**Lecture 1 | Introduction to Convolutional Neural Networks for Visual Recognition**

What is this class?

* Computer vision
* Uses visual data

Computer vision

* Used in physics, biology, engineering, mathematics, computer science

History of computer vision

* Evolution’s big bang
  + Specific short term of time when # of species were ‘exploded’.
* Camera Obscura
  + Pin hole camera theory
  + Similar to early animals’ eye
* Hubel & Wiesel, 1959
  + To find neural response mechanism or visual recognition mechanism of mammals
  + Stick electrode in cat’s brain & checked what made neuron respond excited
  + Simple cells, responded in specific moving direction
* Block world
  + Reconstruct the structure
  + “The summer vision project”
* How can we recognize structure?
  + Generalized cylinder
  + Pictorial structure
* If object recognition is too hard, maybe we should first do object segmentation
  + Task of taking image and group the pixels into meaningful areas
  + **“Image Segmentation”**
* Face detection
  + AdaBoost to do real-time face recognition
* “SIFT” feature
  + SIFT & object recognition, David Lowe, 1999
* PASCAL visual object challenge (2006~2012)
  + 20 object categories
* ImageNet / ImageNet large scale visual recognition challenge
  + to recognize the objects / overcome the ML bottleneck of overfitting, ImageNet project is launched.
  + Error rate decreased, and made lower error(3.57%) than human(5%)

Goal of this course

* Learn about Convolutional Neural Network
* Focuses on image classification
* Object detection, image captioning

Difference of 1990’s and 2010’s

* # of transistors(speed of computation)
* # of labeled data

**Lecture 2 | Image Classification**

Image classification: a core task in computer vision

* Assume given set of discrete labels and recognize the object
* Image is just a big grid of numbers between [0,255] for 3 channels RGB
* Problem: semantic gap
  + Computer may recognize pixels, but don’t understand that the pixels are for one same object. We need ML to make computer learn this pattern and recognize the pattern
  + Pixel-level to semantic-level
* Challenges
  + Viewpoint variation: tilting camera
  + Illumination: different light condition
  + Deformation: cat’s various position
  + Occlusion: we can see only a part of cat
  + Background clutter: cat’s color may similar to environment
  + Intraclass variation: cat has various color and age(representation)
* Algorithm is unlike simple sorting numbers
* No obvious way to hard-code the algorithm for recognizing a cat, or other classes

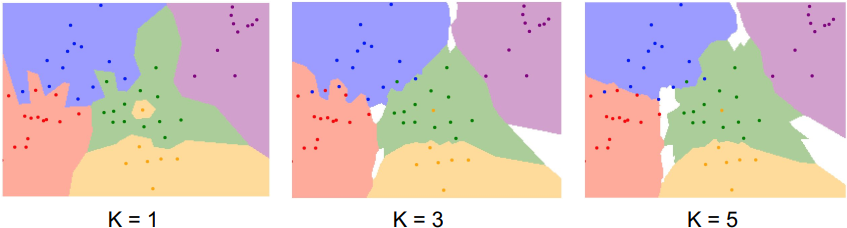
Attempts have been made

* Find edges and corners
  + Edge is one of important point to classify objects
    - Doesn’t work well: brittle and have to start all over again for other objects
* Data-driven approach
  + 1. Collect a dataset of images and labels
  + 2. Use machine learning to train a classifier
  + 3. Evaluate the classifier on new images
  + Has two key modules: train, predict
  + First classifier: nearest neighbor
    - Train: memorize all data and labels
    - Predict: predict the label of the most similar training image
  + Example dataset: CIFAR10
    - 10 classes, 50k training images, 10k testing images

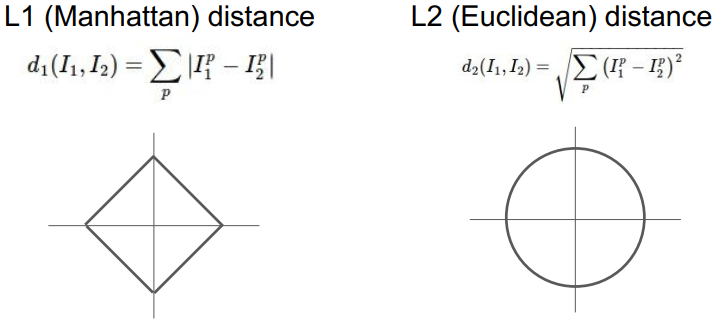
**K-nearest neighbors**

How to compare images? – Distance metric to compare images

* L1 distance (Manhattan)
  + Comparing single pixel of images
  + Add pixel-wise absolute value differences
* With N examples, how fast are training and prediction?
  + Train O(1), predict O(N)
  + Not good: we want fast at prediction. Slow for training is okay
* K-nearest neighbors
  + Instead of copying label from nearest neighbor, take majority vote from K closest points
  + If K is big, tends smother edges



* L2 distance (Euclidean)



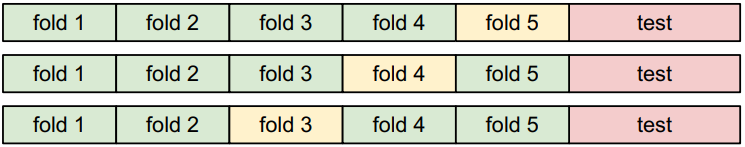
* L1 and L2 distance
  + L1 depends on choice of coordinate system
  + If we rotate the coordinate frame, L1 distance changes but L2 doesn’t.
  + If the input features or individual entries in vector has some important meaning for task, L1 distance is a natural fit
  + If the input vector is a generic vector in some space and don’t know which of the different elements does actually mean, L2 distance is a natural fit.
* K-nearest neighbors with L1/L2 distance
  + Boundaries changes slightly for each method.

Hyperparameters

* What is the best value of **k** to use?
* What is the best **distance** to use?
* Hyperparameter: k and distance, choices about the algorithm that we set rather than learn
* Problem dependent, must try them all out and see what works best

Setting hyperparameters

* Idea 1: choose hyperparameters that works best on the data
  + Very bad idea: k=1 always works perfectly on training data
  + If the k gets bigger, it might cause us to misclassify some of the training data.
  + But it leads to better performance on the data which were not in training data
  + Ultimately, we really care about how our classifier or method performs on unseen data out of training data.
* Idea 2: split data into train and test, choose hyperparameters that work best on test data
  + Seems more reasonable, but also a terrible idea.
  + No idea how algorithm will perform on new data
* Idea 3: split data into train, validation, test and choose hyperparameters on val and evaluate on test
  + Try many hyperparameters at the training set and evaluate on validation set. Choose hyperparameters that best for the validation set, and run once on the test set.
  + Better method
* Idea 4: cross-validation; split data into folds, try each fold as validation and average the results



* + Change validation sets for each fold
  + Useful for small datasets, but not used too frequently in deep learning

k-nearest neighbor on images never used

* Very slow at test time
* Distance metrics on pixels are not informative
* Curse of dimensionality: needs more dots for bigger dimensions(grows exponentialy)

