Deep learning AI from Andrew Ng Class Notes

Qian Yu

Course 1. Neural Network and Deep Learning

Key Takeaways

Neural Network Basics

- The goal (intuition) of the activation function (layer) is the squeeze the weighted sum of weights (W) and input (a) into the range between 0 and 1 (think probability) => how activate(important) a neuron should be
  - $\sigma(W^{[l]} * a^{[l]} + b^{[l]})$
- Neural Network's activation layer is move away from **Sigmoid** function to **Relu** to improve computation speed because Relu allows faster Graduient Decent converge
- Faster neural network computation allows deep learning practitioner to iterate quickly thus try more different approaches and new ideas
- *Loss function* vs *Cost Function* as a function to find global minimum:
  - *Loss Function* is in respect to a single sample
  - *Cost Function* is in respect to whole dataset.
- The intuition of the goal of cost functions are that we want to make prediction as close as the actual. e.g. Using logistical regression as example: We want to ensure that when true label is 1, the model should predict as large as possible, when true label is 0, the model should predict as small as possible.
- **Computation Graph** is very useful framework to work out forward and backward propagation. It is analogous to chain rule of derivative in calculus
- Numpy for Neural Network implementation:
  - Avoid rank 1 array, it shall be reshaped and `np.sum(keep_dim = True)`: `(5,) => (5, 1)`
  - Leverage assert() function to check matrix shape: `assert(a.shape == (5,1)`
- Neural Network is analogous to multiple steps chained (hierchical function) of logistic regression
  - e.g. $z = w * a_1 + b, a_2 = sigmoid(z)$ are connected together
- Vectorize notation:
  - matrix n x m => Vertically (n) is the dimention of layers, horizontally (m) is the sample size
- Sigmoid is an older type of activation function.
  - `Sigmoid` function only make sense to use if the output layer is for binary classifications (output class label of [0, 1])
  - `tanh(z)` is almost always better than `Sigmoid` because it shift Sigmoid curve and centers at 0. This is similar to normalize data which make the learning the next layer easier.
  - Both `Sigmoid` and `tanh` have weakness of having small derivative of Z which cause gradient decent to be converge slowly.
  - Rectify linear unit `Relu` has faster gradient decent with large slop (derivative). `Relu` is the default choice of activation function nowadays. Leaky Relu is also commonly used
- W, dw Z, dz have the same dimension in forward and backward propagation.

- If you initialize the neural network to be 0, all hidden units become symetric, as a result all hidden units just compute the same function over and over which is not very useful
- We initilize W to be random small non-zero number, but inialize b to be 0 is ok. If we initialize w to be large, it will slow down learning since `tanh` or `Sigmoid` will start at very small slope (e.g. For `Sigmoid` slow become very small, when W is large)
- Use matrix dimension to check Neural Network construct is recommended
  - $w_l = (n_l, n_{l-1})$ and $b_l = (n_l, 1)$ for layer l
  - dw has the same dimension of w, db same as b
  - $Z_l = (n_l, m)$, $A_l = (n_{l-1}, m)$ for layer l
- Deep Neural Network Intuition: it using earlier layers to detect simply features first then user later layers to detect more complex features

  e.g. In image recognition

  - layer 1: figure out edges
  - layer 2: finding different part of faces
  - layer 3: recognize or detect face

General NN Building Methodology

The general methodology to build a Neural Network is to:

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

Model Tuning

- The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data.
- regularization

Reference

- http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/
- https://stats.stackexchange.com/questions/211436/why-do-we-normalize-images-by-subtracting-the-datasets-image-mea
- http://scs.ryerson.ca/~aharley/neural-networks/
- http://cs231n.github.io/neural-networks-case-study/

Course 2: Improving deep Neural Networks

Key Takeaways

Train / Test Split

- Deep Learning is a high iterative process (idea => code => experiment) because it is difficult to estimate hyper params the first time even with experience.
- Solving general ML problem often uses train/validate/test split such as 60/20/20. But in the big data era, dev and test can be a much smaller percentage of total.
  - e.g. 1000000 data, only use the necessary to evaluate e.g. 10000. (98% train, 1% dev, 1% test). This is totally depends on the size of data set
- In general, we want to make sure dev and test data come from the same distribution. In deep learn, It can be OK not to have a test set, but only a dev set.

Bias vs Variance

- Bias vs. Variance: it is less of a trade-off in the deep learning era. We need to pay attention to:
  - High dimensionality => train set error not equal to dev set error
  - If training set error is much larger than human performance (opted (bayes) error), than we have high bias
  - We need to first comparing train set error with bayes error to determine whether there is high bias, then compare train set error with dev set errors for checking high variance.
- Recipe to solve Bias/Variance problem in deep learning:
  - If high bias based on training data performance => use bigger network and train longer, conduct NN architectures search until bias reduced to be acceptable level
  - If high variance based on dev set performance => get more data, apply regularization, and perform NN achitecture search
  - May need to iterate between first 2 steps many times
  - In deep learning and big data era, we don't always need to balance the trade-off because getting bigger network and get more data will often improve bias without hurting the model variance (as long as we do proper reguliarzation)

Initialization

- initialization
  - The weights $W^{[l]}$ should be initialized randomly to break the symmetry.
  - It is okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly.
  - Initializing weights to very large random values does not work well due to the nature of activation function.
  - Intializing with small random values does better.
- Different initializations techniques may lead to different results
  - Random initialization
  - Xivar works will for tanh activation
  - He initialization works well for networks with ReLU activations.

