Deep Learning Notes

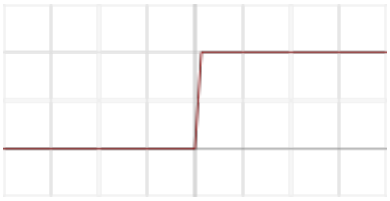
chtunsw@gmail.com

Contents

1 Neural Networks and Deep Learning

1.1 Activation Function

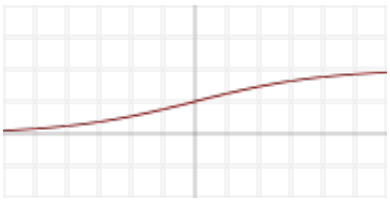
binary step:



$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}, f(x) \in \{0, 1\}$$

$$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

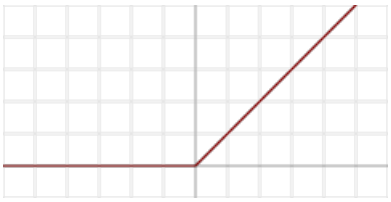
sigmoid function: (mainly used in output layer)



$$f(x) = \sigma(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}, f(x) \in (0, 1)$$

$$f'(x) = f(x)(1 - f(x))$$

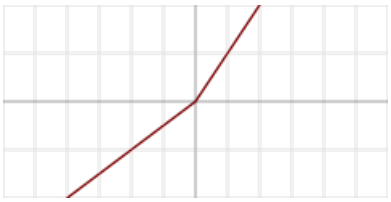
rectified linear unit: (relu, mainly used in hidden layer)



$$f(x) = max\{0, x\} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}, f(x) \in [0, +\infty)$$

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

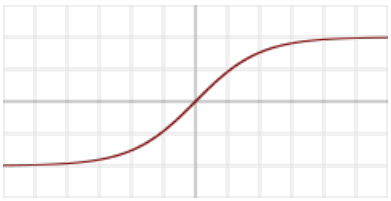
Leaky rectified linear unit: (leaky relu, mainly used in hidden layer)



$$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}, f(x) \in (-\infty, +\infty)$$

$$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

hyperbolic tangent: (tanh, mainly used in hidden layer)



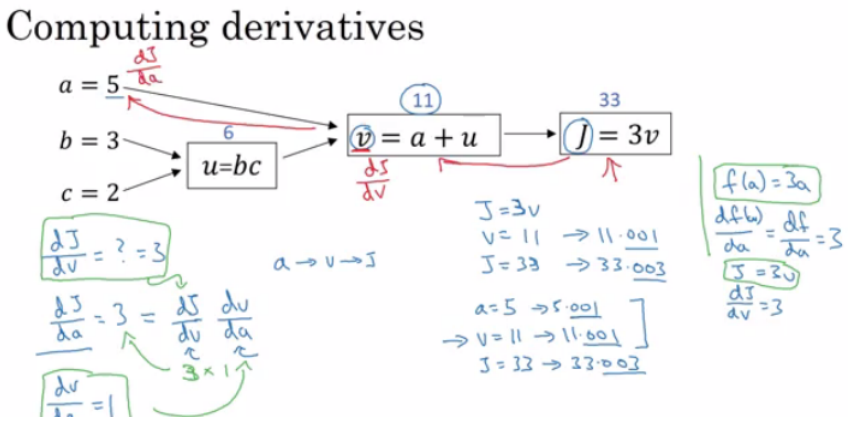
$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, f(x) \in (-1, 1)$$

$$f'(x) = 1 - f(x)^2$$

It turns out that the tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer. Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the vanishing gradient problem.

Linear activation function will output linear activations. No matter how many hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems). You might use linear activation function in the output layer if the output is real numbers (regression problem).

1.2 Computation Graphs of Derivatives:



apply chain rule:

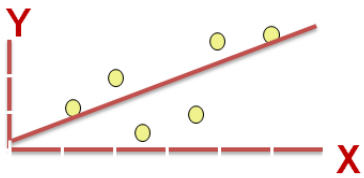
$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial a} = 3 \times 1 = 3$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 3 \times 1 \times 2 = 6$$

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial c} = 3 \times 1 \times 3 = 9$$

1.3 Binary Classification

Use logistic regression to build a binary classifier.



training data:

$$x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$$

$$m \text{ training examples: } (x^{(i)}, y^{(i)}) \; for \; i = 1, \ldots, m$$

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix}, y^{(i)} = y_1^{(i)}$$

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \ldots & x^{(m)} \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(m)} \end{bmatrix}$$

$$X \in \mathbb{R}^{n_x \times m}, Y \in \mathbb{R}^{1 \times m}$$

1.4 Logistic Regression

Logistic regression is a statistical model that uses a logistic function to model a binary dependent variable.

Given x, want $\hat{y} = P(y = 1|x)$, where $0 \leq \hat{y} \leq 1$, $x \in \mathbb{R}^{n_x}$, $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

$$z = w^T x + b$$

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

$$P(y|x) = \begin{cases} \hat{y} & \text{if } y = 1 \\ 1 - \hat{y} & \text{if } y = 0 \end{cases} = \hat{y}^y (1 - \hat{y})^{(1-y)}$$

We want to maximize $P(y|x)$. To make it simpler, because log function is a strictly increasing function, we can maximize $log(P(y|x))$ instead.

$$log(P(y|x)) = log(\hat{y}^y (1 - \hat{y})^{(1-y)}) = ylog(\hat{y}) + (1 - y)log(1 - \hat{y})$$

Or in reverse, we can minimize $-log(P(y|x))$, which is called **loss function**.
loss function: (convex)

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

cost function: (convex)

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

gradient descent: with learning rate α, find best w, b to minimize $J(w,b)$
repeat until convergence {

$$w_j =: w_j - \alpha\frac{\partial}{\partial w_j}J(w,b) = w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x_j^{(i)}$$

$$b =: b - \alpha\frac{\partial}{\partial b}J(w,b) = b - \alpha\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

simultaneously update w_j and b, $j \in [1, n]$

}

vectorized implementation:
repeat until convergence {

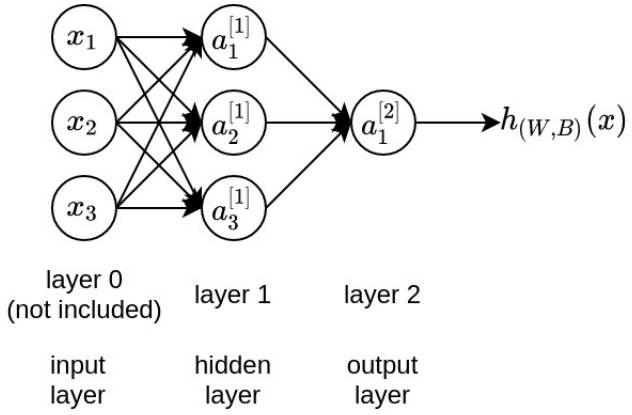
$$w =: w - \alpha\frac{1}{m}X(\sigma(w^T X + b) - Y)^T$$

$$b =: b - \alpha\frac{1}{m}sum\{\sigma(w^T X + b) - Y\}$$

}

1.5 Neural Network

Basic Structure:



$a_i^{[j]}$ = "activation" of unit i in layer j

$W^{[j]}$ = matrix of weights (edges) from layer j - 1 to j

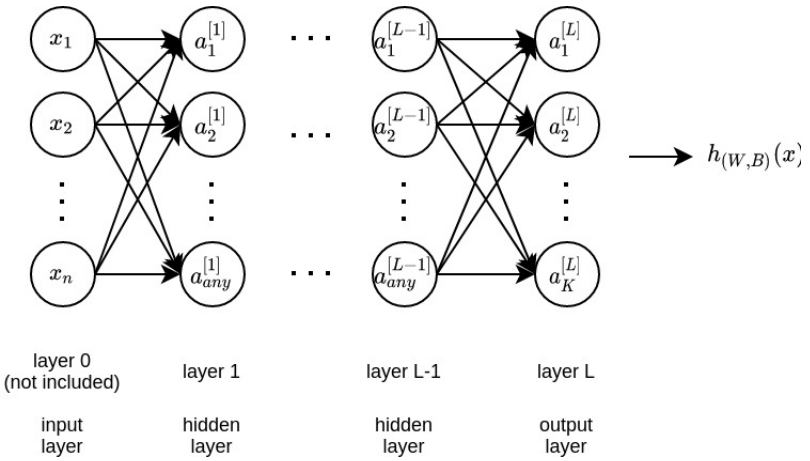
$B^{[j]}$ = vector of biases (nodes) from layer j - 1 to j

$$a_1^{[1]} = \sigma(W_{11}^{[1]}x_1 + W_{12}^{[1]}x_2 + W_{13}^{[1]}x_3 + B_1^{[1]})$$
$$a_2^{[1]} = \sigma(W_{21}^{[1]}x_1 + W_{22}^{[1]}x_2 + W_{23}^{[1]}x_3 + B_2^{[1]})$$
$$a_3^{[1]} = \sigma(W_{31}^{[1]}x_1 + W_{32}^{[1]}x_2 + W_{33}^{[1]}x_3 + B_3^{[1]})$$

$$h_{(W,B)}(x) = a_1^{[2]} = \sigma(W_{11}^{[2]}a_1^{[1]} + W_{12}^{[2]}a_2^{[1]} + W_{13}^{[2]}a_3^{[1]} + B_1^{[2]})$$

If network has s_j units in layer j, s_{j-1} units in layer $j-1$, then $W^{[j]}$ will be of dimension $s_j \times s_{j-1}$, $B^{[j]}$ will be of dimension $s_j \times 1$.

Generalized Model (one vs all):



For a neural network that has:

$$L = \text{total number of layers in the network}$$
$$s_l = \text{number of units in layer } l$$
$$K = \text{number of output units/classes}$$

assume $a^{[0]} = x, a^{[L]} = h_{(W,B)}(x)$, let:

$$z^{[l]} = W^{[l]}a^{[l-1]} + B^{[l]}$$
$$a^{[l]} = \sigma(z^{[l]})$$
$$h_{(W,B)}(x) = a^{[L]} = \sigma(z^{[L]})$$

regularized cost function:

$$J(W,B) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)}log(h_{(W,B)}(x^{(i)})_k) + (1-y_k^{(i)})log(1-h_{(W,B)}(x^{(i)})_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{j,i}^{[l]})^2$$

To reduce over-fitting, we can reduce(penalize) the weight of the features in our function carry by increasing their cost. The λ is the regularization parameter. It determines how much the costs of our theta parameters are inflated.

vectorized implementation: (with different activation function for each layer)

$$Z^{[l]} = W^{[l]}A^{[l-1]} + B^{[l]}$$
$$A^{[l]} = \sigma^{[l]}(Z^{[l]})$$
$$h_{(W,B)}(X) = A^{[L]} = \sigma^{[L]}(Z^{[L]})$$
$$J(W,B) = -\frac{1}{m}np.sum(Y \odot log(h_{(W,B)}(X)) + (1-Y) \odot log(1-h_{(W,B)}(X))) + \frac{\lambda}{2m}np.sum(W \odot W)$$

1.6 Backpropagation Preliminary

matrix calculus: see Wikipedia
chaine rule:
Suppose the variable J depends on the variables $w_1, \ldots, w_p$ via the intermediate variable $z_1, \ldots, z_k$.

$$z_j = z_j(w_1, \ldots, w_p), \forall j \in \{1, \ldots, k\}$$
$$J = J(z_1, \ldots, z_k)$$

Expand J, we can find:

$$\frac{\partial J}{\partial w_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial w_i}, \forall i \in \{1, \ldots, p\}$$

chain rule derivation for matrix:
Suppose J is a real-valued output variable, $z \in \mathbb{R}^m$ is the intermediate variable and $W \in \mathbb{R}^{m \times d}$, $B \in \mathbb{R}^m$, $a \in \mathbb{R}^d$ are the input variables. Suppose they satisfy:

$$z = Wa + B$$
$$J = J(z)$$

Then we can get:

$$\frac{\partial J}{\partial a} = \begin{bmatrix} \frac{\partial J}{\partial a_1} \\ \vdots \\ \frac{\partial J}{\partial a_d} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^m \frac{\partial J}{\partial z_j} \frac{z_j}{a_1} \\ \vdots \\ \sum_{j=1}^m \frac{\partial J}{\partial z_j} \frac{z_j}{a_d} \end{bmatrix} = \begin{bmatrix} \frac{\partial z_1}{\partial a_1} & \cdots & \frac{\partial z_m}{\partial a_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial a_d} & \cdots & \frac{\partial z_m}{\partial a_d} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} = \frac{\partial z}{\partial a} \frac{\partial J}{\partial z} = W^T \frac{\partial J}{\partial z}$$

$$\frac{\partial J}{\partial W_{ij}} = \sum_{k=1}^{m} \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial W_{ij}} = \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} = \frac{\partial J}{\partial z_i} a_j$$

$$\frac{\partial J}{\partial B_i} = \sum_{k=1}^{m} \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial B_i} = \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial B_i} = \frac{\partial J}{\partial z_i}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} \begin{bmatrix} a_1 & \ldots & a_d \end{bmatrix} = \frac{\partial J}{\partial z} a^T$$

$$\frac{\partial J}{\partial B} = \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} = \frac{\partial J}{\partial z}$$

element-wise chain rule:
Assume $z, a \in \mathbb{R}^d$:

$$a = \sigma(z), \text{ where } \sigma \text{ is an element-wise activation}$$
$$J = J(a)$$

Then we have:

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial a} \odot \sigma'(z)$$

Where σ' is the element-wise derivative of the activation function σ.

1.7 Backpropagation

To train the model, we need to update W and B for each epoch: (gradient decent)

$$W := W - \alpha \frac{\partial}{\partial W} J(W, B)$$
$$B := B - \alpha \frac{\partial}{\partial B} J(W, B)$$

We can see $\frac{\partial}{\partial W} J(W, B)$ and $\frac{\partial}{\partial B} J(W, B)$ are hard to get directly. There is an easier way to calculate it. For each training example $(x^{(q)}, y^{(q)}), q \in \{1, \ldots, m\}$, define **loss function**:

$$J = -\sum_{k=1}^{K} [y_k^{(q)} log(h_{(W,B)}(x^{(q)})_k) + (1 - y_k^{(q)}) log(1 - h_{(W,B)}(x^{(q)})_k)]$$

Apply chain rule we have:

$$\frac{\partial J}{\partial W^{[l]}} = \frac{\partial J}{\partial z^{[l]}} (a^{[l-1]})^T$$
$$\frac{\partial J}{\partial B^{[l]}} = \frac{\partial J}{\partial z^{[l]}}$$
$$\frac{\partial J}{\partial a^{[l]}} = (W^{[l+1]})^T \frac{\partial J}{\partial z^{[l+1]}}$$
$$\begin{aligned} \frac{\partial J}{\partial z^{[l]}} &= \frac{\partial J}{\partial a^{[l]}} \odot \sigma'(z^{[l]}) \\ &= (W^{[l+1]})^T \frac{\partial J}{\partial z^{[l+1]}} \odot \sigma'(z^{[l]}) \\ &= (W^{[l+1]})^T \frac{\partial J}{\partial z^{[l+1]}} \odot (a^{[l]} \odot (1 - a^{[l]})) \end{aligned}$$

For $p \in \{1, ..., K\}$:

$$\begin{aligned} \frac{\partial J}{\partial z_p^{[L]}} &= \frac{\partial}{\partial z_p^{[L]}} \sum_{k=1}^{K} -[y_k^{(q)} log(h_{(W,B)}(x^{(q)})_k) + (1 - y_k^{(q)}) log(1 - h_{(W,B)}(x^{(q)})_k)] \\ &= \frac{\partial}{\partial z_p^{[L]}} \{-[y_p^{(q)} log(\frac{1}{1 + e^{-z_p^{[L]}}}) + (1 - y_p^{(q)}) log(1 - \frac{1}{1 + e^{-z_p^{[L]}}})]\} \\ &= -[y_p^{(q)} (1 + e^{-z_p^{[L]}}) \frac{0 - (-e^{-z_p^{[L]}})}{(1 + e^{-z_p^{[L]}})^2} + (1 - y_p^{(q)}) \frac{1 + e^{-z_p^{[L]}}}{e^{-z_p^{[L]}}} \frac{(-e^{-z_p^{[L]}})(1 + e^{-z_p^{[L]}}) - e^{-z_p^{[L]}}(-e^{-z_p^{[L]}})}{(1 + e^{-z_p^{[L]}})^2}] \\ &= -[y_p^{(q)} \frac{e^{-z_p^{[L]}}}{1 + e^{-z_p^{[L]}}} + (1 - y_p^{(q)}) \frac{-1}{1 + e^{-z_p^{[L]}}}] \\ &= -\frac{y_p^{(q)} e^{-z_p^{[L]}} + y_p^{(q)} - 1}{1 + e^{-z_p^{[L]}}} \\ &= \frac{1}{1 + e^{-z_p^{[L]}}} - y_p^{(q)} \\ &= a_p^{[L]} - y_p^{(q)} \end{aligned}$$

Then we get:

$$\frac{\partial J}{\partial z^{[L]}} = \begin{bmatrix} \frac{\partial J}{\partial z_1^{[L]}} \\ \vdots \\ \frac{\partial J}{\partial z_K^{[L]}} \end{bmatrix} = a^{[L]} - y^{(q)}$$

For convenience, define **error term**:

$$\delta^{[l]} = \frac{\partial J}{\partial z^{[l]}}$$

Then we get:

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]}(a^{[l-1]})^T$$

$$\frac{\partial J}{\partial B^{[l]}} = \delta^{[l]}$$

$$\delta^{[l]} = (W^{[l+1]})^T\delta^{[l+1]} \odot (a^{[l]} \odot (1 - a^{[l]}))$$

$$\delta^{[L]} = a^{[L]} - y^{(q)}$$

backpropagation algorithm: (compute $\frac{\partial}{\partial W}J(W,B)$, $\frac{\partial}{\partial B}J(W,B)$)
training set: $(x^{(q)}, y^{(q)}), q \in \{1, \ldots, m\}$
set $\Delta(W)_{ij}^{[l]} = 0, \Delta(B)_i^{[l]} = 0, l \in \{1, \ldots, L\}$
for $q \in \{1, \ldots, m\}$:
    forward propagation: compute $a^{[l]}$ for $l \in \{1, \ldots, L\}$
    compute $\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} = a^{[L]} - y^{(q)}$
    for $l \in \{L-1, \ldots, 1\}$:
        compute $\delta^{[l]} = (W^{[l+1]})^T\delta^{[l+1]} \odot \sigma'(z^{[l]}) = (W^{[l+1]})^T\delta^{[l+1]} \odot (a^{[l]} \odot (1 - a^{[l]}))$
    for $l \in \{1, \ldots, L\}$:
        compute $\Delta(W)^{[l]} := \Delta(W)^{[l]} + \frac{\partial J}{\partial W^{[l]}} = \Delta(W)^{[l]} + \delta^{[l]}(a^{[l-1]})^T$
        compute $\Delta(B)^{[l]} := \Delta(B)^{[l]} + \frac{\partial J}{\partial B^{[l]}} = \Delta(B)^{[l]} + \delta^{[l]}$
compute $\frac{\partial}{\partial W_{ij}^{[l]}}J(W,B) = D(W)_{ij}^{[l]} = \frac{1}{m}\Delta(W)_{ij}^{[l]} + \frac{\lambda}{m}W_{ij}^{[l]}$
compute $\frac{\partial}{\partial B_i^{[l]}}J(W,B) = D(B)_i^{[l]} = \frac{1}{m}\Delta(B)_i^{[l]}$

vectorized backpropagation algorithm: (compute $\frac{\partial}{\partial W}J(W,B)$, $\frac{\partial}{\partial B}J(W,B)$)
with different activation function for each layer:

$$\partial Z^{[L]} = \frac{\partial}{\partial Z^{[L]}}J(W,B) = A^{[L]} - Y$$

$$\partial Z^{[l]} = \frac{\partial}{\partial Z^{[l]}}J(W,B) = (W^{[l+1]})^T\partial Z^{[l+1]} \odot \sigma'^{[l]}(Z^{[l]})$$

$$\partial W^{[l]} = \frac{\partial}{\partial W^{[l]}}J(W,B) = \frac{1}{m}\partial Z^{[l]}(A^{[l-1]})^T + \frac{\lambda}{m}W^{[l]}$$

$$\partial B^{[l]} = \frac{\partial}{\partial B^{[l]}}J(W,B) = \frac{1}{m}np.sum(\partial Z^{[l]}, axis = 1, keepdims = True)$$

1.8 Random Initialization (symmetry breaking)

Initializing bias matrices B with zero is OK. Initializing all the weight matrices W with zero does not work with neural networks, because all hidden units will be completely identical (symmetric) - compute exactly the same function. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our W matrices using the following method:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * 0.01$$

1.9 Parameters vs Hyperparameters

Main parameters of the Neural Network are W and B.
Hyper parameters (parameters that used to find best main parameters) are like:

- Learning rate.
- Number of iteration.
- Number of hidden layers.
- Number of hidden units.
- Choice of activation functions.
- Momentum term.
- Mini-batch size.
- Regularization parameters.

2 Hyperparameter Tuning, Regularization and Optimization

2.1 Train/Dev/Test Sets

find better hyperparameters:
repeat: Idea $\longrightarrow$ Code $\longrightarrow$ Experiment

split dataset:

- Training set: train models (parameters tuning)
- Validation set / Development set: choose the best model (hyperparameters tuning)
- Testing set: estimate model performance in production

ratio of splitting:

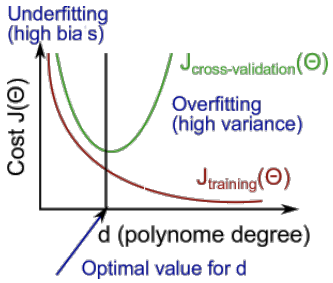
- If size of the dataset is 100 to 1000000: 60%/20%/20%
- If size of the dataset is 1000000 to INF: 98%/1%/1% or 99.5%/0.25%/0.25%

Make sure the dev and test set are coming from the same distribution. For example if cat training pictures is from the web and the dev/test pictures are from users cell phone they will mismatch. It is better to make sure that dev and test set are from the same distribution.

The dev set rule is to try them on some of the good models you've created. Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

2.2 Bias/Variance

The training error will tend to decrease as we increase the degree d of the polynomial. At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.



High bias(underfitting): both $J_{train}(W,B)$ and $J_{cv}(W,B)$ will be high. Also, $J_{cv}(W,B) \approx J_{train}(W,B)$.
High variance(overfitting): $J_{train}(W,B)$ will be low and $J_{cv}(W,B)$ will be high. Also, $J_{cv}(W,B) \gg J_{train}(W,B)$.

underfitting:
High bias, which means hypothesis fits training data poorly, is usually caused by a function that is too simple or using too few features.

overfitting:
High variance, which means hypothesis fits training data well, but does not generalize well to predict new data. It is usually caused by a complicated function with too many features.

2.3 Basic Recipe for Machine Learning

If your algorithm has a high bias:

- Try to make your NN bigger (size of hidden units, number of layers)
- Try a different model (architecture) that is suitable for your data.
- Try to run it longer.
- Different (advanced) optimization algorithms.

If your algorithm has a high variance:

- More data.
- Try regularization.
- Try a different model (architecture) that is suitable for your data.

You should try the previous two points until you have a low bias and low variance. In the older days before deep learning, there was a "Bias/variance tradeoff". But because now you have more options/tools for solving the bias and variance problem its really helpful to use deep learning. With enough training data, training a bigger neural network never hurts.

Regularization: (L2 Regularization)
Add regularization term to the cost function to reduce variance (overfitting):

$$\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{j,i}^{[l]})^2 = \frac{\lambda}{2m}np.sum(W \odot W)$$

weight backpropagation with regularization:

$$W^{[l]} := W^{[l]} - \alpha\partial W^{[l]} = (1 - \frac{\alpha\lambda}{m})W^{[l]} - \frac{\alpha}{m}\partial Z^{[l]}(A^{[l-1]})^T$$

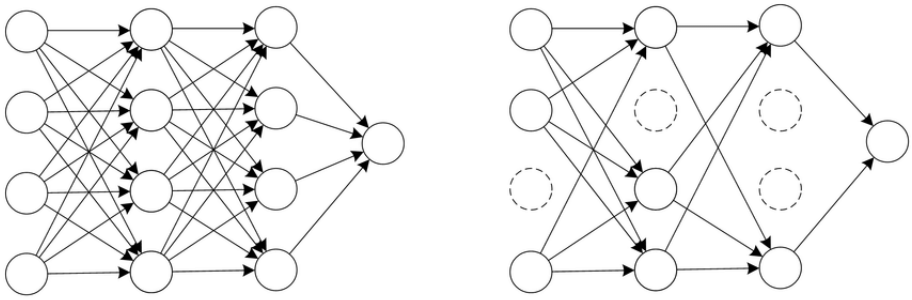
The new term $(1 - \frac{\alpha\lambda}{m})W^{[l]}$ causes the weight to decay in proportion to its size. In practice this penalizes large weights and effectively limits the freedom in your model.

- If λ is too large - $W_{i,j}^{[l]}$ will be close to zeros which will make the NN simpler.
- If λ is good enough it will just reduce some weights and prevent the overfitting.
- If λ is too large, $Z_i^{[l]}$ will be small (close to zero) - assume tanh activation function is used, then it will behave like a linear function, so we will go from non linear activation to roughly linear which would make the NN a roughly linear classifier.
- If λ good enough it will just make some of tanh activations roughly linear which will prevent overfitting.

Implementation tip: if you implement gradient descent, plot the cost function as a function of the number of iterations of gradient descent and you want to see that the cost function decreases monotonically after every elevation of gradient descent with regularization.

