More Neural Networks

Siva Balakrishnan
Data Mining: 36-462/36-662

April 23th, 2019

Outline for Today

- Just a high-level overview of more neural network architectures, cool tricks to get them to behave and applications

Plan for the rest of the semester

- Rest of the lectures (including today) won't be on your final
- Next class, i.e. on Thursday we'll review the second half of the course (mainly dimension reduction, clustering)
- HW will go out today, and be due on Wednesday at midnight
- Remember the project deadlines (i.e. next Tuesday predictions are due, and next Friday write-up is due)

Recap: Neural Network Motivation

- One of the most challenging aspects of designing good classifiers is coming up with good/useful features, i.e. transformations/representations of the input that make learning a good classifier easy.
- Neural networks, roughly, try to *learn* useful transformations of the data that are useful. Surprisingly, this often works (at least when you have enough data, enough compute, and know what you are doing).

Recap: Neural Net Basics

Suppose we want to use a representation:

learned features.

$$y = \phi(x;\theta)^T\beta = \sum_{j=1}^{p'} \underbrace{\phi_j(x;\theta)}_{\text{learned features}} \beta_j$$

We cannot use linear ϕ_j since that just re-creates a linear model (not interesting).

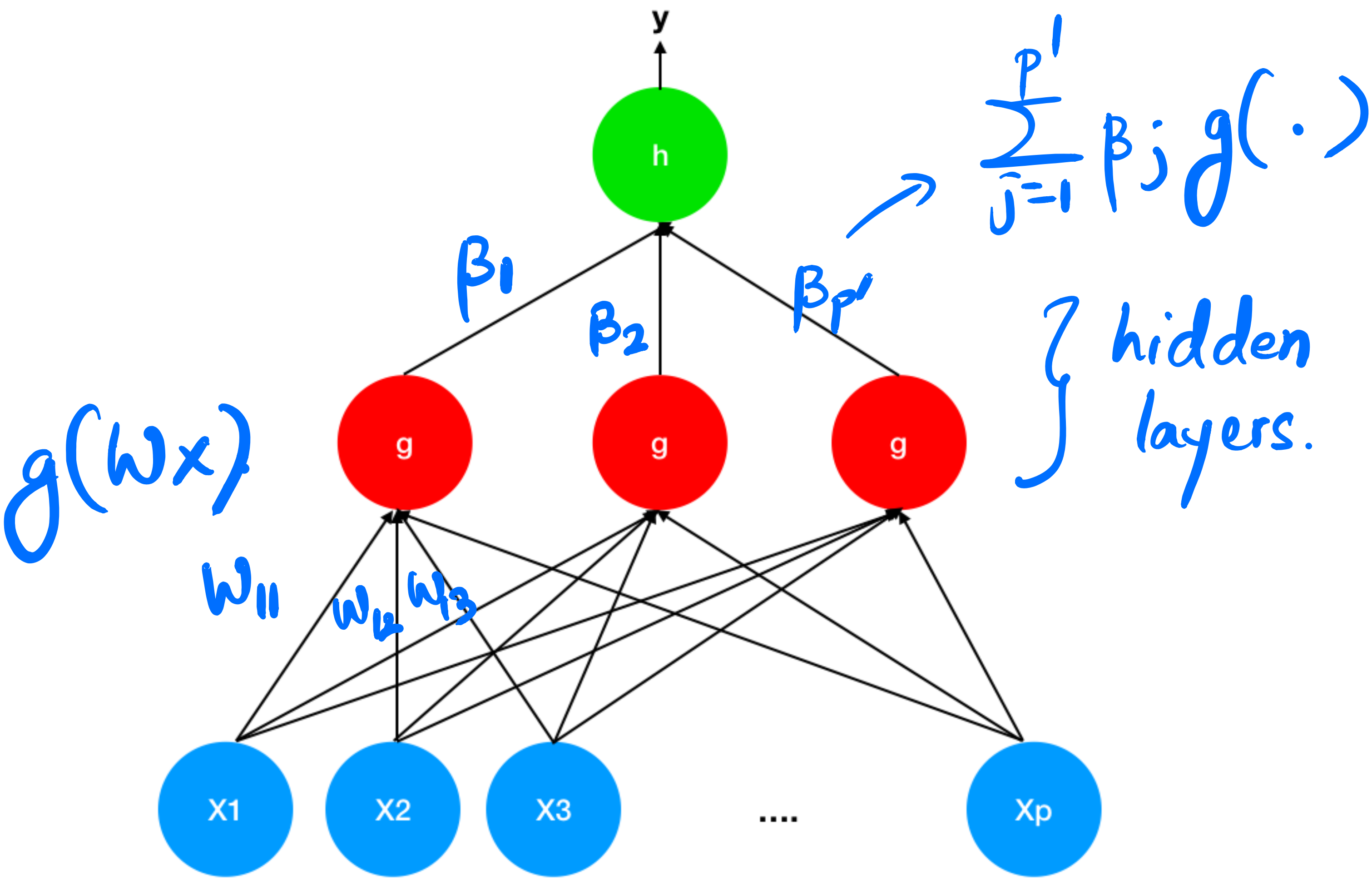
So we instead use:

