Class

- Introduction
- Practical Approaches to Deep Learning
  - Encoding
  - Day / Night Classification
  - Face Verification
  - Neural Style Transfer
  - Trigger Word Detection
- Full Cycle Deep Learning
  - Steps in a Project
  - Getting Data
  - Deploying Model
  - Model Maintainence
- Adversarial Examples and GANs
  - Attacking Networks with Adversarial Networks
  - Defenses Against Adversarial Examples
  - GANs
  - GAN Training
  - GAN Examples
- AI for Healthcare
  - Case Study
- Interpretability of NNs
  - Interpreting the Outputs
    - Saliency Maps
    - Occlusion Sensitivity
    - Class Activation Maps
  - Visualizing NNs from the Inside
    - Gradient Ascent
    - Dataset Search
    - Deconvolution
- Midterm Review
- Prediction Evaluation
  - Confusion Matrices
- Reading Papers and Career Advice
  - Tips for Absorbing Information from Research
  - Career Advice

Coursera Modules

- C1 - Neural Networks and Deep Learning
  - Module 1 - Intro to Deep Learning
    - Supervised Learning with Neural Networks
    - Recent Developments in Deep Learning
  - Module 2 - Neural Networks Basics
    - Binary Classification
    - Logistic Regression
    - Gradient Descent + Derivatives
    - Computation Graph
    - Applying Gradient Descent to Logistic Regression
    - Vectorized Python

Class

Introduction

- ANI - specific systems. AGI - building a generalized intelligence
- Part of a corpus of other AI tools - deep learning, probabilistic graphical models, planning, search, knowledge graphs, game theory. While most have made steady linear progress, deep learning sees a more exponential growth curve.

Practical Approaches to Deep Learning

- Taking the cat logistic classifier, if we wanted to classify multiple animals
  - Could have 3 output neurons instead of 1 - the neurons would be independent of each other. Then we would also need to change our input, now need labeled data that isn't just cat / non-cat but labeled with the outputs we want from the new model.
  - One-hot encoding scheme - each index is responsible for an animal $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} cat \\ dog \\ giraffe \end{bmatrix}$. If an image that has both cats and dogs, could have multiple indices turned on - multi hot encoding
  - Have to consider is our architecture compatible with multi-hot? The neurons are independent, each would determine if cat / not, dog / not, etc. So this is a good architecture for multi-hot since each neuron can focus on one of the categories.

Encoding

- The first neuron layer will be sensitive to simple patterns, the next layer will be sensitive to more complex features like eyes or ears. The second layers receives information that is more complex than the pixels received by the first layer. Third layer, etc represents even more complex features
- This is called encoding - each layer encodes different levels of features.

Day / Night Classification

- Given image, label day (0) or night (1), logistic loss $L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$
- Data - could have labeled pictures of day and night, balance data set. 10k images a good start, but wouldn't wait to gather this many to start working on the problem. Could go to pixel bay and just search for day and night images, might have mistakes but with a big enough data set would be good enough
    - Indoor pictures in general are going to be difficult. Also edge cases at dawn dusk, etc and 10k won't be enough to resolve the edge cases
- Input - do we care about real time, then we have to consider speed. Even if not training time might matter. Low resolution images are probably fine for this problem since we aren't looking at granularity and will make training faster.
    - What is low resolution? Consider orders of magnitude. For face recognition 400 x 400 is pretty classic. In this scenario 64 x 64 is enough, but RGB important here. If you aren't sure, just see if a human can distinguish with different resolutions with accuracy - this is a good guide to what a model might need.
- Output - 0 or 1. If we wanted more specificity for time of day could use one-hot, softmax instead of sigmoid.
- Architecture - shallow network probably works well
- Loss function - binary cross entropy, since this is standard for binary classification tasks.

Face Verification

- School wants to use **face verification** for validating student IDs when students swipe their cards. Do they match their ID pictures?
- Data - University has labeled images of every person with a card. But this is not enough, need more generalizability in the model.
- Input - give camera picture and stored picture 0 or 1 if same person or not. Resolution ~ 400 x 400
- Output - 0 it is the same person, 1 different
- Architecture - say the simplest idea could be compute distance pixel by pixel. Issues with brightness, background colors, faces aren't centered the same way, face could change with aging hair etc.
    - Solution - use encoding. Run the id image and the camera image through the network. The vectors in the network should have much more information than the pixels on their own. Distance of the vectors should be more meaningful than pixel distance
    - Encoder network learns a lower dimensional representation (the encoding) to focus on non-noisy signals
- How do we train this kind of network? Triplets (Anchor, Positive, Negative)
    - What we want is pictures of the same person to have similar encoding and pics of different people should have different encodings. So we generate triplets - an anchor picture, the original picture. A

positive, the same person. A negative, a different person. Now we want to min distance for anchor - positive and max distance anchor - negative.
    - Triplet loss is a loss function where an (anchor) input is compared to a positive input and a negative input. The distance from the anchor input to the positive input is minimized, whereas the distance from the anchor input to the negative input is maximized.
    - Minimize loss function $L = ||Enc(A) - Enc(P)||_2^2 - ||Enc(A) - Enc(N)||_2^2$
- Now we modify the problem - no swipes, the camera just identifies you from the camera. Could use KNN from the DB of faces and compare vectors, could require more than one match to make more robust.
- Another tweak - we want face clustering, all photos of a single person together. Apply K-means clutering to vectors and see given a new picture which group does it fall into, each group is a person.

Neural Style Transfer

- Taking a picture, make it beautiful
- Data - any data
- Input is a content image and style image - want the ouput to be a generated image that is the content image in the style of the style image
- We use a pretrained network on ImageNet - already extracts import information from images.
- Architecture - content is probably something you can get from lower level encoding of a good network. We use a pre-trained model because it extracts important information from images. Extract a content vector C. Feed in the style image, and find a deeper encoding to find a Gram matrix - style S. For image generation - feed in random image, pull out content CG and style SG, compute loss. After many iterations should get image with altered style. Loss:
  $L = ||Style_S - Style_G||_2^2 + ||Content_C - Content_G||_2^2$
- The parameters are not being tuned - the network is fixed. It's the pixels that are tweaked to minimize the loss functions - $\frac{dL}{dx}$. We leverage the knowledge of the pretrained model to extract the content of content image and style of the style image.

Trigger Word Detection

- Given 10 sec audio speech, detect word activate.
- Data - variety of settings/people saying activate, and audio of saying other things. Need a wide distribution of accents, silence, noise
- Input - audio clips where the segment that is activate is specified. Can consider resolution - look at papers / experts for sample rate
- Output is 0, 1? Want 0 or 1 at the spot of the word, not for the whole audio clip. This makes for much more efficient training. Another problem is huge imbalance between positives and negatives. Sequential sigmoid last activation
- Architecture - RNN
- Critical piece is data collection and labelling process. Instead of manual collection and labeling, create DB of positive words, negative words, background noise. BN is freely available online, no problem. Record in the world, just get word by word recordings - just positive word and just negative words. No need to label then, can just write script to insert positive and negative words into background noise, script can automatically label the data. By recombining in different ways, get millions of data points. The way you collect data is critical.

Full Cycle Deep Learning

Steps in a Project

- 1) Select a project. Example: Using Facial Recognition to Unlock Doors
- Get data, design model, train and evaluate the model and iterate, ship and deploy. Finally maintain the system

Getting Data

- How many days will you use to collect data? Important to get some data, but don't wait to long to dive into the problem. Only need a small amount to get started, see how far we get, then collect more data dependent on performance. 1-2 could easily suffice for an initial dataset.
- Get data quickly and train a quick model. Doesn't need to be the most up to date, complicated network. Find something OSS on Github, etc. Look at the results to determine how far you are from the objective. Iterate over this process, slowly improving the dataset and the model.
- Note - this is reasonable for jumping into a new domain. If you already have the domain knowledge, you might already know that you need a minimum number of examples, or a certain model architecture.

Deploying Model

- Edge device - model is running on the physical device. Cloud device - streams the data to the cloud for processing.
- For door problem, cannot have 30 f/s through NN or 24 hour streaming video to the cloud. Instead might feed an image to an activity detector as preprocessing. Most of the time, nothing is changing outside of the front door, no need to run the inference engine during this time.
- Only when some activity is detected do you feed it to the NN for classification.
- How should we deploy an activity detector?
    - Could write a program to take in a picture from 10s ago and current picture, sum up differences in pixels, and trigger for some threshold.
    - Alternatively, train a small NN to predict human or not.
    - Option 1 could lead to higher errors, option 2 could pass fewer images to the higher compute NN. Option 1 could be written quite quickly given its simplicity, option 2 has more to tune.
    - Option 1 can be a good choice for the quick and dirty model. Option 2 might be a backup if we see that option 1 is a problem. Also has very few parameters to tune. Consistent with the theme to do something quick and dirty first to see if it works, then iterate upwards when it does not.
    - The other thing to consider is that the data changes in ML systems. Weather changes with seasons, people dress differently, people look different in other regions, hardware used might change, diversity of population, etc. A network trained to recognize data at a point in time and location will not be robust

Model Maintainence

- Web search - training a system of search ranking, say there is a new figure, celebrity or language changes. The ranking algorithm no longer is relevant because the world has changed.
- Speech recognition - trained on adult voices, but younger people were more likely to use speech

recognition and the younger voices had different performance. Speech recognition attempted to be used in noisier environments, cars, initial dataset did not cover these situations.
- Defect inspection - say in manufacturing, detecting scratches on smart phones. If the lighting changes in the factory, algorithm may cease working.
- When the data changes, have to get new data, retrain the model, redeploy. Using a simple threshold / preprocessing step, it is easy to update or retune this section. Using a neural network means more retraining before you even get to the heart of the model.
- You also want to get ahead on maintainence - deploy updates before users complain. Between cloud and edge approaches to deployment, cloud is easier to maintain. When we deploy an ML system, often set up monitors / dashboard where we can monitor key metrics over time with threshold bands, notified once we exceed those thresholds.
- Often can just sample some portion of the data from devices as a statistical sample to monitor things. Cloud vs edge doesn't affect the day 1 accuracy of the system, but if the system is designed to continue to accumulate data and improve will help keep the system working. Builds a defensive moat to ward off competitors.
- QA: In ML, our problems are statistical more than software bugs. The testing is not binary, right or wrong, but some % accuracy on the test set. Always important to ensure you meet some accuracy criteria.

Adversarial Examples and GANs

Attacking Networks with Adversarial Networks

- Given NN pretrained on ImageNet, we want to find an input image that will be classified as an iguana.
    - Rephrasing what we want: y_hat = y_iguana, some vector for iguana prediction. Loss $L(\hat{y}, y) = \frac{1}{2}\|\hat{y}(W, b, x) - y_{\text{iguana}}\|_2^2$. Gradient wrt X, update X, our data, until we have an iguana classified image.
    - Define the loss function, then optimize the input image. Use an image to run network forward, compute loss, backprop and update our weights. After many iterations the image will be predicted to be iguana.
    - But will the forged image x look like an iguana? We have optimized the pixels that lead the model to predict iguana, this is not the same as something human recognizable as iguana. If our image is 32 x 32 x 3, and each has 256 potential values, space of possible input images is huge $256^{32\times32\times3}$, while space of real images is a small subset. The space of images classified as iguanas will overlap with recognizable pictures of iguanas, but most of that space is random noise to humans.
- Now say we want to find an input image of a cat but classified as iguana in our pretrained NN.
    - Rephrasing - similar to last problem but want x to look like a cat. x should be close to x_{cat}, an image of an actual cat that is classified as cat.
    - Add term to loss function - a regularization that minimizes the distance $L(\hat{y}, y) = \frac{1}{2}\|\hat{y}(W, b, x) - y_{\text{iguana}}\|_2^2 + \lambda||x - x_{cat}||_2^2$
    - Then optimizing over new loss function, we get an image that still appears to be a cat - pixels will still be close to the cat image.
    - There is another space of images that look real to humans, a superset of real images. It overlaps with the space of images classified as iguanas, including outside of the space of real images.

Defenses Against Adversarial Examples

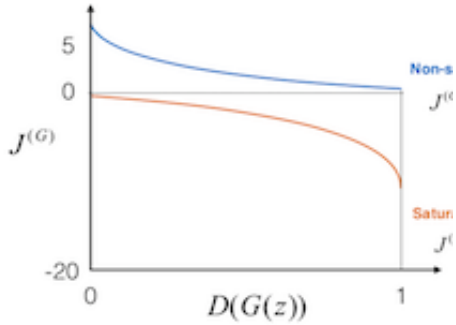
- White box - attacker has access to the model parameters, layers, architecture, these are the examples involving iguana.
- Black box - attacker does not have knowledge of the network, a more typical real world example.
    - Could try to recreate the model and generate adversarial examples - tranferability of adversarial example
    - Ping the model to learn about it before producing adversarial example. Slow tweaks to cat image to see how robust / what it focuses on - a sort of approximation of the gradient.
- Create a SafetyNet - a model whose only goal is to detect perturbation. Downsides, slowly inference time, optimization problem is more complicated
- Train on correctly labeled adversarial examples - label the adversarial examples as cat in training
- Adversarial training - loss function has some reguarization for an adversarial x: $L_{new} = L(W, b, x, y) + \lambda L\left(W, b, x_{adv}, y\right)$. However this will be slow to train, if we bring on adversarial examples for every training example - we basically have an extra loop to train over.
- Fast Gradient Sign Method: Need to understand why NN are vulnerable to adversarial examples
    - Goal is to design a method to generate adversarial examples quickly
    - Take logistic regression $x = \begin{bmatrix} x_1 \\ \vdots \\ x_6 \end{bmatrix} \rightarrow \sigma \rightarrow \hat{y} = \sigma(Wx + b)$
    - We trained the network and got $w = \{1, 3, -1, 2, 2, 3\}, b = 0$. Can we slightly modify x while drastically modifying $\hat{y}$?
    - Propose generate $x^* = x + \varepsilon w^T$. Epsilon since we want small perturbation, $\frac{\partial \hat{y}}{\partial x} \propto w^T$. Take $x = [1, -1, 2, 0, 3, -2]$
    - Get $x^* = \begin{bmatrix} 1 \\ -1 \\ 2 \\ 0 \\ 3 \\ -2 \end{bmatrix} + \begin{bmatrix} 0.2 \\ 0.6 \\ -0.2 \\ 0.4 \\ 0.4 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 1.2 \\ -0.4 \\ 1.8 \\ 0.4 \\ 3.4 \\ -1.4 \end{bmatrix}$ Then $\hat{y}^* = \sigma(wx^* + b) = \sigma(np.dot(w, w) + wx + b) = \sigma(1.6) = 0.83$
    - The impact of the perturbation increases with the dimensionality of the problem. The np.dot term is the sum of weights, as dimensions increase the sum will increase- our output gets pushed further by small changes.
    - Fast gradient sign method: Could also take $x^* = x + \varepsilon sign(w)$. Now we can take this function and easily generate adversarial examples to train on. We just need the sign of the gradient more than an actual approximation of the exact value, allowing this attack to work relatively well in practice.
    - Generally $x = x + \epsilon\, sign(\nabla_x J(w, x, y))$

GANs

- We have sample data of real images, they have a certain distribution.
- We have a sample of generated data, with another distribution. We want to force this distribution close to the true images.

- The idea is to play a game G/D. Feed random code into generator G network. In general this would just produce a random image
- We build a database of real images, and use another network called the Discriminator D, and use this network to train G
- We send real and generated images to D - it is a binary classifier distinguishing real and generated images. In training D, label generated and real images and train as we normally do. The initial image distributions are quite different and it is easy for D to distinguish
- If we backprop all the way through G, G will learn to make the prediction of D wrong. Iterate until G fools D, but D should also be improving as the generated images improve.
- Run GD on batches of real + fake data.

GAN Training

- Cost of discriminator is binary cross entropy. CE term 1 says D should correctly label real data as 1, and term 2 says D should label generated data as 0:
  $J^{(D)} = -\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} \log\left(D\left(x^{(i)}\right)\right) - \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \left(1 - y_{gen}^{(i)}\right) \cdot \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$
- Cost of generator - maximize cost of D. $J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$, but only consider term 2 since term 1 does not depend on G at all. G should try to fool D: by minimizing the opposite of what D is trying to minimize
- Saturating cost for the generator - $J^{(G)} \approx log(1 - x)$, early in the training D is much better than G. Then D(G(z)) is close to 0, and the gradients are very low from $log(1 - x)$. We want to modify the cost so that it is non saturating. We can say minimizing $log(1 - x) \iff max(log(x)) \iff min(-log(x))$. These are roughly equivalent, except $-log(x)$ has high gradients early in the training.
- Saturating cost: $J^{(G)} = \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$. Non saturating cost: $J^{(G)} = -\frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$.
- 

- Another method - train D and G a different number of times. If D stops improving, G will also slow in improving. Could then update D k times for each time we update G. Inner loop in training for D.
- Cannot rely on loss to tell us the quality of output - since it depends on how good the discriminator and generator are. Need to look at the actual output for quality metrics.
- See GanHacks on GitHub for many more ways to train GANs. GANs are hard to train and take many optimization tweaks.

GAN Examples

- Cycle GAN - Convert horses to zebras on images and vice versa
- Data? There are no paired images, so we collect horses and zebra images separately
- Architecture - given horse H -> Generator1(H2Z) -> G1(H) -> Discriminator1. We still haven't introduced the contraint that the generated zebra must be the same as the horse. Take G1(H) -> Generator2(Z2H) ->

G2(G1(H)) -> Discriminator2. Minimize distance between H and G2(G1(H))
- Then for zebra to horse Z -> Generator2 -> G2(Z) -> Discriminator2 and G2(Z) -> Generator1 -> G1(G2(Z))
- 5 loss functions - 2 for G's, 2 for D's, and a cycle loss function used as a regularizer

AI for Healthcare

- Often have imbalanced datasets - some pathologies must more prevalent than others. As we increase the prevalence of disease in data, the algorithm performs better.
- We can weight the loss function - say for logisitic regression, can add larger weight to disease class.
- Noisy label challenge - two radiologists would have different answers for an x-ray. One might be right in reality or it may be a matter of interpretation. This is actually ok for DL tasks - increasing the # of training examples can drastically increase accuracy even learning from the noisy examples - smooths out the noise
- Correcting noise via self training - noisy label -> train classifier -> predict for every example in the training set -> use the prediction of classifier to train a new classifier. Can apply the classifier to many datasets to create the new classifier.
- On image tasks, pretty common to pretrain on ImageNet then modify by training on medical images.

Case Study

- Data augmentation often useful but not in every situation. In MNIST, if you rotate a 6, it becomes a 9, but the label is still a 6. Will confuse the model. Similar with character recognition.
- For transfer learning, may want to freeze the early layers since their weights should directly relate to your task - its the later higher level layers that we want to retune to the specific task as well as modify the output to our purposes.
- Cell segmentation - Determine which parts of a microscope image corresponds to which individual cells. Network outputs a mask for the image that has the fill area for the cells.
- When we use training images not from the dev/test distribution, should still have more images from the dev/test distribution in training than in the dev/test sets.
- If mask algorithm doesn't define the boundaries between cells well, modify the dataset in order to label the boundaries between cells. On top of that, change the loss function to give more weight to boundaries or penalize false positives.
- Now cells labeled as cancerous and benign. Classifier high accuracy, but how can you explain the network's predictions?
- Given an image classified as 1 (cancer present), how can you figure out based on which cell(s) the model predicted 1? Gradient of output w.r.t. input X
- Network could achieve performance beyond a given doctor if a team of doctors can achieve higher accuracy, say 99% for the team and 97% for the individual.

Interpretability of NNs

Interpreting the Outputs

Saliency Maps

- Have a model, but users do not understand the decision process of the network. Say a CNN with softmax that outputs the animal. How do we relate the output to the input

- Look at the gradients - gradient of score of dog with respect to x - derivative will be RGB matrix shape (shape of X) and each entry will indicate if changing the given pixel will change the output or not.
- Look over all pixels - have a saliency map - we can see the silhouette of the dog
- Are you taking the derivative of score of dog pre or post softmax? Always want to take the pre softmax value - this value only depends on the feature of dogs. Post softmax depends on the scores of other animals. You might then find the pixels that minimize the score of other animals rather than maximize the score of dog, since this is equally important in discriminating among animals post softmax.