Dropout:
The dropout regularization eliminates some neurons/weights on each iteration based on a probability. It is used to reduce overfitting. A most common technique to implement dropout is called "Inverted dropout".

$$keep\_prob = 0.8 \text{ (for example)}$$
$$D^{[l]} = np.random.rand(A^{[l]}.shape[0], A^{[l]}.shape[1]) < keep\_prob$$
$$A^{[l]} = A^{[l]} \odot D^{[l]}$$
$$A^{[l]} = keep\_prob^{-1} A^{[l]}$$



(a) Standard Neural Network

(b) Network after Dropout

Drop 20% of units in $A^{[l]}$ by replacing them with 0, then scale $A^{[l]}$ by $keep\_prob^{-1}$.

We need to scale $A^{[l]}$ because $Z^{[l+1]} = W^{[l+1]}A^{[l]} + B^{[l+1]}$. And we want to increase $A^{[l]}$ to not reduce the expected value of output $Z^{[l+1]}$ after dropout.

$D^{[l]}$ is used for forward and back propagation and is the same for them, but it is different for each iteration of training example. At test time we don't use dropout. Otherwise it would add noise to predictions.

Understanding Dropout:

- The intuition is dropout randomly knocks out units in your network. So it's as if on every iteration you're working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.
- Another intuition: can't rely on any single feature, so have to spread out weights.
- Dropout can have different keep_prob per layer.
- The input layer dropout has to be near 1 (or 1 - no dropout) because you don't want to eliminate a lot of features.

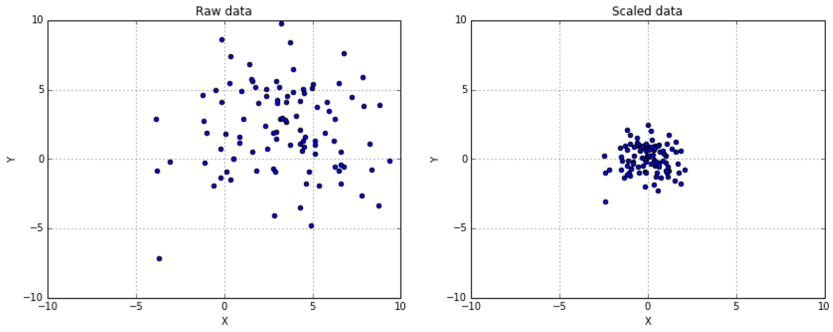
- If you're more worried about some layers overfitting than others, you can set a lower keep_prob for some layers than others. The downside is, this gives you even more hyperparameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and then just have one hyperparameter, which is a keep_prob for the layers for which you do apply dropouts.
- A lot of researchers are using dropout with Computer Vision (CV) because they have a very big input size and almost never have enough data, so overfitting is the usual problem.
- A downside of dropout is that the cost function is not well defined and it will be hard to debug (plot cost by iteration). To solve that you'll need to turn off dropout, set all the keep_prob to 1, and then run the code and check that it monotonically decreases cost and then turn on the dropouts again.

Other regularization methods:

- Data augmentation. For example, you can flip all your pictures horizontally this will give you m more data instances. You could also apply a random position and rotation to an image to get more data. New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
- Early stopping. We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost). We will take these parameters as the best parameters.

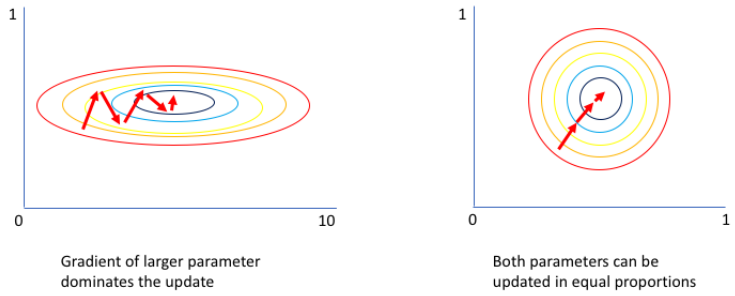
Normalizing Inputs: (Feature Scaling)
All features should be normalized so that they are scaled into a certain range and each feature contributes approximately proportionately to the result. It helps to center the dataset. Also it helps gradient descent converge much faster.

This should be applied to training, dev, and testing sets (but using mean and variance of the training set).



If we normalize inputs, the shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.

Why normalize?



Gradient of larger parameter dominates the update

Both parameters can be updated in equal proportions

mean normalization:

$$x_i = \frac{x_i - mean(x_i)}{max(x_i) - min(x_i)}$$

standardization:
Feature standardization makes the values of each feature in the data have zero-mean (when subtracting the mean in the numerator) and unit-variance.

μ : mean, σ^2 : variance, σ : standard deviation

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_i^{(j)} - \mu_i)^2$$

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$

Gradient Vanishing/Exploding:
In a deep neural network, when the weights or derivatives for each layer get a little bit smaller/larger, the final gradient could be exponentially smaller/larger with respect to L. Too small/large gradient could cause weights update overshot or in tiny steps, and make training process unable to continue.

Weight Initialization:
A partial solution to the Vanishing / Exploding gradients is better or more careful choice of the random initialization of weights.

Some researcher found that: (He Initialization / Xavier Initialization)

If *tanh* is used as activation function, the variance of $W^{[l]}$ should be $\frac{1}{s_{l-1}}$, then $W^{[l]}$ should be initialized as:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * np.sqrt(\frac{1}{s_{l-1}})$$

If *relu* is used as activation function, the variance of $W^{[l]}$ should be $\frac{2}{s_{l-1}}$, then $W^{[l]}$ should be initialized as:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * np.sqrt(\frac{2}{s_{l-1}})$$

Some old paper use this as well:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * np.sqrt(\frac{2}{s_{l-1} + s_l})$$

Gradient checking:
Compose W, B into one big vector Θ. We can approximate the derivative of $J(\Theta)$ with respect to Θ_i as:

$$\frac{\partial}{\partial \Theta_i} J(\Theta) \approx \frac{J(\dots, \Theta_i + \epsilon, \dots) - J(\dots, \Theta_i - \epsilon, \dots)}{2\epsilon}$$

A small value for ϵ such as $\epsilon = 10^{-7}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems. Then we can check if $gradApprox \approx deltaVector$.

- if it is $< 10^{-7}$ - great, very likely the backpropagation implementation is correct
- if around 10^{-5} - can be OK, but need to inspect if there are no particularly big values in $d\_theta\_approx - d\_theta$ vector
- if it is $> 10^{-3}$ - bad, probably there is a bug in backpropagation implementation

gradient checking implementation notes:

- Don't use the gradient checking algorithm at training time because it's very slow. Use gradient checking only for debugging.
- If algorithm fails grad check, look at components to try to identify the bug.
- Don't forget to add regularization term to $J(W, B)$ if you are using L1 or L2 regularization.
- Gradient checking doesn't work with dropout because cost function is not consistent. You can first turn off dropout $keep\_prob = 1.0$, run gradient checking and then turn on dropout again.
- Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

Mini-batch gradient descent:
When training set is getting too big, it is impossible to use entire training set X, Y to do a vectorized gradient descent. Instead we can separate the training set into mini batches, and do a vectorized gradient descent for each of them at a time.

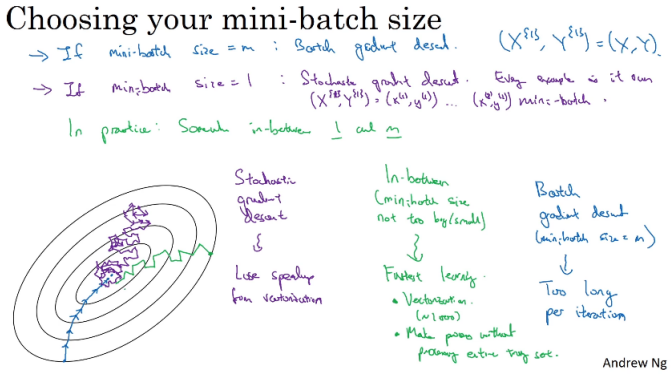
After we iterate through the entire training set (all batches) for one time, we call it an *epoch*. The model should be trained for enough *epochs* until the cost function converges.

Suppose we have m training examples, and we use $batch\_size = 1000$, then we have $num\_batches = \frac{m}{batch\_size}$. The mini batches are $X^{\{t\}}$, $Y^{\{t\}}$ for $t \in [1, num\_batches]$. And we want to train the model with entire training set for $num\_epochs = 10000$ times.

mini-batch gradient descent algorithm:
for $epoch \in \{1, \dots, num\_epochs\}$:
    for $t \in \{1, \dots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        $W =: W - \alpha \partial W$
        $B =: B - \alpha \partial B$

Understanding Mini-Batch Gradient Descent:
In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm (not enough training data in one mini batch, not as effective as batch algorithm). It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function decreases on each iteration).

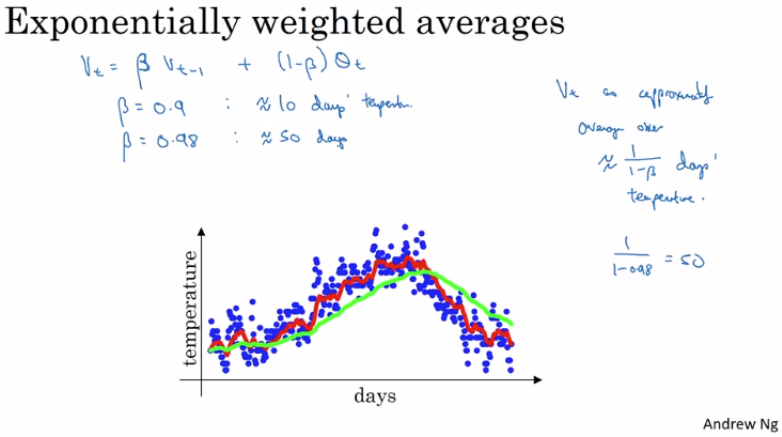
- $batch\_size = m$, Batch gradient descent, too long per iteration
- $batch\_size = 1$, Stochastic gradient descent (SGD), too noisy regarding cost minimization (can be reduced by using smaller learning rate), won't ever converge (reach the minimum cost), lose speedup from vectorization
- $batch\_size \in [1, m]$, Mini-batch gradient descent, have the vectorization advantage, make progress without waiting to process the entire training set, doesn't always exactly converge (oscillates in a very small region, but you can reduce learning rate)



Guidelines for choosing mini-batch size:

- If small training set - use batch gradient descent.
- It has to be a power of 2 (because of the way computer memory is Laid out and accessed, sometimes your code runs faster): $2^6, 2^7, 2^8, 2^9, 2^{10} \ldots$
- Make sure that mini-batch fits in CPU/GPU memory.
- Mini-batch size is a hyperparameter.

2.4 Exponentially Weighted Averages



If we have a daily temperature list like this:

$$\theta_1 = 40, \theta_2 = 49, \theta_3 = 45, \ldots, \theta_{50} = 56$$

We call v_t is the exponentially weighted average of daily temperature over the last $\frac{1}{1-\beta}$ days: ($\beta \in [0, 1)$)

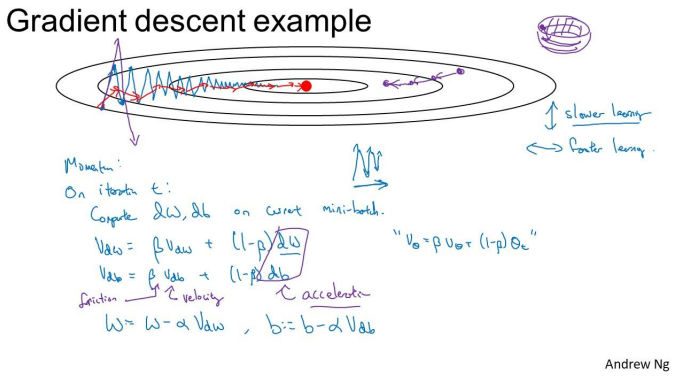
$$v_0 = 0$$
$$v_1 = \beta v_0 + (1-\beta)\theta_1$$
$$v_2 = \beta v_1 + (1-\beta)\theta_2$$
$$\ldots$$
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

Bias Correction of Exponentially Weighted Averages:
The bias correction helps make the exponentially weighted averages more accurate. Because $v_0 = 0$, the initial weighted averages are too small and the accuracy suffers at the start. To fix the bias for the initial estimates we have to use this equation: (As t becomes larger the $1 - \beta^t$ becomes close to 1)

$$v_t = \frac{\beta v_{t-1} + (1-\beta)\theta_t}{1-\beta^t}$$

2.5 Gradient Descent With Momentum

The momentum algorithm almost always works faster than standard gradient descent. It smooths out the steps of gradient descent by using the exponentially weighted average of the gradients. In practice people don't bother implementing bias correction.

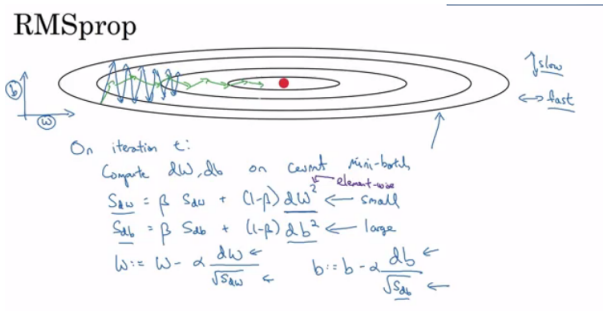


gradient descent with momentum:
$v_{\partial W} = 0, v_{\partial B} = 0$
for $epoch \in \{1, \ldots, num\_epochs\}$:
    for $t \in \{1, \ldots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        $v_{\partial W} =: \beta v_{\partial W} + (1-\beta)\partial W$
        $v_{\partial B} =: \beta v_{\partial B} + (1-\beta)\partial B$
        $W =: W - \alpha v_{\partial W}$
        $B =: B - \alpha v_{\partial B}$

2.6 RMSProp(Root mean square prop)

RMSprop can speed up the gradient descent by dividing the gradient ∂W, ∂B by a manually constructed term: If gradient is small, divided by a small term will make it larger, and make gradient descent faster in this direction; If gradient is large, divided by a large term will make it smaller, and make gradient descent slower in this direction.

It will make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example:



RMSProp:
$s_{\partial W} = 0, s_{\partial B} = 0$
make sure denominator is not too small: $\epsilon = 10^{-8}$
for $epoch \in \{1, \ldots, num\_epochs\}$:
  for $t \in \{1, \ldots, num\_batches\}$:
    vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
    $s_{\partial W} =: \beta s_{\partial W} + (1-\beta)\partial W \odot \partial W$
    $s_{\partial B} =: \beta s_{\partial B} + (1-\beta)\partial B \odot \partial B$
    $W =: W - \alpha \frac{\partial W}{\sqrt{s_{\partial W}+\epsilon}}$
    $B =: B - \alpha \frac{\partial B}{\sqrt{s_{\partial B}+\epsilon}}$

2.7 Adam Optimization Algorithm

Adam (Adaptive Moment Estimation) optimization algorithm combined momentum and RMSProp methods. Hyperparameters for Adam:

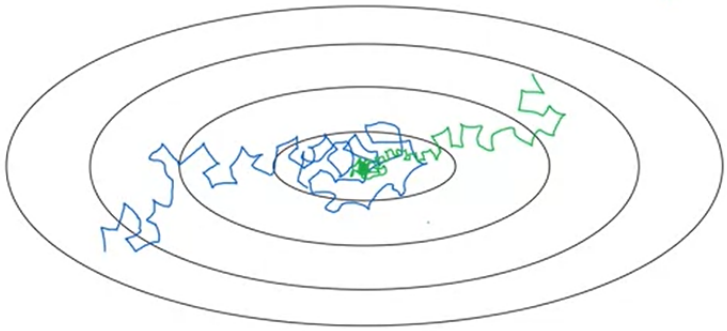
- α: learning rate
- β_1: parameter of the momentum - 0.9 is recommended by default.
- β_2: parameter of the RMSprop - 0.999 is recommended by default.
- ϵ: 10^{-8} is recommended by default.

adam optimization algorithm:
$v_{\partial W} = 0, v_{\partial B} = 0, s_{\partial W} = 0, s_{\partial B} = 0$
make sure denominator is not too small: $\epsilon = 10^{-8}$
for $epoch \in \{1, \ldots, num\_epochs\}$:
  for $t \in \{1, \ldots, num\_batches\}$:
    vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
    $v_{\partial W} =: \beta_1 v_{\partial W} + (1-\beta_1)\partial W$
    $v_{\partial B} =: \beta_1 v_{\partial B} + (1-\beta_1)\partial B$
    $s_{\partial W} =: \beta_2 s_{\partial W} + (1-\beta_2)\partial W \odot \partial W$
    $s_{\partial B} =: \beta_2 s_{\partial B} + (1-\beta_2)\partial B \odot \partial B$
    $v_{\partial W}^{corrected} =: \frac{v_{\partial W}}{1-\beta_1^t}$

$v_{\partial B}^{corrected} =: \frac{v_{\partial B}}{1-\beta_1^t}$
$s_{\partial W}^{corrected} =: \frac{s_{\partial W}}{1-\beta_2^t}$
$s_{\partial B}^{corrected} =: \frac{s_{\partial B}}{1-\beta_2^t}$
$W =: W - \alpha \frac{v_{\partial W}^{corrected}}{\sqrt{s_{\partial W}^{corrected}}+\epsilon}$
$B =: B - \alpha \frac{v_{\partial B}^{corrected}}{\sqrt{s_{\partial B}^{corrected}}+\epsilon}$

2.8 Learning Rate Decay

Slowly reduce learning rate for each epoch helps cost function to converge near the optimum point.



learning rate decay:
initial learning rate: α_0, decay rate: d
for each epoch p: $\alpha_p = \frac{1}{1+d\times p}\alpha_0$

other learning rate decay:

- exponentially decay: $\alpha_p = o.95^p\alpha_0$
- $\alpha_p = \frac{k}{\sqrt{p}}\alpha_0$ or $\alpha_t = \frac{k}{\sqrt{t}}\alpha_0$
- manually learning rate decay.

2.9 Tuning Process

Hyperparameters priority: (as for Andrew Ng)

- α
- β
- s_l
- $batch\_size$
- L
- d
- $\beta_1, \beta_2, \epsilon$

Tune Hyperparameters:

- give each Hyperparameter a range

- randomly pick a value for each Hyperparameter in its range, then train with them
- randomly picking helps to explore Hyperparameters in a larger range (compare with increasing by steps), which can give hints to narrow the range and speed up the tuning process
- use Coarse to fine sampling scheme: when you find some hyperparameter values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space

Using scale to pick hyperparameters
Let's say you have a specific wide range for a hyperparameter from "a" to "b". It's better to search for the right ones using the logarithmic scale rather then in linear scale:

$$a = -4, b = 0$$
$$r \in [a, b]$$
$$r = -4 \times np.random.rand()$$
$$\alpha = 10^r$$

Panda vs. Caviar
Intuitions about hyperparameter settings from one application may or may not transfer to a different one.

- If you don't have much computational resources you can use the "babysitting model" (Panda Approach): Day 0 you might initialize your parameter as random and then start training. Then you watch your learning curve gradually decrease over the day. And each day you nudge your parameters a little during training.
- If you have enough computational resources, you can run several models in parallel and at the end of the day(s) you check the results (Caviar Approach).

Batch Normalization:
Normalized inputs could be shifted in the hidden layers. Batch normalization reduces the problem of input values changing (shifting). And it speeds up learning as well. In practice, normalizing $Z^{[l]}$ is done much more often and that is what Andrew Ng presents.

For a layer l, given a mini batch of samples $Z^{[l](1)}, \ldots, Z^{[l](m)}$:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} Z^{[l](i)}$$
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (Z^{[l](i)} - \mu) \odot (Z^{[l](i)} - \mu)$$
$$Z^{[l](i)}_{norm} = (Z^{[l](i)} - \mu) \oslash \sqrt{\sigma^2 + \epsilon}$$
$$\overset{N}{Z}^{[l](i)} = \gamma^{[l]} \odot Z^{[l](i)}_{norm} + \beta^{[l]}$$

ϵ is added for numerical stability (in case σ is too small). γ, β are learnable parameters of the model, and they make inputs belong to other distribution (with other mean and variance). It makes the NN learn the distribution of the outputs.

Batch normalization does some regularization:

- Each mini batch is scaled by the mean/variance computed of that mini-batch.
- This adds some noise to the values $Z^{[l]}$ within that mini batch. So similar to dropout it adds some noise to each hidden layer's activations.

Gradient Descent with Batch Normalization:
(also can work with Momentum, RMSprop, Adam...)
for $epoch \in \{1, \ldots, num\_epochs\}$:
for $t \in \{1, \ldots, num\_batches\}$:
vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
use $\overset{N}{Z}^{\{t\}}$ to replace $Z^{\{t\}}$
B will be replaced by β
$W =: W - \alpha \partial W$
$\gamma =: \gamma - \alpha \partial \gamma$
$\beta =: \beta - \alpha \partial \beta$

Batch normalization at test time:
When we train a NN with Batch normalization, we compute the mean and the variance of the mini-batch. In testing we might need to process examples one at a time. The mean and the variance of one example won't make sense. We have to compute an estimated value of mean and variance to use it in testing time. We can use the exponentially weighted average of μ, σ^2 across the mini-batches. In practice most often you will use a deep learning framework and it will contain some default implementation of doing such a thing.

Given t mini batches of training examples:
μ_{test} is exponentially weighted average of $\mu^{\{1\}}, \mu^{\{2\}}, \ldots, \mu^{\{t\}}$
σ^2_{test} is exponentially weighted average of $\sigma^{2\{1\}}, \sigma^{2\{2\}}, \ldots, \sigma^{2\{t\}}$

$$Z_{norm} = (Z - \mu_{test}) \oslash \sqrt{\sigma^2_{test} + \epsilon}$$

$$\overset{N}{Z} = \gamma \odot Z_{norm} + \beta$$

Softmax Regression:
Softmax regression is used for multiclass classification/regression. It is normally used in the last layer. Each of values in the output layer represents a probability of the example to belong to each of the classes, and they sum up to 1.

$$t = e^{Z^{[L]}}$$

$$A^{[L]} = \frac{t}{sum\{t\}}$$

Train with Softmax:
With machine learning framework like Tensorflow, Forward Propagation needs to be built, Backpropagation part can be handled automatically .

$$L(\hat{y}, y) = -\sum_{k=1}^{K} y_k log(\hat{y}_k)$$

$$J(W, B) = -\frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

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

$$\partial Z^{[l]} = \ldots$$

Local Optima:

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.

- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.
- Plateaus can make learning slow. Plateau is a region where the derivative is close to zero for a long time. This is where algorithms like momentum, RMSprop or Adam can help.

3 Structuring Machine Learning Projects

3.1 Orthogonalization

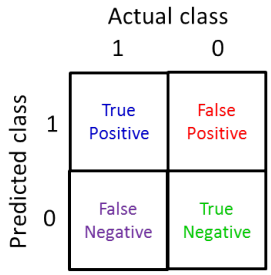
Some deep learning developers know exactly what hyperparameter to tune in order to try to achieve one effect. This is a process we call orthogonalization. In orthogonalization, you have some controls, but each control does a specific task and doesn't affect other controls.

For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that four things hold true - chain of assumptions in machine learning:

- You'll have to fit training set well on cost function (near human level performance if possible). If it's not achieved you could try bigger network, another optimization algorithm (like Adam)...
- Fit dev set well on cost function. If its not achieved you could try regularization, bigger training set...
- Fit test set well on cost function. If its not achieved you could try bigger dev set...
- Performs well in real world. If its not achieved you could try change dev set, change cost function...

3.2 Single Number Evaluation Metric

Confusion Matrix:
Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature. The following example is a classic 2-class confusion matrix. But it can have multiple classes.



Precision & Recall: (deal with skewed classes)
High $F_{score} \in [0, 1]$ (high precision and high recall) represents a good prediction model. (use average value of precision and recall for all classes if there are multiple classes)

$$\text{Precision} = \frac{\text{True positives}}{\text{predicted as positive}} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

$$\text{Recall} = \frac{\text{True positives}}{\text{actual positives}} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

$$F_{score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Satisficing and Optimizing Metrics:
Assume we have N metrics to evaluate the model, normally we will have 1 optimizing metric, and $N - 1$ satisficing metrics. For example: Maximize optimizing metric F_{score}, subject to satisficing metrics $RunningTime < 100ms$, $ModelSize < 1gb$.

3.3 Train/Dev/Test Set Distributions

Dev and test sets have to come from the same distribution. Choose dev set and test set to reflect data you expect to get in the future and consider important to do well on. Setting up the dev set, as well as the validation metric is really defining what target you want to aim at.

Size of the dev and test sets:
An old way of splitting the data was 70% training, 30% test or 60% training, 20% dev, 20% test. The old way was valid for a number of examples < 100000 In the modern deep learning if you have a million or more examples a reasonable split would be 98% training, 1% dev, 1% test.

When to change dev/test sets and metrics:
Orthogonalization for deep learning: break a machine learning problem into distinct steps.

- Figure out how to define a metric that captures what you want to do - place the target.
- Worry about how to actually do well on this metric - how to aim/shoot accurately at the target.

