Contents

1 Regression

Superscript (i) indicates relation to training example i.

1.1 Linear regression

1.1.1 Squared error cost function

Measures how well line fits training data
m = num of training examples
$x^{(i)}$ is training example x value i
$y^{(i)}$ is training example y value i
$\hat{y}^{(i)} = wx^{(i)} + b$

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

$\frac{1}{m}$ finds average error for larger data sets, $\frac{1}{2m}$ makes later calculations neater

1.1.2 Gradient descent

Find w, b for minimum of cost function $J(w,b)$

1. Start with some w, b (commonly $0, 0$)
2. Look around starting point and find direction that will move the point furthest downwards for a small step size

α = learning rate
Must simultaneously update w and b

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

1.2 Multiple linear regression

n_f = number of features
m = number of data points
$\vec{w}$ = vector of weights (length n_f)
$\vec{x}^{(i)}$ = vector of x values for training example i, length n_f
Sum of predictions of all features is the prediction of multiple linear reg

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

Gradient descent

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

Cost function and its partial derivatives

$$J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2$$
$$\frac{\partial}{\partial \vec{w}_j}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})\vec{x}_j^{(i)}$$
$$\frac{\partial}{\partial b}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})$$

1.3 Logistic regression

Sigmoid function

$$g(z) = \frac{1}{1+e^{-z}}$$
$$z = f_{\vec{w},b}(\vec{x})$$
$$\hat{y}^{(i)} = g(f_{\vec{w},b}(\vec{x}^{(i)}))$$

$\hat{y}^{(i)}$ can be interpreted as the "probability" that class is 1, $0 \leq \hat{y}^{(i)} \leq 1$
ex. $\hat{y}^{(i)} = 0.7$ means there is a 70% chance y is 1
Logistic regression requires a new cost function because $f_{\vec{w},b}(\vec{x})$ for logistic regression is non-convex, trapping gradient descend in local minima.
Cost function

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

$$L(\hat{y}^{(i)}, y^{(i)}) = \begin{cases} -\log(\hat{y}^{(i)}) & \text{if } y^{(i)} = 1 \\ -\log(1-\hat{y}^{(i)}) & \text{if } y^{(i)} = 0 \end{cases}$$

Simplified form

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

The loss function will decrease as $\hat{y}_i$ approaches $\vec{y}_i$ on a graph of L vs f.
$\frac{\partial J(\vec{w},b)}{\partial \vec{w}_j}$ and $\frac{\partial J(\vec{w},b)}{\partial b}$ are the same as in linear regression, just the definition of f has changed.

1.4 Softmax regression

Generalization of logistic regression, y can have more than two possible values.
The most probable value of y is the value that when given to L yields the largest loss.
Calculate $\vec{z}_i$ with $\vec{x}$ only consisting of data points that have label i. In implementation, set all y values of data points with label equal to i to 1, and 0 for everything else.
n_f = num features
n_y = number of possible y outputs
W is a matrix of dimensions $n_y \times n_f$.
$\vec{b}$, $\vec{z}$, $\vec{a}$ are vectors of length n_y.

$$1 \leq i \leq n_y$$
$$\vec{z}_i = W_i \cdot \vec{x} + \vec{b}_i$$
$$\vec{a}_i = \frac{e^{\vec{z}_i}}{\sum_{k=1}^{n_y} e^{\vec{z}_k}}$$

$$L(\vec{a}, y) = \begin{cases} -\log \vec{a}_1 \text{ if } y = 1 \\ -\log \vec{a}_2 \text{ if } y = 2 \\ \vdots \\ -\log \vec{a}_n \text{ if } y = n \end{cases} \tag{1}$$

1.5 Feature scaling: z-score normalization

After z-score normalization, all features will have a mean of 0 and a standard deviation of 1
n_f = num features
$\vec{\mu}_j$ = mean of all values for feature j (length n_f)
$\vec{\sigma}_j$ = standard deviation of feature j (length n_f)

$$\vec{x}_j^{(i)} = \frac{\vec{x}_j^{(i)} - \vec{\mu}_j}{\vec{\sigma}_j}$$
$$\vec{\mu}_j = \frac{1}{m}\sum_{i=1}^{m} \vec{x}_j^{(i)}$$
$$\vec{\sigma}_j^2 = \frac{1}{m}\sum_{i=1}^{m} (\vec{x}_j^{(i)} - \vec{\mu}_j)^2$$

1.6 Over / underfitting

Underfit / high bias: does not fit training set well ($wx + b$ fit onto data points with $x + x^2$ shape)

Overfit / high variance: fits training set extremely well but does not generalize well ($w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$ fit onto training set of shape $x + x^2$ can have zero cost but predicts values outside the training set inaccurately)

Addressing overfitting

- Collect more data
- Select features ("Feature selection")
- Reduce size of parameters ("Regularization")

1.6.1 Regularization

Small values of $w_1, w_2, \cdots, w_n, b$ for simpler model, less likely to overfit

Given n_f features, there is no way to tell which features are important and which features should be penalized, so all features are penalized.

$$J_r(\vec{w}, b) = J(\vec{w}, b) + \frac{\lambda}{2m} \sum_{j=1}^{n_f} \vec{w}_j^2$$

Can include b by adding $\frac{\lambda}{2m} b^2$ to J_r but typically doesn't make a large difference.

The extra term in J_r is called the regularization term.

Effectively, $\lambda \propto \frac{1}{w}$. When trying to minimize cost, either the error term or the regularization term must decrease. The larger the lambda, the more the regularization term should decrease to minimize cost, decreasing w parameters.

Regularized linear regression

$$J_r(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left[(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2 \right] + \frac{\lambda}{2m} \sum_{j=1}^{n_f} \vec{w}_j^2$$

For gradient descent, only $\frac{\partial J_r}{\partial \vec{w}_j}$ changes (b is not regularized):

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

Regularized logistic regression

$$J_r(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) + \frac{\lambda}{2m} \sum_{j=1}^{n_f} \vec{w}_j^2$$

