Introduction to Deep Learning

Ian J. Watson
ian.james.watson@cern.ch

University of Seoul

Yonsei University
October 8, 2020

KRF KOREA RESEARCH FELLOWSHIP
해외 우수신진연구자 유치사업

Slides

- Slides available from (this will redirect to my github repo.):

https://git.io/yonsei-slides-2020

- Other Resources:
  - Pytorch documentation: https://pytorch.org/
  - List of resources from tech republic: https://www.techrepublic.com/article/how-to-learn-pytorch-a-resources-guide-for-developers/
- Also, an **old** version of this talk using keras (tensorflow): https://git.io/knu-slides-2019

Goals

- If you're already a deep learning master, not the talk for you!
- Get up and running quickly with Deep Learning
  - In particular, the goal is to build neural networks you can take home today!
- Therefore, we'll use *PyTorch* to get up and running quickly
- Outline of the session:
  - Lecture introducing some of the theory
  - Hands-on covering a few projects:
    - Basic Usage of PyTorch (Iris)
    - Convolutional Neural Net in PyTorch (MNIST)
    - GANs (using MNIST)
  - I'll try to cover as much as possible of machine learning and deep learning, this is too much!
- For this lecture, I recommend using *google colaboratory*
  - Machine learning education and research tool setup by google, all the packages are installed, just need a google account to sign in `https://colab.research.google.com`
- If you don't have a google account, please sign up during this lecture (or, check the hands-on to setup your computer if you're comfortable

Introduction - Machine Learning

- **Machine learning** refers to sets of algorithms (techniques) that can "learn" from experience
- Given inputs and expected output can automatically learn to associate patterns in the input to the output and generalize on unseen inputs.
- As opposed to traditional algorithms which are explicitly programmed to always act in specific ways
- Example: Does this signal in my detector correspond to a photon or a hadron? Feed the algorithm thousands of (simulated) photons & hadrons, and it will learn to distinguish them
  - More specifically, we will build a parameterized model which gives a "probability" for an input datapoint to be photon or hadron, and the algorithms changes the parameters to better describe the photons or hadron examples you feed it

Introduction - Deep Learning

- *Deep learning* - **learning** refers to machine learning, **deep** refers to a special property of the technique but generally means the technique operates on much lower level information than older machine learning algorithms, and goes through multiple levels of processing (all automated by the learning algorithms, of course)
- Ex: in the old days (pre-2012), image recognition might proceed by first using specialized edge detectors feeding to specialized shape detectors feeding to older machine learning algorithms operating on high level info. Today, we would just feed raw pixels into a deep learning algorithm and let it figure out the intermediate representations.

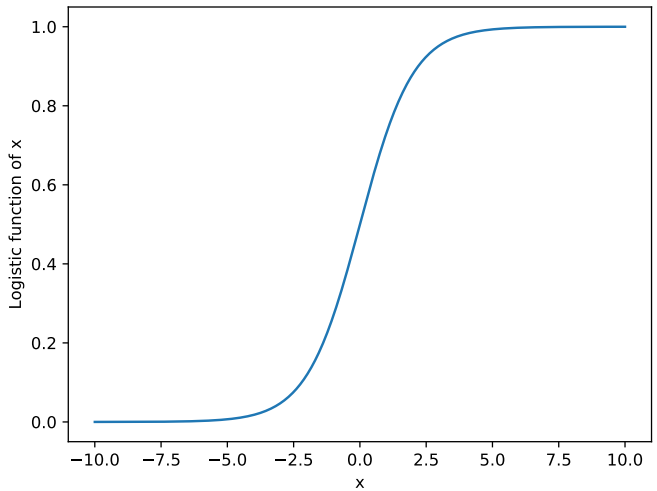
Introduction - Deep Learning Applications

- Typical applications include:
- Image classification: cat vs dog, or photon vs hadron
- Image segmentation: draw a box around all the people in this image, or e.g. all the galaxies in this image from the Hubble telescope
- Voice recognition: "Hey Siri" vs all the other phrases you say around a phone
- Machine translation: this series of utterances in Korean correspond to these in English
- Hopefully, you can see how some of these could be useful to us in science
- Today, we'll mostly be considering image analysis, but the techniques are similar for the other problems

Machine Learning: Neural Network

- Framework for Machine Learning: given a set of data, and set of expected outputs (typically categories), build a system which learns how to connect data to output
- A **Neural Network** is one type: connect stacks of tensor operators with fixed linear and non-linear transformations
- We'll start by building up a neural network classifier from the ground up, this will give us some insight into how machine learning works, and how neural networks find their results

The Logistic Function



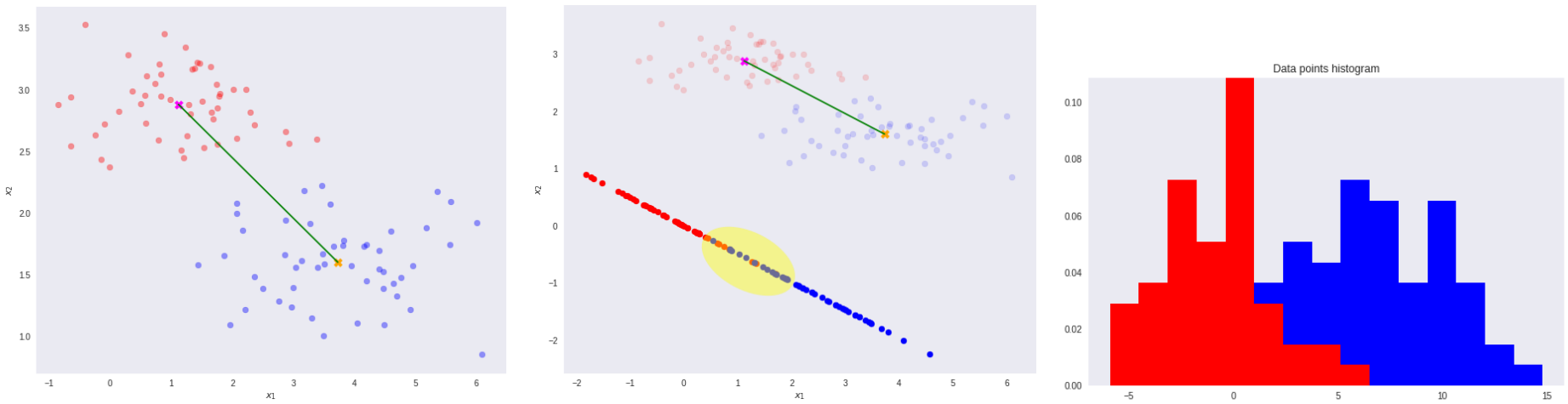
- The logistic (or sigmoid) function is defined as $f(x) = \frac{1}{1+e^{-x}}$
  - Looks like a classic "turn-on" curve
- Concentrate on the case of two classes (cat/dog or electron/photon), and ask what we want from a classifier output
  - We need to distinguish between the two classes using the output:
  - If the value is 0, it represents the classifier identifying one class (cat)
  - If its near 1, the classifier is identifies the other class (dog)
  - Thus, we need to transform the input variables into 1D, then pass through the logistic function
- This is a simple classification technique called *logistic regression*