Occlusion Sensitivity

- Pass in the dog image and occlude a square in the top left .See how the probability changes after the softmax.
- Track it in a probability map - square of image where we track whether confidence increases or decreases when we move the occlusion across the image.
- End with a probability map that shows me where the dog is in the image.
- When there are multiple areas of low confidence - we see the network using other features of the image to determine the true class - like writing on the car helps it figure out car wheel - probability declines when writing is occluded.
- For dog with humans - hiding the human face, the confidence of dog increases. Softmax declining probability of human face helps it with its dog confidence.

Class Activation Maps

- Say we want a realtime visualization of the model's decision process. Where are we losing the decision process data? Where we flatten the CNN to FC layers
- Convolutional layers shrink the height and width but add depth - the top left of the image stays in the top left of the volume.
- But once we flatten, we lose all localized information.
- Instead then we can convert the last layer -> global average pooling -> FC -> softmax
- Global average pooling - takes one slice of height width and averages to one number - keeps depth but reduces the H and W to a single number. Then we pass this to an FC layer -> softmax, but each unit in FC corresponds to a single depth layer in the volume. If the 3rd depth layer has a strong weight into softmax, can map back to that depth
- Get a feature map for each slice in the volume. Weighted across the whole volume by the weights of the last FC layer, get the general class activation map for dog. Some feature maps may activate to human, some to wheel, etc, but the weights ensure the dog feature map carries the most weight for the global map.
- This is a very fast method - can perform in real time. But note that we need to train the last FC layer since we replaced the layer in our original model. This is our new model for prediction as well - we discarded our old model, we might get slightly different probabilities but should be similar if well trained.
- Starting from scratch, you do not need to change the last layer - just start using the GAP / FC / softmax

Visualizing NNs from the Inside

Gradient Ascent

- Try to explain what the model thinks a dog is. Without data, if you asked the model to generate a dog, what would it produce

- If we try to use the backprop to produce the most dog image, the network will minimize the probability of not dog since we are using the softmax. Will end up with an image that is most not dog
- Instead use the score pre-softmax for dog, plus a regularizer to keep the pixels small and constrained, $L = s_{dog}(x) - \lambda \|x\|_2^2$, essentially constraining x to look natural. Keep the weights fixed and use gradient ascent on the input image to maximize this loss. Then the gradient ascent $x = x + \alpha \frac{\partial L}{\partial x}$.
- Repeat the process: forward propogate image x, compute L, backprop dL/dx, update x's pixels with grad ascent.
- With better regularization, you can get closer in colors etc. Class model visualization
- See it produces image of geese for goose. Confidence of model goes up for more geese in an image, but still should be able to label one goose.
- What makes a good regularizer? If you have a skewed data distribution to the right, and model is focused around left. An underfitted regularizer will not impact the quality of the image, while an overfitted regularizer will bind it too closely to the given data, but general idea is to push the distribution towards the data.
- This method can be applied to any activation in the network in order to interpret what a neuron is detecting. Instead of loss defined by class score $L = S_{dog}(x) - R(x)$, sub in a specific activation from somewhere in the network: $L = a_j^{[l]}(x) - R(x)$

Dataset Search

- Given a filter, what examples in the dataset lead to a strongly activated feature map.
- Take one activatation - store in memory the top 5 images that activate this feature map the most. Can do it for any feature map, see different patterns - one detects shirts, one detects edges.
- The top 5 images are cropped - trying to map the activation in a later layer back to the original image. If the activation is in the lower right of the map, then we take the lower right of the image. The deeper we go, the more of the image that maps to an activation.
- If it is totally random, maybe the feature did not learn anything useful.

Deconvolution

- In a generator - to get 64 x 64 x 3 image would need a FC layer of size 64 x 64 x 3 units or use a deconvolution
- Encode information in reduced volume (H and W) then expand back out to original size
- Keep the max activation of a feature map, then reverse the network. Unpool, relu, deconvolution through the layers to get to the reconstruction in the image of what activated that portion of the feature map
- Unpool: But maxpool is not invertible since it only stores the max from the region it pools. But we can cache switches - they tell us the positions in the matrix that the max value came from. A matrix of 0's and 1's indicating we took a maxpool from this position or not. We don't get back the full matrix, so we set the rest of the values to 0. We care about the more discriminative feature so we can stick with just the max values.
- Pass switches from forward prop to unpools to make this work
- Deconv:

- $\begin{bmatrix} 0 \\ 0 \\ x_1 \\ \vdots \\ x_8 \\ 0 \\ 0 \end{bmatrix}$ $\rightarrow$ 1D conv with size 4, stride 2, pad 4 -> y.
  - We have $y_1 \dots y_5$ using the formula from the Conv section.
  - Equations for conv: $y_1 = 0w_1 + 0w_2 + x_1w_3 + x_2w_4$, $y_2 = x_1w_1 + x_2w_2 + x_3w_3 + x_4w_4$... $y_5 = x_7w_1 + x_8w_2 + 0w_3 + 0w_4$. A convolution in 1D is nothing more that a matrix vector multiplication.
  - $y = Wx$ for y (5 x 1), x (12 x 1) so W is (5 x 12).
  - So $\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 & 0 & \dots & 0 & \\ 0 & 0 & w_1 & w_2 & w_3 & w_4 & 0 & \dots \\ \dots & & & & & & & \\ \dots & & & & & & & \\ \dots & & & & & & & \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ x_1 \\ \vdots \\ x_8 \\ 0 \\ 0 \end{bmatrix}$
  - So we need $x = W^{-1}y$ - assume that W is invertible and also orthogonal: $W^TW = WW^T = I$
  - Then this is easier $x = W^Ty = \begin{bmatrix} w_1 & 0 & \dots \\ w_2 & 0 & \dots \\ w_3 & w_1 & \dots \\ w_4 & w_4 & \dots \\ 0 & w_3 & \dots \\ 0 & w_4 & \dots \\ 0 & 0 & \dots \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$
  - But we lose something in the transformation - not really dealing with our filter anymore.
  - So instead we pad our y vector with 0's - 0 between every y value and 0's above and below.
  - Take W and have sliding filter w_4, w_3, w_2, w_1. X is 12 x 1, y is 3 0's the 5 y's the 3 0's plus padding 0's 15 x 1. So W' here is (12 x 15).
- In summary: Soft pixel transformation on y - this is padding it with 0's
- Flip the filter W
- Divide the stride by 2: zeros are inserted *between* input units, which makes the kernel move around at a slower pace than with unit strides. Since we originally mapped from larger to smaller, we need this spacing to do the reverse. Taking fractional stride.
- ReLU does not really change.

Midterm Review

- Sigmoid - outside of activation region has very small gradient - vanishing gradient, and non centered at origin. Tanh also has shrinking gradient away from center.
- ReLU - problem that gradient is 0 when negative z. But not really a vanishing gradient effect since linear for positive z.
- How many bits for image mapping - need to rep cat vs not cat for all images 1 bit, 256 ^(64 x 64 x 3). If it

took 4 bits to represent each class have 256 ^(64 x 64 x 3 x 4)

- Part b - W1 taking a 64 x 64 x 3 space to a 100 space, so W1 is 64 x 64 x 3 x 100. W2 takes us from 100 dimensional space to 1 value, so 100 x 1. Each has 64 bits per weight, so 64 x (64 x 64 x 3 x 100 +100 x 1)
- Number of parameters with L hidden layers, 1 input and ouput layer
  $\sum_{i=0}^{L}\left(n^{|i+1|} \times n^{[l]}+n^{[l+1]}\right)=\sum_{i=0}^{L}\left(n^{[l+1]} \times\left(n^{[l]}+1\right)\right)$
- Weight initialization - these are all variances on the slide, not SD as the second number. Want activations to be around the same size throughout the network. Think of tanh, don't want to be out on the edges.
- He initializtion - half of those inputs are zero in Relu so we multiply by 2.
- Assumptions are only valid at the beginning of training. Linear activation and tanh are close for small inputs. 2 / (nprev + ncurrent) allows same initialization for forward and backward since the prev layer is relative there. Initialization on backward pass - we aren't re-initializing, just thinking about how we want weights to change through forward and backward pass.
- Noisy updates on MBGD add jitter, more likely to avoid getting stuck in saddle points.
- ConvNets - # parameters, filter f, each side of a filter is a parameter (2d or 3d), f x f, f x f x nc +1 bias per filter.
- Output is 2D even if filter is 3d. Then the number of filters corresponds to the channels in the ouput activation.
- Padding can be in neither valid nor same categories - any p that isnt 0 or (f-1)/2
- Filter size is constant in the network, so it may not capture an appropriate feature if image is huge or small - not scale invariant. Not rotation invariant, applying the same filter to a rotated image will have different features. It is translation invariant - cat in top left corner to top right, the same filter is applied to local regions.
- Pooling is down sampling - reduce size of activations while retaining important information. Pooling also has a filter and stride.
- Number of weights - 9 x 9 x 3 - for every filter we have 9 x 9 and input image had 3 channels. We have 32 such filters so total weights 9 x 9 x 3 x 32. Bias is just 1 per filter so 1 x 32
- Pooling has no weights and biases. FC number of weights number units x dims of volume flattened.
- Batch Norm has a slight regularization effect but shouldnt be used for regularization. At test norm, you use the exponentially weighted average computed during training as your normalizing values. Mitigates covariate shift - later layers depends on early layers, if they shift then so will the later layers - want to avoid shifts and keep things normalized.
- Adversarial - using backprop to update the image instead of the network.
- D(G(z)) = 1 when discriminator output says real, 0 when D says fake. 0 towards beginning more when generator is bad.
-

	Bias	Variance
Regularizing the weights	↑	↓
Increasing the size of the layers (more hiden units per layer)	↓	↑
Using dropout to train a deep neural network	↑	↓
Getting more training data (from the same distribution as before)	–	↓

Prediction Evaluation

Confusion Matrices

-

	Truth Pos	Truth Neg
Predicted Pos	TP	FP
Predicted Neg	FN	TN

- Accuracy: sum along the upper-left to bottom right diagonal over the total N. $\frac{TP+TN}{N}$. Good for balanced classes between positive and negative.
- Recall (True Positive Rate, Sensitivity): how many of the positive examples did we catch? $\frac{TP}{TP+FN}$
- Precision: how many of positive predictions are actually positive: $\frac{TP}{TP+FP}$
- F1 Score: how do we trade off between precision and recall. Looks at TP, FP, FN boxes. $\frac{2}{1/P+1/R} = \frac{TP}{TP+\frac{FP+FN}{2}}$

Reading Papers and Career Advice

Tips for Absorbing Information from Research

- Search web to compile a list of papers / blog posts / etc. Skip around reading the papers, only achieving a partial understanding. Only reading the really interesting or cutting edge papers more fully. Get to know what is already outdated.
- After reading 5-20 papers, can get a good sense to implement something reasonable. 50-100 papers in a narrow field is near mastery of what is out there.
- Within one paper, make multiple passes. First looking at the title, abstract, figures. Second, read the intro, conclusion, figure captions, skim the rest, skip related work. Third, read the paper, skimming the math.
- Questions to try to answer - what did the authors try to accomplish? What are the key elements of approach? What can you use yourself? What other references to you want to follow?
- To understand the math, best to try to rederive the math from scratch. Same for code, could download and run open source implementations, but re-implementing from scratch helps you understand.
- The batch - Andrew's group publishes a newsletter

Career Advice

- What do recruiters look for? Skills, evidence of meaningful work. Show that you can do things, learn new skills.
- Strongest candidates have breadth of areas in ML they know about and depth in at least one area. DL reserved for specific problems, need to be able to distinguish when to use different models and skills.
- Depth comes from meaningful project work, research, internships, etc. Better to have few deep projects than may shallow pieces of work.
- Coursework should be a way to build foundational knowledge towards reading research and using more advanced ideas.

Coursera Modules

C1 - Neural Networks and Deep Learning

Module 1 - Intro to Deep Learning

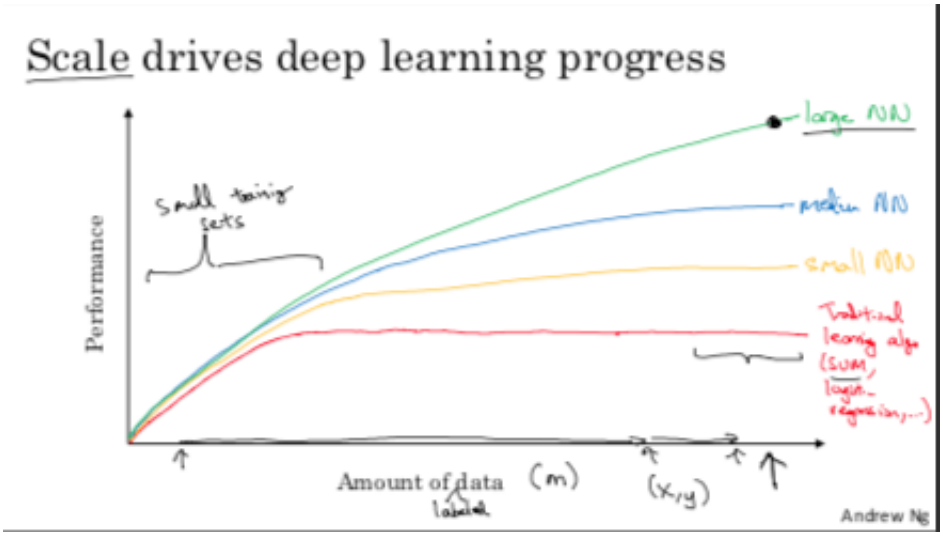
- Housing price prediction - say you have 6 houses with data on size and price. This is a classic supervised learning problem, could fit a regression, might also have a kink at 0 to prohibit negative prices. One could think of this model as a simple neural network.
- Size x -> O -> price y. The intermediate step is a neuron - a function that transforms the input to an output
- We often see the 0 kinked function in NN - ReLU function (Rectified Linear Unit). Linear with pos slope above a certain value, zero below.
- Given additional features about a house: could take in size and # bedrooms as inputs to family size node. Zip code is input into walkability node. Wealth and zip code inputs to school quality node. Finally family size, walkability, school quality are inputs to a node predicting price. Input and output is x(size, # bedrooms, zip code, wealth) and y(price), while the things in the middle are figured out by the network itself
- The hidden units take in all of the inputs in x - this means they are densely connected layers.

Supervised Learning with Neural Networks

- Have input x and want to learn a function mapping to an output y
- Different NN types may be useful for different applications - real estate, advertising might use standard NN. Image recognition uses CNNs. Sequence data like speech recognition typically uses RNNs
- Structured data - databases of data. Unstructured data - raw audio, images, text. While often hear about NNs on unstructured, much of the economic value of NNs has been realized from structured data.

Recent Developments in Deep Learning

- Plot of amount of data vs. performance. While most learning methods plateau and cease to improve with a greater amount of data, NN's continue to improve, and the deeper the network the more they continue to improve.
- Amount of labeled data has increased in recent times. We denote size of training set by m.
- In regime of small training sets, the relative ordering of the learning methods is not well defined - not obvious which method is best. Only in large m do we see consistently large NN's dominating.
- Improvements in data size, computation, and algorithmic innovations. Many algorithmic improvements have allowed for faster training - such as switching from a sigmoid to a ReLU. The extremes of the sigmoid have very small derivative - gradient is small so learning is slow. ReLU has slope of 1 for all points above the kink - makes gradient descent much faster.
- Training a network is iterative - idea -> code -> experiment -> repeat. The learning cycle for the programmer becomes much tighter, allowing more creative and interactive modeling.

- 

Module 2 - Neural Networks Basics

Binary Classification

- Input -> 1 or 0. Binary values could stand for anything (cat or non cat). y is the output label
- Computer images stored in 3 RGB matrices. If image 64 x 64 pixels, have 3 x 64 x 64 pixel values. We unroll these pixel values into a feature vector x. $x = \begin{bmatrix} 255 \\ 231 \\ \dots \end{bmatrix}$. 64 x 64 x 3 = 12288. Say n = $n_X = 12288$
- (x, y) is a single training example. $x \in \mathbb{R}^{n_x},\ y \in \{0, 1\}$
- m training examples: $\{(x^{(1)}y^{(1)}), \dots, (x^{(m)}y^{(m)})\}$. When we use $m = m_{train}$ vs m_{test} = number of test examples.
- Define matrix $X = [\, x^{(1)} \quad x^{(2)} \quad \dots \quad x^{(m)} \,]$ with dimensions $n_x \times m$. Each $x^{(i)}$ is a column. X.shape = $n_x \times m$
- Define matrix $Y = [\, y^{(1)} \quad y^{(2)} \quad \dots \quad y^{(m)} \,]$ so $Y \in \mathbb{R}^{1 \times m}$. Y.shape = 1 x m

Logistic Regression

- Given x want $\hat{y} = P(y = 1|x)$. We know $x \in \mathbb{R}^{n_x}$, then parameters: $w \in \mathbb{R}^{n_x}$ and $b \in \mathbb{R}$
- Could try output $\hat{y} = w^T x + b$ - this is linear regression, but not great here because it does not force $0 \leq \hat{y} \leq 1$.
- Instead use $\hat{y} = \sigma(w^T x + b)$ - wrapped in the sigmoid function. Let $z = w^T + b$. Then $\sigma(z) = \frac{1}{1+e^{-z}}$.
- If z is large then $\sigma(z) \approx 1$. If z is very small then $\sigma(z) \approx 0$
- We will handle w and b separately instead of folding into a new x vector ($\hat{y} = \sigma(\theta^T x)$, inputing a θ_0 as b)
- Loss Function
  - Given $\left\{\left(x^{(1)}, y^{(1)}\right), \dots, \left(x^{(m)}, y^{(m)}\right)\right\}$ want $\hat{y}^{(i)} \approx y^{(i)}$. Note the superscript i, refers to data associated with the ith training example.
  - Loss to measure how good our estimate is compared to true value of y. We won't use squared error here since it leads to local optima.
  - Loss function: $L(\hat{y}, y) = -(y log(\hat{y}) + (1 - y) log(1 - \hat{y}))$. The loss function is a measure for a single training example.
  - If y = 1, $L(\hat{y}, y) = -log(\hat{y})$. We want $log(\hat{y})$ to be as large as possible, then want $\hat{y}$ to be large -

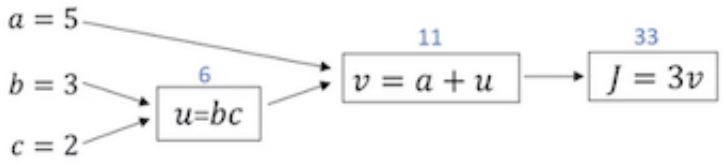
since bounded above by 1, want it to be close to 1.
    - If y = 0, $L(\hat{y}, y) = -log(1-\hat{y}))$ - want $log(1-\hat{y})$ to be large, which means we want $\hat{y}$ as small as possible. Bounded below by 0 so we push it to be as close to 0 as possible.
- Cost Function
    - Loss over all training examples
    - $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(\hat{y}^{(i)}) + (1-y^{(i)})log(1-\hat{y}^{(i)})]$

Gradient Descent + Derivatives

- Want to find w,b that minimize J(w,b). Graphically, we have planar axes w and b, and z-axis = J(w,b) - the cost function is a surface above w, b and want to find the value of w, b at the sink of the surface. J is a convex function, so we should be able to find a global minimum.
- Initialize w and b to some values - almost any initialization is fine, but often use 0. Gradient descent starts at the initial point and takes a step in the direction of steepest descent. Iterate over and over until convergence around optimum is reached.
- Gradient Descent Procedures
    - Repeat: $w := w - \alpha\frac{\partial J(w,b)}{\partial w}$.
        - Note α is the learning rate. Will define $dw := \frac{\partial J(w,b)}{\partial w}$.
        - If w is too large, the slope will be positive and we subtract off a positive number making w smaller. If w is too small, the slope is negative and subtracting off the derivative will make w larger with each step.
    - In same loop, repeat $b := b - \alpha\frac{\partial J(w,b)}{\partial b}$
        - Will define in code $db := \frac{\partial J(w,b)}{\partial b}$.