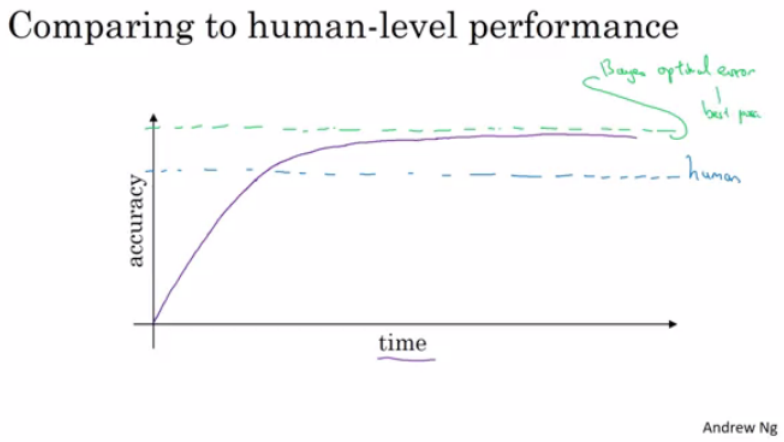
Conclusion: if doing well on your metric + dev/test set doesn't correspond to doing well in your application, change your metric and/or dev/test set.

3.4 Human-level Performance

We compare to human-level performance (as baseline level performance) because of two main reasons:

- Because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance.
- It turns out that the workflow of designing and building a machine learning system is much more efficient when you're trying to do something that humans can also do.

After an algorithm reaches the human level performance the progress and accuracy slow down.



You won't surpass an error that's called "Bayes optimal error" (the minimum possible error that can be made). There isn't much error range between human-level error and Bayes optimal error.

Humans are quite good at a lot of tasks. So as long as Machine learning is worse than humans, you can:

- Get labeled data from humans.
- Gain insight from manual error analysis: why did a person get it right?

- Better analysis of bias/variance.

Avoidable bias:
Suppose that the cat classification algorithm gives these results:

Humans	1%	7.5%
Training error	8%	8%
Dev Error	10%	10%

The human-level error as a proxy (estimate) for Bayes optimal error. Bayes optimal error is always less (better), but human-level in most cases is not far from it. You can't do better than Bayes error unless you are overfitting.

$$\text{Avoidable Bias} = \text{Training error} - \text{Human (Bayes) error}$$

$$\text{Measure of Variance} = \text{Dev error} - \text{Training error}$$

In the left example, because the Avoidable Bias is larger (7%) then we need to focus on the bias. In the right example, because the Measure of Variance is larger (2%) then we need to focus on the variance.

You might have multiple human-level performances based on different groups of people. Choose the best human-level performance as it is closer to Bayes optimal error.

These techniques allow you to make decisions more quickly as to whether you should focus on trying to reduce the bias or trying to reduce the variance of your algorithm. This tend to work well until you surpass human-level performance, whereupon you might no longer have a good estimate of Bayes error that still helps you make this decision really clearly.

Surpassing human-level performance:
In some problems, deep learning has surpassed human-level performance. Like:

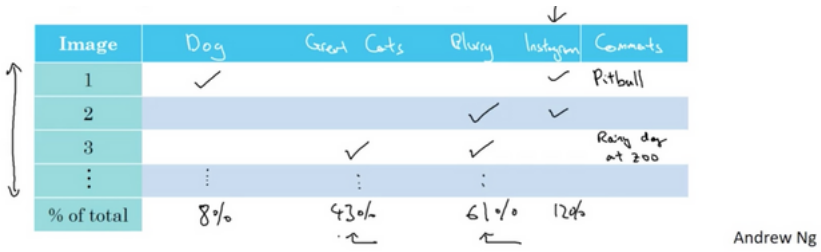
- Online advertising.
- Product recommendation.
- Logistics (predict transit time)
- Loan approval.

These examples are learning on structural data with lots of data, and they are not natural perception task. Humans behave well in natural perception tasks like medical, computer vision and speech recognition, and it's harder to surpass human-level performance in these tasks.

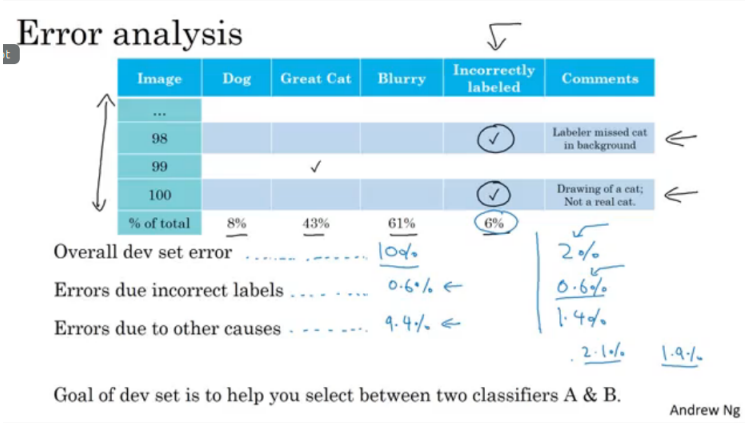
3.5 Error analysis

Error analysis - process of manually examining mistakes that your algorithm is making. It can give you insights into what to do next. Error analysis helps you to analyze the error before taking an action which could take unnecessary efforts.

Sometimes, you can evaluate multiple error analysis ideas in parallel and choose the best idea, by creating a spreadsheet of them. This quick counting procedure, which takes small numbers of hours, can really help you make much better prioritization decisions, and understand how promising different approaches are to work on.



Incorrectly Labeled Data:
DL algorithms are quite robust to random errors in the training set but less robust to systematic errors. Only fix these labels for training set if you can. If you want to check for mislabeled data in dev/test set, try error analysis with the mislabeled column.



If other errors take up the majority of dev set error, fixing mislabeled data is not the first priority. If mislabeled data is a significant part of dev set error, then you need to fix it.

Consider these guidelines while correcting the dev/test mislabeled examples:

- Apply the same process to your dev and test sets to make sure they continue to come from the same distribution.
- Consider examining examples your algorithm got right as well as ones it got wrong. (Not always done if you reached a good accuracy)
- Train and dev/test data may now come from a slightly different distributions. It's very important to have dev and test sets to come from the same distribution. But it could be OK for a train set to come from slightly other distribution.

Build First System Quickly, Then Iterate:
The steps you take to make your deep learning project:

- Setup dev/test set and metric.
- Build initial system quickly.
- Use Bias/Variance Analysis and Error Analysis to prioritize next steps.

Training and Dev/Test set on different distributions:
A lot of teams are working with deep learning applications that have training sets that are different from the dev/test sets due to the hunger of deep learning to data.

There is a strategy to split data into train, dev/test sets:

- Use data from real application feedback data for dev/test sets (for example user image, experience or feedback). This sets a more accurate target and reduces bias for the DL problem.
- Use purchased data, collected data from different sources, plus the real application feedback data for training set. This provides enough amount of data to reduce variance.

Bias and Variance with mismatched data distributions:
Bias and Variance analysis changes when training and Dev/test set is from the different distribution, because you need to consider the influence of distribution.

To solve this issue we create a new set called train-dev set as a random subset of the training set (so it has the same distribution with training set, and we don't use this part of training set to train the model) and we get:

$$\text{Avoidable Bias} = \text{Training error} - \text{Human (Bayes) error}$$

$$\text{Variance} = \text{Training-dev error} - \text{Training error}$$

$$\text{Data mismatch} = \text{Dev error} - \text{Training-dev error}$$

$$\text{Degree of Overfitting to Dev set} = \text{Test error} - \text{Dev error}$$

Addressing data mismatch:
There aren't completely systematic solutions to this, but there some things you could try.

- Carry out manual error analysis to try to understand the difference between training and dev/test sets.
- Make training data more similar to dev/test sets, or collect more data similar to dev/test sets.

If your goal is to make the training data more similar to your dev set, one of the techniques you can use is Artificial data synthesis. (Combine some of your training data with some other data and produce some data that similar to the dev/test set distribution.) For example, combine normal audio with car noise to get audio with car noise example, generate cars using 3D graphics.

Be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples because your NN might overfit these generated data (like particular car noise or a particular design of 3D graphics cars).

3.6 Transfer learning

Apply the knowledge you took in a task A and apply it in another task B. For example, you have trained a cat classifier with a lot of data, you can use part of the trained NN to solve x-ray classification problem.

To do transfer learning, delete the last layer(s) of NN and it's weights and:

- Option 1: if you have a small data set - keep all the other weights as fixed weights. Add new last layer(s) and initialize the weights for the new layer(s), then feed the new data to the NN and learn the new weights.

- Option 2: if you have enough data you can keep less fixed weights and retrain the weights for more layers.

Training on task A called pre-training and Option 1 and 2 are called fine-tuning.

When transfer learning make sense:

- Task A and B have the same input X (e.g. image, audio).
- You have a lot of data for the task A you are transferring from and relatively less data for the task B your transferring to.
- Low level features from task A could be helpful for learning task B.

3.7 Multi-task learning

In multi-task learning, you try to have one neural network do several things at the same time.

For example, build an object recognition system that detects pedestrians, cars, stop signs, and traffic lights (image has multiple labels). Then Y shape will be $(4, m)$ because we have 4 classes and each one is a binary one.

Multi-task learning makes sense:

- 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.
- Can train a big enough network to do well on all the tasks.

If you can train a big enough NN, the performance of the multi-task learning compared to splitting the tasks is better. Today transfer learning is used more often than multi-task learning.

3.8 End-to-end deep learning

Some systems break a task into multiple steps, and build a NN for each step. An end-to-end deep learning system implements all these steps with a single NN.

For example:

- Face recognition system. In practice, the best approach is the multi-steps approach for now. It uses one NN for face detection, and another NN which takes two faces as inputs, then outputs if the two faces are the same person or not.

$$\text{Image} - - > \text{Face detection} - - > \text{Face recognition}$$

$$\text{Image} - - - - - - - - - - - - - - - > \text{Face recognition}$$

- Machine translation system. Here end-to-end deep leaning system works better because we have enough data to train the NN.

$$\text{English} - - > \text{Text analysis} - - > \cdots - - > \text{French}$$

$$\text{English} - > \text{French}$$

To build the end-to-end deep learning system that works well, we need a big dataset (more data than in non end-to-end system). If we have a small dataset, the multi-steps approach could work better.

Whether to use end-to-end deep learning:
Key question: Do you have sufficient data to learn a function of the complexity needed to map X to Y? When applying supervised learning you should carefully choose what types of X to Y mappings you want to learn depending on what task you can get data for.

Pros of end-to-end deep learning:

- By having a pure deep learning approach, your NN learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.
- Less hand-designing of components needed.

Cons of end-to-end deep learning:

- May need a large amount of data.
- Excludes potentially useful hand-design components (it helps more on the smaller dataset).

Multi-steps approach could be better if a complex system can be divided into multiple tasks and they have a good mapping from data to tasks (autonomous driving), and you are able to build ML/DL models for individual components.

4 Convolutional Neural Networks

4.1 Computer Vision and Convolution

Computer vision is one of the applications that are rapidly active thanks to deep learning. Some of the applications of computer vision using deep learning includes self driving cars, face recognition, new types of art...

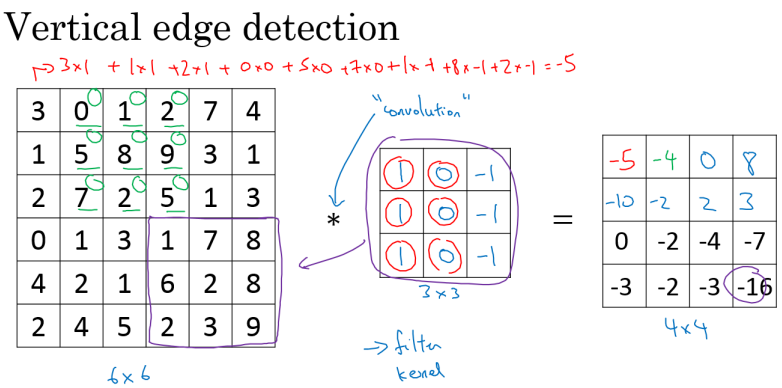
Examples of computer vision problems:

- Image classification.
- Object detection: detect object and localize them.
- Neural style transfer: changes the style of an image using another image.

One of the challenges of computer vision problem that images can be so large and we want a fast and accurate algorithm to work with that. For example, a $1000 \times 1000 \times 3$ image will represent 3 million feature/input to the full connected neural network. If the first hidden layer contains 1000 nodes, then we will have to learn weights of the shape [1000, 3 million] which is 3 billion parameter only in the first layer and that's so computationally expensive! One of the solutions is using convolution layers instead of the fully connected layers.

Edge Detection Examples
The convolution operation is one of the fundamentals blocks of a CNN. One of the examples about convolution is the image edge detection operation. Early layers of CNN might detect edges then the middle layers will detect parts of objects and the later layers will put the these parts together to produce an output.

In an image we can detect vertical edges, horizontal edges, or full edges. Here is an example of convolution operation to detect vertical edges:



We slide the 3×3 filter(kernel) matrix through the image matrix, each time take 1 step, then get the dot product of the matched matrix and the filter matrix, and the output will be a 4×4 matrix.

In TensorFlow the convolution operation is tf.nn.conv2d. In Keras it is Conv2d.

Notice that in math textbooks the convolution operation needs to flip the filter before getting the dot product. The "convolution" operation in deep learning field is called cross-correlation operation in mathematics field.

The vertical edge detection filter will find the 3×3 places in an image where it is bright on the left and dark on the right (or in reverse). If we apply this filter to a white region followed by a dark region, it should find the edge in between the two colors as a positive value. But if we apply the same filter to a dark region followed by a white region it will give us a negative value. To solve this we can use the abs function to make it positive.

Vertical edge detection matrix:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Horizontal edge detection matrix:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

There are a lot of ways we can put number inside the horizontal or vertical edge detections. For example, the vertical Sobel filter (The idea is taking care of the middle row):

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

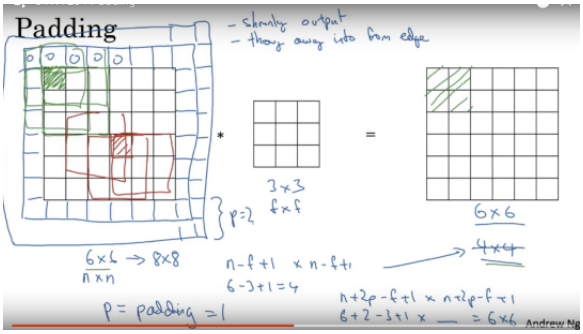
Scharr filter (The idea is taking great care of the middle row):

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

To implement convolution in a NN, instead of manually choosing the filter matrix, we can use a matrix of weights that can be learned in the training process, and use the data to learn the right filter matrix for the model. This approach is much more robust than manually choosing the filter.

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}$$

Padding:
In the last section we saw that a 6x6 matrix convolved with 3x3 filter/kernel gives us a 4x4 matrix. Also we noticed that the pixels in the center of image are used multiple times in the convolution, but the ones at the edge of the image are used less.



So the problems with convolutions are:

- We want to apply convolution operation multiple times, but if the image shrinks every time we will lose a lot of data in this process.
- The edge pixels are used less than central pixels in an image.

To solve these problems we need to use padding. Assume the input image is of shape $n \times n$, the filter is of shape $f \times f$, and we add p padding pixels (of value 0) to the input image. Then the output matrix will be of shape:

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

Choice of padding:

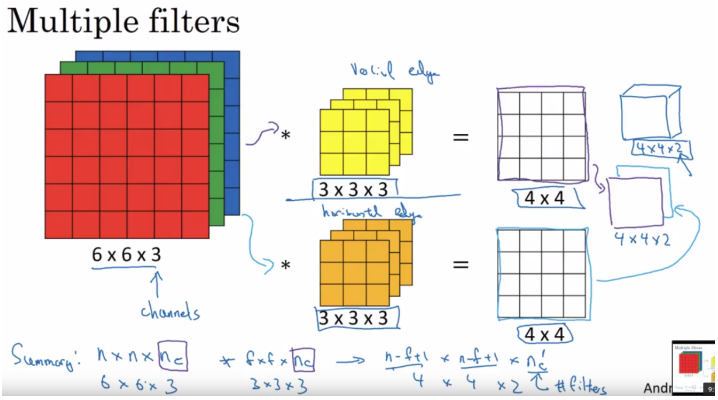
- Valid Convolution: $p = 0$
- Same Convolution: Pad so the output size is the same as input size, using $p = \frac{f-1}{2}$. This works when f is odd, and f is usually odd in a filter because it's nice to have a central pixel.

Strided convolution:
When deal with large images, instead of moving 1 step each time, we can move the filter matrix s steps each time when sliding through the image. Notice that the sliding doesn't happen when there is not enough room to do that. Then the output matrix will be:

$$floor(\frac{n + 2p - f}{s} + 1) \times floor(\frac{n + 2p - f}{s} + 1)$$

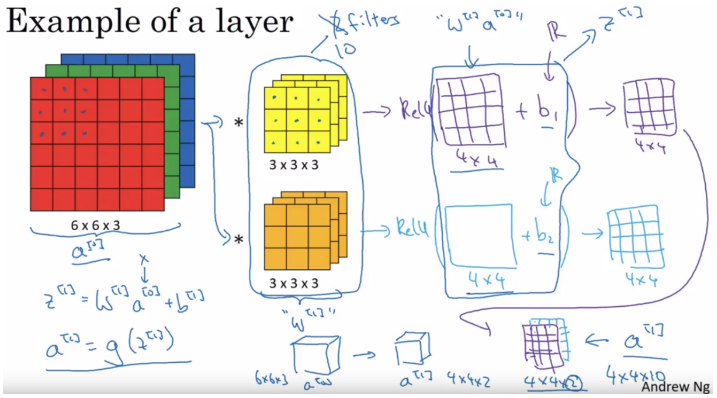
Convolutions Over Volumes
We can convolve an image of shape $(height, width, number\ of\ channels)$ (like RGB images) with stacked filters of shape $(height, width, same\ number\ of\ channels)$. To get the convolution of all the channels, we get the the convolution of each channel first and add them together.

The advantage of using multiple stacked filters is that it helps to detect multiple features in the image.



Assume the image is of shape $n \times n \times n_c$, with $n_c^{'}$ stacked filters of shape $f \times f \times n_c$, the output would be of shape $(n - f + 1) \times (n - f + 1) \times n_c^{'}$.

Convolutional Layer (Generalized)



If layer l is a convolutional layer,

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

$$filter\ width/height = f^{[l]}$$

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

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

$$filter\ channels = n_c^{[l-1]}$$

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

$$n_H^{[l]} = floor(\frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1)$$

$$n_W^{[l]} = floor(\frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1)$$

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

We can find that:

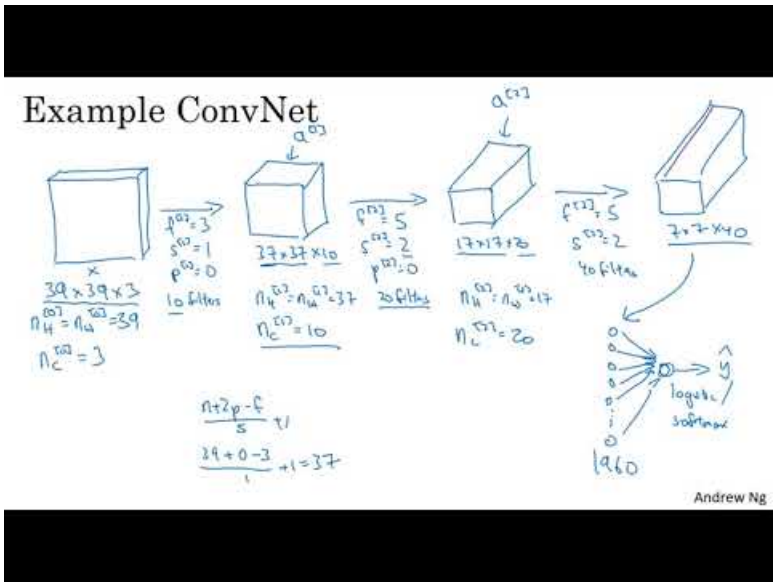
$$Each\ filter\ is\ of\ shape : f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$$

$$Weights\ is\ of\ shape : f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$$

$$Bias\ is\ of\ shape: n_c^{[l]}$$

$$A^{[l]}\ is\ of\ shape: m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$$

A simple convolution network example:
Here is a simple CNN example:



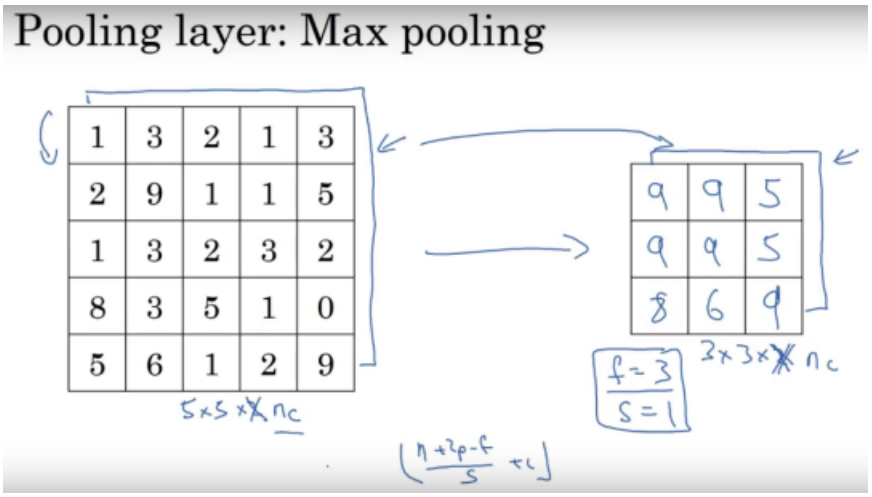
The image goes through 3 convolutional layers, then the output is flattened into a vector so that it can be fed into a logistic regression (or softmax) unit, to be able to classify the image.

Notice that for the outputs of the convolutional layers, when it goes deeper it's size is shrinking, while it's number of channels is increasing. You can find this trend in many other CNNs.

Types of layer in a convolutional network:

- Convolution Layer.
- Pooling Layer.
- Fully Connected Layer.

Pooling Layers:
Other than the conv layers, CNNs often use pooling layers to reduce the size of the inputs, speed up computation, and to make some of the features it detects more robust.



For a pooling layer, given an input image of shape $n_H \times n_W \times n_c$, with filter of shape $f \times f$, stride s, and for most of time $p = 0$, the filter slides through the image for each layer and only choose the max value of the matched region (max pooling). So the output will be of shape:

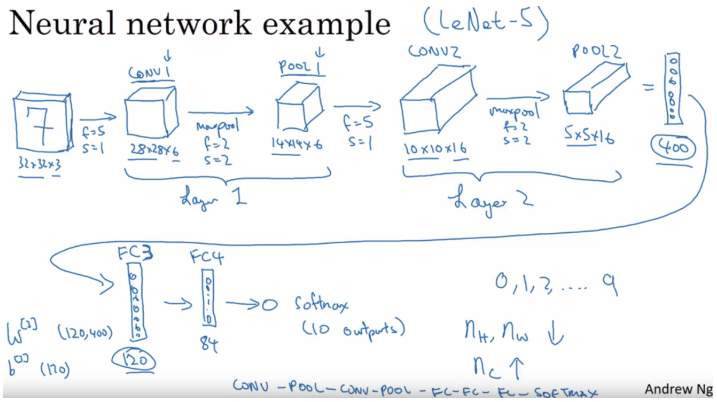
$$floor(\frac{n_H + 2p - f}{s} + 1) \times floor(\frac{n_W + 2p - f}{s} + 1) \times n_c$$

Notice that the parameters for pooling layer are fixed (manually picked) and they are not going to be learned during the training process.

Average pooling is taking the averages of the values of the matched regions instead of taking the max values. Max pooling is used more often than average pooling in practice.

CNN Example:

$$Input\ Image \longrightarrow Conv \longrightarrow Pool \longrightarrow Conv \longrightarrow Pool$$
$$\longrightarrow Flatten \longrightarrow Fully\ Connected \longrightarrow Fully\ Connected \longrightarrow Softmax$$



	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	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

From the table we can find:

- Hyperparameters are a lot. For choosing the value of each you can follow the guideline that we will discuss later or check the literature and takes some ideas and numbers from it.
- Usually the input width/height of convolution layer decreases over layers while the number of channels increases.
- A CNN usually consists of one or more convolution layers (Not just one as the shown examples) followed by a pooling layer.
- Fully connected layers has the most parameters in the network.

- To consider using these building blocks together you can look at other working examples first to get some intuitions.

Why Convolutions:

- Parameter sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
- Sparsity of connections: Fully connected layer causes overfitting if training set is small, and it has far more weights to train. In each convolution layer, each output value depends only on a small number of inputs which makes it translation invariance, and that reduces overfitting.

To train a CNN, given training set $[(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})]$

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

Use gradient descent to optimize parameters to reduce J.

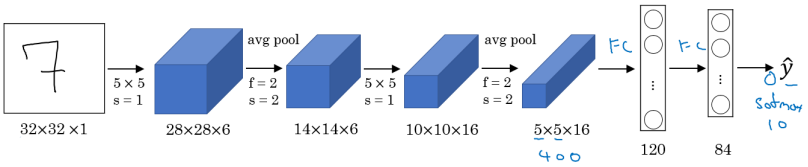
4.2 Deep CNN Models (case studies)

We learned about convolution layer, pooling layer, and fully connected layer. It turns out that computer vision researchers spent the past few years on how to put these layers together. To get some intuitions you can see the models that has been made. Some neural network architecture that works well in some tasks can also work well in other tasks.