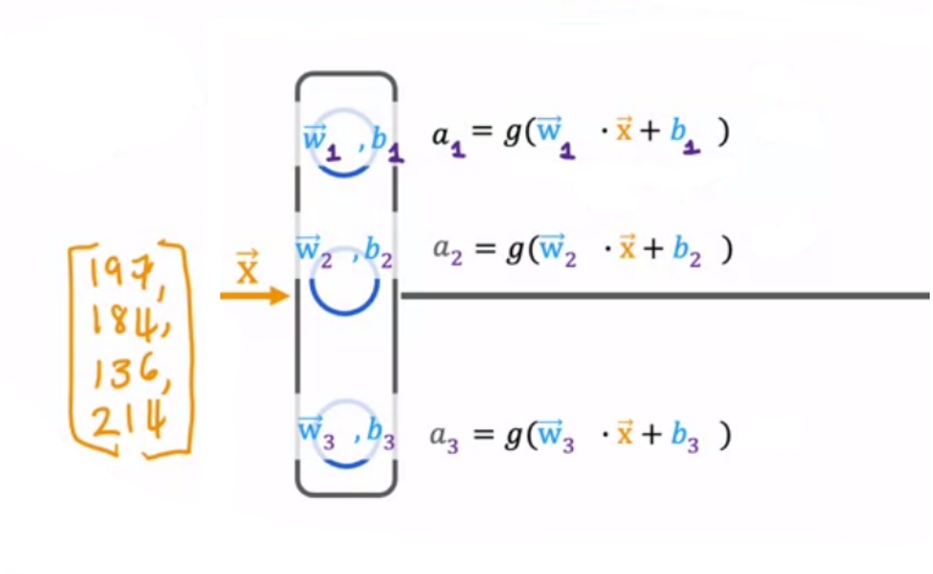
For gradient descent, only $\frac{\partial J_r}{\partial w_j}$ changes (b is not regularized):

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

2 Neural networks

2.1 Intuition

The input layer, or the data, is layer 0. Hidden layers start from layer 1, and the output layer is the last layer. Superscript square brackets are commonly used to refer to some layer; for example, $\vec{a}^{[i]}$ refers to the vectors of activations in layer i.



where g is some activation function (ex. sigmoid). $\vec{a}$ will be the output of this layer, which is a vector of the activations $[a_1, a_2, a_3]$ from the previous layer.

A more compact way of expressing the activations in a layer is $a_j^{[\ell]} = g(\vec{w}_j^{[\ell]} \cdot \vec{a}^{[\ell-1]} + b_j^{[\ell]})$.

2.2 Activation functions

Linear: $g(z) = z$, used for ex. change in stock prices tomorrow
Sigmoid: $g(z) = \frac{1}{1+e^{-z}}$, used for binary classification
ReLU: $g(z) = \max(0, z)$, used for non-negative predictions ex. housing prices
When choosing $g(z)$ for the output layer:

- Sigmoid for binary classification
- Linear for linear regression with both positive / negative results
- ReLU is similar to linear but if output values can only be non-negative

Choosing $g(z)$ for hidden layers:

- ReLU is the most common (faster, doesn't flatten out like sigmoid)
- Linear is typically not used in hidden layers, because they remove complexity from the model's predictions. A neural network with many layers that all use the linear activation function is not any better than a single regression using the activation function of the output layer.

2.3 Training the model

Derivatives can be computed by changing a function's argument by some small ϵ and seeing how the function changes as a reuslt of its argument change. The derivative is then change in function / change in argument.

For example, given cost function $J(w) = w^2$, $\frac{\partial J}{\partial w}\Big|_{w=3}$ can be determined with $\frac{J(w+\epsilon)-J(w)}{\epsilon}$. Values for ϵ can be as small as 0.0001, because ϵ should simulate an infinitely small number.

2.3.1 Forward propagation

$n^{[\ell]}$ = number of neurons in layer ℓ
$n_f = n^{[0]}$ = num features
m = number of examples

$Z^{[\ell]}$ is a matrix of dimensions $n^{[\ell]} \times m$ for $\ell > 0$ which holds the values to pass to the activation function of layer ℓ. The $n^{[\ell]}$ rows hold one feature of the m data points in each neuron in layer ℓ. Not applicable for input layer.

$A^{[\ell]}$ is a matrix of dimensions $n^{[\ell]} \times m$ for $\ell > 0$ which holds activation values that come from $g^{[\ell]}(Z^{[\ell]})$.

$g^{[\ell]}$ is the activation function for layer ℓ.

$W^{[\ell]}$ is a matrix of dimensions $n^{[\ell]} \times n^{[\ell-1]}$ for $\ell > 0$, which holds $\boldsymbol{w}$ values for each neuron in layer ℓ. Not applicable for input layer.

$\boldsymbol{b}^{[\ell]}$ is a list of b values for each neuron in layer ℓ, length $n^{[\ell]}$.

$A^{[0]} = X$ is a matrix of dimensions $n_f \times m$ which holds all input data.

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

2.3.2 Back propagation

Including variables from the forward propagation section:
$L =$ last layer
Y is a matrix of dimensions $n^{[L]} \times m$ which holds training data labels.
$\mathrm{d}Z_i^{[\ell]}$ is the ith column of $\mathrm{d}Z$, giving a vector of length $n^{[\ell]}$.
For output layer:

$$\mathrm{d}Z^{[L]} = A^{[L]} - Y$$
$$\mathrm{d}W^{[L]} = \frac{1}{m}\,\mathrm{d}Z^{[L]} A^{[L-1]T}$$
$$\mathrm{d}\boldsymbol{b}^{[L]} = \frac{1}{m}\sum_i \mathrm{d}Z_i^{[L]}$$

For hidden layers:

$$\mathrm{d}Z^{[\ell]} = W^{[\ell+1]T}\,\mathrm{d}Z^{[\ell+1]} * g^{[\ell]\prime}(Z^{[\ell]})$$
$$\mathrm{d}W^{[\ell]} = \frac{1}{m}\,\mathrm{d}Z^{[\ell]} A^{[\ell-1]T}$$
$$\mathrm{d}\boldsymbol{b}^{[\ell]} = \frac{1}{m}\sum_i \mathrm{d}Z_i^{[\ell]}$$

2.3.3 Back propagation derivation

Using loss function $L(a, y) = -y \log a - (1-y)\log(1-a)$:
dz (output layer)

$$\frac{dL}{dz} = \frac{dL}{da}\frac{da}{dz}$$
$$\frac{da}{dz} = \frac{d}{dz}g(z) = g'(z)$$
$$\frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$
$$\frac{dL}{dz} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)(a(1-a)) = a - y$$

dz (hidden layer)

$$\frac{dL}{dz^{[l]}} = \frac{dL}{dz^{[l+1]}}\frac{dz^{[l+1]}}{dz^{[l]}}$$
$$z^{[l+1]} = w^{[l+1]}a^{[l]} + b^{[l+1]}$$
$$\frac{dz^{[l+1]}}{dz^{[l]}} = w^{[l+1]}\frac{da^{[l]}}{dz^{[l]}}$$
$$\frac{dL}{dz^{[l]}} = \frac{dL}{dz^{[l+1]}}w^{[l+1]}\frac{da^{[l]}}{dz^{[l]}} = \frac{dL}{dz^{[l+1]}}w^{[l+1]}\sigma'(z^{[l]})$$

which in simplified notation is $dz^{[l]} = dz^{[l+1]}w^{[l+1]}\sigma'(z^{[l]})$.
dw

$$\frac{dL}{dw^{[l]}} = \frac{dL}{dz^{[l]}}\frac{dz^{[l]}}{dw^{[l]}} = dz^{[l]}a^{[l-1]}$$

because

$$z = w \cdot a + b \rightarrow \frac{dz}{dw} = a$$

db

$$\frac{dL}{db^{[l]}} = \frac{dL}{dz^{[l]}}\frac{dz^{[l]}}{db^{[l]}} = dz^{[l]}$$

2.4 Convolutional neural network

2.4.1 Edge detection

Starting with vertical edge detection: Vertical edges can be detected by applying a "filter" to an image. ex. with a 6x6 grayscale image convolved with a 3x3 filter:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \end{pmatrix}$$

To convolve the image, place the top left corner of the filter in the top left corner of the image. The sum of the element wise multiplication between the filter and the area of the image it covers is the top left element of the output matrix. Shift the filter one over and repeat, and when the right edge of the filter reaches the right edge of the image shift the filter down one and to the left edge of the image and repeat the process again.

