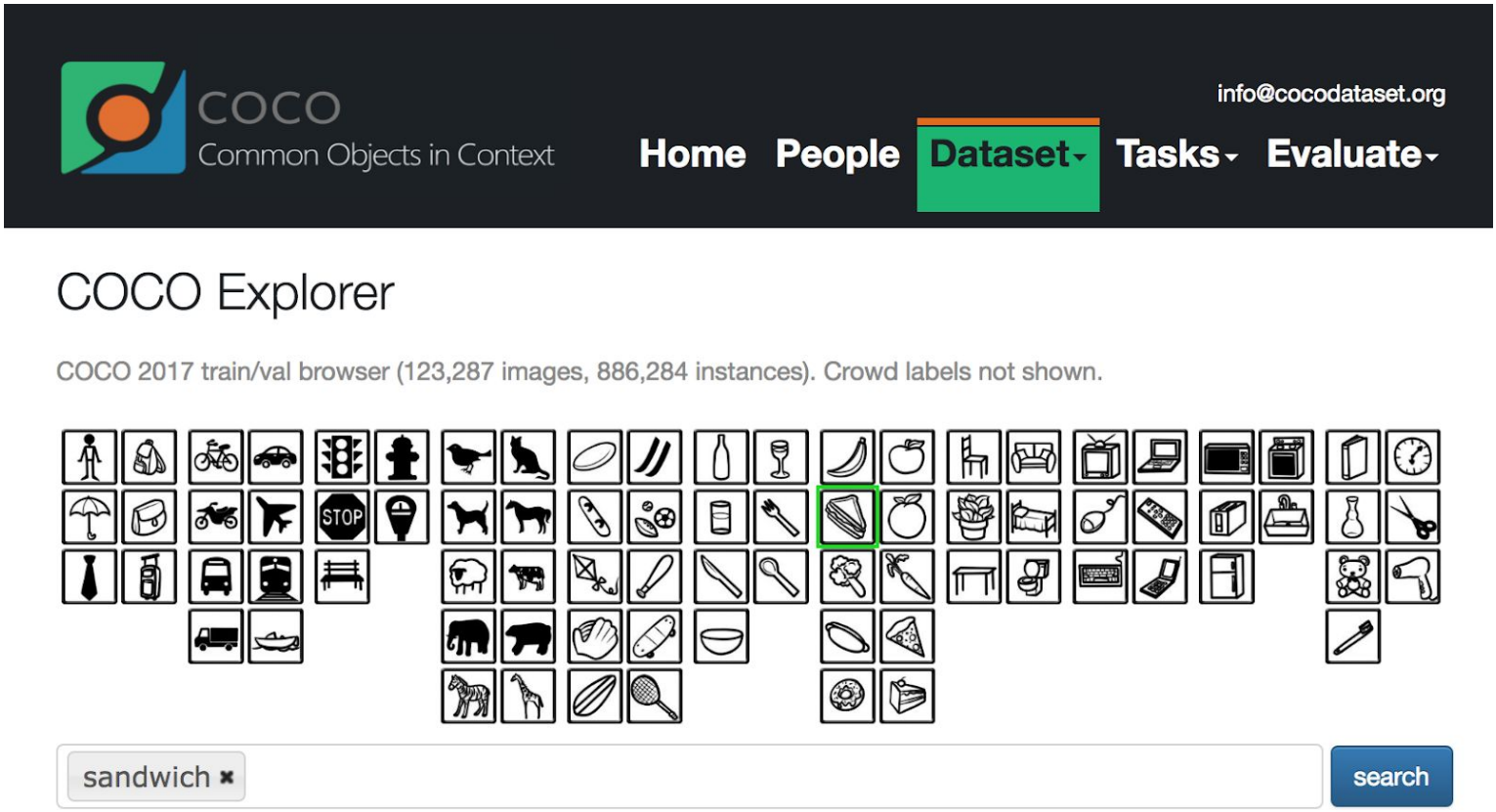
COCO Dataset

The COCO dataset is one of the largest, publicly available image datasets and it is meant to represent realistic scenes. What I mean by this is that COCO does not overly pre-process images, instead these images come in a variety of shapes with a variety of objects and environment/lighting conditions that closely represent what you might get if you compiled images from many different cameras around the world.

To explore the dataset, you can check out the dataset website.

