Deep Learning Specialization

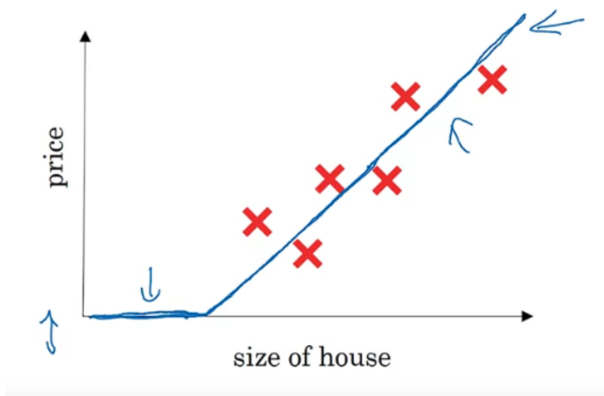
Declan Lim

August 17, 2022

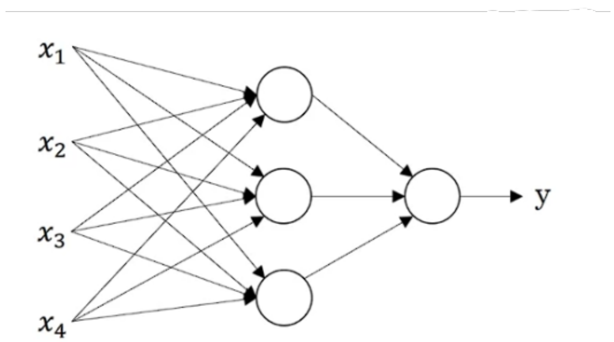
1 Neural Networks and Deep Learning

1.1 Introduction to Deep Learning

- Takes input x to a "neuron" and gives some output y



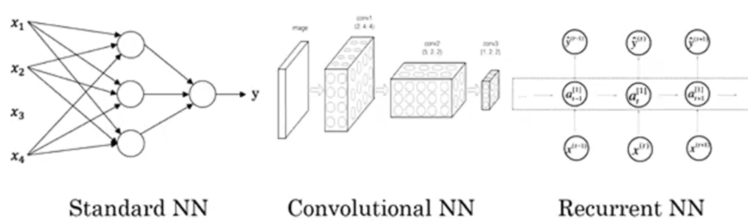
  - Simple neural network has a single input, neuron and output
  - x: size of the house
  - y: price of the house
  - Hypothesis (blue line) is a ReLU (Rectified Linear Unit)
- More complex neural networks can be formed by "stacking" neurons



- Every input layer feature is interconnected with every hidden layer feature
  - The neural network will decide what the intermediate features will be
- Most useful in supervised learning settings

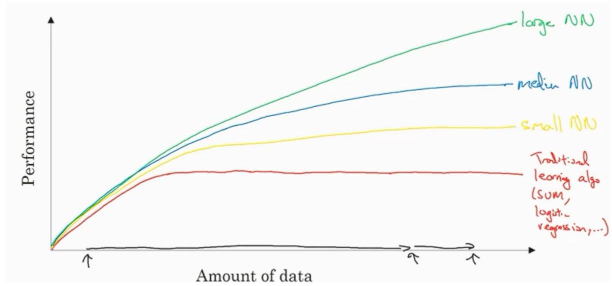
1.1.1 Supervised Learning

- Aims to learn a function to map an input x to an output y
  - Real estate: predicting house prices from the house features
  - Online advertising: showing ads based on probability of user clicking on ad
  - Photo tagging: tagging images based on objects in the image
  - Speech recognition: generating a text transcript from audio
  - Machine translation: translating from one language to another
  - Autonomous driving: returning the positions of other cars from images and radar info
- Different types of neural network used for different tasks
  - Standard neural network: real estate and online advertising
  - Convolutional neural network (CNN): image data
  - Recurrent neural network (RNN): audio and language data (sequenced data)
  - Hybrid neural network: Autonomous driving (more complex input)



- Supervised learning can be applied to structured and unstructured data
  - Structured data has features with well defined meanings
  - Unstructured data has more abstract features (images, audio, text)

- Deep learning has only recently started to become more widespread
  - Given large amounts of data and a large NN, deep learning will outperform more traditional learning algorithms
  - For small amounts of data, any performance of the algorithm depends on specific implementation
- "Scale drives deep learning progress"
  - Both the scale of the data and the NN
- Recent algorithmic innovations with increase scale of computation



- Idea to switch from sigmoid activation function to ReLu function increased NN performance
- Ends of sigmoid function have close to 0 gradient so and therefore result in small changes in θ
- ReLu function has gradient of 1 for positive values

- Neural network process is iterative
  - Increasing speed at which a NN can be trained allows different ideas to be tried

1.2 Neural Network Basics

1.2.1 Logistic Regression as a Neural Network

- Logistic regression used for binary classification
- For a colour image, of 64×64 pixels, will have total 12288 input features
  - Image is stored as 3 separate matrices for each colour channel
  - All pixel intensities should be unrolled into a single feature vector

$$n = 12288$$

$$x \in \mathbb{R}^{12288}$$

  - For a matrix `X` of shape (a, b, c, d), want a matrix `X_flatten` of shape $(b * c * d, 1)$

```
X_flatten = X.reshape(X.shape[0], -1).T
```

Notation

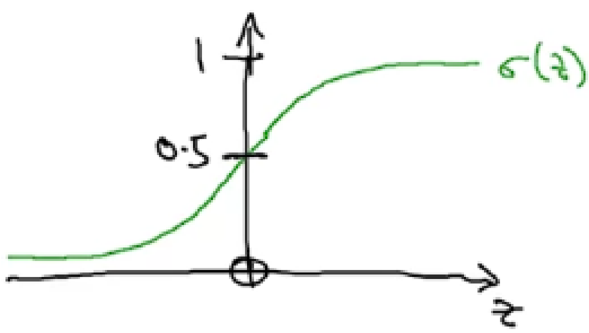
$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$$

- (x, y): single training example
  - $x \in \mathbb{R}^{n_x}$ (n_x = number of features)
  - $y \in \{0, 1\}$

- $(x^{(i)}, y^{(i)})$: i^{th} training example
- $m = m_{train}$
- $m_{test} = \#$ of test examples
- $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(3)} \\ | & | & & | \end{bmatrix}$
  - $X \in \mathbb{R}^{n_x \times n}$
- $Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}$
  - $Y \in \mathbb{R}^{1 \times m}$

Logistic Regression

- Given x, want $\hat{y} = P(y = 1|x)$
  - Since $\hat{y}$ is a probability, want $0 \leq \hat{y} \leq 1$
- Parameters: $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$
- Output: $\hat{y} = \sigma(w^T x + b)$



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$
$$z = w^T x + b$$

- Aim is to learn parameters w and b such that $\hat{y}$ is a good estimate of the probability
- Previous convention had θ vector with an additional θ_0 parameter
  - Keeping θ_0 (b) separate from the rest of the parameters is easier to implement

Cost Function

- Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$
- Squared error function not used for logistic regression loss function

- Optimization problem becomes non convex and will have local optima

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

- If $y = 1$:
  - $\mathcal{L}(\hat{y}, y) = -\log(\hat{y})$
  - Want large $\log(\hat{y}) \therefore$ want large $\hat{y}$
  - $\hat{y}$ has a max of 1 $\therefore$ want $\hat{y} = 1$
- If $y = 0$:
  - $\mathcal{L}(\hat{y}, y) = -log(1-\hat{y})$
  - Want large $\log(1-\hat{y}) \therefore$ want small $\hat{y}$
  - $\hat{y}$ has a min of 0 $\therefore$ want $\hat{y} = 0$
- Cost function:

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

  - Average loss function over all training examples

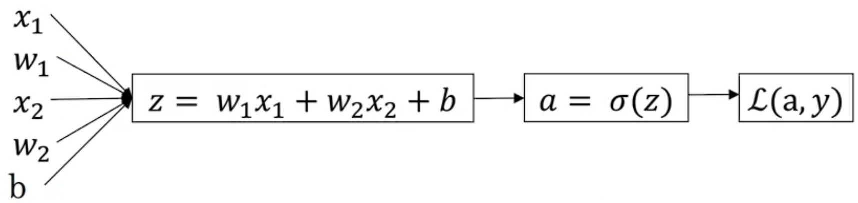
Gradient Descent

- Want to find values of w and b that minimize the cost function $J(w,b)$
  - For logistic regression, w and b usually initialized to 0
- One iteration of gradient descent will take a step in the direction of steepest descent

```
Repeat {
    w := w - α ∂J(w,b)/∂w
    b := b - α ∂J(w,b)/∂b
}
```

- Using the computation graph:

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



$$
\begin{aligned}
\frac{\partial \mathcal{L}(a,y)}{\partial z} &= \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} \\
&= (-\frac{y}{a} + \frac{1-y}{1-a}) \times a(1-a) \\
&= a - y
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_1} &= x_1 \times \frac{\partial \mathcal{L}}{\partial z} \\
\frac{\partial \mathcal{L}}{\partial w_2} &= x_2 \times \frac{\partial \mathcal{L}}{\partial z} \\
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial z}
\end{aligned}
$$

- Partial derivative over all training examples calculated by taking the average `dw1`

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

Initialize `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 += x_1^(i) dz^(i)
    dw2 += x_2^(i) dz^(i)
    db += dz^(i)

J /= m
dw1 /= m
dw2 /= m
db /= m

w1 := w1 - α dw1
w2 := w2 - α dw2
b := b - α db
```

- Above implementation requires `for` loop over all features for all training examples
  - Vectorization can be used to remove explicit `for` loops
  - Vectorization required for deep learning to be efficient

1.2.2 Vectorisation in Python

- Deep learning performs best on large data sets
  - Code must be able to run quickly to be effective on large data sets

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

$$w \in \mathbb{R}^{n_x} \quad x \in \mathbb{R}^{n_x}$$

- Non vectorized implementation:

```python
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

- GPUs and CPUs both have parallelization instructions (SIMD: Single Instruction Multiple Data)
  - If built in functions are used, `numpy` will use parallelism to perform computations faster
- For logistic regression, need to calculate z and a values for each training example

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

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

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$w \in \mathbb{R}^{n_x} \quad X \in \mathbb{R}^{n_x \times m}$$

$$\begin{aligned} \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} &= w^T X + \begin{bmatrix} b & b & \dots & b \end{bmatrix} \\ &= \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & \dots & w^T x^{(m)} + b \end{bmatrix} \end{aligned}$$

- In Python:

```python
Z = np.dot(w.T, X) + b
```

  - Python will broadcast the value `b` so it can be added to the matrix
- Vectorized implementation of sigmoid function can be used on `Z` to calculate `A`