Computation Graph

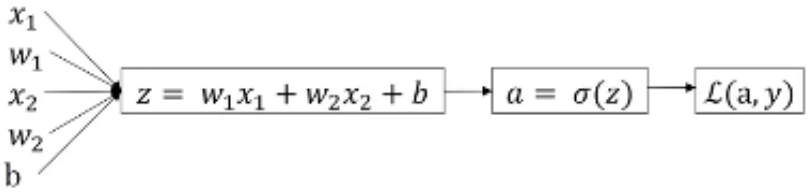
- J(a,b,c) = 3(a + bc)
- 3 distinct steps to compute this function.
    - 1) u =bc 2) v = a + u 3) J = 3v
- Take these three steps and place them in a computation graph:



- For output variable J that we want to optimize, we compute the value of J from a left to right pass. To compute derivatives we will compute from right to left
- Taking derivatives - $\frac{dJ}{dv}$? Since J = 3v, v=11, a small change in v increases J by 3 times as much. $\frac{dJ}{dv} = 3$
- $\frac{dJ}{da}$ - J does not depend directly on a, so we rely on the chain rule. $\frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da}$
    - Note for variables in code, use the notation $dvar := \frac{dFinalOutputVar}{dvar}$

- Continuing $\frac{dJ}{du} = \frac{dJ}{dv}\frac{dv}{du}$ - same as for a since at the same level. For the next level down $\frac{dJ}{db} = \frac{dJ}{du}\frac{du}{db}$ - we have already found $\frac{dJ}{du}$, so we can plug in our result from the prior step - at each step can just use a single chain rule. This is why it is most efficient to compute the derivatives from right to left.

Applying Gradient Descent to Logistic Regression

- For a single example using Loss:
  - $\hat{y} = a = \sigma(z)$
  - 

  - $z = w_1x_1 + w_2x_2 + b \rightarrow \hat{y} = a = \sigma(z) \rightarrow L(a, y)$
  - $da = \frac{\partial L(a,y)}{\partial a} = -y/a + \frac{1-y}{1-a}$, $dz = \frac{\partial L(a,y)}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = a - y$, $\frac{\partial L}{\partial w_1} = x_1 dz$, etc.
- For an entire training set using Cost:
  - Now using $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$
  - $\frac{\partial}{\partial w_1}J(w, b) = \frac{1}{m}\sum^{m}\frac{\partial}{\partial w_1}L(a^{(i)}, y^{(i)})$ - simply use the output from the previous step and average over all training examples.
  - Initialize J = 0, $dw_1 = 0,\ dw_2 = 0,\ db = 0$
  - For i=1 to m:

$$z^{(i)} = \omega^T x^{(i)} + b$$
$$a^{(i)} = \sigma\left(z^{(i)}\right)$$
$$J = \left[y^{(i)} \log a^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right]$$

    - $dz^{(i)} = a^{(i)} - y^{(i)}$
    $dw1+ = x_1^{(i)} dz^{(i)}$
    $dw_2+ = x_2^{(1)} dz^{(i)}$
    $db+ = dz^{(i)}$
  - Finally divide J, dw1, dw2, db by m since we are taking averages. At the end each accumulator variable equal to something like $dw_1 = \frac{\partial J}{\partial w_1}$, etc.
  - Then $w_1 := w_1 - \alpha \times dw_1$, etc. This is a single step of gradient descent - we then need to repeat many times.

Vectorized Python

- Recall $z = w^T x,\ w \in \mathbb{R}^n,\ x \in \mathbb{R}^n$
- `z = np.dot(w, x)+ b` computes $w^T x + b$

```
import numpy as np

# vectorized version
a = np.random.rand(100000)
b = np.random.rand(100000)
c = np.dot(a,b)

# loop version
c = 0
for i in range(100000):
 c += a[i] * b[i]
```

- GPU / CPU both have SIMD, single instance multiple data - vectorized computations allow for parallelization
- Whenever possible, avoid explicit for-loops
- Example: $u = Av$ then $u_i = \sum_j A_{ij} v_j$. Vectorized: `u = np.dot(A,v)`
- Example: vector v, want to exponentiate each value. `u = np.exp(v)`. Other useful element-wise functions: np.log, np.abs, np.maximum
- In logistic regression
  - Need to compute $z^{(1)} = w^T x^{(1)} + b,\ a^{(1)} = \sigma\left(z^{(1)}\right)$ for each data point
  - Construct 1 x m matrix
    $Z = \left[z^{(1)} z^{(1)} \cdots z^{(m)}\right] = \omega^\top X + [bb \ldots b] = \left[w^T x^{(1)} + b \quad \ldots \quad w^T x^{(m)} + b\right]$
  - In python: `Z = np.dot(w.T, x) + b` - here python uses broadcasting, expanding b to a row vector of the constant repeated to the right dimensions.
  - Similarly $A = \left[a^{(1)} \ldots a^{(m)}\right] = \sigma(Z)$
    $Z = W^\top X + b$
    $= np \cdot \text{dot}(W^\top, X) + b$
    $A = \sigma(Z)$
    $dZ = A - Y$
  - $dw = \frac{1}{m} X dZ^T$
    $db = \frac{1}{m} np.\text{sum}(dZ)$
    $w := w - \alpha dW$
    $b := b - \alpha db$
- Gradients
  - Define $dZ = \left[dz^{(1)} \ldots dz^{(m)}\right], Y = \left[y^{(1)} \ldots y^{(m)}\right]$. Then $dZ = A - Y$
  - `db = 1/m * np.sum(dZ)`, `dw = 1/m * X * dZ.T`
- Broadcasting
  - Say we have matrix 3 x 4. Want to sum for each columns and divide to get percents instead of gross.

```
import numpy as np
A = np.array([
  [56., 0., 4.4, 68.],
  [1.2, 104., 52.0, 8.],
  [1.8, 135., 99.0, 0.9]])
cal = A.sum(axis=0)
#broadcasting - dividing 3x4 matrix by 1x4
percentage = 100*A/cal.reshape(1,4)  #note - don't actually need reshape here
A.shape #(3,4)
np.mean(np.array([1,2],[3,4]), axis = 0) #array([2,3])
np.exp(X)
np.sqrt(X)
np.dot(v, w) # inner product or matrix multiplication (or np.matmul)
```

- Python is autoexpanding during broadcasting - generally (m, n) # (1,n) -> (m,n); (m, n) # (m,1) -> (m,n). Can also perform with single row / col vectors and scalars.
- Note calling vectors with np can create a rank 1 array in python - neither column nor row. Specify actual dimensions in calls to get expected behavior, should see 2 square brackets not 1. Debugging tip: throw in `assert(a.shape == (5,1))` to check periodically the data looks as expected

Module 3 - Shallow Neural Networks

- Starting with a single hidden layer NN. Let $a = x^{[0]} = x_1, x_2, x_3$ be the input layer, fully connected to the hidden layer with 4 nodes. Those in turn connect to the output layer, with a single node, whose output is $\hat{y} = a^{[2]}$. Superscript bracket notation - the layer of the NN
- For four unit hidden layer: $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$. 2 layer NN, because we do not count the input layer. Each $a_i^{[1]}$ has an associated $w^{[1]}, b^{[1]}$
- Idea should be like the logistic regression, but simply repeated many times. First perform $z = w^T x + b$, then $a = \sigma(z)$. In the NN with layers, have $z_i^{[1]} = w_i^{[1]T} x + b_i^{[1]}$ and $a_i^{[1]} = \sigma(z_i^{[1]})$ for layer 1, node i. Repeat for each node in the layer - but clearly we are going to vectorize these equations.
- Vectorizing, each w_i^T is a row in the W matrix, so for layer 1 stack the equations for the different nodes: $W^{[1]T} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$. Similarly $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$
- Taking the next layer, can perform the same operations, though the dimensions of the matrices can change depending on the input / output size.

Vectorized Implementation across Training Data

- Each $x^{(j)} \rightarrow a^{[2](i)} = \hat{y}^{(i)}$ - eg example i and layer 2 output. We run these layer equations over each i training examples
  $z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]},\ a^{[1](i)} = \sigma\left(z^{[1](i)}\right),\ z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]},\ a^{[2](i)} = \sigma\left(z^{[2](i)}\right)$

- Each training example is a column in the matrix X - n x m matrix. To vectorize over all examples and noting $X = A^{[0]}$, $Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]},\ A^{[1]} = \sigma\left(Z^{[1]}\right),\ Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]},\ A^{[2]} = \sigma\left(Z^{[2]}\right)$ - same equations just over matrices.
- $Z^{[i]}$ is a matrix with each row is a node in the layer with examples running from 1 to m. $A^{[i]}$ is a matrix with each row a node in the layer with columns ranging 1 - m. For Z and A, columns are different training examples and vertically we have different nodes. For X, columns are different training examples and vertically we have different features.
- $W^{[1]}x^{(1)}$ gives you a column vector, true for each training example. Instead we place each column x in X to get the vectorized version.

Activation Functions

- Activation functions can be different for layers, so may write $g^{[1]}, g^{[2]}$ to specify the layer associated with each activation.
- Sigmoid $a = \frac{1}{1+e^{-z}}$ on [0, 1]. Never use on hidden layers besides output layer. Others are strictly superior otherwise.
    - Derivative $\frac{d}{dz}g(z) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = g(z)(1 - g(z))$. In a NN, this would be equivalent to $a(1-a)$. Can see that if z = 10, $g(z) \approx 1,\ g'(z) \approx 0$
    - Can be interpreted as a probability, so typical output of logistic regression
    - Problems: 1) Output is not centered at 0. Saturation (ie far out of on curve), gradient is near 0 and learning is slow.
- Tanh $a = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ on [-1,1]. Similar shape as sigmoid with different range.
    - Almost strictly superior to sigmoid bc the range allows for a mean 0 - centers the data so learning for the next layer is a bit easier. One exception is the output layer, where you might want the output to be a 0 or 1 for classification. However when z is very large or small, then the slope becomes very small as well, slowing learning in gradient descent.
    - Derivative: $\frac{d}{dz}g(z) = 1 - (\tanh(z))^2$. In NN a = g(z) and $g'(z) = 1 - a^2$
- ReLU - $a = max(0, z)$ on $[0, \infty]$.
    - Discontinuity at 0, but essentially not a problem, can just hard code or pretend derivative is 1 or 0. When z is negative, derivative is 0, but could modify to maintain a slight positive slope for z negative - Leaky ReLU. Advantage to either is that for the majority of z's, the slope is very different from 0 (z > 0), allowing faster learning.
    - Derivative: $g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$. Can in practice ignore 0, since change z is exactly 0 is near 0.
    - Cheap to calculate and empirically converges faster. Major problems are gradient is 0 for z < 0 and output not centered at 0.
- Leaky ReLU - $a = max(0.01z, z)$ on $[-\infty, \infty]$. Can modify the z coefficient or make it part of the learning function.
    - Derivative: $g'(z) = \begin{cases} 0.01 & \text{if} \quad z < 0 \\ 1 & \text{if} \quad z > 0 \end{cases}$
- If you aren't sure which will work best, no problem in trying different ones and testing against a validation set.
- Why do we need a non-linear activation function? Without one, $\hat{y}$ is simply a linear function of X, can just plug in from dependent equations and refactor to see this. Could potentially use a linear function at the output layer, but otherwise definitely want something non-linear.

Gradient Descent

- Parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$
- Cost function $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} L(\hat{y}, y)$
- Repeat: Compute predictions $\hat{y}^{(i)}, i = (\dots, m)$, calculate derivatives of J wrt W and b for each layer, update each parameter.
- Forward propogation - the formulas derived above for Z, A for each layer.
- Back propogation:
  - Summary: single example equations:
    - $$dz^{[2]} = a^{[2]} - y$$
      $$dW^{[2]} = dz^{[2]} a^{[1]^T}$$
      $$db^{[2]} = dz^{[2]}$$
      $$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}\left(z^{[1]}\right)$$
      $$dW^{[1]} = dz^{[1]} x^T$$
      $$db^{[1]} = dz^{[1]}$$
  - Summary: vectorized equations:
    - $dZ^{[2]} = A^{[2]} - Y$
    - $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$ - 1/m comes from cost function J, now that we are averaging the losses
    - $db^{[2]} = \frac{1}{m} np.sum\left(dZ^{[2]}, \text{ axis } = 1, \text{ keepdims } = \text{ True}\right)$
    - $dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}\left(Z^{[1]}\right)$ - Note $\times$ is an element-wise product.
    - $$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
      $$db^{[1]} = \frac{1}{m} np.\text{sum}\left(dZ^{[1]}, \text{ axis } = 1, \text{ keepdims } = \text{ True}\right)$$
- Basic idea here: each portion is the loss function derivative wrt some parameter. For $L = -ylog(a) - (1-y)log(1-a)$ we get $da = \frac{d}{da} L = -\frac{y}{a} + \frac{1-y}{1-a}$. Then $a = \sigma(z)$ and $\frac{dL}{dz} = dz = \frac{dL}{da}\frac{da}{dz} = da\frac{dg(z)}{dz} = da \times g'(z)$

Random Initialization

- Initializing the weights of the parameters - cannot simply initialize to 0
- Say a network with n=2, 2 hidden nodes and 1 output layer
- We can initialize the bias to 0's without a problem
- If $W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ then $a_1^{[1]} = a_2^{[1]}$, $dz_1^{[1]} = dz_2^{[1]}$ - our hidden units are identical and compute the same function. The whole network is symmetric, and will update the weights exactly the same at each iteration. W remains a rank-1 matrix through every iteration.
- Instead $W^{[1]} = np.random.randn((2,2)) \times 0.01$, $b^{[1]} = np.zero((2,1))$, same for $W^{[2]}, b^{[2]}$. We want to initialize the weights to small values, since large values of W make large values of z and will be far out on the activation function -> the derivative will be very small and learning will be slow.
- We can choose a different constant than 0.01 for different situations - will be important in deeper learning.

Module 4 - Deep Neural Networks

Deep L-layer NN

- Define variable L = # of layers, so last layer = $n^{[L]}$, $\hat{y} = a^{[L]}$
- $n^{[l]}$ = # units if layer l. $a^{[l]}$ = activations in layer l. $a^{[l]} = g^{[l]}(z^{[l]})$,
- Forward propogation
  - $z^{[1]} = W^{[1]}x + b^{[1]},\ a^{[1]} = g^{[1]}(z^{[1]})$
  - Generally $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]},\ a^{[l]} = g^{[l]}(z^{[l]})$
  - etc down to layer 4 for L = 4, and the final variable $a^{[4]} = \hat{y}$
  - For vectorized formulae: $Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]},\ A^{[1]} = g^{[1]}(Z^{[1]})$ then $\hat{Y} = A^{[4]}$
  - Note we actually do loop over the layers to iteratively calculate the activations for each layer. Here `for l in range(1,4)`

Matrix Dimensions

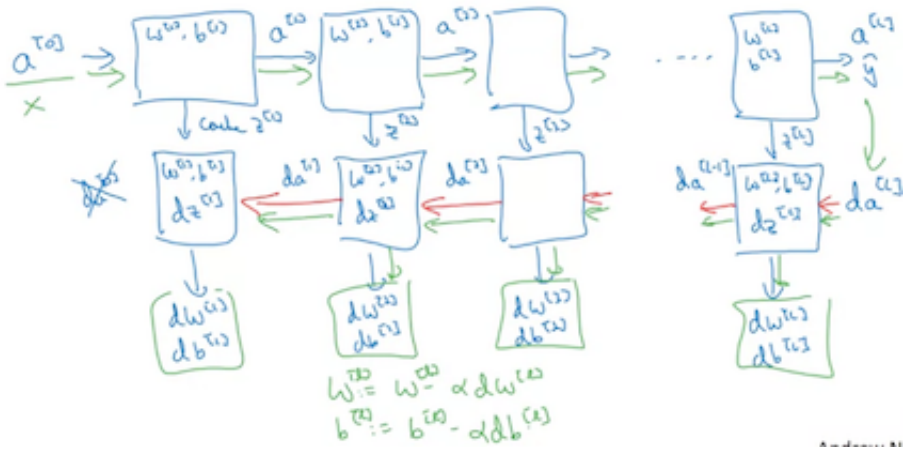
- One Vector
  - For example L = 5
  - $n^{[0]} = 2,\ n^{[1]} = 3,\ n^{[2]} = 5,\ n^{[3]} = 4,\ n^{[4]} = 2,\ n^{[5]} = 1$
  - $z^{[1]} = (n^{[1]}, 1) \leftarrow X = (n^{[0]}, 1)$ so W must be $W^{[1]} : (n^{[1]}, n^{[0]})$
  - W is a transformation matrix from X space to Z space.
  - General formula, $W^{[\ell]} : \left(n^{[\ell]}, n^{[\ell-1]}\right)$ and $b^{[\ell]} : (n^{[\ell]}, 1)$
  - For backprop: $dW^{[l]} : (n^{[l]}, n^{[l-1]})$, $db^{[l]} : (n^{[l]}, 1)$
- Vectorized Matrix Dimensions
  - $Z^{[l]} = (n^{[1]}, m)$, $X = (n^{[0]}, m)$, $W^{[1]} : (n^{[1]}, n^{[0]})$
  - $b^{[\ell]} : (n^{[\ell]}, m)$ by broadcasting
  - $Z^{[l]},\ A^{[l]} : (n^{[l]}, m)$

Deep Representation

- Deeper layers represent more complex expressions - encoding
- You can compute some functions with small L-layer deep neural network that shallower networks would require exponentially more hidden units to compute. Imagine making an XOR tree computing XOR between a string of variables, can perform it pairwise in a tree similar to a NN - depth of network $O(log(n))$. With just one hidden layer, this network would need to consider 2^n combinations in one layer.

Building Blocks of NN

- When going forward input $a^{[l-1]}$ has output $a^{[l]}$. Cache $z^{[l]}, W^{[l]}, b^{[l]}, a^{[l-1]}$
- For backprop, input $da^{[l]}$ has output $da^{[l-1]}, dW^{[l]}, db^{[l]}$, cache($z^{[l]}$)
- The cache contains the Z functions, since the actual output is a, Z after activation. We need the cache to compute the derivatives.
-



- Feed forward:

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

  $A^{[1]} = g^{[1]}\left(Z^{[1]}\right)$
- $Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$

  $A^{[2]} = g^{[2]}\left(Z^{[2]}\right)$

  $A^{[L]} = g^{[L]}\left(Z^{[L]}\right) = \mathcal{Y}$
- Backprop formulas for any layer:
  - $dZ^{[l]} = dA^{[l]} * g^{[1]\prime}\left(Z^{[1]}\right)$
  - $dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[l-1]T}$
  - $db^{[l]} = \frac{1}{m} np.\text{sum}\left(dZ^{[l]},\ \text{axis}\ = 1,\ \text{keepdims}\ =\ \text{True}\right)$
  - $dA^{[l-1]} = W^{[l]T} dZ^{[l]}$
  - When doing logistic regression, input for backdrop $da^{[l]} = -y/a + \frac{(1-y)}{(1-a)}$

Parameters and Hyperparameters

- Hyperparameters - learning rate, # iterations, # hidden layers L, # hidden units $n^{[i]}$, activation function
- We need to tell our learning algorithm these features. They control the parameters learned on the data
- We will add to this list - momentum, mini batch size, regularizations, ...
- Need to iterate through trying values for the hyperparamters to see their effect on the model.