Regularization

- Regularization

  - Although L1 in theory will compress the model size to due to its sparsity natural, but in practice, L1 is less used and less effective.
  - L2 is used much more often in machine learning

- NN regularization:
  - Frobenious Norm (sum of square matrix) also called weight decay. It is to set W close to 0 to make a simplier network to prevent overfitting
    * Intuition: take tanh activatioon, if z is small. Since $z = W * a + b$, then we are mostly in the linear region, make the overall model more linear, thus prevent overfitting
    * To visual examination of regularization: plot J with regularizaiton term. (cost of gradient decent)
  - Drop out regularization: - Inverted dropout to ensure the expected value of Activation stay the same so it will not impact the test prediction by normalize parameter after dropout - Randomly drop out different hidden unit at different model training iteration - No drop out at test time since we are making predict - Intuition: do not rely on any single feature, so we spread out the weights by shrink weights similar to L2 - Computer vision uses drop-out very frequently since usually we don' have a lot of image data - With drop-out, the cost curve plot will not be monotonic, it is recommended to turn-off dopout first, plot the cost curve to ensure the general NN implementation works before turn-on drop out

- Other method to prevent overfitting - Data augmentation: e.g. image random flip, crop, add fake examples - Early stopping: stop when dev set error go back up. (no great to do Orthogonalization) - Orthogonalization: only focus on optimization or prevent overfitting

- Normalize Inputs
  - Subtract mean $x = x - \mu$
  - normalize the variance calculate σ, then do $\frac{x}{\sigma}$ so that all feature of different dimension has same scale of variance
  - Apply the same μ and σ to the test set
  - normaliznge features make the cost function (surface) more evenly contour instead of elongated contour, making the optimization function easier to converge

Vanish Gradient

- Vanish/exploding gradient problem: activation decrease or increase exponentially with the depth of the network
  - Solution 1: Partial carefully weighted random initialization. Different activation has different techniques. - Relu: $\sqrt{\frac{2}{n^{[l-1]}}}$ - tanh uses Xavier $\sqrt{\frac{1}{n^{[l-1]}}}$, or $\sqrt{\frac{2}{(n^{[l-1]} * n^{[l]}}}$
  - gradient checking:
    * reshape all params (W, b) and its derivative (dW, db) to a big vector theta
    * Only for debug need to turn off for training

Batch GD vs Stochastic GD vs Mini-Batch GD

- Batch GD take gradient steps with respect to all m examples on each step
- A variant of this is Stochastic Gradient Descent (SGD), which is equivalent to mini-batch gradient descent where each mini-batch has just 1 example.

(Batch) Gradient Descent:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost += compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

Stochastic Gradient Descent:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost += compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

- Mini batch gradient decent is in between batch gradient decent and stochastic gradient decent with better performance and speed when having large sample size
  - The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
  - You have to tune a learning rate hyperparameter α.
  - With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).
- Implementation of mini-batch
  - if sample size (m) < 2000 use batch
  - Random Shuffling samples
  - Partitioning samples with mini-batch size
  - typical mini-batch size 64, 128, 256, 512, 1024

Mini-batch Gradient Decent

```
X = data_input
Y = labels
# Shuffle data samples
shuffle_X = randam.permutation(X, seed = 1)
shuffle_Y = randam.permutation(Y, seed = 1)
parameters = initialize_parameters(layers_dims)
# Determine the partition
num_of_batches = math.floor(m/mini_batch_size)
# Create mini_batch list by partition data
mini_batches = build_mini_batch(shuffle_X, shuffle_Y)
for i in range(0, num_of_batches):
    # Forward propagation
    a, caches = forward_propagation(minibatch_X[:,j], parameters)
    # Compute cost
    cost += compute_cost(a, minibatch_Y[:,j])
    # Backward propagation
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

Momentum

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

- Momentum takes into account the past gradients to smooth out the update.
  - store the 'direction' of the previous gradients in the variable v.
  - Formally, this will be the exponentially weighted average of the gradient on previous steps.
  - v as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.
- Exponentially weighted average

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

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

- where L is the number of layers, β is the momentum and α is the learning rate. All parameters should be stored in the `parameters` dictionary. Note that the iterator `l` starts at 0 in the `for` loop while the first parameters are $W^{[1]}$ and $b^{[1]}$ (need to shift `l` to `l+1` when coding).

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.

- You have to tune a momentum hyperparameter β and a learning rate α.

- Momentum implementation and select β

  - The velocity is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps.
  - If $\beta = 0$, then this just becomes standard gradient descent without momentum.
  - The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
  - Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.
  - Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.

Adam

- Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) and Momentum.

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).

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

- The update rule is, for $l = 1, ..., L$:

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

- where:
  - t counts the number of steps taken of Adam
  - L is the number of layers
  - β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
  - α is the learning rate
  - ε is a very small number to avoid dividing by zero
- Implementation
  - store v and s parameters in the `parameters` dictionary

Hyperparameters

- Importance of hyperparameters
  - prioirty 1: learning rate
  - priority 2: β (0.9 is a good value), num of hidden units, mini-batch size
  - priority 3: num of layers, learning rate decay, Adam parameters (β_1, β_2, ϵ) almost no need to tune
- Do **NOT** use grid search approach for hyperparameter tuning for deep learning (too costly due to over-tuning the less important parameters)
  - Use random search approach
  - Use coase to fine: zoom into the region of good hyperparameter space
- Use scale for hyperparamter tuning
  - Sample on the log scale for hyper-parameter to evently cover the space as α and β are in log scale
- Hyperparameter tuning process / Method
  - Babysitting 1 model if no computation resources
  - Train many models in parallel if there are a lot of computation resources
- Batch normalization as a good, faster approach to select and tune hyperparameters
  - Idea: since normalize inputs helps improving training, normalize hidden units $A^{[l]}$ (or better $Z^{[l]}$) can also speed up training of $W^{[l]}$ and $b^{[l]}$
  - ϵ is to prevent divided by 0
  - γ and β allow us to make hidden units that has mean diff from 0 and variance diff from 1
  - β here is different from Adam's β parameter
  - Batch normal applies in mini-batch GD
  - Batch normal eliminate $b^{[l]}$ (make them 0) since it zero out the mean of hidden layer $Z^{[l]}$ thus $b^{[l]}$ is meaning less, instead of we add $\beta^{[l]}$

$$\begin{cases} \nu = \frac{1}{m}\sum Z^{(i)} \\ \sigma^2 = \frac{1}{m}\sum (Z_i - \mu)^2 \\ Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \hat{Z}^{\ (i)} = \gamma Z^{(i)}_{norm} + \beta \\ \gamma = \sqrt{\sigma^2 + \epsilon} \beta = \mu \end{cases}$$

- Batch Normal Implementation