$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix}$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dz = A - Y$$
$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$
$$dw = \frac{1}{m} X (dz)^T$$

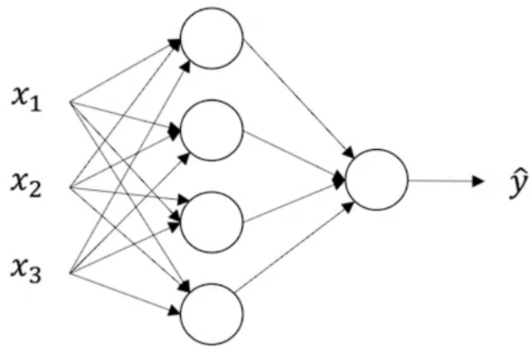
```python
Z = np.dot(w.T,X) + b
A = sigmoid(Z)
dz = A - Y
dw = 1/m * np.dot(X, dz.T)
db = 1/m * np.sum(dz)

# Gradient descent update
w = w - alpha * dw
b = b - alpha * db
```

- `for` loop is required to run multiple iterations of gradient descent

1.3 Shallow Neural Networks

- A neural network will have stacked logistic regression units in each layer
  - Logistic regression output from one layer will be fed to another layer



- Input layer of the neural network contains the feature x_1, x_2, x_3
  - $a^{[0]} = X$
- Intermediate layers in the network are hidden layers
  - Hidden layers do not have "true" values in the training set
- Final layer in the network is the output layer
  - Generates the predicted value $\hat{y}$
- Above diagram is a 2 layer NN
  - Input layer is layer 0
- Each layer will have parameters w and b associated with them
- Each node in the NN will perform logistic regression with its inputs

$$z_i^{[l]} = w_i^{[l]T} x + b_i^{[l]} \quad \rightarrow \quad a_i^{[l]} = \sigma(z_i^{[l]})$$

$$W^{[1]} = \begin{bmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \\ - & w_4^{[1]T} & - \end{bmatrix}$$

$$a^{[0]} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

$$
\begin{aligned}
z^{[1]} &= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \\
&= \begin{bmatrix} w_1^{[1]T} a^{[0]} + b_1^{[1]} \\ w_2^{[1]T} a^{[0]} + b_1^{[1]} \\ w_3^{[1]T} a^{[0]} + b_1^{[1]} \\ w_4^{[1]T} a^{[0]} + b_1^{[1]} \end{bmatrix} \\
&= w^{[1]} a^{[0]} + b^{[1]}
\end{aligned}
$$

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

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

- Vectorized method should be able to work on all training examples at one time
  - Vector for each training example can be stacked horizontally in a matrix
  - Vertical dimension will be the number of units in a layer (n_x for the input layer)

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

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

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

$$
\begin{aligned}
Z^{[1]} &= W^{[1]} X + b^{[1]} \\
A^{[1]} &= \sigma(Z^{[1]}) \\
Z^{[2]} &= W^{[2]} A^{[1]} + b^{[2]} \\
A^{[2]} &= \sigma(Z^{[2]})
\end{aligned}
$$

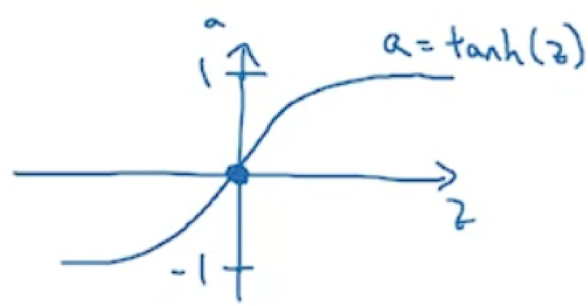
1.3.1 Activation Functions

- After z values are calculated, activation function must be run to get the activation value a

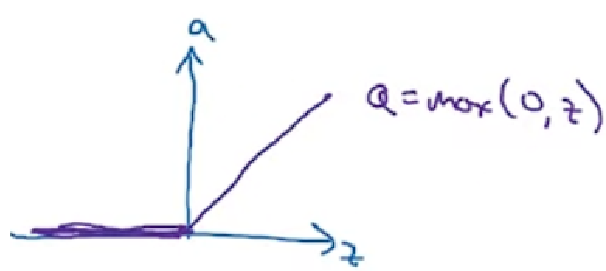
$$a_{sigmoid} = \frac{1}{1+e^{-z}}$$

- Alternatively $a^{[1]} = g(z^{[1]})$ where g is a non linear function
- tanh function almost always performs better than the sigmoid function
  - Equivalent to a transformed version of the sigmoid function
  - tanh function is odd and is "centered" around the origin
  - The mean of the data will be closer to 0 and will help with learning in the next layer

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



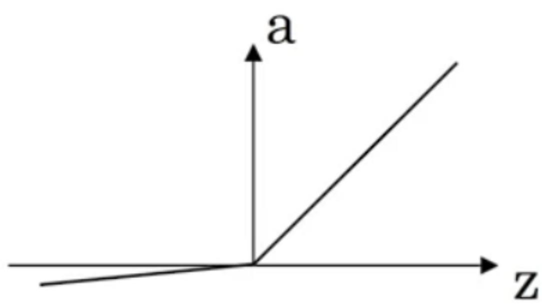
- For binary classification, the final output layer can use the sigmoid function
  - Want the value of $\hat{y}$ to be between 0 and 1
- For both the sigmoid and tanh functions, when z is large, the gradient is very small
  - Results in a slower gradient descent
- ReLU function has a gradient of 1 when z is positive



– Gradient is 0 when z is negative

- For majority of the ReLU function, gradient is very different from 0
  – Will typically allow NN to learn much faster than sigmoid or tanh function
- ReLU function should be used as the default activation function
- The leaky ReLu function has a slight positive gradient when z is negative

$$a_{leakyReLU} = \max(0.01z, z)$$



- For a NN to compute more complex functions, activation function must be non linear
  – If a linear activation function is used, final output of the NN can only be a linear function
  – Multiple linear activation neurons with a sigmoid as the output neuron is equivalent to standard logistic regression
- Linear activation function can be used in the output layer if output is a real number
- Derivative of the activation function must be calculated for backpropagation
  – Sigmoid function

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

$$\begin{aligned}\frac{d}{dz}g(z) &= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)\\ &= g(z)(1-g(z))\end{aligned}$$

  – tanh function

$$\begin{aligned}g(z) &= \tanh(z)\\ &= \frac{e^z-e^{-z}}{e^z+e^{-z}}\end{aligned}$$

$$\frac{d}{dz}g(z) = 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2$$
$$= 1 - g(z)^2$$

- ReLU function

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

- Leaky ReLU function

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

1.3.2 Gradient Descent for Neural Networks

- For a single hidden layer NN, parameters are: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$
  - $w^{[1]} \in \mathbb{R}^{n_1 \times n_0}$
  - $b^{[1]} \in \mathbb{R}^{n_1 \times 1}$
  - $w^{[2]} \in \mathbb{R}^{n_2 \times n_1}$
  - $b^{[2]} \in \mathbb{R}^{n_2 \times 1}$
- Cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$
- For one iteration of gradient descent:

$$w^{[1]} := w^{[1]} - \alpha dw^{[1]},\ \ b^{[1]} := b^{[1]} - \alpha db^{[1]}$$
$$w^{[2]} := w^{[2]} - \alpha dw^{[2]},\ \ b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

  - Gradient descent step will take place after backpropagation calculates the derivatives
- Forward propagation:

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$

- Backpropagation:

$$
\begin{aligned}
dz^{[2]} &= A^{[2]} - Y \\
dw^{[2]} &= \frac{1}{m} dz^{[2]} A^{[1]T} \\
db^{[2]} &= \frac{1}{m} \texttt{np.sum}(dz^{[2]}, \texttt{axis = 1, keepdims = True}) \\
dz^{[1]} &= w^{[2]T} dz^{[2]} \times g^{[1]\prime}(z^{[1]}) \\
dw^{[1]} &= \frac{1}{m} dz^{[1]} X^T \\
db^{[1]} &= \frac{1}{m} \texttt{np.sum}(dz^{[1]}, \texttt{axis = 1, keepdims = True})
\end{aligned}
$$

1.3.3 Random Initialization

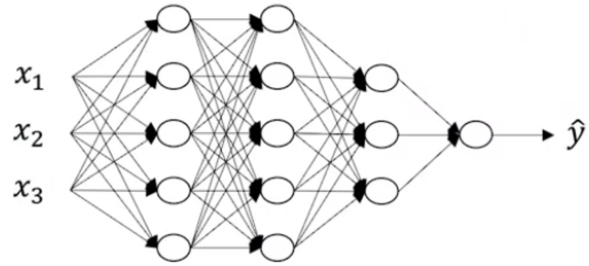
- Weights must be initialized randomly for a NN
  - Weights can be initialized to 0 for logistic regression
  - The bias terms b can be initialized
- If weights are initialized to 0, all neurons in a layer will compute the same hypothesis

```
W1 = np.random.randn((2,2)) * 0.01
b1 = np.zero((2,1))
```

- Weights should be initialized to small random values
  - If weight is too large, activation value $z^{[1]}$ will be large
  - If sigmoid or tanh function is used, derivative will be very small and learning will be very slow
- Different constant for `np.random.randn` should be used for deeper neural networks

1.4 Deep Neural Networks

- Logistic regression is equivalent to a 1-layer NN
- Deep NN have more hidden layers
  - Number of hidden layers in the network can be a parameter for the ML problem



- Above network has 4 layers, $L = 4$
- $n^{[l]}$ = number of units in layer l
- $a^{[l]}$ = activations in layer l

- The inputs x are the activations of the first layer, $x = a^{[0]}$
  - Prediction $\hat{y}$ will be the activations of the last layer, $\hat{y} = a^{[L]}$
- Forward propagation for a deep NN will follow the same pattern for all layers

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

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

- For a vectorized implementation

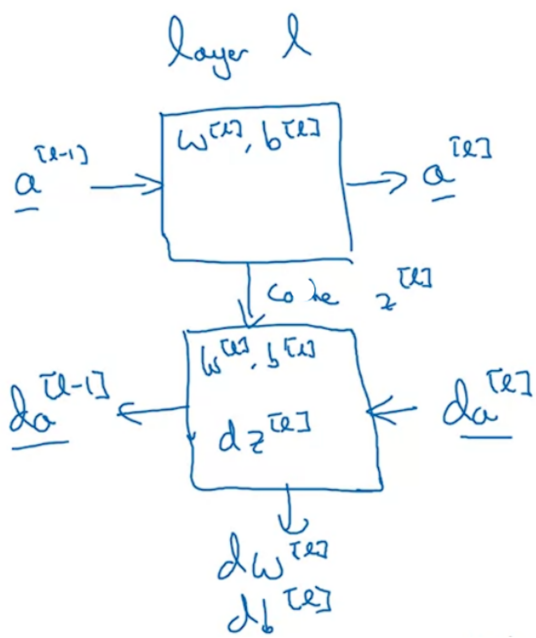
$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

