DEEP LEARNING

[Binary Classification]

Do we know it's a algorithm for <u>binary classification</u>, is or is not ($y=1$ or $y=0$).

We might have an input of an image and determine if something is there; an image is stored by 3 separate matrices one for each color channel, ie $R \in \mathbb{R}^{n \times n}$, $G \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times n}$. Then we extend the matrices into a long vector (unroll), putting all values in a column. For example if the image is 64×64 pixels, it's real dimension is $64 \times 64 \times 3 = 12288$ values, unrolled make a vector of 12288 values.

Notation to be used during the course:

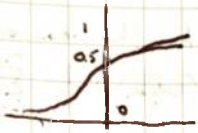
- Single training example: represented by a pair (x, y) where $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
- Training sets: m training examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(m)}, y^{(m)})$
  - m_{train} m_{test}
- The training set inputs $X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix}$ (n_x rows, m columns) $\therefore X \in \mathbb{R}^{n_x \times m}$ & easier than transposing
- For output labels Y: stack them in columns $Y = [y^{(1)}, y^{(2)}, y^{(3)}, \ldots y^{(m)}]$, $Y \in \mathbb{R}^{1 \times m}$

[Logistic Regression]

Learning algorithm for classification, Given 'x', output the prediction $\hat{y}$ what is the probability that $\hat{y}$ equals one with input x, $\hat{y} = P(y=1|x)$; $x \in \mathbb{R}^{n_x}$, $0 \le \hat{y} \le 1$

Parameters $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output $\hat{y} = \sigma(w^T x + b)$ — σ = sigmoid



$\sigma(z) = \frac{1}{1+e^{-z}}$

if z is large, $\sigma(z) \approx 1$

if z is small, $\sigma(z) \approx 0$

When we program neural networks, we usually keep the parameter w and parameter b separate, where here b corresponds to a intercept.

In some conventions an extra feature is defined as $x_0 = 1$, so that now $x \in \mathbb{R}^{n+1}$ and [redacted] then define $\hat{y} = \sigma(\theta^T x)$, in this alternate convention we have vector parameters θ from $\theta^{(0)}$ to $\theta^{(n)}$, and so $\theta^{(0)}$ plays the role of b and the rest as w:

$$x_0 = 1, \quad x \in \mathbb{R}^{n+1}$$

$$\hat{y} = \sigma(\theta^T x)$$

$$\theta = \begin{bmatrix} \theta_{(0)} \\ \theta_{(1)} \\ \vdots \\ \theta_{(n)} \end{bmatrix} \begin{matrix} \} b \\ \} w \end{matrix}$$

however we want use this notation as it's more convenient to keep b and w separated.

[Logistic Regression Cost Function]

To recall: $\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$, want $\hat{y}^{(i)} \approx y^{(i)}$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b), \text{ where } \sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}, \text{ want } \hat{y}^{(i)} \approx y^{(i)}$$

$x^{(i)} \; / \; y^{(i)} \; / \; z^{(i)}$

Loss error function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ (but not suitable for Logistic Regression)

so:

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$$

Good for logistic regression for a single example

This equation is balanced to determine a value for $y=1$ or $y=0$; by replacing y with one $\{0,1\}$, one of the terms is eliminated and it gives a big or small value. If $y=1$ we want the value as big as possible, but if $y=0$, we want it as small as possible.

The cost function applies the loss error function for all the samples

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) =$$

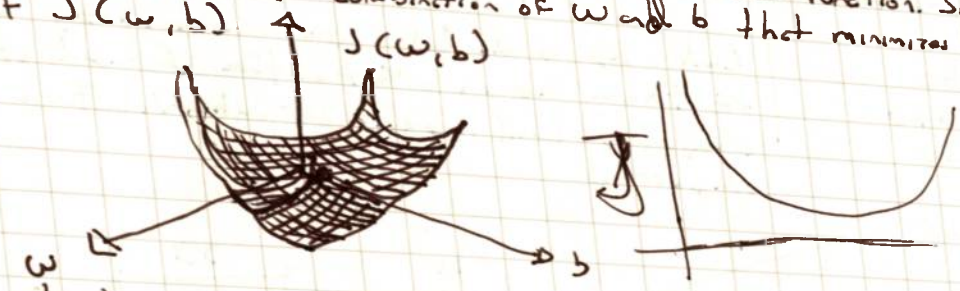
applied to parameters

~~$\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) =$~~

$$-\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)}) \right]$$

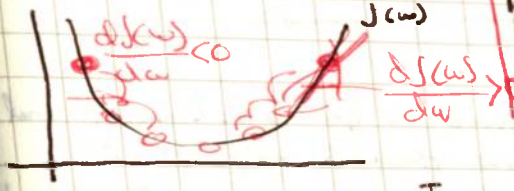
descent]

Used to learn the parameters w and b of the cost function. So we want to find the combination of w and b that minimizes the value of $J(w,b)$



After iteration, we tend to converge on the ~~local~~ global optima.

A plotted curve:



Repeat {

$$w := w - \alpha \frac{dJ(w)}{dw}$$

}

α learning rate which controls the size of the steps.

From calculus, we know the derivative tells us the slope of a point in a graph, thus by dividing the height and width of the point we get a triangle and it's possible to compute it's current slope; this applies on both ends, if the derivative is lower than zero, then it will move to the right or bottom on each iteration, whereas if the derivative is higher than zero, it will move to the bottom left after each iteration.

On the previous equation we've defined for J of w, but our cost function is a function of both $J(w$ and $b)$, so the inner loop of gradient descent ends up as:

$J(w,b)$:

$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

$$b := b - \alpha \frac{d\,J(w,b)}{db}$$

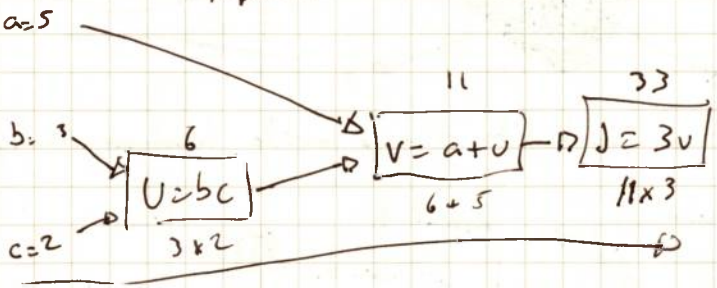
how much the slope function slopes into w direction.

[Computation Graph]

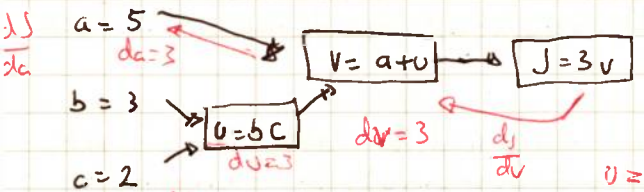
$J(a,b,c) = 3(a+bc)$

$u = bc$
$v = a+u$
$J = 3v$

Each step is divided into sub samples, and each of them has an effect on the forward steps changing c will impact u which in turn will impact J



To perform the derivatives, we ~~also~~ have to go backward and determine by how much the value changes ~~of~~ with respect of a modification in the previous step. As values changes are dependant one after another, the resultants are chained and by that fact as well the derivatives. Thus enables us to use the chain rule of derivatives. $a \to v \to J$ $\frac{dJ}{dv} \cdot \frac{dv}{da} = \frac{dJ}{da}$



$\frac{dJ}{dv} = 3 = \frac{dJ}{du} \cdot \frac{du}{dv}$ (3 = 3 · 1)

$u = 6 \to 6.001$
$v = 11 \to 11.003$
$J = 33 \to 33.009$

$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db}$

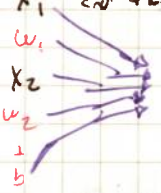
$b = 3 \to 3.001$
$u = bc \to 6 \to 6.002 = c = 2$
$J = 33.006$ ← $v = 11.002$
$J = 3v$

[Logistic Regression Gradient Descent] - derivatives computation for 6.0.

Logistic Regression recap:

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

Supposing we have two features x_1 and their weights



$$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$$

"da" $= \frac{d\mathcal{L}(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

$dz = a - y$

$\frac{\partial \mathcal{L}}{\partial w_1}$, "$dw_1$" $= x_1 dz$,

$dw_2 = x_2 dz$

$db = dz$

[Logistic Regression on "m" examples]

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y)$$

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

And the derivative of the weights is also equal to the sum of the derivatives

$$\frac{\partial}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \frac{d}{dw_1} \left(\mathcal{L}(a^{(i)}, y^{(i)})\right)$$

Algorithm: ~~cross out~~ assuming $n=2$

$J=0$; $dw_1=0$, $dw_2=0$; $db=0$ → computation

for $i=1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log (1-a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$ } $n=2$ — for loop over the features

$db \mathrel{+}= dz^{(i)}$

$J /= m$

$dw_1 /= m$; $dw_2 /= m$; $db /= m$

One step of gradient descent

At the end of the computation:

$dw_1 = \frac{dJ}{dw_1}$ $dw_2 = \frac{dJ}{dw_2}$ and $db =$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

The for loops aren't optimal, so best is to do vectorization.

[Vectorization]

columns, Fields

As a way to optimize the calculations, taking in advantage the modern CPU architectures that allow the use of SIMD (Single Instruction Multiple Data), it's possible to take advantage of matrix operations or vector operations, several of the steps can be greatly optimized.

It becomes possible to eliminate the use of for loops and increase the speed by in magnitudes. order of magnitudes.

To ~~cccl~~ compute $dz^{(i)}$, we need to do $a^{(i)} - y^{(i)}$, and as we have vectors representing the whole array of $dZ = [dz^{(1)}\ dz^{(2)}\ dz^{(m)}]$ we can just do ~~with numpy~~:

$$A = [a^{(1)}, a^{(2)}, \dots, a^{(m)}] \quad Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$$

$$dZ = A - Y = [a^{(1)} - y^{(1)},\ a^{(2)} - y^{(2)}\ \ a^{(m)} - y^{(m)}]$$

So with one line of code [$dZ = A - Y$] we can compute it all at the same time.

Now in the case of w, we had to multiply $x^{(i)}$ with $dz^{(i)}$, this for all the features "n". We can as well get rid of the for loop to vectorize it, applying matrix multiplications, which multiply and do the same sum.

Similar (the same) happens with b: $db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$, and in python we can do: ~~1/m np.dot~~ $1/m \cdot np.sum(dz)$,

for $dw = \frac{1}{m} X\, dz^T = \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix} =$

$$\frac{1}{m} [x^{(1)} dz^{(1)} + x^{(2)} dz^{(2)} + \dots + x^{(m)} dz^{(m)}]$$

Vectorized Version:

$Z = w^T X + b$

$= np.dot(w^T, X) + b$

$A = \sigma(Z)$

$dZ = A - Y$

$dw = 1/m\ X\, dZ^T$

$db = 1/m\ np.sum(dZ)$

$w = w - \alpha\, dw$

$b = b - \alpha\, db$

* Simple iteration of gradient descent

Forward Prop and backward Prop.

Broadcasting in Python

When we do operations between matrices and real numbers, Python uses a feature called broadcasting, and what it does is to replicate the value (real, horizontal or vertical vector) to match the shape of the (m,n) matrix.

If we have for example:

$np.sum(axis=0)$ = Vertical sum
$np.sum(axis=1)$ = horizontal sum

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 \text{ it becomes } \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

or $(1,n)$

$$\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + [100\ 200\ 300] \text{ it becomes } \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100&200&300\\100&200&300\end{bmatrix} = \begin{bmatrix}101&202&303\\104&205&306\end{bmatrix}$$

$$\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \underset{(m,1)}{\begin{bmatrix}100\\200\end{bmatrix}} \text{ becomes } \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100&100&100\\200&200&200\end{bmatrix} = \begin{bmatrix}101&102&103\\204&205&206\end{bmatrix}$$

General principle

(m,n) matrix $\overset{+}{-}{\times}{/}$ $(1,n) \Rightarrow (m,n)$ it will copy n times

$(m,1) \Rightarrow (m,n)$ it will copy n times.

$$\underset{(m,1)}{\begin{bmatrix}1\\2\\3\end{bmatrix}} + 100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$$

$$[1\ 2\ 3] + 100 = [101\ 102\ 103]$$

Notes on Python and vectors

Do not use one rank ~~vectors~~ arrays, always define the dimensions, i.e.

$a = np.random.random(5,1)$ = Column vector

$a = np.random.randn(1,5)$ = Row vector

assert(a.shape == (5,1)) → to verify the shape is correct

Logistic Regression test in Jupyter

train_set_x has shape (209, 64, 64, 3)
A B C D

A = rows
B = height
C = width
D = RGB → 3 color representation

train_set_y has shape (209,) - Flags → 1 = cat, 2 = non-cat

test_set_x (50, 64, 64, 3)
test_set_y (50,)
confirmed by str(... .shape)

From (209, 64, 64, 3), shape[n] gives
shape[0] = 209
[1] = 64
[2] = 64
[3] = 3

∴ on using shape values (0, 1, 2, 3)

I might be wrong but seems numpy handles the columns and rows in an inverted way (n, m) instead of (m, n) as in Matlab

From (209, 64, 64, 3), we can flatten the dataset to get a numpy array of shape (64×64×3, 1), where each column will represent a flattened image

Flattened = ~~normal.shape[0]~~
normal.reshape(normal.shape[0], -1).T

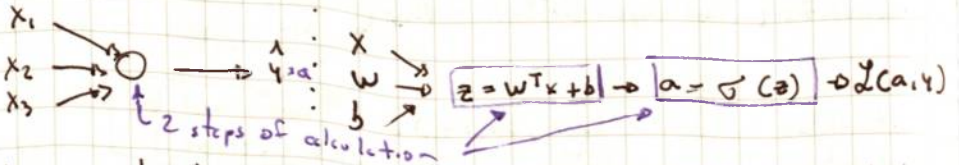
So (209, 64, 64, 3).reshape(209, -1).T -1 = lets numpy to figure out what to do with the rest, in this case, one dimension with 209 things concatenated

After the images in the arrays have been flattened, then we need to standerize the dataset, meaning that we substract the mean of the whole numpy for each example and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient to just divide every row of the dataset by 255, which is the maximum value for a pixel channel (i.e. 255; 6 2 5 5; 0 0 0)

train_set_x = train_set_x_flatten / 255
test_set_x = test_set_x_flatten / 255

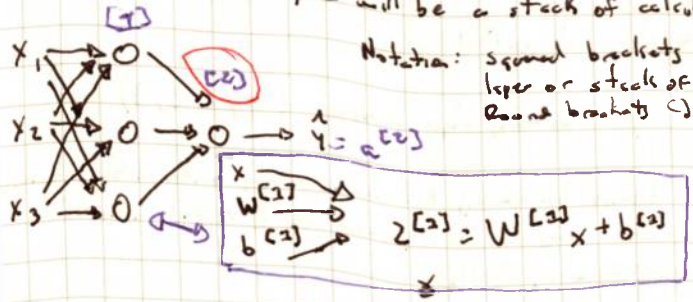
[Shallow Neural Networks]

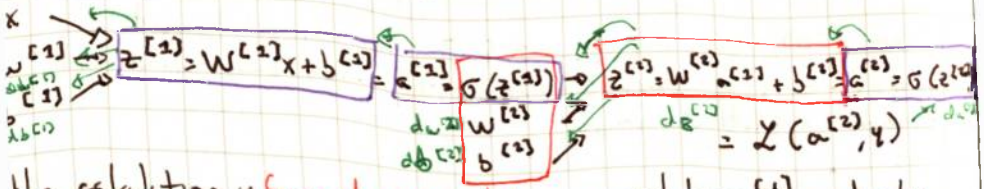
Last module we took logistic regression and we've seen how the following model worked:



In a neural network, z will be a stack of calculations, for z and a

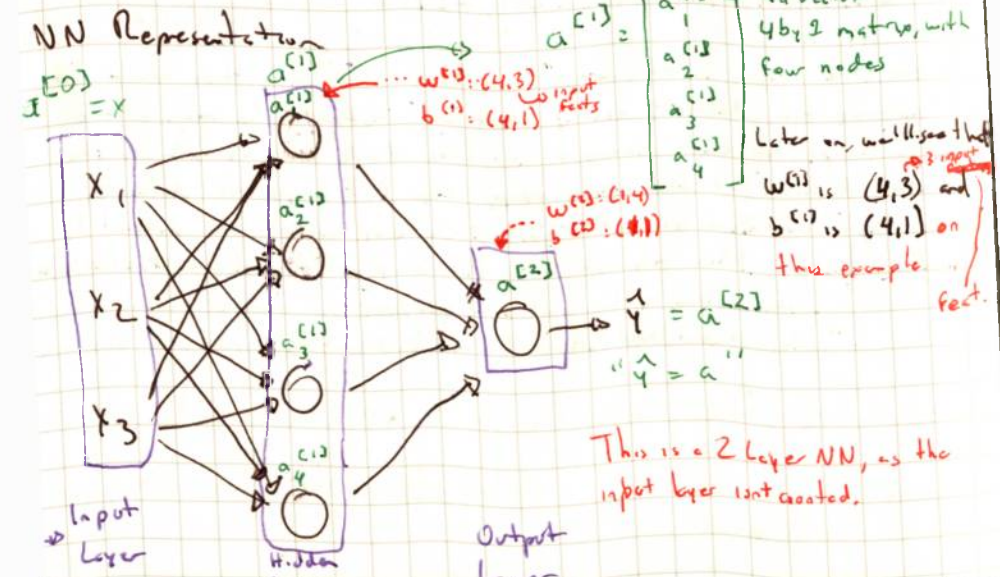
Notation: squared brackets [ ] denote to the layer or stack of nodes and it's values.
Round brackets ( ) = individual training example





the calculation w forward propagated across each layer [l], and also the back propagation will perform the derivatives across.

NN Representation



This is a 2 Layer NN, as the input layer isn't counted.

As we know, each node performs two operations, it computes the weight vs feature multiplication + bias, and then the activation sigmoid function. That said in vectorized approach would be:

$$z = w^T x + b \rightarrow \sigma(z).$$

As this repeats on each node on the NN, taking in consideration as an example, the previous NN drawn, we get:

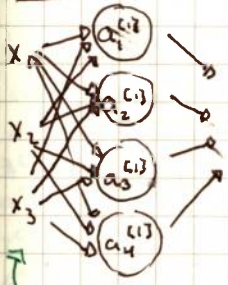
$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]} \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]} \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]} \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]} \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

Computing the above in for loops would be efficient, so we can vectorize them.

$$z^{[1]} = \begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix}_{4,3} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}_{(4,1)} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Given that, what we see for the first layer of the neural network, given the input x:



$W^T = W^{[2]}$
$b = b^{[2]}$

$$z^{[1]}_{(4,1)} = W^{[1]}_{(4,3)} x^{a^{[0]}}_{(3,1)} + b^{[1]}_{(4,1)}$$
$$a^{[1]}_{(4,1)} = \sigma(z^{[1]}_{(4,1)})$$
$$z^{[2]}_{(1,1)} = W^{[2]}_{(1,4)} a^{[1]}_{(4,1)} + b^{[2]}_{(1,1)}$$
$$a^{[2]}_{(1,1)} = \sigma(z^{[2]}_{(1,1)})$$

[Vectorizing across multiple examples]

for i = 1 to m:

$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

Capital X contains the vector X with each training examples stacked in columns

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

One of the properties of this notation is that these matrices, horizontally index training examples and vertically the nodes on the NN.

$$A = \begin{bmatrix} \circ & \circ & \circ & \cdots & \circ \\ \circ & \circ & \circ & \cdots & \circ \\ \circ & \circ & \circ & \cdots & \circ \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \circ & \circ & \circ & & \circ \end{bmatrix}$$

first node on the first training example

Second hidden unit on the first training example

training examples

Activations of the first hidden unit on the last training example (m)

Different nodes on the NN

$$Z^{[1]} = W^{[1]} X + b^{[1]} \quad \text{or } A^{[0]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

We went to lowercase to uppercase by stacking in columns:

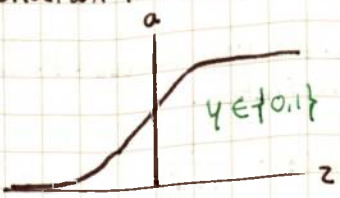
$$Z^{[1]} = \begin{bmatrix} | & | & | & & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} & \cdots & z^{[1](m)} \\ | & | & | & & | \end{bmatrix}$$

$$A^{[2]} \begin{bmatrix} | & | & | & & | \\ a^{[1](1)} & a^{[1](2)} & a^{[1](3)} & \cdots & a^{[1](m)} \\ | & | & | & & | \end{bmatrix}$$

training examples

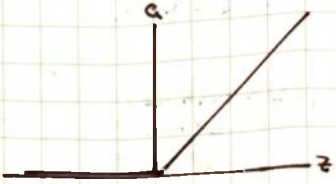
hidden units

[Activation Functions]

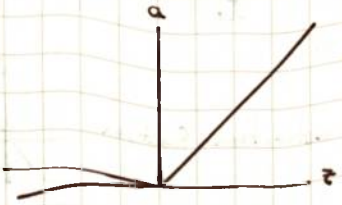


Sigmoid: $a = \frac{1}{1+e^{-z}}$

tanh $= a = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$



ReLU $a = \max(0, z)$



Leaky ReLU $a = \max(0.01z, z)$

Recommendations:

- Sigmoid: Don't use it, except at the final layer if doing binary classif.
- Tanh: much better, strictly superior
- ReLU: the default, the most common
- Leaky ReLU: Values greater or equal to zero (no negatives)

(Why do we need a activation function?) (non linear)

To compute interesting functions with a neural network, we do need to pick a non linear activation function.