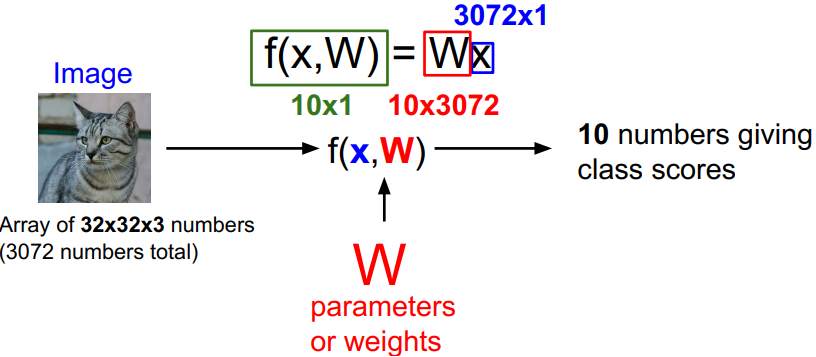
k-nearest neighbors: summary

* In image classification, we start with a training set of images and labels, and must predict labels on the test set
* The k-nearest neighbors classifier predicts labels based on nearest training examples
* Distance metric and k are hyperparameters
* Choose hyperparameters using the validation set; only run on test once at the very end

**Linear classification**

Parametric approach

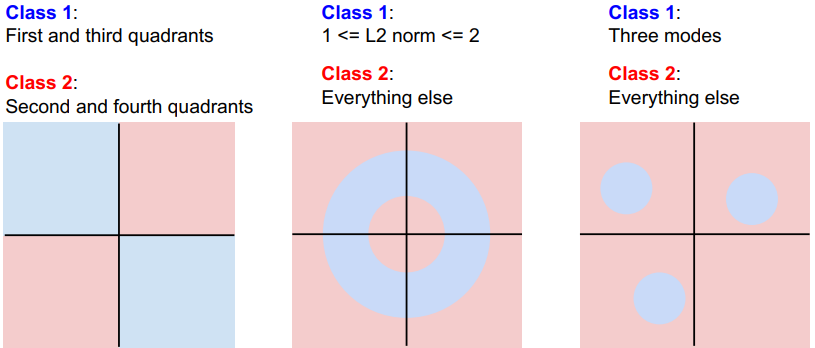
* Recall CIFAR10; 50k different image with 32\*32\*3
* Input data: image, array of 32\*32\*3 numbers (3072 numbers total)
* Make function of input data and parameters or weights(W)
* Output: 10 numbers giving class scores



* Simple function: f(x, W) = Wx +b (b: bias)

Interpreting a linear classifier

* Make a single linear line of classifying objects
* If object is over the line, then it is classified with that group
* Hard cases for s linear classifier



* + Hard to make a linear line to separate into two groups
  + Parity problem separating odd from even/multimodal situations

**Lecture 3 | Loss Functions and Optimization**

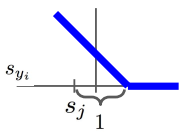
Recall from last time

* Challenges of recognition: semantic gap
* Data-driven approach, kNN, CIFAR-10
* Linear classifier: f(x, W) = Wx+b
  + High score means high probability of classification into specific class
  + TODO: define a loss function & minimize the loss function(optimization)

**Loss function**

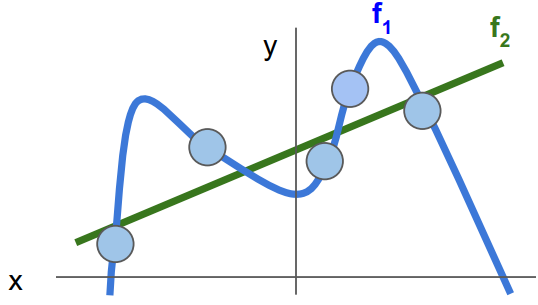
* Represents how good or bad the current classifier is
* Given a dataset of examples where is image and is integer label (base truth)
* Loss over the dataset is a sum of loss over examples:

Multiclass SVM loss

* Let .
* Sum up all incorrect categories’ scores.
* Some type of Hinge loss 
* Q. why +1 at the score?
  + A. Actually, it is an arbitrary value. If we rescale the data, value 1 doesn’t matter.
* Maximum possible loss is inf., minimum possible loss is 0.
* If W is too small so all s is about 0, loss will be N-1 for each class.
* What if we use ?
  + This will become different classifier.
* Why don’t we use squared errors?
  + This is a difference of caring errors of class.
* Suppose that we found a W such that L = 0, is this W unique?
  + No. 2W is also same.

Regularization

* We order to model that to find best fit in training data. (blue curves)
* However, what we really concern is how it works on test data.
* It may cause overfitting. (different with green line)



* To prevent overfitting, we add new regularization term at the loss function.
* Lambda means regularization strength. One of hyperparameter.

Types of regularization

* L2 regularization:
* L1 regularization:
* Elastic net (L1 + L2):
* Max norm regularization
* Dropout
* Batch normalization, stochastic depth, fractional pooling, etc.
* L2 regularization likes to “spread out” the weights.
* L1 regularization likes many 0s in weights. Especially, opposite tend of L2 regularization.

Why regularize?

* Express preferences over weights
* Make the model simple so it works on test data
* Improve optimization by adding curvature

Softmax classifier (Multinomial Logistic regression)

* Want to interpret raw classifier scores as probabilities.
* Softmax function:
* Min loss is 0, max loss is inf.

Softmax vs. SVM

* SVM cares only getting correct score to be greater than a margin above the incorrect scores.
* Softmax loss wants to drive the probability mass all the way to one.

Recap

* We have some dataset of (x, y)
* We have a score function:
* We have a loss function: softmax, SVM, regularization loss

**Optimization**

* We want to make loss function decrease
* Strategy 1: random search / very bad idea
* Strategy 2: follow the slope

Gradient

* In 1-D, derivative of a function is
* In multiple dimensions, gradient is the vector of partial derivatives along each dimension
* The slope in any direction is the dot product of the direction with the gradient.
* The direction of steepest descent is the negative gradient.
* We can use numeric gradient, but it is too slow, approximate.
* We can use calculus to compute an analytic gradient. It is exact and fast.
* We want .

Gradient descent

* Initialize W in some random value and compute loss and gradient.
* Update W in opposite direction of gradient.
* However, we have to determine learning rate(how far we move).

Stochastic Gradient Descent (SGD)

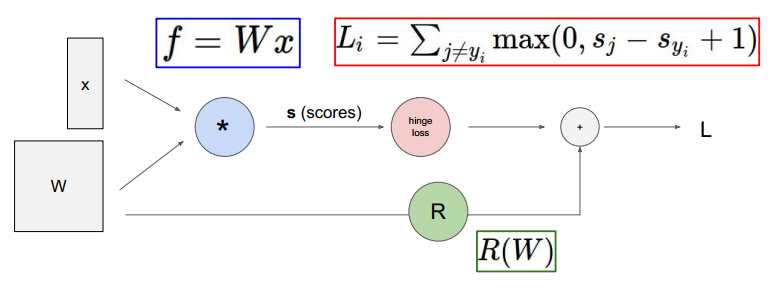
* Full sum is expensive when N is large.
* Approximate sum using a minibatch of examples 32/64/128 common.

**Lecture 4 | Introduction to Neural Networks**

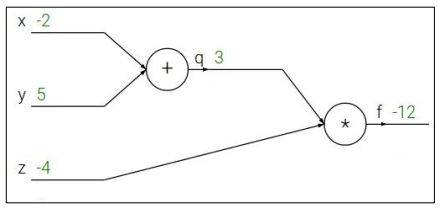
Where we are?

