Table of Contents

1. Intro **5**

2. Linear Regression with One Variable **5**

2.1. Notation 5

2.2. Cost Function 5

2.3. Gradient Descent 6

2.3.1. Derivatives 6

3. Linear Regression with Multiple Variables **7**

3.1. Gradient Descent 8

3.1.1. Feature Scaling/Normalization 8

3.1.2. Debugging Gradient Descent 8

3.1.3. Feature Engineering 9

3.2. Polynomial Regression 9

3.3. Normal Equation 9

4. Classification - Logistic Regression **10**

4.1. Linear Decision Boundary Model 10

4.2. Non-linear Decision Boundary Model 10

4.3. Cost Function 10

4.4. Loss Function 11

4.5. Simplified Loss Function 11

4.6. Gradient Descent 12

5. Regularization **12**

5.1. Cost Function with Regularization for Linear Regression 12

5.2. Gradient Descent of Linear Regression with Regularization 13

5.3. Gradient Descent of Logistic Regression with Regularization 13

6. Tips on Model Building **13**

6.1. Model Selection 14

6.2. Bias and Variance 14

6.2.1. Polynomial and Bias/Variance 14

6.2.2. Regularization and Bias/Variance 15

6.2.3. Neural Network and Bias/Variance 15

6.3. Error Metrics For Skewed Datasets 15

7. Neural Networks **15**

7.1. Vectorization 16

7.2. Activation Functions 17

7.3. Neural Network Computation Graph 17

7.4. Getting Your Matrix Dimensions Right 18

7.5. Building Blocks of Deep Neural Networks 18

7.5.1. Why Deep Representations 18

7.5.2. Deep Neural Network Implementation 18

7.5.3. Parameters vs Hyperparameters 19
7.6. Improving Deep Neural Networks 19
7.6.1. Train / Dev / Test sets 19
7.6.2. Bias and Variance 19
7.6.3. Regularizing Neural Network 20
7.6.4. Dropout Regularization 21
7.6.5. Setting up Optimization Problem 22
7.6.6. Optimization 23
7.7. Hyperparameter Tuning 25
7.7.1. Batch Normalization 25
7.7.2. Hyperparameter Tuning Strategy 26
7.8. Multi-task Learning 27
7.9. End-to-end Deep Learning 28
7.10. TensorFlow 28
7.10.1. Round-off errors 28

8. Convolutional Neural Networks (CNN) 29
8.1. Foundations of Convolutional Neural Networks 29
8.1.1. Edge Detection Example 29
8.1.2. Padding 29
8.1.3. Strided Convolutions 30
8.1.4. Convolutions Over Volume 30
8.1.5. One Layer of a Convolutional Network 30
8.1.6. Why Convolutions 31
8.2. Classic Networks 31
8.2.1. Classic Networks 32
8.2.2. ResNets 33
8.2.3. Networks in Networks and 1x1 Convolutions 34
8.2.4. Inception Network 34
8.2.5. Practical advices for using ConvNets 35
8.3. Object Detection 37
8.3.1. Detection Algorithms 37
8.3.2. Landmark and Object Detection 38
8.3.3. Convolutional Implementation of Sliding Windows 38
8.3.4. YOLO algorithm 38

9. Sequence Models 39
9.1. Recurrent Neural Networks 39
9.1.1. Notation 39
9.1.2. Recurrent Neural Network Model 40
9.1.3. Different Types of RNNs 41
9.1.4. Language Model and Sequence Generation 41
9.1.5. Vanishing Gradients with RNNs 42
9.1.6. Bidirectional RNN 45
9.1.7. Deep RNNs 46

9.2. NLP and Word Embeddings 46
9.2.1. Word Embeddings 46
9.2.2. Using word embeddings 47
9.2.3. Properties of word embeddings 47
9.2.4. Learning Word Embeddings: Word2vec & GloVe 47
9.2.5. Applications using Word Embeddings 50
9.3. Sequence Models & Attention Mechanism 52
9.3.1. Basic Sequence-to-Sequence Models 52
9.3.2. Image Captioning 54
9.3.3. Bleu Score 54
9.3.4. Attention Model 55
9.3.5. Speech recognition - Audio data 58
9.4. Transformers 59

10. Decision Tree Model 61
10.1. Decisions 61
10.2. Decision Tree Learning Process 62
10.3. Regression Tree 62
10.4. Decision Trees vs Neural Networks 63

11. Unsupervised Learning 63
11.1. Clustering 63
11.1.1. K-means Algorithm 63
11.1.2. Cost Function 64
11.1.3. Choosing number of clusters 64
11.2. Anomaly Detection 64
11.2.1. Anomaly Detection Algorithm with Gaussian (Normal) Distribution 64
11.2.2. Anomaly Detection vs. Supervised Learning 65
11.2.3. Error Analysis for Anomaly Detection 65
11.3. Recommender Systems 65
11.3.1. Cost Function 65
11.3.2. Cost Function With Unknown Features 66
11.3.3. Collaborative Filtering 66
11.3.4. Gradient Descent 66
11.3.5. TensorFlow Implementation of Collaborative Filtering 67
11.3.6. Limitations of Collaborative Filtering 68
11.3.7. Content-based Filtering Neural Network 68

12. Reinforcement Learning 68
12.1. Key Factors 68
12.2. Bellman Equation 69
12.3. Markov Decision Process (MDP) 70
12.4. Algorithm 70
12.4.1. Deep Reinforcement Learning Architecture 70
12.4.2. Greedy policy 70

12.5. Limitations of Reinforcement Learning 71

13. Machine Learning Project Lifecycle 71

13.1. Scoping 71

13.1.1. Scoping process 71

13.1.2. Ethical consideration 71

13.1.3. Milestones & Resourcing 71

13.1.4. Setting Up Your Goal 71

13.2. Data 72

13.2.1. Define Data 72

13.2.2. Train/Dev/Test Distributions 72

13.2.3. Mismatched Training and Dev/Test Set 73

13.2.4. Establish Baseline 74

13.2.5. Comparing to Human-level Performance 74

13.2.6. Label and Organize Data 76

13.2.7. Meta-data, Data Provenance and Lineage 76

13.2.8. Data Augmentation 76

13.2.9. From Big Data to Good Data 76

13.3. Modeling 76

13.3.1. Iterative Loop of ML Development 77

13.3.2. Key Challenges 77

13.3.3. Tips For Getting Started on ML Project 77

13.3.4. Building First System 77

13.3.5. Error Analysis 77

13.3.6. Auditing Framework 78

13.3.7. Experiment Tracking 78

13.4. Deployment 78

13.4.1. Key Challenges 79

13.4.2. Key Ideas 79

13.4.3. Common Deployment Patterns 79

13.4.4. Monitoring 79

1. Intro

Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed. A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.
In general, any machine learning problem can be assigned to one of two broad classifications:

- **Supervised learning** - we are given a dataset and already know what our correct output should look like, having the idea that there is a relationship between the input and the output. Supervised learning problems are categorized into "regression" and "classification" problems:
    - In a **regression** problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function.
    - In a **classification** problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.
- **Unsupervised learning** - allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning, there is no feedback based on the prediction results.

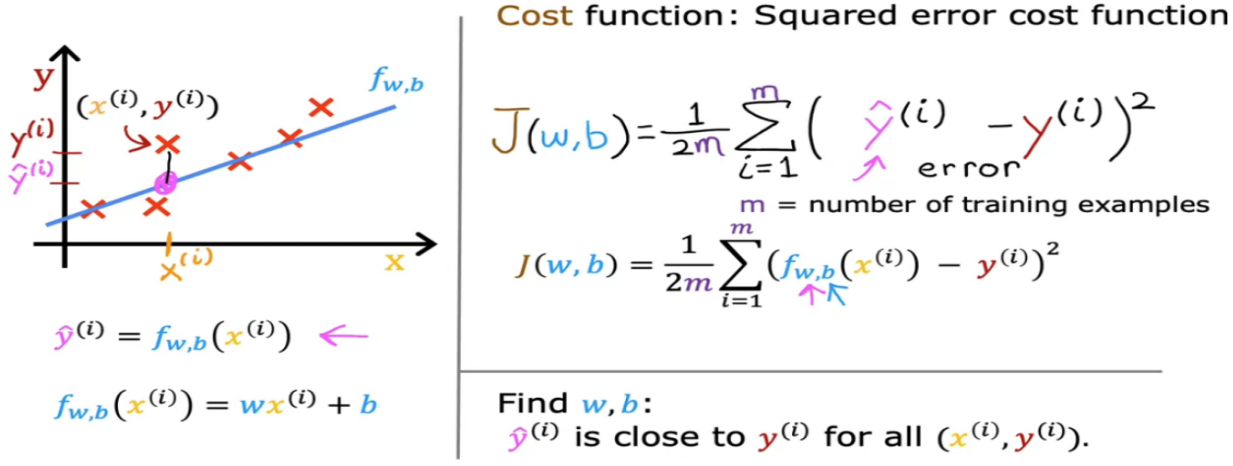
2. Linear Regression with One Variable

2.1. Notation

x = to denote the "input" variables or input features,
y = to denote the "output" or target variable
m = number of training examples
$(x^{(i)}, y^{(i)})$ ith training example
$\hat{y}$ is the estimate or the prediction for y.
$f(x) = wx + b$ - univariate linear regression model
w = weight
b = bias
w, b sometimes referred to as coefficients, weights

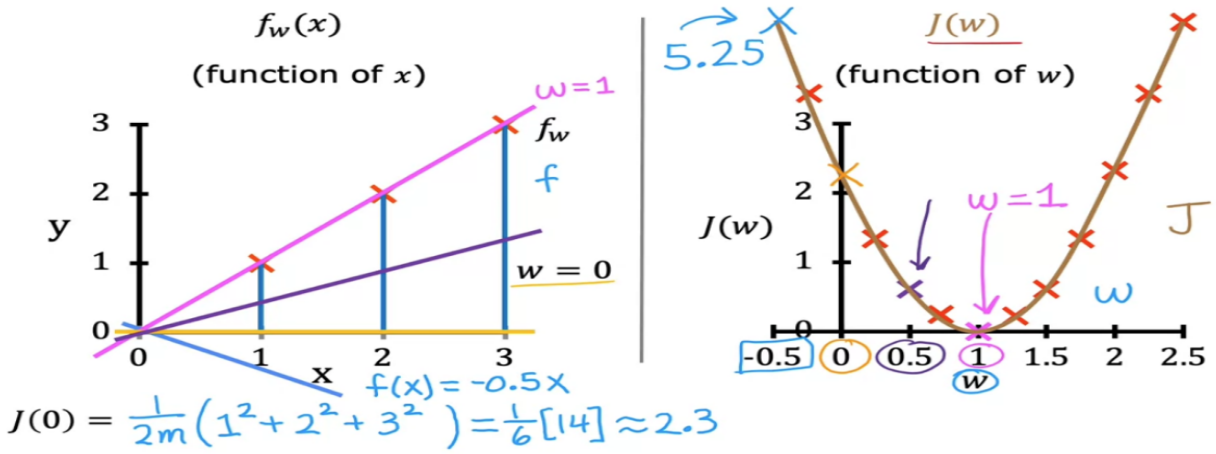
2.2. Cost Function

We can measure the accuracy of our hypothesis function by using a **Cost Function** (Squared error function or Mean squared error). Squared error measures the difference between the model's predictions and the actual true values for y.



2.3. Gradient Descent

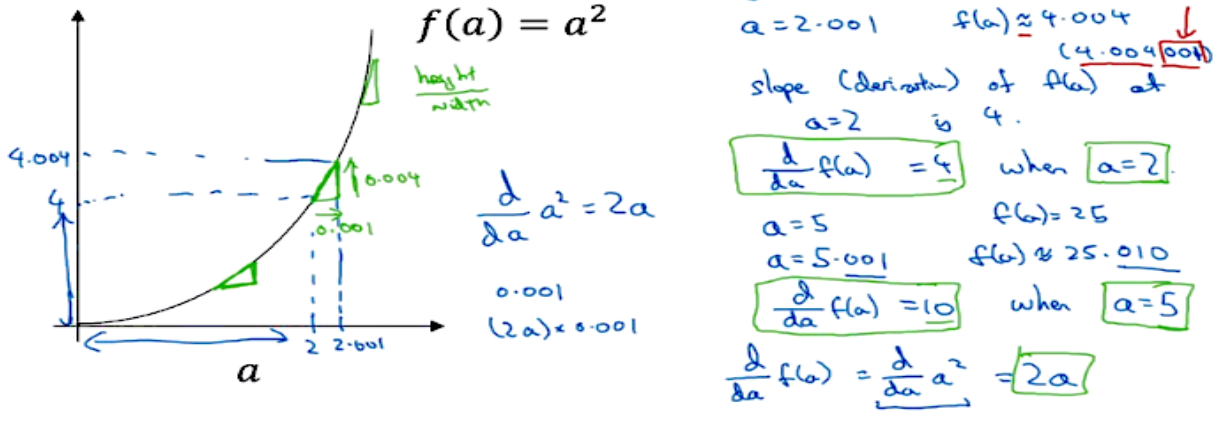
Gradient descent will minimize the cost function J of w, b. It is also used in minimizing cost in the neural networks however its cost is not a parabola shaped like square errors. Hence it can have multiple minima and maxima in a neural network. Squared error cost can only have one minimum and therefore it's called convex function.



2.3.1. Derivatives

Intuition About Derivatives

- The formal definition of derivative - how much does f(a) go up whenever you nudge a to the right by an infinitesimal (infinitely small) amount
- Derivative is same if the function is straight line
- Derivative can be different at various points in the function because of the ratio of height and width
- The function below has derivative 2a (from calculus textbooks)



Gradient Descent Algorithm (repeat until convergence)

$w = w - \alpha\frac{\partial}{\partial w}J(w,b)$

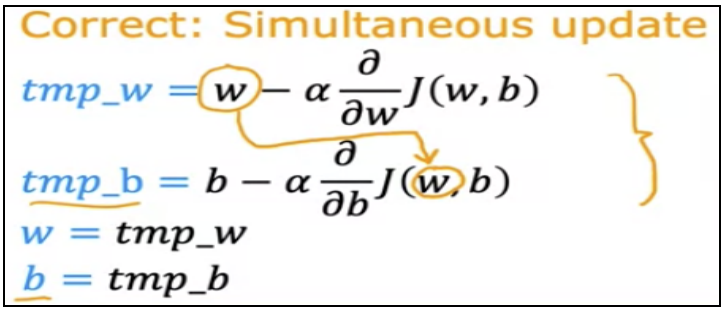
$b = b - \alpha\frac{\partial}{\partial b}J(w,b)$

Notation

α = learning rate

j represents the feature index number.

$\frac{\partial}{\partial \theta_1}J(\theta_0, \theta_1)$ derivative of J



At each iteration j should simultaneously update the parameters $\theta_1, \theta_2, \theta_3, \theta_4 \ldots, \theta_n$ Updating a specific parameter prior to calculating another one on the $j(th)$ iteration would yield a wrong implementation. When the slope of the random point in J(w,b) is negative, the value of w increases, and when it is positive the value decreases.
On a side note, we should adjust our parameter α to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value implies that our step size is wrong. The intuition behind the convergence is that $\frac{d}{d\theta_1} J(\theta_1)$ approaches 0 as we approach the bottom of our convex function. At the minimum, the derivative will always be 0 and thus we get: $\theta_1 = \theta_1 - \alpha * 0$

Gradient Descent Algorithm of Univariate Linear Regression

$$w = w - \alpha \frac{\partial}{\partial w} J(w,b) \qquad b = b - \alpha \frac{\partial}{\partial b} J(w,b)$$

$$\frac{\partial}{\partial w} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) x^{(i)} \qquad \frac{\partial}{\partial b} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

$$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) x^{(i)} \qquad b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

3. Linear Regression with Multiple Variables

Linear regression with multiple variables is also known as "multivariate linear regression".

Size in feet²	Number of bedrooms	Number of floors	Age of home in years	Price ($) in $1000's
x_1	x_2	x_3	x_4	
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

$j = 1 \ldots 4$, $n = 4$, $i = 2$

x_j = j^{th} feature
n = number of features
$\vec{x}^{(i)}$ = features of i^{th} training example
$x_j^{(i)}$ = value of feature j in i^{th} training example

$\vec{x}^{(2)} = [1416 \ 3 \ 2 \ 40]$, $x_3^{(2)} = 2$

The multivariable form of the hypothesis function accommodating these multiple features are as follows:
$f_{w,b}(x) = w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_n x_n + b$
Vectorization will both make your code shorter and also make it run much more efficiently.
f = np.dot(w,x) + b

3.1. Gradient Descent

One feature

repeat {

$$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) x^{(i)}$$

$\frac{\partial}{\partial w} J(w,b)$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

simultaneously update w, b

}

n features ($n \geq 2$)

repeat {

$j = 1$: $w_1 = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}) x_1^{(i)}$

$\frac{\partial}{\partial w_1} J(\vec{w},b)$

$\vdots$

$j = n$: $w_n = w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}) x_n^{(i)}$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})$$

simultaneously update w_j (for $j = 1, \cdots, n$) and b

}

3.1.1. Feature Scaling/Normalization

Techniques to find gradient descent faster. Rule of thumb ideal range $-1 \leq x_1 \leq 1$ but max $-3 \leq x_1 \leq 3$. With normalization, cost function will be more round and easier to optimize when features are all on similar scales.

- **Feature scaling** involves dividing the input values by max of the input variable, resulting $-1 < x_1 \leq 1$
  - E.g min=0,max=2000 $x_1 = \frac{size}{2000}$
- **Mean normalization** involves subtracting the average value for an input variable and dividing it by the range of input variables (i.e. the maximum value minus the minimum value). All features are centered around 0, can be negative and positive values $-1 \leq x_1 \leq 1$
  - E.g min=0,max=2000 $x_1 = \frac{size-\mu}{2000-0}$
- **Z-score normalization** just like mean normalization just divides by standard deviation not range.
  - E.g min=0,max=2000 $x_1 = \frac{size-\mu}{\sigma_1}$

3.1.2. Debugging Gradient Descent

Learning curve chart

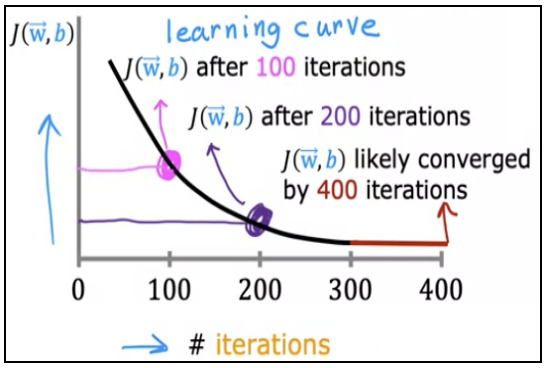
Make a plot with a number of iterations on the x-axis and J(θ)/J(w,b) on the y-axis. J(w,b) should decrease after every iteration. The number of iterations to reach convergence varies. If the learning rate α is sufficiently small, then J(θ) will decrease on every iteration.

Values of α to try:

0.001 -> 0.01 -> 0.1 -> 1

Then multiply by 3

0.003 -> 0.03 -> 0.3 -> 3



Automatic convergence test

Declare convergence if J(θ) decreases by less than ε in one iteration, where ε is some small value such as 10^{-3}. However, in practice, it's difficult to choose this threshold value.

To summarize:

If α is too small: slow convergence.

If α is too large: may not decrease on every iteration and thus may not converge.

3.1.3. Feature Engineering

Feature engineering is using intuition to design new features, by transforming or combining original features. We can combine multiple features into one. E.g.: Let's assume we have Frontage (x_1) and Depth (x_2) of the house. By combining x_1 and x_2 we can create a new variable Area x_3 and improve our model.

3.2. Polynomial Regression

Polynomial regression will let you fit curves, non-linear functions, to your data. We can change the behavior of the curve of our hypothesis function by making it a quadratic, cubic, square root function, or any other form.

For example, if our hypothesis function is $f_{w,b}(x) = w_1 x + b$ then we can create additional features based on x_1 to get

The quadratic function - $f_{w,b}(x) = w_1 x + w_2 x_1^2 + b$

The cubic function - $f_{w,b}(x) = w_1 x + w_2 x_1^2 + w_3 x_1^3 + b$

The square root function - $f_{w,b}(x) = w_1 x + w_2 \sqrt{x} + b$

3.3. Normal Equation

Normal Equation is an alternative to gradient descent. The minimization explicitly and without resorting to an iterative algorithm. In the normal equation method, we will minimize J by explicitly taking its derivatives with respect to the θj ’s, and setting them to zero. This allows us to find the optimum theta without iteration. There is no need to do feature scaling with the normal equation. The normal equation formula is $\theta = (X^T X)^{-1} X^T y$

Gradient Descent	Normal Equation
Need to choose alpha	No need to choose alpha
Needs many iterations	No need to iterate
$O(kn^2)$ complexity	$O(n^3)$ need to calculate inverse of $X^T X$
Works well when n is large	Slow if n is very large

4. Classification - Logistic Regression

Binary classification is a classification problem where there are only two possible outputs. Usually labeled as False or True, 0 or 1. Category is also known as class e.g negative class or positive class. Logistic regression fits a S shaped curve to the dataset. The output predicts the probability of positive class.

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

$$f_{w,b}(x) = g(wx + b) = \frac{1}{1 + e^{-(wx+b)}}$$

The decision boundary is the line where you're just almost neutral about whether y is 0 or y is 1.

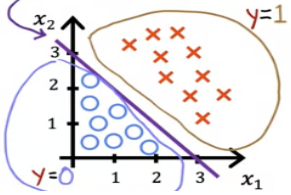
4.1. Linear Decision Boundary Model

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(\underset{1}{w_1}x_1 + \underset{1}{w_2}x_2 + \underset{-3}{b})$$

Decision boundary $z = \vec{w} \cdot \vec{x} + b = 0$

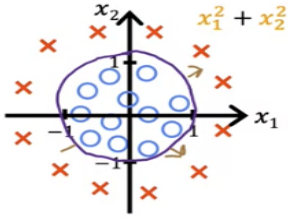
$z = x_1 + x_2 - 3 = 0$

$x_1 + x_2 = 3$



$$f_{w,b}(x) = g(w_1x_1 + w_2x_2 + b) = \frac{1}{1 + e^{-(w_1x_1+w_2x_2+b)}}$$

4.2. Non-linear Decision Boundary Model



$x_1^2 + x_2^2 \geq 1$

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(\overbrace{\underset{1}{w_1}x_1^2 + \underset{1}{w_2}x_2^2 + \underset{-1}{b}}^{z})$$

decision boundary $z = x_1^2 + x_2^2 - 1 = 0$

$x_1^2 + x_2^2 = 1$

$$f_{w,b}(x) = g(w_1x_1^2 + w_2x_2^2 + b) = \frac{1}{1 + e^{-(w_1x_1^2+w_2x_2^2+b)}}$$

4.3. Cost Function

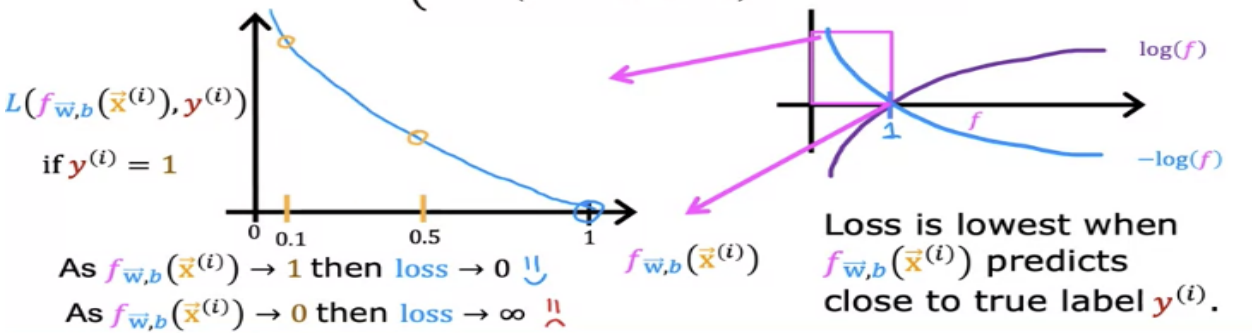
Logistic regression has non-convex cost and squared error cost will not work. Therefore we need a loss function.

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}[(L(f_{w,b}(x^{(i)}), y^{(i)})]$$

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(f_{w,b}(x^{(i)})) + (1-y^{(i)})log(1-f_{w,b}(x^{(i)}))]$$ **Explained below**

4.4. Loss Function

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$



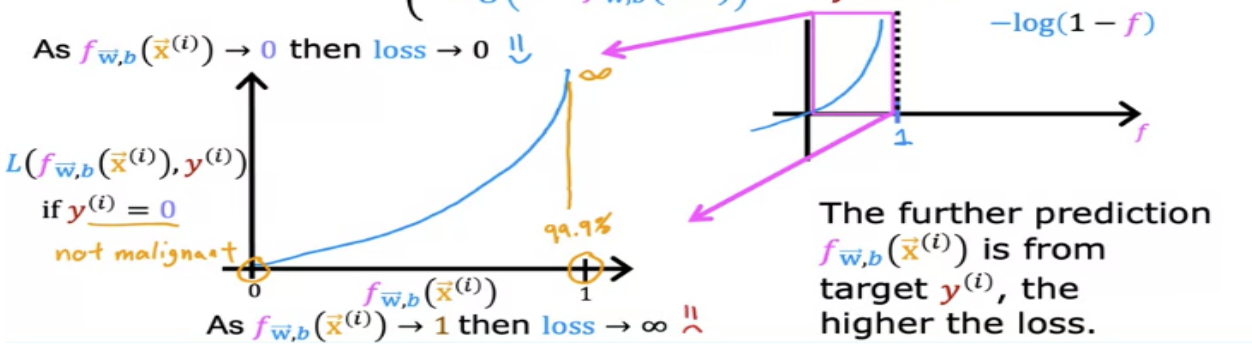
As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 1$ then loss $\to 0$

As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 0$ then loss $\to \infty$

Loss is lowest when $f_{\vec{w},b}(\vec{x}^{(i)})$ predicts close to true label $y^{(i)}$.

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$

As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 0$ then loss $\to 0$



As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 1$ then loss $\to \infty$

The further prediction $f_{\vec{w},b}(\vec{x}^{(i)})$ is from target $y^{(i)}$, the higher the loss.

4.5. Simplified Loss Function

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -\underbrace{y^{(i)}}_{0}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) - \underbrace{(1-y^{(i)})}_{(1-0)}\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)$$

if $y^{(i)} = 1$:

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -1\ \log\left(f(\vec{x})\right)$$

if $y^{(i)} = 0$:

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -(1-0)\log(1-f(\vec{x}))$$

4.6. Gradient Descent

repeat {

looks like linear regression!

$$w_j = w_j - \alpha \left[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})x_j^{(i)}\right]$$

$$b = b - \alpha \left[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})\right]$$

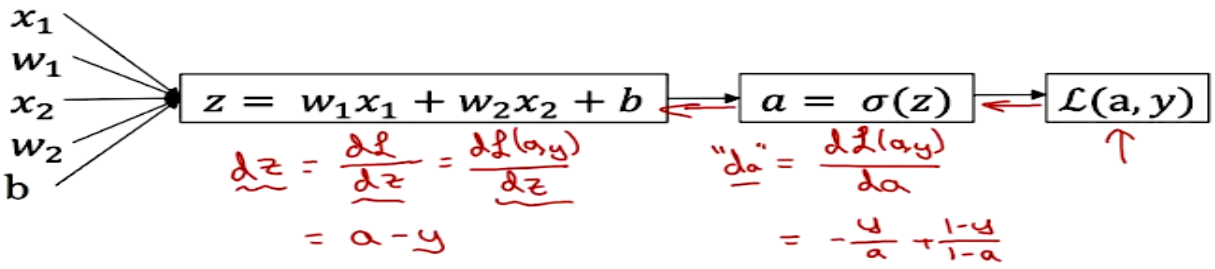
} simultaneous updates

Same concepts:
- Monitor gradient descent (learning curve)
- Vectorized implementation
- Feature scaling

Linear regression $f_{\vec{w},b}(\vec{x}) = \vec{w}\cdot\vec{x} + b$

Logistic regression $f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{(-\vec{w}\cdot\vec{x}+b)}}$

Logistic Regression Computation Graph



Pseudo code

```
J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
    z(i) = w^T x(i) + b
    a(i) = σ(z(i))
    J += -[y(i) log a(i) + (1 - y(i)) log(1 - a(i))]
    dz(i) = a(i) - y(i)
    dw1 += x1(i) dz(i)
    dw2 += x2(i) dz(i)
    db += dz(i)
J = J/m, dw1 = dw1/m, dw2 = dw2/m, db = db/m
```

for $j = 1 \ldots n$: dw_j ... $n_x = 2$

5. Regularization

Regularization encourages the learning algorithm to shrink the values of the parameters without necessarily demanding that the parameter is set to exactly 0. It lets you keep all of your features, but prevents the features from having an overly large effect. If you have a lot of features, you may not know which are the most important features and which ones to penalize. So the way regularization is typically implemented is to penalize all of the features (all the Wj parameters) that will usually result in fitting a smoother, simpler, less weekly function that's less prone to overfitting.

