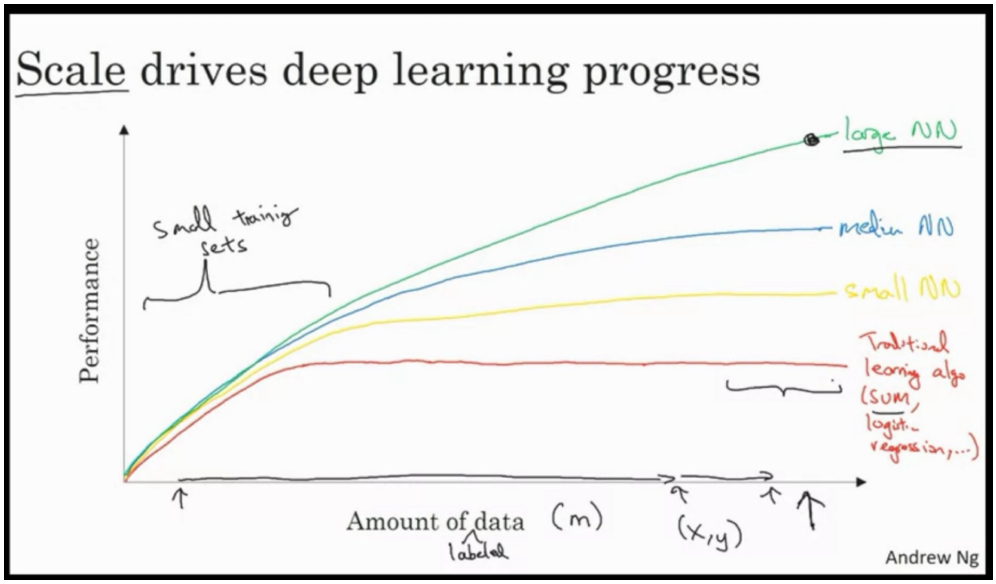
Deep Learning (Coursera)

Neural Networks and Deep Learning

Why deep learning take off?



Course Outline

1. Nerural Networks and Deep Learning
2. Improving Deep Neural networks: Hyperparameter tuning, Regularization and Optimization
3. Structuring your Machine Learning project
4. Convolutional Neural Networks
5. Natural Language Processing: Building sequence models

Logistic Regression as a Neural Network

Notation

m denote the **number of training examples**

n denote the **number of features**

A single training example: $(x, y), x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$

m training example: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

$m_{train} = \#train\ examples, m_{test} = \#test\ examples$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_x \times m}$$ (more easier to implement)

> $Y = [\, y^{(1)} \quad y^{(2)} \quad \cdots \quad y^{(m)} \,] \in \mathbb{R}^{1 \times m}$

Logistic Regression

Given x, want $\hat{y} = P(y = 1|x), x \in \mathbb{R}^{n_x}$

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

Output: $\hat{y} = \sigma(w^\top x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

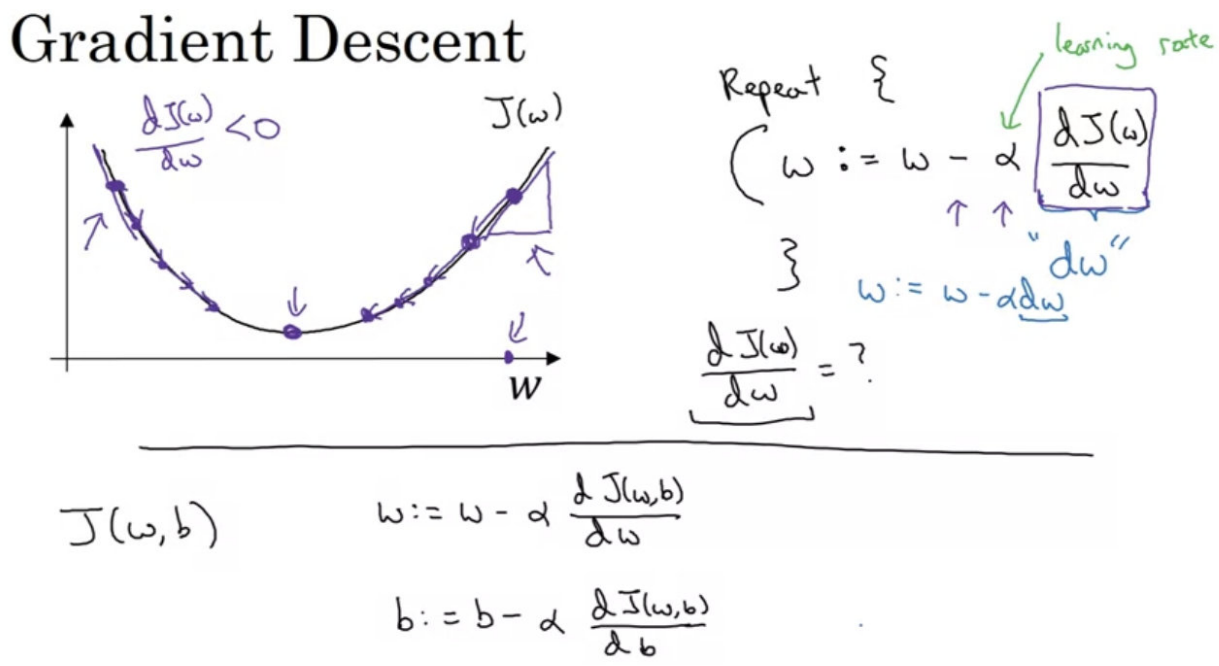
Loss function:

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

Cost function:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y) = -\frac{1}{m} \sum_{i=1}^{m} [y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

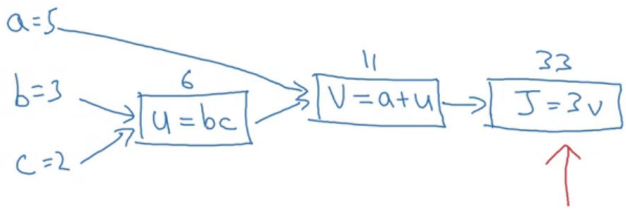
Gradient Descent



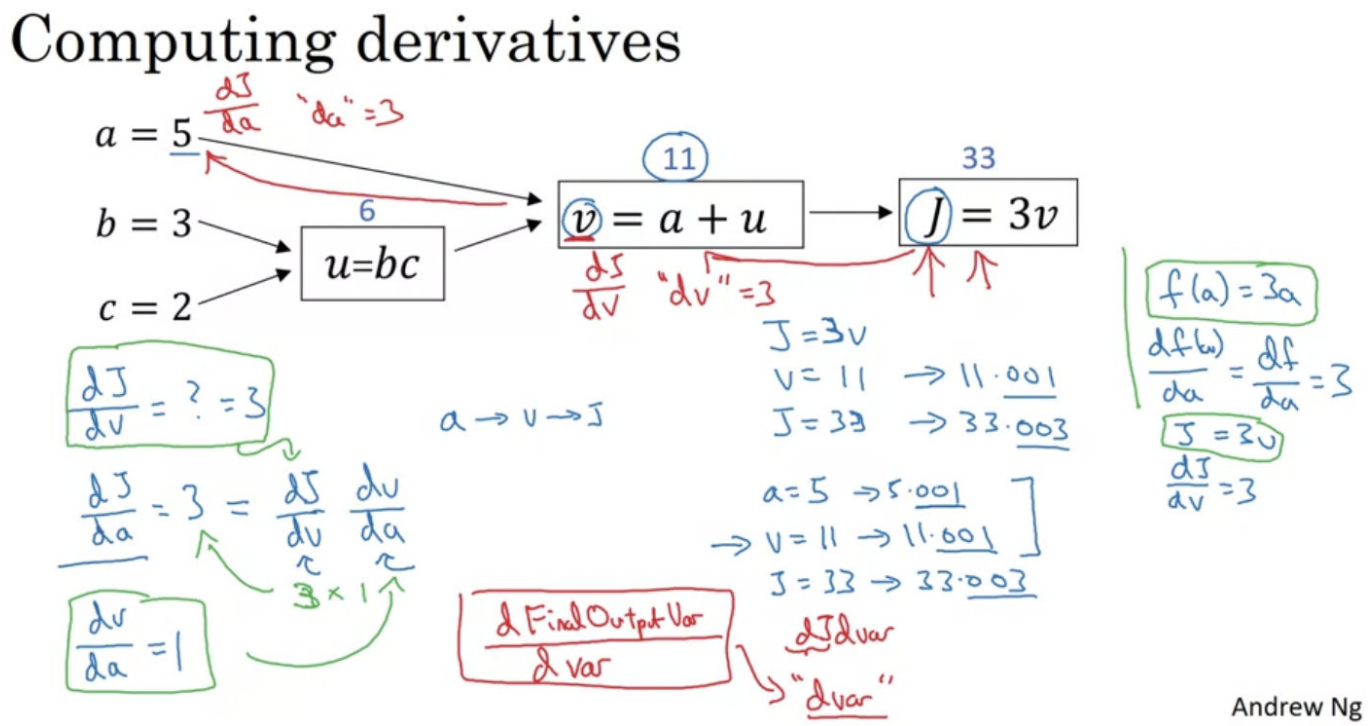
Andrew Ng

Computation Graph

Forward

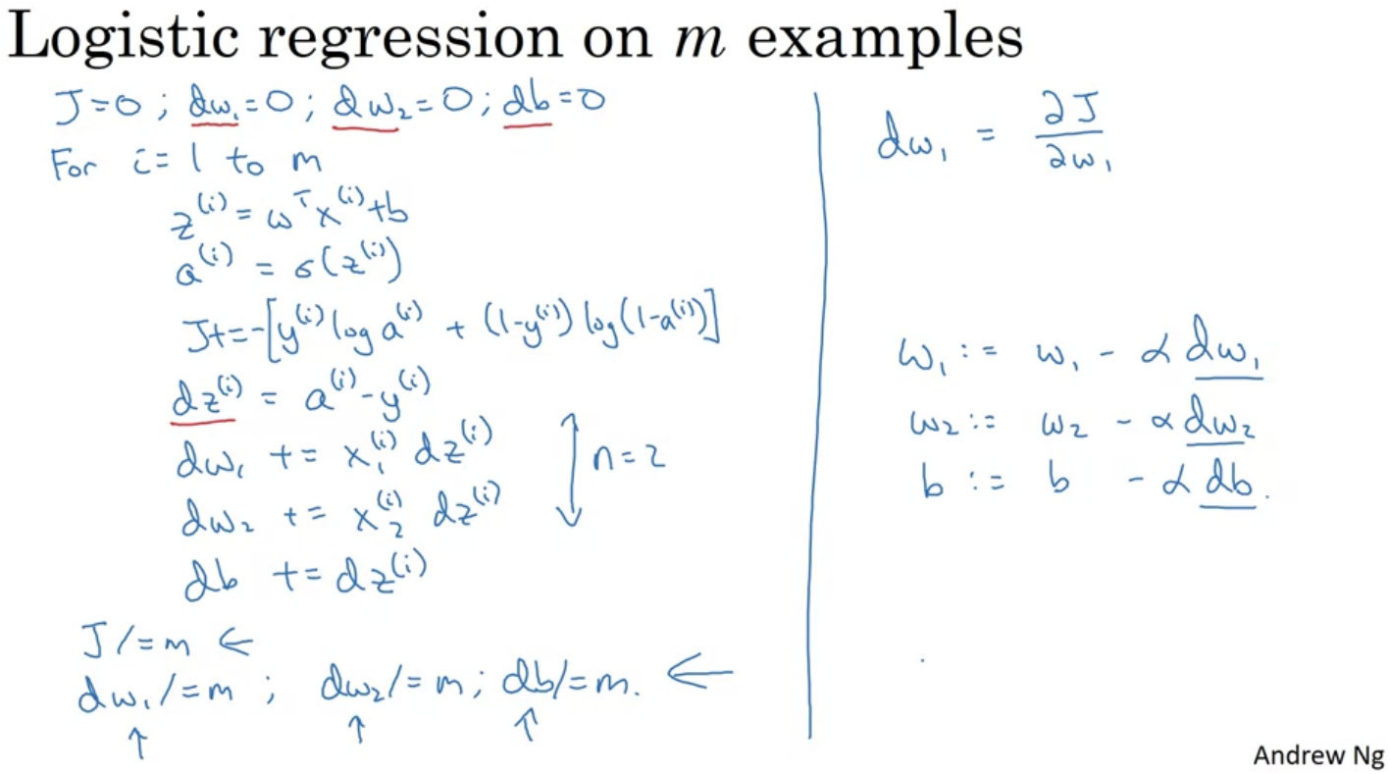


Backward

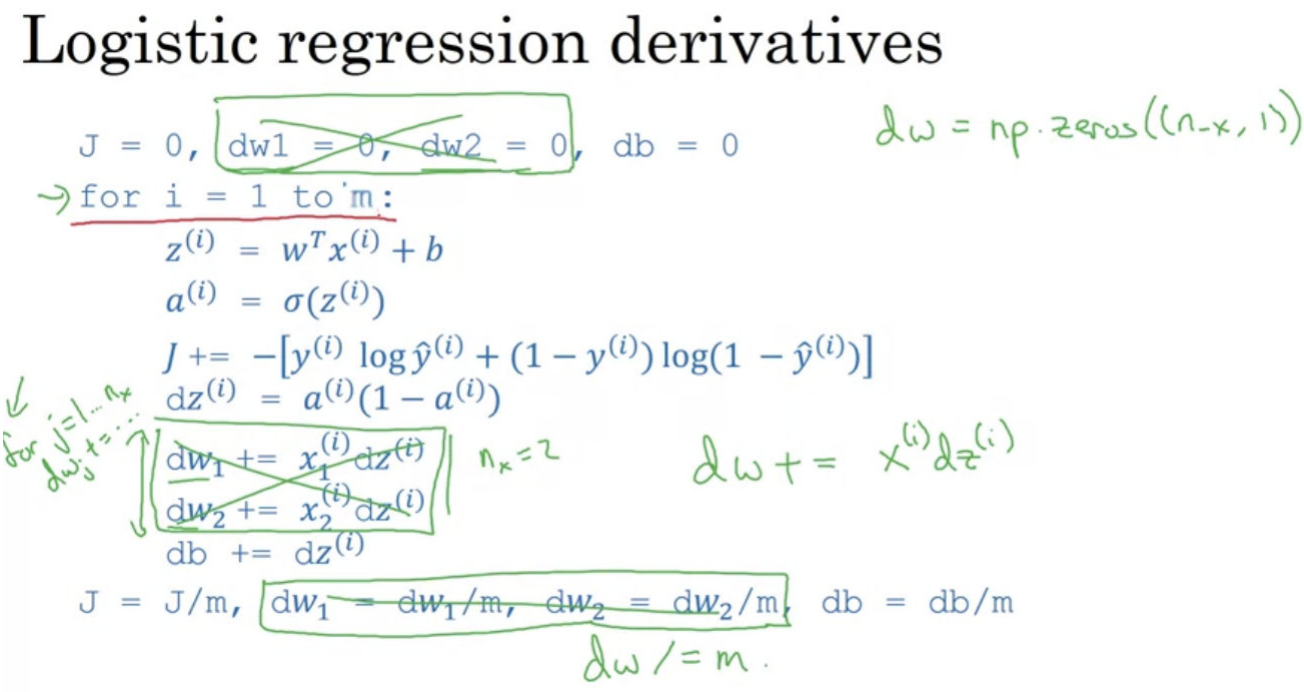


$$\frac{dL(a,y)}{dz} = \frac{dL}{da}\frac{da}{dz} = (-\frac{y}{a} + \frac{1-y}{1-a})a(1-a) = y(a-1) + a(1-y) = a - y$$

Logistic regression on m examples



Vectorization



Verctorizing Logistic Regression

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_x \times m}$$

$$\begin{aligned} [z^{(1)} \quad z^{(2)} \quad \cdots \quad z^{(m)}] &= [w^\top x^{(1)} + b \quad w^\top x^{(2)} + b \quad \cdots \quad w^\top x^{(m)} + b] \\ &= [w^\top x^{(1)} \quad w^\top x^{(2)} \quad \cdots \quad w^\top x^{(m)}] + [b \quad b \quad \cdots \quad b] \\ &= [w^\top \quad w^\top \quad \cdots \quad w^\top] \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} + \mathbf{b} \\ &= \mathbf{w}^\top \mathbf{X} + \mathbf{b} \end{aligned}$$

```
Z = np.dot(w.T, X) + b # b is real number
```

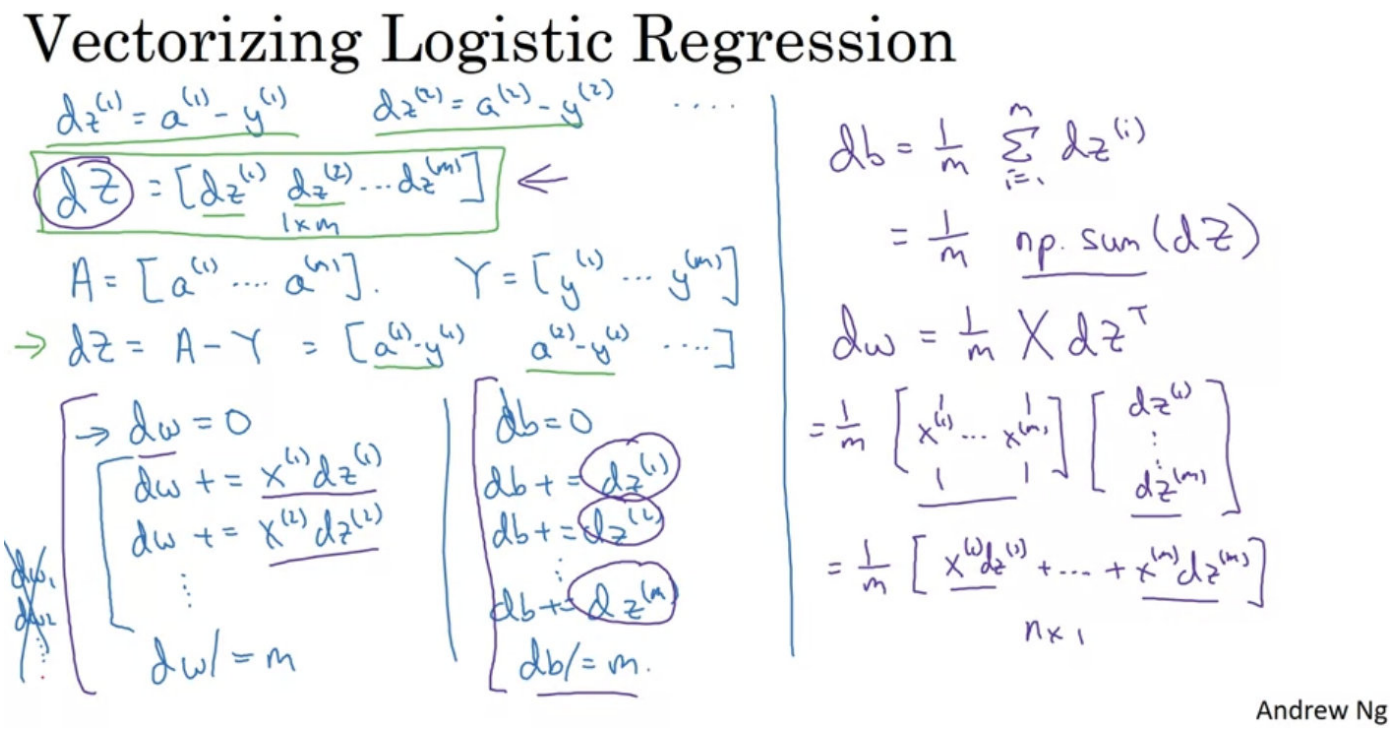
$$A = [a^{(1)} \quad a^{(2)} \quad \cdots \quad a^{(m)}] = \sigma(Z)$$

Gradient computation

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

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$dw = \frac{1}{m} X (dZ)^\top$$



Andrew Ng

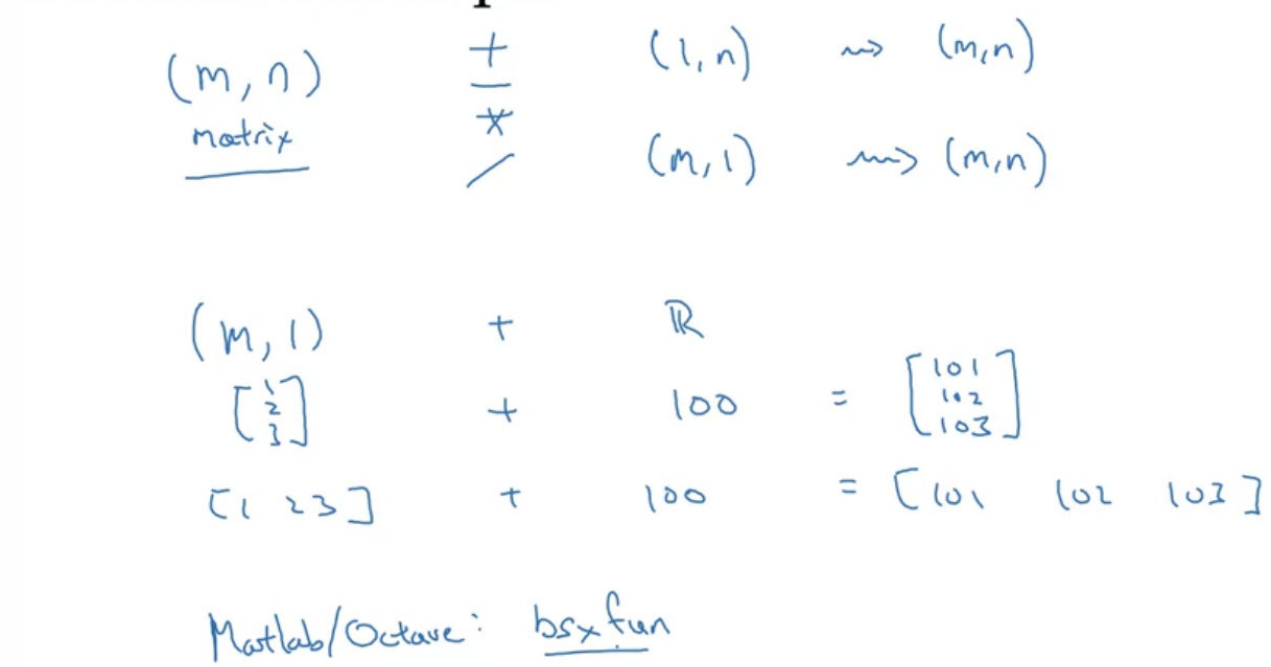
```
Z = np.dot(w.T, X) + b
A = sigmoid(Z)
dZ = A - Y
dw = 1/m * X * dZ.T
db = 1/m * np.sum(dZ)

w -= lr * dw
b -= lr * db
```

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

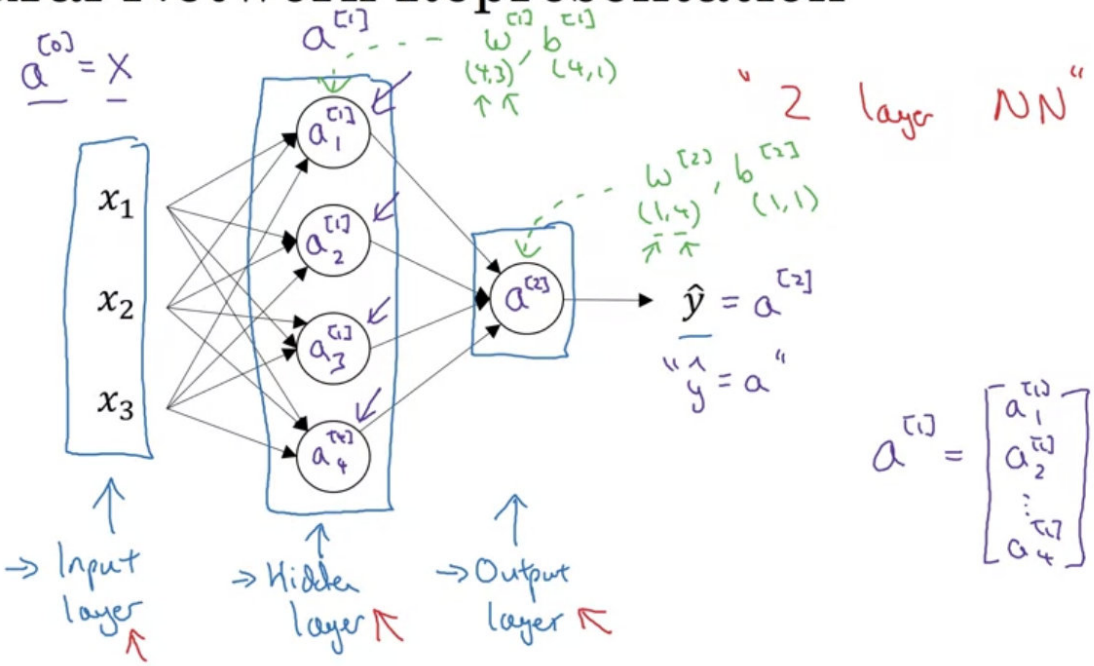
Broadcasting in Python

General Principle



Shallow Neural Network

Neural Network Representation



Andrew Ng

$a_{i \leftarrow node}^{[l] \leftarrow layter}$

Vetorization

Single example

$$W^{[l]} = \begin{bmatrix} - - (w_1^{[l]})^\top - - \\ - - (w_2^{[l]})^\top - - \\ \vdots \\ - - (w_n^{[l]})^\top - - \end{bmatrix}$$

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

Across the training set

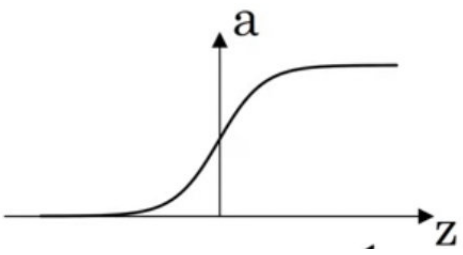
$$Z^{[l]} = \begin{bmatrix} | & | & & | \\ z^{[l](1)} & z^{[l](2)} & \cdots & z^{[l](m)} \\ | & | & & | \end{bmatrix}$$
$$A^{[l]} = \begin{bmatrix} | & | & & | \\ a^{[l](1)} & a^{[l](2)} & \cdots & a^{[l](m)} \\ | & | & & | \end{bmatrix}$$
$$A^{[l]} = \sigma(W^{[l]} A^{[l-1]} + b^{[l]}),\ A^{[0]} = X$$

Activation function

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

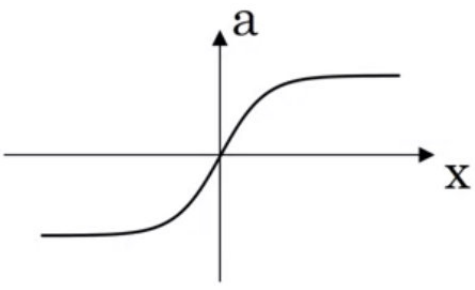
g is the activation function

Sigmoid function (Never use this except for output layer)



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

Hyperbolic tangent function (tanh)

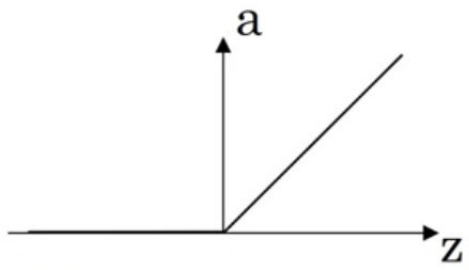


$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

A shifted version of sigmoid function, works **better** than sigmoid.

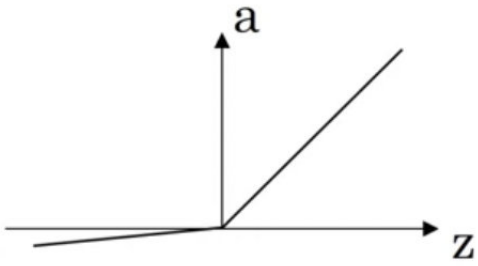
However, for **both** sigmoid and tanh fucntion, when the z is either **very large** or **very small**, the **gradient becomes very small**.

Rectified linear unit function (ReLU) -- Default choice



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

Leaky ReLU



$$a = max(0.01z, z)$$

Why do we need non-linear activation function?

If there is no non-linear activation function, there is no hidden layers, because it is just the combination of linear function.

Derivatives of activation function

Sigmoid

$$\frac{d}{dz}g(z) = g(z)(1 - g(z))$$

tanh

$$\frac{d}{dz}g(z) = 1 - (\tanh(z))^2 = 1 - (g(z))^2$$

ReLU

$$\frac{d}{dz}g(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

Leaky ReLU

$$\frac{d}{dz}g(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

The gradient at $z = 0$ for ReLU and Leaky ReLU is technically not defined. BUT we just take 1.

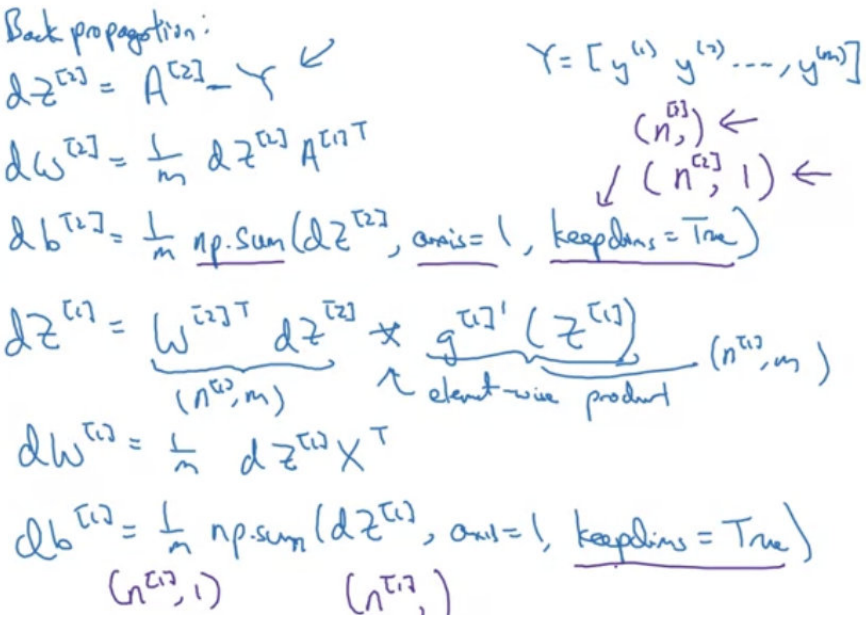
Gradient decent for neural networks

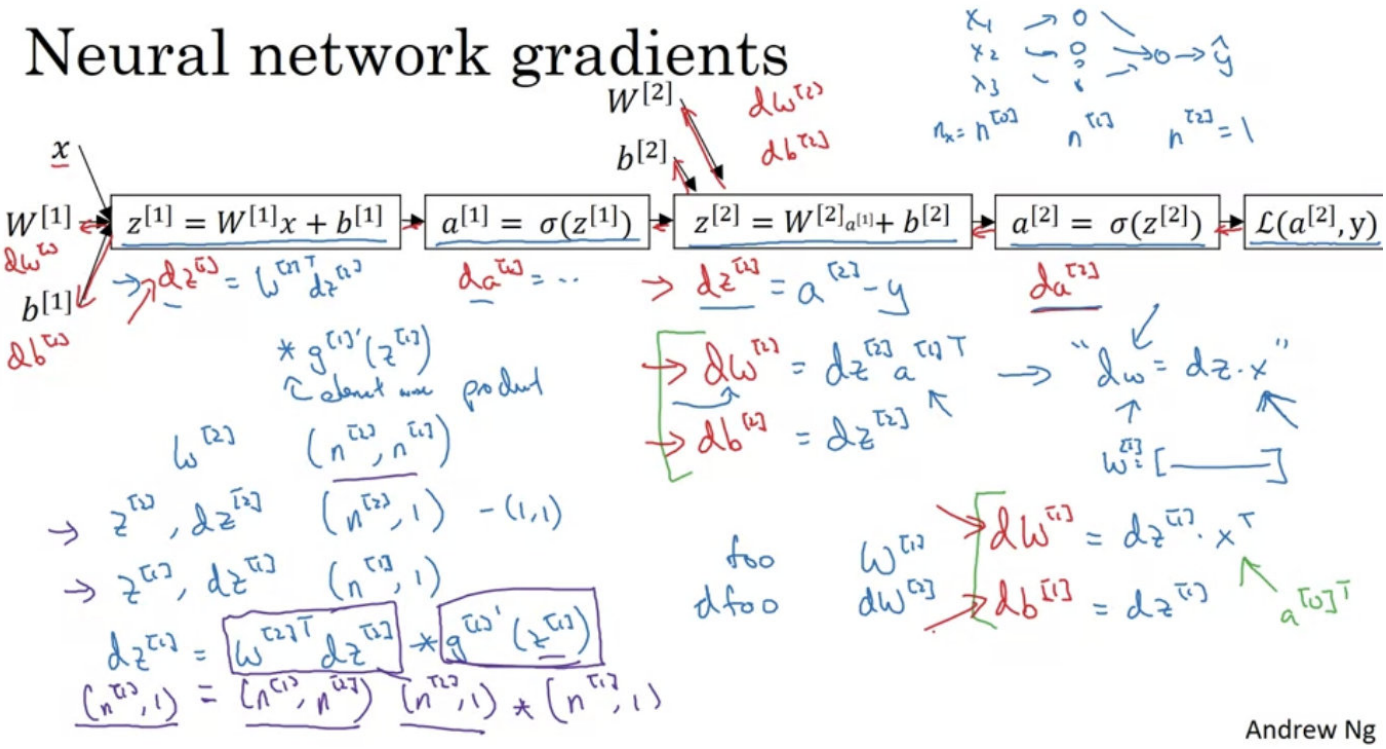
Forward propagation

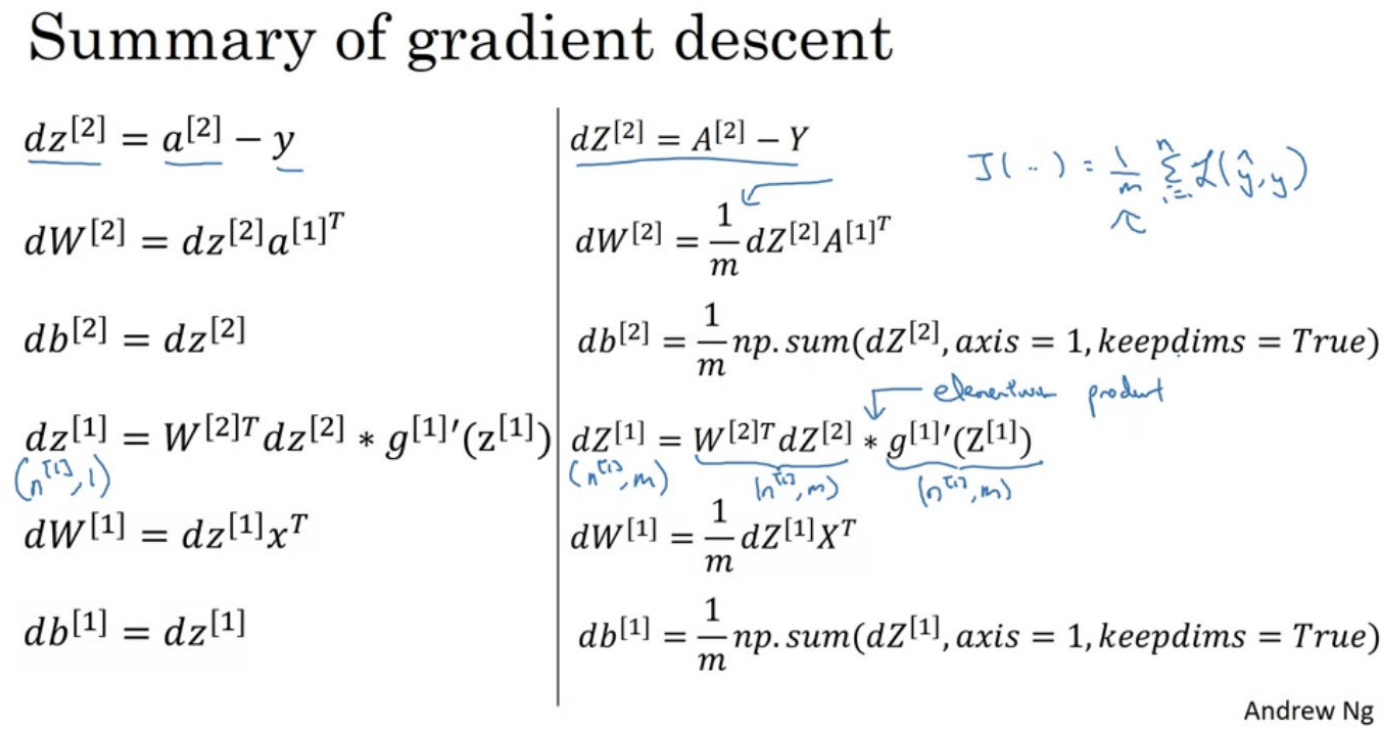
$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = \sigma(Z^{[l]}),\ A^{[0]} = X$$

Backward propagation

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







Random initialization

The purpose of random initialization is to **break symmetry**.

$w^{[1]}$ = np.random.randn((2 ,2)) * 0.01

$b^{[1]}$ = np.zeros((2, 1))

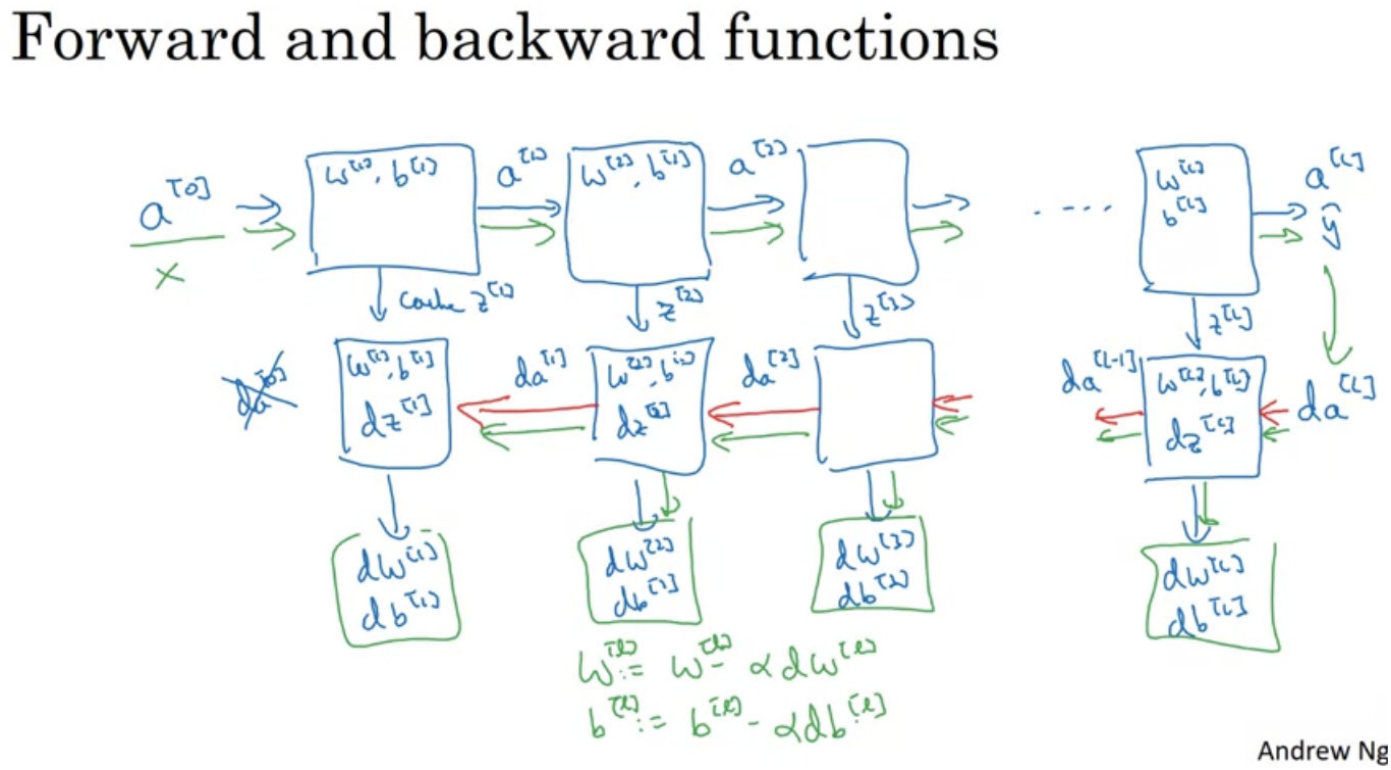
b do not have symmetric breaking problem.

The general methodology to build a Neural Network is to:

1. Define the neural network structure (# of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
    - Implement forward propagation
    - Compute loss
    - Implement backward propagation to get the gradients
    - Update parameters (gradient descent)

Deep Neural Network

> **Notation**
>
> L is the number of layers
>
> $n^{[l]}$ is the number of units in layer l, $n^{[0]} = n_x$
>
> $a^{[l]}$ is the activations in layer l
>
> $w^{[l]}$, $b^{[l]}$ are parameters for $z^{[l]}$
>
> $x = a^{[0]}$, $\hat{y} = a^{[L]}$



Forward propagation for layer l

Input $a^{[l-1]}$

Output $a^{[l]}$, cache($z^{[l]}$)

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = \sigma(Z^{[l]}),\ A^{[0]} = X$$

The dimension of $W^{[l]}$ and $b^{[l]}$

$$W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$$

$$b^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}$$

Backward propagation for layer l

Input $da^{[l]}$

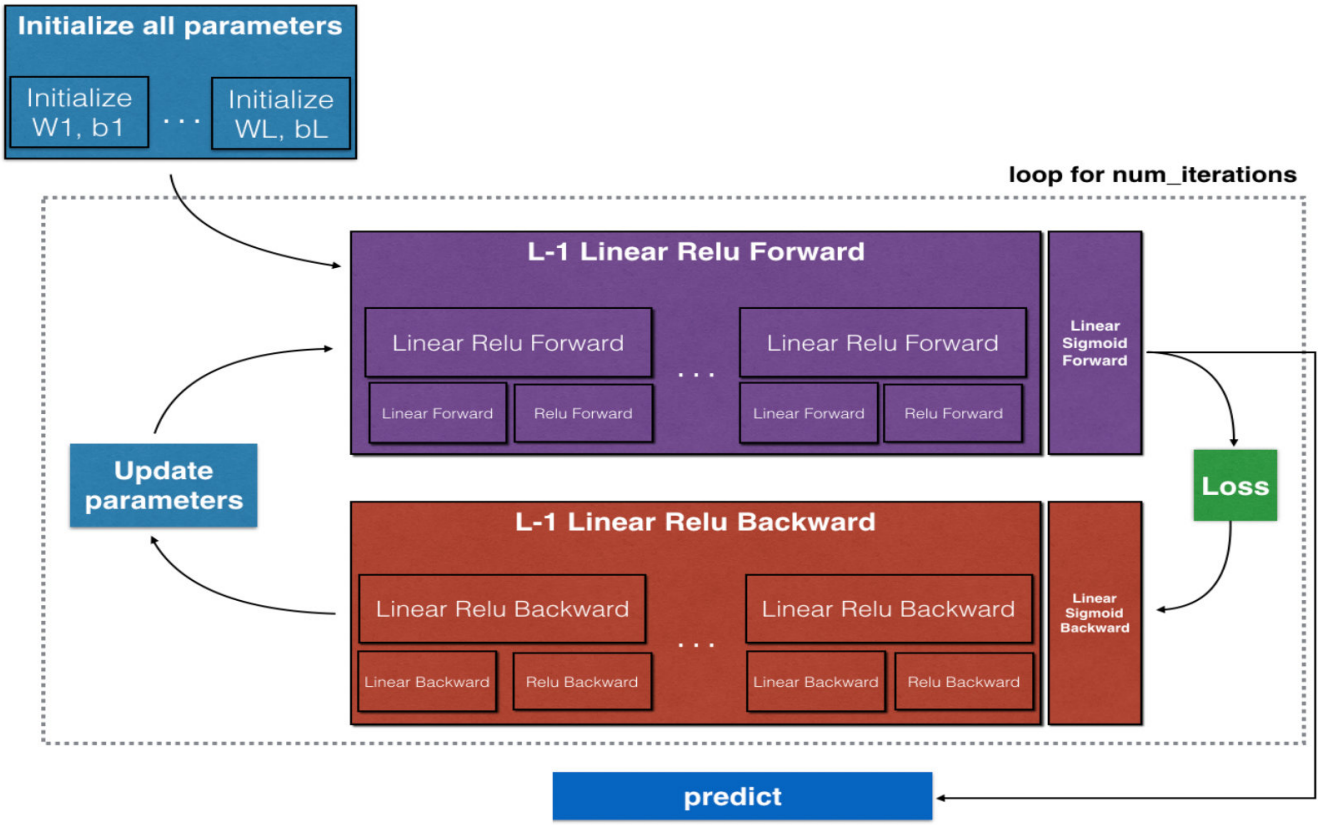
Output $da^{[l-1]},\ dW^{[l]},\ db^{[l]}$

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

$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$
$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$
$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

For final layer

$$dA^{[L]} = \sum_{i=1}^{m} (-\frac{y^{(i)}}{a^{(i)}} + \frac{1-y^{(i)}}{1-a^{(i)}})$$



Hyperparameters

- Learning rate α
- #interations
- #hidden layer Lq
- #Hidden units $n^{[l]}$
- choice of activation function
- ...

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

Train/dev/test sets

- Small dataset -- 70/30, 60/20/20 traditional ratio.
- large dataset -- set the dev and test set smaller than tranditional ratio.

MAKE SURE the dev and test sets come from **same distribution**.

Bias and Variance

High Bias

- More complex model (bigger network)
- ...

High Variance

- More data
- Regularization
- ...

Regularization

Logistic regression

l2 regularization

$$\frac{\lambda}{2m}\|w\|_2^2 = \frac{\lambda}{2m}\sum_{j=1}^{n_x} w_j^2 = \frac{\lambda}{2m} w^\top w$$

l1 regularization (w will be sparse -- lots of 0 in w)

$$\frac{\lambda}{m}\sum_{j=1}^{n_x}|w| = \frac{\lambda}{m}\|w\|_1$$

Neural Network

Frobenius norm

$$\frac{\lambda}{2m}\sum_{l=1}^{L}\|W^{[l]}\|_F^2$$

$$\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(W_{i,j}^{[l]})^2$$

$$dW^{[l]} = (from\ backprap) + \frac{\lambda}{m}W^{[l]}$$

L2 regularization is also called **weight decay** ($1 - \frac{\alpha\lambda}{m} < 1$).

$$\begin{aligned} W^{[l]} &:= W^{[l]} - \alpha[(from\ backprap) + \frac{\lambda}{m}W^{[l]}] \\ &:= W^{[l]} - \frac{\alpha\lambda}{m}W^{[l]} - \alpha(from\ backprap) \\ &:= (1 - \frac{\alpha\lambda}{m})W^{[l]} - \alpha(from\ backprap) \end{aligned}$$

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{2}$$

Dropout regularization

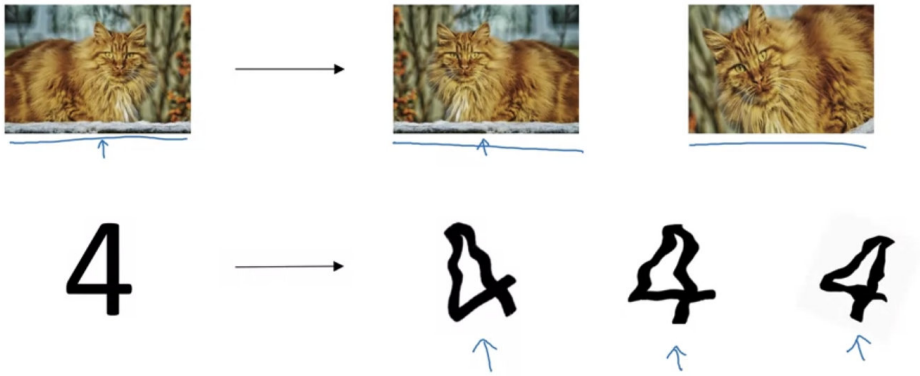
Implementing dropout ("Inverted dropout") -- with l=3 layer

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3) # a3 *= d3
a3 /= keep_prob # Because a3 is reduced by (1 - keep_prob)
```

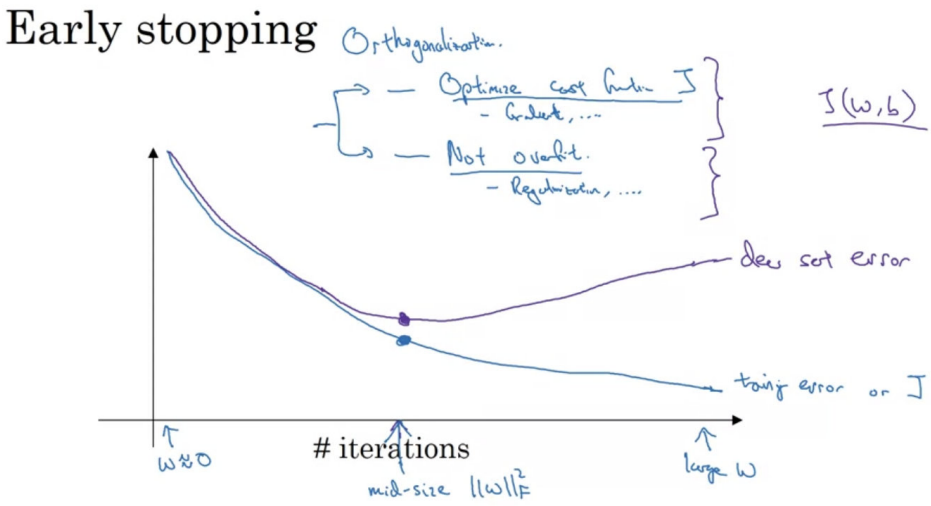
Intuition: Can't rely on any one feature, so have to spread out weights.

Other regularization methods

- Data augmentation



- Early stopping



Setting Up Optimization Problem

Normalizing inputs

1. Move the training set until it has 0 mean.

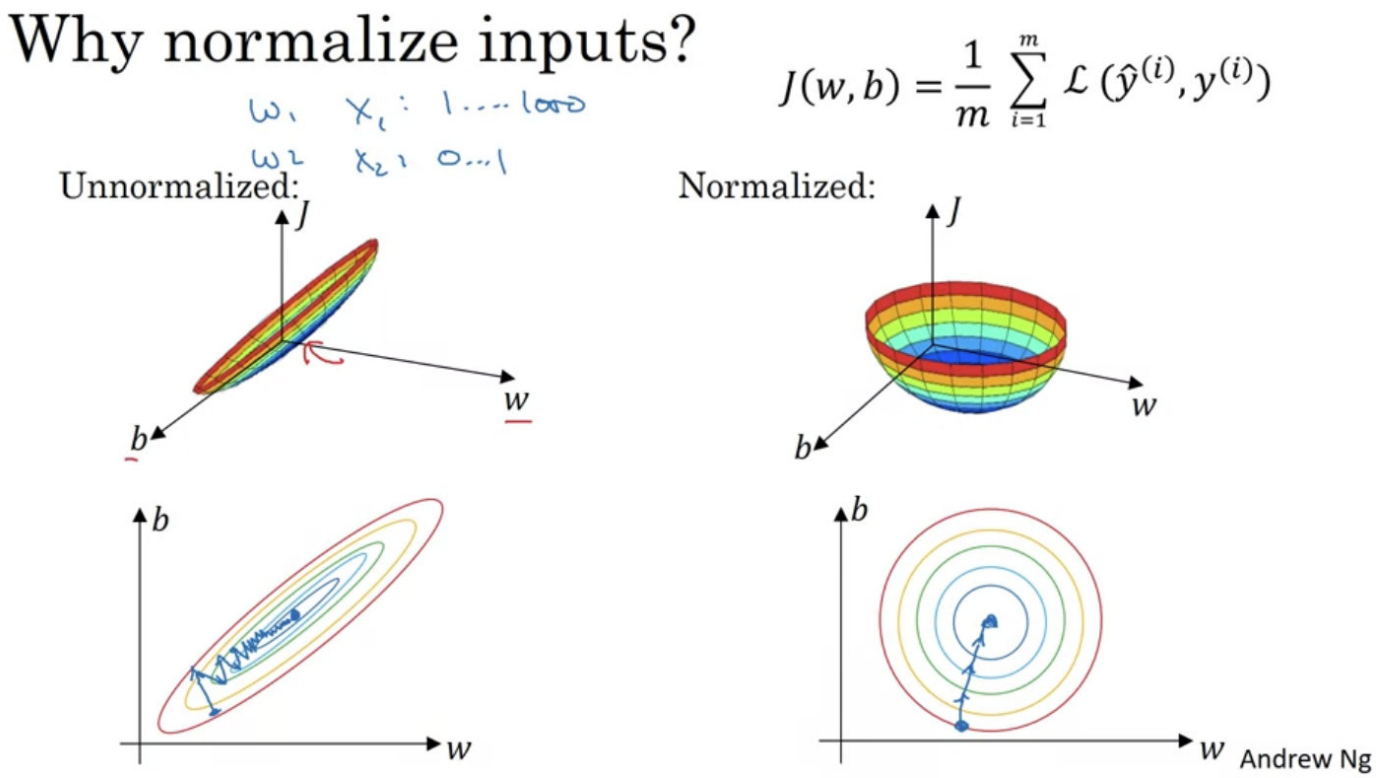
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
$$x := x - \mu$$

2. Normalize the varience

$$\sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2}$$

$$x := \frac{x}{\sigma}$$

Use same μ and σ to normalize the training set and test set.



Vanishing/exploding gradients

For a very deep network,

$w > 1 \rightarrow$ exploding gradient

$w < 1 \rightarrow$ vanishing gradient

Weight initialization for deep networks (partial solution for v/e gradients)

ReLU

He Initialization

$$W^{[l]} = \text{np.random.randn(shape)} * \sqrt{\frac{2}{n^{[l-1]}}}$$

tanh

Xavier initialization

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

Other version

$$W^{[l]} = \text{np.random.randn(shape)} * \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$$

Numerical approximation of gradients

For a very small ϵ,

$$f'(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

Use this formular for **gradient checking**.

Gradient checking for a neural network

Take $W^{[1]}$, $b^{[1]}$, $\cdots$, $W^{[L]}$, $b^{[L]}$ and reshape into a big vector θ.

Take $dW^{[1]}$, $db^{[1]}$, $\cdots$, $dW^{[L]}$, $db^{[L]}$ and reshape into a big vector $d\theta$.

$$\text{For each i:}$$

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

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

To see whether $d\theta_{approx} = d\theta$

$$\text{Check} \quad \frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$$

Notes for grad check:

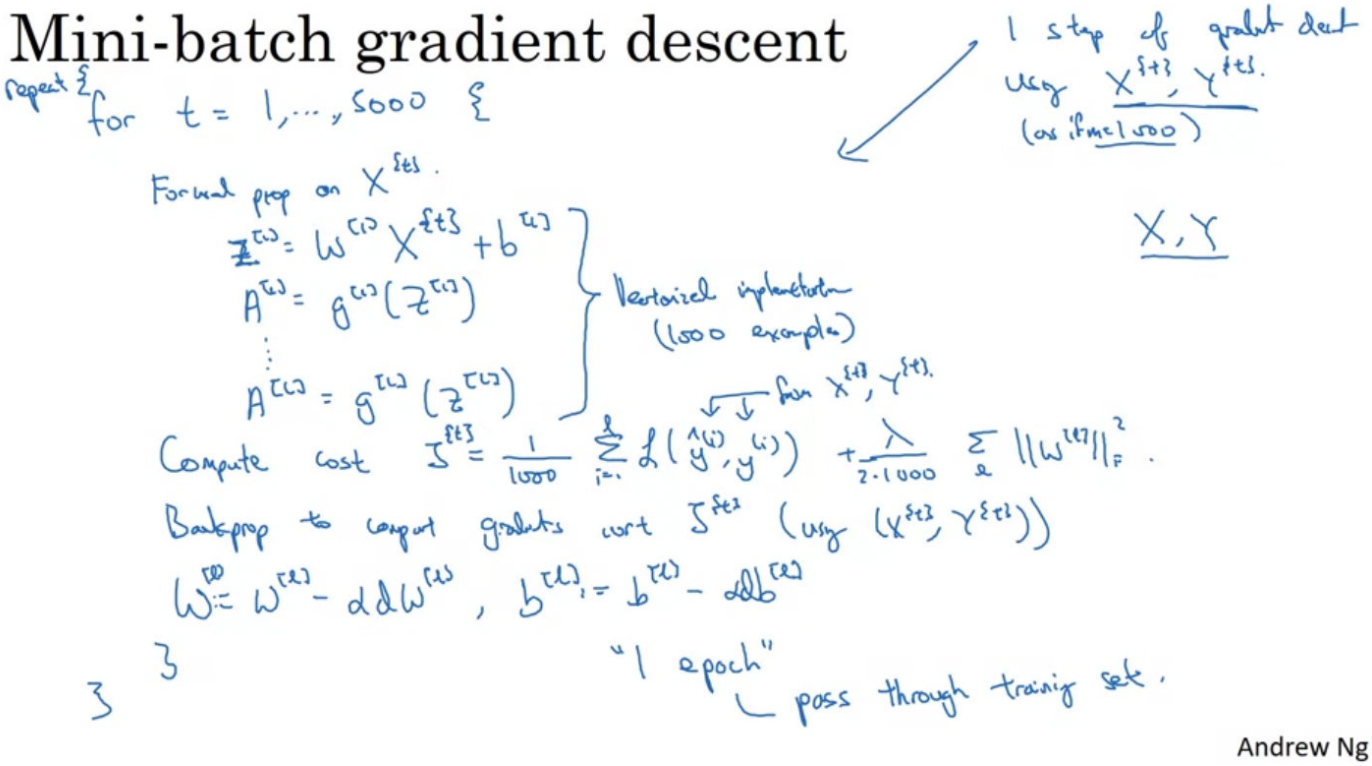
- Don't use in training - only to debug.
- If algorithm fails grad check, look at components ($db^{[l]}$, $dw^{[l]}$) to try to indentify bug.
- Remeber regularization.
- Doesn't work with dropout.
- Run at random initialization; perphaps again after some traning.

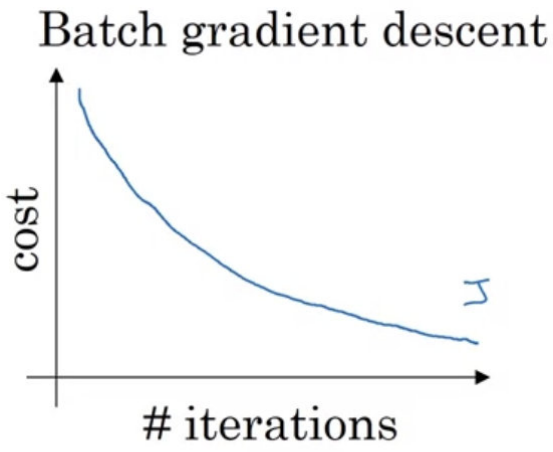
Optimization Algorithms

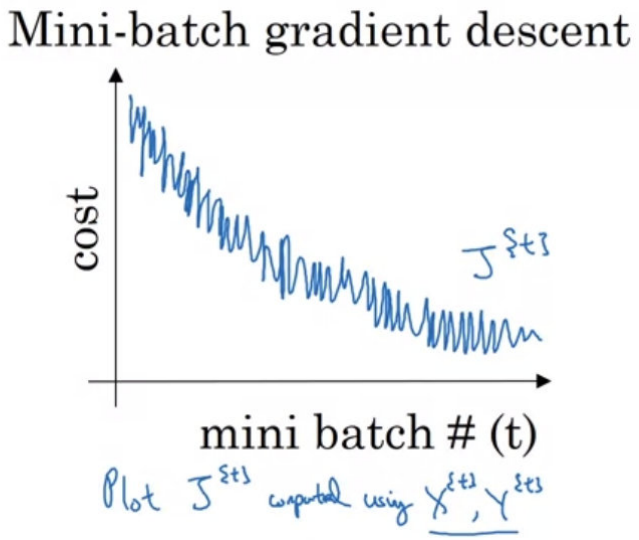
Mini-batch gradient descent

Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$

> $x^{(i)} \rightarrow$ the ith example
>
> $z^{[l]} \rightarrow$ the z value for the L layer of the neural network
>
> $X^{\{t\}}, Y^{\{t\}} \rightarrow$ mini-batch







Choosing the mini-batch size

If nimi-batch size = m: Batch gradient descent.

If nimi-batch size = 1: Stochasitc gradient descent.

IF small train set (m <= 2000): Use batch GD

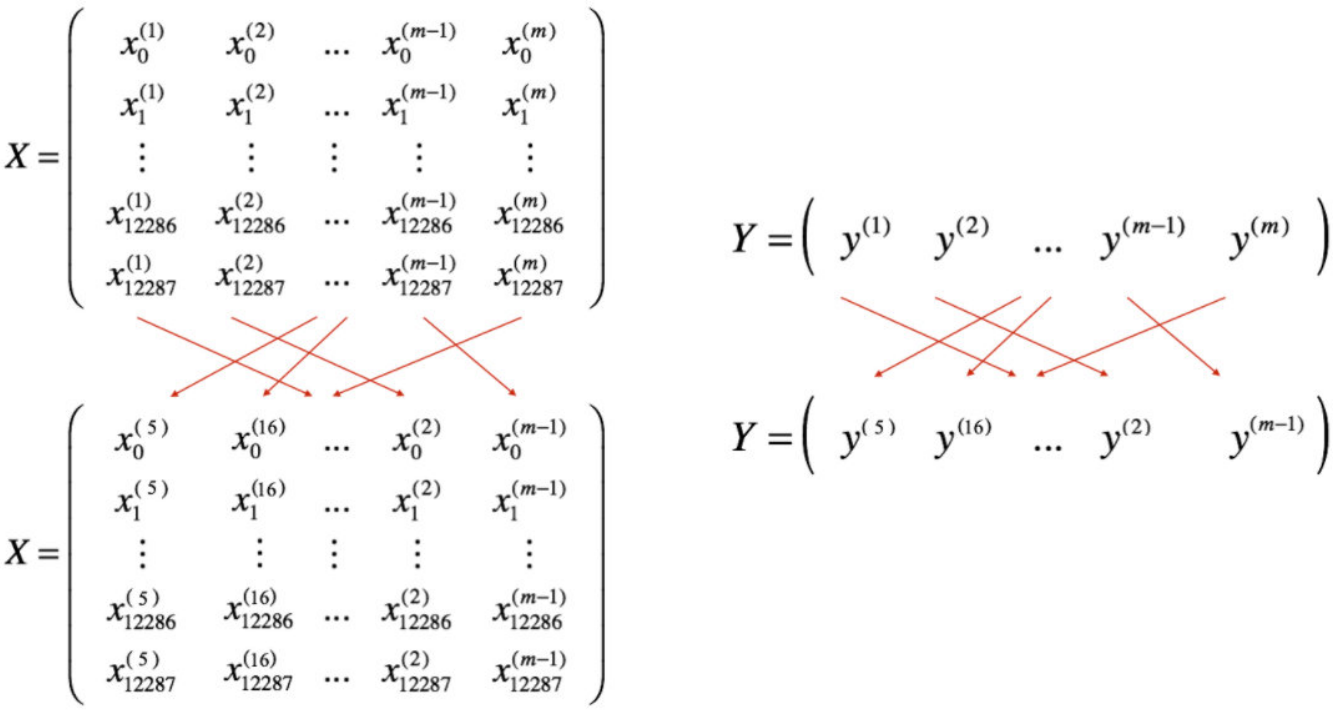
Typical mini-batch size: 32, 64, 128, 256, 512

The size is to the power of 2 (the computer will run faster).

Make sure mini-batch **fit in CPU/GPU memory**.

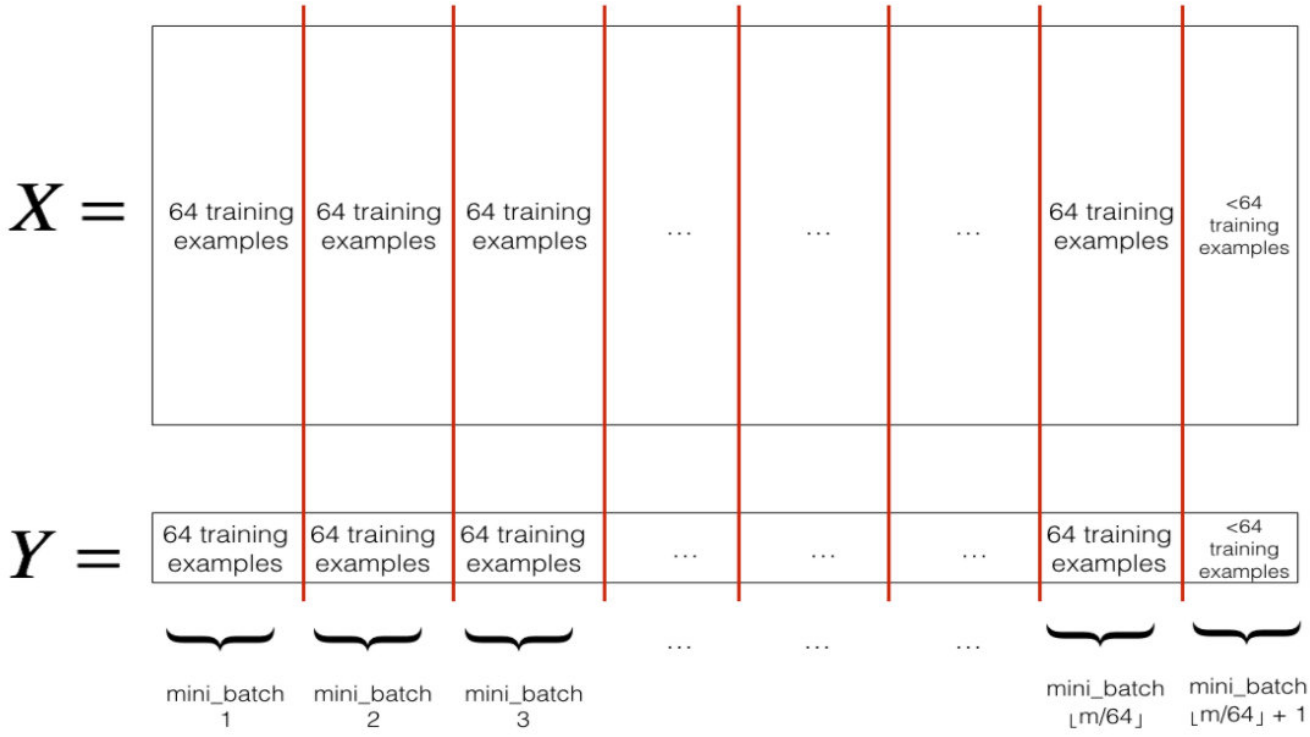
Set Up

1. **Shuffle**: Create a shuffled version of the training set (X, Y) as shown below. Each column of X and Y represents a training example. Note that the random shuffling is done synchronously between X and Y. Such that after the shuffling the i^{th} column of X is the example corresponding to the i^{th} label in Y. The shuffling step ensures that examples will be split randomly into different mini-batches.



```
permutation = list(np.random.permutation(m))  #m为样本数
shuffled_X = X[:, permutation]
shuffled_Y = Y[:, permutation].reshape((1,m))
```

2. **Partition**: Partition the shuffled (X, Y) into mini-batches of size `mini_batch_size` (here 64). Note that the number of training examples is not always divisible by `mini_batch_size`. The last mini batch might be smaller, but you don't need to worry about this. When the final mini-batch is smaller than the full `mini_batch_size`, it will look like this:

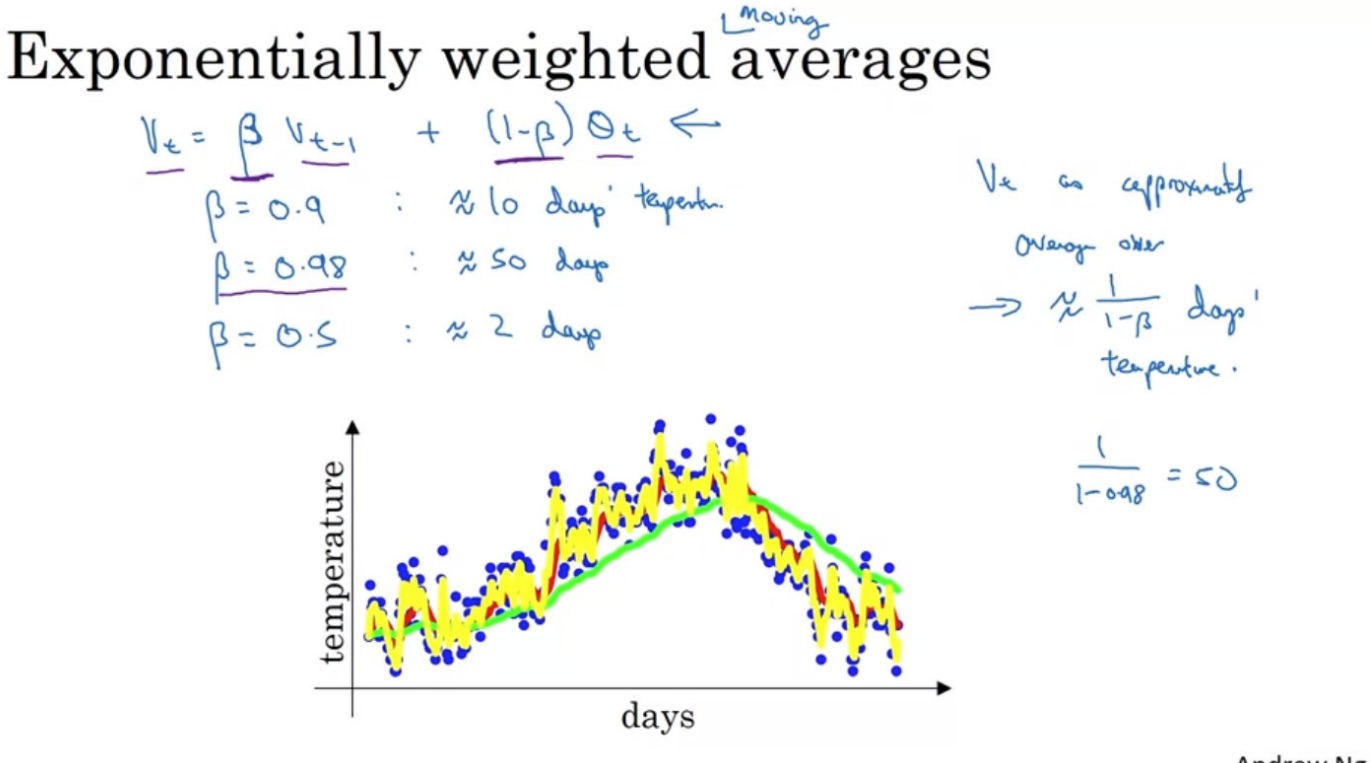


Exponentially weighted averages

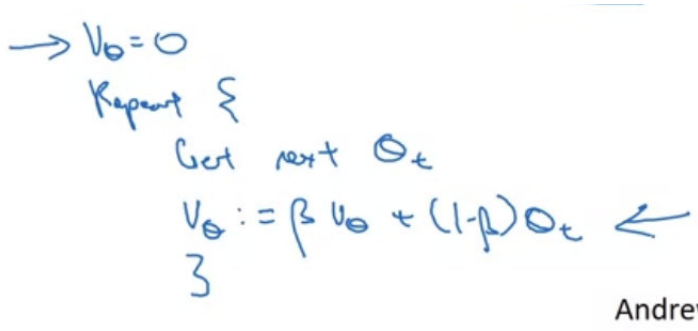
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

V_t is approximately averaging over $\frac{1}{1-\beta}$ examples

$$(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e} \approx 0.35$$



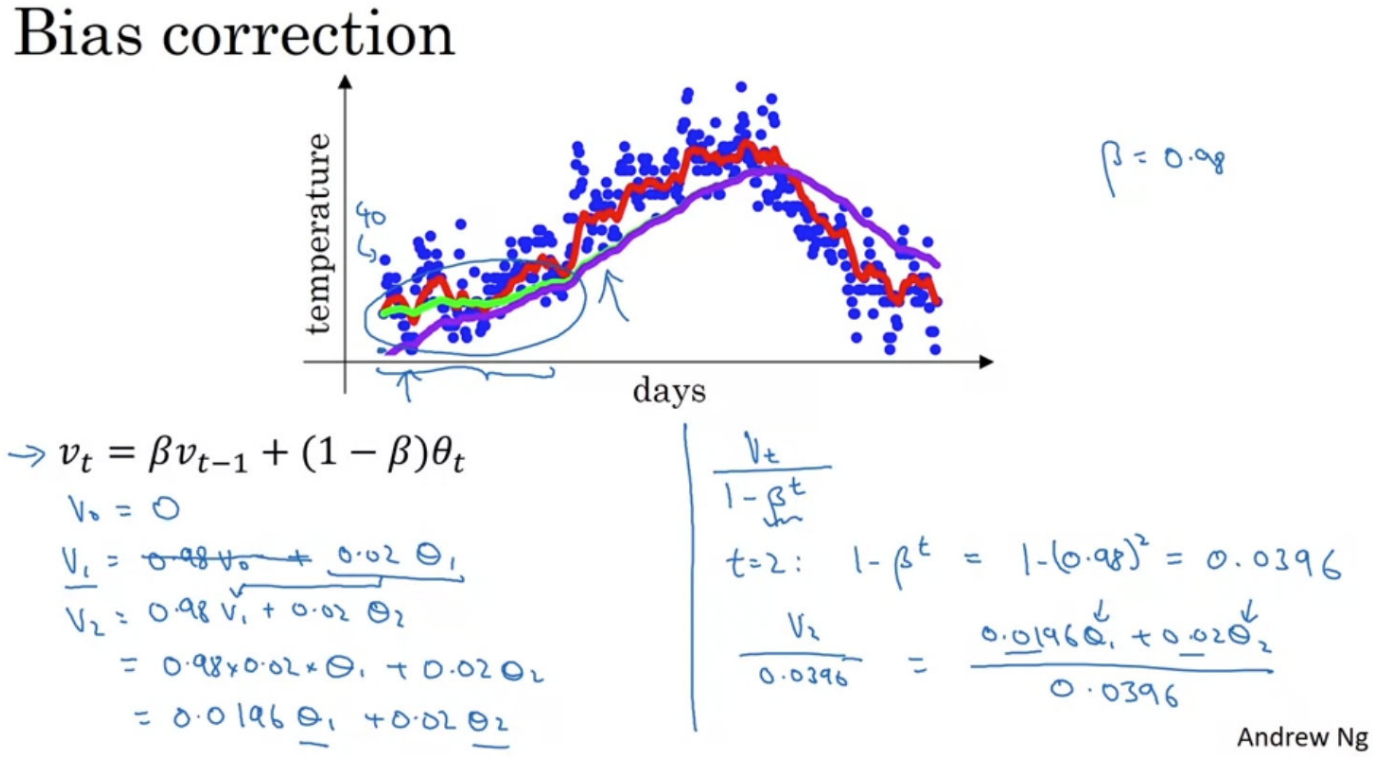
Andrew Ng



The **efficiency** of the exponentially weighted averages is because it the computer can overwrite V_θ over and over (and only 1 line code), and this requires very small memory. In contrast, the direct computation of average requires very large memory when averaging over many examples.

Bias correction

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



Normally, people just wait for warming up in practice.

Gradient descent with momentum

Now, implement the parameters update with momentum. The momentum update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

In one sentence, the basic idea is to **compute an exponentially weighter average of gradients, and then use that gradient to update weights instead**.