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

  - Explicit for loop will be used to loop over the layers in the network
  - b will still be a column vector but will apply correctly due to broadcasting
  - When working with W and A matrices, A will be for the previous layer so the dimensions will fit
- When debugging NN, can look at dimensions of all the matrices
- For a non vectorized implementation:
  - $W^{[l]} : (n^{[l]}, n^{[l-1]})$
  - $b^{[l]} : (n^{[l]}, 1)$
  - Dimensions of dw and db should be the same as the dimensions of W and b
  - $a^{[l]}, z^{[l]} : (n^{[l]}, 1)$
- For a vectorized implementation, z vectors and a vectors will be stacked horizontally for all training examples

- $Z^{[l]}, A^{[l]} : (n^{[l]}, m)$

- Deep NN tend to work better as each layer can compute increasingly complex functions
  - Face recognition: edge detection $\rightarrow$ individual features $\rightarrow$ large parts of the face
  - Audio: low level waveforms $\rightarrow$ phonemes $\rightarrow$ words $\rightarrow$ sentences
- Functions that can be computed with a "small" deep neural network require exponentially more hidden units in a shallower network
- For each forward propagation step, the value of $z^{[l]}$ should be cached for backpropagation
  - Values of $w^{[l]}$ and $b^{[l]}$ can also be stored in the cache so they can be accessed for backpropagation



- All forward propagation steps will carried out until the hypothesis, $\hat{y}$ is found
  - Using cached values, all backpropagation steps will be carried out until $dz^{[1]}$
  - Parameters $W^{[l]}$ and $b^{[l]}$ can be updated accordingly

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

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

- Backpropagation will also follow a pattern for all layers in the NN
  - $dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$
  - $dW^{[l]} = dz^{[l]} a^{[l-1]T}$

  - $db^{[l]} = dz^{[l]}$
  - $da^{[l-1]} = W^{[l]T} dz^{[l]}$
- For a vectorized implementation:
  - $dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$
  - $dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$
  - $db^{[l]} = \frac{1}{m}$`np.sum`$(dZ^{[l]}$`, axis=1, keepdims=True`$)$
  - $dA^{[l-1]} = W^{[l]T} dZ^{[l]}$

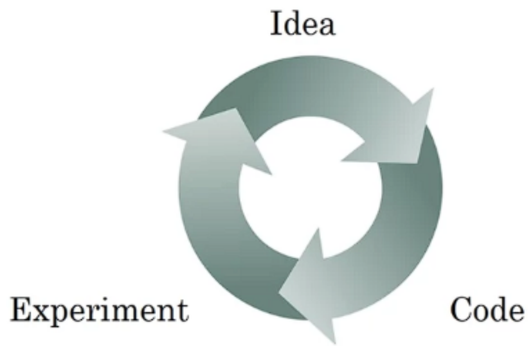
1.4.1 Parameters vs Hyperparameters

- Parameters of the NN are the W and b matrices
- NN also has a number of associated hyperparameters:
  - Learning rate α
  - Number of iterations z‘’
  - Number of layers in the network
  - Number of hidden units
  - Choice of activation function
- Hyperparameters will control the values of W and b
- Deep learning has many more hyperparameters than earlier eras of machine learning
  - Applying deep learning becomes an empirical process
- Intuitions about hyperparameters may be different across different applications

2 Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

2.1 Practical Aspects of Deep Learning

- Applying ML is a highly iterative process
  - Very hard to choose "correct" values for hyperparameters



- Deep learning used in many different areas
  - NLP
  - Computer vision
  - Speech analysis
  - Structured data
    * Advertisement
    * Search engines
    * Computer security
    * Logistics
- Intuitions from one subject area often don't transfer to another application
- Success of deep learning can depend on speed of iteration
  - Choice of split of the data can influence speed of iteration
- Whole dataset should be split into training, development and test set
  - Dev set should be used rate performance of different models
  - Final model should be evaluated on the test set
  - Split will allow better evaluation of bias and variance of the model
- Previous eras of ML had a 60/20/20 split between dataset

- For the big data era, a smaller percentage of data is given to the dev and test sets
  - For 1,000,000 examples, can allocate just 10,000 examples each to dev and test set
  - 10,000 examples is enough to run the algorithm and get a good idea about the algorithm performance
- Recent trends also show mismatched training and test set distributions
  - For images, training set may have very high quality images while test set may have lower quality
  - Dev and test set should come from the same distribution
- Dataset might be split to not include a test set
  - Dev set can be used to get to a "good" model
  - Since data is fit to the dev set, there is no unbiased estimate of performance
  - When data doesn't include a test set, dev set is usually referred to as "test" set
  - Resulting model may overfit to the dev set

2.1.1 Bias and Variance

- In the deep learning era, there tends to be less of a discussion about the bias/variance trade off
- In 2 dimensions, data can be plotted to look for high bias or variance
  - High bias classifiers underfit the data
  - High variance classifiers overfit the data
- For higher dimensions, training set error and dev set error can be used
  - High variance classifier has low training error and high dev set error
  - High bias classifier has high training error and high dev set error
  - Classifier with high bias and high variance will have high training error and even higher dev set error
- Above ideas only work with the assumption that the optimal error is 0%
  - Training and dev set must also come from the same distribution

2.1.2 Basic Recipe for Machine Learning

- Train initial algorithm and reduce bias of algorithm to an "acceptable value"
  - Use a larger network
  - Train algorithm for longer

- Reduce variance of the algorithm by getting more data
  - Add regularization terms to the cost function
- Bias and variance can also be reduced by using a more appropriate NN architecture
- In the big data era, bias and variance can be reduced without affecting each other
  - Training a bigger network typically reduce the bias
  - Getting more training data will typically reduce the variance
- Using regularization will have a bias variance trade off

2.1.3 Regularization

- Adding regularization will usually help in reducing variance and prevent overfitting
  - Regularization will only affect how the weights change during backpropagation
  - For forward propagation, regularization has no effect
- For logistic regression:

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

$$\begin{aligned} ||w||_2^2 &= \sum_{j=1}^{n_x} w_j^2 \\ &= w^T w \end{aligned}$$

  - Above method is L_2 regularization after the L_2 norm (Euclidean norm) of w
  - b values can also be regularized but will have a much smaller effect than w
- L_1 regularization adds the term:

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

  - Using L_1 regularization will result in w being sparse
  - Can be seen to compressing the model
- L_2 regularization is more common for deep learning
- Regularization parameter λ will be set using the cross validation set
  - `lambda` is a reserved keyword in Python

- For a neural network:

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

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

  - $||W^{[l]}||_F^2$ known as the Frobenius norm of the matrix

- Since new term added to cost function, $\frac{\partial J}{\partial W^{[l]}}$ will be different

$$dW^{[l]} = ... + \frac{\lambda}{m}W^{[l]}$$

$$W^{[l]} = W^{[l]} - \frac{\alpha\lambda}{m}W^{[l]} - \alpha(...)$$

  - Also known as weight decay as value of W will decrease on every iteration

$$W^{[l]} - \frac{\alpha\lambda}{m}W^{[l]} = \left(1 - \frac{\alpha\lambda}{m}\right)W^{[l]}$$

  - Value of $\left(1 - \frac{\alpha\lambda}{m}\right)$ will be slightly less than 1

- Adding regularization term will penalize the weight matrix from being too large
  - As the value of λ is increased, the weights in w will get closer to 0
  - Each hidden unit will have a smaller effect and the resulting NN will be simpler
- When using the tanh function, penalizing w will make $z^{[l]}$ smaller
  - For a small $z^{[l]}$, tanh function is roughly linear
  - If all hidden units in the network are roughly linear, the result of the NN will also be roughly linear

Dropout Regularization

- Each layer in the NN has a probability of eliminating a node
  - When a node is eliminated, all outgoing links from the node are also deleted
  - Each example will be trained on a smaller network so will have less chance of overfitting
- For each different training example, the NN is reset and randomly eliminates nodes again
- Inverted dropout:

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
a3 /= keep_prob
```

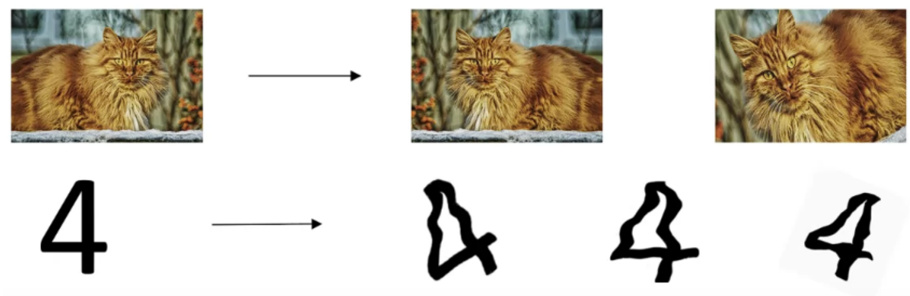
  - For keep_prob $= 0.8$ each node has a 0.2 chance of being removed
  - Activation values should be scaled by keep_prob so the expected value of z can stay constant
  - On each pass through the training set, a different set of units should be zeroed out
- At test time, dropout should not be used as it will create noise in the predictions
- A single hidden unit cannot rely on a specific feature as it may not be used on each iteration
  - Weights for the unit will be spread out between the units
  - Has the same effect as shrinking the weights like L2 regularization
  - The equivalent L2 penalty on different weights depends on the size of the activations being used for the weight
- keep_prob can be varied between the layers
  - Larger layers may be more prone to overfitting and can have a larger keep_prob
  - For small layers with a very small chance of overfitting, keep_prob can be set to 1
- Many dropout implementations started with computer vision
  - Input size for computer vision is extremely large
- Cost function is not well defined when dropout is used
  - Can set keep_prob to 1 and check for monotonically decreasing J
  - When J is decreasing, then can reduce the value of keep_prob to use dropout

Other Regularization Methods

- Getting more training data will almost always help overfitting
  - May not be possible to get more training data or very expensive
- Data augmentation will create new examples and can help reduce overfitting
- For an image dataset:
  - Flipping the image horizontally

– Randomly cropping and distorting the image

- Magnitude of image transformation depends on classifier
  - For a cat dataset, image should not be flipped vertically
  - For OCR, distortions and rotations can be slightly more extreme



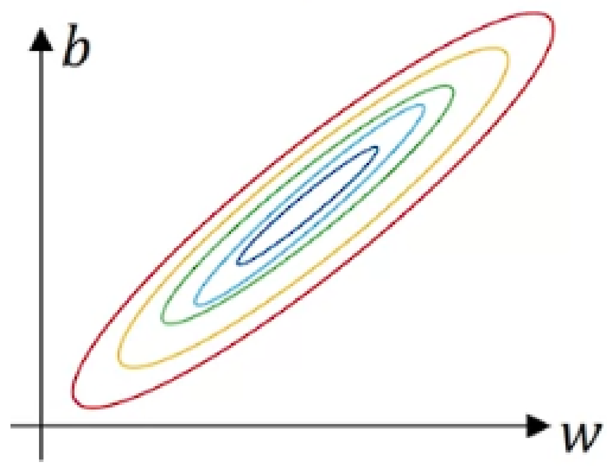
- Early stopping can be used to prevent overfitting from happening
  - If the NN is overfitting the data, the dev set error will initially decrease before increasing
  - Training of the NN can be stopped when the dev set error is lowest and the data has not been overfit
- Using early stopping links the task of optimizing J and not overfitting the data
  - Early stopping will prevent the cost function from being optimized
- L2 regularization is a better method to prevent overfitting
  - Requires a choice for the value of λ and is much more computationally expensive

2.1.4 Setting up the Optimization Problem

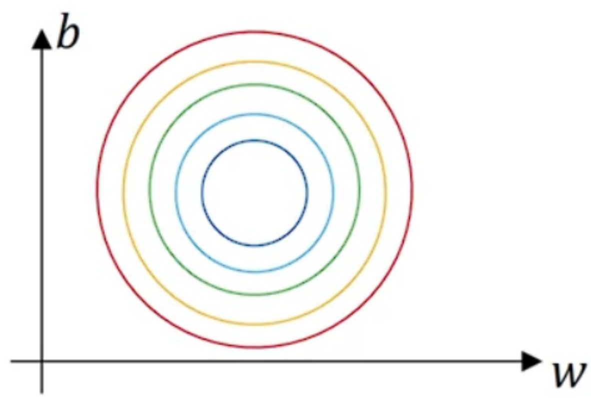
- Normalization can be used to speed up the training of a NN
  - Subtract the mean:
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
$$x := x = \mu$$
  - Normalize the variance:
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2$$
$$x/ = \sigma$$
- When normalizing a training set, test set and training set should be processed together

  - All the data must go through the same transformation

- For data that is not normalized, the cost function will be very elongated
  - The gradient will be quite shallow and will take longer to converge
  - Algorithm will require a smaller learning rate



- On average, normalized data will have a cost function that is more symmetric
  - Gradient descent will converge faster and can use a larger learning rate



Vanishing/Exploding Gradients

- For very deep neural networks, the derivatives can get exponentially big or small
- If the weights of a NN are all the same, the prediction $\hat{y}$ will x to the Lth power
  - For $W^{[l]} > I$ the gradient will explode
  - For $W^{[l]} < I$ the gradient will vanish

- Some modern applications use 152 layer NN
  - Require careful initialization of the weights to ensure correct training
- For a single neuron:
  - The output $\hat{y}$ will be the sum of all $w_i x_i$

$$z = w_1 x_1 + w_2 x_2 + ... + w_n x_n$$

  - For a large n, want a smaller w_i
  - Want $\text{Var}(w_i) = \frac{1}{n}$

```
W^[l] = np.random.randn(shape) * np.sqrt(1/n^[l-1])
```

  - Variance of Gaussian random variable can be set by multiplying by sqrt tem
- For ReLU activation function, the variance should be set to $\frac{2}{n}$
  - tanh activation uses Xavier initialization $\frac{1}{n^{[l-1]}}$
  - Yoshua Bengio multiplied random variable by $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$
- Initialization of weights aims to set weight matrices close to 1
  - Helps to prevent $\hat{y}$ from vanishing or exploding too quickly
- Variance parameter can be tuned as another hyperparameter

Gradient Checking

- Can be used to ensure implementation of backpropagation is correct
- Requires numerical approximations of gradients
  - For a function f at a point θ, gradient can be approximated by looking at $\theta + \epsilon$ and $\theta - \epsilon$
  - Approximation is closer when double sided estimate is used
- If g is the derivative of f:

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

- Using the 2 sided difference will give a much better estimate but is more computationally expensive

- The derivative of a function at a point is the limit of the numerical approximation

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

  - For a non 0 value of ϵ, the error of the approximation is $O(\epsilon^2)$
  - For the single sided numerical approximation, the error is $O(\epsilon)$
- To perform gradient checking on a NN:
  1. Reshape $W^{[1]}, b^{[1]}, ..., W^{[L]}, b^{[L]}$ into a single vector θ
  2. Reshape $dW^{[1]}, db^{[1]}, ..., W^{[L]}, b^{[L]}$ into a single vector $d\theta$
  3. For every i in θ, calculate:

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

  4. Check if $d\theta_{approx}$ and $d\theta$ are reasonably close to each other

     For $\epsilon = 10^{-7}$:

$$\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2} \approx 10^{-7}$$

- Grad check should be only be used when debugging
  - Calculating $d\theta_{approx}$ is very computationally expensive
- If regularization is used, correct cost function must be used to calculate the gradient
- If dropout is used, J is not well defined and cannot use grad check
  - Cost function J that is optimized by dropout is defined by summing over all subsets of nodes that could be eliminated on each iteration
  - Can implement grad check with a `keep_prob` of 1 before turning on dropout
- Implementation of gradient descent may be correct when W and b are close to 0
  - Can run grad check just after random initialization
  - After training the network for a number of iterations, can run grad check again

2.2 Optimization Algorithms

2.2.1 Mini Batch Gradient Descent

- For gradient descent, vectorization will allow computation over all m training examples
  - If m is very large, then vectorization will still be very slow
- Gradient descent requires the whole training set to be processed for a single step of gradient descent

- Data from training set can be split into mini batches

$$X^{\{1\}} = [x^{(1)}, x^{(2)}, ..., x^{(1000)}]$$

$$Y^{\{1\}} = [y^{(1)}, y^{(2)}, ..., y^{(1000)}]$$

- Mini batch gradient descent looks at one mini batch on each iteration of gradient descent
- For each mini batch in the training set:
  - Run forward propagation on $X^{\{t\}}$

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

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

    ...

    $A^{[l]} = g^{[l]}(Z^{[l]})$
  - Compute cost: $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2\times1000}\sum_{l}||W^{[l]}||_F^2$
  - Use backpropagation to calculate gradients wrt $J^{\{t\}}$
  - Update weights

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

    $b^{[l]} := b^{[l]} - \alpha db^{[l]}$
- A single pass through the training set is known as an epoch
- Algorithm can continue to run for multiple passes through the training set until an optimal solution is found
- For batch gradient descent, the cost should decrease on each iteration
  - If the cost doesn't decrease per iteration, then the algorithm has a bug
- For mini batch gradient descent, the cost will trend downwards but will be more noisy
  - Algorithm is being trained on a different batch of results on each iteration
- When running mini batch gradient descent, must choose the size of the mini batch
  - For mini batch size = m: Batch gradient descent
  - For mini batch size = 1: Stochastic gradient descent
- For stochastic gradient descent, each example may be good or bad for gradient descent
  - On average the cost function will be minimized for gradient descent
  - Path taken by gradient descent will be very noisy

- Stochastic gradient descent will never converge and just oscillate around the minimum
- Choice of mini batch size should be between 1 and m
  - Batch gradient descent will take very long for a single iteration
  - Stochastic gradient descent will lose all the speed from vectorization
- For a small training set ($m \leq 2000$), can just use gradient descent
- Otherwise can try a mini batch size from 64-512
  - Code may run faster if the mini batch size is a power of 2
- A single mini batch should be able to fit in the whole CPU/GPU memory

Advanced Optimization Algorithms

- Some advanced algorithms require the use of exponentially weighted averages
- Moving averages can be calculated for data such as daily temperature

$$V_0 = 0$$

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

  - V_t is the approximated average temperature over the last $\frac{1}{1-\beta}$ days
  - If β is larger then the average will adapt slower to changes in the data
- Exponentially weighted average can be found by summing the daily temperature with an exponentially decaying function
- If $\beta = 0.9$:

$$V_{100} = 0.1\theta_{100} + (0.1)(0.9)\theta_{99} + (0.1)(0.9)^2\theta_{98} + (0.1)(0.9)^3\theta_{97} + ...$$

- When calculating the exponentially weighted average, the same variable v should be used and overwritten each time
  - Implementation will be much more efficient than calculating average manually from the past 10 values
- For large values of β, initial average will be much lower than they should be

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

- Bias correction can be used to ensure initial values are correct estimations of the averages
  - As t becomes larger, denominator becomes closer to 1

Momentum

- Gradient descent with momentum uses an exponentially weighted average of the gradients to update the weights
  - Almost always performs better than standard gradient descent

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

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

    $W = W - \alpha V_{dw}$

    $b = b - \alpha V_{db}$

- Taking the average of the gradients will slow down any unnecessary oscillations in the algorithm
  - Algorithm may oscillate at first but will start to take more direct steps to the minimum
- $\beta = 0.9$ is a common choice for most applications of momentum

RMSprop

- RMSprop takes the weighted average of the squares of the derivatives
- Derivatives will get divided by the RMS before the weights are updated

    $S_{dW} = \beta_2 S_{dW} + (1-\beta)dW^2$

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

    $W = W - \alpha \frac{dW}{\sqrt{S_{dw}}}$

    $b = b - \alpha \frac{db}{\sqrt{S_{db}}}$

- Updates in the direction of oscillation will be divided by a large number
  - Will allow the learning rate to be larger and therefore allows faster training
- In practice, very small value ϵ is added to the denominator for more numerical stability

Adam Optimization Algorithm

- Adam optimization shown to work well for a range of deep learning architectures
  - Merges Momentum and RMSprop to one algorithm
  - "Adam" stands for adaptive moment estimation
- On iteration t:

    Compute dW, db using the current mini batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)dW, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1)db$

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

$V_{dw}^C = \frac{V_{dw}}{1-\beta_1^t}, \quad V_{db}^C = \frac{V_{db}}{1-\beta_1^t}$

$S_{dw}^C = \frac{S_{dw}}{1-\beta_2^t}, \quad S_{db}^C = \frac{S_{db}}{1-\beta_2^t}$

$W := W - \alpha \frac{V_{dw}^C}{\sqrt{S_{dw}^C} + \epsilon}$

$b := b - \alpha \frac{V_{db}^C}{\sqrt{S_{db}^C} + \epsilon}$

- Must choose many hyperparameters to run Adam optimization
  - α: needs to be tuned to the specific NN
  - β_1: 0.9 (default)
  - β_2: 0.999 (default)
  - $\epsilon : 10^{-8}$ (default)

Learning Rate Decay

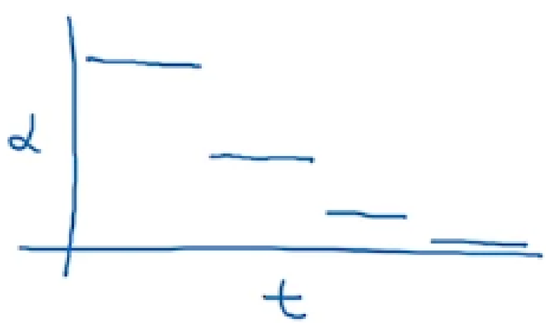
- For mini batch gradient descent, the algorithm will oscillate around the minimum point
- If the learning rate is reduced over time, then the oscillations will become smaller
  - During the initial steps of learning, algorithm can afford to take large steps
  - As the algorithm starts to converge, smaller steps are preferred

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

- Other formulas can be used to decay the learning rate
  - Exponential decay

$$\alpha = 0.95^{\text{epoch num}} \alpha_0$$

  - Discrete staircase



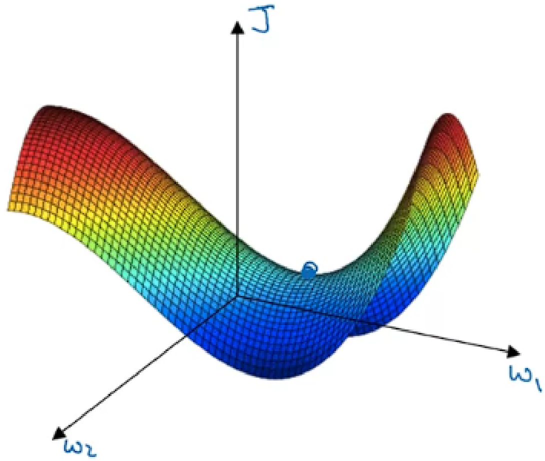
- Square root of epoch number

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

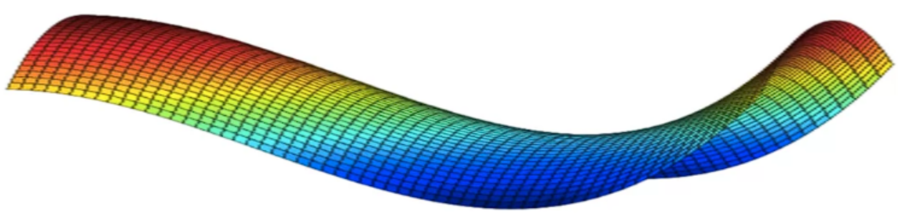
- Manual decay can be used for larger models that take a longer time to train

Local Optima

- Initial ideas believed that a cost function with many points of 0 gradient would have many local optima
  - When training a NN, most points with 0 gradient are saddle points



- For a point with 0 gradient, Each direction can either be a convex or concave function
  - For a local optima, must have a convex function in all directions
  - In a high dimensional space, chance of all directions being convex functions is very small
- Intuitions about lower dimensional spaces may not transfer to high dimensional spaces
- Plateaus are areas where the gradient is near to 0 for a large area



  - Will take a very long time to move down off the plateau

  - Learning will be slow but unlikely to get stuck in a local optima

- Optimization algorithms like Adam can help to speed up the training

2.3 Hyperparameter Tuning, Batch Normalization and Programming Frameworks

2.3.1 Hyperparameter Tuning

- Deep neural networks have many hyperparameters associated with the actual network and the training implementation
  - Numbers of layers and hidden units
  - Learning rate or method for learning rate decay
  - Hyperparameters for momentum or Adam optimization
  - Mini batch size
- Most important hyperparameter is the learning rate
  - Secondary importance can be given to momentum (β), number of hidden units and the mini batch size
  - Number of layers and learning rate decay can be tuned last
  - Parameters for Adam optimization usually don't need to be tuned
- In practice, random values for the hyperparameters should be sampled and tested
  - If values are arranged in a grid, fewer distinct values can be tested
  - Choosing random values for the hyperparameters gives a higher chance of finding an optimum value for important hyperparameters
- Can use coarse to fine sampling scheme to find optimum values
  - Sample initial values and find which values work the best
  - "Zoom in" to the area and take more samples in the smaller region
- For some hyperparameters (number of layers / hidden units), can sample over a reasonable range
- Some hyperparameters may not have an even distribution (Learning rate between 0.0001 and 1)
  - Can use a log scale to ensure the numbers are better distributed