5.1. Cost Function with Regularization for Linear Regression

$$\min_{\vec{w},b} J(\vec{w},b) = \min_{\vec{w},b}\left[\overbrace{\frac{1}{2m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2}^{\text{mean squared error}} + \overbrace{\frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2}^{\text{regularization term}}\right]$$

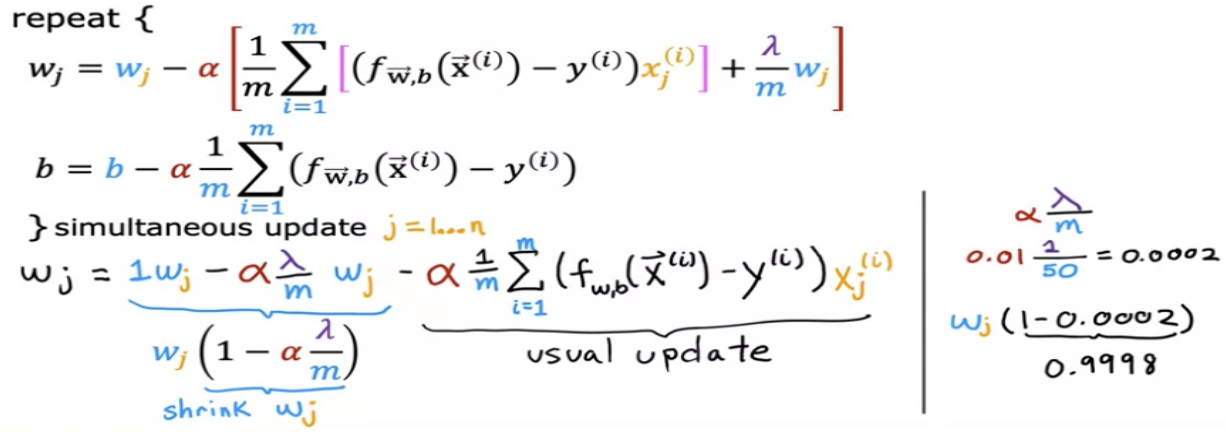
fit data

Keep w_j small

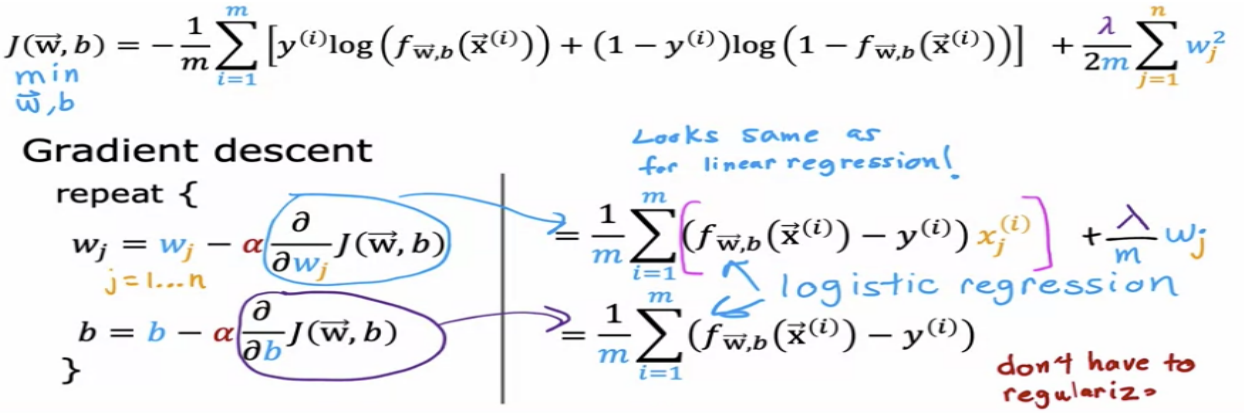
$\lambda = 0$

λ balances both goals

5.2. Gradient Descent of Linear Regression with Regularization



5.3. Gradient Descent of Logistic Regression with Regularization



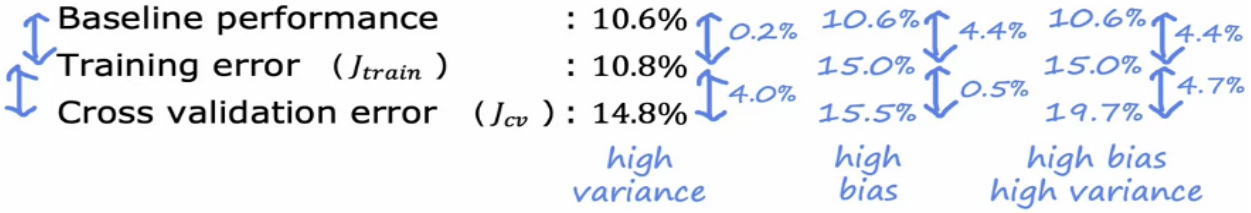
6. Tips on Model Building

You have implemented regularized linear regression but it makes unacceptably large error predictions.
Next steps to fix the issue:

- Get more training data (fixes high variance)
- Try smaller set of features (fixes high variance)
- Try getting additional features (fixes high bias)
- Try adding polynomial features (fixes high bias)
- Try decreasing lambda (fixes high bias)
- Try increasing lambda (fixes high variance)

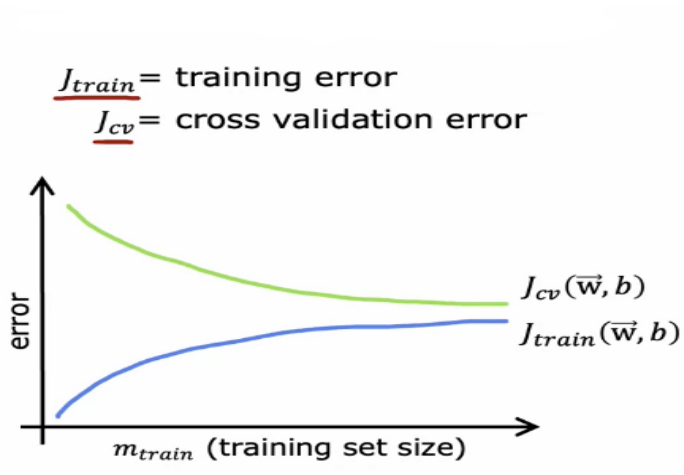
Establishing a baseline level of performance (error)

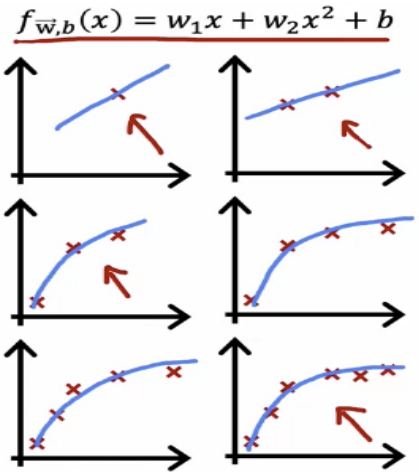
- Check if the model error higher than human error (speech recognition)
- Competing algorithms performance
- Guess based on prior experience
- High difference between baseline and training = high bias / training and dev = high variance



Learning curves

- Train set size increases | Training error (J train) increases and Dev error (J cv) decreases
- Dev error will be always higher than training error
- If model has high bias adding data will not improve it
- If model has high variance adding more data will improve it



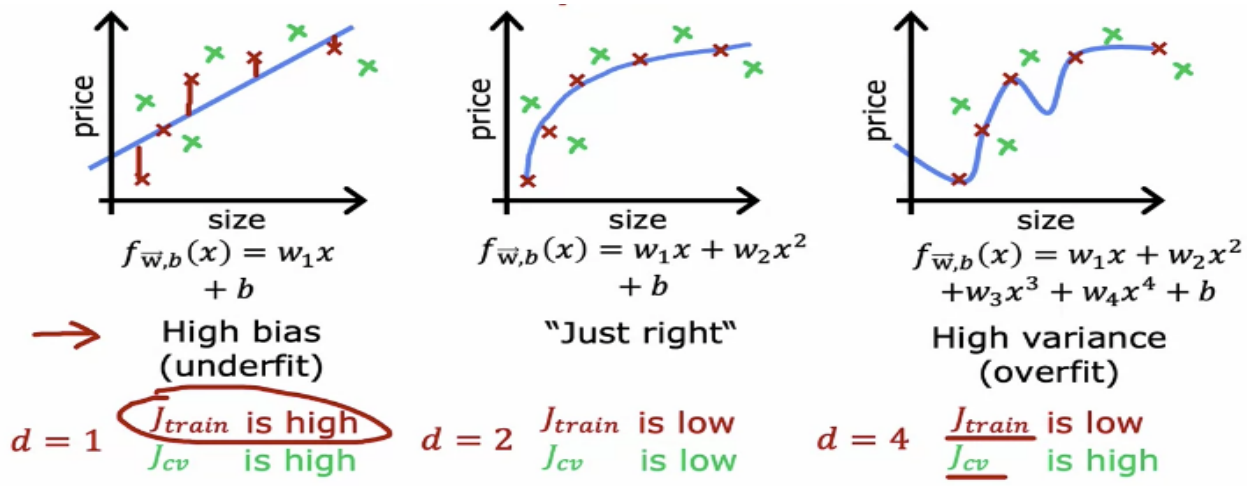


6.1. Model Selection

- Split data in 3 train / dev set / test set
    - The name cross-validation refers to that this is an extra dataset that we're going to use to check the accuracy of different models. You may also hear people call this extra dataset or validation set or development set.
- For regression models try adding polynomials (1 to 10) and see the prediction for cross-validation data set
- For neural networks try adding layers and increase number of neurons and see prediction of dev set
- Estimate generalization error (error when model sees new data) using test dataset

6.2. Bias and Variance

6.2.1. Polynomial and Bias/Variance



Fix overfitting:

- Collect more data
- Feature selection
- Regularization

6.2.2. Regularization and Bias/Variance

- Large Lambda = high bias (underfit)
- Small Lambda = high variance (overfit)
- Intermediate Lambda = just right!
- Try different lambdas (10) with cross validation

6.2.3. Neural Network and Bias/Variance

- Large neural networks are low bias machines
    - Requires big GPUs
    - Slow training
    - Large neural network will usually do as well or better than a smaller one so long as regularization is chosen appropriately
- Adding data fixes variance
    - Not always possible

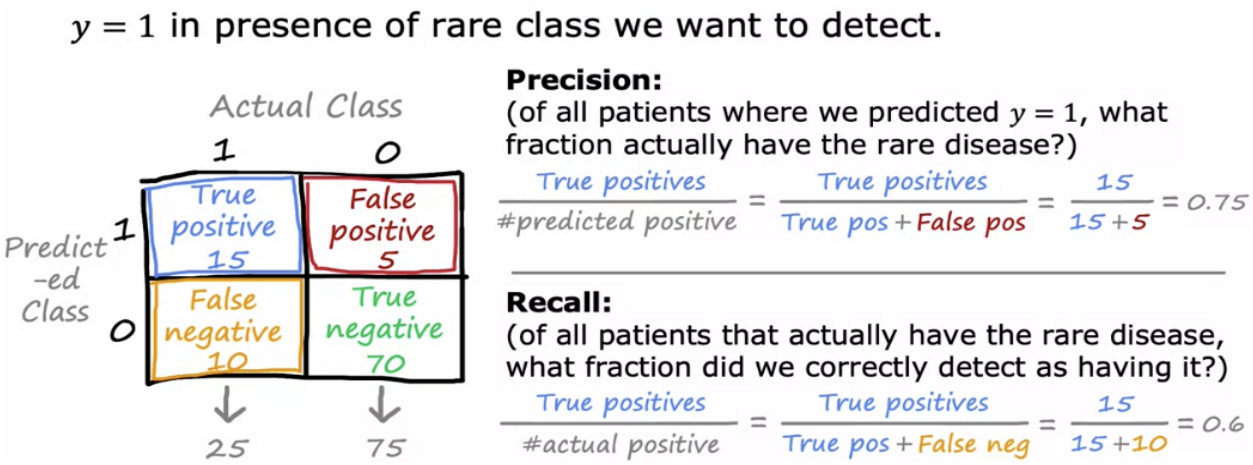
6.3. Error Metrics For Skewed Datasets

Skewed dataset is where the ratio of positive to negative examples are high, very far from 50-50, then it turns out that the usual error metrics like accuracy don't work that well.

Precision (positive predictive value) is the fraction of relevant instances among the retrieved instances.

Recall (sensitivity) is the fraction of relevant instances that were retrieved.

You can adjust the condition of predicting 0 or 1 by moving the threshold from 0.5 to another value and compare precision and recall values. When predict 1 is > 0.5 result in higher precision, lower recall and vice versa.



F1 score calculates average of precision and recall but in a way so that if any of the values are really small it will change accordingly.

	Precision (P)	Recall (R)	F_1
Model 1	88.3	79.1	83.4% ←
Model 2	97.0	7.3	13.6%

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

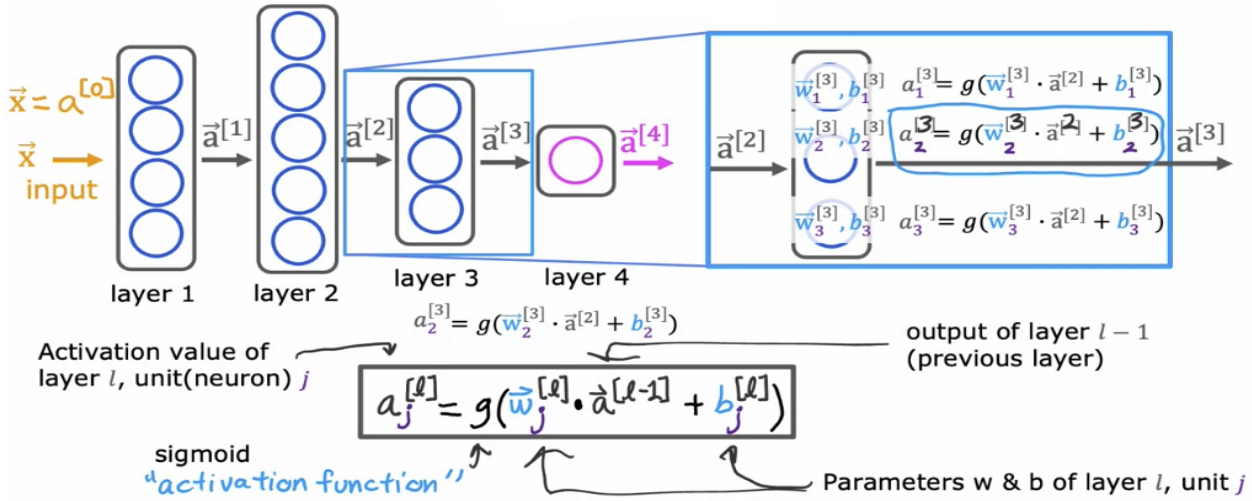
7. Neural Networks

The original motivation was to write software that could mimic how the human brain or how the biological brain learns and thinks. The artificial neural network uses a very simplified Mathematical model of what a biological neuron does. The neural network does feature engineering and makes the learning problem easier for itself. Multilayer perceptron is a neural network with multiple layers. Input layer is not counted towards dimensions of NN, below example is 4 layer NN.

Notaion

$a = f(x)$ = activation

$a^{[l]}$ = activation of layer l



Forward propagation neural network algorithm making computations left to right. (usually predicting)

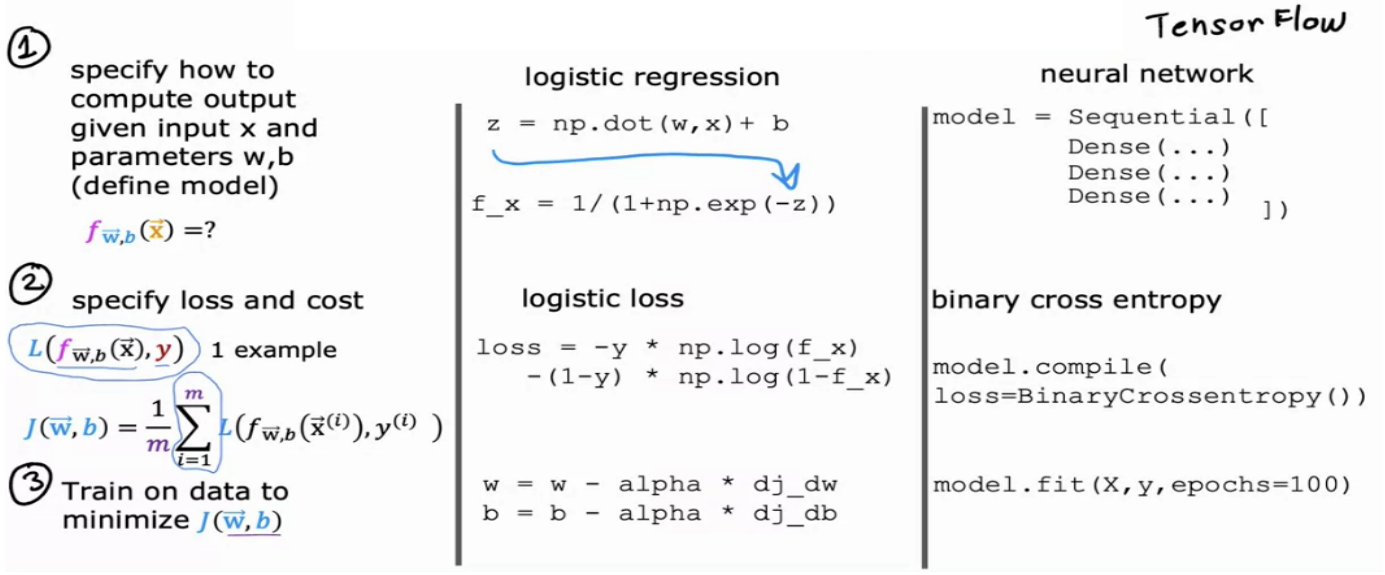
Back propagation neural network algorithm making computations right to left. (usually training the model)

Shallow NN is a NN with one or two layers. **Deep NN** is a NN with three or more layers

7.1. Vectorization

You can improve the performance of the model learning by carrying out matrix multiplication instead of for loop.

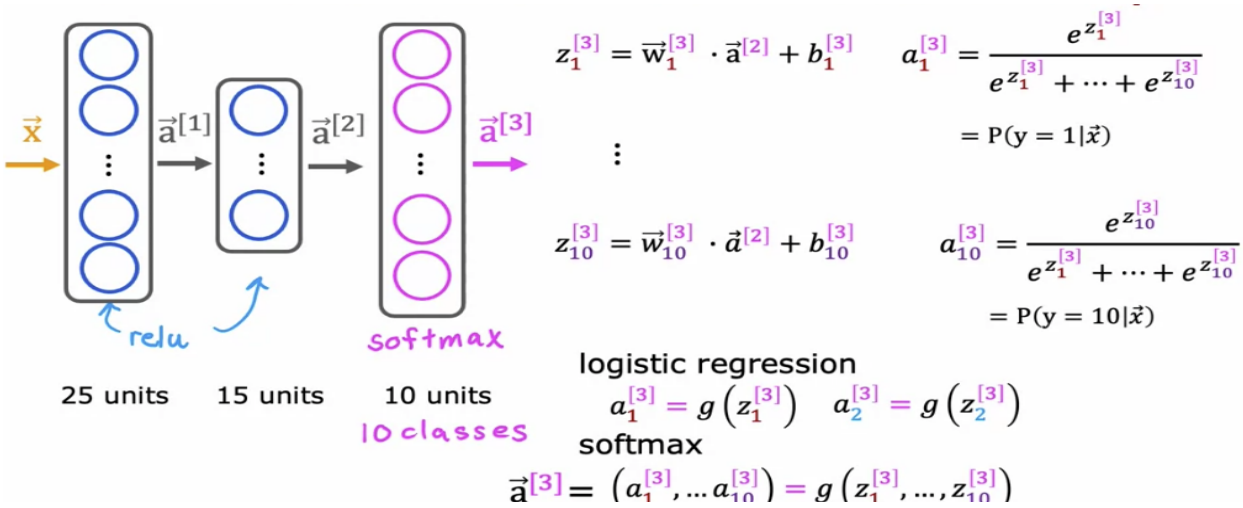
```
def dense(A_in,W,b):
    Z = np.matmul(A_in,W) + B
    A_out = g(Z)
    return A_out
```



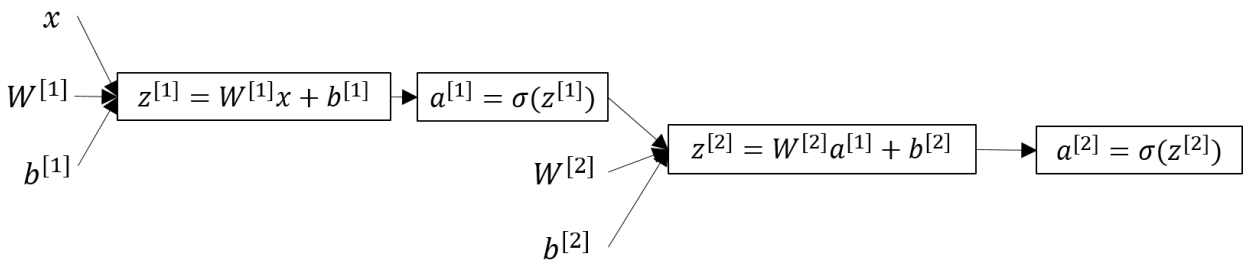
7.2. Activation Functions

	Linear (No activation)	**Sigmoid**	**Tanh**	**ReLU (Rectified Linear Unit)**	**Softmax**
Function	g(z) = z where z = wx+b	g(z) = 1/1+e**-z where z = wx+b	g(z) = e**z - e**-z / e**z+e**-z where z = wx+b	g(z) = max(0,z)	$g(z) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$
Use	Use for output neuron only of regression where **y** can be negative	Use only output layer **y** of binary classification	Use for hidden layers	Use for hidden layers / use for output layer **y** of regression where can't be negative	Use for output layers **y** of multiclass regression
Comments	Useless in hidden layers	If z very large or small the slope will be close to 0 and will slow down gradient descent	If z very large or small the slope will be close to 0 and will slow down gradient descent	z can be either 0 or positive number	Probability of jth

NN with Softmax output layer



7.3. Neural Network Computation Graph



7.4. Getting Your Matrix Dimensions Right

- The best way to debug your matrix dimensions is by a pencil and paper.
- Dimension of W is (n[l], n[l-1]). Can be thought of from right to left.
- Dimension of b is (n[1], 1)
- dw has the same shape as W, while db is the same shape as b
- Dimension of Z[l], a[l], dz[l], and da[l] is (n[l], m)

7.5. Building Blocks of Deep Neural Networks

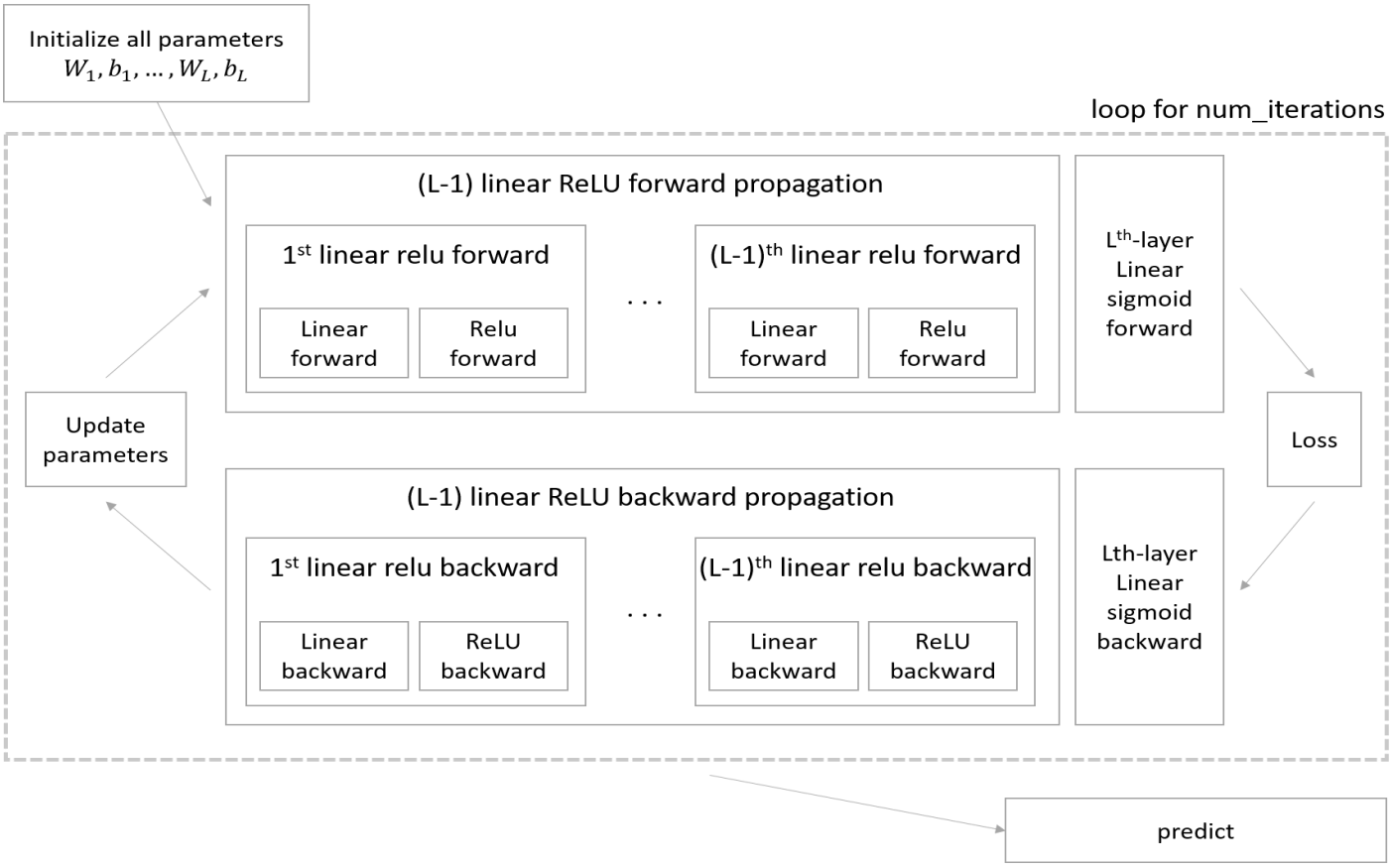
7.5.1. Why Deep Representations

- Deep neural networks with multiple hidden layers might be able to have the earlier layers learn lower level simple features and then have the later deeper layers then put together the simpler things it's detected in order to detect more complex things like recognize specific words or even phrases or sentences.
- If there aren't enough hidden layers, then we might require exponentially more hidden units to compute in shallower networks.

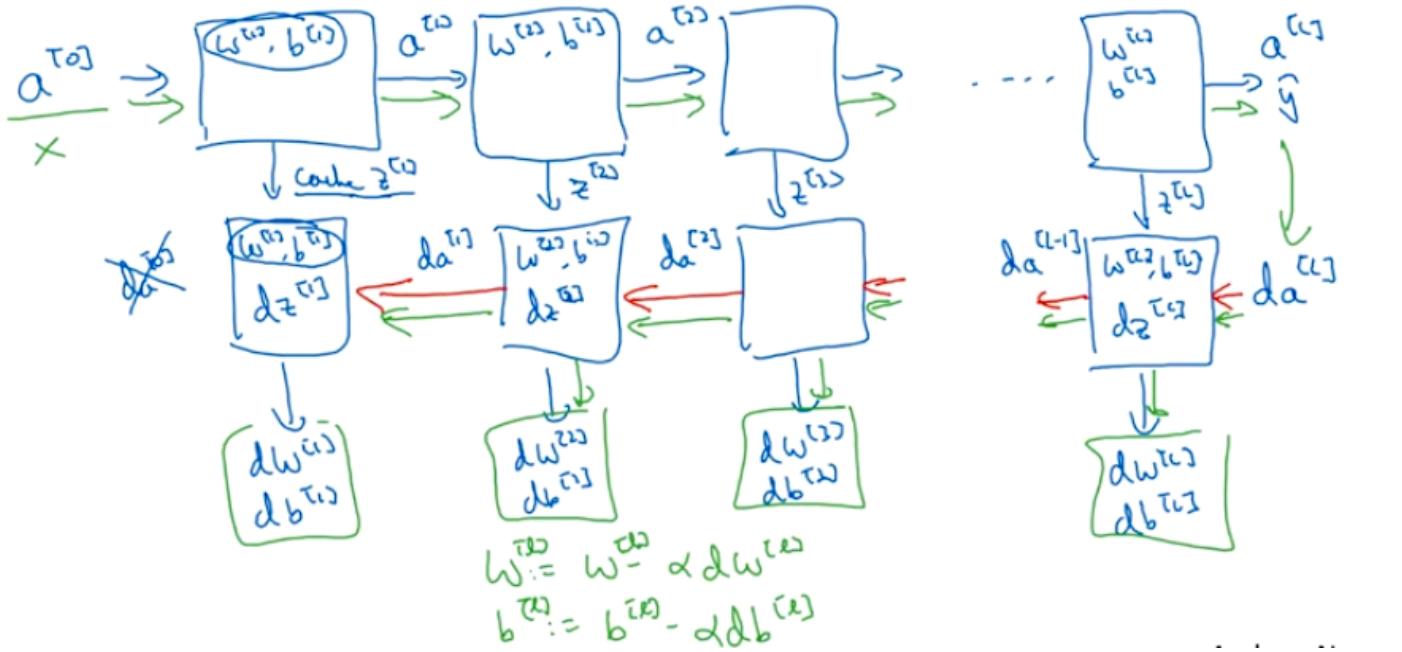
7.5.2. Deep Neural Network Implementation

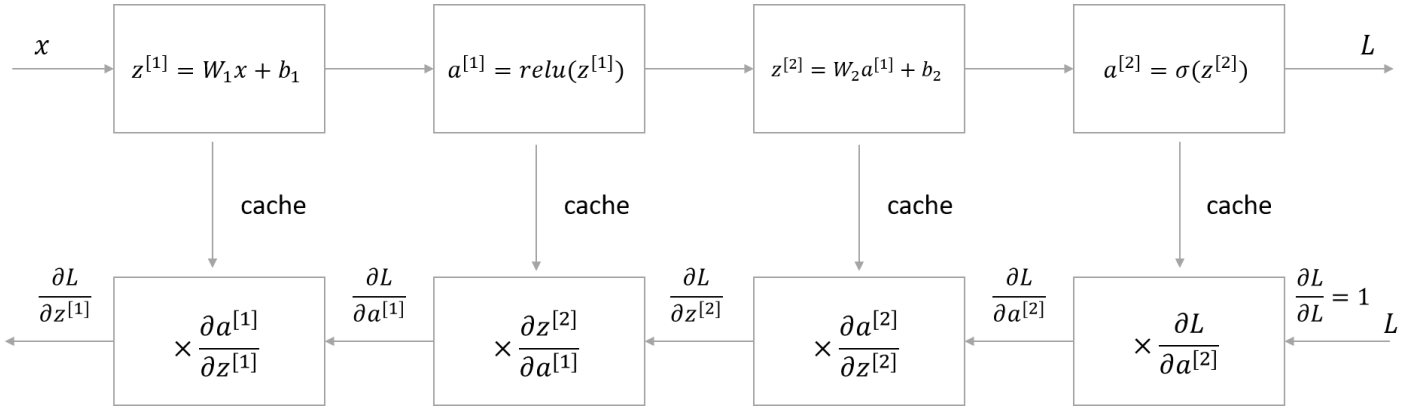
Implementation steps:

1. Initialize parameters / Define hyperparameters
2. Loop for num_iterations:
3. Forward propagation
4. Compute cost function
5. Backward propagation
6. Update parameters (using parameters, and grads from backprop)
7. Use trained parameters to predict labels
8. Forward and Backward Propagation



In the algorithm implementation, outputting intermediate values as caches (basically Z and A) of each forward step is crucial for backward computation.





7.5.3. Parameters vs Hyperparameters

Parameters:

- weight matrices W of each layer
- bias terms b of each layer

Hyperparameters (parameters that control the algorithm):

- number of hidden units n[l]
- learning rate
- number of iteration
- number of layers L
- choice of activation functions

7.6. Improving Deep Neural Networks

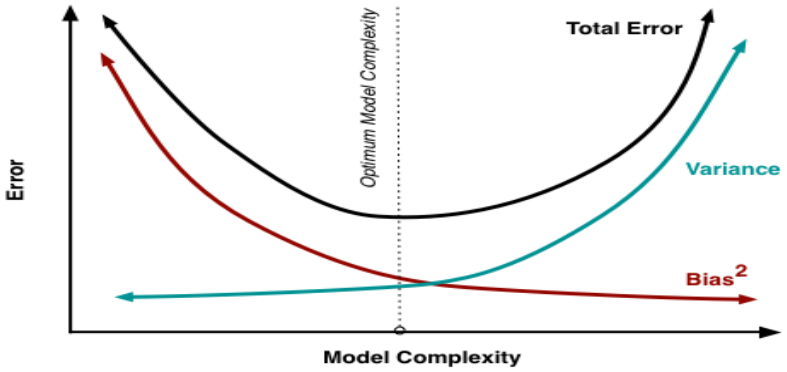
7.6.1. Train / Dev / Test sets

- Make sure the dev and test set are coming from the same distribution
- So the trend on the ratio of splitting the models:
    - If size of the dataset is 100 to 1,000,000==> 60/20/20
    - If size of the dataset is 1,000,000 to INF ==> 98/1/1 or 99.5/0.25/0.25

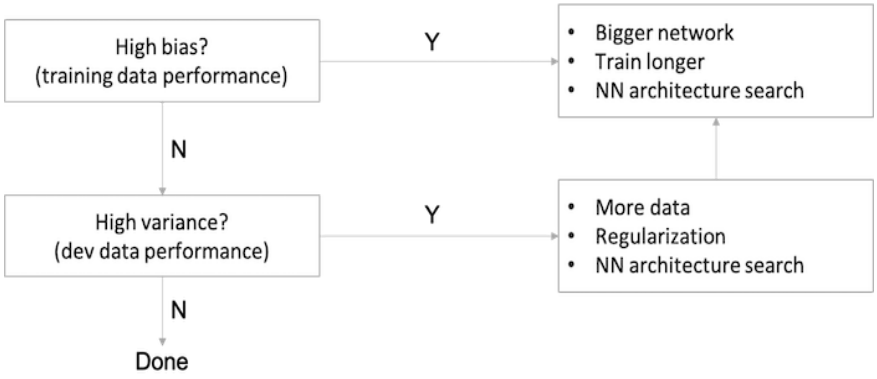
7.6.2. Bias and Variance

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
Error type	high variance	high bias	high bias, high variance	low bias, low variance

When we discuss prediction models, prediction errors can be decomposed into two main subcomponents we care about: error due to "bias" and error due to "variance". There is a tradeoff between a model's ability to minimize bias and variance.
Understanding these two types of error can help us diagnose model results and avoid the mistake of over- or under-fitting.



For a high bias problem, getting more training data is actually not going to help. In the modern deep learning, big data era, getting a bigger network and more data almost always just reduces bias without necessarily hurting your variance, so long as you regularize appropriately. The main cost of training a big neural network is just computational time, so long as you're regularizing.



7.6.3. Regularizing Neural Network

Why regularization reduces overfitting

If we set regularization lambda to be very big, then weight matrices will be set to be reasonably close to zero, effectively zeroing out a lot of the impact of the hidden units. Then the simplified neural network becomes a much smaller neural network, eventually almost like a logistic regression. We'll end up with a much smaller network that is therefore less prone to overfitting.
Taking activation function g(Z)=tanh(Z) as example, if lambda is large, then weights W are small and subsequently Z ends up taking relatively small values, where g and Z will be roughly linear which is not able to fit those very complicated decision boundary, i.e., less able to overfit.
b is just one parameter over a very large number of parameters, so no need to include it in the regularization.

Regularization	Formula	Description
L2 regularization	$\|\|w\|\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$	most common type of regularization
L1 regularization	$\|\|w\|\|_1 = \sum_{j=1}^{n_x} \|w_j\|$	w vector will have a lot of zeros, so L1 regularization makes your model sparse

Regularization for a Neural Network:

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

For the matrix w, this norm is called the Frobenius norm. Its definition looks like L2 norm but is not called the L2 norm:

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

Regularization of gradient:

$$dw^{(l)} = (backprop) + \frac{\lambda}{m} w^{(l)}$$

With regularization the coefficient of w is slightly less than 1, in which case it is called weight decay.

$$w^{(l)} = w^{(l)} - \alpha * dw^{(l)} = w^{(l)} - \alpha[(backprop) + \frac{\lambda}{m} w^{(l)}] = (1 - \frac{\alpha\lambda}{m}) w^{(l)} - \alpha(backprop)$$

Other Regularization Methods

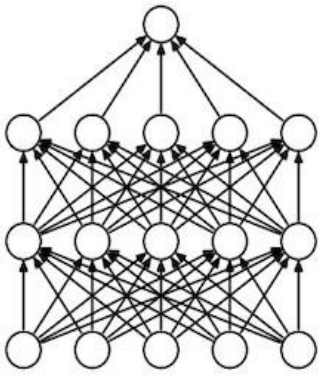
Data augmentation: getting more training data can be expensive and sometimes can't get more data, so flipping horizontally, random cropping, random distortion and translation of image can make additional fake training examples.

Early stopping: stopping halfway to get a mid-size w.

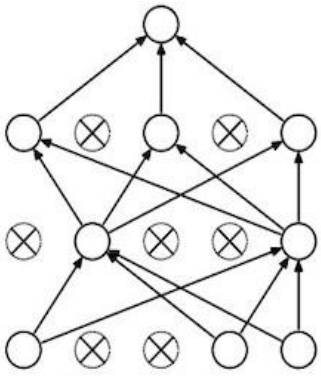
- Disadvantage: early stopping couples two tasks of machine learning, optimizing the cost function J and not overfitting, which are supposed to be completely separate tasks, to make things more complicated.
- Advantage: running the gradient descent process just once, you get to try out values of small w, mid-size w, and large w, without needing to try a lot of values of the L2 regularization hyperparameter lambda.

7.6.4. Dropout Regularization

- Dropout is another powerful regularization technique.
- With dropout, what we're going to do is go through each of the layers of the network and set some probability of eliminating a node in the neural network. It's as if on every iteration you're working with a smaller neural network, which has a regularizing effect.
- Inverted dropout technique, a3 = a3 / keep_prob, ensures that the expected value of a3 remains the same, which makes test time easier because you have less of a scaling problem.



(a) Standard Neural Net



(b) After applying dropout.

Understanding Dropout

- Can't rely on any one feature, so have to spread out weights, which has an effect of shrinking the squared norm of the weights, similar to what we saw with L2 regularization, helping prevent overfitting.
- For layers where you're more worried about over-fitting, the layers with a lot of parameters, you can set the key prop to be smaller to apply a more powerful form of drop out. E.g. For a layer with 10 neurons you can set keep_prob = 0.5 while a layer with 3 neurons = 0.8.
- Downside: more hyperparameters to search for using cross-validation if keep_prop given for some layers.
- Frequently used in computer vision, as the input size is so big, inputting all these pixels that you almost never have enough data, prone to overfitting.
- Cost function J is no longer well-defined and harder to debug or double check that J is going downhill on every iteration. So first run code turn off dropout and make sure old J is monotonically decreasing, and then turn on dropout in order to make sure that no bug in dropout.