```
for t = 1 ... num of mini_batches:
  compute forward pop on X[t]
  in each hidden layer us Batch Norm to replace Z[l] with Z_new[l]
  use backpro to compute dW, dbeta, dgamma for each layer
  update params by doing
    W[l] = W[l] - alpha*dW[l]
    beta[l] = beta[l] - alpha*dbeta[l]
    gamma[l] = gamma[l] - alpha*gamma[l]
```

```
- It works with batch GD, mini-batch GD, Adam (RMSprop)
```

- Batch Normalization Intuition
  - It help the later layer in the network to be less impacted by the shifing/changes of early layer's hidden units as it uses β and γ parameters to govern. It make the later layers learn faster
  - It also has a slight regularization effect since it adds some noise to values of hidden unit's activation within that minibatch (similar to dropout)
- Batch Norm at test time
  - Unlike training time we don't have minibatch size to calculate mean and sigma during the test time.
  - Use exponentially weighted average (running average) (across mini-batch) to keep track of mean and sigma of each layer

Softmax Regression (multi-class)

- Softmax generalized logistic regression to C classes
  - Softmax activation layer with 4 classes

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

$$a_i^{[l]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

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

Deep Learning Framework

- How to choose?
  - Ease of programming (development and deployment)
  - Running speed
  - Truly open source (good governance)
- Tensor flow
  - It is basically building a computation graph
  - Only need to build forward prop and backprop is auto-created

Reference

- Adam paper: https://arxiv.org/pdf/1412.6980.pdf

Course 3 Structure Machine Learning Projects

Key Takeaways

Machine Learning Strategy

- Orthogonalization process: have distinct set of knobs to try for each step of the following ML process

```
1. Fit training set well on cost function (close to human level performance)
    - e.g. Bigger network, Different optimizer such as Adam
2. Fit dev set well on cost function
    - e.g. Different regularization, bigger training set
3. Fit test set well on cost function
    - e.g. bigger dev set
4. Performs well in real world
    - e.g. Change dev set or cost function
```