```
r = -4 * np.random.rand
learning_rate = 10 ** r
```

- Can look for a range $10^a ... 10^b$ and take a random sample $r \in [a, b]$

- For exponentially weighted averages, β will be around 0.9-0.999
  - Equivalent to averaging over the last 10 days or last 1000 days
  - Can sample values for $1 - \beta$ for $r \in [-3, -1]$
- For exponentially weighted averages, the sensitivity of the results is very high when β is close to 1
  - A change from 0.999 to 0.9995 will change the average from 1000 to 2000 examples
- Intuitions about the hyperparameters won't always transfer across applications
  - Ideas found in one application can still be applied to other applications
- Hyperparameters can become stale over time with changing data or hardware
  - Hyperparameters should be reevaluated every few months to ensure values are optimal
- Depending on resources, can babysit a single model or train models in parallel
  - For a single model, hyperparameters can be tweaked over time depending on training performance
  - If resources allow, can train the same model with many different hyperparameters and choose the best model

2.3.2 Batch Normalization

- Inputs to a NN can be normalized to speed up learning

$$X = \frac{X - \mu}{\sigma}$$

- Batch normalization normalizes the input values $Z^{[l]}$ to each layer
  - Can instead normalize the values $A^{[l]}$ after the activation function
- Given intermediate values $z^{(1)}, ..., z^{(m)}$:

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

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

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

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

- γ and β are learnable parameters of the model
  - Allows the mean and variance of $\tilde{z}$ to be set to any value

- If $\gamma = \sqrt{\sigma^2 + \epsilon}$, $\beta = \mu$, then $\tilde{z}^{(i)} = z^{(i)}$
  - May not want mean 0 and standard deviation 1 for the activation function
- NN will have new parameters $\beta^{[1]}, \gamma^{[1]}, ..., \beta^{[L]}, \gamma^{[L]}$
  - Will be updated like normal parameters

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

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

- Batch normalization is typically applied to mini batch gradient descent
  - Mean and variance will be calculated from the mini batch being used
- When using batch normalization, normalization step removes the need for $b^{[l]}$ parameters
  - When subtracting the mean from the z values, the constant will get cancelled out
  - Mean of the $\tilde{Z}$ values will be decided by the $\beta^{[l]}$ parameters
- Batch normalization will make weights deeper in a network more robust to changes earlier in the network
  - Data can have a covariate shift where the distribution changes after a generalization
  - Function mapping from X to Y can be the same but model may need to be retrained
  - Batch normalization will reduce the amount of movement of the distribution of the hidden values
- Even if there is a covariate shift in the data, batch norm will make the z values have the same mean and variance
  - The individual layers in the network will be more independent of each other
- Batch norm will also add a slight regularization effect
  - Each mini batch is scaled by the mean/variance of the specific mini batch
  - Normalizing with the mean/variance of the individual mini batch will add noise to the activations
  - Similar to dropout where the algorithm will not rely on any single hidden unit
  - Noise added to the z values is very small so dropout can be used as well
- If a larger mini batch size is used, noise is reduced and will have a smaller regularization effect
- At test time, data will typically be processed one example at a time

- Cannot calculated the mean/variance of a single example
- Mean/variance can be estimated using exponentially weighted averages across the mini batches

2.3.3 Multi Class Classification

- Logistic regression can be generalized to apply to multiple classes

$$C = \text{number of classes}$$

- Output layer for the NN will have C units
  - Each unit will be the probability of each class
  - Sum of all numbers in the vector must be 1
- Softmax layer used in the output layer to output vector of probabilities
  - $Z^{[L]}$ values are calculated as normal: $Z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$
  - Use the softmax activation function

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

    $a^{[L]} = \frac{t}{\sum_{i=1}^{C} t_i}$
- Softmax activation function has a vector for its input and output
  - Other activation functions had a single value for input and output
- Largest input to softmax function will result in the largest output
  - "Hard max" function would return 1 for the largest input and 0 for the other inputs
- If $C = 2$, softmax reduces to logistic regression
- Softmax classifier cannot be trained as a normal NN

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

- Loss function will only be active for the ground truth class in the training set

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

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

2.3.4 Deep Learning Frameworks

- For larger NNs, using a framework can save a lot of time
- Can look at the community behind the frameworks and the strengths
  - Ease of programming (development and deployment)
  - Running speed
  - Truly open (open source with good governance)
  - Application of NN

Tensorflow

- Assume a simple cost function:

$$J(w) = w^2 + 10w + 25$$

```python
import numpy as np
import tensorflow as tf

w = tf.Variable(0, dtype=tf.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def train_step():
    with tf.GradientTape() as tape():
        cost = w ** 2 - 10 * w + 25
    trainable_variables = [w]
    grads = tape.gradient(cost, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))

for i in range(1000):
    train_step()
```

- No need to compute backpropagation steps with tensorflow
- More complex tensorflow program will have cost as a function of variables

```python
w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optmizers.Adam(0.1)
```