```
VdW, Vdb = zeros(same dimension with W and b)
On iteration t:
  Compute dW, db on current mini-batch
  VdW = beta * VdW + (1 - beta) * dW
  Vdb = beta * Vdb + (1 - beta) * db
  W = W - learning_rate * VdW
  b = b - learning_rate * Vdb
```

The most common value for β is 0.9.

In some literature, $(1-\beta)$ is omitted. That is just scaling the VdW or Vdb by $\frac{1}{1-\beta}$ (it will affect the learning rate).

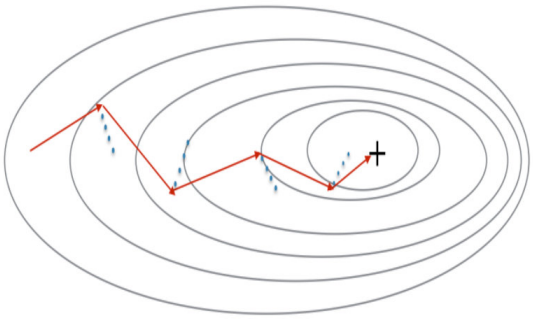


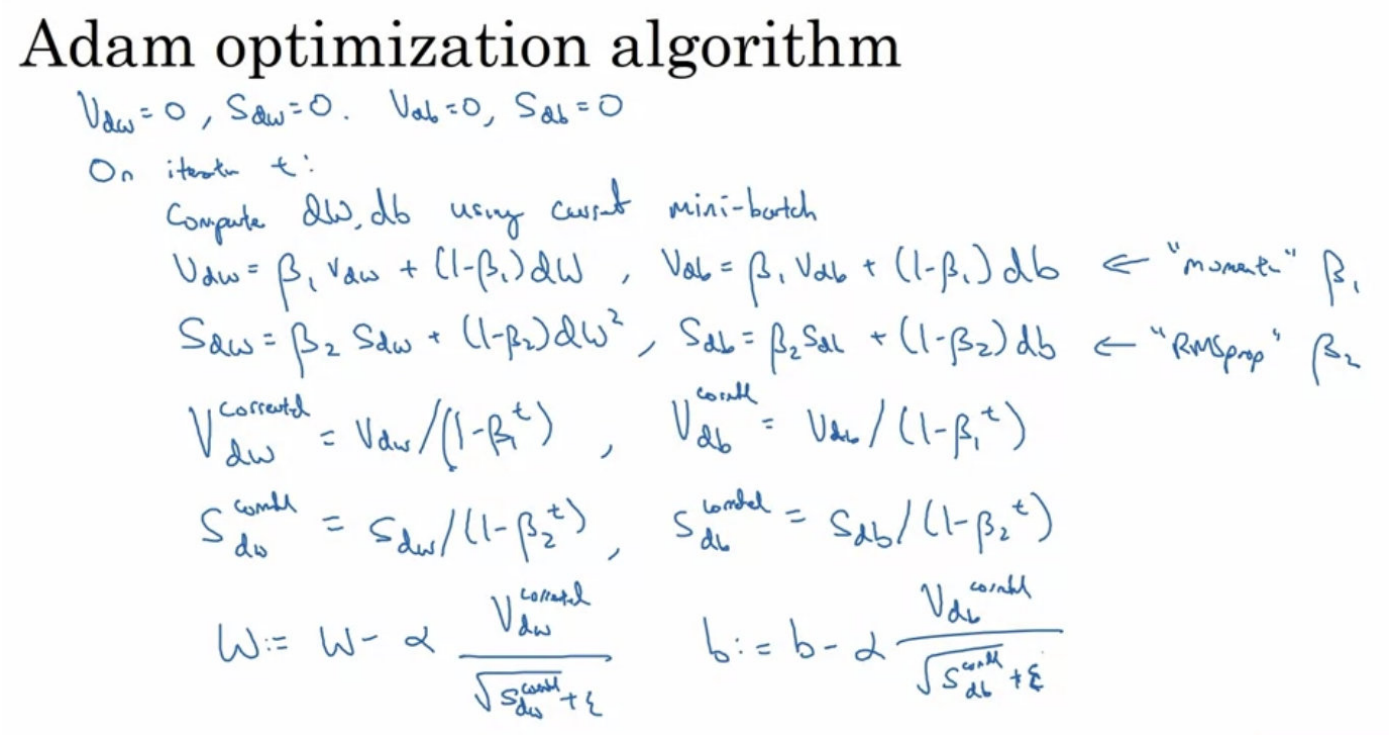
Figure 3 : The red arrows show the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step. Rather than just following the gradient, the gradient is allowed to influence v and then take a step in the direction of v.

RMSProp (Root Mean Square Prop)

```
VdW, Vdb = zeros(same dimension with W and b)
On iteration t:
  Compute dW, db on current mini-batch
  SdW = beta * VdW + (1 - beta) * dW ** 2
  Sdb = beta * Vdb + (1 - beta) * db ** 2
  W = W - learning_rate * dW / (sqrt(SdW + epsilon)) #make sure not 0
  b = b - learning_rate * db / (sqrt(Sdb + epsilon))
```

Adam optimization algorithm (Adaptive Moment Estimation)

```
VdW = 0, SdW = 0, Vdb = 0, Sdb = 0
On iteration t:
  Compute dW, db on current mini-batch
  VdW = beta_1 * VdW + (1 - beta_1) * dW #'momentum' beta_1
  Vdb = beta_1 * Vdb + (1 - beta_1) * db #'momentum' beta_1
  SdW = beta_2 * VdW + (1 - beta_2) * dW ** 2 #'RMSProp' beta_2
  Sdb = beta_2 * Vdb + (1 - beta_2) * db ** 2 #'RMSProp' beta_2
  VdW_corrected = VdW / (1 - beta_1 ** t)
  Vdb_corrected = Vdb / (1 - beta_1 ** t)
  SdW_corrected = SdW / (1 - beta_2 ** t)
  Sdb_corrected = Sdb / (1 - beta_2 ** t)
  W = W - learning_rate * VdW_corrected / (sqrt(SdW_corrected + epsilon))
  b = b - learning_rate * Vdb_corrected / (sqrt(Sdb_corrected + epsilon))
```



How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, \dots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1-\beta_1)\frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v_{dW^{[l]}}^{corrected} = \frac{v_{dW^{[l]}}}{1-(\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1-\beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s_{dW^{[l]}}^{corrected} = \frac{s_{dW^{[l]}}}{1-(\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{corrected}}{\sqrt{s_{dW^{[l]}}^{corrected}}+\varepsilon} \end{cases}$$

Hyperparameters choice:

α: needs to be tuned

β_1: 0.9 (recommend) $\rightarrow dw$ (first moment)

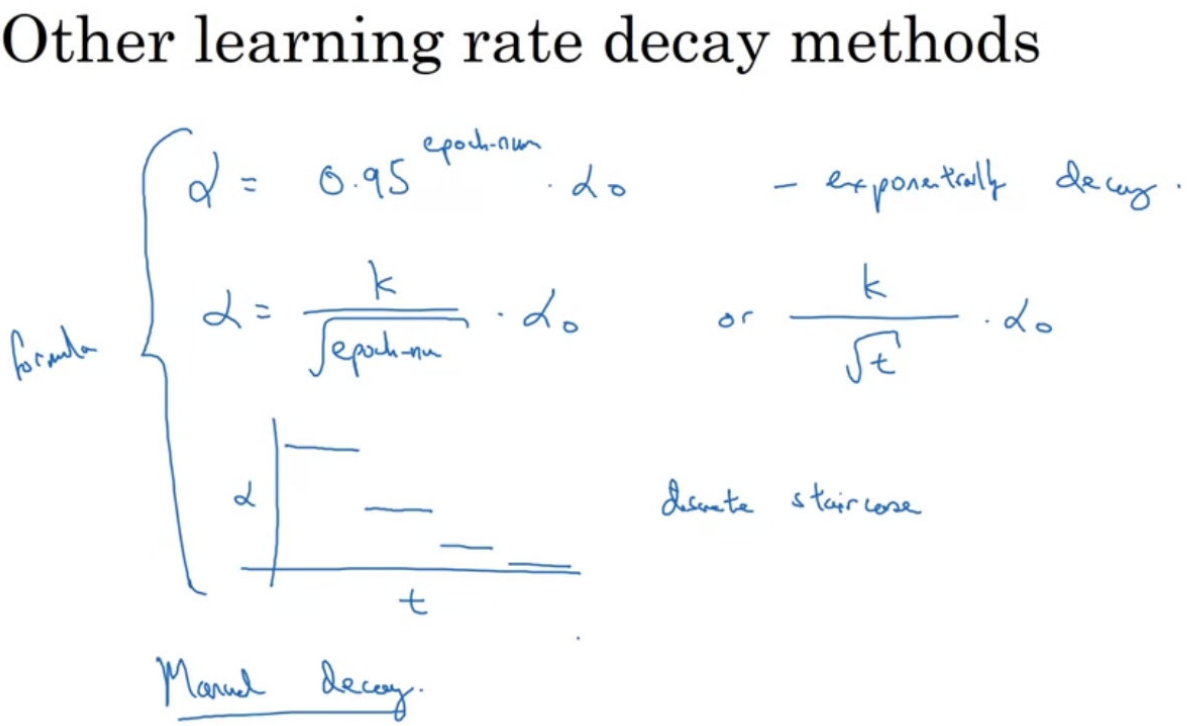
β_2: 0.999 (recommend) $\rightarrow dw^2$ (second moment)

ϵ: 10^{-8} (recommend)

Learning rate decay

$$1\ epoch = 1\ pass\ through\ data$$

$$\alpha = \frac{1}{1 + decay\ rate \times epoch\_num}\alpha_0$$



Andrew Ng

Hyperparameter Tuning

(When there are many hyperparameters) **Use `random sampling` instead of `grid search`!**

Appropriate scale for hyperparameters

It seems more reasonable to search for hyperparameters on a log scale.