If we decide to avoid the activation function and pass the values straight into the next layer, we are just computing another simple variation of the equation (linear), which:

$z^{[1]} = W^{[1]}x + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]}) = z^{[1]}$ ⟹

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

$a^{[2]} = g^{[2]}(z^{[2]}) = z^{[2]}$

$a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]}$

$a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} =$ ($a^{[1]}$)

$= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} =$

$= (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]})$

W' b'

A linear hidden layer is more or less useless because the composition is itself a linear function ($z = mx + b$)

$= W'x + b'$

- Exception: if doing ML on a regression problem

[Derivatives of activation functions]

When we implement back propagation for the NN, we need to compute the slope or derivative of the activation functions.

- Sigmoid activation Function

$$g(z) = \frac{1}{1+e^{-z}}$$

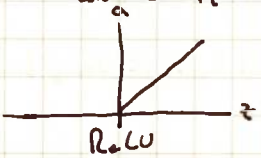
Alternative Notation: $g'(z) = \frac{d}{dz} g(z)$

$$\frac{d}{dz} g(z) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = g(z)(1-g(z))$$
$$= a(1-a)$$

- Tanh activation function

$$g(z) = \tanh(z)$$
$$= \frac{e^{z} + e^{-z}}{e^{z} + e^{-z}}$$
$$= 1 - (\tanh(z))^2$$
$$g'(z) = 1 - a^2$$

- ReLU and Leaky ReLU



$$g(z) = \max(0, z)$$
$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undef} & \text{if } z = 0 \end{cases}$$

$$g(z) = \max(0.001z, z)$$
$$g(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

[Gradient descent for Neural Networks]

Forward Propagation:

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = g(Z^{[2]})$

Back propagation:

$Y = [y^{(1)}, y^{(2)}, \ldots y^{(m)}]$

$dZ^{[2]} = A^{[2]} - Y$

$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$

$db^{[2]} = \frac{1}{m}$ np.sum($dZ^{[2]}$, axis = 1, keepdims = True)

(sum across 1 axis horizontally) — $(n^{[1]}, 1)$

$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$

$(n^{[1]}, m)$ $(n^{[1]}, m)$ (element wise product)

$dW^{[1]} = 1/m\ dZ^{[1]} X^T$

$db^{[1]} = 1/m$ np.sum($dZ^{[1]}$, axis = 1, keepdims = True)

$(n^{[1]}, 1)$ $(n^{[1]}, 1)$

[Random Initialization]

It's important for a NN to initialize the weights to some random values, instead of zeros. Zeros works fine for linear regression but for NN's is a problem. The main issue is that the hidden units will compute the same values, which by the way, be multiplying by zero, they get annulated.

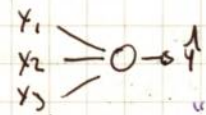
To do it, we can use numpy as

$W^{[1]}$ = np.random.randn((2,2)) * 0.01 ← very small random values to not get to the steepest or smallest part of the function

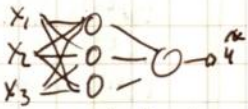
$b^{[1]}$ = np.zero((2,1)) As long as W is random, this won't break symmetry

[Week 4]

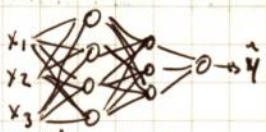
What is a deep neural network



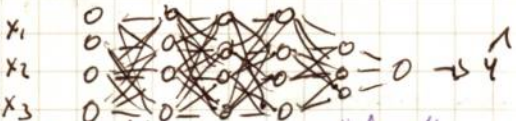
logistic regression "shallow"



1 hidden layer

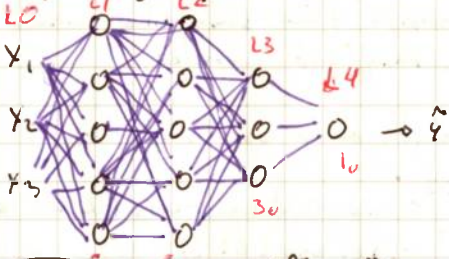


2 hidden layers



5 hidden layers "deep"

Is good to try one by one, perhaps adding layers



Notation

L = Number of layers

$n^{[l]}$ = # units in layer l

In the simple: $a^{[l]}$ = units in layer l

$L = 4$

$n^{[0]} = n^{(x)} = 3 \quad n^{[1]} = 5, \; n^{[2]} = 5, \; n^{[3]} = 3 \quad n^{[4]} = n^{[L]} = 1$

$a^{[l]} = g^{[l]}(z^{[l]})$

[Forward propagation in a Deep Network]

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

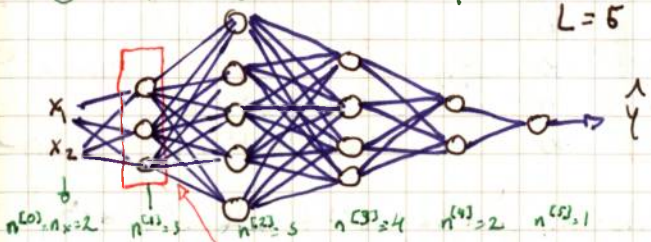
→ 1 training example

l = layer

→ ok to use for loops on forward prop to iterate between layers

[Getting the Matrix dimensions right]

When implementing a DNN, one of the tools to use is to check correctness of the code is to pull a piece of paper and just work thru the dimensions of matrix we are working with.



Taking in consideration $z^{[1]} = W^{[1]} \cdot x + b^{[1]}$

$z^{[1]}$: $(3,1)$ $(n^{[1]}, 1)$

$W^{[1]}$: $(3,2)$ $(n^{[1]}, n^{[0]})$

x: $(2,1)$ $(n^{[0]}, 1)$

$b^{[1]}$: $(3,1)$ $(n^{[1]}, 1)$

$$W^{[l]} = (n^{[l]}, n^{[l-1]})$$

and $b^{[l]} = (n^{[l]}, 1)$

$W^{[1]}$: $(n^{[1]}, n^{[0]})$

$W^{[2]}$: $(5,3) = (n^{[2]}, n^{[1]})$

$z^{[2]}$: $W^{[2]} \cdot a^{[1]} + b^{[2]}$

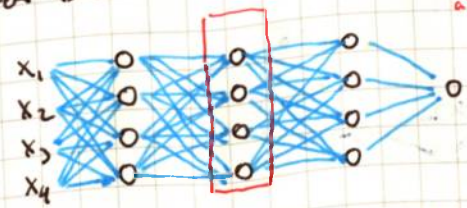
$(3,1) \rightarrow (5,3) \leftarrow (3,1)$ and so on...

$W^{[2]} = (4,5)$

$W^{[4]} = (2,4)$, $W^{[3]} = (1,2)$

[Building Blocks of Deep Neural Networks]

We've already seen the basic building blocks of forward propagation and back propagation, the key components to build a deep net. Here is a network with a few layers. Let's pick a layer and look at the computations only for that layer.



Layer l: $W^{[l]}, b^{[l]}$

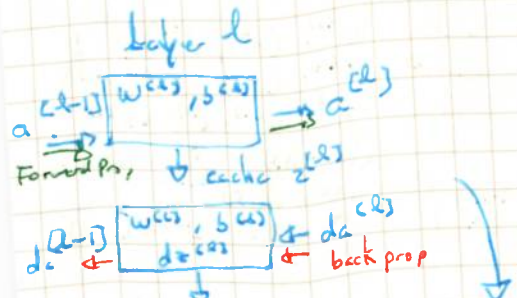
forward: input $a^{[l-1]}$, output $a^{[l]}$

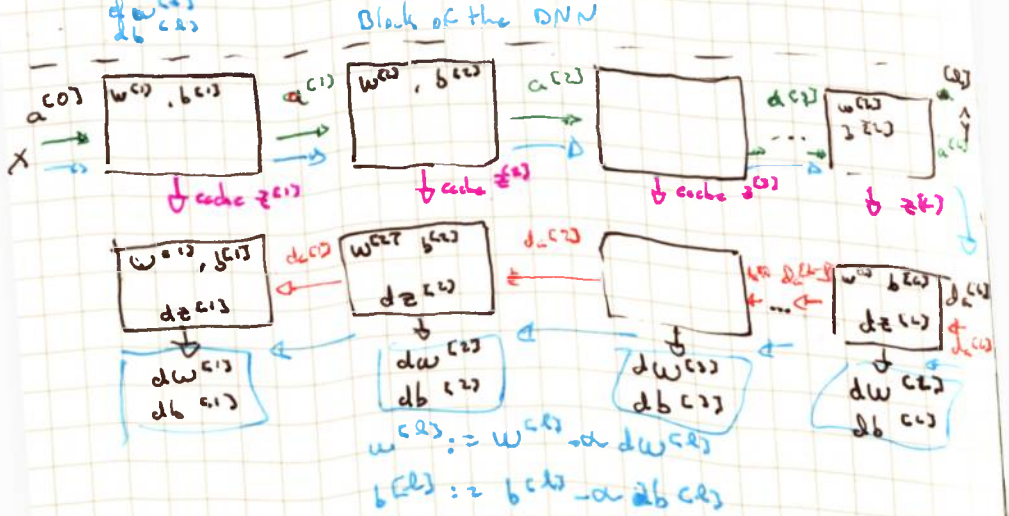
$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

Turns out that for later use, would be useful to cache $z^{[l]}$, because would be useful for the backward propagation step. For the backward prop step, we will implement a function that inputs $da^{[l]}$ and outputs $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

cache($z^{[l]}$)





$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$

$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$

[Forward and Backward Propagation] - implementation

→ Input $a^{[l-1]}$

→ Output $a^{[l]}$, cache $(z^{[l]})$

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

Vectorized:

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Forward Prop →

Backward propagation for layer l

→ Input: $da^{[l]}$

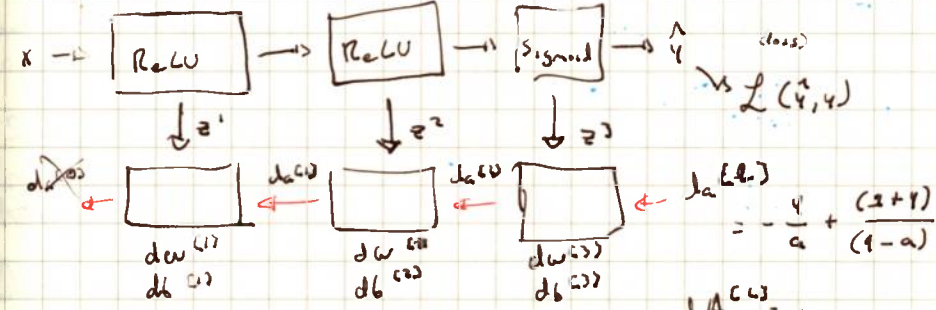
→ Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]}) \quad \text{(element wise prod)}$$
$$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

Vectorized:

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$
$$db = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$$
$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

Summary:



$$da^{[L]} = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$$

$dA^{[L]} = .$

[Parameters vs hyperparameters]

Parameters are those given from the data: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots$

Hyperparameters are those that control the ultimate parameters W and b because they are controlling the result, they determine the final values of W and b:

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]} \ldots$

Hyperparameters:
- Learning rate α
- # iterations
- # hidden layers L
- # hidden units $n^{[1]}, n^{[2]}, \ldots$
- choice activation function

There are other parameters in deep learning such as:
- Momentum
- Mini batch size
- Regularization parameters.

[Course 2: Improving Deep Neural Networks
- Hyperparameter tuning,
- Regularization and
- Optimization

[Setting up the Machine Learning application]

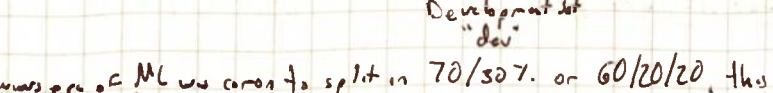
· Train / Dev. / Test sets

When training a NN, we have to make a lot of decisions, such as, the amount of layers, nodes, activations for hidden layers, so in practice applying ML is a highly iterative process: idea → code → Experiment

· As there are many different fields of applications, is not just possible to bring the experience or intuitions from one field area to the next, for example, bringing knowledge from NLP into Computer Vision or from speech recognition into Ads, or from security into logistics.

One of the things that determine how quickly we can make progress is, how efficiently we can iterate the cycle of IDEA → CODE → TEST, and of the processing power is somehow bounded into the CPU's or GPU's capabilities, we want to make it fast.

Setting up our datasets well, in terms of train, development and tests, can help out on that steps. At least to execute it faster. Dividing the data into 3 subsets:



In the previous era of ML we used to split in 70/30% or 60/20/20, this were considered reasonable when the amount of data was about 1000 or 10K or 100K samples. In the current era of Big Data, where the amount of samples is extremely big, (i.e. 1000000), the dev and test set has become a small percentage

For example, if we have 1M of samples, perhaps 1% (10K) is enough to have as dev and test purposes, i.e. 98/1/1%, there are cases where 99.5/0.25/0.25% or 99.5/0.4/0.1%.

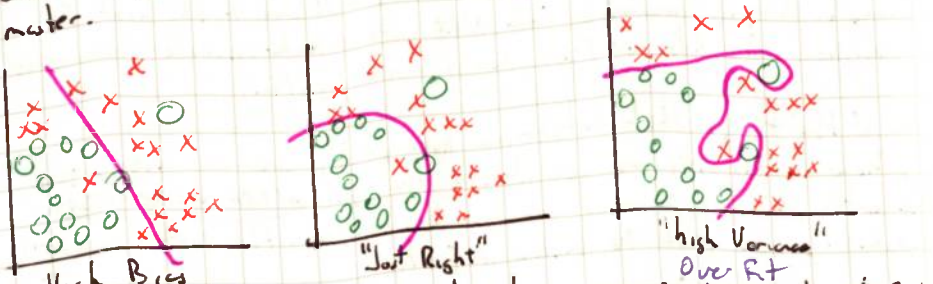
(Mismatched train/test distribution)

It might happen that we train our NN with very high quality samples, for example, if the ML Algorithm is supposed to recognize cats, we must get samples not only of professional pictures with perfectly framed cats. while the users of our app might take pics of their cats with low res phones, or the cats being blurry, out of frame, etc. So its good idea to find realistic samples.

- Make sure dev and test sets come from the same distribution

(Bias / Variance)

Good practitioners of ML, have ortend to be very sophisticated in understanding of bias and Variance. It sounds simple but gets complex to master.



Continuing with the cat classification, if we have a set of train/dev examples. We would look at the key numbers:

	Case 1	Case 2	Case 3	Case 4
Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	Has high Variance	High Bias	High Bias and High Variance	low bias low Variance

This analysis is predicated on the assumption that human level performance gets nearly 0% error, or more generally, that the optimal error is nearly 0%, ≈0%. but if the optimal (Bayes) error is 15%, then the case 2 would have low bias and low variance.

To take away, by looking at the training set error, we can get a sense of how well we are fitting at least the training data, and then we can looking how much higher is the error in dev set so, that gives us a sense how bad the variance is going.

[Basic Recipe for ML]

Ask

- Does the algorithm has high bias? → Y: • train bigger net • train longer • (try another NN architecture)
  ↓ N
- High Variance? (does not perform) → Y: • More data • Regularization • (another NN architecture)
  ↓

Done

The Bias-Variance tradeoff, there were not tools to ~~reduce~~ reduce one without increasing the other.
Having a bigger network, almost always reduces the Bias without hurting the Variance
- Getting more data, always reduce the variance and doesn't hurt the bias very much

[Regularization]

If we suspected the NN is overfitting the data (High Variance), we can try regularization, as training data might be expensive.

Telling from the cost function

Cost $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 + \frac{\lambda}{2m} b^2$

omitted as it's irrelevant, impact every all the other parameters

L2 regularization → $\|w\|_2^2 = \sum_{j=1}^{n_x} (w_j)^2 = w^T w$

• square euclidian norm of the parameter vector w

Lambda λ is the regularization parameter, and in python we'll refer to it as lambd because the word lambda is a reserved word.

How we apply regularization in a neural network

In the NN, we have the cost function which is the function of all parameters:

$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$

$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$

$w: (n^{[l]}, n^{[l-1]})$ matrix

"Frobenius norm"

From backprop: $dw^{[l]} := (\text{From backprop}) + \frac{\lambda}{m} w^{[l]}$

$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$

"Weight decay"

Because slowly we decay by reducing the dw by a little less than 1

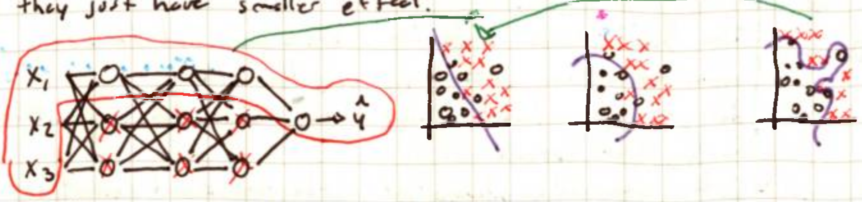
$w^{[l]} = w^{[l]} - \alpha\left[(\text{From backprop}) + \frac{\lambda}{m} w^{[l]}\right]$

$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha(\text{From backprop})$

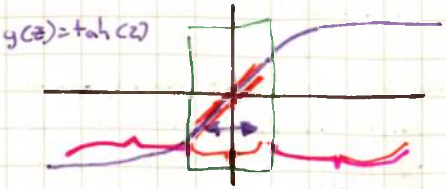
$= \left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]} - \alpha(\text{From backprop})$

[How does regularization prevent overfitting?]

If we set the λ to be a high value, then $\frac{\lambda}{2m}\sum_{l=1}^{L}||w^{[l]}||_F^2$ it will cause the weight matrices w to be close to zero! $w^{[l]} \approx 0$ and that will cancel out several neurons, thus simplifying the NN, taking us from a complex function with high variance (i.e., complex n degree function) into a simpler and lower grade function, reducing the impact of a lot of hidden units, as if we were using logistic regression. They, what happens is that they just have smaller effect.



Another visualization with the tanh function:



As long as z is quite small, so if z takes a small set of values, then it's using only the linear regime of the tanh function. If z uses larger values, it starts to become less linear.

So if it takes a small set of feature values, $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$ and/or g(z) will be roughly linear. And if every layer is linear, even on a DNN, it will only calculate a linear function.

[Dropout Regularization]

Dropout consist into randomly drop out (inutilized) hidden nodes, so it simplifies the network. Then apply back prop on them and repeat.

In code: For example, with layer $l=3$, and "keep-prob" = 0.8 (80% to keep 20% remove)

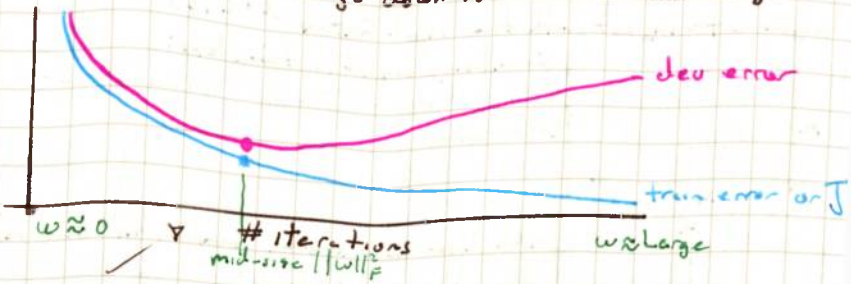
```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep-prob
a3 = np.multiply(a3, d3)  # a3 *= d3
a3 /= keep-prob
```

↳ bajamos uno de los valores al verificar cuales son menores a keep-prob, esto da una matriz de booleanos. Para conservar los valores en $a^{[L-1]}$, subimos los valores de los nodos "vivos" en 80%.