```
- Note: `early stop` is not an orthogonal knob. It simultaneously affect both training set fit and dev s
```

Set up ML Goals

- Use single real number evaluation metric
  - Having a good dev set + single real number evaluation metric to speed up the iteration process of improving ML algorithm (Idea => Code => Experiment)
  - e.g. Use F1 score instead of both precision and recall
  - e.g. In stead of track ML models performace across different segments use average performance of all segments
- Satisficing and optimizing metric
  - e.g. Accuracy is optimizating metrics => to maximize. Running time is satisicing metric => to meet a criteria
  - If we have N metric: pick 1 for optimizaing metrics, N-1 will be satistifing metrics
- Train/dev/test set distributions
  - dev set + metric define the target
  - dev set and test set should have same distribution
  - Choose a dev set and test set to reflect data expecting to get in the future and consider important to do well on
- Size of dev and test sets
  - when sample size is < 10,000 (old era), we normally do 60% train, 20% dev, 20% test.
  - when sample size is >> 10,000 (new era), we do 98% train, 1% dev, 1% test.
    - Set test set to be big enought to give high confidence in the overall performance of the system
    - Train + dev set (without test set) maybe OK for certain application
- When to change dev/test sets and metrics
  - When business and stakeholder's insight require the changes of metrics
  - The future source of data (test data) may force to change the metrics
  - Iterate fast but make changes when those needs happens

Model Performance

- Humen-level performance
  - Deep-learning algorithm often surpass human-level performance
  - Deep-learning algorithm cannot surpass bayers optimal error (very best theoretical function for mapping from x to y which can never be passed)

  - Humans are good at many tasks, as long as ML is worse than human, we can apply the following tactics:
    - Get labelled data from humans
    - Gain insight from manual error analysis: why did a person get this right?
    - Better analysis of bias / variance
- Avoidable bias
  - Use Human-level error as a proxy for Bayes error (performance of a human expert on a task) instead of 0% (as in general ML)
  - If training error $\gg$ human-level error, focus on improving bias
  - If training error $\approx$ human-level error and dev error $>$ human-level error, focus on improving variance
  - Use human-level error as a proxy for bayers error (computer vision, NLP)
  - Avoidable bias = training error - human-level error (bayer's error)
  - Avoidable variance = dev error - training error
- Surpassing human-level performance ML Applications
  - With Structured data, not natural preception problems, with a lot of data
    - Online advertising
    - Product recommendations
    - predict transit time
    - Loan approvals
  - preception problems:
    - Speech recognition
    - Some image recognition
    - Medical: EGG, Skin cancers
- Improve model performance
  - Assumptions:
    - Fit the training set pretty well => Achieve low avoidable bias
    - The training set performance generalized well to the dev/test set
  - Process:
    1. Measure avoidable bias = training error - Human-level error and variance = Dev error - Training error
    2. If avoidable bias is the main problem
       - Traing bigger model
       - Traing loger/better optimization algorithmes (momentum, RMSprop, Adam)
       - NN architecture/hyperparameters search (try CNN, RNN)
    3. If variance is the main problem
       - More data of dev set
       - Regularization (L2, dropout, data augmentation)
       - NN architecture/hyperparameters search

Error Analysis

- Error analysis procedure
  - Manually exam mislabled dev set examples and calculate % of mislabel determine the upper bound impact (ceiling in performance) of making the correction
  - Expand similar process to multiple ideas in parallel (a table)
- Deal with incorrectly labeled example
  - Deep learning algorithm are robust to **random** errors in the training set (if your total sample size is large and error % is small)
  - Deep learning algorithm are NOT robust to **systematic** errors
  - Discover and mark mislabel via error analysis
  - Whether to fix it?
    - Does it bring significant value to improve dev set errors? What is % of errors are from mislabel

        error?
        * Keep in mind the goal of dev set is to select between classifier A or B
    - Apply same process to dev and test set to make sure they continue to come from the same distribution
    - Consider examing examples algorithm got right as well as got wrong
    - If train and dev/test data may come from slightly different distribution after this process but this is OK.
- Build the first system quickly then iterate
    - quickly setup a dev/test set and metric
    - Build initial system quickly
    - Use bias/variance/analysis & error analysis to prioritize next steps

Mismatched training and dev/test set

- If we just merge train and dev/test set and randomly shuffle => make train/dev/test have the same distribution
    - But the % of data in the dev/test set that come from another distribution is much higher than training set
- Have dev/test has the same distribution (from 1 source), add some data of that source to training set
    - Although training set will have different distribution from dev/test set, but it will product better performance in the long run
- If training and dev/test data come from different distribution, we can no longer use the same bias/variance analysis method
    - Create a new train-dev set (carve from training set) that has the same distribution of training set
    - We compare human level error vs train error vs train-dev error vs dev error to determine whether it is bias or variance problem
        * training set error - human-level error => avaiable bias
        * training dev set error - train set error => variance
        * dev set error - train-dev set error => data mismatch
        * test set error - dev set error => degree of overfitting to the dev set
