- Gradient descent learning with multiple inputs:

a) Using gradient descent to update a weight.

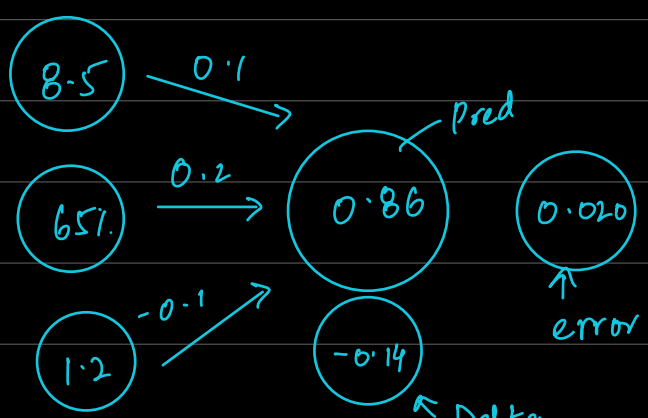
b) Using same technique to update a network containing multiple weights.

weight_delta = input * delta

delta = (pred - true)

derivative

```
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output

weights = [0.1, 0.2, -0.1]
def neural_network(input, weights):
    pred = w_sum(input, weights)
    return pred
```




```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]
```

```
win_or_lose_Binary = [1,1,0,1]
true = win_or_lose_Binary[0]
input = [toes[0], wlrec[0], ntoes[0]]
pred = neural_network(input, weights)
error = (pred - true) **2
delta = pred - true.
```

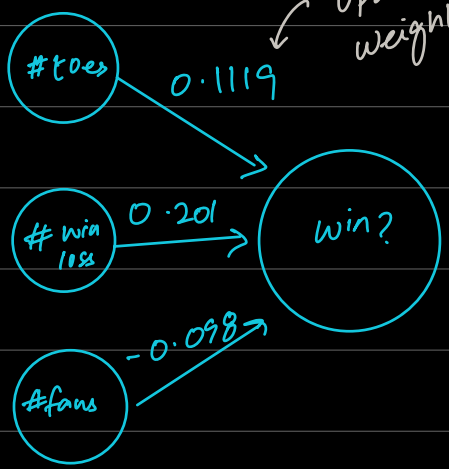
-1.19 ↓ error goes down by -1.19
1 ↑ if weight goes up
1 unit

```
def ele_mul(number, vector):
    output = [0,0,0]
    assert(len(output) == len(vector))
    for i in range(len(vector)):
        output[i] = number * vector[i]
    return output
```

*) Each weight_delta is calculated by taking output delta and multiplying it by its input.

```
pred = neural_network(input, weight)
error = (pred - true) ** 2
delta = pred - true
weight_deltas = elem_mul(delta, input)
```

Learn



alpha = 0.01

```
for i in range(len(weights)):
    weights[i] -= alpha * weight_deltas[i]
```

Learning or slowly updating weights using derivatives so that error reduces to 0.

old weights, weight_deltas

$0.1 - (-1.19 * 0.01) = 0.1119 = \text{weights}[0]$

$0.2 - (-0.091 * 0.01) = 0.2009 = \text{weights}[1]$

$-0.1 - (-0.168 * 0.01) = -0.098 = \text{weights}[2]$

Pred − True

*) **Delta**: A measure of how much higher or lower you want a node's value to be, to predict perfectly given the current training example.

input * Delta — to calculate for each input

*) **Weight_delta**: is an estimate of the direction and amount you should move a weight to reduce node_delta, inferred by derivative.

↳ make it close to zero.

A derivative-based estimate of the direction and amount you should move a weight to reduce the node_delta, accounting for scaling, negative reversal, stopping.

Weight_delta → derivative based estimate of {direction and amount} → you should move a weight

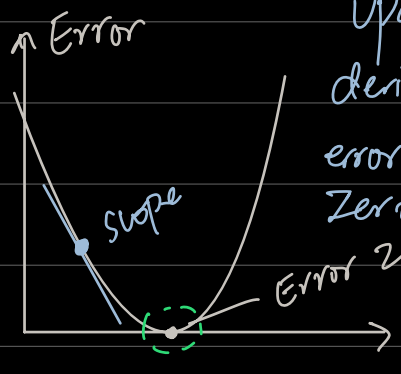
→ to reduce node delta $\frac{dy}{dx}$ = $\frac{\text{change in node\_delta}}{\text{change in weight}}$ — goal

weight_delta or derivative $\frac{dy}{dx} = \frac{1}{2} \frac{\text{change in } y}{\text{change in } x}$ $\frac{\text{increase of } y}{\text{increase of } x}$

$\frac{\text{change in } x}{2}$ - increase of x

*) Alpha controls how quickly the network learns.

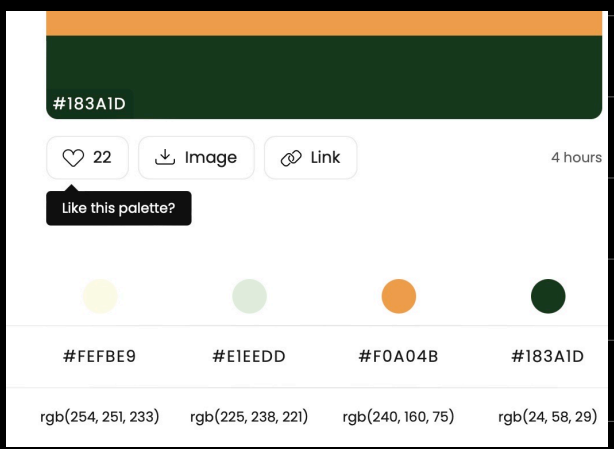
*) With neural networks trying to find the lowest point on this big error plane, where the lowest point refers to the lowest error.



updating weights using slope/derivative so that error reduces to Zero. ZERO point that we want to reach

- Gradient descent with multiple inputs and outputs:

a) generalizes to arbitrarily large networks.



$256 \times 256 \times 256$

Each pixel for Black & white image consists of 8 bits (1 byte) typically. Whereas each colored pixel consists of 24 bits (3 bytes)

rgb - [8 bits, 8 bits, 8 bits]

$2^8 = 256$ different permutations exist of a single Byte

4 permutations

∴ each Byte can represent a number from 0 - 256 and different combinations lead to different color.

two coins → 2

$2^1 = 4$

possible outcomes of coin $2^{[1,0]}$

$2^8 \cdot 2^8 = 2^{16} = 65000$ around

$2^8 \cdot 2^8 \cdot 2^8 = 2^{24}$

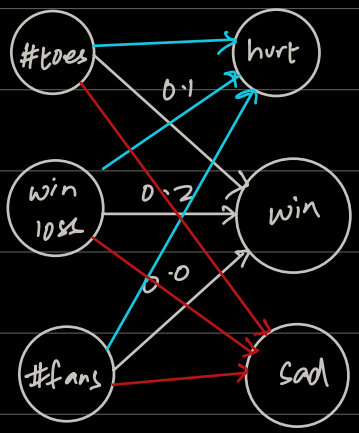
$2^{24} = 16,777,216$ different color variations.

2^{24} ← Blocks or bits

↑ Possible outcomes 1 or 0

= ∴

inputs Predictions



#toes %win #fans

weights = [[0.1, 0.1, -0.3] #hurt
[0.1, 0.2, 0.0] #win
[0.0, 1.3, 0.1]] #sad

```
def w_sum(a, b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output
```

```
def vect_mat_mul(vec, matrix):
    assert(len(vec) == len(matrix))
    output = [0, 0, 0]
    for i in range(len(vectori)):
        output[i] = w_sum(vec,
            matrix[i])
    return output
```

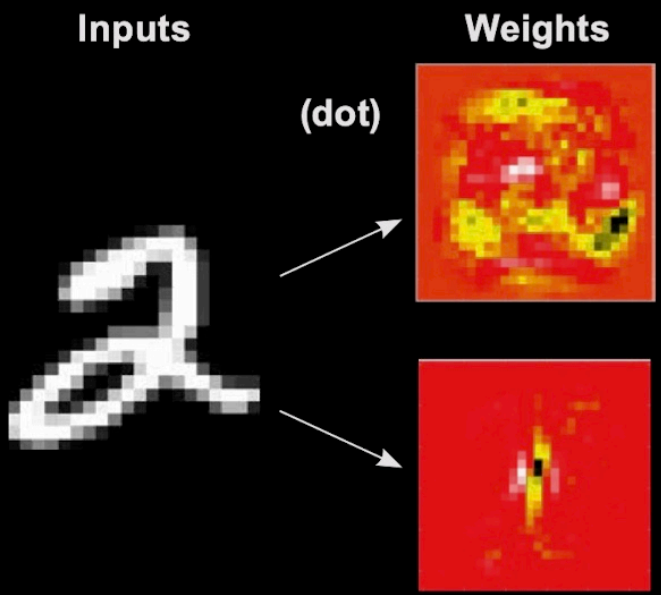
```
def neural_network(input, weights):
    pred = vect_mat_mul(input,
        weight)
    return output
```

```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

hurt = [0.1, 0.0, 0.0, 0.1]
win = [1, 1, 0, 1]
sad = [0.1, 0.0, 0.1, 0.2]

alpha = 0.01
input = [toes[0], wlrec[0], nfans[0]]
true = [hurt[0], win[0], sad[0]]
pred = neural_network(input,
                      weights)
error = [0, 0, 0]
(pred - true) ← delta = [0, 0, 0]

for i in range(len(true)):
    error[i] = (pred[i] - true[i]) **2
    delta = pred[i] - true[i]
```

*) Calculating each weight_delta and putting it on each weight.

*) A dot product is a loose measurement of similarity between two vectors.

*) If weight vector is similar to input vector for number = 2 image then dot product score / value is high and prediction is accurate



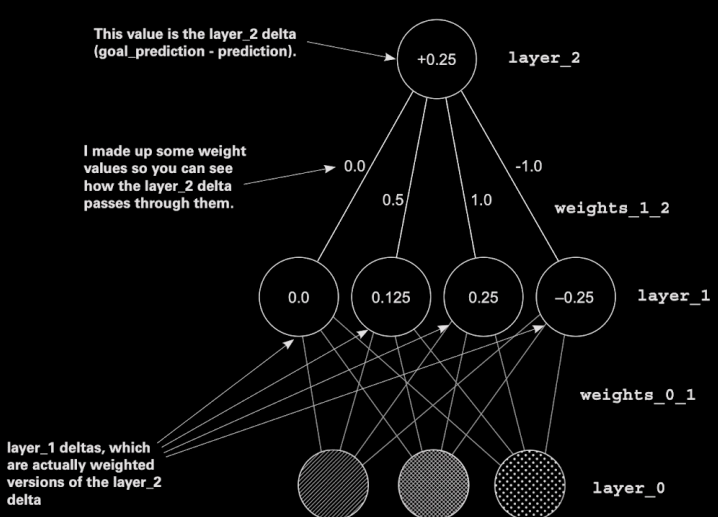
*) The greatest challenge with deep learning is to generalize and not just memorize. Therefore, requires greater amount of data.

- The Prediction is a weighted sum of the inputs. The learning algorithm reward inputs that correlate with output with upward pressure (toward 1) on their weight while penalizing inputs with decorrelation with downward pressure. The weighted sum of inputs find perfect correlation to

input and output by weighting decorrelated inputs too.

Backpropogation: Long - distance error attribute

- The prediction from layer_1 to layer_2 is weighted sum of the values at layer_1. If layer_2 is too high by x amount, then the higher weights contributed more.
- The ones with lower weights from layer_1 to layer_2 contributed less.
- Those weights also describe how much each layer_1 node contributes to the layer_2 prediction.
- The delta at layer_1 can be figured out by multiplying delta with each respective weights for layer_1. This process of moving delta signal is called backpropogation.



*) Regularization is advantageous because if a weight has equal pressure upward and downward then it is not good for anything. In essence, regularization aims to say that only weights with strong correlation can

- Linear vs non Linear:

Stay on. Every else should be silenced as contributing noise. (Trains faster).

For any two multiplications, I can accomplish the same thing using single multiplication. Which is bad.

⇒ If trained the neural network as it is now, it wouldn't converge.

Problem:

⇒ For any two consecutive weighted sums of the input, there exists a single weighted sum with exactly identical behavior. Anything that the three-layer network can do, the two layer network can also do.

- From a correlation standpoint. Each node in the middle layer subscribes to a certain amount of correlation with each input node.
- If the weight from an input to the middle layer is 1.0, then it subscribes to exactly 100% of that nodes' movement. If that nodes goes up by 0.3, the middle node will follow.
- If the weight connecting two nodes is 0.5, each node in the middle layer subscribes to exactly 50% of the nodes movement.

- Each hidden node subscribes to a little correlation from the input nodes.
- Since, the new data set has no correlation. The middle layer provides the ability to selectively correlate with the input.
- The middle layer should sometimes correlate with the input and sometimes not. This way middle layer not just x% correlated to one input and y% correlated to another input. Instead, it can be X% correlated to one input only when it wants to be, but other times not be correlated at all. Also known as Conditional correlation or sometimes correlation.

⇒ By turning off any middle node whenever it would be negative, you allow the network to sometimes subscribe to correlation from various inputs. which was impossible for two - Layer neural networks.

⇒ "If the node is negative, set it to 0" logic is non linearity.