At test time we should not use dropout, otherwise we will get just noise on the predictions

[Early Stopping]

As we run gradient descent, plot the cost function J, with early stop, we also plot our dev set error, and what we find is that the dev set error will go down for a while and then go up.



We stop halfway when we see it was performing better, by stopping halfway, we will have a mid size value of w and similar to L2 regularization, it's likely at that point it will overfit less.

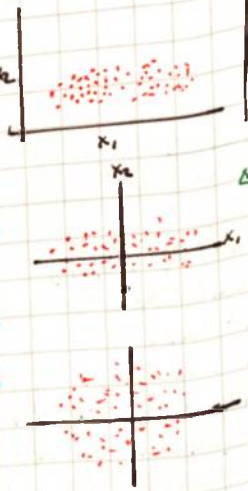
It does have a downside (early stopping), because we should consider that lowering the cost of function J is one thing and regularizing the dev set is completely different thing. This is called orthogonalization and it's point is about thinking or doing one task at a time.

Doing early stopping, breaks whatever we are doing to reduce the cost function J and also simultaneously tries to not overfit.. instead of using two tools to solve different problems, we are mixing them into one.

[Normalizing test sets]

One of the techniques in NN, that will speed up the trains. is if we normalize the inputs. Let's see what that means:

Imagine a trains set with two input features $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



Normalizing consist in two steps:

1: Substract mean: $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$, then $x := x - \mu$ for every trains ex.

2: Normalize

We have now x_1 every example has zero mean, then the second steps is:

2: Normalize the variance: $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} **2$ ← element wise square

now σ^2 is a vector with the variances for each of the features.

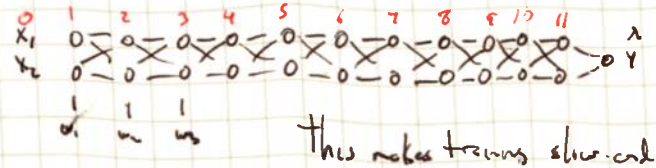
Now we take every example and divide it by sigma squared vector σ^2 : $x /= \sigma^2$ and we end up with:

IMPORTANT: If we use this to scale the trains data, then use it as well on the test set, the same values of μ and σ^2. They have to be normalized/scaled on the same way because we want them defined by the same transformation.

[Vanishing / exploding gradients]

When we have a very deep NN, we have chained several multiplications each one by one will grow exponentially.

If the values of the weights are greater than 1 (1.01, 1.5, etc) then after several layers, the value will grow greatly. Similar happens when the weight is less than 1, 0.9 or less, exponentially it will decrease to a value so small. Therefore the gradients might explode or vanish.

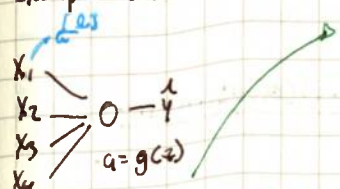


This makes trains slow and difficult, but it can be partially solved carefully picking the careful choose of the random initialization.

[Weight Initialization for Deep Networks]

To minimize the effects of the vanishing/exploding gradients, it's possible to adjust the weights on each layer of the DNN by taking in consideration the values on the previous layer.

Example with a single neuron.



the larger n is, the smaller we want it

One reasonable thing to do is to set the variance of $w_i = \frac{1}{n}$ (n = # of input features)

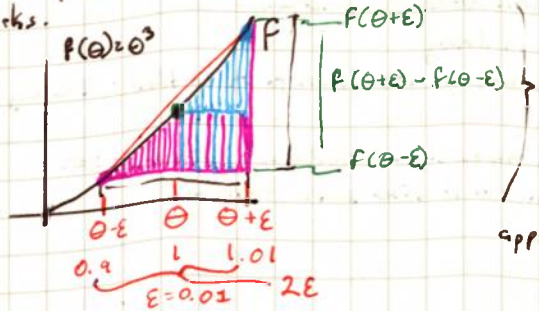
$w^{[l]}$ = np.random.randn(shape) * np.sqrt($\frac{1}{n^{[l-1]}}$)

Other variant, if using tanh activation function is the **Xavier initialization** $\sqrt{\frac{1}{n^{[l-1]}}}$

ReLU

[Numerical Approximation of Gradients]

When we implement back propagation, we will find that there is a test called gradient checking that can help us check that the back propagation is correct. This because sometimes we wrote all the equations and we might not be 100% sure if we got all the details right. So in order to build up the gradient checking, let's see how numerical approximation works.



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

$$\frac{(1.01)^3 - (0.99)^3}{2 \times 0.01} = 3.0001$$

$$g(\theta) = 3\theta^2 = 3$$

approx error = 0.0001

The value is very close to the real derivative of $f(\theta)$, the real value is 3.0301, thus we end with a real error of 0.03, which just for checking is good enough, but not for real calculations. Also an important note is that running it together with the back prop, will duplicate the time required to compute and converge. so the test should be checked after the calculations are verified to be implemented properly.

[Gradient Check for a neural network]

The NN will have some set of parameters $W^{[1]}, b^{[1]}, \ldots W^{[L]}, b^{[L]}$ so we need to take them and reshape them into a big vector θ, thus taking $W^{[1]}$ matrix and make it vector, same for $b^{[1]}$, same for W^{L} and $b^{[L]}$ concatenate them. so instead of $J(W^1, b^1, \ldots, W^L, b^L)$, we get $J(\theta)$. and then Then do the same for $dW^1, db^1, \ldots, dW^L, db^L$ and reshape them into a big vector $d\theta$,

for each i:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \theta \ldots \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots \theta_i, \ldots)}{2\varepsilon}$$

$$\approx d\theta[i] = \frac{dJ}{d\theta_i}$$

then check $$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} + \| d\theta \|_2}$$

in practice $\varepsilon = 10^{-7}$, so if the formula gives 10^{-7} or less than it's OK

10^{-5} ... ok

10^{-3} — worry!

[Gradient Checking Implementation Notes] - practical.

- Don't use in training, only in debug
- If the algorithm fails grad check, look at components to try to identify the bug
- Remember the regularization
- Doesn't work with dropout [because eliminates random nodes], turn it off
- Run at random initialization, run it after some iterations.

Week 6

(Optimization Algorithms)

[Mini Batch Gradient Descent]

As trainn on large datasets is slow, and considering that we might need to run the model several times to find the right set of parameters and optimization algorithms. With the big data on our time, deep learning and its deep layers, even by using vectorization, still takes a long while to complete the training, and as it's empirical, we need to optimize it

As we know, vectorization allow us to efficiently compute on m examples, so vectorizing the training set we would have:

$X = [\underbrace{x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(1000)}}_{X^{\{1\}}} \mid \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}} \mid \dots \mid \underbrace{\dots x^{(m)}}_{X^{\{5000\}}}]$ (n_x, m)

$Y = [\underbrace{y^{(1)}, y^{(2)}, y^{(3)}, \dots, y^{(1000)}}_{Y^{\{1\}}} \mid \underbrace{y^{(1001)} \dots y^{(2000)}}_{Y^{\{2\}}} \mid \dots \mid \underbrace{y^{(m)}}_{Y^{\{5000\}}}]$ $(1, m)$

If we have a dataset of $m = 5000.000$, we could split it in groups For example of 1000 samples, thus we end up with 5000 smaller groups or samples; for this we will introduce a new notation, "curly braces" and each group of 1000 samples will be noted with a superscript (superscript) denoting the group or batch we are refferns to; batches are separated as well as X and Y. Getting for example a minibatch $t = X^{\{t\}}, Y^{\{t\}}$

The process to follow is as usual with like if it were all the training set but on this case, it's limited to one pass on the batch only once

- for $t = 1, \dots, 5000$ {

  Forward prop on $X^{\{t\}}$ — Vectorized implementation for 1000 examples

  $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

  $A^{[1]} = g^{[1]}(Z^{[1]})$

  $\vdots$

  $A^{[L]} = g^{[L]}(Z^{[L]})$

  Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \| W^{[l]} \|_F^2$ (for $X^{\{t\}}, Y^{\{t\}}$)

  Backprop to compute gradients with respect to $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

  $W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$, $b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$

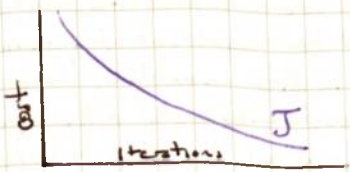
  }

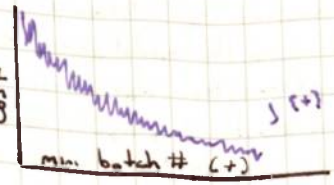
the code above to 1 pass to the training set, this is called epoch and it means a single pass thru the training set.

With mini batch gradient descent a single pass thru the training set (1 epoch), allow us to take "t" gradient descent steps (1000 ie). Where as normal training, a gradient descent step would be done after

[Understanding batch gradient descent]

Training a mini batch gradient descent, would look like:





The reason of the bumps costs on minibatch, is that each minibatch has different samples and some of them might be more expensive than others

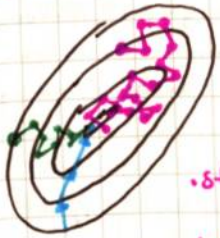
[Choosing the minibatch size]

There are two extreme cases:

If the minibatch size is m (all samples): We end up with batch gradient descent (running samples it's own)

If the minibatch size is equal to 1: Stochastic gradient descent (1 sample)

Lets looks what these cases would do while trying to optimize the cost function, on the contour functions.



- Will take large steps towards the global minima.
- Will start randomly and will jump wildly, it will never converge but will end close.

- Stochastic Gradient Descent
  - Looses speed from vectorization

In between (batch size not too big)
- Fastest learning
- Vectorization ✓
- Progress without waiting for long ✓

Batch Gradient Descent
- too long iterations

- In practice: Something in between ● and ●

[Guidelines:]

- If training set is small: batch gradient descent ($m \leq 2000$)
- Big training set: 64, 128, 256, 512 (powers of 2, $2^6, 2^7, 2^8, 2^9$)
- Make sure the minibatch fits in CPU/GPU

This, however, are not the best approaches, not bad, but there are better than gradient descent.

[Exponentially Weighted Averages]

This is a key component of several optimization algorithms.

Let's take a sample of the average temperature during a year

$\theta_1 = 4.4\,C^\circ$

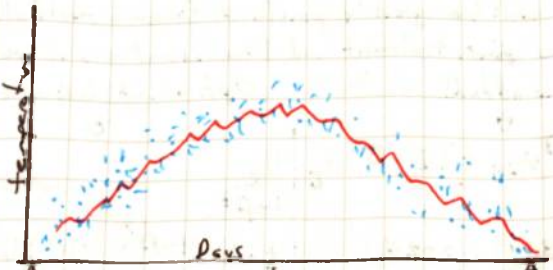
$\theta_2 = 9.4\,C^\circ$

$\theta_3 = 7.2\,C^\circ$

$\vdots$

$\theta_{180} = 15.6\,C^\circ$

$\theta_{181} = 13.33\,C^\circ$



$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1\,\theta_1$

$V_2 = 0.9 V_1 + 0.1\,\theta_2$

$V_3 = 0.9 V_2 + 0.1\,\theta_3$

$\vdots$

$V_t = 0.9 V_{t-1} + 0.1\,\theta_t$

The data looks a bit noisy and if we want to compute the trends of the local average or moving average we initialize $V_0 = 0$ and then average it with a weight of 0.9 times whatever was the previous value, plus 0.1 times that day temperature.

the formula then becames $V_t = 0.9 v_{t-1} + 0.1 \theta_t$, which generalizing we can call the 0.9 weigth as beta β, (being previously equal to 0.9), so we have:

$$\beta V_{t-1} + (1-\beta)\theta_t$$

Averaging V_t can think as $V_t \approx \frac{1}{1-\beta}$ days (samples), so for example

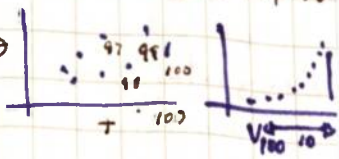
$\beta = 0.9$: $\approx$ 10 days temp

$\beta = 0.98$: $\approx$ 50 days temp - gives priority to the past values and reduces the current archive's weight

$\beta = 0.5$: $\approx$ 2 days temp - less influence from the past and more to current

$V_{100} = 0.1\theta_{100} + 0.9(V_{99}) = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9 V_{98}) = 0.1\theta_{100} + 0.9 V_{98} \ldots$

$= 0.1\theta_{100} + 0.1 \times 0.9\,\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + 0.1(0.9)^4\theta_{96} + \ldots$ and all of this coefficients add up to one up to a detail called bias correction.



$0.9^{10} = 0.35 \approx \frac{1}{e}$ $\qquad \frac{(1-\varepsilon)}{0.9} = \frac{1}{e}$

Implementation:

$V_0 = 0$

$V_1 = \beta V_0 + (1-\beta)\theta_1$

$V_2 = \beta V_1 + (1-\beta)\theta_2$

$V_3 = \beta V_2 + (1-\beta)\theta_3$

...

In programming

```
V = 0
for (each) {
    get next θ_t
    V = βV + (1-β)θ_t
}
```

[Bias Correction]

If we use exponentially weighted averages, our start will be impacted greatly on the values, as the $V_0 = 0$, ~~over value~~ the first iteration and subsecent iterations will be greatly penalized. Because multiplying by β will cancell out the θ_1 first element and the second one is very small (the one that is added).

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$V_0 = 0$

$V_1 = 0.98(0) + 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2$

$= 0.98(0.020) + 0.02\theta_2$

$= 0.0196(\theta_1) + 0.020_3$

⋮

Optimized (Bias correction)

$$\frac{V_t}{1-\beta^t}$$

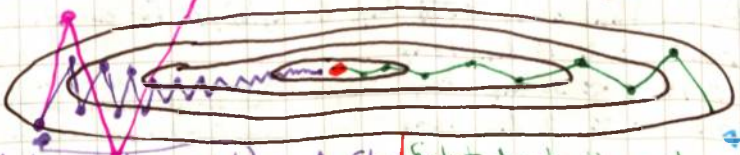
exponent

$t=2$: $1-\beta^t = 1-(0.98)^2 = 0.0396$

$$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

[Gradient Descent with Momentum]

The basic idea is to use the Exponential Weighted Averages From the gradients and then use that gradient to update the the weight itself.



If we start gradient descent here and after some iteration of gradient descent either batch or mini-batch, it will take large steps and slowly oscilate towards the minima, and if we are using a much larger learning rate, we may end up overshooting and diverging.

Gradient Descent with momentum:

On iteration t:

Compute dW, db on the current batch

$V_{dW} = \beta V_{dW} + (1-\beta) dW$ - Moving Avg

$V_{db} = \beta V_{db} + (1-\beta) db$

$W = W - \alpha V_{dW}$, $b = b - \alpha V_{db}$

$\beta = 0.9$ (usually) In practice Bias Correction is not used while implementing momentum

Another way to look at it, is that we want in the vertical axis to have a slower learning, where as on the horizontal axis, we want Faster learning. By using exponential weighted averages, we will smooth down and average the steps of gradient descent.

A way to rationalize the effect of the exponential weighted averages's impact on gradient descent, is to see the derivatives dW, db as providing acceleration to the descent, while β gives friction and V_{dW}, V_{db} provide speed:

$V_{dW} = \beta V_{dW} + (1-\beta) dW$ (velocity, friction, acceleration)

[RMS Prop]

Root-Mean Square Prop has a very similar behavior to gradient descent with momentum, however it uses more use of squares and square roots to penalize the weights and aid or reward the biases with the same purpose, to optimize the descent and approximation to the global minima, while avoiding to bounce laterally.

On iteration t:

Compute dw, db on current mini-batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2$ ← element wise

$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2$

β_2 = to not mix it with β from EWA

ε = Very small value to avoid numerical error if S_{dw} or S_{db} become too small $\approx \varepsilon = 10^{-8}$

$$w = w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}, \quad b = b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

[Adam Optimization Algorithm] = ADaptive MOMENT ESTIMATION

The ADAM optimization algorithm comes from the fusion of Gradient Descent with Momentum and R.M.S Prop. This algorithm has proven to be very efficient in several applications of ML. The implementation is as follows:

$V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\, dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db$ ← "momentum" β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2$ ← "RMSProp" β_2

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$, $V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t)$, $S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$

$$w = w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \qquad b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Hyperparameter choice

α : needs to be tuned

β_1 : 0.9

β_2 : 0.999

ε : 10^{-8}

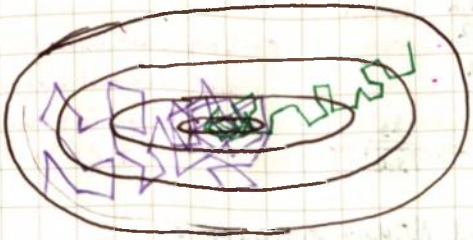
Adam Paper Author's recommended Values

(dw)

(dw^2) db^2

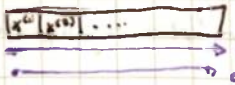
[Learning Rate Decay]

One of the things that may help to speed up our learning algorithm is to slowly reduce the learning rate over time, this is called learning rate decay. If we reduce slowly the learning rate α, during the initial phases, the α will be yet big thus allowing faster learning, while decreasing, the G.D. will approach to the minima, and with smaller oscillations, or not that big as with a constant α. This will allow for enable the algorithm to approach very confidently to the minima, with more variations that can be negligible.



With constant learning rate α

With decaying learning rate α



Remember: 1 epoch = 1 pass on all the training set data

$$\alpha = \frac{1}{1 + \text{decay rate} * \text{epoch num}} * \alpha_0$$

with $\alpha_0 = 0.2$ and decay rate = 1

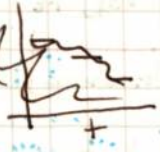
epoch #	α
1	0.1
2	0.67
3	0.5
4	0.4

Other learning rate decay methods

Exponential decay ($0 < \alpha < 1$): $\alpha^{\text{epoch num}} * \alpha_0$

or $\alpha = \frac{k}{\sqrt{\text{epoch num}}} * \alpha_0$

or $= \frac{k}{\sqrt{t}} * \alpha_0$



Or discrete staircase by halving each iteration

[Hyperparameter Tunning]

On DNN, as we know, we have to tune up ~~on~~ the ~~ma~~ hyperparameters, which are:

- Learning rate α
- Momentum β → 0.9
- Adam's β_1, β_2 and ε → 0.9, 0.999, 10^{-8}
- # layers
- # hidden units
- learning rate decay
- mini-batch size

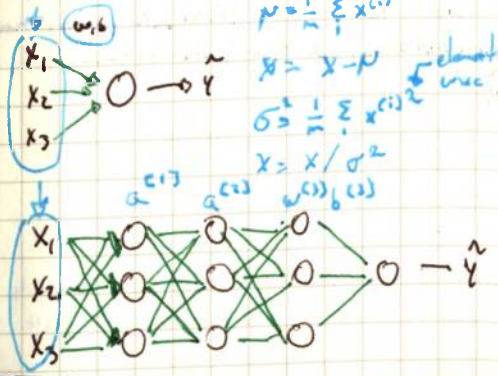
In order of recommendation

1. Learning rate α
2. Momentum β
2. Mini-batch size
2. # hidden units
3. Number of layers
3. Learning rate decay
4. Adam parameters (defaults are ok)

- try random parameters

[Batch Normalization] - batch norm

- Makes the hyperparameter search much easier, much more robust
- In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization, created by two researchers, Sergey Ioffe and Christian Szegedy. Batch normalization makes our hyperparameter search problem much ~~faster~~ easier, makes our NN much more robust.
- The choice of hyperparameters is a much bigger range of hyperparameters that work well and will also enable us to much easily train even very deep neural networks.
- Similar as it was done on inputparameters, when we normalized them in order to turn the elongated contours of our learning problem into more "round", which also led the gradient descent algorithm to be optimized
- The same principle applies on each hidden layer, where the activations can be normalized as well, to make the next layer more efficiently.