apply some non-linearity

$$\phi_j(x;\theta) = g(x^T\theta_j + c)$$

where g is some non-linear function (often sigmoid, or ReLU).

Recap: Representing Classifiers/Regressors as Networks

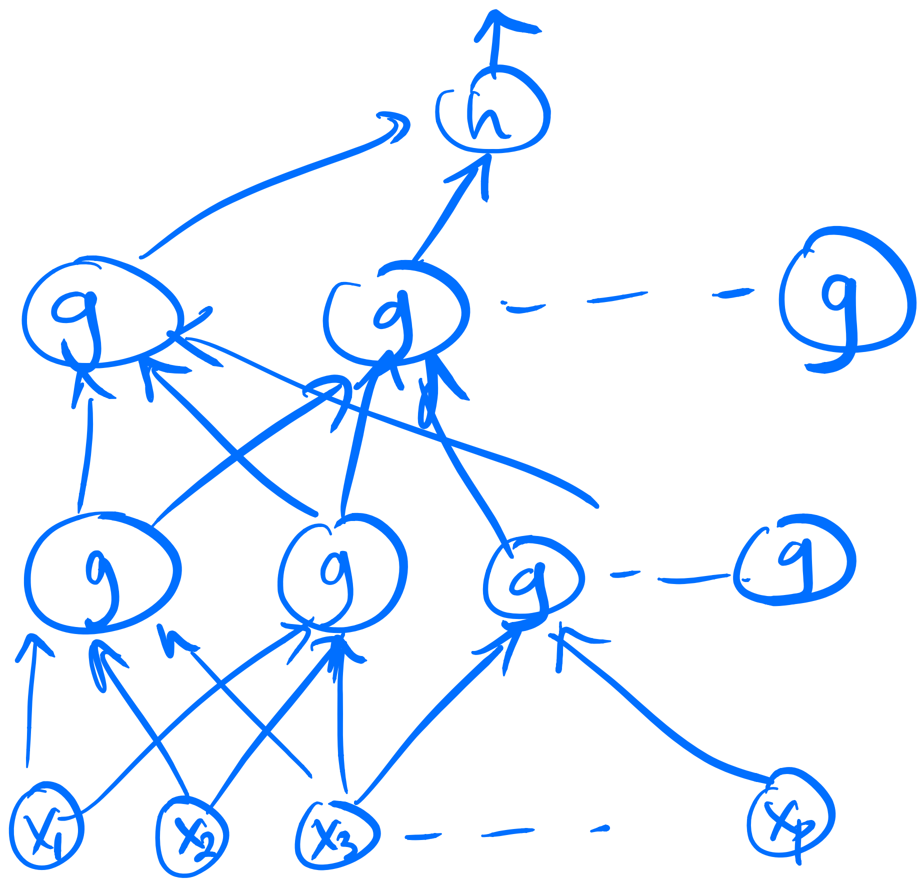


Recap: *Deep* Neural Networks

To get more complex features, we can just recurse! Build more layers of features, each of which is a non-linear function of a linear transformation of the previous.

linear transf + non-linearity → first "learned" features

This is the basic idea of *feed-forward neural networks* or *multilayer perceptrons*.



Some complicated function of inputs $X_1 \dots X_p$.

Recap: Already Lots of choices

This is a very flexible architecture.

→ change non-linearity

→ widths of layers.

→ depth of network.

→ choose to make different connections.