⇒ Many kinds of non linearities exist. Simplest one

that will be used is called relu

```
import numpy as np
np.random.seed(1)

def relu(x):                    # returns x if x>0;
    return (x>0) * x            returns 0 otherwise.

def relu2deriv(output):         # returns 1 for input >0;
    return output>0             0 otherwise.

streetlights = np.array([[1,0,1], [0,1,1],
                         [0,0,1], [1,1,1]])

walk_vs_stop = np.array([[1,1,0,0]]).T      # Transpose.

alpha = 0.2                     # 3 inputs ← 3,4 ← connection or weights from each input
hidden_size = 4
weights_0_1 = 2*np.random.random((3,hidden_size)) - 1      # (4,1)
weights_1_2 = 2*np.random.random((hidden_size,1)) - 1
```

```
for iteration in range(60):
    layer_2_error = 0
    for i in range(len(streetlights)):                    # exclusive
        layer_0 = streetlights[i:i+1]
        layer_1 = relu(np.dot(layer_0, weights_0_1))
        layer_2 = np.dot(layer_1, weights_1_2)
        layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]))**2   # gives single value out of arrays
        layer_2_delta = (layer_2 - walk_vs_stop[i:i+1])               # -ve (Pred < true)   Pred - True
        layer_1_delta = layer_2_delta.dot(weights_1_2.T) * relu2deriv(layer_1)
        weights_1_2 += alpha * layer_1.T.dot(layer2_delta)            # gradient or weight_delta
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)           # weight_delta or derivative
    if (iteration % 10 == 9):
        print("Error:" + str(layer_2_error))
```

#function taking multiple arguments because of numpy array and processing it at the same time.

Transpose (1,1) → (1,4) (4,1) delta multiplied to every weight.

Back Propogation

result → (4,1)

(4,1) dot (1,1)

(1,4)

(3,1) ← layer_0.T.dot(layer_1_delta) ⇒ (3,4)

*) Every input of layer_0 gets multiplied with every delta of lay

purpose, as you'll see in a moment.

- Neural networks seek to find direct and indirect correlation b/w an input layer and an output layer, which are determined by the input and output datasets.

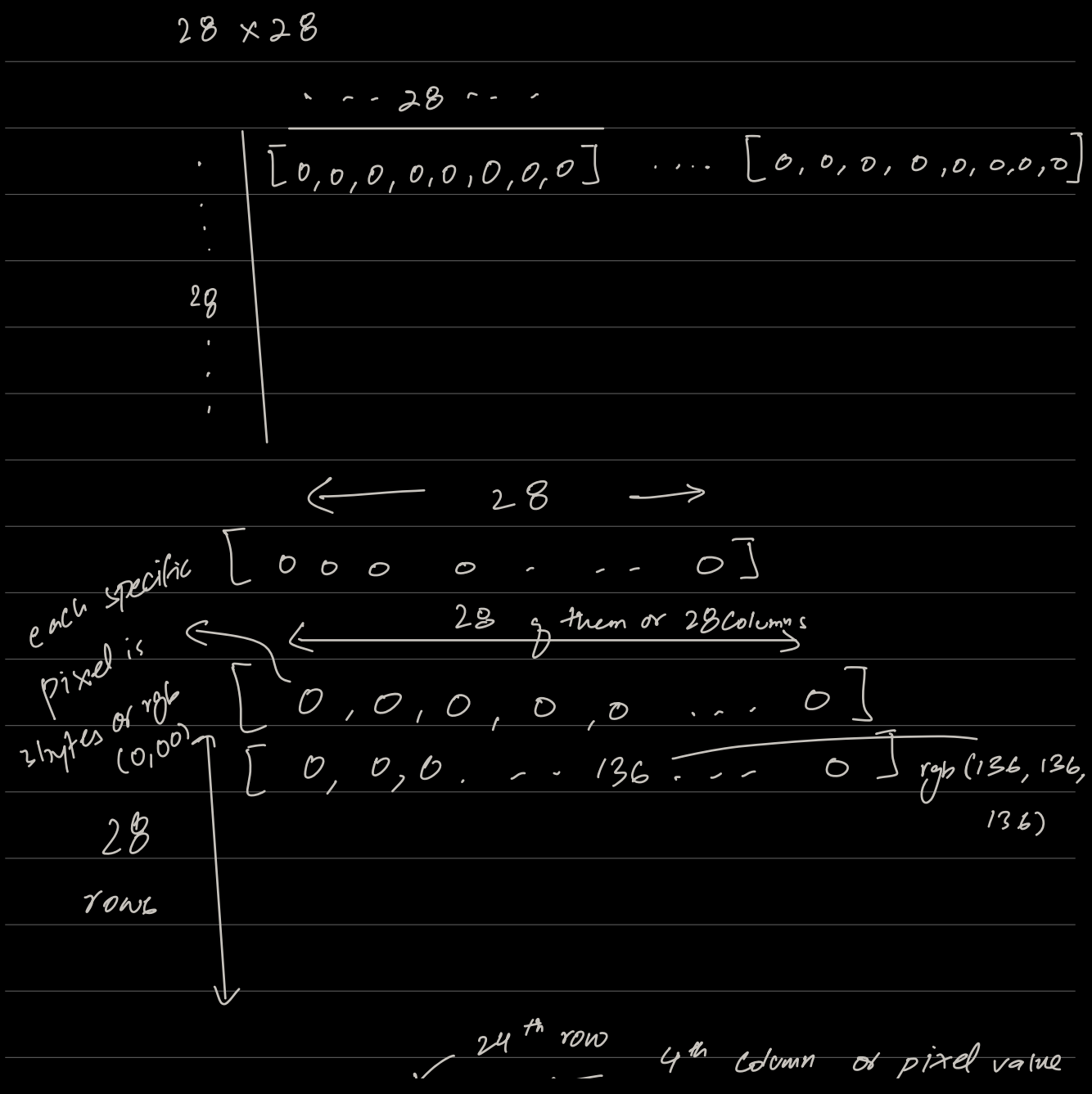
- Local Correlation: Any given set of weights optimizes to learn how to correlate its input layer to with what the output layer says it should be.

- Global Correlation: What an earlier layer says it should be can be determined by taking what a later layer says it should be and multiplying it by weights in b/w them. This way later layers can tell earlier layers what kind of signal they need, to ultimately find the correlation with the output. This cross communication is called back propagation.

- In a neural network lines are weight_matrices and input nodes are like vectors.

*) Good neural architectures channel signal so that correlation is easy to discover. Great architectures also filter noise to help prevent overfitting.
  - ↳ incorrect correlation.
  - ↳ learned to accurately predict but somehow forgot to include a valuable input.

*) The correlation is located wherever the weights were set to high numbers.


28 x 28
28
[0,0,0,0,0,0,0,0] .... [0,0,0,0,0,0,0,0]
28
28
each specific pixel is 3 bytes or rgb (0,0,0)
28 of them or 28 Columns
[0,0,0,0,0 ... 0]
[0,0,0 ... 136 ... 0]
rgb (136, 136, 136)
28 rows
24th row
4th column of pixel value

x_train[0][24][4]

First image

(x_train, y_train), (x_test, y_test)

Training dataset

testing dataset

input pixels

output/result based on specific image.

numpy.reshape(4, 3)

4 rows of 3 columns

x_train[0], y_train[0]

First image

output

1000 images

Columns

(1000, 784)

784 columns or pixels of one image

[ ]

1000 of them or 1000 images

rows

0/255, 136/255 → 0.5333

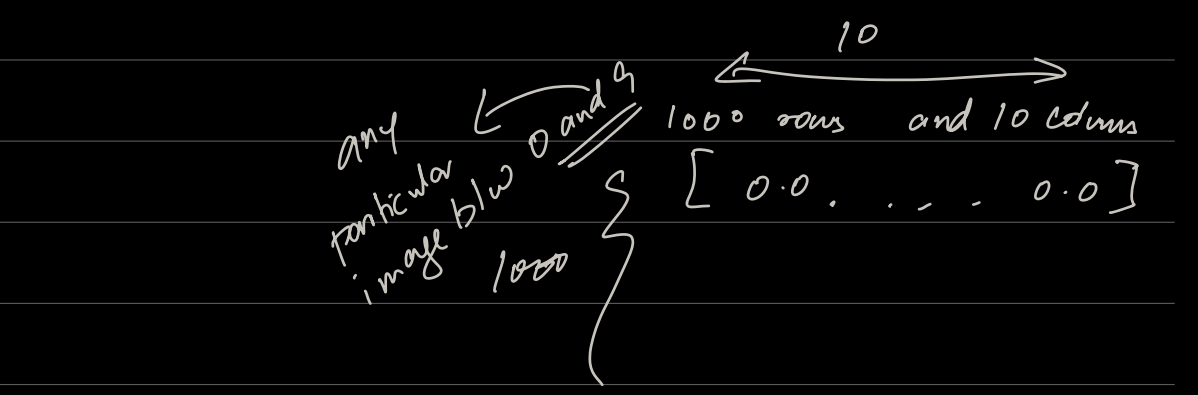
*) every pixel divided by 255.

but the color represented by

each pixel does not change.

*) One_hot_labels — numpy array

np.zeros((len(labels), 10))



Labels = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

for i, l in enumerate(labels): # going through 1000 labels with or indices with each index representing actual output number b/w 0 and 9, set it to 1 so that we know it is True value.

one_hot_labels[i][l] = 1

labels = one_hot_labels

with each index with a value of 1 representing the actual true value/output or what number does the image represents.

784 rows, 40 columns

layer1_ 40 input nodes.

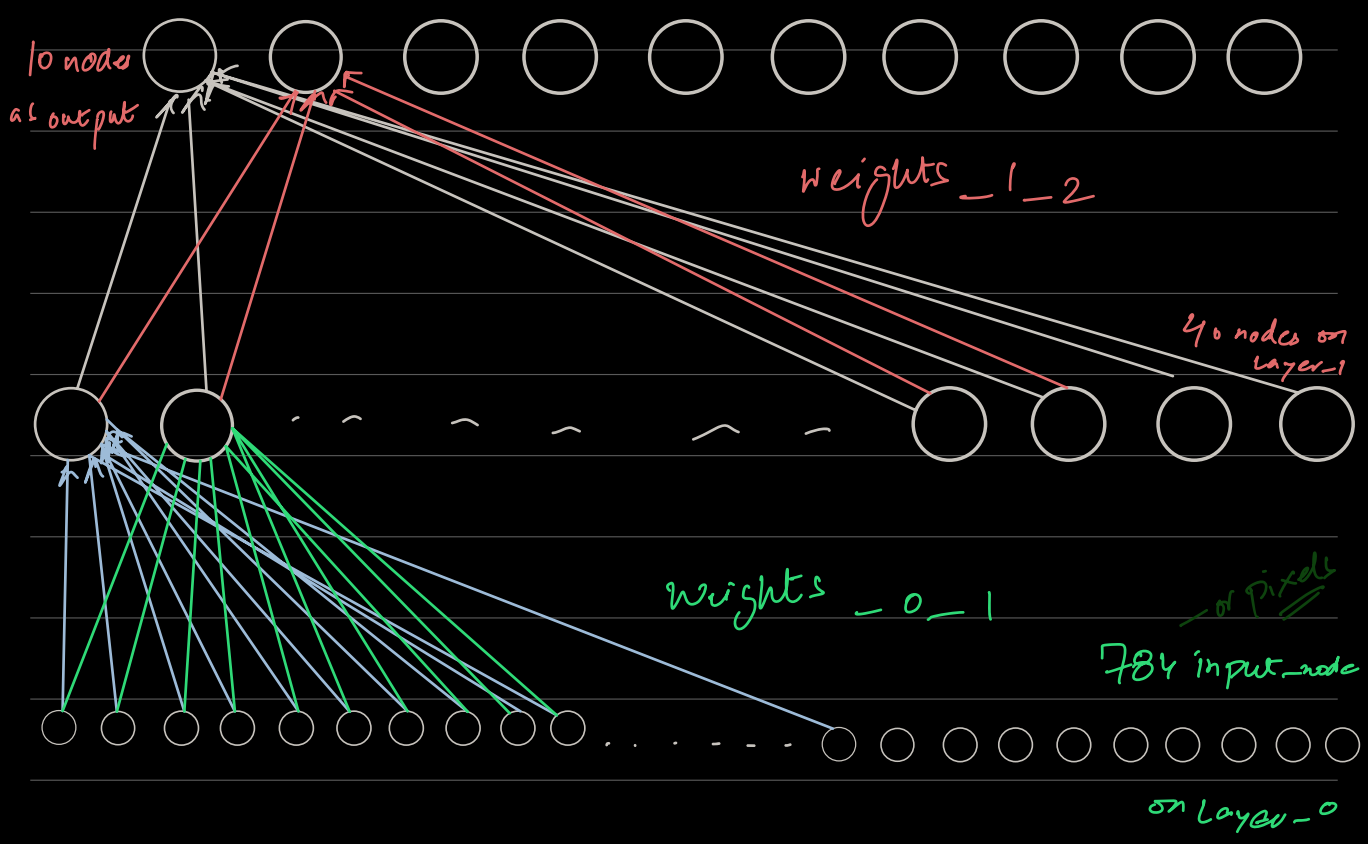
Layer_0 784 input nodes

$[1 \times 784] \; [784 \times 40]$

784 columns

input nodes

1×40



```
import sys, numpy as np
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

images, labels = (x_train[0:1000].reshape(1000,28*28) \
                                        255, y_train[0:1000])
one_hot_labels = np.zeros((len(labels),10))

for i,l in enumerate(labels):
    one_hot_labels[i][l] = 1
labels = one_hot_labels

test_images = x_test.reshape(len(x_test),28*28) / 255
test_labels = np.zeros((len(y_test),10))
for i,l in enumerate(y_test):
    test_labels[i][l] = 1

np.random.seed(1)
relu = lambda x:(x>=0) * x
relu2deriv = lambda x: x>=0
alpha, iterations, hidden_size, pixels_per_image, num_labels = \
                                        (0.005, 350, 40, 784, 10)
weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0, 0)

    for i in range(len(images)):
        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)
        error += np.sum((labels[i:i+1] - layer_2) ** 2)
        correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(labels[i:i+1]))
        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
                                    * relu2deriv(layer_1)
        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

    sys.stdout.write("\r"+ \
                     " I:"+str(j)+ \
                     " Error:" + str(error/float(len(images)))[0:5] +\
                     " Correct:" + str(correct_cnt/float(len(images))))
```

Returns x if x > 0; returns 0 otherwise

Returns 1 for input > 0; returns 0 otherwise

This results in matrix generated with nums from -0.1 to 0.1

generated random matrix with numbers from 0 to 1.

```
....
I:349 Error:0.108 Correct:1.0
```

*) The neural network learns to take a dataset of 1000 images and learn to correlate each input image with the correct label.