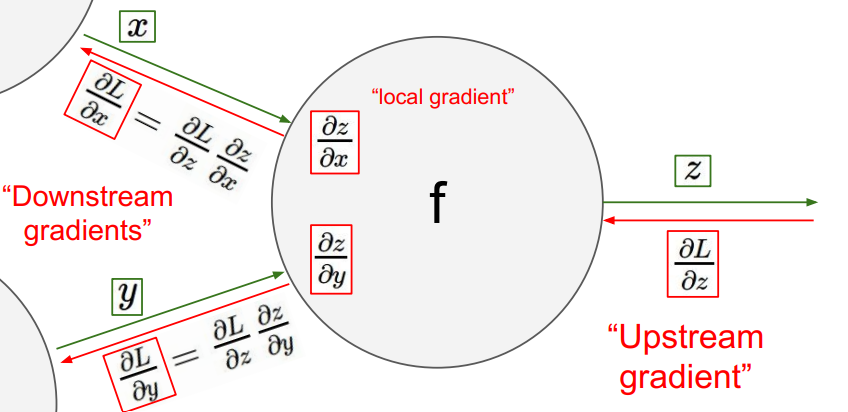
* Scores function:
* SVM loss:
* Data loss + regularization:
* We want
* There were numeric/analytic gradient and both has pros and cons

Computational graphs

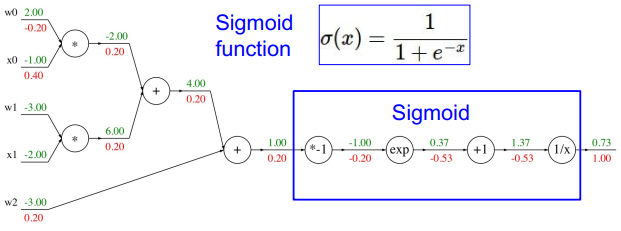


Backpropagation: a simple example

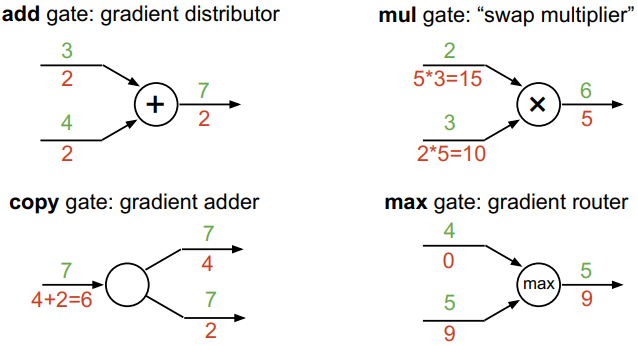
* 
* We want
* Chain rule:



* Another example

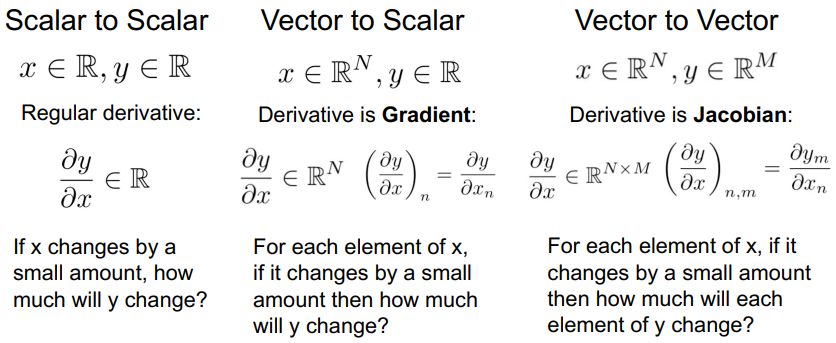


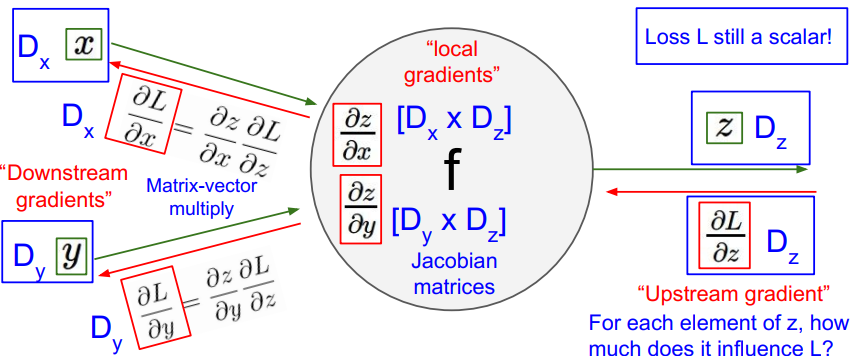
* Computational graph representation may not be unique. We can change some of graph to sigmoid function at the example.

Patterns in gradient flow

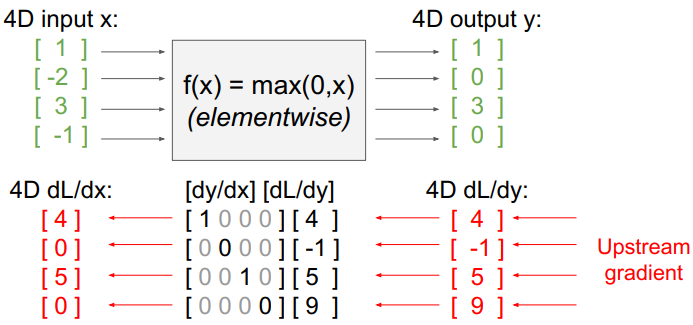
* Add gate: gradient distributor
* Mul gate: “swap multiplier”
* Copy gate: gradient adder
* Max gate: gradient router

Backprop with vectors



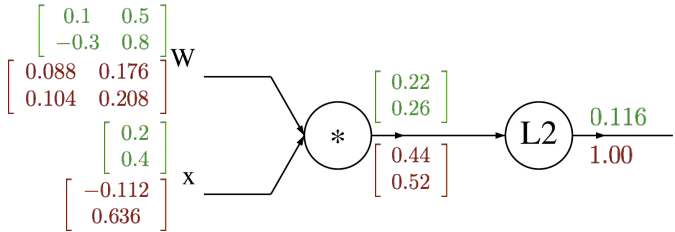


* is a vector with Dz dimension
* Each and become Jacobian matrices with [Dx \* Dz] and [Dy \* Dz] size
* become matrix-vector multiply and results Dx dimension vector



* The size of Jacobian matrix is 4 \* 4 (4 squared).
* If the size of input increases, making Jacobian matrix is very inefficient.
* Actually, Jacobian matrix has 0s for non-diagonal elements(diagonal matrix). So we don’t have to make entire Jacobian matrix.

Vectorized example



Summary

* Neural nets will be very large: impractical to write down gradient formula by hand for all parameters
* backpropagation = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
* implementations maintain a graph structure, where the nodes implement the forward() / backward() API
* forward: compute result of an operation and save any intermediates needed for gradient computation in memory
* backward: apply the chain rule to compute the gradient of the loss function with respect to the inputs

Neural networks: without the brain stuff

* before: linear score function:
* now: 2-layer neural network:
* “Neural Network” is a very broad term. These are more accurately called “fully-connected networks” or “multi-layer perceptrons” (MLP)

Next time

* Many activation functions
  + Sigmoid, tanh, ReLU, Leaky ReLu, Maxout, ELU
* Neural networks architectures

**Lecture 5 | Convolutional Neural Networks**

A bit of history

* Perceptron: Frank Rosenblatt, ~1957
* Adaline/Madaline: Widrow and Hoff, ~1960
* First time back-propagation becams popular: Rumelhart et al., 1986
* First strong results: speech recognition and Imagenet classification, 2012
* Receptive fields of single neurons in the cat’s striate cortex: Huble & Wiesel, 1959
* Hierarchical organization of cells
  + Simple cell: response to light orientation
  + Complex cells: response to light orientation and movement
  + Hypercomplex cells: response to movement with an end point
* AlexNet: ImageNet classification with deep CNN, 2012