C2 - Hyperparameter tuning, Regularization and Optimization

Module 1 - Practical Aspects of Deep Learning

Training / Dev / Test Sets

- So many hyperparameters to determine - layers, hidden units, learning rates, activation functions. This is why deep learning is such an iterative process. Often intuitions in one application area do not transfer well to another application
- Break data into training, validation / dev set, finally a test set. Use the validation set for CV and tuning, finally get results from test set.
- In the big data era, dev and test sets become much smaller % of total data than classic 70/30 split. Say

for 1mm examples, might just keep 10k for dev, 10k test.
- Mismatched training and dev / test sets. If you are pulling data from different sources, this will turn out poorly. Make sure dev and test are definitely from the same distribution, but many people will pull in additional training data that may come from different sources that may exhibit other features.
- Not having a test set might be ok - only needed if you want an unbiased estimate of your error.

Bias - Variance

- Flexible - high variance, Rigid - high bias.
- Key numbers are the train set error and dev set error. Great train set error but poor dev set error - we have high variance and have likely overfit.
- If instead, train set error is 15% and dev set error is 16%, assuming humans have good performance on this task, the algorithm is underfitting the data and has high bias.
- If instead 15% on training error and 30% on dev set error - have high variance and high bias. Worst of both worlds. Can construct high bias and high variance models that are highly linear in some areas while highly overfit in other regions - happens most in high dimensional inputs.
- All of this predicated on knowing the human error or optimal Bayes error rate. If instead Bayes were higher, then we would have to adjust how we interpret these numbers

Recipe for ML

- Bias: After training, do we have high bias? Look at the training performance. If we are not fitting the training too well, could increase the network size, train longer, change our architecture. Repeat trying solutions until our bias is reduced and we fit the training data pretty well.
- Variance: Next look at the dev set performance. If we have high variance, can we get more data, regularize, try another NN architecture?
- Notice these bias and variance solutions can be quite different - important to diagnose your problem correctly.
- Bias-Variance Tradeoff in DL - so long as you can make the NN deeper and get more data, we can reduce the bias or variance respectively without really any tradeoff from the other measure.

Regularization

- Frobenius norm: $\left\|w^{[l]}\right\|^2 = \sum_{i=1}^{n^l} \sum_{j=1}^{n^{[l-1]}} \left(w_{i,j}^{[l]}\right)^2$
- If you have high variance / overfitting, regularization is a good alternative to getting more data
- L2 norm for a vector: $\|\omega\|_2^2 = \sum_{j=1}^{n_x} \omega_j^2 = w^\top w$
- L2 regularization: $J(\omega, b) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y(i)\right) + \frac{\lambda}{2m} \|\omega\|_2^2$ - note we omit b from regularization
- L1 $\frac{\lambda}{2m} \|w\|_1$ instead will produce a sparse w, but in practice L2 is used much more often.
- In a neural network, we are dealing with matrices - $\frac{1}{m} \sum_{i=1}^{n} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m} \sum_{l=1}^{L} \left\|\omega^{(l)}\right\|_F^2$
- In gradient descent, we calculated dW from backprop, then update W
- Now we calculate the same dW, but add $\frac{\lambda}{m}\omega^{[l]}$. This is also called weight decay, since it is equivalent to multiplying W by a factor less than 1.
- Imagine fitting a large and deep NN, and it is overfitting the data. Our regularizer penalizes large weight matrices, so many of the hidden units carry small weights and the network is equivalent to a simpler NN with fewer hidden units.
- Additionally, look at the tanh activation function. With a large regularization parameter, z will be constrained to be small as well. This puts in the in linear range of tanh around 0 with a strong slope -

then every layer is roughly linear, and our model will have a near linear decision boundary.
- In gradient descent, make sure you are plotting the new definition of the cost function J - otherwise may not see a monotonic decrease as we expect.

Dropout Regularization

- Randomly eliminate a set of nodes across different layers. We are now training a smaller network on each example.
- Inverted Dropout - a method of implementing dropout, say with l=3. d3 is the dropout vector for layer 3: `d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob` for `keep_prob= 0.8`, the probability that a unit will be kept in the model. Then there is a 20% chance that the node in the layer will be dropped. Then `a3 = np.multiply(a3,d3)`, then d3 will zero out the entries of a3 that correspond by the dropout. Finally `a3 /= keep_prob` to reinflate the a3 weights to ensure the expected value of a3 remains the same.
- At test time, we do not use dropout and run the test points on the full network. This would just add noise to your predictions, since the dropout is part of random process.
- Note: In general, the number of neurons in the previous layer gives us the number of columns of the weight matrix, and the number of neurons in the current layer gives us the number of rows in the weight matrix.
- Intuition: we cannot put too much weight on any given unit, since it may not be present in the next iteration of training. Weights become spread over more units, reducing the size of weights on any given unit.
- `keep_prob` can vary by layer. The largest hidden layer would have the largest weights, so you might set keep_prob to be lower than other layers where we don't worry about overfitting. Smaller layers, we probably want minimal if any dropout.
- Computer vision makes heavy use of dropout since they almost never have enough data for the size of the network, but in other application areas would wait to see overfitting before trying dropout.
- Cost function J is less well defined and we lose the debugging tool of seeing a monotonically decreasing J in gradient descent. Can run the network without dropout just as a debugging tool to double check.

```
D1 = np.random.rand(A1.shape[0], A1.shape[1]) #init dropout
D1 = D1 <= keep_prob #convert to binary 0, 1 with threshold
A1 = D1 * A1 #shut down some neurons of A1
A1 = A1 / keep_prob #reflate the value of remaining neurons to keep expectation the same
```

Other Regularization Methods

- Data augmentation - say with image, you could flip or mirror image your training data to increase data size. Not as good as new data but a good proxy. Could also apply random distortions or cropping.
- Early stopping - as you run gradient descent, you plot the cost function J and it should decrease monotonically. With early stopping also plot the dev set error - this will be more of a U convex shape. Can stop training the neural network at the dev set min instead of minimizing J all the way. By stopping halfway, our W is smaller than the fully trained network.
    - Orthogonalization - you want to think about one objective at a time. Downside of early stopping is we are not considering optimizing J and preventing overfitting separately. Easier to search of all hyperparameters with L2 reg, but it is more computationally expensive since we have to search

over the lambda space.

Normalizing Inputs

- Formula for normalization: $\frac{x-\mu}{\sigma}$ - 0 mean and standard variance. The variance along each predictor variable is 1
- We normalize same mean and variance to standardize test and training set
- If we didn't normalize, cost function likely to be distorted to odd curve, which distorts the weighting values - elongated eliptical curves instead of circles. Gradient descent is likely to go much more directly to the minimum, instead of bouncing around.

Vanishing / Exploding Gradients

- Training difficult with very large and very small derivatives. Especially problematic in very deep networks, ignoring bias - $y = W^{[l]}W^{[l-1]}\dots W^{[0]}x$. Note $a^{[1]} = W^{[0]}x$, $a^{[2]} = W^{[1]}W^{[0]}x$, etc. We are taking large powers of the weight matrices, so the value of y can explode for weights over 1 (W > I). For weights less than 1 (W < I), the exponents shrink the weights to zero.
- Learning slows for tiny gradients, takes too large steps for large.
- Weight Initialization Fix: Say we have 4 inputs in X. $z = w_1x_1 + \dots w_nx_n$ - for larger n we want smaller w's to balance out. Can set $Var(w_i) = 1/n$. In practice, set $W^{[l]} =$ `np.random.randn(slope)*np.sqrt(2/n^{[l-1]})` (better to set variance to 2 instead of 1). This is the fix generally for ReLU
- For tanh, we tend to use Xavier initialization: $\sqrt{\frac{1}{n^{[l-1]}}}$ or $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$. These initializations for variances can be tuned as well.
- In practice, Xavier initialization rule $var(W^{[l]}) = \frac{2}{n^{[l-1]}n^{[l]}}$ for tanh activations (like setting the variance in np.random.randn, we use this in the initialization).
- He Initialization, used for ReLU activations: uses $var(W^{[l]}) = \frac{4}{n^{[l-1]}n^{[l]}}$. With ReLU cannot make the same linear approximation we used in tanh.

Gradient Approximation and Checking

- Numerical approximation - better approximations of derivatives by taking triangle around point we are interested in: $\frac{f(\theta+\epsilon)-f(\theta-\epsilon)}{2\epsilon} \approx g(\theta)$ - notice this is the difference quotient in the limit. The error is on the order of $O(\epsilon^2)$ instead of one side epsilon with error on order of $O(\epsilon)$
- Gradient checking for debugging and verification. Take all parameters W, b, concatenate and reshape into a big vector θ. Take the dW, db, etc into a big vector $d\theta$.
- $J(\theta) = J(\theta_1, \theta_2, \dots)$. For each i, $d\theta_{approx}[i] = \frac{J(\theta_1,\dots,\theta_i+\epsilon,\dots)-J(\theta_1,\dots,\theta_i-\epsilon,\dots)}{2\epsilon} \approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$
- We take $\frac{||d\theta_{approx}-d\theta||_2}{||d\theta||_2+||d\theta||_2} \approx 10^{-7}$ is a good range for the approximation to be close to the expected gradient. If it climbs to 10^{-3}, should be cause for worry.
- This is not used for training, just debugging - backprop is reserved for actual training. Remember need to include any regularization in checking, doesn't work with dropout.
- If algorithm fails grad check, look at components, see which have the largest differences. Remember if you included regularization, you need to include it in the grad check (note this won't work for dropout since it is randomized). Run at random initialization, then try again after some training since the problem may be cause by extremely small weights that will grow over time.

Module 2 - Optimization Algorithms

Mini-batch Gradient Descent

- Vectorization allows you to compute on m examples. If m = 5mm, we have to process the whole dataset to take a single step in gradient descent
- If we split up the training set into mini-batches of say 1000, and call these batches $X^{\{1\}}, X^{\{2\}}, \ldots$ for $X^{\{1\}} =^{(1)}, \ldots, x^{(1000)}$. Then we have mini-batch t $X^{\{t\}}, Y^{\{t\}}$
- Running mini-batch gradient descent - for t = 1,...,5000 we take 1 step of grad descent using $X^{\{t\}}, Y^{\{t\}}$, vectorizing calcs over those 1000 examples in each mini-batch. Perform backprop to compute gradients for the batch - this is 1 epoch of training.
- Here, the cost function may not monotonically decrease, since we minimize over new training batches at each epoch. Should be jittery, but trend downwards. Some mini-batches may be easier to minimize than others, leading to non-monotonic minimization.
- Choosing mini-batch size - if batch size = m, we have batch gradient descent. At the other extreme, could set batch size to 1, producing stochastic gradient descent where every example is its own mini-batch $X^{\{t\}}, Y^{\{t\}} = x^{(t)}, y^{(t)}$
    - SGD won't march directly to the minimum like BGD. Examples may lead it in an oscillating or jittery path.
    - In practice, use something in between. BGD takes too long to process the large dataset at each step. SGD loses all the speedup from vectorization (note the jitter can be controlled via learning rate). MBGD gives us the fastest learning in practice; we get vectorized speed ups while making progress with smaller datasets.
    - If m less than 2000, just use BGD. Otherwise often use MB sizes of 64, 128, 256, 512. Make sure MBs fit in CPU / GPU memory

Exponentially Weighted Averages

- Say the temperature in a city over time - cold in the winter, warm in the summer. Might be useful to have a moving average. Initialize $V_0 = 0,\ V_1 = 0.9V_0 + 0.1\theta_1$ and generally $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
- V_t is approximately average over $\approx \frac{1}{1-\beta}$ days' temperature. With a high value of beta, our curve gets much smoother, but the curve shifts as it take more days of changes to move the curve - it reacts much more slowly to changes day over day. Similarly, over smaller beta like 1/2, we average over a small number of days and the curve is highly variable
- Notice this formulas are recursive. We essentially have an exponentially decaying function in the thetas of prior days. Note $(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$ - use this as the gauge to see how far back weights hold power for a given epsilon/beta.
- Implementation - initialize, V=0, then each day update V $V_\theta := \beta V_\theta + (1-\beta)\theta_t$. This takes very little memory to compute this weighted average.
- Bias correction - when we initialize to 0, we get a poor estimate of the first data points. For day t, use $\frac{V_t}{1-\beta^t}$ instead. When t is large, denominator becomes 1, but for small t, the denominator boosts the initial values.

Gradient Descent with Momentum

- Compute exponentially weight average of your gradient and use this to update your weights.
- If you have elliptical level curves, GD tends to oscillate back and forth, forcing us to use a slower learning rate. On one axis (shorter) we want slower learning and the other (longer) axis want faster learning.

- Momentum - on iteration t, compute dW, db on current MB. Compute $V_{dW} = \beta V_{dW} + (1-\beta)dW$, the MA for the derivatives of W. $V_{db} = \beta V_{db} + (1-\beta)db$. Then $W = W - \alpha V_{dW}$, $b := b - \alpha V_{db}$. In long axis direction, derivatives are aligned and GD moves quickly in this direction, but in the shorter axis derivates cancel out and the moving average dampens the oscillations.
- The dW term is like the acceleration and V_{db} is the velocity. The beta acts as friction on the ball rolling down the bowl.
- β is a new hyperparameter we tune alongside α, though commonly beta is 0.9, the average over the last 10 gradients. In practice, most do not perform bias correction, since within ten iterations we should have a good approximation for the gradient. Additionally, often omit $(1-\beta)$, and run $V_{dW} = \beta V_{dW} + dW$, though this is a bit less intuitive.

RMSprop

- Root mean squared prop. Speed up learning on the long axis, slow it on the short.
- On iteration t, compute dW, db on current MB. $S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2$ where square is element wise operation. $S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$. Then $W = W - \alpha \frac{dW}{\sqrt{S_{dW}} + \epsilon}$, $b = b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$. If we say W is along the long direction, it will have small dW, and b along short direction will have big db. Net effect is the the short axis direction, we divide by a larger number dampening updates and the opposite for the long direction. In practice we are in higher dimensions, could be some W vector.
- Note the epsilons are to ensure numerical stability - ie. not dividing by zero or a number close to zero.

Adam Optimization

- Adaptive Moment Estimation. Init: $V_{dW} = 0, S_{dW} = 0,\ V_{db} = 0,\ S_{db} = 0$
- On iteration t: compute dW, db using current MB. Use momentum and RMSprop equations to update the parameters above.
- Implement bias correction $V_{dW}^{corrected} = V_{dW}/(1-\beta_1^t),\ V_{db}^{corrected} = V_{db}/(1-\beta_1^t),\ S_{dW}^{corrected} = S_{dW}/(1-\beta_2^t),\ S_{db}^{corrected} = S_{db}/(1-\beta_2^t)$
- Then to update parameters - $W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$ and $b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$
- Many hyperparameters: α needs to be tuned, $\beta_1 : 0.9,\ \beta_2 : 0.999,\ \epsilon : 10^{-8}$

Learning Rate Decay

- Slowly reduce your learning rate over time. Formula: $\alpha = \frac{1}{1 + decayRate \times \text{ epoch } Number} \alpha_0$
- Algorithm in GD may not fully converge to minimum, keeps taking steps around the min. If we reduce alpha over time, we oscillate in a tighter region around the min as we get closer. We can afford to take larger steps at the beginning and want smaller steps when we are close.
- Can try a variety of values for hyperparameters α_0 and decay rate
- Could also rate decay in other ways. For example $\alpha = 0.95^{\text{epoch num}} \alpha_0$ or $\alpha = \frac{k}{\sqrt{\text{epoch num}}} \alpha_0$, discrete stepwise function, manual decay
- Learning rate decay likely lower down on the list of things to try to improve your NN

Local Optima and Saddle Points

- In high dimensions - actually most points of zero gradients are not local optima but are saddle points. This is because you would need all p dimensions forming a cup or cap together, but much more likely to have some dimensions with positive and others with negative derivatives.
- Problem of plateaus - areas of zero for a large portion of a surface. This means it will take a long time to

find a way towards an area of more extreme gradient.

Module 3 - Tuning, Batch Normalization, Programming Frameworks

Hyperparameter Tuning

- $\alpha, \beta, \beta_1, \beta_2, \epsilon$, # layers, # hidden units, learning rate decay, mini-batch size
- Importance: 1 - learning rate. 2 - beta, hidden units, mini batch size, 3 - # layers, learning rate decay
- Best tuning method - use random values, do not use grid search. We have a large search space, we do not know in advance which hyperparameters will be most important for your application. A lot of pairwise options, fixing one hyperparameter does not make sense, since may often get similar output for fixing one hyperparameter.
- Also work coarse to fine - first sample space coarsely, if there is a region that looks good, sample more finely within this smaller space.
- This does not mean sampling uniformly at random - we have to consider the scale of values. Say picking # of hidden units, 50 -100 is a reasonable search area. Number of layers, could look at just 2-4. In both examples could sample uniformly at random within this range.
- For learning rate, lambda could range 0.0001 - 1 - if we sampled uniformly at random, 99% of values would be between 0.1 - 1. Instead makes more sense to sample from the log scale, so we dedicate more resources to searching within orders of magnitude. In python `r = -4 * np.random.rand()` and `alpha = 10**r`, producing alpha range from $10^{-4} - 10^0$. Generally set `r =range(a,b)` to get $10^a - 10^b$ range for alpha.
- Beta for exponentially weighted averages. $\beta = 0.9....0.999$ equivalent to using last 10 values vs last 1000 values. Makes sense to explore range of $1 - \beta = 0.1....0.0001$ and now can apply the same method as for the learning rate, though the values of the left and right are swapped. `r = range(-3,-1)` and $1 - \beta = 10^r$. Why is this so important - as beta approaches 1, the sensitivity to that change increases $\beta = 0.999 \rightarrow 0.9995$ goes from average over 1000 to average over last 2000 values.

Tuning in Practice

- Re-test hyperparameters occasionally - intuitions within a single application do get stale, changes to anything from hardware to software can cause the need to retune
- Often people are babysitting one model - on day 0 initialize params at random, then day 1 try throwing in some tweaks, every day you modulate some values. This is what happens without the computation ability to train many models in parallel
- Otherwise train models in parallel set to different values of hyperparameters and look at their ability to learn over time.
- Panda - the babysitting approach, have few offspring and carefully curate. Caviar - train in parallel, like fish having many offspring.
- Choosing between them depends on your computational resources - Caviar is great if you can afford it.

Batch Normalization

- Makes your NN much more robust to choice of hyperparameters
- Normalizing features in NN can speed up learning as we have seen. Turn the contours from elongated to much more circular / round.
- In a deeper model. can we normalize the values of $a^{[l]}$ so as to train $w^{[l+1]}$, $b^{[l+1]}$ faster? This is what batch normalization does - we normalize $Z^{[l]}$
- Given some intermediate values in NN, $z^{(1)}, \ldots, z^{(m)}$ for layer ℓ. Computer the mean $\mu = \frac{1}{m} \sum_i z^{(i)}$,

$\sigma^2 = \frac{1}{m}\sum_i \left(z^{(i)} - \mu\right)^2$ then normalized z $z^{(i)}_{norm} = \frac{z^{(i)-\mu}}{\sqrt{\sigma^2+\epsilon}}$ (eps for sigma = 0, just in case).