Can we normalize $a^{[2]}$ so as to train $W^{[3]}, b^{[3]}$ faster ($k \neq 1$)

In practice, what is commonly normalized is z, not A, one or the other but z is

[Implementing Batch Norm]

Given some intermediate values in the NN $z^{(1)} \dots z^{(m)}$ (layer, sample) $\rightarrow z^{[l](i)}$

$$\mu = \frac{1}{m}\sum_i z^{(i)} \quad \text{- mean (u)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2 \quad \text{- variance}$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$ to avoid NaN, if $\sigma^2 = 0$

So now we have every component of z with mean 0 and variance 1, but we don't want the hidden units to always have mean 0 and variance 1. Maybe it makes sense for hidden units to have a different distribution, so for this we'll compute $\tilde{z}$ hidden:

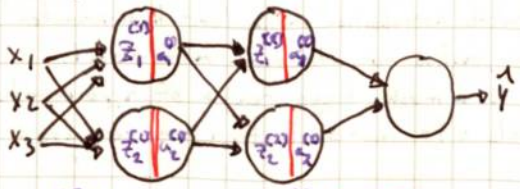
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

γ, β: learnable parameters of the model

the effect of γ and β is that allows us to set the mean of $\tilde{z}$ to be whatever we want it to be.

- So, if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, they would equal the denominator and denominator, then the effect of $\gamma z^{(i)}_{norm} + \beta$ will be inverting the equation, then $\tilde{z}^{(i)} = z^{(i)}$, it's computing essentially the identity function.
- To fit it into the neural network, instead of $z^{(i)}$, we use $\tilde{z}^{(i)}$ for the later computations of the neural network.

[Adding Batch Norm to a network]



$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow \dots$

Using the normalized values

- In practice we don't need usually to implement Batch norm, because the ML packages already have this available, for example with Tensor Flow: tf.nn.batch_normalization
- In practice it is implemented with mini batches, and the batch normalization looks only on the current mini batch!

$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow g^{[1]}(\tilde{z}^{[1]}) \rightarrow a^{[1]} \rightarrow \dots$

$X^{\{2\}} \rightarrow z^{[1]} \xrightarrow[BN]{} \dots \rightarrow \tilde{z}^{[1]} \rightarrow \dots$

The term b gets cancelled as we apply normalization and all z's have similar distributed values, so the bias element becomes a constant and we know it won't change the outcome.

[Implement Gradient Descent with Batch Norm]

For $t = 1 \dots$ num_minibatches

compute forward prop on $X^{\{t\}}$

in each hidden layer use batch norm to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

use back prop to compute $dW^{[l]}$, $db^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

update parameters:

$$\left.\begin{aligned} W^{[l]} &:= W^{[l]} - \alpha\, dW^{[l]} \\ \beta^{[l]} &:= \beta^{[l]} - \alpha\, d\beta^{[l]} \\ \gamma^{[l]} &:= \gamma^{[l]} - \alpha\, d\gamma^{[l]} \end{aligned}\right\} \text{G.D.}$$

Works with momentum, RMSProp, Adam

[Batch Norm as regularization]

- Each minibatch is scaled by the mean/variance computed on just that mini-batch.
- Adds some noise to the values of $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

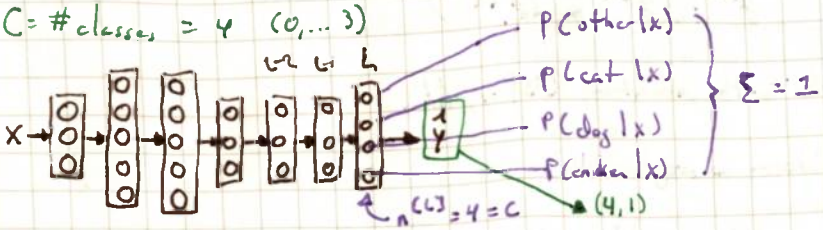
[Batch Norm at Test Time]

On the test samples we won't have access to the training minibatches to compute the usual mean and variance on each minibatch. To accomplish it, we can compute the exponential weighted average, to keep track of the latest average value, which in turn will be used as μ and σ^2, but with running average at test.

{Multiclass Classification}

In difference of logistic regression, that we used to do binary classification there is a generalization of logistic regression called **Softmax** regression, that lets make predictions where we want to recognize 1 to C classes, for example if we want to recognize a set cats[1], dogs[2], chickens[3] and others[C].

$C = \#classes = 4 \quad (0, \dots 3)$



The standard way model for getting your network to do this, uses what's called a Softmax layer in order to generate these outputs.

- In the final layer of the neural network, we are going to compute as usual the linear part of the layer: $z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$. Now having computed z, we need to apply what's called the Softmax activation function.
- The activation function is a bit unusual for the softmax layer, but this is what it does:

Activation function: compute temp variable t

$t = e^{(z^{[L]})} \qquad (z = (4,1)) \therefore t = (4,1)$ [element wise]

$$a^{[L]}_{(4,1)} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i}, \quad a^{[L]}_i = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

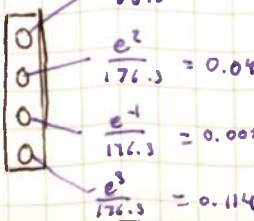
Applied for example:

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \qquad \sum_{j=1}^{4} t_j = 176.3$$

$$a^{[L]} = \frac{t}{176.3}$$

$$a^{[L]}_{(4,1)} = g^{[L]}(z^{[L]}) \quad \hookrightarrow (4,1)$$



[Deep Learning Frameworks]

At this moment we've learnt how to implement almost from scratch deep learning algorithms. Now, it turns out that is not optimal to implement all of them by scratch everytime but it's a better approach to use frameworks that are out there, already optimized and growing month by month. A list of the leading frameworks available is:

- Caffe/ Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- Paddle Paddle
- Tensorflow
- Theano
- Torch
- Accord (.Net)

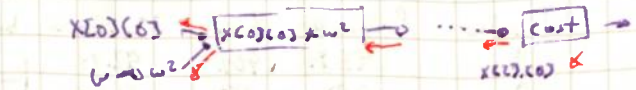
Criteria to choose:
- Ease of programming: Desk, testing, deployment
- Running speed :
- Truly Open : Not only open source, but also good governance

[Code example for Tensor Flow]

```
import numpy as np
import tensorflow as TF

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0], dtype=tf.float32)
x = tf.placeholder(tf.float32, [3, 1])
cost = x[0][0]*w**2 + x[1,0]*w + x[2][0] # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x: coefficients})

print(session.run(w))
```

susceptible, i.e. Adam

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```



Course 3 : ML Strategy

[Orthogonalization]

Orthogonal means 90°, like on a hyperplane ⊥

Chain of assumptions on ML

- Fit training set well on cost function (≈ human-level performance) — bigger NN / Adam
- Fit dev set well on cost function — Regularization / Bigger training set
- Fit test set well on cost function — Bigger dev set
- Performs well in real world — change the dev set or cost function.

Orthogonalization in ML: its about tweaking the right hyperparameters to obtain the right result. Moving or altering one HP to reduce or maximize a particular effect should be avoided. Each HP is designated to fulfill a specific situation and problem, therefore is not good to mix them to get a better result, (ie, tuning the regularization parameter and increasing the ~~regularization~~ optimization algorithm depending on a single "knob".

[Single Numeric Evaluation Metric]

Having a single measure unit to evaluate the performance of the model is to have of course some iterations and evaluate them. For example using Precission and Recall:

Precission: Of the correctly identified samples, what percentage of them are from the whole set, (ie, if a set has 100 pictures, 95 are of cats and 5 are something else. If the 95 cats are properly recognized then we have a precission of 95% because the 95% of the set was properly recognized

Recall : From the sample above, if only 90% of the cats were recognized, (that's 90% of the 95 cats on the set), then the percentage of actual cats recognized would be 94.7%.

Away to combine them into one single number to measure the performance is to use the F1 score or "harmonic mean".

F1 Score will allow us to choose the best tradeoff between Prec. and Rec.

$$\frac{2}{\frac{1}{P}+\frac{1}{R}}$$

Classifier	Precission	Recall	F1 Score
A	95%	90%	92.4%
B	98%	85%	91.0%

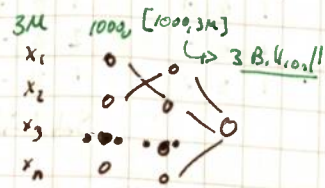
Course 4 : Convolutional Neural Networks

Computer Vision Problems

Image classification : Cat or not?
Object detection : Figure out the position
Neural Style transfer : With content image and a style image, merge them.

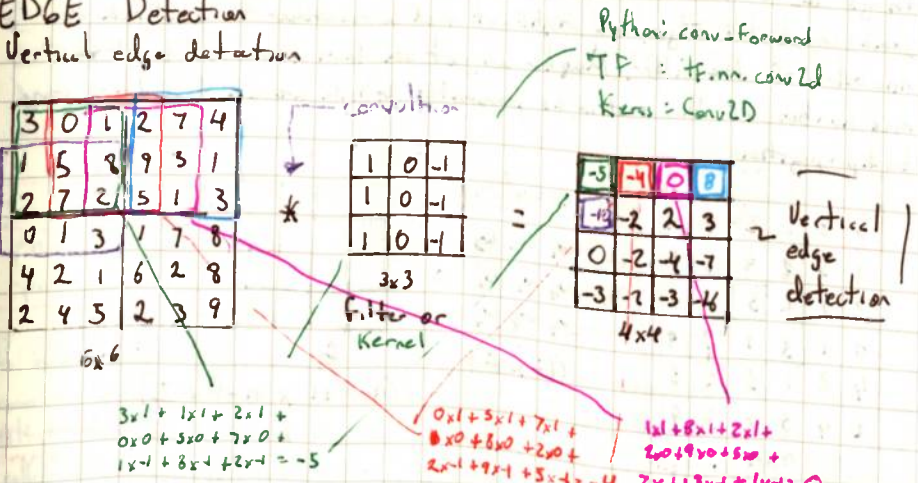
Challenges : Inputs can get really big! In the previous problems we have worked on samples of 64x64 x3 images (12288 pixels), but that's very small. Working with larger images (ie, 1000x1000), we end up with 3 million pixels to process (1000x1000x3)

Passing the 3M parameters to a 1000 units hidden layer, would imply to have a (1000 x 3M) matrix which means the matrix will have 3 billon parms.
This causes overfitting and computational requirements become very high

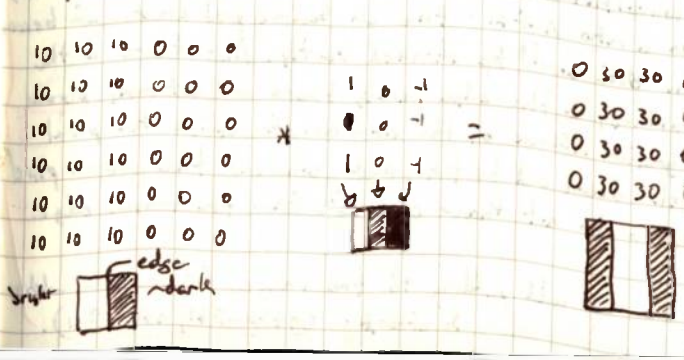


EDGE Detection

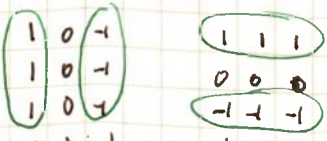
Vertical edge detection

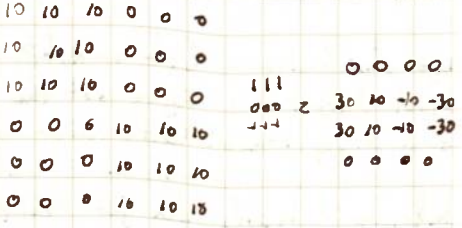


Simplified example:



Vertical and horizontal edge detection

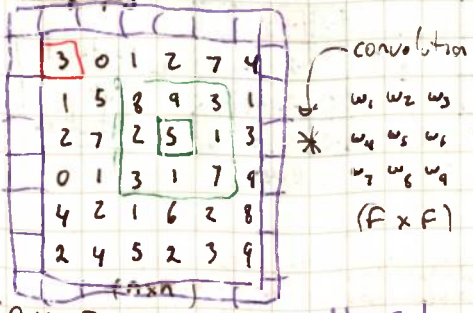




Learning to detect edges

1 0 -1	1 0 -1	3 0 -3
1 0 -1	2 0 -2	10 0 -10
1 0 -1	1 0 -1	3 0 -3
	Sobel filter	Scharr filter

There has been large debate among Computer Vision researchers about the values on the filters to use. But now, instead of hand picking the values, with the rise of Deep Learning, one of the things we've learned is that when we really want to detect edges in some complicated edge, we might not need a researcher to hand pick the nine values of the filter. Maybe we just can learn them and treat the nine numbers of this matrix as parameters which we can then learn using backpropagation



[Padding]

p = padding = 1

Same size as original

As we've seen, doing convolution does shrinks the data, after each hidden layer, so it happens on a ratio n-f+1; thus can become problematic if because after several layers the data would be shrinked vastly. Also there is the problem that taking convolutions, on the edge pixels, there are not many pixels around, so it causes greater influence on the convolution, for example the pixel in red has only pixels down and to the right, whereas the green pixel has neighboring pixels. So to not throw away info from the image's edge.

To fix the problems, we can add padding to the image, for example adding one pixel around. Like in blue color. So after *: $(n+2p-f+1, n+2p-f+1)$.

Valid and Same convolutions:

"Valid": no-padding, $n \times n * f \times f \rightarrow (n-f+1 \times n-f+1)$

$6 \times 6 * 3 \times 3 \rightarrow (4 \times 4)$

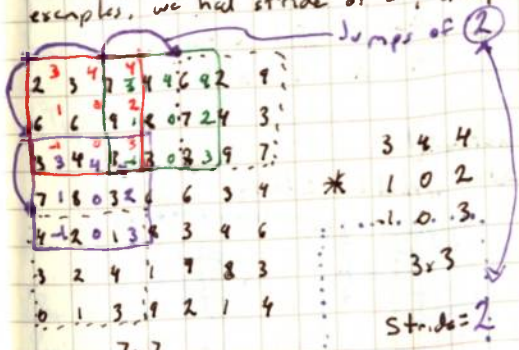
"Same": Pad, so that the output size is the same as the input size

$(n+2p-f+1 \times n+2p-f+1) * (f \times f) = n \times n$

$p = \frac{f-1}{2}$ by convention in CV, p should be odd

[Strided Convolutions]

It's another piece of the basic building blocks of convolutions as used in Convolutional Neural Networks. What basically means is that we define as stride, the number of steps to "jump" on the previous convolution examples, we had stride of 1, example



$$\begin{bmatrix} 91 & 100 & 83 \\ 69 & 91 & 127 \\ 44 & 72 & 74 \end{bmatrix}$$

$$\left\lfloor \frac{n+2p-f}{2} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{2} + 1 \right\rfloor$$

$$\frac{7+0-3}{2} = \frac{4}{2} + 1 = 3$$

There is a convention to $\lfloor \text{floor} \rfloor$ the dimensions, so they are not decimal.

[Summary]

- $n \times n$: image
- padding: p

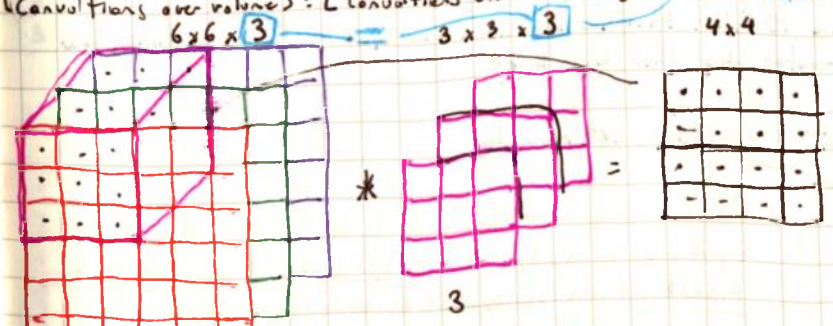
$f \times f$: filter

stride : s

$$\left\lfloor \frac{n+2p-f}{s} \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} \right\rfloor$$

In math textbooks, this convolution is reffered as "cross-correlation", and convolution (the real one), is defined as flipping horizontally and vertically the filter. However in the DNN literature and community, the "cross-correlation" is called convolution, as the result is practically the same and it saves coding time.

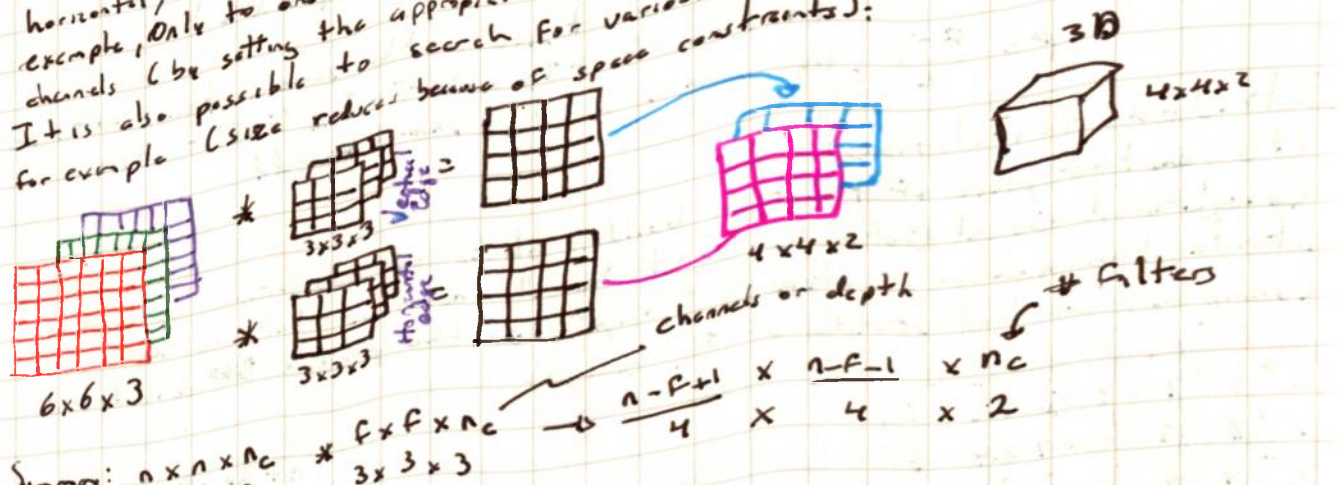
[Convolutions over volume]: [Convolutions on RGB Images]



[Multiple Filters]

As we learnt before, it's possible to use the filters horizontal, diagonal or in a specific "degree". The filters can example, only to one color channel (on a 3 channel image), or to all channels (by setting the appropiate channel's values on the filter).

. It is also possible to search for various edges (if we stack the filters for example (size reduced because of space constraints):

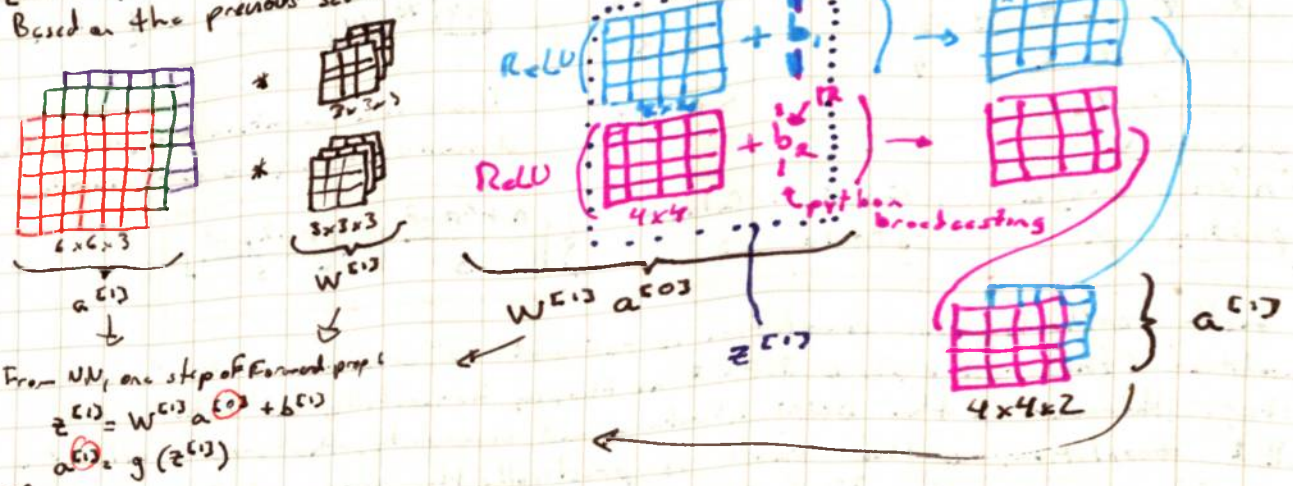


Summary: $n \times n \times n_c * f \times f \times n_c \rightarrow \frac{n-f+1}{4} \times \frac{n-f+1}{4} \times n_c$

$6 \times 6 \times 3 \quad 3 \times 3 \times 3 \qquad 4 \times 4 \times 2$

[One layer of a convolutional neural network]

Based on the previous section where we've applied the filters:



From NN, one step of Forward prop:

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = g(z^{[1]})$

- Number of parameters in one layer

If we have 10 filters that are $3 \times 3 \times 3$ in one layer of a neural network, how many parameters does that layer have?