```
r = -4 * np.random.rand() # r in [-4, 0]
alpha = 10 ** r
```

When tuning hyperparameters for exponentially weighted averages, instead of sample β, we sample $(1 - \beta)$ on a log scale ($\beta = 1 - 10^r$).

Batch Normalization

Give some intermediate values in NN $z^{(1)}, \cdots, z^{(m)}$

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

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

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

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

where γ and β here are learnable parameters, γ and β **control the mean and variance** of $\tilde{z}^{(i)}$.

Implementing gradient descent

for $t = 1 \ldots$ numMiniBatches

Compute forward prop on $X^{\{t\}}$.

In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$.

Use backprop to compute $dw^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$

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

$\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$

$\gamma^{[l]} := \ldots$

Works w/ momentum, RMSprop, Adam.

Batch Norm at test time

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

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

μ, σ^2: estimate using exponentially weighted average (across mini-batches).

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$

$\mu^{\{1\}[l]}$ $\mu^{\{2\}[l]}$ $\mu^{\{3\}[l]}$ $\rightarrow$ μ

θ_1 θ_2 θ_3 σ^2

$\sigma^{2\{1\}[l]}$ $\sigma^{2\{2\}[l]}$ ...

$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ $\tilde{z} = \gamma z_{norm} + \beta$

Andrew Ng

Softmax regression

Activation function

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{i=1}^{C} e^{z^{[l]}}}$$

where C is the number of classes.

Loss function

$$L(\hat{y}, y) = -\sum_{j=1}^{C} y_j \log \hat{y}_j$$

Back propagation

$$dZ^{[L]} = \hat{y} - y$$

Structuring Machine Learning Projects

Orthogonalization

Tune exactly one knob to tune the system.

Satisficing and Optimizing Metric

$$\textbf{maximize } optimizing\ metric \ \textbf{ s.t. } satisficing\ metric$$

N metrics: `1` optimizing, `N - 1` statisficing.

Guideline for choosing dev/test set

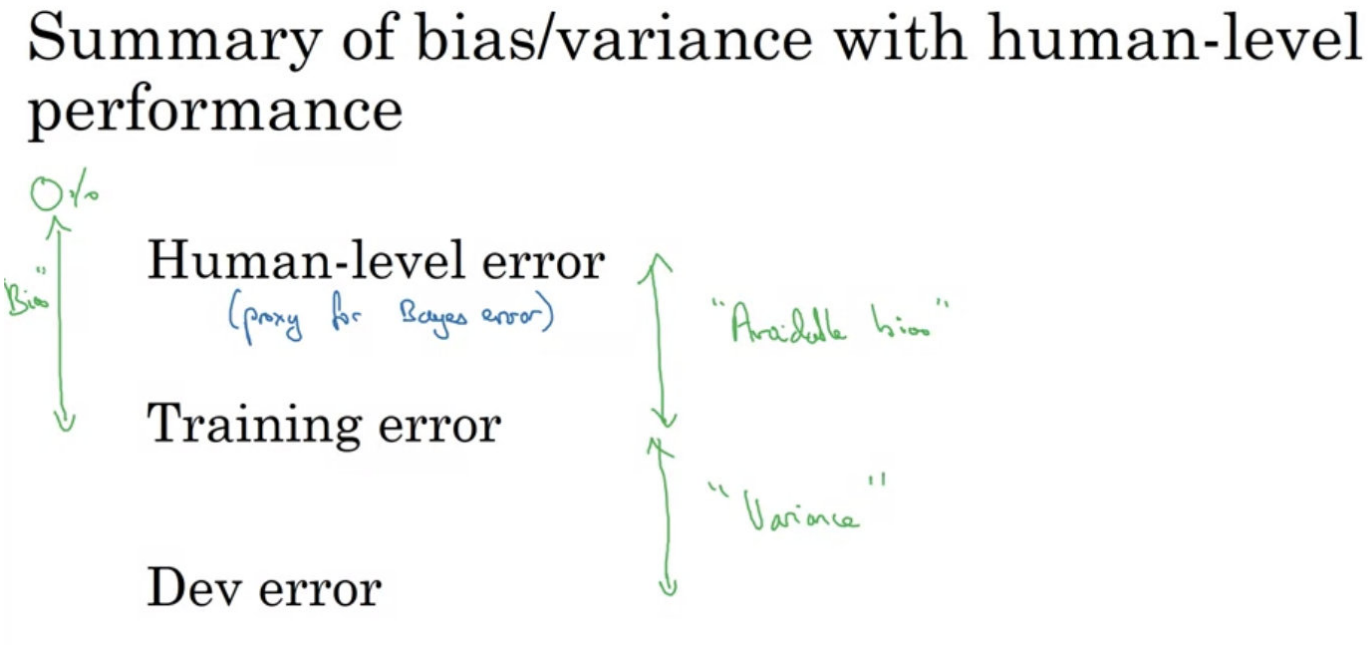
Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on. And the dev set and test set should come from `same distribution`.

Set the test set to be **big enough to give high confidence** i the overall performance of the system.

Comparing with human-level error

Bayes error: the best possible error any function could achieve. **Human-level error** is an `approximation` of Bayes error.

Avoidable bias: the difference between bayes errior and training error.



Improving Model Performance

Two fundamental assumptions of supervised learning

1. You can fit the training set pretty well. `Avoidable bias`
2. The traning set performance generalizes pretty well to the dev/test set. `Variance`

Avoidable bias

- Train bigger model
- Train longer/better optimization algorithms (momentum, RMSProp, Adam...)
- NN architecture/hyperparameters search

Variance

- More data
- Regularization (l2, dropout, data augmentation)
- NN architecture/hyperparameters search

Error analysis

Find a set of mislabeled examples in dev set, and look at the mislabeled examples for false positives and false negatives. Then count up the nimber of errors that fall into various different categories.

Error analysis

Image	Dog	Great Cat	Blurry	Incorrectly labeled	Comments
...					
98				✓	Labeler missed cat in background
99		✓			
100				✓	Drawing of a cat; Not a real cat.
% of total	8%	43%	61%	6%	

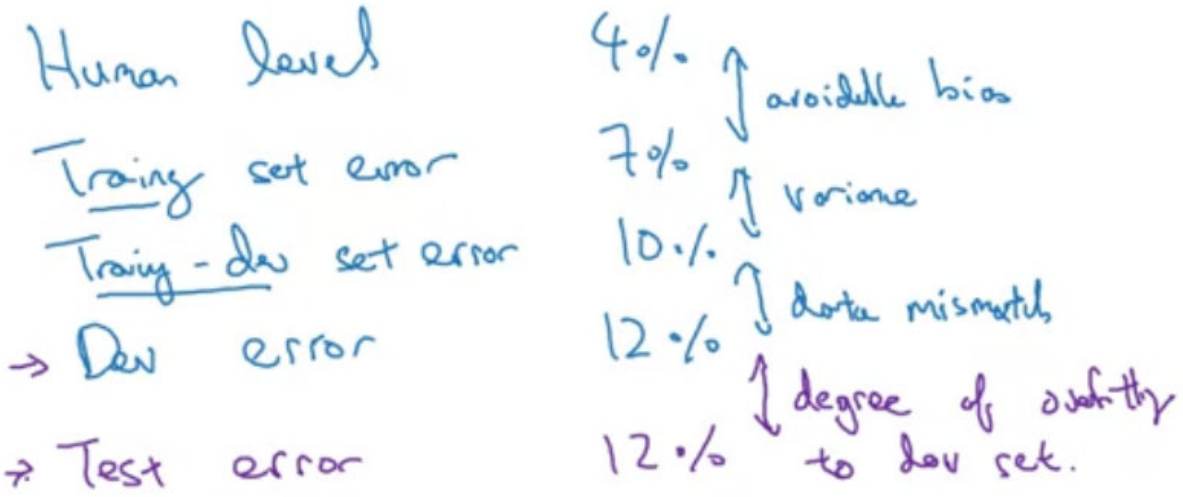
Clearning Up Incorrectly Labeled Data

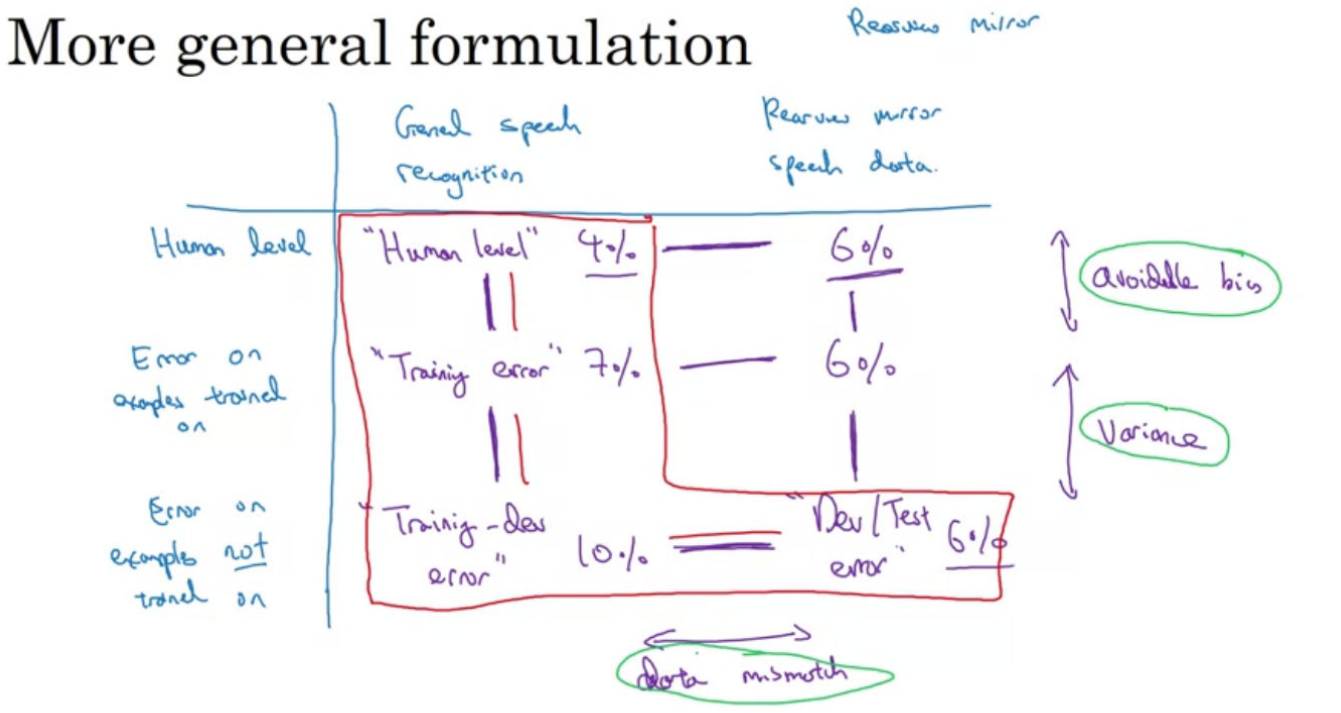
DL algoritms are quite **robust** to random errors in the **training set**.

When correcting incorrect dev/test set examples

- Apply same process to dev and test sets to make sure they continue to come from the same distribution.
- Consider examining examples the algorithm got right as well as ones it got wrong.
- Train and dev/test data may now come from slightly different distributions.

Bias/varience on mismatched training and dev/test sets





Andrew Ng

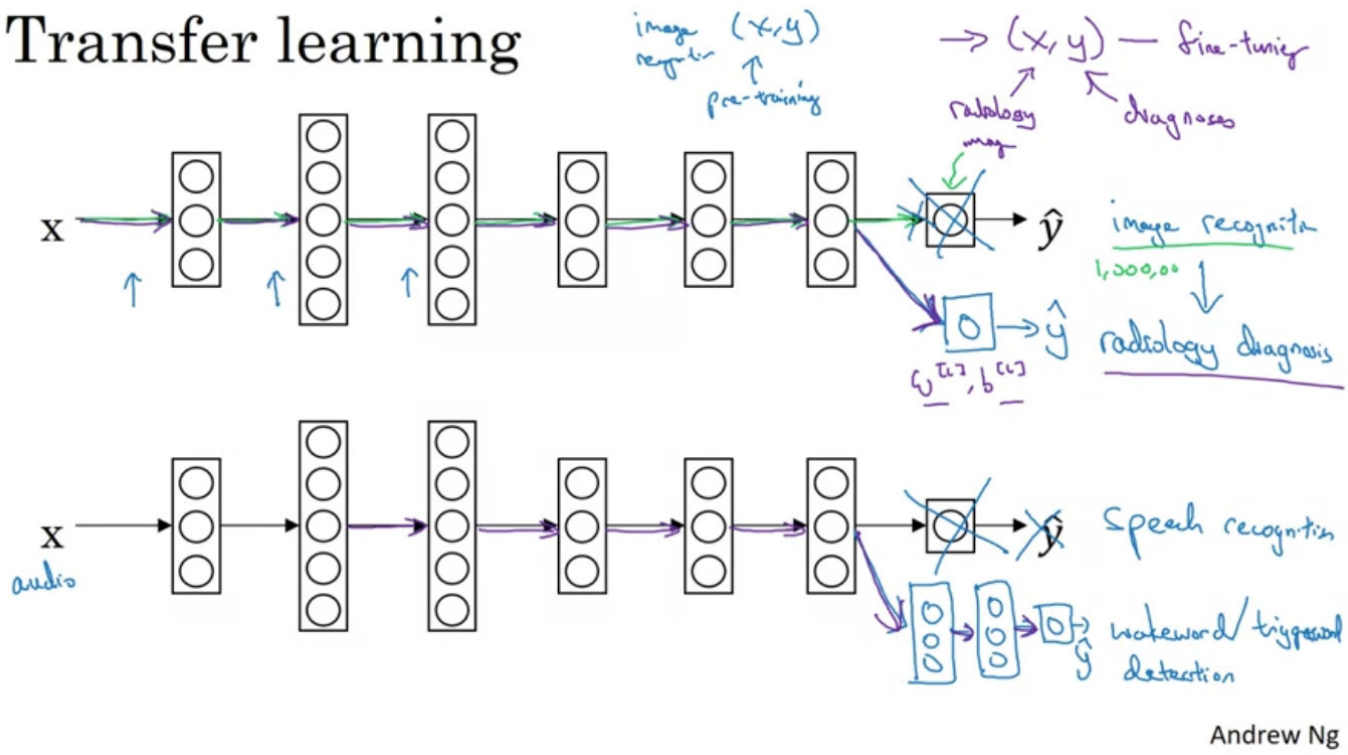
Addressing Data Mismatch

- Carry out **manual error analysis** to try to understand difference between training and dev/test sets.
    - e.g. noisy
- Make training data more similar; or colloect more data similar to dev/test sets.
    - Artificial data synthesis

Transfer Learning

Transfer learning makes sense when you have **a lot of** data for the problem you are `transferring from`, and relatively **less** data for the problem you are `transferring to`. To be more specific,

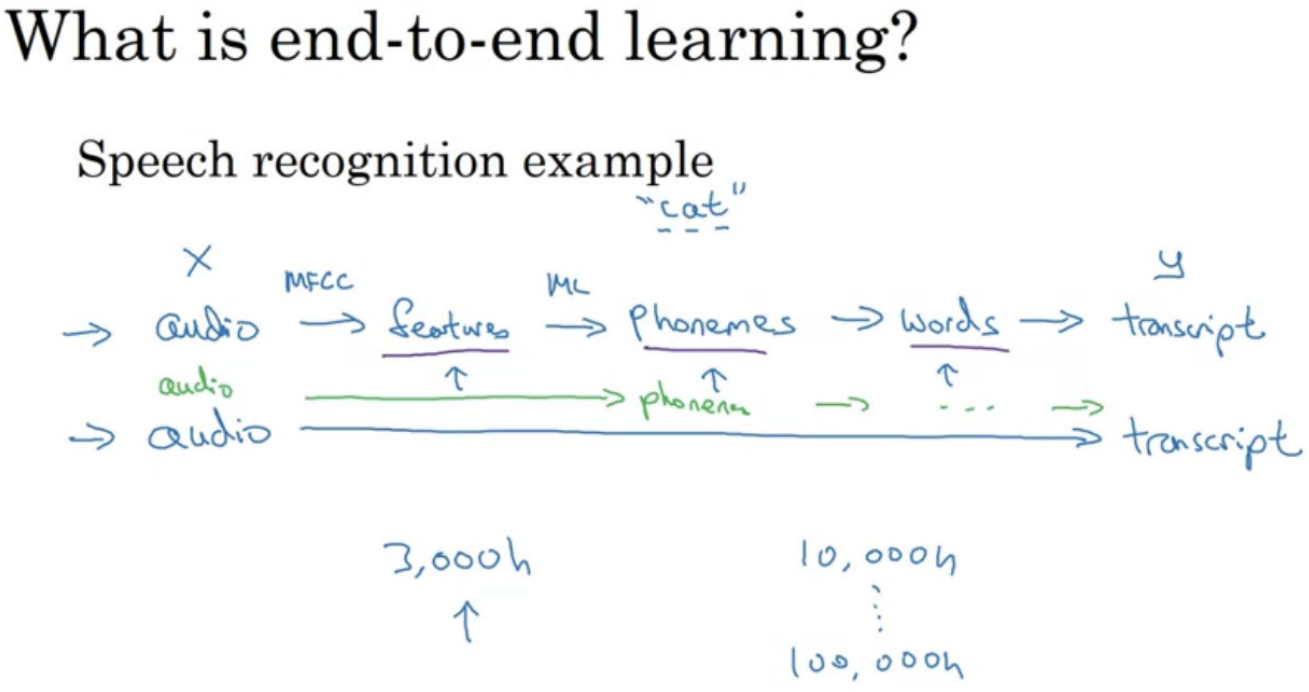
- Task A and B have the **same input x**.
- You have a lot more data for A than B
- Low level features from A could be helpful for B.



Multi-task learning

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually: Amount of data you have for each task is quite similar.

End-to-end Deep Learning



Pros and cons of end-to-end deep learning

Pros

- Let the data speak
- Less hand-desogning of components needed

Cons

- May need large amount of data
- Excludes potentially useful hand-designed components

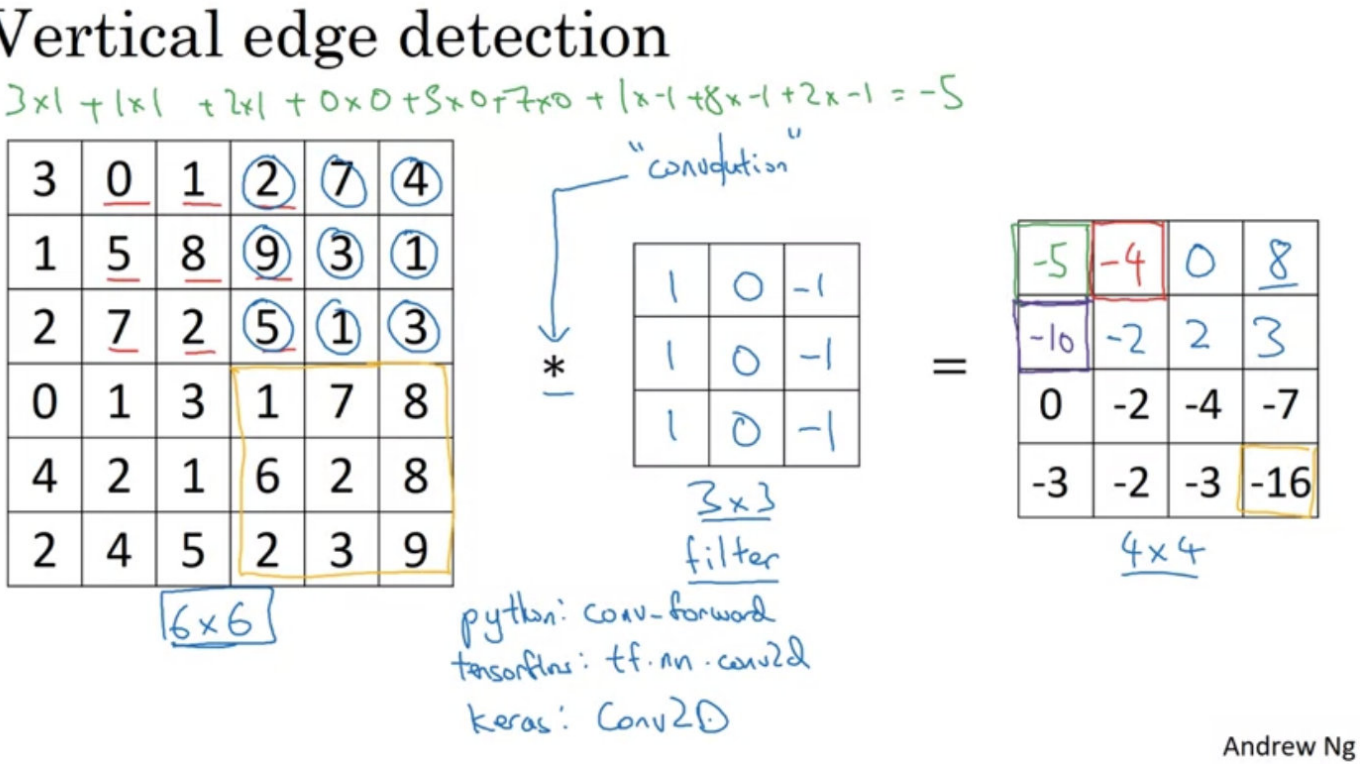
Convolutional Neural Networks

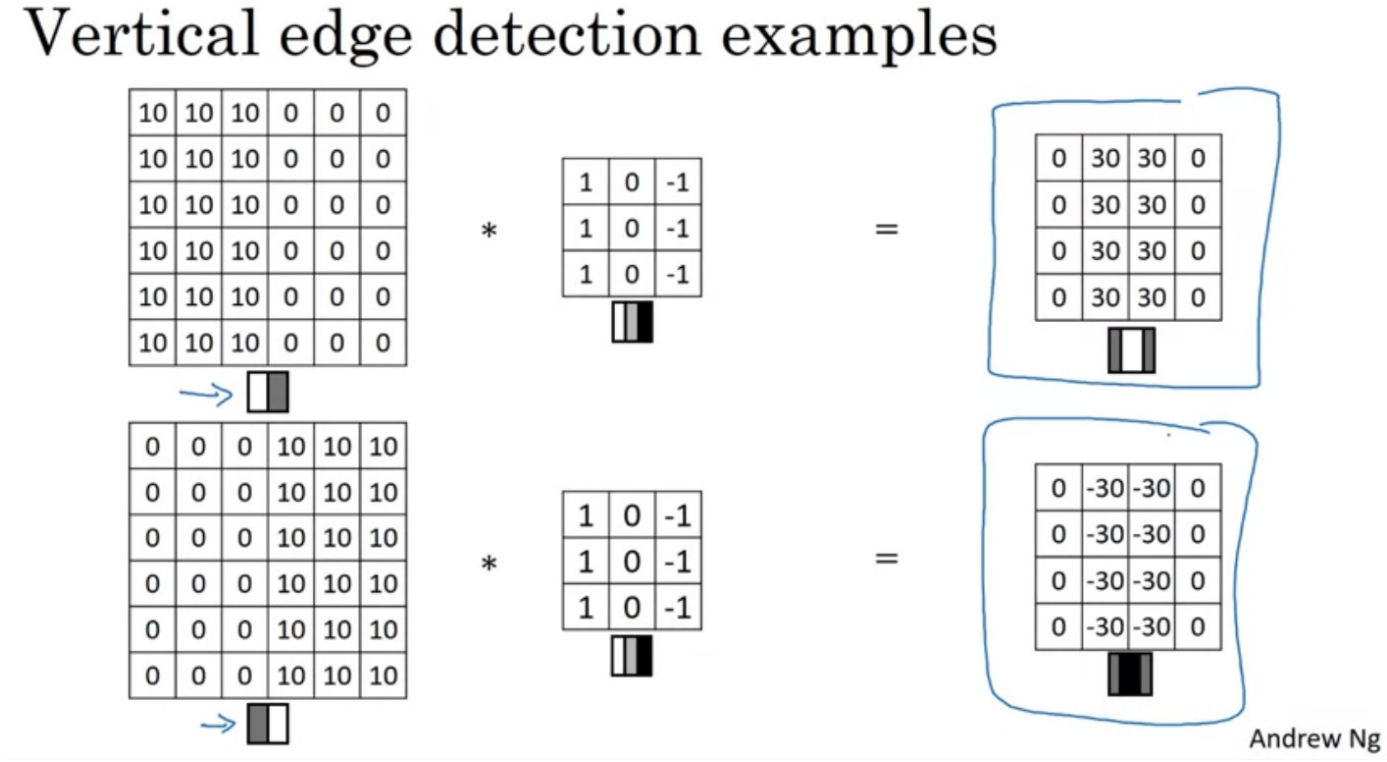
Computer Vision

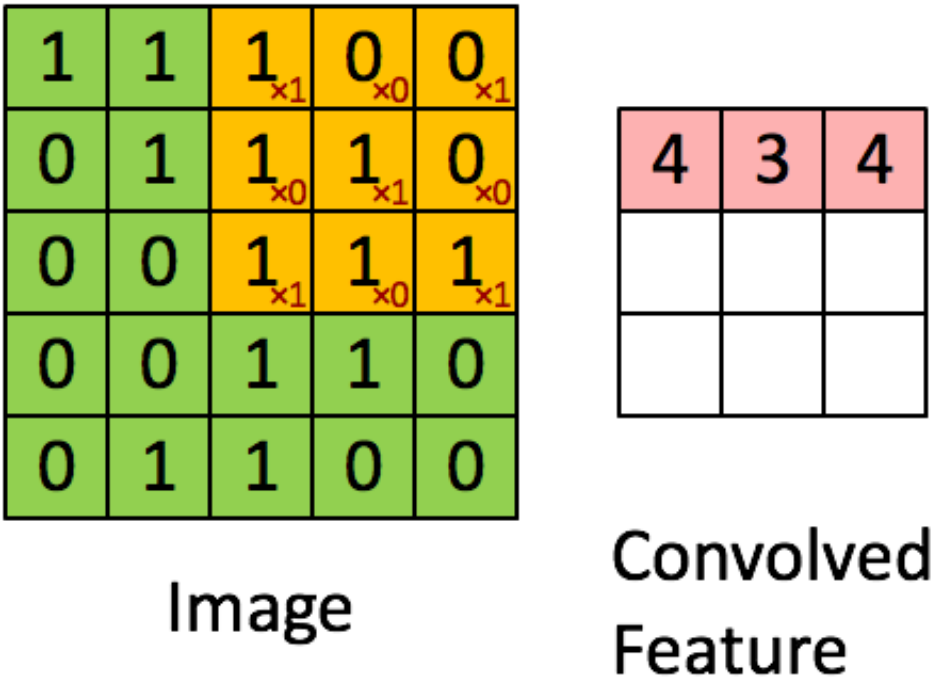
Image Classification, Object Detection, Neural Style Transfer...

The problem is that, when we use a large image, the input size will be extremely large. So, we need to implement **convolution operation**.

Edge Detection Example







Padding

A $n \times n$ image convolved by a $f \times f$ filter, the dimention of output will be $n - f + 1 \times n - f + 1$.

There are two major darwbacks:

- The image shrinks every time when applying convolutional operator.
- The pixels on the corners or on the edges are use much less in the output.

In order to fix both of these problems: **Pad** the image with p (padding amount) before applying convolutional operation. Then the output size becomes $n + 2p - f + 1 \times n + 2p - f + 1$.

Valid and Same convolutions

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

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

$$n + 2p - f + 1 = n$$
$$p = \frac{f - 1}{2}$$

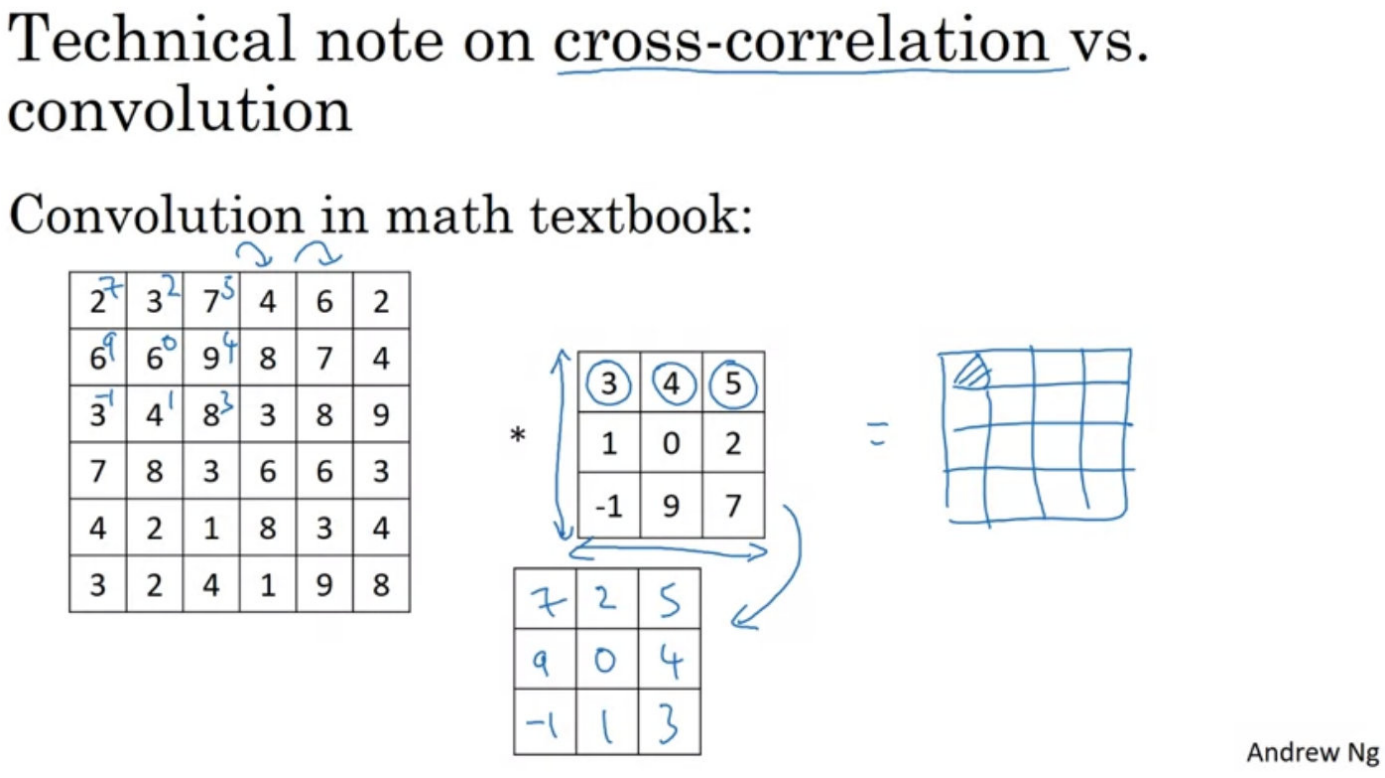
f is usually (almost always) `odd`.

Strided convolution

$n \times n$ image, $f \times f$ filter, padding p, stride s:

$$n \times n \ * \ f \times f \to \lfloor \frac{n + 2p - f}{s} + 1 \rfloor \times \lfloor \frac{n + 2p - f}{s} + 1 \rfloor$$

Thenical note on cross-correlation vs. convolution



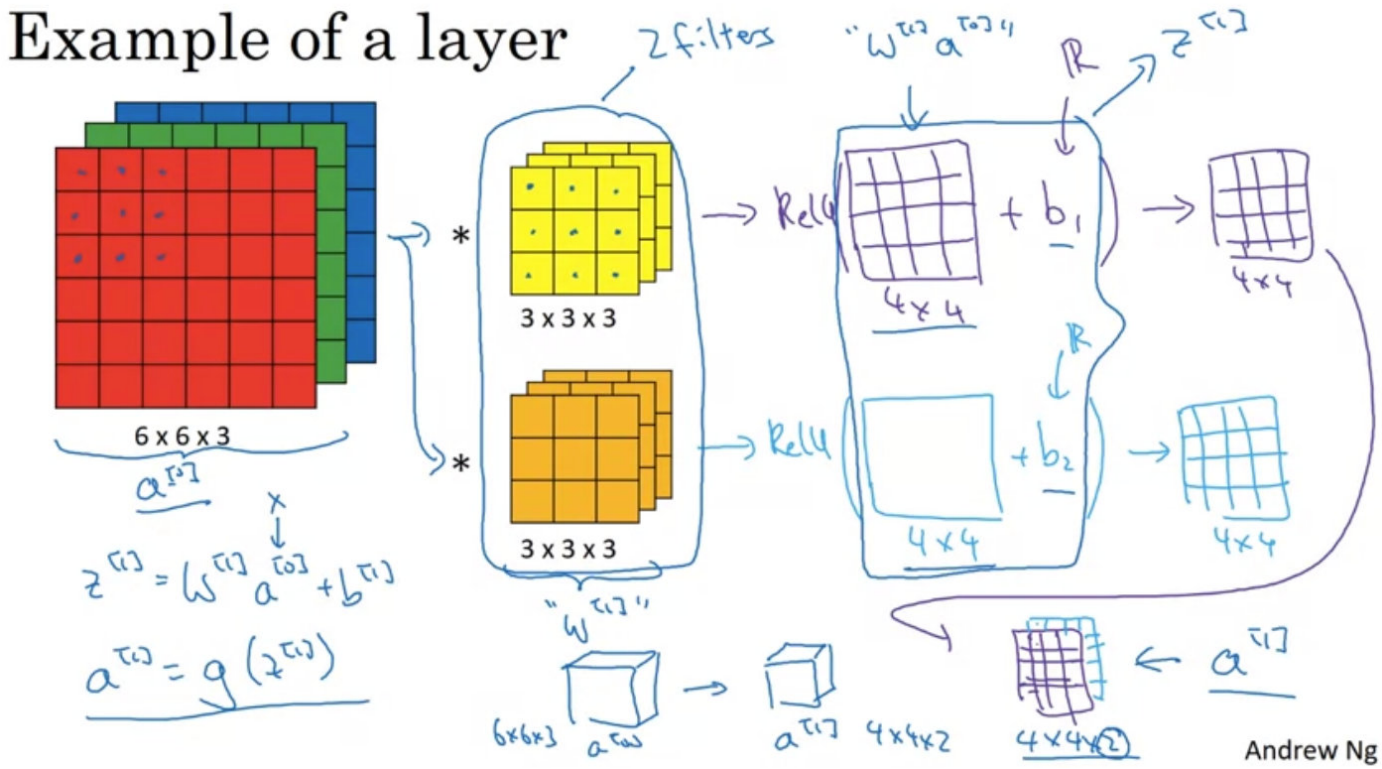
Convolutions over volumes

The number of channels of the image is equal to the number of channels of filter. If we use multiple filters, we can stack the outputs together to get the final output.

Summary (stride = 1)

$$n \times n \times n_c \ * \ f \times f \times n_c \rightarrow n - f + 1 \times n - f + 1 \times n_c'(\#filters)$$

One Layer of a Convolutional Network



Summary of notation

If layer l is a convolution layer:

$f^{[l]} = \text{filter size}$

$p^{[l]} = \text{padding}$

$s^{[l]} = \text{stride}$

$n_C^{[l]} = \text{number of filters}$

Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$

Output: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

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

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

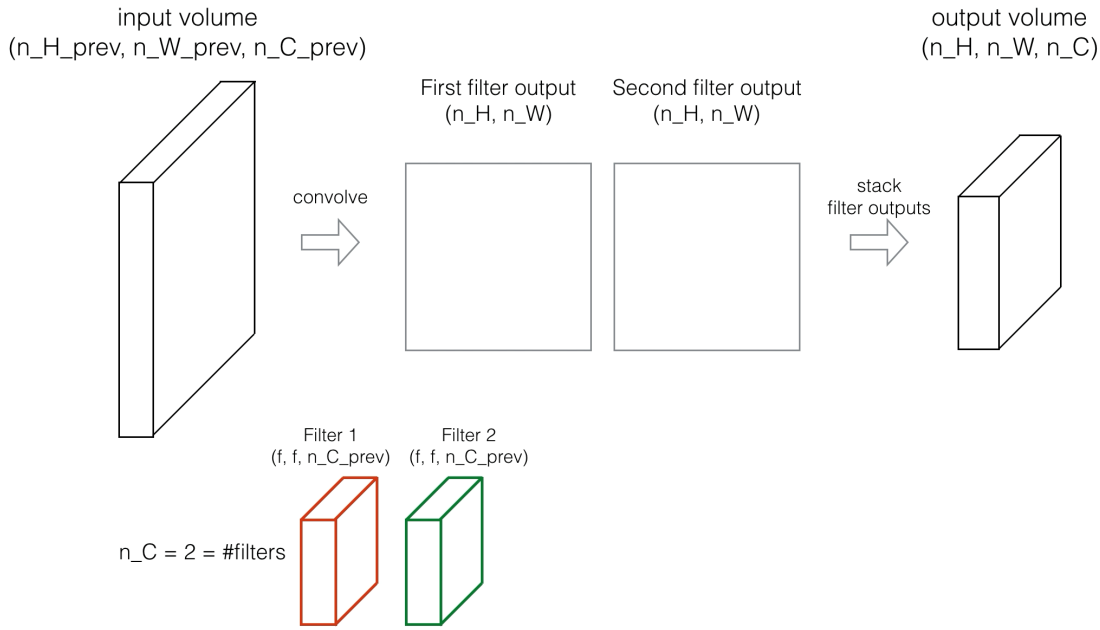
Each filter is: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$

Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$, $A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

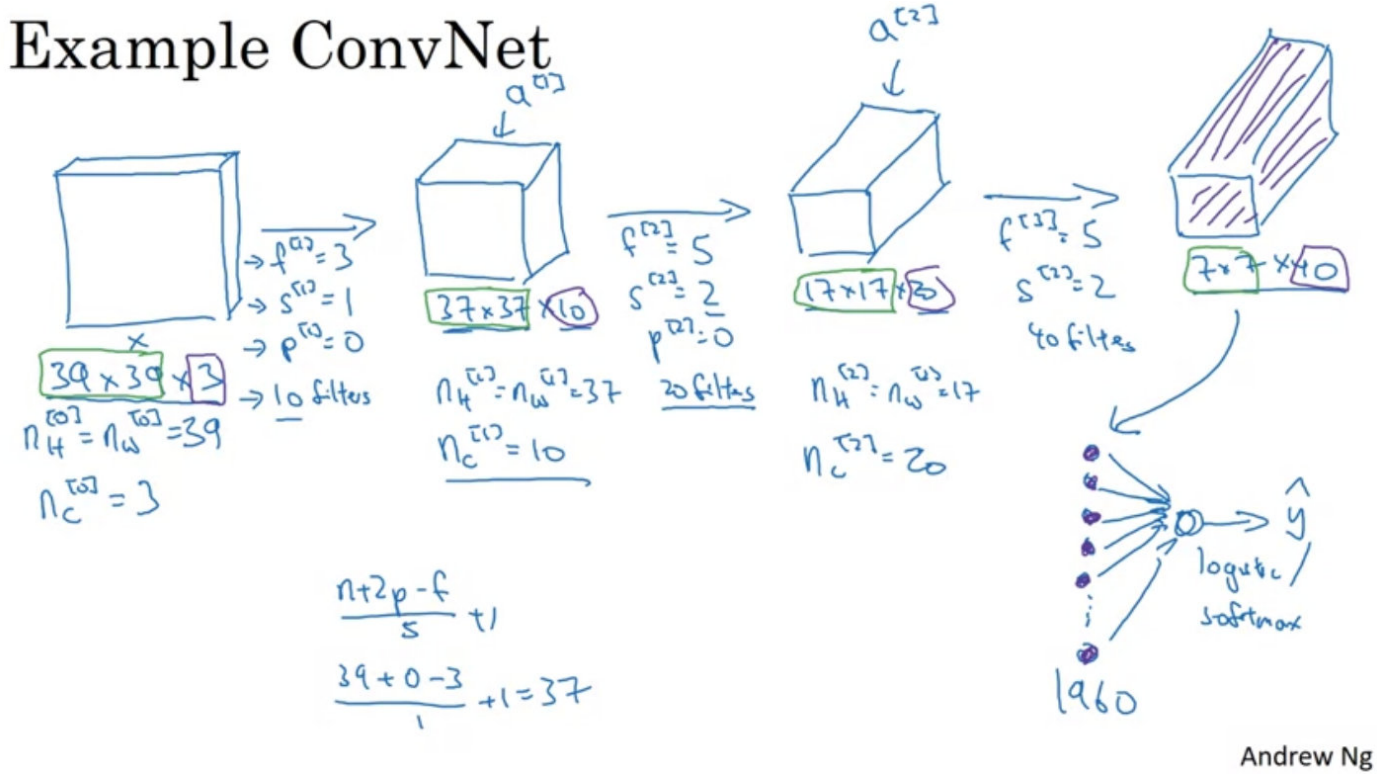
Weights: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$

Bias: $n_C^{[l]} \ - \ (1,1,1,n_C^{[l]})$

How do convolutions work?



Simple ConvNet Example

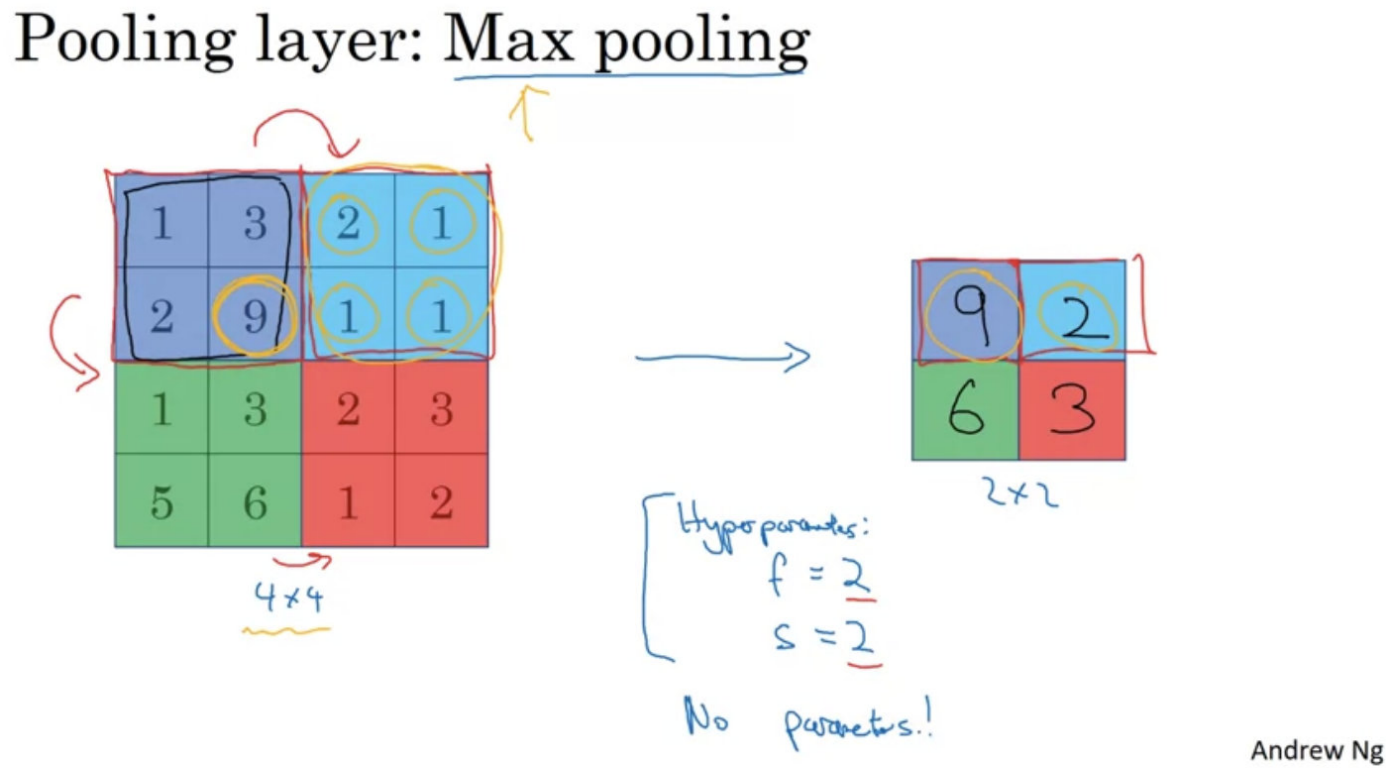


Types of layer in a convolutional networks:

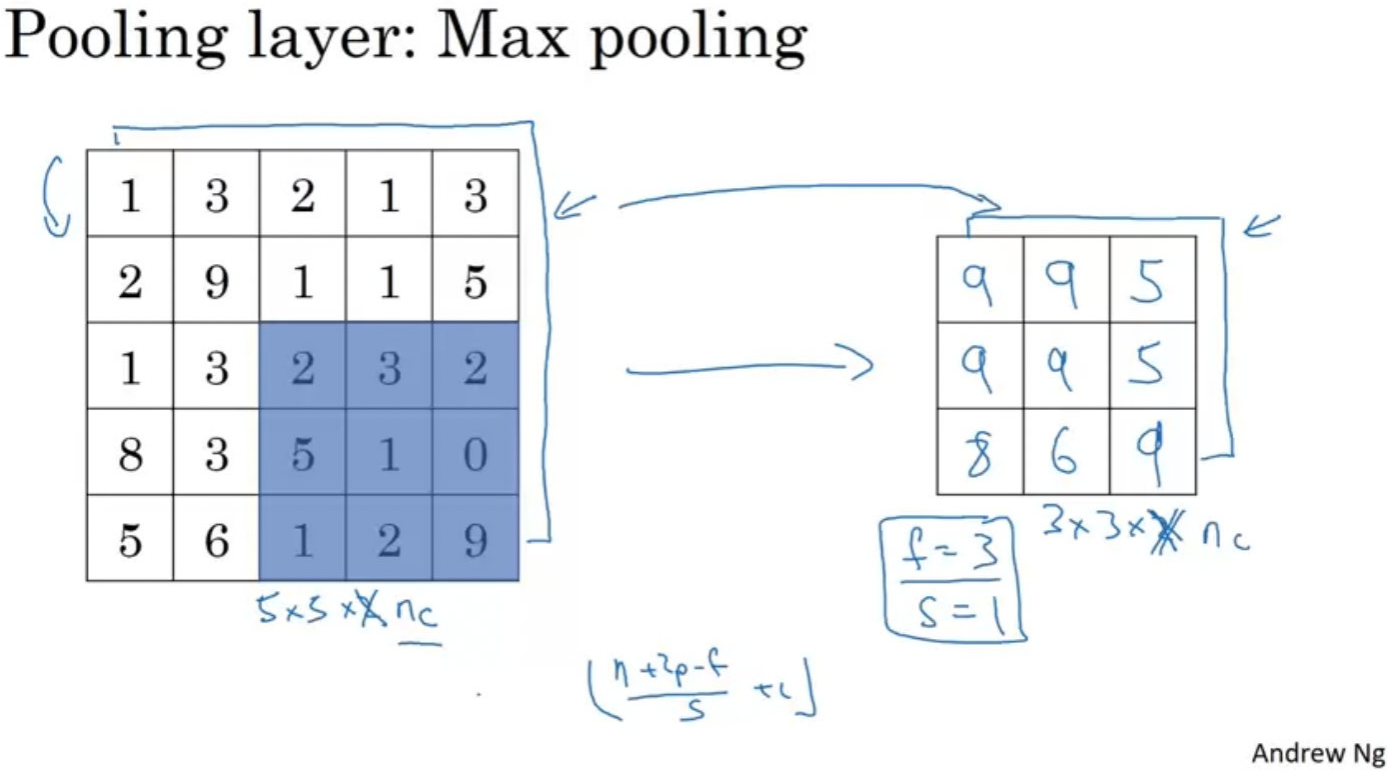
- Convolution (CONV)
- Pooling (POOL)
- Fully connected (FC)

Pooling Layers

Max pooling



Andrew Ng



Andrew Ng

The max pooling computation is done `independently` on each of n_C channels.

Average pooling (not often used)

Insted of finding maximum, find the average of the "block".

In **very deep neural networks**, you mighe use average pooling to collapse the representations.

Summary of pooling

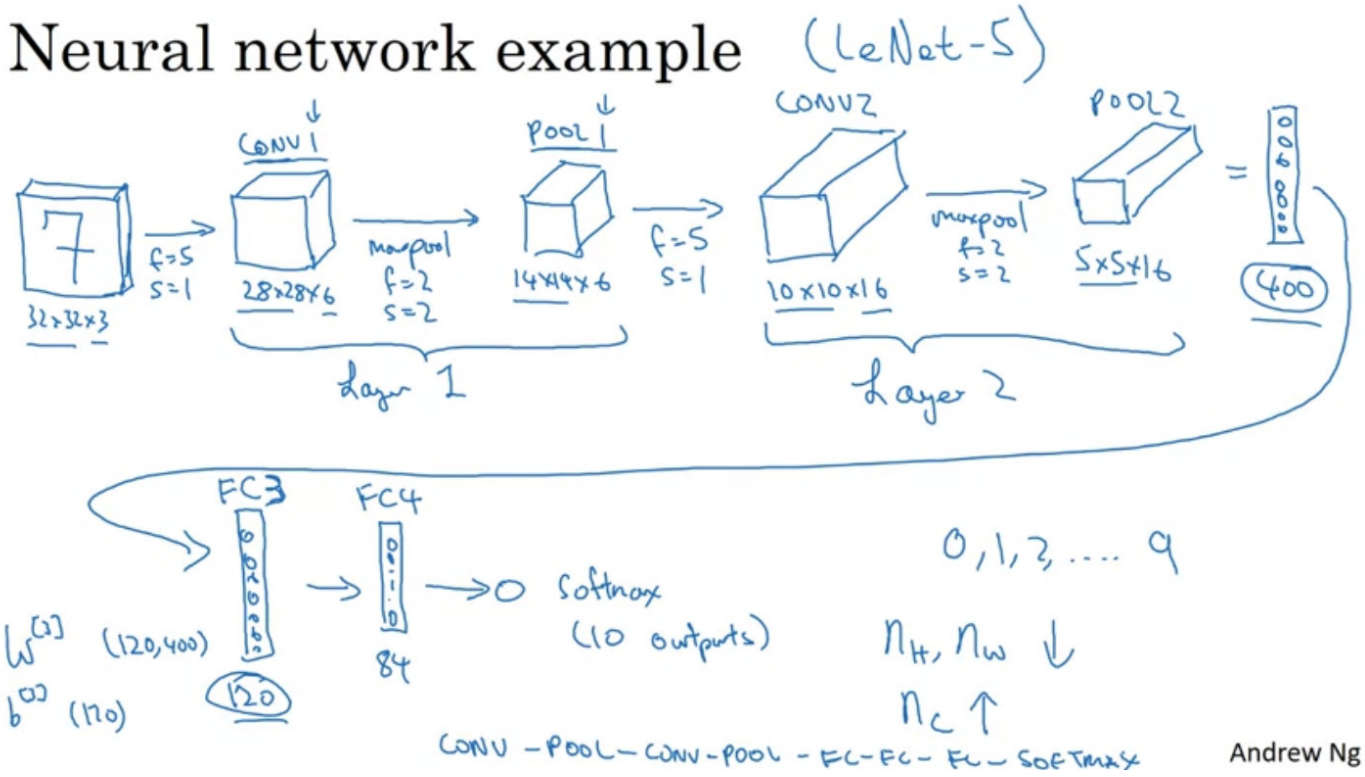
$$n_H \times n_W \times n_C \rightarrow \lfloor \frac{n_W^{[l-1]} - f^{[l]}}{s^{[l]}} + 1 \rfloor \times \lfloor \frac{n_W^{[l-1]} - f^{[l]}}{s^{[l]}} + 1 \rfloor \times n_C$$

Hyperparameters: normally $f = 2, s = 2$ (this will half the height and width)

There is `no parameters to learn!`

In fact, pooling layers modify the input by choosing one value out of several values in their input volume. Also, to compute derivatives for the layers that have parameters (Convolutions, Fully-Connected), we **still need to backpropagate the gradient through the Pooling layers**.

CNN Examples (Similar to LeNet-5)



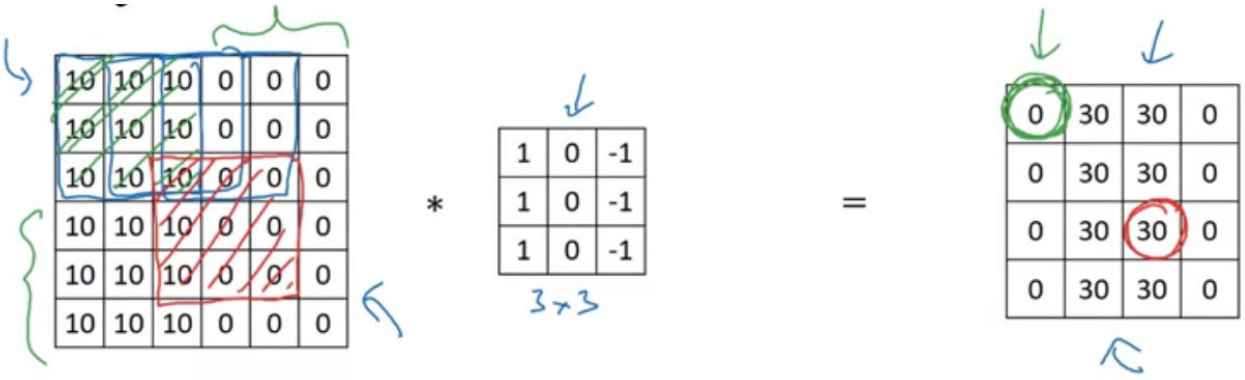
Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Why Convolutions?

The reasons why ConvNet needs much less parameter):

- **Parameter sharing**: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probaby useful in another part of the image.
- **Sparsity of connections**: In each layer, each output value depends only on a small number of inputs.



Through these two mechanisms, a neural network has a lot fewer parameters which allows it to be trained with smaller training set and is less prone to be overfitting.

What you should remember:

- A convolution extracts features from an input image by taking the dot product between the input data and a 2D array of weights (the filter).
- The 2D output of the convolution is called the feature map
- A convolution layer is where the filter slides over the image and computes the dot product
  - This transforms the input volume into an output volume of different size
- Zero padding helps keep more information at the image borders, and is helpful for building deeper networks, because you can build a CONV layer without shrinking the height and width of the volumes

- Pooling layers gradually reduce the height and width of the input by sliding a 2D window over each specified region, then summarizing the features in that region

Terminology

Window, kernel, filter, pool

The words `kernel` and `filter` are used to refer to the same thing. The word `filter` accounts for the amount of `kernels` that will be used in a single convolution layer. `Pool` is the name of the operation that takes the max or average value of the kernels.

Convolutional Layer Backward Pass

Computing dA:

This is the formula for computing dA with respect to the cost for a certain filter W_c and a given training example:

$$dA \mathrel{+}= \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw} \tag{1}$$

Where W_c is a filter and dZ_{hw} is a scalar corresponding to the gradient of the cost with respect to the output of the conv layer Z at the hth row and wth column (corresponding to the dot product taken at the ith stride left and jth stride down). Note that at each time, you multiply the the same filter W_c by a different dZ when updating dA. We do so mainly because when computing the forward propagation, each filter is dotted and summed by a different a_slice. Therefore when computing the backprop for dA, you are just adding the gradients of all the a_slices.

Computing dW:

This is the formula for computing dW_c (dW_c is the derivative of one filter) with respect to the loss:

$$dW_c \mathrel{+}= \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw} \tag{2}$$

Where a_{slice} corresponds to the slice which was used to generate the activation Z_{ij}. Hence, this ends up giving us the gradient for W with respect to that slice. Since it is the same W, we will just add up all such gradients to get dW.

Computing db:

This is the formula for computing db with respect to the cost for a certain filter W_c:

$$db = \sum_{h} \sum_{w} dZ_{hw} \tag{3}$$

As you have previously seen in basic neural networks, db is computed by summing dZ. In this case, you are just summing over all the gradients of the conv output (Z) with respect to the cost.

Pooling Layer - Backward Pass

Next, let's implement the backward pass for the pooling layer, starting with the MAX-POOL layer. Even though a pooling layer has no parameters for backprop to update, you still need to backpropagate the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

Max Pooling - Backward Pass

Before jumping into the backpropagation of the pooling layer, you are going to build a helper function called `create_mask_from_window()` which does the following:

$$X = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \quad \rightarrow \quad M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{4}$$

As you can see, this function creates a "mask" matrix which keeps track of where the maximum of the matrix is. True (1) indicates the position of the maximum in X, the other entries are False (0). You'll see later that the backward pass for average pooling is similar to this, but uses a different mask.

Why keep track of the position of the max? It's because this is the input value that ultimately influenced the output, and therefore the cost. Backprop is computing gradients with respect to the cost, so anything that influences the ultimate cost should have a non-zero gradient. So, backprop will "propagate" the gradient back to this particular input value that had influenced the cost.

Average Pooling - Backward Pass

In max pooling, for each input window, all the "influence" on the output came from a single input value--the max. In average pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.

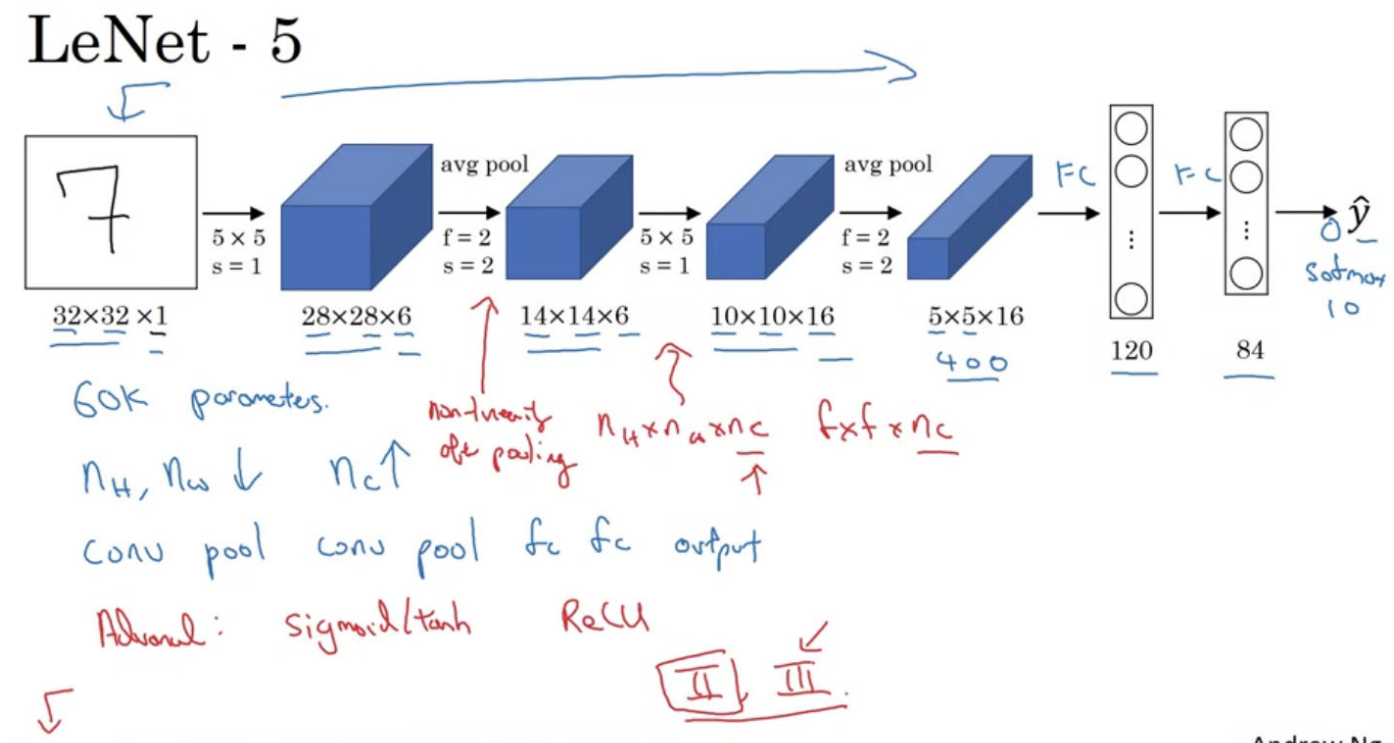
For example if we did average pooling in the forward pass using a 2x2 filter, then the mask you'll use for the backward pass will look like:

$$dZ = 1 \quad \rightarrow \quad dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix} \tag{5}$$

This implies that each position in the dZ matrix contributes equally to output because in the forward pass, we took an average.

Case studies

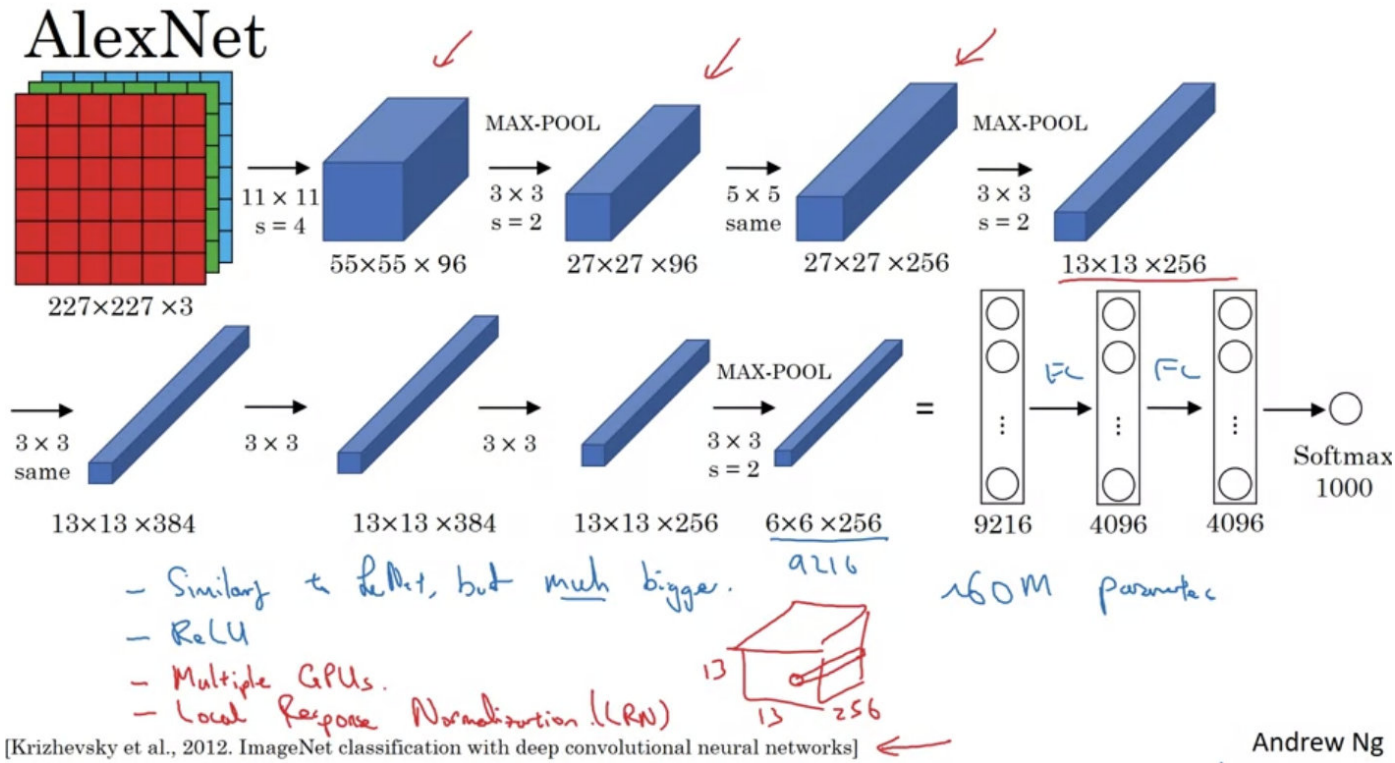
LeNet - 5



[LeCun et al., 1998. Gradient-based learning applied to document recognition]

Andrew Ng

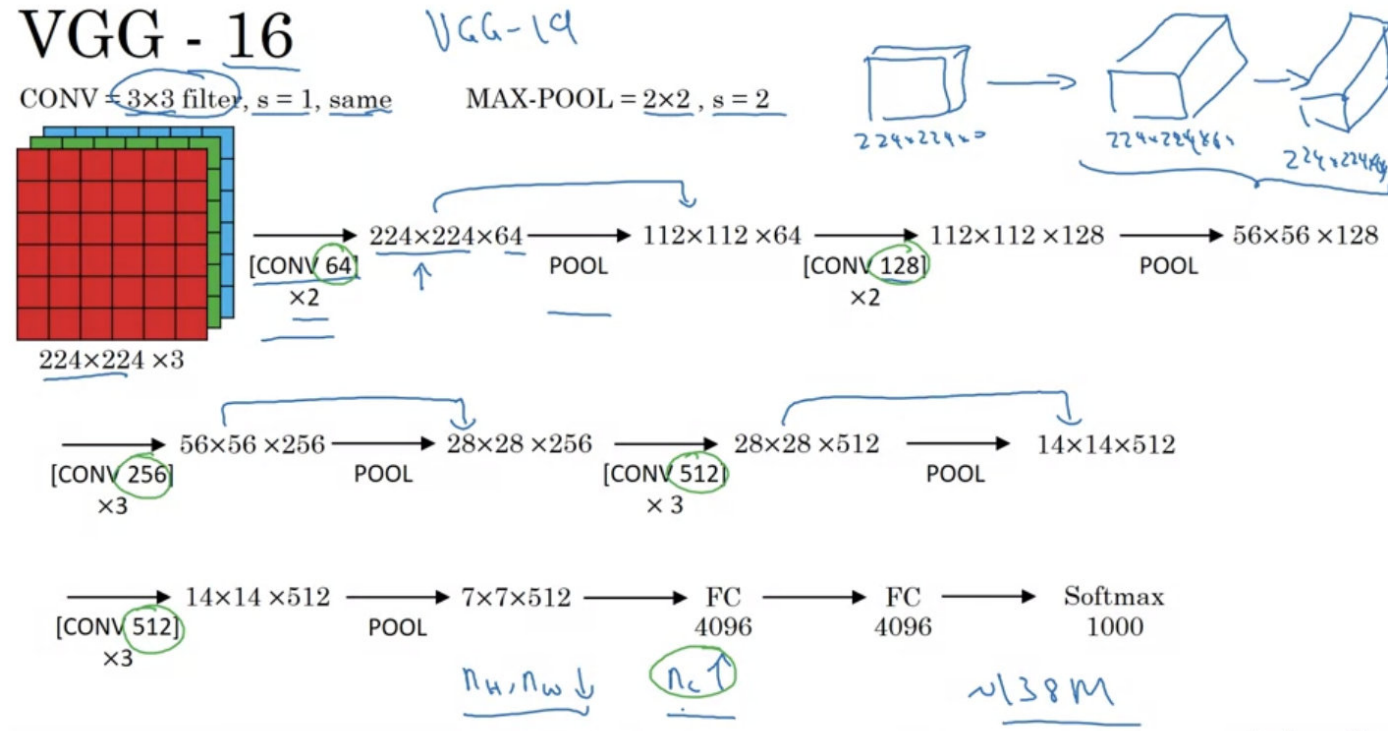
AlexNet



[Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]

Andrew Ng

VGG - 16 (16 refers to there are 16 layers with weights)



[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]

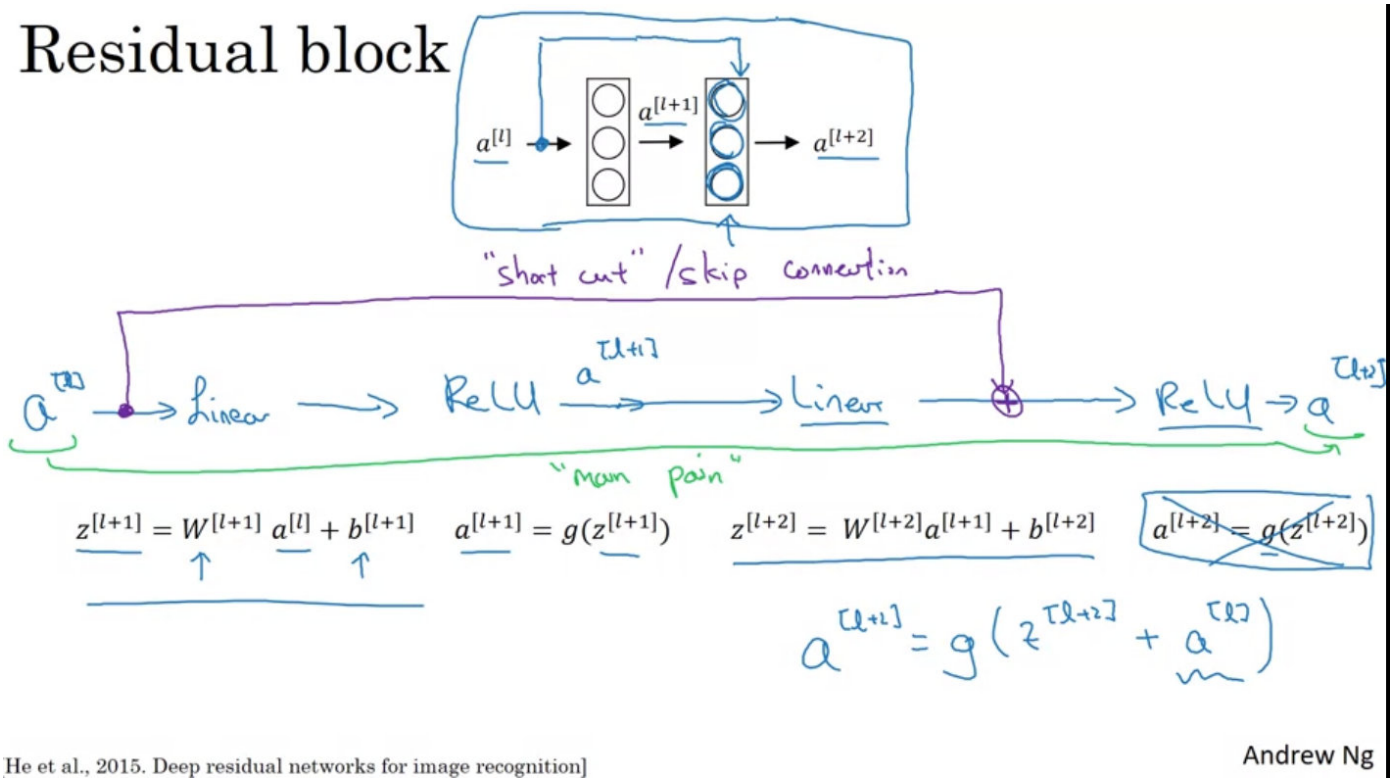
Andrew Ng

Residual Networks (ResNets)

Very deep neural networks are difficult to train beacause of **vanishing** and **exploding** gradient types of problems. The **skip connections** allows you to take activation from one layer and suddenly feed it to another layer even much deeper in the neural network. Using that, you will bulid **ResNet** whihc enables you to train very deep networks.

Residual block (Identity block)

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



He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

The `identity block` is the standard block used in ResNets, and corresponds to the case where the input activation (say $a^{[l]}$) has the **same dimension** as the output activation (say $a^{[l+2]}$).

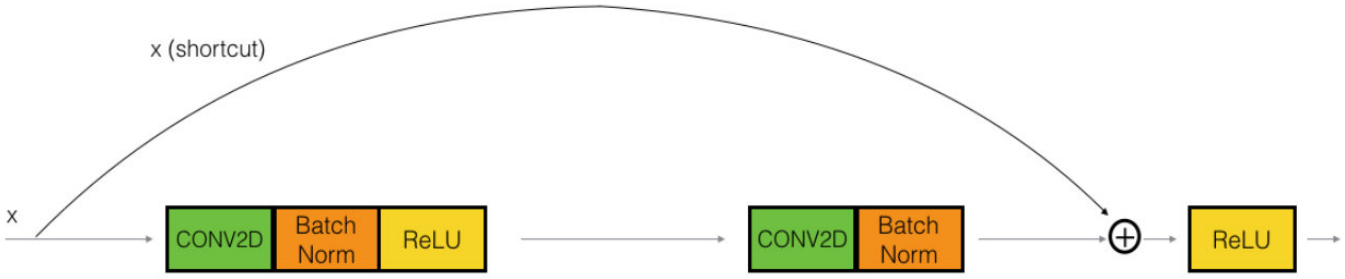


Figure 3 : Identity block. Skip connection "skips over" 2 layers.

The ResNet `"convolutional block"` is the second block type. You can use this type of block when the input and output dimensions **don't match up**. The difference with the identity block is that there is a CONV2D layer in the shortcut path:

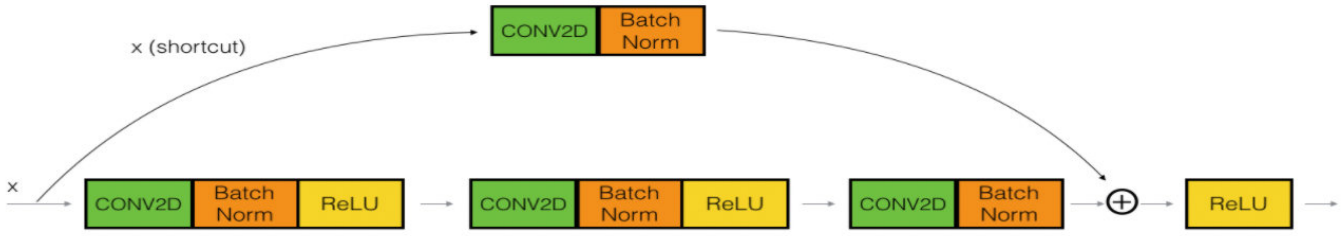
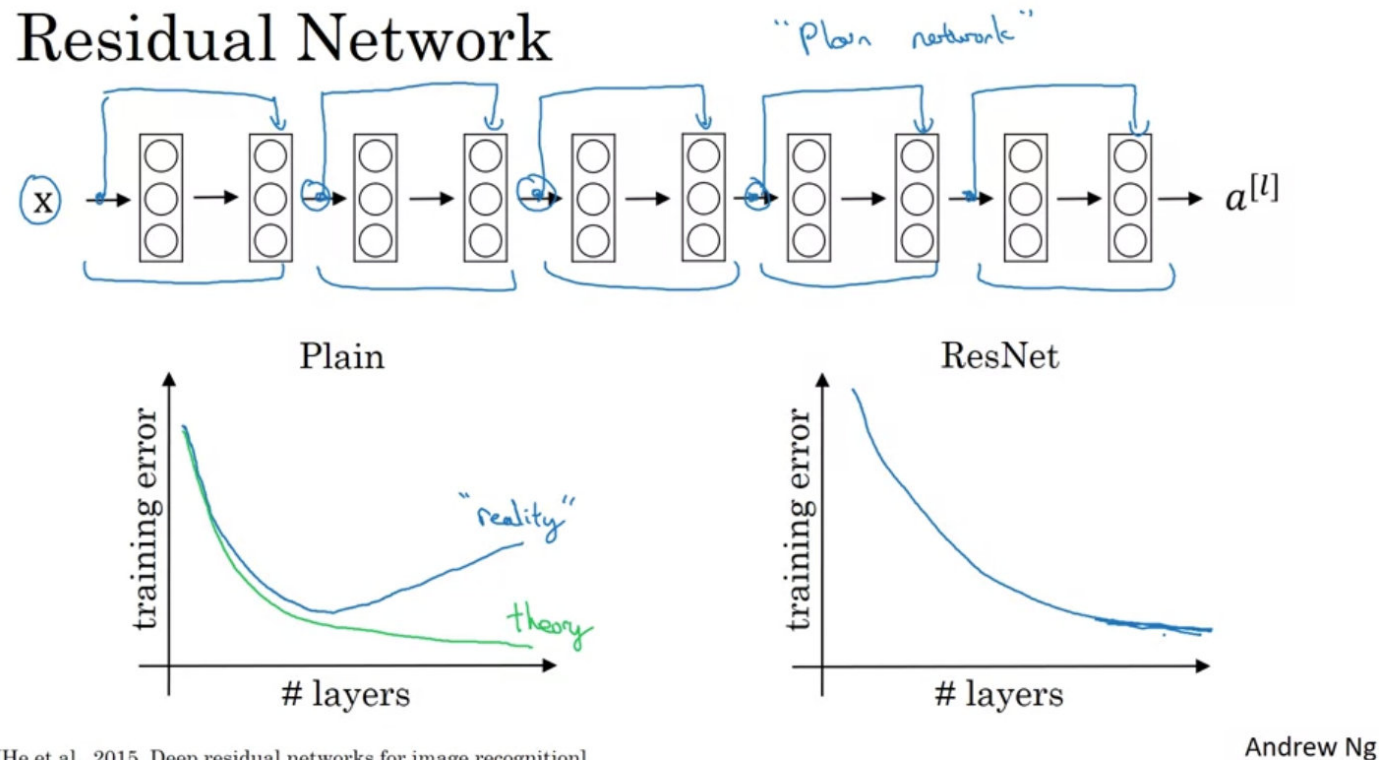


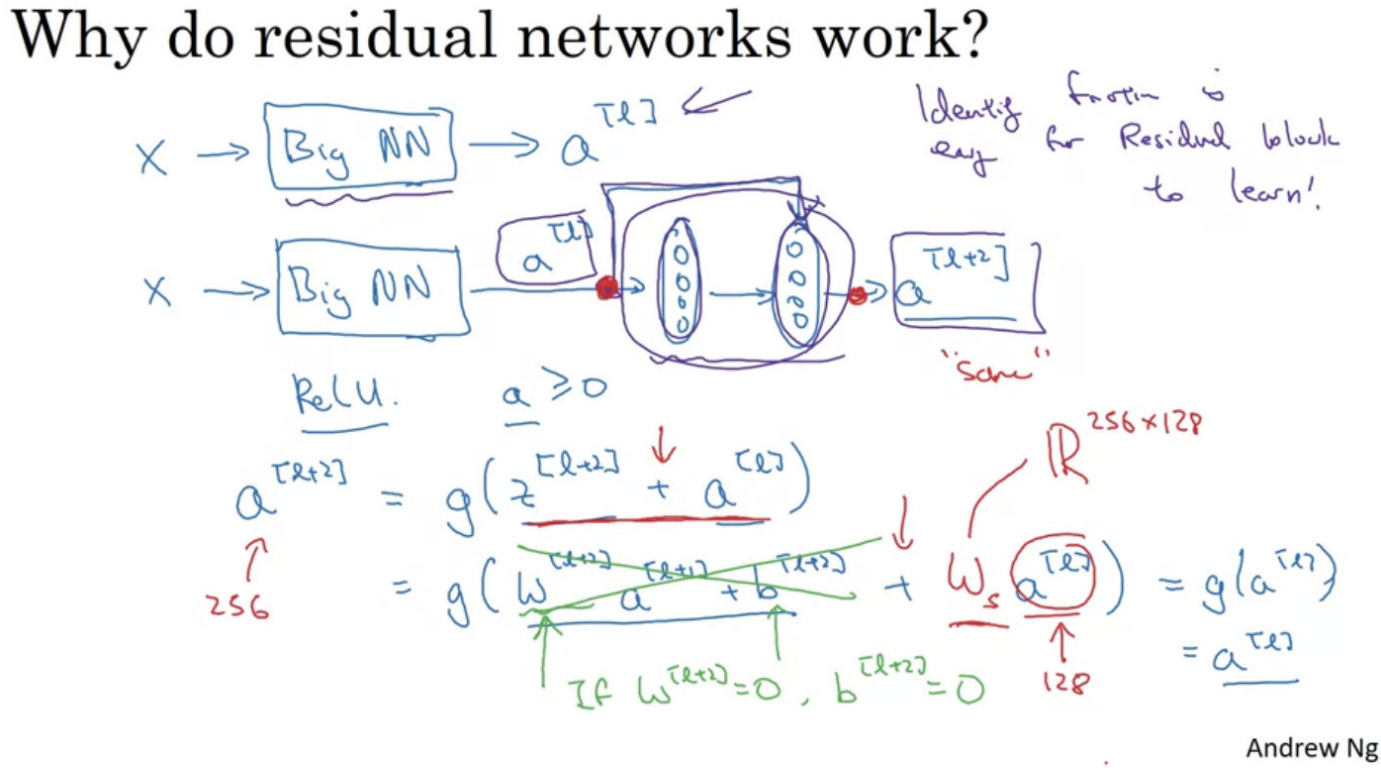
Figure 4 : Convolutional block

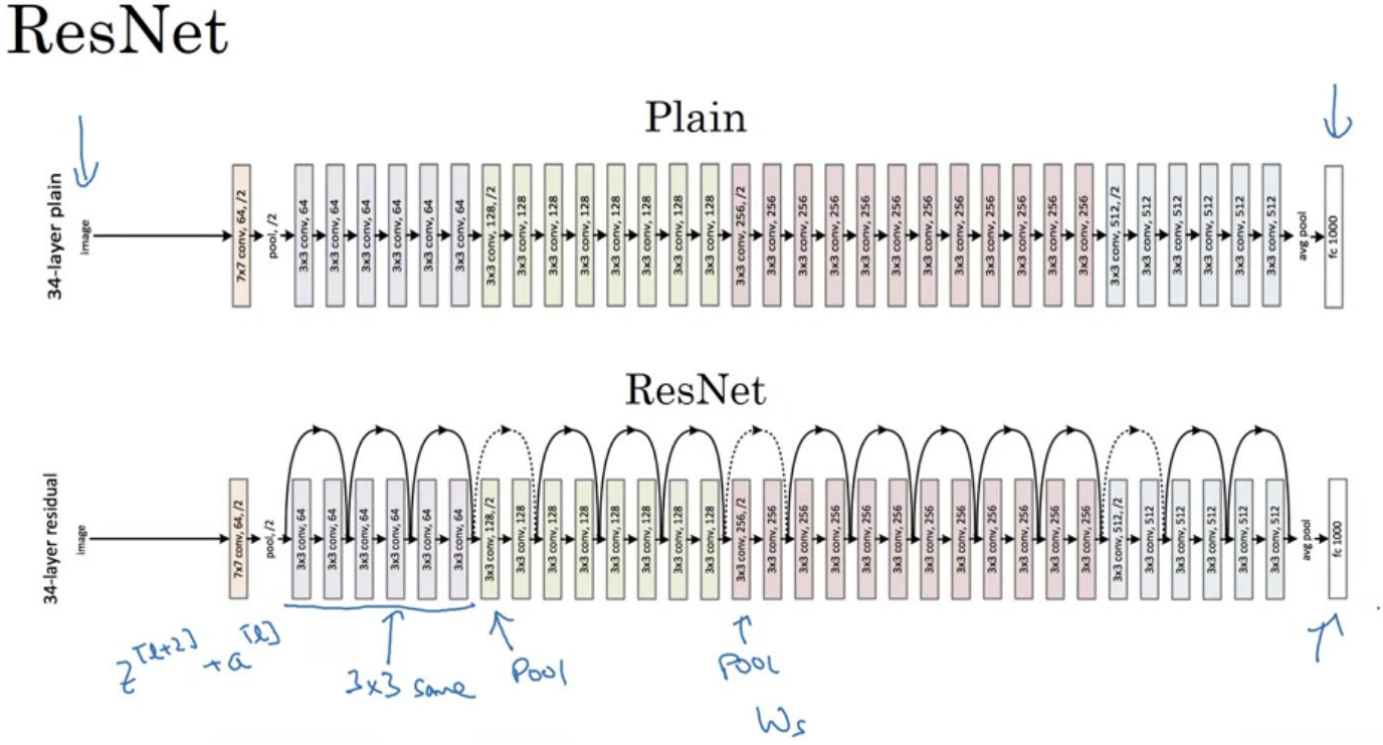
In theory, having a deeper network should only help. But in practice, having a plain network (no ResNet) is very deep means that your optimization algorithm just has a much harder time training. And so, in reality, your training error gets **worse** if you pick a network that is **too deep.**



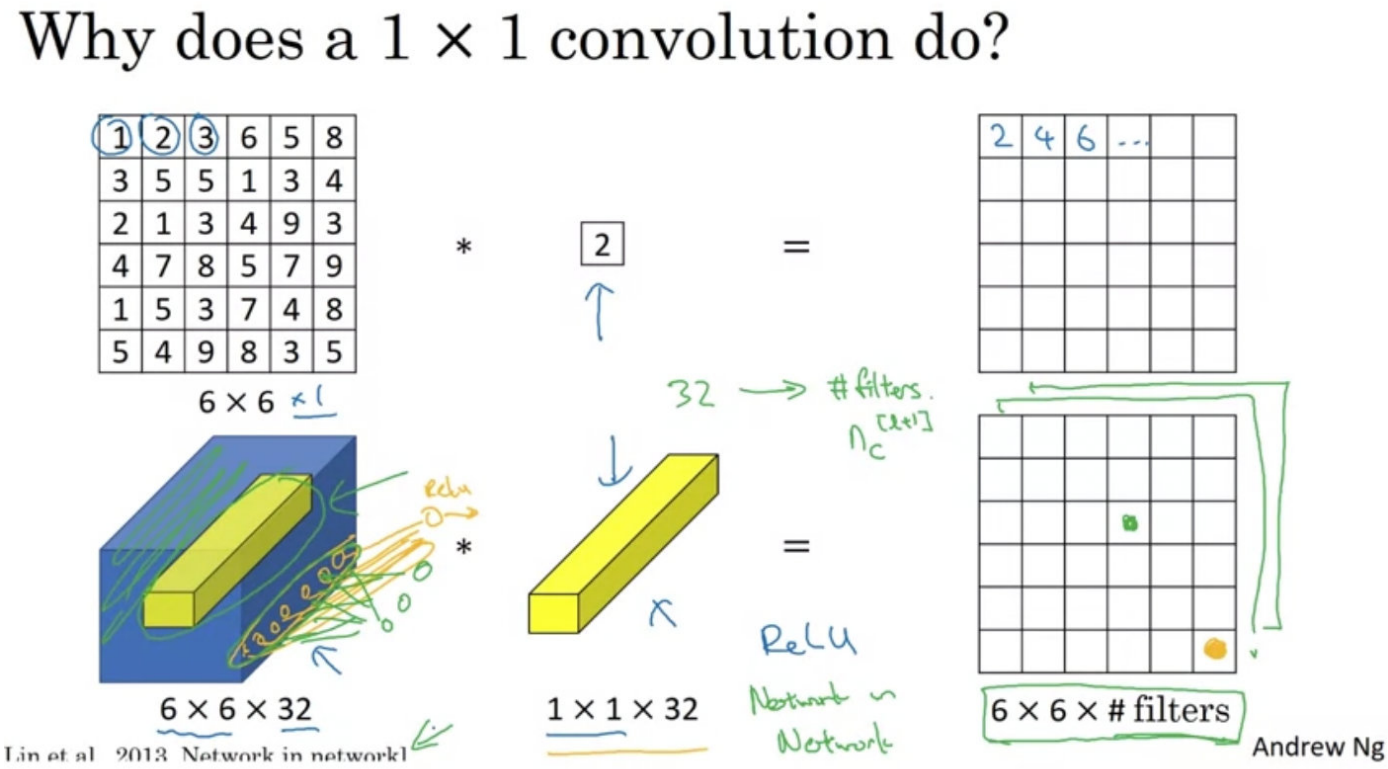
Why ResNet works?

The main reason is that it is so easy for these extra layers to learn the **identity function** that you are kind of guranteed that **it doesn't hurt performance** and thena lot the time you may be get lucky and then even **helps performance**.



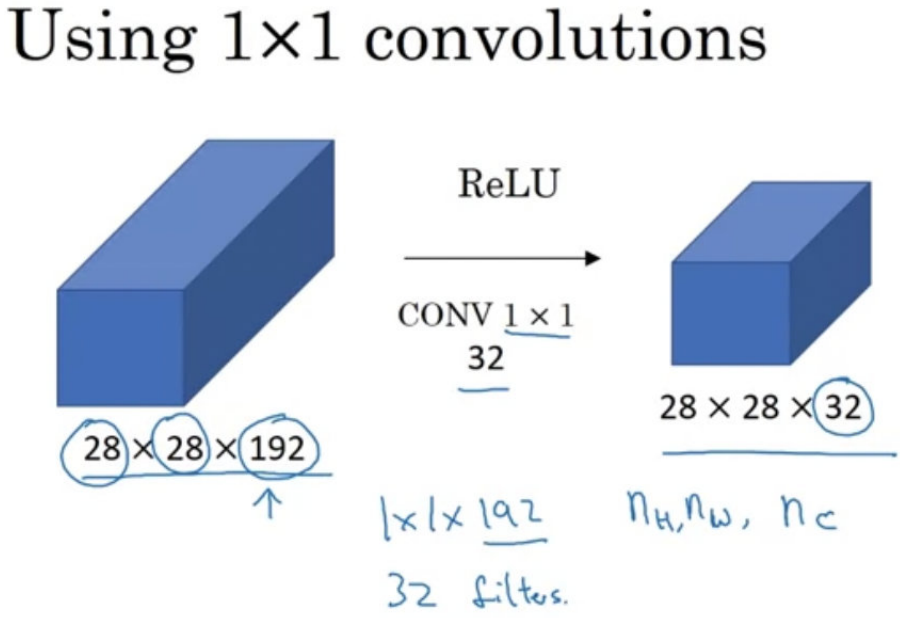


Network in Network and 1x1 convolutions



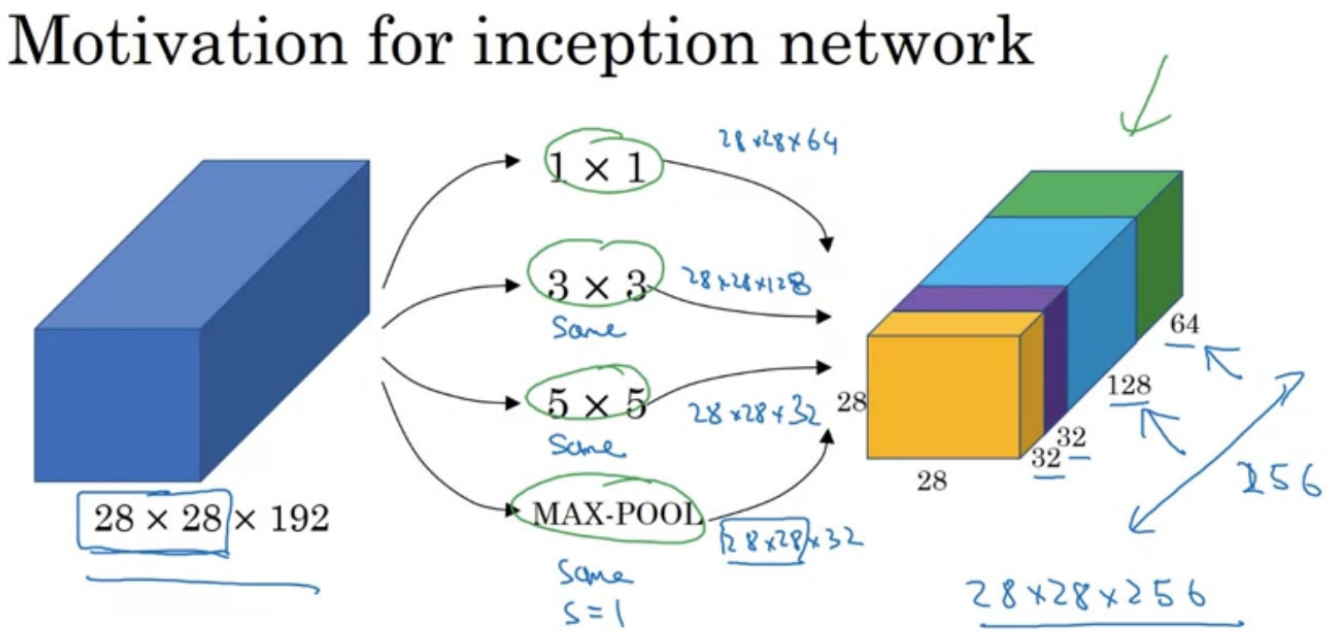
The 1 x 1convolution has nonlinearity, it allows you to learn a more complex function of your network by adding another layer.

The 1 x 1 convolution can **shrink the number of channels**.



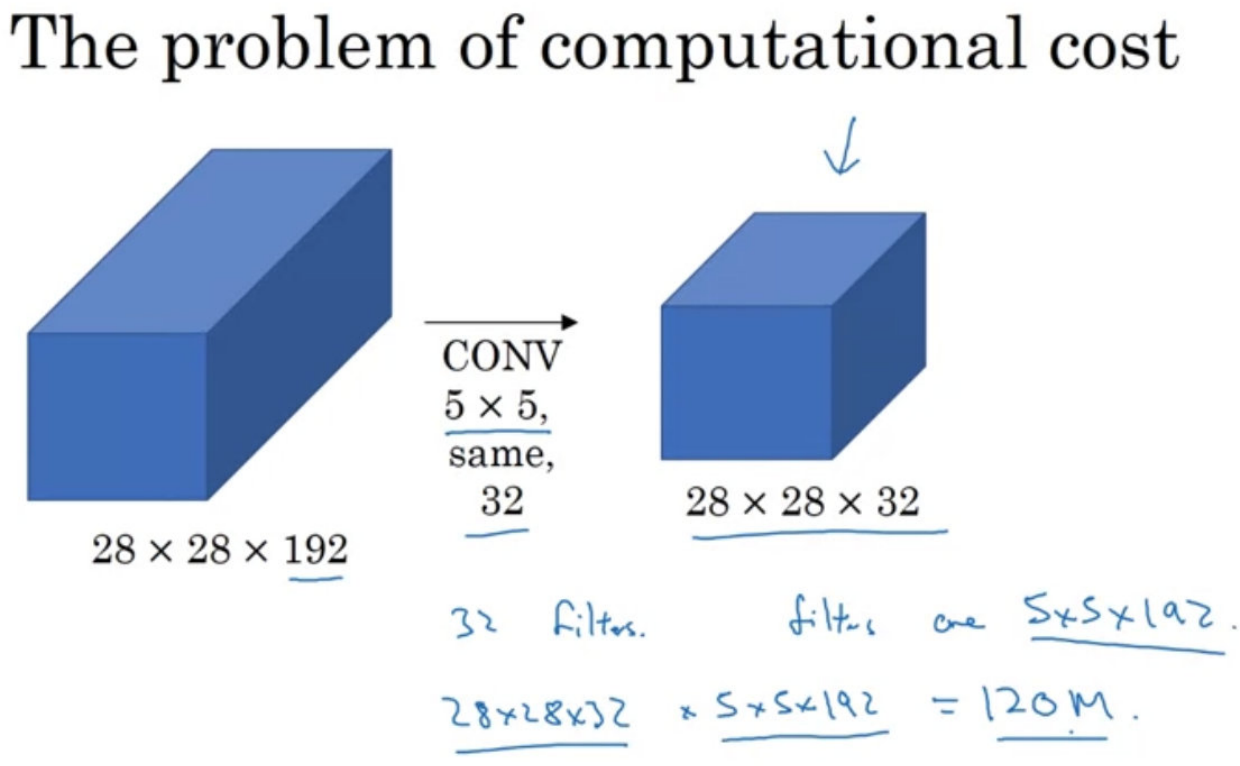
Inception Network

Motivation for inception network



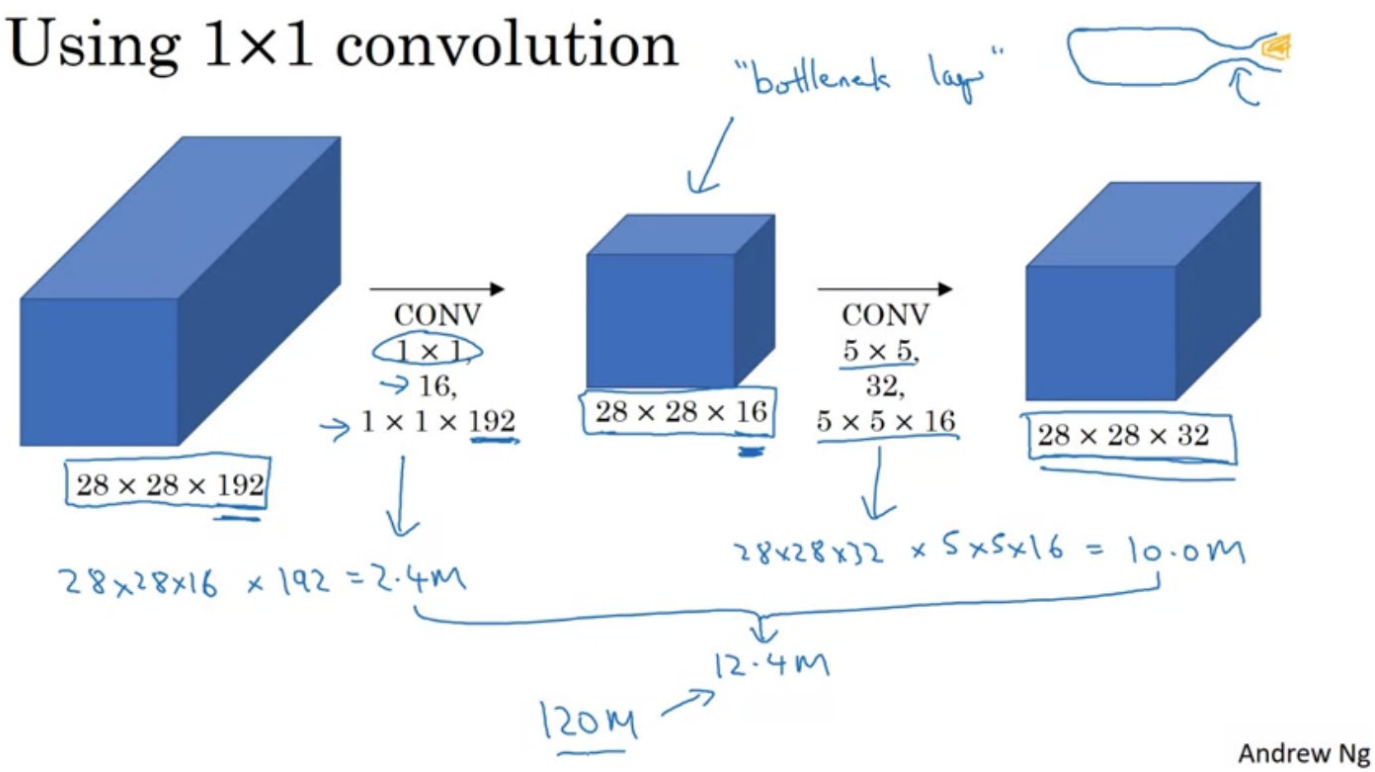