Logistic regression

- Setup: we have data from two different classes, which can be described by the same independent variables, and we want to distinguish them based on those independent variables
- We want to build a function such that data from one class goes close to 1, from the other close to 0
- We will build a linear function of the variables, then pass it through the logistic function, and try to minimise the distance of data from 0 (for one class), or 1 (for the other)
- $y_i = \sigma(\vec{\beta} \cdot \vec{x_i}) + \epsilon_i$, $y_i = 0$ if x_i from class 0, 1 if x_i from class 1
  - $\vec{\beta} \cdot \vec{x_i} = \beta_0 + \beta_1 x_1 + \ldots \beta_k x_k$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$ the logistic function
- The machine-learning goal would be find the values of β which best classifies the classes

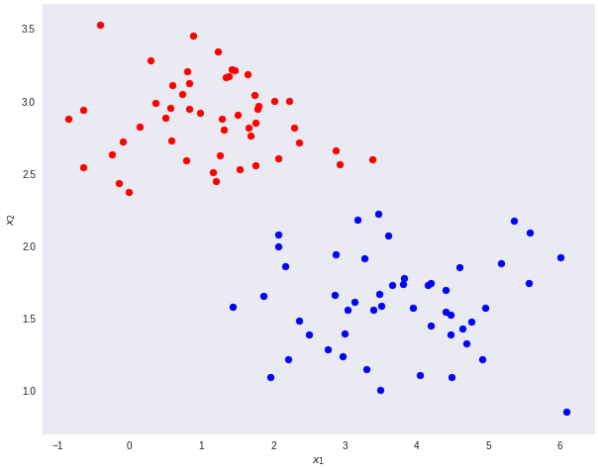
- Note: the logistic function is also called the sigmoid function, logistic curve, turn on curve, etc. depending on the context its used in

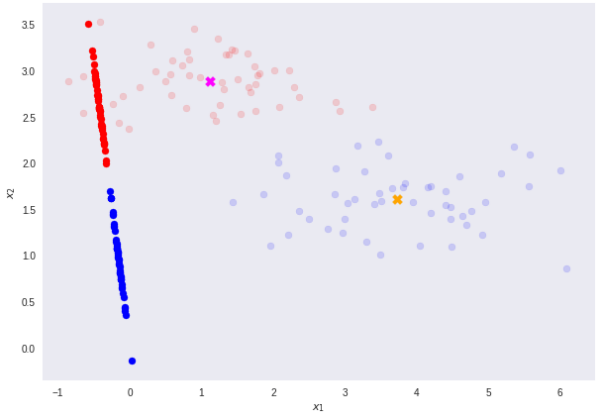
Illustration: 1D Projection

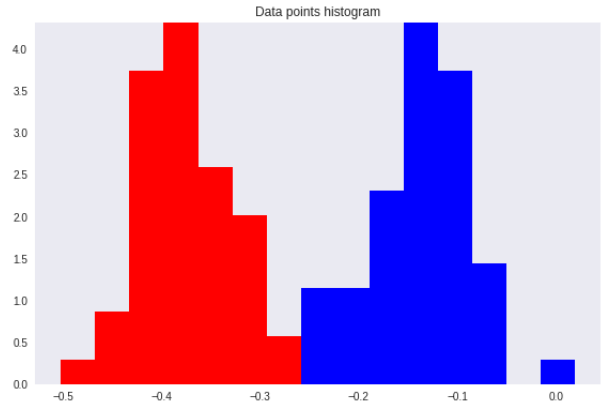


- $\vec{\beta} \cdot \vec{x}$ is a projection of the data onto a line
- Red and blue are two classes which can be measured in (x_1, x_2)
- We can take the mean of each class (left), form a line between, then project the data onto the line (middle) giving a distribution (right)
  - We have reduced the 2D data into a 1D projection
- After the projection, the logistic rejection chooses a cut point (via β_0) then sends things below the cut to 0, above to 1
- Here, we see some separation between the classes but a lot of overlap. We can do better

Illustration: Better Fit



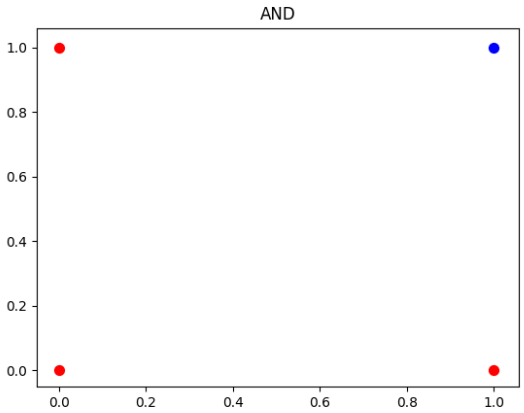


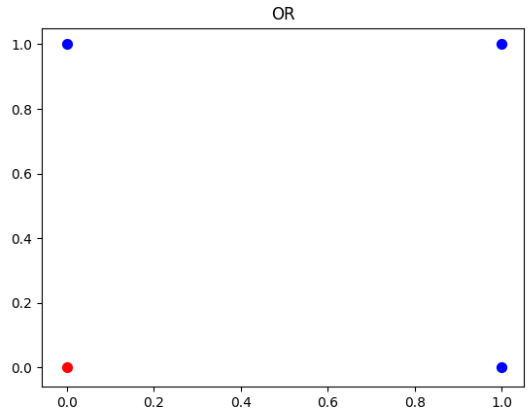


- Finding the best discriminant for our illustrative dataset shows that these two classes are fully separable
  - Find β which minimizes a **loss function** which gives a penalty for misclassifying data: $MSE = \frac{1}{m}\sum_i \left(f(x_i;\beta) - y_i\right)^2$ (the Mean-Squared Error loss), we want to **minimize the loss** over parameters β
  - This is the usual goal in ML: setup a parameterized model and then define a function to minimize which gives the best parameters
- The Logistic Regression will place the cut point between the data and so all red go to 0, blue go to 1 after passing through the logistic function

From
https://medium.freecodecamp.org/an-illustrative-introduction-to-fishers-linear-discriminant-9484efee15ac

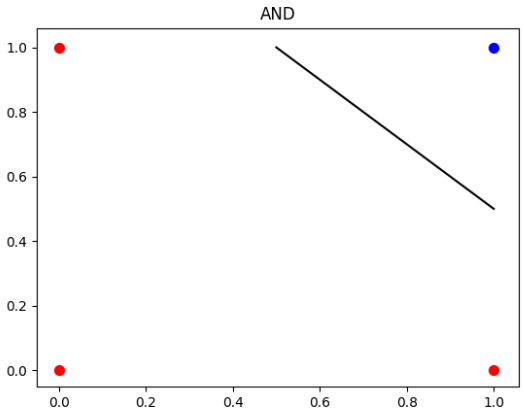
Some very simple examples for simple logistic regression