70.7 % test accuracy — accuracy of the neural network on the data network was not trained on.

*) One way to view weights of a neural network is as a high-dimensional shape. As you train, this shape molds around the shape of data, learning to distinguish one pattern from another. The images in testing slightly different from patterns in the train dataset.

*) A neural network that overfits has learned the noise in the dataset instead of making decisions based on the true signal.

↳ learned noise instead of making decisions of true signal

*) The mold got worse at the training dataset as you imprinted more forks because it learned more-detailed information about the

training dataset.

*) How to get neural network to train only the "signal" (the essence of dog) and ignore the noise. One way is early stopping.

*) Large amount of Noise comes from fine-grained detail of image

*) Most of the signal (for objects) is found in the general shape and in color of the image.

True Signal ← :) general shape of object
← :) Color of object/image

*) Simplest regularization: Early Stopping:

- Stop training the network when it starts getting worse.
  - ↳ Don't let the network train long enough.
  - ↳ Early stopping - cheapest form of regularization.
  - ↳ Subset of methods that help the neural network to learn the signal and ignore the noise.

Encourage generalization - often by making it difficult for a model to learn the fine-grained details of training data.

*) Industry Standard Regularization: (Dropout)

The method: Randomly turn off neurons (set them to 0) during training.

↳ During training randomly setting neurons in network to 0.

↳ This way neural networks train exclusively using random subsections of the neural network.

↳ Dropout makes a big network act like a little one by randomly training little subsections of the network at a time, and little networks don't overfit.

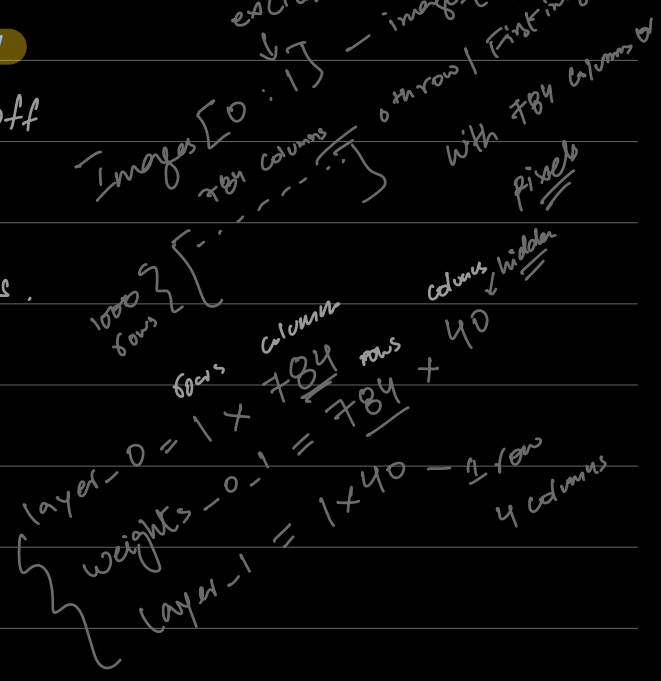
↳ Larger stones can capture more nuanced detail

↳ Form of training a bunch of networks and averaging them.

↳ neural networks learn by trial and error, and that every neural network learns a

little differently

↳ Randomly turning off 50% of the input layer nodes or neurons.



⇒ Modeling probabilities and non-linearities: Activation Functions:

- Activation function
- Standard Activation functions ← sigmoid / Tanh
- Standard output Activation functions — Softmax
- Activation function installation instructions.

- What is an activation function:

A function applied to the neurons in a Layer during prediction. Have used an activation relu in the three-layer neural network. The relu function had the effect of turning all negatives numbers to 0.

*) Constraints for Activation function:

- Constraint 1: The function must be continuous and and infinite in Domain, which means that there should be output number for any input.

- Constraint 2: Good activation functions are monotonic, never changing direction, which means the function should be 1:1, and must be either always increasing or decreasing

- Constraint 3: Good Activation functions are non-Linear (they squiggle or turn).

To Create sometimes Correlation you had to allow the neurons to Selectively correlate to input

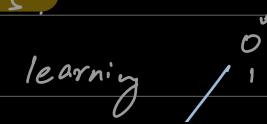
neurons such that a very -ve signal from one input into a neuron could reduce how much it correlated to any input (relu). Facilitated by any function that curves.

- Given a neuron with an activation function, you want one incoming signal to be able to increase or decrease how correlated the neuron is to all the other incoming signals.

Constraint 4: Good Activation functions (and their derivatives) should be efficiently computable.

*) You'll often want to make multiple binary probabilities in one neural network. 98

*) When a neural network has hidden layers, predicting multiple things at once can be beneficial. Often the network will learn something when predicting one label that will be useful to one of the other labels.

eg: learning  = 9

*) Sigmoid Activation function can be of help in this instance because it models individual probabilities separately for each output node.

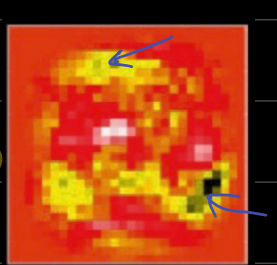
*) The Core Issue: Input have Similarity
- Different Numbers share Characteristics. The Network should know about this.

*) The average 2 shares quite a bit in common with average 3.

*) General rule, Similar inputs create similar outputs.

*) Side effect? Most images share lots of pixels in the middle of images, so the network will start trying to focus on the edges.

*) Consider the 2-detector node weights?
The heaviest weights are the end points of the 2 toward the edge of the image. It is best individual indicator of 2,



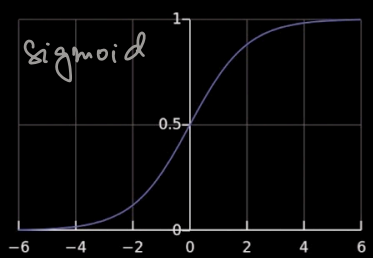
but the best overall is network that sees the entire shape for what it is.

*) Therefore, want an activation function that won't penalize labels that are similar.

*) Activation function that can pay attention to all the information that can be indicative of any potential input.

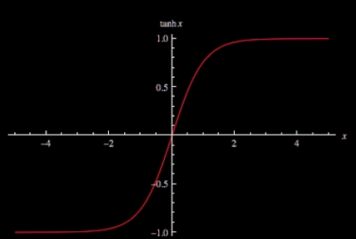
*) softmax Computation:

- Softmax raises each input value exponentially and then divides by the layer's sum.



(Image: Wikipedia)

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

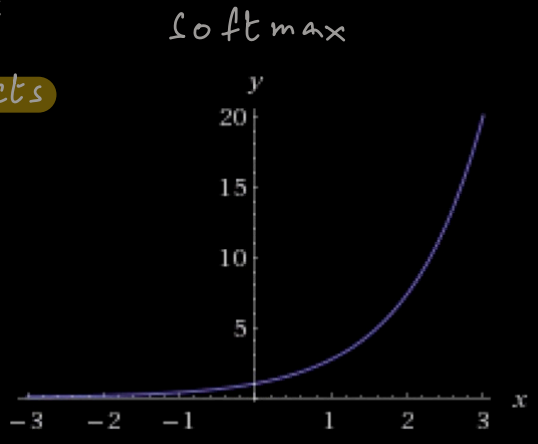


(Image: Wolfram Alpha)

*) For each value x (node's value), compute e^x, and then divide each value by sum of all nodes in the layer.

*) The higher softmax predicts one value, the lower it predicts all the others.

*) It encourages the network to predict one output with very high probability.



*) Adding an Activation function to forward propagation is straight forward. In the case of relu (rectified linear activation unit function), the function is applied to all the input nodes, where negative values are turned to 0's.

*) However, properly compensating for activation function in back propogation is bit nuanced.

*) Wherever, relu had forced a layer_1 value to be 0, we also multiplied the delta by 0.

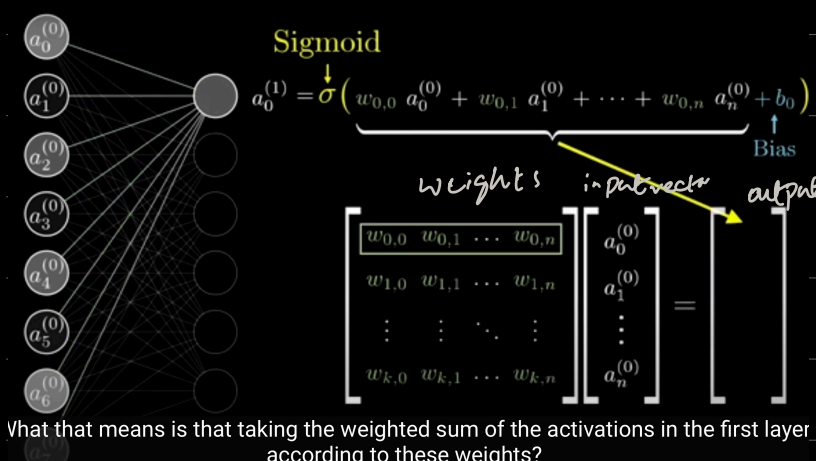
Reason:

Because a Layer_1 value of 0 had no effect on the output prediction, it shouldn't

have an impact on the weight update either. It was not responsible for the error.

*) The purpose of delta at this point is to tell earlier layers "make my input higher or lower next time". It modifies the delta propagated from the following layer to take into account whether this node contributed to error.

*) :: When you back propagate multiply back propagated delta from layer 2 (layer_2_delta . dot (weights_1_2 .T)) by the slope of relu at the point predicted in forward propagation. — Rule applies to other activation function — Sigmoid



What that means is that taking the weighted sum of the activations in the first layer according to these weights?

Following is a small table for the functions you've seen so far, paired with their derivatives. `input` is a NumPy vector (corresponding to the input to a layer). `output` is the prediction of the layer. `deriv` is the derivative of the vector of activation derivatives corresponding to the derivative of the activation at each node. `true` is the vector of true values (typically 1 for the correct label position, 0 everywhere else).

Function	Forward prop	Backprop delta
relu	`ones_and_zeros = (input > 0)` `output = input*ones_and_zeros`	`mask = output > 0` `deriv = output * mask`
sigmoid	`output = 1/(1 + np.exp(-input))`	`deriv = output*(1-output)`
tanh	`output = np.tanh(input)`	`deriv = 1 - (output**2)`
softmax	`temp = np.exp(input)` `output /= np.sum(temp)`	`temp = (output - true)` `output = temp/len(true)`

Note that the `delta` computation for `softmax` is special because it's used only for the last layer. There's a bit more going on (theoretically) than we have time to discuss here. For now, let's install some better activation functions in the MNIST classification network.

⇒ Neural Networks that understand Language:

King − man + Woman == ?

NLP (Natural language processing): Field that is exclusively dedicated to the automated understanding of human language. Deep learning's approach to this field.

matrix of numbers

Raw text → Matrix of # → Supervised Learning → What you want to know

*) Converting text into numbers in such a way that the correlation b/w input and output is most Obvious to the network. Resulting in faster training and better generalization.

*) Need to know what the input/output dataset looks like.

*) Topic classification

- Predicting Whether people post positive or negative reviews.
- IMDB movie reviews dataset is a collection of review → rating pairs (review (text), rating (number))

*) Input reviews are usually few sentences and output ratings are b/w 1 and 5 stars. — Sentiment dataset.

*) Training neural network that can use the input text to make accurate predictions of the output score.

*) Adjusting the range of stars to be b/w 0 and 1 instead of 1 and 5, so that can use binary softmax to model output as probability.

*) Input data is text instead of numbers and it's variable-length text.

*) What about this data will have correlation with output.

*) Several words in this dataset would have a bit of correlation. Words such as terrible and unconvincing have negative correlation with the rating. ∴ As they increase in frequency in any input datapoint (review) ⇒ rating tend to decrease.

*) Words by themselves would have significant correlation with sentiment.

⇒ Capturing word correlation in input data.

- Creating input matrix representing the vocabulary of review.
- Creating a matrix with each row (vector) representing

a review, and each column representing if a review contains particular word in the vocabulary

words in vocabulary
[c_1 c_2 . . .] — review

set — gets rid of duplicate elements.

tokens — list of sets, where every set refers to a review

vocab or words — length — [74074 and a list of words]

word2index = { }

{ 'profession' : index (1) , 'some word' : index (74000) }

input_data set ⇒ every index or indices in rows or reviews represent/refers some word.

⇒ every row/review consists of list or vector of indices which are values for keys or words appearing in that particular review.

target_data set if positive — 1
negative — 0

Topics: (We will use question/Answer dataset available on

1) Introduction to RNNs ✓ (pytorch.org)
2) RNNs ✓
3) Long short-term memory networks (LSTMs)
4) Implementation of RNN and LSTM.
5) Hyper parameters (might be used)
6) Seq2seq (Encoder | Decoder)
7) Actual project Implementation

⇒ Chatbot: RNNs — LSTMs — Implementation — Hyper parameters — seq2seq

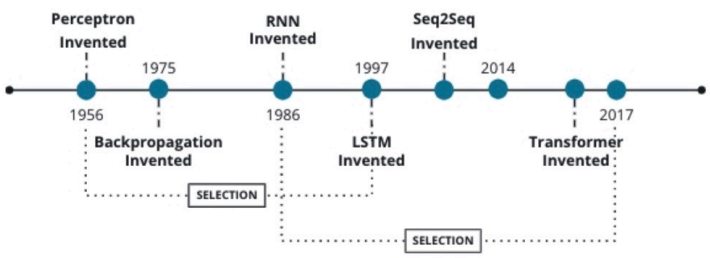
- Implementing different RNN solutions.
- RNNs modification to include a concept of memory.
- Advanced Implementation of LSTM - seq2seq
- Seq2seq uses Encoder/Decoder architecture, which led to transformers, BERT, GPT3.