The filter shown only detects left to right light to dark correctly, and given an image where the colors are inversed, the output matrix would have inversed values as well.

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & -3 & -3 & 0 \\ 0 & -3 & -3 & 0 \\ 0 & -3 & -3 & 0 \\ 0 & -3 & -3 & 0 \end{pmatrix}$$

A convolutional neural network would be able to learn values for the filter that would be more optimal than hand-coded ones:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix}$$

2.4.2 Padding

An issue with convolution is that the image shrinks every convolution, and edge information is used much less than center information because more areas overlap with center pixels than edge pixels. Given input image of size $n \times n$ and filter of size $f \times f$, the output image size is $(n - f + 1) \times (n - f + 1)$.

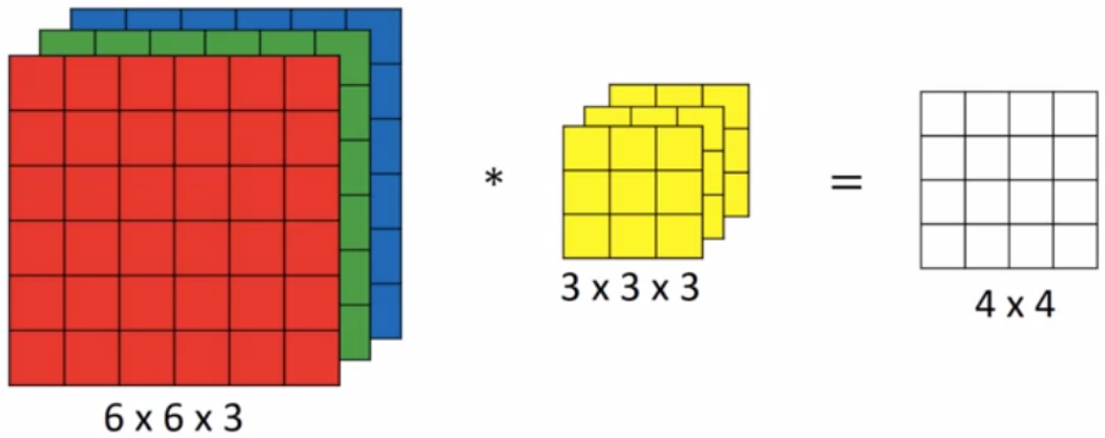
Padding is just adding extra border pixels around the image which are usually set as 0, with thickness denoted as p. The image has new dimensions $(n + 2p) \times (n + 2p)$ for any $p > 0$, so to get the original image back $2p = f - 1 \rightarrow p = (f - 1)/2$. Because of the division by 2, f is usually odd.

2.4.3 Strided convolution

Instead of only taking one step at a time when moving the filter across the image, the filter moves with some set stride (ex. 2 instead of the default 1)

The output image dimensions in strided convolution are $(\frac{n+2p-f}{s} + 1) \times (\frac{n+2p-f}{s} + 1)$, and the padding to ensure same output dimensions is $p = \frac{(n-1)s+f-n}{2}$.

2.4.4 Convolution over a volume



Essentially the same as convolution over a 2D matrix. The number of channels in the filter and the image have to be the same (3 in this case).

Multiple filters can be used by convolving the input image with each filter and then stacking all the output images together. For example, given two filters and one input image, the output matrix size will be 4x4x2, because there are two different filtered 2D output matrices.

2.4.5 One layer of a cnn

Recall from forward prop:

$$Z^{[1]} = W^{[1]}A^{[0]} + \boldsymbol{b}^{[1]}$$
$$A^{[1]} = g(Z^{[1]})$$

where $A^{[0]}$ is analogous to X. The input 6x6x3 image is X, the filter is W, $\boldsymbol{b}$ is added to the output 4x4 matrix (element wise) which becomes Z, and then some activation function g is applied to that Z, giving the current layer's A. The number of filters can be seen as the number of features.

2.4.6 Cnn notation

Superscript $[l]$ means relating to layer l.
$f^{[l]}$ = filter size ($f \times f$)
$p^{[l]}$ = padding thickness
$s^{[l]}$ = stride
Input dimensions are $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$.
Output dimensions are $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$.
$n^{[l]} = \frac{n^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}} + 1$, which is applicable to n_H and n_W.
Each filter is $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
Activations $a^{[l]}$ are $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$.
Vectorized: $A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
Weights $w^{[l]}$ are $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
Vectorized: $W^{[l]} = f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
Bias $\boldsymbol{b}$ is just a vector of length $n_c^{[l]}$, though it becomes more convenient to represent it as $1 \times 1 \times 1 \times n_c^{[l]}$ later.

2.4.7 Example cnn (not vectorized)

Input image $39 \times 39 \times 3$, first layer uses a set of 10 3×3 filters, second layer uses a set of 20 5×5 filters, third layer uses 40 5×5 filters. All layers use no padding, and stride is chosen arbitrarily.

Input image is $39 \times 39 \times 3$, so $n_W^{[0]} = n_H^{[0]} = 39$ and $n_c^{[0]} = 3$.

For first layer, $f^{[1]} = 3$, $s^{[1]} = 1$, $p^{[1]} = 0$. Because there are 10 filters, there will be 10 channels in the first layer output, or $n_c^{[1]} = 10$, and the dimensions of a single channel in the first layer's output is calculated with $n^{[l]} = \frac{n^{[l-1]}+2p^{[l]}-f^{[l]}}{s^{[l]}} + 1$, which gives 37 for both width and height. The activations will then be $37 \times 37 \times 10$.