Note: A common mistake when using dropout is to use it both in training and testing. You should use dropout only in training.

7.6.5. Setting up Optimization Problem

Vanishing / Exploding gradients

In a very deep network derivatives or slopes can sometimes get either very big or very small, maybe even exponentially, and this makes training difficult.

The weights W, if they're all just a little bit bigger than one or just a little bit bigger than the identity matrix, then with a very deep network the activations can explode. And if W is just a little bit less than identity, the activations will decrease exponentially.

Weight Initialization for Deep Networks

A partial solution to the problems of vanishing and exploding gradients is better or more careful choice of the random initialization for neural networks.

For a single neuron, suppose we have n features for the input layer, then we want Z = W1X1 + W2X2 + ... + WnXn not blow up and not become too small, so the larger n is, the smaller we want Wi to be.

- It's reasonable to set variance of Wi to be equal to 1/n
- It helps reduce the vanishing and exploding gradients problem, because it's trying to set each of the weight matrices W not too much bigger than 1 and not too much less than 1.
- Generally for layer l, set W[l]=np.random.randn(shape) * np.sqrt(1/n[l-1]).
  - For RELU activation, set Var(W)=2/n by W[l]=np.random.randn(shape) * np.sqrt(2/n[l-1]). (aka He initialization by Kaiming He)
  - For tanh activation, W[l]=np.random.randn(shape) * np.sqrt(1/n[l-1]). (Xavier initialization)
  - W[l]=np.random.randn(shape) * np.sqrt(2/(n[l-1]+n[l])) (Yoshua Bengio)
- 1 or 2 in variance Var(W)=1/n or 2/n can be a hyperparameter, but not as important as other hyperparameters.

A well chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error

Implementation tips:

- The weights W[l] should be initialized randomly to break symmetry and make sure different hidden units can learn different things. Initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing.
- It is however okay to initialize the biases b[l] to zeros. Symmetry is still broken so long as W[l] is initialized randomly.
- Initializing weights to very large random values does not work well.
- Hopefully initializing with small random values does better. The important question is: how small should these random values be? The initialization works well for networks with ReLU activations. In other cases, try other initializations.

Numerical approximation of gradients

Numerically verify implementation of derivative of a function is correct and hence check if there is a bug in the backpropagation implementation.

Two-sided difference formula is much more accurate:

- In two side case, $f'(\theta) = lim(f(\theta + \varepsilon) - f(\theta - \varepsilon))/(2\varepsilon)$, error term ~ O($\varepsilon$^2)
- In one side case, $f'(\theta) = lim(f(\theta + \varepsilon) - f(\theta))/(\varepsilon)$, error term ~ O($\varepsilon$)
- $\varepsilon < 1, so\ O(\varepsilon\text{^}2) < O(\varepsilon)$

Gradient checking

Implementation steps:

1. Take W[1],b[1],...,W[L],b[L] and reshape into a big vector θ: J(W[1],b[1],...,W[L],b[L])=J(θ).
2. Take dW[1],db[1],...,dW[L],db[L] and reshape into a big vector dθ.
3. For each i: dθ_approx[i] = (J(θ1,θ2,...,θi+ε,...)-J(θ1,θ2,...,θi-ε,...))/(2ε). (Should have dθ_approx[i] ≈ dθ[i])
4. Check diff_ratio = norm_2(dθ_approx-dθ) / (norm_2(dθ_approx)+norm_2(dθ)) ≈ eps:
5. diff_ratio ≈ 10^-7, great, backprop is very likely correct.
6. diff_ratio ≈ 10^-5, maybe OK, better check no component of this difference is particularly large.
7. diff_ratio ≈ 10^-3, worry, check if there is a bug.

Gradient checking implementation notes

- Don't use in training - only to debug
- If an algorithm fails a grad check, look at components to try to identify bug
- Remember regularization
- Doesn't work with dropout (you can first check grad, then turn on dropout)
- Run at random initialization; perhaps again after some training

7.6.6. Optimization

Mini-batch Gradient Descent

Vectorization allows you to process all M examples relatively quickly if M is very large, but it can still be slow. For example, m = 5,000,000 (or m = 50,000,000 or even bigger), we have to process the entire training sets of five million training samples before we take one little step of gradient descent.

We can use the mini-batch method to let gradient descent start to make some progress before we finish processing the entire, giant training set of 5 million examples by splitting up the training set into smaller, little baby training sets called mini-batches. In this case, we have 5000 mini-batches with 1000 examples each.

Notations:

(i): the i-th training sample

[l]: the l-th layer of the neural network

{t}: the t-th mini batch

Understanding Mini-batch Gradient Descent

Size	**Method**	**Description**	**Guidelines**
=m	Batch Gradient Descent	Cost function decreases on every iteration; Too long per iteration (epoch).	For a small training set (<2000).
=1	Stochastic Gradient Descent	Too noisy regarding cost minimization (can be reduced by using smaller learning rate) Wander around minimum never converges; Lose speedup from vectorization, inefficient.	Use a smaller learning rate when it oscillates too much.
between 1 and m	Mini-batch Gradient Descent	Faster learning: vectorization advantage and make progress without waiting to process the entire training set; Not guaranteed to always head toward the minimum but more consistently in that direction than stochastic descent; Not always exactly converge, may oscillate in a very small region, reducing the learning rate slowly may also help.	Mini-batch size is a hyperparameter; Batch size better in [64, 128, 256, 512], a power of 2;

Exponentially Weighted Averages

Moving averages are favored statistical tools of active traders to measure momentum. There are three MA methods:

MA methods	**Calculations**
Simple Moving Average (SMA)	Calculated from the average closing prices for a specified period
Weighted Moving Average (WMA)	Calculated by multiplying the given price by its associated weighting (assign a heavier weighting to more current data points) and totaling the values
Exponential Moving Average (EWMA)	Also weighted toward the most recent prices, but the rate of decrease is exponential

The exponentially weighted average adds a fraction β of the current value to a leaky running sum of past values.

General Equation: $\nu_t = \beta\nu_{t-1} + (1 - \beta)\theta_t$

- If we plot this it will represent averages over ~ (1 / 1 (1 - beta)) entries:
  - beta = 0.9 will average last 10 entries
  - beta = 0.98 will average last 50 entries
  - beta = 0.5 will average last 2 entries
- Best beta average for our case is between 0.9 and 0.98

Because when we're implementing the exponentially weighted moving average, we initialize it with V0=0, subsequently we have the following result in the beginning of the iteration:
V1 = 0.98*V0 + 0.02*θ1 = 0.02 * θ1
V2 = 0.98*V1 + 0.02*θ2 = 0.0196 * θ1 + 0.02 * θ2
As a result, V1 and V2 calculated by this are not very good estimates of the first two data points. So we need some modification to make it more accurate, especially during the initial phase of our estimate to avoid an initial bias. This can be corrected by scaling with 1/(1-β^t) where t is the iteration number.

Gradient descent with momentum

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using momentum can reduce these oscillations.

- Gradient descent with momentum, which computes an EWA of gradients to update weights almost always works faster than the standard gradient descent algorithm.
- Algorithm has two hyperparameters of alpha, the learning rate, and beta which controls your exponentially weighted average. The common value for beta is 0.9.
- Don't bother with bias correction

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$

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

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

Implementation tips:

- If β = 0, then this just becomes standard gradient descent without momentum.
- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, β = 0.9 is often a reasonable default.
- It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.

RMSprop

RMSprop(root mean square), similar to momentum, has the effects of damping out the oscillations in gradient descent and mini-batch gradient descent and allowing you to maybe use a larger learning rate alpha. The algorithm computes the exponentially weighted averages of the squared gradients and updates weights by the square root of the EWA.

Adam

Adam (Adaptive Moment Estimation) optimization algorithm is basically putting momentum and RMSprop together and combines the effect of gradient descent with momentum together with gradient descent with RMSprop. This is a commonly used learning algorithm that is proven to be very effective for many different neural networks of a very wide variety of architectures. In the typical implementation of Adam, bias correction is on.
Implementation tips:

- It calculates an exponentially weighted average of past gradients, and stores it in variables V_dW,V_db (before bias correction) and V_dW_c,V_db_c (with bias correction).
- It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables S_dW,S_db (before bias correction) and S_dW_c,S_db_c (with bias correction).
- It updates parameters in a direction based on combining information from "1" and "2".
- Tune learning rate, beta1 (parameter of the momentum, for dW) is set 0.9, beta2 (parameter of the RMSprop, for dW^2) is set 0.999, ε (avoid dividing by zero) is set 10^-8. You can tune beta1 and beta2 but not necessary.

Learning rate decay

The learning algorithm might just end up wandering around, and never really converge, because you're using some fixed value for alpha. Learning rate decay methods can help by making learning rate smaller when optimum is near.

$\alpha = \frac{1}{1+decayRate \times epochNumber}\ \alpha_0$

There are several decay methods:

Decay factor	**Description**
0.95^epoch_num * α	exponential decay
k/sqrt(epoch_num) * α or k/sqrt(t) * α	manual decay
discrete staircase	piecewise constant
manual decay	

The problem of local optima

- First, you're actually pretty unlikely to get stuck in bad local optima, but much more likely to run into a saddle point, so long as you're training a reasonably large neural network, save a lot of parameters, and the cost function J is defined over a relatively high dimensional space.
- Second, that plateaus are a problem and you can actually make learning pretty slow. And this is where algorithms like momentum or RMSProp or Adam can really help your learning algorithm.

7.7. Hyperparameter Tuning

Importance of hyperparameters (roughly):

1. learning rate alpha
2. momentum term beta, mini-batch size, number of hidden units
3. number of layers, learning rate decay, Adam (beta1, beta2, epsilon)

Tuning tips:

- Choose points at random, not in a grid
  - Search for hyperparameters on a log scale.
- Optionally use a coarse to fine search process

Hyperparameters tuning in practice: Panda vs. Caviar

Panda approach: Not enough computational capacity: babysitting one model

Caviar approach: training many models in parallel

7.7.1. Batch Normalization

Normalizing activations in a network

- Batch normalization makes your hyperparameter search problem much easier, and makes your neural network much more robust.
- What batch norm does is it applies that normalization process not just to the input layer, but to the values even deep in some hidden layer in the neural network. So it will apply this type of normalization to normalize the mean and variance of z[i] of hidden units.
- One difference between the training input and these hidden unit values is that you might not want your hidden unit values to be forced to have mean 0 and variance 1.
  - For example, if you have a sigmoid activation function, you don't want your values to always be clustered in the normal distribution around 0. You might want them to have a larger variance or have a mean that's different than 0, in order to better take advantage of the nonlinearity of the sigmoid function rather than have all your values be in just this linear region (near 0 on sigmoid function).
  - What it does really is it then shows that your hidden units have standardized mean and variance, where the mean and variance are controlled by two explicit parameters gamma and beta which the learning algorithm can set to whatever it wants.

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

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

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

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

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

Use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$

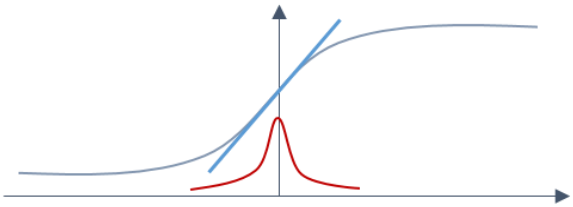
if $\gamma = \sqrt{\sigma^2 + \varepsilon}$

$\beta = \mu$

then

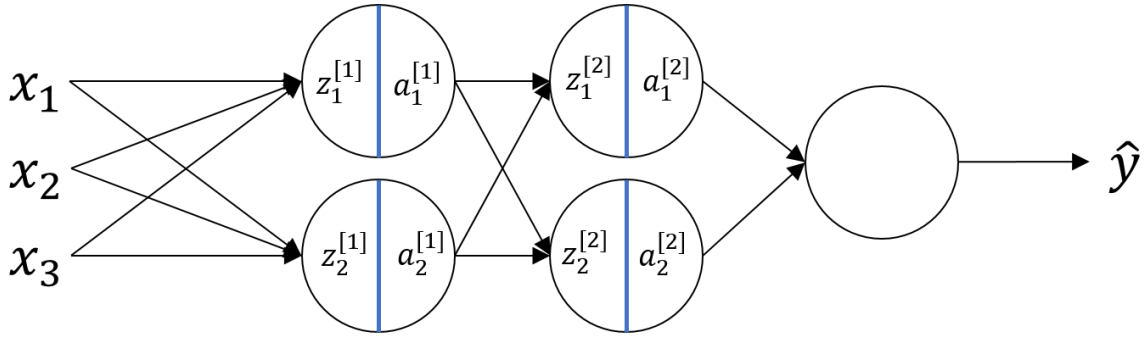
$\tilde{z}^{(i)} = z^{(i)}$

(γ, β is learnable parameters)



Fitting Batch Norm into a neural network

- $\beta[1],\gamma[1],\beta[2],\gamma[2],\cdots,\beta[L],\gamma[L]$ can also be updated using gradient descent with momentum (or RMSprop, Adam). $\beta[l],\gamma[l]$ have the shape with z[l].
- Similar computation can also be applied to mini-batches.
- With batch normalization, the parameter b[l] can be eliminated. So w[l],β[l],γ[l] need to be trained.
- The parameter β here has nothing to do with the beta in the momentum, RMSprop or Adam algorithms.



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

Parameters:

$w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \cdots, w^{[L]}, b^{[L]}$

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \cdots, \beta^{[L]}, \gamma^{[L]}$

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

$tf.nn.batch\_normlization$

7.7.2. Hyperparameter Tuning Strategy

Ideas to improve a machine learning system: collect more data, collect more diverse training sets, train algorithms longer with gradient descent, try Adam instead of gradient descent, bigger network, smaller network, dropout, L2 regularization , network architecture (activation functions, number of hidden units) and etc.

In order to have quick and effective ways to figure out which of these ideas and maybe even other ideas, are worth pursuing and which ones we can safely discard, we need ML strategies.

Orthogonalization: In the example of TV tuning knobs, orthogonalization refers to that the TV designers had designed the knobs so that each knob kind of does only one thing.

Chain of assumptions in ML	**Tune the knobs**
Fit training set well on cost function	Bigger network / Better optimization algorithm (Adam)
Fit dev set well on cost function	Regularization / Bigger training set
Fit test set well on cost function	Bigger dev set
Performs well in real world	Change dev set or cost function

7.8. Multi-task Learning

Multi-task learning refers to having one neural network do several tasks simultaneously.

When to use multi-task learning:

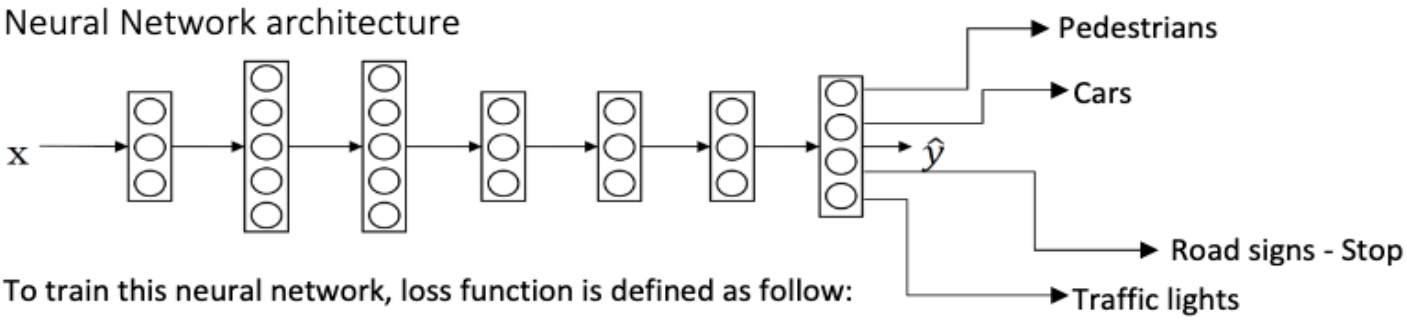
- Training on a set of tasks that could benefit from having shared lower-level features
- Usually: Amount of data you have for each task is quite similar
- Can train a big enough neural network to do well on all tasks

The input $x^{(i)}$ is the image with multiple labels

The output $y^{(i)}$ has 4 labels which are represents:

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} \text{Pedestrians} \\ \text{Cars} \\ \text{Road signs - Stop} \\ \text{Traffic lights} \end{matrix}$$

$$Y = \begin{bmatrix} | & | & | & | \\ y^{(1)} & y^{(2)} & y^{(3)} & y^{(4)} \\ | & | & | & | \end{bmatrix} \qquad Y = (4, m) \qquad Y = (4,1)$$



To train this neural network, loss function is defined as follow:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\left(y_j^{(i)}\log\left(\hat{y}_j^{(i)}\right)+\left(1-y_j^{(i)}\right)\log\left(1-\hat{y}_j^{(i)}\right)\right)$$

When multi-task learning makes sense

- Training on a set of tasks that could benefit from having shared lower-level feature
- Usually: amount of data you have for each task is quite similar
- Can train a big enough neural network to do well on all the tasks

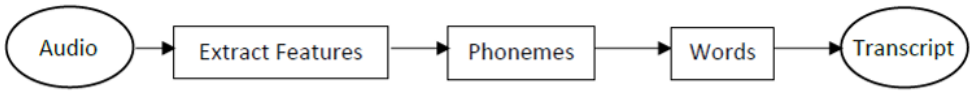
7.9. End-to-end Deep Learning

What is end-to-end deep learning

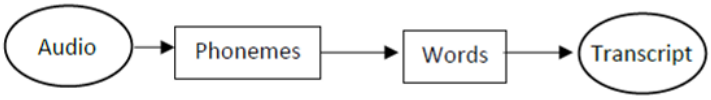
End-to-end deep learning is the simplification of a processing or learning system into one neural network. End-to-end deep learning cannot be used for every problem since it needs a lot of labeled data. It is used mainly in audio transcripts, image captures, image synthesis, machine translation, steering in self-driving cars, etc.

Example - Speech recognition model

The traditional way - small data set



The hybrid way - medium data set



The End-to-End deep learning way – large data set

Whether to use end-to-end deep learning

Before applying end-to-end deep learning, you need to ask yourself the following question: Do you have enough data to learn a function of the complexity needed to map x and y?

1. Pros:
   a. Let the data speak. By having a pure machine learning approach, the neural network will learn from x to y. It will be able to find which statistics are in the data, rather than being forced to reflect human preconceptions.
   b. Less hand-designing of components needed. It simplifies the design workflow.
2. Cons:
   a. Large amount of labeled data. It cannot be used for every problem as it needs a lot of labeled data.
   b. Excludes potentially useful hand-designed components. Data and any hand-design's components or features are the 2 main sources of knowledge for a learning algorithm. If the data set is smaller than a hand-design system is a way to give manual knowledge into the algorithm.

7.10. TensorFlow

7.10.1. Round-off errors

It turns out that while the way we have been computing the cost function for softmax is correct, there's a different way of formulating it that reduces these numerical round-off errors, leading to more accurate computations within TensorFlow.
Add (from_logits=True)
If one of the z's really small than e to negative small number becomes very, very small or if one of the z's is a very large number, then e to the z can become a very large number and by rearranging terms, TensorFlow can avoid some of these very small or very large numbers and therefore come up with more accurate computation for the loss function.

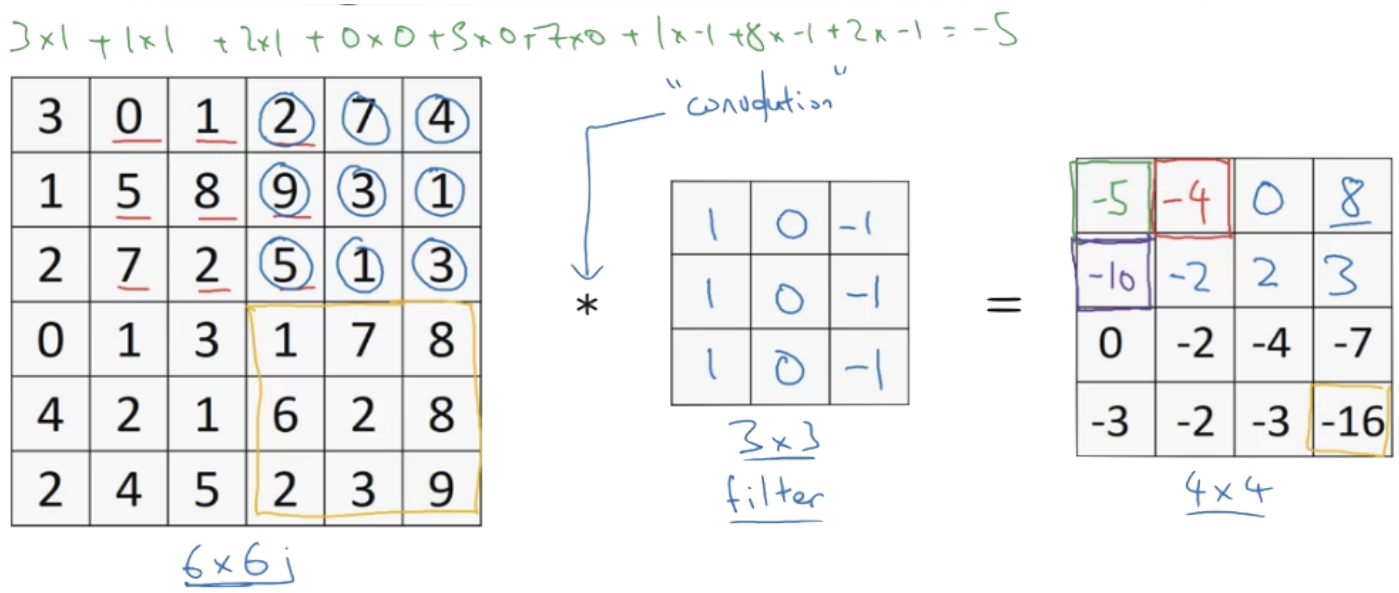
8. Convolutional Neural Networks (CNN)

8.1. Foundations of Convolutional Neural Networks

8.1.1. Edge Detection Example

- The convolution operation is one of the fundamental building blocks of a convolutional neural network.
- Early layers of the neural network might detect edges and then some later layers might detect parts of objects and then even later layers may detect parts of complete objects like people's faces.
- Given a picture for a computer to figure out what are the objects in the picture, the first thing you might do is maybe detect edges in the image.
- The convolution operation gives you a convenient way to specify how to find these vertical edges in an image.

A 3 by 3 filter or 3 by 3 matrix may look like below, and this is called a vertical edge detector or a vertical edge detection filter. In this matrix, pixels are relatively bright on the left part and relatively dark on the right part.
Convolving it with the vertical edge detection filter results in detecting the vertical edge down the middle of the image.



Edge Detection Filters

Vertical			Horizontal			Sobel			Scharr			Back Propagation		
1	0	-1	1	1	1	1	0	-1	3	0	-3	w1	w2	w3
1	0	-1	0	0	0	2	0	-2	10	0	-10	w4	w5	w6
1	0	-1	-1	-1	-1	1	0	-1	3	0	-3	w7	w8	w9

8.1.2. Padding

Padding is usually applied in the convolutional operation in order to fix the following two problems:

- Every time you apply a convolutional operator the image shrinks.
- A lot of information from the edges of the image is thrown away.

Notations:
image size: n x n
convolution size: f x f
padding size: p
Output size after convolution:
Without padding: (n-f+1) x (n-f+1)
With padding: (n+2p-f+1) x (n+2p-f+1)

How much padding to add:

- Valid convolutions: no padding
- Same convolutions: pad so that output size is the same as the input size,
  - f is usually odd
  - p = (f-1)/2
  - e.g. f = 5 then p = 2

8.1.3. Strided Convolutions

Notation: stride s
Output size after convolution: floor((n+2p-f)/s+1) x floor((n+2p-f)/s+1)
Conventions:

- The filter must lie entirely within the image or the image plus the padding region.
- In the deep learning literature by convention, a convolutional operation (maybe better called cross-correlation) is what we usually do not bother with a flipping operation, which is included before the product and summing step in a typical math textbook or a signal processing textbook.
- In the latter case, the filter is flipped vertically and horizontally.

8.1.4. Convolutions Over Volume

For a RGB image, the filter itself has three layers corresponding to the red, green, and blue channels.
Height x Width x Channel

`n x n x nc` * `f x f x nc` --> `(n-f+1) x (n-f+1) x nc'`

8.1.5. One Layer of a Convolutional Network

Types of layer in a convolutional network:

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

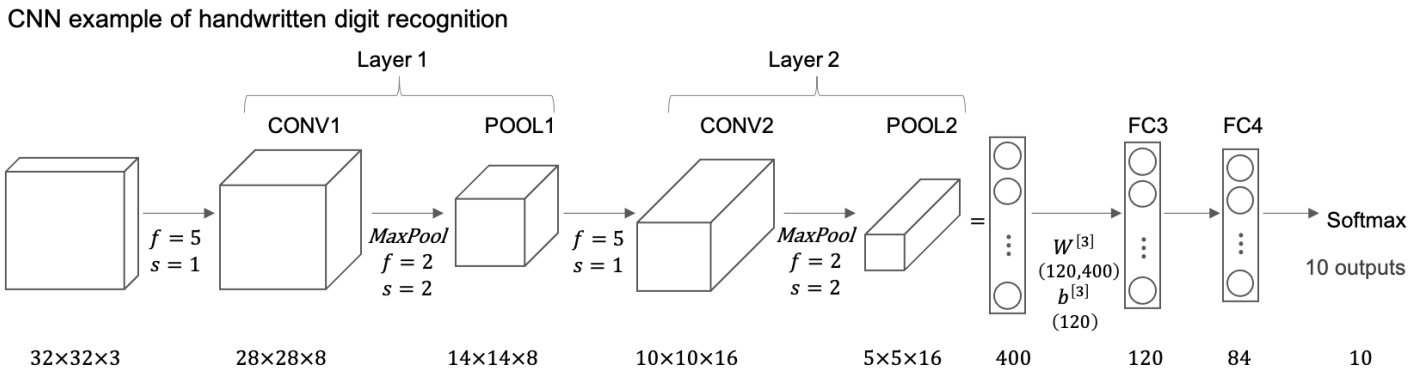
Pooling Layers

- Max Pooling is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map
- One interesting property of max pooling is that it has a set of hyperparameters but it has no parameters to learn. There's actually nothing for gradient descent to learn
- Formulas that we had developed previously for figuring out the output size for the CONV layer also work for max pooling.
- When you do max pooling, usually, you do not use any padding

CNN Example

- Because the pooling layer has no weights, has no parameters, only a few hyperparameters
- As you go deeper usually the height and width will decrease, whereas the number of channels will increase
- The CONV layers tend to have relatively few parameters and a lot of the parameters tend to be in the fully collected layers of the neural network
- The activation size tends to maybe go down gradually as you go deeper in the neural network. If it drops too quickly, that's usually not great for performance as well

size	notation
filter size	$f^{[l]}$
padding size	$p^{[l]}$
stride size	$s^{[l]}$
number of filters	$n_c^{[l]}$
filter shape	$f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
input shape	$n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
output shape	$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
output height	$n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
output width	$n_W^{[l]} = \lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
activations `a[l]`	$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
activations `A[l]`	$m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
weights	$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
bias	$(1, 1, 1, n_c^{[l]})$



Layer shapes of the network:

Layer	**Activation Shape**	**Activation Size**	**# parameters**
Input	(32,32,3)	3072	0
CONV1	(f=5,s=1)	6272	608 =(5*5*3+1)*8
POOL1	(14,14,8)	1568	0
CONV2	(f=5,s=1)	1600	3216 =(5*5*8+1)*16
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48120 =400*120+120
FC4	(84,1)	84	10164 =120*84+84
Softmax	(10,1)	10	850 =84*10+10

8.1.6. Why Convolutions

There are two main advantages of convolutional layers over just using fully connected layers.

- 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: In each layer, each output value depends only on a small number of inputs.

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

Convolutional structure helps the neural network encode the fact that an image shifted a few pixels should result in pretty similar features and should probably be assigned the same output label. And the fact that you are applying the same filter in all the positions of the image, both in the early layers and in the late layers, helps a neural network automatically learn to be more robust or to better capture the desirable property of translation invariance.

8.2. Classic Networks

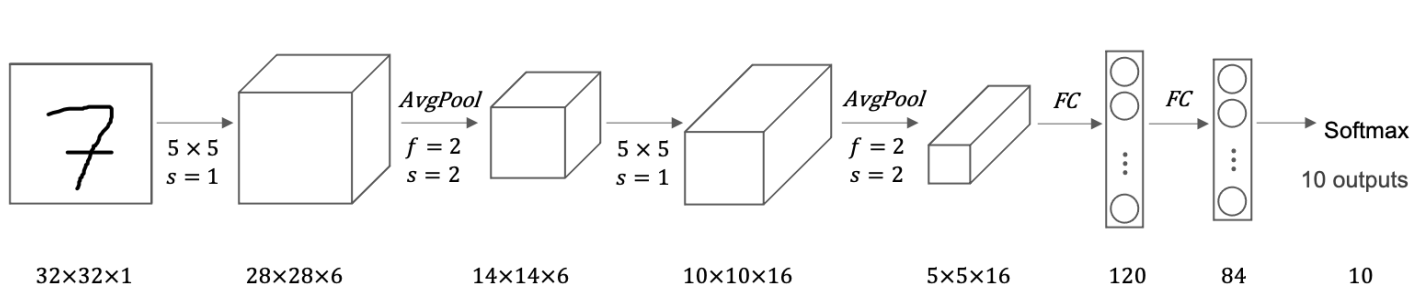
Why look at case studies

It is helpful in taking someone else's neural network architecture and applying that to another problem.

- Classic networks
  - LeNet-5
  - AlexNet
  - VGG
- ResNet
- Inception

8.2.1. Classic Networks

LeNet-5



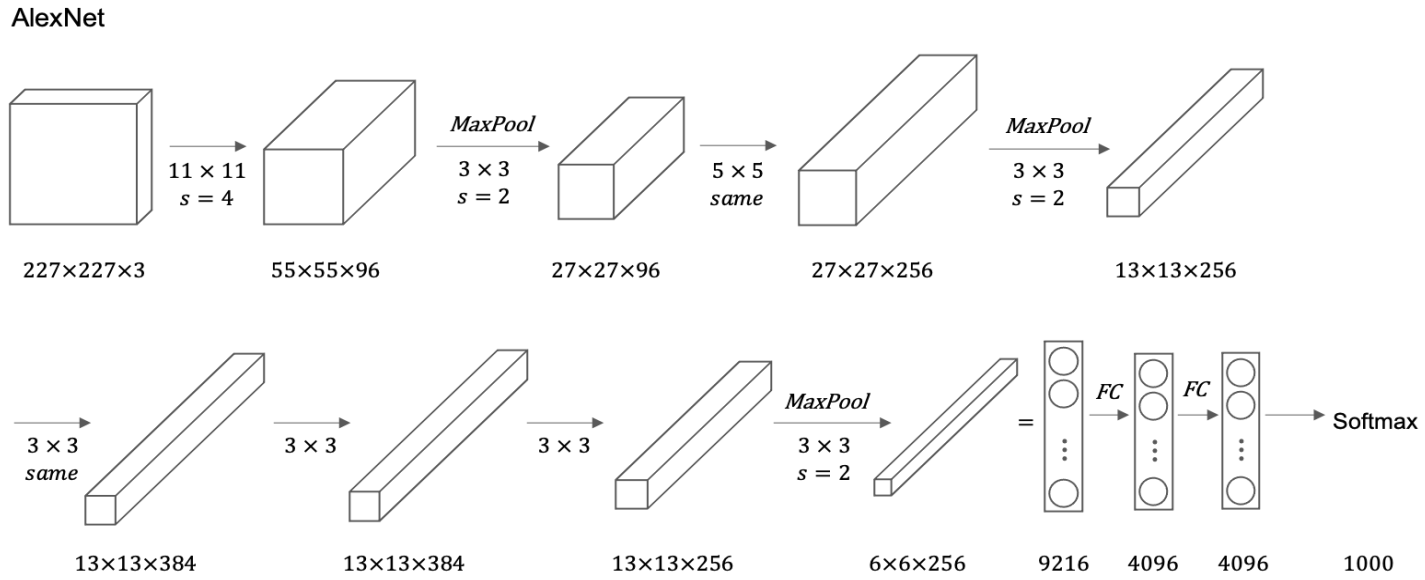
Some difficult points about reading the LeNet-5 paper:

- Back then, people used sigmoid and tanh nonlinearities, not ReLU
- To save on computation as well as some parameters, the original LeNet-5 had some crazy complicated way where different filters would look at different channels of the input block. And so the paper talks about those details, but the more modern implementation wouldn't have that type of complexity these days
- One last thing that was done back then I guess but isn't really done right now is that the original LeNet-5 had a non-linearity after pooling, and I think it actually uses sigmoid nonlinearity after the pooling layer

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

Andrew Ng recommends reading the paper and focusing on section two which talks about this architecture, and take a quick look at section three which has a bunch of experiments and results, which is pretty interesting. Later sections talked about the graph transformer network, which isn't widely used today.