- Requirements:
  1) Pytorch, Numpy, Python
  2) Build a simple neural network in pytorch.
  3) Load csv into a data from pandas.
  4) Perform matrix multiplication with Numpy.

a) Object Oriented

b) Visualizing data using plots in Jupyter notebook.
   - matplotlib
   - Jupyter notebook.

- Data processing and Object-oriented architecture design patterns.
- Building new architectures for problems.



Historical Timeline of RNNs

This chart shows the history of RNNs. Specifically, it describes how different techniques contributed to its development and how new ideas accelerated improvements through time. Our journey starts with the creation of the Perceptron in 1956 by Frank Rosenblatt. Frank Rosenblatt is considered by many to be the father of artificial intelligence. His creation, the Mark I Perceptron, is currently held in the Smithsonian Institute in Washington, D.C.

After the advent of the Perceptron, it took time before the other cornerstone of modern neural networks, the Backpropagation algorithm, was invented by Paul John Werbos in 1975 as a way to efficiently compute update gradients of the weights of a multi-layer network. David Rumelhart, Geoffrey Hinton, and Ronald Williams popularized this technique for neural networks.

Next, we progress to the RNN, which forms the foundation for this course. The RNN was invented in 1986 using a new technique to model sequential data called recurrence. This architecture was further modified by Jurgen Scmidhuber and Sepp Hochreiter from 1991 to 1997 into the LSTM to solve the vanishing gradient problem.

In 2014, the encoder-decoder architecture called Seq2Seq was introduced by Google to improve performance on sequence-to-sequence tasks. This architecture was carried forward by Google Brain into the advent of the Transformer in 2017. That is a lot of exciting ground to cover!

- RNN:

1) Modeling Complex equations.
   - to predict next value in a time series.
   - model differential equations.
2) Natural Language processing - understanding complexities of word relationships in NLP
   - generate text
   - interactive chatbots.

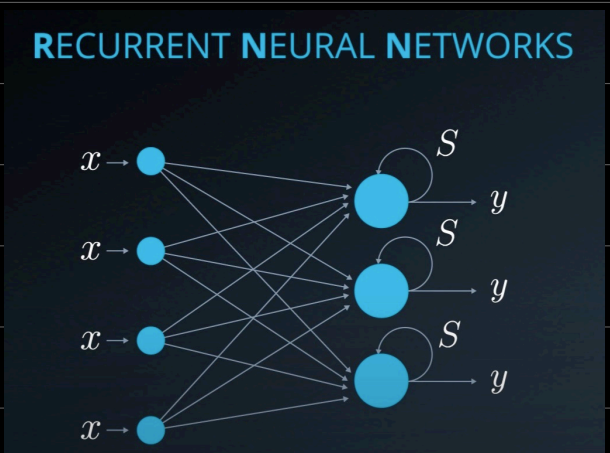
Tools

- Python development environment.
- Pytorch - create neural networks.
- Pandas / Numpy
- Jupyter as IDE

=> Recurrent Neural Network:

1) Recurrence - occuring often
2) Memory - to produce output of the network.
- Back propogation.

=> Our current output does not belong on our current input but past input.



*) They can capture dependencies or dependencies over time

*) Perform same task for each element in the input sequence.

- Simple RNN ELMAN Network.
- Training RNNs
- LSTM

*) RNNs give us a way to incorporate memory in our neural networks and are critical in analyzing sequential data.

*) Associated with text processing and text Generation because of the way sentences are structured as a sequence of words.

*) RNNs and Text Generation:

TV script generator — a model that takes in a series of words as input and outputs a likely next word, forming a text, one word at a time.

*) Similarly, RNNs can be used to generate text given any other data. Giving an RNN a feature vector from an image can be used to generate a descriptive caption.

*) RNNs excel at sequence based tasks.

- Vanishing gradient problem:

In training our network, while using back propogation process, we adjust our weight_matrices using gradient/weight_delta/derivative/slope. In the process, gradients are calculated by continuous multiplication of derivatives. The value of derivatives

may be so small that these continous multiplications may cause the gradient to practically "vanish".

LSTM - overcome vanishing gradient problem.

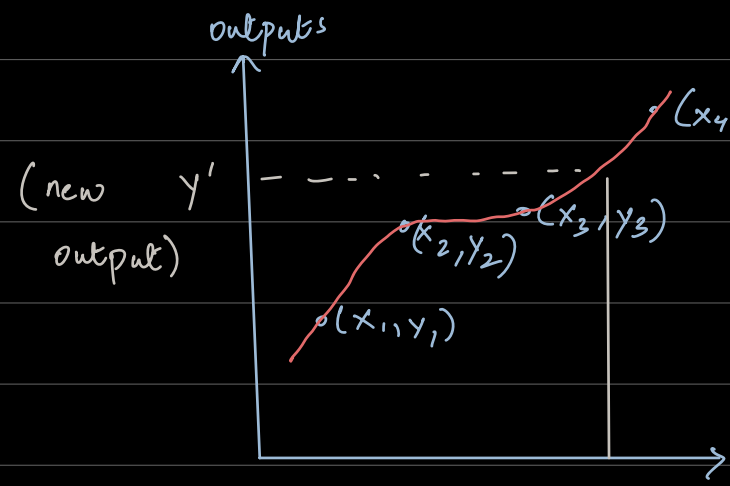
RNN and LSTM

1) Speech recognition
2) Time series
3) movie selection (Netflix)
4) Stock market
5) NLP (chat bots)

⇒ Feed Forward Neural Networks:

- Can have many hidden layers b/w the inputs and the outputs.
- Single hidden layer
- Simulate a artificial neural network by using a non-linear function approximation.

output vector → $\bar{y}$ = $F(\bar{x}, W)$ ← input vector, weights



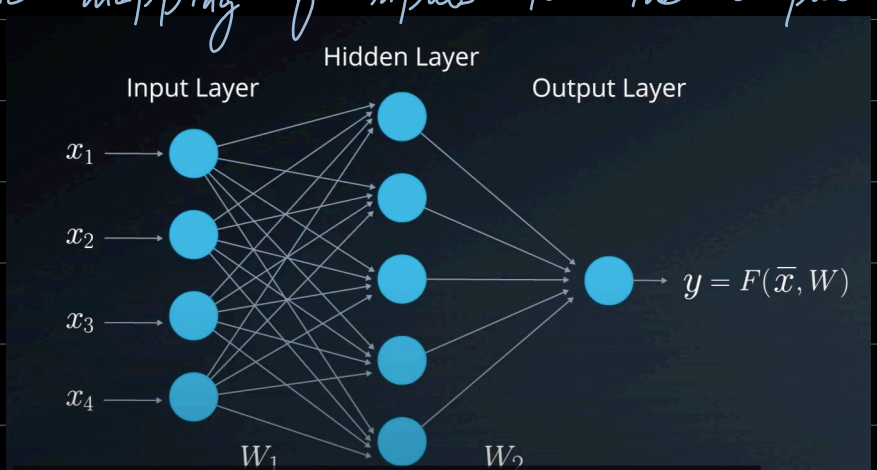
Non-Linear Function

Approximation

- Capturing features or finding correlation of certain pixels with certain features.

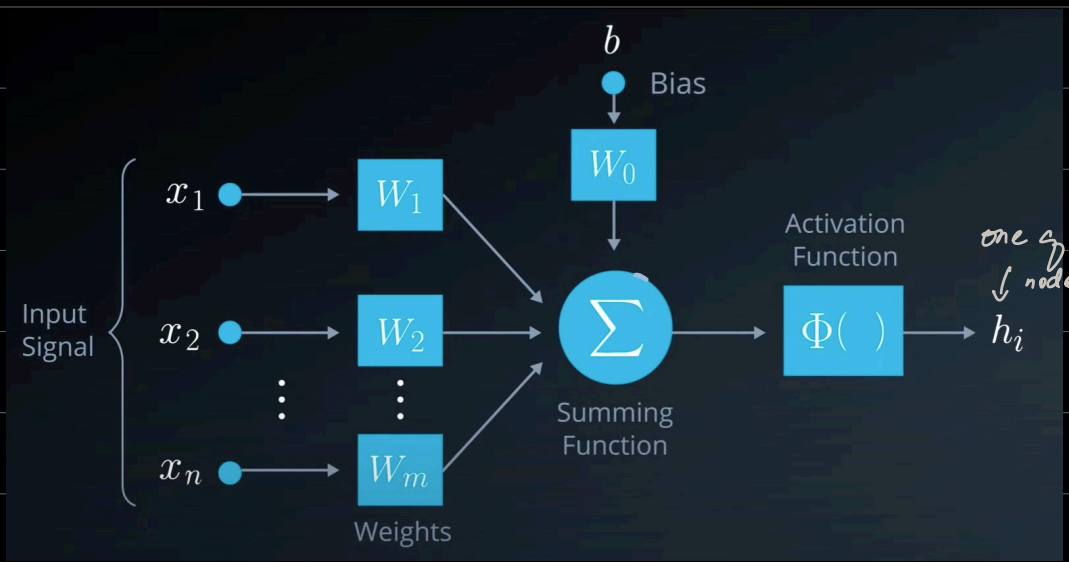
Applications :
- Classification : Image Classification (Pixels)
- Regression : Time-series Forcasting (predicting stock prices)

- Finding best set of weights that yield a good output.
- static mapping of inputs to the output.



- Training: Training set or data, where neural network is trained to generalize beyond the training set or data.

- Evaluation: new inputs ⟶ Desired outputs.



⇒ During the training phase, we take data set (also called training set), which includes many pairs of inputs and their corresponding targets (outputs). Main goal is to find set of weights that would map the inputs to desired outputs. In evaluation phase, we apply our new inputs and expect to obtain the desired outputs.

⇒ Training phase :-

Feedforward and Backpropagation — steps repeated until we decide that our system has reached the best set of weights — error reduced to near zero.

-) W_k — weight matrix k
-) W_{ij}^{k} — is the ij element of weight matrix k.
  - i ↑ row
  - j ↖ column

⇒ Calculate output value of perceptron

$(\text{weights})^T$ - Transpose

$$\begin{bmatrix} 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = 8 + 0 = 8$$

input vector

input

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 8 \end{bmatrix}$$

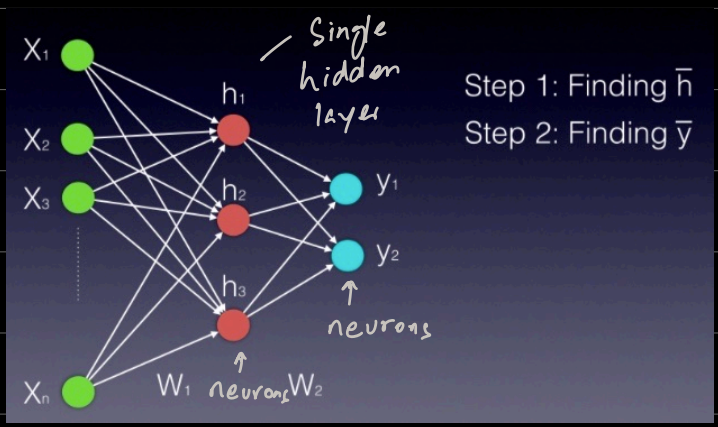
input

weights

Activation Function $\Rightarrow y = \dfrac{1}{(1 + e^{-z})}$

$$y = \frac{1}{(1 + e^{-8})} = 1.0003$$

Activation function model output as probability



⇒ Softmax encourages the network to predict one output with very high probability.

⇒ Soft max is a useful activation function when we want all output values to be b/w 0 and 1, and their sum to be 1.

Error Functions:

⇒ Mean squared Error function = (pred - true) ** 2

⇒ Cross Entropy

$$\bar{x} \qquad\qquad W_1$$

$$= \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ \vdots & \vdots & \vdots \\ w_{n1} & w_{n2} & w_{n3} \end{bmatrix}$$

$$\bar{h}' = \begin{bmatrix} h_1' & h_2' & h_3' \end{bmatrix}$$

← matrix multiplication gives us

Hidden neurons

$$h'_1 = x_1 \cdot w_{11} + x_2 \cdot w_{21} + \dots x_n \cdot w_{n1}$$

$f(x) = \tanh(x)$ – Activation function which makes sure that values of neurons are b/w -1 and 1

$\sigma(x) = \frac{1}{1 + e^{-x}}$ – Sigmoid Activation function makes sure that our values are b/w 0 and 1.

Activation function.

$$\bar{h} = \phi(\bar{h'}) \quad \text{or} \quad \bar{h} = \phi(\bar{x} \cdot W_1)$$

$\bar{h}$ – hidden vector of neurons' input

$\Rightarrow$ Given $\bar{h}$ $\rightarrow$ Find $\bar{y}$ (Apply step 1 again or matrix multiplication)

$$h_1 = \phi\left(\sum_{i}^{n} x_i w_{i1}\right)$$

h_1 – one hidden neuron

$$f(x,y) = 7x^2 - x^3y^4 + 5x^4y^3$$

partial derivative of function with respect to x.