Recap: Outputs

We can stick a bunch of final functions on top to get different outputs. Let h be the output of the final layer.

- Continuous outcome: just use a $w^T h + b$ linear function! ← single y.
- Binary outcome: Just use logistic

$$\frac{1}{1+e^{-w^T h}} =$$

- Multiple categories: use multinomial-logistic, i.e. we produce K outputs of the form:

$$y_i = \frac{e^{w_i^T h}}{\sum_j e^{w_j^T h}}.$$

Recap: Fitting the Model

The basic idea:

- We use a loss function to measure how well we are fitting, and then try to find weights that make our loss small (back-propagation) .

training data o/p

For continuous outputs, we might look $\|y - \widehat{y}\|_2^2$.

o/p of network

For binary or multiple category outputs, we might look at the log likelihood of y_i:

$$\text{Logistic:} \quad \begin{cases} \log\left(\frac{1}{1+e^{-w^T h}}\right) & y_i = 1 \\ \log\left(\frac{e^{-w^T h}}{1+e^{-w^T h}}\right) & y_i = 0 \end{cases}$$

$$\text{Multinomial:} \quad \log\left(\frac{e^{w_{y_i}^T h}}{\sum_j e^{w_j^T h}}\right)$$

Roughly, get the output of the network to match the y we are trying to predict, on the training data.

Summary so far

- (Deep) Neural networks are a flexible class of non-linear functions. We can fit them to data by minimizing some loss function using gradient descent (back-propagation).
- Current research broadly focuses on three questions (that we can briefly ponder):
  1. **Representation:** Lots of classifiers we have learned (trees, boosting, nearest-neighbors, kernel SVMs...) all fit non-linear functions to data. So what exactly are these networks *good* at representing and why are they useful?
  2. **Generalization:** They are extraordinarily flexible classifiers. People have shown for instance that using (sufficiently big) neural networks we can fit random noise labels (and get 0 training error).
     Understanding when/why these networks don't overfit and how to regularize them is important (we'll talk about the basics in a few slides).
  3. **Optimization:** It is not obvious that gradient descent should be able to find us useful weights (think of it like local search). Also, we need some tricks to deal with massive data sets and large networks.

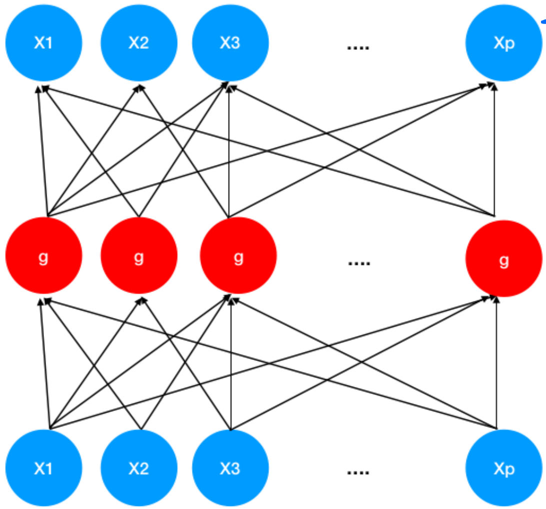
how do we ensure they don't overfit.

Cool Idea 1: How do we do unsupervised learning with neural networks?

- Suppose we want to do either clustering or dimension reduction using a neural network.
- We do not have any y values to fit our network to.
- **Seemingly stupid idea:** Lets pretend our input is also our output. This is called an autoencoder.

→ don't have ys.
→ Replace y by input x.

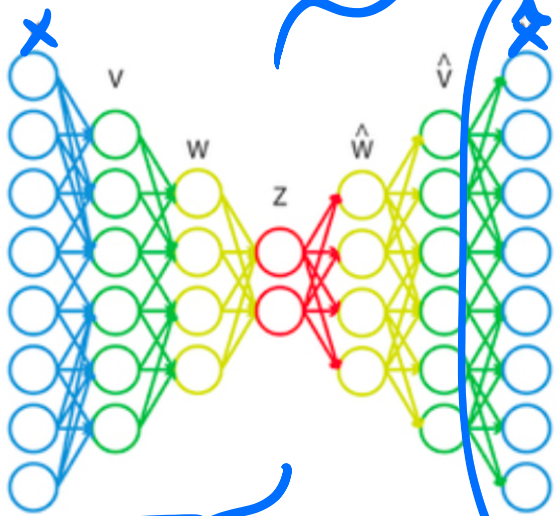
maybe if it's wide enough it will learn the identity.



