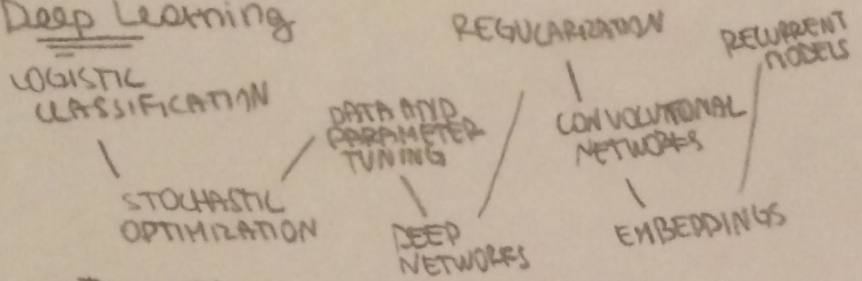
Deep Learning



TensorFlow on IPython notebooks

Neural networks

A lot of important work on that happened in the 80's and in the 90's. Not many real world applications were found and they completely disappeared in the 00's.
Only in the last few years neural networks made a big comeback thanks to lots of data and fast GPU

Supervised classification

classification is the task of taking an input and giving it a label

you have lots of examples called the training set that have already been classified

Given a new example, you have to figure out to which class it belongs to

classification, or prediction, is the central building block of machine learning

Logistic classifier (or linear)

$WX + b = Y$ ← prediction
weights → W, input vector → X, biased term → b

Training Finds W and b to get the best Y

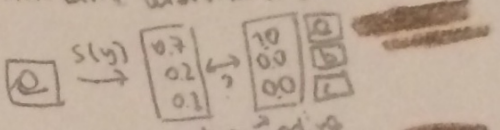
$WX + b = Y$ (X)

$\begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$ → SOFTMAX → $\begin{bmatrix} p=0.7 \\ p=0.2 \\ p=0.1 \end{bmatrix}$ (a), (b), (c)
(scores → probs)

scores are also called logits

$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$ ← softmax

IF you multiply scores by 10, the probabilities will get close to 1 either 0 or 1. Dividing by 10 makes the probabilities closer to uniform distribution



one-hot encoding
1.0 for the correct class

that works well until you have too many classes

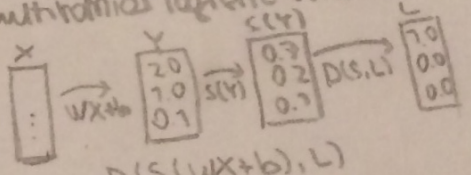
Cross entropy

measures the distance between those two probability vectors

$D(S,L) = -\sum_i L_i \log(S_i)$

for distance

Multinomial logistic classification



$D(S(WX+b), L)$

We need to find W and b such that we have a low distance for the correct class and a high distance for the incorrect class

training loss → $\frac{1}{N}\sum_i D(S(Wx_i+b), L_i)$

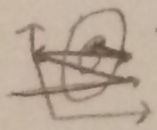
measure D averaged over the entire training set
we want the loss to be small

To find the weights that causes the loss to be the smallest we use gradient descent

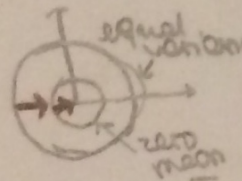
$-\alpha \Delta f_{(W\ldots)}$

Adding really small values to a very large value can introduce errors in python

MEAN $X_i = 0$ ← we want our variables to have that whenever possible
VARIANCE $\sigma(X_i) = \sigma(X_j)$



the optimizer has to do a lot of searching to go and find a good solution

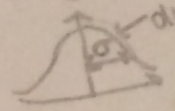


This makes it a lot easier for the optimizer to do its job

Images

$\frac{R-128}{128}$ $\frac{G-128}{128}$ $\frac{B-128}{128}$

weight initialization



draw weights randomly from here. σ determines the order of magnitude of the outputs at the initial point of the optimizer

It's usually better to begin with a low sigma (distribution uncertain about things)

$w \leftarrow w - \alpha \Delta \frac{\delta}{w}$
$b \leftarrow b - \alpha \Delta \frac{\delta}{b}$

until we reach a minimum loss function

Measuring performance

Run classifier in your images with labels and see how many it gets right.

That's my error measure.

Then you try your classifier on images it has never seen and your performance is not as good.

Imagine the classifier compares the new image with any of the other already seen images and gives you back the label

Every classifier you'll build will tend to try and memorize the training sets, it will usually do that very well - you want to help it generalize to new data

Take a small subset of the training set. Don't use it in training. Then measure the error on that test data

Training a classifier is usually a process of trial and error.