ConvNets are everywhere

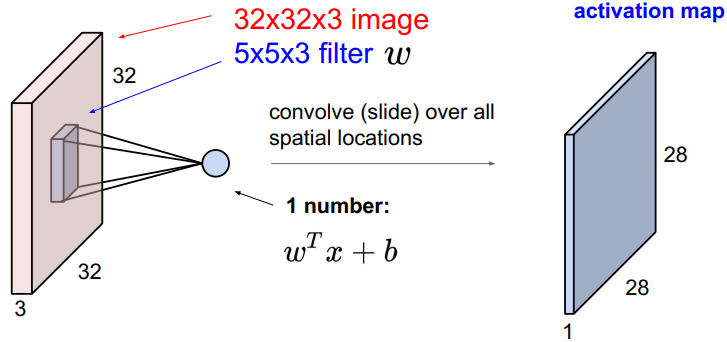
* Image classification
* Image retrieval
* Image detection
* Image segmentation
* Self-driving cars
* Face recognition
* Classifying video
* Pose estimation
* Playing games – Atari
* Image style transfer

Fully connected layer

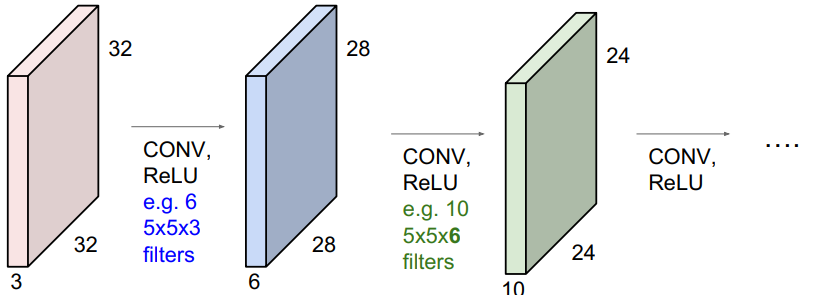
* 32\*32\*3 image -> stretch to 3072\*1 vector
* Input: 3072\*1 vector -> Wx: 10\*3072 weights -> acvitation: 10\*1 vector

Convolution layer

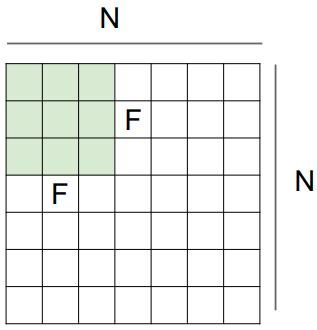
* We keep the original structure of image(32\*32\*3).
* Then, we consider 5\*5\*3 filter, which has full depth(3).
* We locate the filter on the top of spatial location in the image, and to the dot product.
* We need to stretch out to calculate the dot product.
* **Convolve** the filter with the image – slide over the image spatially, computing dot products.
* Total output is 28\*28\*1 layer.



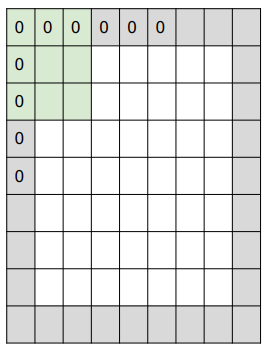
* We can make multiple activation layers with different filter and stack them to make another convolution layers.
* ConvNet is a sequence of convolution layers, interspersed with activation functions.



A closer look at spatial dimensions

* Assume we have 7\*7 input and 3\*3 filter with stride 1.
  + We will get 5\*5 output.
* Assume all the same bud stride is 2.
  + 3\*3 output.
* Stride 3?
  + Ii doesn’t fit.
* Output size: **(N – F) / stride + 1**
  + If N = 7, F = 3, stride 1 -> 5 / stride 2 -> 3 / stride 3 -> 2.33

In practice: common to zero pad the border

* Input: 7\*7 / filter: 3\*3 applied with stride 1
* Pad with 1 pixel border with 0. What is the output?
* We can get 7\*7 output.
* Zero padding makes same output size to input
* In general, Conv layers with stride 1, filters of size F\*F, zero padding is **(F-1)/2**
* With no zero padding, regenerated activation layers shrink too fast. This makes loss of data and make doesn’t work well.

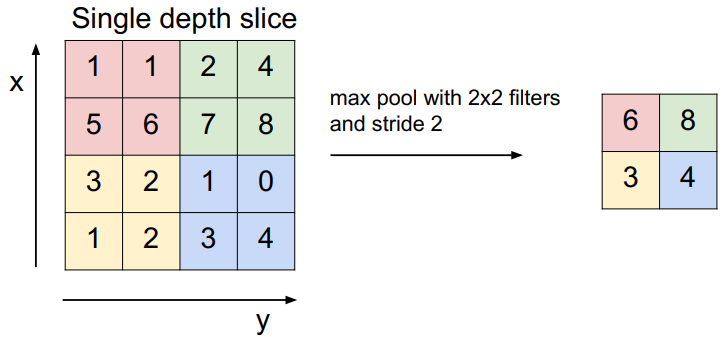
Examples

* Input volume: 32\*32\*3
* 10 5\*5\*3 filters with stride 1, padding 2
* Q1. What is output volume size?
  + A1. 32\*32\*10 / (32+2\*2-5)/1+1=32
* Q2. Number of parameters in this layer?
  + A2. 760. Each filter has 5\*5\*3+1=76 params(1 for bias) and 76\*10=760

Brain/neuron view of Conv layer

* Activation map is a 28\*28 sheet of neuron outputs.
* Each is connected to a small region in the input.
* All of them share parameters.
* 5\*5 filter -> 5\*5 receptive field for each neuron.

Pooling layer

* Makes the representations smaller and more manageable
* Operates over each activation map independently
* Downsampling process
* One common way: max pooling
* Commonly, there are no overlaps and zero padding for pooling
* Why max pooling than average pooling?
  + A value can meaning of signal of neuron. So we want significant signal.

Fully connected later(FC layer)

* At the last, we want to have a fully connected layer.
* We take the ConvNet result output, and put in the vanilla neural network to get score.

Summary

* ConvNets stack Conv, pool, FC layers
* Trend towards smaller filters and deeper architectures
* Trend towards getting rid of pool/FC layers (just Conv)
* Historically architectures looked like

**[(CONV-RELU)\*N-POOL?]\*M-(FC-RELU)\*K,SOFTMAX**

where N is usually up to ~5, M is large, 0 <= K <= 2.

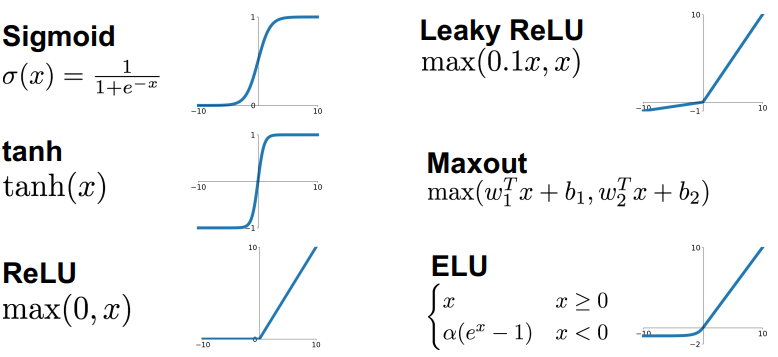
* + but recent advances such as ResNet/GoogLeNet have challenged this paradigm

**Lecture 6 | Training Neural Networks I**

Mini-batch SGD

* 1. Sample a batch of data
* 2. Forward prop it through the graph (network), get loss
* 3. Backprop to calculate the gradients
* 4. Update the parameters using the gradient