$3 \times 3 \times 3 + 1$ (bias) $= 28$

$28 \times 10 = 280$ parameters.

One nice thing about this, is that doesn't matter how big the input image is, 5K x 5K, 1K x 1K; the number of parameters remains fixed! It can be applied to very large images.

[Summary of notation]

$f^{[l]}$ = filter size
$p^{[l]}$ = padding
$s^{[l]}$ = stride
n_c = number of filters

Input : $n_h^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$
Output : $n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

Volume : $n_h^{[l]} = \left\lfloor \frac{n_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$

$n_w^{[l]} = \left\lfloor \frac{n_w^{[l-1]} + 2p^{[l]} + f^{[l]}}{s^{[l]}} + 1 \right\rfloor$

Each filter is: $f^{[l]} \times f^{[l]} \times n_c$

Activations: $a^{[l]} \to n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$
↓ vectorized
$A^{[l]} \to m \times n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l]}$ ← filters in layer l

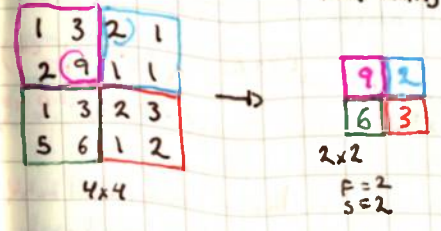
bias: $n_c \to (1, 1, 1, n_c^{[l]})$

Why Convolutions:
- Reduces drastically the amount of parameters into resemble seen
- Parameter sharing: A feature detector (such as a vertical edge detector) that is useful in one part of the image, is probably useful in another part as well.
- Sparsity of connections: In each layer, each output value depends only on a small number of inputs.

[Types of layers in a convolutional network:]
- Convolution (Conv)
- Pooling (pool)
- Fully Connected (FC)

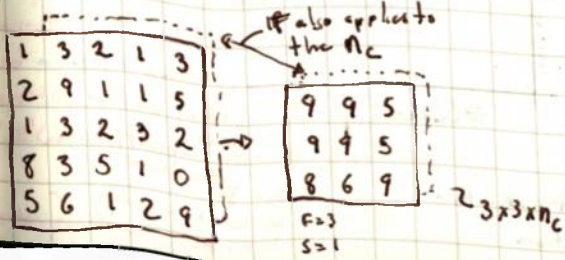
[Pooling Layers]
Pooling is used to shrink the output from very large channels; the principle is very similar to Conv, where a filter is passed over the pixels, however instead of multiplying and adding, only the "max" is taken. As Conv, it also has a filter size, and a stride; the padding is practically never used (but is possible to add).
An example

Max Pooling



There is another version called Average pooling, which does what we can deduct, instead of taking the max

A common setting is:
$f=2, s=2$ or $f=3, s=2$



[Classic networks]

LeNet-5 : • trained on grayscale images (32x32x1)
• 60K parameters.
• tanh, nw ↓ n_c ↑
• conv → pool → conv → pool FC, FC, output

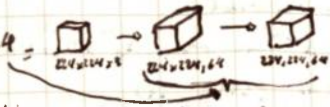
• sigmoid + tanh
• Le Cun et al, 1998
• read chapter II and ~III

AlexNet: • trained in color, RGB (227x227x3)
• Much bigger than LeNet but similar, 60Million params
• conv - pool - conv - pool - conv - conv - conv - pool - FC - FC - FC - softmax
• Local response normalization (LRN)

• ReLU
• Krizhevsky et al, 2012 (Important)
• Multiple GPUs

VGG-16 (or 19)

• Conv = 3x3, s=1, same padding
• max pool = 2x2, s=1
• RGB (224, 224, 3)

Conv 64 x2 = 

• 138 Million params • 16 = layers
• number of channels Doubles on each stack of layers

• Conv 64x2 → pool 64 → conv 128x2 → pool 128 → conv 256x2 → pool 256 → conv 512x3 → pool 512 → conv 512x3 → pool 512 → FC → FC → Softmax • Simonyan & Zisserman 2015, Very Deep Conv.

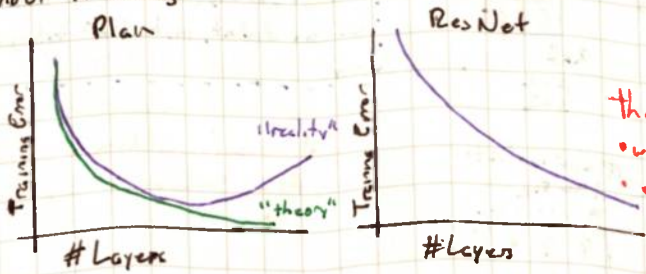
[Residual Networks - ResNets]

Very Very deep networks are difficult to train because of vanishing and exploding types of problems. To avoid that, is possible to take the activation from one layer and suddenly feed it into another layer even much deeper in the NN.

- The standard 'plain' version are known, follows a succession of:

$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]} \rightarrow a^{[l+1]} = g(z^{[l+1]}) \rightarrow z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]} \rightarrow a^{[l+2]} = g(z^{[l+2]})$

Behaviour in training:



$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

$= g(W^{[l+2]} a^{[l+1]} + b^{[l+2]} + a^{[l]})$

the dimensions should match
• with "same" padding or
• extending the dimension on the activation by adding another matrix of weights (multiplication) "W_s"

$W_s = \mathbb{R}^{(proj/empty)}$
→ learned parameter or
→ ...

He et al., 2015. Deep Residual networks for image recognition

... in Networks → 1×1 convolutions]

- Lin et al., 2013. Network in Network

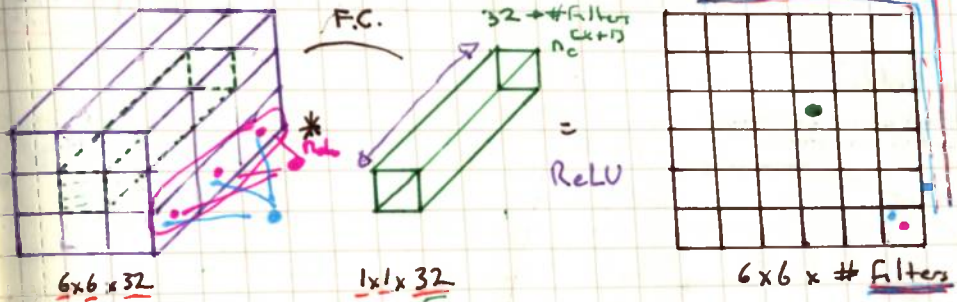
Filters as we have seen are larger, ie, (2×2), (3×3), and this one is just a real number (1,1); on first glance it doesn't seem useful as to what it does is just multiply the original value times the filter, however it makes sense while working with several channels; what it will actually do, is to look at each of the different positions and take an element wise product within the number of channels of the original volume and the volume of the filter.

1	2	3	6	5	8
3	5	5	1	3	4
2	1	3	4	9	3
4	7	8	5	7	9
1	5	3	7	4	8
5	4	9	8	3	5

6×6×1

* [2] =

2	4	6	...	...	...
...	...	...	...	...	...

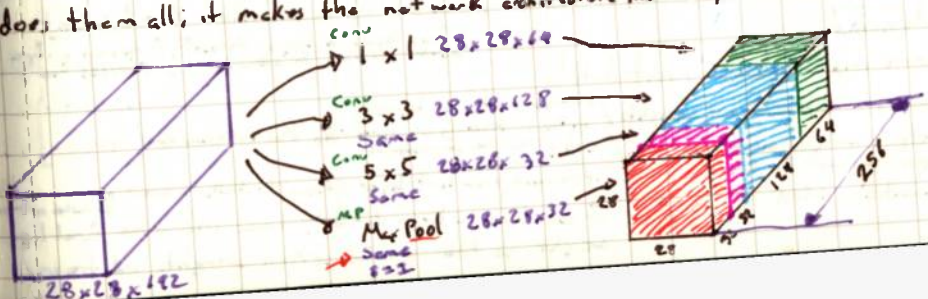


This idea has been very influential on several NN architectures, including the Inception Network.

- If we want to reduce the height and width, we can use a pooling layer, but if the number of channels got big, then we can use a # of 1×1 filters to shrink the number of channels, if needed, otherwise can be the n_c kept equal.

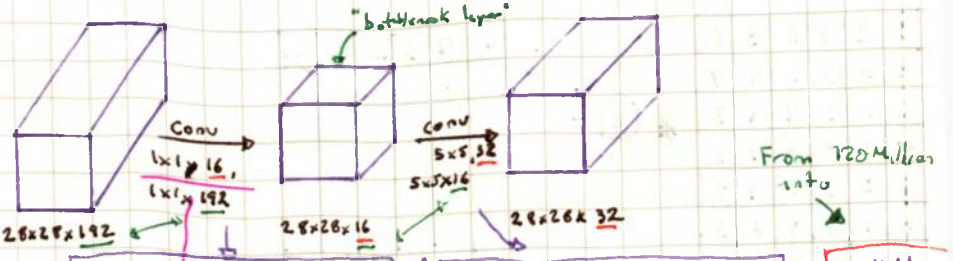
[Inception Network Motivation]: Szegedy et. al. 2014: Going Deeper with convolutions

When designing a layer for a ConvNet, we might have to pick if we want a 1×1 or 3×3 or 5×5 filter, or if we want a pooling layer... What the inception network does, is that does them all; it makes the network architecture more complicated but works remarkable well!



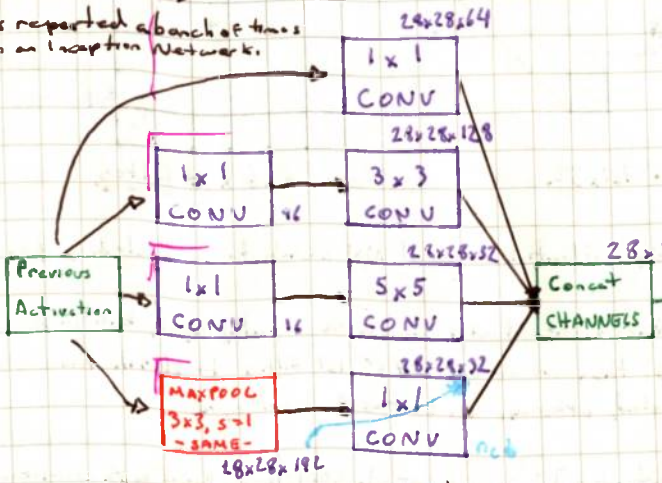
- The problem of the computational cost.
- Taking from the previous example the convolution 5×5, 32, where the original parameter sample of 28×28×192 we have
Input = 28×28×192 → Conv 5×5, 32 (same) → 28×28×32; this mean we have: 32 Filters 5×5×192 ; which give up the multiplication of (28×28×32)×(5×5×192), which in turn equals to 120,422,400 ≈ 120 Million of ~~output values~~ multiplications.

We can use a 1×1 convolution to reduce the amount of computation needed, we input the volume, use a 16×2 convolution, to reduce the volume, and then run the 5×5 convolution.



[Inception Module]

Is reported a bunch of times on an Inception Network.

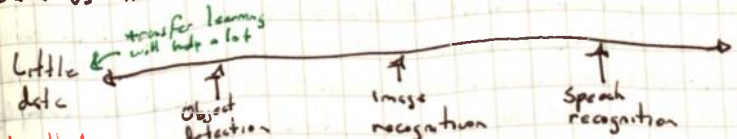


On the Final layers, are added softmax outputs layers that help to regularize the outputs, helps to ensure the features computed are on hidden/intermediate layers are not too bad to predict the class on the output layer. An inception network is called GoogLeNet.

[Practical Advices]

- Use open source implementations: contribute!
- Transfer learning: use pre-trained weights (if you have a large dataset, freeze only a few layers at the hidden units (freeze less))
  - Freeze pretrained layers
  - retrain / initialize parameters
  - precompute the last layer output to avoid processing the entire network
  - if we have a huge dataset, then use the final layer's weights as the initialization weights and train.
- To seriously encouraged when we have a huge dataset for training
- PCA color augmentation (RGB)

Data vs Hand Engineering



Many 'hand-engineering' ('hacks') } hard skilled task to locating objects and it's bounds

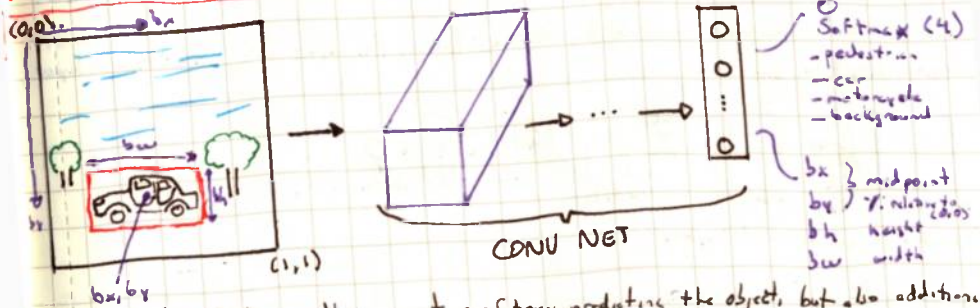
The sources of Knowledge: Labeled data (x, y); Hand engineered features/network architecture/other components

-Week 3 - Object Detection -

It's one of the areas of computer vision that is exploding and working much better than just a couple years ago. We already know and do object classification (telling if in a picture is a cat, dog, etc), but now in the detection, it's needed to locate one or many objects and their locations in boundings boxes, where in the picture are.

- Classification with localization

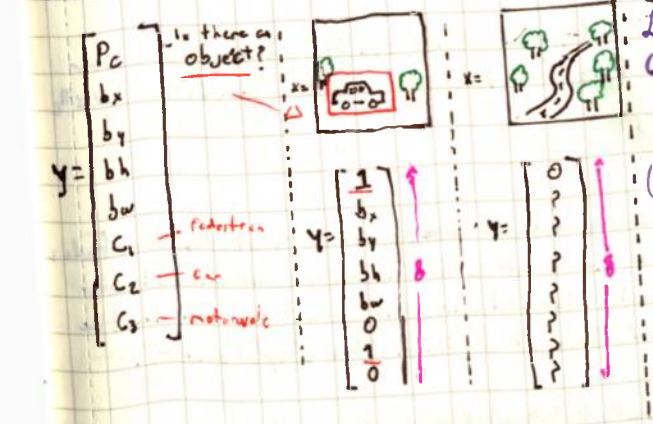
bh and bw can be ≥ 1



The idea is to output from the convnet a softmax predicting the objects, but also additional parameters identifying the object location (b_x, b_y = midpoint of the object, from origin) and it's height and width (b_h, b_w), so we can calculate it's bounding box.

Need to output: b_x, b_y, b_h, b_w, class label (1-4)

- Defines the target label (y) ← Class: 1 = pedestrian, 2 = car, 3 = motorcycle, 4 = other



Loss Function:

$$\mathcal{L}(\hat{y}, y) = (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \dots + (\hat{y}_8 - y_8)^2 \quad (\text{if } y_1 = 1)$$

$$(\hat{y}_1 - y_1)^2 \quad (\text{if } y_1 = 0)$$

[Landmark Detection]

On the previous section we saw how we can get a NN to output 4 numbers (b_x, b_y, b_h, b_w) to specify the bounding box of an object we want the NN to localize. For landmarks, we want a NN to output (x, y) of numbers of important points of an image. For example, if we want to detect the corners of the eyes or the landmarks on a face:

(l_x, l_y)

l_{1x}, l_{1y}
l_{2x}, l_{2y}
l_{3x}, l_{3y}
l_{nx}, l_{ny}

→ ConvNet → $\begin{bmatrix} 1 \\ l_{1x} \\ l_{1y} \\ l_{2x} \\ l_{2y} \\ \vdots \\ l_n \end{bmatrix}$ → Face?

129 output units

It we have 64 landmarks then the NN should output 129

Used for AR CGI, detect mo...

- landmarks should not be consistent across different images -

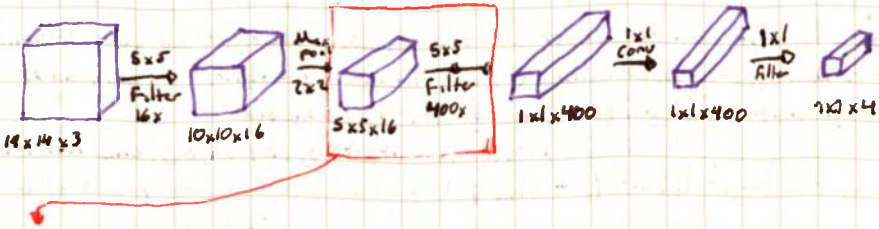
[Sliding Window Object Detection]

Although it is computationally inefficient, it is possible to detect objects on a picture by sliding small to big "squares" which are one by one fed into a ConvNet to identify if there is an object or not. For this the ConvNet should be trained with close ups of the object we want to detect  . This sliding windows are not recommended, too slow.

[Convolutional Implementation of Sliding Windows]: Turning FC layers into convolutional layers

Original detection algorithm:

Turned into convolutional layers



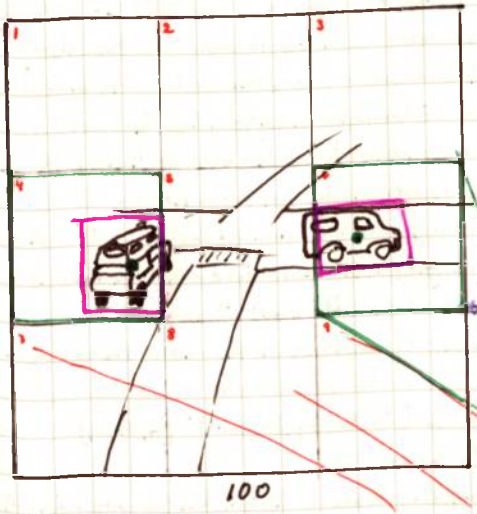
Each 5x5 filter has 16 channels, and applying it into the 5x5 volume, it will cover entirely the volume (no need to stride), so the output is going to be $1 \times 1 \times n_c$

If we input larger images, the conv net will ~~take~~ scan it with stride, the stride is defined or ruled by the Maxpool size. So with a MP of 2, the conv net will slide over the image with strides of 2, and at the last filter, the convnet will mark on the output volume, in which "window" the object was found. (See stapled paper).

- Sermanet et. al, 2014, Overfeat: Integrated recognition, localization using ConvNets

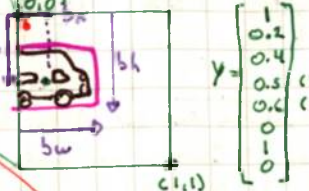
[Bounding Box Predictions]: YOLO algorithm: You Only Look Once, Redmond et al 2015 with the convolutional implementation of sliding windows, we've found a computationally efficient way to output bounding boxes, however is ~~not~~ could not be allways accurate; because it might happen that the sliding window's stride skips a crucial element of the object, or it was too small to fit it, or the object was wider.