- Don't want hidden units to always have mean 0, var 1 so instead define $\tilde{z^{(i)}} = \gamma z^{(i)}_{norm} + \beta$ where γ, β are learnable parameters of the model. Use GD or some other algo, update these parameters like all other parameters of the NN. If for example $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu,$ we would invert this equation and $\tilde{z}^{(i)} = z^{(i)}$. Now for later computations in the network, use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$
- This allows us to construct the range of values for z that we want, while the hidden units have standardized means and variances, not necessarily 0 and 1 respectively.
- In practice take $x \rightarrow z^{[l]} \rightarrow \beta^{[l]}, \gamma^{[l]} \rightarrow \tilde{z}^{[l]} \rightarrow a^{[l]} = g^{[l]}(\tilde{z}^{[l]}) \rightarrow z^{[l+1]}$ etc
- Parameters, all W and b, now also $\beta^{[l]}, \gamma^{[l]}$, then have to find gradients, optimization algorithm, to train.
- Working with mini-batches - $X^{\{1\}} \rightarrow z^{[1]} \rightarrow BN \rightarrow \tilde{z}^{[1]} \ldots$ all for one mini-batch. Then repeat for the next minibatch, at each iteration, just using the data from the batch.
- Note: $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$, but the batch norm will subtract out the b term! So with BN we can eliminate b terms st $z^{[l]} = W^{[l]}a^{[l-1]}$ and $\tilde{z}^{[l]} = \gamma^{[l]}z^{[l]}_{norm} + \beta^{[l]}$
- Finally note $\beta^{[l]}, \gamma^{[l]}$ size $(n^{[l]} \times 1)$

How Batch Norm Works

- Makes weights deeper in the network more robust to changes to earlier layers
- Say we have trained NN on images of black cats, and now want it to recognize a variety of colored cats. Want more generalization - data distribution changing = covariate shift. May need to retrain learning algorithm
- In a deep NN, from perspective of middle layer, gets some values $a^{[l-1]}_{1-n}$, but those are also dependent on prior values. These values are changing all the time and suffer from covariate shift. BN reduces the amount that the distribution of the prior a's shift around. The mean and variance of these layers stay the same, even as the values shift.
- Additionally, batch norm has a regularization effect. Each mini batch is scaled by the mean / var of just that mini batch, which mean it contains some noise from the sample.
    - The scaling process is then also noisy, so similar to dropout, adds noise to each activation in the hidden layers. By using larger minibatches, the regularization effect decreases since samples are lessed biased. Not really a good option for intended regularization, but important to realize it has this effect regardless

Batch Norm at Test Time

- At test time, may not have a mini batch to take a mean or variance over. With one example, you do not have these sample stats
- Instead estimate mean and variance using exponentially weighted average across minibatches
- Say have $\mu^{\{i\}[l]}$ for minibatch i in layer l. These are your θ_i in exponentialyl weighted average and get an overall μ. Do similar for variance σ^2. Then at test time, plug these into $z^{(i)}_{\text{norm}} = \frac{z^{(i)-\mu}}{\sqrt{\sigma^2+\varepsilon}}$

Softmax for Multiclass

- Formula: $a^{[L]} = \frac{e^{Z^L}}{\sum_{i=i}^4 t_i}$
- Softmax regression is a generalization of logistic regression to multiple classes, say recognizing cats, dogs, chicks, coded 1,2,3 and 0 for other.
- C = # of classes, in example 4 for C in [0,1,2,3]. Number of units in output layer = C, ie. $n^{[L]} = C$

- Each unit in the last layer has a different conditional probability, eg $P(cat|x), P(dog|x)\ldots$
- In last layer, compute $z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}$. Then compute the softmax activation, for $t = e^{z^{[L]}}$ where t is also (4,1). Then $a^{[L]} = \frac{e^{z^L}}{\sum_{i=i}^4 t_i} = \frac{t}{\sum_{i=i}^4 t_i}$ where $\sum_{i=i}^4 t_i$ is the sum of t's across the 4 elements of Z.
- Output of NN will be $a^{[L]}$ is (4,1), 4 different probabilities expressing the conditional probability for each category. This is the first activation we have seen that takes in vectors and outputs vectors, instead of mapping $\mathbb{R} \to \mathbb{R}$

Training a Softmax NN

- Example: $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$ and $g^{[L]}\left(z^{[L]}\right) = \begin{bmatrix} e^5 / \left(e^5 + e^2 + e^{-1} + e^3\right) \\ e^2 / \left(e^5 + e^2 + e^{-1} + e^3\right) \\ e^{-1} / \left(e^5 + e^2 + e^{-1} + e^3\right) \\ e^3 / \left(e^5 + e^2 + e^{-1} + e^3\right) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}.$
  Soft since we do not map the last vector to a vector with a single 1 and rest 0 - we have a confidence level to our prediction.
- Loss function: $L(\hat{y}, y) = -\sum_{j=1}^C y_j log(\hat{y}_j)$
  - Say target y = [0,1,0,0]. Then $y_1 = y_3 = y_4 = 0$ and left with $-y_2 log(\hat{y}_2) = -log(\hat{y_2})$. To minimize, need $\hat{y_2}$ large, so we take the MLE for maximize the likelihood for y_2.
  - Cost function is same form as always - average over loss
- Gradient Descent - key equation for backprop $dz^L = \hat{y} - y = \frac{\partial J}{\partial z^L}$

Tensorflow

- Many programming frameworks. Choosing one - easy of programming, running speed, OSS w/ good governance
- Cost function J to min, $J = w^2 - 10w + 25 = (w-5)^2$. This one is simple, but many will not be.

```
import numpy as np
import tensorflow as tf

coefficient = np.array([[1.],[-10.],[25.]])

# define vars and training func
w = tf.Variable(0, dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1]) # for our training data, var whose value we assign
later.
cost = tf.add(tf.add(w**2, tf.multiply(-10, w)),25)
# cost = w**2 - 10*w + 25 also works
# cost = x[0][0]*w**2 + x[1][0]*w + x[2][0] if we have data to use here
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session() #can use with tf.Session() as session:
session.run(init)
sess.run(w)

# run 1 step of GD
#session.run(train)
```

```
session.run(train, feed_dict=[x:coefficients]) # this gets the data into the trainin
function
print(session.run(w))

# run 1000 steps of GD
for i in range(1000):
  #session.run(train)
  session.run(train, feed_dict=[x:coefficients])

print(session.run(w))
```

C3 - Structuring Machine Learning Projects

Module 1 - ML Strategy (1)

Orthogonalization

- Understanding what to tune to see certain effects
- Want adjustment to affect a single aspect of your model.
- Chain of assumptions: fit training set well on cost function, fit dev set well on cost function, fit test set well, system performs in real world.
- Different knobs to tune for each part of this chain - say for training, tune bigger network, etc. For dev set, regularization, bigger training set. For test set, bigger dev set. For real world performance, change dev set or cost function.
- Early stopping, while useful, is a messy knob that simultaneously affects the training and dev set performance.

Setting a Goal

- Single number evaluation metric - helps determine if you are moving in the right or wrong direction.
    - Say you are measuring precision, of positive predictions how many are TP, and recall, of all true in data what % were correctly recognized. If one method has better precision other has better recall, what to do?
    - Define a new evaluation metric that combines them - F1 score = average of P and R (formally, $\frac{2}{1/P+1/R}$, harmonic mean).
    - Say tracking performance of different models in different geographies - while detail may be useful eventually, want to track overall average to have a single metric
- Satisficing and Optimizing metrics
    - Care about accuracy and running time of a model. Could combine them into a single metric to optimize, but very different units and meanings.
    - Could choose a classifier that max's accuracy subject to a run time constraint. Accuracy is an optimizing metric then and run time is a satisficing metric - just has to be good enough, not optimized. Lots of trade offs, say accuracy and false positives, etc.
    - For N metrics - choose 1 optimizing, N-1 satisficing.
- Train / dev / test sets
    - Say you are working across many regions. Should not simply split regions into dev or test - want dev and test to come from the same distribution

    - Dev set + single eval metric is like setting a target - the iteration process is meant to hit a bullseye for this combo. If dev set not representative, then could waste a lot of time
    - Try to get test / dev data you expect to get in the future and consider important to do well on.
    - Could see that your dev / test sets don't match real world user data, say blurrier photos, etc. Would need to alter these datasets to perform better on production environment
- Could have a metric that incorrectly ranks algorithms - maybe one performs better on classification error but let's through inappropriate images. Need to change our error definition - don't want to treat these kinds of errors equally. Add a weighting against bad images for example.
- Orthogonalize the problem - think about how to define a metric to evaluate classifiers, place the target. As a separate process, how do we do well on this metric, hit the target.

Comparison to Human Performance

- Often accuracy increases over time until it surpasses human level performance but then leveling off. It approaches the Bayes error
- Possible there may not be much more room to improve beyond human level, but also a harder task to improve
- As long as ML algo is worse than human performance, you can get labeled data from humans, gain insight from manual error analysis, analyze bias and variance, but once humans cannot help with the task it is harder to improve.
- Avoidable Bias
    - Say humans have near perfect accuracy on a task. Say training error is 8%, may want to improve performance on the training set and focus on reducing bias.
    - Instead imagine human performance is actually 7.5% error. Then bias is low compared to humans, and we may want to focus on variance of dev set.
    - The training / dev errors are relative to humans (a proxy Bayes error) - different steps are taken based on this comparison.
    - Difference between Bayes error and training error = Avoidable bias. Difference between training and dev error is the variance.
    - Reducing avoidable bias - train bigger model train longer or better optimization algos, try different NN architecture / hyperparameter search
    - Reducing training - dev variance - more training data, regularization, data augmentation, NN architecture / hyperparameter search
- How to define human level error - if a team of experienced doctors can achieve the lowest bound for a human, then the Bayes error must be less than that. So using the best possible human performance is a good goal to shoot for. But of course this depends on the application of the production system.
- Your estimate for Bayes error changes how much avoidable bias you believe exists and where you focus your attention.
- Surpassing Human Level Performance - Training and dev error below best human performance - have we overfit? Is Bayes error actually lower?

Module 2 - ML Strategy (2)

Error Analysis

- Manually examining the mistakes the NN is making when below human performance

- If classifier is bad on say dog pictures, is it worth adding effort to make it better. Perform analysis by taking 100 mislabeled dev set examples and count how many are dogs. If 5/100 are dogs, spending time on this problem will have a small effect on our overall error. There is a low ceiling on the improvement. If 50/100 of the examples are dogs, then we could cut our error rate in half by working on this problem.
- Can evaluate multiple ideas in parallel - improve performance on blurry images, fix specific kinds of misclassification, etc.
- Create a table, rows are the individual images, and columns are the ideas to improve NN. Then count what percent of images are attrubuted to each error category. Can also add error categories as we are evaluating. But helps show where the biggest improvements could come from.
- Incorrectly labeled examples - the human label for Y in our dataset is incorrect.
    - In the training set, not a big deal - DL algos are robust to random errors in the training set. But they are less robust to systematic errors, so want to make sure its not a certain skew.
    - In the dev set, can add this as a column in our error analysis table to double check. If it makes a significant difference in the dev set may want to relabel, but otherwise may not be a good use of your time. If our system has 10% error and 6% of those errors are due to incorrect labels, then overall 0.6% error caused by incorrect labels - not a large amount. If same percent of 2% overall error, would be more worthwhile.
    - Correcting dev/test set exampels - apply the same process to dev and test set. Make sure to look at examples algo got right as well as wrong - FP and FN. Train and dev/test can come from slightly different distributions, but keep dev and test tightly aligned.
- Good idea to build a system quickly then iterate to improve. Set a dev/test set and metric, build the system, use bias/variance analysis and error analysis to find the next priority. There can be a lot of problems in tackling a DL problem but not obvious without an initial system what to prioritize. This may be less relevant if you have significant prior experience in an area or are following specific literature.

Mismatched Data Sets

- Training and testing on different distributions - could grab all kinds of data from the web, users, etc, but your goal is to categorize user uploaded data. Can have a dilemma between small training dataset and large dataset from a different distribution.
- One option is to take all of the data, randomly shuffle them, then split into training, dev, test. Advantage that all 3 sets from the same distribution, but dev set will have large proportion from the web instead of the goal we are actually focused on.
- Instead, better to take all of the augmented web data, then maybe half of the user data. Dev and test sets are all user data - this sets the correct target for task we want to do well on. Disadvantage that training distribution is now different, but this is ok.
- Very common for teams to buy data, take data from other applications, etc and drop into the training set. The dev/test sets reserved for the current application data we have. May even place all of our best data into dev and test.
- Bias and Variance
    - When training and dev come from different distributions, performance differences between trianing and dev may not reflect a high variance model - could have different levels of difficulty. Can create a training-dev set - same distribution as training set but not used for training, this is used for the error analysis. If errors are train - 1, train-dev 9, dev - 10 - we have a variance problem

between our train and train-dev. If the gap is between train-dev and dev, not a variance problem, data mismatch problem. Finally, dev - test gaps tells us about the degree of overfitting to the dev set, since come from the same distribution

- Sometimes the training data is much harder than the dev/test distribution, so error can decrease. Can create a general table: comparison of rows 1-2 is avoidable bias, 2-3 variance. Across columns we look at data mismatch

	General data	**Specific data in dev/test**
Human level	Human Level	Human Level
Error on examples trained on	Training Error	Training Error
Error on examples not trained on	Training-Dev Error	Dev/Test error

- When you find mismatch, good to carry out error analysis to understand the differences between sets. Could collect more data similar to dev/test sets for training.
- Could also make training set more similar via artificial data synthesis - eg. adding car noise to clean speech samples. If have much more speech data than car noise, don't want to just repeat the car noise to fit the length of training data - could be overfitting. Similarly for images, if you just synthesize new images you might overfit.

Learning from Multiple Tasks

- Transfer learning - train the NN for a certain task, say image recognition. Then swap in a new dataset, say radiology images and y's diagnoses. We retrain the NN on the new dataset - may just retrain the last layer if small dataset, or train all layers if large dataset. If you retrain the whole network, the original training is called pretraining, and the retraining on the specific data is fine-tuning.
- Take knowledge learned on image recognition and applied to specific application - a lot of the low level features are shared.
- Transfer learning makes sense when we have a lot of data we could transfer from and little data we want to transfer to. We can learn a lot of useful features from the image classification, then use those features for radiology. If we have a lot of radiology data, then unlikely that transfer learning will improve performance at all.
- To tranfer from A to B - want A and B to have the same input x, have a lot more data for task A than B, low level features from A could be helpful for learning B.
- Multi-task learning - try to have one NN do several things at the same time. Self driving cars need to detect pedestrians, cars, signs, lights, etc. Can stack these outputs into a vector, Y is a (4 x m) matrix, 4 for features to classify and m for training examples.
  - To train $\hat{y}^{(i)}_{(4\times1)} = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}_j^{(i)}, y_j^{(i)})$. Different from soft max since one image can have multiple labels.
  - Also works when some of the training data is not fully labeled - some features labeled but maybe don't know if there is a stop sign. Change the inner sum to sum over values of j with 0/1 label - if unknown just omitted
  - Makes sense to do when training on a set of tasks with shared lower level features, amount of data for each task is similar, can train a big enough neural network to do well on all the tasks. Could

have trained one NN for each task, but multi task can suffer if you cannot make a big enough network. Transfer learning is used in practice more often than multi-task.

End-to-end Deep Learning

- Redesigning systems to work without taking all intermediate steps. With a smaller dataset, the prior pipeline may work better, but with large data, the end to end learning works more efficiently.
- Take an image in a camera, could try to design some system mapping image to identity, but the person could approach in any number of ways.
- Instead apply one piece of software to detect the face in the image, then crop the image to center the face, then feed the face to the NN to verify identity. Break complicated sequence into simplier modules.
- This works because we have access to a lot of data for the 2 subtasks - there is a lot of labeled data for recognizing faces, and separately a lot of data for determining identity. By contrast learning everything at once, we don't have global teams working on this problem.
- E2E lets the data speak - reflects more what is in the data than human preconception - eg. used to break down language into phonemes, but why force the learning algo to learn phonemes.
- On the other hand requires large amount of data and excludes potentially useful hand-designed components. When you have a ton of data, may not need designed features but without it, human knowledge may improve a model.
- Key question - do you have sufficient data to learn a function of the complexity needed to map x to y. The function needed to map a hand directly to the age of a child requires a ton of data.
- Not E2E self driving - take image, radar, lidar, then detect cars, pedestrians, etc. Then plan the route given these objects, and execute steering/breaking. The first part is easily done with deep learning, but the route is usually done through motion planning algorithms, and steering through a control system.
- You can use DL to learn individual components, then carefully choose x and y depending on what tasks you can get data for. But E2E for self-driving seems extremely difficult and is less promising.

C4 - Convolutional Neural Networks

Module 1 - Foundations of CNNs

Computer Vision

- Image classification, object detection, neural style transfer
- If you work with larger images, can have 1000 x 1000 x 3 image - now have 3 million input features. Dimensions become enormous

Edge Detection

- Vertical edge detection on a grayscale image, say 6 x 6
- Construct a 3 x 3 filter $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$. We use `*` to denote convolution. We take the filter and convolve over the image, taking the element-wise product and adding together. This is the entry in a new 4 x 4 matrix. For the second element, shift our filter one step to the right, calculate the sum of element-wise products and fill in. Repeat.
- In Python `conv-forward`, in tf `tf.nn.conv2d`, keras `conv2D`, etc.
- Why is this vertical edge detection? Say we have image, filter, and convolved image:

$$\begin{array}{|c|c|c|c|c|c|}\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline\end{array} \times \begin{array}{|c|c|c|}\hline 1&0&-1\\\hline 1&0&-1\\\hline 1&0&-1\\\hline\end{array} = \begin{array}{|c|c|c|c|}\hline 0&30&30&0\\\hline 0&30&30&0\\\hline 0&30&30&0\\\hline 0&30&30&0\\\hline\end{array}$$ - clearly has a strong vertical edge. Can think of our filter as light, middle shade, dark shade. We end up with a light region right in the middle, corresponding to the detected edge right down the middle of the image - otherwise products cancel out in the sum. A vertical edge is a 3 x 3 region where there are bright pixels on the left and dark pixels on the right.

- If we flipped the image but used the same filter, we end up with negative 30s instead of positive - dark segment in the middle. Similar to our vertical filter, we can construct a horizontal filter $\begin{array}{|c|c|c|}\hline 1&1&1\\\hline 0&0&0\\\hline -1&-1&-1\\\hline\end{array}$
- Given a more interesting task:

$$\begin{array}{|c|c|c|c|c|c|}\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline 10&10&10&0&0&0\\\hline 0&0&0&10&10&10\\\hline 0&0&0&10&10&10\\\hline 0&0&0&10&10&10\\\hline\end{array} \times \begin{array}{|c|c|c|}\hline 1&1&1\\\hline 0&0&0\\\hline -1&-1&-1\\\hline\end{array} = \begin{array}{|c|c|c|c|}\hline 0&0&0&0\\\hline 30&10&-10&-30\\\hline 30&10&-10&-30\\\hline 0&0&0&0\\\hline\end{array}$$

- We can also learn these filter weights, can also detect more complex orientations, allowing more robust learning of low level features

Padding

- For n x n image and f x f filter, we get (n - f + 1 x n - f + 1), but we may not want our image to shrink every time we do this. It also treats our pixels unequally - a corner only gets included in one convolution while middle included in many. 2 big problems - shrinking image over layers and discarding edge information
- To fix them, we pad the image with extra pixels. Turn the 6 x 6 into 8 x 8, convolved with 3 x 3, we get another 6 x 6 image. Padding with 0 values. We get an image (n + 2p - f + 1) x (n + 2p - f + 1)
- Valid conv - no padding. Same conv. - pad so that output size is same as input size. This means $p = \frac{f-1}{2}$, note that f is usually odd. Typically want a central pixel

Strided Convolutions

- Stride of 2 say, we take the element-wise product as usually, but instead of moving the filter one step, we move it over 2 steps. For a 7x7 image with 3x3 image, we are just taking three convolutions of first row. Same goes for jumping to new rows - we move it down 2 steps. Our output is then 3 x 3
- Take n x n with f x f filter, padding p and stride s, the output is $\frac{n+2p-f}{s} + 1 \times \frac{n+2p-f}{s} + 1$.
- We take the floor of these dimensions if non integral. We do not take partial convolutions - filter must lie entirely within image + padding

Convolutions over Volumes

- RGB image has additional dimension - 6 x 6 x 3. Now our filter must also have another dimension 3 x 3 x 3. The end dimension of the filter is always the dimension of the channels (colors).