Cool Idea 1: Autoencoders

- Might not be useful because if the hidden layer is wide enough then we can essentially learn the identity map.
- What if it is not wide enough?

decoder.



fit weights in the usual way

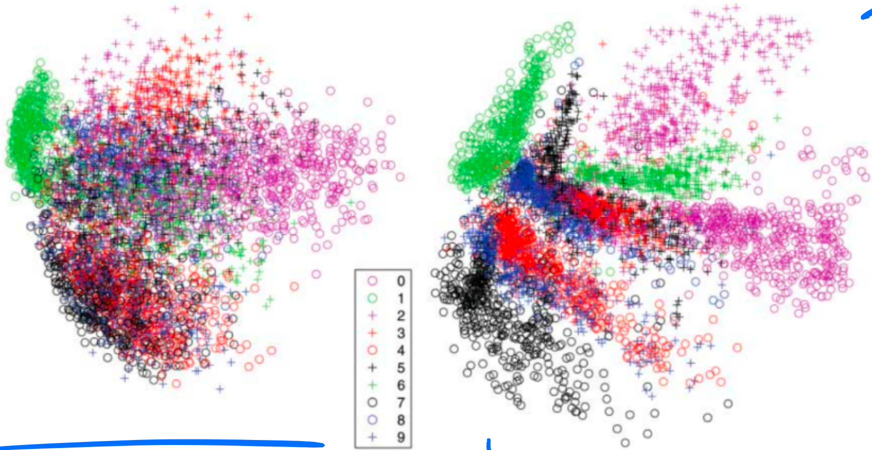
encoder

- We have forced the network to do dimension reduction for us. We just use the representation Z for visualization.

Cool Idea 1: Autoencoders

- An intriguing picture: PCA (left) versus an autoencoder trained on the MNIST dataset.

also does not use labels.



run a clustering algorithm

- Incidentally, autoencoders are also used for denoising.
- How would you use them for clustering?

Switching Gears: Regularization

In real problems, you're going to be looking at problems with many millions of parameters.

We definitely need regularization to avoid overfitting!

One approach is to incorporate regularization on the weights directly into the loss. One might add on

weights from first layer

weight decay

$$\lambda \left(\underbrace{\|W^{(1)}\|_2^2}_{} + \underbrace{\|W^{(2)}\|_2^2}_{} \right)$$

wts from second layer

to the penalty to help regularize all the layer weights.

These L_2 (ridge) penalties are most common, but L_1 (lasso) penalties are also used for sparse weights.

Switching Gears: Regularization

In real problems, you're going to be looking at problems with many millions of parameters.

We definitely need regularization to avoid overfitting!

One approach is to incorporate regularization on the weights directly into the loss. One might add on

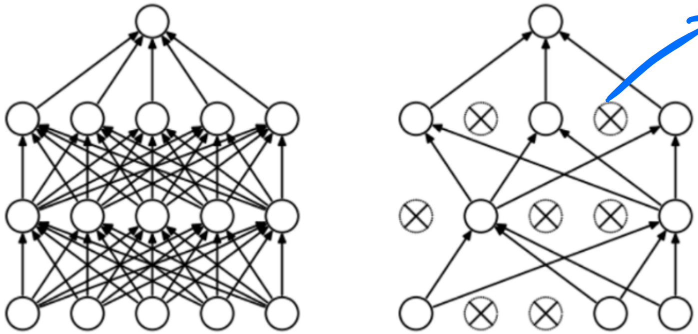
$$\lambda \left(\|W^{(1)}\|_2^2 + \|W^{(2)}\|_2^2 \right)$$

to the penalty to help regularize all the layer weights.

These L_2 (ridge) penalties are most common, but L_1 (lasso) penalties are also used for sparse weights.
In NNs this is called *weight decay*.

Weird Idea 1: Dropout

We saw in random forests that subsampling features can actually improve stability. A similar effect has been observed in deep learning.



drop out nodes / features.

At each iteration, some nodes (or edges) are hidden.

This tends to improve stability and predictions. Like in a random forest we imagine eventually voting between these different neural networks.

Weird Idea 2: Data augmentation

augment training data with changes to it that shouldn't change label.