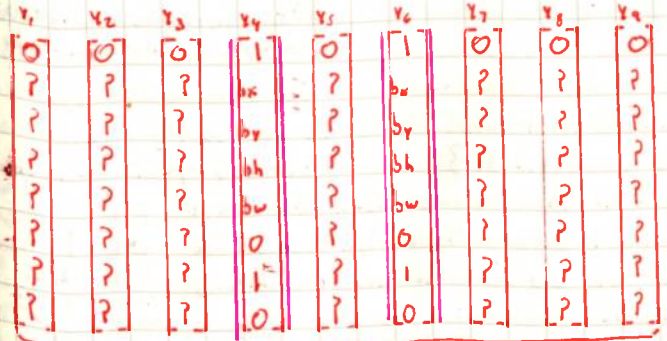
- It works by placing a grid over an image, and the basic idea is to take the image classification and localization algorithm, and apply it on each of the grids of the image. For labeling the image, we specify a label Y which is a vector containing as before: $[p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]^T$, this for each grid cell.



For this sample we assume a (9x9) grid, but finer grids are used in reality such as (19x19).

Also here we look at the midpoint of the object, but Relative to the grid cell where it is located!

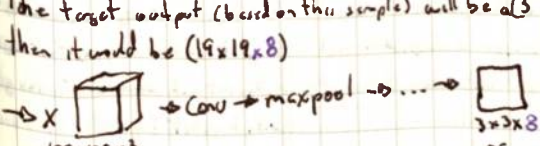




? = numbers that will be ignored (don't care)

Labels for training for each grid cell

The target output (based on this sample) will be a (3 x 3 x 8) volume. With grids of 19x19, then it would be (19x19x8)



This is one of the harder papers to understand, even for senior researchers

[Intersection over Union : IoU]

How to tell if the object detection algorithm is working well?
Having an image with the ground truth bounding box and if our algorithm outputs this bounding box... is this a good or bad outcome?

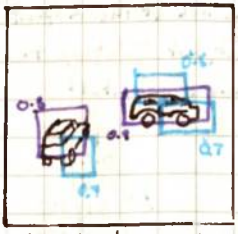
$$IoU = \frac{\text{Size of } \square}{\text{Size of } \square}$$

"Correct" if IoU ≥ 0.5 ... | or 0.6, 0.7 ...
'perfect' if IoU = 1

IoU: is a measure of the overlap between two bounding boxes.

[Non-max Supression]

With finer grids it's almost always sure that the algorithm may find multiple detections of the same objects, Non-max Supression is a way to make sure the algorithm detects each object only once. This because while running the identificators, each cell of the grid might 'think' that it has the center of the object and output a high P_c



Multiple detections of each object

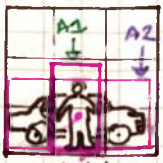
What Non-max supression does, is that cleans up these up the detections so that we end up with one detection per object.
It looks at the probabilities associated with each of the detections P_c
It takes the largest one (the most confided one), then looks at the remaining rectangles with a high overlap (IoU) and supress them.
Non-max means, it's going to supress the rest of the rectangles (max

Note: P_c is evaluated per each class, in our examples, P_c has 3 object

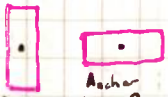
Steps: 1: Each output prediction is $[P_c, b_x, b_y, b_h, b_w]^T$ $[C_1, C_2, C_3]$
2: Discard all boxes with $P_c \leq 0.6$
3: While there are any remaining boxes: a) Pick the box with largest P_c as predic
b) Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

[Anchor Boxes] (Note: choose between 5 to 10 classes; or K-means to choose representative shapes)

What if a grid cell wants to detect multiple objects? We can use the idea of anchor boxes. Below a simplified 3x3 grid for illustration purposes. We can see that both objects share the same midpoint and the same cell.



If we want to detect multiple objects, we can use anchor boxes and define the class label containing as well with anchor boxes:



# of Labels

Normal: $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$

Can't detect more than one object at the same time.

With anchor boxes 1 and 2: $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$ (Anchor box 1 / Anchor box 2)

Anchor Box algorithm:

Previously:
Each object in training image is assigned to grid cell that contains that object's midpoint.
Output y: $3 \times 3 \times 8$ (8 = possible bounding box "classes")

With two anchor boxes:
Each object in training image is assigned to grid cell that contains that object's midpoint and anchor box for the grid cell with highest IoU (grid cell, anchor box from the order of the anchors)
Output $3 \times 3 \times (8 \times 2)$

[Course 4 - Week 4 : Face Recognition and Neural Style Transfer]

[Face Recognition]

1) Face Verification: - Input image ✓, name/ID — 1:1
Building block for ↓
- Output: whether the input image is that of the claimed person

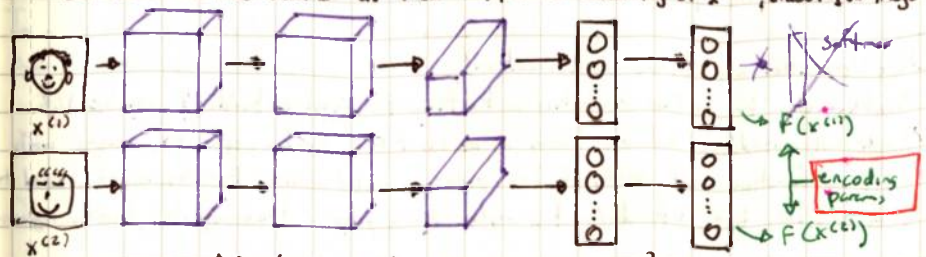
2) Recognition : - Has a database of K persons — 1:K
- Get an input image
- Output ID if the image is any of the K persons (or not recognized)

[One Shot Learning]

What it means, is that for most Face recognition applications, we need to be able to recognize a person given just one example of that person's face. And historically, Deep learning algorithms don't work well if we have only one training example. To address this, we can't rely on a DNN with softmax output layer, as this would require to retrain and add the additional unit. The approach on this case is to have a function to compute the similarity between two images, that means, compare the image on the database vs the image taken on the moment. If the difference is small, then the person is the same, whereas a large number would indicate the person is not the same.

[Siamese Network]: the similarity function.

We are used to seeing pictures of ConvNets where we input an image and trough a sequence of convolutional, pooling and FullyConnected layers, we end up with a feature vector, and sometimes is fed up to a softmax unit to make classification. On the case of Siamese Networks, we focus on the last vector of let's say 128 numbers computed by some FC layer that is deeper in the network.
We can refer to this vector as $f(x^{(i)})$, or as "encoding of $x^{(i)}$", encoding of image 'i'



$$\text{diff}(x^{(1)}, x^{(2)}) = \left\| f(x^{(1)}) - f(x^{(2)}) \right\|_2^2$$

the goal is to find a set of parameters that we can then use to find the diff, and if the difference is small between images of the same subject, we have it!
As we vary the parameters, we end up with different encodings, and what we can do is use back propagation, to and vary all the parameters to ensure that the conditions are satisfied.
Taigman et al 2014. Deep Face closing the gap to human level performance

[Triplet Loss]

One way to learn the parameters of the neural network so that it gives a good encoding for pictures or faces, is to define and apply gradient descent on the triplet loss function.

To apply the triplet loss, is needed to compare pairs of images. For example given a picture (A) as Anchor, compare it with another image of the same object which will be (P) positive, and another from someone else, which will then be (N) negative sample. The goal is to have pairs (A, P) and (A, N), we want the encodings to be very similar for AP, whereas for AN, to be large.

Loss function: Given 3 images A P N

- $\mathcal{L}(A,P,N) = \max\left(\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2, \alpha\right)$ — α - margin

Cost function:

$$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

Of course it's needed to have multiple pictures of the same person, ie, 10 per person so we can have for example:
Training set of 10K pictures of 1K persons

After being trained with enough data, then it can be possible to use just one shot learn

Choosing the triplets A, P, N

- Avoid random election: During training, if APN are chosen randomly, then is very likely that $d(A,P) + \alpha \leq d(A,N)$, so the condition will be easily satisfied
- Choose triplets that are 'hard' to train on, choose those which similarity is very close so that $d(A,P) + \alpha \leq d(A,N)$ is forced to have better computations and push the quantities further away.

[Face Verification and Binary Classification]

Another way to do face verification is to take the final embeddings, maybe 128 dimensional embeddings and use them as an input for logistic regression, where with a sigmoid function we'll verify if the person is valid or not. For this is not needed to have triplets, but just different samples of a person, then apply logistic regress

To avoid the need to compute on real time, this is to have perhaps nightly precomputed embeddings and run the network with the precomputed values. next day.

$$\hat{y} = \sigma\left(\sum_{i=1}^{n} w_i \left|f(x^{(i)})_k - f(x^{(j)})_k\right| + b\right)$$

Schroff et al., 2015, FaceNet: A unified embedding for face recognition and cluster

[Neural Style Transfer]

The goal is to Generate an image based on the Style of one of them and when it's being applied to the Content (source) image.

[Content Cost Function]

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

- Use a hidden layer l to compute content cost — somewhere in the middle - not too shallow, not too deep
- Use pre trained ConvNet (eg. VGG Network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content.

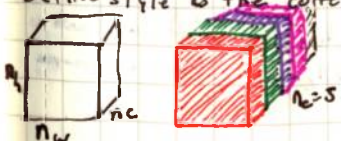
$$\hookrightarrow J_{content}(C,G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

(unrolled into vectors) → element-wise sum of square difference between content in layer l for C and G

[Style Cost Function]

Say we are using layer l's activations to measure "style"

Define style as the correlation between activations across channels.



→ compare channel by channel on both 3D matrices! - check stapled page.

unnormalized co-variance

Style Matrix: Let $a^{[l]}_{i,j,k} = \text{activation at } (i,j,k)$. $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

$G^{[l]}_{kk_1}$ for 1 image:

$$G^{[l](S)}_{kk_1} = \sum_{i=1}^{n_h^{[l]}} \sum_{j=1}^{n_w^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'} \qquad G^{[l](G)}_{kk_1} = \sum_{i=1}^{n_h} \sum_{j=1}^{n_w} a^{[l](G)}_{ijk} \, a^{[l](G)}_{ijk'}$$

Style Matrix

In linear algebra = gram matrix

$$J^{[l]}_{style}(S,G) = \frac{1}{2 n_h^2 n_w^2 n_c^2} \| G^{[l](S)} - G^{[l](G)} \|_F^2 = \frac{1}{2 n_h^2 n_w^2 n_c^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right)^2$$

$+ \beta$

To get more visually attractive results, works better to use the style cost func. from multiple different layers.

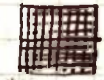
$$J_{style}(S,G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S,G)$$

Overall cost function:

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

1D and 3D Convolutions

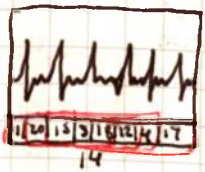
There are sources of 1 dimensional data, such as electrocardiograms, which also can be seen as a time-series problem. However it is possible to compute convolutions as well - For example:



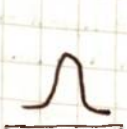
"2D 14x14"

2D 5x5 Filter x16

$14 \times 14 \times 3_c * 5 \times 5 \times 3_c \rightarrow 10 \times 10 \times 16$ } Common sample.



14

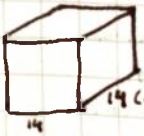


5 x16

$14 * 5 \rightarrow 10$

10×16

} 1d data convoluted with 1d filter and also with 16 channels.



14 14 14 (or other)

3D Volume

5 5 5

3D Filter 1 channel 16 channel

$14 \times 14 \times 14 \times n_c$ (16)

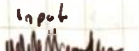
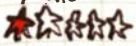
$* 5 \times 5 \times 5 \times n_c$ (16)

$\rightarrow 10 \times 10 \times 10 \times n_c$ (16)

} 3D data such as frames on a movie

- COURSE 5 - Sequence Models -

Sequence Models are one of the most exciting areas in Deep Learning; Models like recurrent Neural Networks or RNNs have transformed speech recognition, Natural Language Processing and other areas. Sequence Models can be useful in:

	input		output	
Speech Recognition		→	Text	- sequence
Music Generation	∅ or a integer	→	Music	- sequence
Sentiment classification	"there is nothing to like"	→		
DNA seq. analysis	AGCCCTGTGAGGAACTAG	→	AG CCCCTGTGAGG AACT AG	
Machine Translation	"hi how are you"	→	"Hola, ¿como estás?	
Video activity recognition		→	action on the video	
Name Entity Recognition	Yesterday, Harry Potter met Hermione Granger	→	Yesterday, Harry Potter met Hermione Granger	

Just to mention some...

All of these problems can be addressed as supervised learning with labeled data x, y; In some, both the input x and the output y can have different lengths, and in some of these examples only either x or y is a sequence.

Notation: As a motivating example, let's say we want to build a sequence model to input a sentence like this:

x = Harry Potter and Hermione Granger invented a new spell.

- And we want a model to tell us where are the people's names in this sentence. So this is a problem called Name-Entity recognition, and this is used for example by search engines to index, let's say, the last 24 hours of news, and looks for people names, so they can be indexed appropriately.
- Name-Entity recognition systems can be used to find people's names, company names, times, locations, country names, currency names and so on, on diff. types of text.

- Now, given the input x, let's say we want a model to output y that has 1 (ones) per input words, and the target output, the desired y tells for each of the input words if that's part of a person's name. Technically, this is not the best output representation, there are more sophisticated output representations that tells not only if it's a name, but tells you where are the start and ends of peoples names in the sentence:

1. $y = [1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0]$

2. $y = [(1 \quad 1) \quad 0 \quad (1 \quad 1) \quad 0 \quad 0 \quad 0 \quad 0]$

For this motivating example, we will stick with this simpler output representation. The input is a sequence of nine words, and we will use superscript $\langle t \rangle$ to denote or index positions. It implies one these are temporal sequences (whether the sequence is temporal or not). On this example we have $x^{\langle 1 \rangle} \dots x^{\langle t \rangle} \dots x^{\langle 9 \rangle}$ and we have also will have for the output the same notation $y^{\langle 1 \rangle} \dots y^{\langle \dots ? \rangle}$

X = Harry Potter and Hermione Granger invented a new spell. $T_x = 9$

$x^{\langle 1 \rangle}$ $x^{\langle 2 \rangle}$ $x^{\langle 3 \rangle}$ $x^{\langle 4 \rangle}$ $x^{\langle 5 \rangle}$ $x^{\langle 6 \rangle}$ $x^{\langle 7 \rangle}$ $x^{\langle 8 \rangle}$ $x^{\langle 9 \rangle}$

Y = 1 1 0 1 1 0 0 0 0 $T_y = 9$

$y^{\langle 1 \rangle}$ $y^{\langle 2 \rangle}$ $y^{\langle 3 \rangle}$ $y^{\langle 4 \rangle}$ $y^{\langle 5 \rangle}$ $y^{\langle 6 \rangle}$ $y^{\langle 7 \rangle}$ $y^{\langle 8 \rangle}$ $y^{\langle 9 \rangle}$

As we know, the length of X and Y, don't necessarily have to match! But on this example, both $T_x = 9$ and $T_y = 9$. Finally, to denote the current sample, we continue using (i), to refer to the ith training example. So:

$X^{(i)\langle t \rangle}$ = t'th element of the training example i; $T_x^{(i)}$ = sequence length for sample i

$Y^{(i)\langle t \rangle}$ = t'th element of the output sequence of train example i; $T_y^{(i)}$ = length of output sequence on i'th

This is our first serious foray into **NLP** or Natural Language Processing, and one of the things we need to decide is, how to represent individual words in the sequence

- To represent a word in a sentence, the first thing we do, is to come up with a vocabulary, sometimes also called a dictionary, and that means, making a list of the words that we will use in our representations. Starting with A ... till the end...

Vocabulary

a 1
aaron 2
and 367
harry 4075
potter 6830
zulu 10,000

One hot encoding:

$X^{\langle 1 \rangle}$ $X^{\langle 2 \rangle}$ $X^{\langle 3 \rangle}$ ... $X^{\langle 7 \rangle}$

4075 6830 367 10,000 words

Real world dictionaries contain from 30 to 50k or 100k words. Big internet companies have dictionaries even of 1 million words

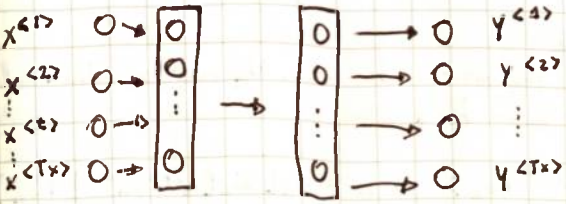
A way to come up with our own dictionary is to take the 'top n' most common words in our sentences and use only those.

If we encounter a word that is not present in our vocabulary, then we can create a new token or fake word, such as <unknown>, more later on

[Recurrent Neural Network]

Why not a standard network?

At first glance we would think on using a standard/naive NN to do the work, to learn the mappings between input words and entities on the output; even we could pad the input and output to match a consistent expected length of input data, however, this doesn't work well



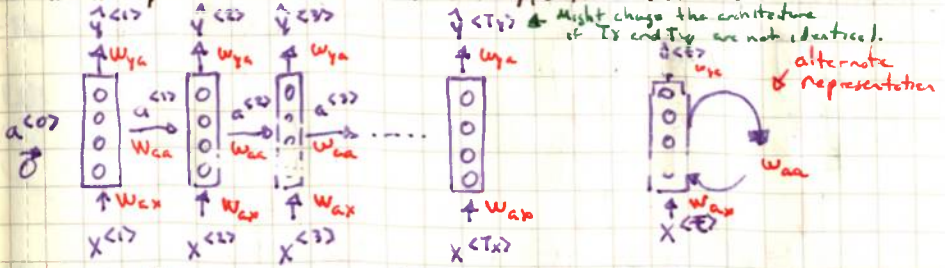
Problems: - Inputs and outputs can be different in different examples.
- Doesn't share features learned across different positions of text.

And the recurrent NN doesn't have these disadvantages.

[Recurrent Neural Networks]

In simple words, in a RNN, each $X^{<t>}$ is being fed into a layer, together with the activation of the layer before. On the first layer, a vector of zeros $\vec{0}$ or a randomly initialized vector is used.

This is repeated ~~until~~ for each time-step, until the last one.



W_{ax} = It's the same across each time step (shared), governs the comp. from $X^{<t>}$ to the hidden layer

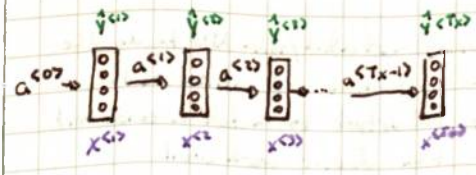
W_{aa} = Activations are governed as well by a shared parameter.