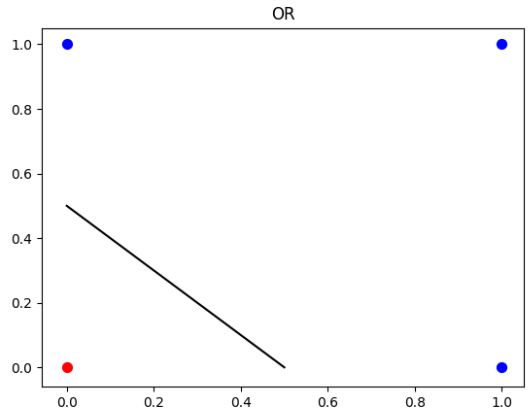




- Let's think about approximating some simple binary functions as a basic classification task
- OR and AND gates
  - OR is 0 (red) if both input are 0, 1 (blue) otherwise
  - AND is 1 if both inputs are 1, 0 otherwise
- Can we find logistic function approximations for this?
  - That is, $f(x_1, x_2)$ returns approximately 1 or 0 at the indicated points

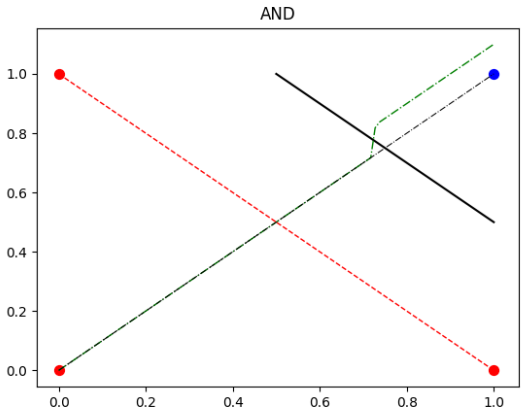
Some very simple examples for simple logistic regression

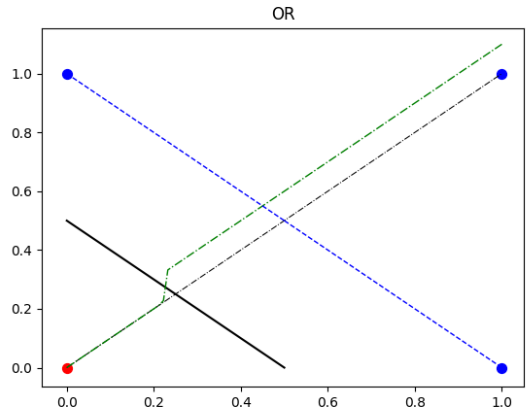




- Let's think about approximating some simple binary functions as a basic classification task
- OR and AND gates
  - OR is 0 (red) if both input are 0, 1 (blue) otherwise
  - AND is 1 if both inputs are 1, 0 otherwise
- Can we find logistic function approximations for this?
  - That is, $f(x_1, x_2)$ returns approximately 1 or 0 at the indicated points
- Yes! Take the projection perpendicular to the line

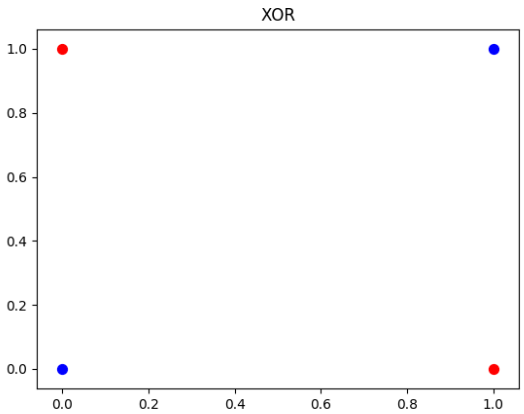
Some very simple examples for simple logistic regression





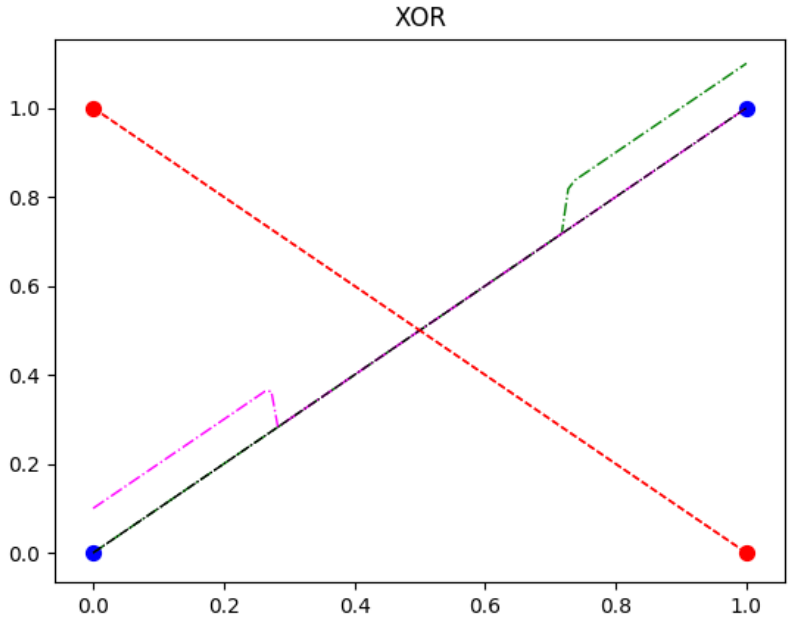
- Let's think about approximating some simple binary functions as a basic classification task
- OR and AND gates
  - OR is 0 (red) if both input are 0, 1 (blue) otherwise
  - AND is 1 if both inputs are 1, 0 otherwise
- Can we find logistic function approximations for this?
  - That is, $f(x_1, x_2)$ returns approximately 1 or 0 at the indicated points
- Yes! Take the projection perpendicular to the line
- and have the logistic turn on at the line (in the 2D plane the logistic function will turn on as a "wave-front" along the black line shown)
  - e.g. $f(x_1, x_2) = \sigma(2x_1 + 2x_2 - 1)$ for OR,
    $f(x_1, x_2) = \sigma(2x_1 + 2x_2 - 3)$ for AND [σ is our logistic function]