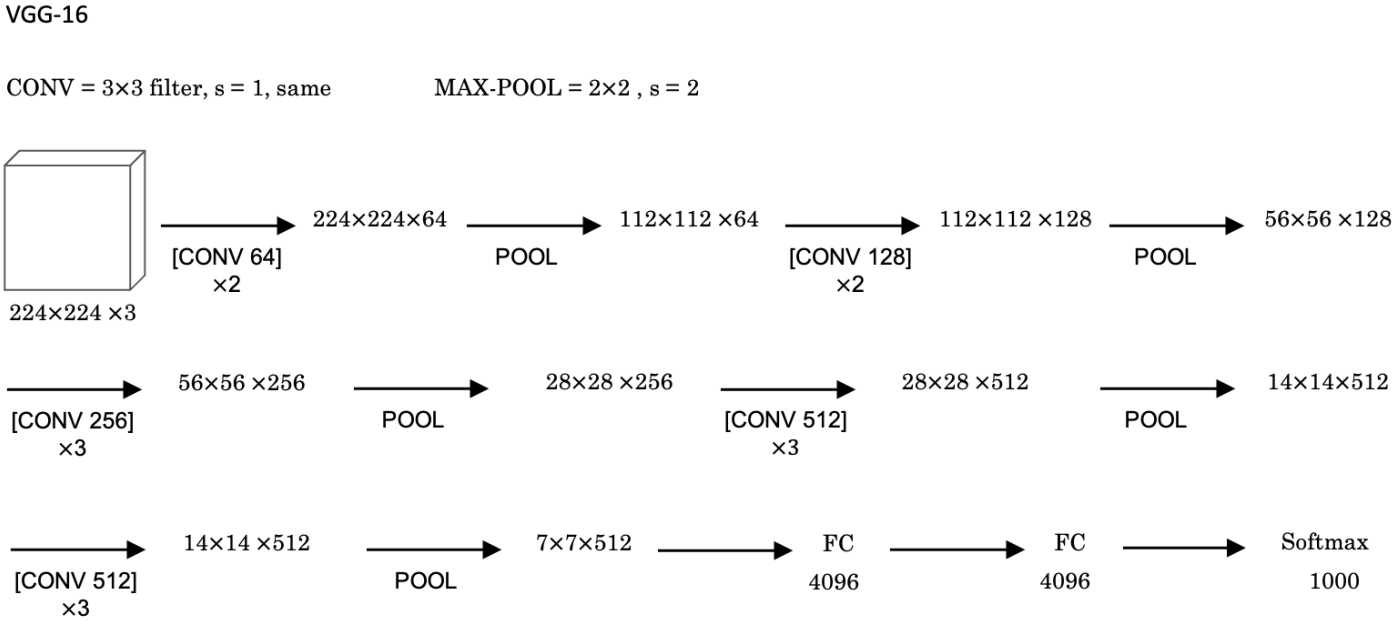
AlexNet



- AlexNet has a lot of similarities to LeNet (60,000 parameters), but it is much bigger (60 million parameters)
- Uses ReLU activation
- The paper had a complicated way of training on two GPUs since GPUs were still a little bit slower back then.
- The original AlexNet architecture had another set of a layer called local response normalization, which isn't really used much.
- Before AlexNet, deep learning was starting to gain traction in speech recognition and a few other areas, but it was really just paper that convinced a lot of the computer vision community to take a serious look at deep learning, to convince them that deep learning really works in computer vision.

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

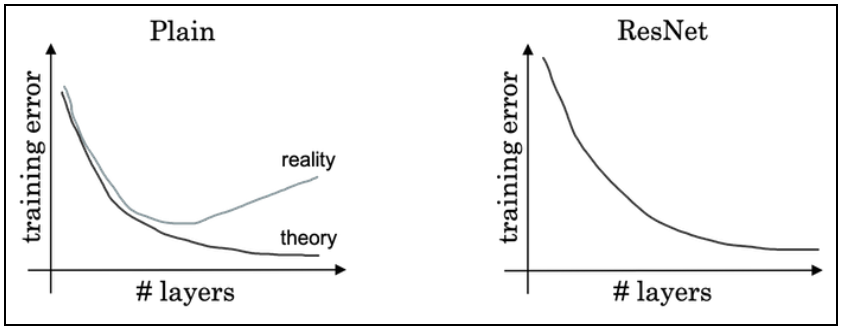
VGG-16



- Filters are always 3x3 with a stride of 1 and are always the same convolutions
- VGG-16 has 16 layers that have weights, total of about 138 million parameters, large even by modern standards
- It is the simplicity, or the uniformity, of the VGG-16 architecture that made it quite appealing
- There are a few conv-layers followed by a pooling layer which reduces the height and width by a factor of 2
- Doubling through every stack of conv-layers is a simple principle used to design the architecture of this network.
- The main downside is that you have to train a large number of parameters.

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

8.2.2. ResNets



- Deeper neural networks are more difficult to train. They present a residual learning framework to ease the training of networks that are substantially deeper than those used previously.
- When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. The paper addresses the degradation problem by introducing a deep residual learning framework. Instead of hoping each few stacked layers directly fit a desired underlying mapping, they explicitly let these layers fit a residual mapping.
- The paper authors show that: 1) Their extremely deep residual nets are easy to optimize, but the counterpart "plain" nets (that simply stack layers) exhibit higher training error when the depth increases; 2) Their deep residual nets can easily enjoy accuracy gains from greatly increased depth, producing results substantially better than previous networks.

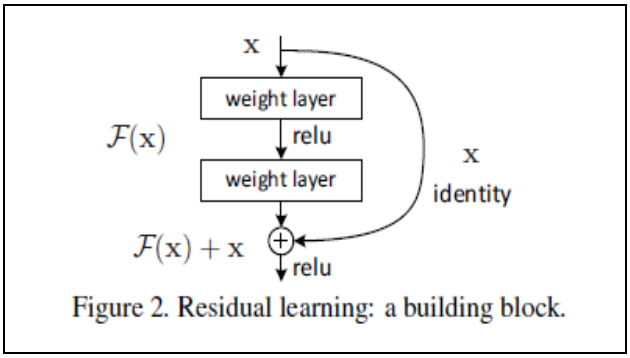
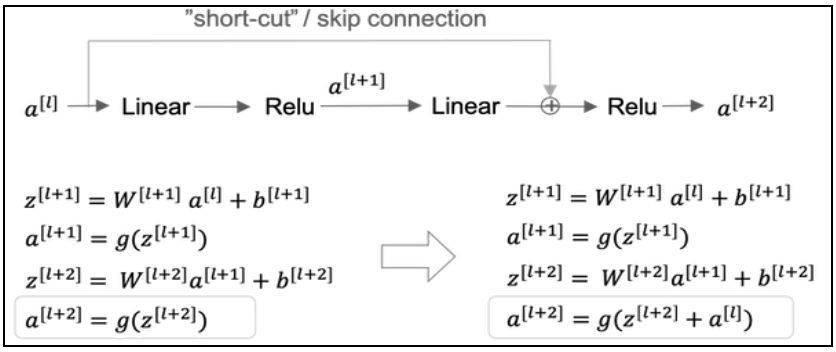


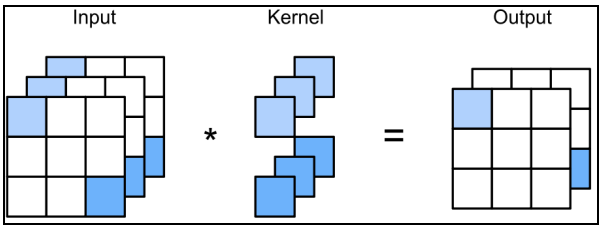
Figure 2. Residual learning: a building block.

Formally, denoting the desired underlying mapping as H(x), they let the stacked nonlinear layers fit another mapping of F(x):=H(x)-x. The original mapping H(x) is recast into F(x)+x. If the added layers can be constructed as identity mappings, a deeper model should have training error no greater than its shallower counterpart.
Paper: [Deep Residual Learning for Image Recognition]



8.2.3. Networks in Networks and 1x1 Convolutions

- At first, a 1×1 convolution does not seem to make much sense. After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously does not.
- Because the minimum window is used, the 1×1 convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the 1×1 convolution occurs on the channel dimension.
- The 1×1 convolutional layer is typically used to adjust the number of channels between network layers and to control model complexity.



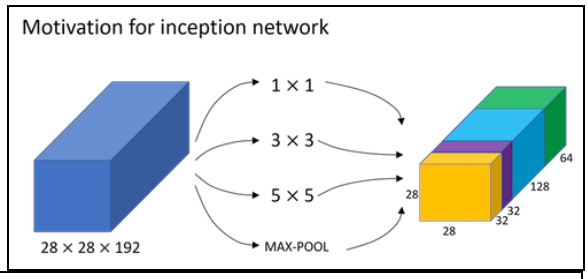
The 1×1 convolutional layer is equivalent to the fully-connected layer, when applied on a per pixel basis.

- You can take every pixel as an example with n_c[l] input values (channels) and the output layer has n_c[l+1] nodes. The kernel is just nothing but the weights.
- Thus the 1x1 convolutional layer requires n_c[l+1] x n_c[l] weights and the bias.

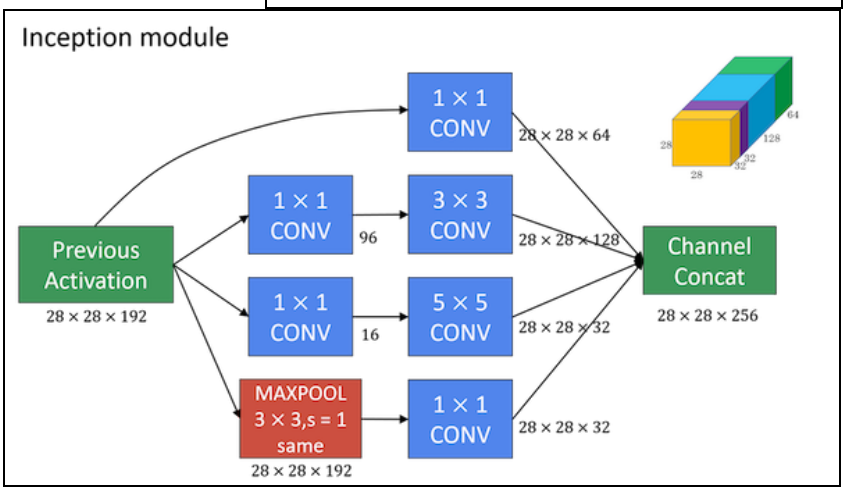
The 1x1 convolutional layer is actually doing something pretty non-trivial and adds non-linearity to your neural network and allows you to decrease or keep the same or if you want, increase the number of channels in your volumes.
Paper: [Network in Network]

8.2.4. Inception Network

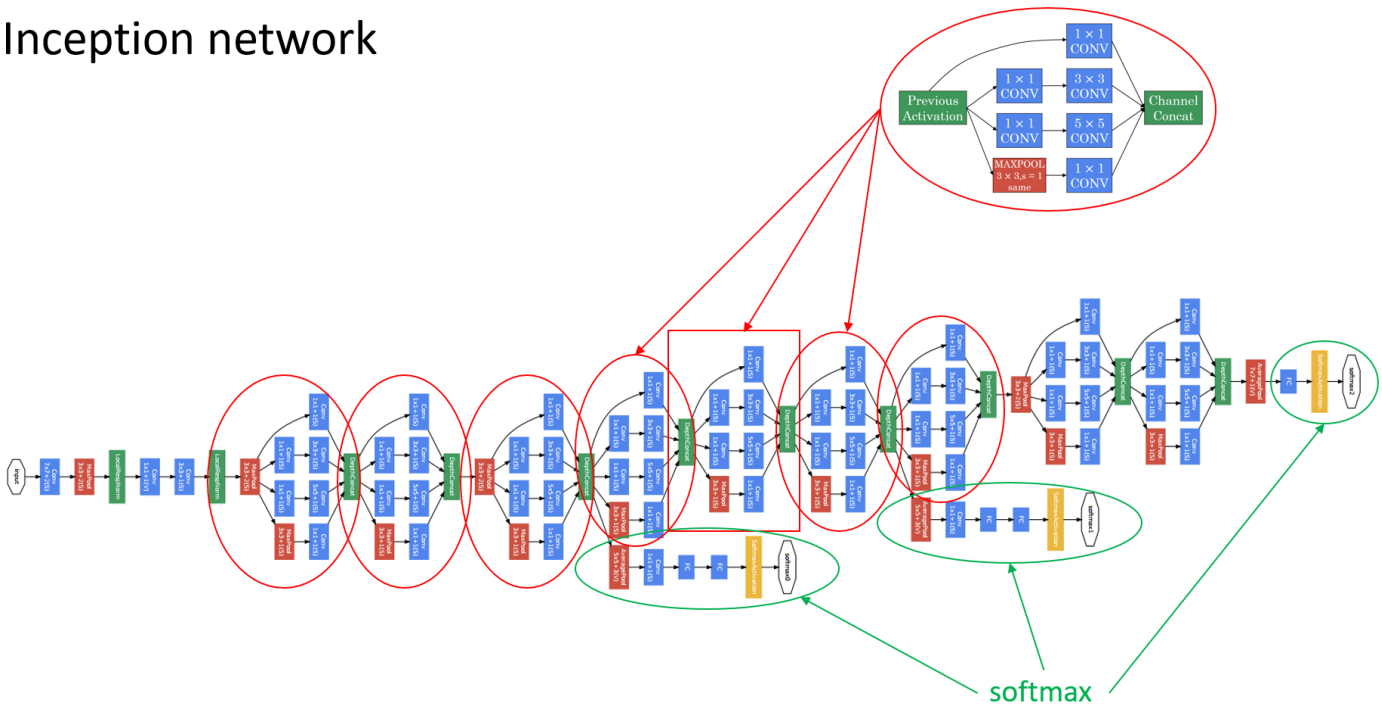
When designing a layer for a ConvNet, you might have to pick, do you want a 1 by 3 filter, or 3 by 3, or 5 by 5, or do you want a pooling layer? What the inception network does is it says, why shouldn't you do them all? And this makes the network architecture more complicated, but it also works remarkably well.



And the basic idea is that instead of you need to pick one of these filter sizes or pooling you want and commit to that, you can do them all and just concatenate all the outputs, and let the network learn whatever parameters it wants to use, whatever the combinations of these filter sizes it wants. Now it turns out that there is a problem with the inception layer as we've described it here, which is computational cost.



- In order to really concatenate all of these outputs at the end we are going to use the same type of padding for pooling.
- What the inception network does is more or less put a lot of these modules together.



Paper: [Going Deeper with Convolutions]

8.2.5. Practical advices for using ConvNets

Using Open-Source Implementation

- Starting with open-source implementations is a better way, or certainly a faster way to get started on a new project.
- One of the advantages of doing so also is that sometimes these networks take a long time to train, and someone else might have used multiple GPUs and a very large dataset to pretrain some of these networks. And that allows you to do transfer learning using these networks.

Transfer Learning

The computer vision research community has been pretty good at posting lots of data sets on the Internet so if you hear of things like ImageNet, or MS COCO, or PASCAL types of data sets, these are the names of different data sets that people have post online and a lot of computer researchers have trained their algorithms on.

- ImageNet: ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images.
- Microsoft COCO: COCO is a common object in context. The dataset contains 91 object types of 2.5 million labeled instances across 328,000 images.
- PASCAL: PASCAL-Context Dataset This dataset is a set of additional annotations for PASCAL VOC 2010. It goes beyond the original PASCAL semantic segmentation task by providing annotations for the whole scene. The statistics section has a full list of 400+ labels.

Sometimes these training takes several weeks and might take many GPUs and the fact that someone else has done this and gone through the painful high-performance search process, means that you can often download open source ways that took someone else many weeks or months to figure out and use that as a very good initialization for your own neural network.

- If you have a small dataset for your image classification problem, you can download some open source implementation of a neural network and download not just the code but also the weights. And then you get rid of the softmax layer and create your own softmax unit that outputs your classification labels.
- To do this, you just freeze the parameters which you don't want to train. A lot of popular learning frameworks support this mode of operation (i.e., set the trainable parameter to 0).

- Those early frozen layers are some fixed function that doesn't change. So one trick that could speed up training is that we just pre-compute that layer's activations and save them to disk. The advantage of the save-to-disk or the pre-compute method is that you don't need to recompute those activations everytime you take an epoch or take a path through a training set.
- If you have a larger label dataset one thing you could do is then freeze fewer layers. If you have a lot of data, in the extreme case, you could just use the downloaded weights as initialization so they would replace random initialization.

Data Augmentation

Having more data will help all computer vision tasks.

Some common data augmentation in computer vision:

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

Color shifting: Take different values of R, G and B and use them to distort the color channels. In practice, the values R, G and B are drawn from some probability distribution. This makes your learning algorithm more robust to changes in the colors of your images. One of the ways to implement color distortion uses an algorithm called PCA. The details of this are actually given in the AlexNet paper, and sometimes called PCA Color Augmentation.

Implementation tips:

A pretty common way of implementing data augmentation is to really have one thread, almost four threads, that is responsible for loading the data and implementing distortions, and then passing that to some other thread or some other process that then does the training.

- Often the data augmentation and training process can run in parallel.
- Similar to other parts of training a deep neural network, the data augmentation process also has a few hyperparameters, such as how much color shifting do you implement and what parameters you use for random cropping.

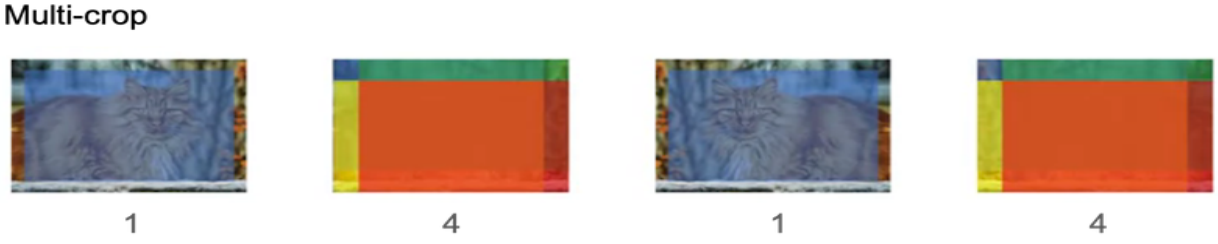
Tips for doing well on benchmarks/winning competitions:

(1) Ensembling

- Train several networks independently and average their outputs (not weights)
- That maybe gives you 1% or 2% better, which really helps win a competition
- To test on each image you might need to run an image through 3 to 15 different networks, so ensembling slows down your running time by a factor of 3 to 15
- So ensembling is one of those tips that people use doing well in benchmarks and for winning competitions.
- Almost never use in production to serve actual customers
- One big problem: need to keep all these different networks around, which takes up a lot more computer memory.

(2) Multi-crop at test time

- Run classifier on multiple versions of test images and average results
- Used much more for doing well on benchmarks than in actual production systems
- Keep just one network around, which doesn't suck up as much memory, but it still slows down run time quite a bit



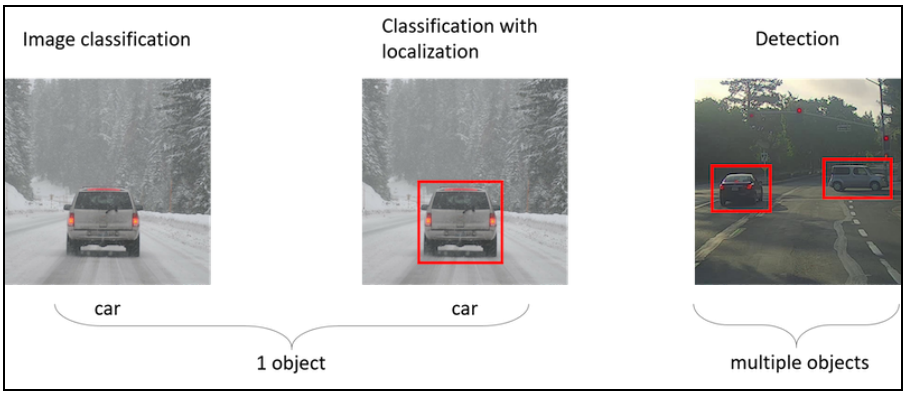
Use open source code:

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

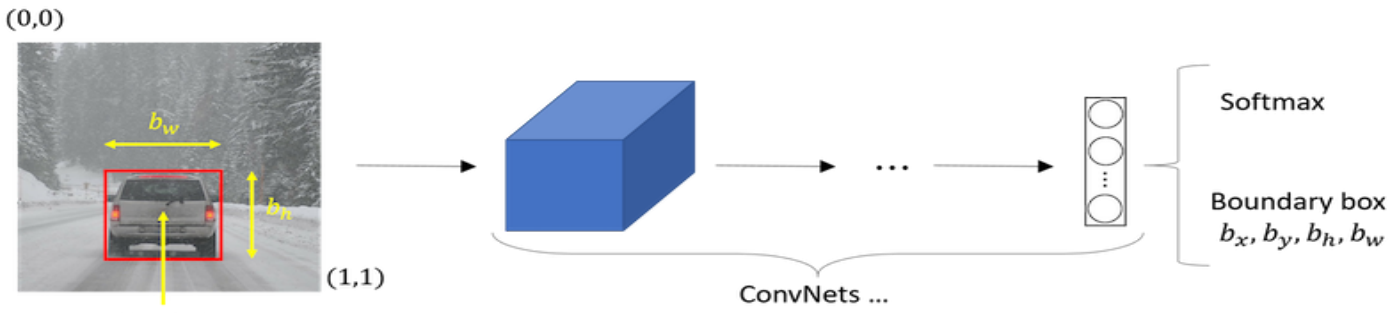
8.3. Object Detection

8.3.1. Detection Algorithms

- The classification and the classification of localization problems usually have one object.
- In the detection problem there can be multiple objects.
- The ideas you learn about image classification will be useful for classification with localization, and the ideas you learn for localization will be useful for detection.



Classification with localization



$b_x = 0.5$
$b_y = 0.7$
$b_h = 0.3$
$b_w = 0.4$

1 - pedestrian
2 - car
3 - motorcycle
4 - background

Defining the target label y

1 - pedestrian
2 - car
3 - motorcycle
4 - background

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

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

X=

$$\begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

$$\mathcal{L}(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + \cdots + (\hat{y}_8 - y_8)^2, & y_1 = 1 \\ (\hat{y}_1 - y_1)^2, & y_1 = 0 \end{cases}$$

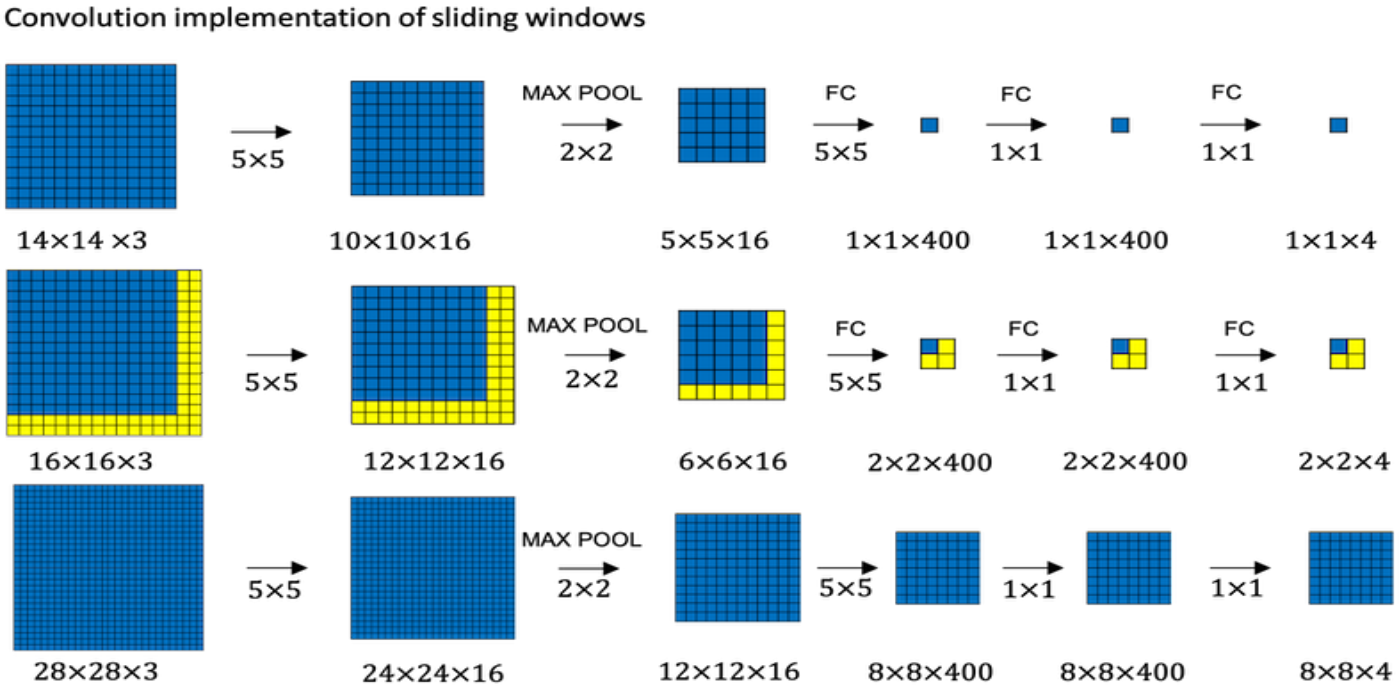
The squared error is used just to simplify the description here. In practice you could probably use a log-like feature loss for the c1, c2, c3 to the softmax output.

8.3.2. Landmark and Object Detection

In more general cases, you can have a neural network just output x and y coordinates of important points in the image, sometimes called landmarks. If you are interested in people pose detection, you could also define a few key positions like the midpoint of the chest, the left shoulder, left elbow, the wrist, and so on. The identity of landmark one must be consistent across different images like maybe landmark one is always this corner of the eye, landmark two is always this corner of the eye, landmark three, landmark four, and so on.
Disadvantage of sliding windows detection is computational cost. Unless you use a very fine granularity or a very small stride, you end up not able to localize the objects accurately within the image.

8.3.3. Convolutional Implementation of Sliding Windows

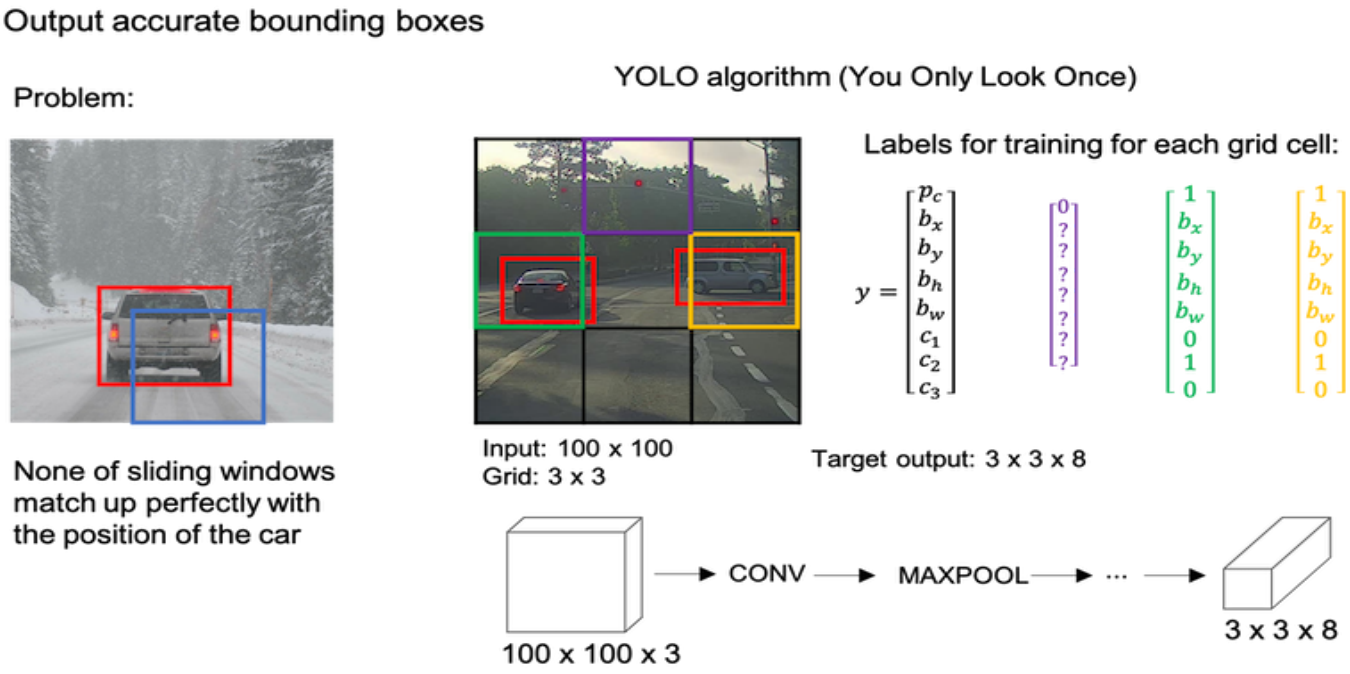
To build up towards the convolutional implementation of sliding windows let's first see how you can turn fully connected layers in a neural network into convolutional layers. What the convolutional implementation of sliding windows does is it allows four processes in the convnet to share a lot of computation. Instead of doing it sequentially, with the convolutional implementation you can implement the entire image, all maybe 28 by 28 and convolutionally make all the predictions at the same time.



8.3.4. YOLO algorithm

The convolutional implementation of sliding windows is more computationally efficient, but it still has a problem of not quite outputting the most accurate bounding boxes. The perfect bounding box isn't even quite square, it actually has a slightly wider rectangle or slightly horizontal aspect ratio.
The basic idea is you're going to take the image classification and localization algorithm and apply that to each of the nine grid cells of the image. If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.
The advantage of this algorithm is that the neural network outputs precise bounding boxes as follows.

- First, this allows in your network to output bounding boxes of any aspect ratio, as well as, output much more precise coordinates than are just dictated by the stride size of your sliding windows classifier.
- Second, this is a convolutional implementation and you're not implementing this algorithm nine times on the 3 by 3 grid or 361 times on a 19 by 19 grid.



Intersection Over Union

IoU is a measure of the overlap between two bounding boxes. If we use IoU in the output assessment step, then the higher the IoU the more accurate the bounding box. However IoU is a nice tool for the YOLO algorithm to discard redundant bounding boxes.

Non-max Suppression

One of the problems of Object Detection as you've learned about this so far, is that your algorithm may find multiple detections of the same objects. Rather than detecting an object just once, it might detect it multiple times. Non-max suppression is a way for you to make sure that your algorithm detects each object only once.

- It first takes the largest Pc with the probability of a detection.
- Then, the non-max suppression part is to get rid of any other ones with a high (defined by a threshold) IoU between the box chosen in the first step.

If you actually tried to detect three objects, say pedestrians, cars, and motorcycles, then the output vector will have three additional components. And it turns out, the right thing to do is to independently carry out non-max suppression three times, one on each of the output classes.

9. Sequence Models

9.1. Recurrent Neural Networks

Examples of sequence data: Speech recognition, Music generation, Sentiment classification, DNA sequence analysis, Machine translation, Video activity recognition, Named entity recognition

9.1.1. Notation

For a motivation, in the problem of Named Entity Recognition (NER), we have the following notation:

- x is the input sentence, such as: Harry Potter and Hermione Granger invented a new spell
- y is the output, in this case: 1 1 0 1 1 0 0 0 0
- x<t> denotes the word in the index t and y<t> is the corresponding output
- In the i-th input example, x(i)<t> is the t-th word and Tx(i) is the length of the i-th example
- Ty is the length of the output. In NER, we have Tx = Ty

Words representation introduced in this video is the One-Hot representation.

- First, you have a dictionary where words appear in a certain order
- Second, for a particular word, we create a new vector with 1 in position of the word in the dictionary and 0 everywhere else

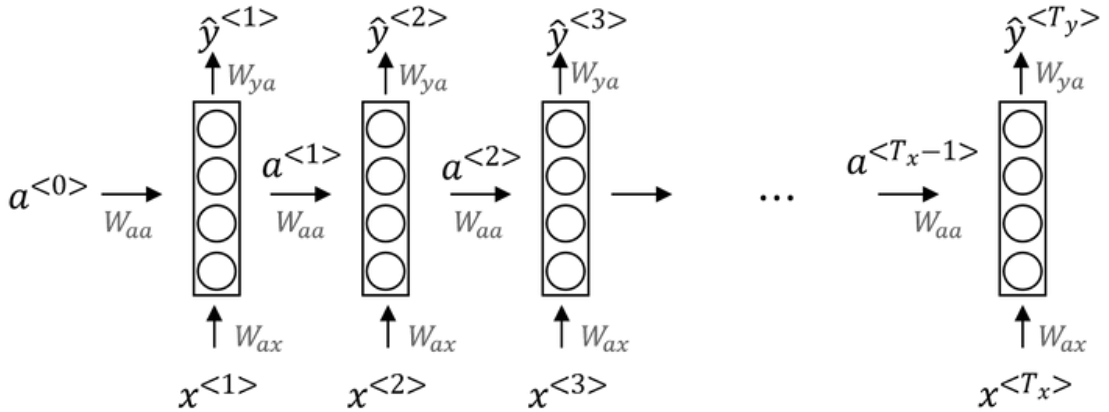
For a word not in your vocabulary, we need to create a new token or a new fake word called an unknown word denoted by <UNK>.

9.1.2. Recurrent Neural Network Model

Recurrent Neural Networks:

- At each time step, the recurrent neural network passes on as activation to the next time step for it to use.
- The recurrent neural network scans through the data from left to right. The parameters it uses for each time step are shared
- One limitation of unidirectional neural network architecture is that the prediction at a certain time uses inputs or uses information from the inputs earlier in the sequence but not information later in the sequence
  - He said, "Teddy Roosevelt was a great president."
  - He said, "Teddy bears are on sale!"
  - You can't tell the difference if you look only at the first three words.

Forward Propagation



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

$a^{<1>} = g_1(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$ Maybe $tanh$ or $relu$

$\hat{y}^{<1>} = g_2(W_{ya}a^{<1>} + b_y)$ Maybe $sigmoid$ for ner

$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$

$\hat{y}^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$

Instead of carrying around two parameter matrices Waa and Wax, we can simplify the notation by compressing them into just one parameter matrix Wa.

Simplified RNN notation

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

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

$\Rightarrow$

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

$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$

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

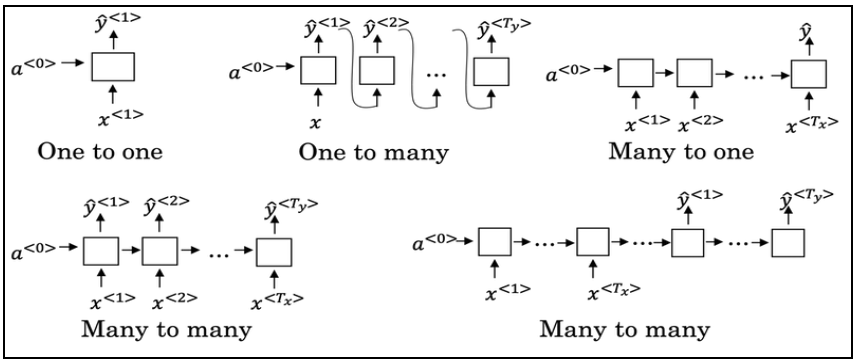
Backpropagation through time: In the backpropagation procedure the most significant message or the most significant recursive calculation is which goes from right to left, that is, backpropagation through time.