    - It is possible that dev/test set error is better than training and train-dev set, this means we train on harder data for the deep learning algorithm
        * We need to determine the human level error of the dev/test data set
- Address data mismatch
    - Carry out manual error analysis to learn the difference
    - Using artificial data synthesis or other technique to reduce difference between training and dev/test

Transfer Learning

- One can take the knowledge the deep NN learned from 1 tasks and apply that knowledge to a seperate task.
- For example:
    - You can train an image classifier task then apply on a different image classification task (radiolog diagnosis)
    - If the new task has small data set, retrain the last layer (keep other layer fixed) or modified the last few layers
    - If the new task has large data set, retrain all layer (keep the same architecture)
        * The first task is pretraining
        * The next task is fine tuning
    - Intuition
        * Low leverl features (such as edges, shapes, etc) can be learnined with earlier layers (regardless of data set)
    - When make sense

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

Multi-task learning

- In image recognition, we can learn multiple tasks using the same deep NN
- When make sense
    - Training on a set of tasks that could benefit from having shared lower-level features
    - Amount of data you have for each task is similar
    - Can train a big enought NN to do well on all the tasks

End to end deep learning

- Put multiple statge of deep learning to 1 deep NN (simplified the system)
- examples:
    - Audio $\rightarrow$ features $\rightarrow$ phonemes $\rightarrow$ words $\rightarrow$ transcript
    - audio $\Longrightarrow$ transcript (end to end)
    - Image recognition
    - Machine translation
    - Estimation of child's age
- When to use?
    - Pros:
        * Let's the data speak
        * Less hand-esigning of components needed
    - Cons:
        * May need large amount of data of both input end and output end
        * Exclude potentially useful hand-designed components (human knowledge)
    - Ask
        * Do we have sufficient data to learn a function of the complexity needed to map x to y?

Course 4 Convolutional Neural Networks

Key Takeaways

Computer Vision Problems

- Applications
    - Image classification
    - Object detection
    - Neural style transfer
    - Challenge
        * Large image has very high dimension (1000 x 1000 x 3 with 1000 samples = 3 Million dimensions)

Edge detection

- Convolution operation (*) astris with filter
    - perform element wise multiplication and add up
    - move 1 step and repeat

- e.g. 6×6 matrix convolve with a 3×3 filter produce a 4×4 resulting matrix
- Vertical edge detector
  - 3×3 pixel (filter) with bright edge on right, dark edge on the right, (don't care about the middle)
  - e.g. $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$
- Horizontal edge detector
  - e.g. $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$
- Dark to light or light to dark edge transition

$$\text{light to dark edge transition} \begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

$$\text{dark to light edge transition} \begin{bmatrix} 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \end{bmatrix}$$

- There are different kinds of filters
  - Sobel filter
  - Schorr filter
  - We can learn the filter by training the parameters
  - Unlining convolution operation is the key

Padding

- $n \times n$ image convolve with $f \times f$ filter (without padding) yields $(n - f + 1) \times (n - f + 1)$ image, image shrink quickly after a few layers of filter
- The corner of the image used less than the center of the image (loss of information)
- We pad 0s around the edge (both size 2p)
  - $(n + 2p) \times (n + 2p)$ conolve with $f \times f$ gives $(n + 2p - f + 1) \times (n + 2p - f + 1)$ size image
  - p can be 1 or 2 or more
- Valid and Same convolutions
  - **Valid**: no padding $n \times n * f \times f \Rightarrow (n - f + 1) \times (n - f + 1)$
  - **Same**: Pad so output size is the same as the input size. $(n + 2p) \times (n + 2p) * f \times f \Rightarrow (n + 2p - f + 1) \times (n + 2p - f + 1)$ where $p = \frac{f-1}{2}$
  - The size of the filter f is usually odd number since we can get natual integer padding since $p = \frac{f-1}{2}$. The odd size filter also give us a center position

Strided Convolution

- Instead of convolve with step of 1, we move with a stride size s
  - $n \times n * f \times f \Rightarrow (\frac{n+2p-f}{s} + 1) \times (\frac{n+2p-f}{s} + 1)$
  - If the filter stride over across the edge of image, we will not calculate the output. As a results, the dimension of the output image will be floored $floor(Z) = \lfloor Z \rfloor = \lfloor (\frac{n+2p-f}{s} + 1) \rfloor \times \lfloor (\frac{n+2p-f}{s} + 1) \rfloor$

- The convolution of deep learning is also called cross-correlation in signal processing languague. The convolution in signal processing require additional step of fliping the filter matrix which we do not do in image processing.

Convolution of Volumes (RGB images)

- $n \times n \times 3 * f \times f \times 3 \Rightarrow (\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1)$
    - Each convolution operation now have $n * n * 3$ number of multiplcations then **sum them up** to 1 number output. 3D image * 3D filter = 2D output
    - $f \times f \times 3$: we can have a filter that does different operation for different color channels
- Multiple filter: We have apply multiple filter to the input image e.g. vertical edge, horizontal edge, we will product 3D output where the last dimension is the num of filters
    - $n \times n \times n_c * f \times f \times n_c \Rightarrow (\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1) \times n_c^{'}$
        - n_c num of color channel (usually 3) (Sometime it als called the depth of volume)
        - $n_c^{'}$ num of different filters

One layer of a convolutional net

- $n \times n \times 3$