Very simple example with issues for Logistic Regression



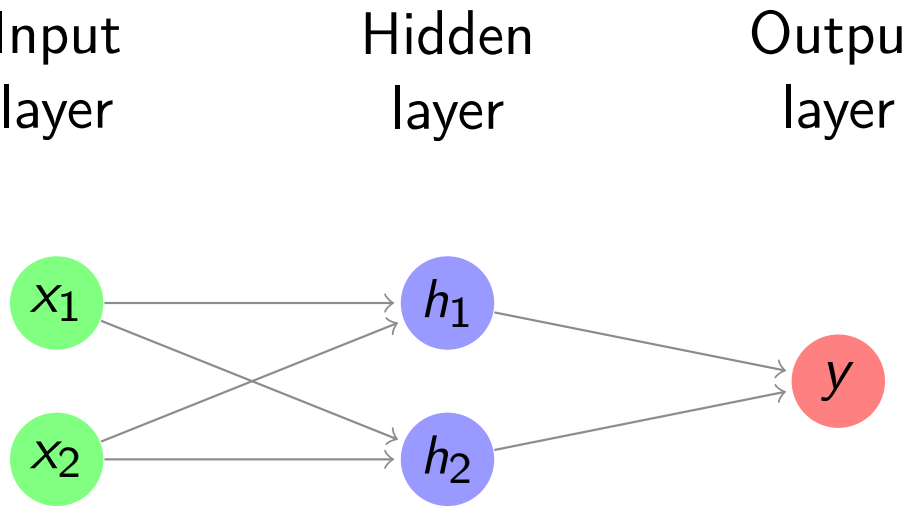
- So, with logistic regression we can separate classes that act like AND/OR gates
- Now consider the XOR gate: 1 if both inputs are the same, 0 otherwise
- The XOR gate can't be generated with a logistic function!
- Try it: no matter what line you draw, can't draw a logistic function that turns on only the blue!

How to Fix: more logistic curves!



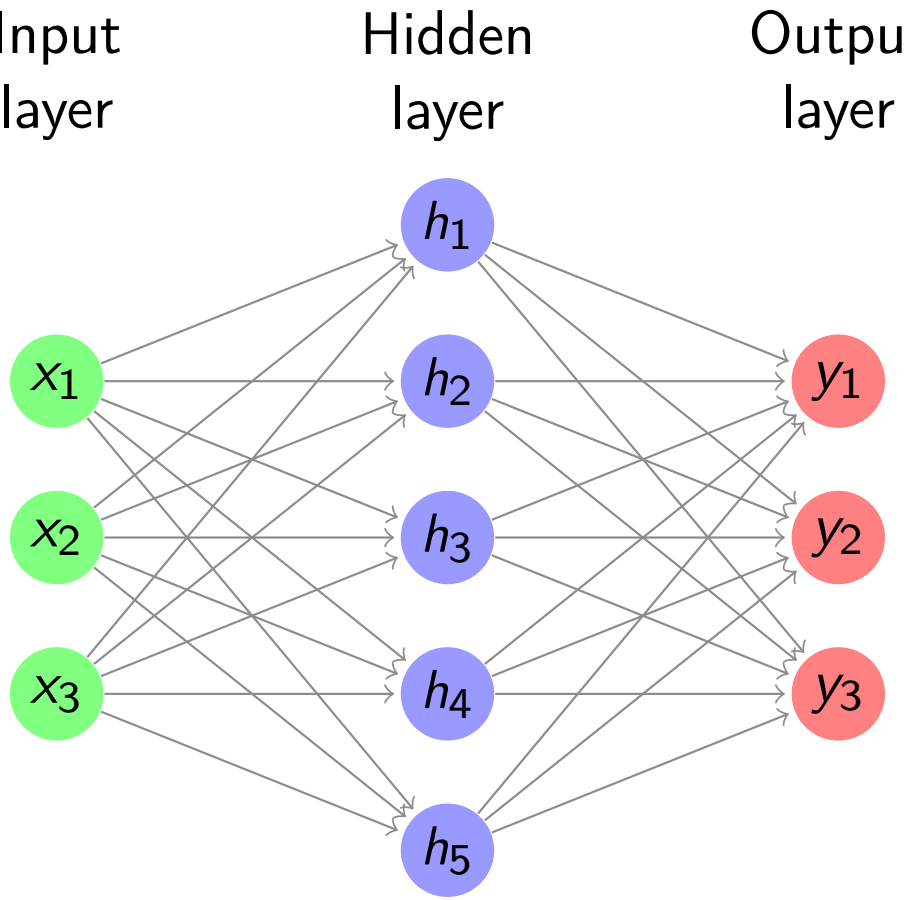
- Can fix by having 2 turn-on curves, one turning on either of the blue points, then summing the result
- $f(x_1, x_2) = \sigma(2x_1 + 2x_2 - 1) + \sigma(-2x_1 - 2x_2 + 1)$

The Feed-Forward Neural Network



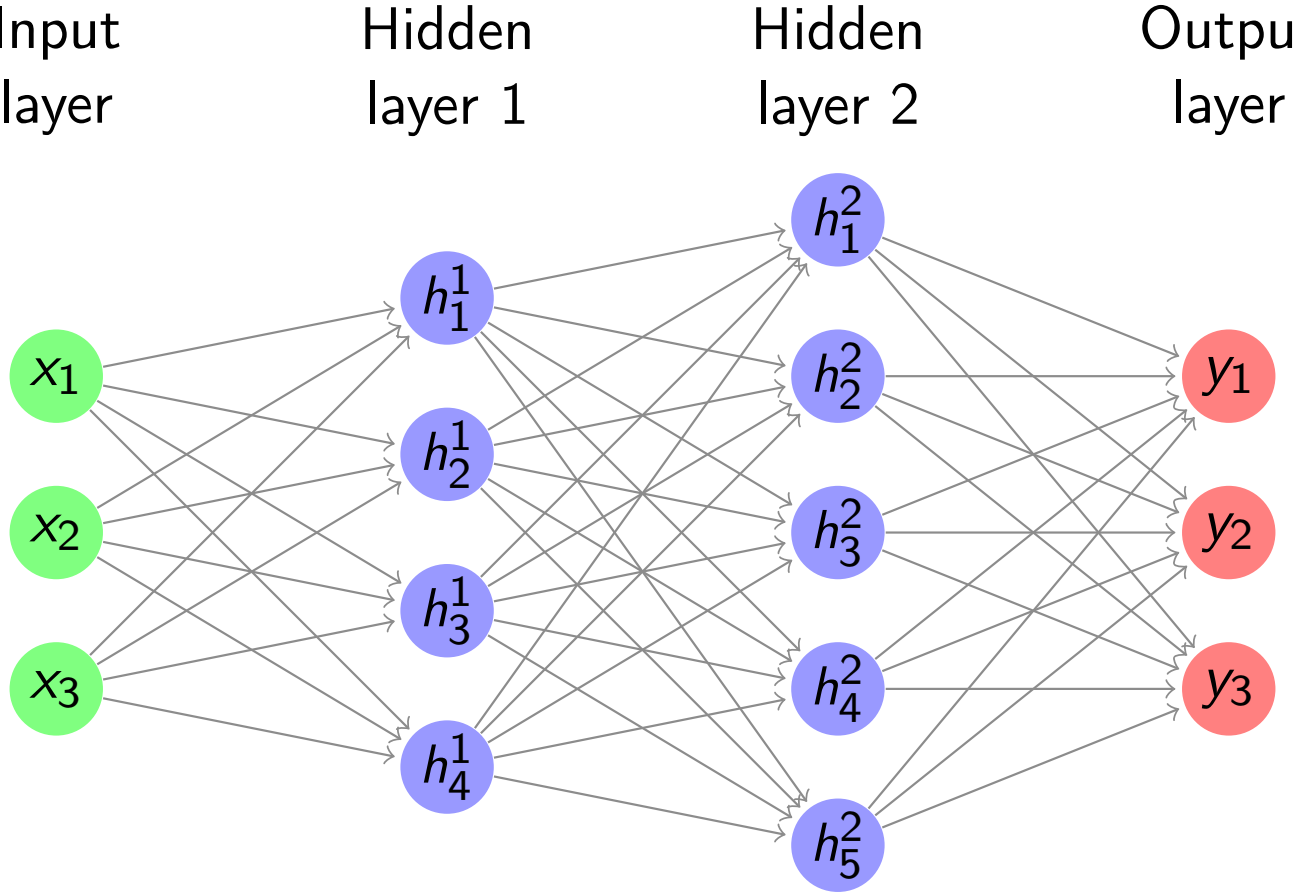
- Consider the structure of what we just made
  - $y = f(x_1, x_2) = \sigma(-1 + 2x_1 + 2x_2) + \sigma(1 - 2x_1 - 2x_2)$
- Decompose the function into:
  - the *input layer* of $\hat{x}$,
  - the *hidden layer* which calculates $h_i = \beta_i \cdot x$ then passes it through the *activation function* σ, (called "sigmoid" in NN terms)
    - There is an extra β_0, called the *bias*, which controls how big the input into the node must be to activate; σ is implicit in the diagram
  - the *output layer* which sums the results of the hidden layer and gives y
    - $y = 0 + 1 \cdot \sigma(h_1) + 1 \cdot \sigma(h_2)$