**Activation functions**



Sigmoid function

* Squashes number to range [0,1]
* Has a nice interpretation as a saturating ‘firing rate’ of a neuron
* Has 3 problems
  + Saturated neurons “kill” the gradients
    - For a very negative/positive value, sigmoid function gives ~0 gradients to downstream backpropagation
  + Sigmoid outputs are not zero-centered
    - If the input to a neuron is always positive, gradients on w will always all positive or negative(same with upstream backpropagation sign). This makes inefficient gradient update.
    - We want a zero mean data X for input, to not get a gradient update problem.
  + Exp() id a bit compute expensive

tanh(x)

* Squashes numbers to range [-1, 1]
* Zero centered – good
* Still kills gradients when saturated

ReLU (Rectified Linear Unit)

* Computes
* Does not saturate in + region
* Very computationally efficient
* Converges much faster than sigmoid/tanh in practice (6 times)
* More biologically plausible than sigmoid
* Problems
  + Not zero centered output
  + An annoyance: killing the gradient for the half of the regime(- area)
    - Some of networks will be dead ReLU, and will never activate and updated
    - People like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Leaky ReLU

* Does not saturate
* Computationally efficient
* Converges much faster than sigmoid/tanh in practice
* **Will not “die”**

Parametric rectifier (PReLU)

* Backprop into alpha(parameter)

Exponential linear units (ELU)

* All benefits of ReLU
* Closer to zero mean outputs
* Negative saturation regime compared with leaky ReLU and adds some robustness to noise
* Computation requires exp()

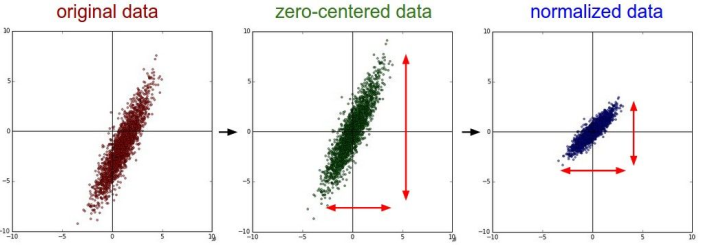
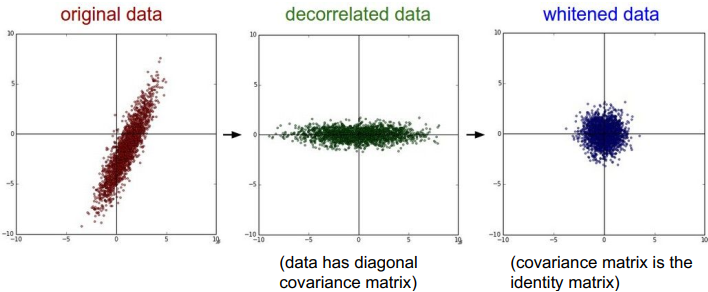
Maxout “neuron”

* Does not have the basic form of dot product -> nonlinearity
* Generalizes ReLU and Leaky ReLU
* Linear regime, does not saturate, doen’t die
* Problem: doubles the number of parameters and neuron

In practice

* Use ReLU, be careful with learning rates
* Try Leaky ReLU / Maxout / ELU
* Try tanh but don’t expect much
* Don’t’ use sigmoid

Data preprocessing

**** ****

Preprocess the data

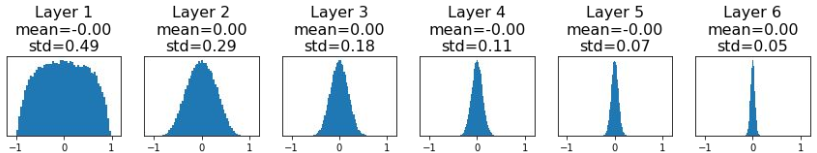
* Zero mean them, normalize it with standard deviation.
* All features are in the same range and contribute equally.
* In practice, PCA and whitening of data can used.
* We also do this for test data.
* In practice for images: center only
  + Consider CIFAR-10 example with [32,32,3] images
  + Subtract the mean image (AlexNet) / mean image = [32,32,3] array
  + Subtract per-channel mean (VGGNet) / mean along each channel = 3 numbers
  + Suvtract per-channel mean and divide bt per-channel std (ResNet) / mean along each cahnnel = 3 numbers
  + It is not common to do PCA or whitening

Weight initialization

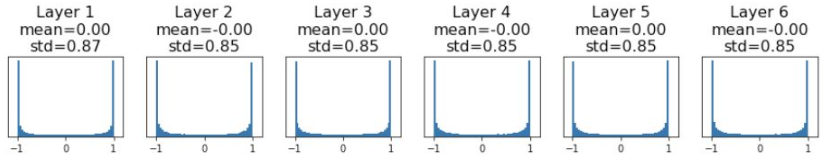
* Q. what happens when W=0 init is used?
  + A. all the neurons will do the same thing and same output.

First idea: small random numbers (gaussian with zero mean and 1e-2 standard deviation)

* W = 0.01 \* np.random.randn(Din, Dout)
* Works okay for small networks, but problems with deeper networks.
* As we multiply by this W by small numbers at each layer, this quickly shrinks and collapses all these values. So all activation become zero
* Q. what to the gradients dL/dW look like?

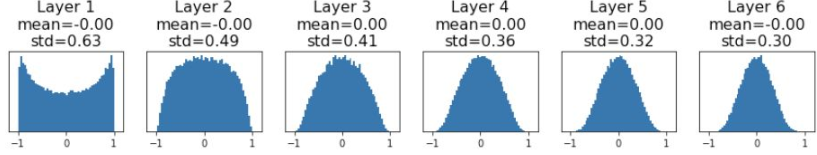


How about increasing std?

* W = 0.05\* np.random.randn(Din, Dout)
* Almost all neurons completely saturated, either -1 and 1.
* Gradients will be all zero, and no update.
* 

Xavier initialization

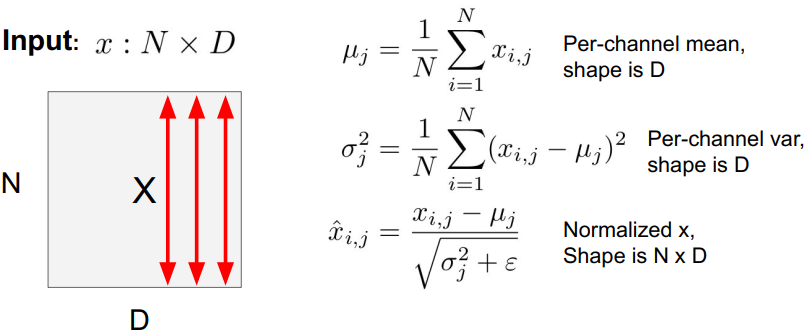
* W = np.random.randn(Din, Dout) / np.sqrt(Din)
* Reasonable initialization – mathematical derivation assumes linear activations



* What about ReLU?
  + It breaks because it kills half of our inputs
  + For ReLU, we need to correct std = sqrt(2 / Din)
  + This makes nicely scaled for all layers

Batch normalization

* We want to keep activations in a gaussian range.
* Consider a batch of activations at some later. To make each dimension zero-mean unit-variance, apply down function
* <- this is a vanilla differentiable function
* 1. Compute the empirical mean and variance independently for each dimension.
* 2. Normalize.



* This process is usually inserted after FC convolutional layers, and before nonlinearity.
* Learning will recover the identity function.
* Summary
  + This improves gradient flow through the network
  + Allows higher learning rates
  + Reduces the strong dependence on initialization, slightly reduces the need for dropout.
* Q. Why do we need to learn beta or gamma?
  + A. We want extra flexibility for control the degree of saturation
* We can’t calculate beta/gamma at the test time, so we want to use learned parameter during learning. So we use empirical mean and variance at the test time.