    - $*f \times f \times 3 \Rightarrow Relu((\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1) + b1)$
    - $*f \times f \times 3 \Rightarrow Relue(\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1) + b2)$
    - $\dots$ $n^{'}$ filters => $n_c^{[l+1]}$ number of filter of previous layer is the number of channel of the current layer
    - $\Rightarrow (\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1) \times n^{'}$
- Reference to general NN, assume 3 color channel, 5 filters

$$Z^{[i]} = W^{[i]}a^{[i-1]} - b^{[i]}$$
$$a^{[i]} = g(Z^{[i]})$$

where

$$a^{[i-1]} : n \times n \times 3$$
$$W^{[i]} : f \times f \times 3$$
$$Z^{[i]} : [(\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1)] + b$$
$$a^{[i]} : Relu(Z^{[i]}) = (\frac{n+2p-f}{s}+1) \times (\frac{n+2p-f}{s}+1) \times 5$$

- Example: 10 filters of 3x3x3 in 1 layer = 3 * 3 * 3 * 10 + 10 = 280 parameters
- Parameter size is irelevant of input image size making it less prone to overfitting
- Formal notation:
    - $f^{[l]}$ = filter size (layer l)
    - $p^{[l]}$ = padding size (layer l)
    - $s^{[l]}$ = stride size (layer l)
    - $n_c^{[l]}$ = num of filters (layer l)
        - Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
    - Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ (Height x Width x Channel)
    - Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ (Height x Width x Channel)
    - Activation: $a^{[l]} : n[l]_H \times n[l]_W \times n[l]_c$
    - batch size (mini-batch size): m

  - Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
  - Bias: $n_c^{[l]} \Rightarrow (1, 1, 1, n_c^{[l]})$
  - where:

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

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

$$A^{[l]} \Rightarrow m \times n[l]_H \times n[l]_W \times n[l]_c$$

A Typical Convolutional Neural Network (ConvNet)

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

Pooling layer

- Max pooling (mostly used)
  - max of element of the regions (e.g. 2 x 2)
    * Hyperparameters: f = 2, s = 2
  - Intuition: the max elemnt of a region usually represent important features detected.
  - No parameters to learn from max pooling
  - Output size of max pooling (same as conv) $\lfloor \frac{n+2p-f}{s} - 1 \rfloor$
  - Max pooling perform independently on each channels
- Avg pooling
  - Very deep NN sometimes we use it.
- Hyperparameters of pooling layer
  - f: filter size (common: f = 2, 3)
  - s: stride (common: s=2)
  - Max or average pooling
  - Usually do not use padding p = 0
  - no parameters to learn

$$n_H \times n_w \times n_c \Rightarrow \lfloor \frac{n_H + 2p - f}{s} + 1 \rfloor \times \lfloor \frac{n_W + 2p - f}{s} + 1 \rfloor \times n_c$$

CNN Example (typical)

- LeNet-5 similar: input -> conv1 -> pool1 -> conv2 -> pool2 ->(flatten)-> FC3 -> FC4 -> softmax

$$input : 32 \times 32 \times 3 \Rightarrow f = 5, s = 1$$

$$layer1 : 28 \times 28 \times 8 (conv1) \Rightarrow f = 2, s = 2$$

$$layer1 : 14 \times 14 \times 8 (pool1)$$

$$layer2 : 10 \times 10 \times 16 (conv2) \Rightarrow f = 2, s = 2$$

$$layer2 : 5 \times 5 \times 16 (pool2)$$

$$400 \times 1 (flattern) \Rightarrow 120 \times 1 (FC3)$$

$$84 \times 1 (FC4) \Rightarrow softmax \text{ 10 outputs}$$

- Each Layer Break downs

	Activation shape	Activation Size	# parameters
Input (f=5, s=1, p=0, n_c=8)	(32 x 32 x 3)	3072	0
CONV1 (f=2, s=2, p=0, n_c=8)	(28 x 28 x 8)	6272	(5x5+1)x8 = 208
POOL1 (f=5, s=1, p=0, n_c=16)	(14 x 14 x 8)	1568	0
CONV2 (f=2, s=2, p=0, n_c=16)	(10 x 10 x 16)	1600	(5x5+1)x16 = 416
POOL2	(5 x 5 x 16)	400	0
FC3	(120 x 1)	120	(120x400+1) = 48001
FC4	(84 x 1)	84	(84x120+1) = 10081
softmax	(10, 1)	10	(10x84+1) = 841

Why Convolutions

- Parameter share: A feature detector (edge detector) that is usefl in one part of image is also useful in another part
- Sparsity of connections: in each layer, each output value depends only on a small number of inputs => Much smaller number of parameters to train, less pone to overfitting. Good at capture **translation variance** (if some pixal shifted, it can still capture the property)

Why Case Study

- Neural network architecture often works well on similar problems
- To Read classic research paper
- Classic networks:
  - LeNet-5
  - AlexNet
  - VGG
  - **ResNet** (Very deep)
  - **Inception Network**

Classic Network Architecture

- LeNet - 5
  - Pattern: Conv + pool -> Conv pool -> fc -> fc -> output
  - In the past
    - People use sigmoid/tanh instead of Relu
    - Apply different filter on different channels (save computation and parameters), nowadays we apply the every filter to every channel since we have more computer power
  - Focus on Section II and III if reading this paper

$$input : 32 \times 32 \times 1 \rightarrow f = 5, s = 1$$

$$28 \times 28 \times 6 (conv1) \rightarrow f = 2, s = 2$$

$$14 \times 14 \times 6 (avg\ pool) \rightarrow (f = 5, s = 1)$$

$$10 \times 10 \times 16 (conv2) \rightarrow f = 2, s = 2$$

$$5 \times 5 \times 16 (avg\ pool)$$

$$120 \times 1 (FC)$$

$$84 \times 1 (FC)$$

$$softmax \rightarrow \hat{y}$$

- AlexNet
  - much larger than LeNet - 5 with 60M parameters
  - Use Relu
  - Use multiple GPUs (but outdated)
  - Use local response normalization (outdate approach less effective now)
- VGG - 16
  - Simpler with less hyperparameters
  - CONV with 3x3 filters, s=1, same; max-pool = 2x2, s=2
  - 16 layer with 138 M parameters
  - Uniform across each layer, Double filter every step

ResNets

- Residual block
  - Repalce $a^{[l]} \rightarrow linear\ activation \rightarrow Relu \rightarrow a^{[l+1]} \rightarrow linear\ activation \rightarrow Relu \rightarrow a^{[l+2]}$ with a short cut before the Relu activation
  - $Z^{[l+2]} = W^{[l+2]} * g(W^{[l+1]} * a^{[l]} + b^{[l+1]}) + b^{[l+2]}, a^{[l+2]} = g(Z^{[l+2]}) \Rightarrow a^{[l+1]} = g(Z^{[l+2]} + a^{[l]})$
  - Short-cut is also refer to as skip connection => Pass information deeper in the NN network
  - Stack residual blocks to a deep residual network (vs. plain network)
  - ResiNet allow the training error to decrease with num of layer increase much deep into the network
    * Training error of a plain network will bounce up as the num of layer increase into a deep network.