- Every other variable except x - treat as constant

$$f_x = 7(2x) - (3x^2)y^4 + 5(4x^3)y^3$$

$$f_x = \boxed{14x - 3x^2y^4 + 20x^3y^3}$$ first partial derivation.

$$f_y = 0 - x^3(4y^3) + 5x^4(3y^2)$$

$$f_y = -4x^3y^3 + 15x^4y^2$$

$z = f(x,y)$

$\frac{dz}{dx} = f_x$, $\frac{dz}{dy} = f_y$

↓ − 1 ← error

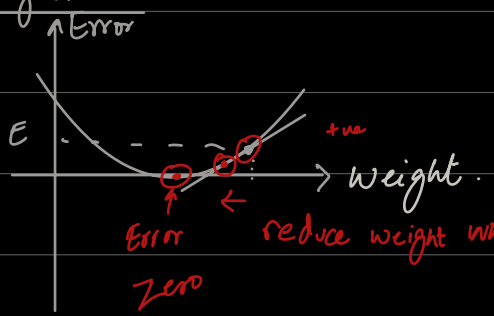
↑ 2 ← weight

Slope \ increase weight

↑ 1 Slope /

↓ 2 reduce weight

=> Back propogation:



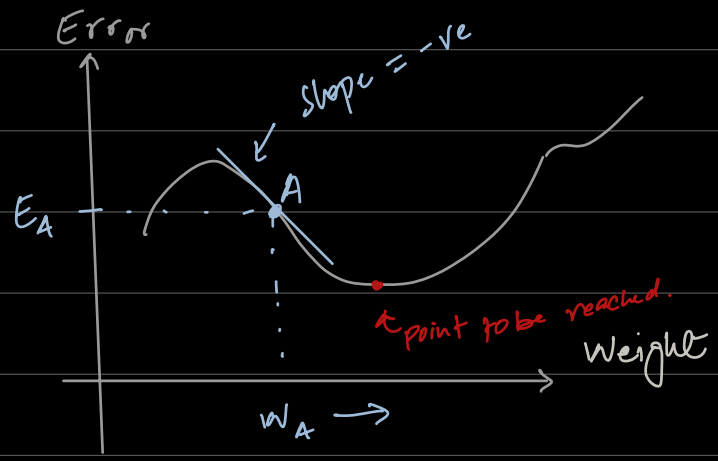
Reduce weight when gradient/slope is positive.

Increase weight when gradient/slope is negative.

- Partial derivative with respect to each weight.
- Error function consists of many weights / variables.

⇒ <u>Back propogation</u>: stochastic gradient Descent with chain rule.

- Finding set of weights that minimizes the network error.



- Assume only dealing with one input.

Since, Slope = -ve we have to change weight in opposite direction ∴ Have to increase weight so that the error goes down (vice versa)

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

(α: step size)

- The weight changes in the opposite direction of the partial derivative of the Error with respect to w.
- Partial derivative lets us measure how the error is impacted by each weight separately.

- Vector of partial-derivatives of the network error each with respect to different weight.

Vector of partial derivatives.

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

weight_deltas

= input × delta

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

(2) $$W_{new} = W_{previous} + \Delta W_{ij}^{k}$$

loss function $\Rightarrow E = \frac{(\bar{d} - \bar{y})^2}{2}$ } To model error.

↑

error function

$\bar{d}$ – desired output

$\bar{y}$ – calculated output.

$$\underset{\text{input}}{\begin{bmatrix} x_1 & x_2 \end{bmatrix}} \quad \underset{\text{weights}}{\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}}$$

$$\begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \underline{y} \text{ or } \underline{\text{output}}$$

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

$$E = \frac{(\text{true} - \text{pred})^2}{2}$$

$$\Delta w_{ij} = -\alpha \left(\frac{\partial E}{\partial w_{ij}} \right)$$

Partial derivative of loss function e with respect to the weight.

constant: d, output/pred: y

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

constant: y

$$F(x) = \frac{(x-y)^2}{2} = g(f(x))$$

$$F_x = \frac{1}{2}(2)(x-y) = x - y = g'(f(x))$$

$$F_y = -\frac{1}{2}(2)(x-y) = y - x$$

with respect to y

$$= +\frac{\alpha}{2} \cdot \left(\frac{\partial (d-y)^2}{\partial w_{ij}} \right)$$

$$E = (d-y)^2$$

$$\frac{dE}{dy} = +2(d-y)$$

$(d-y)$ or delta

$$\frac{dE}{dw} = \frac{dE}{dy} \cdot \frac{dy}{dw}$$

$\frac{dE}{dw} = ?$

$$y = \sum_{i=1} (h_i w_i)$$

$y = h_1 w_1$ ← neuron

$\frac{dy}{dw} = h_1$ * Delta $= \frac{dE}{dw}$

$f(g(x)) = \sin(x^2)$

$f(y) = \sin(y)$ $\quad y = g(x) = x^2$

$f'(y) = \cos y$ $\quad g'(x) = 2x$

$f'(g(x)) \cdot g'(x)$

$f'(y) \cdot g'(x)$

$\cos y \cdot 2x \quad - \frac{a}{2}$

$f(g(a)) = \frac{(d-y)^2}{w_{ij}}$

$f(g(x)) = f'(g(x)) \cdot g'(x)$

pred ↓ true ↓

$C = (a - y)^2$

$\frac{dC}{da} = 2(a-y)$ $\quad f(g(x)) = g'(x) \cdot f'(g(x))$

constant =

$$\frac{dC}{dw} = \underbrace{i}_{\text{input}} \cdot \underbrace{\frac{da}{dw}}_{\text{change in output over change in weight}}^{T} \circ \frac{dC}{da}$$

Change in output
over change in weight

$$a = i \cdot w$$

$$\frac{da}{dw} = i$$

$$E = \frac{(d-y)^2}{2} \quad \left\{ \text{loss function} \right.$$

*) $\Delta w_{ij}^{k} = \alpha \left(- \frac{dE}{dw} \right)$ ← Partial derivative of loss function

$$y = \sum_{i=1}^{2} (h_i w_i)$$

$$\Delta w_{ij}^{k} = \alpha \left(- \frac{dE}{dw} \right) = - \frac{\alpha}{2} \left(\frac{d \overbrace{(d-y)^2}^{E}}{d w_{ij}} \right) \frac{dE}{dw_{ij}}$$

desired / output

$$E = (d-y)^2$$

$$\frac{dE}{dy} = -2(d-y)$$

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy} \cdot \frac{dy}{dw_{ij}} \quad \left\{ \text{Chain rule} \right.$$

$$\frac{dE}{dw_{ij}} = -2(d-y) \cdot \frac{dy}{dw_{ij}}$$

$$\Delta w_{ij}^{k} = -\frac{\alpha}{2}\left(-2(d-y)\cdot\frac{dy}{dw_{ij}}\right)$$

$$\Delta w_{ij}^{k} = \alpha\left[(d-y)\cdot\frac{dy}{dw_{ij}}\right] \quad \text{gradient or } \delta_{ij}$$

$$y = \sum_{i=1}^{3}(h_i w_i) \qquad \delta_{ij} = \frac{dy}{dw_{ij}}$$

delta * weight = node_delta

node_delta * input(xi) = weight_delta

$$\Delta w_{ij}^{k} = \alpha\,(d-y)\,\frac{dy}{dw_{ij}}$$ (gradient)

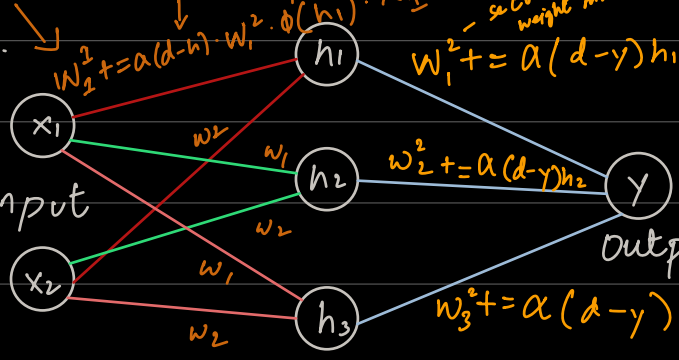
One instance of weight update for layer 2.

delta

$W_2^2 \mathrel{+}= \alpha(d-y)\cdot w_1^2\cdot\phi'(h_1)\cdot x_1$

second weight matrix

$W_1^2 \mathrel{+}= \alpha(d-y)h_1$



These weight updates carried out in the end or after layer 2 update.

output

$$y = \sum_{i=1}^{3} (h_i w_i)$$

$$x_1 w_1 + x_2 w_2$$

W^2 – 2nd weight matrix

W^1 – 1st weight matrix

$$h_1 = \sum_{i=1}^{2} (x_i w_i)$$

$$h_1 = x_1 w_1 + x_2 w_2$$

$$h_2 = x_1 w_3 + x_2 w_4$$

$$h_3 = x_1 w_5 + x_2 w_6$$

$$h_1 = \phi(x_1 w_1 + x_2 w_2)$$

function of function

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_1 & w_3 & w_5 \\ w_2 & w_4 & w_6 \end{bmatrix} \quad W^1$$

Activation function applied.

$$h_j = \phi_j \left(\sum_{i=1}^{2} x_i w_i^1 \right)$$

$j \to 3$

hidden function

$$W^2 = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$$

$$h_1 = x_1 w_1 + x_2 w_2$$

$$h_2 = x_1 w_1 + x_2 w_2$$

$$h_3 = x_1 w_1 + x_2 w_2$$

$$y = \sum_{i=1}^{3} (h_i w_i^2)$$

$$y = hw$$

$$\frac{dy}{dw} = h$$

hidden neurons vector.

$$\frac{dy}{dw_{ij}^{2}} = h_i$$

Partial derivative with respect to w is everything else treated as constants.

⇒ Step 2 of Backpropogation:

$$\boxed{\Delta w_{ij} = \alpha (d-y) h_i}$$

{ First weight update (Backwards)

⇒ step 2 of Back propogation.

Activation

$$h_j = \phi_j \left(\sum_{i=1}^{2} x_i w_i^1 \right)$$

chain rule.

gradient

$$\frac{dy}{dw_{ij}^{1}} = \frac{dy}{dh} \cdot \frac{dh}{dw_{ij}^{1}}$$

$$y \cdot x_i \cdot \phi_j$$

$$\frac{dh}{dw_{ij}^{1}}$$

$$\Delta w_{ij}^{1} = \alpha (d-y) \frac{dy}{w_{ij}^{1}}$$

$$\boxed{w'_{new} = w'_{prev} + \alpha (d-y) \cdot w_j^2 \cdot \phi'_j \cdot x_i}$$

$$y = \sum_{i=1}^{3} (h_i w_i^2)$$

$$\frac{dy}{dh} = w_i^2 \quad \# \text{ weight - 2 matrix or vector.}$$

↑ Partial derivative with respect to h.

Activation - second function. — $f(g(x)) \Rightarrow$ derivation $= f'(g(x)) \cdot g'(x)$

$$h_j = \phi_j \left(\sum_{i=1}^{2} \overbrace{x_i w_i}^{\text{one function}} \right)$$

$$h = x w$$

$$\frac{dh}{dw} = x$$

$$\frac{dh_j}{dw_{ij}^2} = \phi_j' \circ \left(\overbrace{\sum_{i=2}^{2} (x_i w_i)}^{\text{input passed to the activation function derivative. — a neuron}} \right) \circ x_i$$

NB:

∴ Each neuron j in the hidden layer will have its own value for ϕ — activation function and

for ϕ' - activation function derivative.

$$\therefore \frac{dh_j}{dw_{ij}^1} = \phi'_j \cdot x_i$$

(x_i: input vector; ϕ'_j: derivative of activation function.)

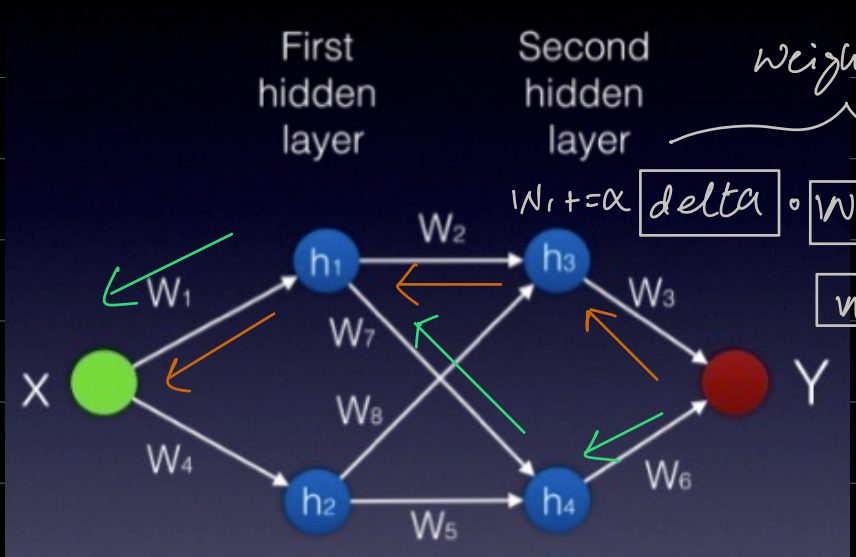
gradient

$$\therefore \frac{dy}{dw_{ij}^1} = \frac{dy}{dh_j} \cdot \frac{dh_j}{dw_{ij}^1} = w_j^2 \cdot \phi'_j \cdot x_i$$

weight update

$$\therefore \Delta w_{ij}^1 = \alpha (d - y) \cdot w_j^2 \cdot \phi'_j \cdot x_i$$

$$\therefore w'_{new} = w'_{prev} + \alpha (d - y) \cdot w_j^2 \cdot \phi'_j \cdot x_i$$



weight_delta for w_1

$w_1 += \alpha$ [delta] $\cdot$ [$w_3 \cdot w_2 \cdot x$ + $w_6 \cdot w_7 \cdot x$]