The basic idea is that instead of you needing to pick one of these filter sizes or pooling you want and committing to that, you can do them all and just concatenate all the outputs, and **let the network learn whatever parameters it wants to use, whatever the combinations of these filter sizes it wants.**

The problem of computational cost

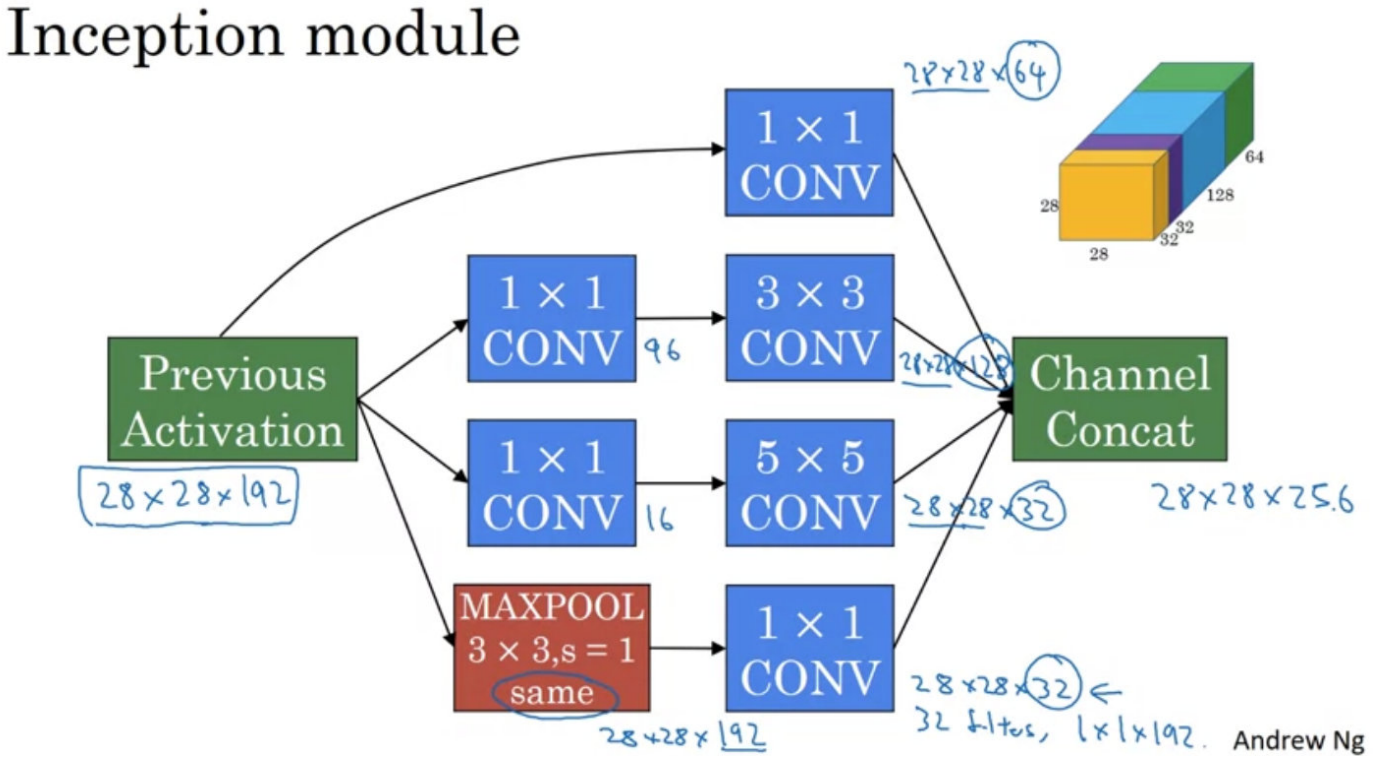


$$\#\text{multiplies} = \#\text{multiplies to compute each of output values} \times \#\text{output values}$$

Using 1 x 1 convolution



Inception module

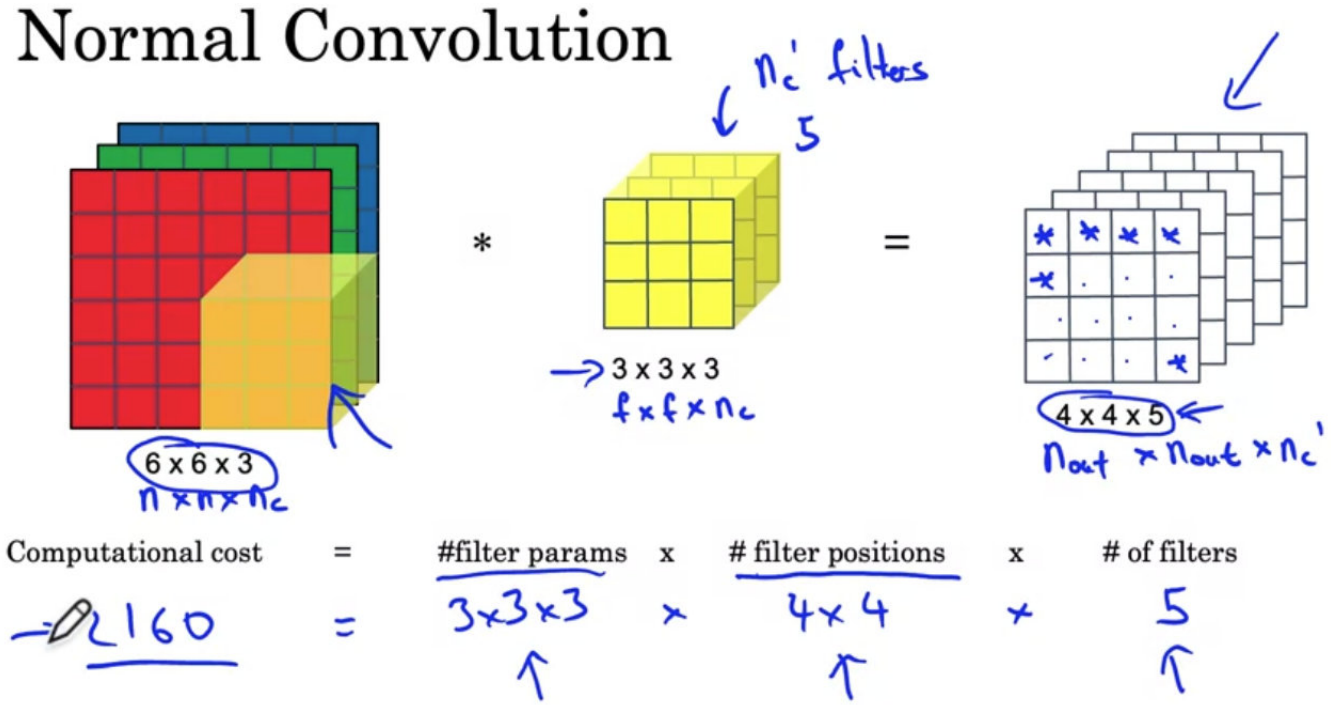


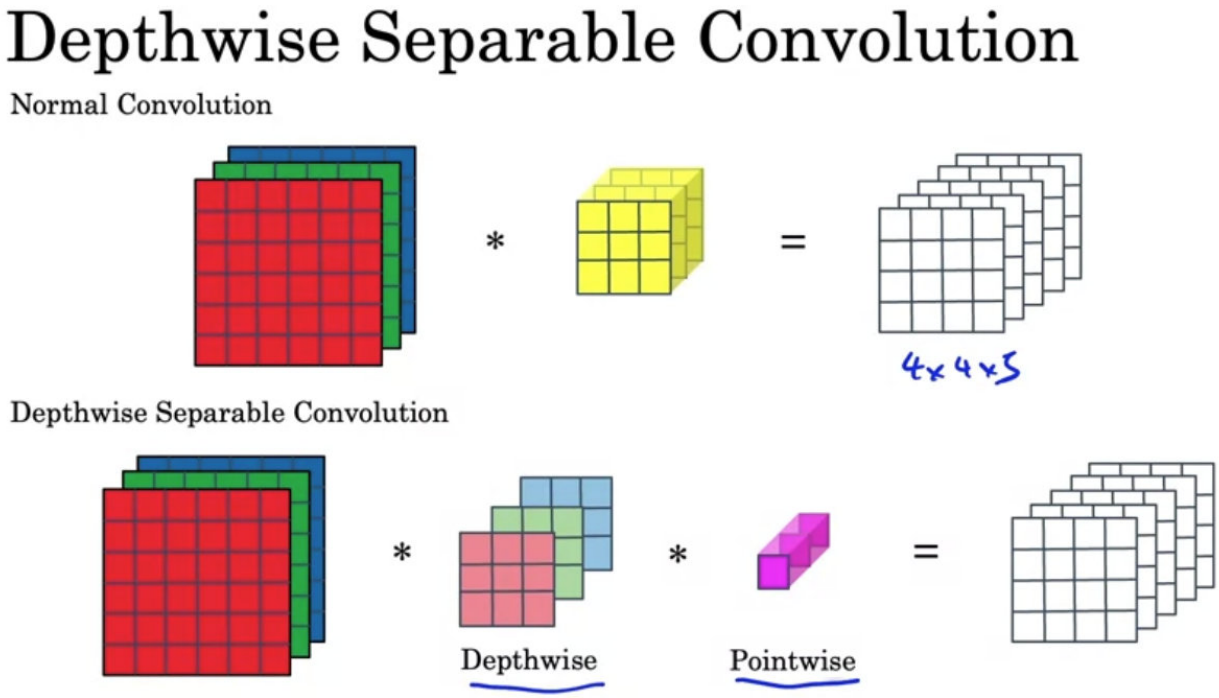
MobileNet

Motivation for MobileNets

- Low computational cost at deployment
- Useful for mobile and embedded vision applications

Key idea: Normal vs. depthwise-separable convolutions





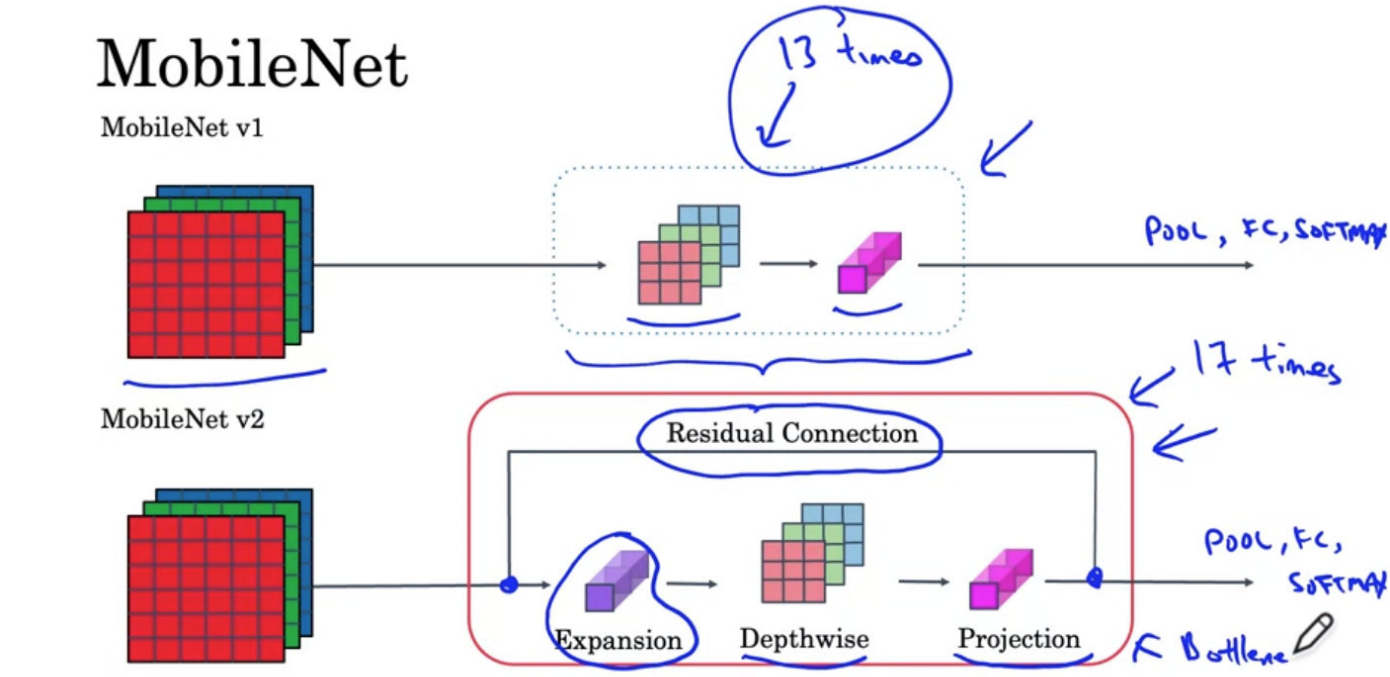
Cost of normal convolution: **2160**

Cose of depth wise separable convolution: 432 + 240 = **672**

The ratio is equal to

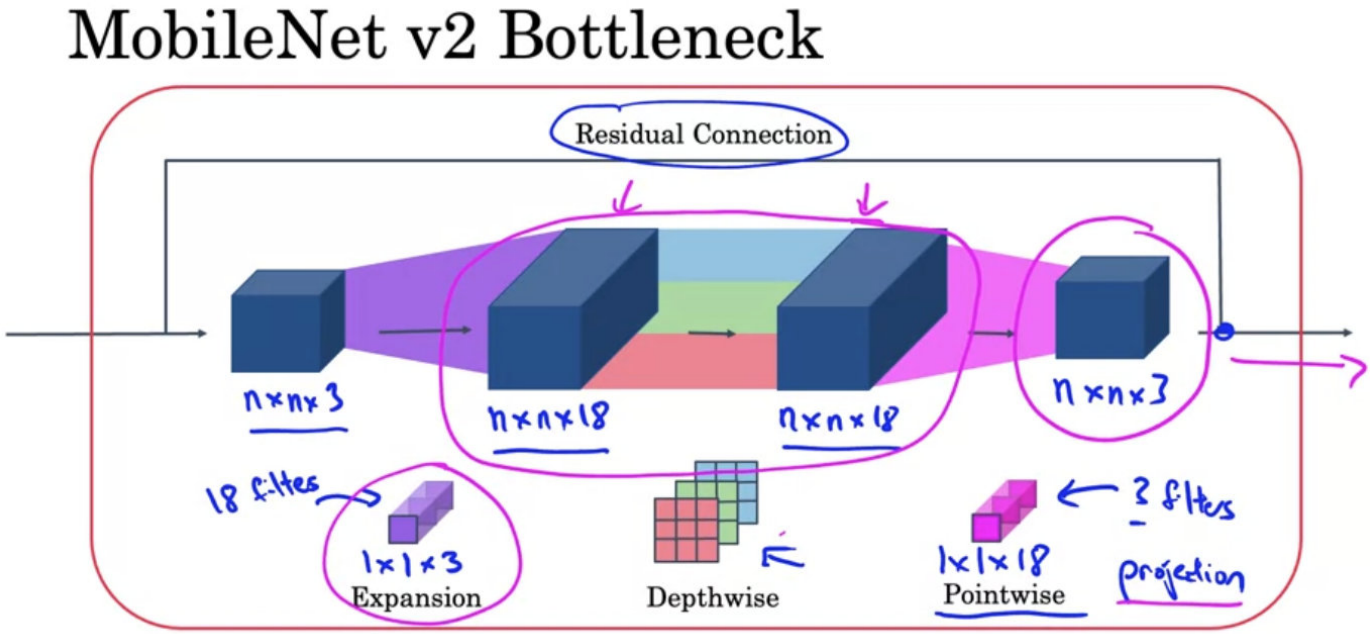
$$ratio = \frac{1}{n'_C} + \frac{1}{f^2}$$

MobileNet architecture



[Sandler et al. 2019, MobileNetV2: Inverted Residuals and Linear Bottlenecks] Andrew Ng

MobileNet v2 Bottleneck



What you should remember:

- MobileNetV2's unique features are:
  - Depthwise separable convolutions that provide lightweight feature filtering and creation

    - Input and output bottlenecks that preserve important information on either end of the block
- Depthwise separable convolutions deal with both spatial and depth (number of channels) dimensions

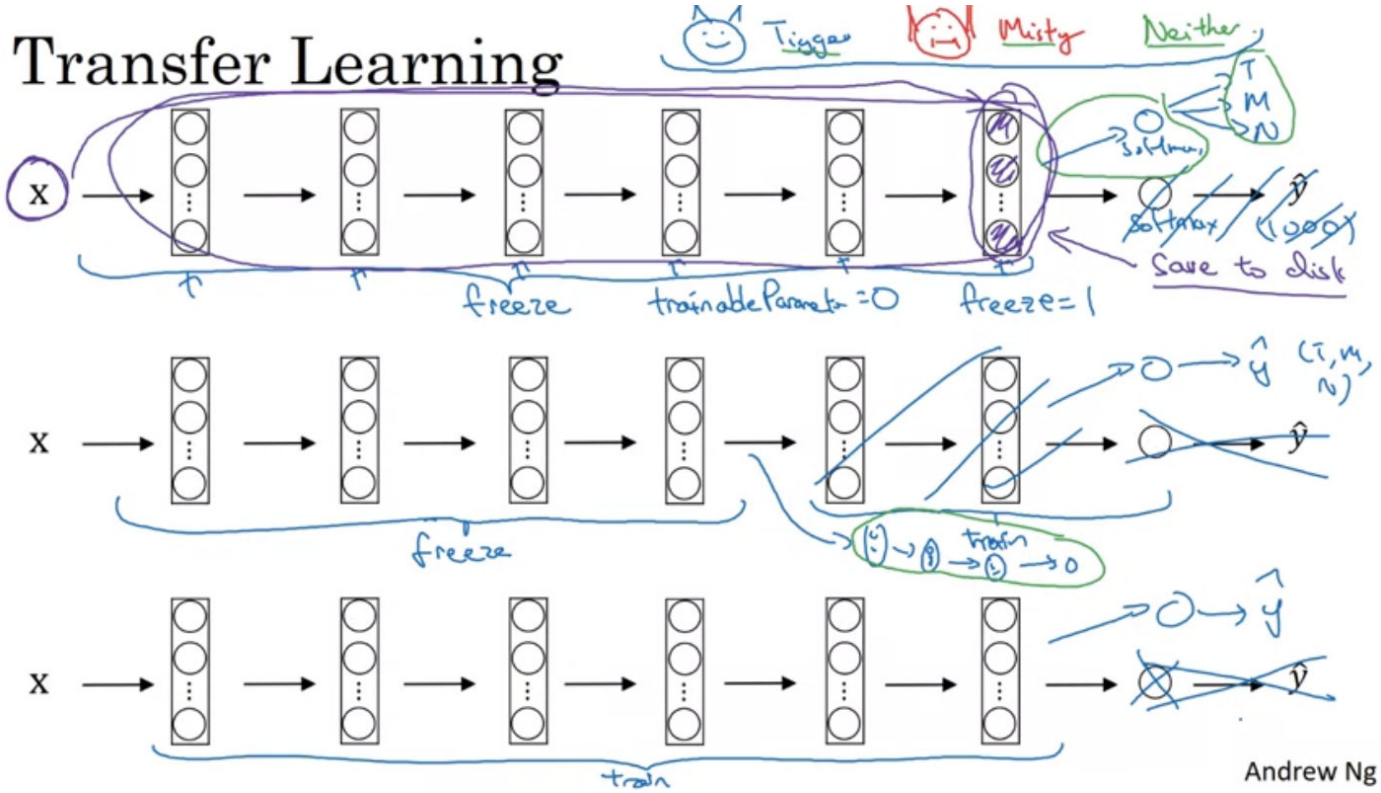
EfficientNet

Three things you can do to scale things up or down are:

- **r**: resolution (of images)
- **d**: depth (of networks)
- **w**: width (of layers)

The EfficientNet can help you find a way to scale up or down networks.

Transfer Learning



You could try fine-tuning the model by re-running the optimizer in the last layers to improve accuracy. When you use a smaller learning rate, you take smaller steps to adapt it a little more closely to the new data. In transfer learning, the way you achieve this is by unfreezing the layers at the end of the network, and then re-training your model on the final layers with **a very low learning rate**. Adapting your learning rate to go over these layers in smaller steps can yield more fine details - and higher accuracy.

What you should remember:

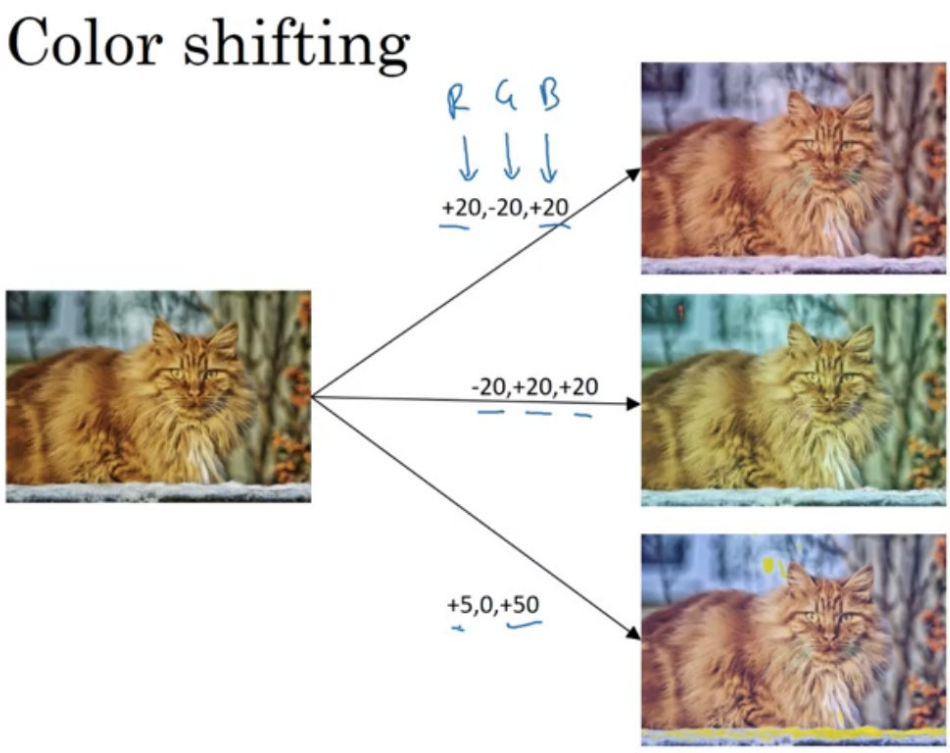
- To adapt the classifier to new data: Delete the top layer, add a new classification layer, and train only on that layer
- When freezing layers, avoid keeping track of statistics (like in the batch normalization layer)
- Fine-tune the final layers of your model to capture high-level details near the end of the network and potentially improve accuracy

Data Augmentation

Common augmentation method

- Mirroring
- Random cropping
- Rotation
- Shearing
- Local warping
- ...

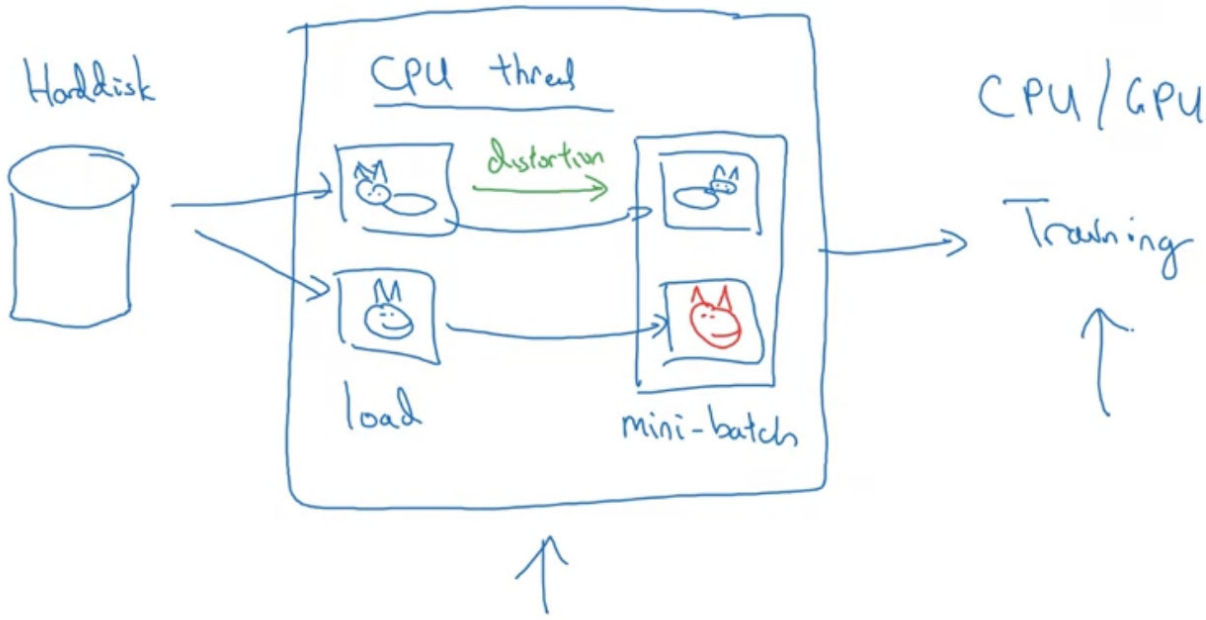
Color shifting



This make the learning algorithm more robust to changes in the colors of the images.

"PCA color augmentation" (AlexNet): keep the overall color of the tint the same.

Implementing distortions during training



Andrew Ng

Tips for doing well on benchmarks/winning competitions

Ensembling

3-15 networks → $\hat{y}$

- Train several networks independently and average their outputs

Multi-crop at test time

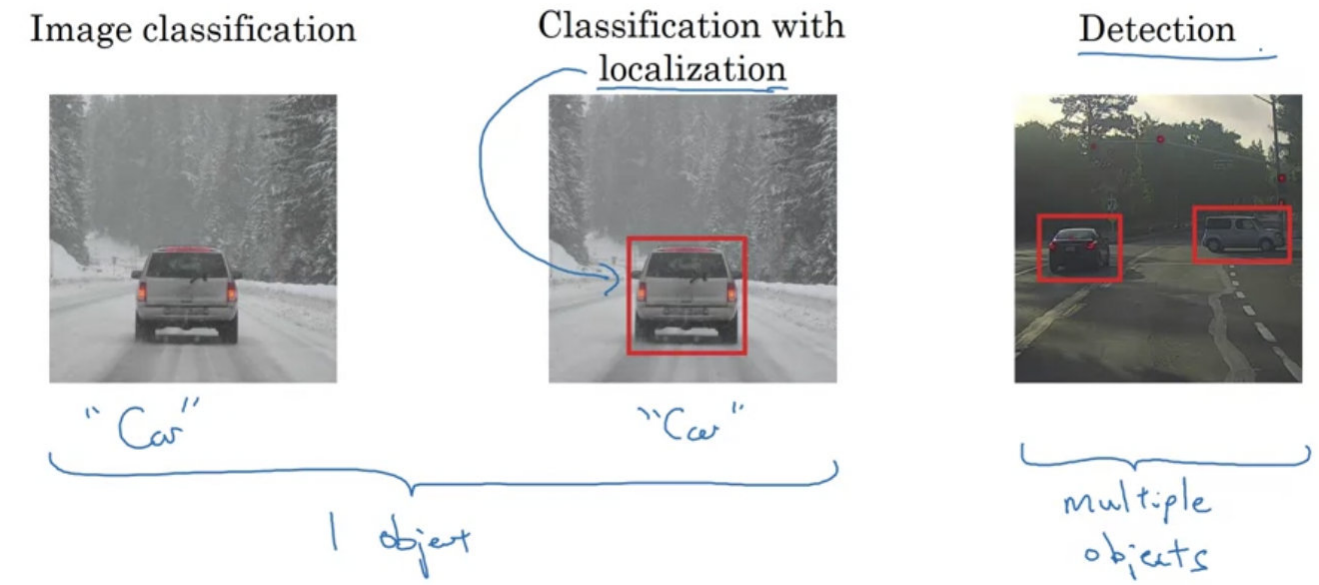
- Run classifier on multiple versions of test images and average results

10-crop

Andrew Ng

Object Detection

Object localization

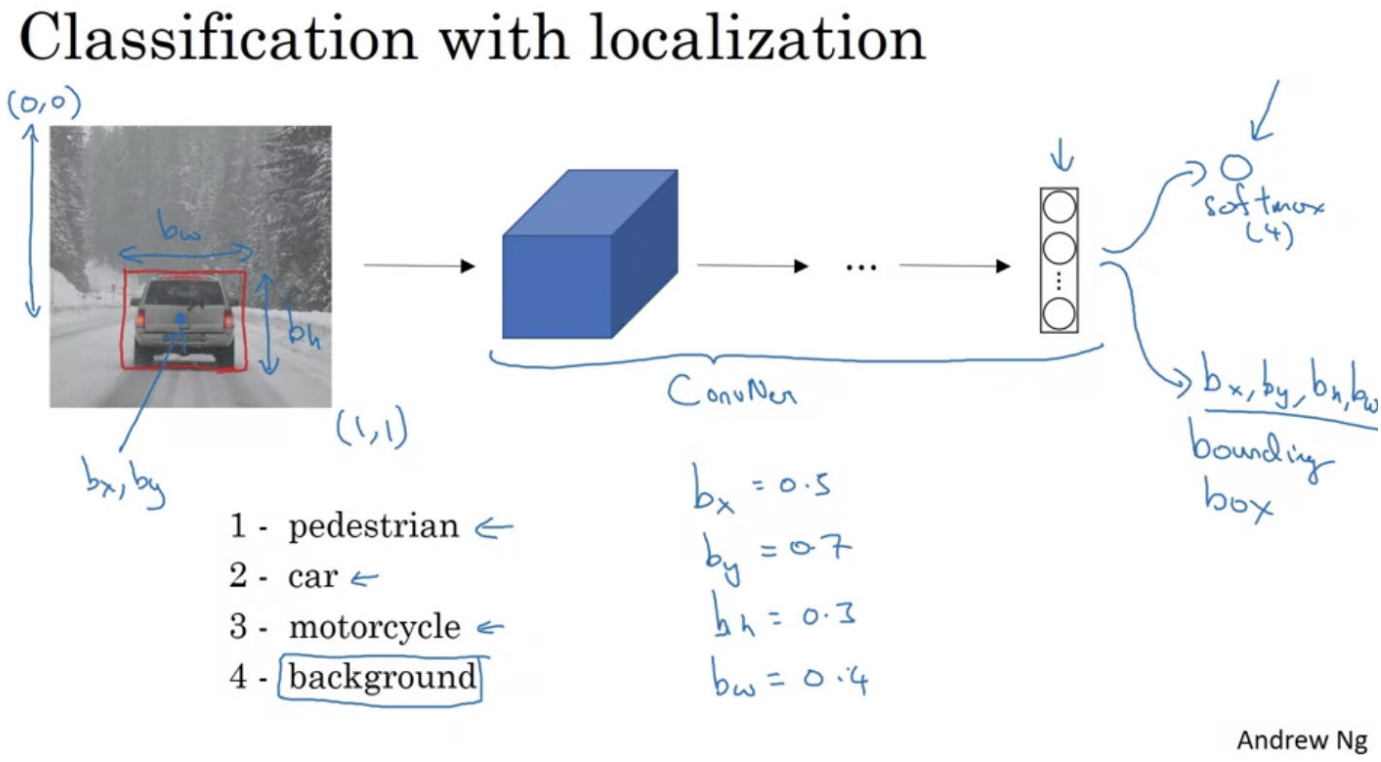


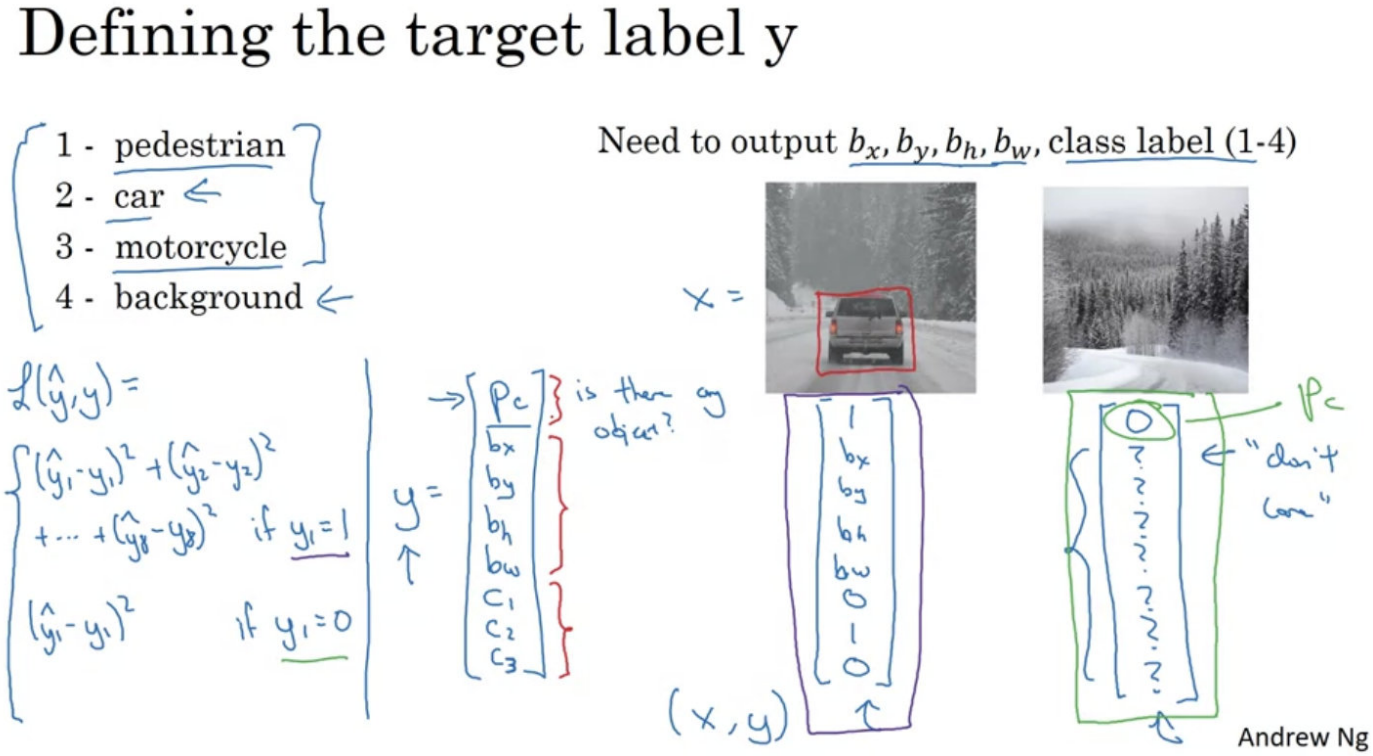
In this section, the `upper left` of the image is **(0, 0)**, and at the `lower right` is **(1, 1)**.

The bounding box is denoted by:

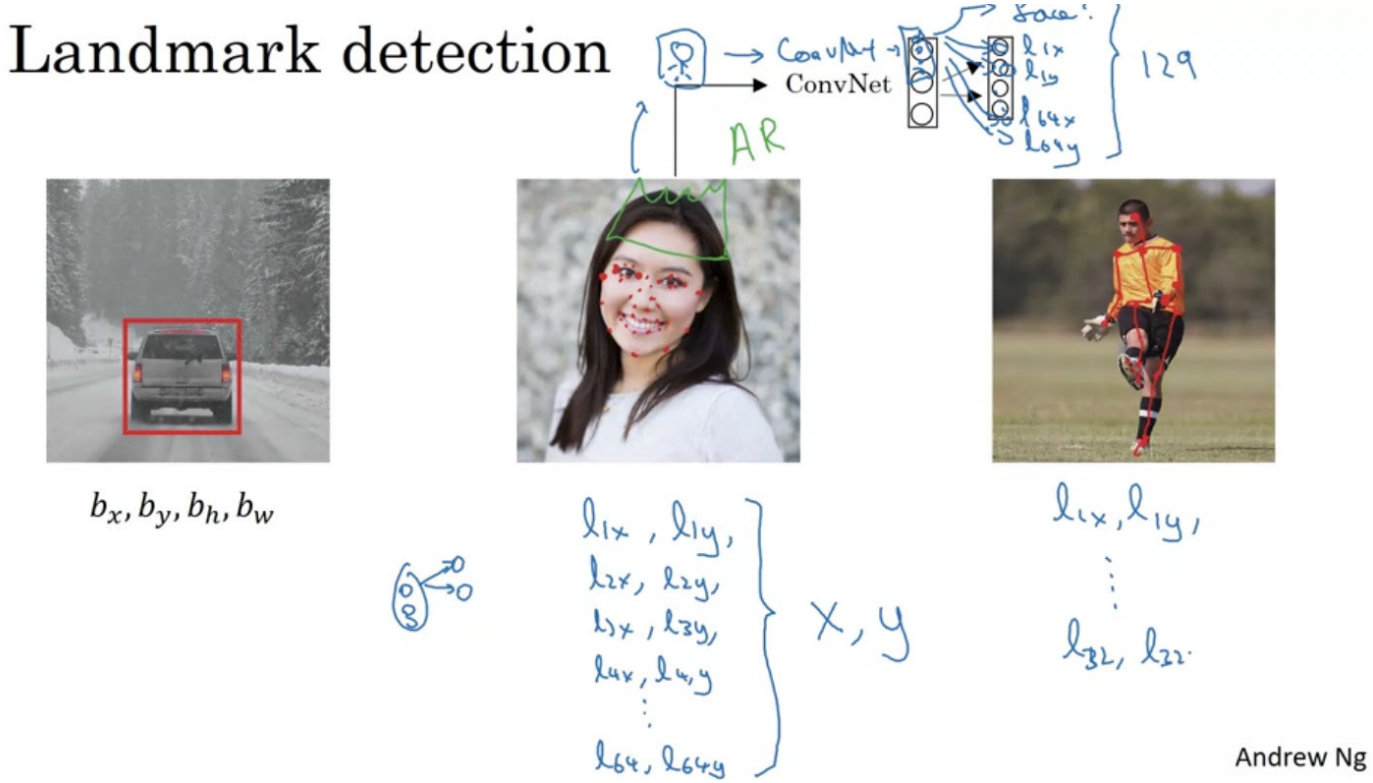
$$(b_x, b_y, b_h, b_w)$$

where b_x, b_y are the coordinates of the midpoint, b_h, b_w are height and weight respectively.





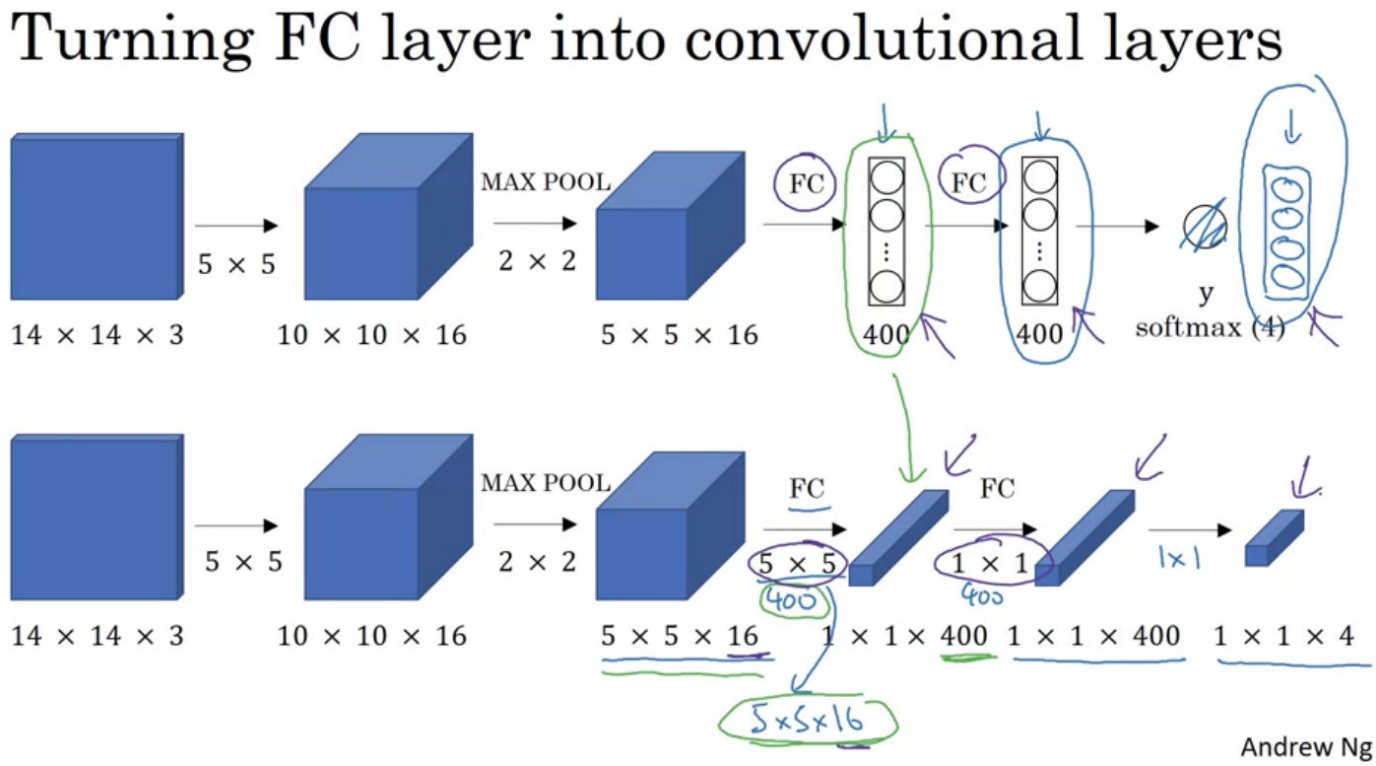
Landmark detection

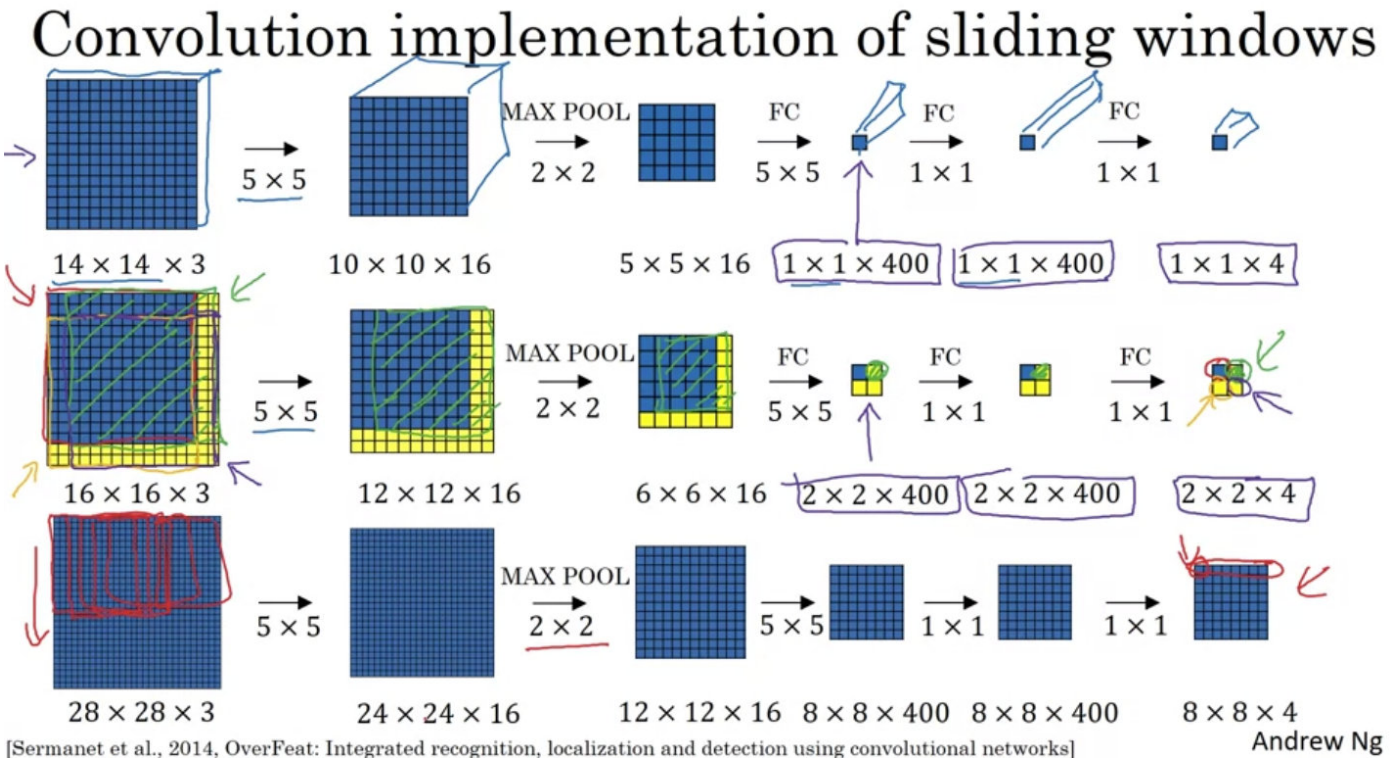


Object detection

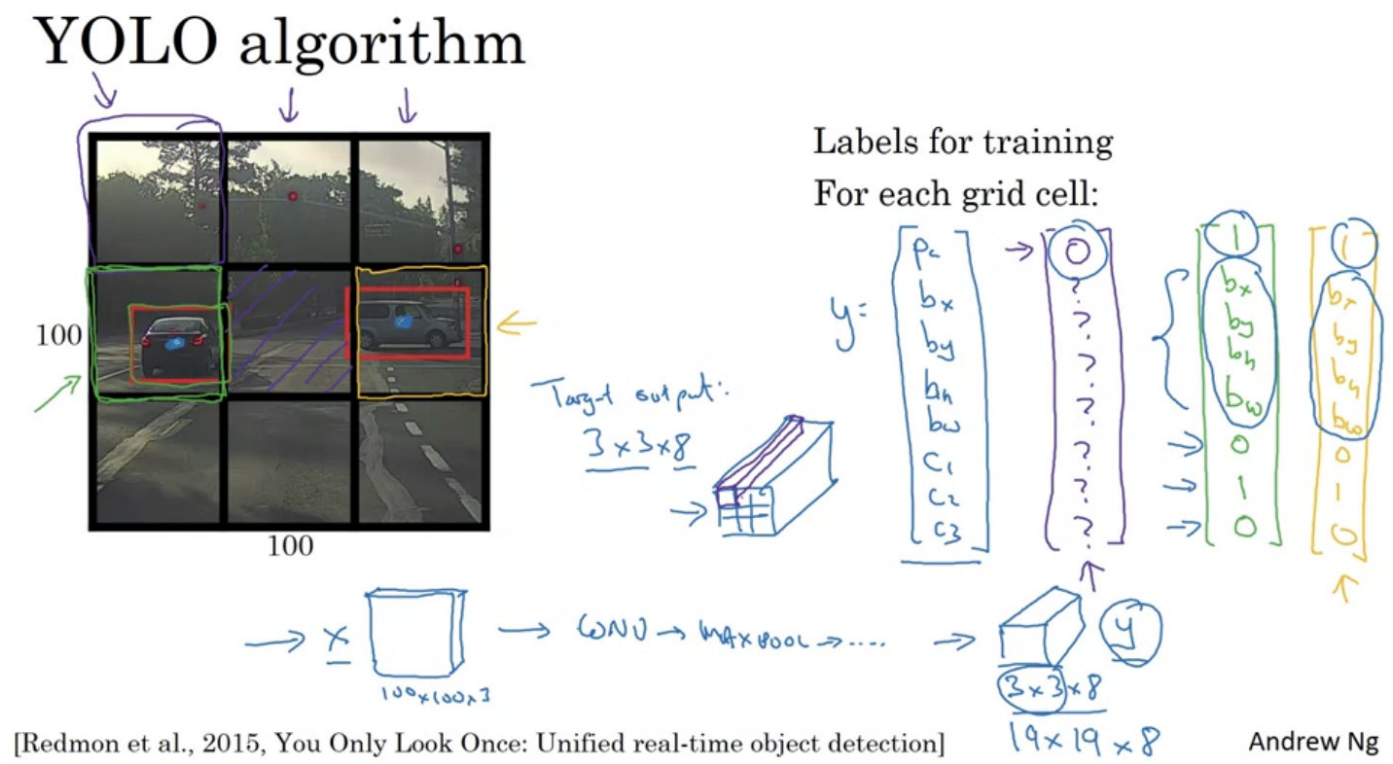
Before the raise of deep learning, people used **sliding windows** detection with simple linear classifer over hand-engineer features in order to perform object detection.

Convolutional implementation of sliding windows



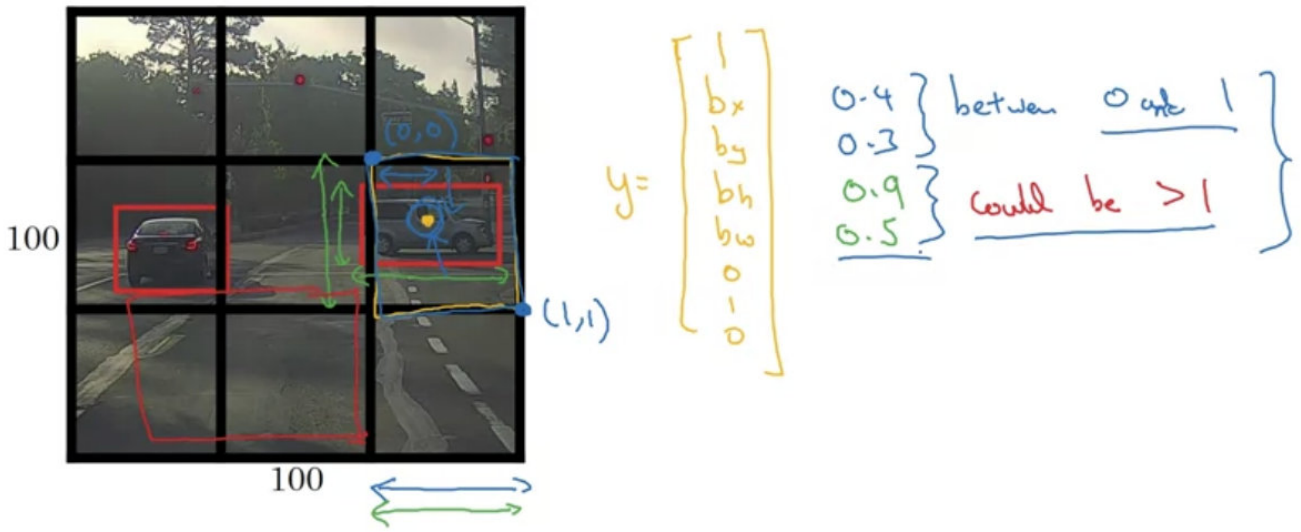


Bounding box predictions



The bounding box for YOLO (b_x, b_y, b_h, b_w) is specified relative to the grid cell. And the width and height of the bounding box are specified as `fractions` of the overall width and height of the grid cell.

Specify the bounding boxes

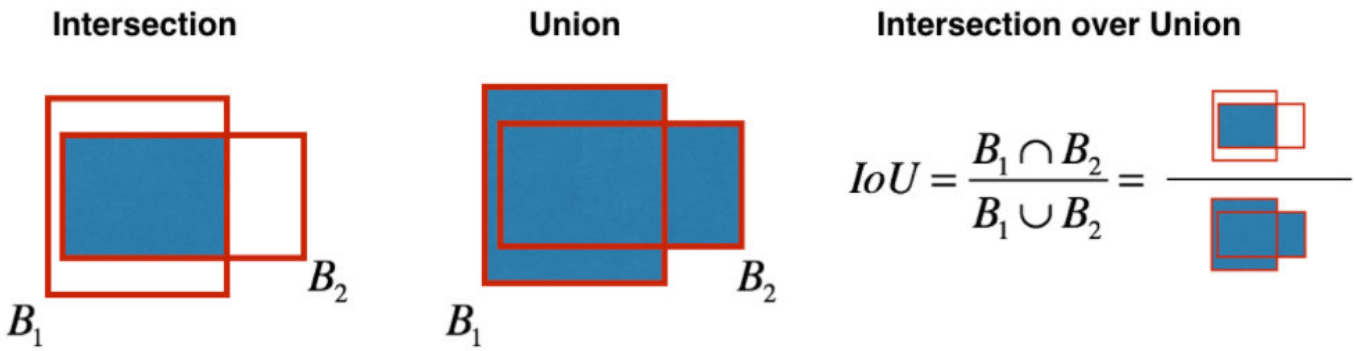


Intersection over Union (IoU)

$$IoU = \frac{Area\ of\ intersection}{Area\ of\ union}$$

"Correct" if $IoU \geq 0.5$

More generally, IoU is a measure of the overlap between two bounding boxes.



Non-max suppression

Non-max suppression is a way to make sure the algorithm detects each object **only once**.

Non-max means that you are going to output your **maximal probabilities classifications**, but suppress the close-by ones that are non-maximal.

Non-max suppression algorithm

19× 19

Each output prediction is: $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix}$

Discard all boxes with $p_c \leq 0.6$

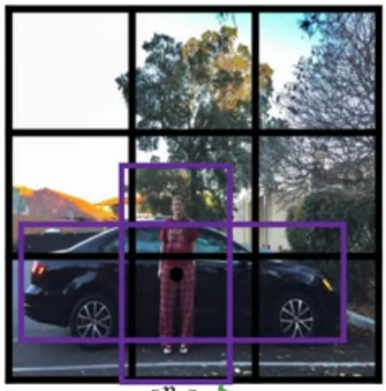
While there are any remaining boxes:

- Pick the box with the largest p_c Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

Andrew Ng

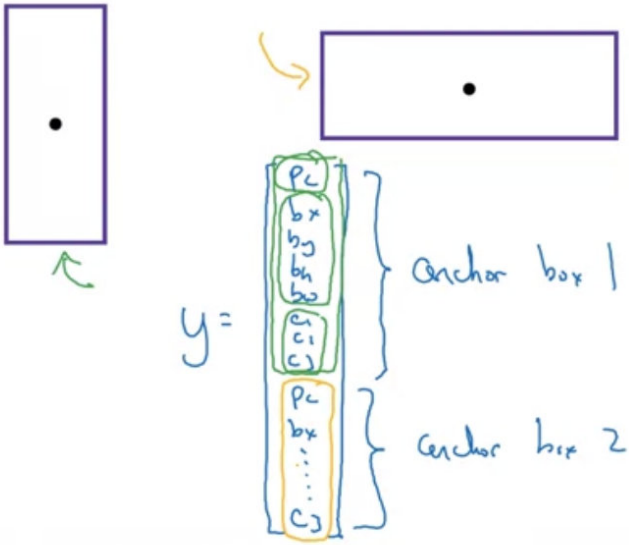
Anchor boxes

Overlapping objects:



$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Anchor box 1: Anchor box 2:



[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

Andrew Ng

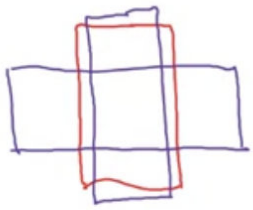
Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y:

$3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

(grid cell, anchor box)

Output y:

$3 \times 3 \times 16$

$3 \times 3 \times 2 \times 8$

Andrew Ng

Two cases that the algorithm does not handle well:

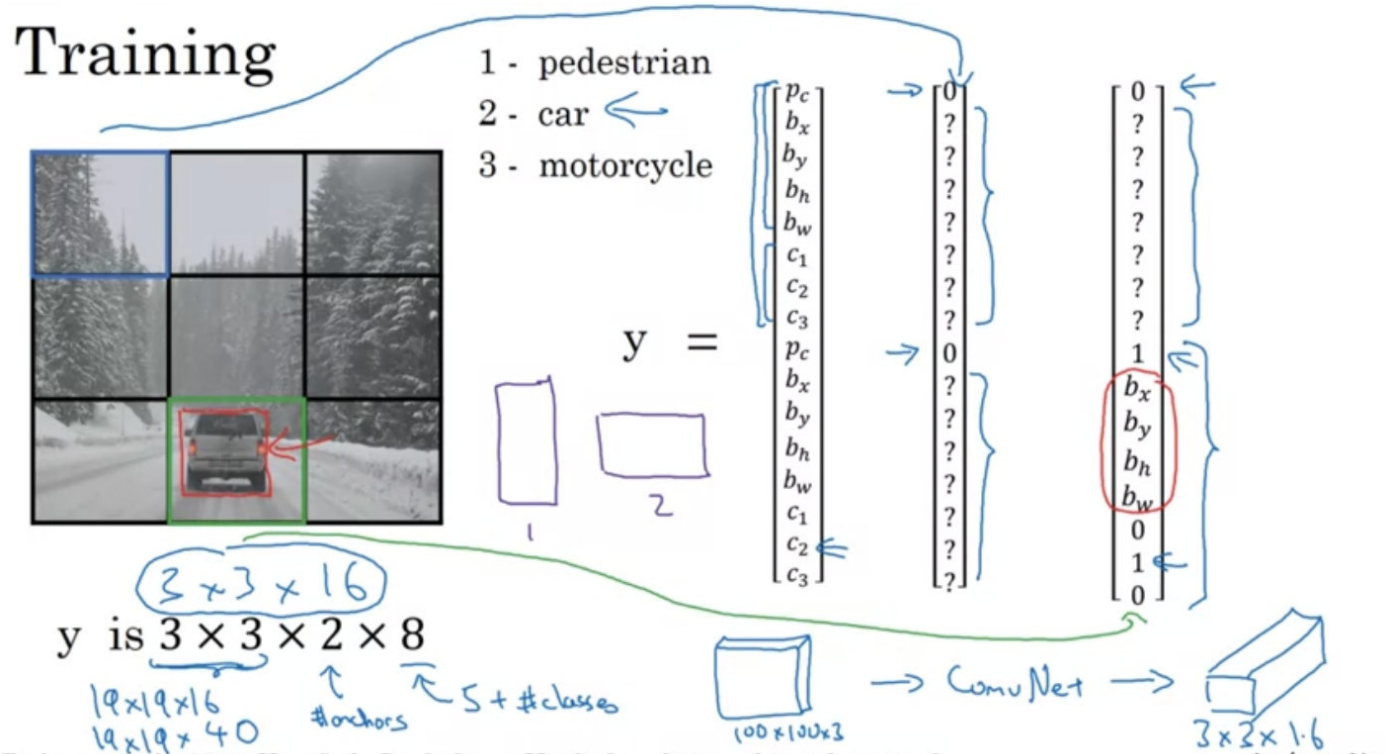
- If there are **more than two** objects in the same grid cell.
- Two objects are associated with same grid cell but both of them have the **same anchor box shape**.

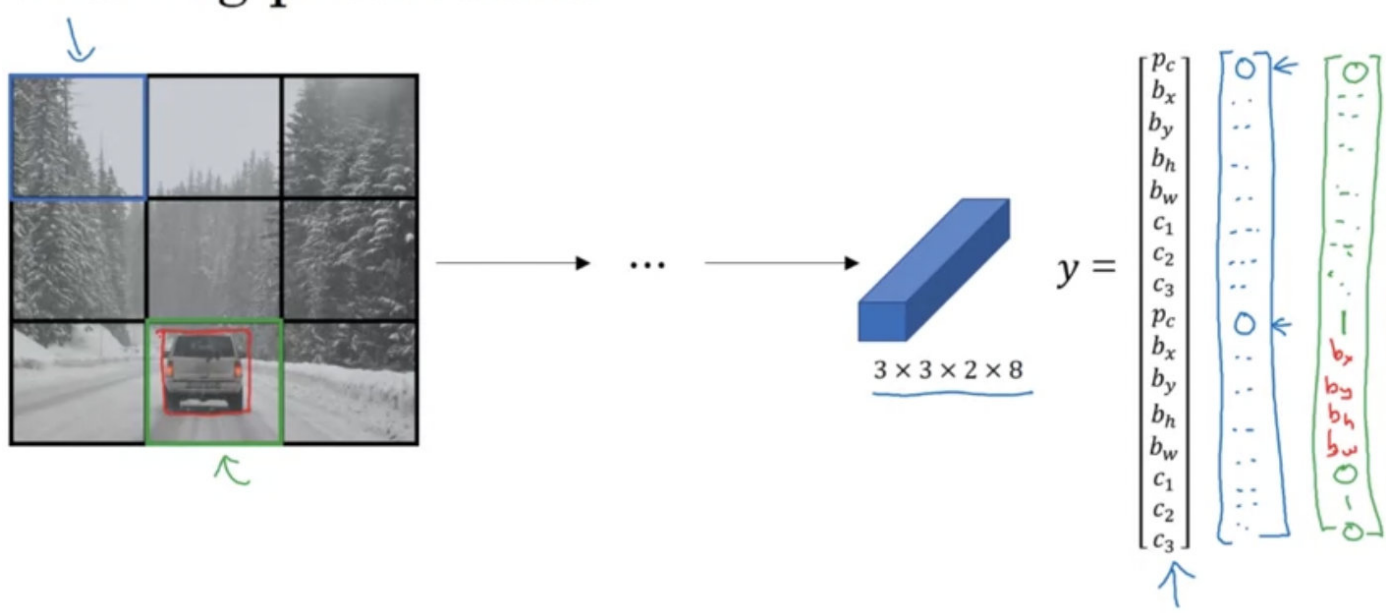
How to choose anchor boxes?

People used to just choose anchor boxes by hand or choose maybe 5 or 10 anchor box shapes that spans a variety of shapes that seems to cover the types of objects you seem to detect. As a much more advanced version is to use a **K-means algorithm** to group together two types of objects shapes you tend to get. And then to use that to select a set of anchor boxes that are most stereotypically representative of the multiple, of the dozens of object classes you're trying to detect.

YOLO Algorithm

This algorithm "only looks once" at the image in the sense that it requires **only one forward propagation** pass through the network to make predictions.


Training
1 - pedestrian
2 - car
3 - motorcycle
y =
y is 3 × 3 × 2 × 8
3 × 3 × 16
19 × 19 × 16
19 × 19 × 40
#anchors
5 + #classes
100 × 100 × 3
ConvNet
3 × 3 × 16
Redmon et al., 2015, You Only Look Once: Unified real-time object detection]
Andrew Ng


Making predictions
3 × 3 × 2 × 8
y =

Outputting the non-max supressed outputs

- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

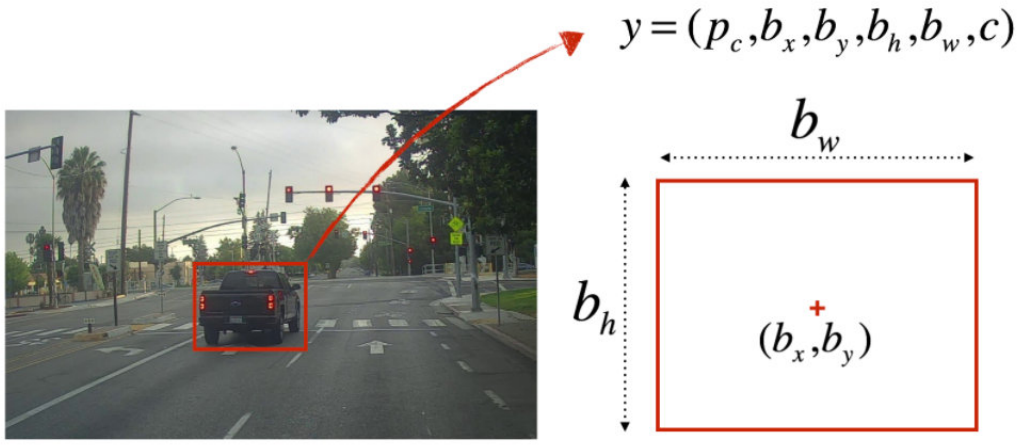
Programming Assignment Example

Inputs and outputs

- The **input** is a batch of images, and each image has the shape (m, 608, 608, 3)
- The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers $(p_c, b_x, b_y, b_h, b_w, c)$ as explained above. If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.

Anchor Boxes

- Anchor boxes are chosen by exploring the training data to choose reasonable height/width ratios that represent the different classes. For this assignment, 5 anchor boxes were chosen for you (to cover the 80 classes), and stored in the file './model_data/yolo_anchors.txt'
- The dimension for anchor boxes is the second to last dimension in the encoding: $(m, n_H, n_W, anchors, classes)$.
- The YOLO architecture is: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).



$p_c = 1$: confidence of an object being present in the bounding box

$c = 3$: class of the object being detected (here 3 for "car")

Figure 1: Definition of a box

Encoding

Let's look in greater detail at what this encoding represents.

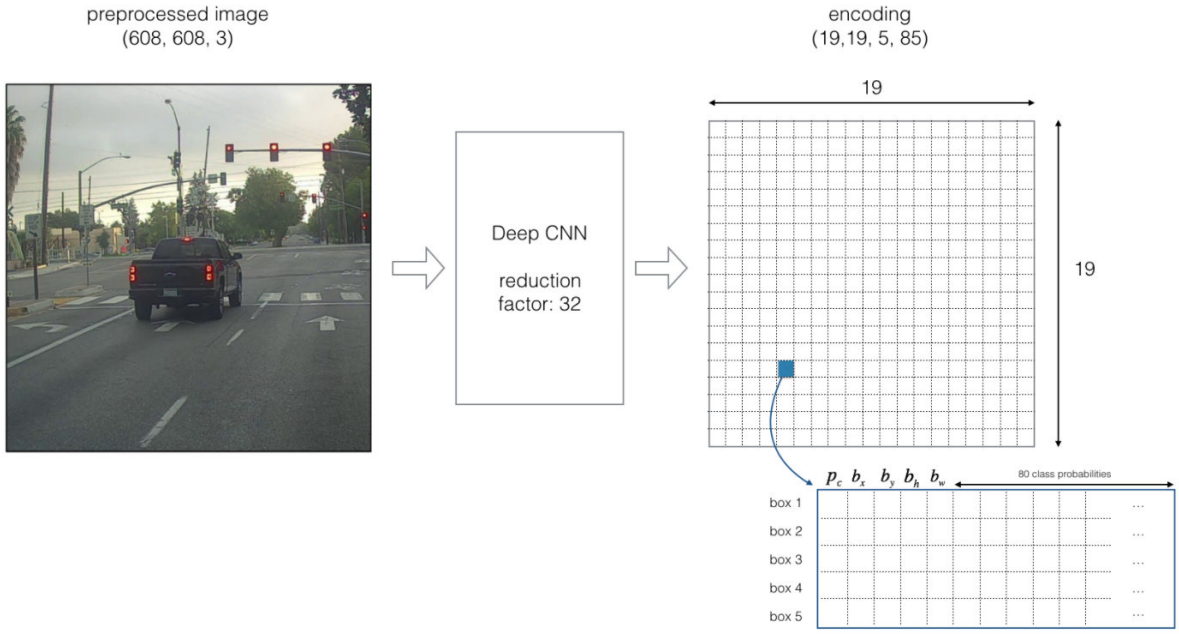


Figure 2 : Encoding architecture for YOLO

If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Since you're using 5 anchor boxes, each of the 19 x19 cells thus encodes information about 5 boxes. Anchor boxes are defined only by their width and height.

For simplicity, you'll flatten the last two dimensions of the shape (19, 19, 5, 85) encoding, so the output of the Deep CNN is (19, 19, 425).

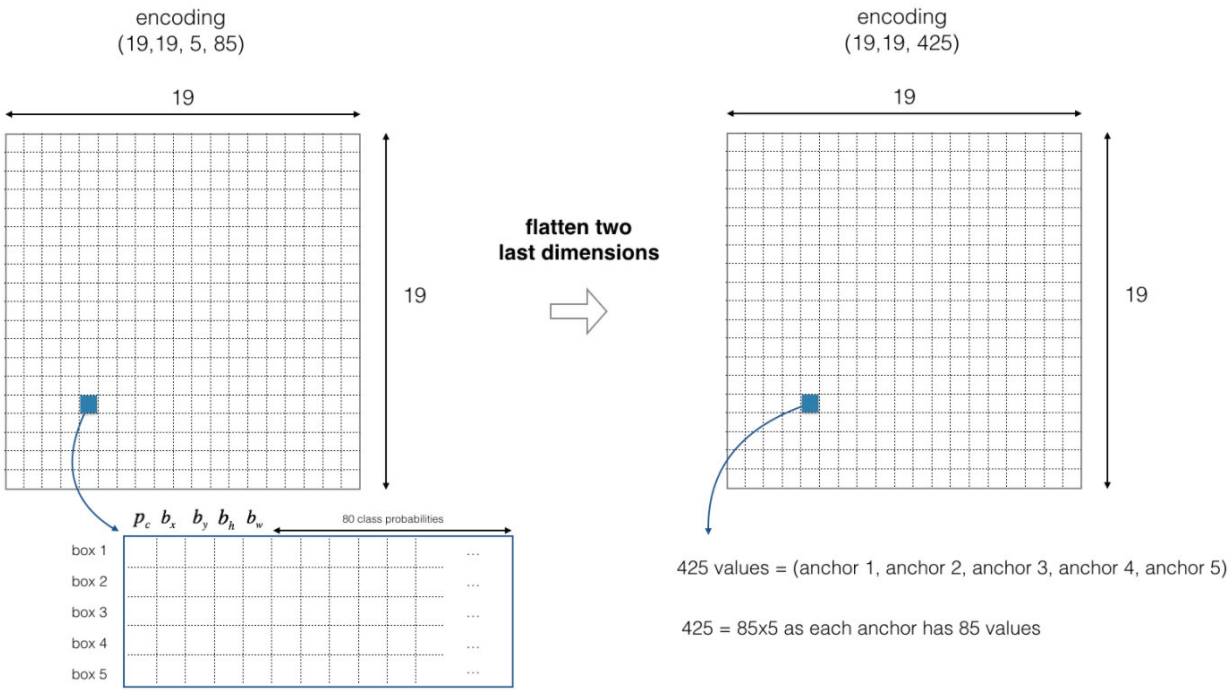


Figure 3 : Flattening the last two last dimensions

Class score

Now, for each box (of each cell) you'll compute the following element-wise product and extract a probability that the box contains a certain class.

The class score is $score_{c,i} = p_c \times c_i$: the probability that there is an object p_c times the probability that the object is a certain class c_i.

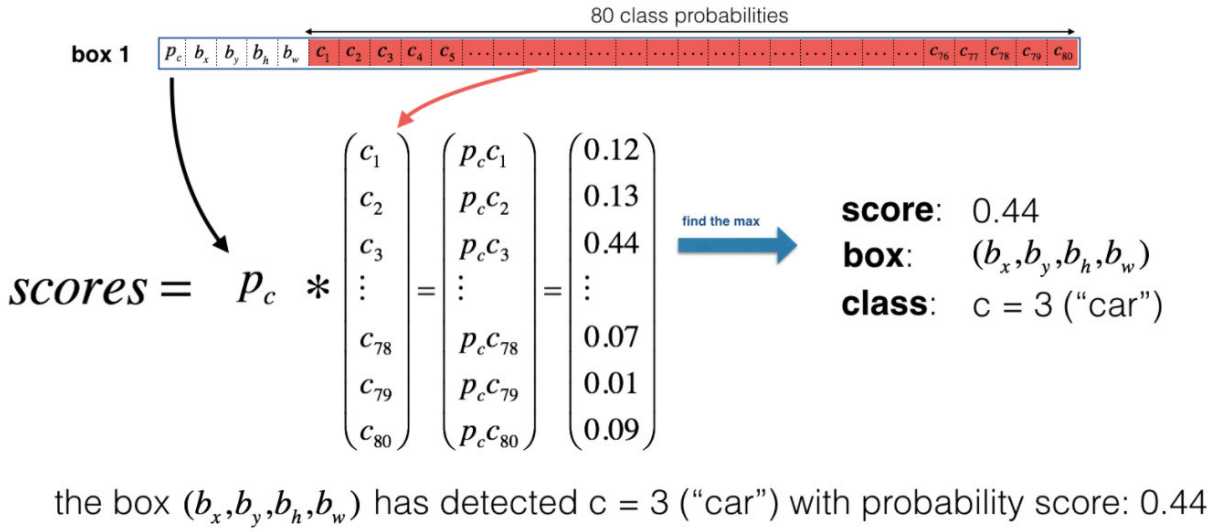


Figure 4: Find the class detected by each box

Visualizing classes

Here's one way to visualize what YOLO is predicting on an image:

- For each of the 19x19 grid cells, find the maximum of the probability scores (taking a max across the 80 classes, one maximum for each of the 5 anchor boxes).
- Color that grid cell according to what object that grid cell considers the most likely.

Doing this results in this picture:

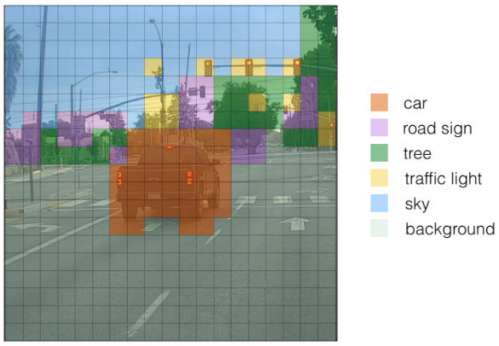


Figure 5: Each one of the 19x19 grid cells is colored according to which class has the largest predicted probability in that cell.

Note that this visualization isn't a core part of the YOLO algorithm itself for making predictions; it's just a nice way of visualizing an intermediate result of the algorithm.

Visualizing bounding boxes

Another way to visualize YOLO's output is to plot the bounding boxes that it outputs. Doing that results in a visualization like this:

Figure 6: Each cell gives you 5 boxes. In total, the model predicts: 19x19x5 = 1805 boxes just by looking once at the image (one forward pass through the network)! Different colors denote different classes.

Non-Max suppression

In the figure above, the only boxes plotted are ones for which the model had assigned a high probability, but this is still too many boxes. You'd like to reduce the algorithm's output to a much smaller number of detected objects.

To do so, you'll use **non-max suppression**. Specifically, you'll carry out these steps:

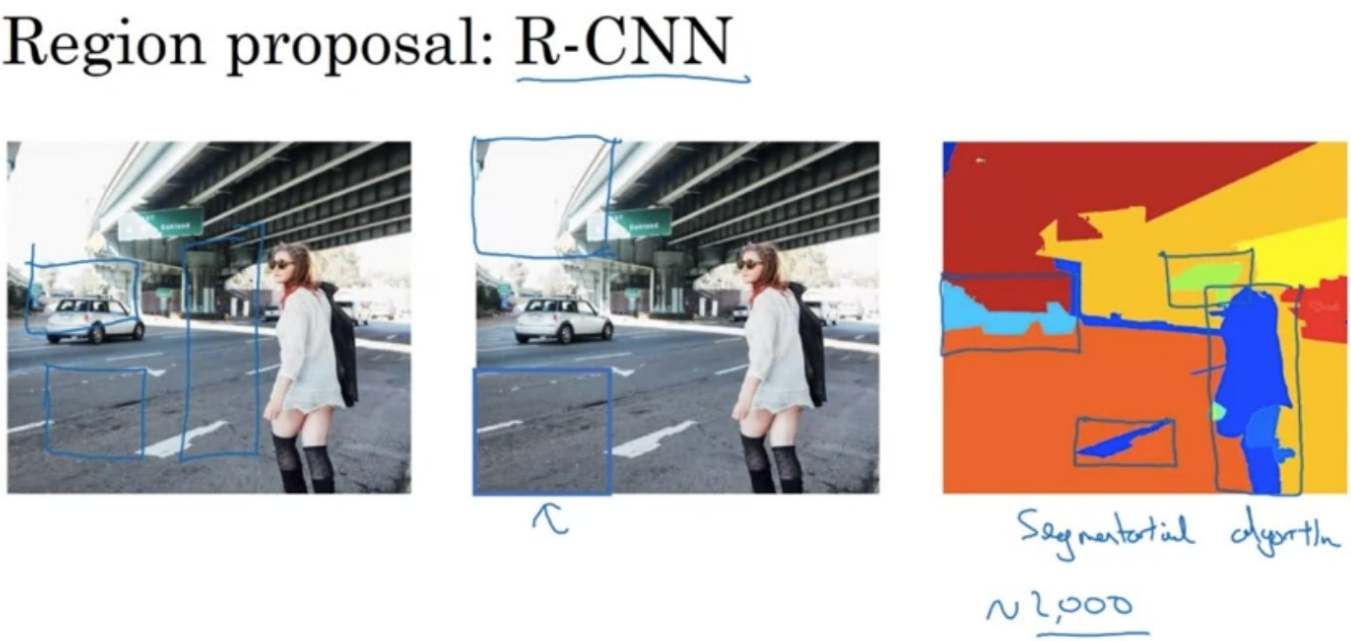
- Get rid of boxes with a low score. Meaning, the box is not very confident about detecting a class, either due to the low probability of any object, or low probability of this particular class.
- Select only one box when several boxes overlap with each other and detect the same object.

The model gives you a total of 19x19x5x85 numbers, with each box described by 85 numbers. It's convenient to rearrange the (19,19,5,85) (or (19,19,425)) dimensional tensor into the following variables:

- `box_confidence` : tensor of shape $(19, 19, 5, 1)$ containing p_c (confidence probability that there's some object) for each of the 5 boxes predicted in each of the 19x19 cells.
- `boxes` : tensor of shape $(19, 19, 5, 4)$ containing the midpoint and dimensions (b_x, b_y, b_h, b_w) for each of the 5 boxes in each cell.
- `box_class_probs` : tensor of shape $(19, 19, 5, 80)$ containing the "class probabilities" $(c_1, c_2, \ldots c_{80})$ for each of the 80 classes for each of the 5 boxes per cell.

Figure 7 : In this example, the model has predicted 3 cars, but it's actually 3 predictions of the same car. Running non-max suppression (NMS) will select only the most accurate (highest probability) of the 3 boxes.

Region proposal: R-CNN



R-CNN

Propose regions. Classify proposed regions one at a time. Output label + bounding box.

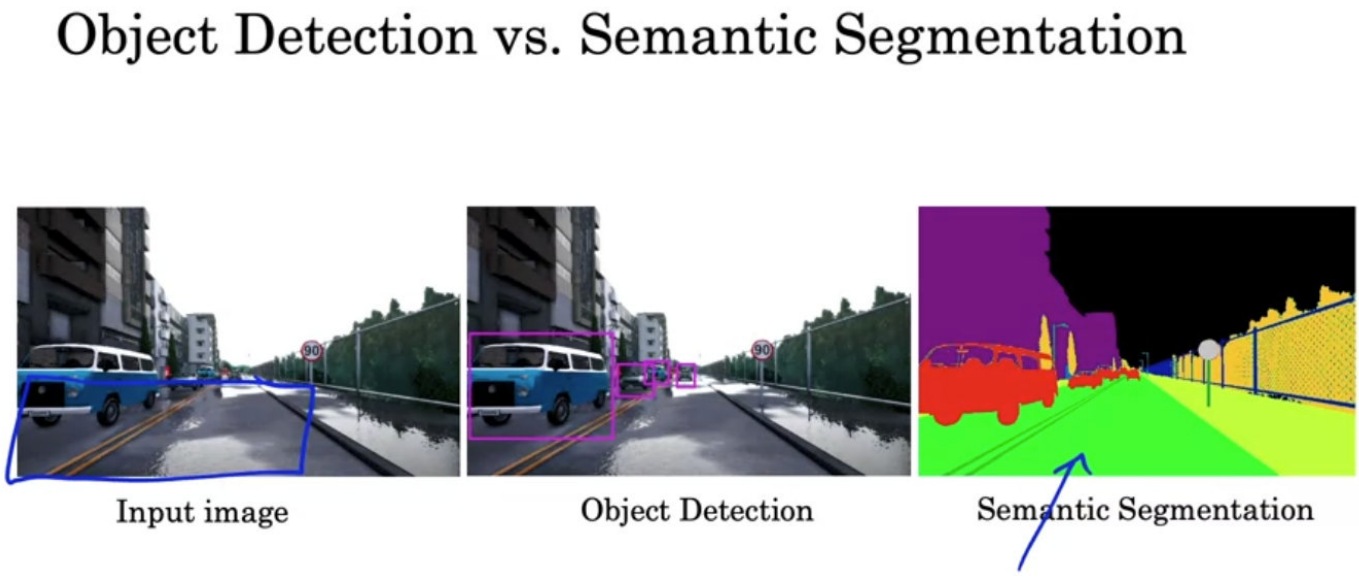
Fast R-CNN

Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.

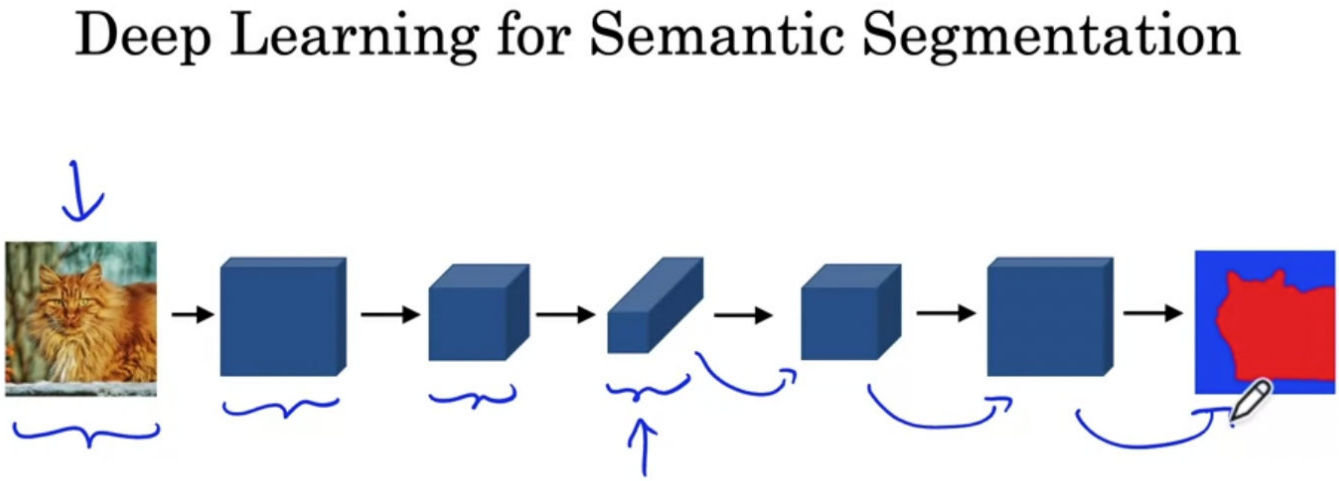
Faster R-CNN

Use convolutional network to propose regions.

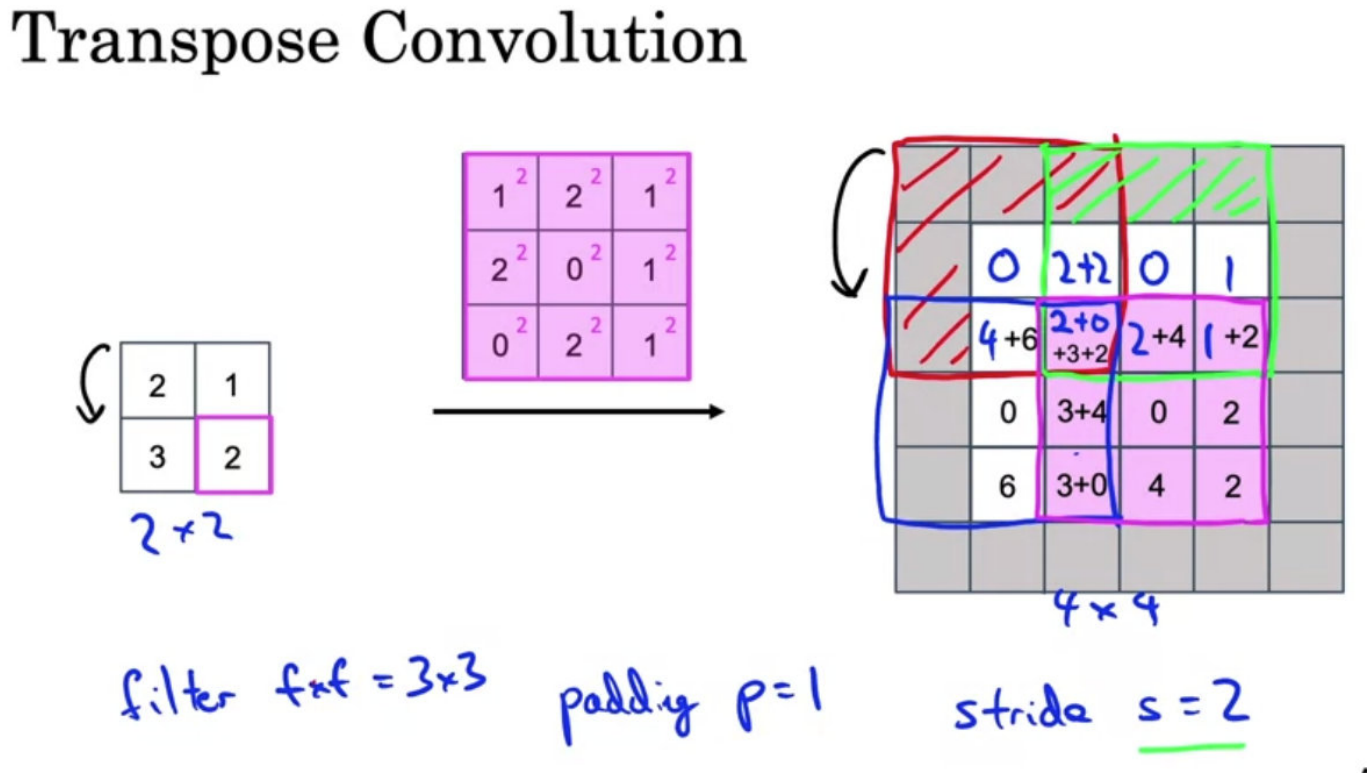
Semantic Segmentation



U-Net



Transpose Convolutions

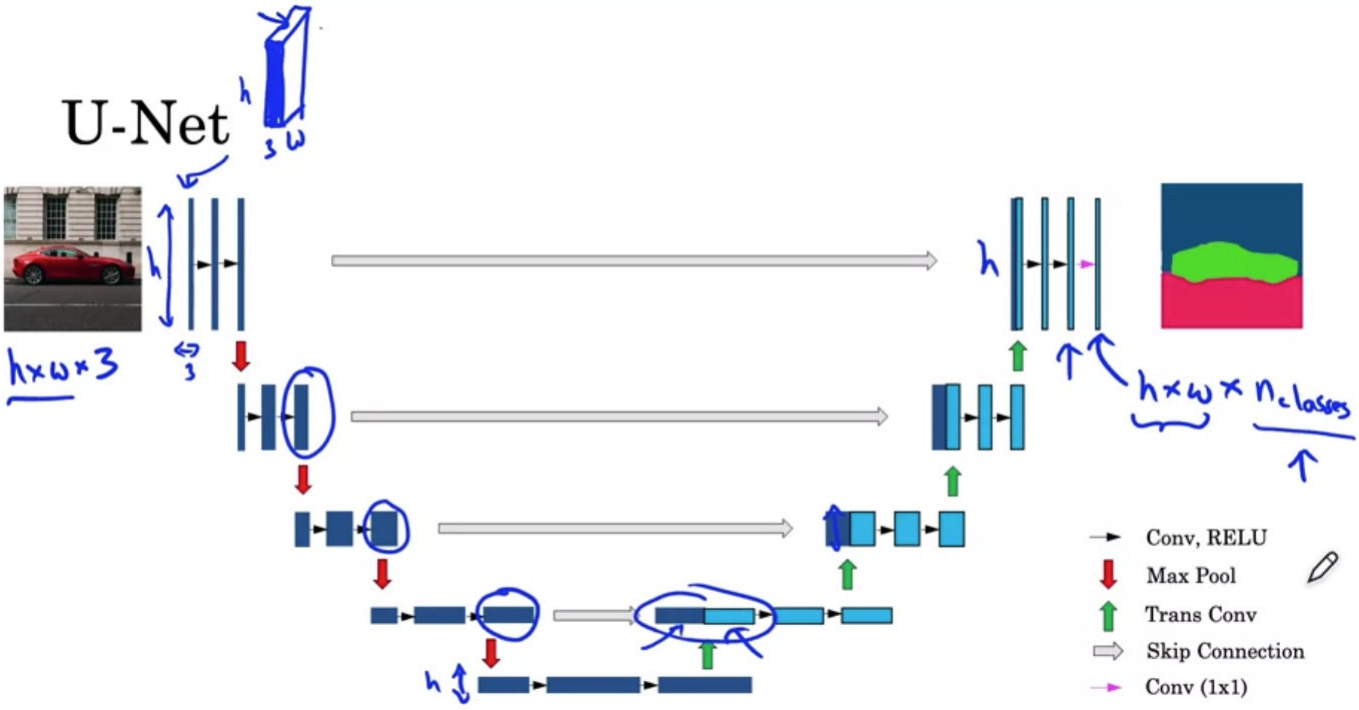


U-Net Architecture

U-Net builds on a previous architecture called the Fully Convolutional Network, or FCN, which replaces the dense layers found in a typical CNN with a transposed convolution layer that upsamples the feature map back to the size of the original input image, while preserving the spatial information. This is necessary because the dense layers destroy spatial information (the "where" of the image), which is an essential part of image segmentation tasks. An added bonus of using transpose convolutions is that the input size no longer needs to be fixed, as it does when dense layers are used.

Unfortunately, the final feature layer of the FCN suffers from information loss due to downsampling too much. It then becomes difficult to upsample after so much information has been lost, causing an output that looks rough.

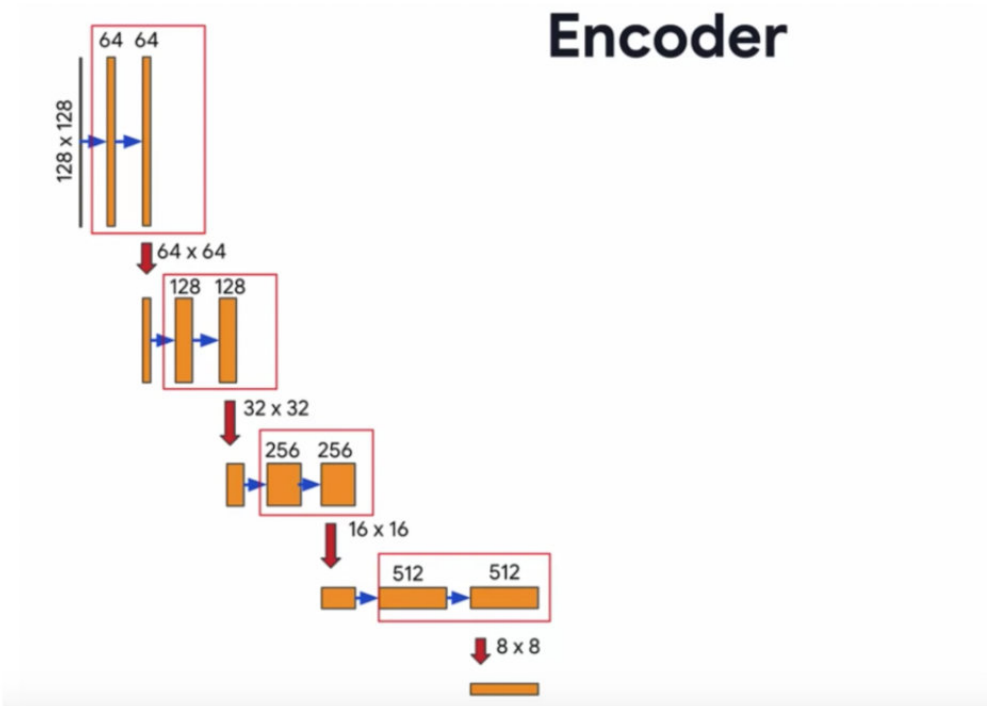
U-Net improves on the FCN, using a somewhat similar design, but differing in some important ways. Instead of one transposed convolution at the end of the network, it uses a matching number of convolutions for downsampling the input image to a feature map, and **transposed convolutions** for upsampling those maps back up to the original input image size. It also adds **skip connections**, to retain information that would otherwise become lost during encoding. Skip connections send information to every upsampling layer in the decoder from the corresponding downsampling layer in the encoder, capturing finer information while also keeping computation low. These help prevent information loss, as well as model overfitting.



Encoder

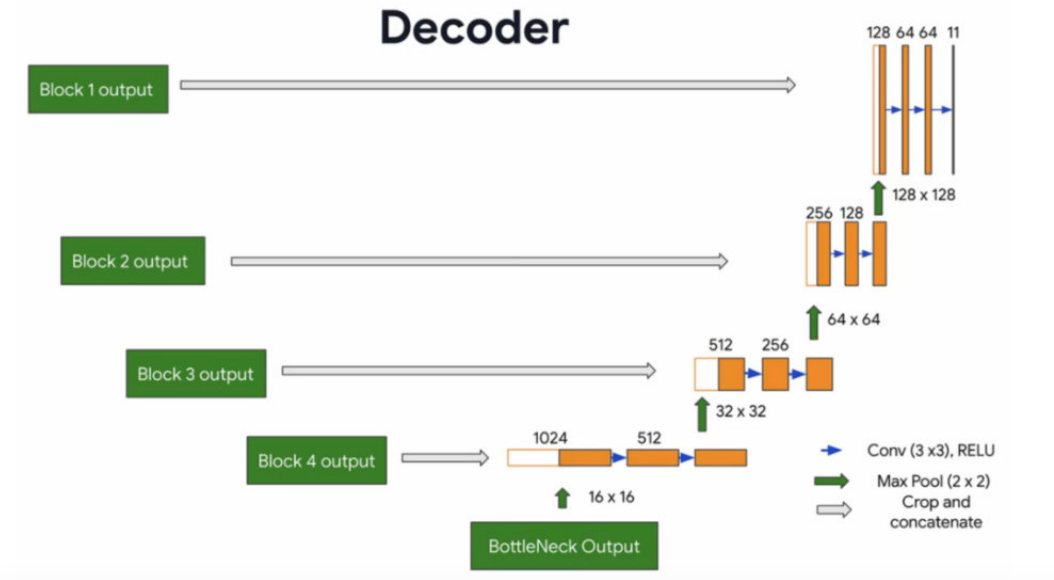
The encoder is a stack of various conv_blocks:

Each `conv_block()` is composed of 2 **Conv2D** layers with ReLU activations.

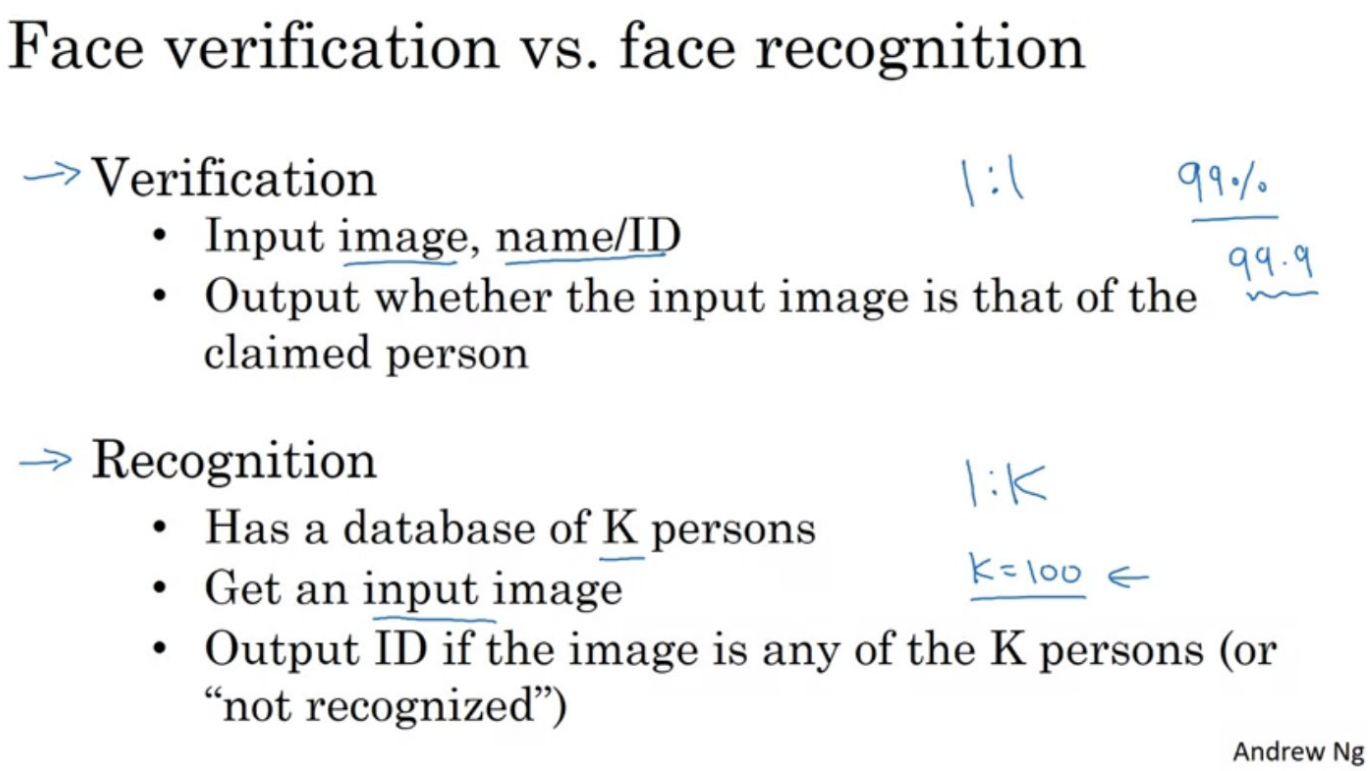


Decoder

The decoder, or upsampling block, upsamples the features back to the original image size. At each upsampling level, you'll take the output of the corresponding encoder block and concatenate it before feeding to the next decoder block.



Face Recognition



One-shot learning

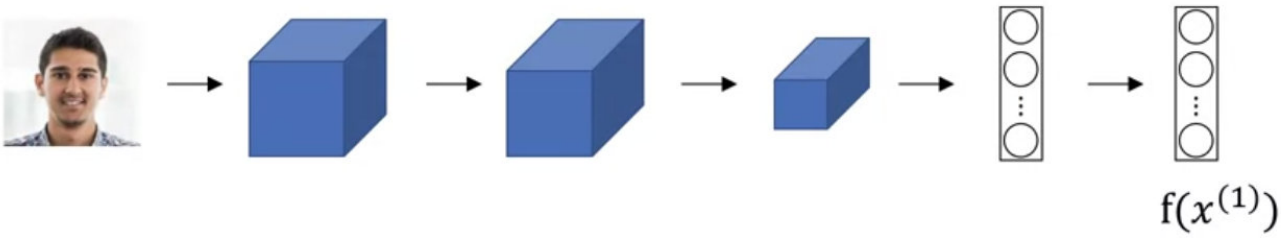
Learning from **one example** to recognize the person again. The method is learn a `similarity function`.

$$d(\text{img1}, \text{img2}) = \text{degree of difference between images}$$

If $d(\text{img1}, \text{img2}) \leq \tau \rightarrow$ 'same', $d(\text{img1}, \text{img2}) > \tau \rightarrow$ 'diff'

Siamese Network

Goal of learning



Parameters of NN define an encoding $f(x^{(i)})$ 128

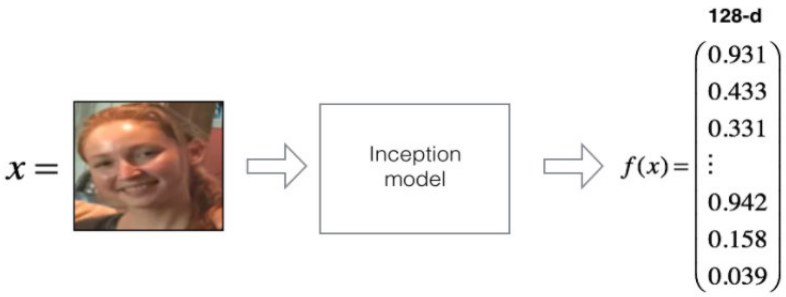
Learn parameters so that:

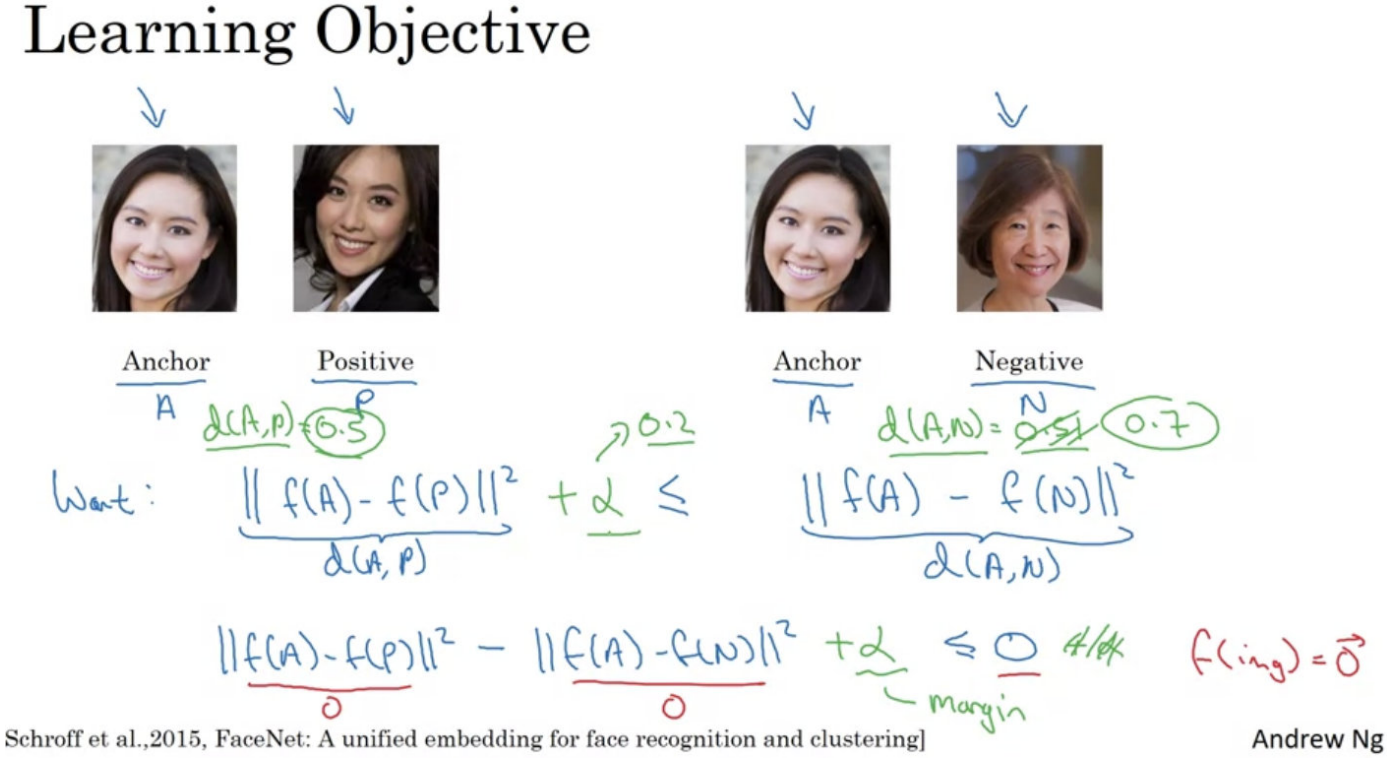
If $x^{(i)}, x^{(j)}$ are the same person, $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is small.

If $x^{(i)}, x^{(j)}$ are different persons, $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is large.

Triplet loss

For an image x, its encoding is denoted as $f(x)$, where f is the function computed by the neural network.





Training will use triplets of images (A, P, N):

- A is an "Anchor" image--a picture of a person.
- P is a "Positive" image--a picture of the same person as the Anchor image.
- N is a "Negative" image--a picture of a different person than the Anchor image.

These triplets are picked from the training dataset. $(A^{(i)}, P^{(i)}, N^{(i)})$ is used here to denote the i-th training example.

You'd like to make sure that an image $A^{(i)}$ of an individual is closer to the Positive $P^{(i)}$ than to the Negative image $N^{(i)}$ by at least a margin α:

$$||f\left(A^{(i)}\right) - f\left(P^{(i)}\right)||_2^2 + \alpha < ||f\left(A^{(i)}\right) - f\left(N^{(i)}\right)||_2^2$$

You would thus like to minimize the following "triplet cost":

$$\mathcal{J} = \sum_{i=1}^{m} \Big[\underbrace{|| f(A^{(i)}) - f(P^{(i)}) ||_2^2}_{(1)} - \underbrace{|| f(A^{(i)}) - f(N^{(i)}) ||_2^2}_{(2)} + \alpha\Big]_+$$

Here, the notation "$[z]_+$" is used to denote $max(z, 0)$.

Notes:

- The term (1) is the squared distance between the anchor "A" and the positive "P" for a given triplet; you want this to be small.
- The term (2) is the squared distance between the anchor "A" and the negative "N" for a given triplet, you want this to be relatively large. It has a minus sign preceding it because minimizing the negative of the term is the same as maximizing that term.
- α is called the margin. It's a hyperparameter that you pick manually. e.g. $\alpha = 0.2$.

Training set: 10k pictures of 1k persons -- Take 10k pictures and use it to generate the treplets (A, P, N).