- Output is 2D 4 x 4 since the filter is covering all color dimensions at once. Filter has 27 numbers and do the same element-wise multiplication and sum to get a single number for the output matrix.
- Could have different 3 x 3 filters across colors or the same - will change the feature we are detecting.
- Multiple filters
    - Combine what we are trying to detect at a given time. We take say a vertical edge filter, then also convolve by a 2nd edge detector. Now we have 2 4 x 4 output matrices, and we can stack these - 4 x 4 x 2 output volume. Can say $(n \times n \times n_c) \times (f \times f \times n_c) \rightarrow n - f + 1 \times n - f + 1 \times n_c'$

One Layer CNN

- Image + filter -> ReLU((4 x 4) + b) -> (4 x 4). Perform for each filter, then stack our final outputs.
- This is a single layer of a CNN. Can think of the first 4 x 4 output as W x a, Then (4 x 4) + b equivalent to Z, then passed into activation.
- With 10 filters that are 3 x 3 x 3 in one layer of a NN, how many parameters does that layer have? Well filter 1 has 3 x 3 x 3 + bias = 28 params. Over 10 filters, total of 280 parameters. Fewer parameters than FC units means less prone to overfitting
- Notation: $f^{[l]}$ = filter size, $p^{[l]}$ = padding, $s^{[l]}$ = stride.
- Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
- Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- Volume size for layer (output): $n_H^{[l]} = \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$, same for $n_W^{[l]}$
- Number of filters: $n_c^{[l]}$
- Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
- Activations: $a^{[l]} = n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ or with matrix in BGD, $A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ since $n_c^{[l]}$ is the number of filters in layer l
- Bias: $n_c^{[l]} - \left(1, 1, 1, n_c^{[l]}\right)$
- Generally in a many layer network - the size of the image shrinks while the number of channels grows
- In a larger network usually combine types of layers - convolution, pooling, and fully connected layers (FC).

Pooling Layers

- Say we take a 4 x 4, f = 2, s = 2 we get a 2 x 2 with no overlap from convolutions.
- Max pooling, take the max from each of these 4 regions in the 4 x 4. Often used simply because it works well in practice. We can also do it with overlapping convolutions, take the max instead of element-wise products.
- If you have a 3D input, then the output will have that extra dimension as well - perform pooling on each channel separately.
- Average pooling - instead of taking the max's we take the average. Used less but sometimes used to collapse dimensions 7 x 7 x 100 -> 1 x 1 x 100
- So for pooling, we have hyperparameters filter size, stride, and max / average.
- Could add extra hyperparameter for padding, but generally never used.
- Input is $n_H \times n_W \times n_C$ and output is $\frac{n_H - f}{s} + 1 \times \frac{n_W - f}{s} + 1 \times n_c$
- There are no parameters to learn by pooling - it is a fixed function.

CNN Example

- Image 32 x 32 x 3 for digit recognition. We will build something similar to LeNet - 5

- Conv layer: f = 5, s = 1, p = 0. Output 28 x 28 x 6 from 6 filters (CONV1)
- Pool Layer: max pooling, f=2, s=2. Output: 14 x 14 x 6 (POOL1)
- We call CONV1 + POOL1 = Layer 1
- Conv layer: f = 5, s = 1, 16 filters. Output: 10 x 10 x 16 (CONV2)
- Pool Layer: f = 2, s = 2. Output: 5 x 5 x 16 (POOL2)
- Layer 2 = CONV2 + POOL2
- Fatten POOL2 into 400 x 1 vector
- Fully connected layer: FC3, fully connected 120 units, where $w^{[3]} = (120, 400)$
- Add another FC: 84 units (FC4). Feed this to softmax unit with 10 outputs.
-

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	608
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	3216
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48120
FC4	(84,1)	84	10164
Softmax	(10,1)	10	850

Why Convolutions?

- Parameter sharing
  - If we made FC from images, could have 3000 units in one layer connected to 5000 in the next. Instead we just have ~150 in a convolutional layer.
  - A feature detector useful in one part of the image is probably useful in other parts - in this way we share parameters over different positions in the input. True for low and high level features.
- Sparsity of connections
  - In each layer each output value depends only on a small number of inputs. A single value in an output matrix is just a collection of say 9 inputs.
  - Translation invariance - if we shifted pixels over a bit, CNN is still pretty robust, since we are looking locally and applying the same filter over and over.

Module 2 - Deep Convolutional Model Case Studies

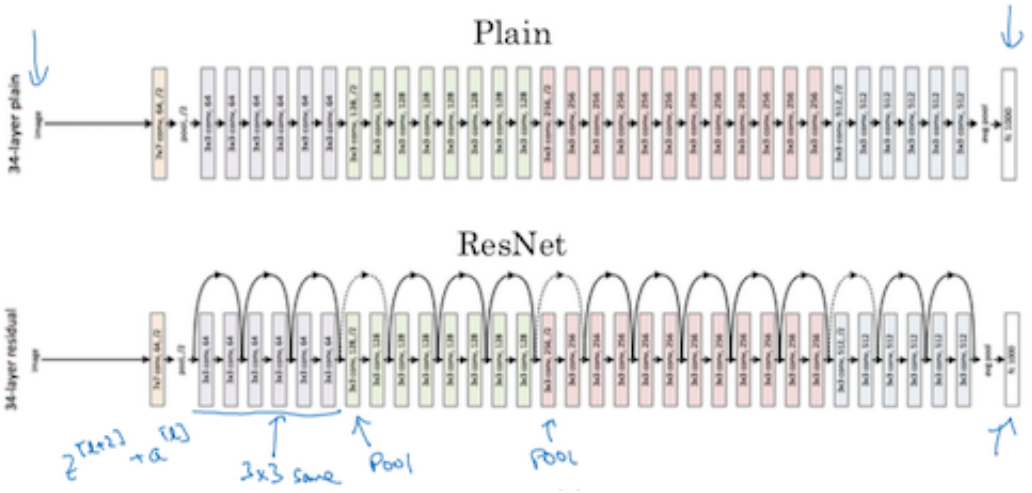
Classic Networks

- LeNet-5
  - Goal was to recognize handwritten digits, grayscale images 32 x 32 x 1
  - Conv: Use 6 5x5 filters, s = 1 -> 28 x 28 x 6
  - Avg Pool: f = 2, s = 2, output: 14 x 14 x 6
  - Conv: 16 filters that are 5 x 5. Output: 10 x 10 x 16
  - Avg Pool: f = 2, s = 2, Output: 5 x 5 x 16
  - FC: 120 -> FC: 84 -> y_hat with 10 possible values (would use softmax today)
  - Small by modern standards - 60k parameters.
  - Notice the height and width tend to decrease over layers (no padding) but channels increase. Also structure of conv, pool, conv, pool, fc, fc, output is a classic architecture.

- AlexNet
    - Inputs 227 x 227 x 3 images
    - Conv: 11 x 11 filters, s = 4. Ouput: 55 x 55 x 96
    - Max Pool: f = 3, s = 2
    - More Conv and Pool with same parameters. Gets down to a 6 x 6 x 256 -> 9216 FC nodes. Ends in a softmax
    - Similar to LeNet but much bigger. This one has 60m parameters. Also used ReLU activation which helped a lot.
- VGG-16
    - Conv layers f = 3 and s = 1, same convolutions. Number of channels doubles at each convolutional step, each pool halves the size of the image. End with a squat, deep 7 x 7 x 52 before FC 4096.
    - Max Pool layers f = 2, s = 2
    - The layers get longer as number of filters increase, since this determines the third dimension. The pooling layers cause the image to shrink by 2 at each pool, but conv keep dimensions the same except for the third dimension.
    - Has around 138m parameters. Roughly doubling number of filters at each conv layer.

ResNets

- Residual block - instead of running an output of l through the next linear -> ReLU -> linear block we give $a^{[l]}$ a shortcut. Now $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$
- Injected after the linear part but before the ReLU part. Using residual blocks allows you to train much deeper networks
- We add these skip connections to a regular network to get many residual blocks stuck together.
- Usually, training error has a u shape vs the number of layers in reality, despite the monotonic decline that is expected from theory. With a resnet, we get the proper monotonic decline
- The skip connections helps with vanishing and exploding gradient problems.
- Note $a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(w^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$ - $w^{[l+2]} = 0$, that whole term goes away and you just get $a^{[l]}$. Easy to get $a^{[l+2]} = a^{[l]}$, then adding those extra layers between them leaves a fallback value to learn. If the hidden units actually learn something useful, then they can improve on $a^{[l]}$ but shouldn't hurt if they cannot.
- Also note that we assume $z^{[l+2]}, a^{[l]}$ have same dimensions - so usually keep the dimensions constant in the res block so this works. If you changed dimensions could add $W_s a^{[l]}$ instead that transforms the dimensions correctly.
- Can add to a CNN with 3 x 3 same conv layers to preserve dimensions. Whenever we have pooling layers, need to make the dimension adjustment.



-

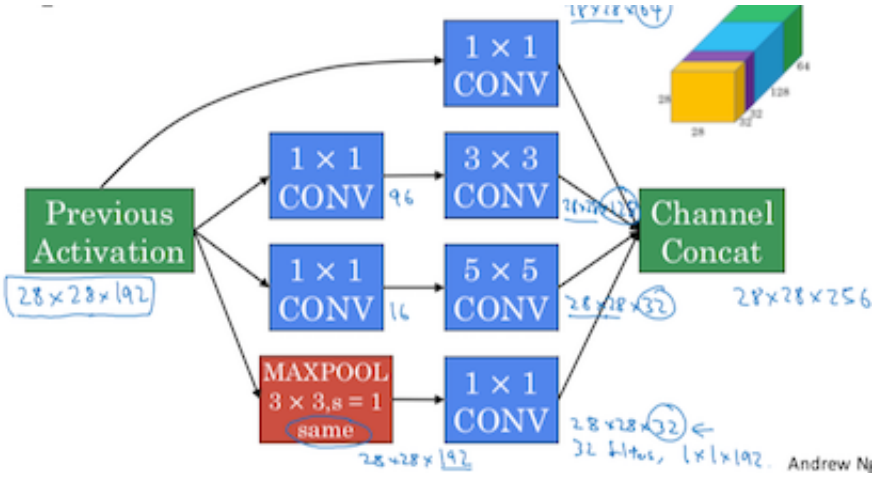
1x1 Convolutions

- A 1x1 filter on 6 x 6 x 1 image ends up just scaling the original image by the number in the filter. But if you have a 6 x 6 x 32, your filter is 1 x 1 x 32 to get 6 x 6 x (# of filters) output. So it multiplies each of 32 channels by a scalar, summing and applying a ReLU.
- With multiple filters, rebuilding different slices. Like a FC network, applies to each position. Also called “Network in Network”
- Why is this useful? To shrink H or W just pool. But to shrink the # of channels, use y filters that are 1 x 1. Then the output of the image will be H x W x y. If we keep the number of channels the same, then we are just adding some non-linearity that allows for learning a more complex function.

Inception Network

- Given a 28 x 28 x 192 volume. Use 1 x 1 convolution to output 28 x 28 x 64. But what if you also want to try a 3 x 3? We can do both and stack them - the H and W stay the same but can change the channel dimension.
- Can max pool also, but need to use pooling to keep the 28 x 28 dimensions.
- Input some volume, and get any # of channels as the output without having to pick any filter size - just do them all and stack.
- However the computational cost does increase. Just implementing the 5 x 5, same, 32 filter: have 32 filters, each 5 x 5 x 192 and you need to compute 28 x 28 x 32 to get the output - this is 120 m computations.
- We can reduce this using a 1 x 1 convolution: go from 28 x 28 x 192 -> 28 x 28 x 16 -> use 5 x 5 filter -> 28 x 28 x 32. The 1x1 is called a “bottle neck” layer.
    - First computation is 28 x 28 x 16 x 192 = 2.4 m, and second part is 28 x 28 x 32 x 5 x 5 x 16 = 10m. So now we have roughly 1/10 of the computations needed before.
- With max pooling, we add a same parameter to ensure we add padding to get the same dimensions. Then need a 1x1 to shrink the number of channels.
- Once we feed the activations through a 1x1, the larger filters we desire (3x3, 5x5 etc) after the 1x1s, we can just concat the whole stack again. Then can repeat these inception blocks to create a full inception network.
- Finally, can add some sidebranches that try to take a hidden layer and make a prediction through a softmax. Helps ensure the features computated even in the hidden units aren’t too bad - can have a

regularizing effect.

- 

Transfer Learning

- Can download weights that someone else has trained on their architecture. Then use this as the initializiation for your problem and get much faster training.
- Get rid of the last softmax layer, then attach your own with the categories you care about. Freeze the early hidden layers and just train the softmax layer.
- This can help get good performance even with a small dataset. Should be easy to do inside the DL framework using.
- Could also freeze some of the early layers and continue training the last few layers. Either use the last few weights as initialization or throw those away and start over. Depends on how much data you have - the more you have the more layers you can train.
- With a lot of data, can train the whole NN, just leaving the original weights as initialization. Basically you should almost always do transfer learning unless you have a very large dataset.

Data Augmentation

- Mirroring on vertical axis. Random cropping, though some error since you can take bad crops. Rotation shearing, rotation, local warping.
- Color shifting - add to the RGB channels different distortions. Can draw the additions at random from some distribution. May use PCA color augmentation - find the colors that are most important.
- With a very large dataset, can have a CPU thread constantly loading images off the HD, then implement the distortion in the thread. The threads form a mini-batch and we pass it directly to the training

State of Computer Vision

- Data vs hand engineering - on a scale of lots of data to little data, image recognition is behind other tasks like speech recognition. Object detection is even further behind.
- When there is more data, we can have a simpler architecture, less hand engineering. With less data, we need more hand-engineering and hacks.
- The learning algorithm has two sources of knowledge - the labeled data and hand engineered features/ architectures. Without the data we push harder on the other. Transfer learning is also useful in this case.
- Tips for benchmarks / competitions
    - Ensembling - train several networks independently and average the outputs ($\hat{y}$). Good for

benchmarks but super costly for real production
    - Multi-crop at test time - run classifier on multiple versions of test images and average the results. 10-crop for example.
- Always consider open source architectures and papers and implementations / transfer learning weights

Module 3 - Detection Algorithms

Object Localization

- Localization - drawing a bounding box around a car in an image say.
- Detection: Can also have multiple detection for different objects in the photo.
- Typical ConvNet might output to a softmax with a predicted class for an image, say 4 possible outputs. To localize we change our NN to have more output units, four more numbers b_x, b_y, b_h, b_w that place a bounding box around the object that we classified. b_x, b_y indicate the midpoint with b_h, b_w determining the heigh and width of the bonding box.
- In total we need to output those 4 coordinates and a class label 1-4. Target label y is vector $y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$:
    - 1st component p_c is there any object? 1 for object detected, 0 for just background
    - 2nd - 5th component: bx, by, bw, bh
    - Last components: c1, c2, c3 - typical softmax, classifying a single object in the photo.
- When pc is zero - we do not care about the rest of the y vector, since we aren't bounding anything, no detected objects.
- Loss function $L(\hat{y}, y) = (\hat{y}_1 - y_1)^2 + \ldots + (\hat{y}_8 - y_8)^2$ if $y_1 = 1$
    - If $y_1 = p_c = 0$, then just need $L = (\hat{y}_1 - y_1)^2$, since we just care about the accuracy of p_c in this case.

Landmark Detection

- X and Y coordinates of important parts of an image you want the algo to recognize.
- Say you want the corner of people's eyes - just have additonal outputs for the coordinate of interest (x,y). If you want more points, $(l_{1x}, l_{1y}), \ldots$ for all points of interest.
- Network can tell us a large number of key points on recognizing a face. Ouput vector where first indicates if it is a face, then over 64 landmarks we would have an additional 128 data points output in the vector. This is important for face filters and special effects.
- We do need a labeled dataset for this task though. Often human intensive
- Pose detection works similarly. Need consistent labels across images, landmark 1 refers to the same part of the image across examples.

Object Detection - Sliding Windows

- Say for car detection, create labeled datset with closely cropped images of cars as X, and y indicates if

car or not. Then train a ConvNet that classifies if car or not for cropped images

- Then use sliding window detection - take a local region of a real photo, and have the convnet determine in car is in this region. Slide the window across the whole image and it will classify which windows have cars and which don't.
- We then repeat the sliding window with a larger window, again feed the windows to the convnet for car classification. Can continue with larger and larger windows. The hope that so long as a car is somewhere in the image, the convnet will recognize for some window size.
- Obviously this has a high computational cost. Using a coarse stride can reduce the number of windows but this might hurt performance of classification.
- People use to use linear methods for the classifier but now convnets make this process slower. Luckily we can fix this cost by using a convolutional implementation of the sliding windows

Convolutional Sliding Windows

- How to turn FC into convolutional layers? If we have 5x5x16 filters at the end of the conv net, we can look at this as a 1 x 1 x 400 output, then pass through to 1 x 1 x 400 layer, but this is equivalent to an FC layer. We can do this a couple of times then pass to a 1 x 1 x 4 layer, which corresponds to the 4 class softmax output.
- Using this is sliding windows, we start with a 14 x 14 x 3 image. We pass through the modified NN and we end with a 1 x 1 x 4.
- Our test image is 16 x 16 x 3. We could slide the 14 x 14 window around 4 times to cover the image, but this is highly duplicative. Doing this convolutionally, we just run the conv net using the 5 x 5 x 16 filters, and we shrink down to a 2 x 2 x 40 volume run through 1 x 1 filters. Finally get a 2 x 2 x 4. Each quadrant of the output corresponds to the upper / lower left/ right regions of the original image.
- We run the classifier once on the image and the computational expense is shared across regions. Adding larger images will correspond to larger outputs from the classifier conv net.
- Still have a problem that the bounding boxes will not be too accurate so still have to adjust for that.

Bounding Box Predictions

- YOLO - you only look once, can give us better bounding boxes
- Place grid on the image, medium fine. Apply to localization algorithm to each section of the grid.
- For each grid cell specify label y where like previously, y is an 8 dimensional vector $y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$. Say top left grid cell is sky, there is no label and we do not care about the rest of the values.
- Taking an object that spans multiple grid cells, YOLO takes the midpoint of the object and assigns it to

the grid cell that lives in. Then y indicates there is an object, $y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$ say indicating the car is in that grid cell. With 9 grid cells on a 100 x 100 image, target output is 3 x 3 x 8, since each of the 3 x 3 grid cells has one of these 8 dimensional y vectors. This is the depth of the output volume.

- 100 x 100 x 3 -> conv -> pool -> etc -> 3 x 3 x 8. Use backprop to train network to take any image X and output to this volume
- First run forward prop, then for each 3 x 3 output, read the pc, and if object, have the coordinates for the object to backprop. If multiple objects per cell might make grid finer.
- There is a lot of shared computation over all these steps since convolution
- Bounding box encoding - set coordinates for grid cell, then midpoint is relative to the grid cell as is the height and width. If grid is unit square, then the midpoint is between 0 and 1, but height and width could be outside of grid dimensions if object spills out into other grid cells.

Intersection Over Union

- How do you tell if your object detection algo is working well?
- Intersection over union (IOU) computes this statistic of 2 bounding boxes - the ideal and the computed. Compute size of intersection of boxes / size of union.
- Judged "correct" if IOU $\geq$ 0.5 typically. Could be more stringent

Non-Max Suppression

- Might detect an object multiple times - this ensures it is only detected once
- A fine grid over the image, while cars only have one midpoint, in practice we are running a classification and localization for each grid cell and algorithm may assign midpoints to multiple grid cells
- Over 19 x 19 grid, want to clean up the multiple detections per car. Looks at probabilities Pc of detection for each grid cell. Takes the highest confidence Pc.
- Then takes the remaining bounding boxes and suppresses their detections. Just output the maximal Pc detection
- On 19 x 19 grid, get 19 x 19 x 8 output volume. Say we only have cars to detect, so y is length 5 instead.
    - Discard all boxes Pc below 0.6 say.
    - While remaining bounding boxes,
        - pick the box with highest Pc and output that as prediction
        - Discard any remaining box with IOU > 0.5 with the box output in the prior step

Anchor Boxes

- What if a grid cell wants to detect multiple objects, say human in front of car
- Predefine two different shapes - anchor box shapes. In general we could use more anchor boxes.