Here are some classic CNN networks:

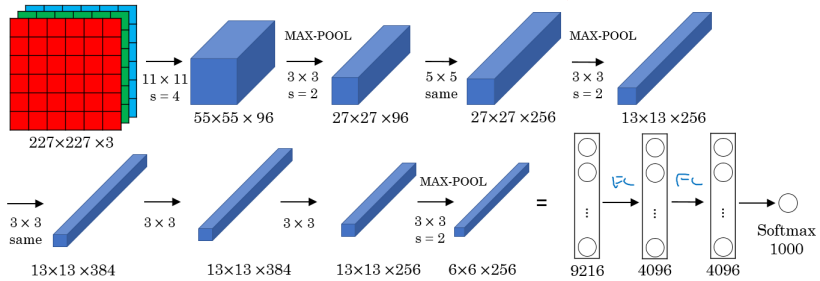
- LeNet-5
- AlexNet
- VGG
- ResNet
- Inception

LeNet-5:



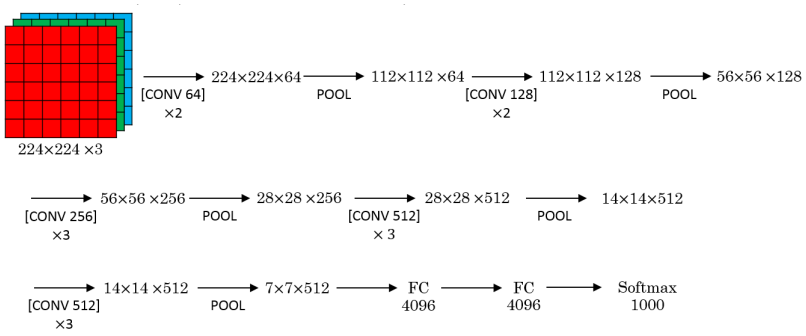
- The goal for this model was to identify handwritten digits in a $32 \times 32 \times 1$ gray image.
- This model was published in 1998. The last layer wasn't using softmax back then.
- It has 60k parameters.
- The width/height of the image decreases as the number of channels increases.
- $Conv \longrightarrow Pool \longrightarrow Conv \longrightarrow Pool \longrightarrow FC \longrightarrow FC \longrightarrow softmax$. This type of arrangement is quite common.
- The activation function used in the paper was Sigmoid and Tanh. Modern implementation uses RELU in most of the cases.

AlexNet:



- The goal for this model was the ImageNet challenge which classifies images into 1000 classes.
- $Conv \longrightarrow Max-pool \longrightarrow Conv \longrightarrow Max-pool \longrightarrow Conv \longrightarrow Conv \longrightarrow Conv \longrightarrow Max-pool \longrightarrow Flatten \longrightarrow FC \longrightarrow FC \longrightarrow Softmax$
- Similar to LeNet-5 but bigger. It has 60 Million parameters compared to 60k parameters of LeNet-5.
- It used the RELU activation function.
- The original paper contains Multiple GPUs and Local Response normalization (RN). Multiple GPUs were used because the GPUs were not so fast back then. Researchers proved that Local Response normalization doesn't help much.
- This paper convinced the computer vision researchers that deep learning is very important.

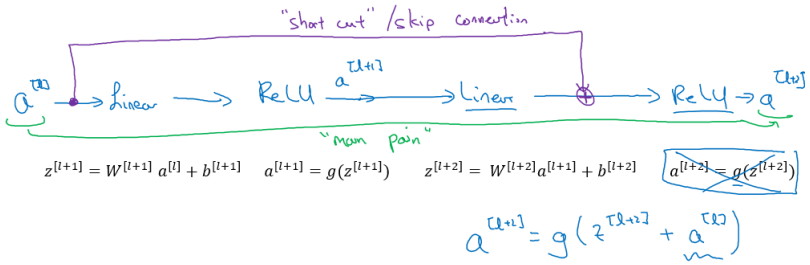
VGG-16:



- A modification for AlexNet. Instead of having a lot of hyperparameters it has a simpler network architecture.
- Focus on having only these blocks:
  - CONV: 3×3 filter, $s = 1$, same padding
  - MAX-POOL: 2×2 filter, $s = 2$
- This network is large even by modern standards. It has around 138 million parameters. Most of the parameters are in the fully connected layers.
- It has a total memory of 96MB per image for only forward propagation! Most memory are used in the earlier layers.
- Number of filters increases from 64 to 128 to 256 to 512. 512 was made twice.
- Pooling was the only one who is responsible for shrinking the image width/height.

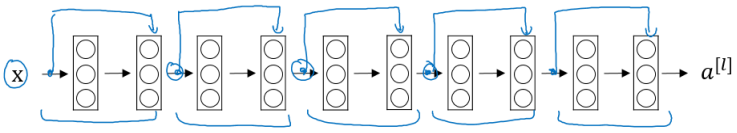
- There are another version called VGG-19 which is a bigger version. But most people uses the VGG-16 instead of the VGG-19 because it does the same.
- VGG paper is attractive because it tries to make some rules regarding using CNNs, like doubling the number of filters for convolution layers.

Residual Network:
Very, very deep NNs are difficult to train because of vanishing and exploding gradients problems. In this section we will learn about Residual Network which skips connections and take the activation from one layer and feed it directly to a much deeper layer. It allows you to train large NNs even with more than 100 layers.

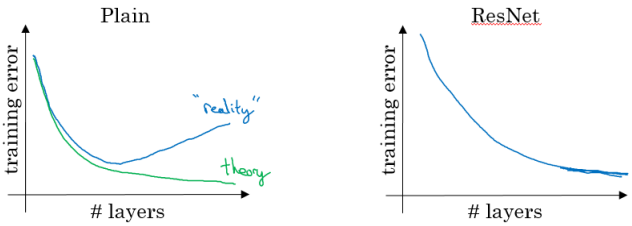


The NN above adds a shortcut/skip connection before the second activation. The researchers find that you can train a deeper NN by stacking blocks like this (residual blocks). Assume $z^{[l+2]}$ and $a^{[l]}$ have the same dimension:

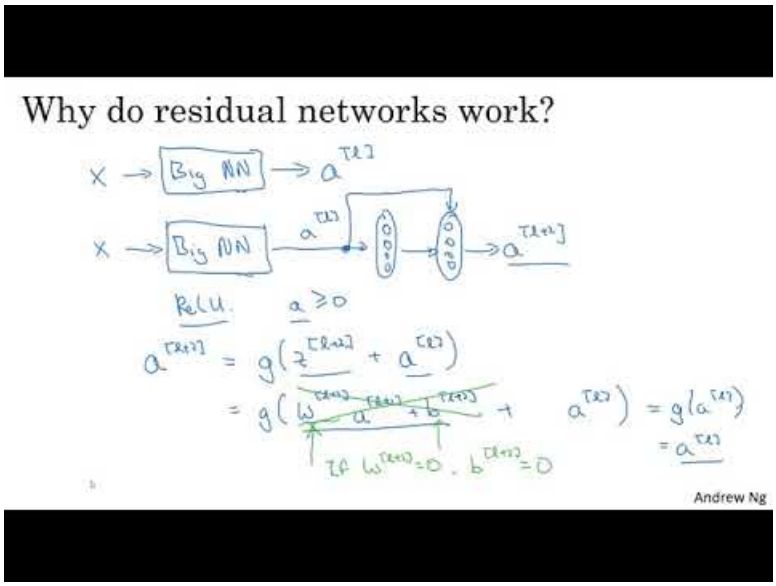
$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$



In the normal NN - Plain networks - the theory tells us that if we go deeper we will get lower training error, but because of the vanishing and exploding gradient problems the training error increases as it goes deeper. While for a Residual Network the training error keeps decreasing when going deeper.



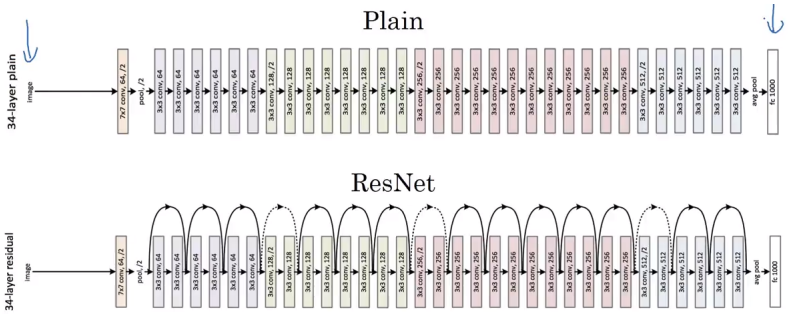
Why ResNets Work:



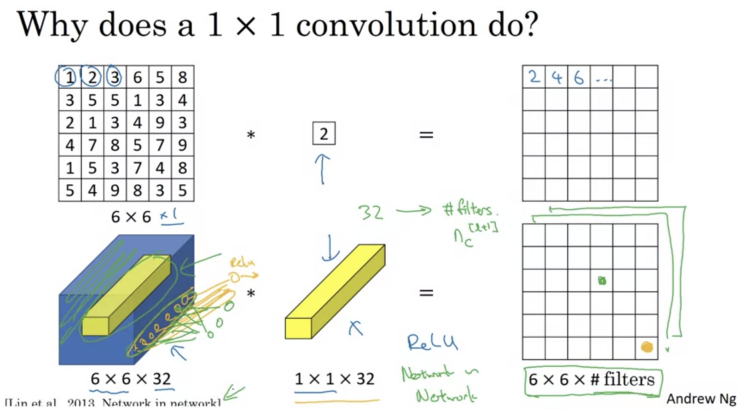
Notice that for above example $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$, if we apply L2 normalization it's not hard to make $z^{[l+2]} = 0$ by punishing the weights of $l+2$ layer, then $a^{[l+2]} = g(a^{[l]}) \approx a^{[l]}$. This means it's easy for Resnet to learn an identity function in a residual block, and this can achieve the same performance of an NN with less layers. It's also possible that a residual block can learn better weights than an identity function.

To make $z^{[l+2]}$ and $a^{[l]}$ have the same dimension, we can use Same Convolution. Or we can apply a weight matrix w_s to $a^{[l]}$ so that $a^{[l+2]} = g(z^{[l+2]} + w_s a^{[l]})$. w_s can be fixed matrix or can be learned in the training process. Another way is to add zero padding to $a^{[l]}$.

For example, the Resnet below used a lot of 3×3 Same Convolution layers.



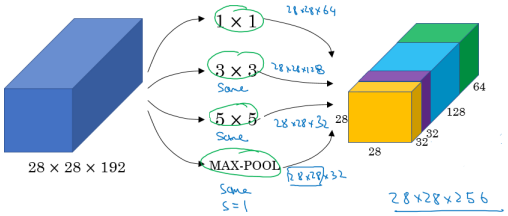
Network In Network (One by One Convolution):



A 1×1 convolution can be used for:

- Increase/decrease the number of channels (feature transformation).
- Shrinking the number of channels can save a lot of computations.
- It works similiar to a fully connected layer.

Inception Network Motivation:
When you design a CNN you have to choose what layer to use. Will you pick a 3×3 Conv or 5×5 Conv or maybe a max pooling layer. But in an inception network, they can be used all at once.



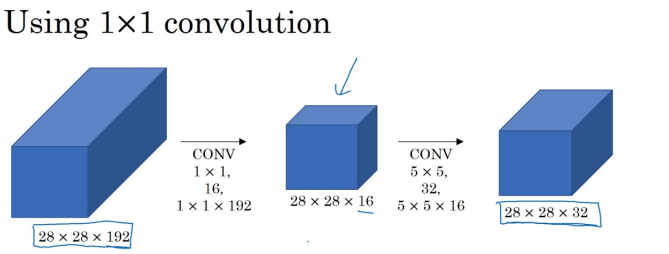
For the example above, the input of the inception module is $28 \times 28 \times 192$ and the output is $28 \times 28 \times 256$. We can apply all the Convs and pools we might want and then let the NN learn and decide which it wants to use most.

If we just focus on the 5×5 Conv in the example. There are 32 same filters of 5×5, and the input is $28 \times 28 \times 192$. Output is $28 \times 28 \times 32$. The total number of multiplications needed here are:

$$number\ of\ outputs \times filter\ size \times filter\ size \times input\ channels$$

$$= 28 \times 28 \times 32 \times 5 \times 5 \times 192 \approx 120\ mil$$

120 mil multiply operations is still a problem for the modern day computers. Using a 1×1 convolution we can reduce the multiply operations significantly.



For the 1×1 Conv layer, the number of multiplications:

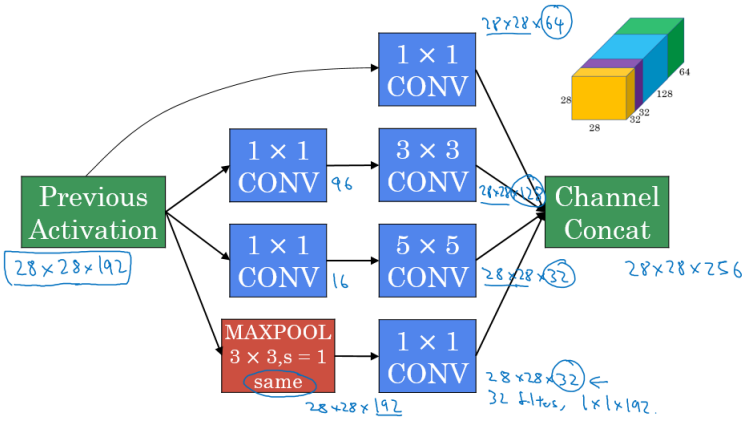
$$28 \times 28 \times 16 \times 1 \times 1 \times 192 \approx 2.5\ mil$$

For the second Conv layer, the number of multiplications:

$$28 \times 28 \times 32 \times 5 \times 5 \times 16 \approx 10\ mil$$

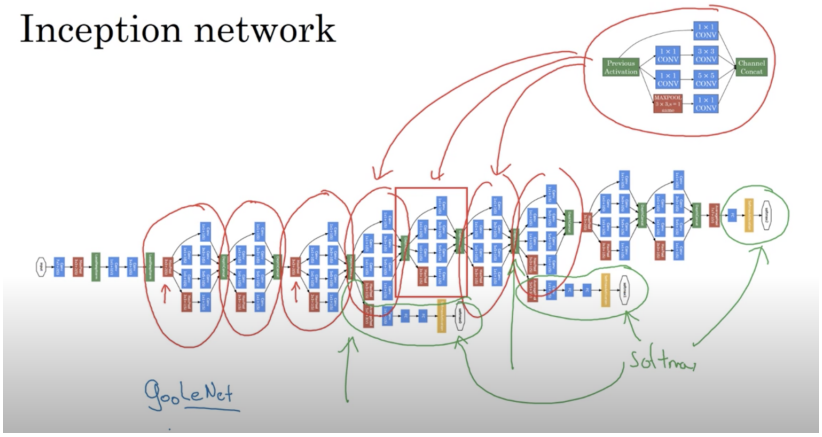
So the total number of multiplications reduces from 120 mil to 12.5 mil approx. The 1×1 Conv layer in this case is also called a Bottleneck Layer. If it is used for a proven reason, it can reduce the computation and it won't hurt the performance.

Inception Network:
Inception Network is made up with a series of Inception Modules. An Inception Module can have multiple parallel layers/blocks, and the output will concat all the channels from parallel layers/blocks. Here is an example of Inception Module:

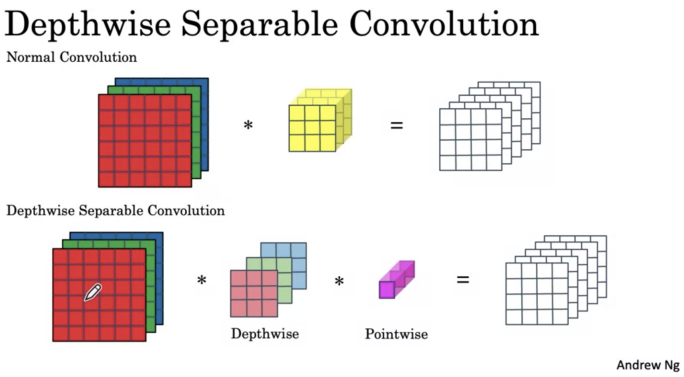


Inception Network example: GoogLeNet

- A Max-Pool block is sometimes used before the inception module to reduce the dimensions of the inputs.
- There are 2 Sofmax side branches which are used to predict the output label using the hidden layer. It helps to ensure that even the intermediate layers are not too bad to predict the output class of an image. It turns out that this has a regularization effect on the Inception Network and it helps prevent overfitting.

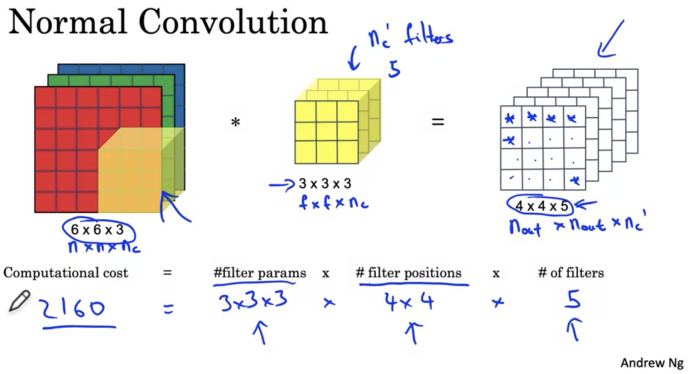


MobileNet:
MobileNet (Depthwise Seperable Convolution) can reduce the calculation of an NN. Using MobileNet will allow you to build and deploy new networks that work even in low compute environment, such as a mobile phone.



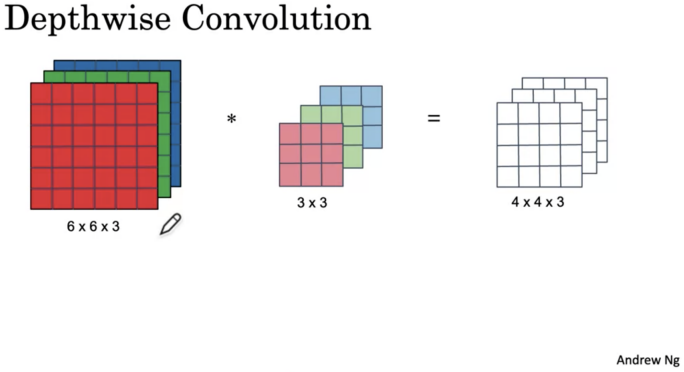
The number of multiplications that this Normal Convolution needs is:

$$4 \times 4 \times 5 \times 3 \times 3 \times 3 = 2160$$



Depthwise Separable Convolution can be divided into 2 steps: Depthwise Convolution and Pointwise Convolution.

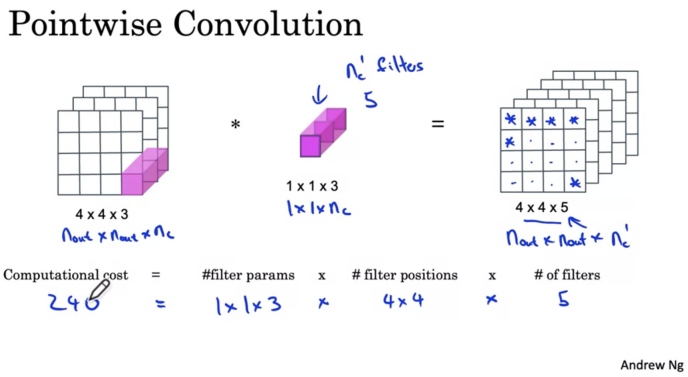
For the Depthwise Convolution, apply each channel of the filter to each channel of the input, then you get the output for each channel.



The number of multiplications that this Depthwise Convolution needs is:

$$4 \times 4 \times 3 \times 3 \times 3 = 432$$

For the Pointwise Convolution, it works as an one by one convolution.



The number of multiplications that this Pointwise Convolution needs is:

$$4 \times 4 \times 5 \times 1 \times 1 \times 3 = 240$$

So in total, the number of multiplications that this Depthwise Separable Convolution needs is:

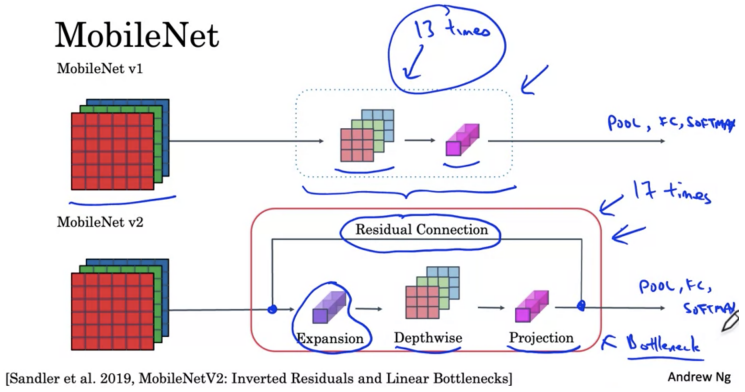
$$432 + 240 = 672$$

We can see Depthwise Separable Convolution saves a lot of computations from Normal Convolution. In a general case, for a Normal Convolution with filter size f, number of filters $n_c^{'}$, researchers found that:

$$\frac{Depthwise\ Separable\ Conv\ Computations}{Normal\ Conv\ Computations} = \frac{1}{n_c^{'}} + \frac{1}{f^2}$$

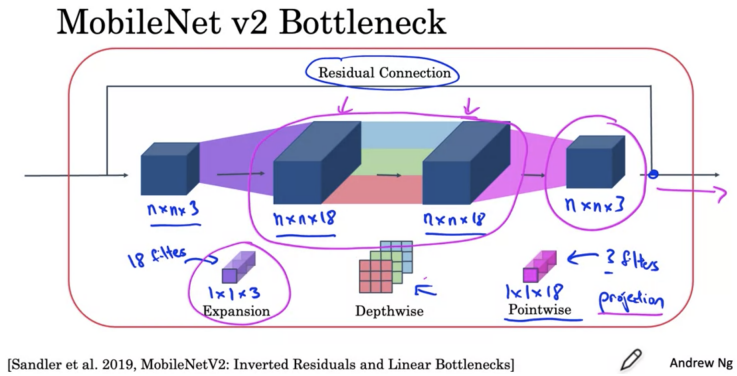
MobileNet V2:
For a MobileNet V1 network, it uses 13 Depthwise Separable Convolution Layers, followed by Pooling Layer, Fully Connected Layer and Softmax Layer in order to do classification.

For a MobileNet V2 network, it uses 17 "Bottleneck Blocks", followed by Pooling Layer, Fully Connected Layer and Softmax Layer in order to do classification.



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

The MobileNet V2 network uses a Bottleneck Block. It adds a Residual Connection in the block, and also an Expansion Convolution in front of the Depthwise Convolution. It's Projection Convolution is just another name of "Pointwise Convolution".



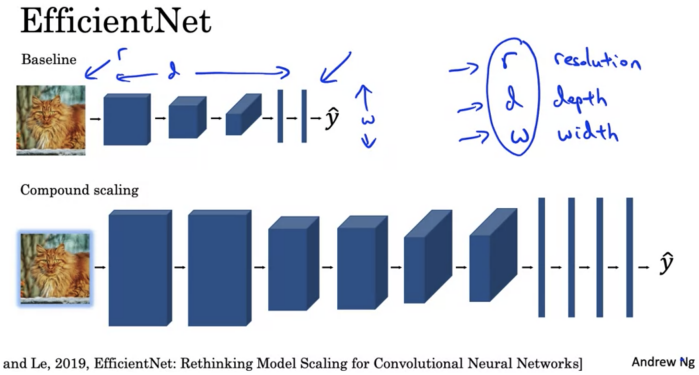
The Residual Connection works as a Residual layer. The Projection Convolution works as a Pointwise Convolution.

The Expansion Convolution expands the channels of the input. It increases the size of the representation of input within the bottleneck block, and this allows the neural network to learn a richer function. Then the Projection Convolution projects it back down to a smaller input, so that it won't hit the memory limit of a mobile device.

The cool thing about the Bottleneck Block is that it enables a richer set of computations, thus allow your neural network to learn richer and more complex functions, while also keeping the amounts of memory that is the size of the activations you need to pass from layer to layer, relatively small. That's why the MobileNet V2 can get a better performance than MobileNet V1, while still continuing to use only a modest amount of compute and memory resources.

EfficientNet:
EfficientNet gives you a way to automatically scale up or down neural networks for a particular device. If you have a little bit more computation, maybe you need a slightly bigger neural network and hopefully you get a bit more accuracy, or if you are more computationally constraint, maybe you want a slightly smaller neural network that runs a bit faster, at the cost of a little bit of accuracy.

Here are three things you can scale in an EfficientNet: input image resolution r, depth of network d, and width of layers w. Check an open source application that implements EfficientNet to learn how to scale r, d and w for a certain device for the best performance.

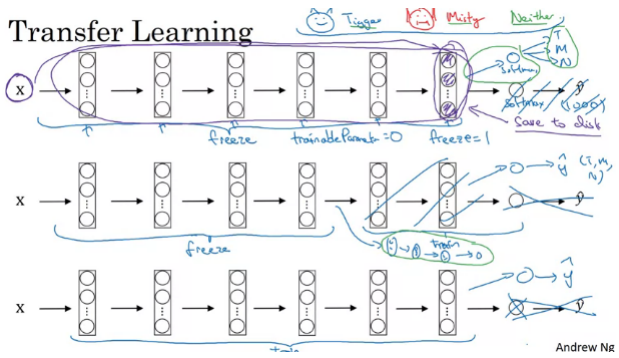


Using Open-Source Implementation:
You've now learned about several highly effective neural network and ConvNet architectures. It turns out that a lot of these neural networks are difficult to replicate because a lot of details about tuning of the hyperparameters such as learning decay and other things that make some difference to the performance.

It's sometimes difficult even for a higher deep loving PhD student, even at the top universities to replicate someone else's polished work just from reading their paper.

Fortunately, a lot of deep learning researchers routinely open source their work on the Internet, such as on GitHub. If you see a research paper whose results you would like to build on top of, one thing you should consider doing is looking online for an open source implementation. Some advantage of doing this is that you might download the network implementation along with its parameters/weights. The author might have used multiple GPUs and spent some weeks to reach this result and its right in front of you after you download it.