When we talked about the digits data, you might have wondered whether we could just generate new data by transforming our existing images: translate, rotate, thicken them a little.

This is very common in deep learning! It provides a larger training set, and also automatically incorporates invariances.

Cool Idea 2: Other architectures

So far, we've been talking about general feed-forward networks with dense connections.

In reality, many different architectures are used for neural networks representing different kinds of problems.

The two most common:

- Convolutional neural networks (CNNs): For regular, grid-like inputs where we want to share some of the basic processing and use local information.
- Recurrent neural networks: Sequential data

We will briefly discuss CNNs.

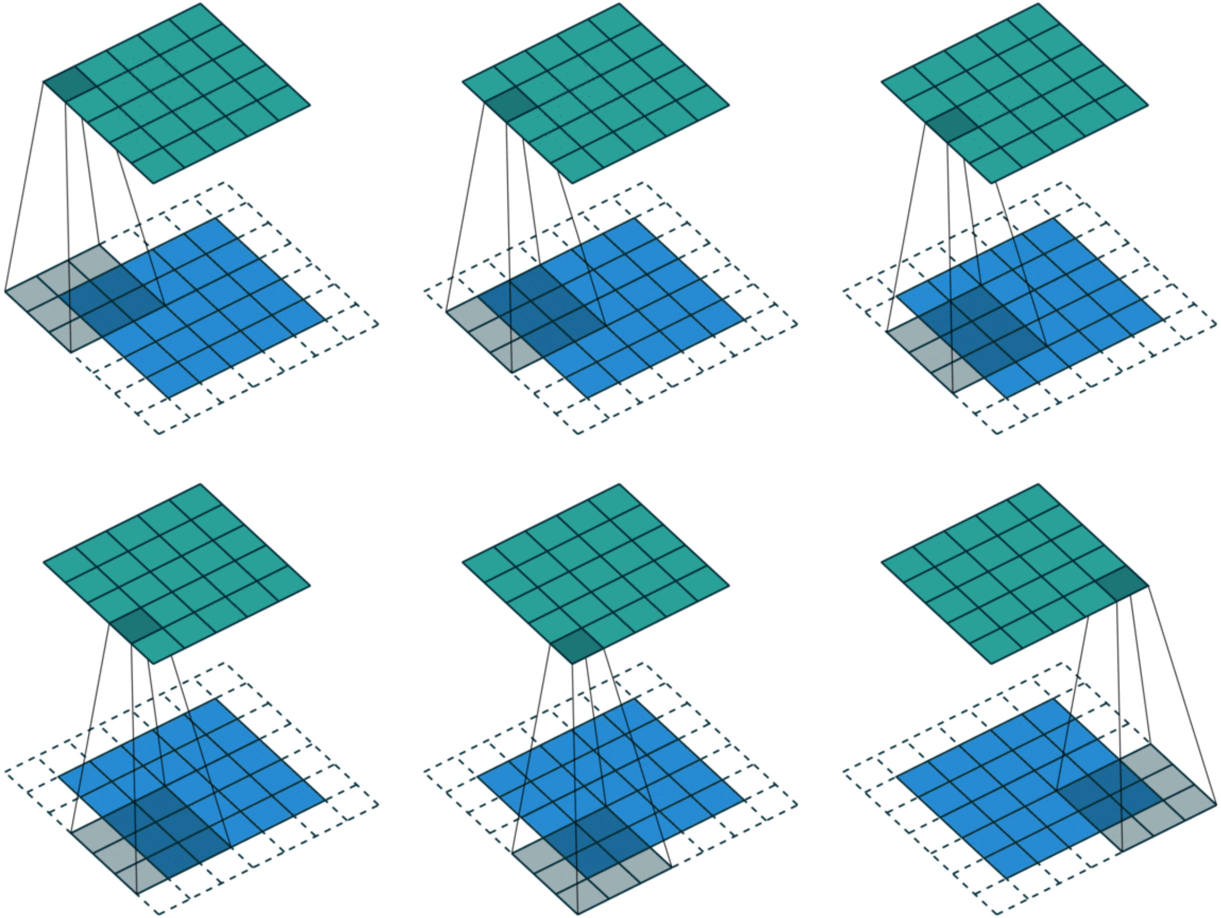
Cool Idea 2: CNNs

In image processing, the early stages of processing are likely similar everywhere on your image: edge detectors, corner detectors, smoothers, etc.