Feed-Forward Neural Network



- In general, we could have several input variables, and output variables
- In the case of classification, we would usually have a final *softmax* applied to $\hat{y}$, but could use any *activation* φ here also
  - *softmax* normalizes the output layer so it sums to 1: $f_k(x) = \frac{e^{-y_k}}{\sum_i e^{-y_i}}$

Feed-Forward Neural Network



- We can even have several hidden layers
  - The previous layer acts the same as an *input layer* to the next layer
- We also call each node in the network a *neuron*, for historical reasons
- The deep learning algorithms we will see later are just variations on this theme, using more complicated transformations

Universal Approximation Thereom

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$ such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi \left(w_i^T x + b_i \right)$$

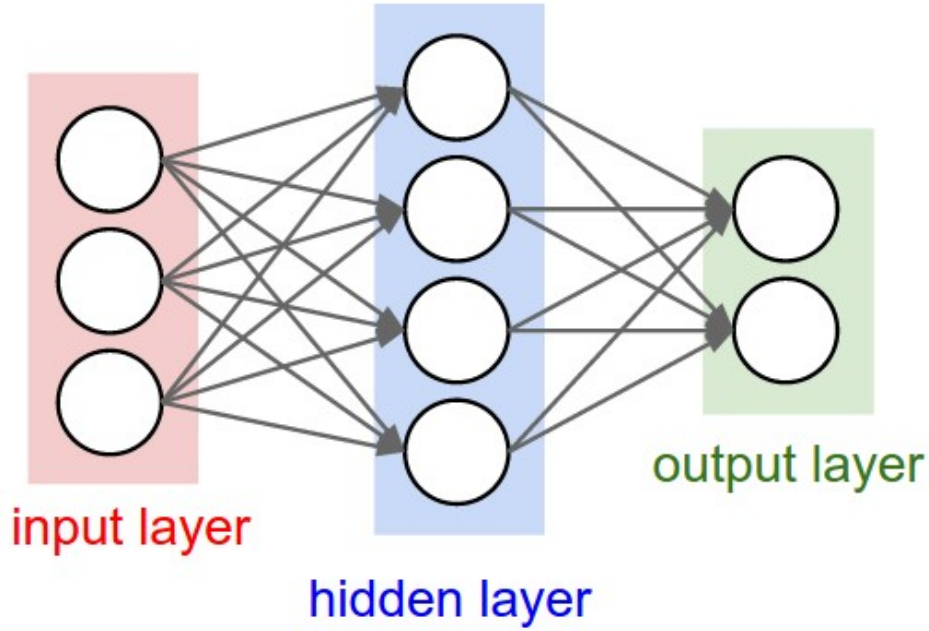
as an approximate realization of the function f; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$. This still holds when replacing I_m with any compact subset of $\mathbb{R}^m$.

- In brief: with a hidden layer (of enough nodes), any (sensible) function $f : \mathbb{R}^m \to \mathbb{R}$ can be approximated by a feed-forward NN
  - Any (sensible) activation φ can work, not just σ
- There is a simple, graphical proof for those who are interested: http://neuralnetworksanddeeplearning.com/chap4.html

Neural Networks Review



- Example shown: input vector $\vec{x}$, goes through $\vec{y}_{hidden} = W\vec{x} + \vec{b}$, then $\vec{y}_{output} = \sigma(\vec{y}_{hidden})$ (σ is some non-linear turn-on curve)
- I.e. hidden layer combines $\vec{x}$ by some weights, then if the weighted sum passes a threshold $\vec{b}$, we turn on the output (with the $\sigma(x) = 1/(1 + e^{-x})$ to gate the ops)
- Need to **train** the weight matrix W and the bias vector b and optimize a "loss" function that represents a distance from the target output