9.1.3. Different Types of RNNs

There are different types of RNN based on inputs and outputs:

- One to One (one input to one output)
- One to Many (one input to many output)
- Many to One (many input to one output)
- Many to Many (many input to many output)

See more details on RNN by Karpathy



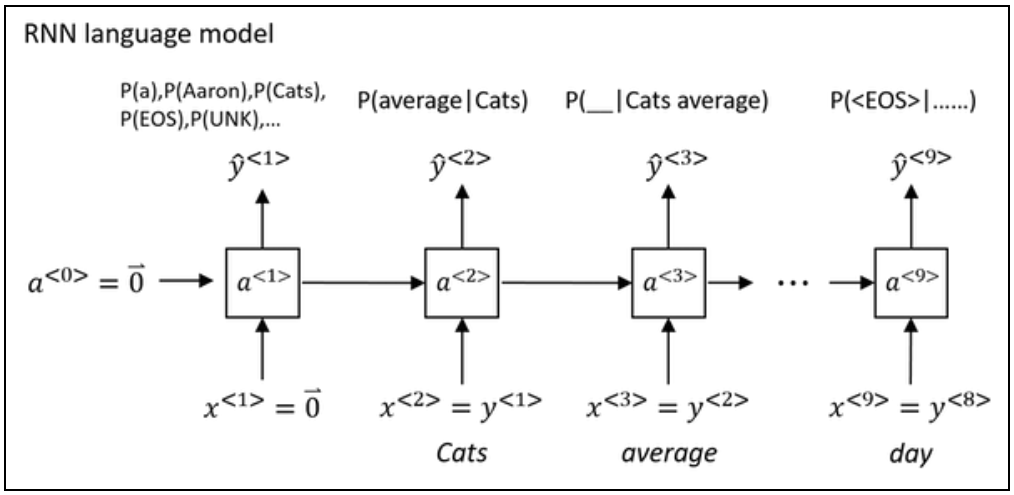
9.1.4. Language Model and Sequence Generation

So what a language model does is to tell you what is the probability of a particular sentence. For language model it will be useful to represent a sentence as output y rather than input x. So what the language model does is to estimate the probability of a particular sequence of words *P*(y<1>, y<2>, ..., y<T_y>).

How to build a language model?

Cats average 15 hours of sleep a day <EOS> Total 9 words in this sentence.

- You need Training set: Large corpus of english text
- The first thing you would do is to tokenize the sentence
- Map each of these words to one-hot vectors or indices in vocabulary
  - Maybe need to add extra token for end of sentence as <EOS> or unknown words as <UNK>
  - Omit the period. if you want to treat the period or other punctuation as explicit token, then you can add the period to you vocabulary as well
- Set the inputs x<t> = y<t-1>
- What a1 does is it will make a softmax prediction to try to figure out what is the probability of the first words y<1>. That is what is the probability of any word in the dictionary. Such as, what's the chance that the first word is Aaron?
- Until the end, it will predict the chance of <EOS>
- Define the cost function. The overall loss is just the sum over all time steps of the loss associated with the individual predictions



If you train this RNN on a large training set, we can do:

- Given an initial set of words, use the model to predict the chance of the next word
- Given a new sentence y<1>,y<2>,y<3>, use it to figure out the chance of this sentence: p(y<1>,y<2>,y<3>) = p(y<1>) * p(y<2>|y<1>) * p(y<3>|y<1>,y<2>)

Sampling novel sequences

After you train a sequence model, one way you can informally get a sense of what is learned is to have it sample novel sequences.

How to generate a randomly chosen sentence from your RNN language model:

- In the first time step, sample what is the first word you want your model to generate: randomly sample according to the softmax distribution
  - What the softmax distribution gives you is it tells the chance of the first word is 'a', the chance of the first word is 'Aaron', the chance of the first word is 'Zulu', or the chance of the first word referring to <UNK> or <EOS>. All these probabilities can form a vector
  - Take the vector and use np.random.choice to sample according to distribution defined by this vector probabilities. That lets you sample the first word
- In the second time step, remember in the last section, y<1> is expected as input. Here take ŷ<1> you just sampled and pass it as input to the second step. Then use np.random.choice to sample ŷ<2>. Repeat this process until you generate an <EOS> token
- If you want to make sure that your algorithm never generates <UNK>, just reject any sample that comes out as <UNK> and keep resampling from vocabulary until you get a word that's not <UNK>

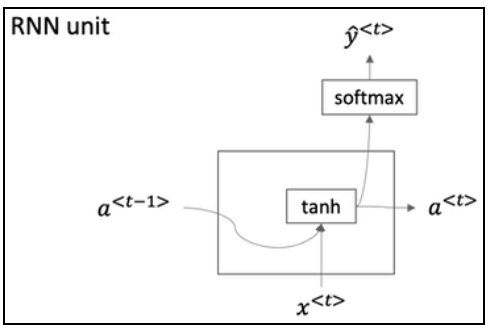
Character level language model:
If you build a character level language model rather than a word level language model, then your sequence y1, y2, y3, would be the individual characters in your training data, rather than the individual words in your training data. Using a character level language model has some pros and cons. As computers get faster there are more and more applications where people are, at least in some special cases, starting to look at more character level models.

- Advantages:
  - You don't have to worry about <UNK>
- Disadvantages:
  - The main disadvantage of the character level language model is that you end up with much longer sequences
  - And so character language models are not as good as word level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence
  - More computationally expensive to train

9.1.5. Vanishing Gradients with RNNs

- One of the problems with a basic RNN algorithm is that it runs into vanishing gradient problems.
- Language can have very long-term dependencies, for example:
  - The cat, which already ate a bunch of food that was delicious ..., was full.
  - The cats, which already ate a bunch of food that was delicious, and apples, and pears, ..., were full.
- The basic RNN we've seen so far is not very good at capturing very long-term dependencies. It's difficult for the output to be strongly influenced by an input that was very early in the sequence.
- When doing backprop, the gradients should not just decrease exponentially, they may also increase exponentially with the number of layers going through.
- Exploding gradients are easier to spot because the parameters just blow up and you might often see NaNs, or not a number, meaning results of a numerical overflow in your neural network computation.
  - One solution to that is apply gradient clipping: it is bigger than some threshold, re-scale some of your gradient vector so that it is not too big.
- Vanishing gradients is much harder to solve and it will be the subject of GRU or LSTM.

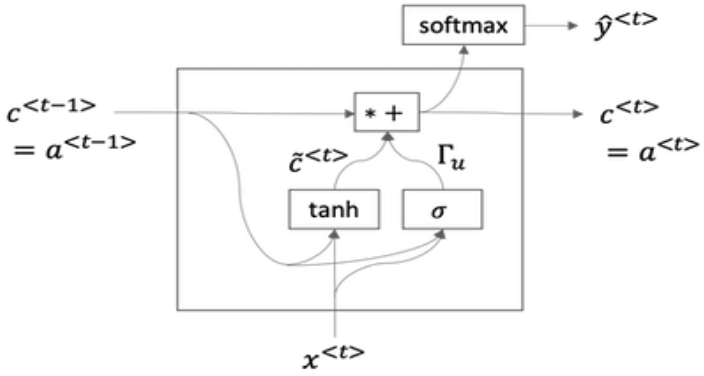
Gated Recurrent Unit (GRU)
Gate Recurrent Unit is one of the ideas that has enabled RNN to become much better at capturing very long range dependencies and has made RNN much more effective.
A visualization of the RNN unit of the hidden layer of the RNN in terms of a picture:



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

- The GRU unit is going to have a new variable called c, which stands for memory cell.
- c̃<t> is a candidate for replacing c<t>.
- For intuition, think of Γu as being either zero or one most of the time. In practice gamma won't be exactly 0 or 1.
- Because Γu can be so close to zero, can be 0.000001 or even smaller than that, it doesn't suffer from much of a vanishing gradient problem
- Because when Γu is so close to zero this becomes essentially c<t> = c<t-1> and the value of c is maintained pretty much exactly even across many many time-steps. So this can help significantly with the vanishing gradient problem and therefore allow a neural network to go on even very long range dependencies.
- In the full version of GRU, there is another gate Γr. You can think of r as standing for relevance. So this gate Γr tells you how relevant is c<t-1> to computing the next candidate for c<t>.

GRU



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

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

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

Full GRU:

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

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

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

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

Implementation tips:

- The asterisks are actually element-wise multiplication.
- If you have 100 dimensional or hidden activation value, then c<t>, c̃<t>, Γu would be the same dimension
  - If Γu is 100 dimensional vector, then it is a 100 dimensional vector of bits, the value is mostly zero and one
  - That tells you of this 100 dimensional memory cell which are the bits you want to update. What these element-wise multiplications do is it just tells the GRU unit which bits to update at every time-step. So you can choose to keep some bits constant while updating other bits
  - In practice gamma won't be exactly zero or one

Long Short Term Memory (LSTM)

Fancy explanation: [Understanding LSTM Network]

For the LSTM we will no longer have the case that a<t> is equal to c<t>.

And we're not using relevance gate Γr. Instead, LSTM has update, forget and output gates, Γu, Γf and Γo respectively.

GRU

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

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

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

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

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

LSTM

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

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

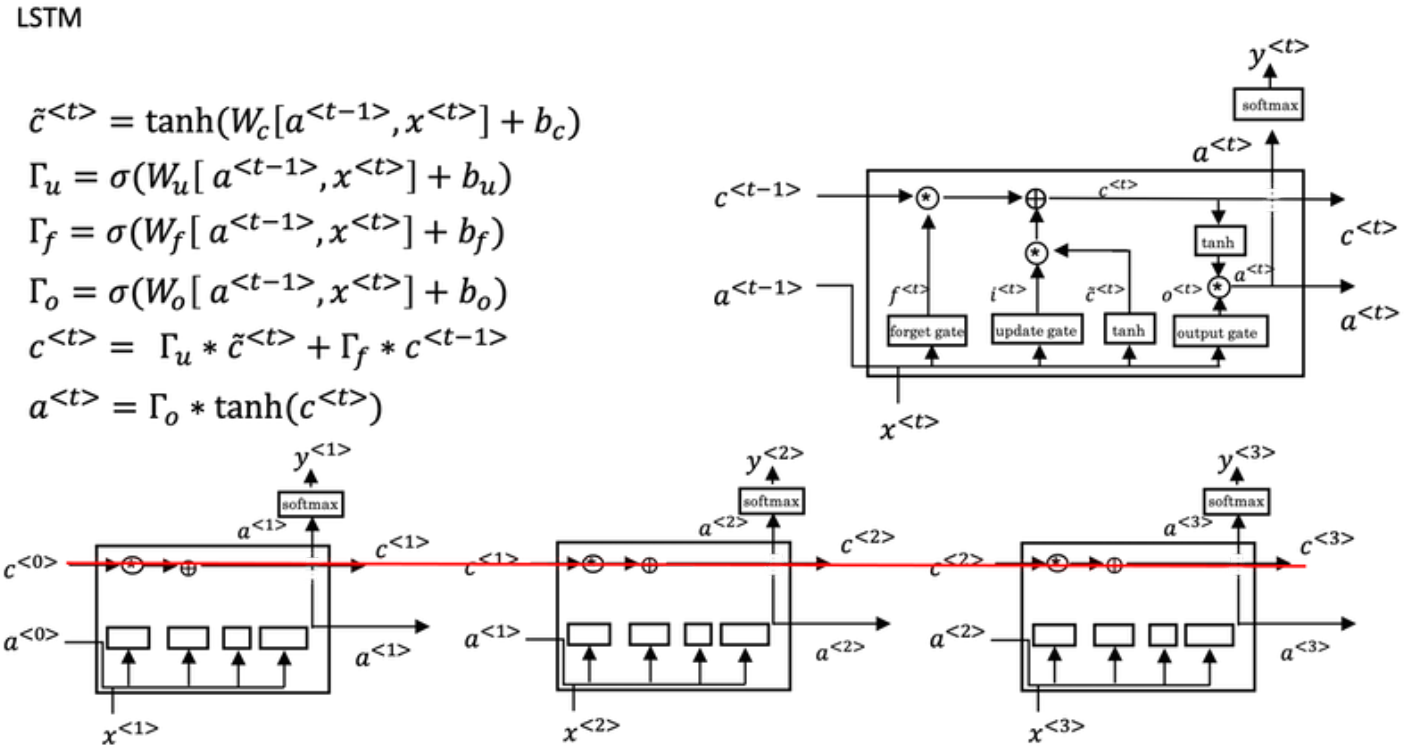
(forget) $$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

(output) $$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

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

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$

One cool thing about this you'll notice is that this red line at the top that shows how, so long as you set the forget and the update gate appropriately, it is relatively easy for the LSTM to have some value c<0> and have that be passed all the way to the right to have your, maybe, c<3> equals c<0>. And this is why the LSTM, as well as the GRU, is very good at memorizing certain values even for a long time, for certain real values stored in the memory cell even for many, many timesteps.



One common variation of LSTM:

- Peephole connection: instead of just having the gate values be dependent only on a<t-1>, x<t>, sometimes, people also sneak in there the values c<t-1> as well.

GRU vs. LSTM:

- The advantage of the GRU is that it's a simpler model and so it is actually easier to build a much bigger network, it only has two gates, so computationally, it runs a bit faster. So, it scales the building to somewhat bigger models.
- The LSTM is more powerful and more effective since it has three gates instead of two. If you want to pick one to use, LSTM has been the historically more proven choice. Most people today will still use the LSTM as the default first thing to try.

Implementation tips:

- forget gate Γf
  - The forget gate Γf<t> has the same dimensions as the previous cell state c<t-1>
  - This means that the two can be multiplied together, element-wise
  - Multiplying the tensors Γf<t> is like applying a mask over the previous cell state
  - If a single value in Γf<t> is 0 or close to 0, then the product is close to 0
  - This keeps the information stored in the corresponding unit in c<t-1> from being remembered for the next time step
  - Similarly, if one value is close to 1, the product is close to the original value in the previous cell state
  - The LSTM will keep the information from the corresponding unit of c<t-1>, to be used in the next time step
- candidate value c̃<t>
  - The candidate value is a tensor containing information from the current time step that may be stored in the current cell state c<t>
  - Which parts of the candidate value get passed on depends on the update gate
  - The candidate value is a tensor containing values that range from -1 to 1 (tanh function)

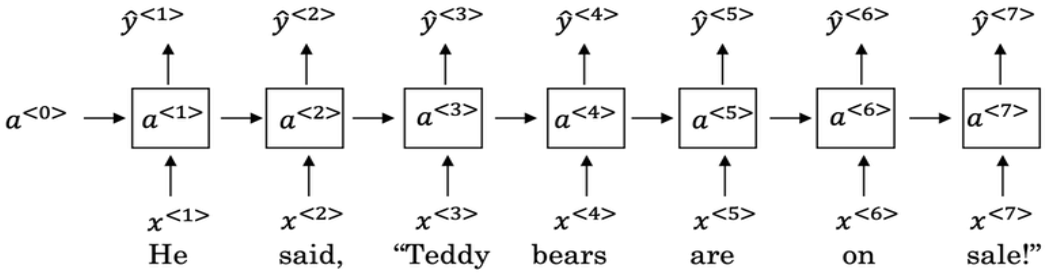
    - The tilde "~" is used to differentiate the candidate c̃<t> from the cell state c<t>
- update gate Γu
    - The update gate decides what parts of a "candidate" tensor c̃<t> are passed onto the cell state c<t>.
    - The update gate is a tensor containing values between 0 and 1
    - When a unit in the update gate is close to 1, it allows the value of the candidate c̃<t> to be passed onto the hidden state c<t>
    - When a unit in the update gate is close to 0, it prevents the corresponding value in the candidate from being passed onto the hidden state
- cell state c<t>
    - The cell state is the "memory" that gets passed onto future time steps
    - The new cell state c<t> is a combination of the previous cell state and the candidate value
- output gate Γo
    - The output gate decides what gets sent as the prediction (output) of the time step
    - The output gate is like the other gates. It contains values that range from 0 to 1
- hidden state a<t>
    - The hidden state gets passed to the LSTM cell's next time step
    - It is used to determine the three gates (Γf, Γu, Γo) of the next time step
    - The hidden state is also used for the prediction y<t>

9.1.6. Bidirectional RNN

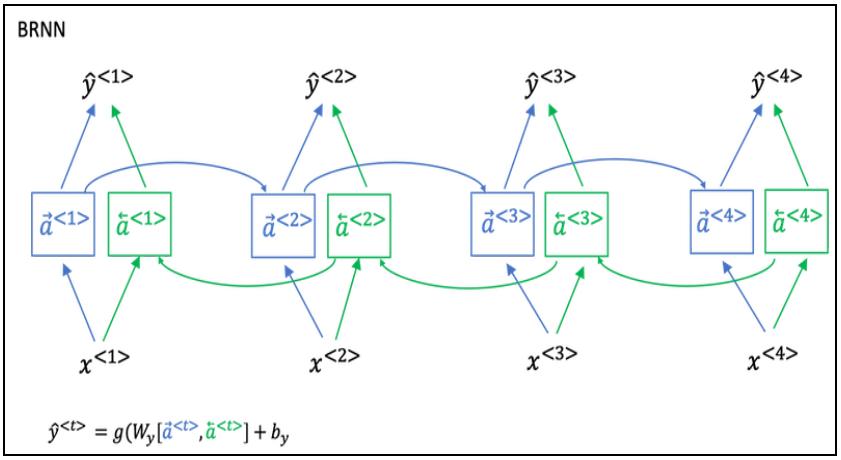
RNN on NER

He said, “Teddy bears are on sale!”

He said, “Teddy Roosevelt was a great President!”



- Bidirectional RNN lets you at a point in time take information from both earlier and later in the sequence.
- This network defines a Acyclic graph
- The forward prop has part of the computation going from left to right and part of computation going from right to left in this diagram.
- So information from x<1>, x<2>, x<3> are all taken into account with information from x<4> can flow through a backward four to a backward three to Y three. So this allows the prediction at time three to take as input both information from the past, as well as information from the present which goes into both the forward and the backward things at this step, as well as information from the future.
- Blocks can be not just the standard RNN block but they can also be GRU blocks or LSTM blocks. In fact, BRNN with LSTM units is commonly used in NLP problems.

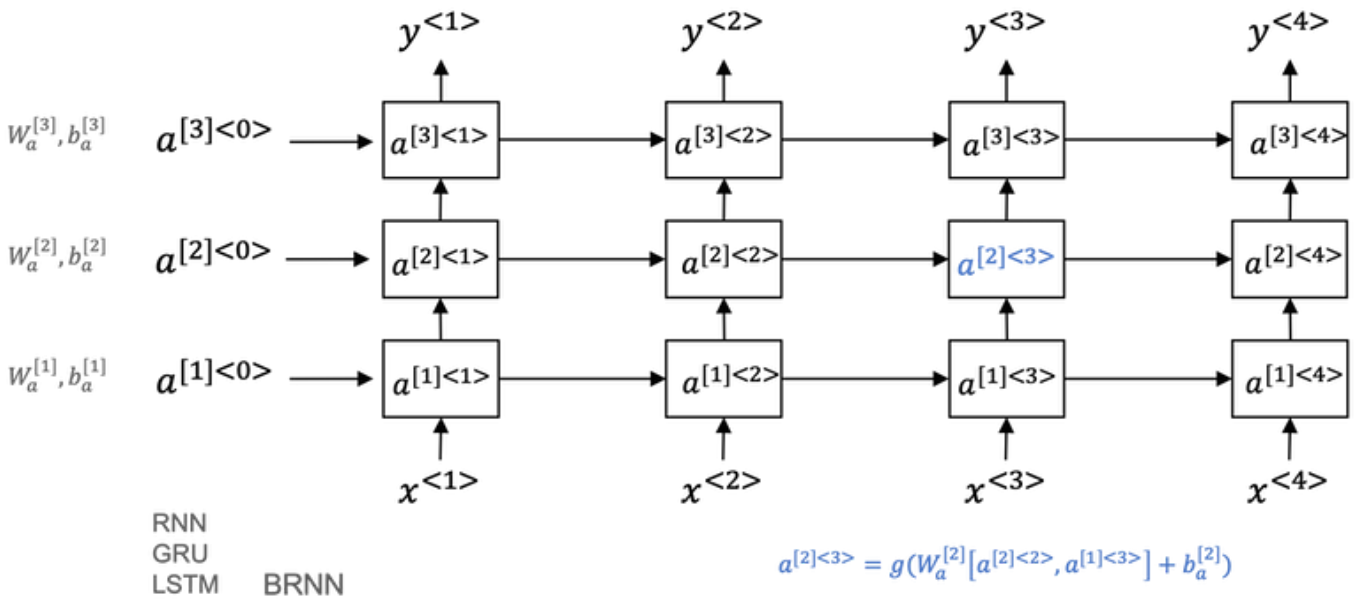


Disadvantage:
The disadvantage of the bidirectional RNN is that you do need the entire sequence of data before you can make predictions anywhere. So, for example, if you're building a speech recognition system, then the BRNN will let you take into account the entire speech utterance but if you use this straightforward implementation, you need to wait for the person to stop talking to get the entire utterance before you can actually process it and make a speech recognition prediction. For a real type speech recognition applications, they're somewhat more complex modules as well rather than just using the standard bidirectional RNN as you've seen here.

9.1.7. Deep RNNs

- For learning very complex functions sometimes it is useful to stack multiple layers of RNNs together to build even deeper versions of these models.
- The blocks don't just have to be standard RNN, the simple RNN model. They can also be GRU blocks or LSTM blocks.
- And you can also build deep versions of the bidirectional RNN.

Deep RNN example



9.2. NLP and Word Embeddings

9.2.1. Word Embeddings

- One of the weaknesses of one-hot representation is that it treats each word as a thing unto itself, and it doesn't allow an algorithm to easily generalize across words
  - Because the dot product between any two different one-hot vectors is zero
  - It doesn't know that apple and orange are much more similar than king and orange or queen and orange
- Instead we can learn a featurized representation (Embedding Vectors)
  - A lot of the features of apple and orange are actually the same, or take on very similar values; so, this increases the odds of the learning algorithm that has figured out that orange juice is a thing, to also quickly figure out that apple juice is a thing
  - The features won't have an easy to interpret interpretation like that component one is gender, component two is royal, component three is age and so on. What they're representing will be a bit harder to figure out
  - The featurized representations will allow an algorithm to quickly figure out that apple and orange are more similar than say, king and orange or queen and orange

9.2.2. Using word embeddings

- Learn word embeddings from large text corpus (1-100B words) (Or download pre-trained embedding online.)
- Transfer embedding to new task with smaller training set (say, 100k words)
- Optional: Continue to finetune the word embeddings with new data
  - In practice, you would do this only if task 2 has a pretty big data set
  - If your label data set for step 2 is quite small, do not bother to continue to fine tune the word embeddings

Word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set.

Word embeddings is useful for:

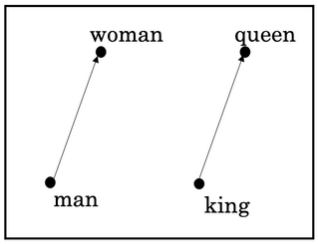
- Named entity recognition
- Text summarization
- Co-reference
- Parsing

Less useful for:

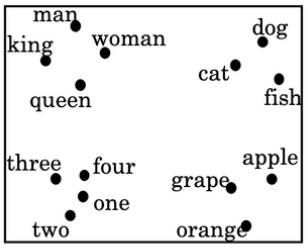
- Language modeling
- Machine translation

9.2.3. Properties of word embeddings

- Word embeddings can be used for analogy reasoning, which can help convey a sense of what word embeddings are doing even though analogy reasoning is not by itself the most important NLP application
- man --> woman vs. king --> queen: eman - ewoman ≈ eking - equeen
- To carry out an analogy reasoning, man is to woman as king is to what?
- To find a word so that eman - ewoman ≈ eking - e?
  - Find word w: argmax sim(ew, eking-eman+ewoman)
  - We can use cosine similarity to calculate this similarity
  - Refer to paper by Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig.
- t-SNE (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique, it takes 300-D data, and it maps it in a very non-linear way to a 2D space. And so the mapping that t-SNE learns, this is a very complicated and very non-linear mapping. So after the t-SNE mapping, you should not expect these types of parallelogram relationships, like the one we saw on the left, to hold true. And many of the parallelogram analogy relationships will be broken by t-SNE



300 D



t-SNE

Embedding matrix

- When you implement an algorithm to learn a word embedding, what you end up learning is an embedding matrix
- Our goal will be to learn an embedding matrix E by initializing E randomly and then learning all the parameters of matrix
- E times the one-hot vector gives you the embedding vector
- Use specialized function to look up an embedding

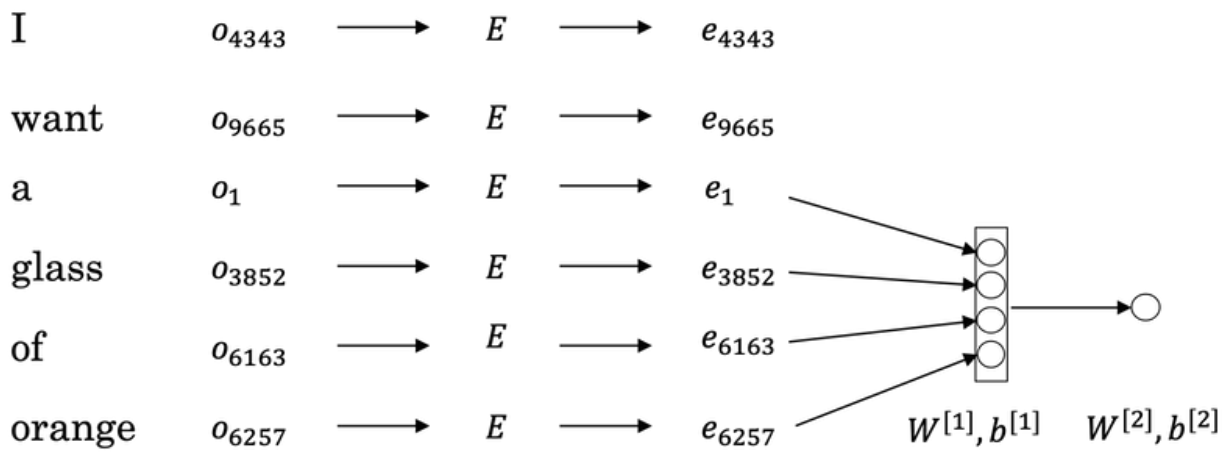
9.2.4. Learning Word Embeddings: Word2vec & GloVe

Learning Word Embeddings

- In the history of deep learning as applied to learning word embeddings, people actually started off with relatively complex algorithms. And then over time, researchers discovered they can use simpler and simpler and simpler algorithms and still get very good results especially for a large dataset
- More complex algorithm: a neural language model paper, by Yoshua Bengio, Rejean Ducharme, Pascals Vincent, and Christian Jauvin: A Neural Probabilistic Language Model

Let's start to build a neural network to predict the next word in the sequence below.

`I want a glass of orange _____`



If we have a fixed historical window of 4 words (4 is a hyperparameter), then we take the four embedding vectors and stack them together, and feed them into a neural network, and then feed this neural network output to a softmax, and the softmax classifies among the 10,000 possible outputs in the vocab for the final word we're trying to predict. These two layers have their own parameters W1,b1 and W2, b2. This is one of the earlier and pretty successful algorithms for learning word embeddings.

Word2Vec

Paper: Efficient Estimation of Word Representations in Vector Space by Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean.

The Skip-Gram model:

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

- In the skip-gram model, we're going to come up with a few context to target errors to create our supervised learning problem; So rather than having the context be always the last four words or the last end words immediately before the target word, what we are going to do is, randomly pick a word to be the context word.
- Randomly pick another word within some window e.g., ±5 words or ±10 words of the context word and we choose that to be target word
  - maybe by chance pick juice to be a target word, that just one word later
  - maybe glass, two words before
  - maybe my
- Set up a supervised learning problem where given the context word, that predicts what is a randomly chosen word within ±10 or ±5 word window of the input context word
- The goal of setting up this supervised learning problem isn't to do well on the supervised learning problem per se. It is that we want to use this learning problem to learn good word embeddings.

Context	Target
orange	juice
orange	glass
orange	my

Model details:

- Context c: orange and target t: juice.
- oc ---> E ---> ec ---> O(softmax) ---> ŷ. This is the little neural network with basically looking up the embedding and then just a softmax unit.
- Softmax: p(t|c), θt: parameter associated with output t. (bias term is omitted)
- Loss: L(ŷ,y) = -sum(yi log ŷi)
- This is called the skip-gram model because it's taking as input one word like orange and then trying to predict some words skipping a few words from the left or the right side.

Model problem:

- Computational speed: in the softmax step, every time evaluating the probability, you need to carry out a sum over all 10,000, maybe even larger 1,000,000, words in your vocabulary

Hierarchical softmax classifier:

- Hierarchical softmax classifier is one of a few solutions to the computational problem
  - Instead of trying to categorize something into all 10,000 categories on one go, imagine if you have one classifier, it tells you is the target word in the first 5,000 words in the vocabulary, or is in the second 5,000 words in the vocabulary, until eventually you get down to classify exactly what word it is, so that the leaf of this tree
  - The main advantage is that instead of evaluating W output nodes in the neural network to obtain the probability distribution, it is needed to evaluate only about log2(W) nodes
  - The hierarchical softmax classifier doesn't use a perfectly balanced tree or perfectly symmetric tree. The hierarchical softmax classifier can be developed so that the common words tend to be on top, whereas the less common words like durian can be buried much deeper in the tree

How to sample context c:

- One thing you could do is just sample uniformly, at random, from your training corpus
  - When we do that, you find that there are some words like the, of, a, and, to and so on that appear frequently
  - In practice the distribution of words p(c) isn't taken just entirely uniformly at random for the training set purpose, but instead there are different heuristics that you could use in order to balance out something from the common words together with the less common words

CBOW:

The other version of the Word2Vec model is CBOW, the continuous bag of words model, which takes the surrounding contexts from the middle word, and uses the surrounding words to try to predict the middle word. And the algorithm also works, which also has some advantages and disadvantages.

Negative Sampling

Paper: Distributed Representations of Words and Phrases and their Compositionality by Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean.

Negative sampling is a modified learning problem to do something similar to the Skip-Gram model with a much more efficient learning algorithm.

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

- To create a new supervised learning problem: given a pair of words like orange, juice, we're going to predict whether it is a context-target pair or not
- First, generate a positive example. Sample a context word, like orange and a target word, juice, associate them with a label of 1
- Then generate negative examples. Take orange and pick another random word from the dictionary for k times
  - Choose large values of k for smaller data sets, like 5 to 20, and smaller k for large data sets, like 2 to 5
  - In this example, k=4. x=(context, word), y=target
- Compared to the original Skip-Gram model: instead of training all 10,000 of them on every iteration which is very expensive, we're only going to train five, or k+1 of them. k+1 binary classification problems are relatively cheap to do rather than updating a 10,000 weights of softmax classifier
- How to choose the negative examples?
  - Sample it according to the empirical frequency of words in your corpus. The problem is you end up with a very high representation of words like 'the', 'of', 'and', and so on.
  - Other extreme method would use p(w)=1/|V| to sample the negative examples uniformly at random. This is also very non-representative of the distribution of English words.

context	word	target?
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

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

- The paper chooses a method somewhere in-between: f(w_i) is the observed frequency of word w_i

GloVe Word Vectors

Paper: GloVe: Global Vectors for Word Representation

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

- X_{ij}: #times j appear in the context of i. (Think X_{ij} as X_{ct})
  - $X_{ij} = X_{ji}$
  - If the context is always the word immediately before the target word, then Xij is not symmetric
- For the GloVe algorithm, define context and target as whether or not the two words appear in close proximity, say within ±10 words of each other. So, Xij is a count that captures how often do words i and j appear with each other or close to each other
- Model: $minimize \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_j + b_j' - logX_{ij})^2$
  - $\theta_i^T e_j$ plays the role of $\theta_i^T e_c$ in the previous sections.
  - We just want to learn vectors, so that their end product is a good predictor for how often the two words occur together.
  - There are various heuristics for choosing this weighting function f that neither gives these words too much weight nor gives the infrequent words too little weight.
    - f(X_{ij}) = 0 if X_{ij} = 0 to make sure 0log0=0
  - One way to train the algorithm is to initialize theta and e both uniformly random, run gradient descent to minimize its objective, and then when you're done for every word, to then take the average.
    - For a given words w, you can have efinal to be equal to the embedding that was trained through this gradient descent procedure, plus theta trained through this gradient descent procedure divided by two, because theta and e in this particular formulation play symmetric roles unlike the earlier models we saw in the previous videos, where theta and e actually play different roles and couldn't just be averaged like that

Conclusion:

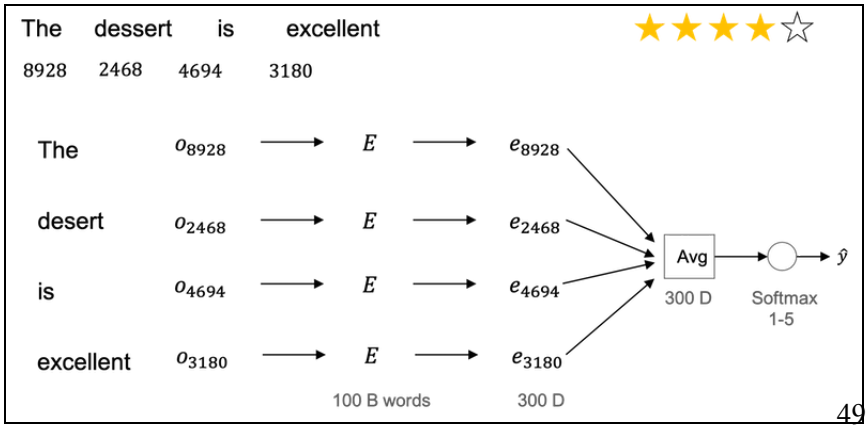
- The way that the inventors ended up with this algorithm was, they were building on the history of much more complicated algorithms like the newer language model, and then later, there came Word2Vec skip-gram model
- But when you learn a word embedding using one of the algorithms that we've seen, such as the GloVe algorithm that we just saw on the previous slide, what happens is, you cannot guarantee that the individual components of the embeddings are interpretable
- Despite this arbitrary linear transformation of the features, you end up learning the parallelogram map for figure analogies that works

9.2.5. Applications using Word Embeddings

Sentiment Classification

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

- One of the challenges of sentiment classification is you might not have a huge labeled dataset
- If this was trained on a very large data set, like a 100 billion words, then this allows you to take a lot of knowledge even from infrequent words and apply them to your problem

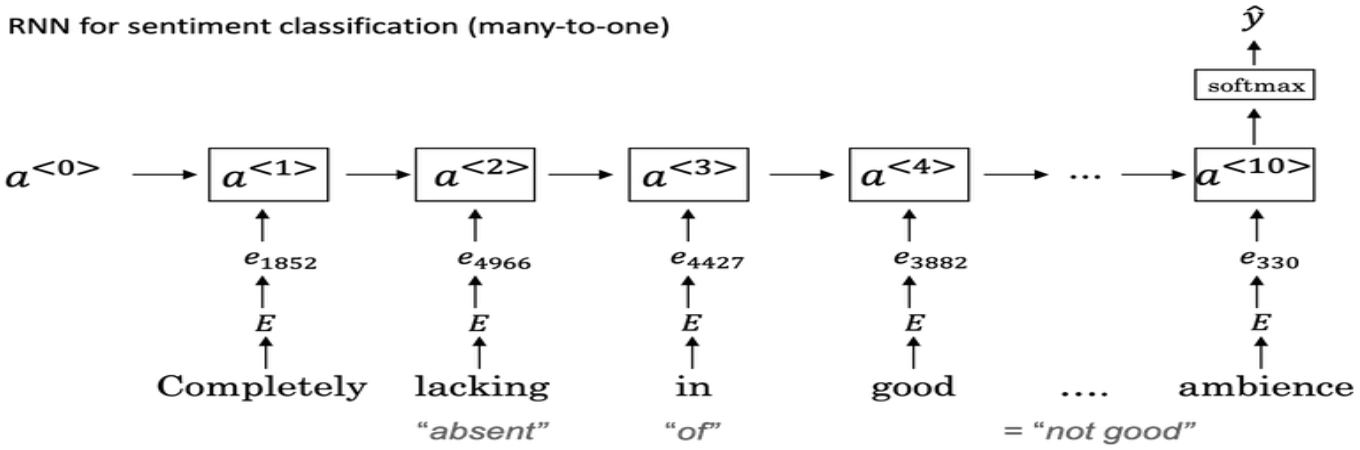


- Notice that by using the average operation here, this particular algorithm works for reviews that are short or long because even if a review that is 100 words long, you can just sum or average all the feature vectors for all hundred words and so that gives you a representation, a 300-dimensional feature representation, that you can then pass into your sentiment classifier
- One of the problems with this algorithm is it ignores word order
  - "Completely lacking in good taste, good service, and good ambiance"
  - This is a very negative review. But the word good appears a lot

A more sophisticated model:

Instead of just summing all of your word embeddings, you can instead use a RNN for sentiment classification.