Furthermore, early processing probably shouldn't depend on long-range relationships across your image. I don't need to see the other corner of an image to decide if I'm looking at an edge.

Convolutional neural networks encode these ideas in their architecture. We want to reduce the number of parameters by re-using them in clever ways.

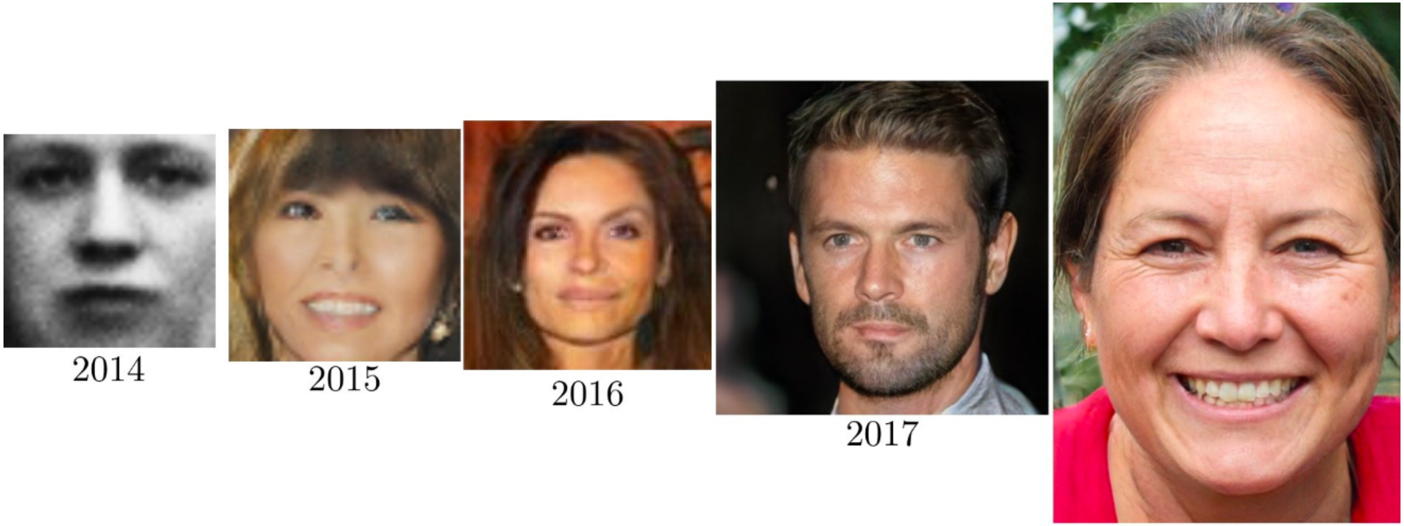
CNNs



Key Idea: Weight Sharing.

Cool Idea 3: Generative Adversarial Networks

Suppose we want to create a generative model, i.e. want to be able to simulate realistic images (possibly with some “features”).