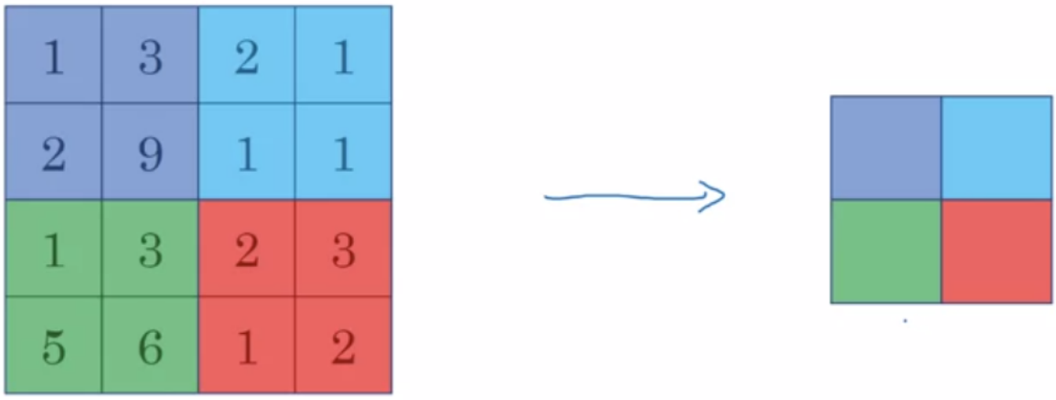
For the second layer, $f^{[2]} = 5$, $s^{[2]} = 2$, $p^{[2]} = 0$, $n_c^{[2]} = 20$. The output volume will then have dimensions of $17 \times 17 \times 20$, where 17 comes from the n_W and n_H calculation and 20 is n_c.

For the third layer, $f^{[3]} = 5$, $s^{[2]} = 2$, $p^{[2]} = 0$. The output volume then has dimensions of $7 \times 7 \times 40$.

After the last layer of filters has been processed, flatten out the final output volume (in this case the $7 \times 7 \times 40$ volume) and feed it to a logistic/softmax regression.

A general trend is that the width and height of each channel in a volume of some layer will be smaller than the previous layer's, while the number of channels increases as the layers go on.

2.4.8 Pooling layers



Max pooling takes the largest number in each shaded region and puts it in the corresponding color in the resulting 2×2 matrix. The hyperparameters can be represented as $f = 2$, $s = 2$.

Max pooling can be helpful because it identifies important features in an image while also reducing its size, which is computationally less expensive when running the cnn.

Average pooling takes the average of all the values in each shaded region instead of the largest.

In pooling layers, there are no parameters to learn, the only parameters a pooling layer has is its hyperparameters f and s.

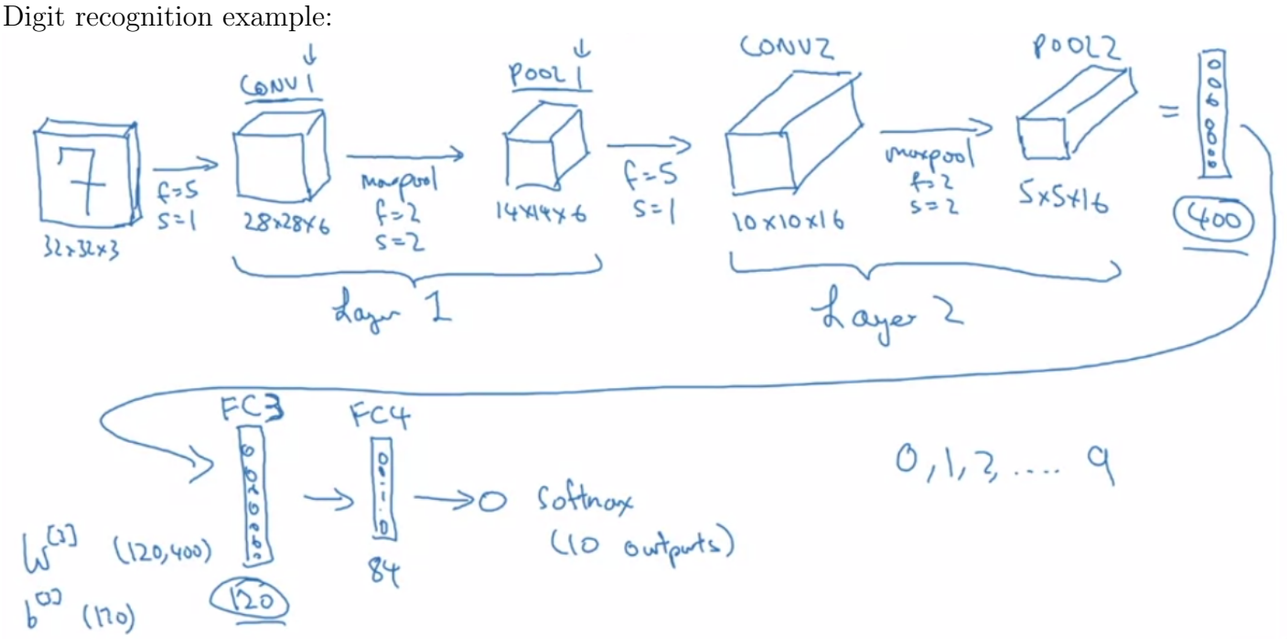
Convolutional and pooling layers are usually grouped together into a single layer because the pooling layer has no parameters to learn.

2.4.9 Fully connected layer

A fully connected layer is just a regular deepnn layer. W dimensions $n^{[l]} \times n^{[l-1]}$, $\boldsymbol{b}$ dimensions $n^{[l]}$.

2.4.10 Complete cnn example

Digit recognition example:



2.4.11 Forward prop

eth example, nth layer l channel, jth layer $l-1$ channel

$$Z_{en}^{[l]} = \sum_{c=0}^{n_c^{[l-1]}} \text{convolve}(A_{ec}^{[l-1]}, W_{cn}^{[l]}) + \boldsymbol{b}_n^{[l]}$$

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

2.4.12 Why convolutions

Each filter only needs to learn its own parameters, and then can be applied anywhere on the data (for example, in edge detection, a vertical edge filter is able to detect vertical edges anywhere on the image after only learning a few parameters.

Each output depends on less inputs compared to a normal neural network, which improves performance.

2.5 Improving model

Cross validation: split data into training and test, use test data to determine how well the model generalizes

2.5.1 Fixing high bias/variance

High bias (underfit): J_{train} high, $J_{train} \approx J_{cv}$
High variance (overfit): J_{train} may be low, $J_{cv} \gg J_{train}$
High bias and high variance: J_{train} high, $J_{cv} \gg J_{train}$
How to fix:

1. Get more training examples (fix high variance)
2. Try smaller sets of features (fix high variance)
3. Add more features (fix high bias)
4. Add polynomial features (fix high bias)
5. Decrease λ (fix high bias)
6. Increase λ (fix high variance)

Neural networks and bias/variance
If J_{train} is high, make the network larger
If J_{cv} is high, get more data

2.5.2 Adding data

Data augmentation: add data with distortions (ex. distorted letters in a letter recognition program)

3 Cnn back prop

A^l and Z^l have dimensions $m \times n_c^l \times n_h^l \times n_w^l$
P^l is the pooled A^l with dimensions $m \times n_c^l \times n_{ph}^l \times n_{pw}^l$, and is equal to A^l if there is no pooling.
F^l is the flattened P^l with length $mn_c^l n_{ph}^l n_{pw}^l$
$0 \leq e \leq m$
$0 \leq n \leq n_c^l$
$0 \leq u \leq n_H^l$
$0 \leq v \leq n_W^l$
$n_{ph}^l = n_H^l / p_H^l$
$n_{pw}^l = n_W^l / p_W^l$
p_H^l = pooling height in layer l
p_W^l = pooling width in layer l
Assume that all pooling layers use max pooling, and that the last layer L is a dense layer with a single softmax neuron.