Transfer Learning:



If you don't have enough data to train your own network, you can go online and download some open source implementation of a neural network and download not just the code but also the weights. What you can do is then get rid of the original softmax layer and create your own softmax layer that outputs the prediction you need.

Because some of the layers are fixed (frozen) in transfer learning, one of the trick that could speed up training is we just pre-compute the activation of frozen part, for all the examples in training set, and save them to disk. Then you can feed the output from the frozen part to your softmax layer to make a prediction, and train the softmax layer on top of that. Using this trick you don't need to recompute those activations every time you take an epoch or pass through a training set.

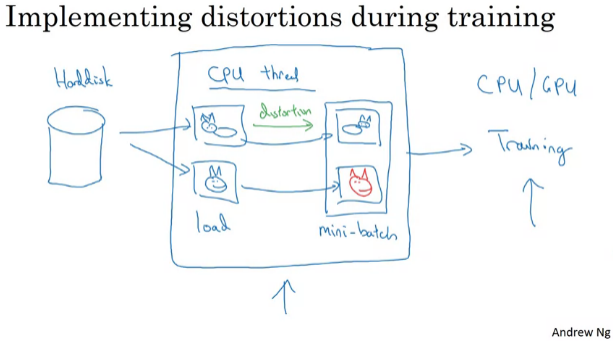
If you have more training data, your can keep and train the last a few layers and use their original weights as initialization, or replace them with your own layers and train them from random initialization.

If you have a lot of training data, you can retrain the whole model with its original weights as initialization, only replace the softmax layer with your own.

Data Augmentation:
Most computer vision tasks could use more data. Data augmentation is one of the techniques that is often used to generate more data and improve the performance of computer vision systems. Common data augmentation operations include:

- Mirroring.
- Random Cropping.
- Color Shifting. (PCA color augmentation)
- Rotation.
- Shearing.
- Local Warping.

Implement distortions during training:
You can load the data and implement distortions in a CPU thread to create a minibatch, then feed it into another CPU/GPU thread for training, so that they can run in parallel.



State of Computer Vision:
Currently there is a lot of data for Speech Recognition, enough data for Image Recognition but only limited data for Object Detection. You can see that with a lot of data, model doesn't have to be complicated to achieve a good performance. Simple model can work well with a lot of data. But if there is not enough data provided, more hand engineering hacks are required when designing the model to make it perform well, which makes the model more complex. Transfer Learning can also help with resolving this problem.

The learning algorithm has two sources of knowledge.

- Labeled data. The (x,y) pairs you use for supervised learning.
- Hand Engineering features/network architecture/other components.

The Computer Vision model is trying to learn a very complex function, and it is heavily relying on Hand Engineering because of the lack of data.

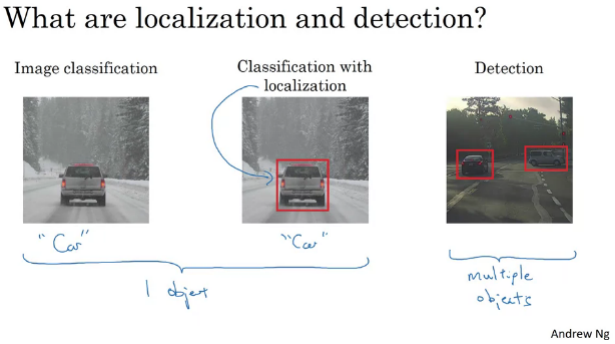
Using following techniques can sometimes give you a better result in a competition. But because it takes extra time in the run time, they are not usually used in production. Tips for doing well on benchmarks/winning competitions:

- Ensembling. Train several networks independently and average their outputs.
- Multi-crop at test time. Crop each of the image in the test set on different regions, do a prediction for each cropped image and average the results.

Use open source code:

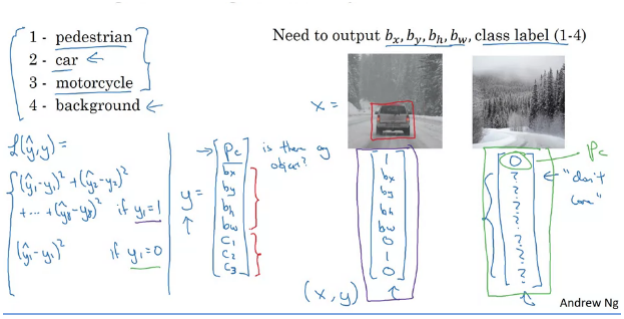
- Use architectures of networks published in the literature.
- Use open source implementations if possible.
- Use pretrained models and fine tune on your data set.

4.3 Object Detection



- Image Classification: No localization. Classify the image.
- Image Classification with Localization: Find the bounding box of the object. Classify the image.
- Object Detection: Find the bounding boxes of all the objects. Classify the objects.

Object Localization:



Assume there is only 1 object in the image. To localize the object in the image and classify it, you need to construct your neural network and make the output contains:

- p_c: Is there an object in the image (1 or 0).
- b_x: Part of bounding box center coordinate (b_x, b_y) (percentage as decimals).
- b_y: Part of bounding box center coordinate (b_x, b_y) (percentage as decimals).
- b_h: Bounding box height (percentage as decimals).
- b_w: Bounding box width (percentage as decimals).
- $c_1, \ldots, c_K$: Is the object of class $c_1, \ldots, c_K$ (1 or 0).

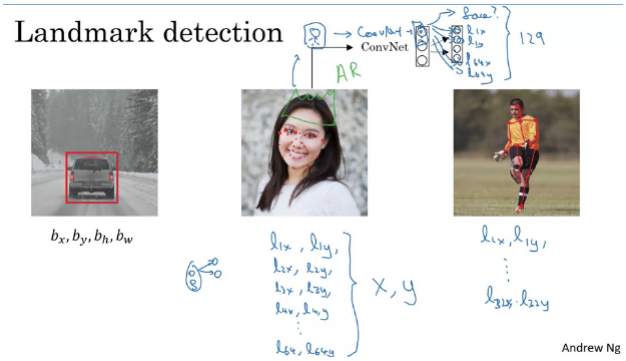
$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ \vdots \\ c_K \end{bmatrix}$$

Assume y is of shape $(P, 1)$. A simple squared error loss function can be:

$$L(\hat{y}, y) = \begin{cases} \sum_{p=1}^{P} (\hat{y}_p - y_p)^2 & \text{if } y_1 = 1, \text{object exists} \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0, \text{object doesn't exist} \end{cases}$$

To get a better loss function, you can use log likelihood loss for $c_1, \ldots, c_K$, squared error for b_x, b_y, b_h, b_w, logistic regression loss for p_c.

Landmark Detection:



You can have a neural network to output coordinates of important points on the image, which are called landmarks. You can construct your neural network and make the output contains:

- p_c: Is there an object on the image (1 or 0).
- $l_{1x}, l_{1y}, \ldots, l_{Mx}, l_{My}$: M important coordinates of the object (percentage as decimals).

$$y = \begin{bmatrix} p_c \\ l_{1x} \\ l_{1y} \\ \vdots \\ l_{Mx} \\ l_{My} \end{bmatrix}$$

Landmark Detection is used in tasks like emotion recognition and pose detection.

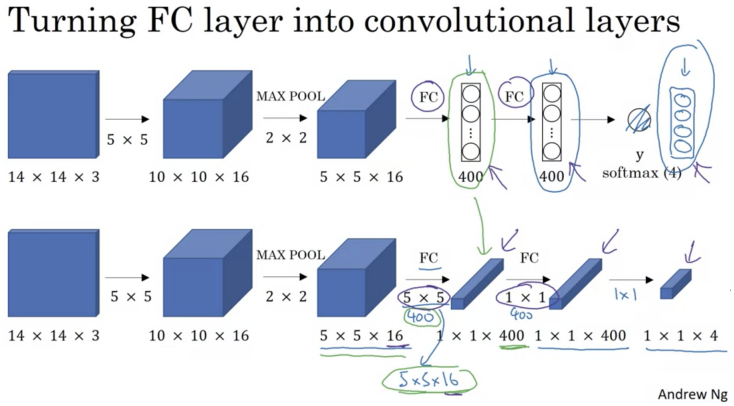
Object Detection:

To detect cars on an image, first you need to train a Conv net on cropped car images and non car images, so that it can recognize car images. Then, apply sliding windows detection algorithm:

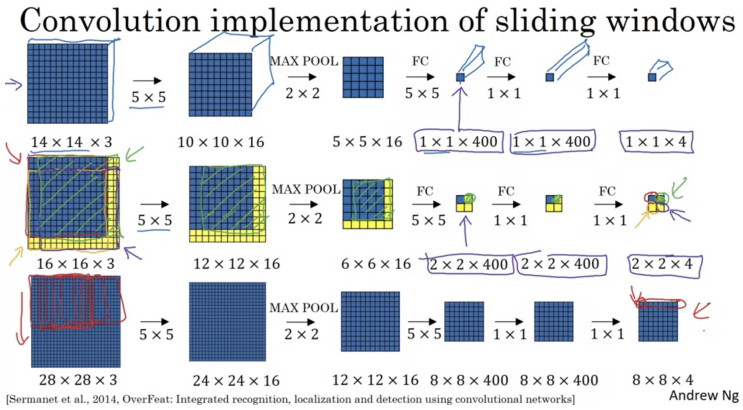
- Pick a window size.
- Slide the window through the image. Make sure it covers all the regions on the image.
- For each matched region feed the region into the Conv net to see if it's a car or not.
- Pick larger/smaller window sizes and repeat the above process.
- Keep the matched regions that contain the cars. If two or more regions intersect choose the region with the highest confidence.

A disadvantage of sliding windows is that it is computationally expensive. To solve this problem, we can implement the sliding windows with a convolutional approach.

Convolutional Implementation of Sliding Windows:



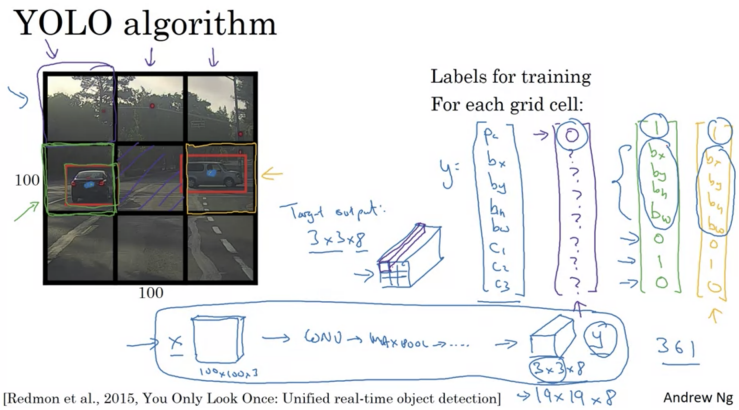
From the image above, we can see that the FC layers in the first NN can be replaced with Conv layers in the second NN, and they are doing the same thing mathematically.



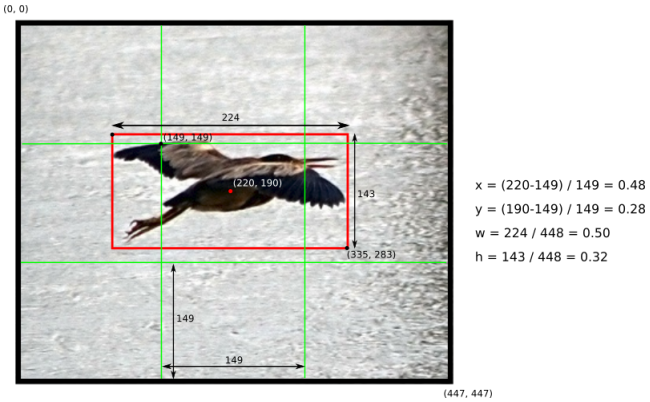
When you feed the matched region of sliding windows each time, a lot of regions intersect with each other, and you are doing duplicated computations. Instead of feeding the matched regions to the NN individually, we can replace this process with a Conv layer, and apply filters as sliding windows to all the matched regions from the image. Then the results are passed to the rest of the NN, in the end we get the predictions for all the matched regions in the final output.

The weakness of this algorithm is that sometimes none of the sliding windows is exactly on the object you want to recognize.

Bounding Box Predictions:
YOLO (You Only Look Once) Algorithm



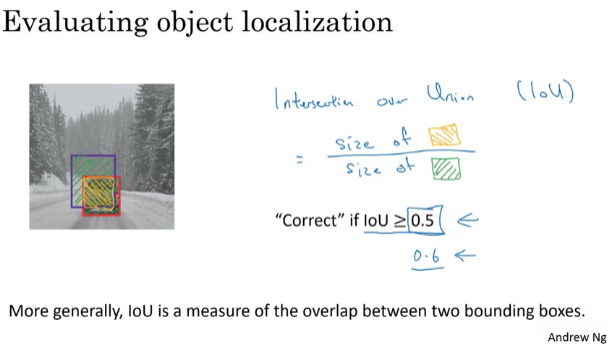
Assume the original image is of shape $(100, 100, 3)$, then you can divide the image into 3×3 grid cells (or 19×19 which avoids multi-objects in one grid cell). Apply Object Localization algorithm to each of the grid cell by using a CNN, in the end we get predictions for all the grid cells in the final output, which is of shape $(3, 3, 8)$. In the training data the object will be assigned to the grid cell which contains the center point of the object.



Notice that b_x, b_y are relative to the position of the grid (top left coner). b_h, b_w are relative to the size of full image. They should be converted to percentage as decimals.

One of the advantages of the YOLO algorithm is that it has a great speed and a Conv Net implementation. YOLO still works when there are more than one objects in the same grid cell and it will be explained later.

Intersection Over Union:

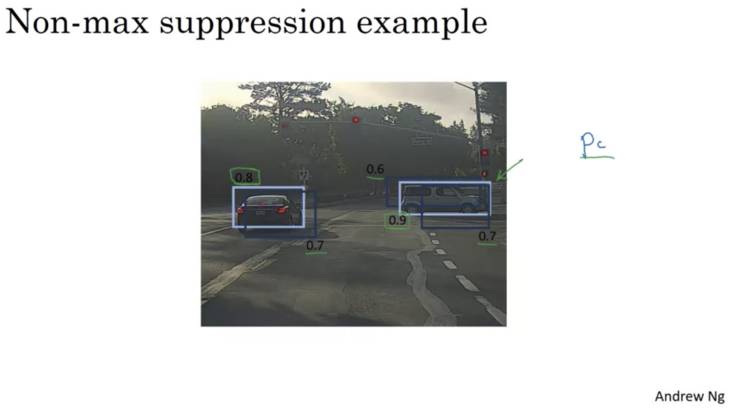


Intersection Over Union is a function used to evaluate the object detection algorithm. You can use this function to convert localization to accuracy. It is computed as the intersection area divided by the union area of predicted bounding box and labeled bounding box.

$$IoU = \frac{Intersection\ Area}{UnionArea}$$

Normally if $IoU \geq 0.5$ then it's considered a correct localization.

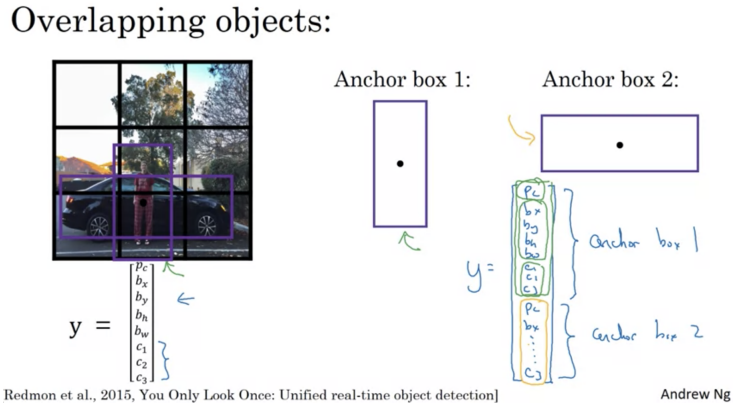
Non-max Suppression:
Non-max Suppression means that you're going to output your maximal probability predictions but suppress the overlapped ones that are non-maximal. This can make sure that YOLO detects the object just once.

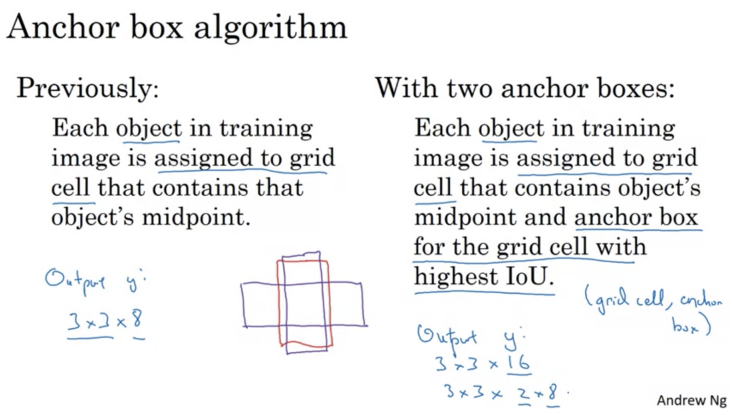


Given a list of predictions made on the image, divide them into different groups by predicted class. For each group:

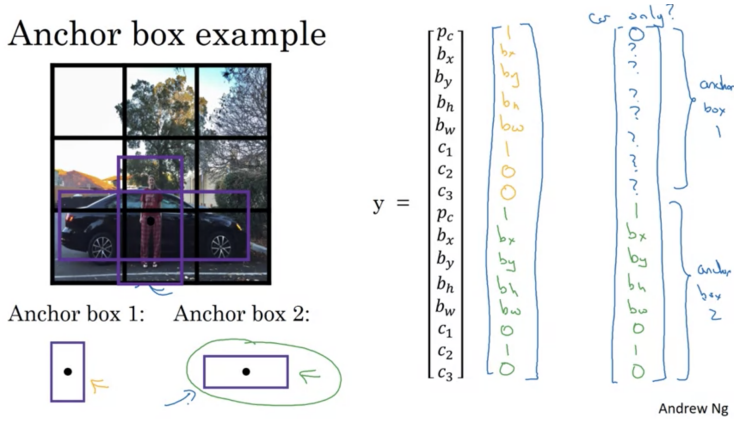
- Discard all predictions with $p_c \leq 0.6$ (Probability of Detection).
- Repeat while there are any remaining boxes in the group:
  - Pick the box with the largest p_c as an output prediction. Then remove it from the prediction group.
  - Discard any remaining box with $IoU \geq 0.5$ with the output box in the previous step.

Anchor Boxes:
In the previous YOLO example, a grid cell only detects one object. What if a grid cell wants to detect multiple objects? You can apply anchor boxes with different size to solve this problem.





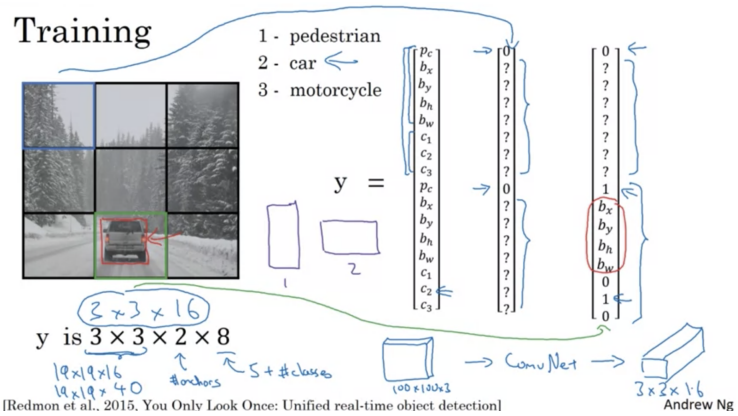
- YOLO without anchor boxes: each object in training image is assigned to grid cell that contains that object's center point.
- YOLO with anchor boxes: Each object in training image is assigned to grid cell that contains object's center point, and the anchor box for the grid cell with highest IoU. (overlap the center points of ground truth bounding box and the anchor box to calculate their IoU)



In the above example, there are 2 anchor boxes with different size(width, height). So the prediction for each grid cell is of shape $(1, 1, 16)$, in which the first 8 numbers are for anchor box 1, and the following 8 numbers are for anchor box 2. And the final output is of shape $(3, 3, 16)$. Then this model is able to detect 2 objects in the same grid cell.

You can manually choose or use some advanced algorithm to generate a lot more anchor boxes (with different width and height) to make the model more powerful.

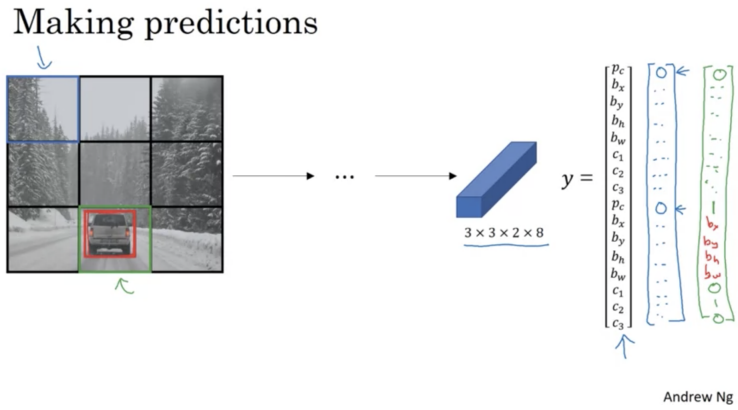
YOLO Algorithm Summary:



Construct training set: assume your model needs to detect 3 types of object: pedestrian, car, motorcycle. You pick 2 anchor boxes for them and the input image is divided into 9 grid cells. So the output of the Conv Net will be of shape $(3, 3, 16)$. You go through each of the grid cells and form the appropriate target output.

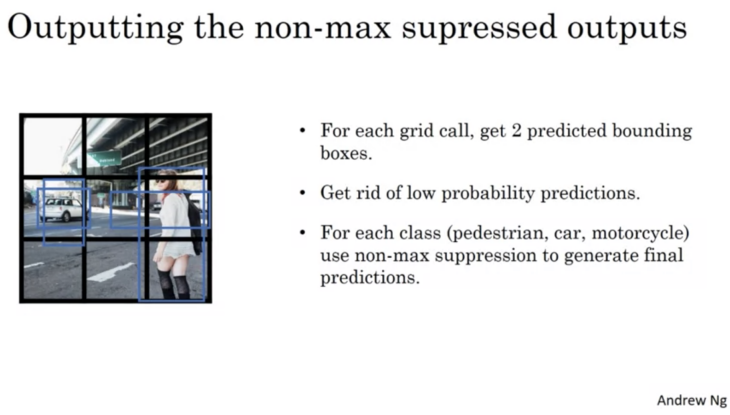
- For the first grid cell, there is nothing to detect, so $p_c = 0$ for both anchor boxes, and in this case we don't care the rest of numbers (?).
- For the 8th grid cell, the bounding box of car object has a higher IoU with the 2nd anchor box, so it is assigned to the lower portion of the prediction. No object is assigned to the 1st anchor box so it's p_c will be 0.

Your model takes a $(100, 100, 3)$ image as input, then use a Conv Net to make the prediction, which is of shape $(3, 3, 16)$.



Expected predictions: given a training sample, for each grid cell

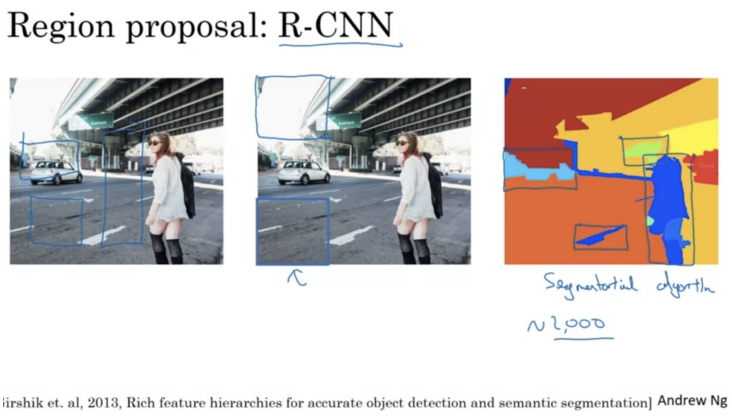
- If there is an object assigned to the grid cell and an anchor box in the training sample, it is expected that the prediction will contain $p_c = 1$, (b_x, b_y, b_h, b_w) as the bounding box of the object, and the correct class (c_1, c_2, c_3) in the corresponding anchor box position.
- If there is no object assigned to an anchor box, $p_c = 0$ is expected for the anchor box position, the rest of numbers for that anchor box position are not taken care of (?).



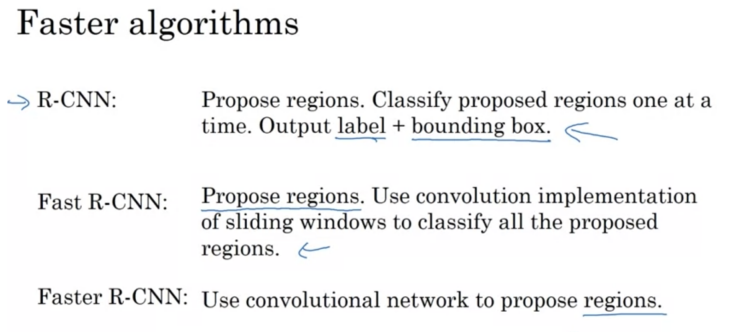
Implement non-max supression: the prediction made by the Conv Net won't be the same as expected outputs

- Each of the grid cell in the prediction will contain 2 predicted bounding boxes with a $p_c \in [0, 1]$.
- Implement non-max supression for all the predicted bounding boxes. The result will be the final output for the YOLO algorithm.