```python
def training(x, w, optimizer):
    def cost_fn():
        return x[0] * w ** 2 + x[1] * w + x[2]

    for i in range(1000):
        optimizer.minimize(cost_fn, [w])

    return w
```

- Tensorflow will create a computation graph from the defined cost function
  - From the computation graph, tensorflow will compute the backpropagation steps

3 Structuring Machine Learning Projects

3.1 ML Strategy

3.1.1 Setting up a ML Project

- A machine learning project may have many ideas that can improve performance
  - Collect more data
  - Use a more diverse training set
  - Train the algorithm over a longer period of time
  - Use a different optimization algorithm (Adam instead of gradient descent)
  - Use a bigger/smaller network
  - Add dropout or L_2 regularization
  - Change the network architecture (activation functions or hidden units)
- Some methods may not be useful for the specific scenario
- ML strategy is changing with deep learning
  - Deep learning algorithms have different options when compared with previous generations
- Orthogonalization is where specific functions can be split up into different areas
- For a supervised learning system to perform well, system requires a chain of assumptions
  - Performance of algorithm on the training set must pass some threshold ($\approx$ human-level performance)
  - Algorithm must be fit well to the dev set
  - Algorithm must be fit well to the test set
  - Algorithm must perform well in the real world
- Each step has specific "knobs" to tune to improve performance in the specific area
  - Training set: bigger network, Adam optimization
  - Dev set: regularization, bigger training set
  - Test set: bigger dev set
  - Real world: change dev set or cost function
- Early stopping can be used but is less orthogonalized
  - Worsens the performance on the training set

  - Improves the performance on the dev set

- Single number evaluation metric can be used to test effectiveness of a model
- F1 score combines precision and recall into a single metric
  - Precision is the percentage of positively classified examples that are actually positive
  - Recall is the percentage of positive examples that are correctly classified
  - F1 score takes the harmonic mean of precision and recall

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

- Having a well defined test set and single number evaluation metric will speed up iteration
- Scenario may have more than one type of metric that is relevant
  - Classification algorithm may value accuracy as well as running time
  - May not make sense to use a numerical function of some metrics
- Accuracy would be a optimizing metric and running time would be the satisficing metric
  - Goal can be to maximize accuracy subject to running time $\leq$ 100ms
- For N different metrics:
  - 1 should be optimizing
  - $N - 1$ should be satisficing
- Dev set and test set should come from the same distribution
  - If different distributions are used, algorithm may perform on the dev set but not on the test set
  - Dev set and test set must have the same target
- Dev set should be used to evaluate the performance of different models
  - Setting up a dev set and an evaluation metric allows teams to iterate quickly

"Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on"

- Previous eras of machine learning had a 60%, 20%, 20%
- Modern eras of machine learning have much larger datasets
  - For 1000000 examples, can assign 1% each to dev and test set

  - Larger amount of data in the training set will help algorithm

> "Set your test set to be big enough to give high confidence in the overall performance of your system"

- Some applications may only use a train and dev set
  - Specific scenario may not require high confidence in the overall performance of the algorithm
  - Must be careful to not overfit the dev set too much
- Evaluation metric may not give a full representation of the specific scenario
  - Cat classifier with very low error may allow some pornographic images through the algorithm
  - Algorithm with slightly higher error but no pornographic images would be preferred
- Evaluation metric should be changed if it doesn't correctly rank the algorithm's performance
  - Standard error function treats all images equally
  - Weight can be added to the error function to weight unwanted images higher
  - Requires labelling of unwanted images in dev and test set
- Task of changing evaluation metric is separate from changing cost function to achieve good performance
- Metric and/or dev/test set should be changed if performance on the application is not linked

3.1.2 Comparing to Human Level Performance

- With advances in deep learning, ML algorithms have much better performance
  - More feasible for algorithms to be competitive with human level performers
- Workflow of designing and building a ML system is more efficient when trying to learn something that humans can do
- For many ML projects, initial learning will be very fast as algorithm approaches human level performance
  - Rate of learning decreases after algorithm surpasses human level performance
  - Algorithm will approach Bayes optimal error
- Bayes optimal error is the best theoretical function for mapping from X to Y
  - For many tasks, human level performance is not very far from Bayes optimal error

  - Once human level performance is surpassed, there may not be many areas to improve in
- If algorithm has lower than human level performance:
  - Get labelled data from humans
  - Gain insight from manual error analysis
  - Better analysis of bias/variance
- If human level performance is much lower than the training and dev set error, can focus on the bias of the algorithm
- If human level performance is close to the training error, can focus on the variance of the algorithm
- Human level performance can be used as an estimate for Bayes error
  - Difference between the Bayes error and training error is the avoidable bias
  - Difference between the training and dev set error can measure the variance
- For specialized tasks, different parties may have different errors for human classification
  - For medical image classification, a team of experienced doctors will have much lower error than an average human
  - Bayes error must be less than or equal to the lowest human error
  - Lowest human error can be used as estimate for Bayes error
- For publishing a paper or deploying a system, human error definition may be different
- When algorithm is very close to human level performance, can be hard to see if bias or variance should be trained
- With deep learning, algorithms in some areas can surpass human level performance
  - Online advertising
  - Product recommendations
  - Logistics
  - Loan approvals
- Above areas are not natural perception problems and come from structured data
  - Currently more challenging for computers to surpass humans in natural perception tasks
- ML has also surpassed humans in some natural perception tasks
  - Speech recognition
  - Some image recognition

    - Medical tasks
- For supervised learning, must assume that the training set can be fit well (low avoidable bias)
    - The training set performance must also generalize well to the dev/test set (low variance)
- For high bias:
    - Train a bigger model
    - Train for longer or use a better optimization algorithm (momentum, RMSprop, Adam)
    - Change the NN architecture or find better hyperparameters
- For high variance:
    - Use more data
    - Use regularization (L_2, dropout, data augmentation)
    - Change the NN architecture or find better hyperparameters

3.1.3 Error Analysis

- Misclassified examples can be manually examined to look for any patterns
    - Finding patterns can give an upper bound of any increase in performance
- Different ideas for error analysis can be evaluated in parallel with a table
    - For each image, can fill in a checkbox for any patterns
    - Percentage of total for each pattern will give an idea of how to best improve performance
- Manual analysis may show new patterns in the errors
- Some errors may be incorrectly labelled examples in the dev/test set
    - Deep learning algorithms are quite robust to random errors in the training set
    - Algorithms are fairly susceptible to systematic errors in the training set
- Incorrectly labelled examples can be recorded in the error analysis table
    - Percentage of error caused by incorrect labels can be calculated to see if fixing labels is a worthwhile task
- Any processes should be applied to the dev and test set at the same time to ensure they come from the same distribution
    - Training set may end up coming from a different distribution than the dev/test set

- Can also look at examples that the algorithm got right to see if got any errors
- For a new ML system, priority should be to build initial system then iterate
  - Set up a dev/test set and evaluation metric
  - Build initial system quickly
  - Use bias/variance and error analysis to prioritize next steps
- Error analysis will give idea for next steps

3.1.4 Mismatched Training and Dev/Test Sets

- Deep learning algorithms perform best with a lot of training data
  - Many teams are putting as much data as possible into training sets
  - Extra data added to the training set will give a different distribution to the training set data
- Other sources of data may have more examples but can come from a slightly different distribution
- Data can be pooled together and randomly split into training, dev and test set
  - All data will come from the same distribution
  - Much of the dev set will come from the additional distribution of images rather than the original distribution
  - Algorithm will optimize to the wrong distribution of images
- Training set can be set to include all images from the additional distribution
  - Examples from the original distribution will be split between the dev and test set
  - Dev and test set will have the correct distribution of images
  - Training set will have a different distribution
- Estimate of bias and variance changes when training set has a different distribution to dev and test set
  - Comparatively high dev set error might mean dev set has more challenging images than training set
  - Data from the dev set will be new to the algorithm and will have a different distribution to the training data
- Portion of the training set can be set as the training-dev set
  - Should not be used for training but will have the same distribution as the training set
- For error analysis, can look at the training set, training-dev set and dev set

  - Large difference between the training error and training-dev error indicates a variance problem
  - Large difference between the training-dev error and dev error indicates a data mismatch problem
  - Large difference between training error and human error indicates high bias problem
  - Difference between the dev error and test error indicates degree of overfitting to the dev set
- For each distribution of data, can look at:
  - Human level error