- Why ResNets work
  - Given A network with num of layer = l and additional residual block
    * $a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(W^{[l+2]} * a^{[l+1]} + b^{[l+2]} + a^{[l]})$
    * If W^{[l+2]} = 0, b^{[l+2]} = 0 (or very small due to regularization)
    * $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$ We are training a identical function which is an easy tasks => This shows that adding residual block does not hurt the learning even with additional layers and may learn things.
  - Residual blocks assums $Z^{[l+2]}$ and $a^{[l]}$ has the same dimension, therefore ResNet often use Same filter
    * If the dimension is different we need to do $g(Z^{[l+2]} + W_s * a^{[l]})$ where W_s is a matrix to align the dimension. or We can even add 0 padding to match dimension

1x1 Convolutions (Network in network)

- An inmage convolve with 1x1 filter across a number of channels (a slice) then apply Relu non-linearity
  - Input: 6x6x32 * 1x1x32 (filter) and apply Relu to the sum => essentially a full connected affine layer with activation => 6x6 output (similar to 1 layer of CNN with 1 filter)
  - We can apply multiple filterss 6x6x32 * 1x1xn_cx#filter = 6x6x#filters
  - It can be used to shrink the number of channels
    * with 28x28x192 input, we can use 32 of 1x1x192 filters to shrink the channel of the volume to 28x28x32
    * As we can use pooling layer to shrink n_W and n_H, we can use CONV 1x1 to shrink the n_c (save compution)

Inception Network

- The idea is the apply different sizes of filter (e.g. 1x1, 3x3, 5x5, event max-pool) on the input CONV layer and stack them together
  - filter will be same to keey the output layer the same dimension $n_W x n_H$ as input
  - We need to pad max-pool to keep the same dimension
  - Motivation: instead of pick the right filter, we try them all and let the network learn
  - High computational cost
    - But we can 1x1 convolution to reduce the volume first than apply a smaller filer to get the same output layer (dimension)
    - Without 1x1 convolution: Input: 28x28x192 -> output: 28x28x32, We use 32 of 5x5x192 filters
      - Cost of computing is 28x28x192 x 5x5x192 = 120M
    - With 1x1 convolution: Input: 28x28x192 -> 28x28x16 -> output: 28x28x32, we use 16 of 1x1x192 to shrink the input to conv of 28x28x16, then we use 32 of 5x5x16 to get 28x28x32 output
      - 28x28x16 is the bottleneck layer to shrink the computation
      - Cost of computation: 28x28x16 x 1x1x192 = 2.4M (first step) + 28x28x32 x 5x5x16 = 10.0M = 12.4M, we shrinked 10x
- Inception Network is inception module repeated multiple times
- Inception module
  - Previous activation
    - $\rightarrow CONV\ 1 \times 1 \times 192(64) \rightarrow 28 \times 28 \times 64$
    - $\rightarrow CONV\ 1 \times 1 \times 192(96) \rightarrow CONV\ 3 \times 3 \times 96(128) \rightarrow 28 \times 28 \times 128$
    - $\rightarrow CONV 1 \times 1 \times 192(16) \rightarrow CONV\ 5 \times 5 \times 16(32) \rightarrow 28 \times 28 \times 32$
    - $\rightarrow maxpool$ 3×3(s=1, same) $\rightarrow CONV\ 1 \times 1 \times 192(32) \rightarrow 28 \times 28 \times 32$
  - Channel concat 28x28x(64+128+32+32) = 28x28x256

Leverage open source and transfer learn

- For computer vision task, we should always start with transfer learning
- Take advantage of open source code to get started
- Use transfer learning:
  - If have small training dataset, can reuse (freeze) the network, swapout the final softmax layer with your own and just train the softmax layer parameters
  - To speed up training, precomputer the layer of activation before the softmax and save to disc, then just train the softmax
  - If have larger dataset, freeze fewer layers, train later layers or swap the last few layer with your own layers
  - If have a lot of data, keep the same architecture, retrain the weight of the whole network

Data Augmentation

- Computer vision tends to have less data than we needs, data augmentation can solve this
- Methods:
  - Mirroring
  - Random cropping
  - Rotation
  - Shearing
  - local warping
  - Color shifting
    - distorting RGB channel

        * PCA color augmentation (AlexNet paper)

Keras Framework

- Excellent for fast prototyping but with more restriction as it adds a level of hierchy
- Assign output to a new layer as connection
- 4 steps process
    1. Create the model by constructing layer one by one
    2. Compile the model by calling `model.compile(optimizer = "...", loss = "...", metrics = ["accuracy"])`
    3. Train the model on train data by calling `model.fit(x = ..., y = ..., epochs = ..., batch_size = ...)`
    4. Test the model on test data by calling `model.evaluate(x = ..., y = ...)`
- If run fit again, it will train from what left from the previous run
- MaxPool layer has defaut stride size as the filter W/H dimension size

Object Localization

- Image classification => Classification with localization => Detection
- localization / classification: 1 object
- Dtection: multiple objects
- Classification with localization
    - class
        * 1 pedestrian
        * 2 car
        * 3 motorcycle
        * 4 background
    - in addition to class also output bounding box: bx, by, bn, bw
        * bx, by is the center location
        * bn, bw is the size of the bounding box
    - Target label y becomes $y = \begin{bmatrix} P_c \\ bx \\ by \\ bh \\ bw \\ c1 \\ c2 \\ c3 \end{bmatrix}$ where P_c indicate whether there is any object?
        * If there is a car, then $y = \begin{bmatrix} 1 \\ bx \\ by \\ bh \\ bw \\ 0 \\ 1 \\ 0 \end{bmatrix}$, If there is no object, $y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$ where ? is `don't care`
    - Loss function:
        * if $y_1 = 1$. $L(\hat{y}, y) = (\hat{y_1} - y_1)^2 + (\hat{y_2} - y_2)^2 + ... + (\hat{y_8} - y_8)^2$
        * if $y_1 = 0$. $L(\hat{y}, y) = (\hat{y_1} - y_1)^2$
- landmark detection
    - Annotate image with landmarks to identify face features or pose
    - The order of the label has to be consistent across all the images (e.g. l1x, l1y is outer corner of left eye, ...)