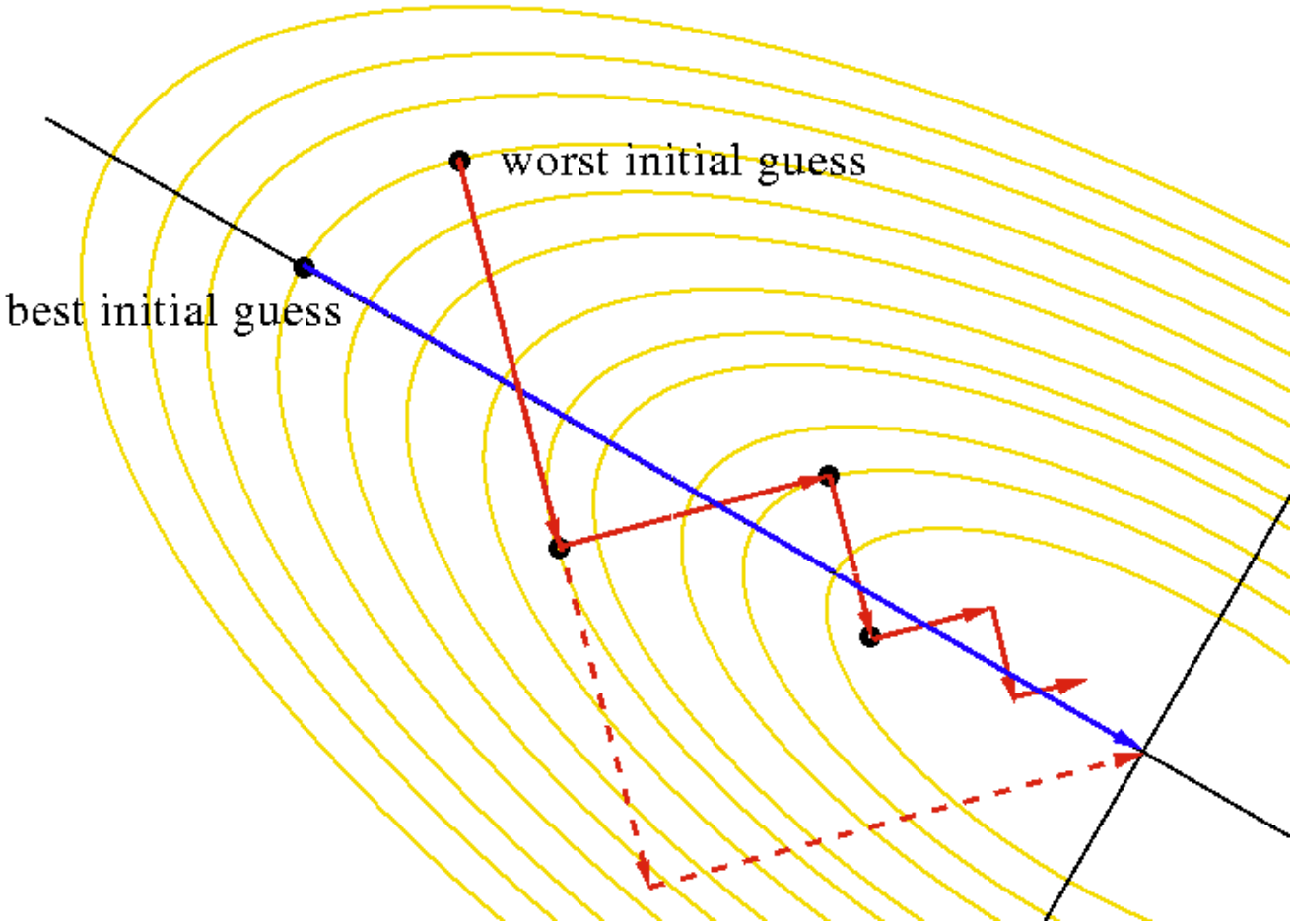
Analogy: Steepest descent

- A climber is trying to find his way down a mountain in deep fog, how should he proceed?
- One idea is to try to always go downhill the fastest way possible
- So, he figures out which direction has the steepest descent (ie which way is downhill), then takes a step in that direction
- After the step, he checks again, and takes another step
- He keeps proceeding in this manner until he cant go downhill anymore, he's reached the bottom

Gradient Descent

- From calculus, $\nabla f(\mathrm{x})$ gives the direction of largest increase of f at x (if its 0, we are at a minimum and done)
- Equivalently, $-\nabla f(\mathrm{x})$ gives direction of largest decrease, so $f(\mathrm{x} - \gamma \nabla f(\mathrm{x})) < f(\mathrm{x})$ (at least, for some γ small enough)
- We will define a sequence x_i to find the minimum:
  - Start with some random position x_0
  - Iterate:
    - Find $\mathrm{x}_{n+1} = \mathrm{x}_n - \gamma_n \nabla f(\mathrm{x}_n)$
    - Stop if $|f(\mathrm{x}_{n+1}) - f(\mathrm{x}_n)| < \epsilon$, i.e. we're not reducing further, so we're close to the minimum
  - Return the final x_n
- γ_n can be different for each iteration, we'll find the best γ_n by checking several possible values
- ϵ is the *tolerance*, how close to a minima do we need to be before stopping (we could also check $|\nabla f(\mathrm{x}_n)| < \epsilon$)

Example function

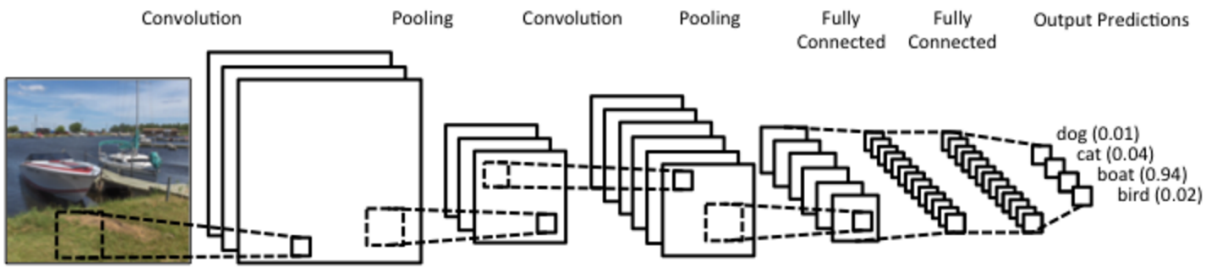


- Shows how the algorithm picks out different paths depending on starting point
- Lines are contours of equal value

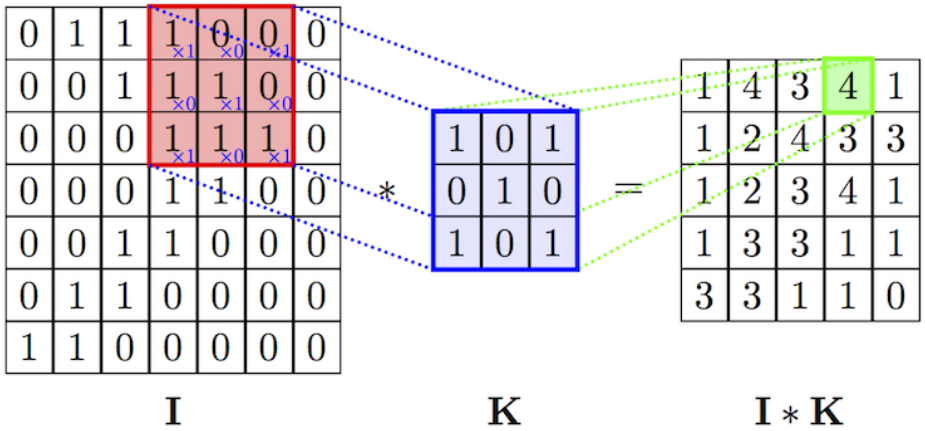
Training Neural Networks: Backpropagation