  - Error on examples trained on
  - Error on examples not trained on
- For data mismatch:
  - Use manual data analysis to try understand the difference between training and dev sets
  - Can try to make the training set more similar to the dev set (collect more examples or use artificial data synthesis)
- For some applications, algorithm may overfit during artificial data synthesis
  - For speech recognition, same recording of noise may be added to many examples
  - As much as possible, should aim to get a large range of examples with data synthesis

3.1.5 Learning From Multiple Tasks

- For some applications, NN trained for one task can be applied to another task
  - NN trained for cat recognition can be retrained for radiology diagnosis
- After initial NN is trained, output layer should be deleted
  - Weights for the output layer should be randomly initialized
  - Dataset can be switched to new application and NN retrained
- If the new dataset is small, can just retrain the last layer of the NN
  - If there is a lot of data, all layers in the NN can be retrained
  - Pre-training is the training of the NN for the original application
- Learning basic feature of images from a large dataset can help performance of algorithm

- Transfer learning works best when there is comparatively more data for the initial training
  - Initial training will not be useful if there is more data in the fine-tuning dataset
  - Both tasks must have the same input type
  - Low level features should be helpful for learning B
- For multi task learning, a single NN will try to learn multiple things at a time
  - Each task will ideally help the other tasks
- For self driving vehicles, many objects need to be identified from input data
  - Pedestrians
  - Cars
  - Different types of signs
  - Traffic lights
- Output from NN will be a vector for each object

$$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

  - Output from NN can have all objects in the same image
  - Softmax regression had only one output label for each image
- NN trained to minimize above cost function is using multi task learning
  - Separate NN could have been trained for each object
  - Basic image features for all NN can be shared
- Multi task learning can also be done if the dataset is incomplete
  - Dataset may have missing values for some objects
  - When calculating the cost, missing values can be ignored in the sum
- Multi task learning should have tasks that benefit from having shared lower level features
  - Amount of data for each task tends to be similar
  - Must be able to train a big enough NN to do well on all tasks
- Transfer learning tends to be more common than multi task learning
  - Multi task learning more common in computer vision

3.1.6 End to End Deep Learning

- End to end deep learning takes multiple stages of processing and combines it into a single NN
- For sound recognition:
  - Individual features of the sound (MFCC)
  - Recognizing phonemes
  - Recognizing words
  - Final transcript
- End to end deep learning requires a lot more data than the standard pipeline
  - A medium sized dataset can use a mixture of end to end learning and the standard pipeline
- For an identity detection algorithm using a camera, algorithm will first detect the person's face
  - Algorithm will then crop the image to the face and use the image to identify the person
  - Algorithm will compare new image to all existing images of recognized people
- For each individual step, there is a lot of data for each step
  - Will be a lot harder to find data for both concurrent steps
- End to end deep learning used for machine translation
- Estimating a child's age from an x-ray more suited to different tasks
  - Much easier to identify bones from x-ray before estimating age
  - Possible to use end to end method with a lot of data
- End to end deep learning requires less hand-designing of components
  - Hand-designing components may be constricting the data
- End to end deep learning requires a large amount of data
  - Hand-designed components could be useful when there is comparatively little data

"Do you have sufficient data to learn a function of the complexity needed to map x to y"

4 Convolutional Neural Networks

4.1 Foundations of Convolutional Neural Networks

- Computer vision has benefitted greatly from deep learning
  - Many current applications of computer vision were not possible a few years ago
  - Some ideas in deep learning also transferable across disciplines
- Computer vision can be split into many subareas:
  - Image classification
  - Object detection
  - Neural style transfer
- For computer vision applications, input from an image can be very large
  - 64×64 color image has 12288 features
  - 1000×1000 color image (1 megapixel) has 3000000 features
- For a 1000×1000 image with 100 hidden units in the first layer, $W^{[1]}$ will have 3 billion parameters
  - Computational requirements will be very large
  - Also hard to get enough data to prevent the NN from overfitting
- For an object detection problem, can start by detecting vertical and horizontal edges in the image
  - Using a grayscale image, a filter can be convolved with the image
  - Each pixel in the filter takes an element wise product and sum over the whole filter
- 6×6 grayscale image convolved with a 3×3 gives a 4×4 image

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

  - Above filter used for vertical edge detection
  - Filter represents area in image that has a light section on the left section and dark on the right section
  - Filter will have better performance on larger images

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

  - Above filter used for horizontal edge detection

- Using the same filter, dark to light and light to dark edges will look different
  - Absolute value can be taken if type of edge detected is not needed
- Different numbers may be used for the edge detection filter
  - Sobel filter

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

  - Scharr filter

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

- Numbers in the filter can be learned with backpropagation
  - Can define what type of edge the filter should learn

4.1.1 Padding and Strides

- For a $n \times n$ image with a $f \times f$ filter, dimensions of the result will be $n - f + 1$
  - Dimensions of the image will shrink with every convolution
  - With the standard convolution operation, corner pixel is only used once
  - Pixels in the center of the image will get used many more times
- Image can be padded with a 1×1 border
  - Original image size will be preserved with convolution operation
  - 0s are typically used for padding
- Dimensions of the new image will be $n + 2p - f + 1$
- Valid convolution has no padding on the input
- Same convolution uses padding such that the output is the same size as the input
  - For same convolution, need $p = \frac{f-1}{2}$
- Size of filter is usually an odd number
  - For an even number, asymmetrical padding is needed for same convolution
  - Odd filter will always have a central pixel to the filter
- Strided convolutions change the size of the step taken by the filter
  - Standard convolution uses a stride of 1

- For an $n \times n$ image with an $f \times f$ filter, size of resultant image is:

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

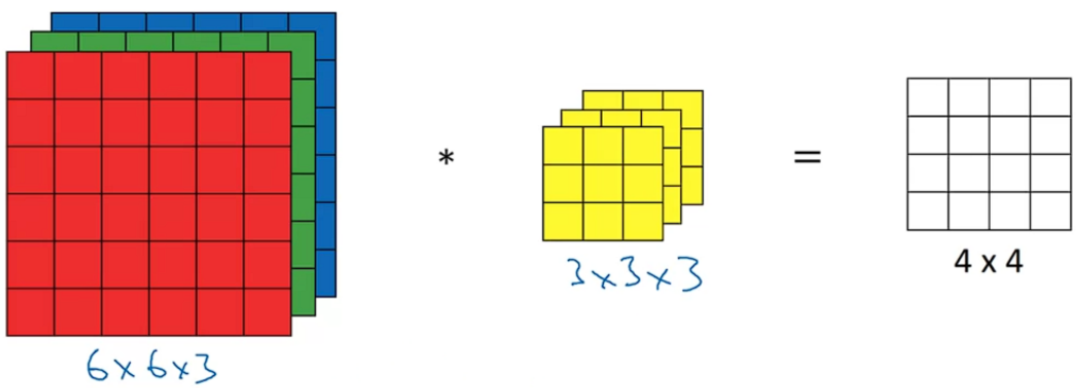
- For a non standard stride length, the filter must be completely within the image for the computation
- Convolution in mathematical literature flips the filter across the horizontal and diagonal before operation

$$\begin{bmatrix} 3 & 4 & 5 \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{bmatrix} \to \begin{bmatrix} 7 & 9 & -1 \\ 2 & 0 & 1 \\ 5 & 4 & 3 \end{bmatrix}$$

  - Convolution operation in deep learning literature known as cross-correlation
  - Flipping the filter in convolution gives associativity to the operation

$$(A * B) * C = A * (B * C)$$

  - Associativity not required for NN so flipping of filter can be omitted
- For a 3 channel RGB image, filter will also have 3 channels
  - Output from the RGB convolution will be a single image



- For RGB image, convolutions in each layer are applied then summed together for each pixel
  - Filter can be set to detect edges in specific colors or all edges
- To detect all edges, vertical filter and horizontal filter can be used
  - Outputs from both filters can be stacked over each other
- Single layer in a convolutional NN will add a bias term and non-linearity to each output
  - Same bias term will be added to all pixels in the image

- If using 10 $3 \times 3 \times 3$ filters, total parameters will be 280
  - Number of parameters is independent of the size of the input
  - Makes CNN less prone to overfitting than standard NN
- For a convolution layer l:
  - $f^{[l]}$ = filter size
  - $p^{[l]}$ = padding
  - $s^{[l]}$ = stride
  - Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
  - Output: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

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

  - Filter: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
  - Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

    $A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
  - Weights: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$
  - Bias: $n_C^{[l]}$
- Each layer in a CNN can have different sizes for padding, filters and stride length
  - A lot of the work for CNNs is choosing the hyperparameters for each layer in the network
- Final output from the CNN can be unrolled and fed to a logistic regression unit to make a prediction
- In a CNN, will have convolution layers, pooling layers and fully connected layers
- Pooling layers reduce the size of the representation and can make detected features more robust
  - Max pooling splits the input into sections and takes the maximum value from each section
  - Max pooling will have a filter size and stride length
  - Max pooling will "preserve" any standout features
- For a 3D input to max pooling, output will have the same 3rd dimension
  - Computation will be applied to each channel separately

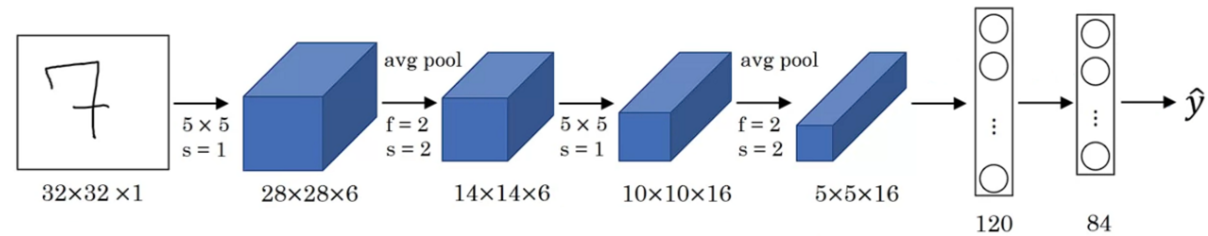
- Average pooling takes the average from each filter
  - Not as commonly used as max pooling
- Padding size of 0 usually used for pooling layers
- A fully connected layer is the same as a layer in a standard neural network
  - FC layer will have W and b parameters
  - Will reduce the dimension of the output of the NN
- Further in the CNN, the height and width of the input will gradually decrease
  - As n_W and n_H decrease, the depth of the input will typically increase
- Typical CNN will have one or more conv layers followed by a pool layer
  - CNN will usually finish with some fully connected layers then a softmax layer
- Conv layers help the network with sparsity of connections
  - Using a $32 \times 32 \times 3$ input image, 6 filters ($f = 5$) will give around 14m parameters
  - Conv layer will have 456 parameters for same calculation