- Defined the class label to be $y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$ of double length - first 8 correspond to box 1 and 2nd 8 correspond to box 2. Then use the first to encode a person and second for car if those boxes are more similar to each of those shapes.
- Each object in training image is assigned to grid cell that contains the objects midpoint and anchor box for the grid cell with highest IOU. (Grid cell, anchor box pair) assigned to object.
- Output y is 3 x 3 x 16 for a 9 square grid now.
- Algorithm still weak if two objects that would be assigned same anchor box are in same grid cell, but happens rarely.

YOLO Algorithm

- Trying to detect peds, cars, motorcycles. y is then 3 x 3 x (2 x 8) (2 for anchors, 8 is 5 (pc, b's) + # of classes). Then for each grid cell create the y vector. Once object detected, we have bounding box coordinates and class label in C part of vector.
- Making a prediction - passes through early portions of algorithms until grid portion. Then for each grid, it outputs a y vector. Then run through non-max suppression.
- For each grid cell, say get 2 predicted bounding boxes, and some exceed a single grid cell. Then run the algorithm as described above for each class to make final predictions (here would run 3 times).

Region Proposals

- R-CNN - regions of CNNs. There are a lot of non interesting grids cells
- Instead of running on all windows, run on just a few. First run a segmentation algorithm to determine separation between objects. Then run the classifier on the segmented blobs
- Find maybe 2000 blobs then place bounding boxes around those. Can be a much smaller number of objects then running over whole image.

Module 4 - Conv Applications

Face Recognition

- Verification - input image, name / id and ouput whether the input image is that of the claimed person
- Recognition - has a database of k persons, get an input image, output id if the image is any of the k persons. If you have 100 persons may need a very high accuracy verification system to make this work. Verification is a building block of the recognition system.

One Shot Learning

- Given a single image or example, need to recognize a person again
- Could try approach of taking an image -> conv net -> softmax, but this is not robust for training, and would have to change the softmax if more people join
- Instead learn a similarity function. $\mathrm{d}(\mathrm{img1}, \mathrm{img2}) =$ degree of difference between images. If d < tau then same else different persons.

- Compare test image against the images in the database, get a number of difference scores. Take the lowest score if below tau threshold or return none at all if differences are all too large

Siamese Network

- Last layer of conv net - FC 128, let's call $f(x^{(1)})$, an encoding of $x^{(1)}$.
- Feed a second image through the network and get a different 128 length vector at the end $f(x^{(2)})$. If these encodings are good representations of these images, then $d\left(x^{(1)}, x^{(2)}\right) = \left\|f\left(x^{(1)}\right) - f\left(x^{(2)}\right)\right\|_2^2$.
- Uses the same weights while working in tandem on two different input vectors to compute comparable output vectors. Often one of the output vectors is precomputed, thus forming a baseline against which the other output vector is compared.
- How do we train this network. Parameters of NN define an encoding $f\left(x^{(i)}\right)$, and learn parameters st for $x^{(i)}, x^{(j)}$, $\left\|\mathrm{f}\left(x^{(i)}\right) - \mathrm{f}\left(x^{(j)}\right)\right\|^2$ is small for same persons and large for different persons

Triplet Loss

- We compare pairs of images. Want same persons to have similar encodings and different to have different encodings.
- Want difference between anchor and pos example to be small and anchor vs neg example to be large. $\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 \leqslant 0$. But this could be trivially satisfied by setting everything to 0. To prevent this solution, modify the objective to be $\leq 0 - \alpha$, so $$\|f(A)-f(P)\|^{2} - \|f(A)-f(N)\|^{2} + \alpha\leqslant 0 $. Equivalent to a margin in SVMs (called margin parameter here).
- Forces the A-N difference to be much bigger than A-P instead of marginally different.
- Given three images A,P,N, we define the triplet loss $L(A, P, N) = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\right)$. Tries to send the left portion to 0. Then cost over all examples $J = \sum_{i=1}^{m} L\left(A^{(i)}, p^{(i)}, N^{(i)}\right)$. Say training set is 10k pictures of 1k different persons. We need multiple pictures of a person to train this system.
- Then once trained this can be a one-shot task.
- How do we choose triplets? If chosen at random, pretty easy to satisfy the $d(A, P) + \alpha \leq d(A, N)$ since most pictures will easily meet this criteria. You want to choose hard triplets, where A-P close to A-N. Increases the computational efficiency of your algorithm since then GD will have to actually do work to distinguish them.

Face Verification and Binary Classification

- Another approach to this problem. Take the encodings from the siamese networks and feed to binary classification output. 1 indicates same person 0 different.
- $y_{hat} = \sigma\left(\sum_{k=1}^{128} w_k \left|f\left(x^{(i)}\right)_k - f\left(x^{(j)}\right)_k\right| + b\right)$. Element wise features we run through logistic regression that we train for different or same persons.
- Could also use chi-squared similarity or other metrics.
- Computational trick: instead of computing the embedding every time, could precompute encodings so we do not need to store the images.

Neural Style Transfer

- Take content image and want to transfer style from another image.
- What are deep conv nets really learning? Want to visualize what the hidden units at different layers are learning
- Pick a unit in layer 1, find the nine image patches that max the unit's activation. Might see a diagonal

edge in all of the images, seems like the unit is detecting that specific edge
- We can go through other hidden units with different low level features they follow. Could be different colors, edges, etc. Relatively simple features at unit 1
- As we go into the deeper layers, image patches are larger portions of the images since H and W shrunk. The later units are activated first for more complex shapes and pattersn and even later for specific human recognizable objects. Say people, car wheels, dogs, legs.

Cost Function for Style Transfer

- Overall cost function $J(G) = \alpha J_{\text{content}}\ (C, G) + \beta J_{\text{style}}\ (S, G)$
- Initiate G randomly, say 100 x 100 x 3 random image
- Use gradient descent to minimize J(G) - G := G - dJ(G)/dG

Content Portion

- Say you use hidden layer l to compute content cost. Usually somewhere in the middle of the layers. Use a pretrained convnet like VGG
- Let $a^{[l](C)}$, $a^{[l](G)}$ be activation of layer l on the images
- If $a^{[l](C)}$, $a^{[l](G)}$ similar both images have similar content. Use $J_{content}(C, G) = \frac{1}{2}\left\|a^{[\ell](c)} - a^{[\ell](G)}\right\|^2$. Element wise sum of squared differences between these activations.

Style Portion

- Say using layer l to define the style of the image. Style is the correlation between activations across channels. Given block nH nW nc, say nc = 5. Look at red and yellow channels, look across positions between these channels and compare the activations and determine correlation.
- For high level features, like comparing how a certain pattern occurs with a certain color.
- Style (Gram) matrix: let $\mathrm{a}_{i,j,k}^{[l]} = \text{activation at } (i, j, k). G^{[l]} \text{ is } \mathrm{n}_{\mathrm{C}}^{[l]} \times \mathrm{n}_{\mathrm{C}}^{[l]}$ for i height, j width, k channel.
- $G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a_{i,j,k}^{[l](G)} a_{i,j,k'}^{[l](G)} \cdot G_{kk'}^{[l](S)} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a_{i,j,k}^{[l](S)} a_{i,j,k'}^{[l](S)}$ for generated image and style image. Measures the correlation between channels k and k'.
- Style Cost Function: $J_{\text{style}}^{|l|}(S, G) = \frac{1}{\left(2 n_H^{[l]} n_W^{[l]} n_C^{[l]}\right)^2} \sum_k \sum_{k'} \left(G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)}\right)^2$
- Get more visually pleasing results if we take a weighting over multiple layers: $J_{\text{style}}^{|l|}(S, G) = sum_l \lambda^{[l]} J_{style}^{[l]}(S, G)$

1D and 3D Generalizations

- Say you have EKG data, a spiky 1D time series. Here we might use a 1D filter that has 5 different value
- 14 x 1 dimensional vector convolved with 5 x 1 filter 16 times gives us 10 x 16. Same idea as in 2D.
- 3D data like a CT scan - has a slice for each level of the brain. Conv net to 3D volume with 3D filter. A 14 x 14 x 14 (x 1 channel) input volume convolved with 5 x 5 x 5 x 1 filter, 16 filters gives us a 10 x 10 x 10 x 16. Then say we apply again with 32 filters -> 6 x 6 x 6 x 32
- Could also use with movie data, where the third dimension is time.

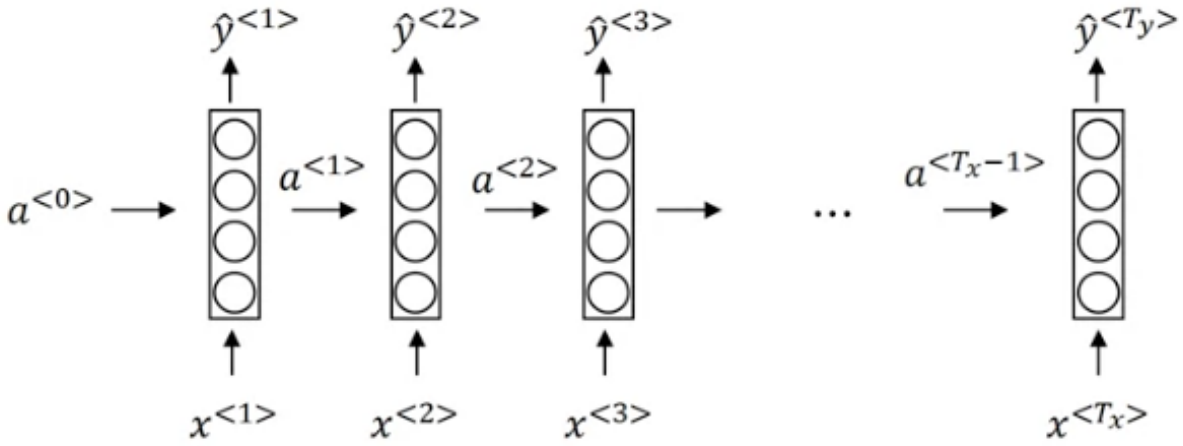
C5 - Sequence Models

Module 1 - Recurrent Neural Networks

Notation

- Speech recognition - x audio clip and y output words. Music generation. Sentiment classification, DNA sequencing. Machine Translation
- Say have sequence model to find names in sentence x - Named entity recognition
- Model output y, [1,1,1,0,0,0,1,0] etc, word is name or not a name. To refer to positions we will use $x^{\langle 1\rangle}, x^{\langle 2\rangle}, \ldots x^{\langle 9\rangle}$, similarly for y. Let $T_x = 9, T_y = 9$, the number of words in input and output
- The t'th element of example i: $x^{(i)<t>}$, lenth of training example input $T_x^{(i)} = 9$
- Vocabulary - need a mapping of all words in our dataset. Vector of words [a, aaron, ..., zulu], each assigned their numerical index. Dictionary sizes of 30-50k is common, but also much larger.
- Use one hot encoding for each word in the sentence -> eg. $x^{<1>} = [0001...0]$. Each vector the length of the vocab, with a single 1 indicating the word in the vocab we are looking at.
- What if we encounter word not in vocab? We can use a placeholder

RNN

- Why can't we use a standard network? Inputs and outputs can be different lengths in different examples. Doesn't share features learned across different positions of text - don't want it just to learn Harry in position 1 is a person's name, need more generalization.
- Also using better representations reduces number of needed parameters
- 

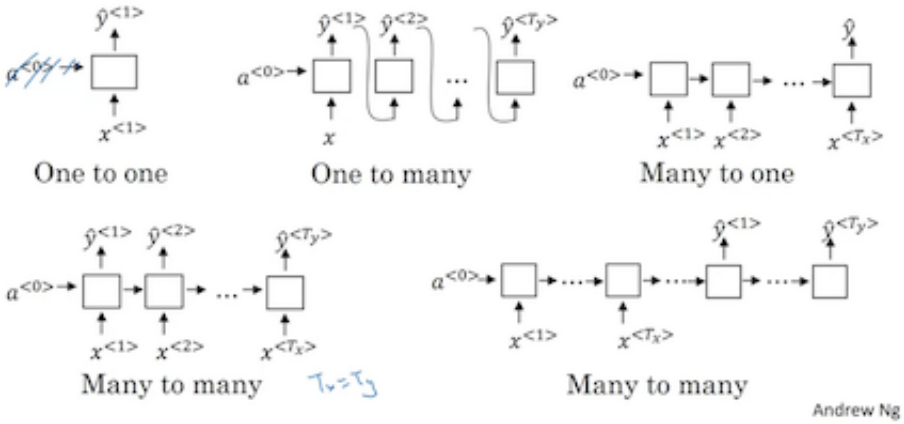
- With RNN, we take $x^{<1>}$ and input into the network, leading to prediction $y^{<1>}$. Then we feed in $x^{<2>}$, and activation is used from prior step $a^{<1>}$ to produce $y^{<2>}$. At each time step, the RNN passes its activation to the next time step to use.
- Can initialize randomly or with all zeros for $a^{<0>}$ (0's more common).
- We will have parameters W_{xa}, W_{aa}, W_{ya} that govern relationships.
- One weakness of this RNN is we are only using information before the current word - we also want to use information from after the current step in the sequence. We will address this with bidirectional RNNs.
- Forward Prop: $a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$ and $y^{<1>} = g(W_{ya}a^{<1>} + b_y)$. Here we use W_{ax} to indicate this will compute some a quantity from an x, while W_{ya} computes a y quantity from an a quantity.
- Often use tanh for activation, sometimes ReLU. Y activation depends on the type of output we are looking for.
- Generally: $a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$ and $y^{<t>} = g(W_{ya}a^{<t>} + b_y)$
- We can simplify these two equations with better notation: let $W_{aa}a^{<t-1>} + W_{ax}x^{<t>} = W_a[a^{<t-1>}, x^{<t>}]$, by stacking $[W_{aa}W_{ax}]$ (combine width and both matrices are of same height).

- $[a^{<t-1>}, x^{<t>}]$ indicates we are stacking these vectors vertically - say first 100 terms are a, next 10k are x, so when we do our matrix multiplcation we get back our original equation.

Backprop Through Time

- At each forward step, feed in x, W, b, and prior a to calculate current a. It outputs a y for our step using W_y and b_y.
- Loss $L^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right) = -y^{<t>}log(\hat{y}^{<t>}) - (1-y^{<t>})log(1-\hat{y}^{<t>})$
- Then total loss is sum over all time steps $L = \sum_{t=1}^{T_y} L^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right)$
- Then our backprop step is passing our gradients back through the steps that created our total loss. We go from the last step in our sequence to the first.

Different Types of RNNs

- T_x does not have to equal T_y. In machine translation, we have different number of words to say the same things in different languages.
- Many to many architecture is what we saw before - input has many sequences as does the output.
- Many to one architecture - For sentiment classification, x is text, y is a number 1-5. Then the RNN can read in the entire sentence and output a single y
- One to many archtecture - music generation, with goal to have network output a set of notes. Input x and have RNN output first value of y, then with no further inputs, output subsequent y values. Often drawn as output feeding back into the next sequence instead of a new x.
- Many to many with different lengths - Encoder: first portion takes in X's in a sequence, passing activations to next steps, but produces no output. Then we pass the output of the encoder to the decoder, which takes in no additional input but produces a sequence of y's.
- 

Language Model and Sequence Generation

- Given any sentence, what is the probability of that sentence occurring given an input sentence?
- The language model estimates the probability of the sequence of words $P(y^{<1>}, \ldots, y^{<T_y>})$
- Train on a large corpus of text.
- First, we tokenize the sentence by formulating our vocabulary. Often also want to model when setences end, by adding a token $< eos >$ to the end.
- Could also treat punctuation as tokens in the vocab. If some words in training set are not in the vocab, then you can replace words with $< unk >$.
- At time 0, we compute some activation $a^{<1>}$, starting with $a^{<0>} = x^{<1>} = 0$. The first activation uses a softmax to make $\hat{y}^{<1>}$ - what is the chance that the first word is any given word in your vocabulary. With

a 10k vocab, we have a 10k softmax.

- Then we take our next step, trying to find the probability of what is the second word, but we give it $y^{<1>}$ - the actual first word in the dataset, so $y^{<1>} = x^{<2>}$ (since x init to 0). Then $\hat{y}^{<2>} = P(y^{<2>}|y^{<1>})$.
- To predict the 3rd word, we give the activation the prior word, etc. At the end, we feed in the last actual word, and hopefully the model predicts the next word is the EOS token.
- Loss function $\mathcal{L}\left(\hat{y}^{<t>}, y^{<t>}\right) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$. Overall loss $\mathcal{L} = \sum_t \mathcal{L}^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right)$.
- Probability of a sentence: $P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>})P(y^{<2>}|y^{<1>})P(y^{<3>}|y^{<1>}, y^{<2>})$

Sampling Novel Sequences

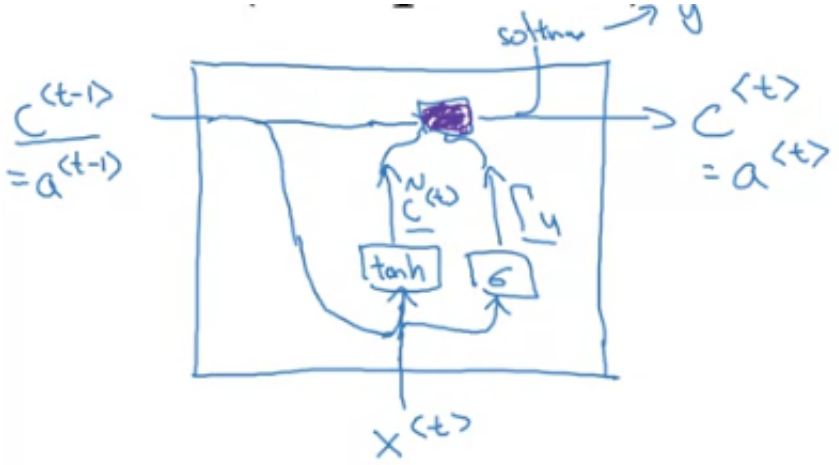
- Use your usual first inputs of 0, and get an output softmax distribution. Whats the chance that the first word is x over whole vocab. Then we can run `np.random.choice` to sample a word from this probability distribution.
- Then pass the word we sampled into the input for the second activation. We again sample the output softmax for the second sequence and pass it to the next activation as input. You could keep sampling until EOS is hit from sampling or decide a fixed length to try.
- If you never want a token, could reject any sample that selects this and take a new sample.
- Could also build a character level RNN - vocab is just alphabet and characters (could use upper and lower case too). Then the sequence will just be individual characters.
- Can use this for sequence generation and novel texts.

Vanishing Gradients with RNN's

- Whether the noun is singular determines the form of a verb. There can be long term dependencies within a sentence.
- But with very deep NN's we have a problem of vanishing gradients, preventing learning in the early layers - ie. the early words in the sentence. Activations are affected only be nearby members of the sequence.
- Exploding gradients tends to be less of problem with RNNs, but can happen and cause numerical overflow
- Can apply gradient clipping in RNN - threshold the values against some fixed number. Pretty robust solution to solving exploding gradients, but vanishing graidents much harder to solve.