Train a classifier, measure its performance. Then you try another one and measure again. And repeat by tweaking the model, exploring the parameters, measure, until you have a nice classifier.

Then you'll deploy your system in a real production environment and you get more data, and you scale your performance on that and it doesn't do nearly as well.

Your classifier has memorized the test data through your decision about which classifier to use & which parameter to tune you gave information to your classifier about the test set.

Take another chunk of your training set and never use it until you have made your final decision.

You can use the validation set to measure your actual error

The bigger your test set, the less noisy your accuracy measurement will be

A change that affects 30 examples of your validation set, it's usually significant and can be trusted.

validation set size > 30 000

↑ means accuracy figures significant to the 10^{-2} and gives you enough resolution to see small improvements

IF some important classes are very rare, this heuristic is no longer good

[redacted] this might be too much data if you have a small training set. cross-validation is a way to mitigate this problem, but it can be slow so getting more data is often better

—

The problem with scaling gradient descent is that you need $-\alpha \Delta \xi(w\ldots)$.

IF $\xi = \sum D_i$ takes n floating point operations, computing its gradient $(-\alpha \Delta \xi(w\ldots))$ takes three times that compute.

ξ depends on every element in the training set. That can be a lot of compute.

Since gradient descent is iterative, [redacted] you have to compute that hundreds of times.

Instead of computing the loss, we're going compute an estimate of it (a really bad one)

The estimate is computing the average loss for a small random fraction of the training data. Between 1 and 1000 training examples.

IF the way you pick examples isn't random enough it no longer works [redacted]

So compute loss and derivative for that example and pretend that the result is the right [redacted] direction to walk to do gradient descent

It is not the right direction, and at times it might increase the real loss. But we compensate by doing this many times.

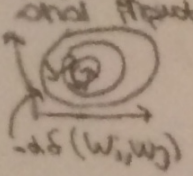
[redacted] doing small steps, each will be cheaper to compute. We take many more steps.

This is vastly more efficient than doing gradient descent.

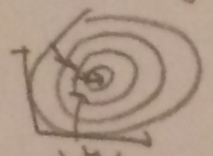
This is called stochastic gradient descent and it scales well with data [redacted] and model size

Helping SGD

To help SGD you want 0 mean and equal variance for initial weights (picked at random) and inputs

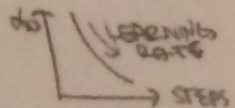


$-\alpha\delta(w_i, w_j)$

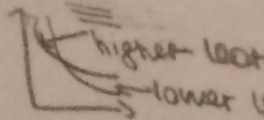


$-\alpha\Pi(w_i, w_j)$

$\Pi = 0.9\Pi + \Delta\delta$ ← the running average



It's beneficial to make steps smaller as you train

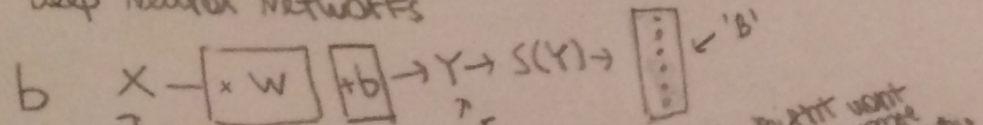


"never trust how quickly you learn it has often little to do with how well you learn"

SGD has many hyper-parameters you can play with IF things don't work try to lower the learning rate First

ADAGRAD takes care of some hyper parameters for you, however it's often worse than well tuned SGD

Deep Neural Networks



(N+1)K PARAMETERS ← you might want to use many more parameters in practice

Also it's linear

$Y = X_i + X_j$ ✓ equivalent with matrix

$Y = X_i \times X_j$ ✗

↓

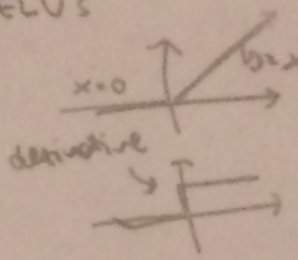
this is what GPUs are for

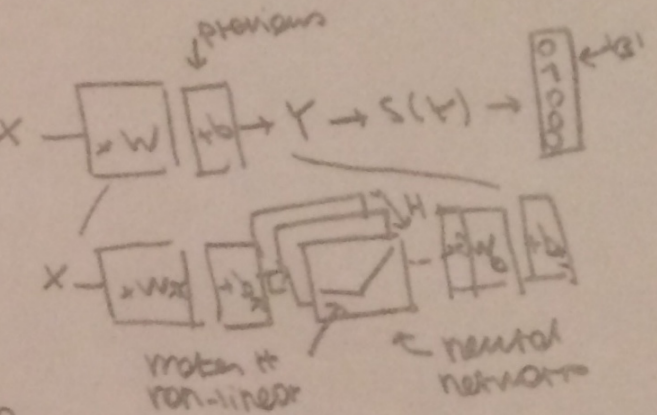
Linear operations are really stable, small changes in the input can never result in big changes in the output