One weakness of the RNN is that uses only information that is earlier in the sequence to make a prediction, it uses past information, but doesn't uses information later in the sequence

- He said, "Teddy Roosevelt was a great President" ← Should be a name
- He said, "Teddy bears are on sale!" ← Should not be a Name

— we will solve this later with Bidirectional RNNs

Forward Propagation



$a^{\langle 0 \rangle} = \vec{0}$

$a^{\langle 1 \rangle} = g(W_{aa} a^{\langle 0 \rangle} + W_{ax} x^{\langle 1 \rangle} + b_a)$

$\hat{y}^{\langle 1 \rangle} = g(W_{ya} a^{\langle 1 \rangle} + b_y)$ ← sigmoid/softmax

$\vdots$

$a^{\langle t \rangle} = g(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a)$

$\hat{y}^{\langle t \rangle} = g(W_{ya} a^{\langle t \rangle} + b_y)$

Simplified RNN Notation:

A way to declutter or simplify the notation is to use a W_a matrix which will contain both, the second a, or the x values

$a^{\langle t \rangle} = g(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a) \rightarrow a^{\langle t \rangle} = g(W_a [a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_a)$

$(100, 100)$ $(100, 10000)$

$\hat{y}^{\langle t \rangle} = g(W_{ya} a^{\langle t \rangle} + b_y) \rightarrow \hat{y} = g(W_y a^{\langle t \rangle} + b_y)$

The simplification comes from stacking W_{aa} and W_{ax} $(100, 100)$ and $(100, 10000)$ respectively

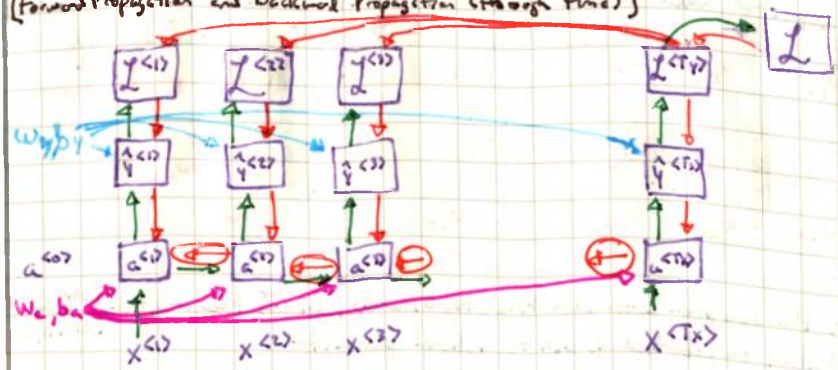
$100 \left\{ \left[W_{aa} \;\vdots\; W_{ax} \right] \right. = W_a$ $\quad (100, 10100)$

100 $\quad 10000$

$[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] = \begin{bmatrix} a^{\langle t-1 \rangle} \\ \cdots \\ x^{\langle t \rangle} \end{bmatrix} \begin{matrix} \} 100 \\ \} 10\,k \end{matrix} \quad 10100$

$[W_{aa} : W_{ax}] \begin{bmatrix} a^{\langle t-1 \rangle} \\ x^{\langle t \rangle} \end{bmatrix} = W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle}$

[Forward Propagation and Backward Propagation (through time)]



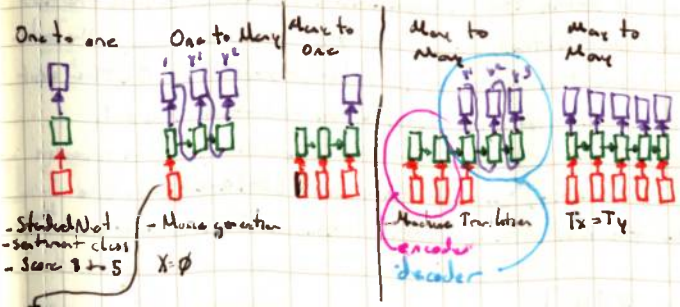
- Forward Propagation
- Backward Propagation
- Backward Propagation Through Time
- Parameters shared for a
- Parameters shared for $\hat{y}$

$$\mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) = -y^{\langle t \rangle} \log \hat{y}^{\langle t \rangle} - (1 - y^{\langle t \rangle}) \log (1 - \hat{y}^{\langle t \rangle})$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle})$$

[Different Types of RNNs]

We've seen so far RNNs where the number of input parameters match the number of output parameters, but that's not the case for different applications.



[Language Model and Sequence Generation]

What is language modelling? : What a language model does, is that given a sentence it's job is to tell you what's the probability of a sentence, for example, given sentences:

- The apple and pear salad
- The apple and pear salad

] pear and peer sound similar but a good language model would say "pear" is the correct word.

P(The apple and pear salad) = 3.2×10^{-13}

P(The apple and peer salad) = 5.7×10^{-10}

→ P(sentence) ? - Fundamental for speech recognition and machine translation

$P(y^{<1>}, y^{<2>}, \dots, y^{<T_y>})$

- Training set : Large corpus of x language text. corpus = NLP terminology for a very large body of set of english text sentences.

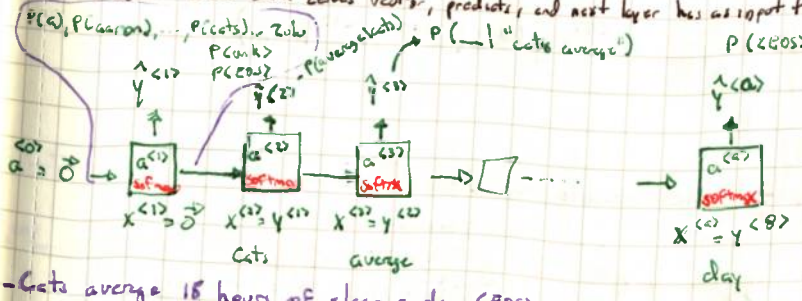
- Tokenize : Cats Average 15 hours of sleep a day <EOS> - end of sentence Token marker

$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ $y^{<4>}$ $y^{<5>}$ $y^{<6>}$ $y^{<7>}$ $y^{<8>}$

- Flag unknown words on the dictionary : The Egyptian Mau → <UNK> is a breed of cats <EOS>

RNN Model:

Start with zeroes vector, predicts, and next layer has as input the first token.



- Cats average 15 hours of sleep a day. <EOS>

- each word is fed after the first layer, then probabilities are calculated in sequence, softmax of the first word, then softmax of the second word including the previous one, and so on, concluding each word on each step; all together to output the probability of a sentence.

Cost function: $\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$ ← Softmax loss function

Loss : $L = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$

Sample for simple sentence y^1, y^2, y^3 :

$P(y^{<1>}, y^{<2>}, y^{<3>}) =$

$P(y^{<1>}) ; P(y^{<2>}, y^{<1>}) ; P(y^{<3>} | y^{<1>}, y^{<2>}) ; \dots$

[Sample Novel Sequences]

After we train a model, one of the ways we can informally get a sense of what's learned is to have sample novel sequences.

To sample, we just want the NN to generate sentences based on the training, to do so, we do a normal sampling scan on the softmax the first word we will use. Meaning, to take the first word that comes out (based on probabilities), this use for example. np.random.choice, on the vector and sample according to the distribution defined by the vector of probabilities. On the next time step, pass the first prediction, and continue in the same fashion until the end.

- A character level language model can also be used, however is computationally more expensive vs the word model, and the vocabulary would be the alphabet [0-9a-zA-Z]. One benefit is that it eliminates the need of using <UNK> tokens, as all letters would be on the dict.

[Vanishing Gradients with RNNs]

Recurrent Neural Networks as we've seen so far, are not very good at capturing very long-term dependencies. For example with the sentences, where there should be coherent, i.e. Plurals/singular nouns

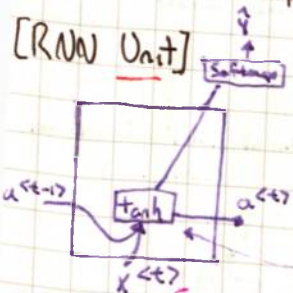
"The cat, which already ate ....., was full" "The cats, which ....., were full"

To address it, we'll address some options, while as for exploding gradients, the procedure is much simpler/easier, by using "Gradient Clipping".

Gated Recurrent Unit (GRU)

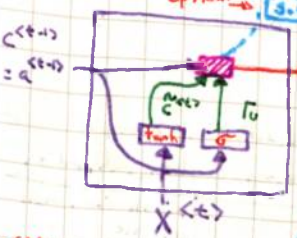
Is a modification to the RNN hidden layer that makes it much better capturing long range connections and helps a lot with the vanishing gradient problems.

[RNN Unit]



$$a^{\langle t\rangle} = g(W_a[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_a)$$

GRU (simplified) - new variable c = cell [memory cell]



c = memory cell

$c^{\langle t\rangle} = a^{\langle t\rangle}$

$\tilde{c}^{\langle t\rangle} = \tanh(W_c[c^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_c)$ — candidate to replace c

$\Gamma_u = \sigma(W_u[c^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_u)$ $0 < \Gamma < 1$

"update"

$$c^{\langle t\rangle} = \Gamma_u * \tilde{c}^{\langle t\rangle} + (1-\Gamma_u) * c^{\langle t-1\rangle}$$

$c^{\langle t\rangle}$ can be a vector, then also $\tilde{c}^{\langle t\rangle}$, and Γ_u as well. In that case the multiplications are element wise!

The cat, which already ate ..., was Full

$\Gamma_u = 1$ $\Gamma_u = 0$ $\Gamma_u = 0$ ... $\Gamma_u = 1$

$c^{\langle t\rangle} = 1$

[Full GRU]:

Modifying a bit the previous equations to account for relevance Γ_r :

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$
$$\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\Gamma_r = \sigma(W_r[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_r)$$
$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle}$$

Γ_r = relevance of word

In academic literature: $\tilde{h}$, u, r, h

[LSTM: Long Short Term Memory]

This is even more powerful than the GRU; This was a really seminal paper that had a huge impact on sequence modeling. This paper is one of the most difficult to read.

- GRU and LSTM

GRU

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$
$$\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\Gamma_r = \sigma(W_r[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_r)$$
$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle}$$
$$a^{\langle t \rangle} = c^{\langle t \rangle}$$

LSTM

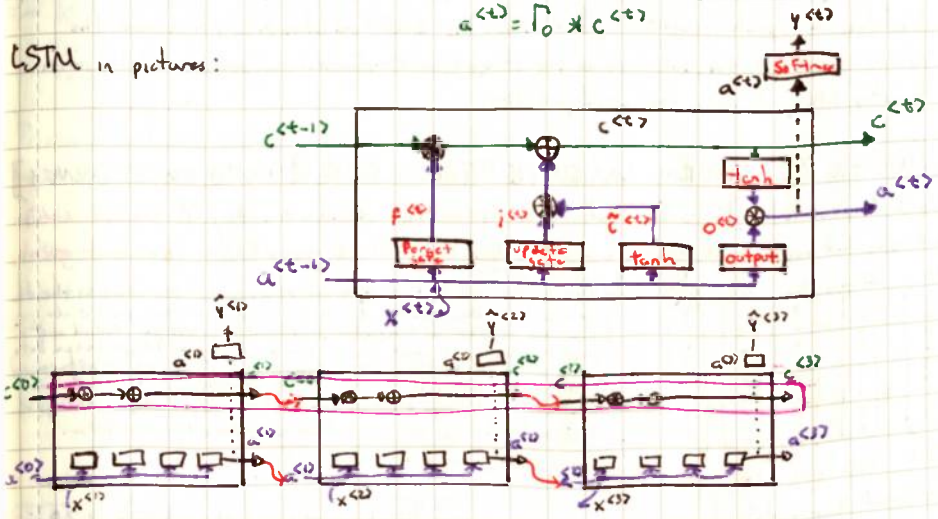
$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$

3 Actors:

$$\Gamma_u = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$ } update
$$\Gamma_f = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$ } forget
$$\Gamma_o = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$ } output

$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + \Gamma_f * c^{\langle t-1 \rangle}$$
$$a^{\langle t \rangle} = \Gamma_o * c^{\langle t \rangle}$$

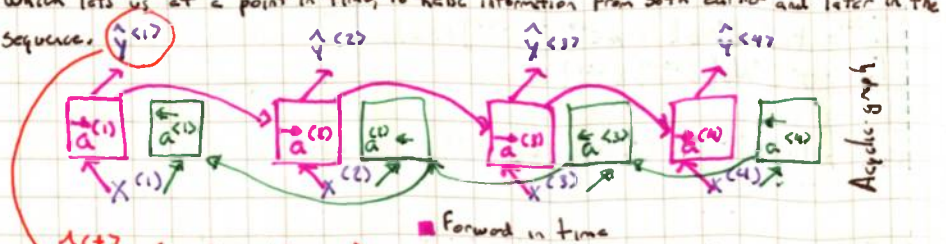
LSTM in pictures:



A common variation is the "peephole connection", meaning that the gate values may depend not just on $a^{\langle t-1 \rangle}$ and $x^{\langle t \rangle}$, but also on the previous memory cell value $c^{\langle t-1 \rangle}$

Bidirectional RNN (BRNN)

By now we've seen most of the key building blocks of RNNs, one is bidirectional RNNs which lets us at a point in time, to take information from both earlier and later in the sequence.



$\hat{y}^{<t>} = g(W_y[\overrightarrow{a}, \overleftarrow{a}] + b_y)$

■ Forward in time
■ backward in time - is forward prop! not back prop
= computations are done with network activation =

For example, at time 3:

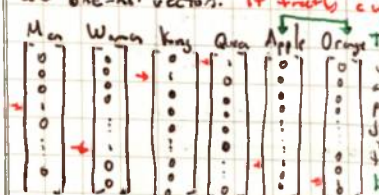
- Predictions there's from $x^{<1>} \rightarrow a^{<1>} \rightarrow a^{<2>} \rightarrow a^{<3>} \rightarrow \hat{y}^{<3>}$ so information from $x^{<1>}, x^{<2>}$ and $x^{<3>}$ counts.
- While info. from $x^{<4>} \rightarrow a^{<4>} \rightarrow a^{<3>} \rightarrow \hat{y}^{<3>}$
- So, this allows at T3 to take as input to take info from the past, present and future

- The blocks can be, not only the standard RNN blocks, but also GRU or LSTM blocks, for NLP or text problems, a BRNN with LSTM is commonly used.

The disadvantages of the Bidirectional Recurrent Neural Networks is that we do need the entire sequence of data before we can make predictions anywhere, for example if we are building a speech recognition system, then the BRNN will let us take into account the entire speech utterance, but if we use this straightforward implementation then we need the person to stop talking to make the actual prediction.

[WEEK 2: NATURAL LANGUAGE PROCESSING & WORD EMBEDDINGS]

[Word Representation]: So far we've been representing words in a vocabulary $|V| = [a, aaron, \ldots, zulu]$ and one-hot vectors: It strictly categorizing into itself and doesn't allow to generalize across words



The relationship between apple and orange is not any closer than the other words like man, woman, king, queen, so even if "orange juice" is also a common sentence as "apple juice", and the NN has learned it as a popular phrase, it won't realize the similarity. Cannot generalize both juices are a similar thing, that's because the inner product between any two one hot vectors is zero, and there is no distance, so it doesn't know that apple and orange are more similar than apple and queen or orange and king, so further we can use rather a Featurized representation, with features on the one-hot vectors:

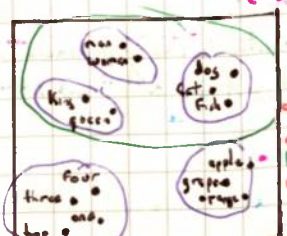
[Featurized Representation]

Gender	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Royal	-1	1	-0.95	0.97	0.00	0.01
age	0.01	0.02	0.93	0.95	-0.01	0.00
Food	0.03	0.02	0.7	0.69	0.03	-0.02
Size	0.09	0.01	0.02	0.01	0.95	0.97
cost						
alive						

- Similarity

i.e. 300 dim vector

Visualizing word Embeddings (300D → 2D): t-SNE

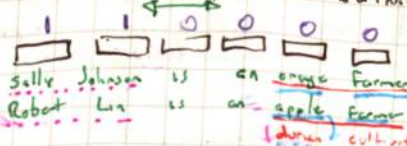


Groups like clustering by similarity and among Features and more Features close in common

t-SNE: ☠ visualization of embeddings

[Word Embeddings]

Once we have featural representations of words, the embedding vectors, we can 'easily' learn an algorithm that can generalize for example, similarities like "Farmer" and "Cultivator" or names, persons. But for that, we should have seen these words on the training set, so it can infer that "cultivator" is also a farmer or "dunam" is a unit, but also even that a "dunam cultivator" is a person.



For this, the algorithm should examine 1 Billion words → 100 Billion → huge but to learn embeddings that group them together.

Take the word embedding and apply it to our named-entity recognition task: Transfer learning to train a smaller "100K" labeled training set.

[Transfer Learning and Word Embeddings]

1. Learn Word Embedding from large text corpus (1–100B words) (or download pre-trained embedding online)
2. Transfer embeddings to new task with smaller training set (say 100K words)
3. Optional: Continue fine-tune the word embeddings with new data.

Man : Woman :: Boy : Girl
Father : Grandfather :: Mother : Grandmother
Tokyo : Japan :: Rabat : Morocco
Big : Bigger :: Tall : Taller

[Properties of Word Embeddings]

Help with analogy reasoning and convey a sense of what these words embeddings are doing. Using the t-SNE algorithm, and assuming 4D vector only:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

e_{5391} = e_{man}, e_{woman}, e_{king}, e_{queen}, e_{apple}, e_{orange}

$e_{man} - e_{woman} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $e_{king} - e_{queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ diff = Gender!

Man → Woman / King → ?

$e_{man} - e_{woman} \approx e_{king} - e_{?}$

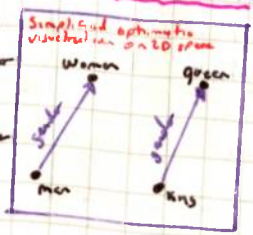
$e_{?} \approx e_{king} - e_{man} + e_{woman}$

So, to find the word ? represented as $e_?$:

$\arg\max_w \text{Similarity}(e_w, e_{king} - e_{man} + e_{woman})$

Previously we talked about using t-SNE to visualize words, what t-SNE does, is that takes 300D data and maps it to a very non linear way to 2D space and so the mapping with t-SNE learns crazy, we can't expect these types of relationships to hold true, but in higher dimensional space, analogies hold.

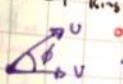
Simplified optimistic visualization in 2D space



The most common similarity function is the cosine similarity:

[Cosine Function]

$\text{Sim}(e_w, e_{king} - e_{man} + e_{woman}) = \text{Sim}(U,V) = \frac{U^T V}{\|U\|_2 \|V\|_2}$ ← euclidian lengths } if similar the product will be large



Another option, but for "dissimilarity" is to use square distance or euclidian distance: $\|U - V\|^2$

How different these two words are from each other.

[Embedding Matrix]

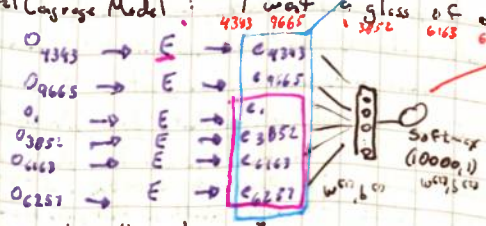
$E \cdot O_j = e_j$ = embedding for word j in the vocabulary

Note: on the relatively high dimensional matrix drug the matrix multiplication is not so expensive because the count of zeros, so we will better use a specialized and more effective way like with Keras's function "Embedding"

Embedding matrix E, initialized randomly and use gradient descent to learn all the parameters of this 300 × 10000 parameter matrix.

[LEARNING WORD EMBEDDINGS]

Neural Language Model: I want a glass of orange ____ ?



- Embedding vectors
- Subscript is the result of the embedding
- It will be stacked and passed into a NN, so then for example, having 6 words (300 D) means we'll pass 1800 parameters.
- Also we can choose to pass for example the last 4 words, so it will be 1200 parameters

[Other context/target pairs]

x = I want a glass of orange juice to go along with my cereal.

target

CONTEXT

- Context: last 4 words
  - 4 words on left and right: a glass of orange ? to go along with
  - Last 1 word: orange ?
  - Nearby 1 word: glass ? } Skip Gram Model

[Word2vec] model: Simpler and computationally efficient to learn this embeddings (Skip-grams) Mikolov et al., 2013. Efficient estimation of word representations in vector space.

In the skip-gram model, what we going to do is to come up with a few context to target pairs to create our supervised learning problem. So rather than having the context be always the last 4 words or last end words immediately after or before the target word, randomly pick a word to be the context word, let's say, the word orange and randomly select another word within some window. Say ±5 or ±10 words of the context word. So it can happen that we choose the immediate one or the one that is one before that.

Obviously this is not an easy learning problem, because between ±n words, there could be way too many words in between.

x = I want a glass of orange juice to go along with my cereal

Context	Target
Orange	juice
Orange	glass
Orange	my

Model: Vocabulary 10000 words

Context c ("orange") (6257) → Target t ("juice") (4834)

$O_c \rightarrow E \rightarrow e_c \rightarrow$ softmax $\rightarrow \hat{y}$

Softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$

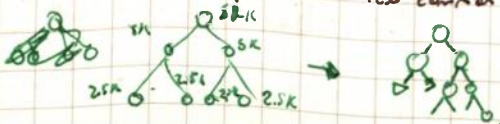
θ_t = parameter associated with output t

$y = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$ ← 4834

$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10000} y_i \log \hat{y}_i$

Paper: Mikolov et al., 2013: Efficient estimation of word representations in vector space.

This 10K is already too slow, even worst with vectors of 1M, or more... ! an alternative is the hierarchical softmax, which is a balanced tree with binary classification. In theory the trees are balanced, however in the application, the trees are sorted by the most common at the top, to the less common at the bottom.