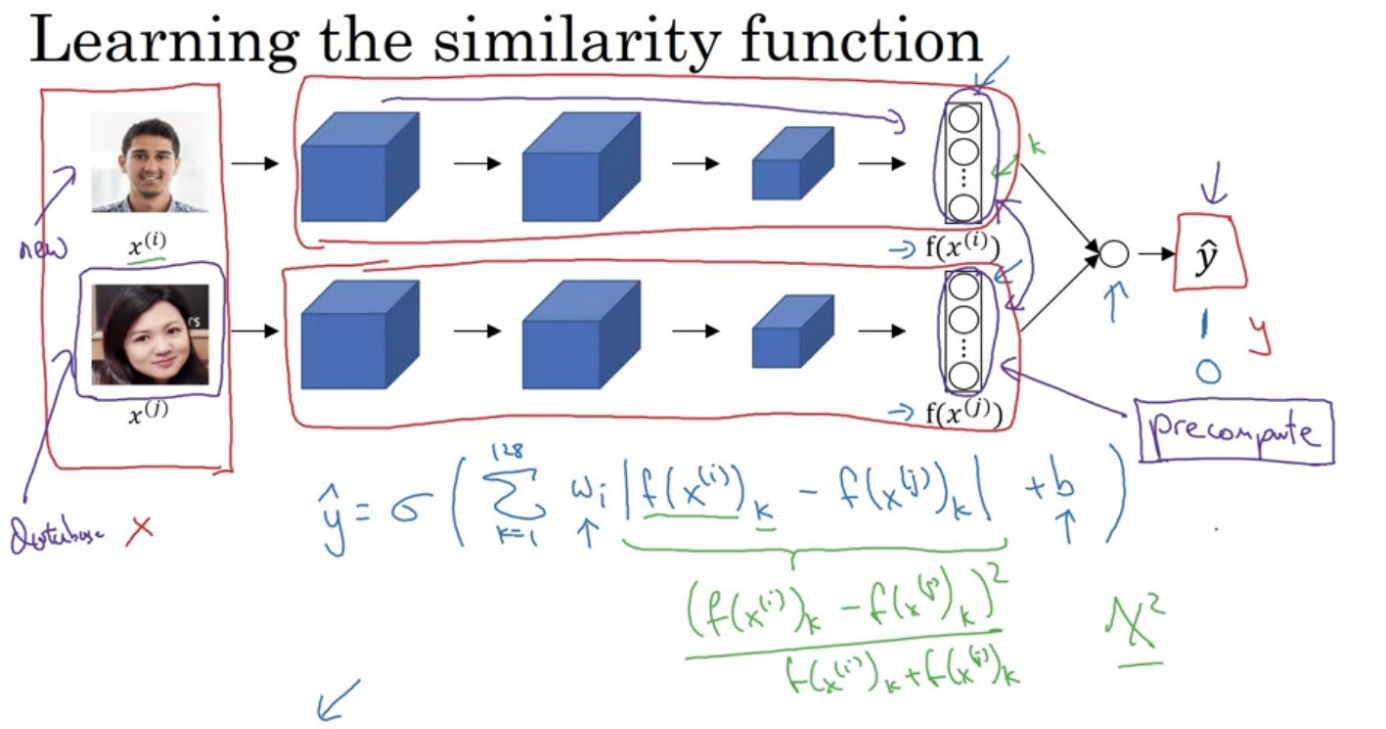
Choosing the triplets A, P, N

During training, if A, P, N are chosen **randomly**, $d(A, P) + \alpha \leq d(A, N)$ is **easily satisfied**.

Choose triplets that are "hard" to train on.

$$d(A, P) \approx d(A, N)$$

Face Verification and Binary Classification



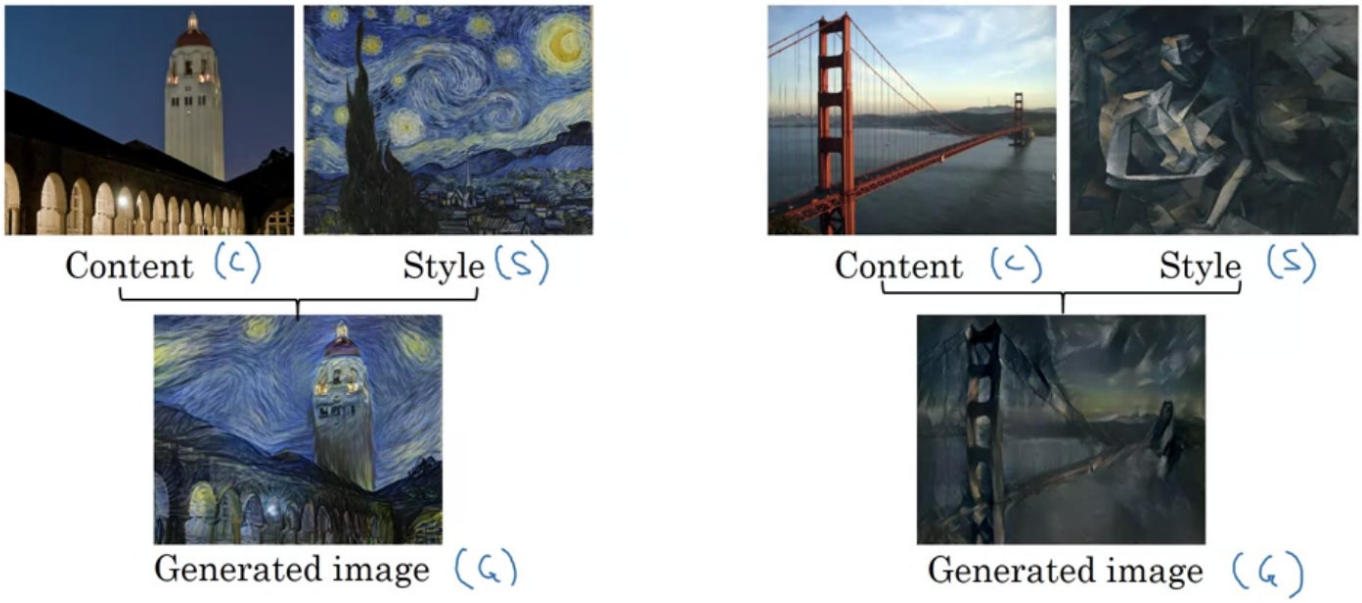
[Taigman et. al., 2014. DeepFace closing the gap to human level performance] Andrew Ng

What you should remember:

- Face verification solves an easier 1:1 matching problem; face recognition addresses a harder 1:K matching problem.
- Triplet loss is an effective loss function for training a neural network to learn an encoding of a face image.
- The same encoding can be used for verification and recognition. Measuring distances between two images' encodings allows you to determine whether they are pictures of the same person.

Neural Style Transfer

Neural style transfer



Cost function

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

Where $J_{content}(C, G)$ measures how similar is the content of the generated image G, to the content fo the content image C, $J_{style}(C, G)$ measures how similar is the style of the generated image G, to the style fo the style image S.

Find the generated image G

1. Initialize G randomly.

$$G : 100 \times 100 \times 3$$

2. Use gradient descent to minimize $J(G)$.

$$G := G - \frac{\partial}{\partial G} J(G)$$

Content cost function

Recap the cost function:

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

- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet.
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l one the images.
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content.

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

Style cost function

Define style as **correlation** between activations across channels.

Style matrix (Gram matrix)

- The style matrix is also called a "Gram matrix" .
- In linear algebra, the Gram matrix G of a set of vectors $(v_1, \ldots, v_n)$ is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j = np.dot(v_i, v_j)$.
- In other words, G_{ij} compares how similar v_i is to v_j: If they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large.

Let $a^{[l]}_{i,j,k} =$ activation at (i, j, k). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$. Particularly, $G^{[l]}_{kk'}$ will measure how correlated are the activations in channel k compared to the activations in channel k'.

$$G^{[l](S)}_{kk'} = \sum_{i}^{n_H^{[l]}} \sum_{j}^{n_W^{[l]}} a^{[l](S)}_{i,j,k} a^{[l](S)}_{i,j,k'}$$

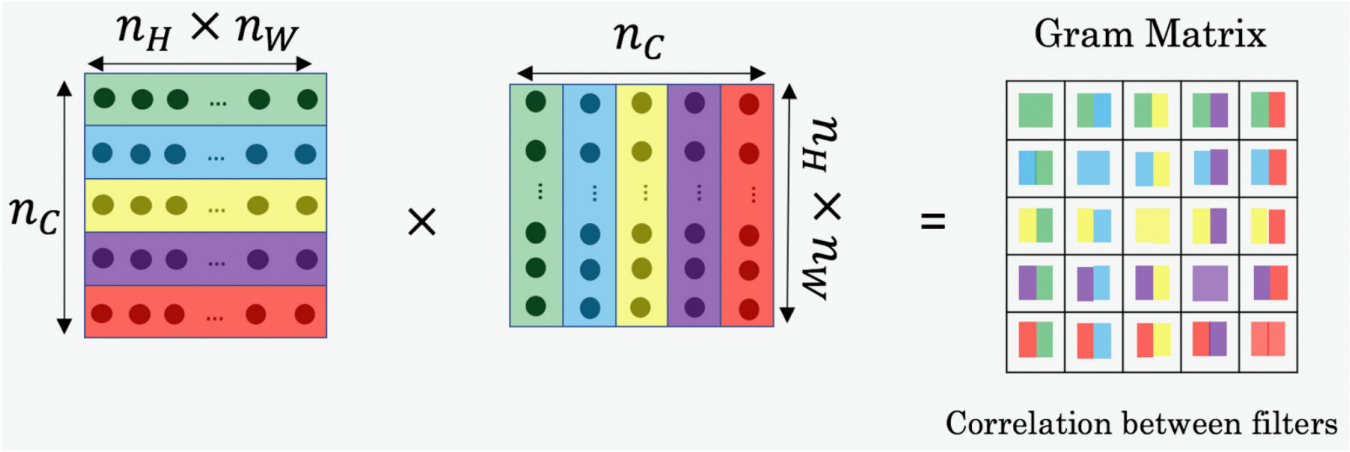
$$G^{[l](G)}_{kk'} = \sum_{i}^{n_H^{[l]}} \sum_{j}^{n_W^{[l]}} a^{[l](G)}_{i,j,k} a^{[l](G)}_{i,j,k'}$$

$$\begin{aligned} J^{[l]}_{style}(S, G) &= \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \|G^{[l](S)} - G^{[l](G)}\|_F^2 \\ &= \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \sum_k \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})^2 \end{aligned}$$

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

Compute Gram matrix G_{gram}

You will compute the Style matrix by multiplying the "unrolled" filter matrix with its transpose:



$G_{(gram)ij}$: **correlation**

The result is a matrix of dimension (n_C, n_C) where n_C is the number of filters (channels). The value $G_{(gram)i,j}$ measures how similar the activations of filter i are to the activations of filter j.

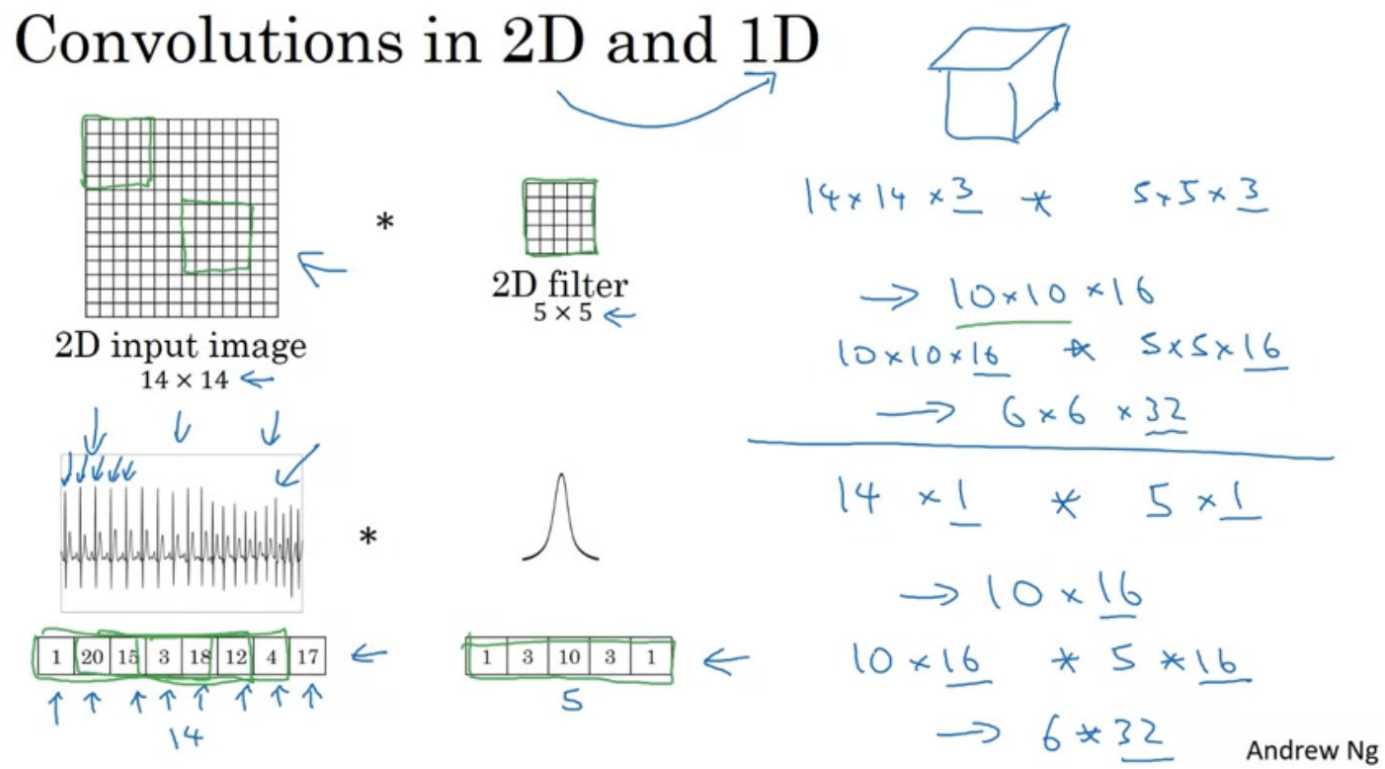
$G_{(gram),ii}$: **prevalence of patterns or textures**

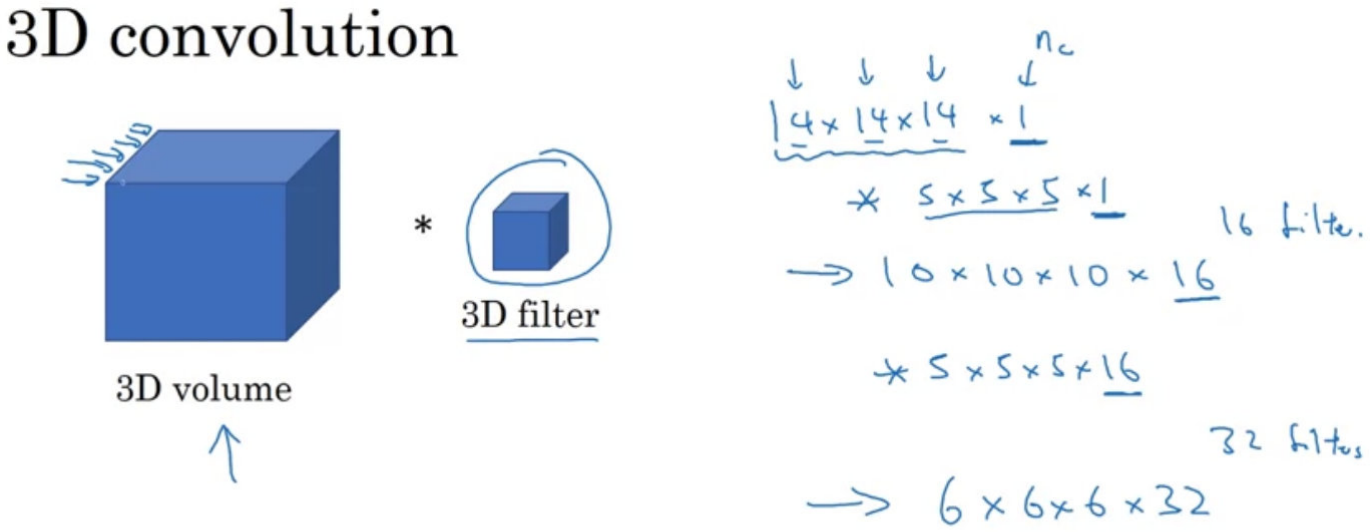
- The diagonal elements $G_{(gram)ii}$ measure how "active" a filter i is.
- For example, suppose filter i is detecting vertical textures in the image. Then $G_{(gram)ii}$ measures how common vertical textures are in the image as a whole.
- If $G_{(gram)ii}$ is large, this means that the image has a lot of vertical texture.

By capturing the prevalence of different types of features ($G_{(gram)ii}$), as well as how much different features occur together ($G_{(gram)ij}$), the Style matrix G_{gram} measures the style of an image.

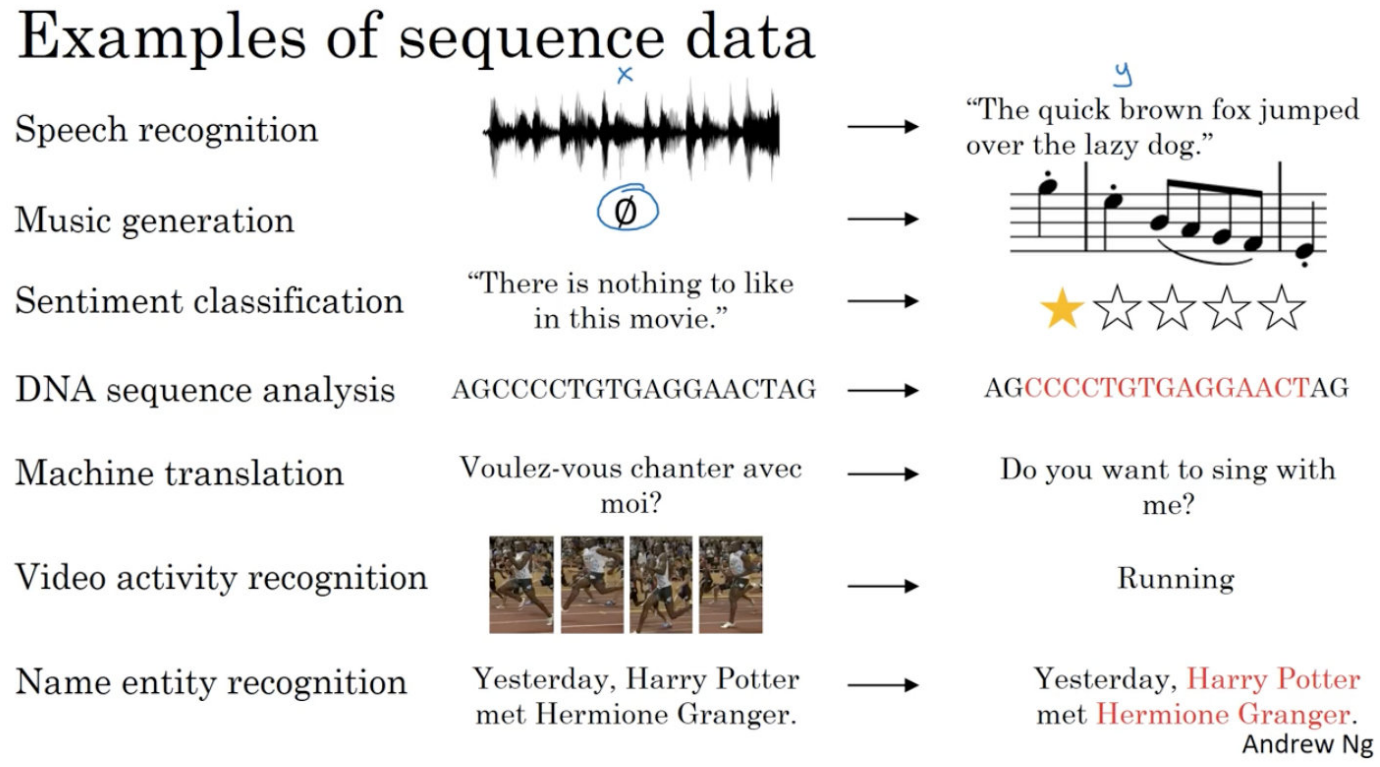
> Neural style transfer is trained as a supervised learning task in which the goal is to input two images (x *x*), and train a network to output a new, synthesized image (y *y*).
>
> `FALSE:` **Neural style transfer is about training on the pixels of an image to make it look artistic, it is `not learning` any parameters.**

1D and 3D Generalizations





Sequence Model



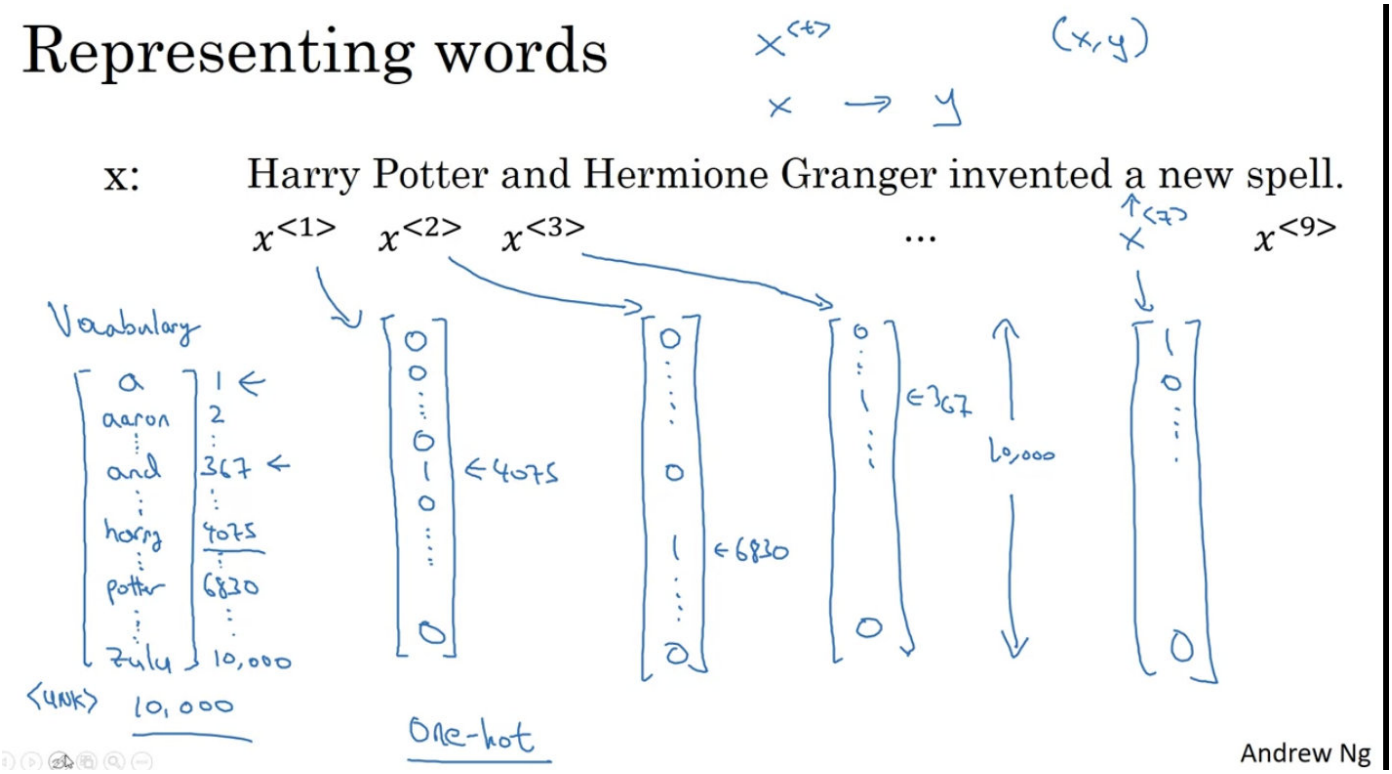
Notation

- Superscript $[l]$ denotes an object associated with the l^{th} layer.
- Superscript (i) denotes an object associated with the i^{th} example.
- Superscript $\langle t \rangle$ denotes an object at the t^{th} time step.
- Subscript i denotes the i^{th} entry of a vector.

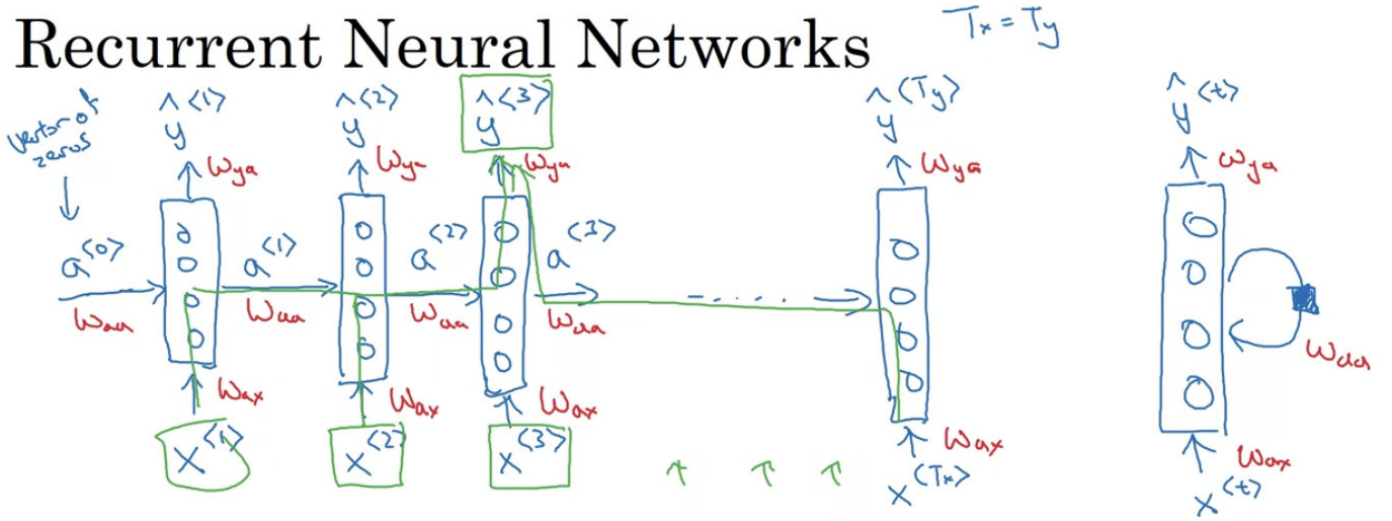
- $T_x^{(i)}$: the input sequence length for training example i.
- $T_y^{(i)}$: the output sequence length for training example i.

Example:

- $a_5^{(2)[3]<4>}$ denotes the activation of the 2nd training example (2), 3rd layer [3], 4th time step <4>, and 5th entry in the vector.



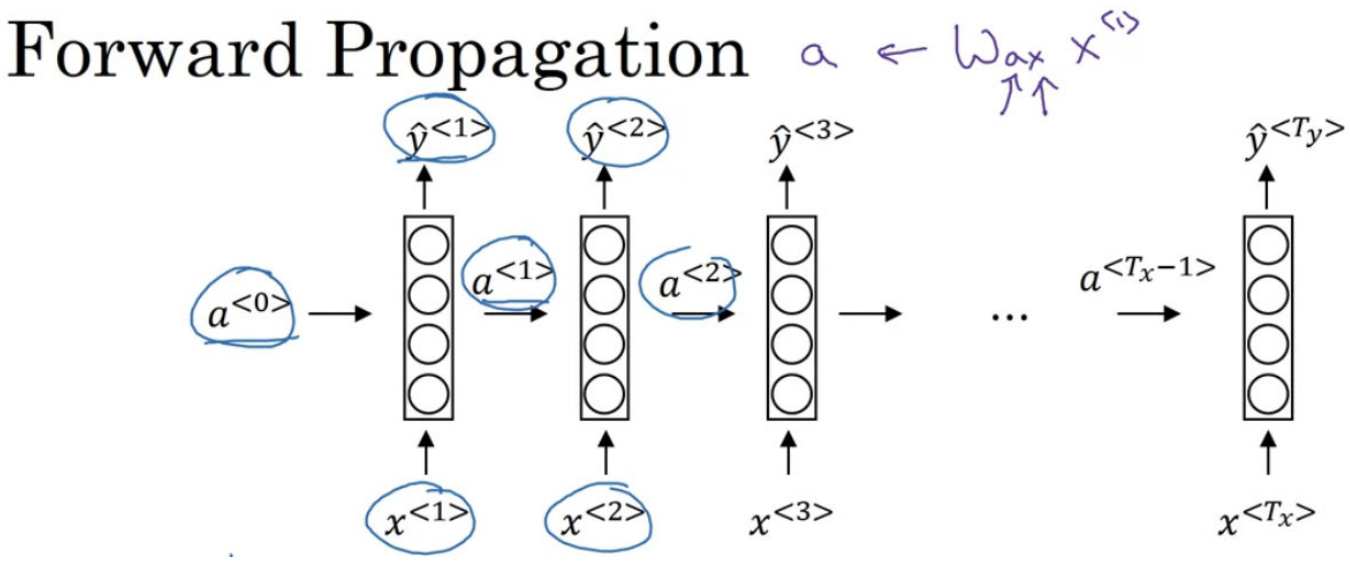
Recurrent Neural Network Model



When making the prediction for y_3, it gets the information **not only from x_3 but also the information from x_1 and x_2.**

One weakness of RNN is that it only uses the information that is **earlier** in the sequence to male prediction.

Forward Propagation



Initialize $a^{<0>} = \vec{0}$.

Then,

$$a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a) \leftarrow \tanh(/\text{ReLU})$$
$$\hat{y}^{<1>} = g(W_{ya}a^{<1>} + b_y) \leftarrow \text{depends on tasks}$$

Generally,

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$
$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

Simplified RNN notation

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$
$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

where

$$W_a = [\,W_{aa}|W_{ax}\,]$$
$$[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$$

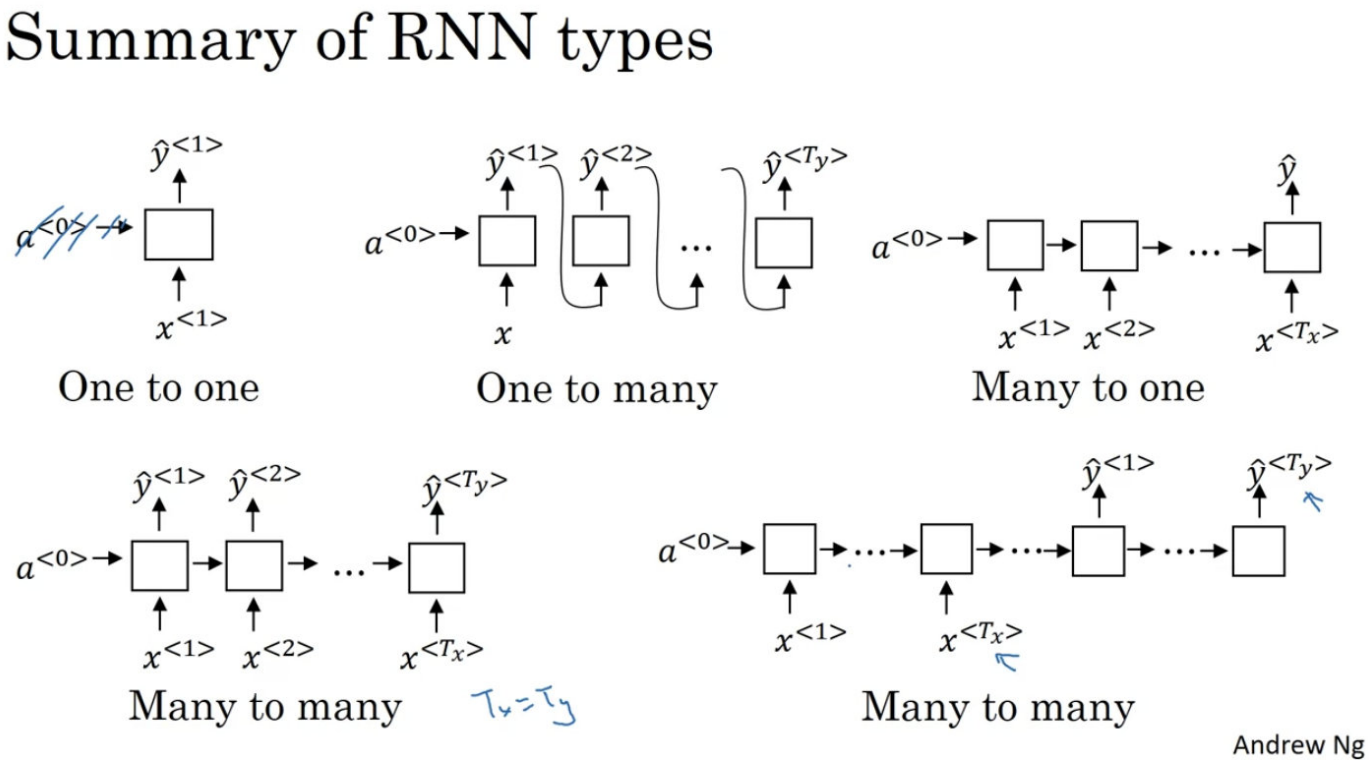
So that,

$$[\,W_{aa}|W_{ax}\,] \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} = W_{aa}a^{<t-1>} + W_{ax}x^{<t>}$$

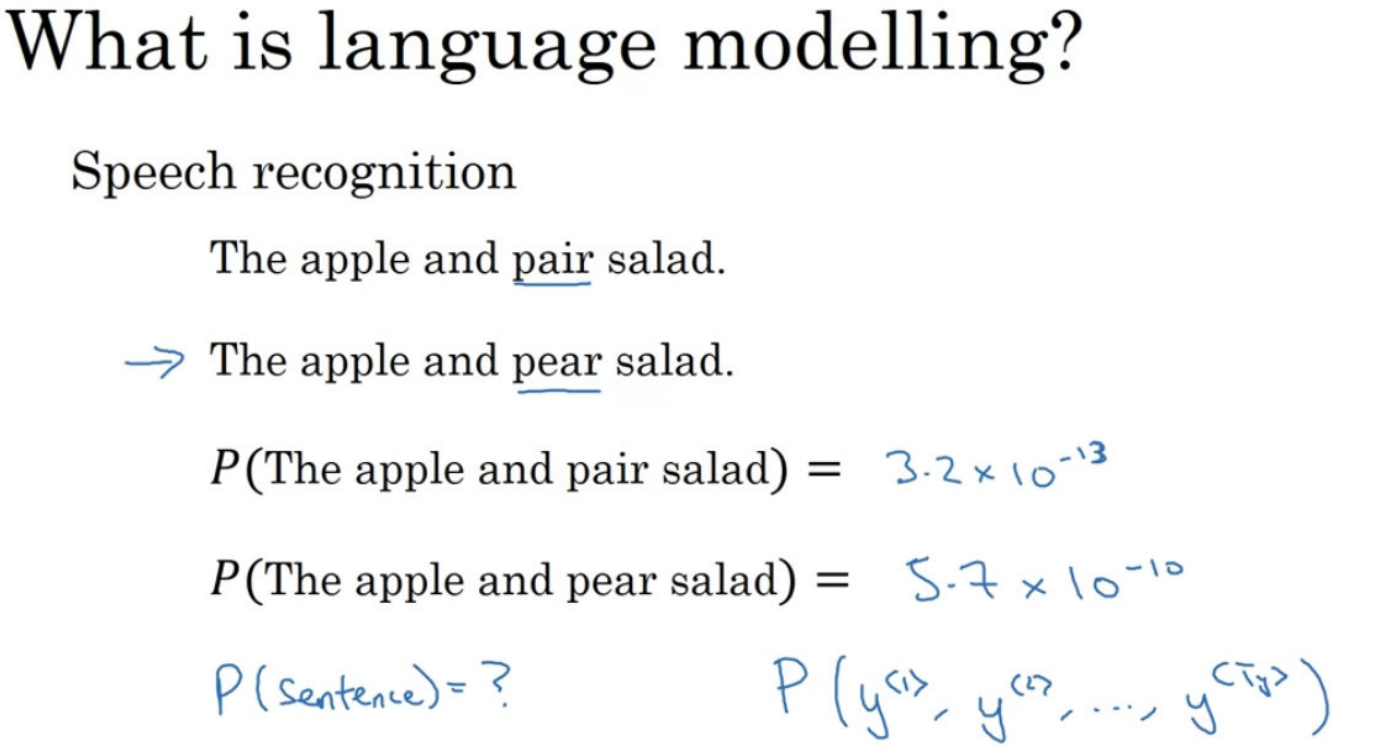
Backpropagation Through Time

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>}\log\hat{y}^{<t>} - (1 - y^{<y>})\log(1 - \hat{y}^{<t>})$$
$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Different Types of RNNs



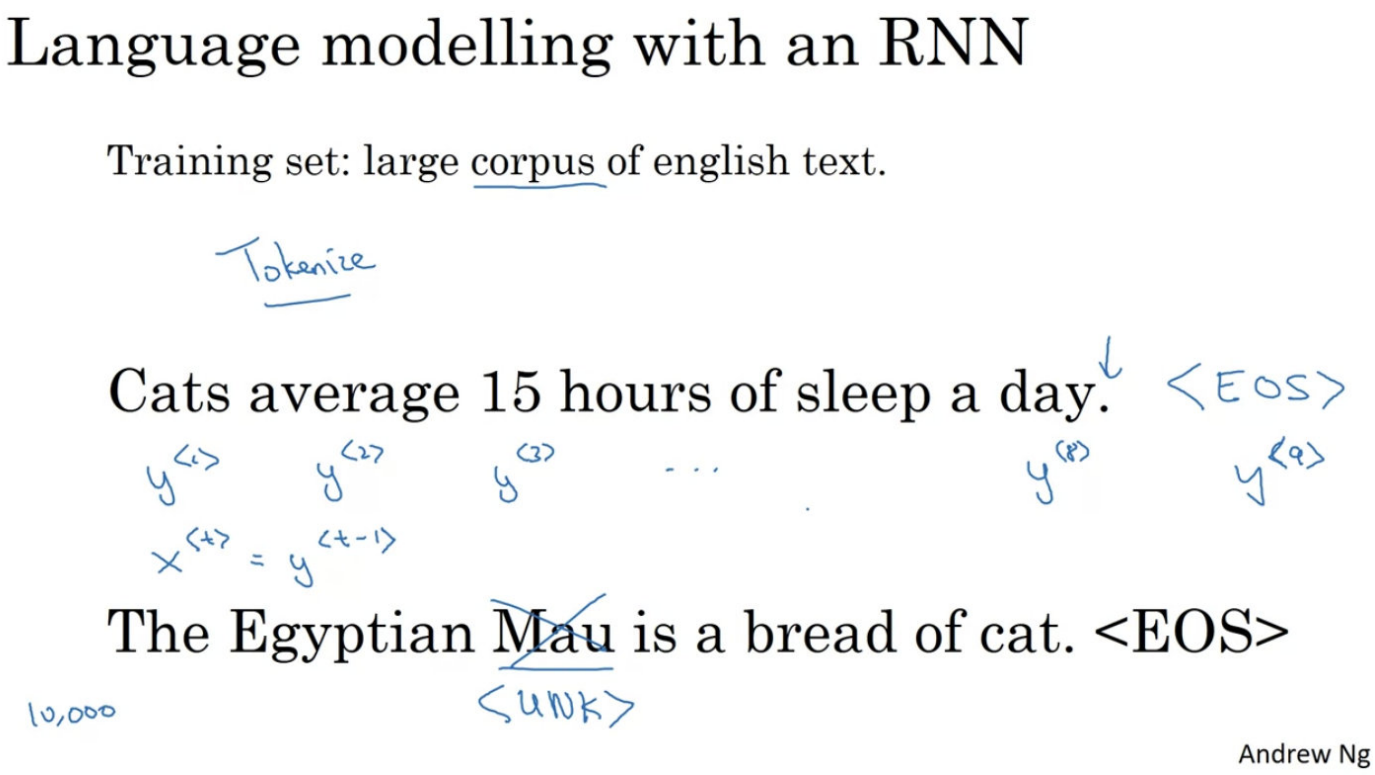
Language Model and Sequence Generation



`The probability of sentence:` the chance that the next sentence you read somewhere out there in the world will be the particular sentence.

The language model represent a sentences as `outputs y` rather than `inputs x`.

The **basic job** of a language model is to estimate the probability of the particular input sequences of words.

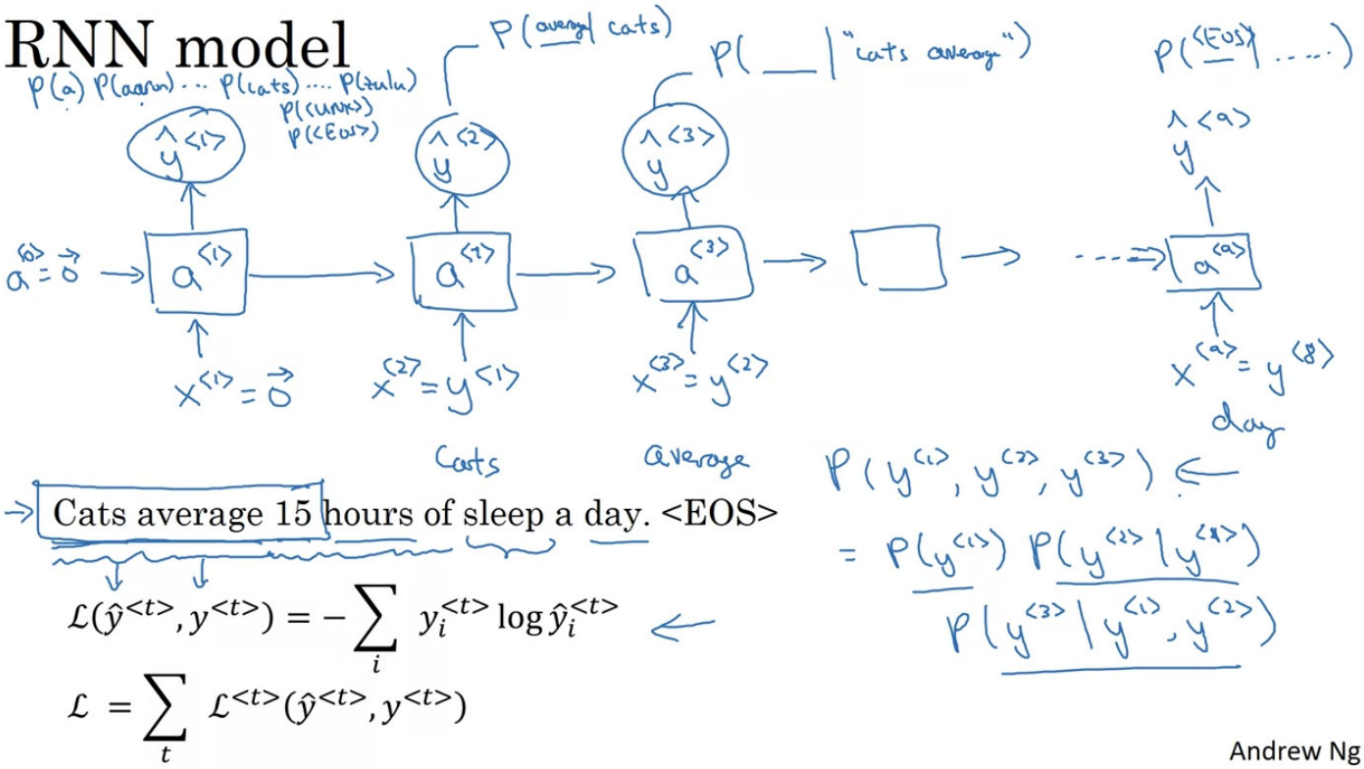


`Corpus:` an NLP terminology that just means a large body or a large set of sentences.

`Tokenize:` form a vocabulary and map each of words in the sentence to **one-hot vectors** or **indicies** in the vocabulary. By tokenization step, you can decide whether or not the **punctuation** should be token as well.

`<EOS>:` "End Of Sentence", an **optional** extra token that can help to figure out when a sentence ends.

`<UNK>:` a unique token stands for **unknown** words.



Each step in the RNN will look at some set of preceding words, such as given the first three words, what is the distribution over the next word? So this learns to predict **one word at a time** going from left to right.

The loss function:

$$L(\hat{y}^{<t>}, y^{<t>}) = -\sum y_i^{<t>} \log \hat{y}_i^{<t>} \leftarrow \text{softmax loss}$$
$$L = \sum_t L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Given a new sentence $(y^{<1>}, y^{<2>}, y^{<3>})$, the chance of this entire sentence would be:

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

Implement RNN

Dimensions of input x

Input with n_x number of units

- For a single time step of a single input example, $x^{(i)\langle t \rangle}$ is a one-dimensional input vector
- Using language as an example, a language with a 5000-word vocabulary could be one-hot encoded into a vector that has 5000 units. So $x^{(i)\langle t \rangle}$ would have the shape (5000,)
- The notation n_x is used here to denote the number of units in a single time step of a single training example

Time steps of size T_x

- A recurrent neural network has multiple time steps, which you'll index with t.
- In the lessons, you saw a single training example $x^{(i)}$ consisting of multiple time steps T_x. In this notebook, T_x will denote the number of timesteps in the longest sequence.

Batches of size m

- Let's say we have mini-batches, each with 20 training examples
- To benefit from vectorization, you'll stack 20 columns of $x^{(i)}$ examples
- For example, this tensor has the shape (5000,20,10)
- You'll use m to denote the number of training examples
- So, the shape of a mini-batch is (n_x, m, T_x)

Definition of hidden state a

- The activation $a^{\langle t \rangle}$ that is passed to the RNN from one time step to another is called a "hidden state."

Dimensions of hidden state a

- Similar to the input tensor x, the hidden state for a single training example is a vector of length n_a
- If you include a mini-batch of m training examples, the shape of a mini-batch is (n_a, m)
- When you include the time step dimension, the shape of the hidden state is (n_a, m, T_x)
- You'll loop through the time steps with index t, and work with a 2D slice of the 3D tensor
- This 2D slice is referred to as $a^{\langle t \rangle}$

Dimensions of prediction $\hat{y}$

- Similar to the inputs and hidden states, $\hat{y}$ is a 3D tensor of shape (n_y, m, T_y)
  - n_y: number of units in the vector representing the prediction
  - m: number of examples in a mini-batch
  - T_y: number of time steps in the prediction
- For a single time step t, a 2D slice $\hat{y}^{\langle t \rangle}$ has shape (n_y, m)

Steps:

1. Implement the calculations needed for one time step of the RNN.
2. Implement a loop over T_x time steps in order to process all the inputs, one at a time.

RNN Cell

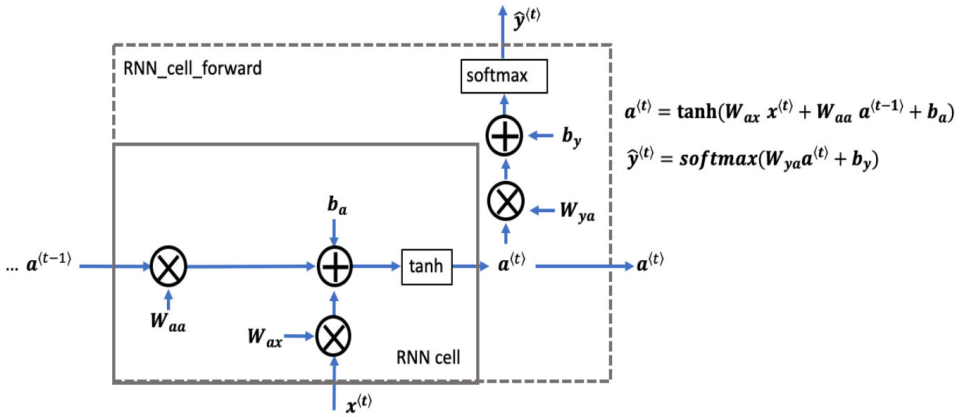


Figure 2: Basic RNN cell. Takes as input $x^{\langle t \rangle}$ (current input) and $a^{\langle t-1 \rangle}$ (previous hidden state containing information from the past), and outputs $a^{\langle t \rangle}$ which is given to the next RNN cell and also used to predict $\hat{y}^{\langle t \rangle}$

`RNN cell` versus **`RNN_cell_forward`**:

- Note that an RNN cell outputs the hidden state $a^{\langle t \rangle}$.
  - `RNN cell` is shown in the figure as the inner box with solid lines
- The function that you'll implement, `rnn_cell_forward`, also calculates the prediction $\hat{y}^{\langle t \rangle}$
  - `RNN_cell_forward` is shown in the figure as the outer box with dashed lines

RNN Forward Pass

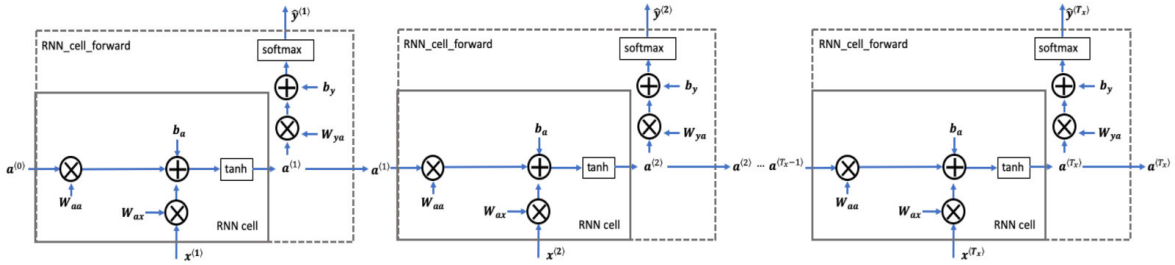


Figure 3: Basic RNN. The input sequence $x = (x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots, x^{\langle T_x \rangle})$ is carried over T_x time steps. The network outputs $y = (y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \ldots, y^{\langle T_x \rangle})$.

- A recurrent neural network (RNN) is a repetition of the RNN cell.
  - If your input sequence of data is 10 time steps long, then you will re-use the RNN cell 10 times
- Each cell takes two inputs at each time step:

    - $a^{\langle t-1 \rangle}$: The hidden state from the previous cell
    - $x^{\langle t \rangle}$: The current time step's input data
- It has two outputs at each time step:
    - A hidden state ($a^{\langle t \rangle}$)
    - A prediction ($y^{\langle t \rangle}$)
- The weights and biases $(W_{aa}, b_a, W_{ax}, b_x)$ are re-used each time step
    - They are maintained between calls to `rnn_cell_forward` in the 'parameters' dictionary

Instructions:

- Create a 3D array of zeros, a of shape (n_a, m, T_x) that will store all the hidden states computed by the RNN
- Create a 3D array of zeros, $\hat{y}$, of shape (n_y, m, T_x) that will store the predictions
    - Note that in this case, $T_y = T_x$ (the prediction and input have the same number of time steps)
- Initialize the 2D hidden state `a_next` by setting it equal to the initial hidden state, a_0
- At each time step t:
    - Get $x^{\langle t \rangle}$, which is a 2D slice of x for a single time step t
        - $x^{\langle t \rangle}$ has shape (n_x, m)
        - x has shape (n_x, m, T_x)
    - Update the 2D hidden state $a^{\langle t \rangle}$, the prediction $\hat{y}^{\langle t \rangle}$ and the cache by running `rnn_cell_forward`
        - $a^{\langle t \rangle}$ has shape (n_a, m)
    - Store the 2D hidden state in the 3D tensor a, at the t^{th} position
        - a has shape (n_a, m, T_x)
    - Store the 2D $\hat{y}^{\langle t \rangle}$ prediction in the 3D tensor $\hat{y}_{pred}$ at the t^{th} position
        - $\hat{y}^{\langle t \rangle}$ has shape (n_y, m)
        - $\hat{y}$ has shape (n_y, m, T_x)
    - Append the cache to the list of caches
- Return the 3D tensor a and $\hat{y}$, as well as the list of caches

What you should remember:

- The recurrent neural network, or RNN, is essentially the repeated use of a single cell.
- A basic RNN reads inputs one at a time, and remembers information through the hidden layer activations (hidden states) that are passed from one time step to the next.
    - The time step dimension determines how many times to re-use the RNN cell
- Each cell takes two inputs at each time step:
    - The hidden state from the previous cell
    - The current time step's input data
- Each cell has two outputs at each time step:
    - A hidden state
    - A prediction

Basic RNN Backward Pass

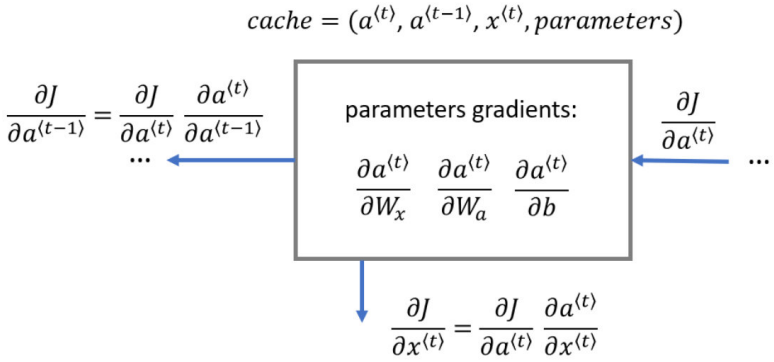


Figure 6: The RNN cell's backward pass. Just like in a fully-connected neural network, the derivative of the cost function J backpropagates through the time steps of the RNN by following the chain rule from calculus. Internal to the cell, the chain rule is also used to calculate $(\frac{\partial J}{\partial W_{ax}}, \frac{\partial J}{\partial W_{aa}}, \frac{\partial J}{\partial b})$ to update the parameters (W_{ax}, W_{aa}, b_a). The operation can utilize the cached results from the forward path.

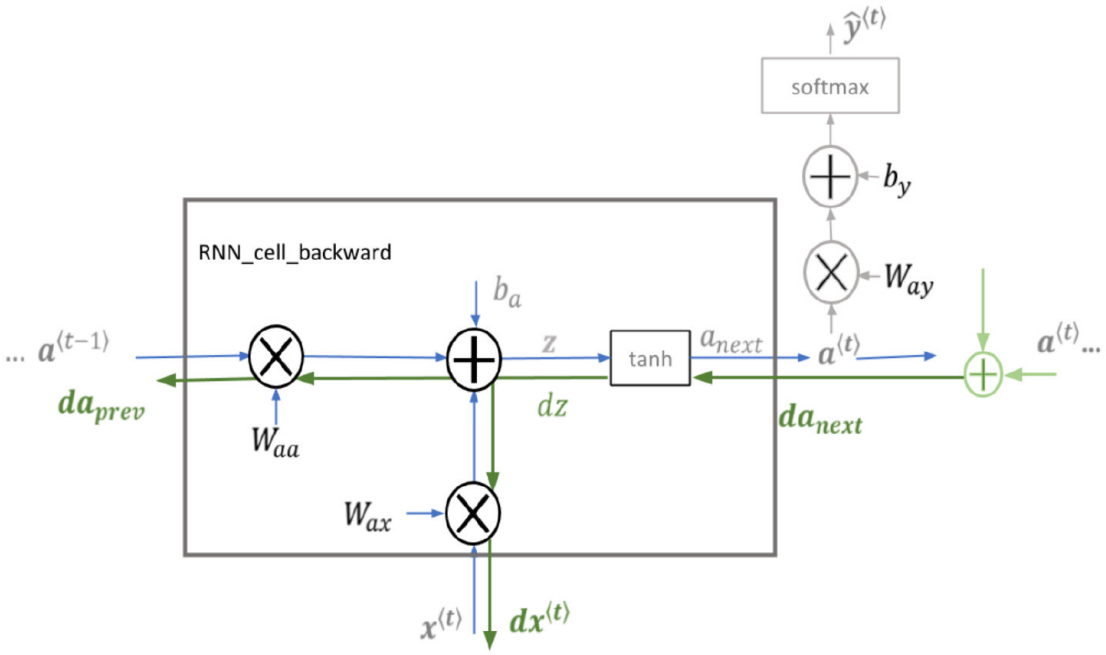


Figure 7: This implementation of `rnn_cell_backward` does **not** include the output dense layer and softmax which are included in `rnn_cell_forward`.

da_{next} is $\frac{\partial J}{\partial a^{\langle t \rangle}}$ and includes loss from previous stages and current stage output logic. The addition shown in green will be part of your implementation of `rnn_backward`.

Equations

To compute `rnn_cell_backward`, you can use the following equations. Here, $*$ denotes element-wise multiplication while the absence of a symbol indicates matrix multiplication.

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

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x) \tag{-}$$

$$dtanh = da_{next} * (1 - \tanh^2(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a)) \tag{0}$$

$$dW_{ax} = dtanh \cdot x^{\langle t \rangle T} \tag{1}$$

$$dW_{aa} = dtanh \cdot a^{\langle t-1 \rangle T} \tag{2}$$

$$db_a = \sum_{batch} dtanh \tag{3}$$

$$dx^{\langle t \rangle} = {W_{ax}}^T \cdot dtanh \tag{4}$$

$$da_{prev} = {W_{aa}}^T \cdot dtanh \tag{5}$$

$$a^{\langle t \rangle} = \tanh(\underbrace{W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a}_{z}) \qquad a^{\langle t+1 \rangle} = \tanh(z^{\langle t \rangle})$$

element-wise

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

Equation (0) $d\tanh = da_{next} * (1 - \tanh^2(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a))$

$$d\tanh = dz = \frac{dJ}{da^{\langle t+1 \rangle}} \cdot \frac{da^{\langle t+1 \rangle}}{dz} = da^{\langle t+1 \rangle} * (1 - \tanh^2(z))$$

$$= - - \text{——} \quad - \quad \text{——}$$

Equation (1) $dW_{ax} = d\tanh \cdot x^{\langle t \rangle T}$ Matrix Cookbook

$$dW_{ax} = \frac{dJ}{dz} \cdot \frac{dz}{dW_{ax}} = d\tanh \cdot x^{\langle t \rangle T}$$

Equation (2) $dW_{aa} = d\tanh \cdot a^{\langle t-1 \rangle T}$

$$dW_{aa} = \frac{dJ}{dz} \cdot \frac{dz}{dW_{aa}} = d\tanh \cdot a^{\langle t-1 \rangle T}$$

Equation (3) $db_a = \sum_{batch} d\tanh$

$$db_a = \frac{dJ}{dz} \frac{dz}{db_a} = \sum_{batch} dz = \sum_{batch} d\tanh$$

vectorization, $\frac{1}{m}$ is omitted

$db^{(i)} = dz^{(i)}$ sum over

Equation (4) $dx^{\langle t \rangle} = W_{ax}^T \cdot d\tanh$

$$dx^{\langle t \rangle} = \frac{dJ}{dz} \frac{dz}{dW_{ax}} = W_{ax}^T \cdot d\tanh$$

order? Transpose?

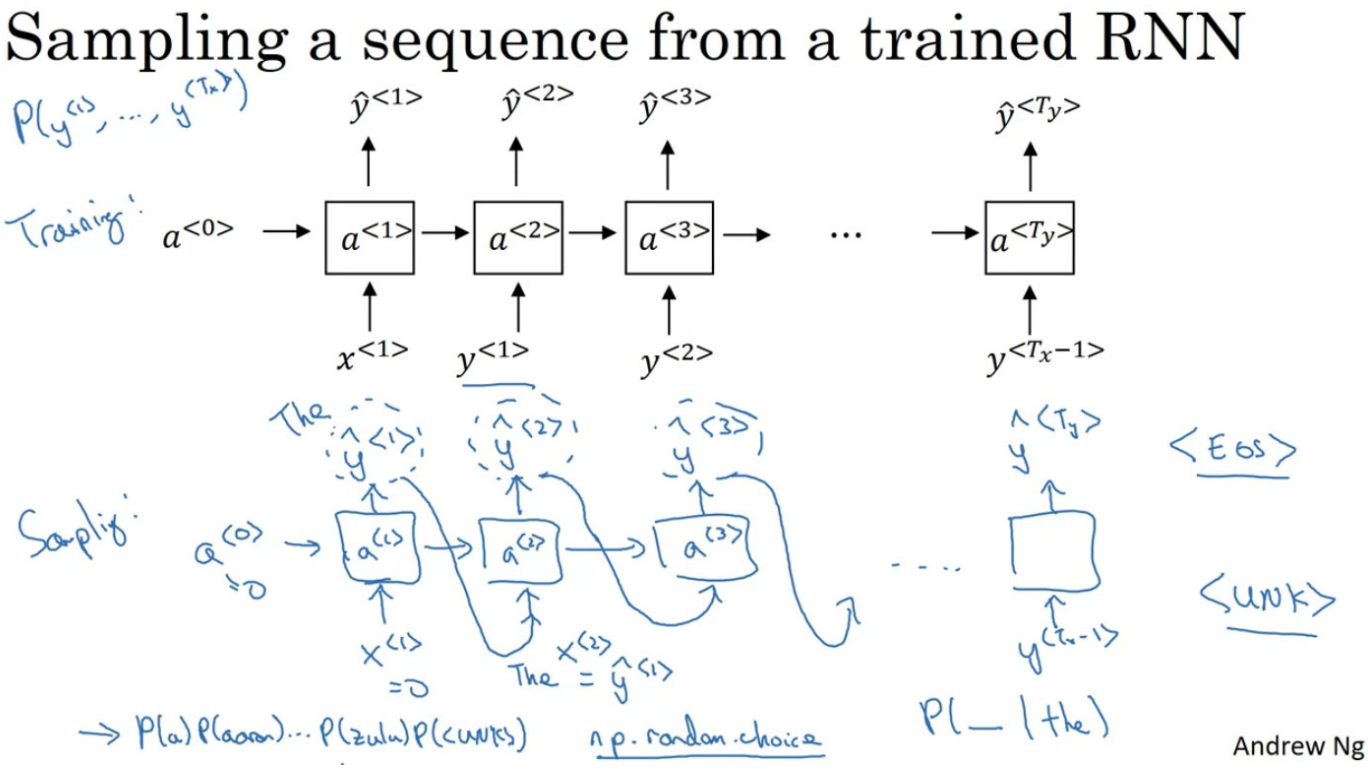
Equation (5) $da_{prev} = W_{aa}^T \cdot d\tanh$

$$da_{prev} = \frac{dJ}{dz} \cdot \frac{dz}{dW_{aa}} = W_{aa}^T \cdot d\tanh$$

Check note about Matrix vector derivatives by CS231n:

vecDerivs.pdf

Sampling Novel Sequences



what you want to do:

1. Randomly sample first word $\hat{y}^{<1>}$ according to the first softmax distribution.