- In the graph, the one-hot vector representation is skipped
- This is an example of a many-to-one RNN architecture



Debiasing Word Embeddings

Paper: [Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings]

Word embeddings may have bias problems such as gender bias, ethnicity bias and so on. As word embeddings can learn analogies like man is to woman like king to queen. The paper shows that a learned word embedding might output:

`Man: Computer_Programmer as Woman: Homemaker`

Learning algorithms are making very important decisions and so I think it's important that we try to change learning algorithms to diminish as much as is possible, or, ideally, eliminate these types of undesirable biases.

1. Identify bias direction
   - The first thing we're going to do is to identify the direction corresponding to a particular bias we want to reduce or eliminate
   - And take a few of these differences and basically average them. And this will allow you to figure out in this case that what looks like this direction is the gender direction, or the bias direction. Suppose we have a 50-dimensional word embedding
     - $g_1 = e_{she} - e_{he}$; $g_2 = e_{girl} - e_{boy}$; $g_3 = e_{mother} - e_{father}$; $g_4 = e_{woman} - e_{man}$
   - g = g1 + g2 + g3 + g4 + ... for gender vector.
   - Then we have
     - cosine_similarity(sophie, g)) = 0.318687898594
     - cosine_similarity(john, g)) = -0.23163356146
     - to see male names tend to have positive similarity with gender vectors whereas female names tend to have a negative similarity, this is acceptable
   - But we also have
     - cosine_similarity(computer, g)) = -0.103303588739
     - cosine_similarity(singer, g)) = 0.185005181365
     - It is astonishing how these results reflect certain unhealthy gender stereotypes

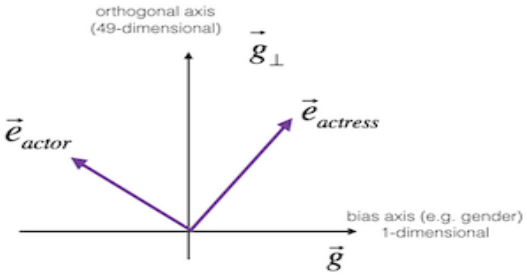
- The bias direction can be higher than 1-dimensional. Rather than taking an average, SVD (singular value decomposition) and PCA might help

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

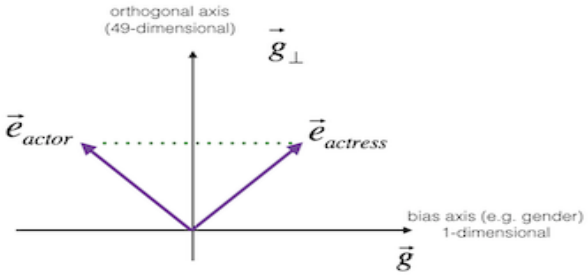
$$e^{bias\_component} = \frac{e \cdot g}{\|g\|_2^2} * g$$

$$e^{debiased} = e - e^{bias\_component}$$

3. Equalize pairs
   - Finally we'd like to make sure that words like grandmother and grandfather are both exactly the same similarity, or exactly the same distance, from words that should be gender neutral, such as babysitter or such as doctor
   - The key idea behind equalization is to make sure that a particular pair of words are equi-distant from the 49-dimensional g⊥



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



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

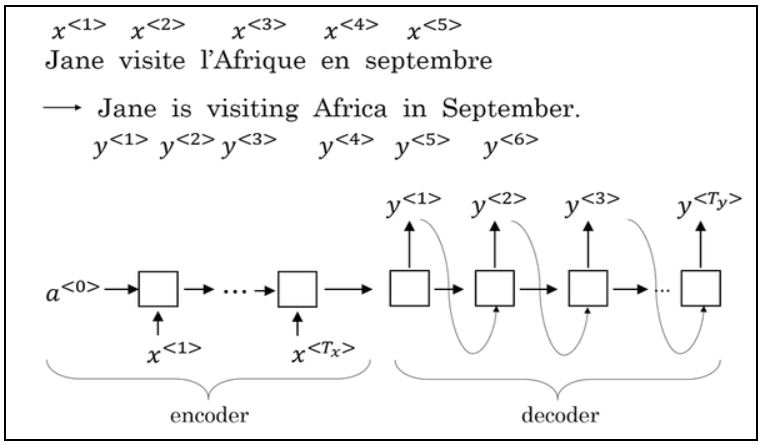
9.3. Sequence Models & Attention Mechanism

9.3.1. Basic Sequence-to-Sequence Models

In this section, we explore sequence-to-sequence models, which are useful for everything from machine translation to speech recognition.

Machine Translation

- Papers:
  - Sequence to Sequence Learning with Neural Networks by Ilya Sutskever, Oriol Vinyals, Quoc V. Le
  - Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation by Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio
- Input a French sentence: Jane visite l'Afrique en septembre, we want to translate it to the English sentence: Jane is visiting Africa in September
- First create a encoder network built as a RNN, and this could be a GRU and LSTM, feed in the input French words one word at a time, the RNN then offers a vector that represents the input sentence
- Then build a decoder network which takes as input the encoding and then can be trained to output the



translation one word at a time until eventually it outputs the end of the sequence

- The model simply uses an encoder network to find an encoding of the input French sentence and then use a decoder network to then generate the corresponding English translation
- In developing a machine translation system, one of the things you need to do is come up with an algorithm that can actually find the value of y that maximizes p(y<1>,...,y<T_y>|x<1>,...,x<T_x>). The most common algorithm for doing this is called **beam search**
  - In the example of the French sentence, "Jane, visite l'Afrique en Septembre"
  - Step 1: Probability of the first word of the English translation
    - Setting beam width B = 3, will choose the most likely three possibilities for the first words in the English outputs and stores away in computer memory to try out these three words
    - Run the input French sentence through the encoder network and then this first step will decode the network with a softmax output overall 10,000 possibilities (if we have a vocabulary of 10,000 words). Then you would take those 10,000 possible outputs p(y<1>|x) and keep in memory which were the top three
    - For example, after this step, we have three words as: in, Jane, September
  - Step 2: consider the next word
    - Find the pair of the first and second words that is most likely it's not just a second where is most likely. By the rules of conditional probability, it's p(y<1>,y<2>|x) = p(y<1>|x) * p(y<2>|x,y<1>)
    - After this step, in september, jane is, jane visit is left, and notice that September has been rejected as a candidate for the first word
    - Because beam width is equal to 3, every step instantiate three copies of the network to evaluate these partial sentence fragments and the output
  - Refinements to Beam Search
    - Length normalization:
      - Beam search is to maximize the probability:

$$\arg\max_y p(y^{<1>}, ..., y^{<T_y>}|x) = \arg\max_y \prod_{t=1}^{T_y} p(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})$$

      - Multiplying a lot of numbers less than 1 will result in a very tiny number, which can result in numerical underflow, so instead we maximizing a log version:

$$\arg\max_y \sum_{t=1}^{T_y} \log p(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})$$

      - If you have a very long sentence, the probability of that sentence is going to be low, because you're multiplying many terms less than 1 and so the objective function (the original version as well as the log version) has an undesirable effect, that maybe it unnaturally tends to prefer very short translations
      - A normalized log-likelihood objective:

$$\arg\max_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log p(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})$$

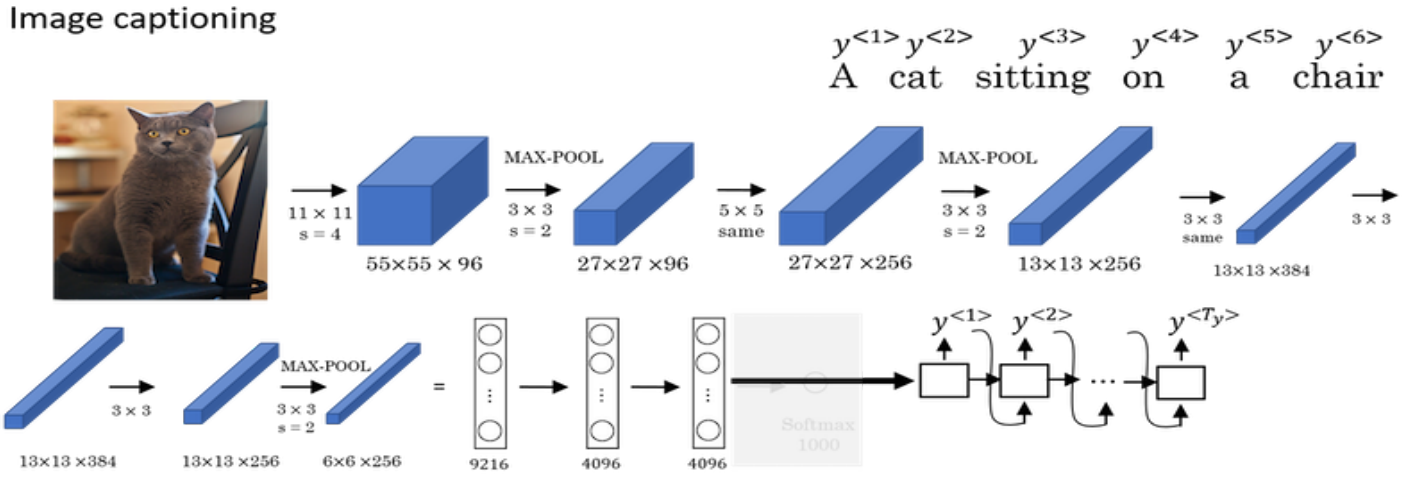
      - α is another hyperparameter, α=0 no normalizing, α=1 full normalization
    - How to choose beam width B?
      - If beam width is large: considers a lot of possibilities, so better result and consuming a lot of different options, so slower and memory requirements higher
      - If beam width is small: worse result but faster, memory requirements lower
      - Choice of beam width is application dependent and domain dependent: when B gets very large there is often diminishing returns; B=10 is common in a production system, whereas B=100 is uncommon
  - Error analysis in beam search
    - Beam search is an approximate search algorithm, also called a heuristic search algorithm, so it doesn't always output the most likely sentence

- In order to know whether it is the beam search algorithm that's causing problems and worth spending time on, or whether it might be the RNN model that's causing problems and worth spending time on, we need to do error analysis with beam search
- Getting more training data or increasing the beam width might not get you to the level of performance you want
- You should break the problem down and figure out what's actually a good use of your time

9.3.2. Image Captioning

This architecture is very similar to the one of machine translation.

- Paper: Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN) by Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, Alan Yuille.
- In the AlexNet architecture, if we get rid of this final Softmax unit, the pre-trained AlexNet can give you a 4096-dimensional feature vector of which to represent this picture of a cat. And so this pre-trained network can be the encoder network for the image and you now have a 4096-dimensional vector that represents the image. You can then take this and feed it to an RNN, whose job it is to generate the caption one word at a time.



9.3.3. Bleu Score

Paper: BLEU: a Method for Automatic Evaluation of Machine Translation by Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu.

BLEU stands for bilingual evaluation understudy.

- The reason the BLEU score was revolutionary for machine translation was because it gave a pretty good, not perfect, single real number evaluation metric, so that accelerated the progress of the entire field of machine translation
- The intuition behind the BLEU score is we're going to look at the machine generated output and see if the types of words it generates appear in at least one of the human generated references and so these human generated references would be provided as part of the dev set or as part of the test set
  - One way to measure how good the machine translation output is to look at each of the words in the output and see if it appears in the references
  - An extreme example:
    - French: Le chat est sur le tapis
    - Reference 1: The cat is on the mat
    - Reference 2: There is a cat on the mat
    - MT output: the the the the the the the
    - Precision: 7/7. This is not a particularly useful measure because it seems to imply that this MT output has very high precision

    - Instead, what we're going to use is a modified precision measure in which we will give each word credit only up to the maximum number of times it appears in the reference sentences
        - Modified precision: 2/7. The numerator is the count of the number of times the word, the, appears. We take a max, we clip this count, at 2
- In the BLEU score, you don't want to just look at isolated words. You may want to look at pairs of words as well. Let's define a portion of the BLEU score on bigrams
    - MT output: The cat the cat on the mat
    - Modified bigram precision: 4/6
- Generally, Bleu score on n-grams is defined as:

$$p_n = \frac{\sum_{\text{ngram}\in\hat{y}} count_{clip}(\text{ngram})}{\sum_{\text{ngram}\in\hat{y}} count(\text{ngram})}$$

- Combined Bleu score = $\text{BP} exp\left(\frac{1}{4}\sum_{n=1}^{4} p_n\right)$
    - BP is for brevity penalty. Preventing short sentences from scoring too high
    - BP = 1, if MT_output_length > reference_output_length, or
    - BP = exp(1 − reference_output_length / MT_output_length), otherwise

9.3.4. Attention Model

Paper: [Neural Machine Translation by Jointly Learning to Align and Translate](#) by Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio.

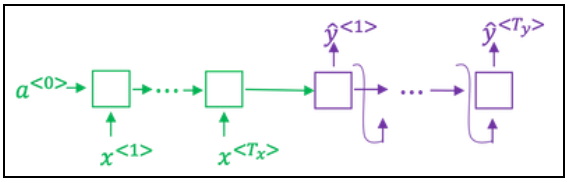
There's a modification to Encoder-Decoder architecture called the Attention Model that makes all this work much better.

The French sentence:

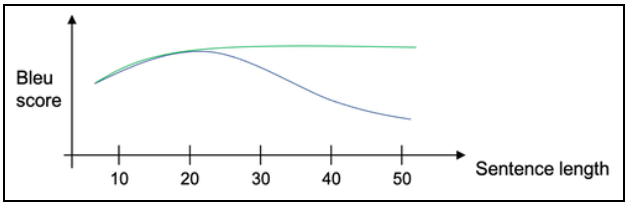
Jane s'est rendue en Afrique en septembre dernier, a apprécié la culture et a rencontré beaucoup de gens merveilleux; elle est revenue en parlant comment son voyage était merveilleux, et elle me tente d'y aller aussi.

The English translation:

Jane went to Africa last September, and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too.

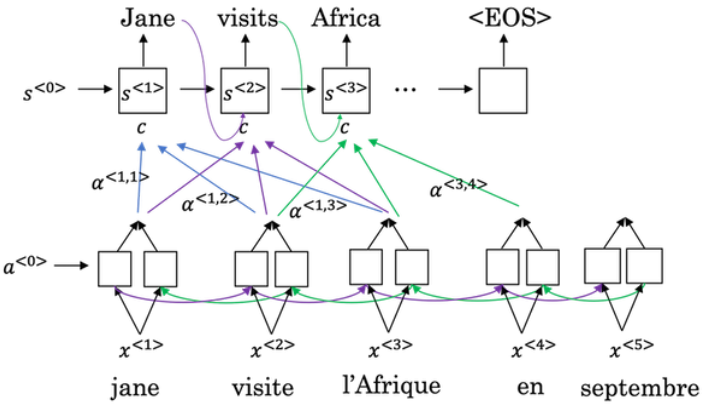


- The way a human translator would translate this sentence is not to first read the whole French sentence and then memorize the whole thing and then regurgitate an English sentence from scratch. Instead, what the human translator would do is read the first part of it, maybe generate part of the translation, look at the second part, generate a few more words, look at a few more words, generate a few more words and so on



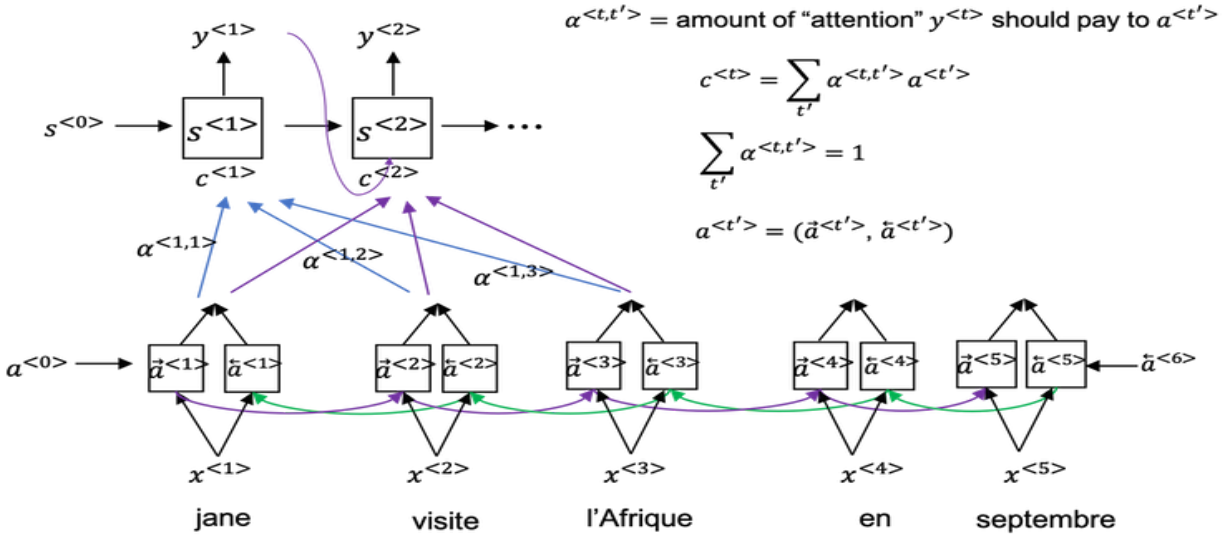
- The Encoder-Decoder architecture above works quite well for short sentences, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down (The blue line)
- The Attention model which translates maybe a bit more like humans looking at part of the sentence at a time, machine translation systems performance can look like the green line above
- What the Attention Model would be computing is a set of attention weights and we're going to use α<1,1> to denote when you're generating the first words, how much should you be paying attention to this first piece of information here and α<1,2> which tells us what we're trying to compute the first word of Jane, how much attention we're paying to the second word from the inputs, and α<1,3> and so on.

- Together this will be exactly the context from, denoted as C, that we should be paying attention to, and that is input to the RNN unit to try to generate the first word.
- In this way the RNN marches forward generating one word at a time, until eventually it generates maybe the <EOS> and at every step, there are attention weighs α<t,t'> that tells it, when you're trying to generate the t-th English word, how much should you be paying attention to the t'-th French word.

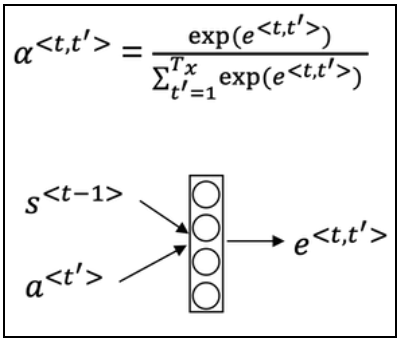


Model

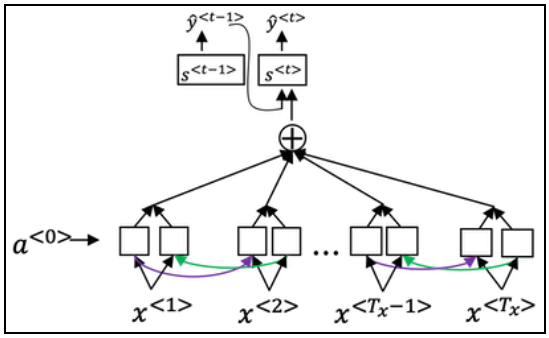
Assume you have an input sentence and you use a bidirectional RNN, or bidirectional GRU, or bidirectional LSTM to compute features on every word. In practice, GRUs and LSTMs are often used for this, maybe LSTMs are more common. The notation for the Attention model is shown below.



- Compute e<t,t'> using a small neural network:
  - The intuition is, if you want to decide how much attention to pay to the activation of t', it seems like it should depend the most on what is your own hidden state activation from the previous time step. And then a<t'>, the features from time step t', is the other input.
  - So it seems pretty natural that α<t,t'> and e<t,t'> should depend on s<t-1> and a<t'> . But we don't know what the function is. So one thing you could do is just train a very small neural network to learn whatever this function should be. And trust the backpropagation and trust gradient descent to learn the right function.

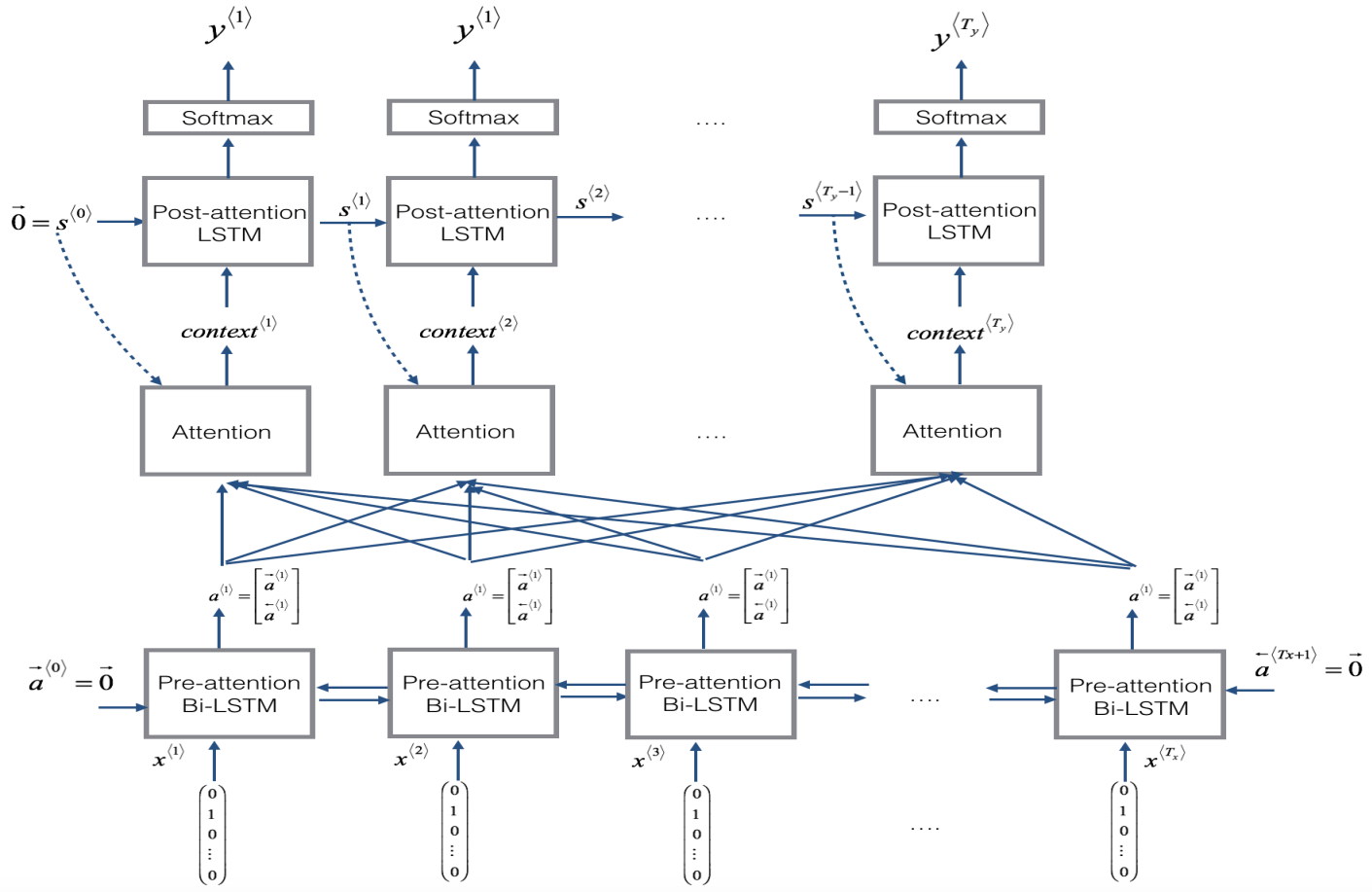


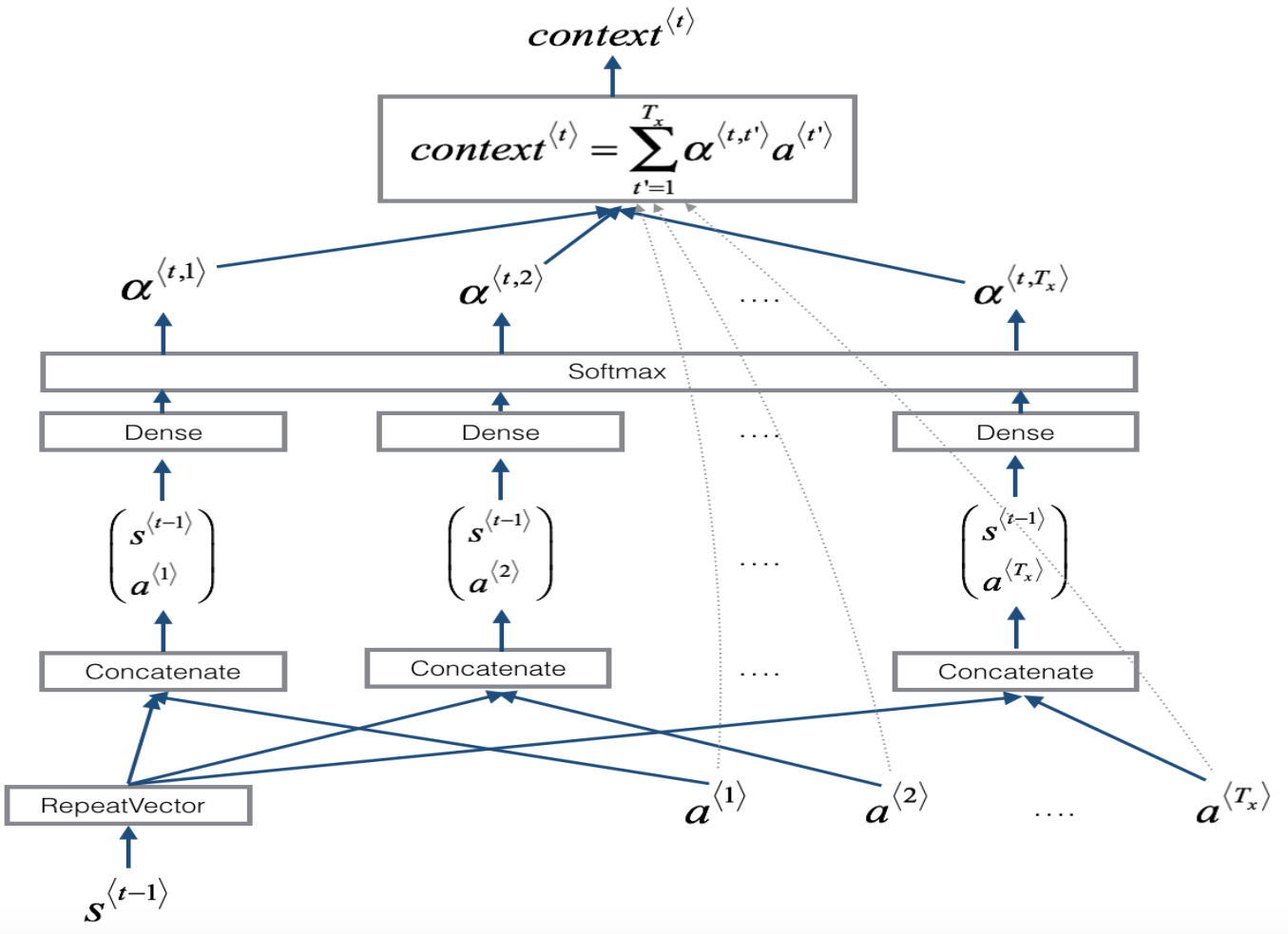
- One downside to this algorithm is that it does take quadratic time or quadratic cost to run this algorithm. If you have Tx words in the input and Ty words in the output then the total number of these attention parameters are going to be Tx * Ty.



Implementation tips:

- The first diagram below shows the attention model.
- The second diagram below shows what one "attention" step does to calculate the attention variables α<t,t'>.
- The attention variables α<t,t'> are used to compute the context variable context<t> for each timestep in the output (t=1, ..., Ty).

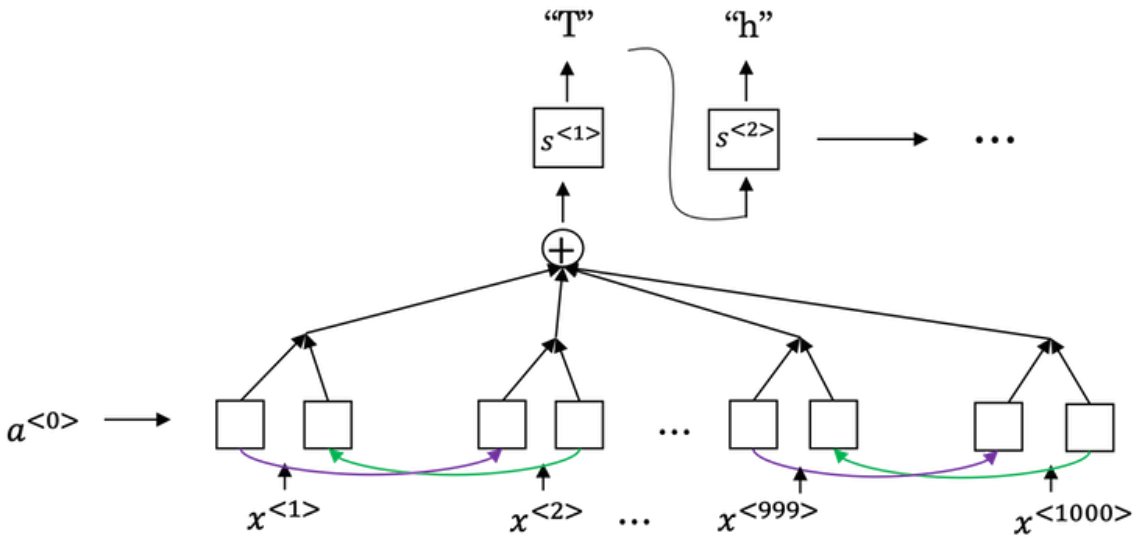

Softmax
Post-attention LSTM
Attention
Pre-attention Bi-LSTM


Softmax
Dense
Concatenate
RepeatVector

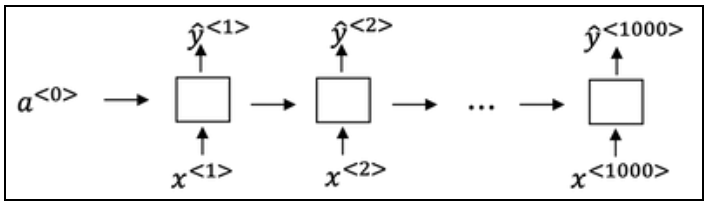
9.3.5. Speech recognition - Audio data

What is the speech recognition problem?
You're given an audio clip, x, and your job is to automatically find a text transcript, y.
How to build a speech recognition system?

- Attention model for speech recognition: one thing you could do is actually do that, where on the horizontal axis, you take in different time frames of the audio input, and then you have an attention model try to output the transcript like, "the quick brown fox"



- CTC cost for speech recognition: Connectionist Temporal Classification
  - Paper: Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks by Alex Graves, Santiago Fernandes, Faustino Gomez, and Jürgen Schmidhuber
    - For simplicity, this is a simple RNN, but in practice, this will usually be a bidirectional LSTM and bidirectional GRU and usually, a deeper model. But notice that the number of time steps here is very large and in speech recognition, usually the number of input time steps is much bigger than the number of output time steps.



  - For example, if you have 10 seconds of audio and your features come at a 100 hertz so 100 samples per second, then a 10 second audio clip would end up with a thousand inputs. But your output might not have a thousand alphabets, might not have a thousand characters.
  - The CTC cost function allows the RNN to generate an output like ttt_h_eee___[]___qqq__, here _ is for "blank", [] for "space".
  - The basic rule for the CTC cost function is to collapse repeated characters not separated by "blank".