Region Proposals (R-CNN):

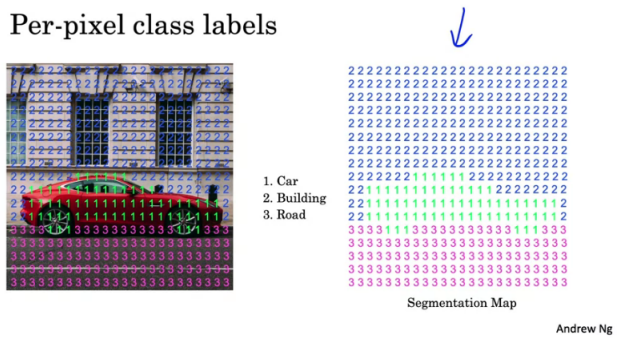


When using sliding window or YOLO, sometimes the model has to make predictions for some blank regions, where there is obviously no object. R-CNN uses a segmentation algorithm to pick only a few regions on the image that make sense to run your Conv Net on. In the example above, it picks 2000 blocks using the segmentation algorithm, and then make predictions on these blocks.

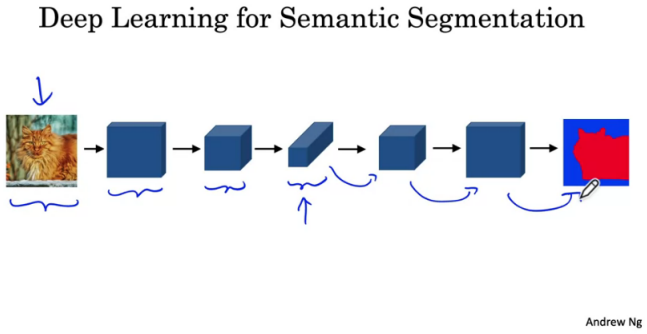


R-CNN is slow. Fast R-CNN uses a convolutional implementation of sliding window. Faster R-CNN uses Conv Net to propose regions. But they are still slower than YOLO, which is recommended by Andrew.

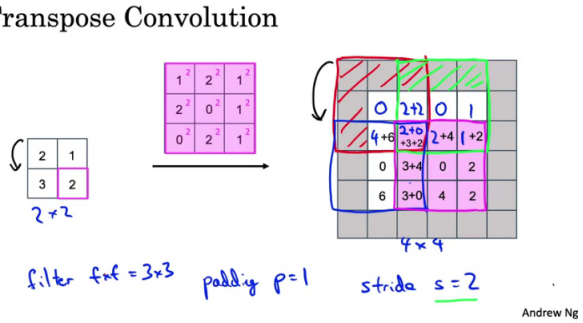
Semantic Segmentation with U-Net:
Instead of drawing bounding boxes, semantic segmentation labels every pixel in the image to a class.



In order to label every pixel on the image, the width/height of Conv layers shrinks first, then grows back. While the number of channels increases at first, then shrinks back. Transpose convolution is used to expand the width/height and reduce the channels.



Transpose Convolutions:

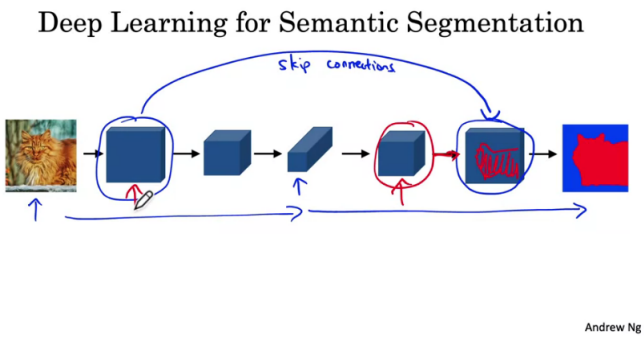


Assume the input is of shape $(2, 2)$ and we want an output of shape $(4, 4)$, with a filter of shape $(3, 3)$, padding $p = 1$ (apply to output), stride $s = 2$ (apply to output).

- Pick the first element on the input, multiply it with the filter, and map the result to the beginning area of the output. Notice that the values mapped to the padding are ignored.
- Repeat until input is traversed: Move 1 step on the input, multiply current input element with the filter, move s steps on the output and map the result to the current output area. If there is overlapping with previous output areas, add the overlapped values together.

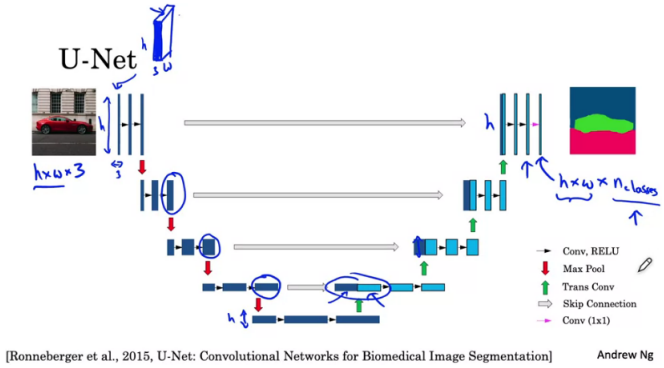
There are many ways to map the input to an output with larger width/height. Why do it in this way? It turns out the model performs well after this type of layers learn the weights.

U-Net Intuition:



The U-Net uses transpose convolutions to recover the dimensions. Also the skip connections (copy and concat) from earlier layers to later layers make the model perform better. The reason is that the earlier layers contain a lot of high level resolution low level feature information that later layers don't have (lost when the width/height shrinks). Combined with the low resolution high level spacial/contextual information provided by the later layers, it makes the model do a better prediction.

U-Net Architecture:



The example above is a U-Net. The input is a image of shape $(h, w, 3)$, and the output is of shape $(h, w, n_{classes})$, where $n_{classes}$ dimension contains the confidence of each class the pixel may belong to.

Notice that the activations from layers on the left hand side is copied and concatenated with the activations of layers on the right hand side, which is called skip connection.

4.4 Face Recognition

Face recognition system identifies a person's face. It can work on both images and videos.

Face verification vs. Face recognition

- Verification:
  - Input: image, name/ID. $(1 : 1)$
  - Output: whether the input image is that of the claimed person.
- Recognition:
  - Has a database of K persons
  - Get an input image
  - Output ID if the image is any of the K persons (or not recognized)

One Shot Learning:
It's not a good idea to train a neural network to take an image and figure out which person it is based on the knowledge in the database. Because every time when a new guy is added into the system, you need to train the model again.

Instead, you can train a model to compare 2 images and output the degree of difference between them.

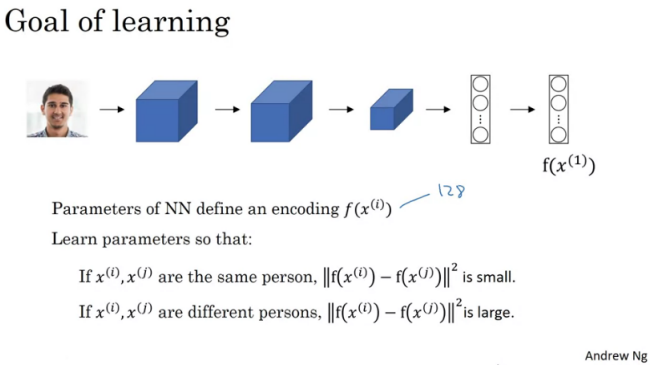
$$d(img1, img2) = \text{degree of difference between images}$$

$$Same\ Person = \begin{cases} true, & if\ d(img1, img2) \leq \tau \\ false, & if\ d(img1, img2) > \tau \end{cases}$$

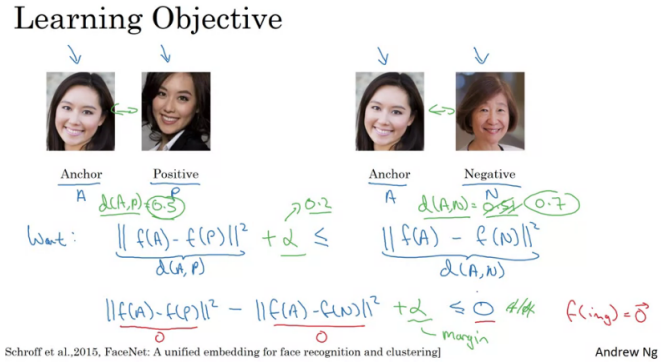
With this model, you can compare the input image with all the images in the database, if you can find one with a difference small enough, then output true, otherwise output false.

Siamese Network:
To compare 2 person images, we can build an NN to output the encodings of different images, then compare the encodings to see the degree of difference.

Assume the siamese network outputs $f(x^{(i)})$ for input $x^{(i)}$, $f(x^{(j)})$ for input $x^{(j)}$. if $||f(x^{(i)}) - f(x^{(j)})||^2$ is small, $x^{(i)}$ and $x^{(j)}$ are same person. If it's large, they are different person.



Triplet Loss:



To get triplet loss, we will be looking at 3 images at a time: Anchor, Positive, Negative, where Anchor is the target person, Positive is the same person, Negative is a different person. We expect the degree of difference between A and P is smaller than that between A and N:

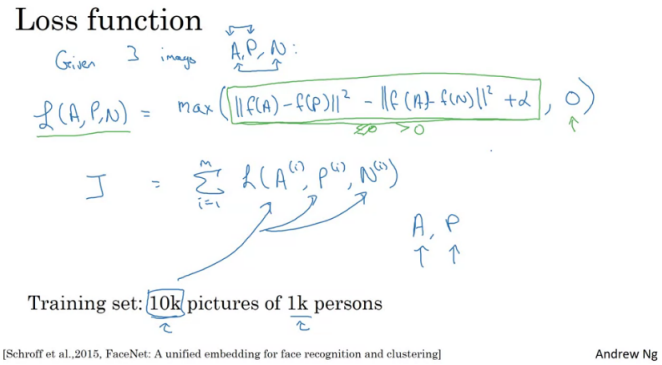
$$d(A, P) \leq d(A, N)$$

$$||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2$$

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 \leq 0$$

To push the gap between $||f(A) - f(P)||^2$ and $||f(A) - f(N)||^2$ larger, we can add a margin α to further separate apart these 2 differences.

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 \leq -\alpha$$
$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0$$



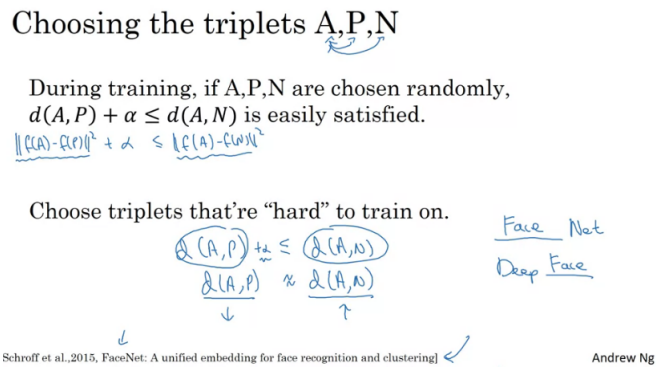
Given images A, P, N, define triplet loss function L:

$$L(A, P, N) = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0)$$

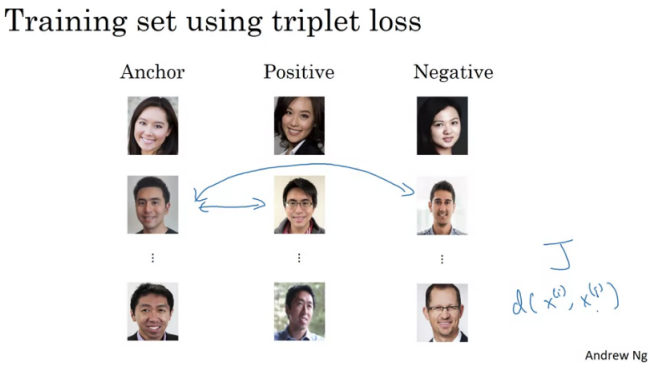
With m training examples, define cost function J:

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

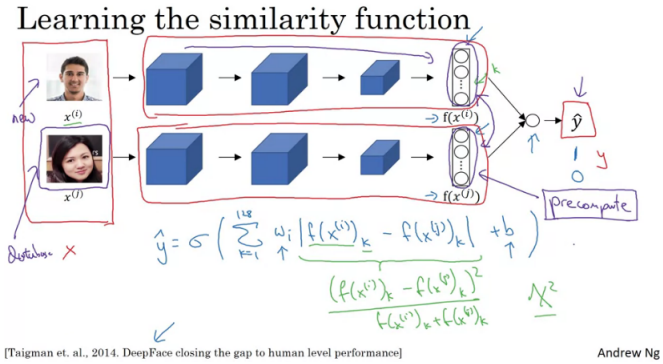
Notice that you need multiple pictures of the same person in your training set to use this loss function.



If you choose the A, P, N randomly for the training set, the constraint $||f(A)-f(P)||^2-||f(A)-f(N)||^2+\alpha \leq 0$ is very easy to satisfy and the model won't be able to learn much from it. To learn a more powerful model, we need to choose triplets that hard to train on, where $d(A, P) \approx d(A, N)$.



Face Verification and Binary Classification:



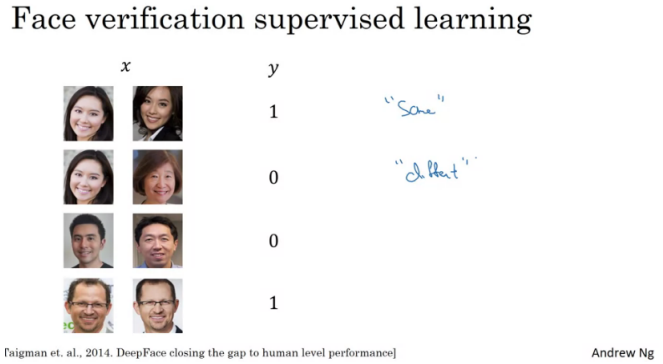
Face Verification can also be solved as a binary classification problem. You can build a pair of NNs to output the encodings of the 2 images, and feed them into a logistic regression unit to make a prediction (0 for different person, 1 for same person).

Assume the encodings has K elements in total, then the logistic regression unit:

$$\hat{y} = \sigma(\sum_{k=1}^{K} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b)$$

Or use χ^2 similarity:

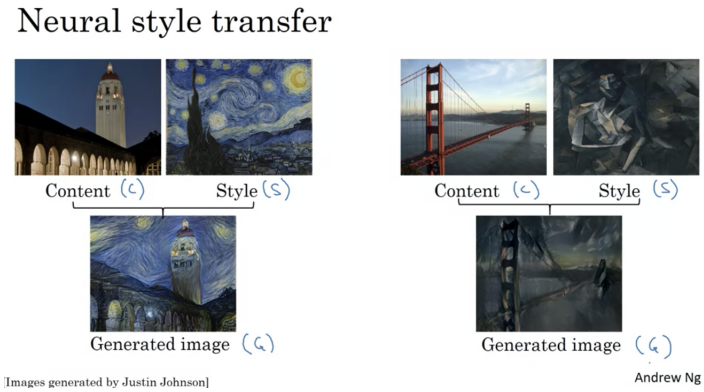
$$\hat{y} = \sigma(\sum_{k=1}^{K} w_k \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k} + b)$$



You can precompute the encodings for the images in the database and store them on the disk, so that you don't need to compute them in the running time to make a prediction.

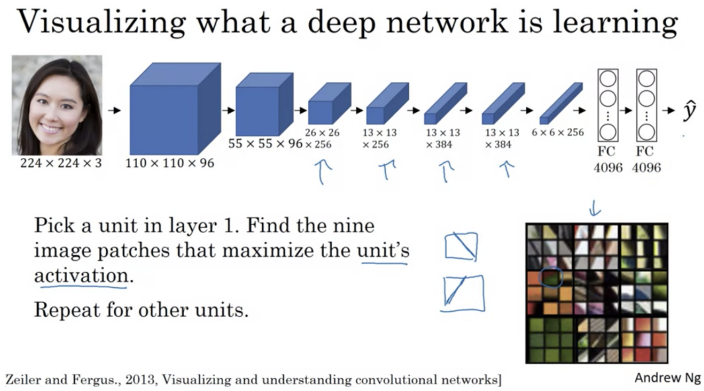
4.5 Neural Style Transfer

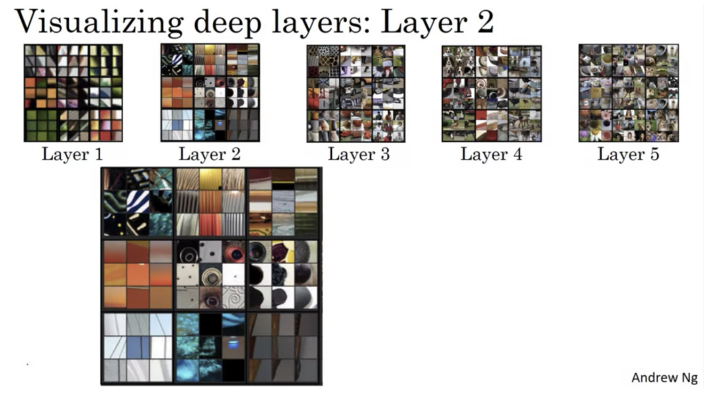
Neural Style Transfer can combine the features from 2 images and generate a new image based on that. It takes the content from Content image and the style from Style image.



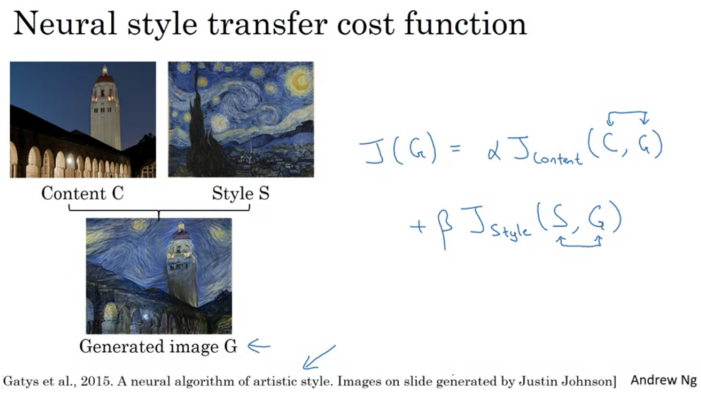
What are deep ConvNets learning?
Given an image classifier NN. Observe 1 unit in a layer. Find the training image patch that maximizes the activation of this unit. Crop out the areas on the original images, in this image patch, that are associated with this unit. Do this to other units in other layers as well to observe the difference between the areas cropped out from the original images associated with those units.

We can find that, for the first a few layers, the cropped areas are small and they contain some simple features like edges. When it goes deeper, the cropped areas are larger and they contain more complex shapes like circle and square. In the deepest layers, the cropped areas are large and they contain complex objects like person and dog.





Neural Style Transfer Cost Function:



Use a pre-trained ConvNet to extract features from "content" and "style" images in the hidden layers (e.g. VGG). The output of this pre-trained ConvNet can be ignored, since we are going to update generated image rather than the weights of the model to reduce the cost, and we are going to use the following cost function:

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

Find the generated image G:

- Initiate G randomly: $100 \times 100 \times 3$
- Use gradient descent to minimize $J(G)$:

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

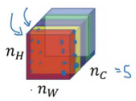
Notice that "content" and "style" are not strictly defined. They are just compared with generated image in different ways to get different costs.

Content Cost Function:

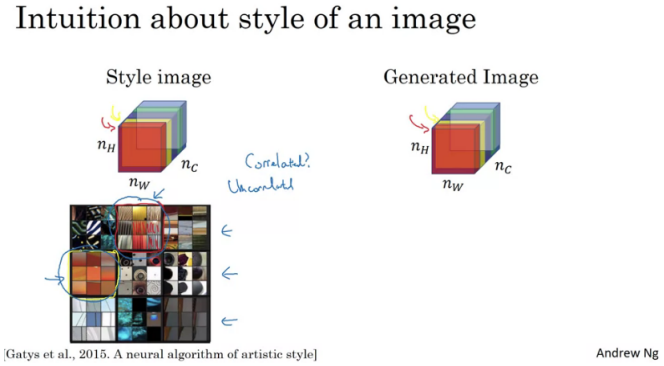
- Choose hidden layer l in the middle of NN (neither too shallow nor too deep) to compute the content cost.
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images. If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content.

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

Style Cost Function:



Correlation of pixels between different channels: Correlated means if a pixel appeared in a specific channel then a specific pixel will appear in another channel at the same position (Depends on each other). The correlation tells you how pairs of pixels (correlated features) might occur or not occur together in the different channels.



Assume $a^{[l]}$ is the activation of layer l, and it is of shape $(n_H^{[l]}, n_W^{[l]}, n_c^{[l]})$. To measure the correlation between each channels of $a^{[l]}$, define style matrix $G^{[l]}$ of shape $(n_c^{[l]}, n_c^{[l]})$:

$$\text{between channel } k, k' \in [1, n_c^{[l]}],\ G_{kk'}^{[l]} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l]} a_{ijk'}^{[l]}$$

$$G^{[l]} = \begin{bmatrix} G_{1,1}^{[l]} & \cdots & G_{1,n_c^{[l]}}^{[l]} \\ \vdots & \ddots & \vdots \\ G_{n_c^{[l]},1}^{[l]} & \cdots & G_{n_c^{[l]},n_c^{[l]}}^{[l]} \end{bmatrix}$$

Feed image S and G into NN, for each layer l, we can get style matrix for style image $G^{[l](S)}$, and for generated image $G^{[l](G)}$. The correlation of style image channels should appear in the generated image channels. Define style cost function $J_{style}(S, G)$:

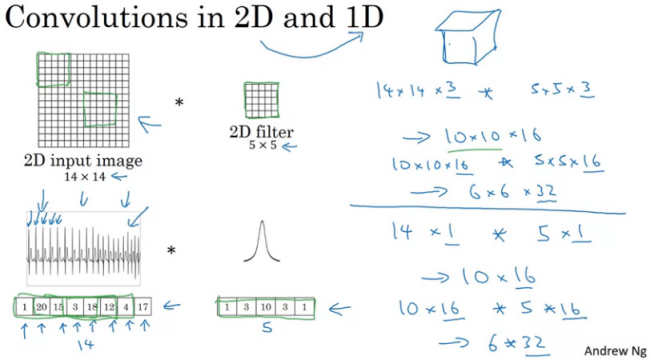
$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_c^{[l]})^2} ||G^{[l](S)} - G^{[l](G)}||^2$$

$$= \frac{1}{(2n_H^{[l]} n_W^{[l]} n_c^{[l]})^2} \sum_{k=1}^{n_c^{[l]}} \sum_{k'=1}^{n_c^{[l]}} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)})^2$$

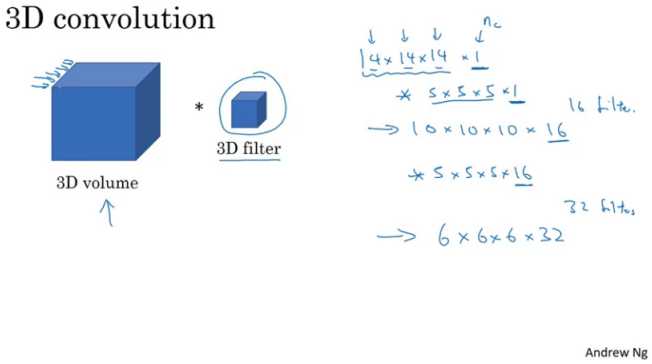
$$J_{style}(S, G) = \sum_{l=1}^{L} \lambda^{[l]} J_{style}^{[l]}(S, G)$$

It turns out you can get better visually result if you use style cost from all the different layers (both earlier ones and deeper ones). Choose the right hyper parameter $\lambda^{[l]}$ to get a better result.

Multiple Dimension Convolution Generalizations:
You can apply convolution to muti dimensional data with the same convolution idea. Choose a filter of the right dimension and slide it through the muti dimensional input data to get the output.



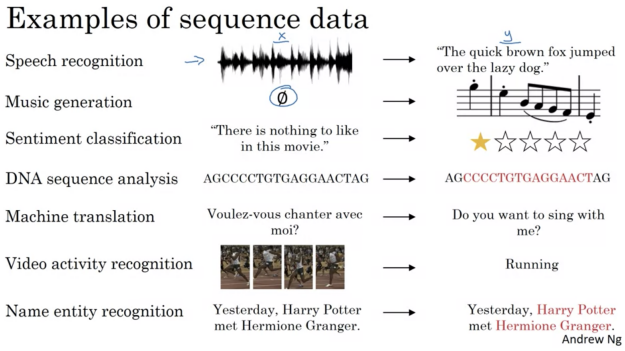
For example, you can use a 1d filter to detect heartbeat from 1d voltage data. Or you can apply a 3d filter to a 3d CT scan to detect tumor.



5 Sequence Models

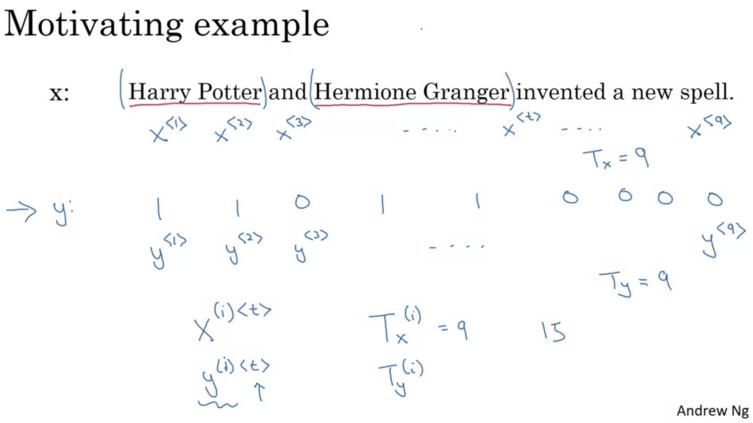
5.1 Recurrent Neural Networks

Models like recurrent neural networks or RNNs have transformed speech recognition, natural language processing and other areas. They are used to process data in sequence format.