Weight update rule of weight matrix W_2

or partial derivative of y with respect to W_2

$$y = h_3 w_3$$

$$y = h_3 w_3 + h_4 w_6$$

$$\boxed{\frac{dy}{h_4} = w_6} \qquad \boxed{\frac{dy}{h_3} = w_3}$$

$$h_4 = h_1 w_7 + h_2 w_5 \qquad h_3 = h_1 w_2 + h_2 w_8$$

$$\boxed{\frac{dh_4}{dh_1} = w_7} \qquad \boxed{\frac{dh_3}{dh_1} = w_2}$$

$$h_1 = x w_1$$

$$h_2 = x w_4$$

$$\boxed{\frac{dh_1}{dw_1} = x}$$

$$\frac{dy}{dw_1} = \left(\left[\frac{dh_1}{dw_1} \cdot \frac{dh_3}{dh_1} \cdot \frac{dy}{dh_3} \right] + \left[\frac{dh_1}{dw_1} \cdot \frac{dh_4}{dh_1} \cdot \frac{dy}{dh_4} \right] \right)$$

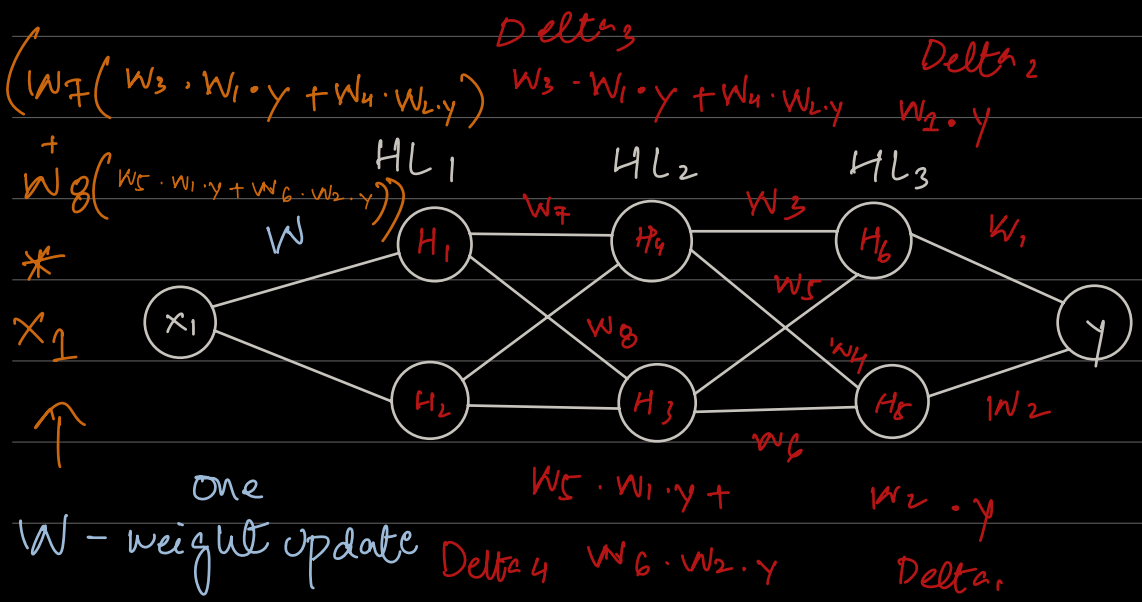
W_1 - Weight update

$$W_1 += \alpha(d-y) * \frac{dy}{dW_1}$$

gradient.

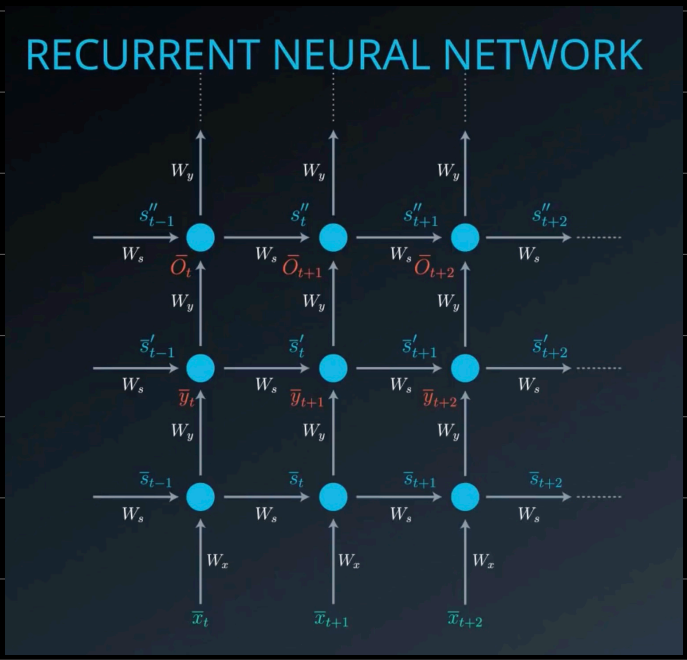
α alpha, $(d-y)$ delta

$$W_1 += \alpha(d-y)\left[(W_3 \circ W_2 \circ x) + (W_6 \cdot W_7 \cdot x)\right]$$



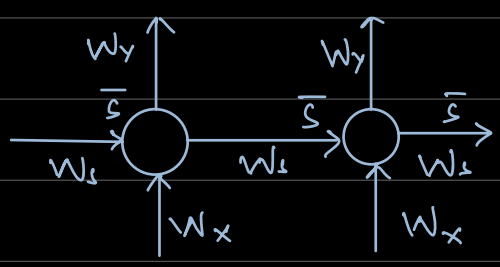
⇒ Recurrent Neural Network :

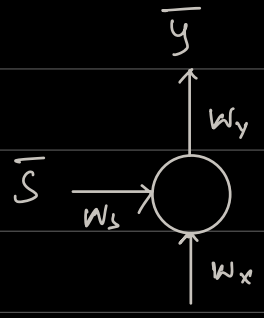
Recurrent - occuring repeatedly



- Perform same task for each element in the input sequence
- Capturing information in previous inputs by maintaining internal memory elements also known as states.

Memory Elements



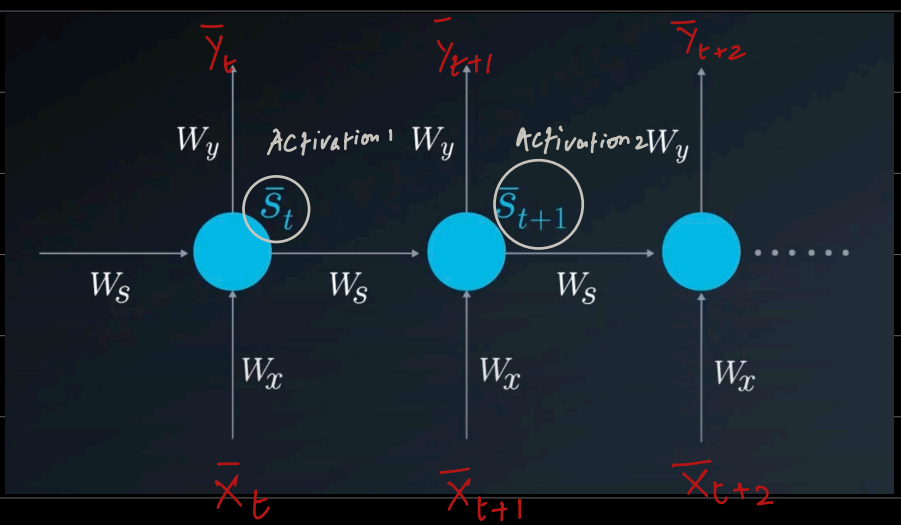


$\bar{x}_t$ $\bar{x}_{t+1}$ $\bar{x}$

*) Temporal dependencies: Current output does not only depend on the current input, but also on a memory element, which takes into account the past inputs — use RNN

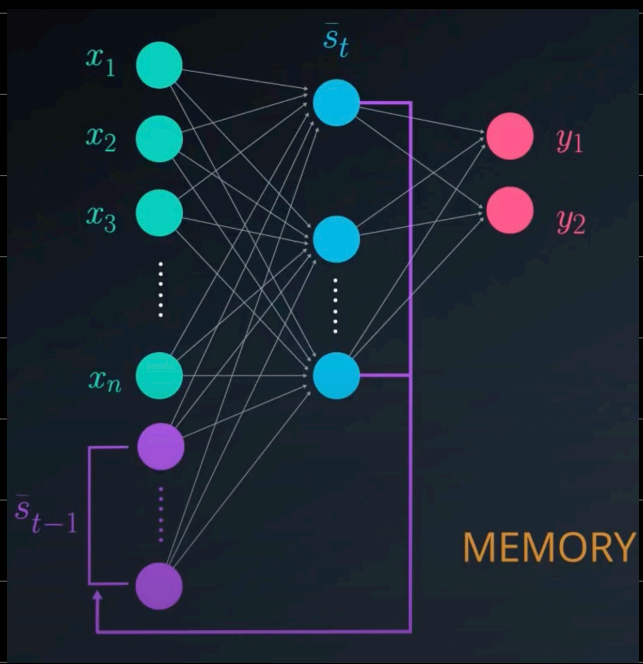
*) A good use of RNN — is predicting the next word in a sentence, which requires looking at the last few words rather than the current one only

Ex: To be or not to _



i) We train with sequences because previous inputs matter.

2) The second difference, stems from memory elements that RNN's host. Current inputs, as well as activations of neurons serve as inputs to the next time step.



*) The Feed forward scheme changes and includes feedback or memory elements — memory as output of the hidden layer, which will serve as additional input to the network at the following training step. This way the system learns to optimize the weight matrix.

RNN uses:

- Sequences as inputs in the training process.
- memory elements.

- FFNN: The output at any time t is a function of the current input and the weights alone.

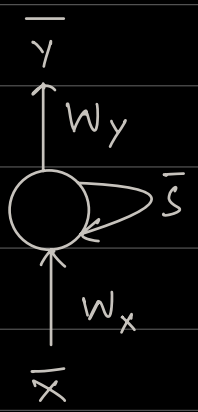
Time t

$$\bar{h}_t = F(\bar{x}_t, W_1)$$

- RNN: The output at time t does not only depend on current inputs and weights, but also on previous inputs.

Time t

$$\bar{y}_t = F(\bar{x}_t, \bar{x}_{t-1} \ldots \bar{x}_{t-n}, W)$$

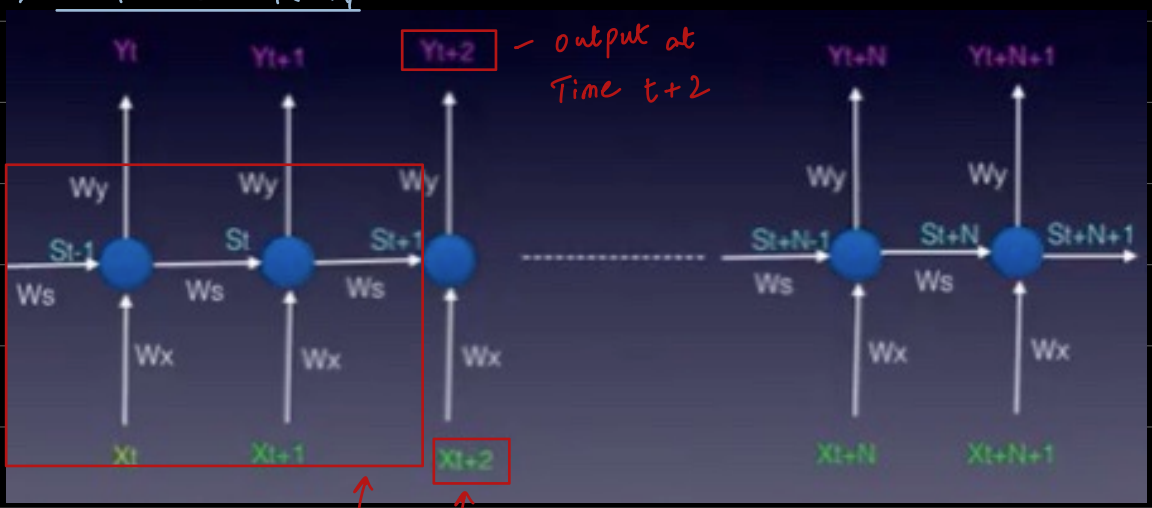


- W_x represents weight matrix connecting the inputs to the state layer.
- W_y represents weight matrix connecting the

State to output.

- Ws represents weight matrix connecting the the state from previous time step to the state in the next time step.

⇒ Unfolded RNN:



- Trained on sequences as inputs.
- Memory elements.