Choosing hyperparameters

* 1. Check initial loss
  + Turn off weight decay, sanity check loss at initialization
  + e.g. log(C) for softmax with C classes
* 2. Overfit a small sample
  + Try to train to 100% training accuracy on a small sample oftraining data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization
  + Loss not going down? LR too low, bad initialization
  + Loss explodes to Inf or NaN? LR too high, bad initialization
* 3. Find LR that makes loss go down
  + If the loss barely changes, lr is too small.
  + If the loss is barely changed, but the accuracy jumped up to ~20% in a softmax function. This is because the probabilities are still diffuse, so loss is similar. However, weight can move to correct direction slightly and this makes accuracy jump.
  + NaN means too high lr – loss exploging.
  + Rough range for learning rate we should be cross-validating is [1e-3, 1e-5]
* 4. Coarse grid, train for ~1-5 epochs
* 5. Refine grid, train longer
* 6. Look at loss curves

Hyperparameter optimization

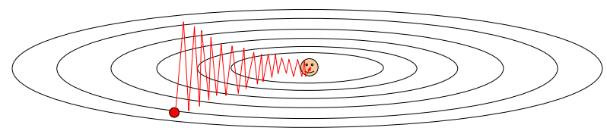
* Cross-validation strategy
  + Coarse -> fine cross-validation in stages
  + First stage: only a few epochs to get rough idea of what params work
  + Second stage: longer running time, finer search
  + Repeat this
* Random search vs. Grid search
  + Random search for hyperparameter is better
* Hyperparameter to play with:
  + Network architecture
  + LR, its decay schedule, update type
  + Regularization(L2/Dropout strength)

**Lecture 7 | Training Neural Networks II**

Last time

* Activation functions
  + Lots of functions, but ReLU is a good default choice
* Weight initialization
  + Initialization too small: activations go to zero, gradients also zero -> no learning
  + Initialization too big: activations saturate(for tanh), gradients zero -> no learning
  + Just right: nice distribution of activations at all layers, learning process nicely
* Data preprocessing
  + Zero center and normalize data -> zero mean and unit variance
  + Before normalization: classification loss is very sensitive to changes in weight matrix; small perturbations in weight matrix could cause large perturbation in outputs -> hard to optimize
  + After normalization: less sensitive to small changes in weights; easier to optimize
* Batch normalization
  + Add additional layer into network and force all of the intermediate activations to zero mean % unit variance.
  + Mean and standard deviation can be learned by small-batch used learning with batch normalization. We can use learned params to test data.
* Babysitting learning
* Hyperparameter search
  + Random search is better than grid search theoretically

Optimization

* Stochastic gradient descent(SGD)
  + Super simple
  + Evaluate gradient and update weights
* However, SGD has problems
  + Consider 2D weight function and 3D loss field
* Poor conditioning: what if loss changes quickly in one direction(e.g. vertically) and slowly in another(e.g. horizontally)
  + Loss function has high condition number: ratio of largest to smallest singular value hf the Hessian matrix is large
  + SGD will move very slow progress along shallow dimension, jitter along steep direction.
  + This problem is more common in higher dimension.
* What if the loss function has a local minima or saddle point?
  + SGD will be stuck at that point, because it has zero gradient.
  + Also more common in high dimension
* ‘stochastic’ can cause problem
  + We usually ‘estimate’ loss and gradient with small batch because of cost of learning full dataset.
  + This means we are not getting true information at each step. Instead, we are just getting some noisy estimate of the gradient at our current point.
  + In some case, we can get long time to reach optimum weight.

SGD + momentum

* SGD:
* SGD+momentum:
* We maintain velocity over time and add gradient estimate into gradient velocity.
* Build up velocity as a running mean of gradients
* Rho() gives ‘friction’; typically 0.9 or 0.99 (high value)
* Now, we can have a velocity even at the local minima or saddle point
* Good initialization for the velocity is just setting into zero.

Nesterov momentum



* Normal momentum update: combine gradient at current point with velocity to get step used to update weights
* Nesterov momentum: look ahead to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction
* For more easy form:
* At the second equation, we have our current point plus our current velocity plus weighted difference between our current velocity and previous velocity.
* Nesterov momentum is kind of incorporating some kind of error-correcting between current/previous velocity

Q. Does it work also at the very ‘sharp’ minima? Doesn’t it skip that?

* A. very sharp minima could be a ‘bad’ minima, which can overfit our model. If more data is added, that minima can vanish off. Usually, flat minima is more robust to us.

AdaGrad

* We are adding squared gradient continuously, and divide current gradient with sqrt of summed squared gradients.
* In code,
  + grad\_squared = 0

while True:

dx = compute\_gradient(x)

grad\_squared += dx \* dx

x -= learning\_rate \* dx / (np.sqrt(grad\_squared) + 1e-7)

* What happens if we have a very high condition number?
  + If the one has high gradient and another one has small gradient for 2D case, small gradient will accelerate and high gradient will slow down.
* What happens to the step size over long time?
  + Steps gets smaller because squared grad increases monotonically
  + If the step is big and near saddle point, model can stuck.

RMSProp

* Slight variation of AdaGrad (Leaky AdaGrad)
* Grad\_squared term is changed to **decay\_rate \* grad\_squared + (1 – decay\_rate) \* dx \* dx**
* So sum of squared gradients decay.
* Decay rate is usually 0.9 or 0.99 (high value)

Adam(almost)

* Combined version of momentum and AdaGrad/RMSProp
* In code,
  + first\_moment = 0

second\_moment = 0

while True:

dx = compute\_gradient(x)

**first\_moment** = beta1 \* first\_moment + (1 - beta1) \* dx # momentum

**second\_moment** = beta2 \* second\_moment + (1 – beta2) \* dx \* dx # AdaGrad

x -= learning\_rate \* first\_moment / (np.sqrt(second\_moment) + 1e-7) # AdaGrad

* What happens at first timestep?
  + After first update, second\_moment may very small. It is because value of beta2 is near one. So, we are dividing with very small number and very large step at the start.
  + This initial large step is not due to geometry of the problem. It is affected by the artifact of the model.

**Adam (full form)**

* In code,
  + first\_moment = 0

second\_moment = 0

while True:

dx = compute\_gradient(x)

first\_moment = beta1 \* first\_moment + (1 - beta1) \* dx # momentum

second\_moment = beta2 \* second\_moment + (1 – beta2) \* dx \* dx # AdaGrad

first\_unbias = first\_moment / (1 – beta1 \*\* t) # bias correction

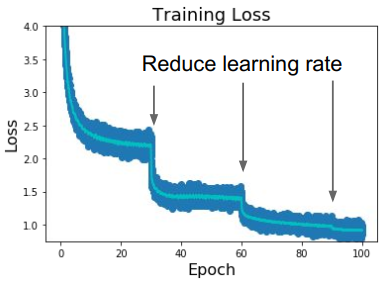
second\_unbias = second\_moment / (1 – beta2 \*\* t) # bias correction

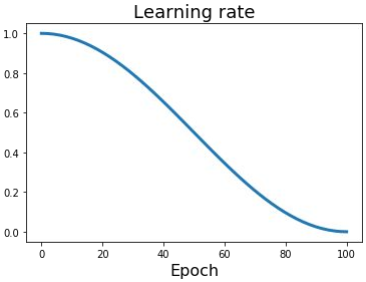
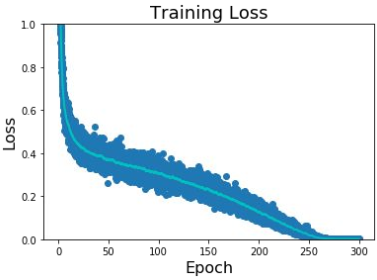
x -= learning\_rate \* first\_unbias / (np.sqrt(second\_unbias) + 1e-7)