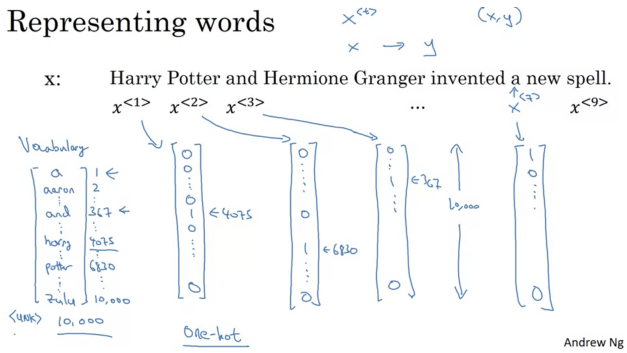


Notation:
Below is a named entity recognition example:

$$x^{(i)}: \text{ a sequence of words}$$
$$y^{(i)}: \text{ a sequence of classes}$$
$$x^{(i)<t>}: \text{ sequence element, a word}$$
$$y^{(i)<t>}: \text{ sequence element class, is name or not}$$
$$T_x^{(i)}: \text{ sequence length of } x^{(i)}$$
$$T_y^{(i)}: \text{ sequence length of } y^{(i)}$$



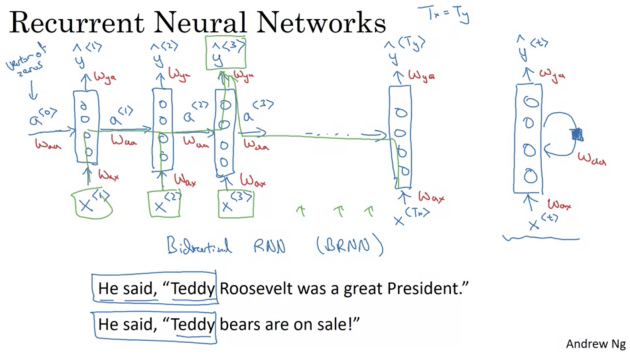
Construct a vocabulary vector to include all the possible words (or only words with most occurrence) in the sequences. We can add an UNK token in the vocabulary which stands for unknown text. Then each $x^{(i)<t>}$ will be an one-hot encoding of vocabulary vector.



Recurrent Neural Network Model:
Normal NN cannot handle sequence data very well:

- Inputs, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of sequence.

In the RNN architecture, the current output $\hat{y}^{<t>}$ depends on the previous activation (memory) and current input. Later we have Bidirectional RNN (BRNN) which uses both information earlier in the sequence as well as information later in the sequence. Below is a Recurrent Neural Network model:



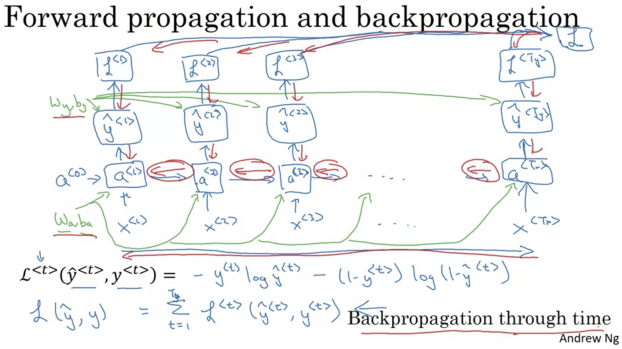
$$a^{<0>} = \vec{0}$$

$$a^{<t>} = g_a(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + B_a)$$
$$\hat{y}^{<t>} = g_y(W_{ya}a^{<t>} + B_y)$$

To simplify this notation:

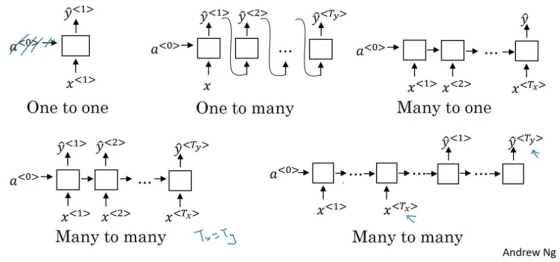
$$W_a = \begin{bmatrix} W_{aa} & W_{ax} \end{bmatrix}$$
$$a^{<t>} = g_a(W_a \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} + B_a)$$
$$\hat{y}^{<t>} = g_y(W_y a^{<t>} + B_y)$$

Backpropagation Through Time:
The backpropagation here is called backpropagation through time because we calculate the gradients for later sequence elements first then previous sequence elements.



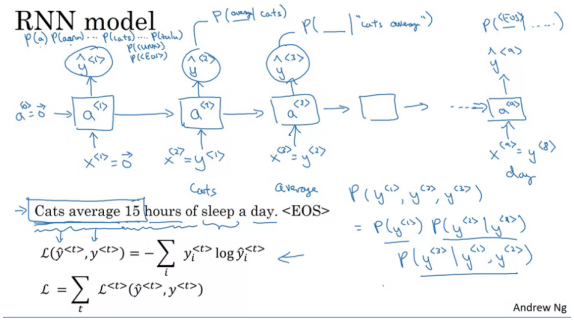
$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>}log(\hat{y}^{<t>}) - (1 - y^{<t>})log(1 - \hat{y}^{<t>})$$
$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

RNN Types:



- one to one.
- one to many: music generation.
- many to one: sentiment classification.
- many to many ($T_x = T_y$).
- many to many ($T_x \neq T_y$): machine translation.

Language Model and Sequence Generation:



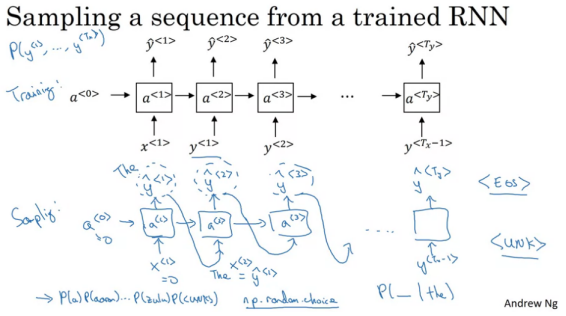
For the text samples, use EOS to represents end of sentence, UNK for unknown words in the vocabulary. Train an RNN model that, given the first a few sequence elements, predicts what's the next sequence element. Each $\hat{y}^{<t>}$ will be a softmax probability distribution of different tokens in the vocabulary based on the given condition (previous sequence elements). Loss function:

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -\sum_{k=1}^{K} y_k^{<t>} log(\hat{y}_k^{<t>})$$

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

Then use the model to sample a sequence:

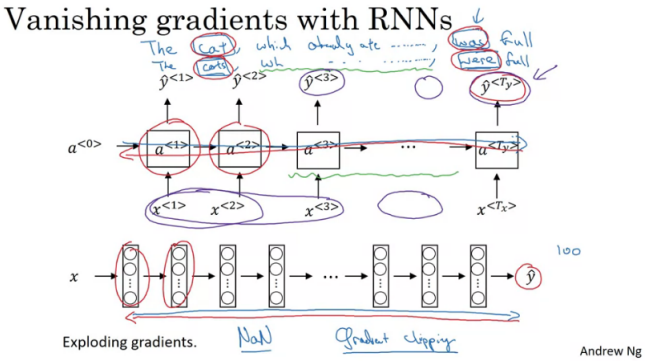
- Use **np.random.choice** to sample a sequence element from $\hat{y}^{<t>}$.
- Feed $\hat{y}^{<t>}$ into model as input in the next step to get $\hat{y}^{<t+1>}$. Use **np.random.choice** to sample a sequence element from $\hat{y}^{<t+1>}$.
- Repeat these 2 steps to get a sequence.



If we build the vocabulary with characters rather than words, we can implement a character-level language model.

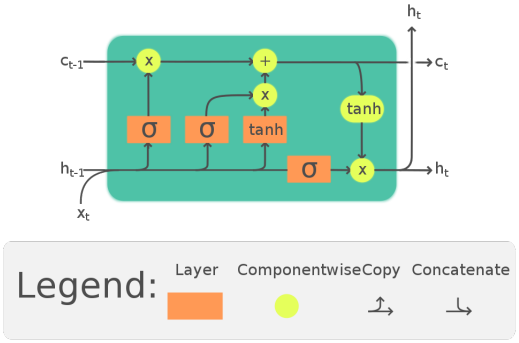
Vanishing Gradients with RNNs:
Similar to a very deep neural network, when doing a backpropagation on an RNN with a lot of steps, vanishing gradients problem can occur. This also causes the difficulty to capture long term dependencies for the RNN model. Because it's hard for the gradients of current step to go all the way back and influence the gradients of previous steps.

For example, it's hard for the model to learn the dependency of "cat" and "was", "cats" and "were", if the sequence in the middle is too long.



Exploding gradients problem can happen as well. We can use gradient clipping to solve it: rescale the gradient if it is larger than some threshold.

Long Short Term Memory (LSTM):



$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot tanh(c_t)$$

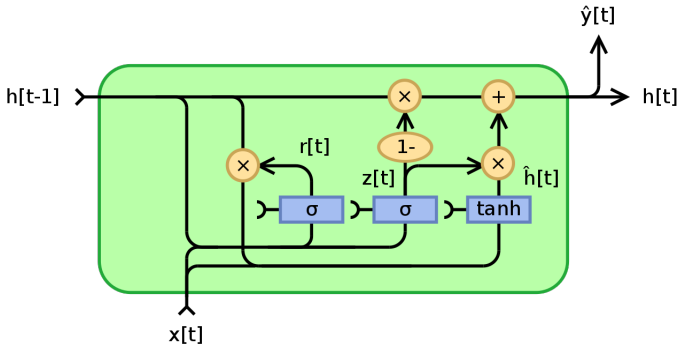
Variables:

- x_t: input vector
- f_t: forget gate's activation vector
- i_t: input gate's activation vector
- o_t: output gate's activation vector
- h_t: output vector
- $\tilde{c}_t$: cell input activation vector
- c_t: cell state vector
- W, U, b: weight matrices and bias vector parameters

A common LSTM unit is composed of a cell state, an input gate, an output gate and a forget gate. The cell state flows through the chain of LSTM units, and stores the memory information. Each LSTM unit has the ability to remove or add information to the cell state, carefully regulated by it's forget gate and input gate. And the output gate controls what parts of the cell state we're gonna put in the output.

LSTM partially solves the vanishing gradient problem, because it allows gradients to flow unchanged if necessary, by making use of gates.

Gated Recurrent Unit (GRU):



$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

Variables:

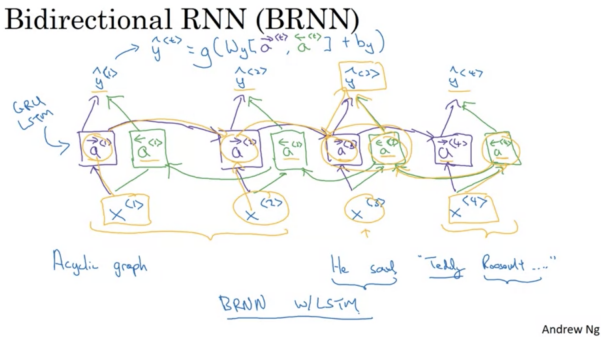
- x_t: input vector
- h_t: output vector
- $\hat{h}_t$: candidate activation vector
- z_t: update gate vector
- r_t: reset gate vector
- W, U, b: weight matrices and bias vector parameters

The GRU has fewer parameters than LSTM. It doesn't have a cell state and it keeps the memory in the output vectors. A reset gate is used to control which part of previous output vector are considered to generate the candidate activation vector. The it uses the update gate to pick the data from candidate activation vector and previous output vector, to generate the output vector.

GRU's performance on certain tasks was found to be similar to that of LSTM. GRUs have been shown to exhibit better performance on certain smaller and less frequent datasets.

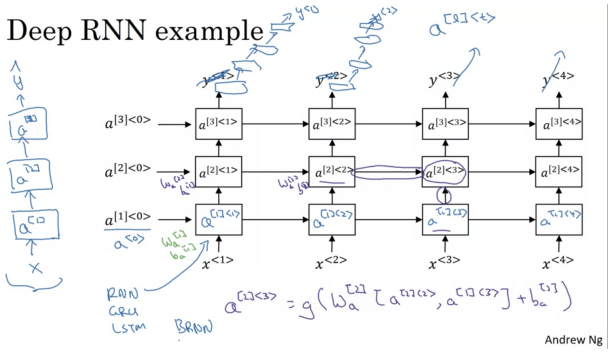
Bidirectional RNN (BRNN):
With normal RNN architecture you can only make prediction with the knowledge of previous sequence elements. In Bidirectional RNN, we add another RNN sequence path in the reverse direction, so we can make prediction with the information from the entire sequence.

$$\hat{y}^{<t>} = g_y(W_{\overrightarrow{y}} \overrightarrow{a}^{<t>} + W_{\overleftarrow{y}} \overleftarrow{a}^{<t>} + B_y)$$



The drawback of BRNN is that it only works when we get the entire sequence. In some problems like real time speech recognition system, it cannot work.

Deep RNN:
Similar to deep neural networks, we can stack RNN sequences to build a Deep RNN model:



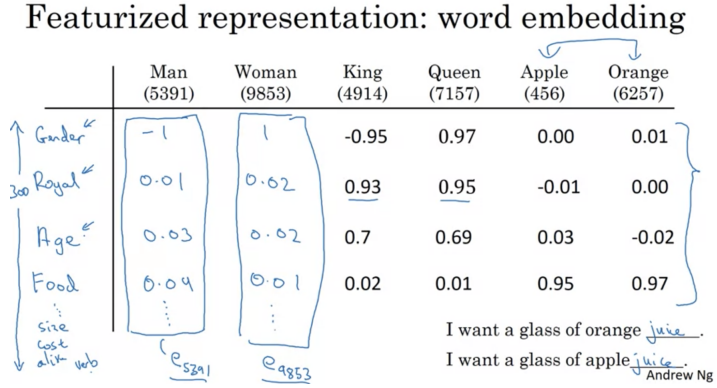
$$a^{[l]<t>} = g_a(W_a^{[l]} a^{[l]<t-1>} + W_a^{[l]} a^{[l-1]<t>} + B_a^{[l]})$$

For the last a few layers, sometimes they are not connected horizontally. RNN usually cannot go too deep since it is very computationally expensive.

5.2 Word Embedding

One hot representation cannot describe the relationship or similarity between different words - they are all independent of each other. For example, a learned model can figure out "I want a glass of orange juice". But it cannot generalize it to figure out "I want a glass of apple juice". Because there is no similarity information included in the one hot encoding of "apple" and "orange".

Featurized Representation: Word Embedding



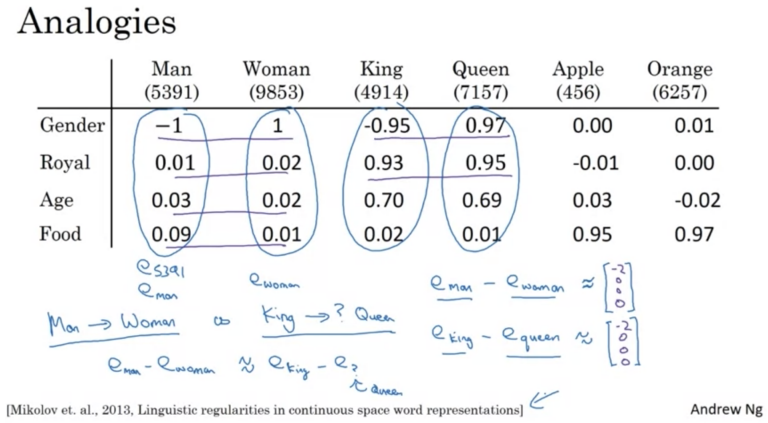
- Featurized Representation can describe the relationship between different words.
- There are ways to learn word embeddings (high dimensional feature vectors).
- Word embeddings can be visualized in 2d using T-SNE algorithm.

Transfer Learning and Word Embeddings:

- Word embeddings can be learned by some RNN model from large text corpus (1 - 100B words). We can download pretrained word embeddings online.
- Transfer word embeddings to new ML tasks with smaller training set (100K words). In this case word embedding is a result of transfer learning.
- Optional. Continue to finetune the word embeddings with new data. Only do this when the training set is large.

Using word embeddings enables RNN models perform well even with small training set. It also reduces the size of the input - from one hot encoding to feature vector.

Properties of Word Embeddings:
One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning. While analogy reasoning may not be by itself the most important NLP application, but it might help convey a sense of what these word embeddings can do.



In the above example, we can find that:

$$e_{man} - e_{woman} \approx e_{king} - e_{queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This vector shows gender is the dominating difference in both pairs.

We can formulate this problem to find the appropriate e_w:

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

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

By choosing the right word to maximize the similarity function:

$$\underset{w}{argmax}\ Sim(e_w, e_{king} - e_{man} + e_{woman})$$

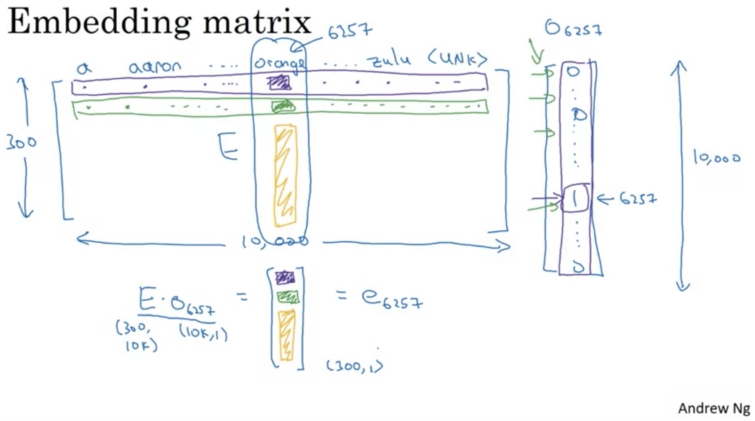
Cosine Similarity - the most commonly used similarity function:

$$Sim(u, v) = \frac{u^T v}{||u||\ ||v||}$$

You can also use Euclidean Distance as a similarity function (but it rather measures a dissimilarity, so you should take it with negative sign).

$$d(u, v) = ||u - v||^2$$

Embedding Matrix:
When you implement an algorithm to learn word embeddings, what you end up learning is an embedding matrix. The following embedding matrix has 10000 words with 300 features.



Assume we have embedding matrix E of shape (300, 10000), one-hot encoding o_{6257} of shape (10000, 1) for word "orange", then the word embedding e_{6257} of shape (300, 1) for word "orange":

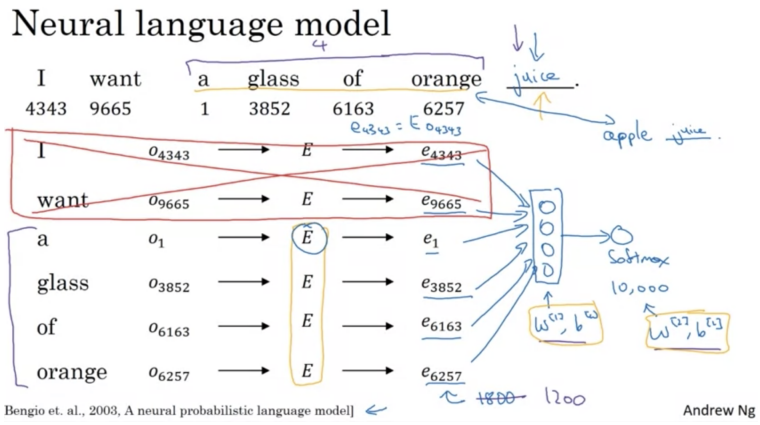
$$e_{6257} = E\ o_{6257}$$

Learning Word Embeddings:
We can build a model to learn word embeddings. We feed one-hot encodings of words in the model, multiplied with embedding matrix to get word embeddings, then feed them into neural network to predict the target word. During backpropagation the embedding matrix will be updated to minimize the cost.

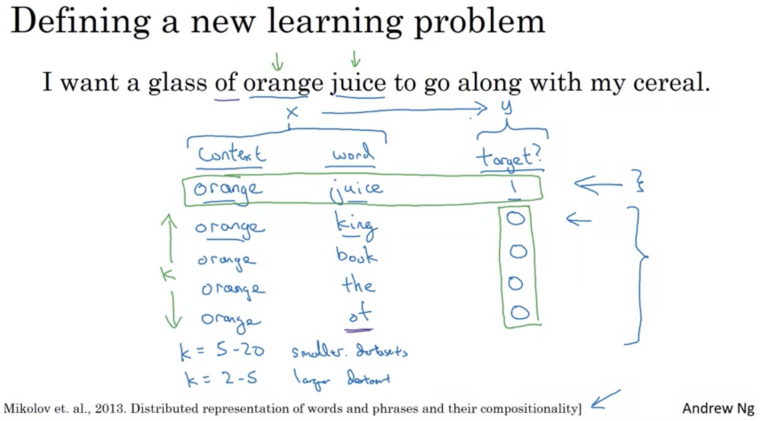
The words we feed into the network that help to make a prediction is called "context". We can pick context as:

- Last 4 words.
- 4 words on the left and on the right.
- Last 1 word.
- Nearby 1 word (skip-gram model).

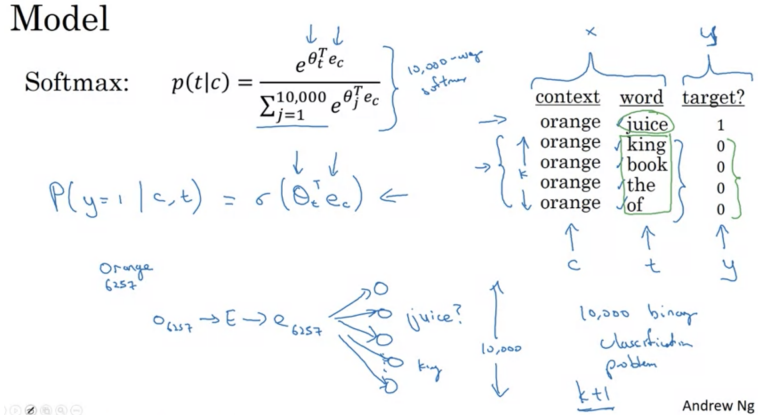
The features learned by the model are sometimes difficult to understand for human. But the embeddings work well in solving NLP problems.



Negative Sampling:
In the last section, a softmax classification of shape (10000, 1) is computationally expensive. Negative sampling can convert this to a binary classification problem.



To prepare the training data, we pick a (context, word) pair as positive sample and the expected output is 1. Then we randomly pick k words from vocabulary to generate the negative samples, whose output value is expected to be 0. Then we can train a model that takes the (context, word) pairs as the input and outputs the target labels.



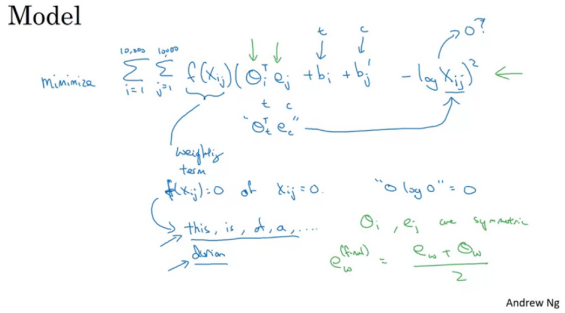
It is recommended that $k \in [5, 20]$ for smaller dataset, $k \in [2, 5]$ for larger dataset.

Assume the observed frequency of a word is $f(w)$, then we can choose the words for negative samples base on this distribution:

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{j=1}^{10000} f(w_j)^{\frac{3}{4}}}$$

This distribution is somewhere in between empirical frequency distribution and uniform distribution.

Global Vectors for Word Representation (GloVe):
GloVe is a simpler model to learn word embeddings. It is getting attentions in recent years because of it's simplicity and efficiency.



Assume x_{ij} is the number of times that word i appears near the context word j (for example in the nearest 10 words). And the size of vocabulary is 10000.

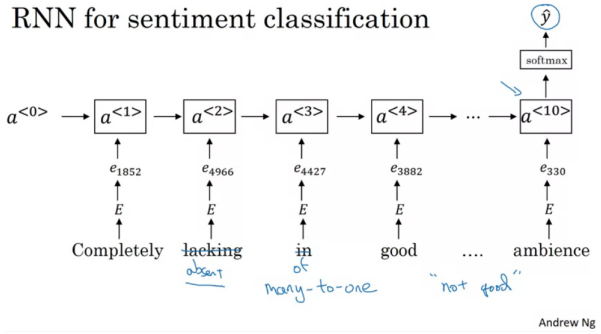
$$J(\Theta, e, B, B') = \sum_{i=1}^{10000} \sum_{j=1}^{10000} f(x_{ij})(\Theta_i^T e_j + B_i + B'_j - log(x_{ij}))^2$$

Intuitively, $\Theta_i^T e_j$ tells your how related word i and word j are. $f(x_{ij})$ is a weighting term used to balance the influence of frequency of words, and also deal with the situation when $x_{ij} = 0$ (it is set to 0 to solve the $log(0)$ problem).

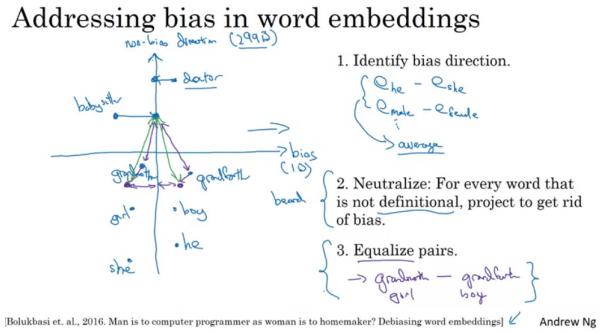
We can find that Θ_i and e_j are symmetric. And they are updated for the same purpose in the backprobagation to minimize the cost function. So we can take $e_w^{(final)}$ as:

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

Sentiment Classification:
Sentiment classification is the automated process of identifying opinions in text and labeling them as positive, negative, or neutral. We can feed the embeddings of the words into RNN to train a model and make a prediction.