```
np.random.choice() # To sample according to distribution
```

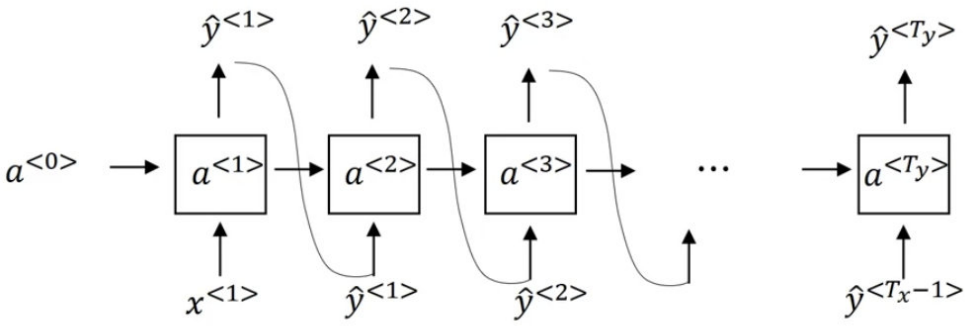
2. Going to next time step, take the $\hat{y}^{<1>}$ which is just sampled as the input to the next time step and sample $\hat{y}^{<2>}$.
3. Repeat 1 $\rightarrow$ 2 until get the last time step (such as when `<EOS>` token is sampled or just decide how many words to sample).

Character-level language model

→ Vocabulary = [a, aaron, …, zulu, <UNK>] ←

→ Vocabulary = [a, b, c, … , z, ␣, ., ,, ;, 0, …, 9, A, …, Z]

$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ $y^{<4>}$ Cat ↑ ↑ ↑ ↑ average … Mau



Andrew Ng

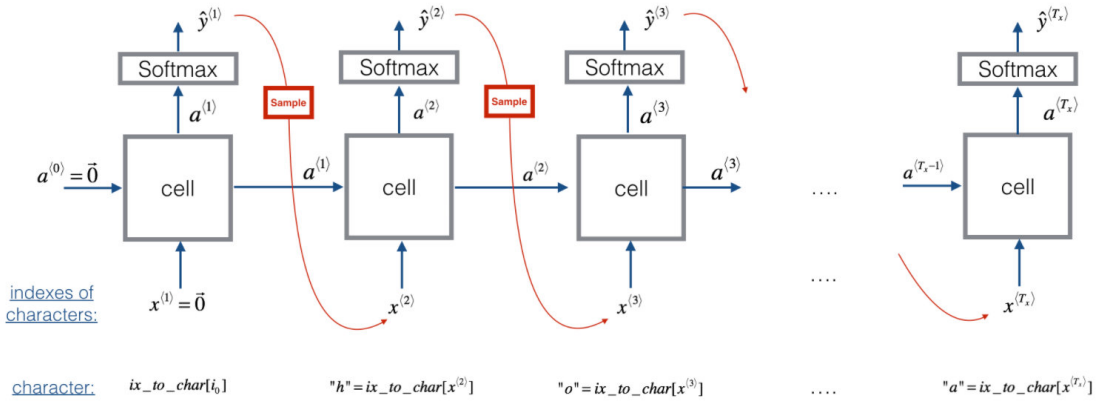
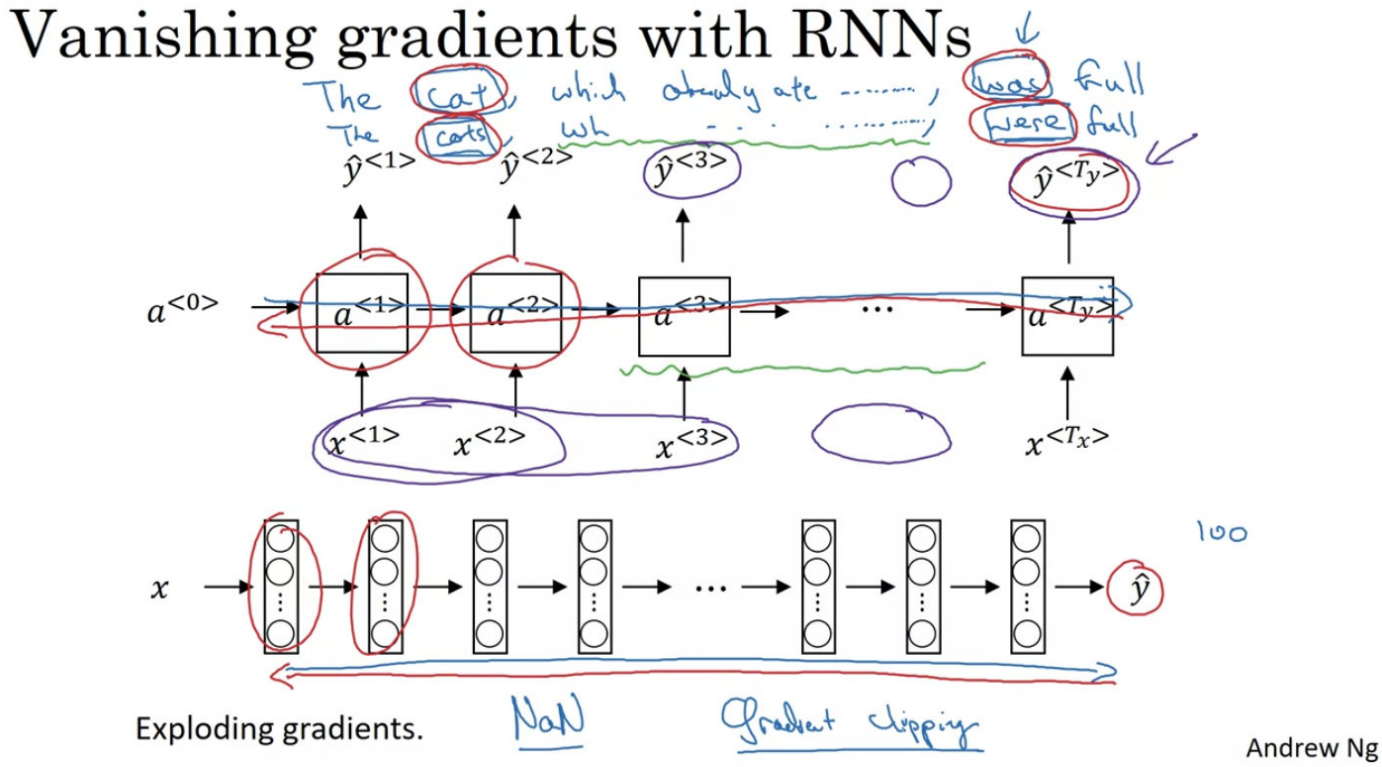


Figure 3: In this picture, you can assume the model is already trained. You pass in $x^{\langle 1 \rangle} = \vec{0}$ at the first time-step, and have the network sample one character at a time.

Pros and cons of Character-level language model

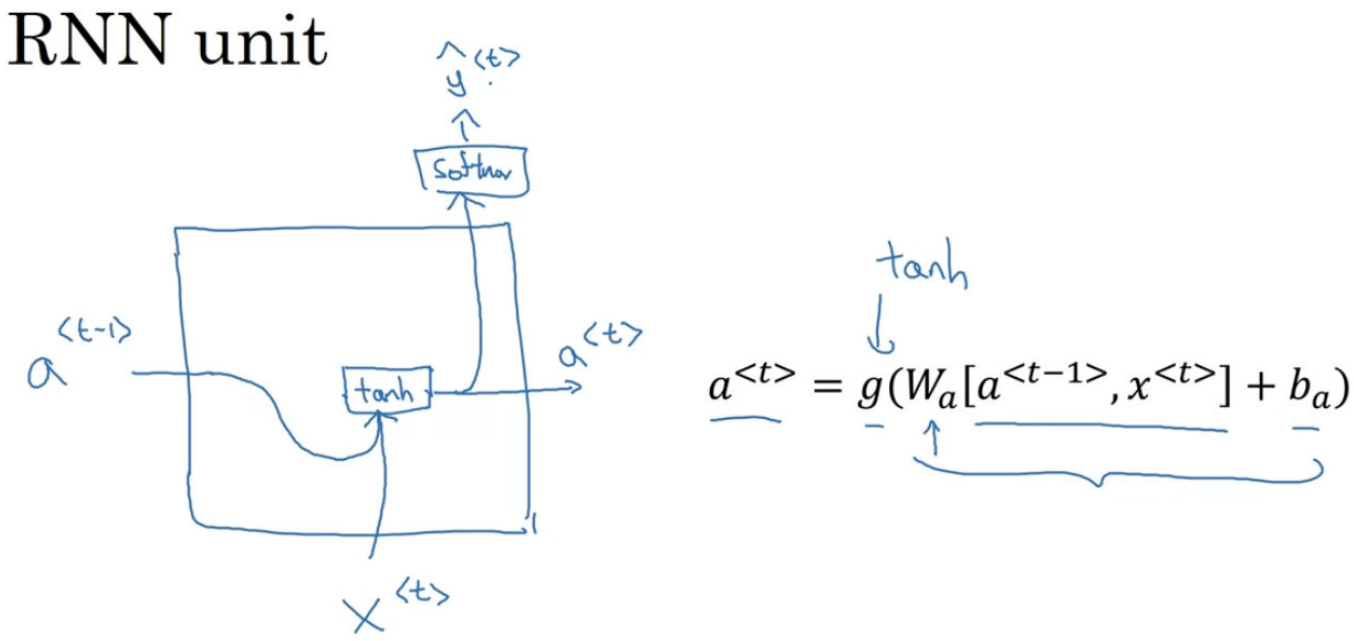
1. Don't have to worry about unknown word tokens.
2. End up with much longer sequences (computational expensive).

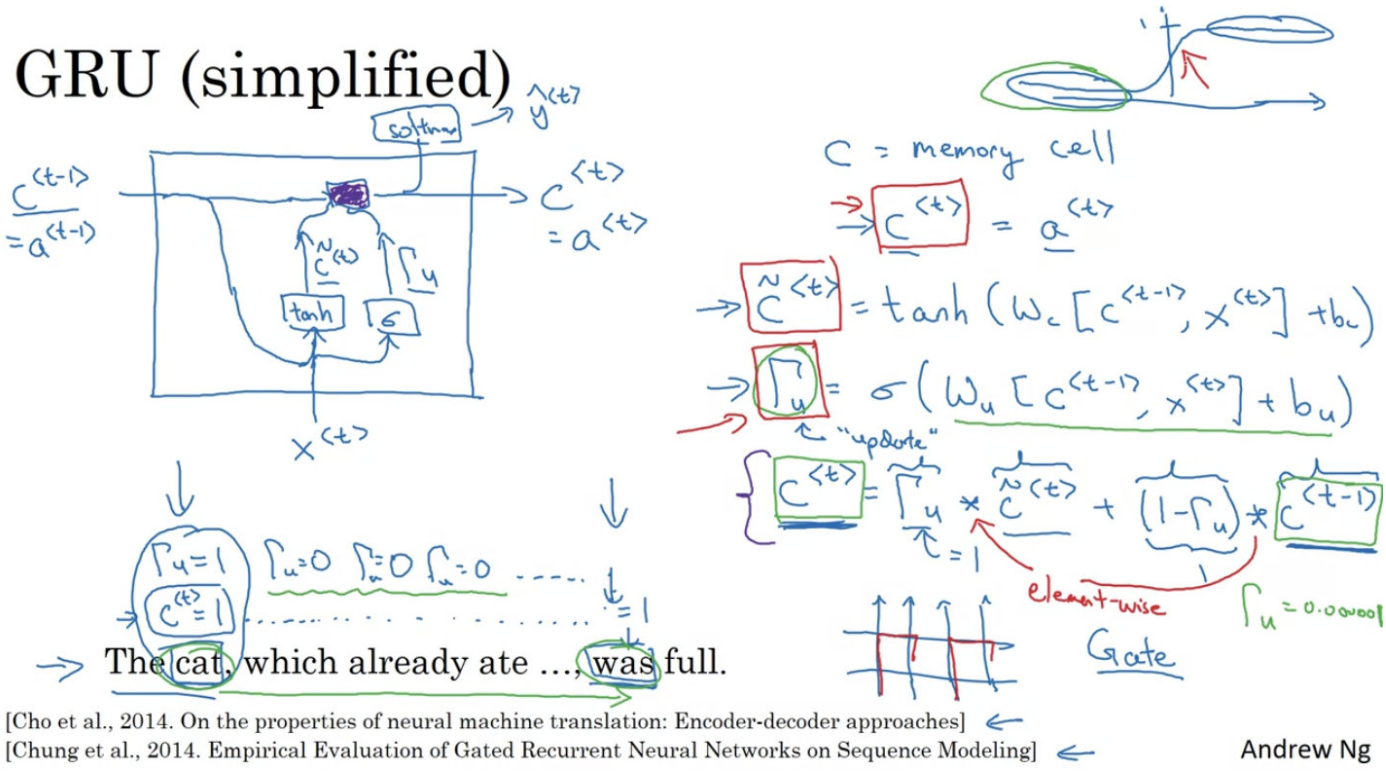
Vanishing Gradients with RNNs



Because of vanishing gradients, the basic RNN model has many **local influences**, meaning that the output $y^{<t>}$ is mainly influenced by values close to $y^{<t>}$, so is difficult for it to be strongly influenced by an input that was **very early** in the sequence.

Gated Recurrent Unit (GRU)





`c:memory cell` will provide a bit of to remember the important information.

The memory cell will have some value $c^{<t>}$ at time t. The GRU will out put an activation value $a^{<t>}$ that is equal to $c^{<t>}$.

$$c^{<t>} = a^{<t>}$$

At every time step, we are going to **consider** overwritting the memory cell with a `candidate` $\tilde{c}^{<t>}$ for replacing $c^{<t>}$.

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

The key idea of GRU is to have a `update gate`

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

The job of gate Γ_u is to decide when do you update memory cell.

The specific equation for GRU is

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

Because the gate is easliy to be set to 0, it is cery good at **maintaining** the value for the cell. And because the gate can be so close to 0, it does not suffer from vanishing gradient problems.

Full GRU

Full GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

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

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

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

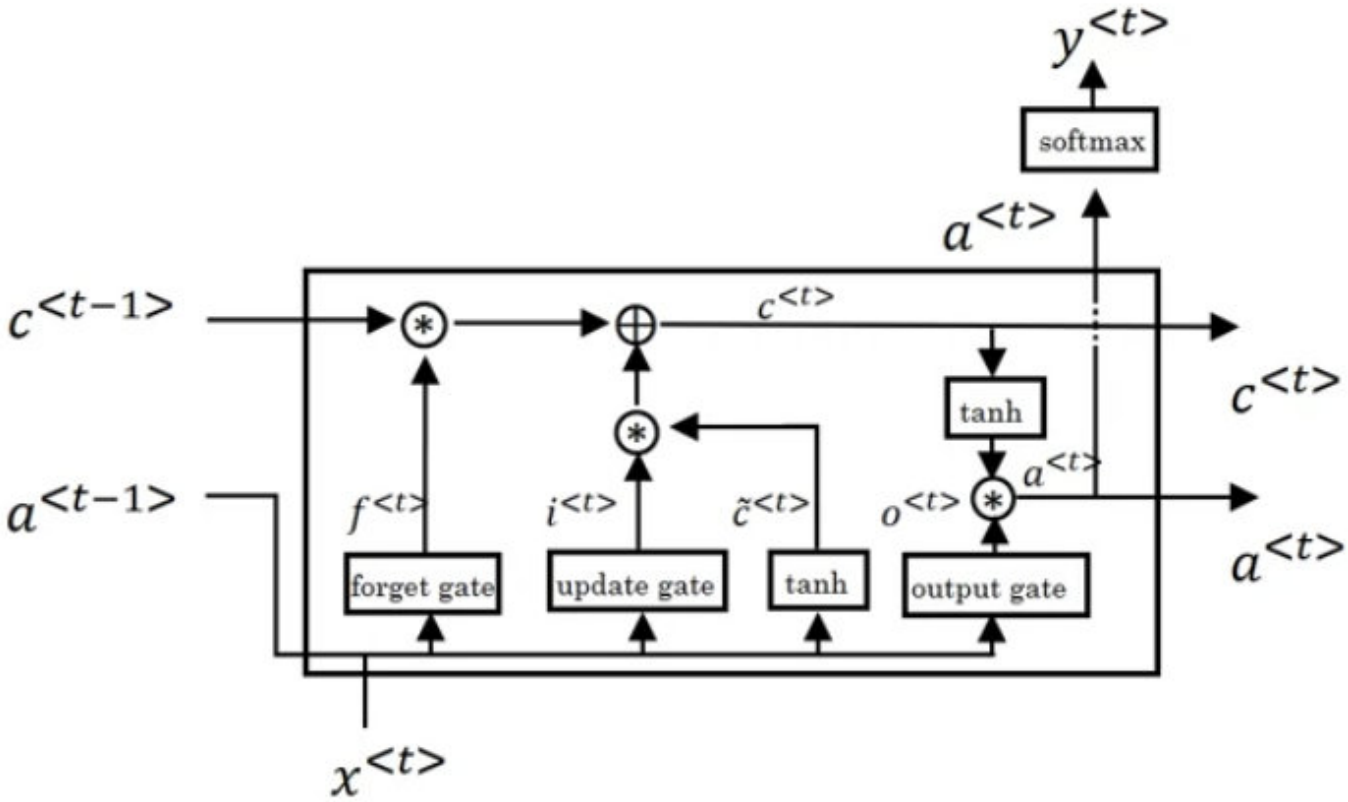
LSTM

The cat, which ate already, was full.

For full GRU, there is an additional gate Γ_r.

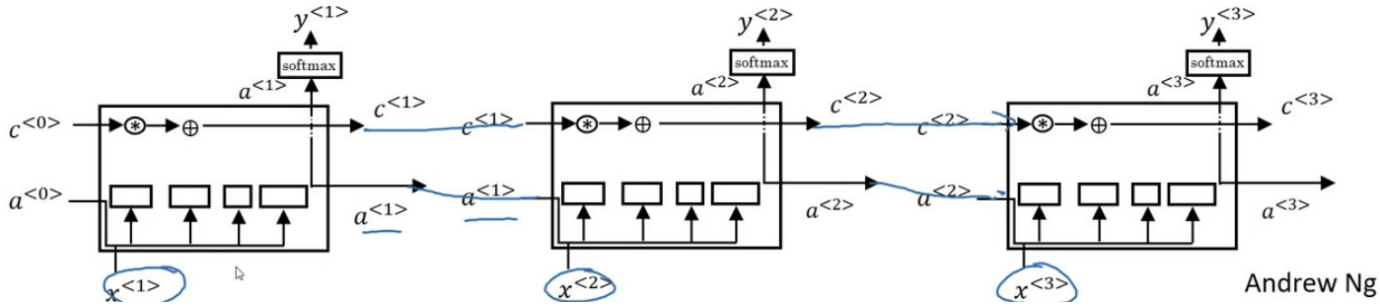
$$\begin{aligned}
a^{<t>} &= c^{<t>} \\
\tilde{c}^{<t>} &= \tanh(W_c[\Gamma_r * c^{<t-1>}, x^t] + b_c) \\
\Gamma_r &= \sigma(W_r[c^{<t-1>}, x^t] + b_r) \\
\Gamma_u &= \sigma(W_u[c^{<t-1>}, x^t] + b_u) \\
c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}
\end{aligned}$$

LSTM (Long Short Term Memory) Unit



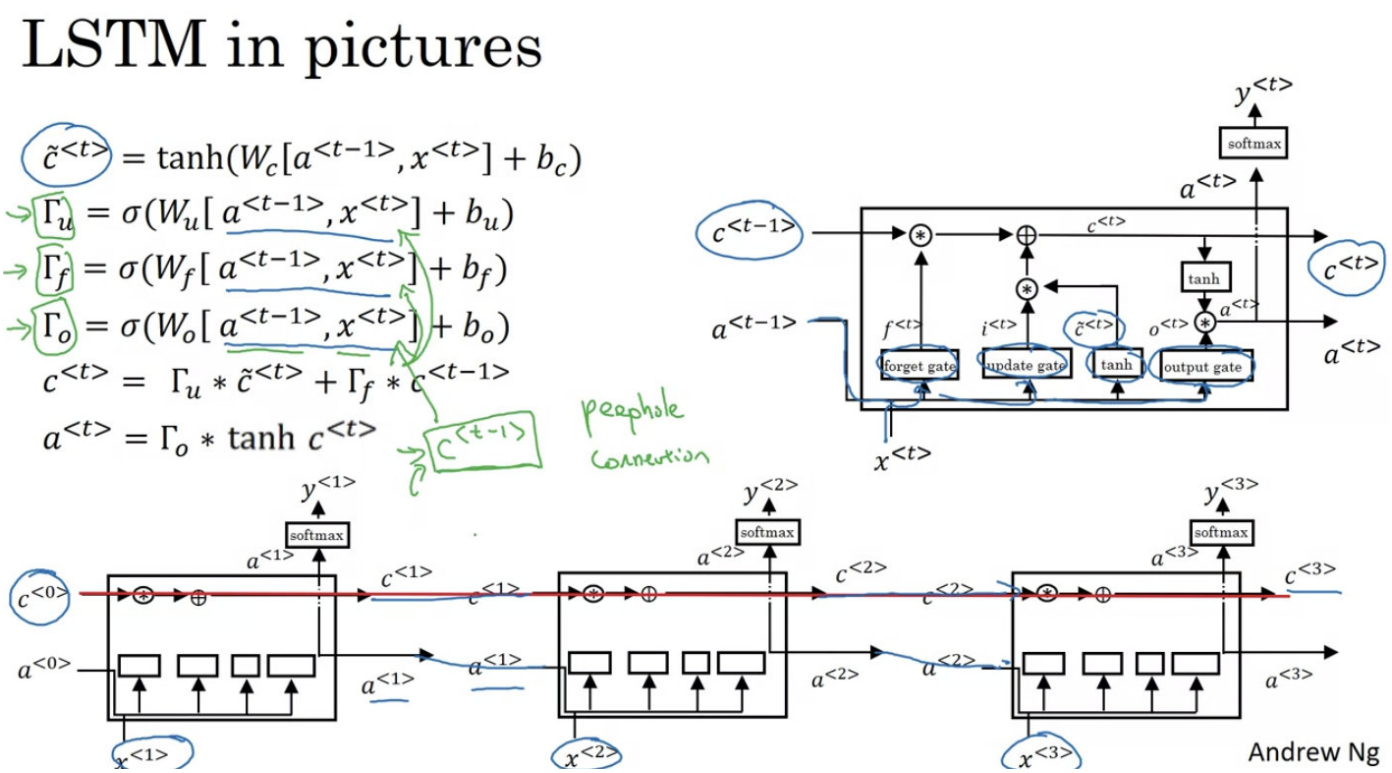
Three gates: `update gate` Γ_u, `forget gate` Γ_f, `output gate` Γ_o

$$
\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_u &= \sigma(W_u[a^{<t-1>}, x^t] + b_u) \leftarrow \text{update} \\
\Gamma_f &= \sigma(W_f[a^{<t-1>}, x^t] + b_f) \leftarrow \text{forget} \\
\Gamma_o &= \sigma(W_o[a^{<t-1>}, x^t] + b_o) \leftarrow \text{output} \\
c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \\
a^{<t>} &= \Gamma_o * \tanh(c^{<t>})
\end{aligned}
$$



Notice that, there is a line at the top shows that so long as you set the **forget** and the **update** gate appropriately, it is **relatively easy** for the LSTM to have some value c_0 to be passed all the way the right to have $c_3 = c_0$. This is why LSTM, as well as the GRU, is very good at memorizing certain values even for a long time.

Peephole LSTM



LSTM and GRU

- The advantage of the GRU is that it is a **simpler** model, so it is actually easier to build a much bigger network.
- LSTM is more **powerful** and more **effective** since it has three gates.

Details of LSTM

The following figure shows the operations of an LSTM cell:

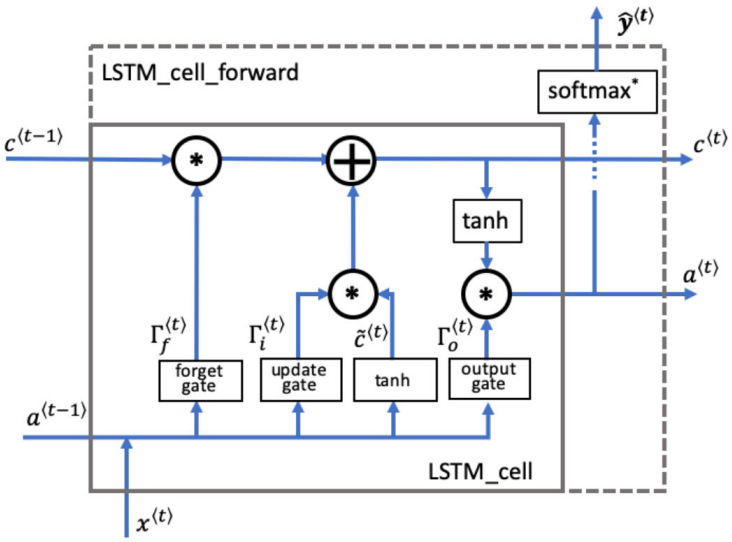


Figure 4: LSTM cell. This tracks and updates a "cell state," or memory variable $c^{\langle t \rangle}$ at every time step, which can be different from $a^{\langle t \rangle}$. Note, the $softmax$ includes a dense layer and softmax.

Forget gate $\mathbf{\Gamma}_f$

- Let's assume you are reading words in a piece of text, and plan to use an LSTM to keep track of grammatical structures, such as whether the subject is singular ("puppy") or plural ("puppies").
- If the subject changes its state (from a singular word to a plural word), the memory of the previous state becomes outdated, so you'll "forget" that outdated state.
- The "forget gate" is a tensor containing values between 0 and 1.
    - If a unit in the forget gate has a value close to 0, the LSTM will "forget" the stored state in the corresponding unit of the previous cell state.
    - If a unit in the forget gate has a value close to 1, the LSTM will mostly remember the corresponding value in the stored state.

Equation

$$\mathbf{\Gamma}_f^{\langle t \rangle} = \sigma(\mathbf{W}_f[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_f) \tag{1}$$

Explanation of the equation:

- $\mathbf{W_f}$ contains weights that govern the forget gate's behavior.
- The previous time step's hidden state $[a^{\langle t-1 \rangle}$ and current time step's input $x^{\langle t \rangle}]$ are concatenated together and multiplied by $\mathbf{W_f}$.
- A sigmoid function is used to make each of the gate tensor's values $\mathbf{\Gamma}_f^{\langle t \rangle}$ range from 0 to 1.
- The forget gate $\mathbf{\Gamma}_f^{\langle t \rangle}$ has the same dimensions as the previous cell state $c^{\langle t-1 \rangle}$.
- This means that the two can be multiplied together, element-wise.
- Multiplying the tensors $\mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle}$ is like applying a mask over the previous cell state.
- If a single value in $\mathbf{\Gamma}_f^{\langle t \rangle}$ is 0 or close to 0, then the product is close to 0.
    - This **keeps** the information stored in the corresponding unit in $\mathbf{c}^{\langle t-1 \rangle}$ **from** being remembered for the next time step. `forget`
- Similarly, if one value is close to 1, the product is close to the original value in the previous cell state.
    - The LSTM will **keep** the information from the corresponding unit of $\mathbf{c}^{\langle t-1 \rangle}$, to be used in the next time step. `remember`

Candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$

- The candidate value is a tensor containing information from the **current** time step that **may** be stored in the current cell state $\mathbf{c}^{\langle t \rangle}$.
- The parts of the candidate value that get passed on depend on the update gate.
- The candidate value is a tensor containing values that range from -1 to 1.
- The tilde "~" is used to differentiate the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ from the cell state $\mathbf{c}^{\langle t \rangle}$.

Equation

$$\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\Big(\mathbf{W}_c[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_c\Big) \tag{3}$$

Explanation of the equation

- The $\tanh$ function produces values between -1 and 1.

Update gate $\mathbf{\Gamma}_i$

- You use the update gate to decide what aspects of the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ to add to the cell state $c^{\langle t \rangle}$.
- The update gate decides what parts of a "candidate" tensor $\tilde{\mathbf{c}}^{\langle t \rangle}$ are passed onto the cell state $\mathbf{c}^{\langle t \rangle}$.
- The update gate is a tensor containing values between 0 and 1.
    - When a unit in the update gate is close to 1, it allows the value of the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ to be passed onto the hidden state $\mathbf{c}^{\langle t \rangle}$
    - When a unit in the update gate is close to 0, it prevents the corresponding value in the candidate from being passed onto the hidden state.
- Notice that the subscript **"i" is used and not "u"**, to follow the convention used in the literature.

Equation

$$\mathbf{\Gamma}_i^{\langle t \rangle} = \sigma(\mathbf{W}_i[a^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_i) \tag{2}$$

Explanation of the equation

- Similar to the forget gate, here $\mathbf{\Gamma}_i^{\langle t \rangle}$, the sigmoid produces values between 0 and 1.
- The update gate is multiplied element-wise with the candidate, and this product ($\mathbf{\Gamma}_i^{\langle t \rangle} * \tilde{c}^{\langle t \rangle}$) is used in determining the cell state $\mathbf{c}^{\langle t \rangle}$.

Cell state $\mathbf{c}^{\langle t \rangle}$

- The cell state is the "memory" that gets passed onto future time steps.
- The new cell state $\mathbf{c}^{\langle t \rangle}$ is a **combination** of the previous cell state and the candidate value.

Equation

$$\mathbf{c}^{\langle t \rangle} = \mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle} + \mathbf{\Gamma}_i^{\langle t \rangle} * \tilde{\mathbf{c}}^{\langle t \rangle} \tag{4}$$

Explanation of equation

- The previous cell state $\mathbf{c}^{\langle t-1 \rangle}$ is adjusted (weighted) by the forget gate $\mathbf{\Gamma}_f^{\langle t \rangle}$
- and the candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$, adjusted (weighted) by the update gate $\mathbf{\Gamma}_i^{\langle t \rangle}$

Output gate $\mathbf{\Gamma}_o$

- The output gate decides what gets sent as the prediction (output) of the time step.
- The output gate is like the other gates, in that it contains values that range from 0 to 1.

Equation

$$\mathbf{\Gamma}_o^{\langle t \rangle} = \sigma(\mathbf{W}_o[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_o) \tag{5}$$

Explanation of the equation

- The output gate is determined by the previous hidden state $\mathbf{a}^{\langle t-1 \rangle}$ and the current input $\mathbf{x}^{\langle t \rangle}$
- The sigmoid makes the gate range from 0 to 1.

Hidden state $\mathbf{a}^{\langle t \rangle}$

- The hidden state gets passed to the LSTM cell's next time step.
- It is used to determine the three gates ($\mathbf{\Gamma}_f, \mathbf{\Gamma}_u, \mathbf{\Gamma}_o$) of the next time step.
- The hidden state is also used for the prediction $y^{\langle t \rangle}$.

Equation

$$\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_o^{\langle t \rangle} * \tanh(\mathbf{c}^{\langle t \rangle}) \tag{6}$$

Explanation of equation

- The hidden state $\mathbf{a}^{\langle t \rangle}$ is determined by the cell state $\mathbf{c}^{\langle t \rangle}$ in combination with the output gate $\mathbf{\Gamma}_o$.
- The cell state state is passed through the `tanh` function to rescale values between -1 and 1.
- The output gate acts like a "mask" that either preserves the values of $\tanh(\mathbf{c}^{\langle t \rangle})$ or keeps those values from being included in the hidden state $\mathbf{a}^{\langle t \rangle}$

Prediction $\mathbf{y}_{pred}^{\langle t \rangle}$

- The prediction in this use case is a classification, so you'll use a softmax.

Equation

$$\mathbf{y}_{pred}^{\langle t \rangle} = \text{softmax}(\mathbf{W}_y \mathbf{a}^{\langle t \rangle} + \mathbf{b}_y)$$

Forward Pass for LSTM

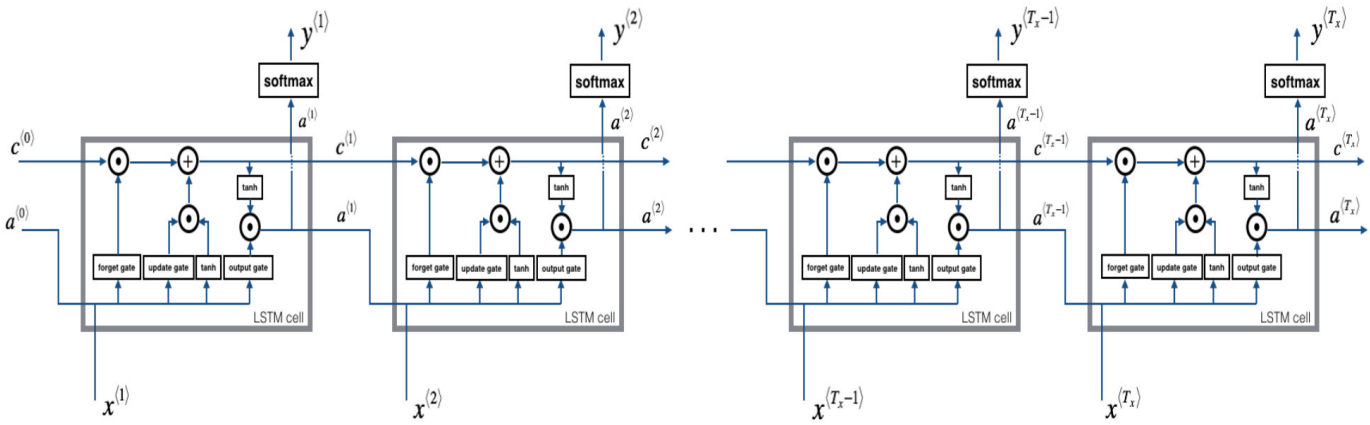


Figure 5: LSTM over multiple time steps.

Instructions

- Get the dimensions n_x, n_a, n_y, m, T_x from the shape of the variables
- Initialize the 3D tensors a, c and y

    - a: hidden state, shape (n_a, m, T_x)
    - c: cell state, shape (n_a, m, T_x)
    - y: prediction, shape (n_y, m, T_x) (Note that $T_y = T_x$ in this example)
    - **Note** Setting one variable equal to the other is a "copy by reference". In other words, don't do `c = a', otherwise both these variables point to the same underlying variable.
- Initialize the 2D tensor $a^{\langle t \rangle}$
    - $a^{\langle t \rangle}$ stores the hidden state for time step t.
    - $a^{\langle 0 \rangle}$, the initial hidden state at time step 0, is passed in when calling the function.
    - $a^{\langle t \rangle}$ and $a^{\langle 0 \rangle}$ represent a single time step, so they both have the shape (n_a, m)
    - Initialize $a^{\langle t \rangle}$ by setting it to the initial hidden state ($a^{\langle 0 \rangle}$) that is passed into the function.
- Initialize $c^{\langle t \rangle}$ with zeros.
    - $c^{\langle t \rangle}$ represents a single time step, so its shape is (n_a, m)
    - **Note**: create `c_next` as its own variable with its own location in memory. Do not initialize it as a slice of the 3D tensor c. In other words, **don't** do `c_next = c[:,:,0]`.
- For each time step, do the following:
    - From the 3D tensor x, get a 2D slice $x^{\langle t \rangle}$ at time step t
    - Call the `lstm_cell_forward` function that you defined previously, to get the hidden state, cell state, prediction, and cache
    - Store the hidden state, cell state and prediction (the 2D tensors) inside the 3D tensors
    - Append the cache to the list of caches

What you should remember:

- An LSTM is similar to an RNN in that they both use hidden states to pass along information, but an LSTM also uses a cell state, which is like a long-term memory, to help deal with the issue of vanishing gradients
- An LSTM cell consists of a cell state, or long-term memory, a hidden state, or short-term memory, along with 3 gates that constantly update the relevancy of its inputs:
    - A **forget** gate, which decides which input units should be remembered and passed along. It's a tensor with values between 0 and 1.
        - If a unit has a value close to 0, the LSTM will "forget" the stored state in the previous cell state.
        - If it has a value close to 1, the LSTM will mostly remember the corresponding value.
    - An **update** gate, again a tensor containing values between 0 and 1. It decides on what information to throw away, and what new information to add.
        - When a unit in the update gate is close to 1, the value of its candidate is passed on to the hidden state.
        - When a unit in the update gate is close to 0, it's prevented from being passed onto the hidden state.
    - And an **output** gate, which decides what gets sent as the output of the time step

LSTM Backward Pass

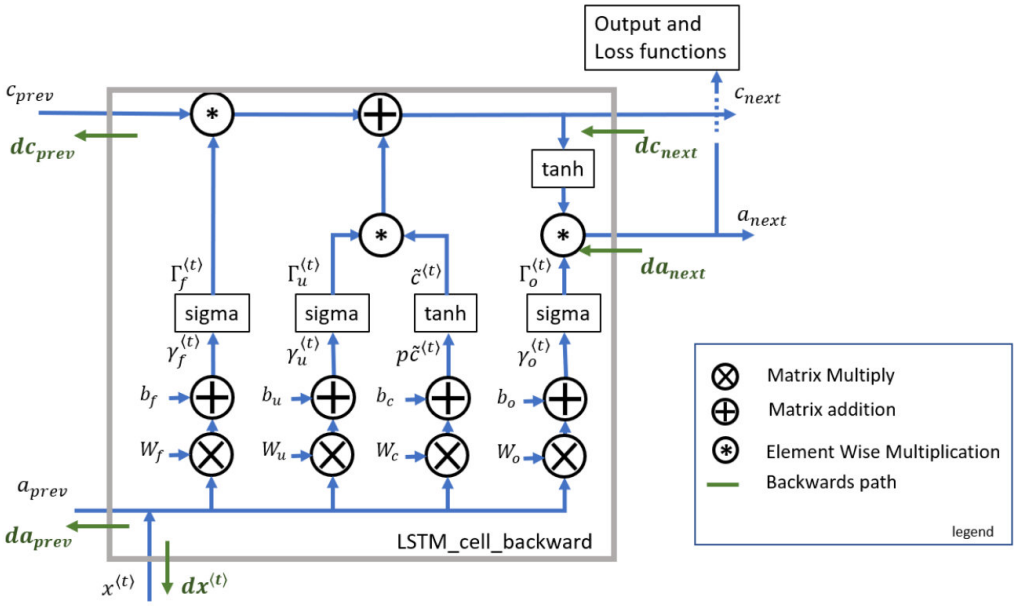


Figure 8: LSTM Cell Backward. Note the output functions, while part of the `lstm_cell_forward`, are not included in `lstm_cell_backward`

Gate Derivatives

Note the location of the gate derivatives (γ..) between the dense layer and the activation function (see graphic above). This is convenient for computing parameter derivatives in the next step.

$$d\gamma_o^{\langle t \rangle} = da_{next} * \tanh(c_{next}) * \Gamma_o^{\langle t \rangle} * \left(1 - \Gamma_o^{\langle t \rangle}\right) \tag{7}$$

$$dp\tilde{c}^{\langle t \rangle} = \left(dc_{next} * \Gamma_u^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * \Gamma_u^{\langle t \rangle} * da_{next}\right) * \left(1 - \left(\tilde{c}^{\langle t \rangle}\right)^2\right) \tag{8}$$

$$d\gamma_u^{\langle t \rangle} = \left(dc_{next} * \tilde{c}^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * \tilde{c}^{\langle t \rangle} * da_{next}\right) * \Gamma_u^{\langle t \rangle} * \left(1 - \Gamma_u^{\langle t \rangle}\right) \tag{9}$$

$$d\gamma_f^{\langle t \rangle} = \left(dc_{next} * c_{prev} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * c_{prev} * da_{next}\right) * \Gamma_f^{\langle t \rangle} * \left(1 - \Gamma_f^{\langle t \rangle}\right) \tag{10}$$

3. Parameter Derivatives

$$dW_f = d\gamma_f^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{11}$$

$$dW_u = d\gamma_u^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{12}$$

$$dW_c = dp\tilde{c}^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{13}$$

$$dW_o = d\gamma_o^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{14}$$

To calculate db_f, db_u, db_c, db_o you just need to sum across all 'm' examples (axis= 1) on $d\gamma_f^{\langle t \rangle}, d\gamma_u^{\langle t \rangle}, dp\tilde{c}^{\langle t \rangle}, d\gamma_o^{\langle t \rangle}$ respectively. Note that you should have the `keepdims = True` option.

$$db_f = \sum_{batch} d\gamma_f^{\langle t \rangle} \tag{15}$$

$$db_u = \sum_{batch} d\gamma_u^{\langle t \rangle} \tag{16}$$

$$db_c = \sum_{batch} d\gamma_c^{\langle t \rangle} \tag{17}$$

$$db_o = \sum_{batch} d\gamma_o^{\langle t \rangle} \tag{18}$$

Finally, you will compute the derivative with respect to the previous hidden state, previous memory state, and input.

$$da_{prev} = W_f^T d\gamma_f^{\langle t \rangle} + W_u^T d\gamma_u^{\langle t \rangle} + W_c^T dp\tilde{c}^{\langle t \rangle} + W_o^T d\gamma_o^{\langle t \rangle} \tag{19}$$