Explore

Click on the explore tab and you should see a search bar that looks like the image below. Try selecting an object by it's icon and clicking search!



A sandwich is selected by icon.

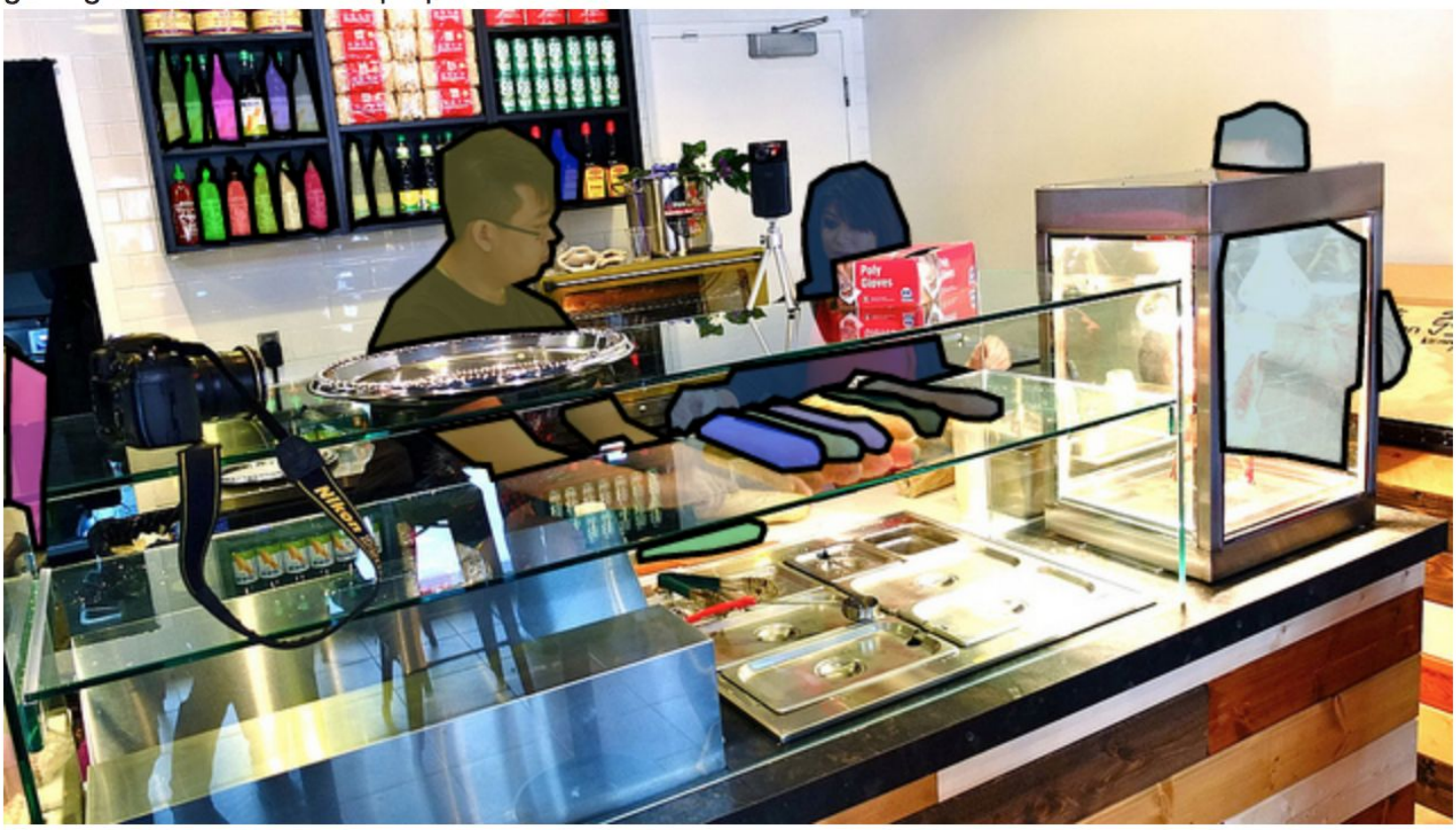
You can select or deselect multiple objects by clicking on their corresponding icon. Below are some examples for what a `sandwich` search turned up! You can see that the initial results show colored overlays over objects like sandwiches and people and the objects come in different sizes and orientations.