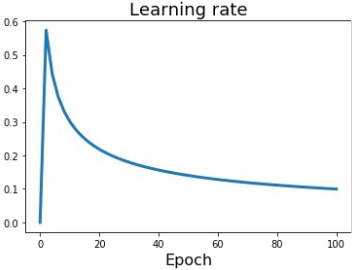
* Bias correction term is added to code.
* With beta1 = 0.9, beta2 = 0.999, and learning\_rate = 1e-3 or 5e-4 is good initialization

Q. Which one of these learning rates is best to use?

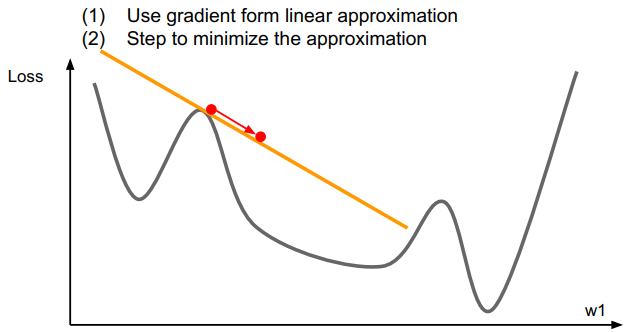
* A. All of them. They all start at large learning rate and decay over time.
* SGD, SGD+momentum, AdaGrad, RMSProp, Adam all have learning rate as a hyperparameter.

Learning rate decay

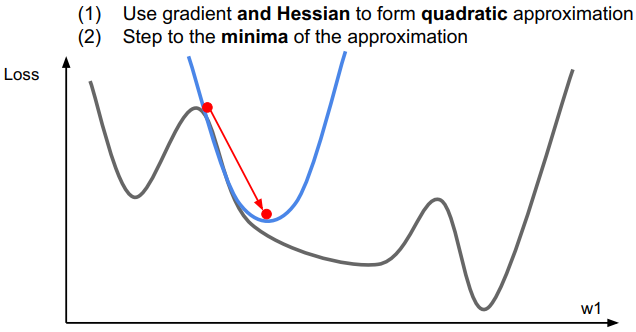
* Step: Learning rate is reduced at a few fixed points.
  + For ResNets, lr is multiplied with 0.1 after each 30 epochs.
  + This prevents model oscillating near minimum and reduces loss more.
* Cosine:
  + : initial learning rate
  + : learning rate at epoch t
  + : total number of epochs
* Linear:
* Inverse sqrt:

Learning rate decay: linear warmup

* High initial learning rate can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this
* Empirical rule of thumb
  + If we increase the batch size by N, also scale the initial learning rate by N

First-order optimization

* 1. Use gradient form linear approximation
* 2. Step to minimize the approximation
* First order Taylor approximation
* This doesn’t hold far for very large regions

Second-order optimization

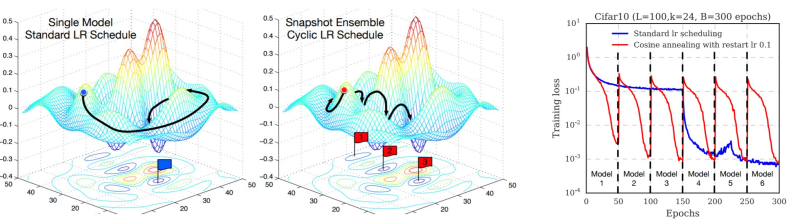
* 1. Use gradient and Hessian to form quadratic approximation
* 2. Step to the minima of the approximation
* Second order Taylor approximation to function, and locally approximate the function to quadratic.
* Second-order Taylor expansion
* Solving for the critical point we obtain the Newton parameter update
* This method doesn’t have(or need) learning rate.
* However, this is bad for deep learning
  + Hessian has O(N^2) elements, inverting takes O(N^3), and N = tens or hundreds of millions
* Quasi-Newton methods(BFGS most popular)
  + instead of inverting the hessian(O(N^3)), approximate inverse Hessian with rank 1 updates over time (O(N^2) each)
* L-BFGS (limited memory BFGS)
  + Does not form/store the full inverse Hessian
  + Usually works very well in full batch, deterministic mode
  + Does not transfer very well to mini-batch setting

Beyond training error

* All these strategies were about reducing training error.
* However, we don’t really care about training error that much. Instead, we really care about unseen data(test data).
* Once we are already good at optimizing our objective function, what can we do to reduce the gap of training data and unseen data?

Model ensembles

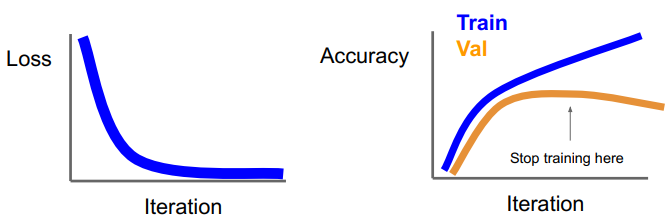
* 1. Train multiple independent models
* 2. At test time, average their results.
* This method enjoy 2% of extra performance
* Tips and tricks
  + Instead of training independent models, use multiple snapshots of a single model during training.
  + Cyclic learning rate schedules can make this work even better.



* + Polyak averaging: Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time

Early stopping

* Stop training the model when accuracy on the validation set decreases.
* Or training for a long time, but always keep track of the model snapshot that worked best on val.



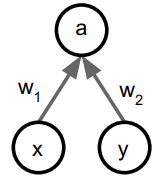
Regularization

* Add term to loss:
* In common use
  + L2 regularization: (weight decay)
  + L1 regularization:
  + Elastic net (L1 + L2):

Regularization: dropout

* In each forward pass, randomly set some neurons(activations) to zero
* Probability of dropping is a hyperparameter; 0.5 is common
* How can this possibly be a good idea?
  + Forces the network to have a redundant representation; prevents co-adaptation of features
  + This can prevent overfitting in some way
  + Dropout is training a large ensemble of models (that share parameters). Each binary model is one model.

Dropout in test time

* Dropout makes output random, and we want to average out the randomness at test-time.
* But real calculation is too hard. So, we need to approximate the integral.
* Consider a single neuron.
  + At test time:
  + At training:
* At test time, multiply by dropout probability
* At test time, all newrons are active always. We must scale the activations so that for each neuron: output at test time = expected output at training time

A common pattern of regularization

* Training: add some kind of randomness
* Testing: average out randomness (sometimes approximate)
* Example: batch normalization
  + Training: normalize using stats from random minibatches
  + Testing: use fixed stats to normalize

Regularization: data augmentation

* We can randomly transform the image in some way during training such that label is preserved.
* Horizontal flips, random crops and scales, color jitter, rotation, stretching, shearing, …

Regularization: DropConnect

* Similar to dropout, but this method randomly zero out some of the values of the weight matrix instead.