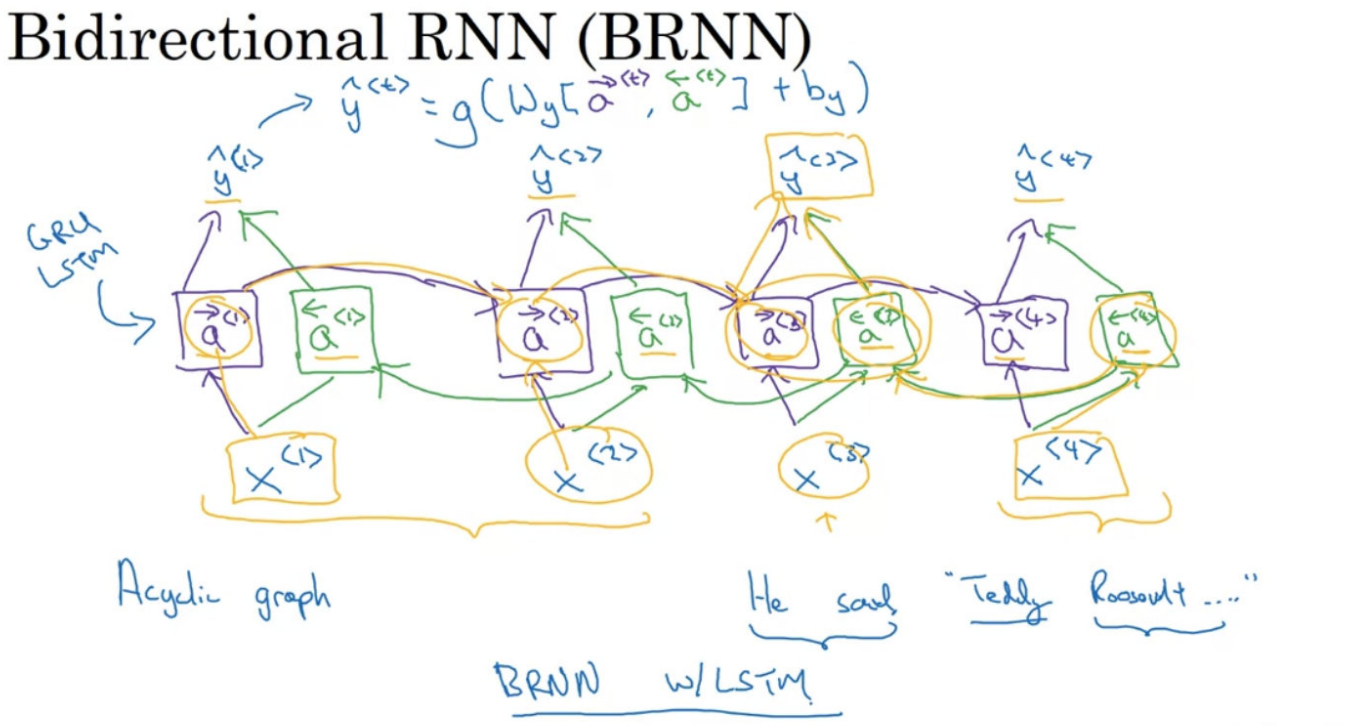
Here, to account for concatenation, the weights for equations 19 are the first n_a, (i.e. $W_f = W_f[:, : n_a]$ etc...)

$$dc_{prev} = dc_{next} * \Gamma_f^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * \Gamma_f^{\langle t \rangle} * da_{next} \tag{20}$$

$$dx^{\langle t \rangle} = W_f^T d\gamma_f^{\langle t \rangle} + W_u^T d\gamma_u^{\langle t \rangle} + W_c^T dp\tilde{c}^{\langle t \rangle} + W_o^T d\gamma_o^{\langle t \rangle} \tag{21}$$

where the weights for equation 21 are from n_a to the end, (i.e. $W_f = W_f[:, n_a :]$ etc...)

Bidirectional RNN (BRNN)



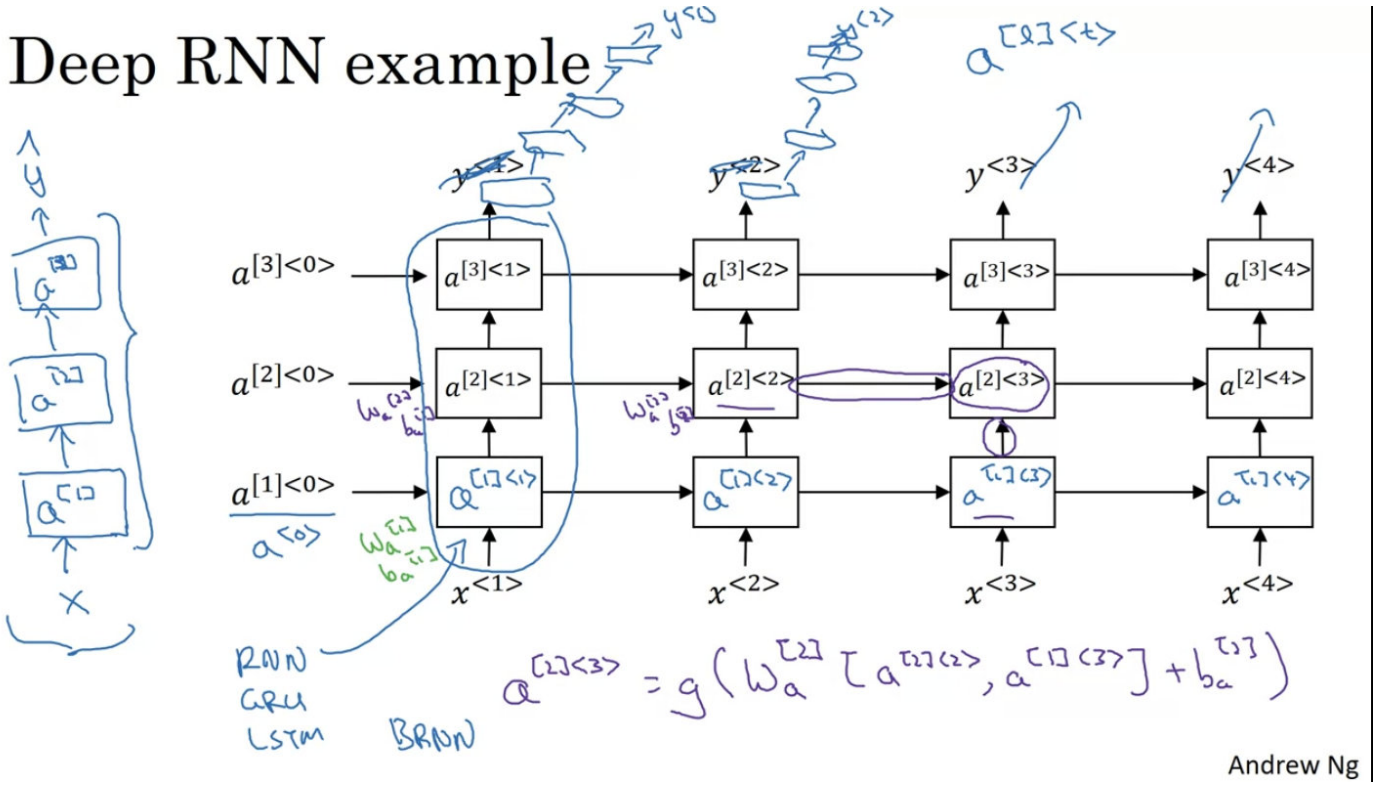
Andrew Ng

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

The blocks can be not just the standard RNN block but they can also be GRU blocks and LSTM blocks.

The disadvantage of the BRNN is that you need the **entire** sequence of data before you can make predictions anywhere.

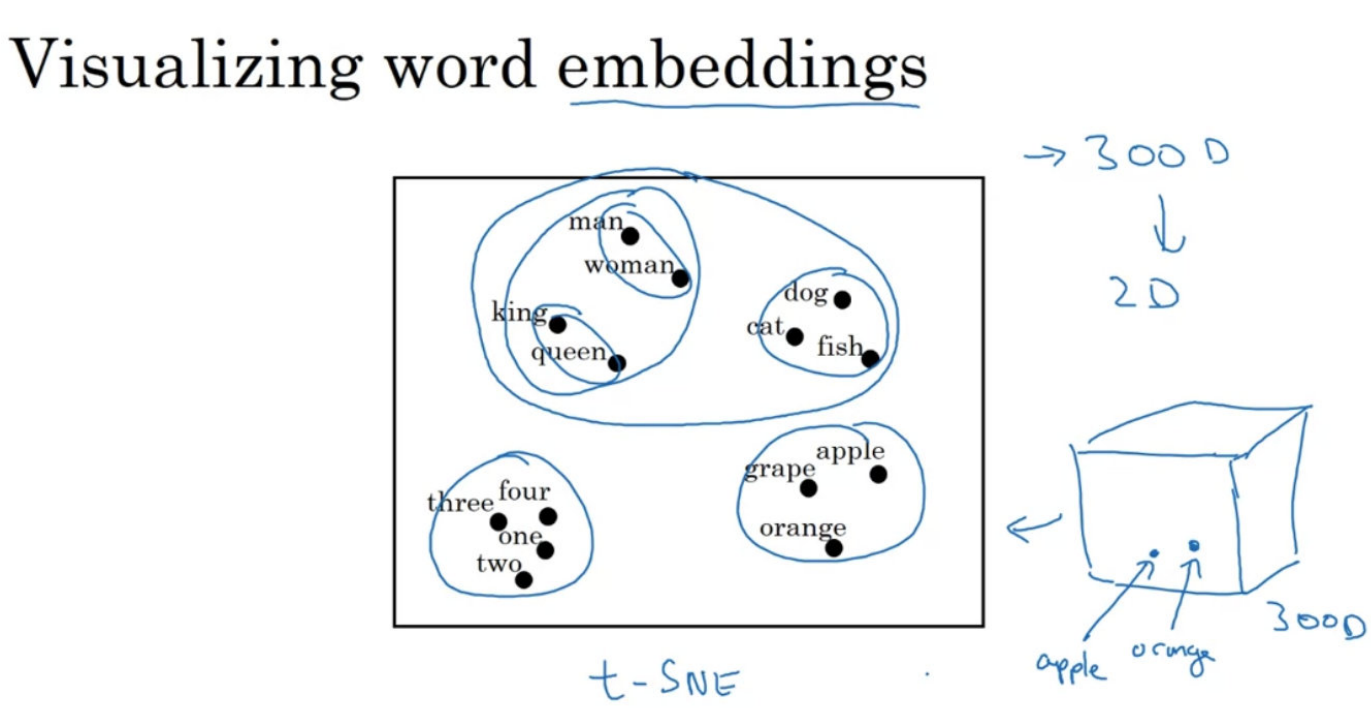
Deep RNNs



Word Embeddings

Representing words by "**features**".

	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



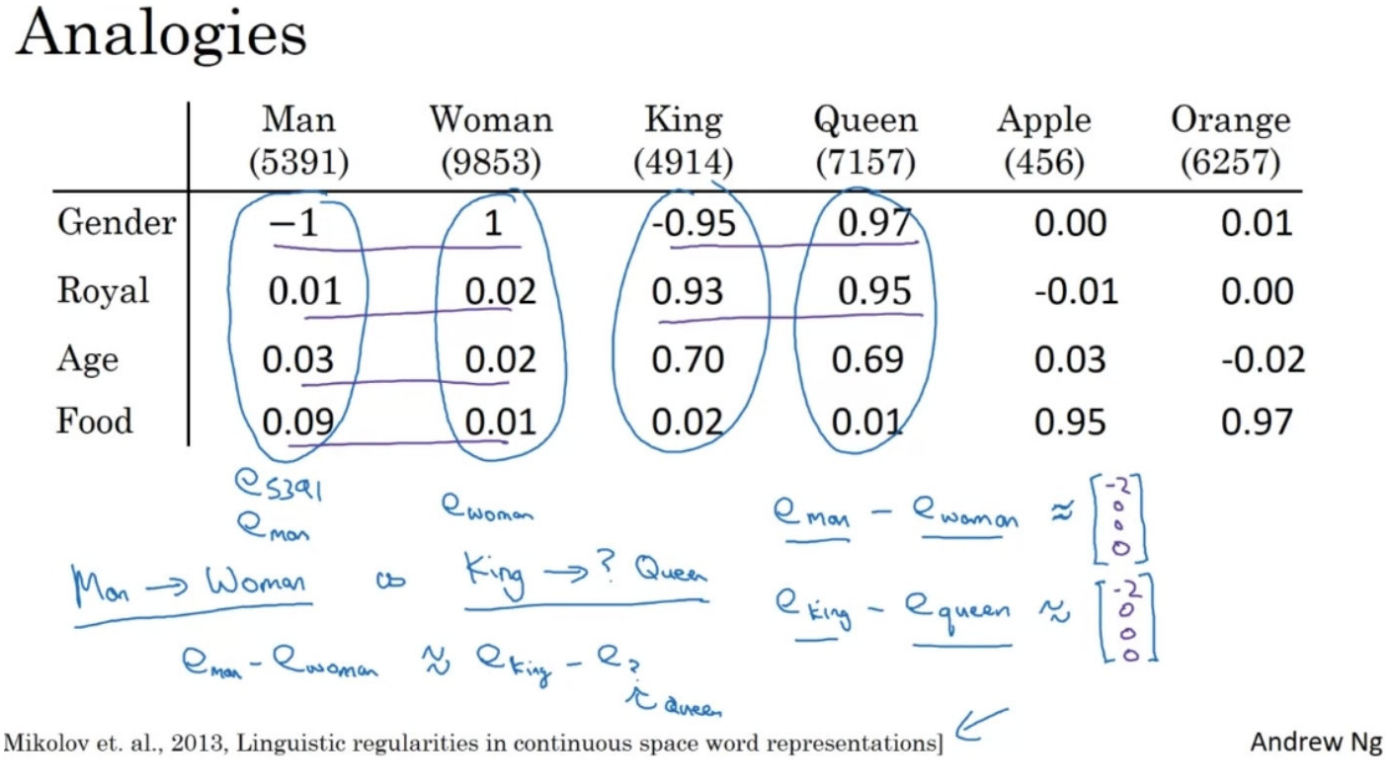
Transfer learning and word embedings

1. Learn word embeddings from large text corpus. (Or download pre-trained embedding online.)
2. Transfer embedding to new task with smaller training set.
3. Optional: Continue to finetune the word embeddings with new data.

The terms `encoding` and `embedding` are used somewhat interchangeably.

Analogies

One of the most facinating properties of word embeddings is that they can also help with **analogy reasoning**.



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

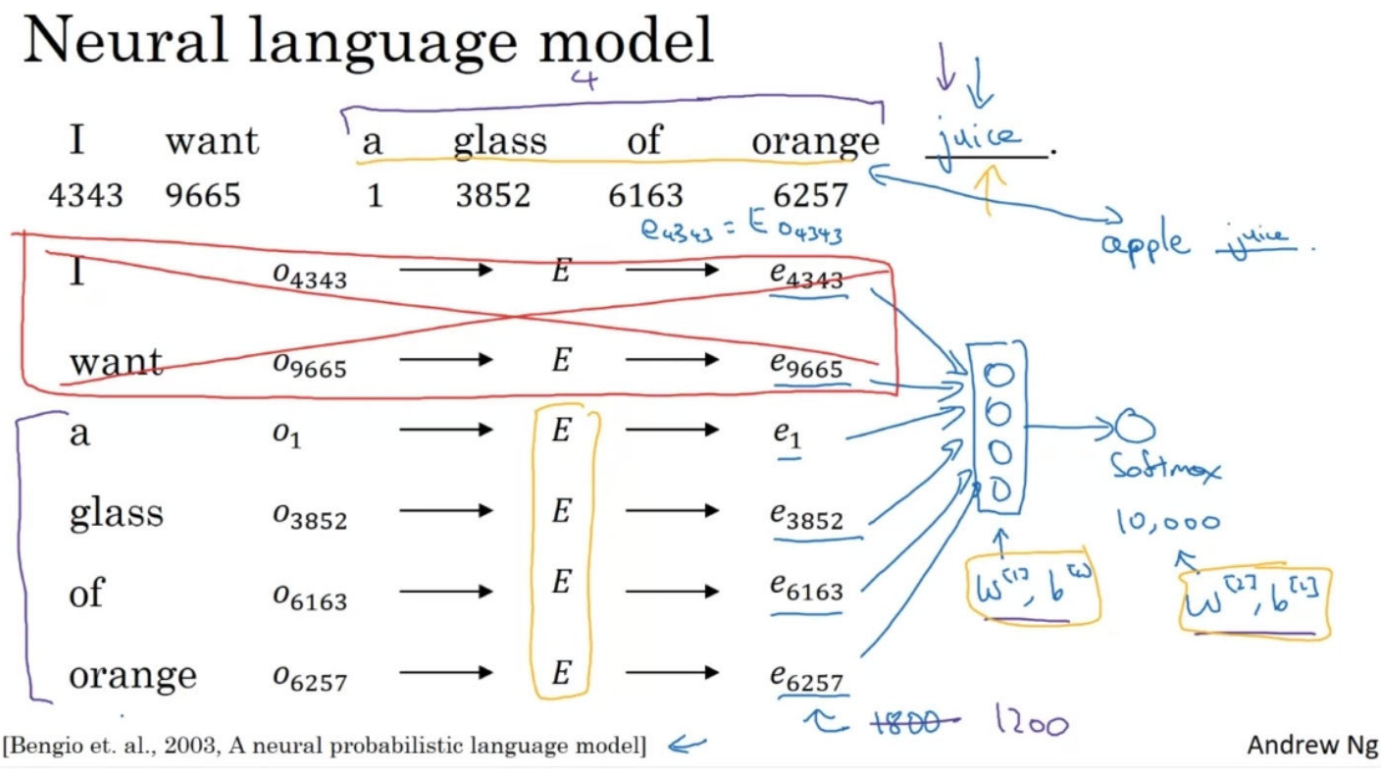
Find word w:

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

Cosine similarity

$$sim(u, v) = \frac{u \cdot v}{||u||_2 ||v||_2} = \cos(\theta)$$

- $u \cdot v$ is the dot product (or inner product) of two vectors
- $||u||_2$ is the norm (or length) of the vector u
- θ is the angle between u and v.
- The cosine similarity depends on the angle between u and v.
  - If u and v are very similar, their cosine similarity will be close to 1.
  - If they are dissimilar, the cosine similarity will take a smaller value.



Embedding Matrix

$$E = \begin{bmatrix} a & aaron & \cdots & orange & \cdots & zulu & \langle UNK \rangle \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \in \mathbb{R}^{300 \times 10{,}000}$$

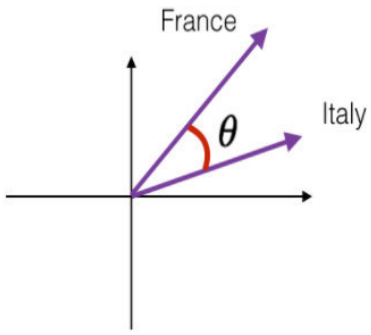
The word orange is the 6257th token in the vocabulary:

$$orange = O_{6257} = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^{10,000}$$

O_{6257} is the one-hot vector of the word orange. Notice that:

$$E \cdot O_{6257} = e_{6257} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \in \mathbb{R}^{300}$$

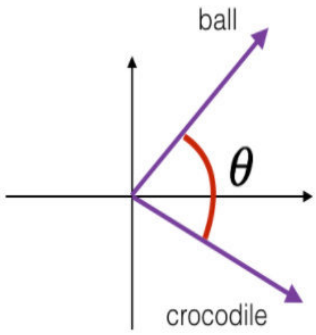
Learning Word Embeddings



France and Italy are quite similar

θ is close to 0°

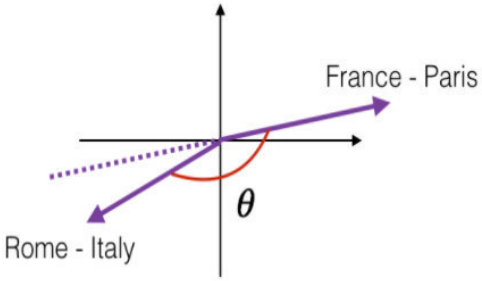
$\cos(\theta) \approx 1$



ball and crocodile are not similar

θ is close to 90°

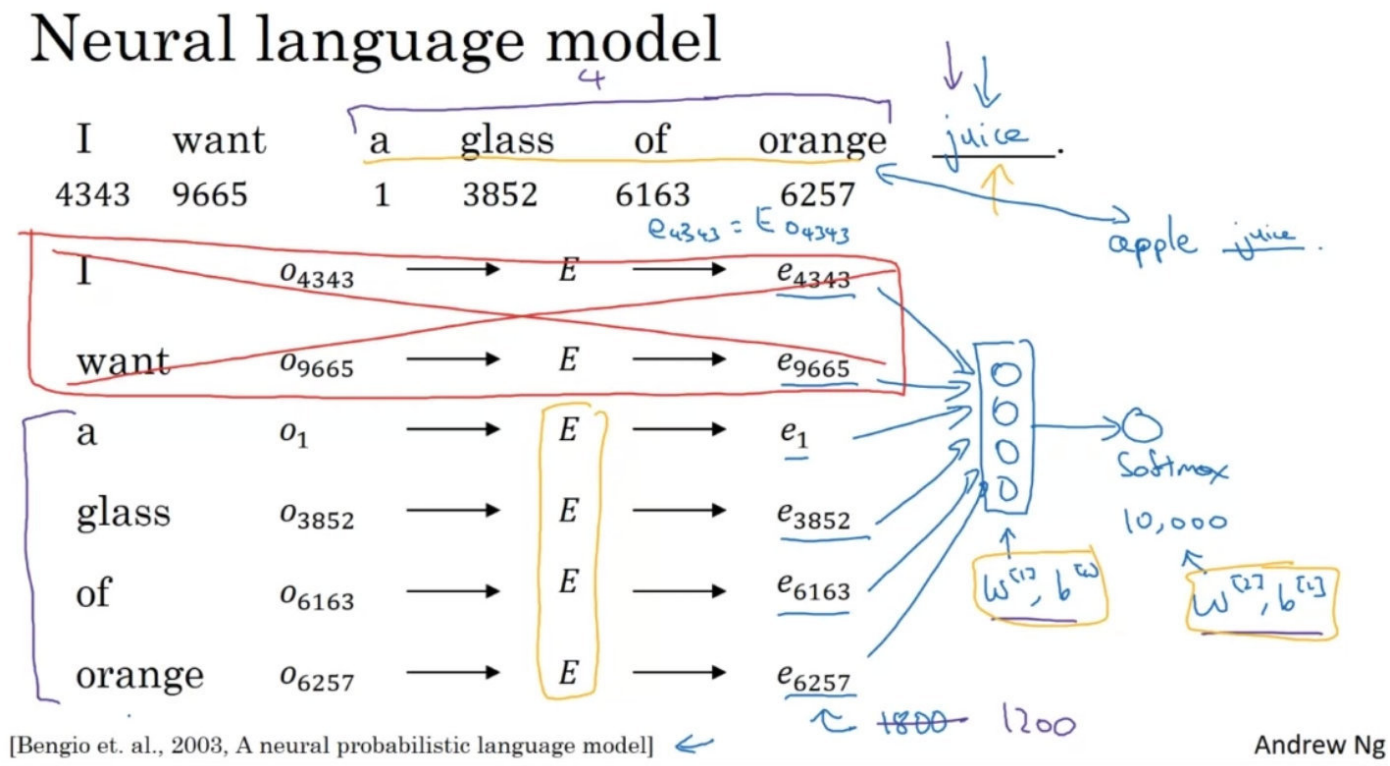
$\cos(\theta) \approx 0$



the two vectors are similar but opposite
the first one encodes (city - country)
while the second one encodes (country - city)

θ is close to 180°

$\cos(\theta) \approx -1$

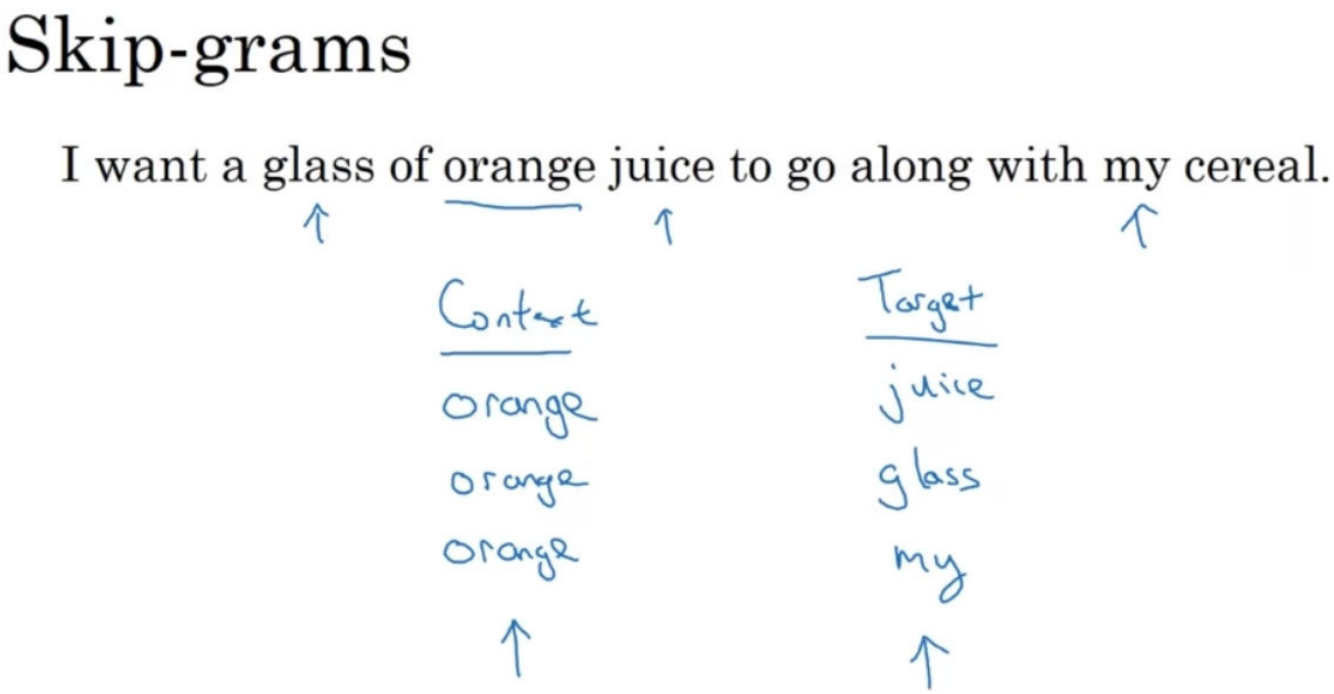


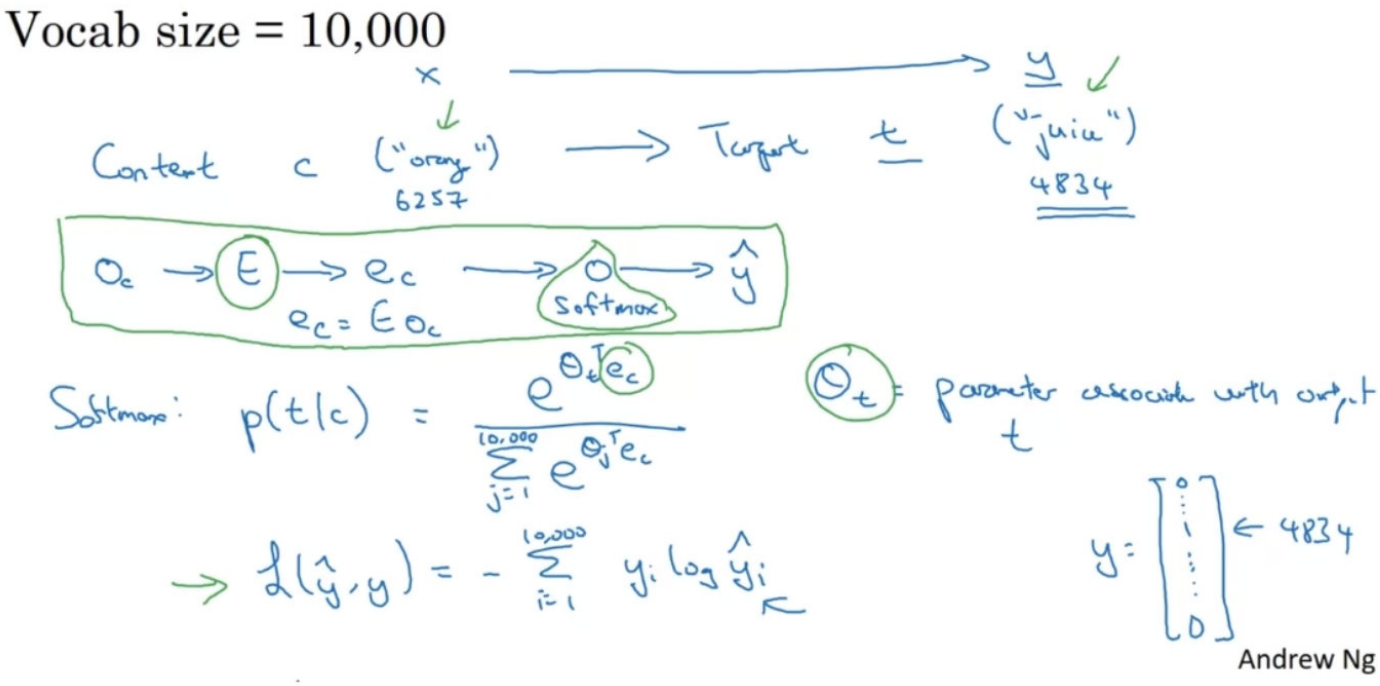
If you really want to **build a language model**, it is nature to use **last few words** as the context. But if the main goal is to **learn word embedding**, then you can use **all of these other context**, and they will result in meaningful word embedding as well.

Word2Vec

Skip-grams

1. Randomly pick a word to be the context word.
2. Randomly pick another word within some window (Say ±5 words) of the context word.





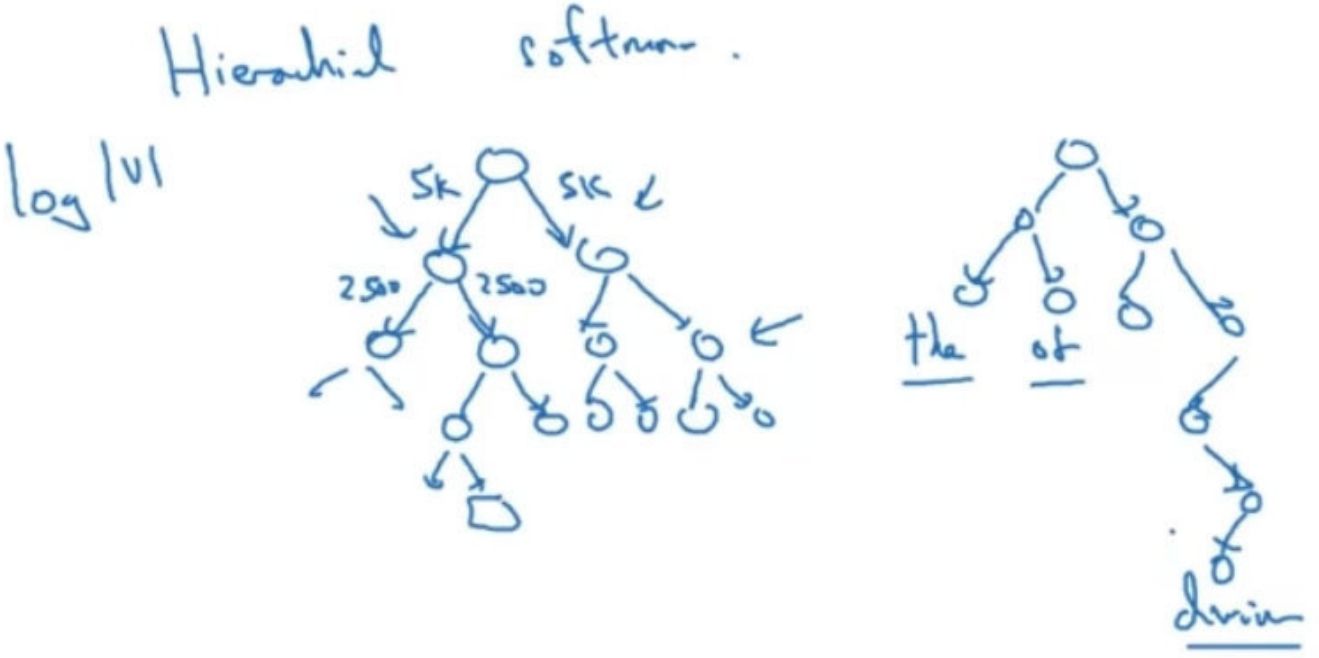
θ_t is the parameter associated with output t. It is the chance of the particular word t being the label.

However, there are a couple problems with using skip-grams.

The primary problem is **computational speed**.

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

To evaluate this probability, you need to carry out a sum over all 10,000 words in the vocabulary. One of the solution is to use **Hierachical Softmax Classifier**. In practice, the hierachical softmax classifer can be developed so the **common words** tend to be on **top**, whereas the less common words can be buried much deeper in the tree.

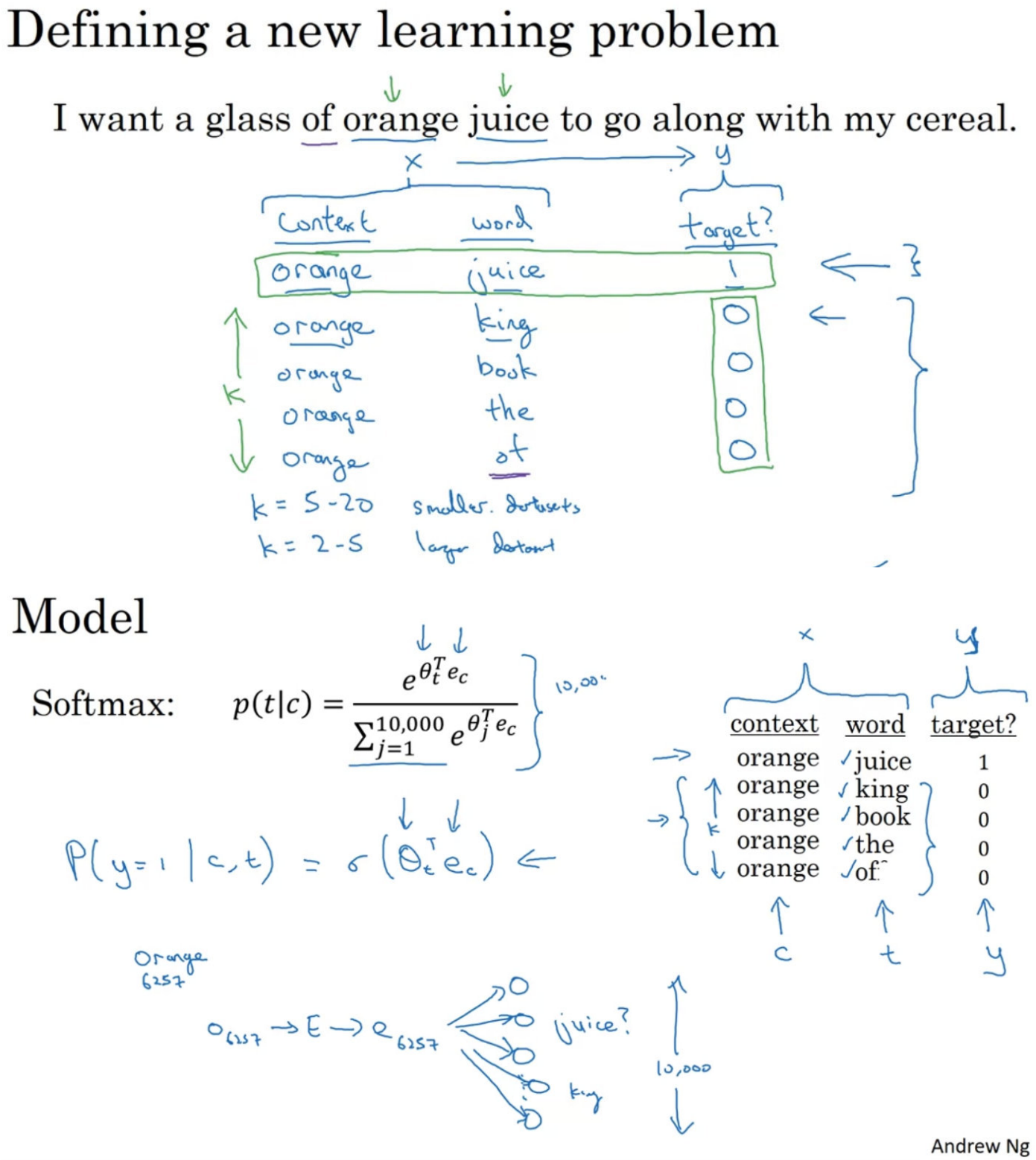


How to sample the context c?

In practice the distribution of words pc isn't taken just entirely uniformly at random for the training set purpose, but instead there are different heuristics that you could use in order to balance out something from the common words together with the less common words.

Negative sampling

To solve the computational speed problems, this algorithm defines a new learning problem.



Think of this as having 10,000 binary logistic regression classifers, but instead of training all 10,000 of them on every iterarion, **we are going to train `k+1` of them (`1` coresponding to actual target word, and `k` randomly chosen negative examples)**.

Selecting negative examples

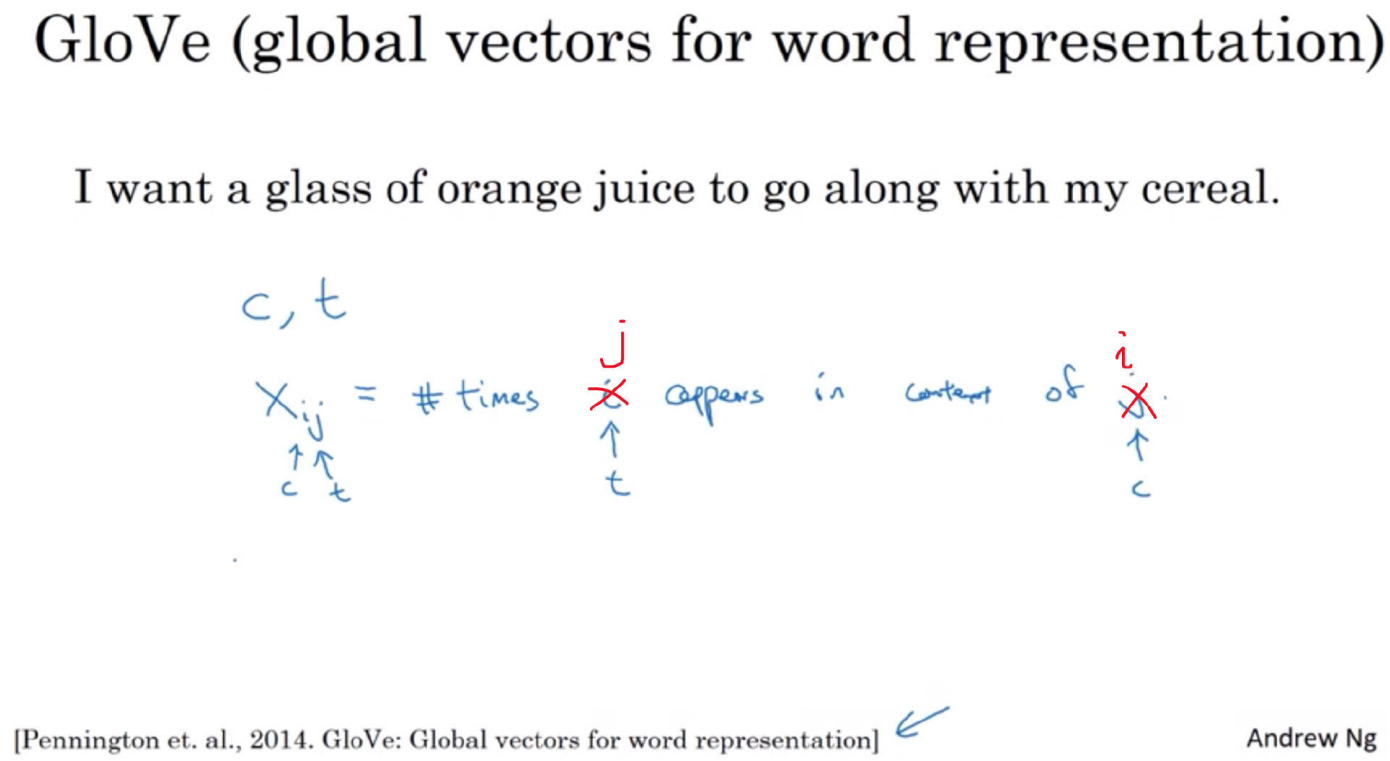
(Empirically) Sample according to:

$$p(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10,000} f(w_j)^{3/4}}$$

where $f(w_i)$ is the observed **frequency** of a particular word in the English language or in the training set corpus.

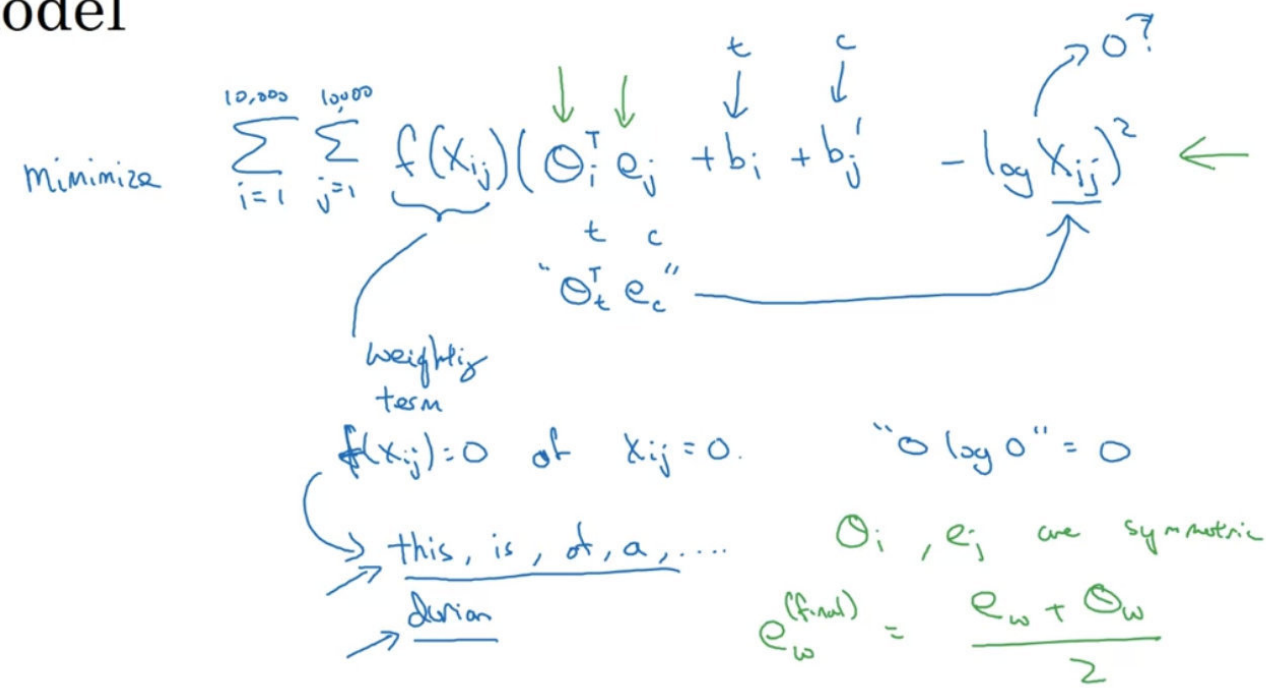
GloVe Word Vectors

GloVe (global vectors for word representation)



For the purposes of the GloVe algorithm, we define context and target as whether or not the two words appear in close proximity.

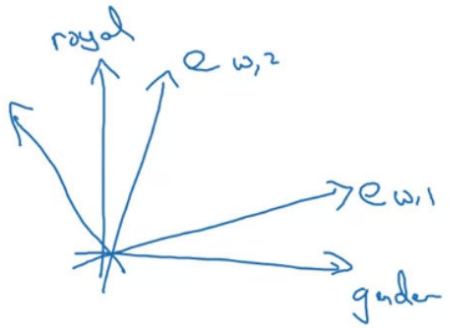
Model



$$minimize \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$$

A note on the featurization view of word embeddings

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	
Gender	−1	1	-0.95	0.97	←
Royal	0.01	0.02	0.93	0.95	←
Age	0.03	0.02	0.70	0.69	←
Food	0.09	0.01	0.02	0.01	←



$$\text{minimize } \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i - b_j' - \log X_{ij})^2$$

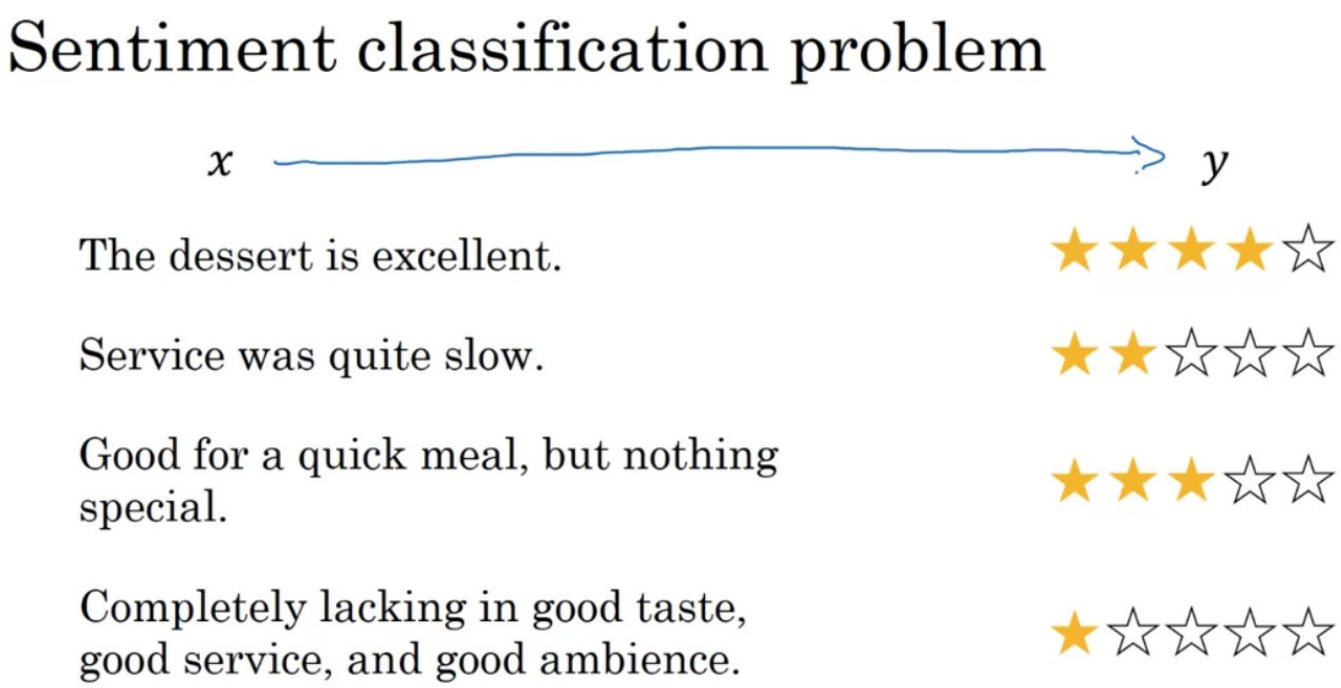
$(A\theta_i)^T (A^{-T} e_j) = \theta_i^T A^T A^{-T} e_j$

Andrew Ng

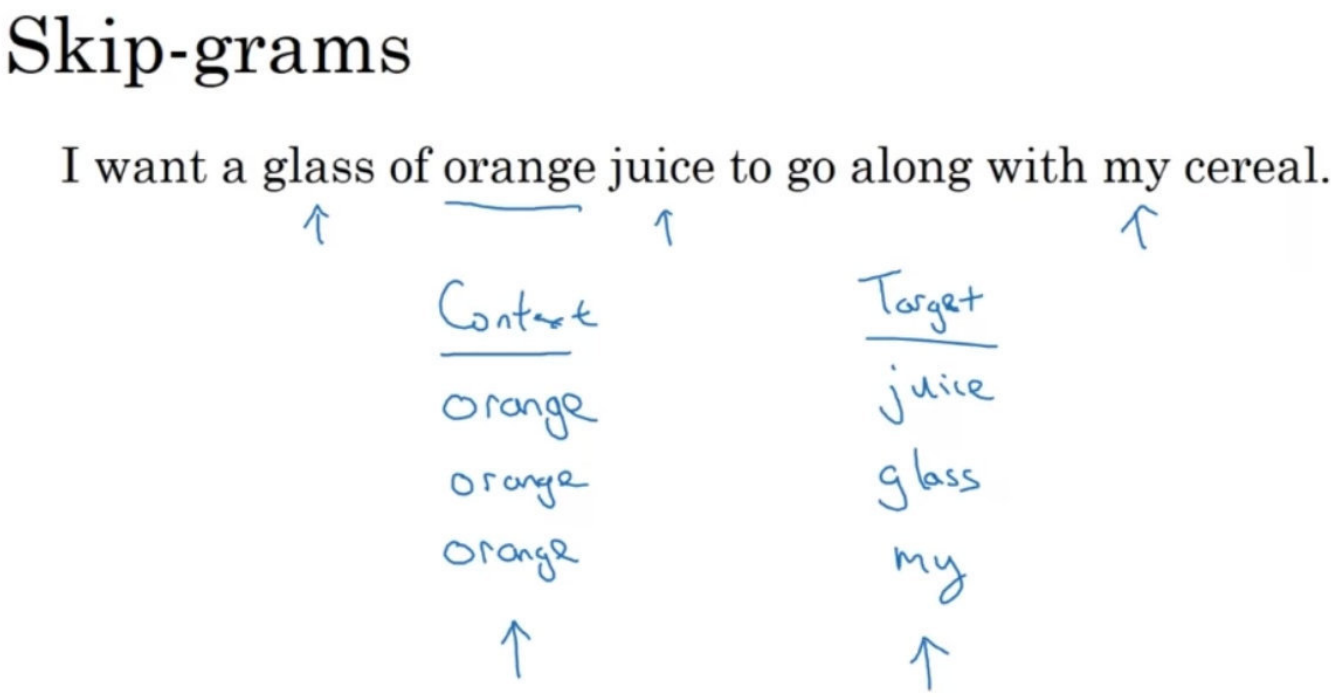
You can't guarantee that the axis used to represent the features will be well-aligned with waht might be easily humanly interpretable axis.

Sentiment Classification

Sentiment classification is the task of looking at a piece of text and telling if someone likes or dislikes the thing they are talking about.

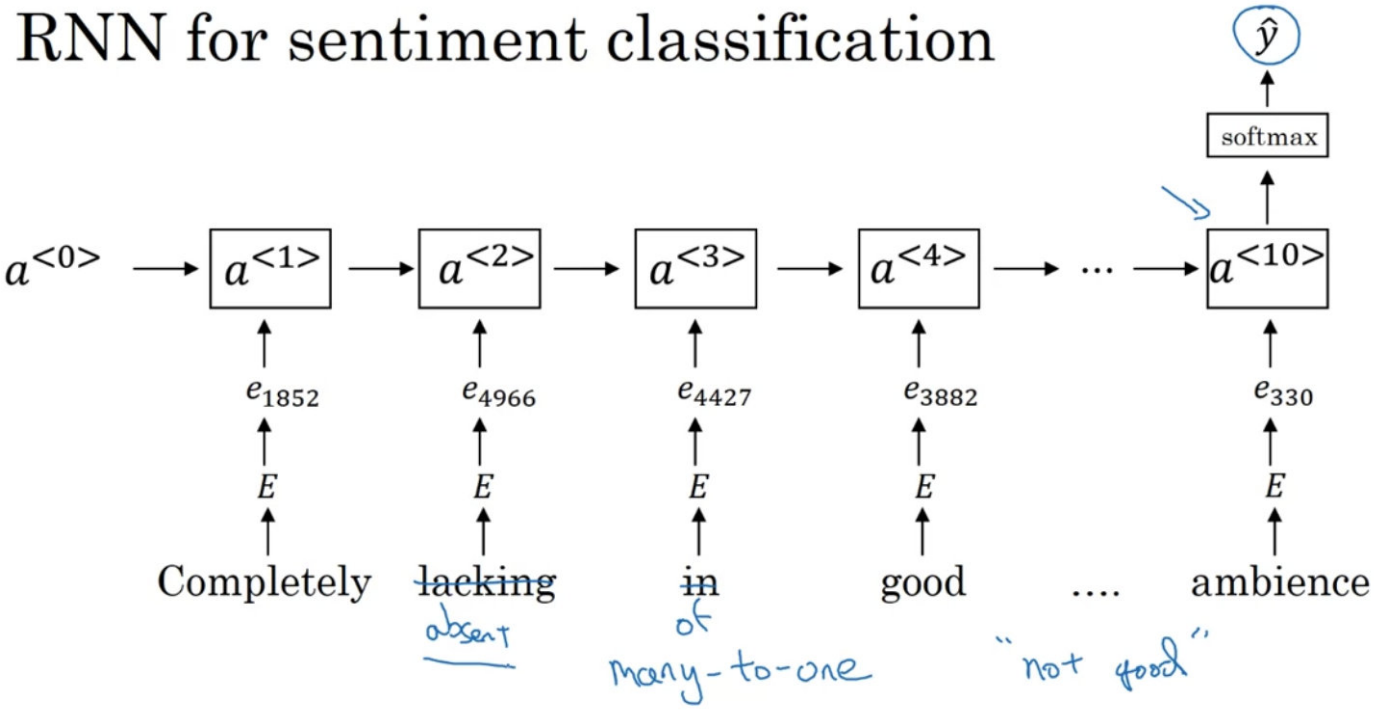


One of the challenges of sentiment classification is you might not have a huge label data set.



However, if you use an algorithm like this that ignores word **order** and just sums or averages all of the embeddings for the different words, the result is not convicing for shown example (because there are many "good" in the review, and the alogorithm may think this is a good review after averaging the word embedding). So, instead of just summing all of the word embeddings, you can instead use a **RNN** for sentiment classification.

RNN for sentiment classification



Debiasing Word Embeddings

The problem of bias in word embeddings

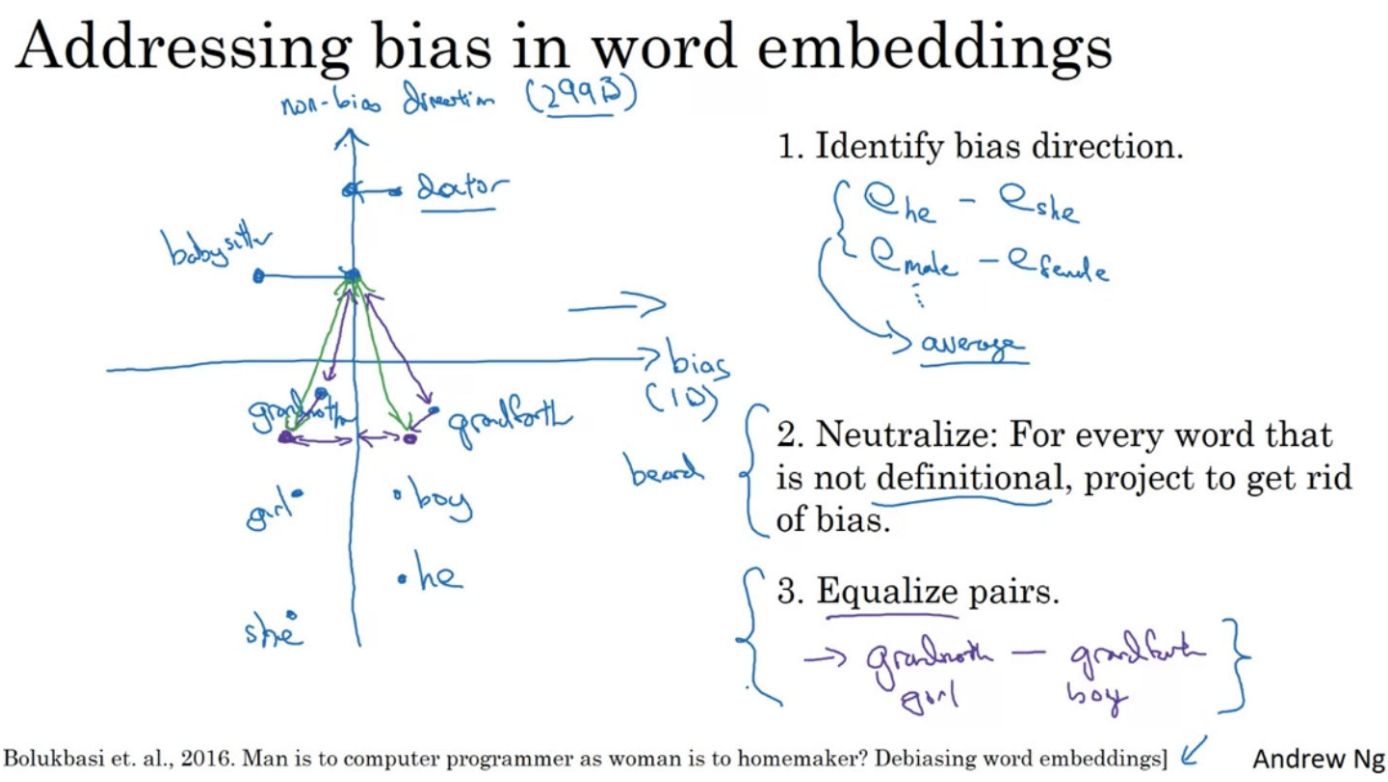
Man:Woman as King:Queen

Man:Computer_Programmer as Woman:Homemaker

Father:Doctor as Mother:Nurse

Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model.

[Bolukbasi et. al., 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings]



Neutralization

The figure below should help you visualize what neutralizing does. If you're using a 50-dimensional word embedding, the 50 dimensional space can be split into two parts: The bias-direction g, and the remaining 49 dimensions, which is called $g_\perp$ here. In linear algebra, we say that the 49-dimensional $g_\perp$ is perpendicular (or "orthogonal") to g, meaning it is at 90 degrees to g. The neutralization step takes a vector such as $e_{receptionist}$ and zeros out the component in the direction of g, giving us $e^{debiased}_{receptionist}$.

Even though $g_\perp$ is 49-dimensional, given the limitations of what you can draw on a 2D screen, it's illustrated using a 1-dimensional axis below.

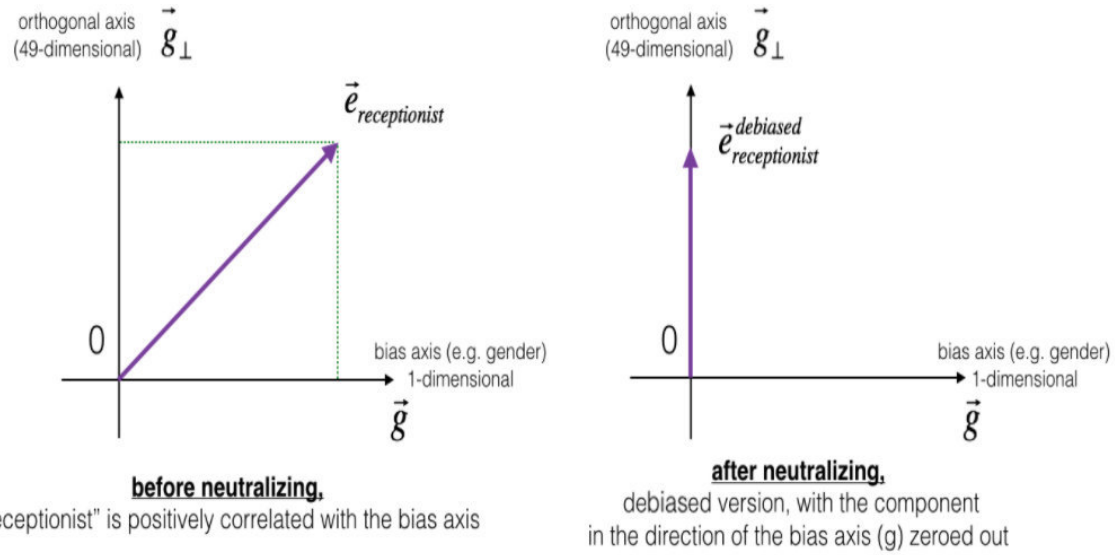


Figure 2: The word vector for "receptionist" represented before and after applying the neutralize operation.

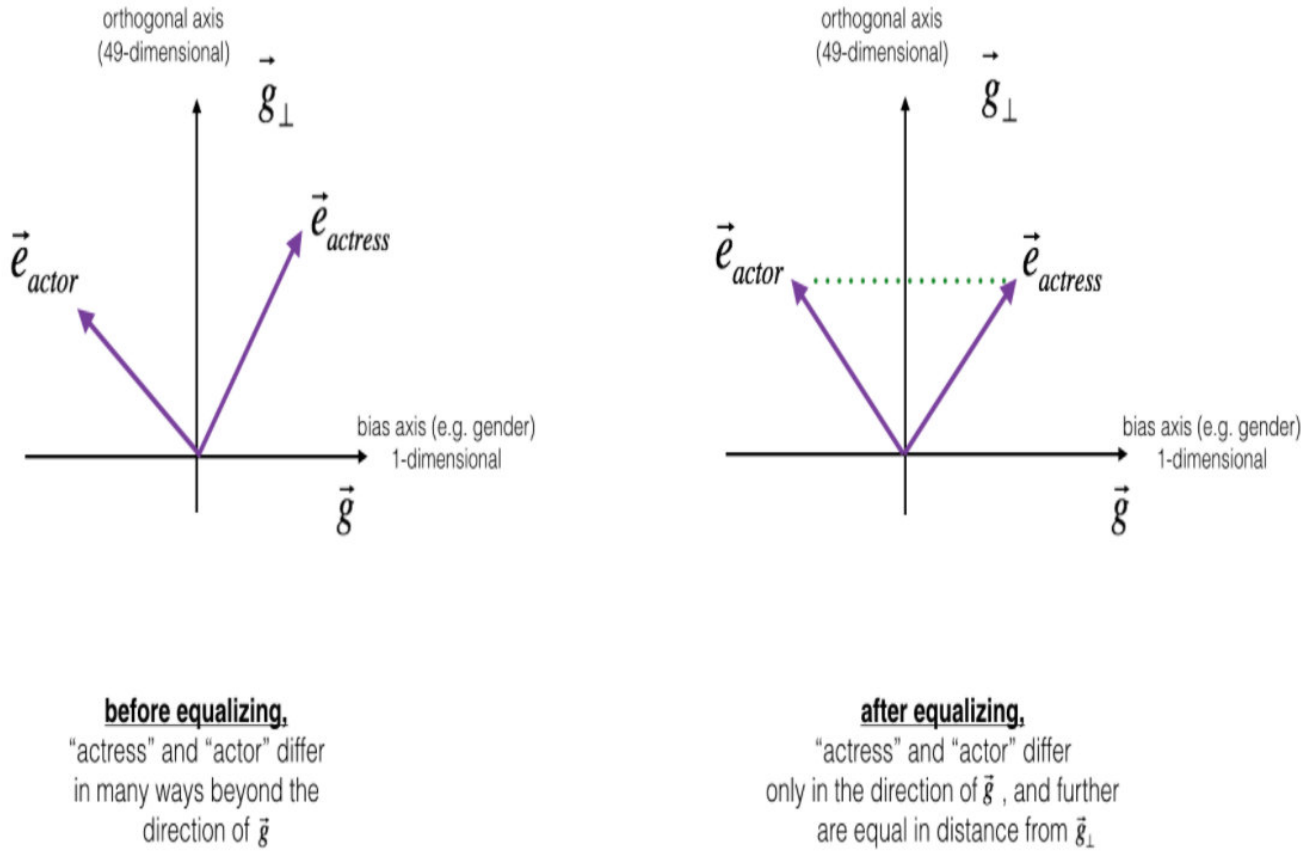
Given an input embedding e, you can use the following formulas to compute $e^{debiased}$:

$$e^{bias\_component} = \frac{e \cdot g}{||g||_2} * \frac{g}{||g||_2} = \frac{e \cdot g}{||g||_2^2} * g$$
$$e^{debiased} = e - e^{bias\_component}$$

where $e^{bias\_component}$ is the projection of e onto the direction g.

Equalization

The key idea behind equalization is to make sure that a particular pair of words are equidistant from the 49-dimensional $g_\perp$. The equalization step also ensures that the two equalized steps are now the same distance from $e^{debiased}_{receptionist}$, or from any other work that has been neutralized. Visually, this is how equalization works:



before equalizing,
"actress" and "actor" differ in many ways beyond the direction of $\vec{g}$

after equalizing,
"actress" and "actor" differ only in the direction of $\vec{g}$, and further are equal in distance from $\vec{g}_\perp$

The key equations:

$$\mu = \frac{e_{w1} + e_{w2}}{2}$$

$$\mu_B = \frac{\mu \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis}$$

$$\mu_\perp = \mu - \mu_B$$

$$e_{w1B} = \frac{e_{w1} \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis}$$

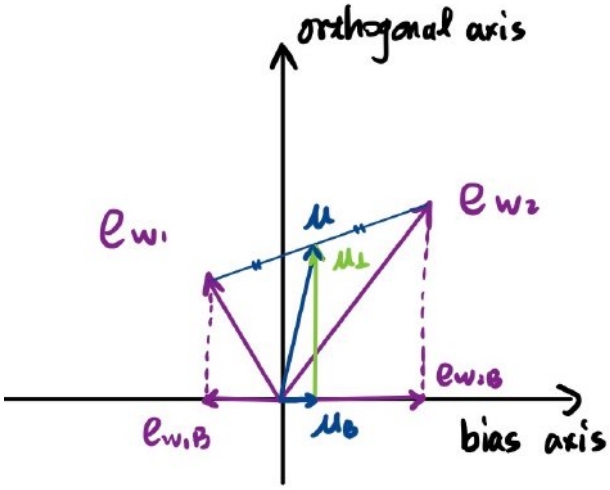
$$e_{w2B} = \frac{e_{w2} \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis}$$

$$e_{w1B}^{corrected} = \sqrt{|1 - ||\mu_\perp||_2^2|} * \frac{e_{\text{w1B}} - \mu_B}{||(e_{w1} - \mu_\perp) - \mu_B||_2}$$

$$e_{w2B}^{corrected} = \sqrt{|1 - ||\mu_\perp||_2^2|} * \frac{e_{\text{w2B}} - \mu_B}{||(e_{w2} - \mu_\perp) - \mu_B||_2}$$

$$e_1 = e_{w1B}^{corrected} + \mu_\perp$$

$$e_2 = e_{w2B}^{corrected} + \mu_\perp$$



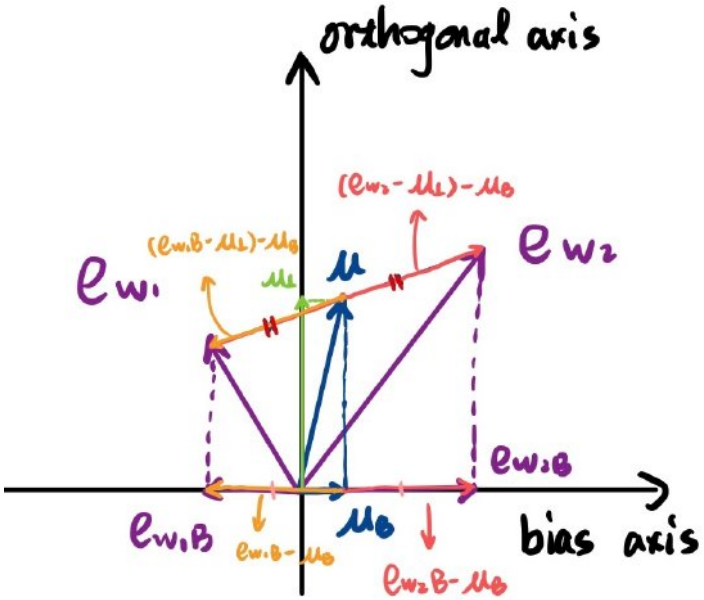
$$u = \frac{e_{w_1} + e_{w_2}}{2}$$

$$u_B = \frac{u \cdot bias\_axis}{\|bias\_axis\|_2^2} * bias\_axis$$

$$u_\perp = u - u_B$$

$$e_{w_1B} = \frac{e_{w_1} \cdot bias\_axis}{\|bias\_axis\|_2^2} * bias\_axis$$

$$e_{w_2B} = \frac{e_{w_2} \cdot bias\_axis}{\|bias\_axis\|_2^2} * bias\_axis$$

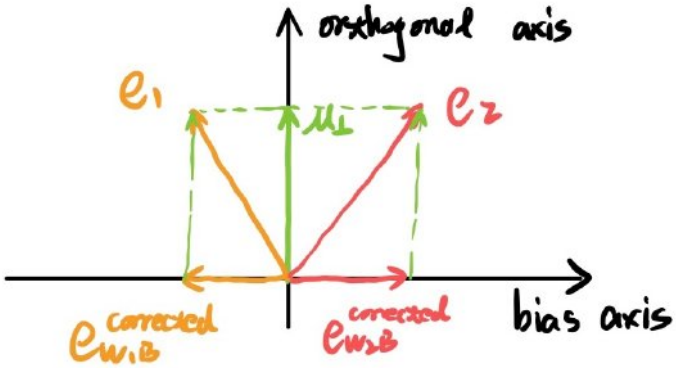


$$e_{w_1B}^{corrected} = \sqrt{|1 - \|u_\perp\|_2^2|} * \frac{e_{w_1B} - u_B}{\|(e_{w_1} - u_\perp) - u_B\|_2}$$

$$e_{w_2B}^{corrected} = \sqrt{|1 - \|u_\perp\|_2^2|} * \frac{e_{w_2B} - u_B}{\|(e_{w_2B} - u_\perp) - u_B\|_2}$$

$$\Downarrow$$

$$\|e_{w_1B}^{corrected}\|_2 = \|e_{w_2B}^{corrected}\|_2$$

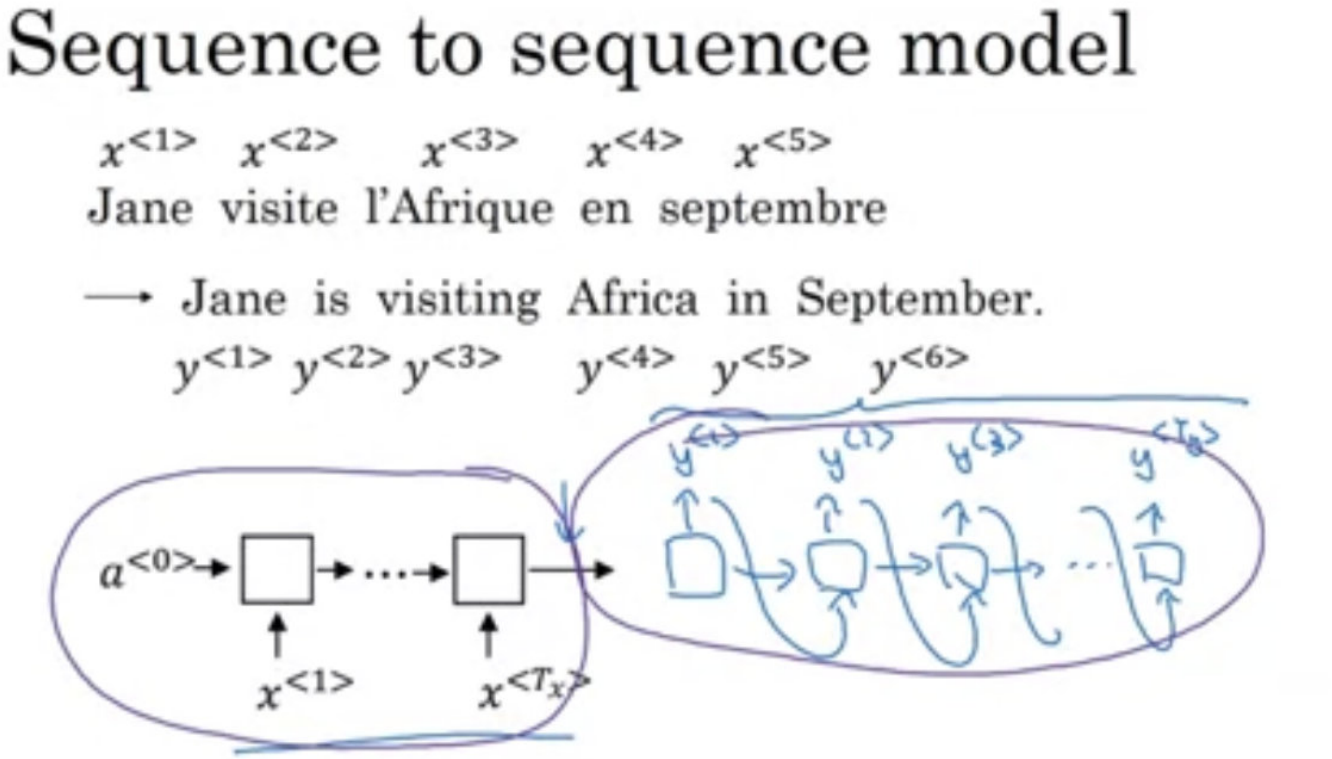


$$e_1 = e_{w_1B}^{corrected} + u_\perp$$

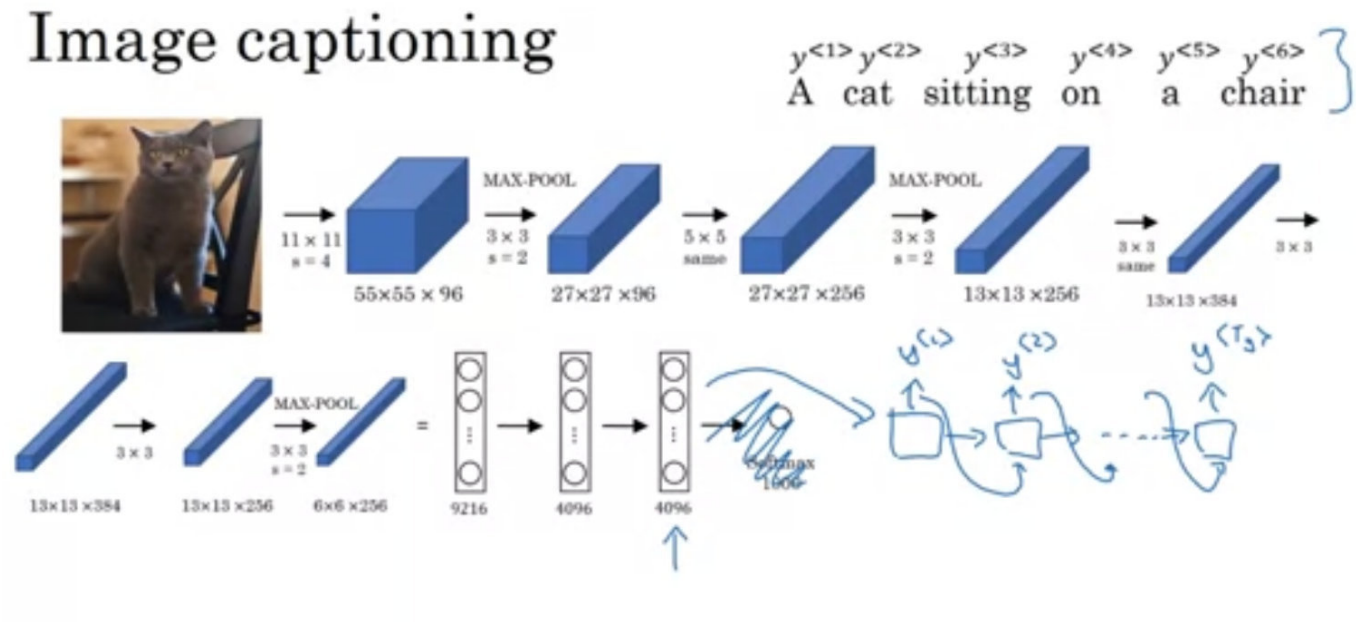
$$e_2 = e_{w_2B}^{corrected} + u_\perp$$

Sequence to Sequence Model

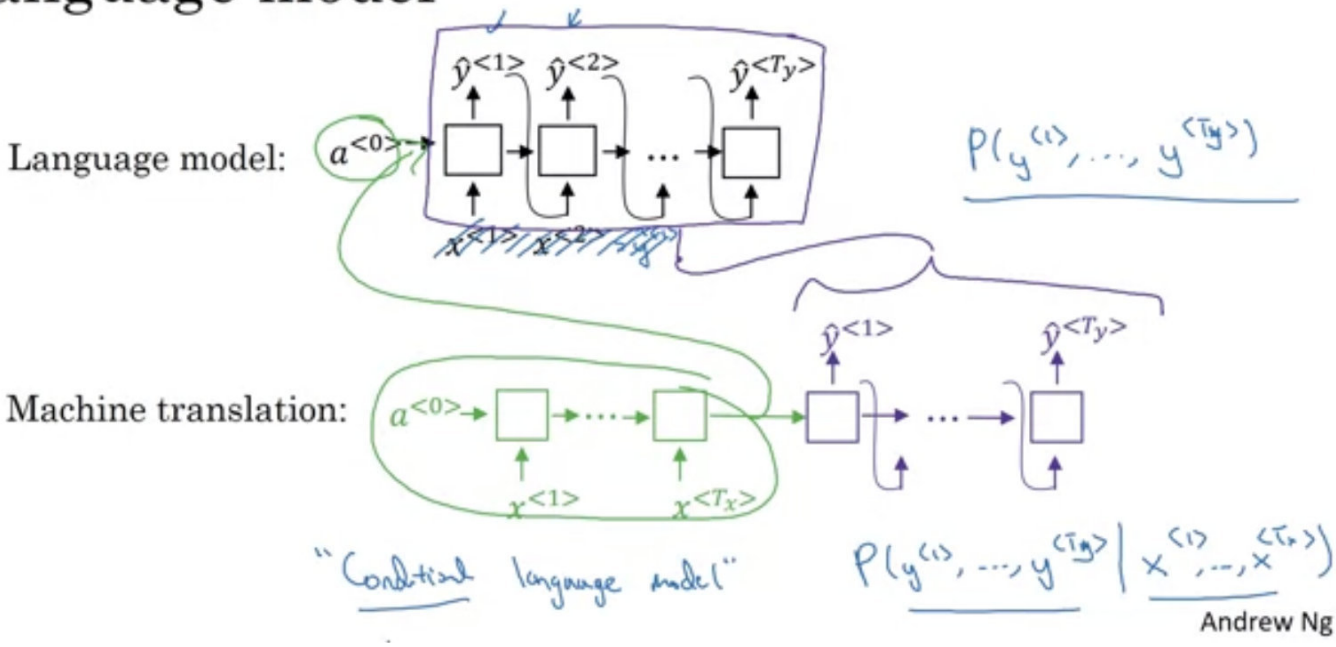
Basic models



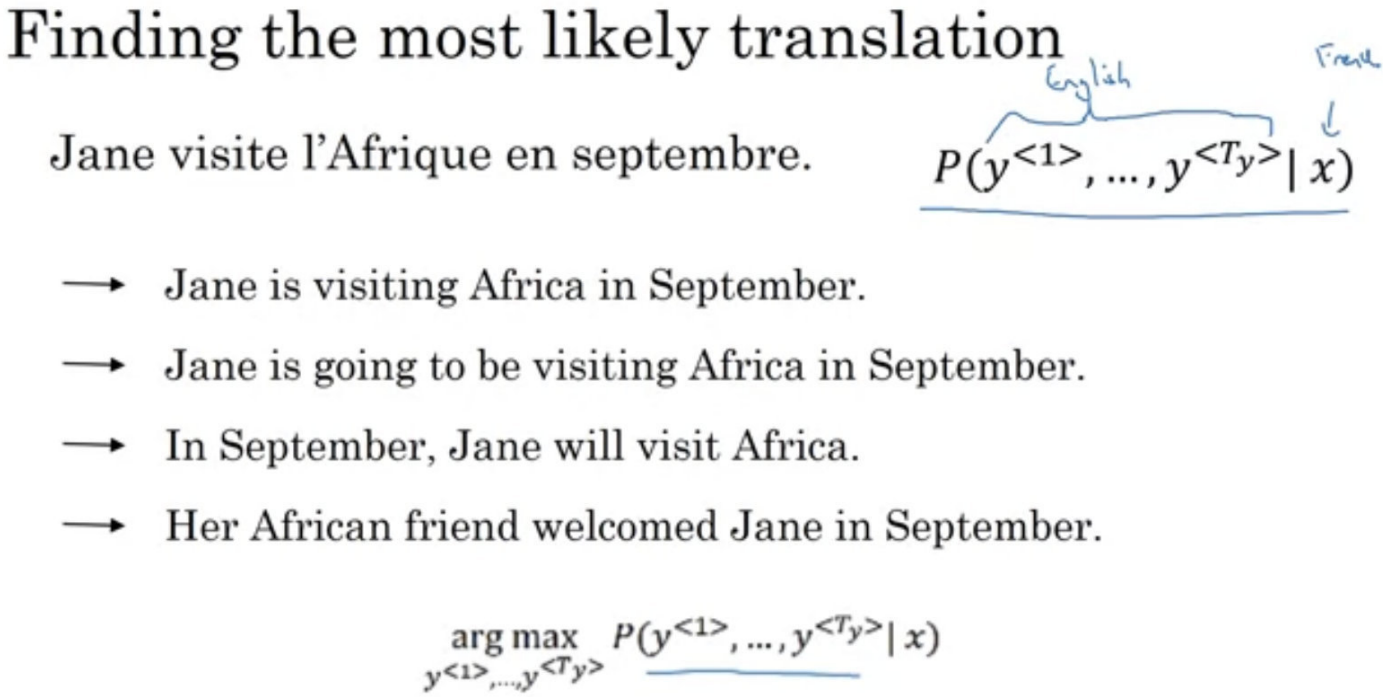
An **encoding** network and a **decoding** network.



Of the key differences is that you do not want to randomly choose (like we did in generating novel text) in translation. You may be want most likely translation.



The translation model can be viewed as a **conditional language model**.



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

Instead of finding the probability distribution $P(y^{\lang1\rang},\cdots,y^{\lang T_y \rang}|x)$, what you would like is to find the English sentence y that **maximizes** that conditional probability.

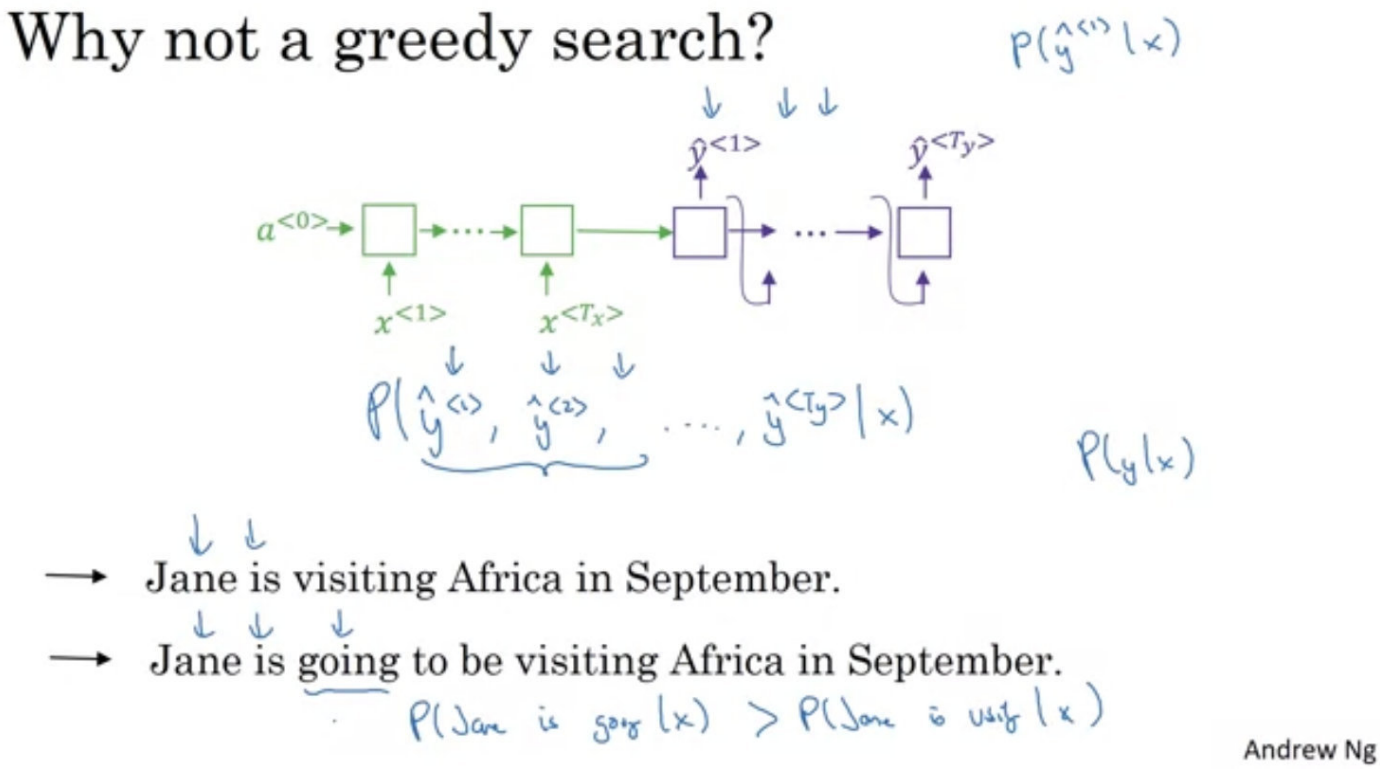
$$\arg\max_{y^{\langle 1\rangle},\cdots,y^{\langle T_y\rangle}} P(y^{\langle 1\rangle},\cdots,y^{\langle T_y\rangle}|x)$$

The most common algorithm for doing this is called **beam search**.

Beam search

Why not greedy search?

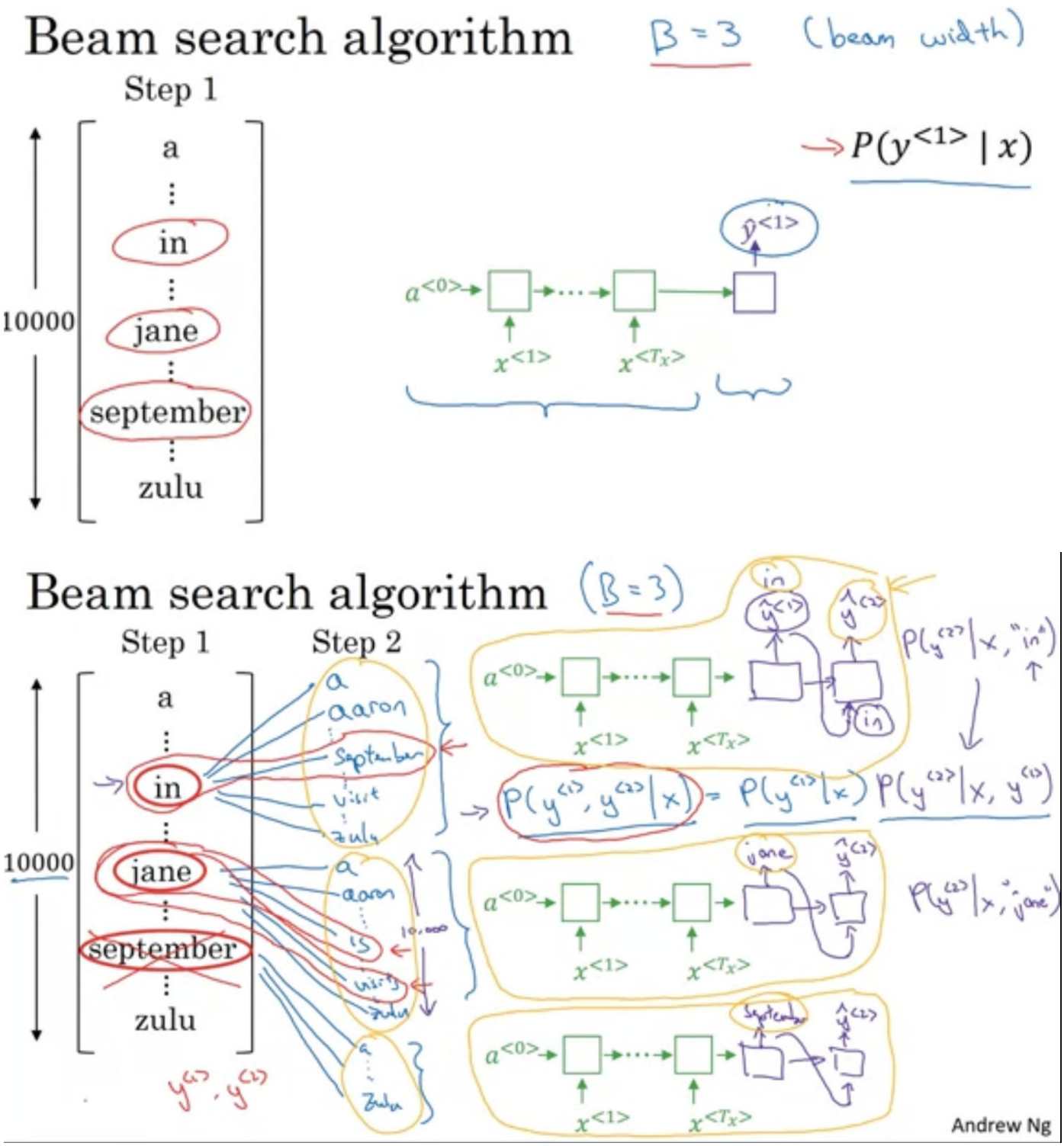
Greedy search is to pick the most likely words every time.



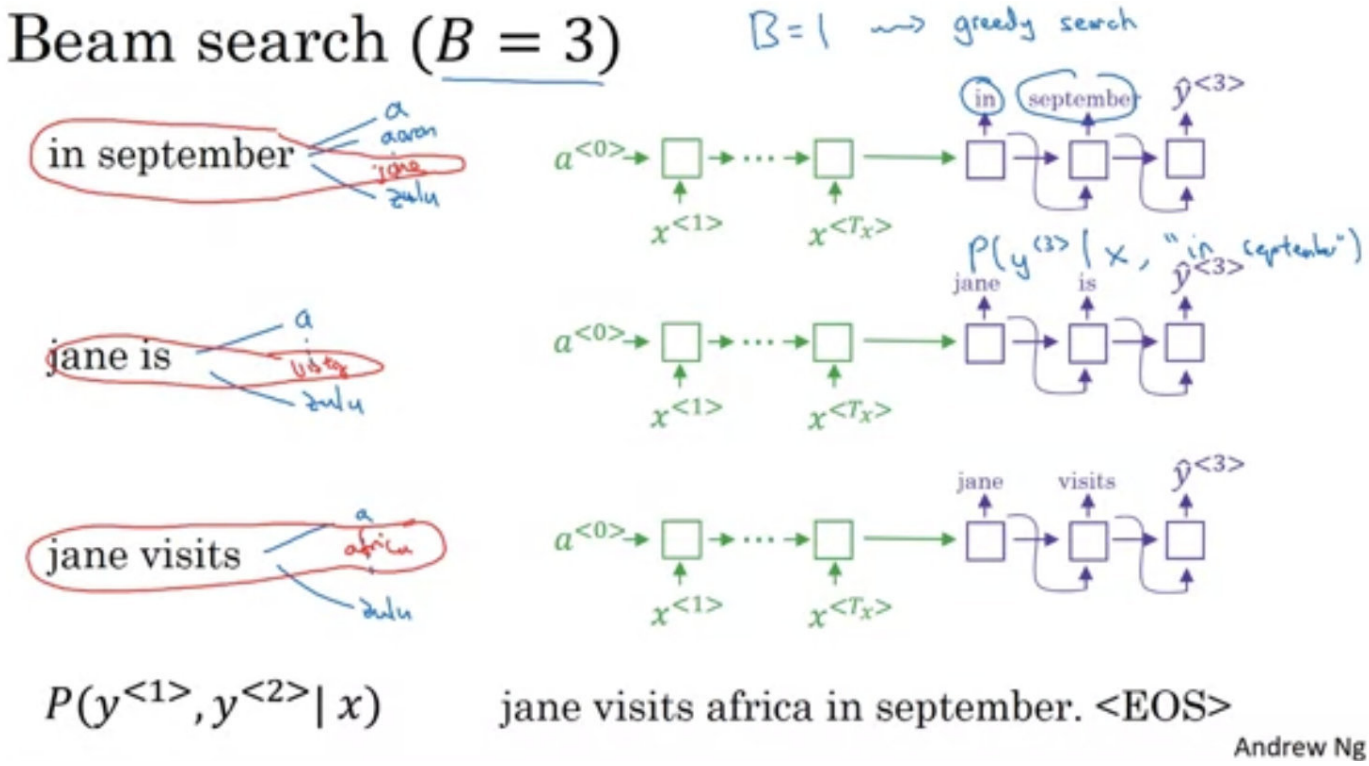
The greedy search may ends up resulting in a **less optimal**, or more **verbose** sentence. Moreover, it is not always optimal to just pick one word at a time. If our vocabulary contains 10,000 words, there will be $10,000^{10}$ possible sentences that are 10 words long in total, it is impossible to rate them all. So, the most common thing to do is use **an approximate search algorithm**. The approximate search algorithm will try to pick the sentence y that maximizes the conditional probability.

Beam search algorithm

The beam search algorithm has a parameter called **beam width** deboted by B. Instead of picking only the one most likely words like gready search dose, beam search can consider **multiple alternatives** accoding to beam width.



$$P(A, B|C) = \frac{P(A, B, C)}{P(C)} = \frac{P(A, B, C)}{P(B, C)} \frac{P(B, C)}{P(C)} = P(A|B, C)P(B|C)$$



Notice that if the beam width is set to be equal to one, then the beam search becomes the greedy search algorithm.

Refinements to beam search

Recall that the objective is to

$$\arg\max_{y^{\langle 1\rangle},\cdots,y^{\langle T_y\rangle}} P(y^{\langle 1\rangle},\cdots,y^{\langle T_y\rangle}|x)$$

Notice that

$$P(y^{\langle 1\rangle},\cdots,y^{\langle T_y\rangle}|x) = P(y^{\langle 1\rangle}|x)P(y^{\langle 2\rangle}|x,y^{\langle 1\rangle})\cdots P(y^{\langle T_y\rangle}|x,y^{\langle 1\rangle},\cdots,y^{\langle T_y-1\rangle})$$

So, the objective function is

$$\arg\max_{y}\prod_{t=1}^{T_y} P(y^{\langle t\rangle}|x,y^{\langle 1\rangle},\cdots,y^{\langle t-1\rangle})$$

Notice that in the objective function, these probabilities are all mumbers less than 1, and multiplying a lot of them will result in a very tiny number, which can result in **numerical underflow**, meaning that it is too small for the floating part represenation in your computer to store accurately.

So, inpractice, we will take logs and the objective funtion becomes

$$\arg\max_{y}\sum_{t=1}^{T_y} \log P(y^{\langle t\rangle}|x,y^{\langle 1\rangle},\cdots,y^{\langle t-1\rangle})$$

Length normalization

For both of these objective functions, however, if you have a very **long** sentence, the probability of the sentence will be very **low** (multiply lots of number smaller than 1, or sum over many negative numbers). So these objective function have an undesirable affect that they tend to prefer **short** tanslations.

To make the algorithm works better, we need to add one more term to normalize it:

$$\arg\max_y \frac{1}{T_y} \sum_{t=1}^{T_y} \log P(y^{\langle t \rangle} | x, y^{\langle 1 \rangle}, \cdots, y^{\langle t-1 \rangle})$$

This will significantly reduce the penalty for outputting longer translations.

In practice, instead of dividing by T_y, sometimes we will use a softer approach

$$\arg\max_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log P(y^{\langle t \rangle} | x, y^{\langle 1 \rangle}, \cdots, y^{\langle t-1 \rangle})$$

where normally $\alpha=0.7$.

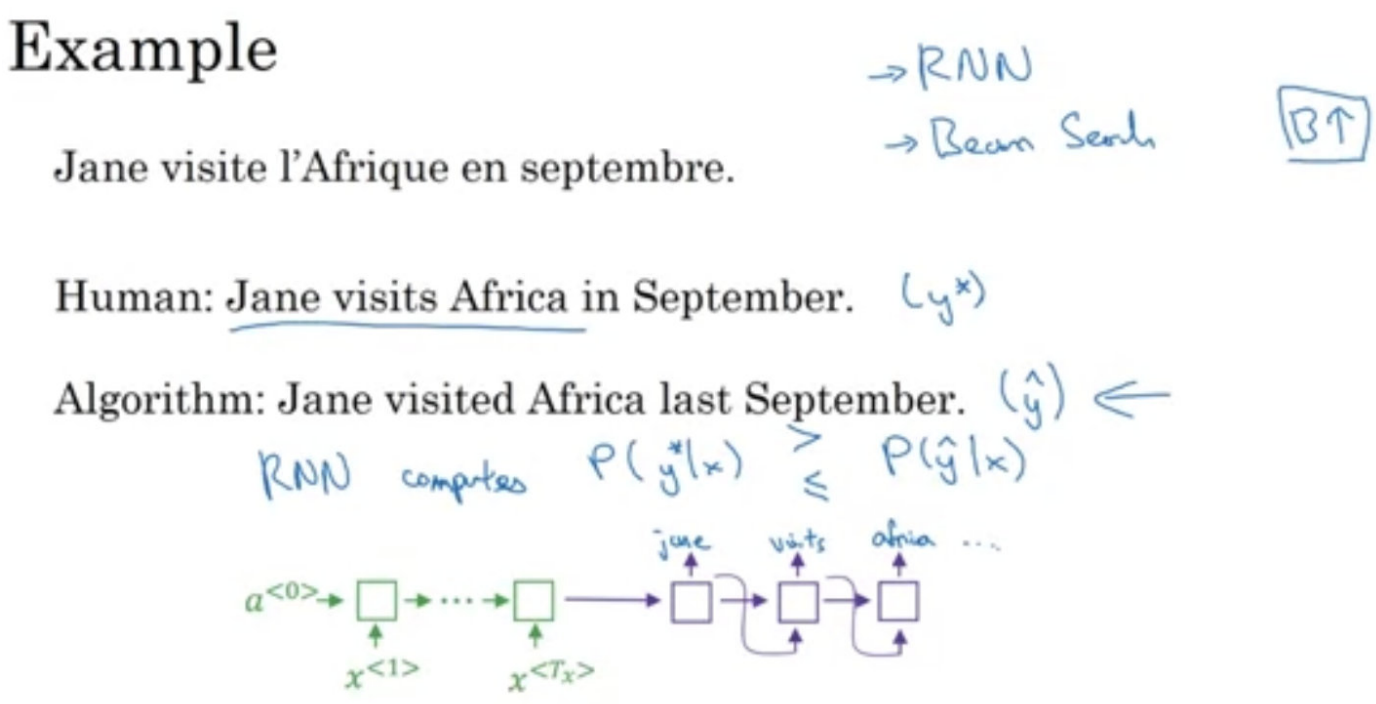
Beam search discussion

How to choose beam width $\mathbf{B}?$

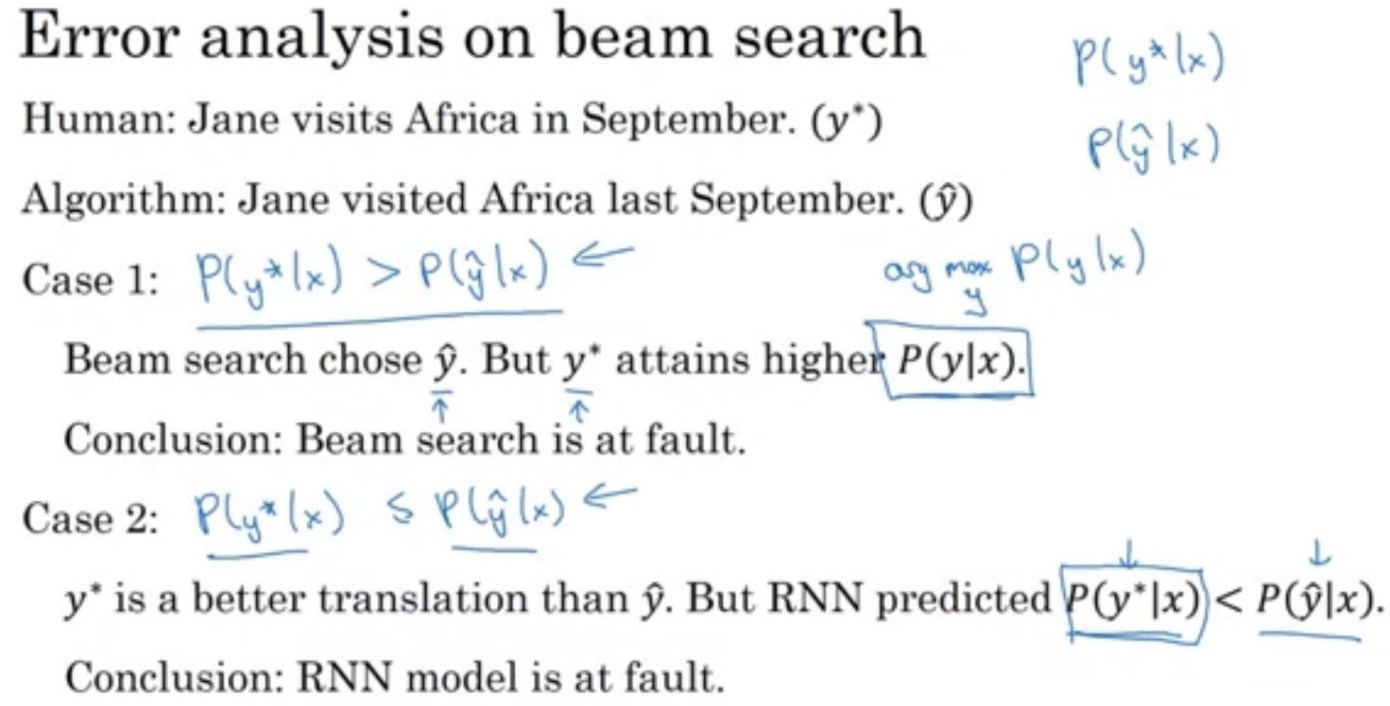
- Large B: better result, but slower
- Small B: worse result, but faster

Unlike **exact search algorithms** like BFS or DFS, Beam Search **runs faster** but is **not guaranteed** to find exact maximum for $\mathop{\arg\max}\limits_yP(y|x)$.

Error analysis in beam search



What you can do is to compute $P(y^*|x)$ and $P(\hat{y}|x)$ and see which one is bigger.

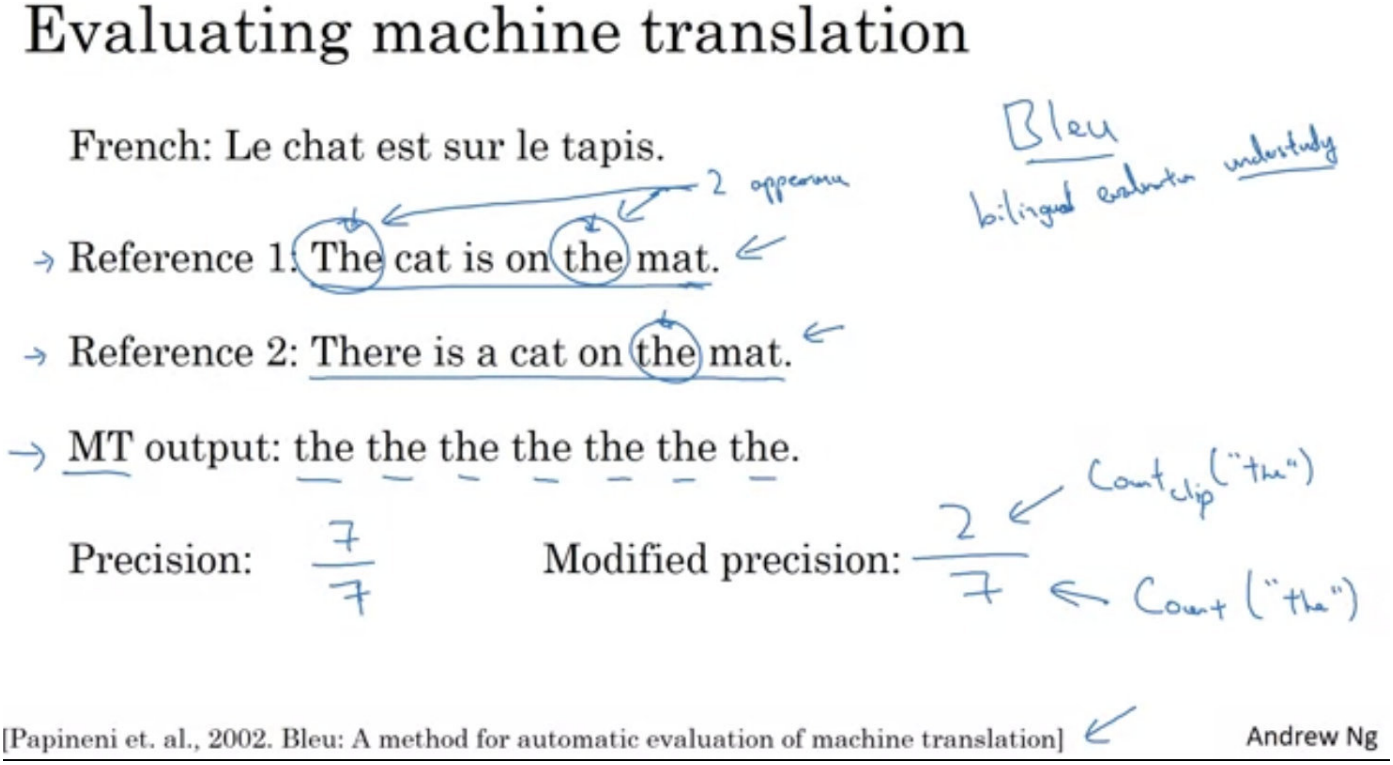


Only when a lot of fractions of faults are due to beam search, then you may need to increase the beam width.

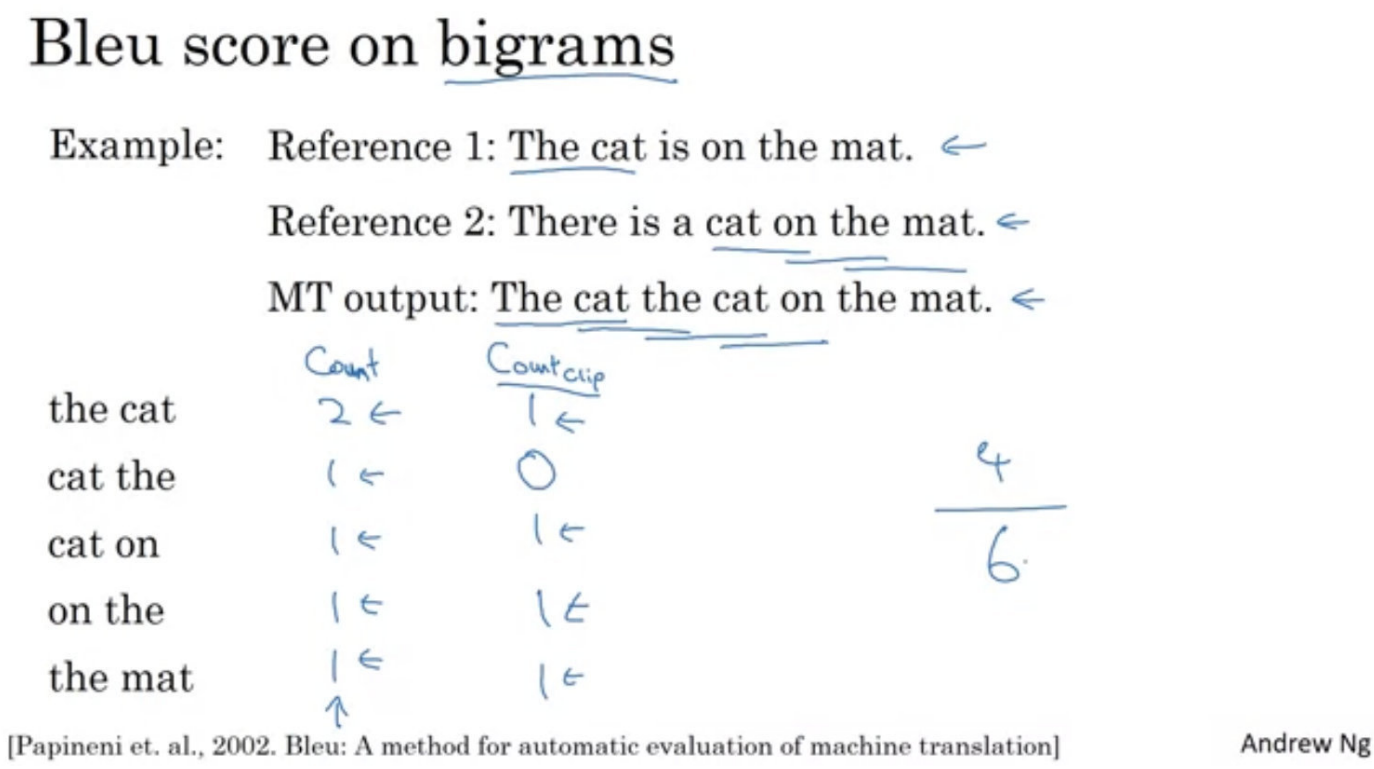
Bleu score

What the Bleu score does is given a machine generated translation, it allows you to automatically compute a score that measures how good is that machine translation.

Bleu stands for **bilingual evaluation understudy**.



[Papineni et. al., 2002. Bleu: A method for automatic evaluation of machine translation]



For unigram (one word), the precision P_1 (1 stands for unigram) is given by:

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

For n-gram (n words pair), the precision P_n is given by:

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

Bleu details

p_n = Bleu score on n-grams only

p_1, p_2, p_3, p_4

Combined Bleu score: $BP \exp\left(\frac{1}{4}\sum_{n=1}^{4} p_n\right)$

BP = brevity penalty

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length}/\text{reference\_output\_length}) & \text{otherwise} \end{cases}$$

exp(1 − reference_output_length/MT_output_length)

[Papineni et. al., 2002. Bleu: A method for automatic evaluation of machine translation]

Andrew Ng

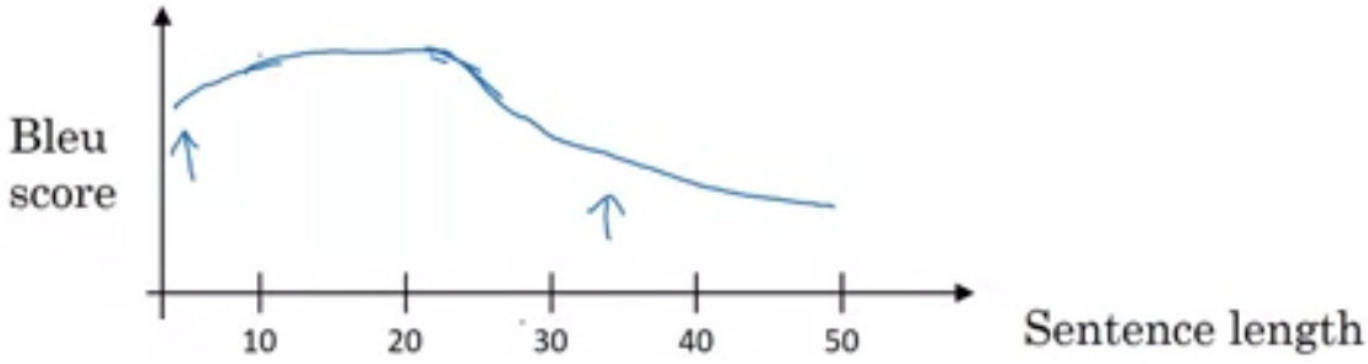
BP stands for **brevity penalty**.

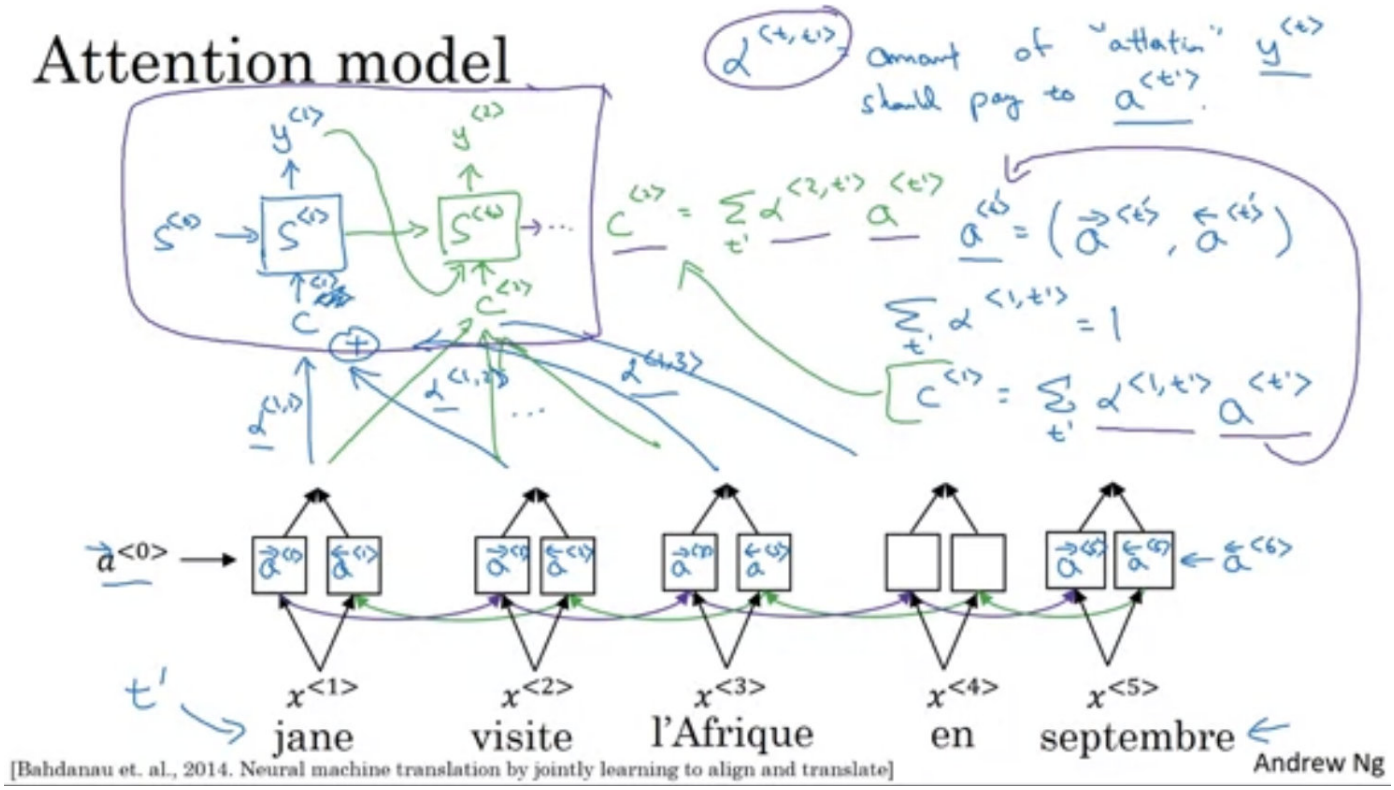