3.1 Deriving $\frac{dL}{db}$ for forward feeding into dense layer

Using chain rule:

$$\frac{dL}{db_n^l} = \sum_{u=0}^{n_H^l} \sum_{v=0}^{n_W^l} \frac{dL}{dZ_{enuv}^l} \frac{dZ_{enuv}^l}{db_n^l}$$

$$\frac{dL}{dZ_{enuv}^l} = \frac{dL}{dA_{enuv}^l} \frac{dA_{enuv}^l}{db_n^l}$$

3.1.1 $\frac{dL}{dA_{enuv}^l}$

If A_{enuv}^l is the max out of the elements in its pool, or there is no pooling layer:
$P_{enyx}^l = A_{enu_{max}v_{max}}^l$, where $x = (v_{max}/p_W)$, $y = (u_{max}/p_H)$

$$\frac{dL}{dA_{enuv}^l} = \frac{dL}{dA_{enu_{max}v_{max}}^l} = \frac{dL}{dP_{enyx}^l}$$

$F_k^l = P_{enyx}^l$, where $k = (n_c^l n_{pm}^l n_{pw}^l)e + (n_{ph}^l n_{pw}^l)n + n_{pw}^l y + x$

$$\cdots = \frac{dL}{dP_{enyx}^l} = \frac{dL}{dF_k^l}$$

Since $\frac{dL}{dA^l_{enu_{max}v_{max}}} = \frac{dL}{dF_k}$, it might be helpful to calculate $\frac{dL}{dF_k}$:

$$\frac{dL}{dF^l_k} = \sum_{i=0}^{n^L} \frac{dL}{dZ^L_{ei}} \frac{dZ^L_{ei}}{dF^l_k}$$

From $Z^L_{ei} = \sum_{j=0}^{k_{max}} (W^L_{ij} F^l_j) + \boldsymbol{b}_i$:

$$\frac{dZ_{ei}}{dF^l_k} = W_{ik}$$
$$\frac{dL}{dF^l_k} = \sum_{i=0}^{n^L} \frac{dL}{dZ^L_{ei}} W_{ik}$$

With $\frac{dL}{dF^l_k}$ solved, it is now possible to solve for $\frac{dL}{dA^l_{enuv}}$ in the case that dA^l_{enuv} is the largest in its pool.

$$\frac{dL}{dF^l} = W^{lT} \cdot \frac{dL}{dZ^L}$$
$$\frac{dL}{dP^l} = \frac{dL}{dF^l}.\text{reshape}(P^l.\text{shape})$$

If A^l_{enuv} is not the largest in its pool:

This part is not applicable for when there is no pooling done.

Because P only holds the maximum value in each pool for A, changes in A^l_{enuv} when A^l_{enuv} is not the largest value will not affect dL/dP and therefore dL/dA, which gives

$$\frac{dL}{dA^l_{enuv}} = \begin{cases} \frac{dL}{dP^l_{enyx}} \text{ if } A^l_{enuv} \text{ is the largest in its pool} \\ 0 \text{ otherwise} \end{cases}$$

Calculating $\frac{dL}{dA^l_{enuv}}$

In forward propagation, cache n, u_{max}, and v_{max} into some array I with the same shape as P^l, which can be accessed with $I[n][y][x] = \begin{bmatrix} u_{max} \\ v_{max} \end{bmatrix}$.

Calculate $\frac{dL}{dP^l}$ using

$$\frac{dL}{dF^l} = W^{lT} \cdot \frac{dL}{dZ^L}$$
$$\frac{dL}{dP^l} = \frac{dL}{dF^l}.\text{reshape}(P^l.\text{shape})$$

and then iterate over all inputs to $\frac{dL}{dP^l}$ (n, y, x) and use

$$\frac{dL}{dA^l_{enu_{max}v_{max}}} = \frac{dL}{dP^l_{enyx}}$$

where u_{max} and v_{max} are fetched from $I[n][y][x]$.

For every $u \neq u_{max}$ and $v \neq v_{max}$, set $\frac{dL}{dA^l_{enuv}}$ to 0.

3.1.2 $\frac{dA^l_{enuv}}{dZ^l_{enuv}}$

$$A = g(Z)$$
$$\frac{dA^l_{enuv}}{dZ^l_{enuv}} = g'(Z^l_{enuv})$$

3.1.3 $\frac{dZ^l_{enuv}}{db^l_n}$

From $Z = \text{convolve}(P^{l-1}, W^l) + b^l$ in forward prop:

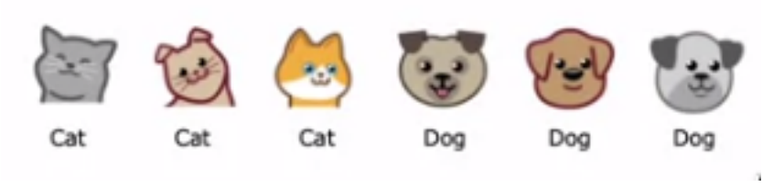
$$\frac{dZ^l_{enuv}}{db^l_n} = 1$$

4 Decision trees

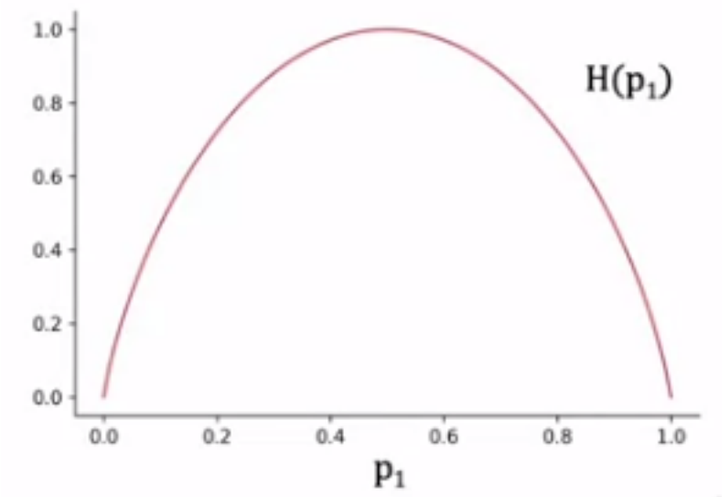
4.1 Measuring purity

4.1.1 Entropy as a measure of impurity

p = fraction of examples that are cats



$p = \frac{1}{2}$
Impurity can be measured with the entropy function $H(p)$

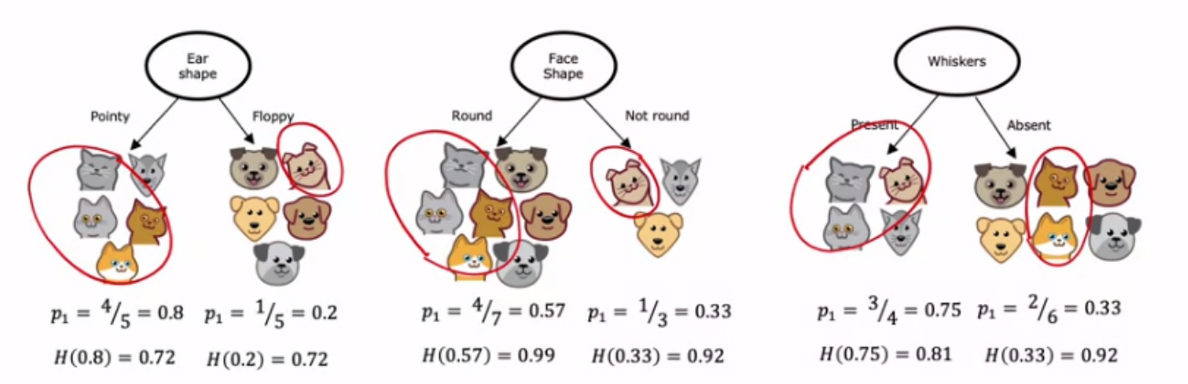


Higher H = less pure, more information gain
Mathematically, H is defined as:

$$H(p) = -p\log_2(p) - (1-p)\log_2(1-p)$$

$0\log(0)$ is defined as 0 for the function H

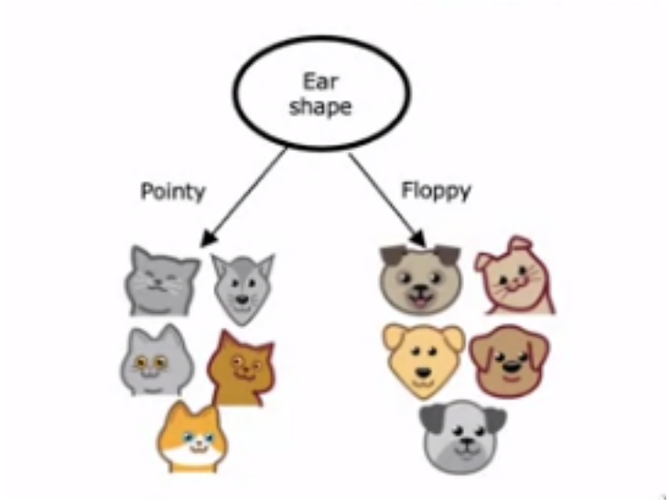
4.2 Choosing a split



To choose which feature to split data on is the best, calculate the weighted average of the entropy on the left and right branches, then choose which split has the highest entropy (least pure, which will give a good split).

Average of ear shape split entropy: $0.5H(0.8) + 0.5H(0.2) = 0.28$
Average of face shape split entropy: $0.7H(0.57) + 0.3H(0.33) = 0.03$
Average of whiskers split entropy: $0.4H(0.75) + 0.6H(0.33) = 0.12$
Ear shape has the largest entropy, so the best choice is to split based on ear shape.

Formal definition of information gain:

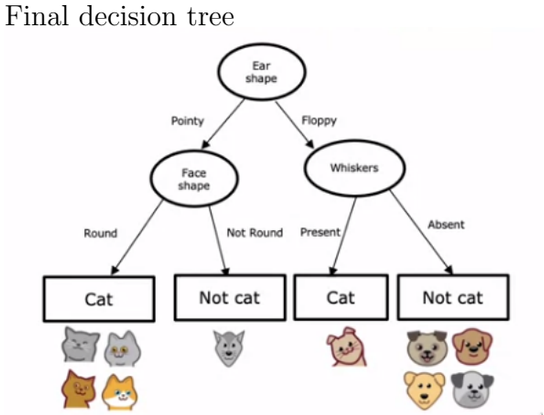


p_{root} = percentage of positive examples (0.5 in this case)
$p_{left} = 4/5$
$p_{right} = 1/5$
$w_{left} = 5/10$
$w_{right} = 5/10$
$H(p_{root}) - (w_{left}H(p_{left}) + w_{right}H(p_{right}))$

4.3 Constructing a decision tree

1. Start with all examples at root node
2. Calculate information gain for all possible features, pick one with highest information gain
3. Split dataset according to selected feature, creating a left and right branch
4. Stop when stopping criteria is met (node is 100% one class, information gain from more splits is less than a threshold, num examples is below a threshold)

Final decision tree



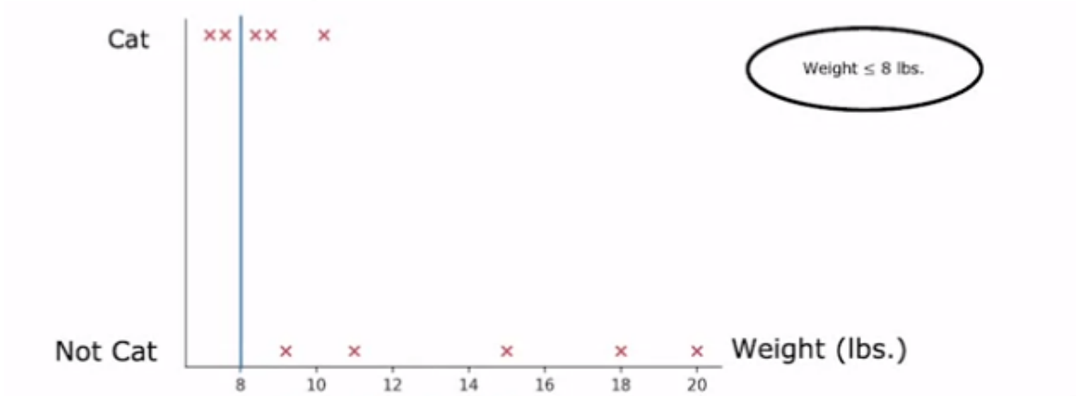
4.4 Features with multiple possible values

Known number of possible values

If a categorical feature can take on k values, create k binary features

ex. Create a true/false feature for pointy ears, floppy ears, and oval ears instead of a single feature "ear shape" which takes on three possible values.