FFNN

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

↑ Activation

RNN

$$\bar{s}_t = \Phi(\bar{x}_t W_x + \overbrace{\bar{s}_{t-1}}^{\text{Previous state}} \cdot W_s)$$

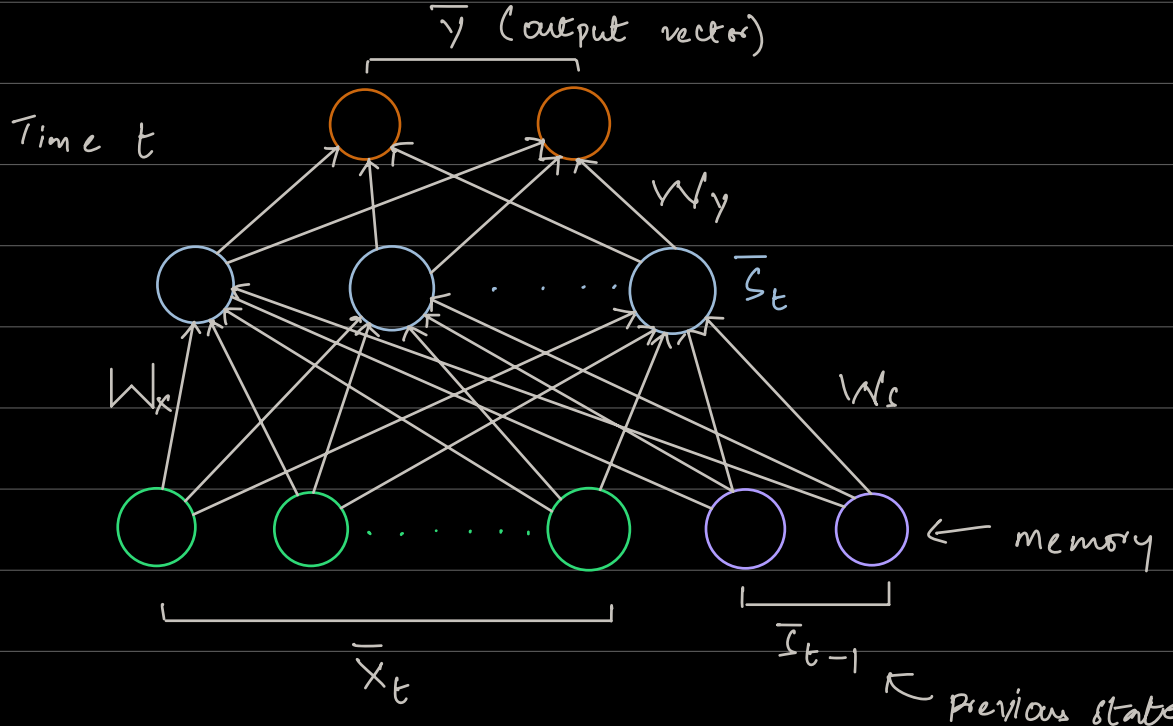
Weight matrix

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

$\bar{y}_t$: output or output vector.

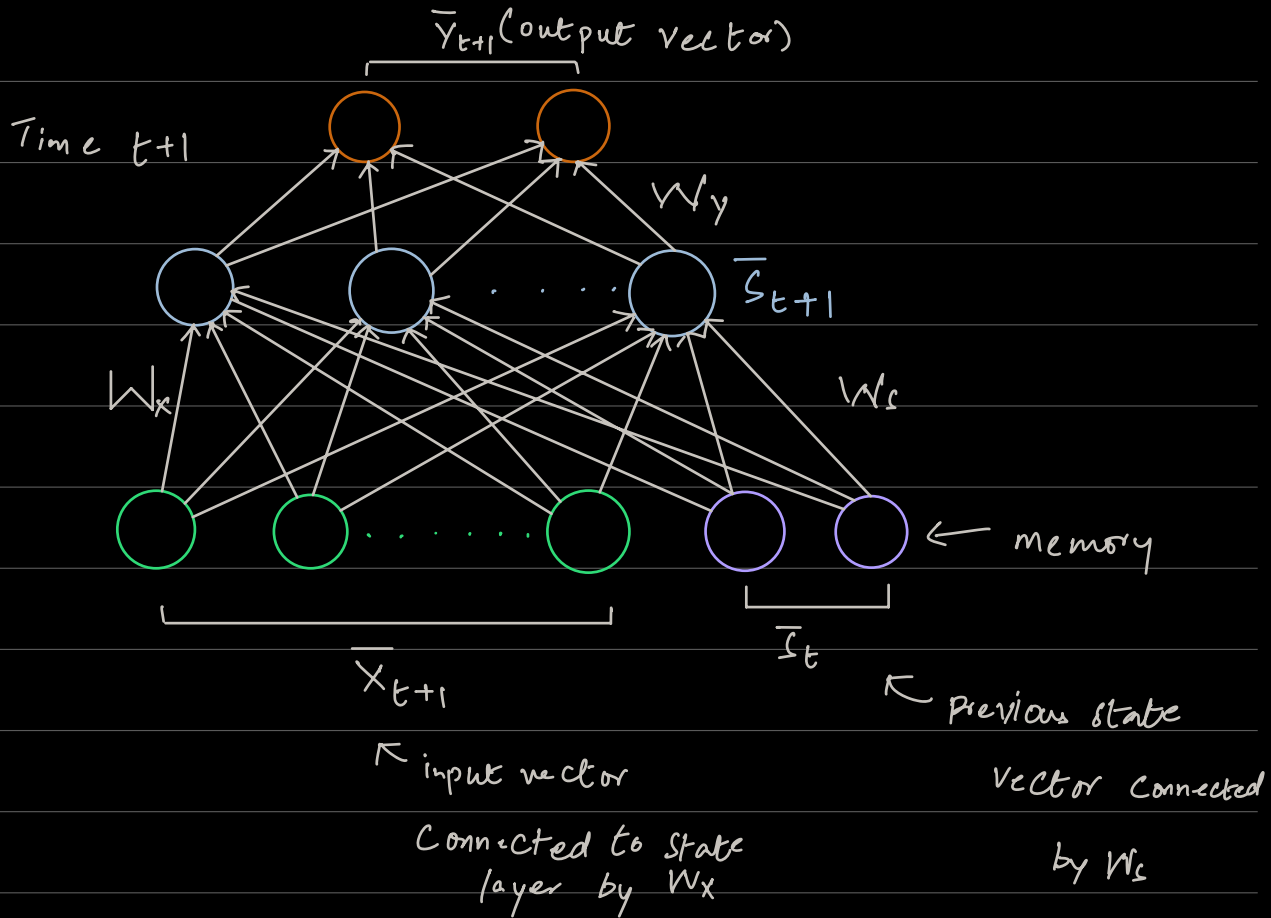
$\bar{s}_t$: state input vector

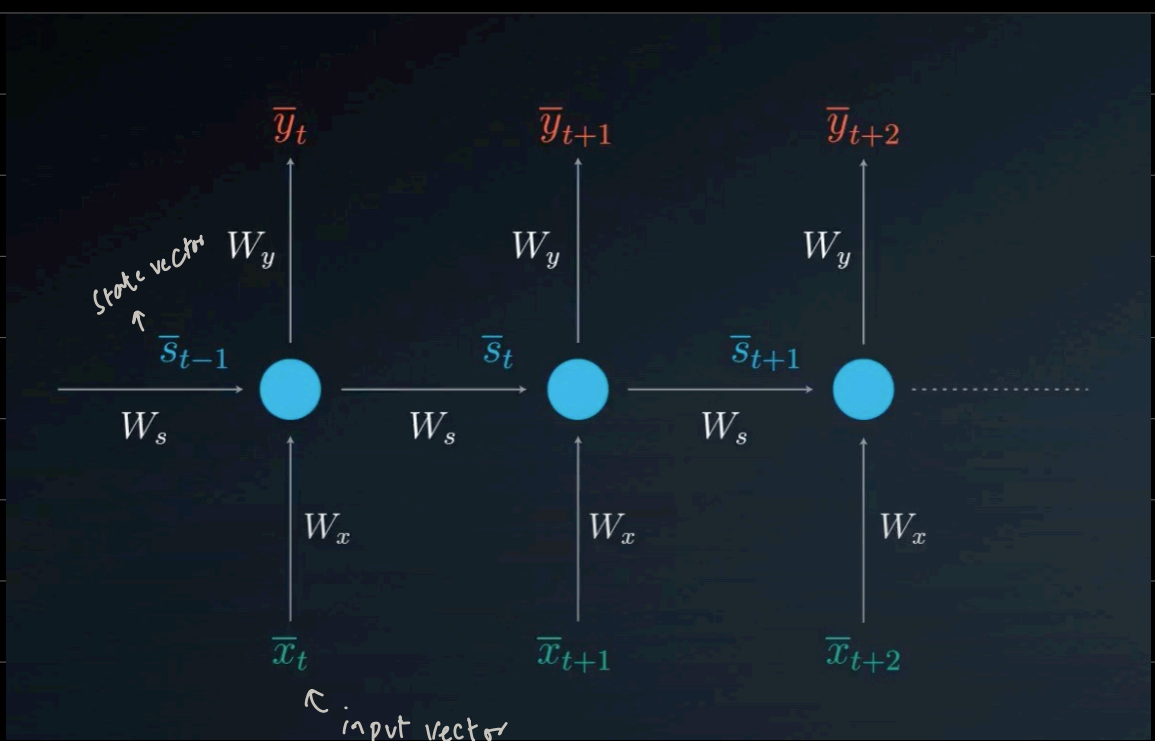
⇒ Elman Network:



$\bar{x}_t$: input vector connected to state layer by W_x

$\bar{s}_{t-1}$: Previous state vector connected by W_s


$\bar{Y}_{t+1}$ (output vector)
Time $t+1$
W_y
$\bar{S}_{t+1}$
W_x
W_s
memory
$\bar{S}_t$
$\bar{X}_{t+1}$
Previous state vector connected by W_s
input vector
Connected to state layer by W_x

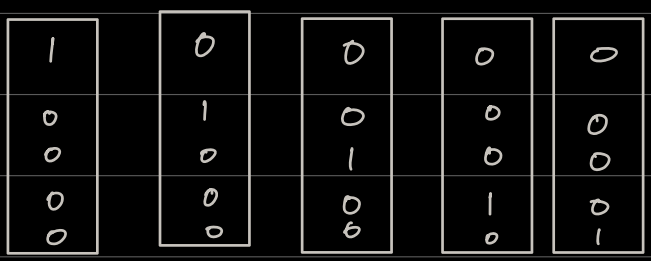

$\bar{y}_t$
$\bar{y}_{t+1}$
$\bar{y}_{t+2}$
W_y
W_y
W_y
state vector
$\bar{s}_{t-1}$
$\bar{s}_t$
$\bar{s}_{t+1}$
W_s
W_s
W_s
W_x
W_x
W_x
$\bar{x}_t$
$\bar{x}_{t+1}$
$\bar{x}_{t+2}$
input vector

- RNN example :

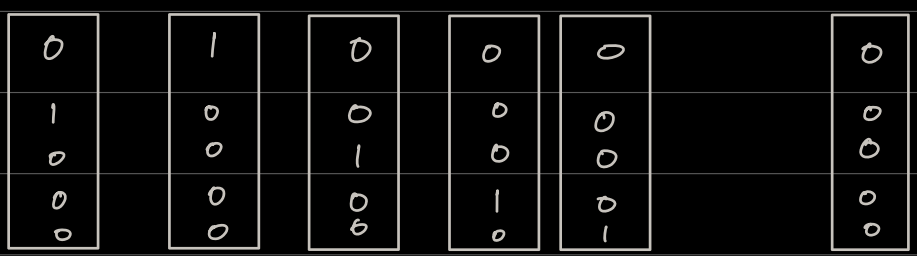
·) Sequence / Word detector :

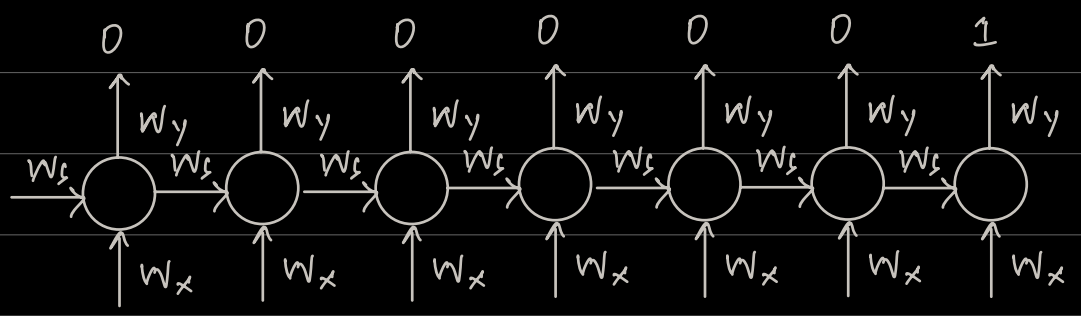
Detecting word "happy"

Arranging letters of Word "happy" in ascending order and creating one-hot vector encoding to be fed as input to the neural network.



Sequence detected Configuration:



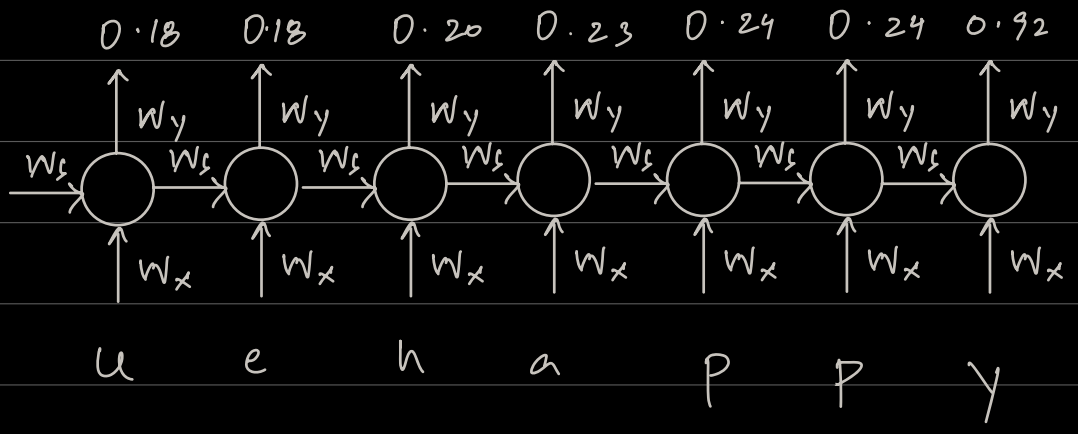


Training:

*) Will keep feeding the network random letters and setting target value to 0 if sequence / word not detected.