2014 2015 2016 2017 2018

Cool Idea 3: Generative Adversarial Networks

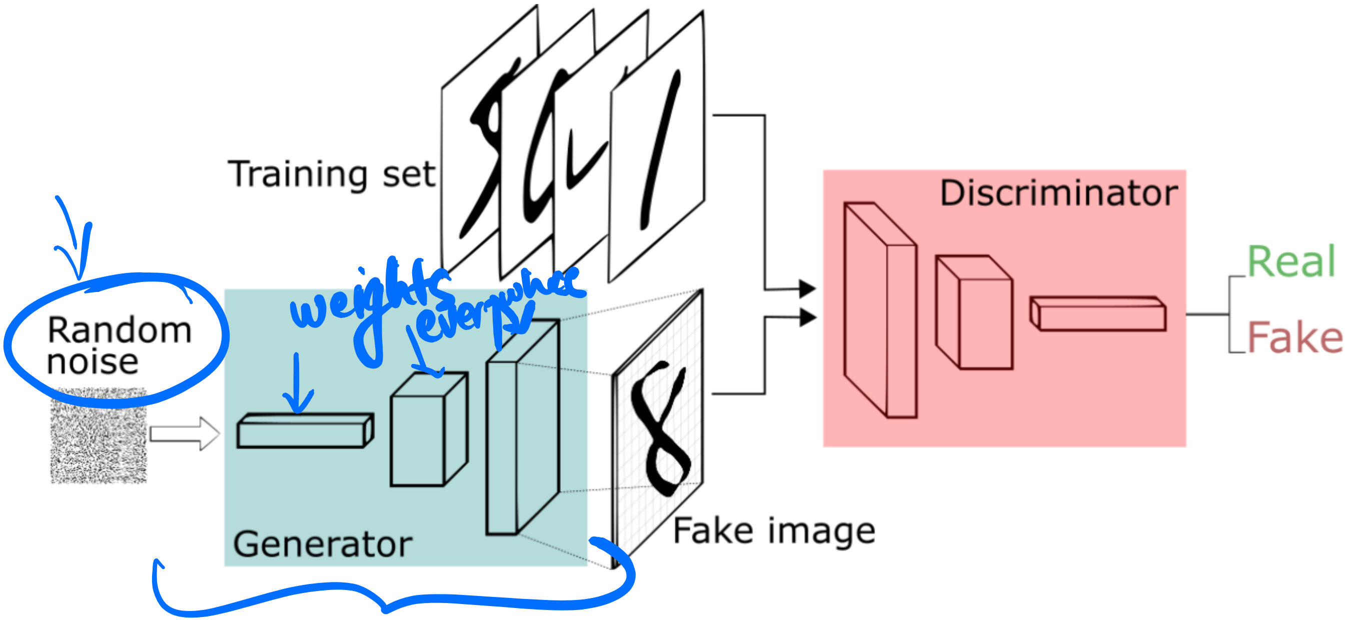
Why?

deepfashion

Cool Idea 3: Generative Adversarial Networks

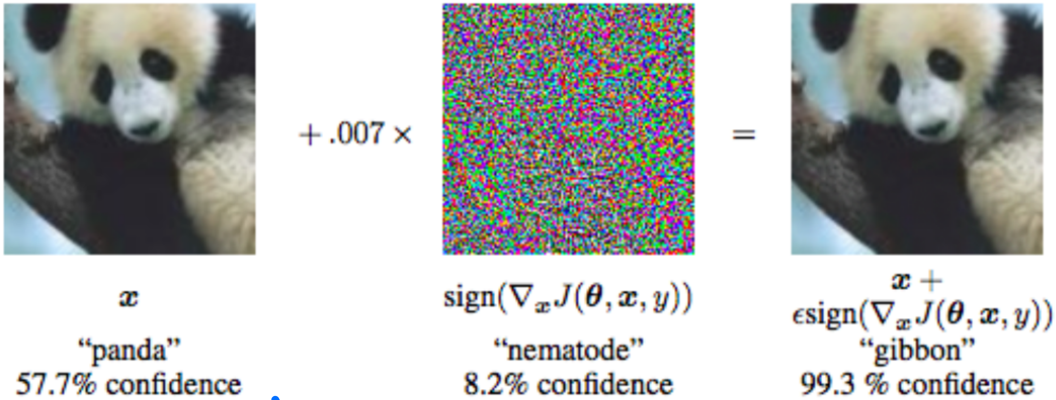
Cool Idea 3: GANs How?

- Statisticians have thought about generative models for 50 years – estimate the distribution and sample from the distribution.
- A completely different idea:



Interesting Observation 4: Adversarial Examples

A now classic example:



fairly sure it's a panda.

- Turns out to not just be a neural network thing, essentially every classifier you have learned about has this problem.
- Why does it matter?

Interesting Observation 4: Adversarial Examples

Why does it matter?

- People have shown you can take stop signs, modify them slightly and have them classified by a neural network as speed limit signs.

"real world" adversarial examples.

Overall, we do not really understand neural networks very well. Despite the fact that they achieve human-level performance on lots of tasks they are brittle and non-human like in many ways (unsurprising). Lots of research to do...

Just a start

This is just a taste of what deep learning looks like. There's a lot of fun to be had learning about this area, but it could easily fill a whole course.

There are several computing packages now, most of which are python friendly. Some of the most popular: Tensorflow, Keras, Torch, Caffe, Theano.

Everything you have learned is about 20 lines of code in Keras (for example).

Lots of books, tutorials, example code out there if you feel like learning more or playing with neural networks (I would be happy to link you to things I like).