Unknown number of possible values



Split on weight ≤ 8 lbs
$p_{root} = 0.5$
$p_{left} = 1$
$p_{right} = 3/8$

$w_{left} = 2/10$
$w_{right} = 8/10$
$H(0.5) - (\frac{2}{10}H(1) + \frac{8}{10}H(\frac{3}{8}))$

To find most optimal information gain (maximize H), make splits between every pair of adjacent data points and choose the one with the highest information gain.

4.5 Tree ensembles

Training multiple decision trees will lead to more accurate predictions since a single decision tree is sensitive to small changes in data.

4.5.1 Sampling with replacement

Take original training set of size m and randomly select from the original training set to create a new training set of size m. Repeated data is expected.

This will create new datasets that are similar to the original dataset, but are slightly different which will create unique decision trees.

4.5.2 Random forest algorithm

From $b = 1$ to B: sampling with replacement to create new dataset, train decision tree on new dataset.

B is commonly around 100. Setting B too large doesn't hurt performance but gives diminishing returns as it increases.

Randomizing feature choice is another way to create more unique decision trees: Given n features, give each decision tree a subset of all features of size k.

A good value for k is $k = \sqrt{n}$.

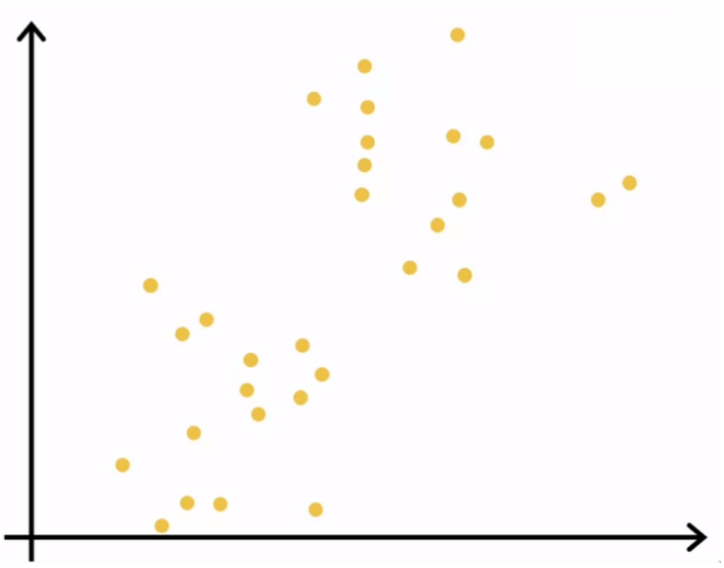
4.5.3 XGBoost

Instead of picking from all training data with equal probability in the random forest algorithm, make it more likely to pick misclassified examples from previous decision trees

5 Unsupervised learning

5.1 Clustering: K-means

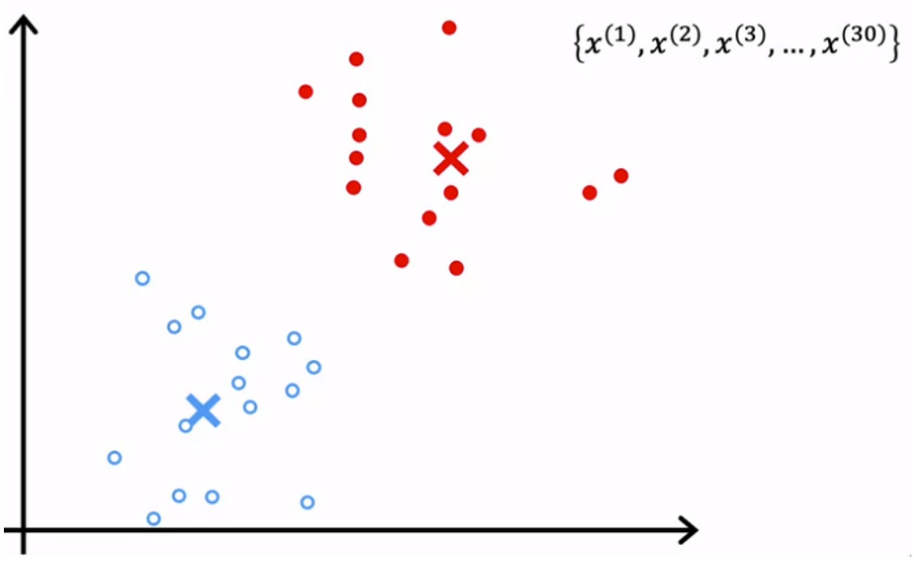
5.1.1 Algorithm



Given a dataset like this, the algorithm will guess the centers of two different clusters (Determining number of clusters will be covered later).

Once two cluster centers (or centroids) are guessed, each data point on the graph will be associated with the centroid it's closest to. The centroid will then move to the average position of all its data points.

Eventually, the centroids will move to the center of the two clusters:



5.1.2 Cost function

k = num clusters
$\vec{c}_i$ = index of cluster (1,2,...,k) to which example $\vec{x}_i$ is currently assigned
μ_i = cluster centroid i
$\mu_{\vec{c}_i}$ = cluster centroid of cluster to which example $\vec{x}_i$ is currently assigned to

$$J(\vec{c}, \vec{\mu}) = \frac{1}{m} \sum_{i=1}^{m} \|\vec{x}_i - \vec{\mu}_{\vec{c}_i}\|^2$$

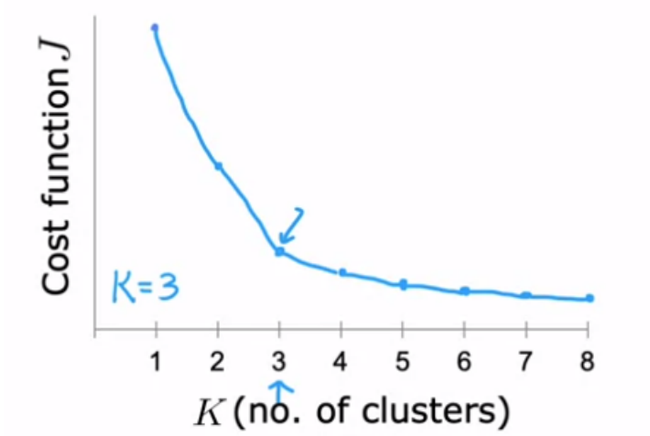
The cost function can be used to determine how well the centroids predicted the clusters, and it can also determine when k-means is converging.

The most optimal centroids can be determined by running k-means multiple times with random initial centroid positions every time, then choosing the result with the lowest cost.

5.1.3 Choosing k

Elbow method

Plot cost as a function of k, choose k where cost begins to decrease at a slow rate.



In this graph, $k = 3$ might be a good number of clusters. Although cost does continue to decrease as k increases beyond 3, the number of clusters is too large and makes for less meaningful clusters.