Regularization: fractional pooling

* Training: use randomized pooling regions
* Testing: average predictions from several regions

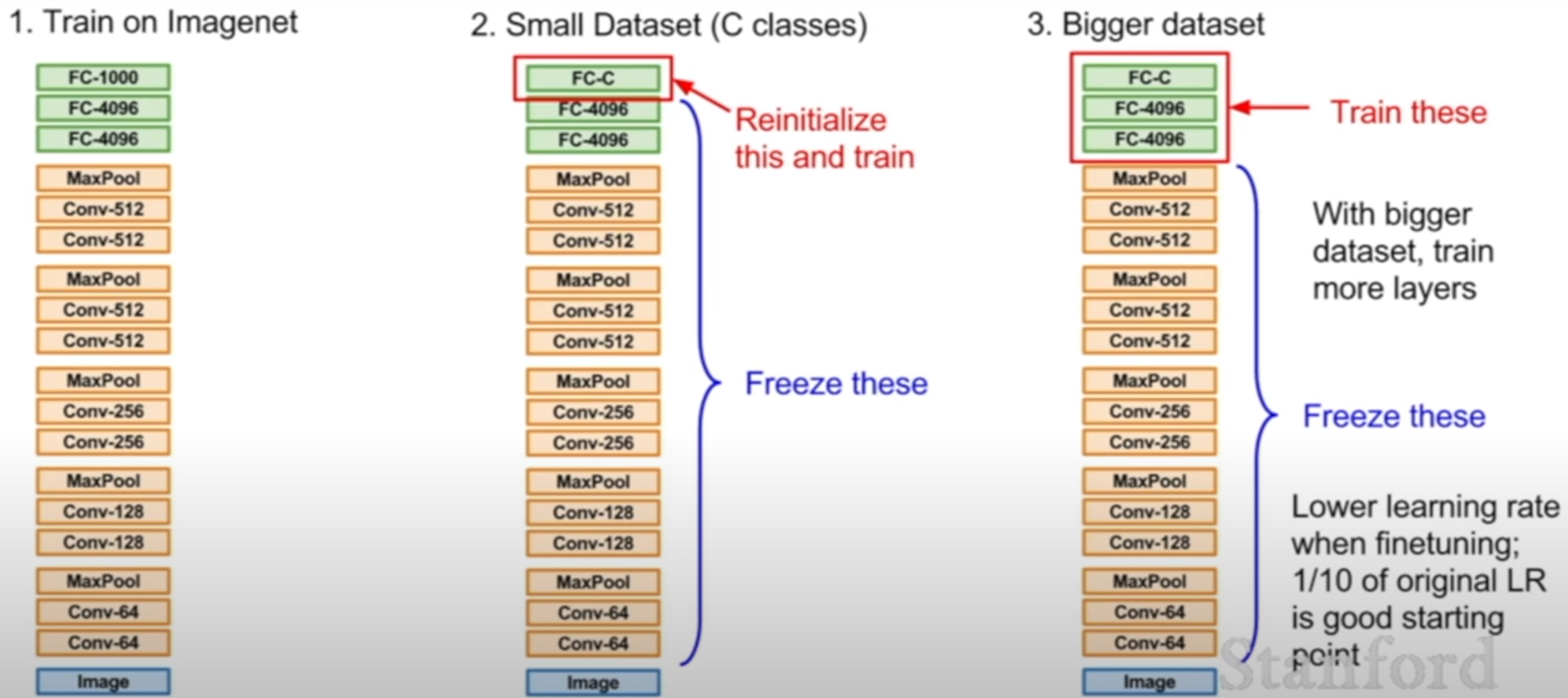
Regularization: stochastic depth

* When we have a very deep network, we are going to eliminate some layers and use only some of subsets of networks during training, and use all at test time.

Regularization methods

* Dropout, batch normalization, data augmentation, dropconnect, fractional max pooling, stochastic depth, cutout(set random image regions to zero at training / use full image at test time), mixup(train on random blends of images at training / use original images at test time)
* Consider dropout for large FC layers
* Batch normalization and data augmentation is almost always a good idea
* Try cutout and mixup for small classification datasets

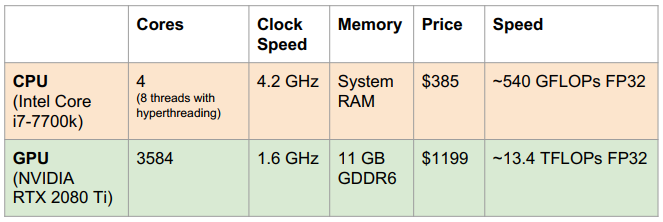
Transfer learning

* Normally, we need a lot of data if we want to train/use CNNs. However, this method uses small data to learn model of CNN.
* 

**Lecture 8 | Deep Learning Software**

CPU vs. GPU

* CPU; central processing unit
* GPU; graphics processing unit
  + NVIDIA vs. AMD
  + In deep learning, mostly NVIDIA GPU is used
* Difference?
  + CPU: fewer cores, but each core is much faster and much more capable(works independently); great at sequential tasks
  + GPU: more cores, but each core is much slower and “dumber”; great for parallel tasks
  + CPU has cash memories, but it is relatively small.
  + GPU has own RAM and own hierarchy.
  + GPU is good for matrix multiplication(parallel work).



Programming GPUs

* CUDA (NVIDIA only)
  + Write C-like code that runs directly on the GPU
  + Higher-level APIs; cuBLAS, cuFFT, **cuDNN**, etc
* OpenCL
  + Similar to CUDA, runs on anything
  + Usually slower
* Udacity: intro to parallel programming
  + For deep learning just use existing libraries
* cuDNN has about 2~3 times better performance than naïve CUDA

Deep learning software

* Caffe (UC Berkeley) -> Caffe2 (Facebook)
* Torch (NYU/Facebook) -> PyTorch (Facebook)
* Theano (U Montreal) -> TensorFlow (Google)
* Paddle (Baidu)
* CNTK (Microsoft)
* MXNet (Amazon)
* Etc

Point of deep learning frameworks

* Quick to develop and test new ideas
* Automatically compute gradients
* Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

TensorFlow: neural net (Pre-2.0 ver.)

* First, define computational graph. Then run the graph many times.
* Create placeholders for input x, weights w1, w2 and targets y.
  + Different at TensorFlow 2.0 version
* Forward pass: compute prediction for y and loss(L2 distance)
  + This phase is just building graph, so there is no computation
* Tell TF to compute loss of gradient with respect w1 and w2
  + No computation
* Enter a session and actually run the graph
  + Create numpy arrays to fill the placeholders
* Run the graph
  + Feed in the numpy arrays for x, y, w1, w2
  + Get numpy arrays for loss, grad\_w1, grad\_w2
* Train the network
  + Run the graph over and over, use gradients to update weights
* Problem
  + Copying weights between CPU/GPU each step
* Solution
  + We can change w1, w2 to variable which remains in graph(GPU)
  + Add assign operations to update w1, w2 as part of the graph
* Keras: high-level wrapper

PyTorch

* Three levels of abstraction
  + Tensor: imperative ndarray, but runs on GPU -> same to numpy array of TF
  + Variable: node in a computational graph; stores data and gradient -> tensor, variablr, placeholder of TF
  + Module: a neural network layer; may store state or learnable weights -> tf.layers, TFSlim, TFLearn, Sonnet, … of TF
* Major difference with TensorFlow
  + PyTorch runs on GPU

Static vs. Dynamic graphs

* TensorFlow: build graph once, then run many time (static)
* PyTorch: each forward pass defines a new graph (dynamic)
* With static graphs, framework can optimize the graph before it runs
* Serialization
  + Static: Once graph is built, can serialize it and run it without the code that built the graph
  + Dynamic: Graph building and execution are intertwined, so always need to keep code around.
* Conditional & loops
  + Very simple for dynamic graph(PyTorch)
  + For static, we build graph only once, so we need special conditional case graphs. Almost making new language.

Dynamic graph applications

* Recurrent networks
* Recursive networks
* Modular networks

Caffe

* Core written with C++, has python & matlab binding