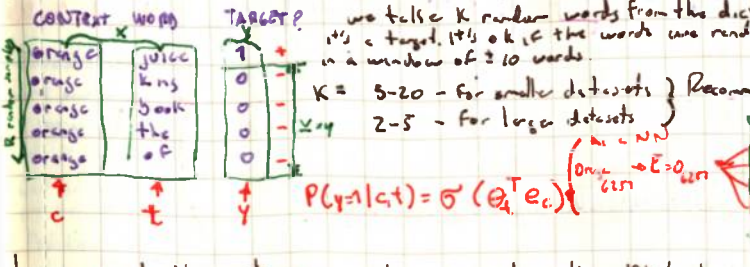


To sample "c": If uniformly at random, it will sample very common words like "the, of, and", so in practice the distribution of words $P(c)$ isn't uniform, there are different heuristics to balance it with less

[Negative Sampling]: Mikolov et. al, 2013: Distributed representation of words and phrases and their compositionality

We already saw how the skip-gram model allows us to construct a supervised learning task. So we map from context to target and how that allows us to learn useful word embedding. But the downside of that, was that the softmax objective was too slow to compute. Now we will address Negative sampling that will allow us to do something similar to the skip-gram model, but with a much more efficient algorithm.

What are we going to do is to create a new supervised learning problem: given a pair of words, we are going to predict "is this a context-target pair?". x = I want a glass of orange juice to go along with my cereal



we take K random words from the dictionary and compare if it's a target. It's ok if the words are randomly pick we even in a window of ± 10 words.

K = 5-20 - for smaller datasets } Recommendations from author
K = 2-5 - for larger datasets

Binary classification (binary logistic)

In comparison to the great expensive softmax, we endup with a 10K logistic regression classification problem, which is going to classify if the word is juice, book, etc. It's heavy 10K but on each iteration is going to train only K+1 logistic units.

[Selecting Negative Samples]

Sample proportionally by the frequency of the word: $P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10000} f(w_j)^{3/4}}$ } Author recommended and validated

[GloVe word vectors]: Pennington et. al, 2014. GloVe: Global Vectors for word representations

-I want a glass of orange juice to go along with my cereal-

context to target

X_{ij} = # times i appear in the context of j

c = context word, t = target word

Might happen that depending on the choice of symmetry, X_{ij} should be before and after over a window of $\pm X$ so $X_{ij} = X_{ji}$; if restricted to be before only then it won't happen.

Model

$$\text{minimize} \sum_{i=1}^{10000} \sum_{j=1}^{10000} f(X_{ij}) \cdot \left(\theta_i^T e_j + b_i + b_j - \log X_{ij}\right)^2$$

Weighting term

θ_i^t, e_j^c

Initialize random run GD Gradient then take the AVG

θ and e are symmetric

$e_w^{(final)} = \frac{e_w + \theta_w}{2}$

$0 \log 0 = 0$ $f(X_{ij}) = 0$ if $X_{ij} = 0$: if X_{ij} is 0 then don't bother to sum over the X_{ij} pair

$f(X_{ij})$ also helps to give proper weights to words that appear less frequently, but also more (but not unduly) weights to more common words. (weighting factor)

- frequent words: this, is, of, a
- less frequent: durian ...

check the paper for implementation of $f(X_{ij})$

Disconnect

[Sentiment Classification]

The idea is to identify the meaning wether positive or negative, for example with rating where we have examples like "good and tasty" → ★★★★☆. "The dessert is excellent" → 5 stars

8928 2468 4694 3180

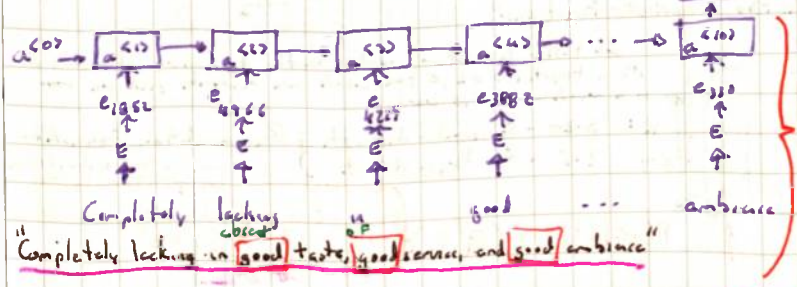
the → o_{8928} → E → e_{8928}
dessert → o_{2468} → E → e_{2468}
is → o_{4694} → E → e_{4694}
excellent → o_{3180} → E → e_{3180}

~10000 ~1800 2000 ~7

→ Avg → softmax 1–5 stars → $\hat{y}$

Simple model, works reasonable well it averages the meanings of words, it ignores word order, so might give false positives

[RNN For Sentiment Classification]



Many to One RNN

generalizes pretty well if we have trained on a big corpus

"Completely lacking in good taste, good service, and good ambiance"

[Debiasing Word Embeddings]: Bolukbasi et al, 2016. Man is to computer programmer as woman is to homemaker

The algorithms interpret the data as it is, so they could make stereotyped or biased assumptions based on the word embeddings; Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model:

- Man : Woman, King : Queen
- Man : Computer programmer, Woman : Homemaker
- Father : Doctor, Woman : Nurse

[Addressing bias in word Embeddings]

1 = Identify bias direction

$e_{he} - e_{she}$

$e_{male} - e_{female}$

$e_{\dots}$:

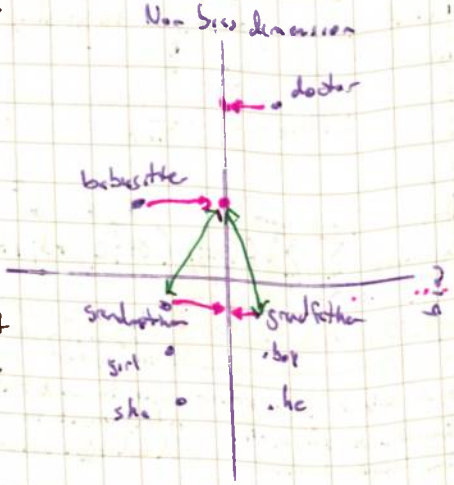
Average

2 = Neutralize for every word that is not definitional, project to get rid of bias.

3 = Equalize pairs

grandmother - grandfather

girl - boy



- - WEEK 3 : Sequence Models and Attention Mechanism]]

for Machine translation to speech recognition.

Sutskever et.al, 2014. Seq 2 Seq learning with NN

Cho et.al. 2014: Learning phrase representation using RNN encoder-decoder for statistical machine translation.

Model]

vence to sequence model

Jane visite l'Afrique en Septembre

Jane is visiting Africa in September

the sentence in French

t thru a GRU/RNN/LSTM

t a vector that ents the input nce

Encoding Network

Decoding Network

se Captioning]

"a cat and a window"

her a Conv Net, here it learn and encode a t of features of the picture

et rid of the softmax layer and use the rmation vector that represents the image

Feed this vector 4096 to a RNN

Full AlexNet Network

4096

Softmax 1000

ao et.al., 2014. Deep Captioning with multimodal RNN

Vinyals et. al., 2014. Show and tell: Neural image caption generator

Karpathy and Li, 2015. Deep visual-semantic alignments for generating image descriptions

Picking the Most likely Sentence

There is some similarity with the sequence to sequence machine translation model and the language models that we have worked on the last few course, but there are some significant diffs. as well.

Language Model:

Allows us to estimate the probability of a sentence or generate novel sentences.

$P(y^{<1>}, \ldots, y^{<T_y>})$

Pretty similar

Machine Translation:

- Encoding Network
- Decoding Network

Instead of passing in array of zeroes, it passes already with encoded trained parameters or representation of the input sentence

This becomes into what Andrew calls "Conditional language model", as the parameters already encoded and representing the input sequence, are passed to the decoding network. This then becomes a calculation between the probability of an english translation, conditioned on a French sentence

$P(y^{<1>}, \ldots, y^{<T_y>} \mid x^{<1>}, \ldots, x^{<T_x>})$

[Finding the most likely translation]

Given the following sentence, the model will give us the probability of different corresponding english translations.

Jane visite l'Afrique en Septembre

$P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$ — ENGLISH | ~French

What we don't want is to sample translations at random!. For example:

1: Jane is visiting Africa in September ✓
2: Jane is going to be visiting Africa in September ~✓
3: In September, Jane will visit Africa ~✓
4: Her African friend welcomed Jane in September ✗!

} Because we might sample a bad translation, so while using the model we want to find the sentence that maximizes the conditional probability.

$$\arg\max_{y^{<1>}\ldots y^{<T_y>}} P(y^{<1>},\ldots,y^{<T_y>}|x)$$

An algorithm to do find the best sentence that maximizes the conditional probability, is called "Beam Search". But still important to mention other approaches that

- Greedy search: Search the best possible word on our language model, and iteratively, do the same over and over. This picking only the most likely words. It will end up choosing by "commonality" rather than doing it by semantic way:
  - Jane is visiting Africa in September
  - Jane is going to be visiting Africa in September

  } "going" might be more common than "visiting, thus".

$$P(\text{Jane is going}|x) > P(\text{Jane is visiting})$$

- NO! -

[BEAM SEARCH]

(For drawing, check the stepped process)

Beam search differentiates itself from greedy search because this algorithm takes "B" samples at a time. If we set B=1, then it will become essentially a greedy algorithm.

1 The first thing it does, is to try to pick the first words of the translation; with B=3, then it will pick the "B" or 3 maximum possibilities (and most likely) at a time (beam width).

So from the vocabulary, of let's say (on the example) of 10000 words, the 3 (B) most likely words will be chosen.

2 Pass each of the first $y^{<1>}$ words into an encoder-decoder network to get the first "B" possible words, (from the vocabulary) $P(y^{<1>}|x)$

3 Having picked the 3 top words, is for each of these choices, which should be the next word, for that, also using the same vocabulary, so for each of them, they will be passed to the network to predict the probability of the first word with a new "top word" from the dictionary, this, by passing each combination to 3 different instances of the network (3 on this case):

$$P(y^{<2>}|x, \text{"in"}),\ P(y^{<2>}|x, y^{<1>}) \text{ or } P(y^{<2>}|x, \text{"Jane"})\ /\ P(y^{<2>}|x, \text{"septem"})$$

On this example, that would mean that there will be 3 x 10000 components, and the search would need to pick the top "B" again

4 Repeat again the process, now pass the top "3" two word sentences to the 3 different instances of the model, and on each pass them "hardwired" and then the probability (conditional) should be now: $P(y^{<3>}|x, \text{"in september"})$ and so on.

5 Repeat again with the next word, until <EOS> -end of sentence-

[Refinements on Beam Search]

[Length normalization]: is a small change to the basic beam search algorithm that can help us to get much better results. Beam search is about maximizing the probability, and with the current algorithm, the probabilities multiplied on each step, tends to reduce the value to a numerical underflow.

$$\arg\max_{y} \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$$

$P(y^{<1>} \ldots y^{<T_y>} | x) = P(y^{<1>}|x) \cdot P(y^{<2>}|x, y^{<1>}) \ldots P(y^{<T_y>}|x, y^{<1>} \ldots y^{<T_y-1>})$

Probabilities are from 0 to 1, thus multiplying several values <1 will gradually reduce the number leading to underflow.

In practice, instead of maximizing this product, we will take logs, as the log of a product becomes the sum of a log, and maximizing this sum of log probabilities, should give the same result in terms of selecting the most likely sentence y. We end with this, with a more numerical stable algorithm, that is less prone to numerical rounding errors or underflow.

$$\arg\max_{y} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$$

However this still has a problem, and it's that it will prefer or "reward" shorter sentences as the value will be higher with less sentences, than with more, because of the nature of the multiplication or the sum of logs, for that it's needed to normalize it, by dividing it will reduce the penalty:

$$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$$

α = heuristic, hyperparameter

$\alpha = 0.7$

$\alpha = 1$ - completely normalized

$\alpha = 0$ - no normalization

To wrap up: As we run beam search, we will see a lot of sentences with length =1, a lot with length =2, a lot with length =3, and so on. Maybe we run the search for 30 steps and we consider output sentences up to 30 (let's call A-1 so that $B = 3$, we will be keeping track of the top 3 possibilities for each of these possible sentence lengths, 1, 2, 3, 4 and so on up to 30. Then we will look at all the output sentences and score them against this score. And so we can take our top sentences and just compute the objective function onto sentences that we have seen through the beam search process. And finally, of all of these sentences that we validate this way, we pick the one that achieves the highest value on the normalized log probability objective. Sometimes it's called a Normalized log likelihood objective. And then that would be the final translation your outputs.

[Beam Search discussion]

- How Beam width B? - The larger it is, the more possibilities we consider, but also computer requirements

Large B: better results, slower

Small B: worse results, faster

Greedy → 1 → Google → 3 → Facebook → 10 → Baidu → 100 → Research Papers → 1000 → 3000

[Error analysis on Beam Search]

It's given for example a perfect translation by a human, but the model outputs a wrong one. It's important to determine if the problem is on the RNN or the BeamSearch algorithm. We can compare $P(\hat{y}|x)$ and $P(\hat{y}|x)$ and determine who's wrong.

Go to the training set and look at the training example and make the error analysis process

Human	Algorithm	$P(\hat{y}\|x)$	$P(\hat{y}\|x)$	At fault
—	—	—	—	—
1	⋮	⋮	⋮	B
⋮	⋮	⋮	⋮	R
⋮	⋮	⋮	⋮	R
⋮	⋮	⋮	⋮	B

Case 1: $P(\hat{y}|x) > P(\hat{y}|x)$ ← but Beam Search

Beam chooses $\hat{y}$, but $\hat{y}$ attains higher $P(y|x)$

Case 2: $P(\hat{y}|x) \leq P(\hat{y}|x)$

$\hat{y}$ is better translation than $\hat{y}$. But RNN predicts $P(\hat{y}|x) < P(\hat{y}|x)$: RNN to at fault

[Bleu Score]: Bilingual Evaluation Understudy

[Papineni et. al., 2012. Bleu: A method for automatic evaluation of machine translation]

- Evaluating Machine translation

To match equally good translations (human), that might be written similarly but mean or express exactly the same. For example.

French: Le chat est sur le tapis

Reference 1: The cat is on the mat (2x)

Reference 2: There is a cat on the mat

MT: the the the the the the the } 7 outputs

We will give a word credit up to the maximum times it appears in the reference sentences

Precision: 7/7 Modified Precision: 2/7

Bleu Score on bigrams

	Count	Count clip
1 the cat	2x	1
2 cat the	1x	0
3 cat on	1x	1 → $\frac{4}{6} = \frac{2}{3}$
4 on the	1x	1
5 the mat	1x	1

Ref 1: the cat is on the mat

Ref 2: There is a cat on the mat

MT: The cat the cat on the mat

- Bleu Score on Unigrams

$$P_1 = \frac{\sum_{\text{unigram} \in \hat{y}} \text{Count}_{\text{clip}}(\text{unigram})}{\sum_{\text{Unigram} \in \hat{y}} \text{Count}(\text{unigram})}$$

(1 unigram: single words in isolation)

$$P_n = \frac{\sum_{n\text{-gram} \in \hat{y}} \text{Count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram} \in \hat{y}} \text{Count}(n\text{-gram})}$$

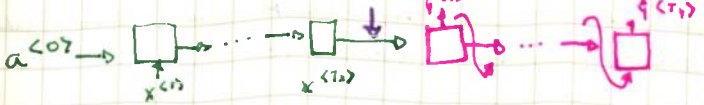
Details: Short translations give high scores, so we add BP (brevity Penalty), which is a factor for adjustment, that penalizes translation systems that output translations that are too short.

$$BP = \begin{cases} 1 & \text{if MT output length} > \text{reference output length} \\ \exp(1 - \text{MT output length} / \text{reference output length}) & \text{else} \end{cases}$$

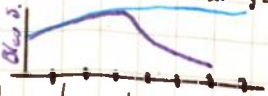
[Attention Model Intuition]

-The problem of long sentences:

(When we input a very large sentence, what we are asking to the green encoder to read in the whole sentence and then memorize it, the whole is stored in the activation, then for the purple decoding network to then generate the translation



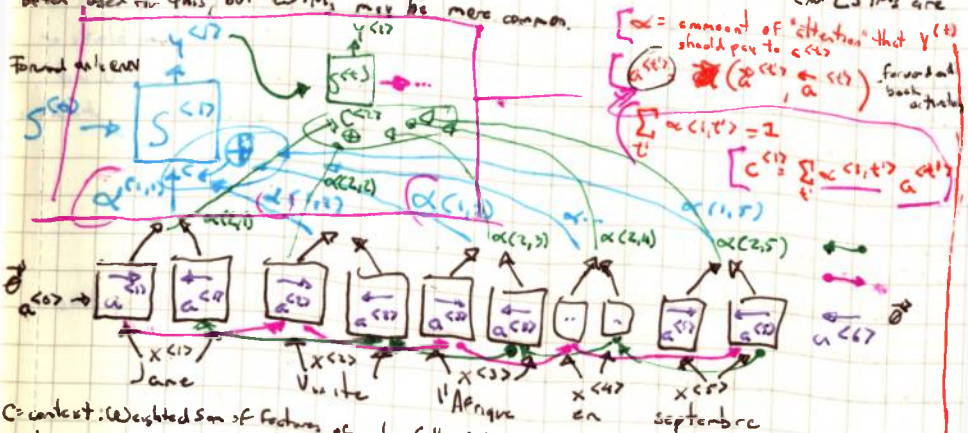
Now, the way a human translator would translate a huge sentence, is not by reading the whole thing, memorize it, and then regurgitate the whole translated version from scratch. Instead the human translator would read the first part of the sentence and generate some words, read more and generate more words, and so on.



(What we see for the encoder – decoder architecture shown above, is that it works quite well for short sentences, so we might achieve relatively high Blue score, but for very long sentences, maybe longer than 30 -40 words, the performance comes down. Later on we will see the the inner workings, but the Attention model shows this, working as humans.

[Attention Model]: Pay attention to apart of the sentence.

Let's assume we have an input sentence, we use a RNN, or Bidirectional GRU or bidirectional LSTM to compute features of every word. In practice, GRUs and LSTMs are often used for this, but LSTMs may be more common.



C= context: Weighted Sum of features of each of the features from the diff timesteps multiplied by the attention α's

α= attention parameter (how much attention), how much the context will depend on each activation from time steps

all weights are summed to (1)

[Bahdanau et. al., 2014. Neural Machine Translation by learning to align and translate]

[Xu et. al., Show, attend and tell: Neural image caption generation with visual attention]

[Speech Recognition - Audio Data]: Sequence to Sequence models applied to audio data

What is the speech recognition problem? Given an audio clip "x" make a text transcript.

Audio is taken on three dimensions: time, frequency and loudness or intensity, thus it requires preprocessing.

Academic has research with 300 to 3000 hours of transcripts, whereas commercial grade systems have 10K or 100K hours of audio data and its transcript.

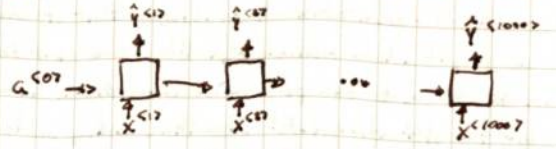
Having similar inputs as the attention model requires T_x, it can be used as well.

CTC Cost for speech recognition [Graves et.al., 2006. Connectionist Temporal Classification: Labeling unsegmented sequential data with RNNs]

(Connectionist Temporal Classification)

"The quick brown fox"

Simplified for demonstrating purposes, as in reality we would need to use: Bidirectional LSTM or Bidi-GRU, and deeper model



Basic rule: collapse repeated characters not separated by "blank"

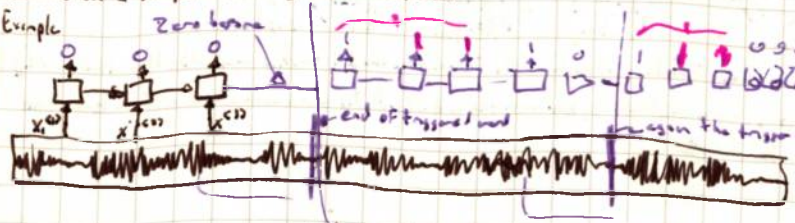
Notice the number of steps is very large (1000), it's fixed, and it's because as shown above, the model can output repeated characters, it will be filled with spaces blanks or the correct characters. So, the input sentence, even if it has 15 characters, the NN will output 1000, but once the repeated characters are collapsed and blanks removed, it will have succeeded. THIS REQUIRES LOOOTS OF DATA

[Trigger Word Detection]: Still evolving technology

The technology behind smart assistants like Alexa, Google home, Siri, etc.

It requires less data to train and it can be used to trigger specific activities, like turning on a lamp, executing a command, etc.

Example



Having to output so many zeros as often might create unbalance, so what we can do is to make the model to output a bit more "ones", so the ratio 1:0 gets more even.