COCO sandwich detections

Captions

COCO is a richly labeled dataset; it comes with class labels, labels for segments of an image, *and* a set of captions for a given image. To see the captions for an image, select the text icon that is above the image in a toolbar. Click on the other options and see what the result is.

the counter of a restaurant with food displayed.
a store has their display of food with workers behind it
a few people that are out front of a cafe
a man prepares food behind a counter with two others.
getting a lesson to how to prepare the food.



Example captions for an image of people at a sandwich counter.

When we actually train our model to generate captions, we'll be using these images as input and sampling *one* caption from a set of captions for each image to train on.

Supporting Materials

Creating COCO, paper

Words to Vectors

At this point, we know that you cannot directly feed words into an LSTM and expect it to be able to train or produce the correct output. These words first must be turned into a numerical representation so that a network can use normal loss functions and optimizers to calculate how "close" a predicted word and ground truth word (from a known, training caption) are. So, we typically turn a sequence of words into a sequence of numerical values; a vector of numbers where each number maps to a specific word in our vocabulary.

To process words and create a vocabulary, we'll be using the Python text processing toolkit: NLTK. in the below video, we have one of our content developers, Arpan, explain the concept of word tokenization with NLTK.

Reference:

- `nltk.tokenize` package: http://www.nltk.org/api/nltk.tokenize.html

Later, you'll see how we take a tokenized representation of a caption to create a Python dictionary that maps unique words in our captions dataset to unique integers.

Training vs. Testing

During training, we have a true caption which is fixed, but during testing the caption is being actively generated (starting with `<start>`), and at each step you are getting the most likely next word and using that as input to the next LSTM cell.

Caption Generation, Test Data

After the CNN sees a new, test image, the decoder should first produce the `<start>` token, then an output distribution at each time step that indicates the most likely next word in the sentence. We can sample the output distribution (namely, extract the one word in the distribution with the highest probability of being the next word) to get the next word in the caption and keep this process going until we get to another special token: `<end>`, which indicates that we are done generating a sentence.

Want to Learn more from Kelvin?

Kelvin has worked with us on one more high-level lesson about fully-convolutional networks (FCN's). To learn more from him and about a complex deep learning model, I suggest you check out the elective section: **Elective: More Deep Learning Models** and go to Fully-Convolutional Networks!