Word Embeddings Bias:
Researchers found there is undesired bias existing in some learned word embeddings, like "man is programmer" and "woman is homemaker". We want to make sure that our word embeddings are free of undesirable forms of bias, such as gender bias, ethnicity bias and so on.

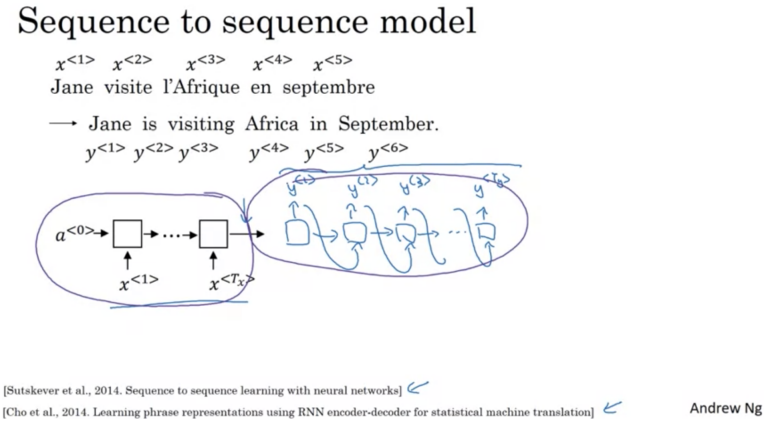


There are several steps we can follow to debias word embeddings:

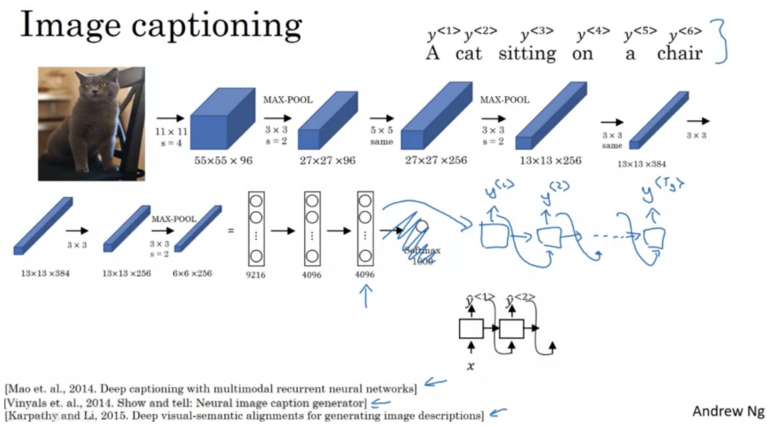
- Identify bias direction: For example to get the bias direction for gender, we can calculate $e_{he} - e_{she}$, $e_{male} - e_{female}$, $e_{man} - e_{woman}$... Then take the average of them.
- Neutralize non-definitional words: For every word that is not definitional, project to non-bias direction axis to get rid of bias. (for words like "programmer" and "homemaker")
- Equalize definitional pairs. For pairs like "grandmother" and "grandfather", we want the only difference in their embedding is the gender. And make sure their embeddings have the same distance to the non-bias direction axis.

5.3 Basic Models

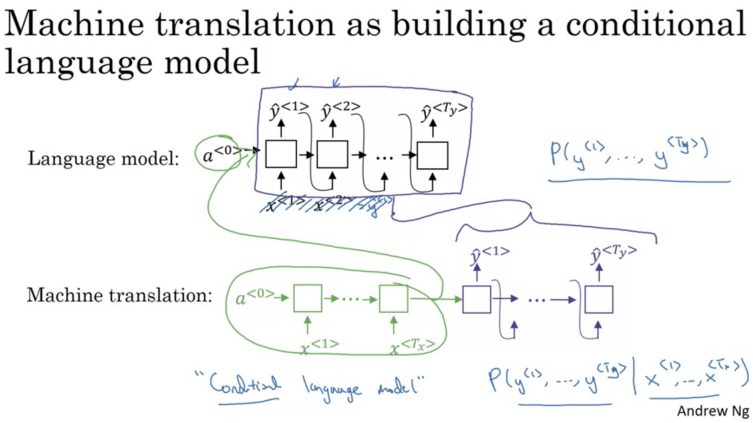
Sequence to Sequence Model:
To translate a sentence from French to English, we can use an RNN encoder to take in the input sequence, then use an RNN decoder to generate the output sequence.



Similarly, image captioning uses the same encode-decode architecture. A trained AlexNet is used to generate the encoding of an image (from the last layer before softmax), then the encoding is fed into an RNN decoder to generate the output sequence.



Picking the Most Likely Sequence:
Machine translation model is considered as a conditional Language model. They both have a decoder to generate output sequence. The difference is that the Machine translation model has an encoder to take in the input sequence, and it's output sequence conditions on the input sequence.



To find the best translation from all the possible translations, we want to find the best output sequence that maximizes the output conditional probability:

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

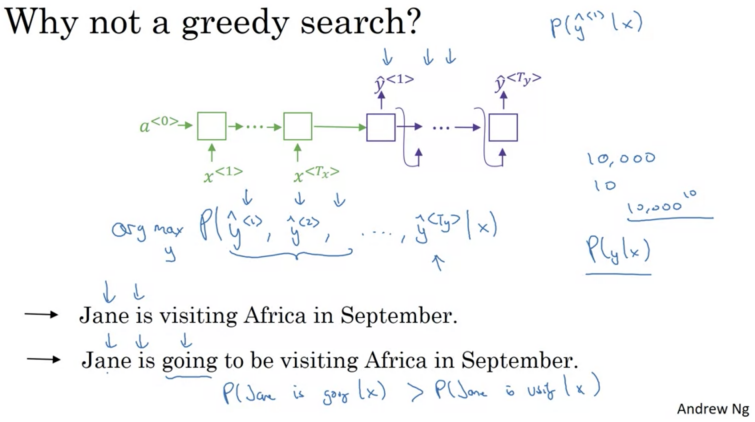
Expand conditional probability for each decoding step in the RNN:

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

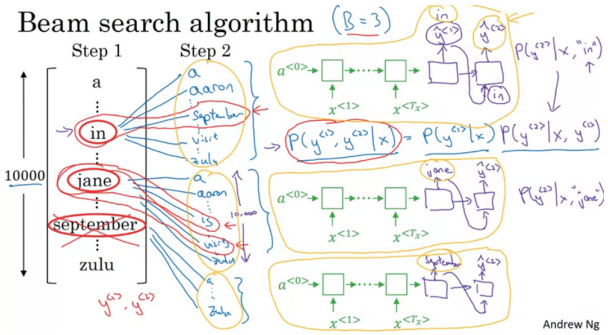
$$\underset{y^{<1>},\ldots,y^{<T_y>}}{argmax} \prod_{t=1}^{T_y} P(y^{<t>}|x,y^{<1>},\ldots,y^{<t-1>})$$

It turns out that the greedy search doesn't work well (always choose the word with highest conditional probability on each step). Because it doesn't always give the best translation with highest possible output conditional probability.

Instead, we need some searching algorithm to give us a better result. Notice that given a large vocabulary, there are almost infinite possible translations. The searching algorithm we need is an approximate algorithm. It will try to pick the best sequence that maximizes the output conditional probability. It usually does a good job, but it is not guaranteed that it always gives the best sequence.



Beam Search:
Instead of only choosing the word with highest conditional probability on each step, Beam search will explore the first B words with highest conditional probability for each step in RNN decoder. So the algorithm will traverse a tree on which each node has B children, and choose the path which results the highest output conditional probability. The example below shows a Beam search algorithm with $B = 3$:



Larger B gives better results but causes longer running time. $B \in [10, 100]$ is often used in production, while $B \in [1000, 3000]$ can be used in research.

Refinements to Beam Search:
The conditional probability on each step is a fraction, most of the time a small fraction. So the output conditional probability is sometimes too small that it causes a numerical overflow in programming languages. To solve this we can use summing logs of conditional probabilities instead.

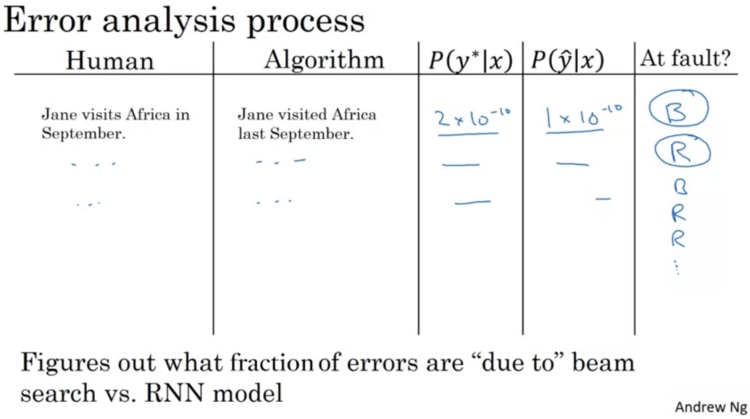
Also, longer sequence causes smaller results in the previous function. To solve this we can normalize the result by multiplying a term $\frac{1}{T_y^\alpha}$. α is a hyperparameter, where $\alpha = 0$ means no sequence length normalization, while $\alpha = 1$ means full sequence length normalization. In practice $\alpha = 0.7$ works well.

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

Error Analysis in Beam Search:
Assume human translation y^* and machine translation $\hat{y}$ are different, and human translation is better.

- $P(y^*|x) > P(\hat{y}|x)$: Beam search choose $\hat{y}$, but y^* has higher output conditional probability. Beam search is at fault.
- $P(y^*|x) \leq P(\hat{y}|x)$: y^* is a better translation than $\hat{y}$. But RNN predicts $P(y^*|x) \leq P(\hat{y}|x)$. RNN is at fault.

Then we can put false translations in the dev set into a table to do error analysis. If Beam search is at fault, try a larger B. If RNN is at fault, try improve network performance.

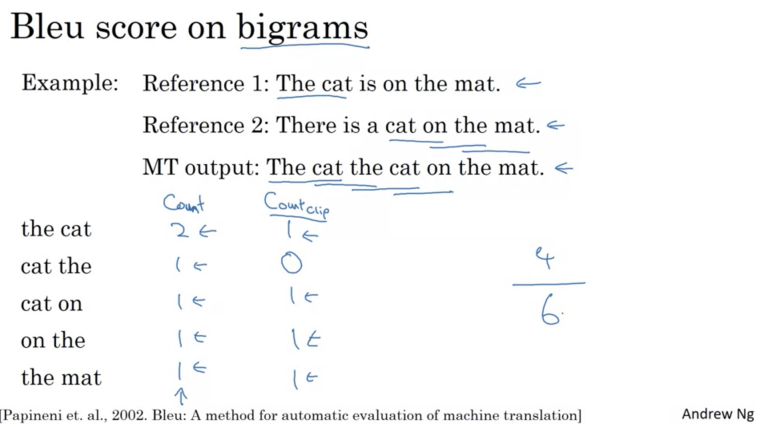


Bleu Score:
Bleu (Bilingual Evaluation Understudy) score is used to deal with situations when there are multiple equally good human translations. We want to give a score to the machine translation to evaluate how similar it is compared to human translations.

Assume we have several human translations and a machine translation. We take the n-grams of the machine translation. Then for each n-gram pair, we count its total appearances in all the human translations as $Count$(n-gram), and its max appearances in one of the human translations as $Count_{clip}$(n-gram). Then the Bleu score on n-gram:

$$p_n = \frac{\sum_{\text{n-gram} \in \text{n-grams of } \hat{y}} Count_{clip}(\text{n-gram})}{\sum_{\text{n-gram} \in \text{n-grams of } \hat{y}} Count(\text{n-gram})}$$

The example below shows the Bleu score on 2-gram:



To get a combined Bleu score, consider $n \in [1, N]$:

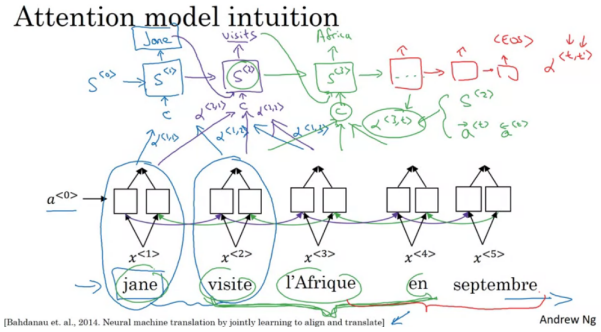
$$P = BP \ exp(\frac{1}{N} \sum_{n=1}^{N} p_n)$$

We found it is easier for short machine translations to get higher Bleu scores. But we don't want all the machine translations to be short. BP means brevity penalty and it is a factor used to penalize the model that only outputs short machine translations. Assume machine translation length mtl, human translation length htl:

$$BP = \begin{cases} 1, & if\ mtl > htl \\ exp(1 - \frac{htl}{mtl}), & if\ mtl \leq htl \end{cases}$$

Attention Model Intuition:
Normal RNN doesn't work well for very long sentence. The attention model looks at one part of the sentence at a time and it can solve this problem.

The following example is a bidirectional RNN. Instead of passing the encoding all at once to the decoder in the normal RNN models, in attention model each encoding step outputs its own encoding. And each decoding step will pick the encodings with attention weight $\alpha^{<t,t'>}$, which implies how much attention the decoding step wants to pay for each encoding.

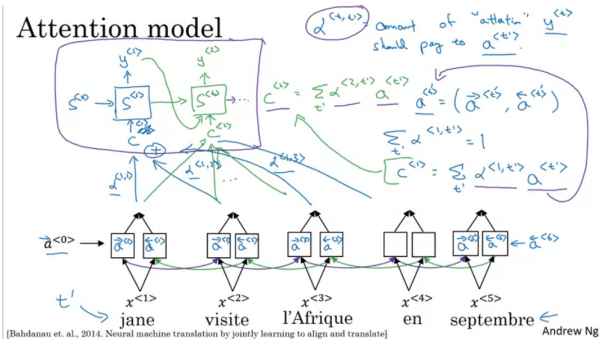


Attention Model:
Assume encoder sequence length T_x, decoder sequence length T_y. The activation for each encoding step is $a^{<t'>}$. The hidden state for each decoding step is $s^{<t>}$. Context $c^{<t>}$ is fed into each decoding step.

$$t' \in [1, T_x], t \in [1, T_y]$$

$$a^{<t'>} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$$

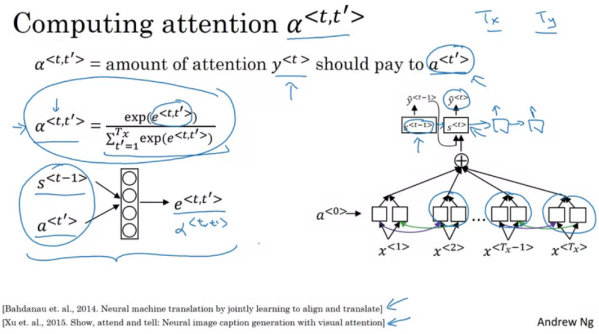
$$c^{<t>} = \sum_{t'=1}^{T_x} \alpha^{<t,t'>} a^{<t'>}$$



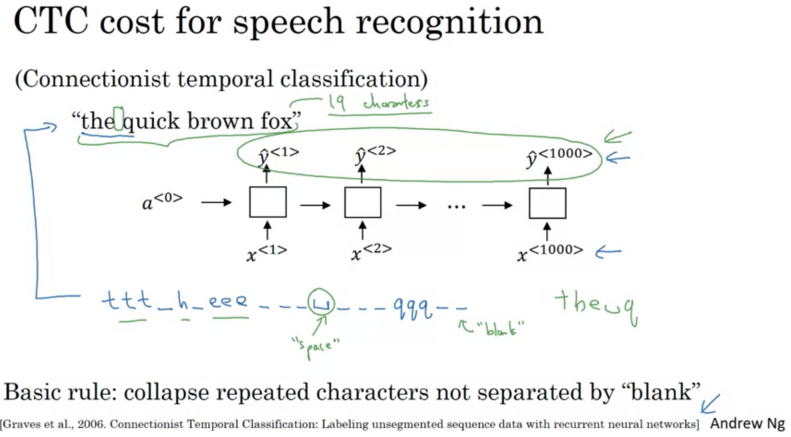
Attention $\alpha^{<t,t'>}$ implies the amount of attention $y^{<t>}$ should pay to $a^{<t'>}$. It is constructed in one-hot encoding so that $\sum_{t'=1}^{T_x} \alpha^{<t,t'>} = 1$.

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

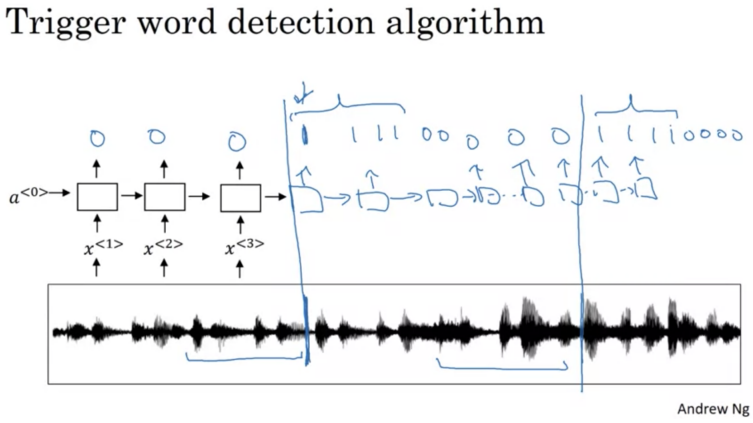
$e^{<t,t'>}$ is computed from an NN with $s^{<t-1>}$ and $a^{<t'>}$ as input. This NN can be learned during the backprobagation process and it is used to generate the attention $\alpha^{<t,t'>}$.



Speech Recognition:
We can use an attention model to translate an audio clip to a sequence of characters. CTC (Connectionist Temporal Classification) is a type of neural network output and associated scoring function. It can be used to generate output sequence in a reasonable format. Basic rule: collapse repeated characters not separated by "blank".



Trigger Word Detection:
For some products, it is important to detect trigger words (e.g. hi siri). We can use 0 for non trigger word, 1 for trigger word to generate the training data. One problem is that for the most of time, there are far more non trigger words than trigger words in the audio samples. And that will cause imbalanced training set. One hack is that we set several steps right after the trigger word to 1, instead of setting one single trigger word to 1. This will make the training set more balanced.



5.4 Transformer Network

Transformer Network Intuition:
Normal RNN architecture has some problems:

- LSTM/GRU use gates to control the flow of information, but they bring extra complexity.
- It is a sequential model. Each unit becomes a bottleneck of flow of information.

Transformer Network can run the computation of the entire sequence in parallel. So the sequence can be fed into the network all at once rather than being processed item by item. It is motivated by the attention based representations and the CNN style of parallel processing.

Self Attention:
For each element $x^{<t>}$ in the input sequence, we have query $q^{<t>}$, key $k^{<t>}$ and value $v^{<t>}$. W^Q, W^K and W^V are learned weights used to generate the right Q, K and V.

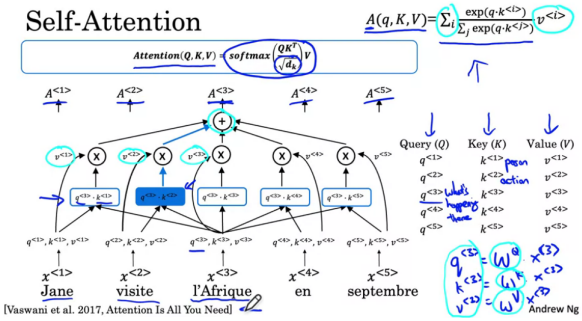
$$q^{<t>} = W^Q x^{<t>}$$
$$k^{<t>} = W^K x^{<t>}$$
$$v^{<t>} = W^V x^{<t>}$$

The concept of "query", "key" and "value" comes from database. "query" implies the question and "key" implies the answer. The product of "query" and "key" implies the relevance between them. "value" implies the representation of the word.

$$A^{<t>} = A(q^{<t>}, K, V) = \sum_{i=1}^{T_x} \frac{exp(q^{<t>} k^{<i>})}{\sum_{j=1}^{T_x} exp(q^{<t>} k^{<j>})} v^{<i>}$$

The following is the original vectorized formula. d_k is the dimension of the key vector. $\sqrt{d_k}$ is used to scale the dot-product so that it doesn't explode.

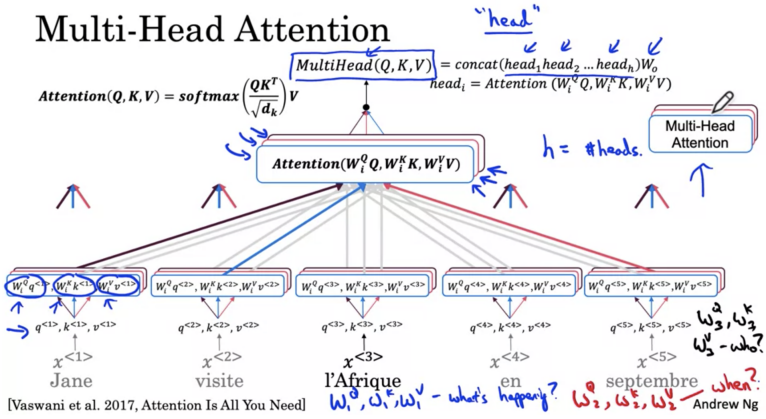
$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



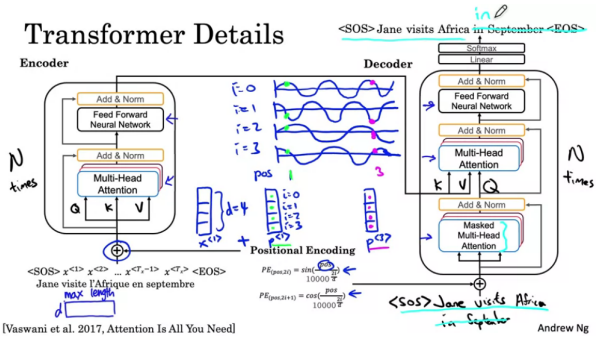
Multi-Head Attention:
Multi-head Attention mechanism runs through self attention mechanism several times in parallel. The independent attention outputs are then concatenated and linearly transformed into the expected dimension. Intuitively, Multi-head Attention allows the model to jointly attend to information from different representation subspaces at different positions.

$$head_i = Attention(W_i^Q Q, W_i^K K, W_i^V V)$$
$$MultiHead(Q, K, V) = concat(head_1, head_2, \ldots, head_h) W^O$$

Assume there are h heads in the Multi-head Attention mechanism. Each of W_i^Q, W_i^K, W_i^V, W^O is a parameter to be learned.



Transformer Network:



Transformer Network has an encoder block and a decoder block. d_{max_length} is the maximum length of sequence the model can take, and the output of (Maksed) Multi-Head Attention layer and feed forward NN are of this shape.

The encoder block contains a Multi-Head Attention layer and a feed forward NN. This encoder block is repeated for N times before fed into the decoder block. (repeat means output of encoder/decoder is fed as input into another encoder/decoder block)

The output of the encoder block is fed into the decoder block to generate "key" and "value". The decoder block has a Masked Multi-Head Attention layer, a Multi-Head Attention layer and a feed forward NN. The Masked Multi-Head Attention layer is used to generate the "query" for the Multi-Head Attention layer. The decoder block is repeated for for N times before making a prediction.

The Positional Encoding of word is used to add unique positional information to each input sequence element. Assume the position of sequence element in the sequence is $pos \in [1, T_x]$, the dimension of word embedding is d, and feature index $i \in [0, d-1]$.

$$PE(pos,i) = \begin{cases} sin(\frac{pos}{10000^{\frac{i}{d}}}), & \text{if i is even} \\ cos(\frac{pos}{10000^{\frac{i-1}{d}}}), & \text{if i is odd} \end{cases}$$

$$p^{<pos>} = \begin{bmatrix} PE(pos,0) \\ PE(pos,1) \\ \vdots \\ PE(pos,d-1) \end{bmatrix}$$

$$x^{<pos>} = x^{<pos>} + p^{<pos>}$$

The positional encodings are also passed to (Maksed) Multi-Head Attention layers and feed forward NNs using residual connections, so that the positional information is passed along the entire architecture.

Add & Norm (similar to Batch Norm) layers are used throughout the architecture to speed up learning process. The normalization used here is layer norm: (with H hidden units in the layer)

$$\mu^{[l]} = \frac{1}{H}\sum_{i=1}^{H} a_i^{[l]}$$

$$\sigma^{[l]} = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^{[l]} - \mu^{[l]})^2}$$

$$Z_{norm}^{[l](i)} = (Z^{[l](i)} - \mu^{[l]}) \oslash \sqrt{\sigma^{[l]2} + \epsilon}$$

$$\overset{N}{Z}^{[l](i)} = \gamma^{[l]} \odot Z_{norm}^{[l](i)} + \beta^{[l]}$$

When decoding, an output sequence element should only depends on the previous sequence elements (not the future ones). Masked Multi-Head Attention is used to mask the sequence elements appearing in the future, so that the model cannot make use of them to make a prediction.

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

Where M is a matrix of 0s and $-\infty$s (0 for previous sequence elements, $-\infty$ for future sequence elements). Then the attention for future sequence elements will be set to 0.

A linear layer and a softmax layer are used after the decoder block to make the prediction.