Object Detection Algorithm

- with sliding window
  - Start with cropped image as labeled training set with a ConvNet
  - Apply a sliding window the full image and slide the window across all region of the image
  - Apply a slightly larger sliding window the full image and slide the window across all region of the image
  - Apply a even slightly larger sliding window the full image and slide the window across all region of the image
  - The goals the capture the object in the image with the sliding window
  - High Computational cost as we ran many convNet
- Implement sliding window convolutionally
  - Implement FC as convNet
    * 14x14x3 convNet 5x5x16 –> 10x10x16 maxpool(2x2) –> 5x5x16 –> FC 400 –> FC 400 –> softmax(4)
    * 14x14x3 convNet 5x5x16 –> 10x10x16 maxpool(2x2) –> 5x5x16 convNet 5x5x400 –> 1x1x400 convNet 1x1x400 –> 1x1x400 convNet 1x1x4
  - Share computation combines all in 1 forward pass
    * run sliding window of 14x14 on 28x28x3 with stride of 2 (run 64 times the above convNet) is the same as the convNet below:
    * 28x28x3 convNet 5x5x16 –> 24x24x16 maxpool(2x2) –> 12x12x16 convNet(5x5x400) –> 8x8x400 convNet(1x1x400) –> 8x8x400 convNet(1x1x4) –> 8x8x4
- Bounding box prediction
  - sliding window may not capture the full object
  - YOLO (You only look once) algorithm fix this
    * Use n x n grid
    * Apply image classification and localization to each grid cells
    * for each grid cell $y = \begin{bmatrix} P_c \\ bx \\ by \\ bh \\ bw \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}$, If there is no object.
    * The center of the object determine whether an object belong to a grid cell (object may overlap mutliple grid cell)
    * Target out volume is n x n x 8 since y has 8 dimension
    * e.g. Input: 100 x 100 x 3 –> Conv -> Maxpool –> . . . . -> 3x3x8
    * If there is an object, we can find the bounding box of the object (assume 1 object per grid cell)
    * Use convolution implementation with single pass (efficient and run fast)
    * How encode bounding box? Grid top left (0, 0), bottom right (1, 1). The center of the bounding box is relative to (0, 0) and (1, 1). The height/width of bounding box can be > 1 if the bounding box is outside of grid cell
- Intersection over union to evaluating object localization
  - $IOU = \frac{size\ of\ intersect}{size\ of\ union}$
  - **correct** if IoU >= 0.5
- Non max Supression
  - cleans up multiple detection of the same object from different grid box

  - Select max probablity bounding box and get rid of non-max ones.
    - Use P_c (the probability of the detection) to select the highest
    - Supress the remaining high IOU bounding box
  - Implementation:
    - output prediction is $y = \begin{bmatrix} P_c \\ bx \\ by \\ bh \\ bw \end{bmatrix}$
    - Discard all boxes with $P_c <= 0.6$
    - while there are any remaining boxes:
      - pick the box with largest P_c output that as prediction
      - Discard any remaining box with IoU >= 0.5 with the box otuput in the previous step.

- Anchar Box
  - Each object is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU
    - different Anchor boxs may fit better for different objects
  - object => (grid cell, anchor box)
  - Allow specialize of different objects

- YOLO Object Detection Algorithm
  - Training set (3 object class example) y is 3x3x2x8 (3x3 grid cell, 2 anchar box, 8 = 5 bounding box + 3 class)
    - In matrix form: $y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$
    - input: 100x100x3 –> convNet –> 3x3x16
  - Prediction
    - ConvNet to detect whether there is an object $P_c = 0/1$ and bounding box
    - Apply non-max supress algorithm
      - For each grid call, get predicted bounding box
      - remove low probability predictions
      - For each class, use non-max suppression to generate final predictions

- Region proposal (R-CNN)
  - Perform a segmentation algorithm and identify (Propose) the region to run CNN
  - Fast R-CNN (use convolution implementation)
  - Faster R-CNN: use convolutional network to proposal regions.

Image Recognition

- image verification (1 to 1) => Image recognition of k faces (1 => K)
- One shot learning: learning from 1 example to recognize the person again
  - Learning a "similarity" function: d(img1, img2) = degree of difference between images
  - if d(img1, img2) $\leqslant$ t "same"
  - if d(img1, img2) $>$ t "different"
- Siamese network
  - train 2 images through the same CNN network (same parameters) to get an encoding representation (embedding)
  - $d(x^{(1)}, x^{(2)}) = \|f(x^{(1)}) - f(x^{(2)})\|_2^2$
- Triplet Loss (how we can define the loss function for Siamese network)
  - anchar (A) <=> positive (P) ; anchar (A) <=> Negative (N)
  - Want: $\frac{\|f(A)-f(P)\|^2}{d(A,P)} + \alpha \leqslant \frac{\|f(A)-f(N)\|^2}{d(A,N)}$
  - or $\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leqslant 0$
  - α is margin to make sure the objective function learns when the difference between triplet pairs is small.
  - Margin will help us to train the parameters that push the anchor positive pair and anchar negative pair further away from each other
  - Official Loss Function
    - Given 3 images A, P, N
    - $L(A, P, N) = max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$
    - $J = \sum_{i=1}^{m} L(A^{(i)}, P^{(i)}, N^{(i)})$
  - Training set 10K pics of 1K person: need multiple pics of same person
  - If choose A, P, N randomly, $d(A, P) + \alpha \leqslant d(A, N)$ is easily satisfied, we want to choose triplet that are hard to train on
- Face Verification and Binary Classification (is the 2 image the same person or not)
  - $\hat{y} = \sigma(\sum_{k=1}^{128} W_i|f(x^{(i)})_k - f(x^{(j)})_i| + b)$
  - where $f(x^{(i)})$ is the encoding for each images we feed their difference to logistic regression output
  - Instead of difference we can also use χ^2

Reference

- Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - [Deep Residual Learning for Image Recognition (2015)] (https://arxiv.org/abs/1512.03385)
- Francois Chollet's github repository: https://github.com/fchollet/deep-learning-models/blob/master/resnet50.py