- The algorithm to train neural networks is called **backpropagation**
- Its essentially a gradient descent implemented taking the network structure into account to speed up evaluation of the partials
- To apply gradient descent, we need a function to minimize, this is our loss function from earlier
  - $L(x_i;\theta) = \sum_i |f(x_i;\theta) - y_i|^2$ for inputs x_i with known output y_i
- We start with the parameters θ set to arbitrary values, usually picked from e.g. the unit gaussian
- We run a forward pass through the network and calculate the loss, keeping track of the values at the intermediate nodes
- Using the chain rule, calculate the derivates *for all weights* backward from the loss to the higher layers to the inputs, in a single pass
- Propagate changes based on the gradient $\Delta\theta_i = -\eta \frac{\partial L}{\partial \theta_i}$
- For more on how backpropagation works: `http://neuralnetworksanddeeplearning.com/chap2.html`

A Convolutional Network



- One of the great advances in image classification in recent times
- We have some filter kernel K of size $n \times m$ which we apply to every $n \times m$ cell on the original image to create a new filtered image.
- It has been seen that applying these in multiple layers of a network can build up multiple levels of abstraction to classify higher-level features.
  - And, importantly, is trainable many, many layers deep

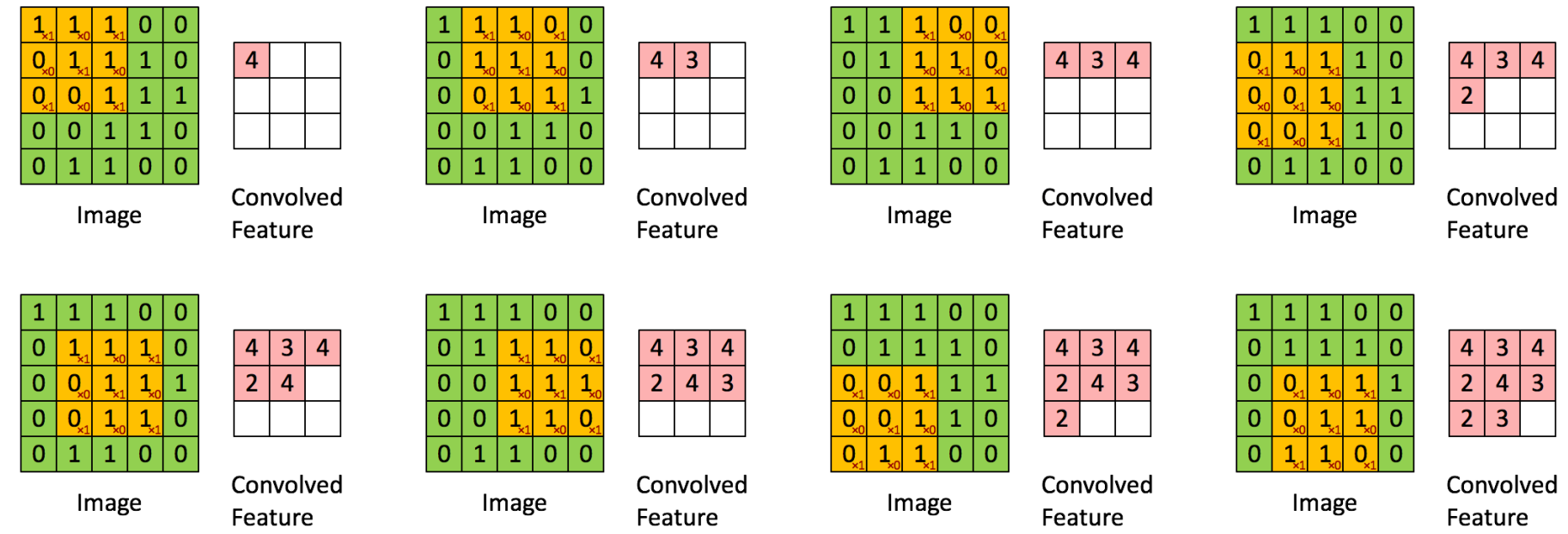


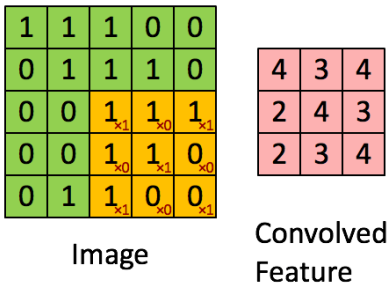
Reference: `http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/`

Convolutional Layers

- A convolution layer is a connection between one layer and the next in a NN with a very specific structure:
  - Typically, it works with a 3d input like an image: channels (red, green, blue), width, height
  - It contains a **kernel** or **filter**, which is a 3d block sized $channel \times n \times m$, n and m are user-specifed, with each element of the block a weight to be set in training
  - The outputs consists of all $n \times m$ *convolutions* of the filter with the image, creating a new one-channel image
    - Discrete convolution, meaning each element of the kernel is multiplied with a pixel in (one channel of) the image, and all are summed together
  - The output of the filter is passed through an activation function, the same as the usual fully-connected layer
- A single convolutional layer generally consists of many convolutional filters, each filter giving one layer in the output
- Networks with convolutional layers are Convolutional Neural Networks: CNN