The derivative of a linear function is constant, so also stable

We would like our parameters inside a big linear function, but we would also want the model to be non-linear

RELUs





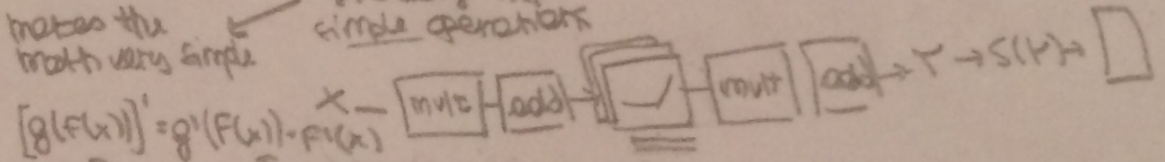
stacking up simple operations

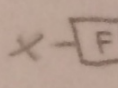
makes the math very simple

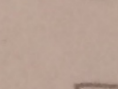
$[g(f(x))]' = g'(f(x)) \cdot f'(x)$

derivative of product

chain rule!

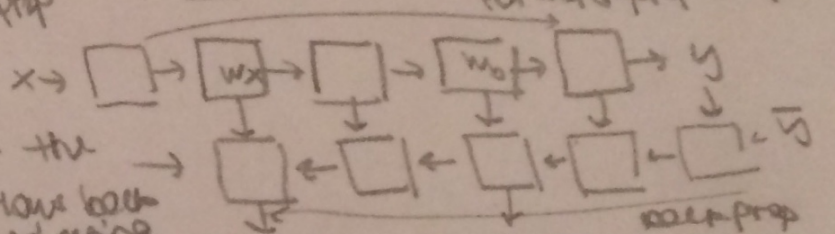






Backprop

(made up of simple blocks)



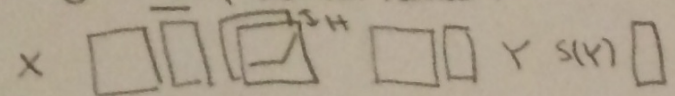
To compute the derivatives data that flows back gets combined using the chain rule. that graph can be derived from the individual operations in your network and frameworks will do it for you

$\Delta W_x \leftarrow$ gradients ΔW_0

$W_x \leftarrow W_x - \alpha \Delta W_x$ $W_0 \leftarrow W_0 - \alpha \Delta W_0$

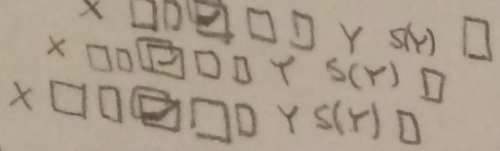
[redacted]

Training a deep neural network



You can make this neural network more complex big increasing H. But doing that is not really efficient in general and it gets really hard to train.

Instead you can add more layers and make your model deeper



You can typically get much more performance by going deeper rather than wider.

Also, many natural phenomena have an hierarchical structure, which deep models naturally capture

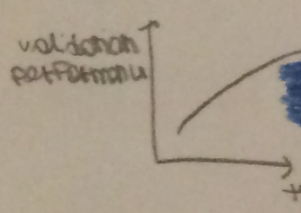
If a model for images usually has lines or edges at the lowest layers and by moving up these have more complicated things like geometric shapes.

Further up these objects

The model matches the abstraction you expect to see in your data and has an easier time learning

Regularization

When doing numerical optimization, the network that's the right size for your data is very hard to optimize, so we always try networks that are way too big for our data, and then we try our best to prevent them from overfitting (training loss problem)



early termination prevent from overfitting

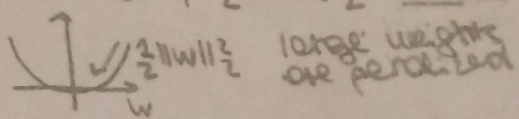
Another way is to apply regularization..

regularizing means applying artificial constraints on your network that implicitly reduce the number of free parameters while not making it more difficult to optimize

stretch pants: they fit just as well (as skinny jeans), but because they flexible they are not harder to fit in