Trigger Word Detection

With a RNN what we really do is to take an audio clip, maybe compute spectrogram features, and that generates audio features x<1>, x<2>, x<3>, that you pass through an RNN. So, all that remains to be done, is to define the target labels y.

- In the training set, you can set the target labels to be zero for everything before that point, and right after that, to set the target label of one. Then, if a little bit later on, the trigger word was said again at this point, then you can again set the target label to be one.
- Actually it just won't actually work reasonably well. One slight disadvantage of this is, it creates a very imbalanced training set, so we have a lot more zeros than we want.

- One other thing you could do, that it's little bit of a hack, but could make the model a little bit easier to train, is instead of setting only a single time step to operate one, you could actually make it to operate a few ones for several times. Guide to label the positive/negative words):
  - Assume labels y<t> represent whether or not someone has just finished saying "activate."
    - y<t> = 1 when that that clip has finished saying "activate".
    - Given a background clip, we can initialize y<t> = 0 for all t, since the clip doesn't contain any "activates."
  - When you insert or overlay an "activate" clip, you will also update labels for y<t>.
    - Rather than updating the label of a single time step, we will update 50 steps of the output to have target label 1.
    - Recall from the lecture on trigger word detection that updating several consecutive time steps can make the training data more balanced.

Implementation tips:

- Data synthesis is an effective way to create a large training set for speech problems, specifically trigger word detection.
- Using a spectrogram and optionally a 1D conv layer is a common pre-processing step prior to passing audio data to an RNN, GRU or LSTM.
- An end-to-end deep learning approach can be used to build a very effective trigger word detection system.

9.4. Transformers

Intuition

Transformer architecture allows you to run RNNs, GRU and LSTM computations for an entire sequence in parallel. You can ingest an entire sentence all at the same time, rather than just processing it one word at a time from left to right.

Paper: Attention Is All You Need by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin.

The major innovation of the transformer architecture is combining the use of attention based representations and a CNN convolutional neural network style of processing.

Attention + CNN:

- Self-Attention: the goal of self attention is, a sentence of five words will end up computing five representations for these five words (A1,A2,A3, A4, A5) and this will be an attention based way of computing representations for all the words in your sentence in parallel
- Multi-Head Attention: it is a for loop over self attention process, so you end up with multiple versions of these representations

Self-Attention

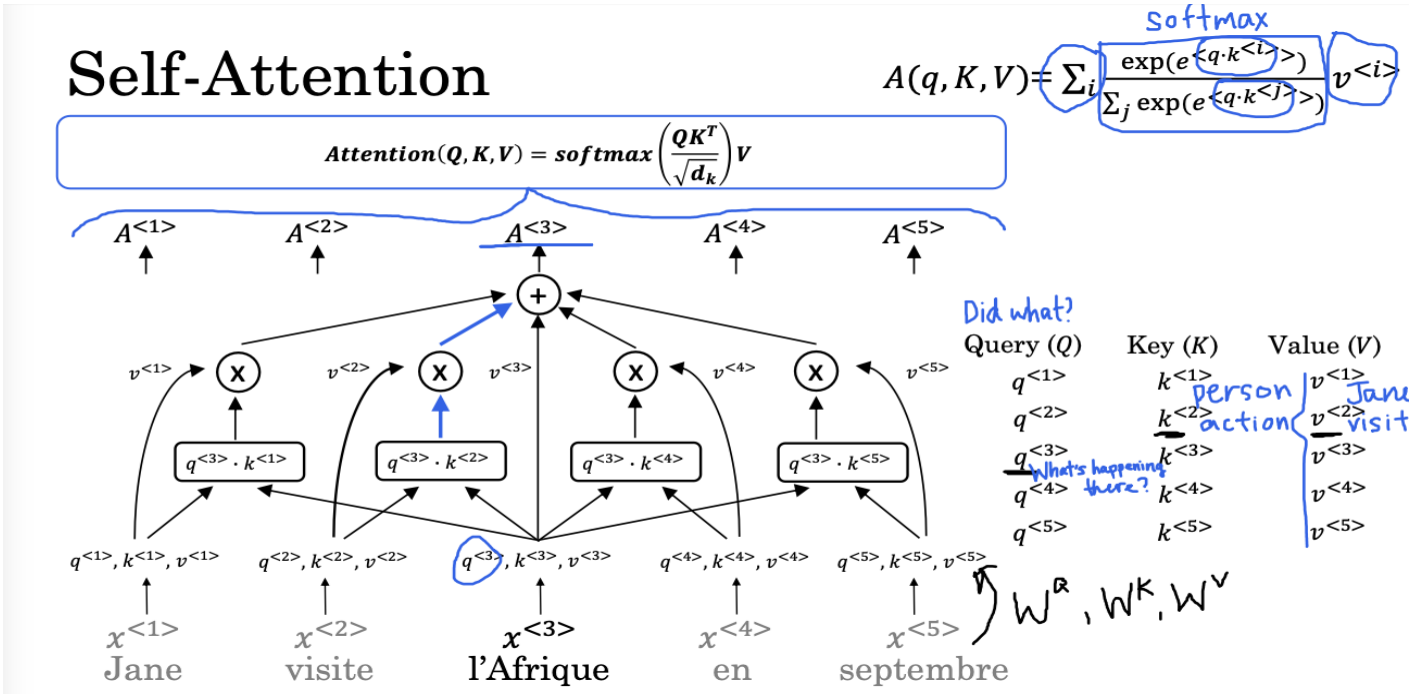
A(q,K,V) = Attention-based vector representation of a word

In the example of the French sentence, "Jane, visite l'Afrique en Septembre" calculate for each word A1 to A5

- Depending on how you're thinking of l'Afrique, you may choose to represent it differently, and that's what this representation A(3) will do, it will look at the surrounding (A1, A2, A4, A5) words to try to figure out how we're talking about Africa in this sentence, and find the most appropriate representation for this
- Same Attention equation but in parallel to all 5 words

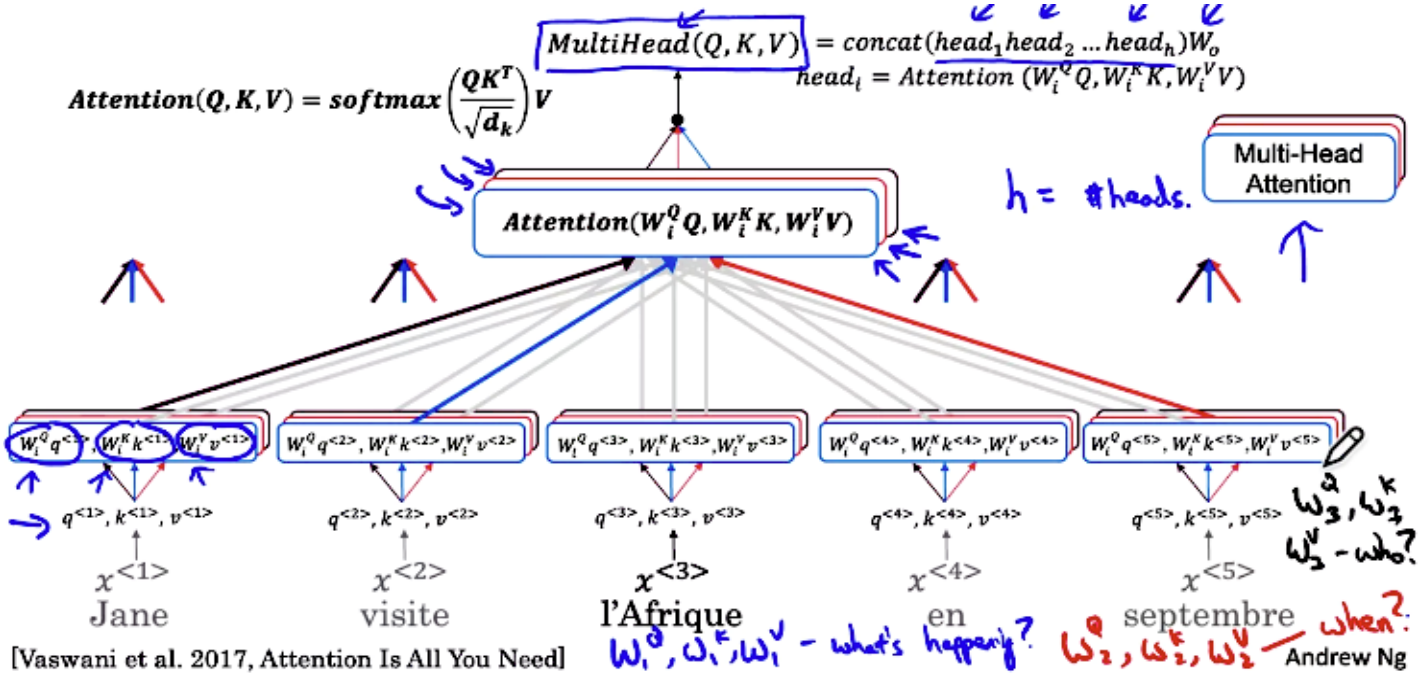
$$A(q,K,V) = \sum_i \frac{exp(qk^{<i>})}{\sum_j exp(qk^{<i>})} v^{<i>}$$

- For each word you have three values called the query q, key k, and value v these vectors are the key inputs to computing the attention value for each words
- WQ, WK, and WV are parameters (weights) of this learning algorithm, and they allow you to pull off these query, key, and value vectors for each word

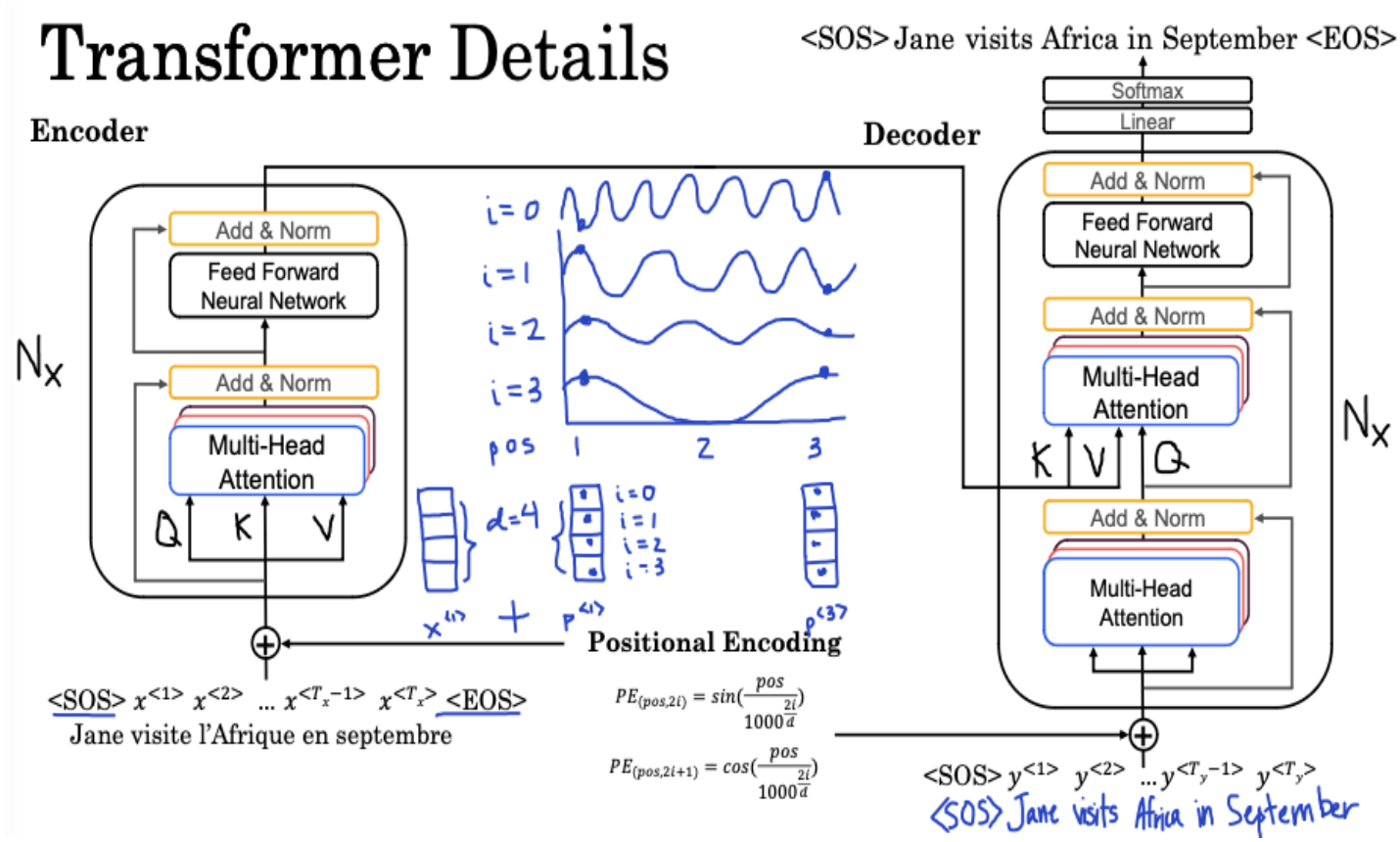


Multi-Head Attention

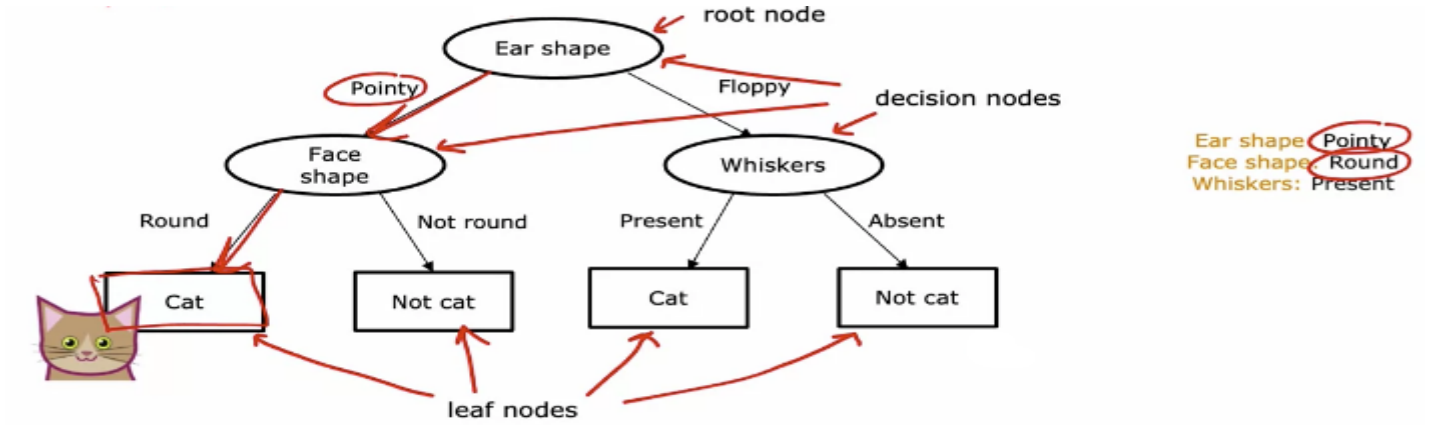
Each time you calculate self attention for a sequence is called a head.



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



10. Decision Tree Model



10.1. Decisions

Decision 1: How to choose what feature to split on at each node?

- Maximize purity (or minimize impurity)

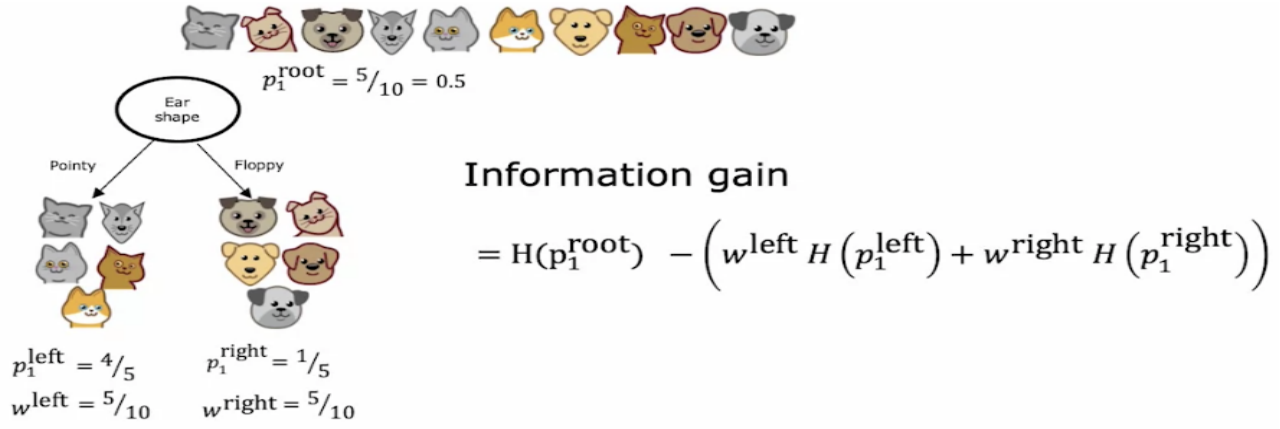
Decision 2: When do you stop splitting?

- When node is 100% one class
- When splitting a node will result in the tree exceeding a maximum depth
- When improvements in purity score are below a threshold
- When number of examples in a node is below a threshold

Entropy is a measure of impurity. Formula $H(p_1) = -p_1 log_2(p_1) - p_0 log_1(p_0)$

In decision tree learning, the reduction of entropy is called information gain.

Choosing split by calculating information gain



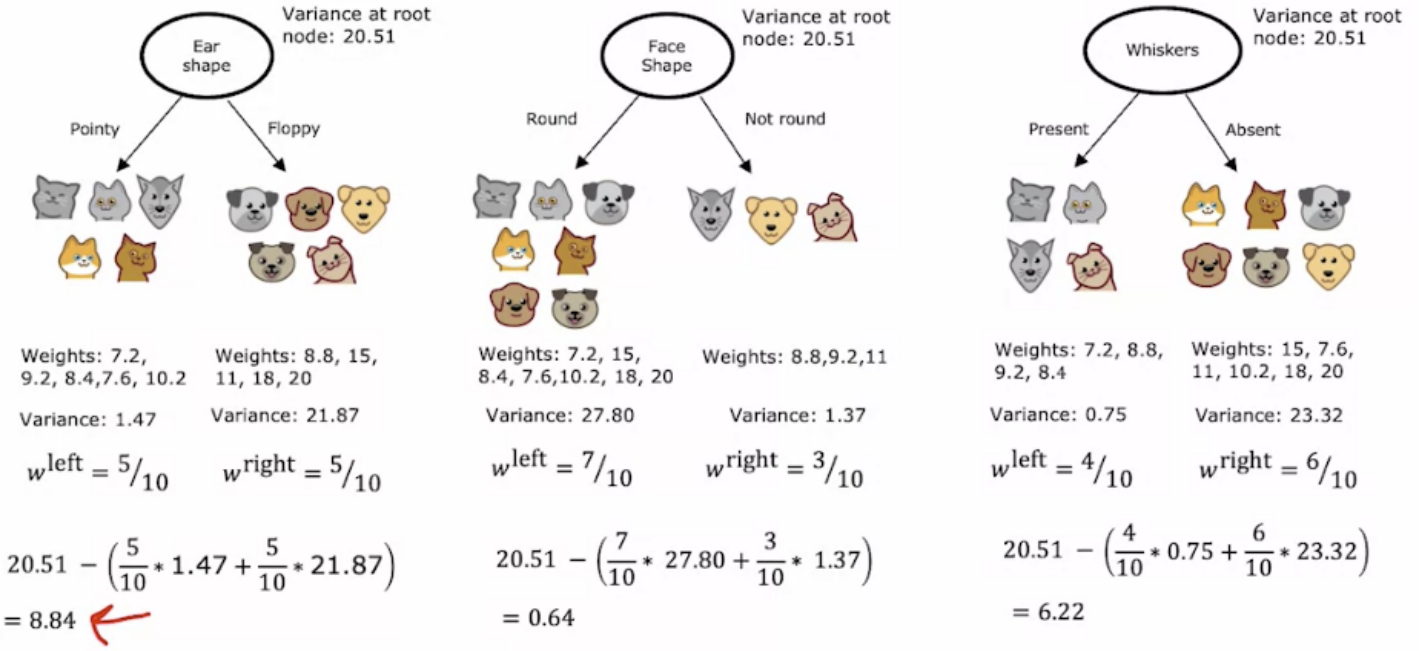
10.2. Decision Tree Learning Process

1. Start with all examples at the root node
2. Calculate information gain for all possible features, and pick the one with the highest information gain
3. Split dataset according to selected feature, and create left and right branches of the tree
4. Keep repeating splitting process until stopping criteria is met:
    a. When node is 100% one class
    b. When splitting a node will result in the tree exceeding a maximum depth
    c. When improvements in purity score are below a threshold
    d. When number of examples in a node is below a threshold

Use one hot encoding for categorical values.

For splitting continuous variables you can plot variables on x axis vs prediction value on y. When consuming splits, you would just consider different values to split on, carry out the usual information gain calculation and decide to split on that continuous value feature if it gives the highest possible information gain.

10.3. Regression Tree



The reason we use an **ensemble of trees** is by having lots of decision trees and having them vote, it makes your overall algorithm less sensitive to what any single tree may be doing because it gets only one vote out of three or one vote out of many, many different votes and it makes your overall algorithm more robust.

The process of **sampling with replacement** lets you construct a new training set that's a little bit similar to, but also pretty different from your original training set. It turns out that this would be the key building block for building an ensemble of trees. Let's take a look in the next video and how you could do that.

Randomizing Forest Algorithm

At each node, when choosing a feature to use to split, if n features are available, pick a random subset of k < n feature and allow the algorithm to only choose from that subset of features. A typical choice for the value of K would be to choose it to be the square root of n.

XGBoost

Algorithm focus more attention on the subset of examples that are not yet doing well on and get the new decision tree, the next decision tree reporting ensemble to try to do well on them.

10.4. Decision Trees vs Neural Networks

Decision trees

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast
- Small decision trees maybe human interpretable

Neural Networks

- Works well on all types of data
- Maybe slower than a decision tree
- Works with transfer learning
- When building a system of multiple models working together, it might be easier to string together multiple neural network

Transfer learning refers to using the neural network knowledge for another application.

When to use transfer learning (transfer from A to B):

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

11. Unsupervised Learning

11.1. Clustering

Clustering algorithm looks at the number of data points and automatically finds data points that are related or similar to each other.

11.1.1. K-means Algorithm

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K$

Repeat {

 # Assign points to cluster centroids

for i = 1 to m

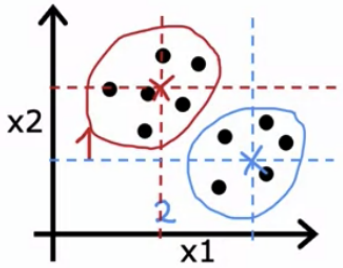
$c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$

 # Move cluster centroids

for k = 1 to K

μ_k := average (mean) of points assigned to cluster k

}



$$\mu_1 = \frac{1}{4}\left[x^{(1)} + x^{(5)} + x^{(6)} + x^{(10)}\right]$$

11.1.2. Cost Function

$c^{(i)}$ - index of cluster (1,2,...,K) to which example $x^{(i)}$ is currently assigned

μ_k - cluster centroid k

$\mu_{c^{(i)}}$ - cluster centroid of cluster to which example $x^{(i)}$ has been assigned

$$J(c^{(1)},\dots,c^{(m)},\mu_1,\dots,\mu_k) = \frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Randomly initialization K cluster centroids

Choose K < m

Randomly pick initial K from training examples. Repeat to find the best local optima. To find the best local optima, compute the cost function J for all of these solutions. And then to pick one with the lowest value for the cost function J.

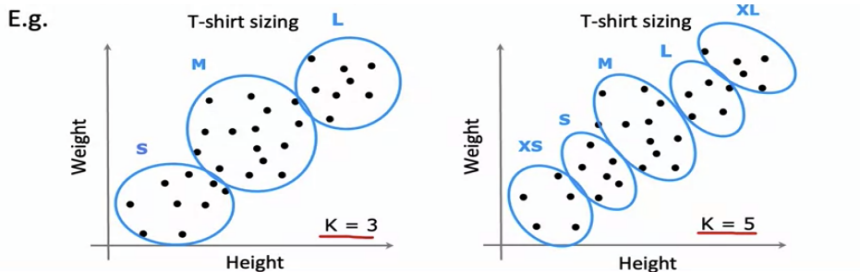
For $i = 1$ to 100 { (50-1000)
  Randomly initialize K-means. (k random examples)
  Run K-means. Get $c^{(1)},\dots,c^{(m)},\mu_1,\mu_1,\dots,\mu_k$
  Computer cost function (distortion)
  $J(c^{(1)},\dots,c^{(m)},\mu_1,\mu_1,\dots,\mu_k)$
}
Pick set of clusters that gave lowest cost J

11.1.3. Choosing number of clusters

Elbow method (not recommended): run K-means with a variety of values of K and plot the cost function or the distortion function J as a function of the number of clusters.

Better Option: Evaluate K-means based on a metric for how well it performs for downstream/later purposes.

E.g.

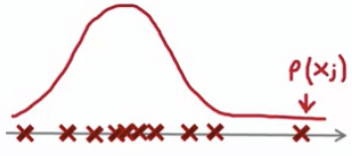


11.2. Anomaly Detection

11.2.1. Anomaly Detection Algorithm with Gaussian (Normal) Distribution

1. Choose n features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1,\dots,\mu_n,\sigma_1^2,\dots,\sigma_n^2$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \qquad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$



3. Given new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j;\mu_j,\sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp(-\frac{(x_j-\mu_j)^2}{2\sigma_j^2})$$

Anomaly if $p(x) < \varepsilon$

Aircraft engines dataset example

10000 good engines

20 flawed engines (anomalous, y = 1)

Training: 6000 good engines (y=0)

CV: 2000 good engines and 10 anomalous

Test: 2000 good engines and 10 anomalous

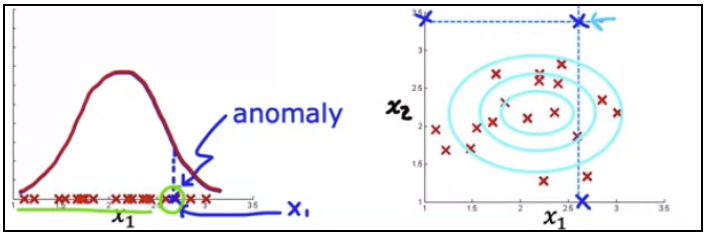
11.2.2. Anomaly Detection vs. Supervised Learning

Anomaly Detection	Supervised Learning
Very small number of positive examples (y=1) and large number of negative examples (y=0) examples. Many different types of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like. Future anomalies may look nothing like any of the anomalous examples we've seen so far. E.g. Fraud	Large number of positive and negative examples. Enough to get a sense of what positive examples are like. Future positive examples likely to be similar to ones in the training set. E.g. Spam

Choosing Features

It is important to use the right features in Unsupervised learning as there is no y.

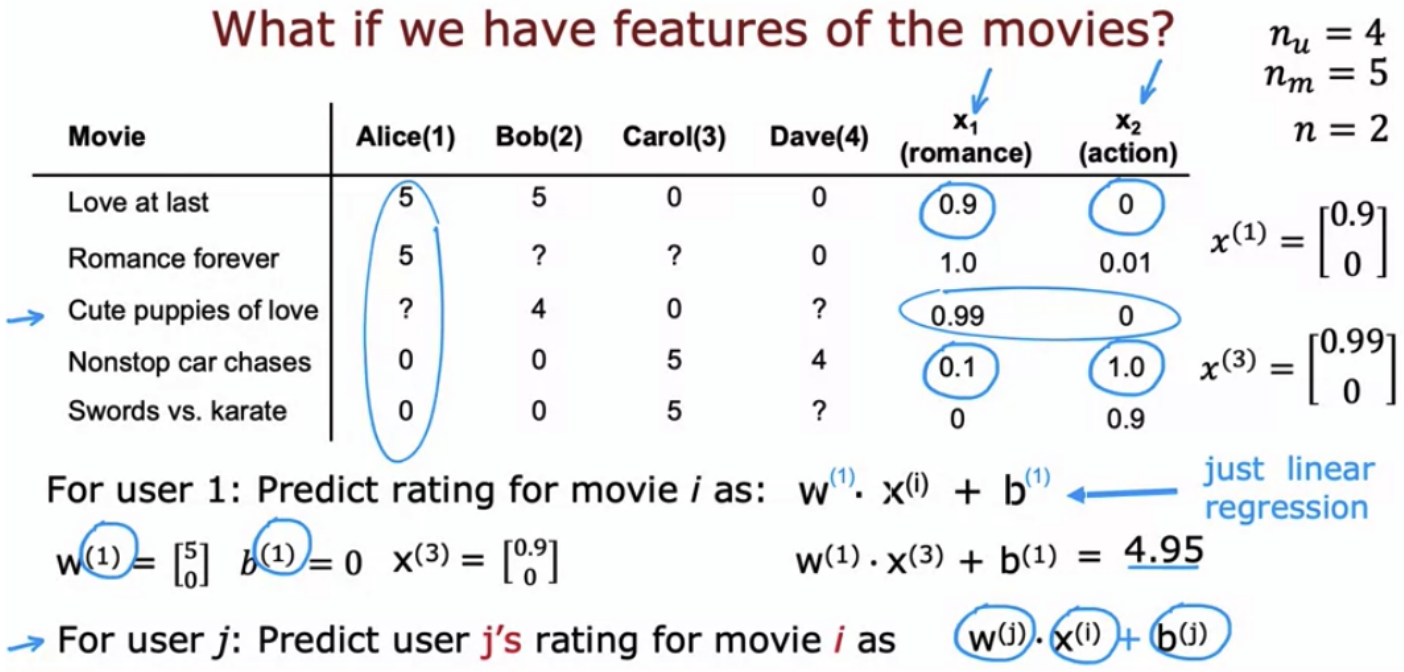
- Make sure the feature has gaussian distribution
- If not gaussian then compute log(x) or log(x+c) or x**2



11.2.3. Error Analysis for Anomaly Detection

Want: $p(x) \geq \varepsilon$ large for normal examples x. $p(x) < \varepsilon$ small for anomalous examples x.
Most common problem: $p(x)$ is comparable (say, both large) for normal and anomalous examples x.

11.3. Recommender Systems



Notation

$r(i,j) = 1$ *if the user j has rated movie i* (0 *otherwise*)

$y^{(i,j)} =$ *rating given by user j on movie i* (*if defined*)

$w^{(j)}, b^{(j)} =$ *parameters for user j*

$x^{(i)} =$ *feature vector for movie i*

$m^{(j)} =$ *no. of movies rated by user j*

11.3.1. Cost Function

To learn parameters $w^{(j)}, b^{(j)}$ for user j :

$$J(w^{(j)}, b^{(j)}) = \frac{1}{2} \sum_{i:r(i,j)} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (w_k^{(j)})^2$$

To learn parameters $w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, \cdots w^{(n_u)}, b^{(n_u)}$ for all users :

$$J\begin{pmatrix} w^{(1)}, \ \ldots, w^{(n_u)} \\ b^{(1)}, \ \ldots, b^{(n_u)} \end{pmatrix} = \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (w_k^{(j)})^2$$

($f(x)$)

Example without x1 and x2 features

11.3.2. Cost Function With Unknown Features

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0 ←	?	? $x^{(2)}$
Cute puppies of love	?	4	0	? ←	?	? $x^{(3)}$
Nonstop car chases	0	0	5	4 ←	?	?
Swords vs. karate	0	0	5	? ←	?	?

$w^{(1)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$, $w^{(2)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$, $w^{(3)} = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$, $w^{(4)} = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$

$b^{(1)} = 0$, $b^{(2)} = 0$, $b^{(3)} = 0$, $b^{(4)} = 0$ ←

using $w^{(j)} \cdot x^{(i)} + b^{(j)}$

$w^{(1)} \cdot x^{(1)} \approx 5$
$w^{(2)} \cdot x^{(1)} \approx 5$
$w^{(3)} \cdot x^{(1)} \approx 0$
$w^{(4)} \cdot x^{(1)} \approx 0$

→ $x^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Given $w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, \cdots, w^{(n_u)}, b^{(n_u)}$

to learn $x^{(i)}$:

$$J(x^{(i)}) = \frac{1}{2} \sum_{j:r(i,j)=1} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

→ To learn $x^{(1)}, x^{(2)}, \cdots, x^{(n_m)}$:

$$J(x^{(1)}, x^{(2)}, \ldots, x^{(n_m)}) = \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

11.3.3. Collaborative Filtering

	$j=1$ Alice	$j=2$ Bob	$j=3$ Carol
Movie1 ($i=1$)	5	5	?
Movie2 ($i=2$)	?	2	3

Cost function to learn $w^{(1)}, b^{(1)}, \cdots w^{(n_u)}, b^{(n_u)}$:

$$\min_{w^{(1)}, b^{(1)}, \cdots, w^{(n_u)}, b^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (w_k^{(j)})^2$$

Cost function to learn $x^{(1)}, \cdots, x^{(n_m)}$:

$$\min_{x^{(1)}, \cdots, x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Put them together:

$$\min_{\substack{w^{(1)}, \ldots, w^{(n_u)} \\ b^{(1)}, \ldots, b^{(n_u)} \\ x^{(1)}, \ldots, x^{(n_m)}}} J(w, b, x) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} (w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (w_k^{(j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

11.3.4. Gradient Descent

repeat {

$$w_i^{(j)} = w_i^{(j)} - \alpha \frac{\partial}{\partial w_i^{(j)}} J(w, b, x)$$

$$b^{(j)} = b^{(j)} - \alpha \frac{\partial}{\partial b^{(j)}} J(w, b, x)$$

$$x_k^{(i)} = x_k^{(i)} - \alpha \frac{\partial}{\partial x_k^{(i)}} J(w, b, x) \}$$

Collaborative filtering with binary labels

1 = watched the movie till the end,
0 = did not watch till the end,
? = did not start watching the movie
Other examples:

1. Did user j purchase an item after being shown?
2. Did user j like an item?
3. Did user j spend at least 30 sec with an item?
4. Did user j click on an item?