  - In every layer, each output value is depends on only a small number of inputs
- Conv layers use parameter sharing
  - A feature detector (filter) that is useful in one part of an image will likely be useful in another part of the image
- Conv layers and FC layers all have associated parameters
  - Cost function can be defined over the parameters
  - Gradient descent or other optimization algorithm can be used to train the network and reduce J

4.2 Deep Convolutional Models: Case Studies

- Intuition about own deep learning problem can be gained by looking at existing research
  - NN architecture and other ideas may be transferrable to other problems
  - Ideas may also be transferrable to other areas of machine learning

4.2.1 LeNet-5

- Goal of LeNet-5 was to recognize handwritten digits
- NN was trained on grayscale images ($32 \times 32 \times 1$)
- Output from the NN had 10 possible values

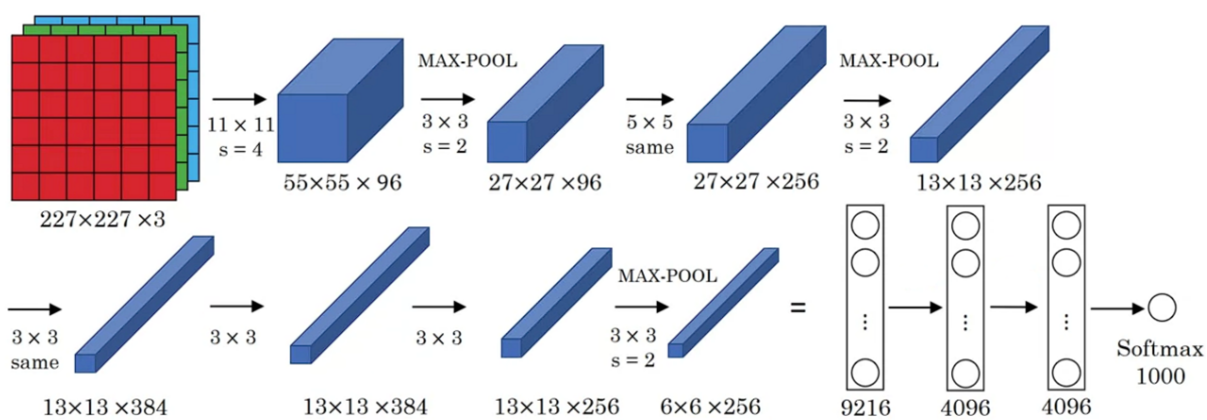


- Modern implementation would use a softmax layer

- When the NN was implemented, no padding was used
- LeNet-5 was "small" compared to other networks
  - Whole NN had 60K parameters
  - Modern NN can have 10m to 100m parameters
- Deeper in the network, n_H and n_W decrease and n_C increase
- Network starts with conv and pool layers, followed by FC layers then output
- Modern computers have the capacity for each filter to have the same number of channels as its input
  - LeNet-5 had a method of making different filters looking at different inputs
- LeNet-5 used sigmoid or tanh activation functions
  - Non linearity was also added after the pooling layers

4.2.2 AlexNet

- Input was a 227 × 227 color image

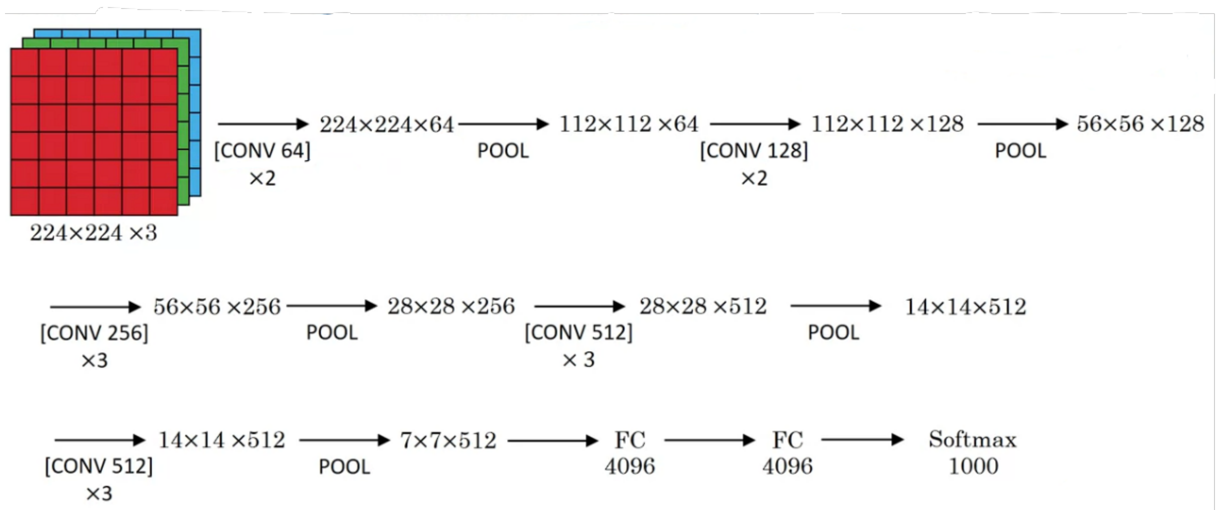


- Similar structure to LeNet-5 but much larger
  - Contains around 60m parameters

- Used ReLU activation functions
- Training of AlexNet was split across multiple GPUs
- AlexNet used a Local Response Normalization layer
  - After some layers, the outputs would be normalized across all the channels
  - Not used very often as research showed layer is not very helpful

4.2.3 VGG-16

- Uses a much simpler network compared to AlexNet
  - Conv layers: 3×3 filter, $s = 1$, same padding
  - Max pool layers: 2×2 filter, $s = 2$
- NN has 16 layers with weights
  - NN has around 138m weights



- NN is much more uniform when compared with other architectures

4.2.4 ResNets

- Very deep networks are hard to train due to vanishing and exploding gradients
- Skip connections use activations from one layer in another layer deeper in the NN
- ResNets created by using a residual block
  - In between $a^{[l]}$ and $a^{[l+2]}$, the activations $a^{[l]}$ will go through two sets of linear and non linear functions
  - $a^{[l]}$ can be added later in the network before the second non-linearity

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

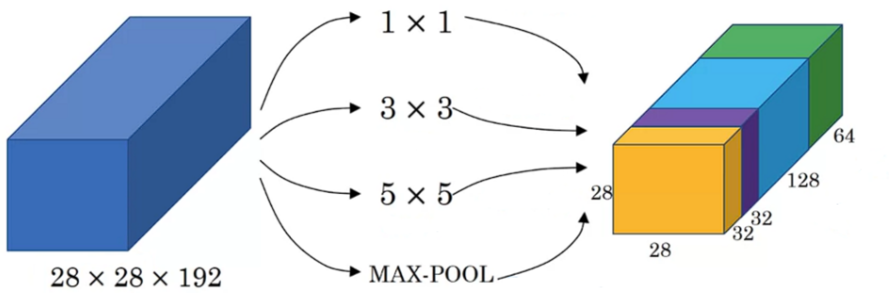
- Residual blocks can be stacked together to form a deep network
  - Residual blocks allow deeper NN to be trained
- For "plain" NN, increasing the number of layers will initially decrease the training error
  - When the number of layers is very large, the NN is hard to train and the training error increases
  - With ResNets, the training error shouldn't increase with the number of layers
- Residual blocks can quite easily learn the identity function
  - Using the ReLU activation, $a^{[l+2]} = g(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$
  - With regularization, W and b will be close to 0
  - $\therefore a^{[l+2]} \approx g(a^{[l]})$
  - Since ReLU activation is used, $a^{[l+2]} \approx a^{[l]}$
- If adding residual blocks is similar to using the identity function, the performance of the network will not be affected
  - Residual blocks can also learn parameters that are better than the identity function
- For ResNets, it is assumed that $z^{[l+2]}$ and $a^{[l]}$ have the same dimensions
  - Same convolutions tend to be used for ResNets
  - If same convolution is not used, $a^{[l]}$ is multiplied by a matrix W_s to create the correct dimension
  - W_s can have parameters that can be learnt or can be a fixed matrix that adds zero padding

4.2.5 Networks in Networks

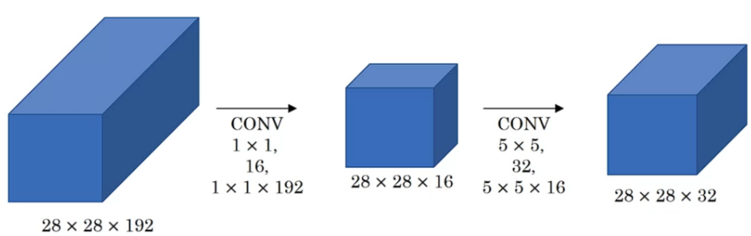
- For a single $1 \times 1 \times 1$ filter, pixels in the image will get multiplied by the filter value
  - If the filter has a depth of more than 1, the filter will take the element wise product of all numbers in the slice
  - Very similar to a neuron taking all the numbers in a slice as input
- Having a 1×1 convolution on an input is the same as having a fully connected NN in each position
- Using 1×1 convolutions known as network in network
- 1×1 convolutions can be used to shrink the depth of an input
  - Pooling layers used to shrink the height and width of the volume

4.2.6 Inception Network

- For CNNs, must choose the size of the filters and order of layers
  - Inception network works with multiple choices of filters and layers at the same time
- Outputs from all possible choices are stacked on top of each other



  - Same padding used for conv layers so outputs have the same dimension
  - Same padding and $s = 1$ must also be used for max pooling layer
  - Output from the layer will be a $28 \times 28 \times 256$ volume
- For the 5×5 filter section of the layer, 120,422,400 calculations are needed
  - Single layer of an inception network can be very computationally expensive
  - 1×1 convolutions can be used to reduce the computational cost by a factor of 10
- 1×1 convolutions can be used in a bottleneck layer to shrink the input



  - With the bottleneck layer, only 12,443,648 calculations are needed
- For an inception module, 1×1 convolution should be used before any filters that have larger dimensions
  - For pooling layers, 1×1 convolutions should be used to shrink the number of channels
- Inception network created from multiple inception modules

4.2.7 MobileNet

- MobileNet networks can be built and deployed in low compute environments
  - Can be used for mobile and embedded vision applications due to low computational cost at deployment
- For a normal convolution:

$$\text{Computational cost} = \#\text{ filter params} \times \#\text{ filter positions} \times \#\text{ of filters}$$

- A depthwise separable convolution separates process into depthwise and pointwise convolution
- Depthwise convolution uses n_C filters of $f \times f$
  - Separate filter used for each channel
  - Output of depthwise convolution will be $n_{out} \times n_{out} \times n_C$
- Pointwise convolution uses filters of size $1 \times 1 \times n_C$
  - ${n_C}'$ filters used to get the correct dimensions in the output volume
- Computational cost of depthwise and pointwise convolutions will be less than the computational cost for a normal convolution
  - In general, the ratio of of computational costs is $\frac{1}{{n_C}'} + \frac{1}{f^2}$
- MobileNet will use a depthwise separable convolution for all convolutions in the network
  - Original MobileNet V1 network had 13 depthwise separable convolutions
  - Last layers of the network were pooling, FC and softmax layers
- MobileNet V2 used residual connections across each layer
  - Convolution also added an expansion layer before the depthwise convolution
  - MobileNet V2 had 17 convolution (bottleneck) blocks with pooling, FC and softmax layers
- Expansion layer similar to the pointwise convolution (projection) but increases the depth of the volume
  - The expansion increases the size of the representation to allow the NN to learn a richer function
  - The projection reduces the depth of the volume to reduce the amount of memory required for the output

4.2.8 EfficientNet

- Specific application can benefit from being scaled to the hardware specifications
- To scale a NN:
  - Higher resolution image
  - Change the depth of the network
  - Change the width of the layers
- r, d, w can be scaled according to available resources
  - Rate of scaling for each variable may not be the same

4.2.9 Practical Advice for CNNs

- A lot of details about CNNs are hard to replicate in practice
  - Open source implementation of code can often be found online
  - Reimplementing the whole algorithm from scratch can help in terms of understanding
- Specific architecture may also take a very long time to train on own device
  - Transfer learning can be used from pre trained networks
- When using a pre trained network, softmax layer can be replaced to suit new application
  - Parameters in the rest of the network can be ignored and softmax output layer can be trained
  - If only the last layer is being trained, the activations input to the softmax layer can be saved separately to prevent extra computation
- If the training set is very big, more layers from the end of the network can be trained
  - Weights from original network can be used for initialization
  - Layers can also be completely removed and trained again from the start
  - Whole network can be retrained if there is enough data
- Most computer vision tasks can benefit from data augmentation
- Mirroring and random cropping are commonly used for data augmentation
  - Mirroring images works well for most applications
  - Random cropping works well as long as the crop is a reasonable subset of the image
- Other methods can be used but can be less effective

- Rotation
- Shearing
- Local warping

- Color shifting can be used in almost all computer vision applications
  - Can help to eliminate any biases caused by specific lighting
  - In practice, color shifting will be more structured (PCA color augmentation)
- When training, a specific CPU thread will be used to apply distortions
  - The data will be processed by the thread then passed to the CPU/GPU for training
  - CPU thread for distortion and CPU for training can run in parallel
- Some applications of deep learning have comparatively more data than other applications
  - Speech recognition has a lot of available labelled data
  - Image recognition has a lot of data but not "enough" for applications
  - Object detection had relatively little labelled data
- Applications with comparatively more data can use simpler algorithms
  - Applications with comparatively less data require more hand engineering of features
- Historically, computer vision relies more on hand engineered features
  - Network architectures are also hand engineered for computer vision
- Researchers also want to do well on benchmarks and win ML competitions
  - Some researchers will use ideas that will specifically help the benchmark
  - Same ideas would not be used in a standard application
- Ensembling can give a slight increase to the performance of an algorithm
  - Several networks are trained independently and the outputs are averaged
  - 3-15 networks can be used but will greatly slow down the running time
- Multi-crop at test time is more computationally expensive and much slower
  - The trained classifier is run on multiple versions of test images and results are averaged
  - 10-crop applies same network to 10 separate crops of the image