L2 regularization

$$\xi' = \xi + \beta \frac{1}{2}\|w\|_2^2$$

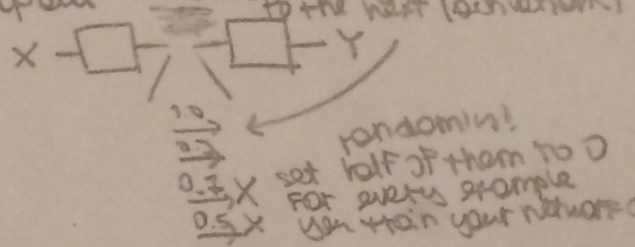


Because it's added to ξ the structure of your network doesn't have to change

$$\beta \frac{1}{2}\|w\|_2^2 \rightarrow \frac{1}{2}(w_1^2 + w_2^2 + w_3^2 + w_4^2 + w_5^2)$$

whose derivative is w

Dropout



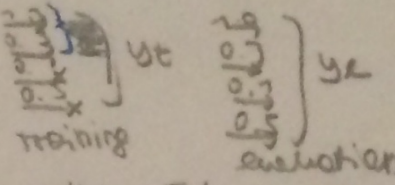
The network can never rely on any given activation to be present, so it is forced to learn a redundant representation for everything to make sure that at least some of the information remains

Makes things more robust and prevents overfitting.

It also makes your network act as a consensus over an ensemble of networks

When evaluating the network trained with dropouts you longer want that randomness.

Take the consensus over these models by averaging the activations



$y_e \sim E(y_t)$

to make sure this holds scale the remaining activation by a factor of 2

This way, when evaluating, you just remove these dropouts and scaling operations

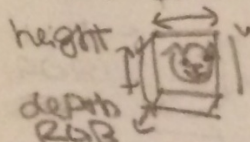
If your network doesn't have to learn the structure of your data from scratch, it is going to perform better

If want your network to see an image has a cat in it. You can tell it that objects in images are largely the same whether they're in any position (translation invariance)

weight sharing: when you know that two inputs can contain the same kind of information, then you share their weights and train the weights jointly for those inputs

Spatial invariants are things that don't change on average across time or space

Convolutional Networks (or convnets) store parameters across space



→ Given an image, run a neural net with n outputs on it and slide it across the image.
The output is another image with different height, width, depth (instead of RGB we now have n other channels)

A convolution

If your patch was the size of the image it wouldn't be different than a layer of neural net

A smaller patch means fewer weights shared across space

A convnet will be a neural network where instead of stacks of matrix multiply layers we will have stacks of convolutions.

It will form a pyramid of smaller but thinner layer going up

256×256×3 → 128×128×16 → 64×64×32 → 32×32×64

The depth corresponds (roughly) to the semantic complexity of your representation

At top you can put your classifier so you have a representation where spatial information has been squeezed out and only parameters that map to content of the image remain

- patch is also called kernel
- each cell is a feature map
- stride is how many pixels you shift each time you move your filter. When 2, the output is roughly a third of the input.

valid padding is when you don't go off the edge

If you go off the edge and pad with 0's such that the output image is the same size as the input image, is same padding

In all cases the depth is the same

An improvement is a better way to reduce the spatial extent of your feature maps in the convolutional pyramid through a method called pooling

Striding is very aggressive removes a lot of information

We can use a small stride and then take all the convolutions in a neighborhood and combine them somehow

That is called pooling.

The most common is looking at the max of all the neighborhood around a point in the feature map

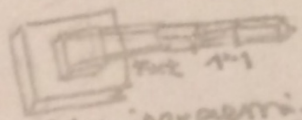
max pooling
- parameter-free, not overfitting
- often more accurate models
- more expensive (lower stride)
- larger parameters needed

typical architecture is

IMAGE → CONV → MAX POOL → CONV → MAX POOL → FULLY CONN → FULLY CONN → CLASSIFIER

average pooling: providing a blurred view of the image

1×1 convolutions, looking at 1 pixel



thin network instead of a linear classifier

very inexpensive way to make your models deeper and have more parameters

Inception module

. . .

Convnets take a lot of compute and require GPUs

Deep models for text

Often we different words to mean almost the same thing

KITTY CAT — when things are similar we'd like to share parameters between them

we have to learn that they are related between them

We would also like to see a word often enough that we automatically learn the meanings