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)
Love at last	1	1	0	0
Romance forever	1	?	?	0
Cute puppies of love	?	1	0	?
Nonstop car chases	0	0	1	1
Swords vs. karate	0	0	1	?

Cost function for binary application

Previous cost function:

$$\frac{1}{2} \sum_{(i,j):r(i,j)=1} \underbrace{(w^{(j)} \cdot x^{(i)} + b^{(j)}}_{f(x)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (w_k^{(j)})^2$$

Loss for binary labels $y^{(i,j)}$: $f_{(w,b,x)}(x) = g(w^{(j)} \cdot x^{(i)} + b^{(j)})$

$$L\left(f_{(w,b,x)}(x), y^{(i,j)}\right) = -y^{(i,j)} \log\left(f_{(w,b,x)}(x)\right) - (1 - y^{(i,j)}) \log\left(1 - f_{(w,b,x)}(x)\right)$$ ← Loss for single example

$$J(w, b, x) = \sum_{(i,j):r(i,j)=1} L(f_{(w,b,x)}(x), y^{(i,j)})$$ ← cost for all examples

$g(w^{(j)} \cdot x^{(i)} + b^{(j)})$

Mean normalization of the dataset will help to improve accuracy and performance. Impute null data with mean of the rating.

11.3.5. TensorFlow Implementation of Collaborative Filtering

Gradient descent algorithm

Repeat until convergence

$w = w - \alpha \frac{\partial}{\partial w} J(w,b,x)$

$b = b - \alpha \frac{\partial}{\partial b} J(w,b,x)$

$X = X - \alpha \frac{\partial}{\partial x} J(w,b,x)$

```
# Instantiate an optimizer.
optimizer = keras.optimizers.Adam(learning_rate=1e-1)

iterations = 200
for iter in range(iterations):
    # Use TensorFlow's GradientTape
    # to record the operations used to compute the cost
    with tf.GradientTape() as tape:

        # Compute the cost (forward pass is included in cost)
        cost_value = cofiCostFuncV(X, W, b, Ynorm, R,
                                   num_users, num_movies, lambda)
                                   # n_u      # n_m

    # Use the gradient tape to automatically retrieve
    # the gradients of the trainable variables with respect to
    #    the loss
    grads = tape.gradient( cost_value, [X,W,b] )

    # Run one step of gradient descent by updating
    # the value of the variables to minimize the loss.
    optimizer.apply_gradients( zip(grads, [X,W,b]) )
```

11.3.6. Limitations of Collaborative Filtering

Cold start problem. How to

- rank new items that few users have rated?
- show something reasonable to new users who have rated a few items?

Use side information about items or users:

- Item: Genre, movie stars, studio, …
- User: Demographics (age, gender, location), expressed preferences, …

Collaborative filtering vs Content-based filtering

Collaborative filtering recommend items to you based on rating of users who face similar ratings as you
Content-based filtering recommend items to you based on features of user and item to find good match

11.3.7. Content-based Filtering Neural Network

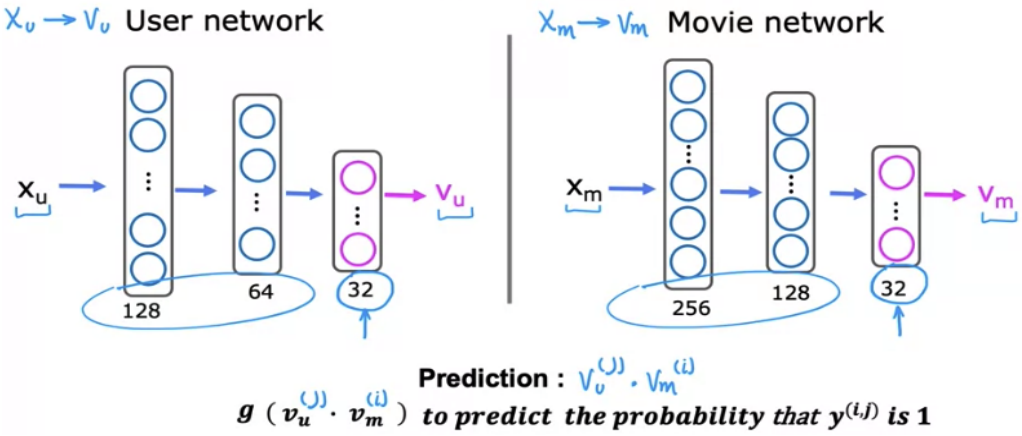
Recommending from a large catalog: Retrieval & Ranking
Retrieval:

- Generate large list of plausible item candidates
  - For each of the last 10 movies watched by the user, find 10 most similar movies
  - For most viewed 3 genres, find the top 10 movies
  - Top 20 movies in the country
- Combine retrieved items into list, removing duplicates and items already watched/purchased

Ranking:

- Take list retrieved and rank using learned model
- Display ranked items to user

Retrieving more items results in better performance but slower recommendations. To analyze/optimize the trade-off, carry out offline experiments to see if retrieving additional items result in more relevant recommendations



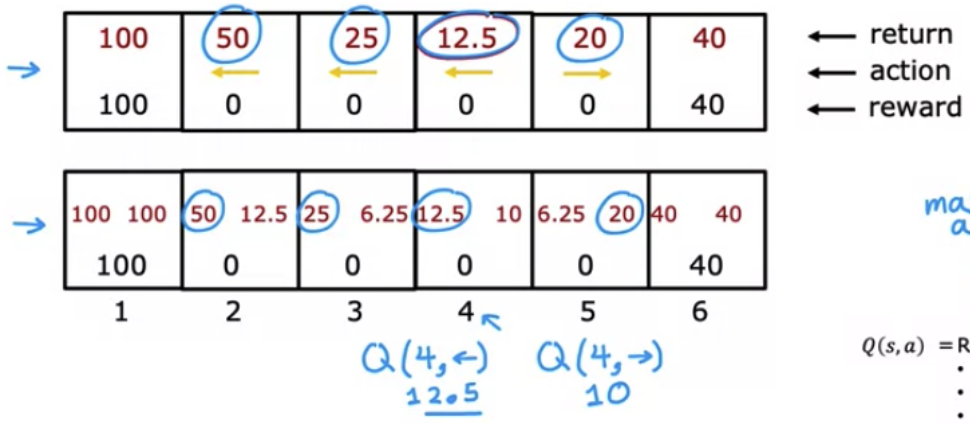
12. Reinforcement Learning

One way to think of why reinforcement learning is so powerful is you have to tell it what to do rather than how to do it. A key input to reinforcement learning is something called the reward. Specifying the reward function rather than the optimal action gives you a lot more flexibility in how you design the system. So the way to think of the reward function is a bit like training a dog. So how do you get a puppy to behave well? You let it do its thing and whenever it does something good, you go, good dog. And whenever they did something bad, you go, bad dog.
The return in reinforcement learning is the sum of the rewards that the system gets, weighted by the discount factor, where rewards in the far future are weighted by the discount factor raised to a higher power.
A policy is a function $a = \pi(s)$ mapping from states to actions, that tells you what action a to take in a given state s

12.1. Key Factors

	Mars rover 	Helicopter 	Chess 
states	6 states	position of helicopter	pieces on board
actions	← →	how to move control stick	possible move
rewards	100, 0, 40	+1, −1000	+1, 0, −1
discount factor γ	0.5	0.99	0.995
return	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$
policy π		Find $\pi(s) = a$	Find $\pi(s) = a$

The state action value function (Q function) is a function typically denoted by the letter uppercase Q. And it's a function of a state you might be in as well as the action you might choose to take in that state.



$\max_a Q(s,a)$

$\pi(s) = a$

$Q(s,a)$ = Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

The best possible return from state s is $\max_a Q(s,a)$.

The best possible action in state s is the action a that gives $\max_a Q(s,a)$.

Q^* Optimal Q function

12.2. Bellman Equation

$Q(s,a)$ = Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

$R(1)=100$ $R(2)=0$ ... $R(6)=40$

s : current state
a : current action
$R(s)$ = reward of current state
s' : state you get to after taking action a
a' : action that you take in state s'

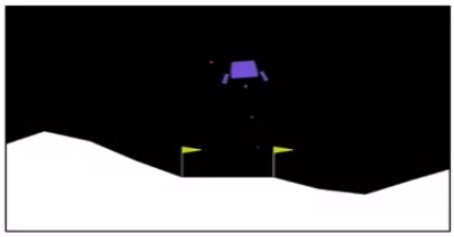
$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

12.3. Markov Decision Process (MDP)

Markov decision process the future depends only on where you are now, not on how you got here.

Continuous state Markov decision process, continuously MTP. The state of the problem isn't just one of a small number of possible discrete values, like a number from 1-6. Instead, it's a vector of numbers, any of which could take any of a large number of values. Whereas for the Mars rover example, the state was just one of six possible numbers. It could be one, two, three, four, five or six. For the car, the state would comprise this vector of six numbers, and any of these numbers can take on any value within its valid range.

12.4. Algorithm



Initialize neural network randomly as guess of $Q(s,a)$.

Repeat {

Take actions in the lunar lander. Get $(s,a,R(s),s')$.

Store 10,000 most recent $(s,a,R(s),s')$ tuples.

Replay Buffer

Train neural network:

Create training set of 10,000 examples using

$x = (s,a)$ and $y = R(s) + \gamma \max_{a'} Q(s',a')$

Train Q_{new} such that $Q_{new}(s,a) \approx y$.

Set $Q = Q_{new}$.

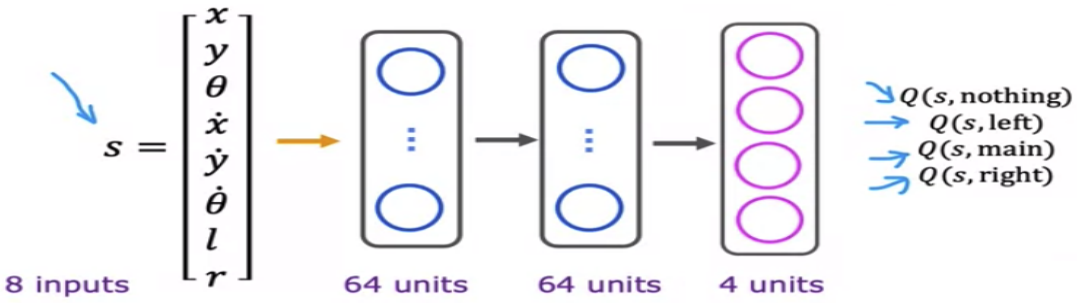
$f_{w,B}(x) \approx y$

x, y

$x^{(1)}, y^{(1)}$

$x^{(10000)}, y^{(10000)}$

12.4.1. Deep Reinforcement Learning Architecture



In a state s, input s to neural network.

Pick the action a that maximizes $Q(s,a)$. $R(s) + \gamma \max_{a'} Q(s',a')$

12.4.2. Greedy policy

How to choose actions while still learning?

In some state s

Option 1: Pick the action a that maximizes Q(s,a)

Option 2:

- With probability 0.95, pick the action a that maximizes Q(s,a). Greedy, 'Exploitation'
- With probability 0.05, pick action a randomly. 'Exploration'

Given a training set with a high number of examples, every step of gradient descent computes sum over all examples.

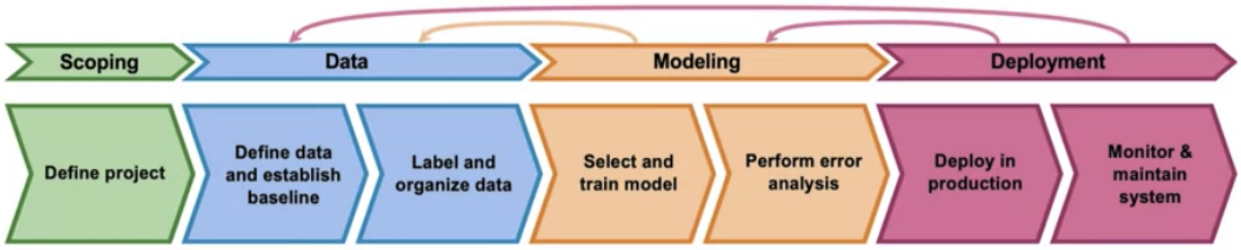
Mini-batch gradient descent looks at the random subset of the data on each iteration.

Soft Update allows you to make a more gradual change to Q or to the neural network parameters W and B that affect your current guess for the Q function Q of s, a. Soft update method causes the reinforcement learning algorithm to converge more reliably and it makes it less likely that algorithm will oscillate or divert or have other undesirable properties.

12.5. Limitations of Reinforcement Learning

- Much easier to get to work in a simulation than a real robot
- Far fewer application than supervised and unsupervised learning
- Exciting research direction with potential for future applications

13. Machine Learning Project Lifecycle



13.1. Scoping

Scoping refers to picking what project to work on and planning out the scope of the project.
Questions:

- What project should we work on?
- What are the metrics for success?
- What are the resources (data, time, people) needed?

13.1.1. Scoping process

1. Brainstorm business problems (not AI problems)
2. Brainstorm AI solution
3. Assess the feasibility and value of potential solutions
   a. To identify feasibility use external benchmark (literature, other company, competitors)
   b. Gather diligence on value from tech and business teams then have both of the teams to agree on metrics
4. Determine milestones
5. Budget for resources

13.1.2. Ethical consideration

- Is this project creating net positive societal value?
- Is this project reasonably fair and free from bias?
- Have any ethical concerns been openly aired and debated?

13.1.3. Milestones & Resourcing

- ML metrics (accuracy, precision/recall, etc.)
- Software metrics (latency, throughput)
- Business metrics (revenue, etc.)
- Resources needed (data, personnel, help from other teams)
- Timeline

If unsure, consider benchmarking to other projects, or build a POC (Proof of Concept) first.

13.1.4. Setting Up Your Goal

Single number evaluation metric (metrics for success)
Evaluation metric allows you to quickly tell if classifier A or classifier B is better, and therefore having a dev set plus single number evaluation metric tends to speed up iterating.
Satisficing and optimizing metric

If we care about the classification accuracy of our cat's classifier and also care about the running time or some other performance, instead of combining them into an overall evaluation metric by their artificial linear weighted sum, we actually can have one thing as an optimizing metric and the others as satisfying metrics. In the cat's classifier example, we might have accuracy as an optimizing metric and running time as satisficing metric.

13.2. Data

13.2.1. Define Data

Major types of data problems

Unstructured data:

- May or may not have huge collection of unlabeled examples x
- Human can label more data
- Data augmentation more likely to be helpful

Structured data:

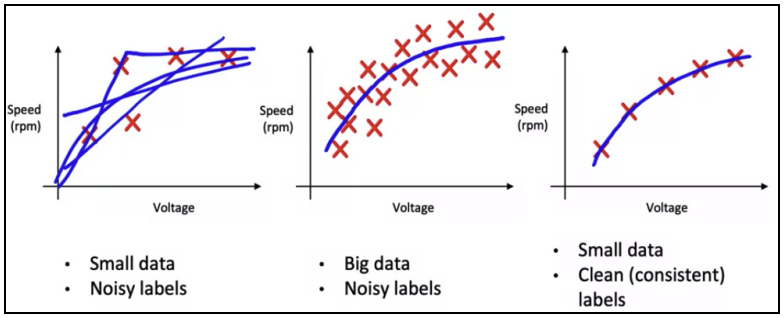
- May be more difficult to obtain more data
- Human labeling may not be possible (with some exceptions)

Small data (<10k):

- Clean labels are critical
- Can manually look through dataset and fix labels
- Can get all the labelers to talk to each other

Big data (>10k):

- Emphasis data process (create a data definition process and share that with all labelers)



Data definition questions (for labeling data)

- What is the input x?
  - Lighting? Contract? Resolution?
  - What features need to be included?
- What is the target label y?
  - How can we assure labelers give consistent labels?

Improving label consistency

- Have multiple labelers label same example
- When there is a disagreement, have MLE, SME and/or labelers discuss definition of y or reach agreement
- If labelers believe that x doesn’y contain enough information, consider changing x
- Iterate until it is hard to significantly increase agreement
- Standardize labels
- Merge classes of labels
- Have a class/label to capture uncertainty
- For big data consider having multiple labelers label every example and using voting or consensus labels to increase accuracy

13.2.2. Train/Dev/Test Distributions

Guideline:

- Choose a dev set and test set to reflect data you expect to get in future and consider important to do well on.
- In particular, the dev set and the test set here, should come from the same distribution.

Size of the dev and test sets

- In the era of big data, the old rule of thumb of a 70/30 is that, that no longer applies. And the trend has been to use more data for training and less for dev and test, especially when you have very large datasets.

- Suppose we have a million training examples, it might be quite reasonable to set up the data so that we have 98% in the training set, 1% dev, and 1% test.
- The guideline is, to set your test set to big enough to give high confidence in the overall performance of your system.

When to change dev/test sets and metrics

In an example of cat classification system, classification error might not be a reasonable metric if two algorithms have the following performance:

Algorithm	**Classification error**	**Issues**	**Review**
Algorithm A	3%	letting through lots of porn images	showing pornographic images to users is intolerable
Algorithm B	5%	no pornographic images	classifies fewer images but acceptable

In this case, the metric should be modified. One way to change this evaluation metric would be adding weight terms.

metric	calculation	notation
classification error	$\frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} \mathcal{L}\{\hat{y}^{(i)} \neq y^{(i)}\}$	L can be identity function to count correct labels
weighted classification error	$\frac{1}{\sum w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathcal{L}\{\hat{y}^{(i)} \neq y^{(i)}\}$	$w^{(i)} = \begin{cases} 1, & if\ x(i)\ is\ not\ pornographic \\ 10, & if\ x(i)\ is\ pornographic \end{cases}$

This is actually an example of an orthogonalization where I think you should take a machine learning problem and break it into distinct steps.

1. Figure out how to define a metric that captures what you want to do. (place the target)
2. Think about how to actually do well on this metric. (shoot the target)

The overall guideline is if your current metric and data you are evaluating on doesn't correspond to doing well on what you actually care about, then change your metrics and/or your dev/test set to better capture what you need your algorithm to actually do well on.

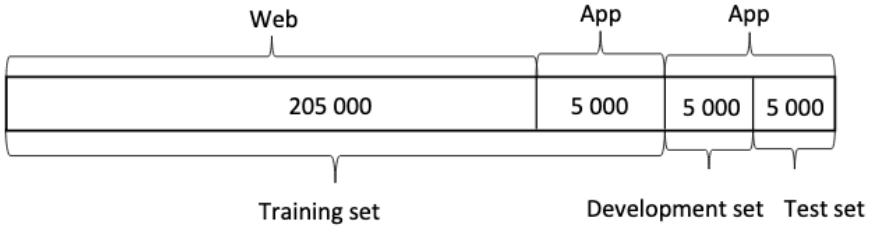
13.2.3. Mismatched Training and Dev/Test Set

Training and testing on different distributions

In the Cat vs Non-cat example, there are two sources of data used to develop the mobile app.

- The first data distribution is small, 10,000 pictures uploaded from the mobile application. Since they are from amateur users, the pictures are not professionally shot, not well framed and blurrier.
- The second source is from the web, you downloaded 200,000 pictures where cat's pictures are professionally framed and in high resolution.

The guideline is that you have to choose a development set and test set to reflect data you expect to get in the future and consider important to do well.



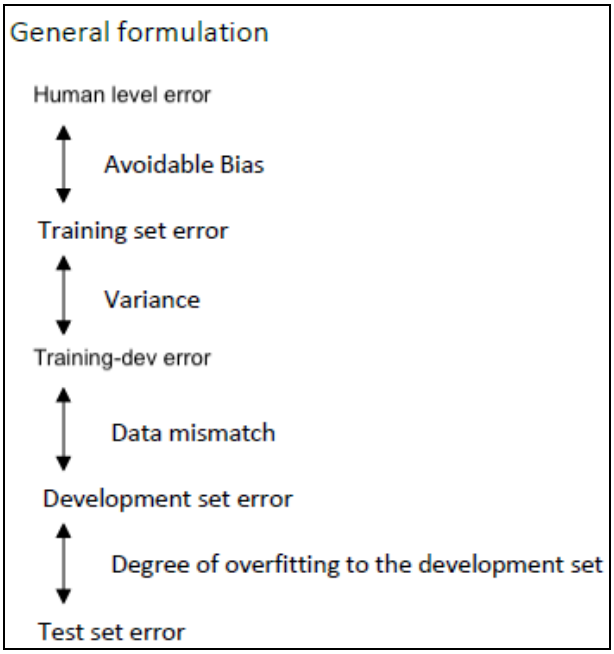
Bias and Variance with mismatched data distributions

Instead of just having bias and variance as two potential problems, you now have a third potential problem, data mismatch.

Training-dev set: same distribution as training set, but not used for training. Split and validate if the difference between training and dev sets error is too high.

This is a general guideline to address data mismatch:

1. Perform manual error analysis to understand the error differences between training, development/test sets. Development should never be done on a test set to avoid overfitting.
2. Make training data or collect data similar to development and test sets. To make the training data more similar to your development set, you can use artificial data synthesis. However, it is possible that you might be accidentally simulating data only from a tiny subset of the space of all possible examples.



13.2.4. Establish Baseline

Baseline helps to indicate what might be possible. In some cases it also gives a sense of what is irreducible error/Bayes error.

- Human Level Performance (HLP)
- Literature search for state-of-the-art / open source
- Quick-and-dirty implementation
- Performance of older system

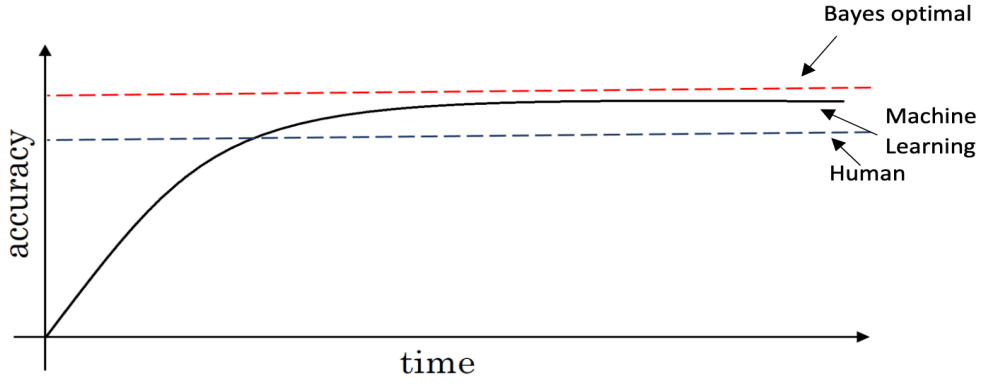
HLP is the accuracy of a human making a guess on y output. You can establish a baseline based on HLP. Usually works well with unstructured data (images, audio, text). When the ground truth label is externally defined, HLP gives an estimate for Bayes error/irreductible error. However, often ground truth is just another human label.

13.2.5. Comparing to Human-level Performance

Why human-level performance

A lot more machine learning teams have been talking about comparing the machine learning systems to human-level performance.

- First, 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.
- Second, the workflow of designing and building a machine learning system is much more efficient when we're trying to do something that humans can also do.



Machine learning progresses slowly when it surpasses human-level performance. One of the reasons is that human-level performance can be close to Bayes optimal error, especially for natural perception problems. Bayes optimal error is defined as the best possible error. In other words, it means that any functions mapping from x to y can't surpass a certain level of accuracy. Also, when the performance of machine learning is worse than the performance of humans, we can improve it with different tools. They are harder to use once it surpasses human-level performance.

Avoidable bias

By knowing the human-level performance, it is possible to tell when a training set is performing well or not.

Performance	Scenario A	Scenario B
Human error	1%	7.5%
Training error	8%	8%
Development error	10%	10%

In this case, the human-level error as a proxy for Bayes error since humans are good at identifying images. If you want to improve the performance of the training set but you can't do better than the Bayes error otherwise the training set is overfitting. By knowing the Bayes error, it is easier to focus on whether bias or variance avoidance tactics will improve the performance of the model.

- Scenario A: There is a 7% gap between the performance of the training set and the human-level error. It means that the algorithm isn't fitting well with the training set since the target is around 1%. To resolve the issue, we use bias reduction techniques such as training a bigger neural network or running the training set longer.
- Scenario B: The training set is doing good since there is only a 0.5% difference with the human-level error. The difference between the training set and the human-level error is called avoidable bias. The focus here is to reduce the variance since the difference between the training error and the development error is 2%. To resolve the issue, we use variance reduction techniques such as regularization or have a bigger training set.

Understanding human-level performance

Summary of bias/variance with human-level performance:

- Human-level error is a proxy for Bayes error.
- If the difference between human-level error and the training error is bigger than the difference between the training error and the development error. The focus should be on bias reduction techniques.
- If the difference between training error and the development error is bigger than the difference between the human-level error and the training error. The focus should be on variance reduction technique

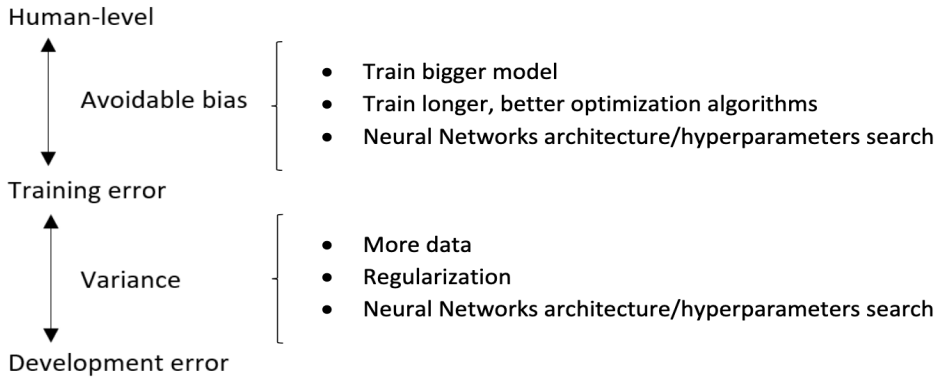
Surpassing human-level performance

There are many problems where machine learning significantly surpasses human-level performance, especially with structured data: Online advertising, Product recommendations, Logistics (predicting transit time), Loan approvals
And these are not natural perception problems, so these are not computer vision, or speech recognition, or natural language processing tasks. Humans tend to be very good at natural perception tasks. So it is possible, but it's just a bit harder for computers to surpass human-level performance on natural perception tasks.

Improving your model performance

There are two fundamental assumptions of supervised learning.

1. The first one is to have a low avoidable bias which means that the training set fits well.
2. The second one is to have a low or acceptable variance which means that the training set performance generalizes well to the development set and test set.



13.2.6. Label and Organize Data

How long should you spend obtaining data?

- Get into iteration loop as quickly as possible
- Instead of asking: How long would it take to obtain m examples? Ask: How much data can we obtain in k days?
- Exception: if you have worked on the problem before and from experience you know you need m examples

When making a decision on where to get data you can have a brainstorm session where you list all your possible data sources. You have to consider data quality, privacy and regulatory constraints.

Labeling data

- Options: In-house vs. outsourced vs. crowdsourced
- Having MLEs label data is expensive. But doing this for just a few days is usually fine
- Who is qualified to label
- Don't increase data by more than 10x at a time (if you have dataset of 1000k in first run, get 3x - 10x for next run)

13.2.7. Meta-data, Data Provenance and Lineage

Keep track of data provenance (where the data came from) and lineage (sequence of steps). Create extensive documentation. Meta-data is data about data. Gathering meta-data can be helpful at the error analysis stage and to keep track of data provenance.

13.2.8. Data Augmentation

Data Augmentation refers to modifying an existing training example to create a new training example usually used for unstructured data. How can you modify or warp or distort or make more noise in your data in a way so that what you get is similar to what you have in your test set. Goal of data augmentation is to create realistic examples that the algorithm does poorly on, but the humans (or other baseline) do well on.

Checklist:

- Does it sound realistic?
- Is the x -> y mapping clear? (e.g., can humans recognize speech?)
- Is the algorithm currently doing poorly on it?

For unstructured data: if the model is large and the mapping x->y is clear, then adding data rarely hurts accuracy. On the other hand if the model is small, then it might hurt the accuracy.

Data Synthesis refers to using artificial data inputs to create a new training example.

13.2.9. From Big Data to Good Data

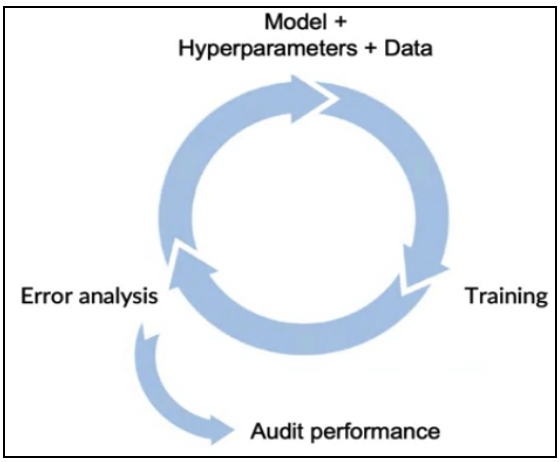
Try to ensure consistently high-quality data in all phases of the ML project lifecycle.

Good data:

- Covers important cases (good coverage of input x)
- Is defined consistently (definition of labels y is unambiguous)
- Has timely feedback from production data (distribution covers data drift and concept dring)
- Is sized appropriately

13.3. Modeling

13.3.1. Iterative Loop of ML Development



- Choose Architecture
- Train Model
- Diagnostics
- Repeat
- Audit Performance

Note: once the decision is made on which model will go to production, good practice would be doing final performance audit before pushing the model to production.
Make sure that a model does well on business metrics / project goals, not dev/test dataset.
Model-centric approach - work on code to improve accuracy
Data-centric approach - work on data quality to improve accuracy

13.3.2. Key Challenges

- Performance on disproportionately important examples
- Performance on key slices of the dataset (E.g. loan approval model: make sure not to discriminate by ethnicity, gender, location, language etc.)
- Skewed data distribution (99% negative case and 1% positive, cancer test result or fraudulent transaction)

13.3.3. Tips For Getting Started on ML Project

- Literature search to see what's possible (courses, blogs, open-source project)
- Find open-source implementations if available
- A reasonable algorithm with good data will often outperform a great algorithm with not so good data
- Take into account deployment constraints, if you plan to deploy the project
- Sanity check for code and algorithm before running on the large dataset

13.3.4. Building First System

Build your first system quickly, then iterate. Depending on the area of application, the guideline below will help you prioritize when you build your system.
Guideline:

1. Set up development/test set and metrics
   a. Set up a target
2. Build an initial system quickly
   a. Train training set quickly: Fit the parameters
   b. Development set: Tune the parameters
   c. Test set: Assess the performance
3. Use bias/variance analysis & error analysis to prioritize next steps

13.3.5. Error Analysis

Error analysis process refers to manually looking through prediction examples and trying to gain insights into where the algorithm is going wrong. Find a set of examples that the algorithm has misclassified examples from the cross validation set and group them into common themes or common properties or common traits. Error analysis is an iterative process just like model development. Create a spreadsheet and tag each case and find patterns.
Useful metrics for each tag:

- What fraction of errors has that tag?
- Of all data with that tag, what fraction is misclassified?
- What fraction of all the data has that tag?
- How much room for improvement is there on data with that tag?

Decide on most important categories to work on based on:

- How much room for improvement there is
- How frequently that category appears
- How easy is to improve accuracy in that category
- How important it is to improve in that category

Type	Accuracy	Human level performance	Gap to HLP	% of data
Clean Speech	94%	95%	1%	60% → 0.6%
Car Noise	89%	93%	4%	4% → 0.16%
People Noise	87%	89%	2%	30% → 0.6%
Low Bandwidth	70%	70%	0%	6% → ~0%

Correcting incorrect dev/test set 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.
- Train and dev/test data may now come from slightly different distributions.

For categories you want to prioritize:

- Collect more data
- Use data augmentation to get more data
- Improve label accuracy/data quality

13.3.6. Auditing Framework

Check for accuracy, fairness/bias, and other problems

1. Brainstorm the ways the system might go wrong
   a. Performance on subset of data (e.g., ethnicity, gender)
   b. How common are certain errors (e.g., FP, FN)
   c. Performance on rare classes
2. Establish metrics to assess performance against these issues on appropriate slices of data
3. Get business/product owner buy-in

13.3.7. Experiment Tracking

What to track?

- Algorithm/code versioning
- Dataset used
- Hyperparameters
- Results

Tracking tools

- Text files
- Spreadsheet
- Experiment tracking system (MLflow, Sage Maker Studio, Comet)

Desirable features

- Information needed to replicate results
- Experiment results, ideally with summary metrics/analysis
- Perhaps also: Resource monitoring, visualization, model error analysis

13.4. Deployment

13.4.1. Key Challenges

Concept drift occurs when the patterns the model learned no longer hold. In essence, the very meaning of what we are trying to predict evolves. Depending on the scale, this will make the model less accurate or even obsolete.
Data drift (feature drift, population, or covariate shift): the input data has changed. The distribution of the variables is meaningfully different. As a result, the trained model is not relevant for this new data.
Software engineering issues
Checklist of questions:

- Real-time or Batch
- Cloud vs Edge/browser
- Compute Resources (CPU/GPU/Memory)
- Latency, throughput (QPS)
- Logging
- Security and privacy

13.4.2. Key Ideas

- Gradual ramp up with monitoring
- Rollback

13.4.3. Common Deployment Patterns

Shadow mode where the ML system shadows the human and runs in parallel. The ML system's output is not used for any decisions drawing this phase. Helps to verify performance of the model.
Canary (ready to let model make decisions) Roll out to a small fraction, around 5%, of traffic initially. Monitor system and ramp up traffic gradually. Helps to spot problems early on.
Blue Green has both old (blue) and new (green) model versions and randomly has the router switch to predict through one of the models. Easy to roll back if something goes wrong.

13.4.4. Monitoring

The best way to monitor models in production is to build dashboards

- Brainstorm the things that could go wrong
- Brainstorm few statistics/metrics that will detect the problem

Examples of metrics

- Software metrics: Memory, compute, latency, throughput, server load
- Input metrics: Avg input length, avg input volume, num of missing values
- Output metrics: num of return null

Monitoring dashboards

- Set thresholds for alarms
- Adept metrics and thresholds over time

Model maintenance can be done through manual or automatic retraining.