The "right" value of k is often ambiguous however, which is an issue with the elbow method.

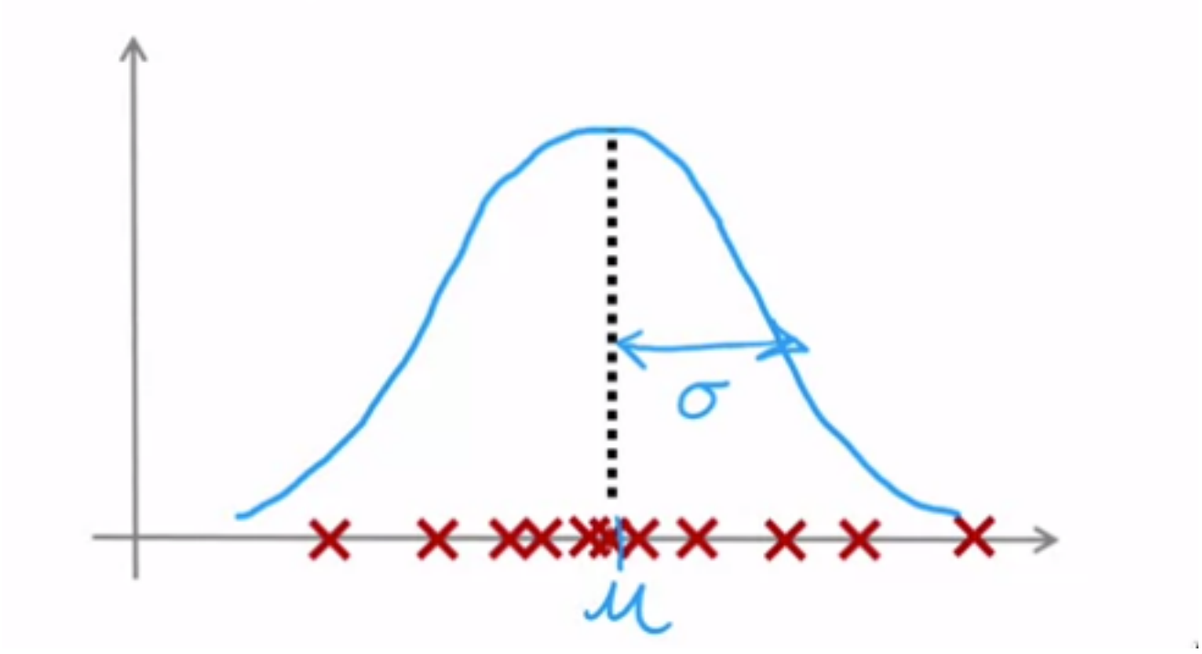
5.2 Anomaly detection

5.2.1 Normal distribution

The equation for the normal distribution is given by

$$p(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Change in μ will shift the curve on the x axis, and change in σ will make the curve thinner or wider. Smaller σ makes curve narrow, larger σ makes curve wide.



Values of μ and σ that produce a normal distribution which will fit the data well can be determined like this:

$$\mu = \frac{1}{m}\sum_{i=1}^{m}\vec{x}_i$$
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(\vec{x}_i - \mu)^2$$

5.2.2 Density estimation

Training set: matrix X of size $m \times n$ (m examples, n features)

$$p(\vec{x}) = \prod_{i=1}^{n} p(\vec{x}_i, \mu_i, \sigma_i^2)$$

5.3 Recommender systems

n = num features
n_i = num items
$r(i,j) = 1$ if user j has rated item i (0 if otherwise)
$y_{i,j}$ = rating given by user j on item i (if defined)
$W_j, \vec{b}_j$ = parameters for user j
X_i = feature vector for item i
$\vec{m}_j$ = number of items user j has rated
For user j, predict rating of item i with $W_j \cdot X_i + \vec{b}_j$
Feature example:

Movie	X_{i1} (Romance)	X_{i2} (Action)	X_{i3} (Horror)
Romance movie ($i = 1$)	1.0	0.1	0.0
Action movie ($i = 2$)	0.0	1.0	0.0
Comedy movie ($i = 3$)	0.5	0.0	0.0
Horror movie ($i = 4$)	0.0	1.0	1.0

Cost function to learn parameters for user j:

$$J(W_j, \vec{b}_j) = \frac{1}{2\vec{m}_j}\sum_{i:r(i,j)=1}(\vec{w}_j \cdot X_i + \vec{b}_j - y_{i,j})^2$$

Learn parameters for all users n_u:
W is a matrix of size $n_u \times n$
$\vec{b}$ is a vector of length n_u

$$J(W, \vec{b}) = \frac{1}{2}\sum_{j=1}^{n_u}\left[\sum_{i:r(i,j)=1}(\vec{w}_j \cdot X_i + \vec{b}_j - y_{i,j})^2\right]$$

5.3.1 Collaborative filtering

Given user parameters $\vec{w}$ and b, predict features.
To learn X_i:

$$J(X_i) = \frac{1}{2} \sum_{j:r(i,j)=1} (\vec{w}_j \cdot X_i + \vec{b}_j - y_{i,j})^2$$

To learn $X_{1...n_m}$:

$$J(X) = \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} (\vec{w}_j \cdot X_i + \vec{b}_j - y_{i,j})^2$$

5.3.2 Cost function

$$J(\vec{w}, \vec{b}, X) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} (\vec{w}_j \cdot X_i + \vec{b}_j - y_{i,j})^2$$

5.3.3 Binary labels

Predict probability of $y_{i,j} = 1$ using $a_{i,j} = g(\vec{w}_j \cdot X_i + \vec{b}_j)$ where $g(z) = \frac{1}{1+e^{-z}}$
Loss for binary labels

$$L(a_{i,j}, y_{i,j}) = -y_{i,j} \log(a_{i,j}) - (1 - y_{i,j}) \log(1 - a_{i,j})$$

Cost function

$$J(\vec{w}, \vec{b}, X) = \sum_{(i,j):r(i,j)=1} L(g(\vec{w}_j \cdot X_i + \vec{b}_j), y_{i,j})$$

5.3.4 Mean normalization

Allows the algorithm to give better predictions for users who have rated very few movies.
Given a chart like this,

Movie	Alice(1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at last	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	?	?

create a matrix

$$\begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & ? & ? \end{bmatrix}$$

$$\mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

After subtracting every value in each row by its corresponding μ:

$$\begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

5.3.5 Finding related items

Find item k with X_k similar to X_i

$$\sum_{l=1}^{n}(X_{k,l} - X_{i,l})^2$$

5.3.6 Content based filtering

Recommends items user might like, in contrast to collaborative filtering which decides if a user would like an item

5.4 Reinforcement learning

5.4.1 State action value function

Denoted by $Q(s, a)$ where s is the current state and a is an action that might be taken in that state.