That would require a lot of labelling. Use unsupervised learning instead

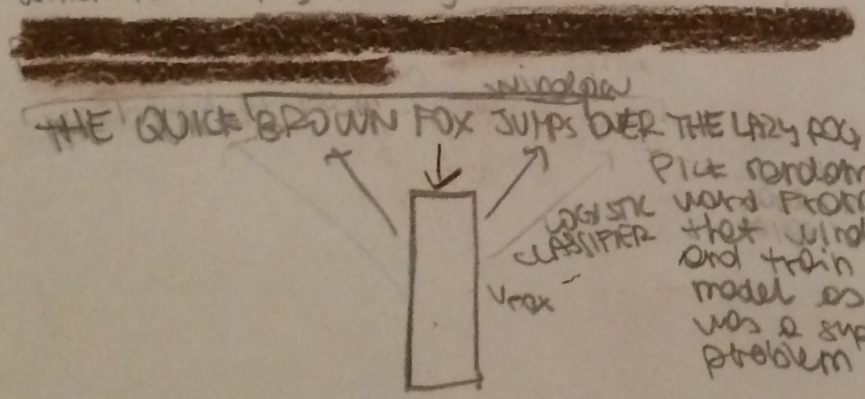
Similar words tend to occur in similar contexts

THE CAT PURRS
THIS CAT HUNTS MICE — it's reasonable to use KITTY

We map words to small vectors called embeddings which are going to be close to each other when words have similar meanings and apart when they don't

Word2Vec

Given text, map each word to an embedding, initialize a random one, and the use that embedding to try and predict context which is simply nearby words



To see that the embeddings are clustering together we can we can do a nearest neighbour lookup of the words closest to any given word

Another way is to reduce the dimensionality of the embedding to two dimensions and to plug the two dimensional representation

The naive way (eg PCA) loses too much information

t-SNE preserves the neighbours in input data

It's better to measure closeness with cosine instead of L2 as the length of the vector is not relevant for the classification

COSINE — $\frac{V_{CAT} \cdot V_{KITTEN}}{\|V_{CAT}\| \|V_{KITTEN}\|}$ — it's good to normalize embedding vectors to unit norm ($\frac{V_{CAT}}{\|V_{CAT}\|}$)

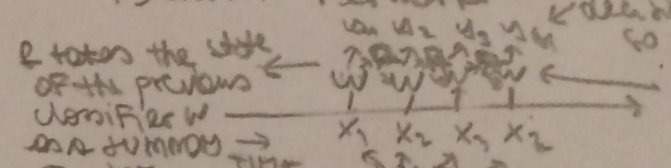
The softmax is really inefficient if done for every word in our vocabulary but you can sample the words that are not the target (sampled softmax)

Semantic Analogy — puppy → dog / kitten → cat

Syntactic Analogy — toilet → toilet / flutter → flutter

$V_{puppy} - V_{dog} + V_{cat} = V_{kitten}$

Recurrent Networks



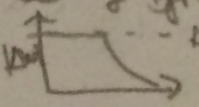
To compute the parameter updates we backpropagate the derivative through time to x_0 (beginning)

These derivatives are going to be applied to the same parameters (a lot of correlated updates)

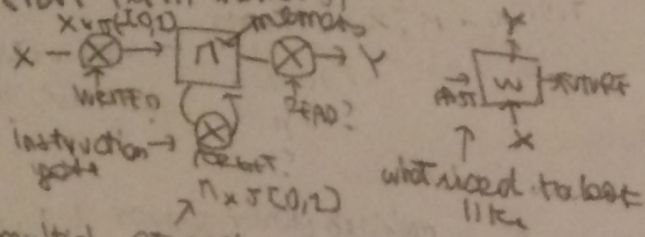
This is bad if SGD that creates correlated updates either the gradients grow exponentially or go down to 0 really quickly (vanishing gradients) (exploding gradients)

Gradient clipping for exploding gradients

$\Delta W = \Delta W \frac{\Delta max}{\max(\|\Delta W\|, \Delta max)}$



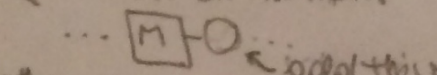
LSTM for vanishing gradients
(long short term memory)



multiply at each gate. $\sigma = 0$ means no input passes

σ is continuous, differentiable, backprop possible

the gating values gets controlled by a tiny logistic regression on the input



The gates help the model keep the memory longer when it needs to

---

regularization with L2 works
dropouts work as long as you use it on X or Y
input output

You can use RNN to predict next thing in your sequence
For that you take the sequence at time t, generate the predictions and then you sample from the predicted distribution, Feed that sample to the next step
just sampling the next prediction is very greedy.
Better way is to just sample 100 times, then you have multiple sequences (or hypotheses) that you could continue predicting from at every step and choose the best sequence out of those in computing the total probability of the characters generated
That way you prevent the sampling from making one bad choice and being stuck with that
Beam search - only keep the most likely few candidates at every step and prune the rest

RNNs not variable length sequences to fixed length vectors thanks to sequence generation discussed then go the other way