Today, BLEU score is used to evaluate many systems that **generate text**, such as machine translation systems, as well as image captioning systems where you would have a neural network generated image caption. It is a useful single number evaluation metric to use when you want your algorithm to generate text, and you want to see whether it has similar meaning as the reference text written by human.

Attention model

The problem of long sequences

It is difficult for a network to memorize long sentence.





We use

$$a^{\langle t' \rangle} = \begin{bmatrix} \overrightarrow{a}^{\langle t' \rangle} \\ \overleftarrow{a}^{\langle t' \rangle} \end{bmatrix}$$

to represent feature vector for time step t'. For gennerating the translation, we use a single direction RNN with state s. This single RNN will generate the translation y. It also has a input called **context** c for every time step.

$$c^{\langle t \rangle} = \sum_{t'} \alpha^{\langle t,t' \rangle} a^{\langle t' \rangle}$$

where α is the **attention parameter**.

$$\alpha^{\langle t,t' \rangle} = \text{amount of attention } y^{\langle t \rangle} \text{ should pay to } a^{\langle t' \rangle}$$

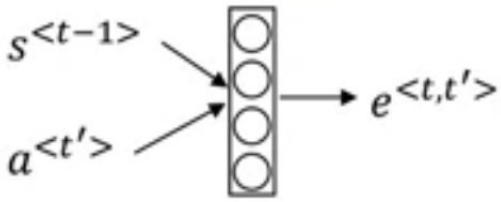
The formula to compute $\alpha^{\lang t,t'\rang}$ is

$$\alpha^{\langle t,t' \rangle} = \frac{\exp(e^{\langle t,t' \rangle})}{\sum_{t'=1}^{T_x} \exp(e^{\langle t,t' \rangle})}$$

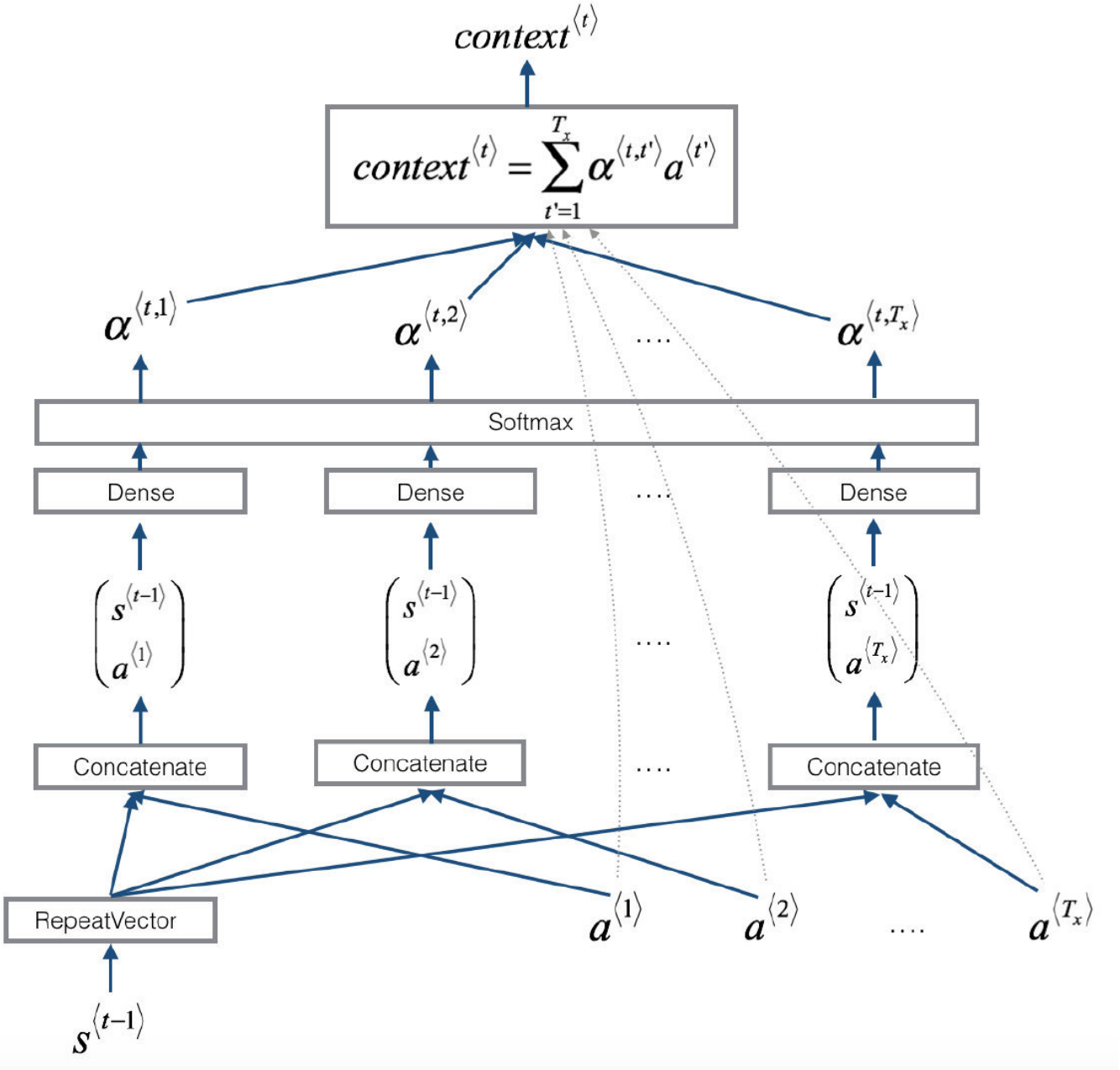
Using the softmax prioritization is just to ensure

$$\sum_{t'} \alpha^{\langle t,t' \rangle} = 1$$

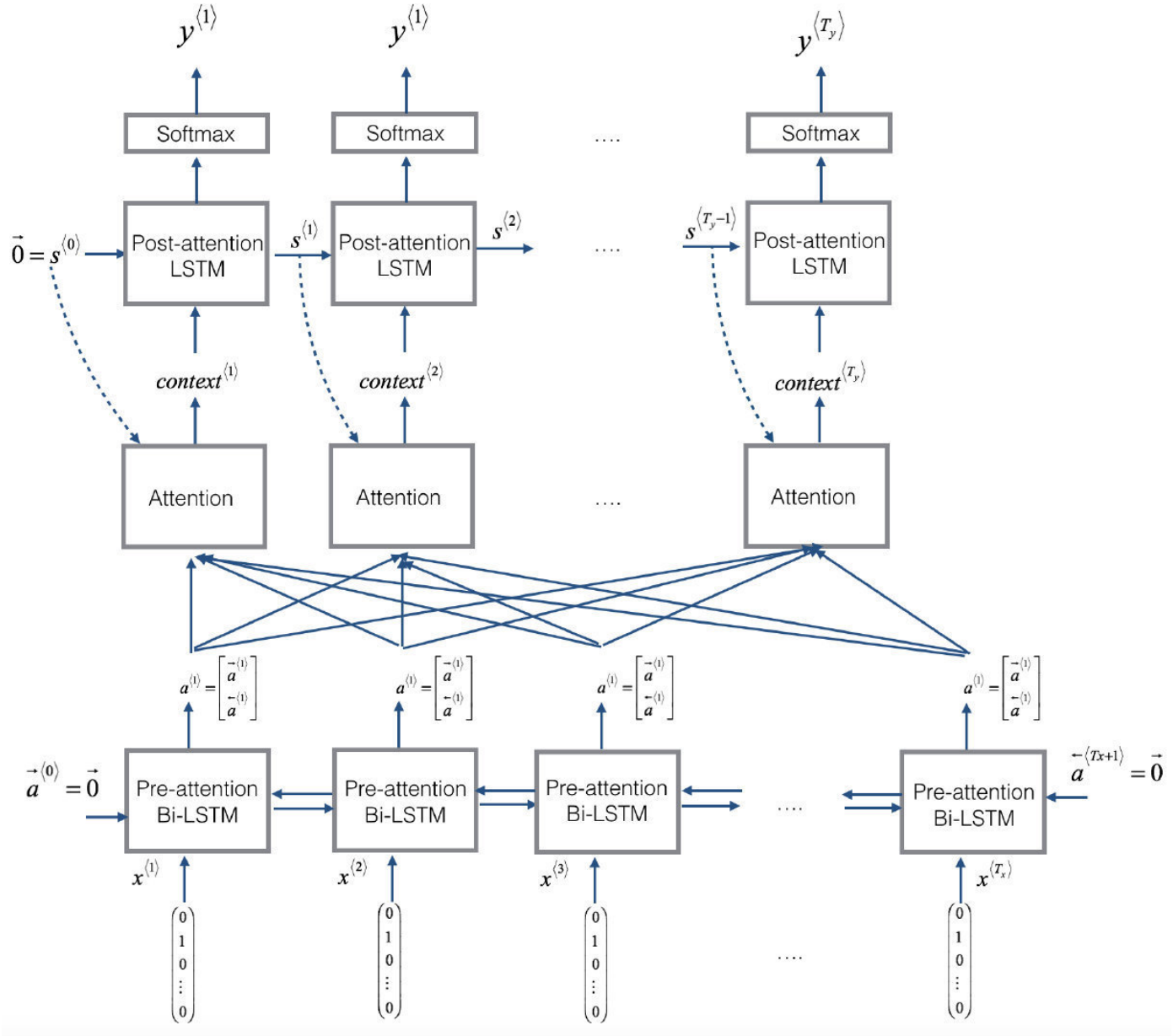
e is called the "**energies**" variable, one way to compute the factors e is to use a small neural networs (usually just one hidden layer) with **softmax** activation as follow:



The intuition is, if you want to decide how much attention to pay to the $\large a^{\lang t'\rang}$, it should depend the most on is what is your own hidden state activation from the previous time step, which is $\large s^{\lang t-1\rang}$, and also the features of original words, which is $\large a^{\lang t'\rang}$. The following picture shows how the context is generated.



The whole attention model is as follow:



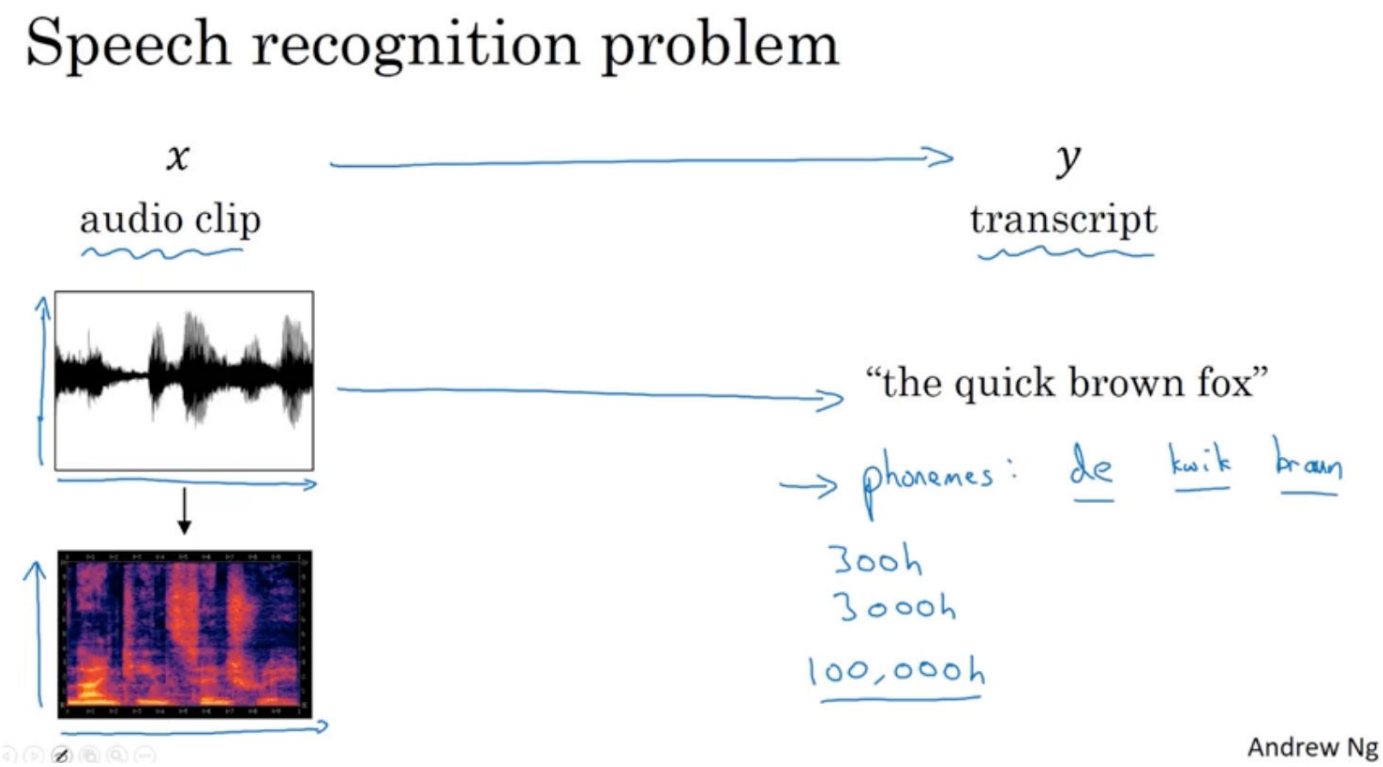
Here are some properties of the model that you may notice:

Pre-attention and Post-attention LSTMs on both sides of the attention mechanism

- There are two separate LSTMs in this model (see diagram on the left): pre-attention and post-attention LSTMs.
- *Pre-attention* Bi-LSTM
  - The pre-attention Bi-LSTM goes through T_x time steps
- *Post-attention* LSTM
  - The post-attention LSTM goes through T_y time steps.
- The post-attention LSTM passes the hidden state $s^{\langle t \rangle}$ and cell state $c^{\langle t \rangle}$ from one time step to the next.

One **downside** to this algorithm is that it does take **quadratic time** or **quadratic cost** to run this algorithm. Because the total number of attention parameters is $T_x \times T_y$.

Speech Recognition



The first figure plots the audio clip, the horizontal axis is time, and the vertical axis is actually air pressure. The common prepocessing step is to convert the audio clip in to a **spectrogram**. The horizontal axis of the spectrogram is time, and the vertical axis is the frequency, and the intensity of different color shows the amount of energy. It shows "how loud is the sound at different frequncy at different time".

What really is an audio recording?

- A microphone records little variations in air pressure over time, and it is these little variations in air pressure that your ear also perceives as sound.
- You can think of an audio recording as a long list of numbers measuring the little air pressure changes detected by the microphone.
- We will use audio sampled at 44100 Hz (or 44100 Hertz).
    - This means the microphone gives us 44,100 numbers per second.
    - Thus, a 10 second audio clip is represented by 441,000 numbers (= $10 \times 44,100$).

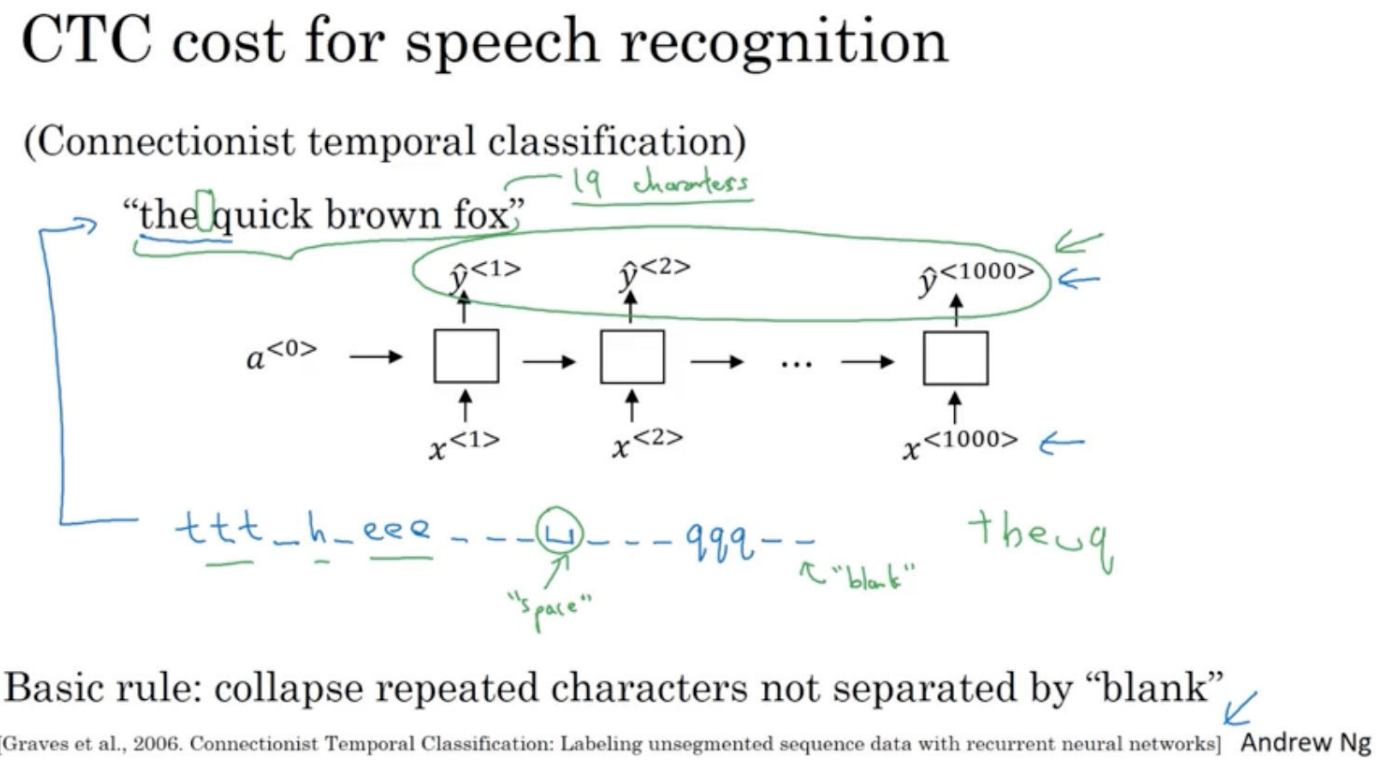
Spectrogram

- It is quite difficult to figure out from this "raw" representation of audio whether the word "activate" was said.
- In order to help your sequence model more easily learn to detect trigger words, we will compute a *spectrogram* of the audio.
- The spectrogram tells us how much different frequencies are present in an audio clip at any moment in time.
- If you've ever taken an advanced class on signal processing or on Fourier transforms:
    - A spectrogram is computed by sliding a window over the raw audio signal, and calculating the most active frequencies in each window using a Fourier transform.

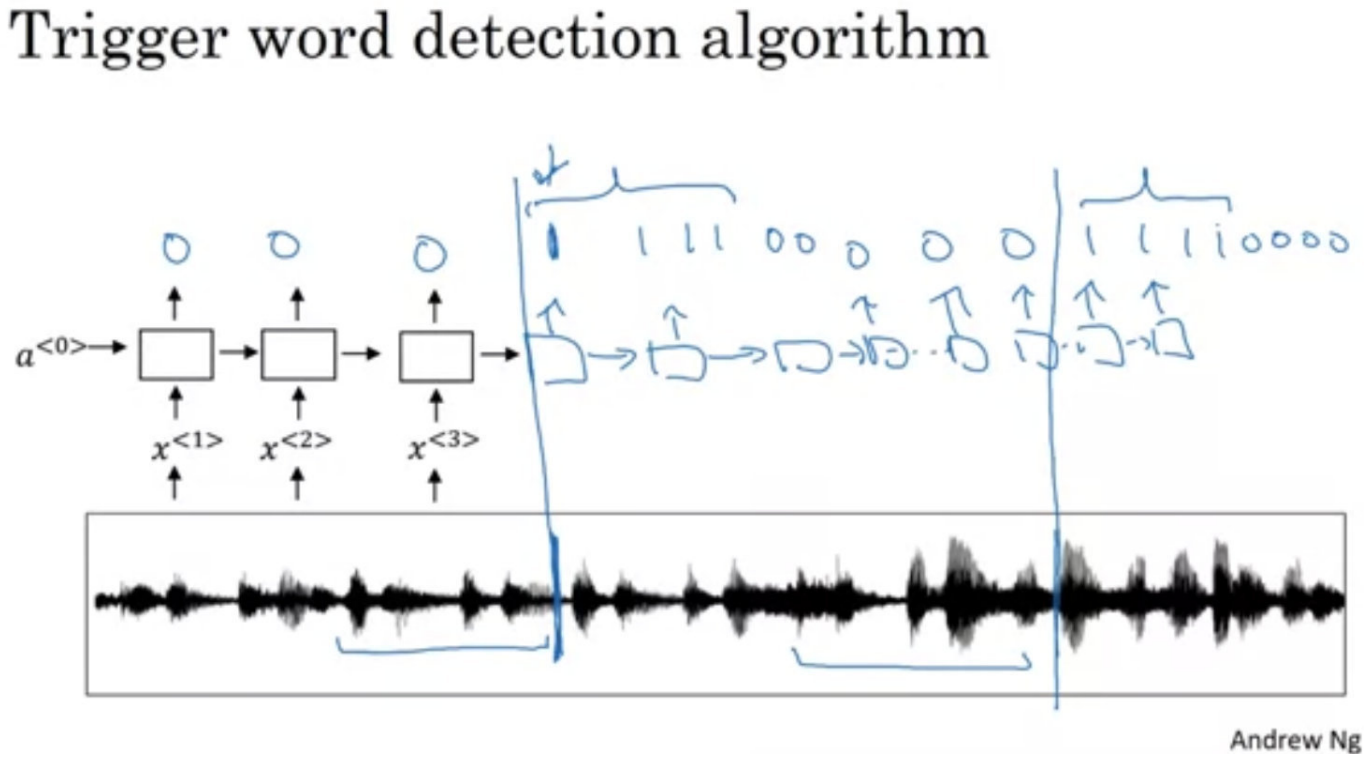
CTC cost for speech recognition

CTC stands for **Conectionist Temporal Classification**.

Basic rule: collapse repeated characters not separated by "blank" (not space).



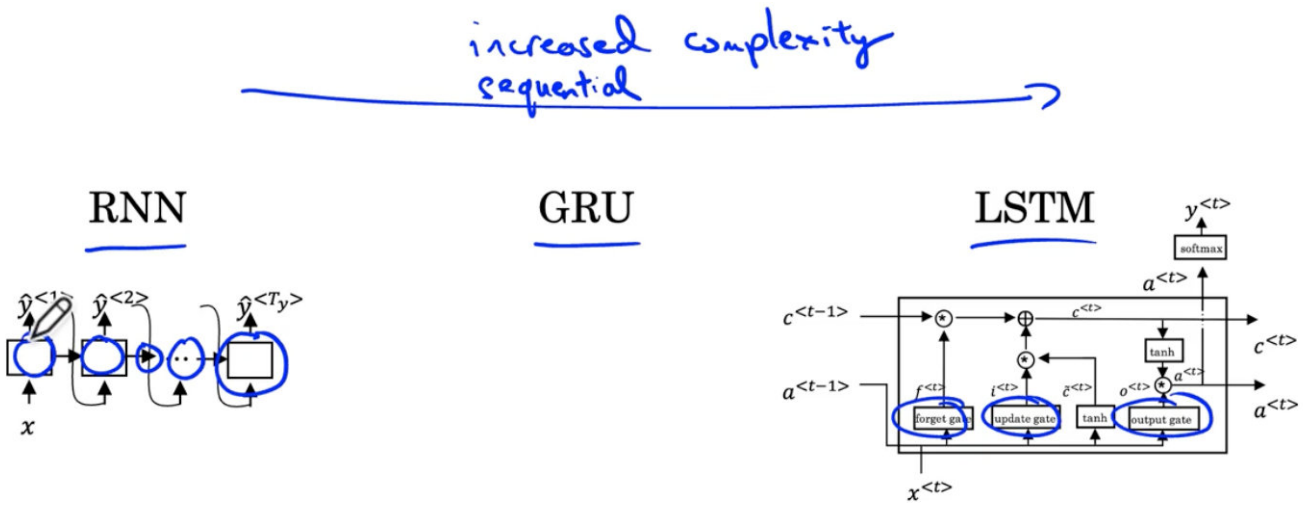
Trigger word detection



Transformers

Transformer network motivation

As we move from RNNs to GRU to LSTM, the models became more and more complex. And all of these models are still **sequential models** in that they ingested the input, maybe one word or one token at a time. And so, each unit was like a **bottleneck** to the flow of information. Because to compute the output of the final unit, for example, you have to compute the outputs of all of the unit that come before.



The transformer architecture allows you to run a lot of these computations for an entire sequence **in parallel**. The major innovation of the transformer architecture is combining the use of **attention based representations** and a **CNN style of processing**. The two key ideas we will go through are:

- Self-attention
- Multi-Head Attention

Self-attention

Self-attention enable us to use attention with a style more like CNNs.

First, we need to compute an **attention-based representation**:

$$A(q, K, V) = \text{attention-based vector representation of a word}$$

We will compute this for each word to get $A^{\langle t \rangle}$. The intuition of $A^{\langle t \rangle}$ is that it will look at the surrounding words to try to figure out the context of this word and find the most appropriate representation for this word.

Recall the RNN attention formula:

$$\alpha^{\langle t,t' \rangle} = \frac{\exp(e^{\langle t,t' \rangle})}{\sum_{t'=1}^{T_x} \exp(e^{\langle t,t' \rangle})}$$

The transformers attention formula is similar:

$$A(q, K, V) = \sum_i \frac{\exp(q \cdot k^{\langle i \rangle})}{\sum_j \exp(q \cdot k^{\langle j \rangle})} v^{\langle i \rangle}$$

The main difference is that for each word you have three values called

$q^{\langle t \rangle}$: query

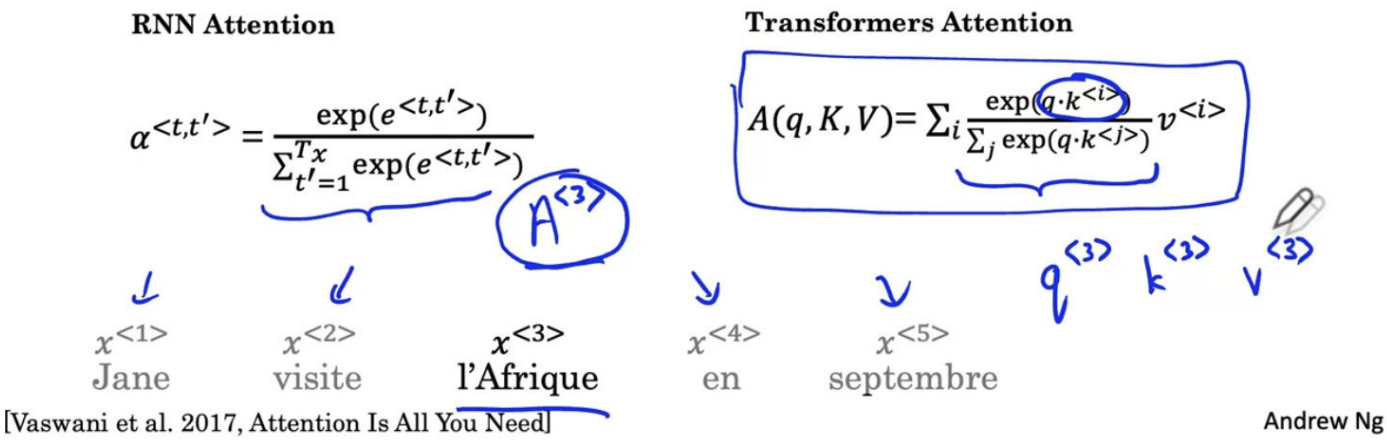
$k^{\langle t \rangle}$: key

$v^{\langle t \rangle}$: value

Self-Attention Intuition

$A(q, K, V)$= attention-based vector representation of a word

↳ calculate for each word $A^{<1>}, \ldots, A^{<5>}$



[Vaswani et al. 2017, Attention Is All You Need]

Andrew Ng

Now, let's go through how to get $A^{\langle t \rangle}$

First, we will associate each of words with three values, the **query, key, and value** pairs. If the $x^{\langle t \rangle}$ is the word embedding, the way that $q^{\langle t \rangle}$ is computed is as a learned matrix:

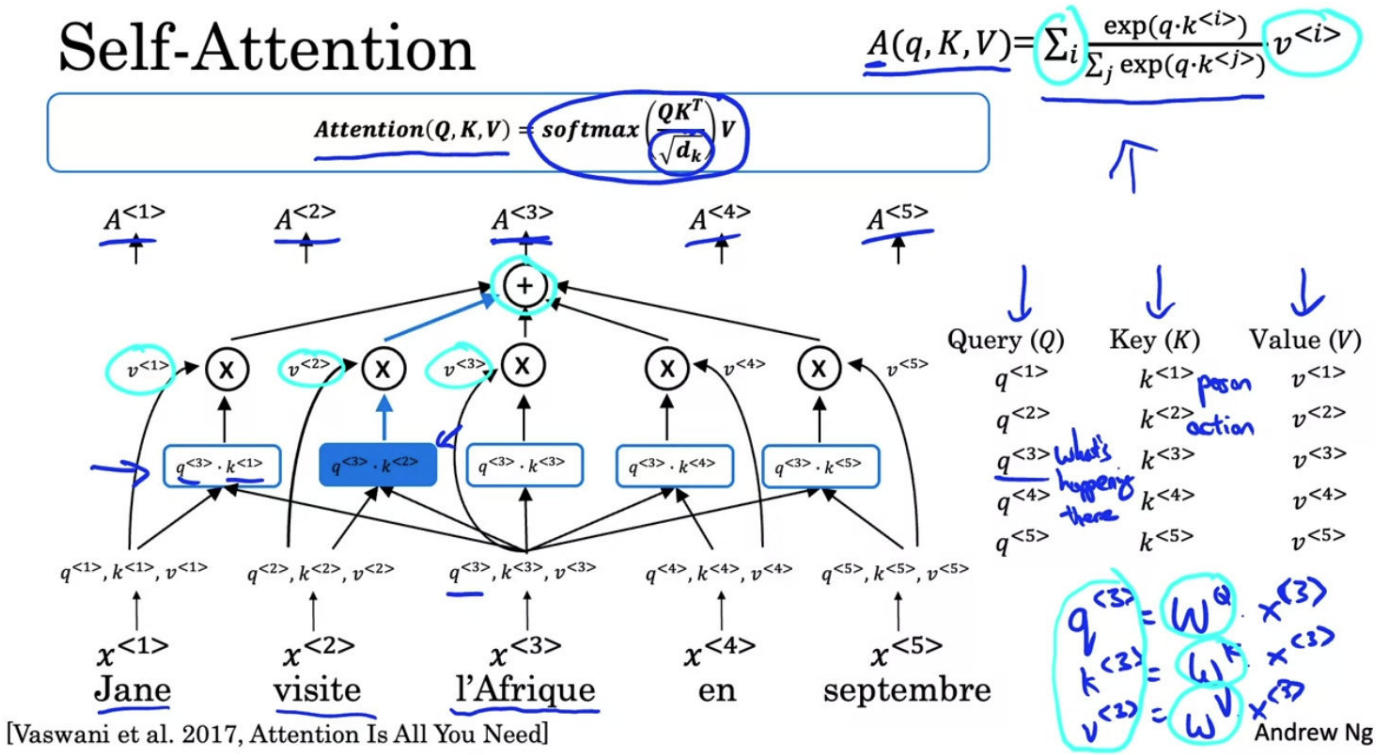
$$q^{\langle t \rangle} = W^Q \cdot x^{\langle t \rangle}$$

And similarly for the key and value,

$$k^{\langle t \rangle} = W^K \cdot x^{\langle t \rangle}$$

$$v^{\langle t \rangle} = W^V \cdot x^{\langle t \rangle}$$

These matrices, W^Q, W^K, W^V, are parameters of this learning algorithm, and they allow you to pull off these query, key, and value vectors for each word.



One intuition behind the intent of these query, key, value vectors

$q^{\langle t \rangle}$ may represent a question like, what is happening to the word $x^{\langle t \rangle}$ when computing $A^{\langle t \rangle}$? In the above example, $q^{\langle 3 \rangle}$ may represent what is happening to l'Afrique (Africa) when we compute $A^{\langle 3 \rangle}$? Dose l'Afrique represent a destination of traveling or the second largest continent in this sentence?

Then, we are goint to computer the inner product between $q^{\langle t \rangle}$ and $k^{\langle 1 \rangle}$, and this will tell us how good is an answer word $x^{\langle 1 \rangle}$ to the question what is happening to the word $x^{\langle t \rangle}$? For example, $q^{\langle 3 \rangle} \cdot k^{\langle 2 \rangle}$ will tell us how good is "visite" an answer to the question of what is happening to l'Afrique (Africa), and so on for the other words in the sequence. The goal of this operation is to **pull off the most imformation that is needed** to help us compute the most useful representation $A^{\langle t \rangle}$.

Next, for the above example, you may find that $q^{\langle 3 \rangle} \cdot k^{\langle 2 \rangle}$ gives you largest value, this may suggest that "visite" gives you **the most relevant contexts** for the question of what is happening to l'Afrique (Africa) (view Africa as a destination for a visit).

In summary for intuition:

- Q: interesting questions about the words in a sentence.
- K: qualities of words given a Q.
- V: specific representations of words given a Q.

Then we will take softmax over these inner products, and multiply them with $v^{\langle t \rangle}$, which is value for word $x^{\langle t \rangle}$. Finally, we sum them up to get $A^{\langle t \rangle}$.

The key advantage of this representation is that the word of $x^{\langle t \rangle}$ is not some fixed word embedding, instead it lets the self-attention mechanism realize that this word is actually associated with another word (Africa is the destination of a visit), and thus computer a richer, more useful representation for this word.

The notation used in literature is like this:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- Q is the matrix of queries
- K is the matrix of keys
- V is the matrix of values
- d_k is the dimension of the keys, which is used to scale everything down so the softmax doesn't explode

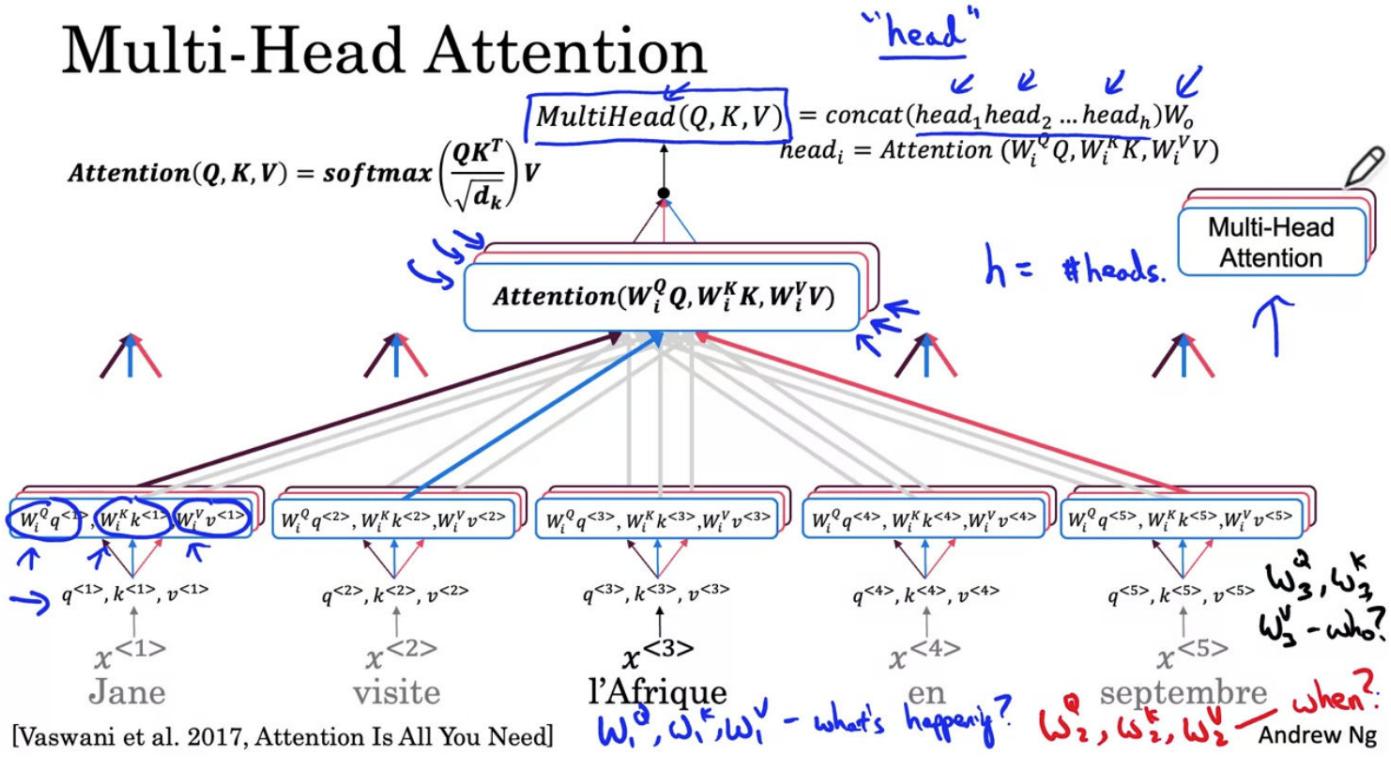
It is the vetorized representation of

$$A(q, K, V) = \sum_i \frac{\exp(q \cdot k^{\langle i \rangle})}{\sum_j \exp(q \cdot k^{\langle j \rangle})} v^{\langle i \rangle}$$

The term $\sqrt{d_k}$ in the denominator is just to scale the dot product so it doesn't explode. The another name for this type of attention is called **the scaled dot-product attention**.

Muli-head attention

Each time we calculate self attention for a sequence is called a **head**.



To calculate the multiple self-attention, we firstly multiply $q^{\langle t \rangle}, k^{\langle t \rangle}, v^{\langle t \rangle}$ with weight matrices W_i^Q, W_i^K, W_i^V, the subscript i represents this is the weight for $head_i$. In literature, we use h to denote the number of heads. Using $h = 8$ heads is common in the literature.

$$h = \#heads$$

For the sake of intuition, we can think of W_i^Q, W_i^K, W_i^V as being learned to help ask and answer different questions (i^{th} question in particular). For example, W_1^Q, W_1^K, W_1^V are learned to help with asking and answering "what is happenting to the word 'Africa'", and the word "visite" gives the best answer (blue arrow). Similarly, W_2^Q, W_2^K, W_2^V are learned to help with asking and answering the second question like "when is happening", so the inner product between the September key and l'Afrique query will have the highest value (red arrow). And W_3^Q, W_3^K, W_3^V may help

with third question "who has something to do with Africa", and this time Jane's value will have the greatest weight in this representation.

After calculating the computation for these heads, the concatenation of these values is used to compute the output of the multi-head attention.

$$MultiHead(Q, K, V) = concat(head_1, head_2, \cdots, head_n)W_0$$

where

$$head_i = Attention(W_1^Q Q, W_1^K K, W_1^V V)$$

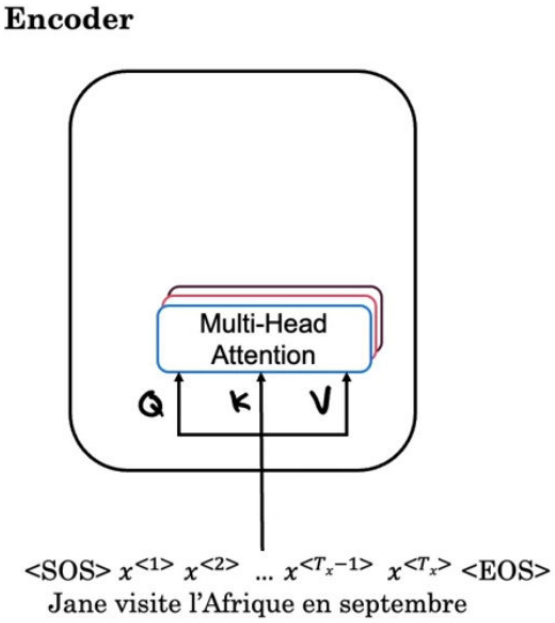
Recall that

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

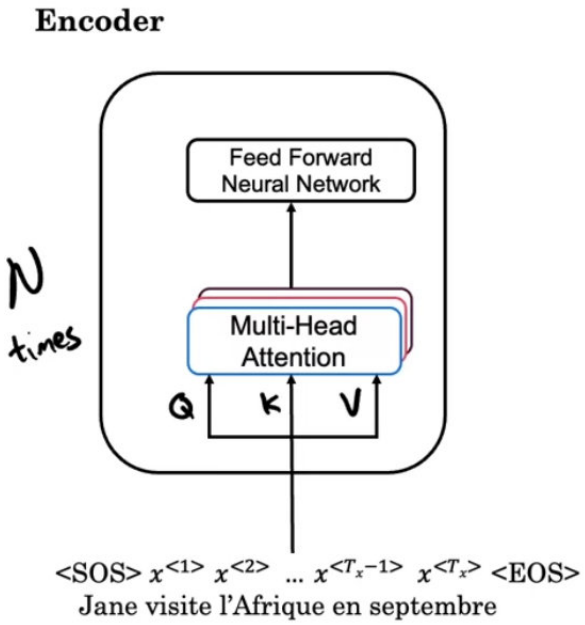
In practice, these multi-head attentions are computed in parallel, because no one head's value depends on the value of any other head.

Transformer network

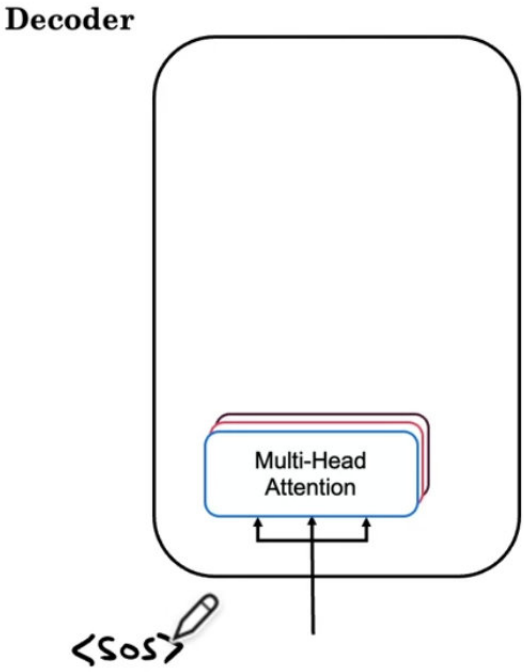
The first step in the transformer is the embeddings of the input sentence get def into an **encoder** block, and the Q, K, V are computed from the embeddings and weight matrices W.



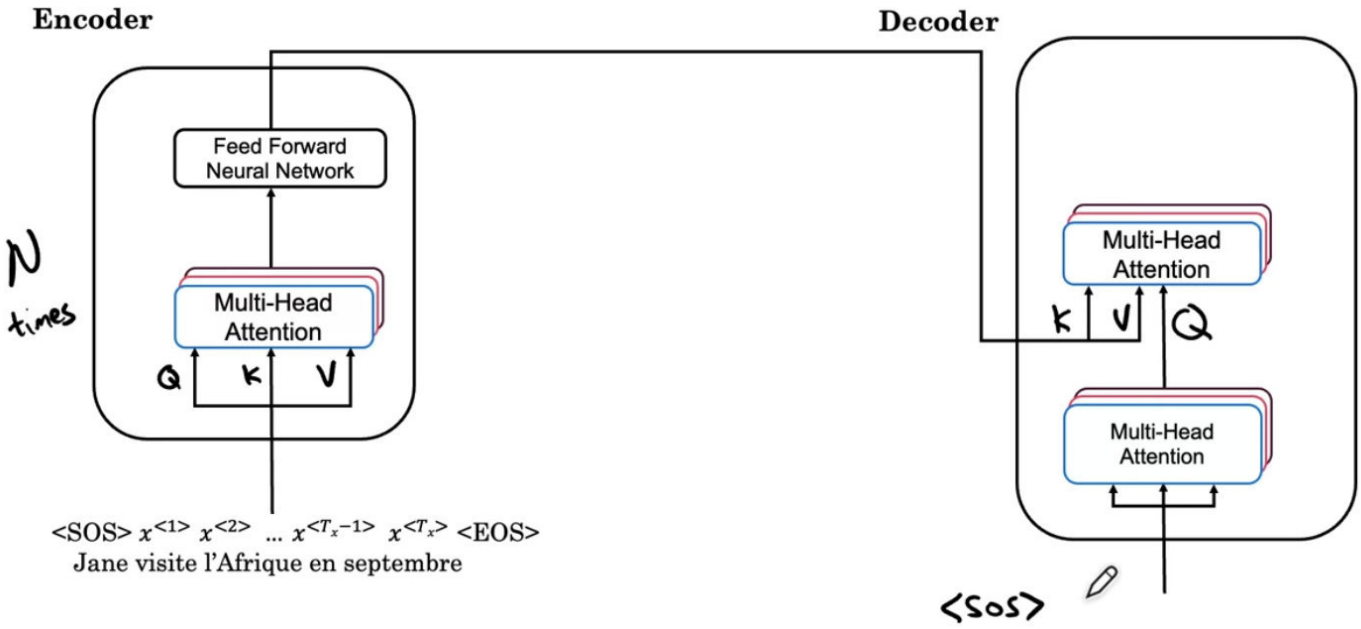
The multi-head attention layer then produces a matrix that can be passed into a feed forward neural network, which helps determine what interesting features there are in the sentence. In the original paper, this encoding block is repeated N times, typically $N = 6$.



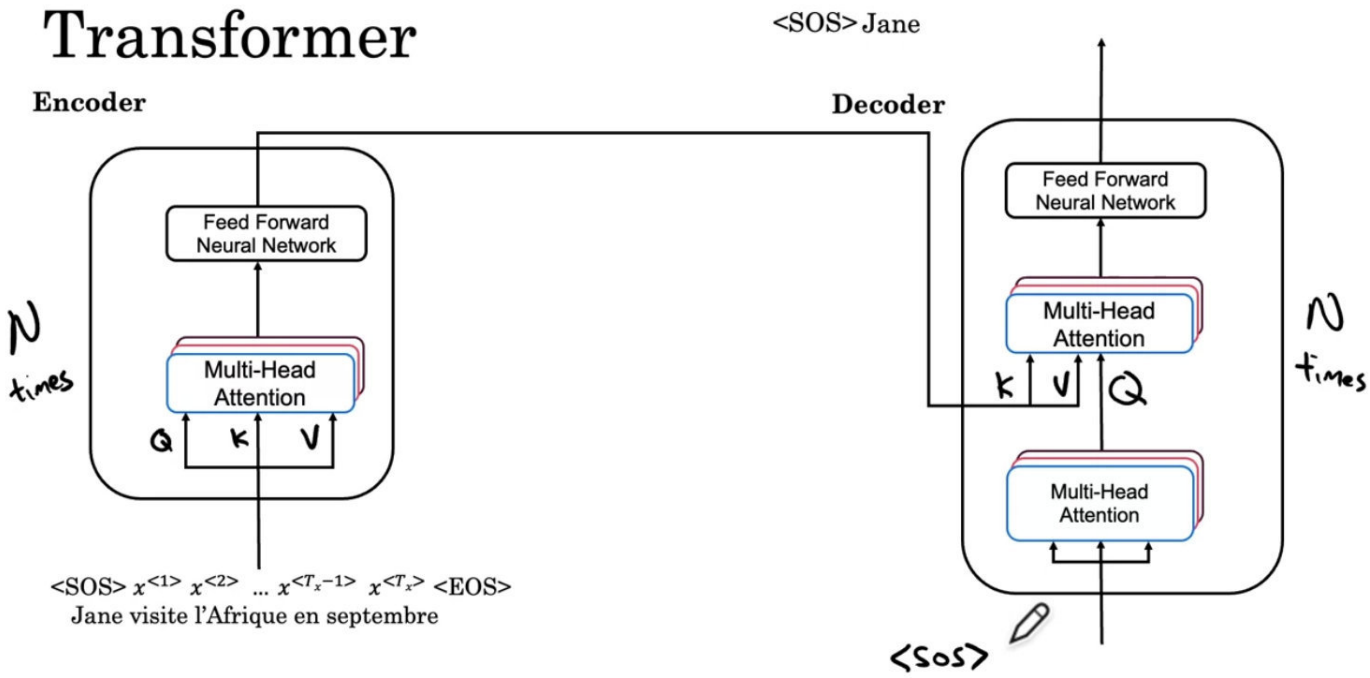
Then the output of the encoder is fed into a **decoder** block. The job of the decoder is to output the English translation. At every step, the decoder block will input the first few words that were generated by decoder and compute Q, K, V for the first multi-head attention block.



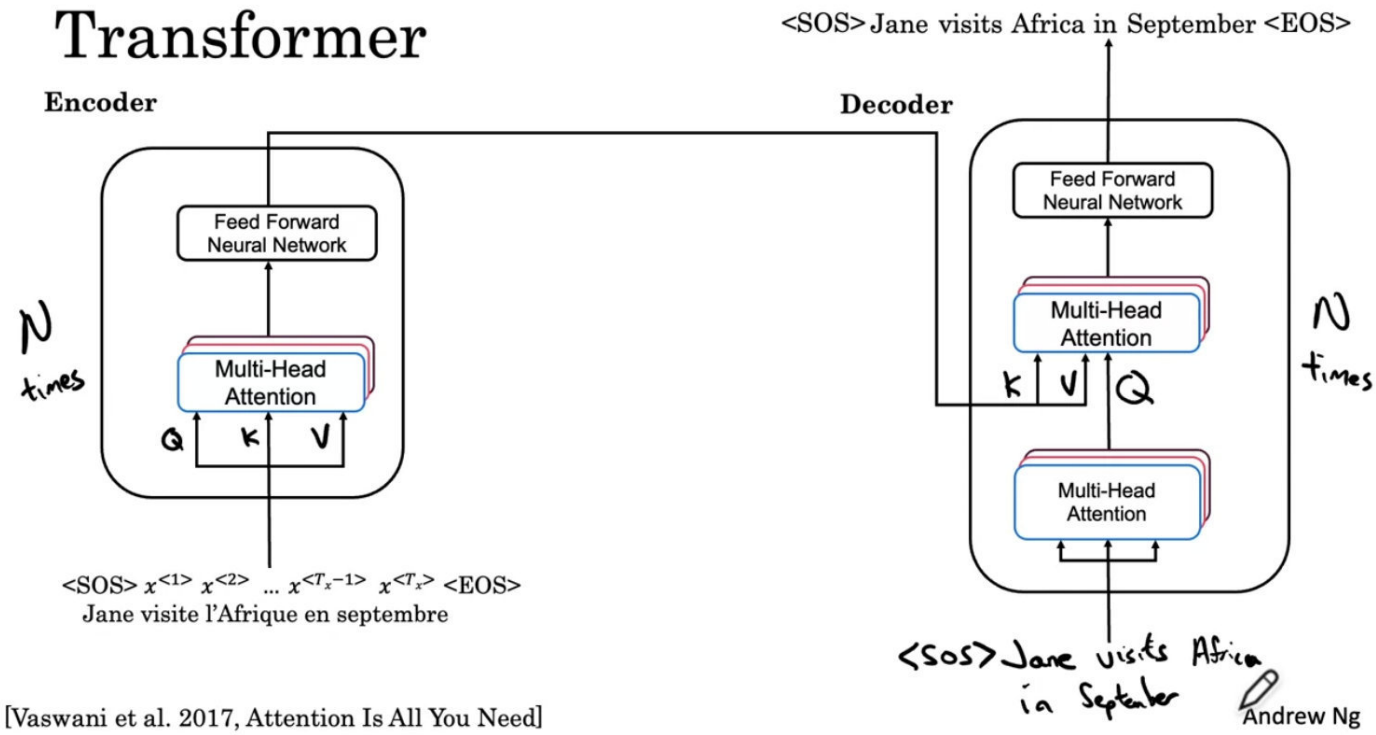
Then the first multi-head attention block will generate Q matrix for next multi-head attention block, and the matrix K,V are derived from the output of the encoder. The intuition for this step is that, the input of the first multi-head attention block is what you have translated of the sentence so far. And this will ask the query to say "what is the next word?", and then we will pull context from K and V which are translated from the French to then try to decide the next word.



The outputs of second multi-head attention block will feed to a feed forward neural network, and this decoder block is also going to be repeated N times. And the job of the feed forward neural network is to predict the next word in the sentence.



And the generated output will be fed to the input as well and predict the next word.



[Vaswani et al. 2017, Attention Is All You Need]

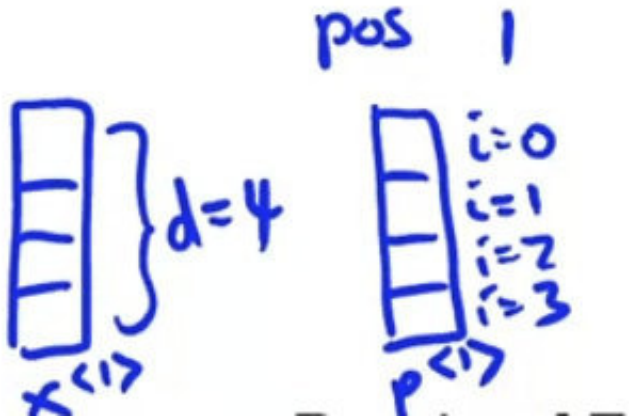
This is the **main idea** of transformer.

Moreover, there is **positional encoding (PE)** of the input. The position within the sentence can be extremely important to translation. However, when you train a Transformer network, you feed your data into the model all at once. While this dramatically reduces training time, there is no information about the order of your data. This is where positional encoding is useful - you can specifically encode the positions of your inputs and pass them into the network using these sine and cosine formulas:

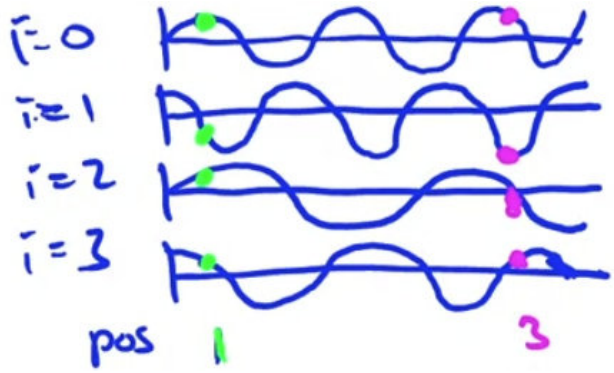
$$PE_{(pos,2i)} = \sin(\frac{pos}{10000^{\frac{2i}{d}}})$$
$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{\frac{2i}{d}}})$$

Where

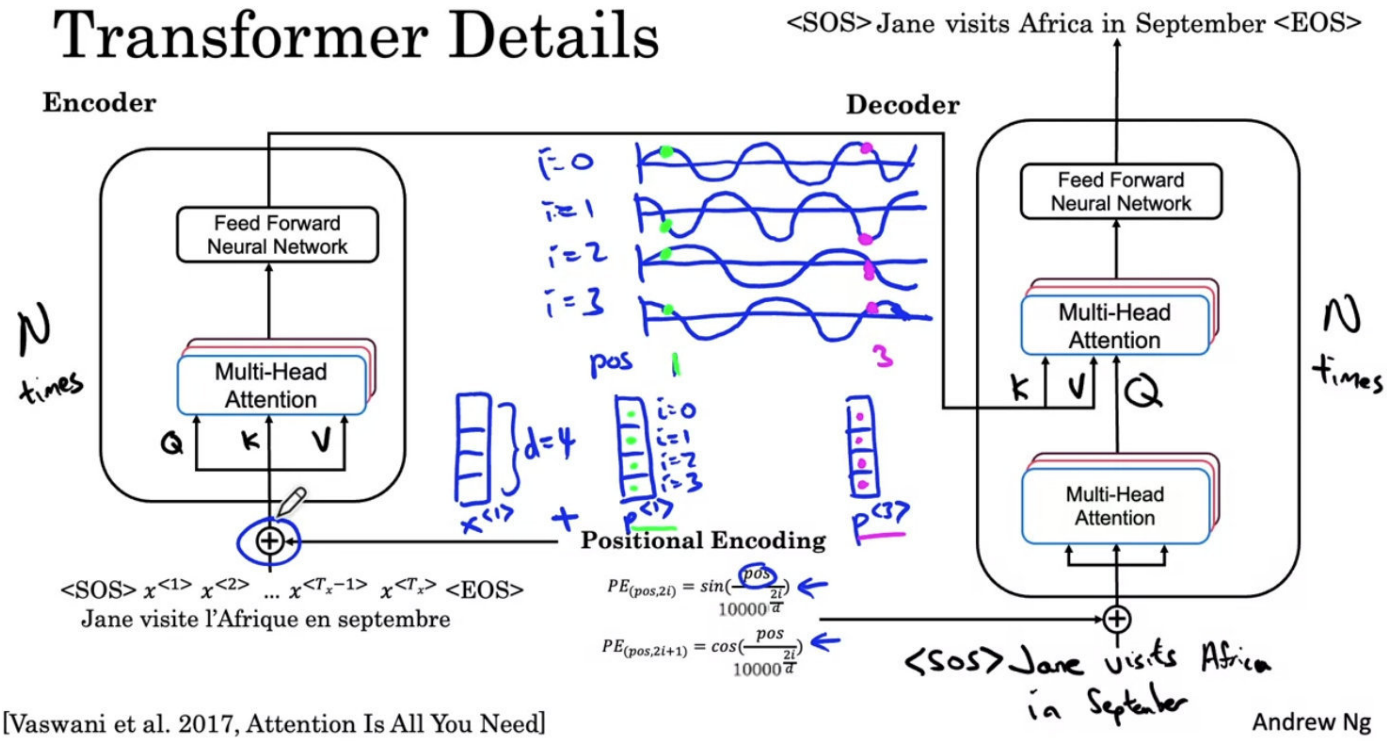
- d is the dimension of the word embbedings (how many features)
- pos is the numerical position of the word ($pos = 1$ for the word Jane).
- i refers to the different dimensions of the encoding
- the positon encoding for word $x^{\langle t \rangle}$ is $p^{\langle t \rangle}$.



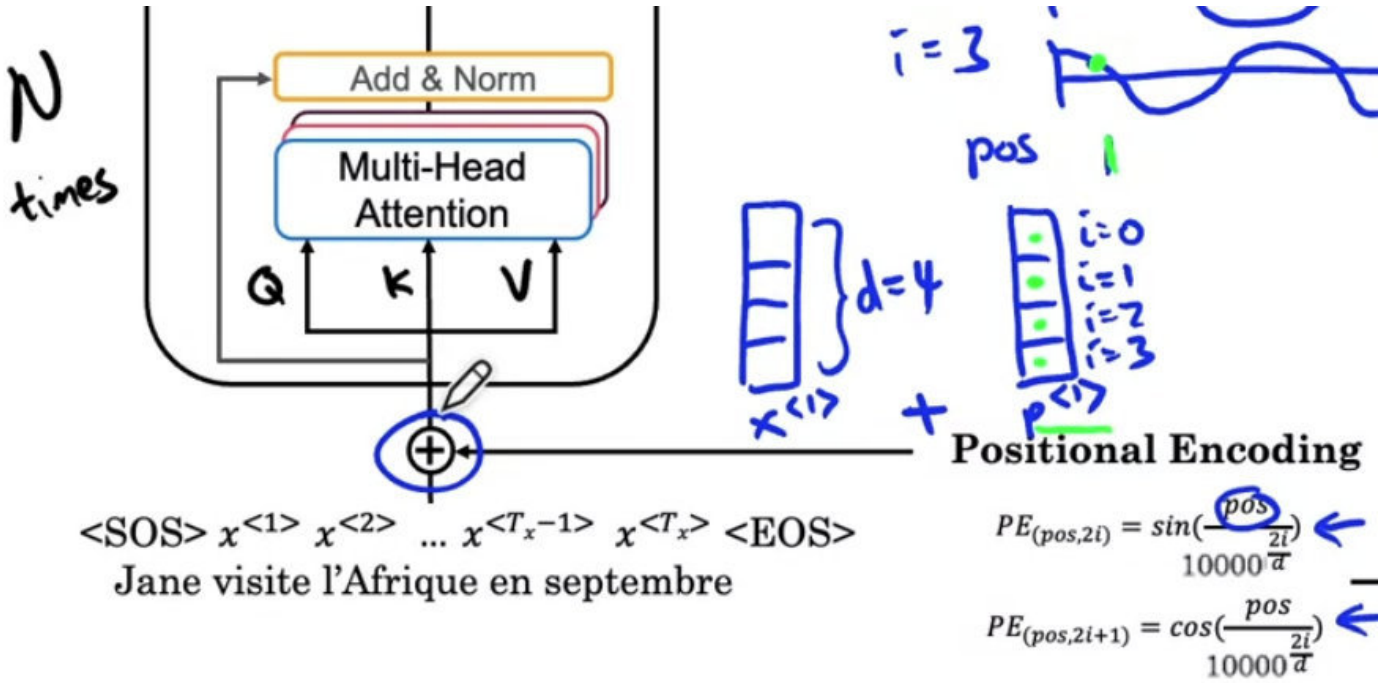
The $q^{\langle t \rangle}$ is unique because the properties of sin and cos function:



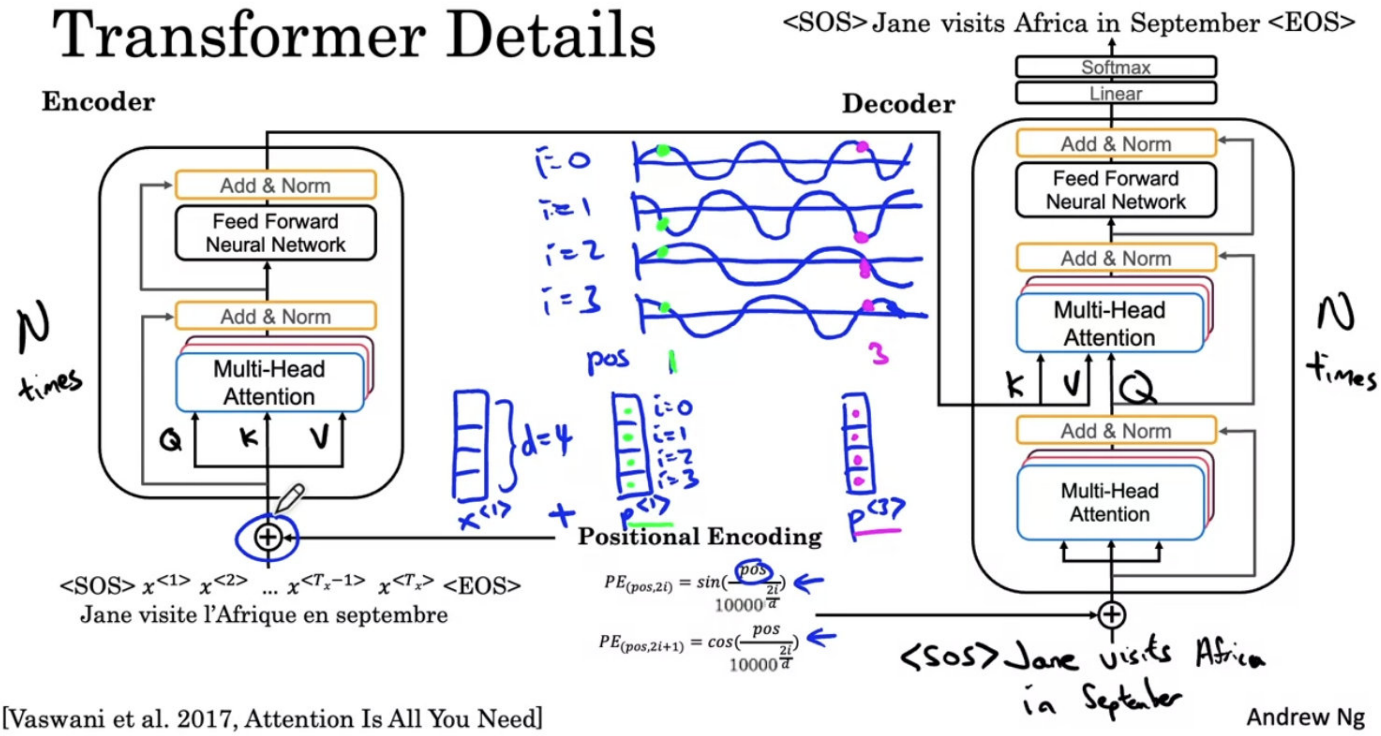
The positional encoding $p^{\langle t \rangle}$ is added directly to the $x^{\langle t \rangle}$, so that each of the word vectors is also influenced by the position of the word. The values of the sine and cosine equations are small enough (between -1 and 1) that when you add the positional encoding to a word embedding, the word embedding is not significantly distorted.



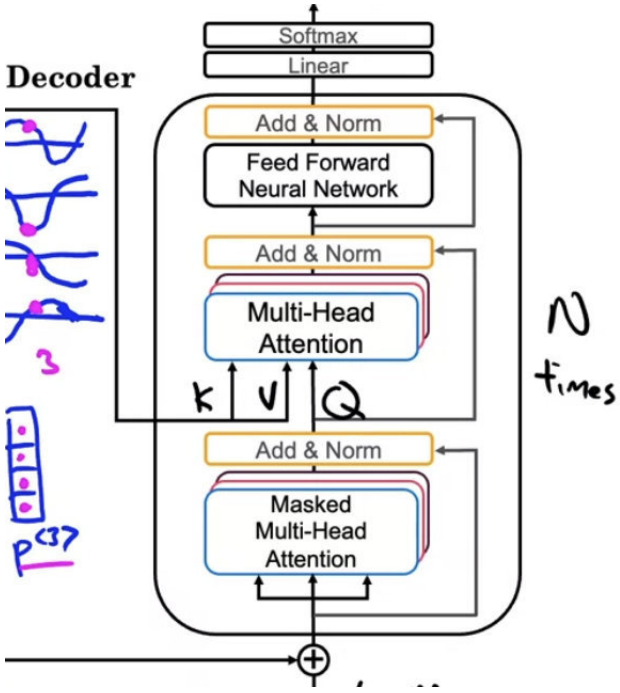
In addition to adding this positional encodings to the embeddings, or the attention representations, we would also pass them through the network with **residual connections**. The add & norm layer is just like the batchnorm layer to speed up learning.



And these bachnorm-like layer and residual connections are repeated throughout the architecture. Finaly, for the output of the decoder block, there are a linear and a softmax layer to predict the next word one at a time.



In literature on transformer, the first multi-head attention block is actually **masked multi-head attention**. This is important only for training process.



When training the network, you have the access to the entire correct outputs. So we do not need to generate the words one at a time during training. Instead, what masking does is it **blocks out** the last part of sentence to mimic what the network will need to do at test time while doing prediction. In other words, all the masked multi-head attention do is to repeatedly pretend that the network had perfectly translated first few words, and hide the remaining words to see if the network can predict the next word accurately when giving perfect inputs.

Finaly, **THIS IS THE TRANSFORMER NETWORK**.