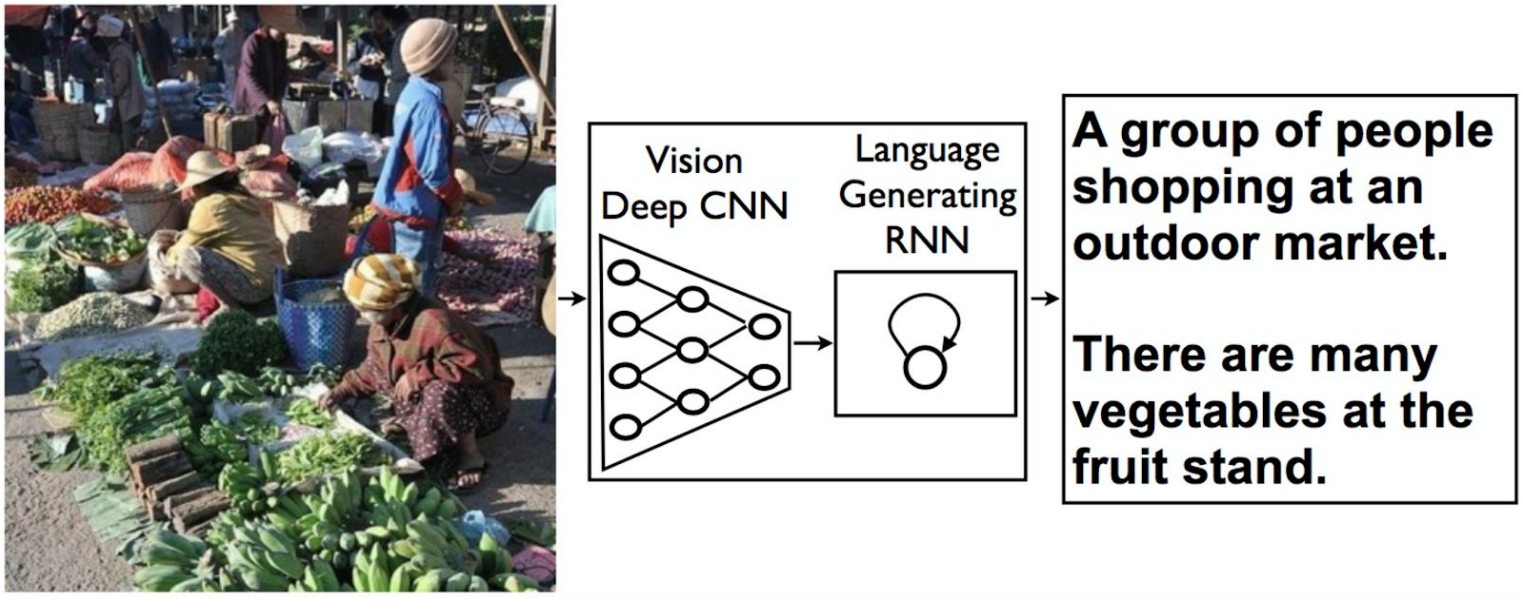


Image Captioning Model

Project Overview

In this project, you will create a neural network architecture to automatically generate captions from images.

After using the Microsoft Common Objects in COntext (MS COCO) dataset to train your network, you will test your network on novel images!

Project Instructions

The project is structured as a series of Jupyter notebooks that are designed to be completed in sequential order:

- 0_Dataset.ipynb
- 1_Preliminaries.ipynb

- 2_Training.ipynb
- 3_Inference.ipynb

You can find these notebooks in the Udacity workspace that appears in the concept titled **Project: Image Captioning**. This workspace provides a Jupyter notebook server directly in your browser.

You can read more about workspaces (and how to toggle GPU support) in the following concept (**Introduction to GPU Workspaces**). This concept will show you how to toggle GPU support in the workspace.

You MUST enable GPU mode for this project and submit your project after you complete the code in the workspace.

A completely trained model is expected to take between 5-12 hours to train well on a GPU; it is suggested that you look at early patterns in loss (what happens in the first hour or so of training) as you make changes to your model, so that you only have to spend this large amount of time training your final model.

Should you have any questions as you go, please post in the Student Hub!

Evaluation

Your project will be reviewed by a Udacity reviewer against the CNN project rubric. Review this rubric thoroughly, and self-evaluate your project before submission. As in the first project, **you'll find that only some of the notebooks and files are graded**. All criteria found in the rubric must meet specifications for you to pass.

Ready to submit your project?

It is a known issue that the COCO dataset is not well supported for download on Windows, and so you are required to complete the project in the GPU workspace. This will also allow you to bypass setting up an AWS

account and downloading the dataset, locally. If you would like to refer to the project code, you may look at this version (in PyTorch 0.4.0) at the linked Github repo.

Once you've completed your project, you may **only submit from the workspace** for this project, linked here. Click Submit, a button that appears on the bottom right of the *workspace*, to submit your project.

For submitting from the workspace, directly, please make sure that you have deleted any large files and model checkpoints in your notebook directory before submission** or your project file may be too large to download and grade.

GPU Workspaces

Note: To load the COCO data in the workspace, you *must have GPU mode enabled*.

In the next section, you'll learn more about these types of workspaces.

LSTM Decoder

In the project, we pass all our inputs as a sequence to an LSTM. A sequence looks like this: first a feature vector that is extracted from an input image, then a start word, then the next word, the next word, and so on!

Embedding Dimension

The LSTM is defined such that, as it sequentially looks at inputs, it expects that each individual input in a sequence is of a **consistent size** and so we *embed* the feature vector and each word so that they are `embed_size`.

Sequential Inputs

So, an LSTM looks at inputs sequentially. In PyTorch, there are two ways to do this.

The first is pretty intuitive: for all the inputs in a sequence, which in this case would be a feature from an image, a start word, the next word, the next word, and so on (until the end of a sequence/batch), you loop through each input like so:

```
for i in inputs:
    # Step through the sequence one element at a time.
    # after each step, hidden contains the hidden state.
    out, hidden = lstm(i.view(1, 1, -1), hidden)
```

The second approach, which this project uses, is to **give the LSTM our *entire sequence*** and have it produce a set of outputs and the last hidden state:

```
# the first value returned by LSTM is all of the hidden states throughout
# the sequence. the second is just the most recent hidden state

# Add the extra 2nd dimension
inputs = torch.cat(inputs).view(len(inputs), 1, -1)
hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # clean out hidden state

out, hidden = lstm(inputs, hidden)
```