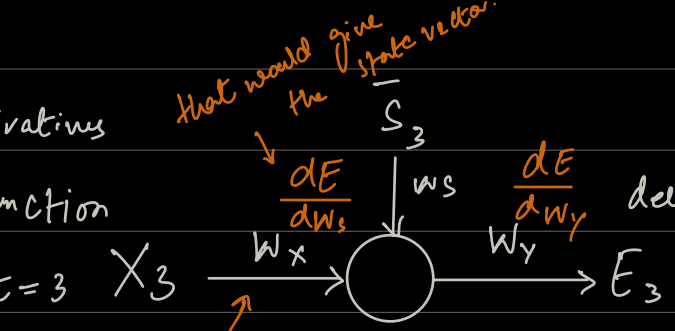
*) When sequence / word detected set target to 1 as occassionally will feed word "happy".

Evaluation phase:



When threshold or probability > 0·9 — sequence is detected

⇒ Back propagation through time:



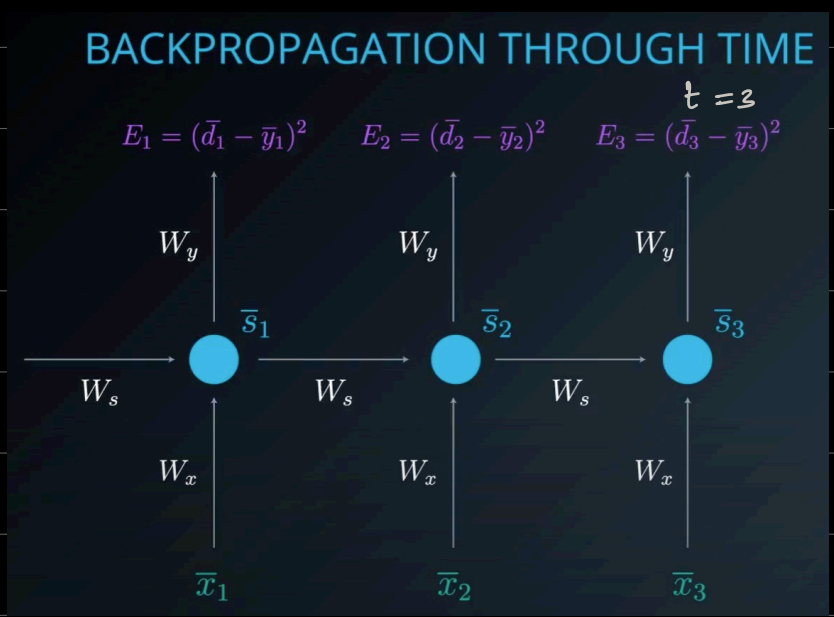
- Find derivatives of loss function at time $t=3$ as a function of all of the weight matrices. (FFNN), but have to consider previous time steps.

delta $(d-y) \cdot \alpha \cdot$ state update for W_y

$E = \bar{s} \cdot W_y$

$\frac{dE}{dW_y} = \bar{s}$

$\frac{dE}{dW_x} = ?, \quad \frac{dE}{dW_s} = ?$



- In this model we have to modify three weight matrices :

  W_x – linking the network input to the state or hidden layer

  W_s – Connecting one state to the next.

  W_y – Connecting this state to the output.

– Adjusting Weight Matrix W_y :

$$E_3 = (\underbrace{d_3}_{\text{desired}} - \underbrace{y_3}_{\text{output}})^2 \qquad y_3 = \bar{S}_3 \cdot W_y$$

$$\frac{dE_3}{W_y} = \frac{dE}{d\bar{y}_3} \cdot \frac{d_{y_3}}{dW_y} \qquad \frac{d_{y_3}}{dW_y} = \bar{S}_3$$

$(d-y)$ = delta

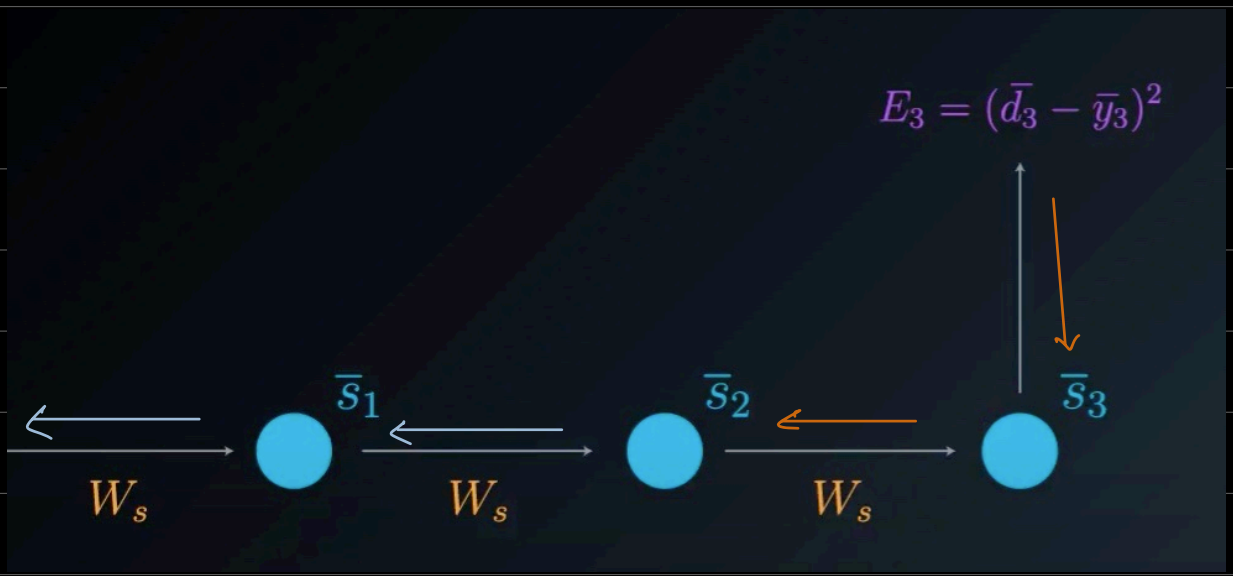
$$W_y = W_y + \alpha (d-y) \cdot \frac{dy_3}{dW_y}$$

(α = alpha; $\bar{S}_3$)

– Adjusting Weight matrix – W_s :

- Weight matrix connecting one state to the next and removing anything that we don't need

for now.



$s_3 = s_2 \cdot w_s \qquad s_2 = s_1 \cdot w_s$

at time $t = 3$

$\frac{d\bar{s}_3}{d\bar{s}_2} = w_s \qquad \frac{ds}{dw_s} = s_1$

$$\frac{dE}{dw_s} = \frac{dE}{dy_3} \cdot \frac{dy}{d\bar{s}_3} \cdot \frac{d\bar{s}_3}{dw_s} \quad \left.\right\} \text{Till } \bar{s}_2$$

$$+$$

$$\frac{dE}{d\bar{y}_3} \cdot \frac{d\bar{y}}{d\bar{s}_3} \cdot \frac{d\bar{s}_3}{d\bar{s}_2} \cdot \frac{d\bar{s}_2}{d\bar{w}_s} \quad \left.\right\} \text{Till } \bar{s}_1$$

$$+$$

$$\frac{dE}{d\bar{y}_3} \cdot \frac{d\bar{y}}{d\bar{s}_3} \cdot \frac{d\bar{s}_3}{d\bar{s}_2} \cdot \frac{d\bar{s}_2}{d\bar{s}_1} \cdot \frac{d\bar{s}_1}{d\bar{w}_s} \quad \left.\right\} \text{Till } \bar{s}$$

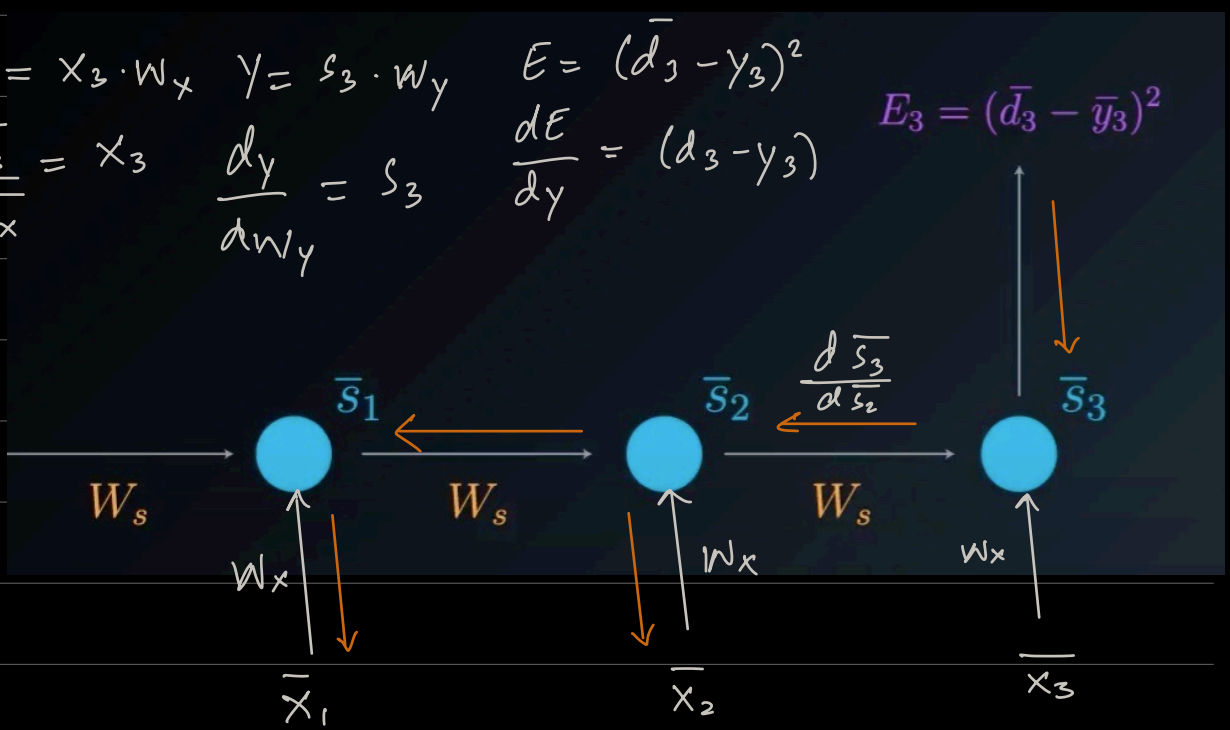
Weight update of W_s

$$\frac{dE_N}{dW_s} = \sum_{i=1}^{n} \frac{dE_N}{d\bar{y}_N} \cdot \frac{d\bar{y}_N}{d\bar{s}_i} \cdot \frac{d s_i}{d W_s}$$

— Adjusting W_x:

$\bar{s}_3 = x_3 \cdot W_x \quad y = s_3 \cdot W_y \quad E = (\bar{d}_3 - y_3)^2$

$\frac{d\bar{s}_3}{d W_x} = x_3 \quad \frac{dy}{dW_y} = s_3 \quad \frac{dE}{dy} = (d_3 - y_3)$



at time $t = 3$:

$$\frac{dE}{dW_x} = \frac{dE}{d\bar{y}_3} \cdot \frac{d\bar{y}_3}{d\bar{s}_3} \cdot \frac{d\bar{s}_3}{dW_x} \cdot \bar{x}_3$$

+

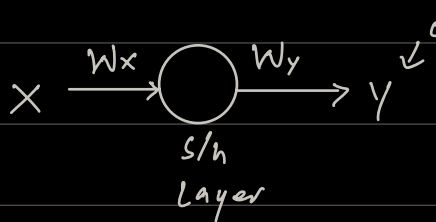
$$\frac{dE}{d\bar{y}_3} \cdot \frac{d\bar{y}_3}{d\bar{s}_3} \cdot \frac{d\bar{s}_3}{d\bar{c}_2} \cdot \frac{d\bar{s}_2}{dw_x}$$

$$\frac{dE_N}{dw_x} = \sum_{i=1}^{n} \frac{dE_n}{d\bar{y}_N} \cdot \frac{d\bar{y}_N}{d\bar{s}_i} \cdot \frac{d\bar{s}_i}{dw_x}$$

+

$$\frac{dE}{d\bar{y}_3} \cdot \frac{d\bar{y}_3}{d\bar{s}_3} \cdot \frac{d\bar{s}_3}{d\bar{c}_2} \cdot \frac{d\bar{s}_2}{d\bar{s}_1} \cdot \frac{d\bar{s}_1}{dw_x}$$

*) Simple Back propogation :



$$y = h \cdot W_y \qquad h = x \cdot w_x$$

Error function

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

$$\frac{dE}{dy} = -(d-y)$$

update to weight matrix Y.

$$\frac{dE}{dw_y} = \frac{dE}{dy} \cdot \frac{dy}{dw_y}$$

$$= -(d-y) \cdot h$$

$$W_y \mathrel{+}= -\alpha (d-y) \cdot h$$

$$\frac{dE}{dW_x} = \frac{dE}{dy} \circ \frac{dy}{dh} \circ \frac{dh}{dW_x} = -(d-y) \cdot W_y \cdot x$$

Chain rule

update to weight matrix W_x

$$W_x \mathrel{+}= -\alpha (d-y) \cdot W_y \cdot x$$

Gradient Calculations

$$\frac{\partial E_N}{\partial W_y} = \frac{\partial E_N}{\partial \overline{y}_N} \cdot \frac{\partial \overline{y}_N}{\partial W_y}$$

$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \overline{y}_N} \cdot \frac{\partial \overline{y}_N}{\partial \overline{S}_i} \cdot \frac{\partial \overline{S}_i}{\partial W_s}$$

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \overline{y}_N} \cdot \frac{\partial \overline{y}_N}{\partial \overline{S}_i} \cdot \frac{\partial \overline{S}_i}{\partial W_x}$$