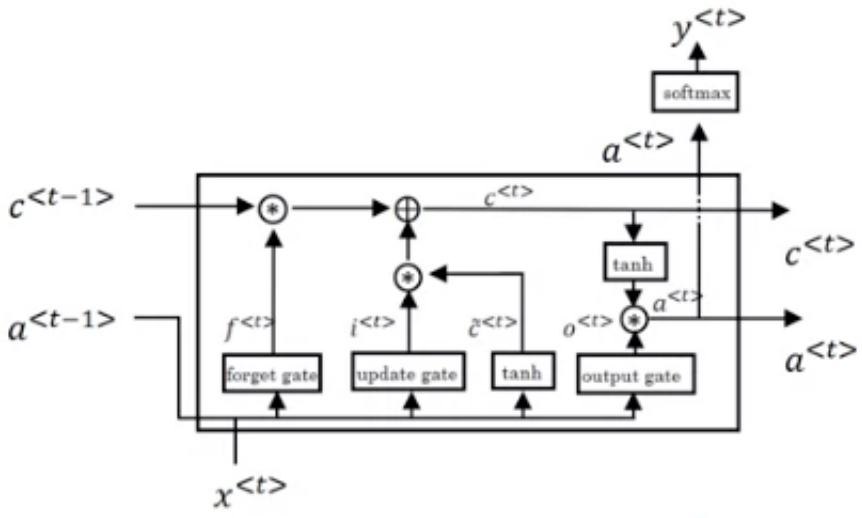
Gated Recurrent Units

- The cat, which already ate ..., was full. Sentence we are trying to get the GRU to recognize cat is singular, was should be singular
- C = memory cell - have $c^{<t>} = a^{<t>}$
- At every time step, consider overwriting C with $\tilde{c}^{<t>} = tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$
- Then it also has a gate: Γ_u, evaluated between 0 and 1. $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$. Here gamma will be far out on the sigmoid function so mostly 0 or 1.
- We have a candidate update and the gate for a decision function. Then can think $c^{<t>}$ at cat is set to 0 or 1 for plural or singular, then the GRU memorize this value until "was" comes along. But once we have used the gate, we can forget it and update it.
- The key equation for updating is $c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>}$.
- Essentially this sets $\Gamma_u = 1$ at cat, then for the values between "cat" and "was" $\Gamma_u = 0$ indicating no updates should be made.

- 

- Because gamma is 0 across many time stamps, this helps with the vanishing gradient problem since most of the time $c^{<t>} = c^{<t-1>}$
- We have $c^{<t>}, c^{<t-1>}, \Gamma_u$ are same dimensions, so note that the multiplication above is element-wise. It tells which dimensions should be updated at each step - the GRU will update some values at each step and leave others unchanged in the vector.
- In practice, we can add another gate: Γ_r for relevance - how relevant is $c^{<t-1>}$ to updating $c^{<t>}$. Then $\tilde{c}^{<t>} = \tanh\left(W_c \left[\Gamma_r * c^{<t-1>}, x^{<t>}\right] + b_c\right)$ for $\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$

LSTM

- Also a type of unit to learn long term effects. Often seen as more powerful / better.
- We now have a separate gate Γ_f for explicitly forgetting prior values. Separate update and forget gates. Additionally an output gate Γ_o
- Equations:
$$\tilde{c}^{<t>} = \tanh\left(W_c \left[a^{<t-1>}, x^{<t>}\right] + b_c\right)$$
$$\Gamma_u = \sigma\left(W_u \left[a^{<t-1>}, x^{<t>}\right] + b_u\right)$$
$$\Gamma_f = \sigma\left(W_r \left[a^{<t-1>}, x^{<t>}\right] + b_f\right)$$
$$\Gamma_o = \sigma\left(W_o \left[a^{<t-1>}, x^{<t>}\right] + b_o\right)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * tanh(c^{<t>})$$
- 

- If you connect these units in parallel - output from 1 becomes the input for another. As long as you set

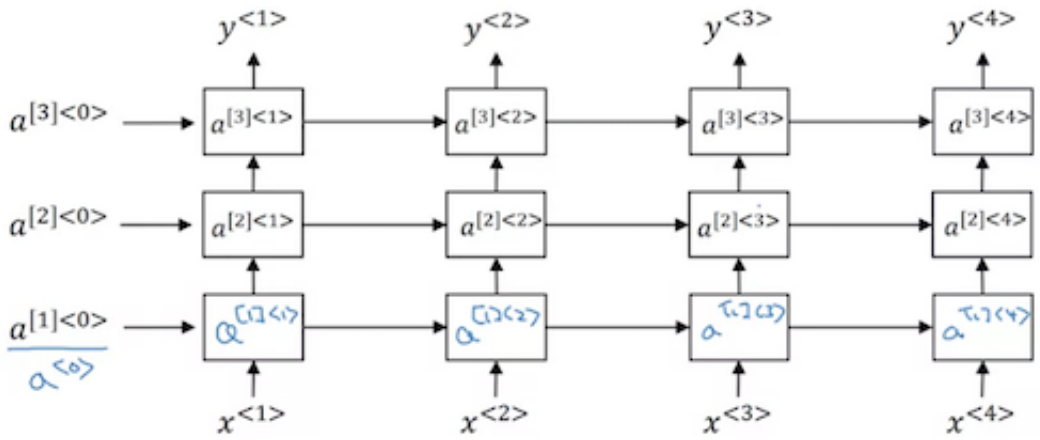
your update/forget gates correctly, can simply pass a single value straight through all of the units, so it is very good at maintaining a gradient.
- Using one vs the other - GRUs are a more recent invention, derived as a simplification. Advantage of GRU is you can build a much bigger network, but is less powerful than the LSTM.

Bidirectional RNNs

- Take information from earlier and later in the sequence. Information presented can work for GRU/LSTM/Standard RNN blocks
- Each X feeds into an activation, activations passed to the next step, y output produced at each step. So far all the same.
- We will add a backward connection, where we have a reserve activation connection.
- The backward connection activations are connected to each other backwards in time. X passes into both forward and backward activations, and both activations feed into the y hat produced by each sequence.
- Then $\hat{y}^{<t>} = g(W_y[a_f^{<t>}, a_b^{<t>}] + b_y)$. Predictions at any time point takes information from past and future.
- BRNN with LSTM blocks is a common structure for these types of networks.
- You do need the entire set of data - ie. the end of the sentence - to make a prediction. May not be best for real time processing of speech.

Deep RNNs

- Can stack layers of RNNs together to build deeper models
- Now we specify the layer $a^{[\ell]<t>}$.



- $a^{[2]<3>}$ has two inputs - from the left and from the bottom. Then $a^{[2]<3>} = g\left(W_a^{[2]}\left[a^{[2]<2>}, a^{[1]<3>}\right] + b_a^{[2]}\right)$
- We don't usually stack too many of these, since they are already quite complex / computationally expensive. The blocks can be RNN, GRU, LSTM.

Module 2 - NLP and Word Embeddings

Word Representation

- So far we have represented words using a vocab + 1 hot vectors. We represent one word as a single 1 in a vector of length of the vocab.

- But it treats each word as a thing unto itself, prevents generalization of words. If we trained a network to see orange juice, it cannot generalize to apple juice. The inner product between any two one hot vectors is 0, and the euclidean distance is also the same between any pair
- Featurized representation: word embedding. Have a matrix of values relating words to each other. Could have row of gender, age, royal, and the words man, woman, king, queen, will have different relationships to those rows.
- Using this word representation, we now see correlation between orange and apple. Therefore the learning algorithm can figure out that apple and orange juice are likely both real things.
- We often take a large dimensional embedding and reduce it to a 2D space for visualization - t-SNE algorithm. Shows clusters of related words.
- Embeddings, since the word is embedded somewhere in high dimensional space.

Using Word Embeddings

- Training a model with embeddings for our named entity task, we can figure out names more easily in related sentences.
- If we have learned word embeddings, then even for words outside of our vocab, the algorithm can still see that a new sentence is close to one it has learned. Embeddings can be huge 1B - 100B words.
- Essentially transfer learning - taking pretrained relationships transferred to your smaller application dataset.
- Now can use lower dimensional vectors - instead 0f 10k one hot sparse vector, have 300 length vector that is dense.
- Similar to the face encodings in the siamese networks we learned for face recognition - encoding and embeddings are quite similar.

Properties of Word Embeddings

- Embeddings can help with reasoning by analogy. If we posed the question Man:Woman as King:?, our embeddings can provide an answer
- Vectors addition $e_{man} - e_{woman} \approx [2, 0, 0...0]$. Similar for King - Queen. It captures that the main difference between man and woman is gender, and king and queen also differ in the same dimension. So in the analogy can take the difference of our provided anology, and try to find a difference that is similar in the embeddings.
- Turning this into an algorithm - Find word w s.t. $argmax_w \ sim(e_w, e_{king} - e_{man} + e_{woman})$.
- Most common similarity function used is cosine similarity: $sim(u, v) = \langle u, v \rangle = \frac{u^T v}{\|u\|_2 \|v\|_2}$. Note this is actually equal to the cosine of the angle between the vectors.

Embedding Matrix

- Take our vocab to an embedding matrix of size 300 x 10k. The columns will be each word in the vocab. We take our one hot vectors for a word O and multiply it by the embedding matrix E: E0*{word1} = (300 x 10k) times (10k, 1) = e*{word1}. Our goal is to learn an embedding matrix E.
- In practice, this isn't efficient way to do this. We use specialized functions instead.

Learning Word Embeddings

- Have a sentence and want to predict the last word. Each word has an index in the vocab.
- Construct one hot vector for each word - 10k dimensional vector O_{4343}
- Then take embedding matrix times one hot $np.dot(E, O_{4343}) = e_{4343}$. Perform for each word. These e

are 300D embedding vectors.
- We can feed these into a NN, to a softmax with 10k outputs corresponding to the vocab.
- Often have a sliding window to just look at the previous, say 4, words.
- Parameters of the model is the matrix E and the weights of the NN layer and softmax layer. Then can just use backprop to train the NN and it learns pretty good word embeddings.
- Target is last word, context is last 4 words. Could experiment with other contexts - words on the left and right say, nearby 1 word, last 1 word. These are all good for learning embeddings over a language model.

Word2Vec

- More efficient way to learn embeddings.
- Skip-grams - come up with context to target pairs to create our supervised learning problem. Randomly pick a word to be a context word, randomly pick another word within some window to be the target.
- Learn mapping from some context c to target t. Each word has a vocab index.
- Take one hot $O_c \rightarrow E \rightarrow e_c$, embedding for the context.
- Then $e_c \rightarrow softmax \rightarrow \hat{y}$. Softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} \theta_t^T e_c}$, where θ_t is the parameter associated with output t.
- Then loss $L(\hat{y}, y) = -\sum_{i=1}^{10,000} y_i log(\hat{y_i})$. Output y hat is a 10k dimensional vector with probabilities within our vocab.
- The problem with this model is computational speed - we are summing over our entire vocabulary embedding vectors. Often sped up through hierarchical softmax - binary classifier that creates a tree telling up if word is in first half or second half of list - tree of classifiers. We will also negative sampling helps speed this up with a different method.
- How to sample context c? If you sample uniformly random, there are some words that appear very frequently while others are almost never selected. Doesn't help you update the embeddings for all words. So instead other samplings are used.

Negative Sampling

- Skip gram more efficiently.
- Given a pair of words, predict whether they are a context target pair. Eg positive ex is orange and juice, negative is orange and king.
- To generate pos examples, sample nearby words from our sentence. For negative, sample one from sentence, one random word from the dictionary.
- Create supervised learning problem, pair of words input, target is predict are they related or randomly chosen.
- We pick the same word and for k examples of that word, we pair it up with other words.
- Logistic regression model $P(y = 1|c, t) = \sigma(\theta_t^T e_c)$. For k negative examples, we have k:1 negative:positive example ratio. Input $O_c \rightarrow E \rightarrow e_c$ leading to 10k binary logistic regression problems, but we will only train 5 of them on a given iteration, 1 positive, k negative.
- Sampling the negative examples: choosen something in between in proportion to word freq and uniformly at random: $p(w_i) = \frac{f(w_i)^{3/4}}{\sum_{i=i}^{10000} f(w_i)^{3/4}}$
- As always you can download open source word vectors to get started more quickly without learning your own.

GloVe

- Global vectors for word representation
- Let x_{ij} = # of times the word j appears in the context of i. (i and j playing the role of t and c respectively).
- Symmetric relationship if we using both directions in the context (+- some number of words)
- Minimize $\sum_{i=1}^{10,000}\sum_{j=1}^{10,000} f(x_{ij})(\theta_i^T e_j + b_i + b_j - logx_{ij})^2$ - how related are words i and j measured by the co-occurrance. Solve using gradient descent. Weighting $f(x_{ij}) = 0$ if $x_{ij} = 0$ to prevent log of 0.
- Stop words - the very frequent non content words. Can use the weighting factor to modulate the weighting of stop words and infrequent words. Finally θ_i, e_j are symmetric, so $e_w^{final} = (e_w + \theta_w)/2$.
- A nice simplification of earlier algorithms.
- Learning the GloVe embeddings does not guarantee that the embeddings are interpretable.

Sentiment Classification

- Input sentence, output star rating.
- Simple Model: Given some words, create the one hot vector, multiply by embedding matrix E (from BERT say), to get e_{word}.
- Take the e vectors, and average them (or sum) to get a 300D vector -> softmax -> yHat. Softmax gives the star rating 1-5.
- Works for long or short reviews since we are aggregating over all the words. This works pretty well, but it ignores word order.
- Instead, use an RNN:
    - Take one hot vectors -> E -> embedding vecots e. Feed the embedding vectors into the RNN. Many to one - with prediction y hat the final output. Much better at taking word sequence into account.

Debiasing Word Embeddings

- Word embeddings reflect the biases in the text used to train the model.
- Addressing gender bias in word embeddings - identify bias direction $e_{he} - e_{she}$ and related queries, take an average, find the bias direction in the embedding space via SVD.
- Neutralize - for every word that is not definitional, project to get rid of bias.
- Equalize pairs - for girl and boy, want only difference in embedding to be in the gender direction. Moves boy and girl to be equidistant from non-bias direction.
- What words should be gender specific? Most words in the english language are not definitional - gender is unrelated to their definition. Linear classifier can tell us what words we need to correct and which we do not.

Module 3 - Sequence Models

Basic Models

- Sequence to Sequence model - say you are performing NMT. We have an input sequence x and output sequence y.
- Encoder - RNN (GRU, LSTM), fed in the french words one at a time. Outputs some encoded vectors. Then a Decoder outputs the translated english words in a sequence
    - Each generated word is the input to generated the next word in the decoder sequence

- Similar architecture works for image captioning. We saw that a conv net can learn an encoding for an image at the last FC layer. This can be the encoder network and feed the FC to an RNN decoder, that captions the image with an output sequence one word at time.

Picking Most Likely Sentence

- NMT is like a conditional language model. We previously saw language models based on probabilities, where each output is fed back into the model to generate the next sequence.
- Now the decoder looks like the language model previously, but it is initiated with the encoder network instead of all zeros.
- Instead of modeling the probability of any sentence, we model it as a conditional probability $P(y^{<1>}, \dots, y^{<T_y>} | x^{<1>}, \dots, x^{<T_x>})$.
- We do not want to sample words at random order from this conditional distribution - we want to find the english sentence y that maximizes this distribution.
- Beam search is the most common algorithm for maximizing this probability
- Why not a greedy search? Pick first word most likely, then next and next most likely. The greedy approach does not maximize the joint probability of the sentence.
- There are too many sentences to choose from to search through all possible sentences from your vocab. We use approximate algorithms to make this work.

Beam Search

- First, it needs to pick the first word for the english translation.
- Given the encoder input, predict the probability of $p(y^{<1>} | x).
- Set beam width (say 3), evaluates three possibilities for a pick at a time. Takes three most likely possible first words and keeps track of all 3 in memory as we build the sentence.
- Step 2: Consider what should be the second word for each choice of first word.
- Takes encoder input, and feed $y^{<1>}$ back into the decoder to generate $y^{<2>}$. Does this for each $y^{<1>}$ kept in the first step, and maximizes the probability of the pair: $p(y^{<1>}, y^{<2>} | x) = p(y^{<1>} | x) p(y^{<2>} | x, y^{<1>})$. We consider 3 times 10,000 = 30,000 possibilities for the second step, since we look at the conditional across our entire vocab.
- At every step we have a beam width number of copies of the network.
- Step 3: third word. For each two word fragment we have selected, we feed into the third unit of the decoder, to look at the probabilities for the third word.
- If we set BW = 1, this becomes the greedy search algorithm, but tends to be much better if we set it higher.

Refining Beam Search

- Beam maxes: $\arg\max_y \prod_{t=1}^{T_y} P\left(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>}\right)$
- Length normalization: Multiplying many number that are less than 1 can result in tiny numbers. So in practice we take the log: $\arg\max_y \sum_{t=1}^{T_y} logP\left(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>}\right)$. Makes for a more stable criterion.
    - Long sentence will have smaller probabilities given that it multiplies (sums) many (log) probabilities together.
    - So instead we can normalize the criterion by the number of words in the translation, reducing the penalty, ie $\arg\max_y \frac{1}{T_y} \sum_{t=1}^{T_y^\alpha} \dots$ with some alpha between 0 and 1 if we want to tune this hyperparameter.

- Choosing beam width B? If large, improved translation but slower. In practice, B might be around 10, depending on resources. There is also diminishing returns, so moving to a massive B might not help much.
- Beam search runs faster than BFS / DFS but not guaranteed to find an exact maximum.

Error Analysis on Beam Search

- We have two components to the model - the RNN encoder decoder and beam search.
- Tempting to just increase B to make translation better. RNN computes P(y|x); most useful to compute $p(y^*|x), P(\hat{y}|x)$ for y* true translation to determine the cause of error.
- Case 1: Beam search chose y hat, but y* achieves higher P(y|x) in RNN. Conclusion: beam search is at fault, since it isn't considering the better translation
- Case 2: y* better translation than y hat but RNN predicted $p(y^*|x) < P(\hat{y}|x)$ - then RNN is at fault, since it is evaluating the better translation as worse.
- For each human algorithm comparison that looks bad, compare these two conditional probabilities and assign fault, determine fraction assigned to each, and spend time improving the piece that would make the biggest differnt to error.

BLEU Score

- Measuring accuracy of your translation
- So long as the NMT is close to the reference from humans, then it is a good translation.
- Precision: Look at output words and see whether they appear in the reference. But just outputting a repeated word in the translation gives a perfect score.
- Modified precision: Word only gets credit for max representation across reference translations. Score against the total number of words in the sentence.
- Bleu on bigrams: Look at all possible adjacent bigrams. Can generate a count of bigram appearance in MT and a clipped count looking at the frequency in the references.
- $P_n = \frac{\sum_{\text{n-grams in } \hat{y}} \text{Count clip}(n-gram)}{\sum_{\text{n-grams in } \hat{y}} \text{Count}(n-gram)}$
- P_n = bleu score on n-grams only. Combined bleu score is $(BP)exp(\sum_{n=1}^{4} p_n)$ for BP = brevity penalty. BP = 1 if MT output length > reference output length, and $\exp(1 - \text{ reference output length } / \text{MT} - \text{output length })$ otherwise.

Attention Model

- With long sentences, we are asking an encoder to read the whole thing, encode it, then have the decoder pop NMT out. But it might be better to do it parts, hard to memorize the whole long sentence. With an attention model, we don't see the same drop in performance for longer sentences that we see with other architectures
- Say we have a bidirectional RNN for NMT and we input a french sentence. Now let's take another RNN to generate the english transation S. For generating the first word, want to mostly look at the first word. So the attention model creates attention weights $\alpha^{<1,1>}$ for the first french word, $\alpha^{<1,2>}$ for the second, etc that are passed into the first decoder block. Then for the second NMT word, $s^{<2>}$ we take inputs $\alpha^{<2,1>}, \alpha^{<2,2>}, \ldots$. Feed into a context that aggregates the weights and inputs it into the unit S.
- For $a^{<t>'} = (a_{left}^{<t>'}, a_{right}^{<t>'})$, french input in bidirectional input RNN. Attention weights satisfy $\sum_{t'} \alpha^{<1,t'>} = 1$ and $c^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{<t>'}$ for c context of step t. Note $\alpha^{<t,t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t>}$.
- The decoder portion looks like a standard RNN with the context vectors as input.

- Attention weights: $\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$. We look at the hidden state of previous activation.
- This algorithm can be slow. This algorithm can be applied to image captioning and other problems beyond NMT.

Speech Recognition

- Given audio clip x, want to create transcript y. Often start by generating a spectrogram for the audio. In the past used phoneme, hand engineered features, now use E2E DL.
- CTC Cost for speech recognition - allows RNN to write some characters with blanks. CTC collapses repeated characters not separated by blacks
- Trigger word detection
    - One option: everything before trigger set target labels to 0, then when trigger said, set target to 1.
    - Create imbalanced dataset, so instead could make more 1's around the instant of trigger word.