Convolutional Filters In Pictures

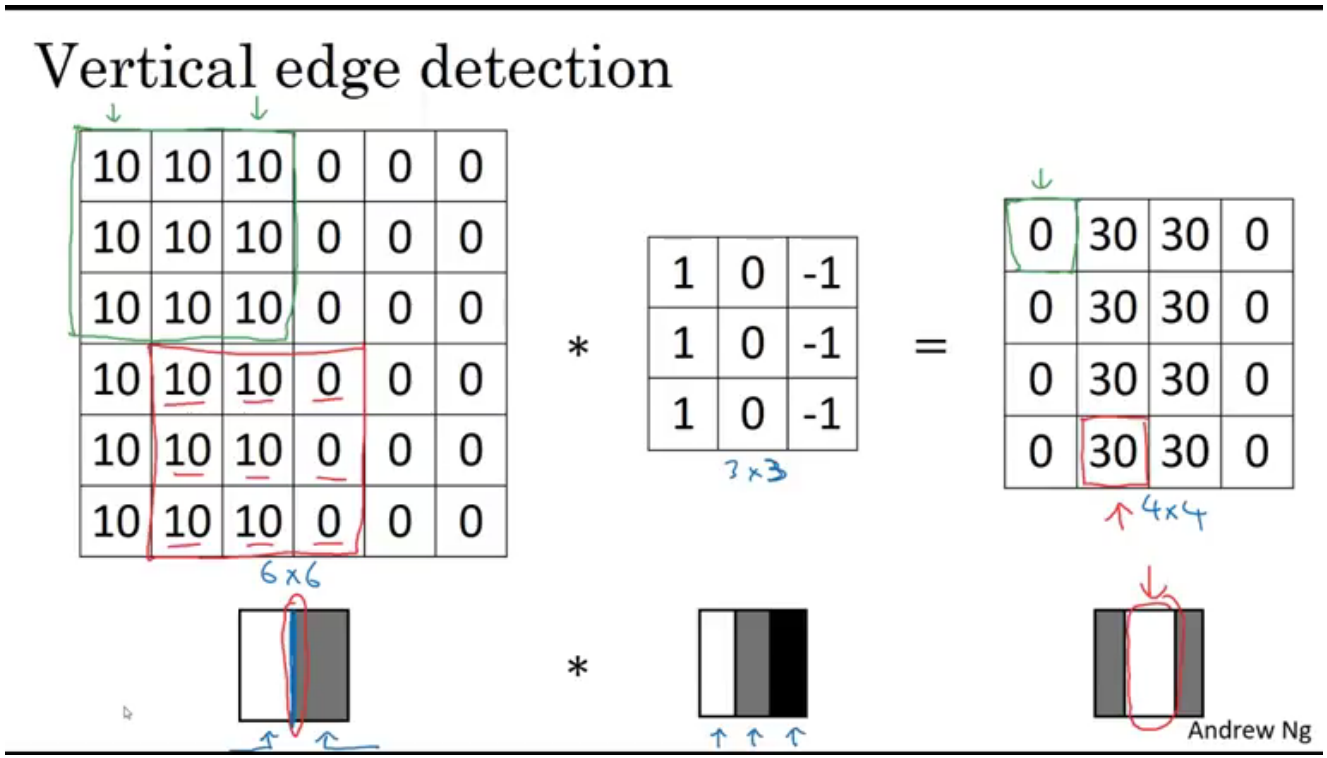




- A filter sliding over the image builds up the output layer, each output is sum of filter elements multiplied by image pixels
- The same filter is used for each pixel, the weights are learnt during training (as well as an output bias)

https://www.kdnuggets.com/2015/11/understanding-convolutional-neural-networks-nlp.html

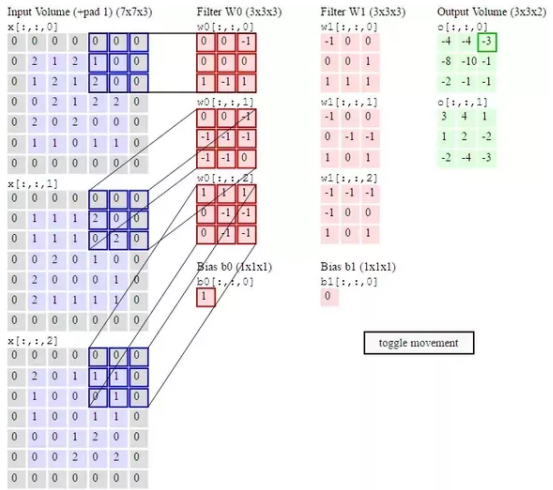
Example Filter



- As an example, here is a 3x3 filter for detecting vertical edges
- The opposing plus and minus sides cancel in a block of color
- At an edge, the filter is either highly positive (white to left of edge), or negative (white to right of edge)
- What would a horizontal edge detector look like?

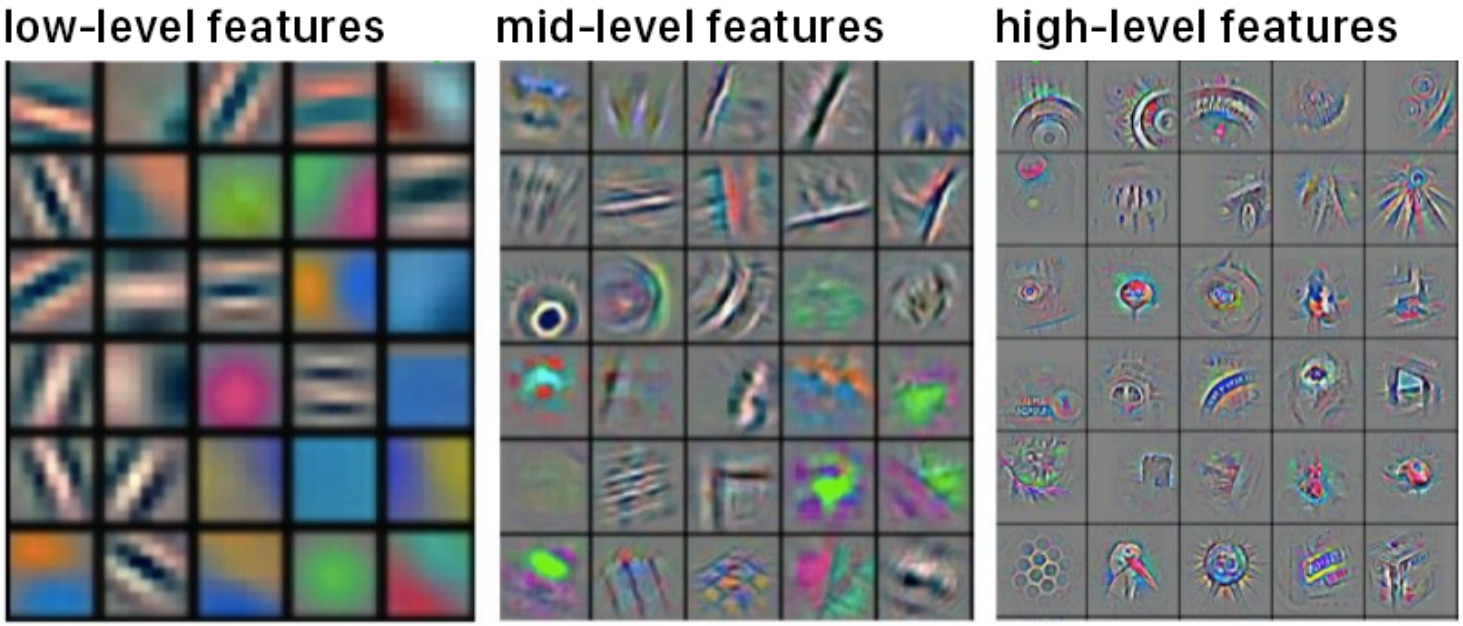
Andrew Ng lecture by way of
https://kharshit.github.io/blog/2018/12/14/filters-in-convolutional-neural-networks

Multiple Filter Outputs



- When multiple filters are used in a single layer, they have the same width and height, so they can be put together in a single output as *channels* $\times$ *width* $\times$ *height*
- This is exactly the image structure which was the input to the network
- This means this convolutional structure can be used several times in series
  - Each successive layer effectively sees a larger part of the image, since each pixel in the output of one layer is from several pixels
- The image shows that a 3-channel input needs filters with a 3x3x3 block, and 2 filters produce a 2 channel output

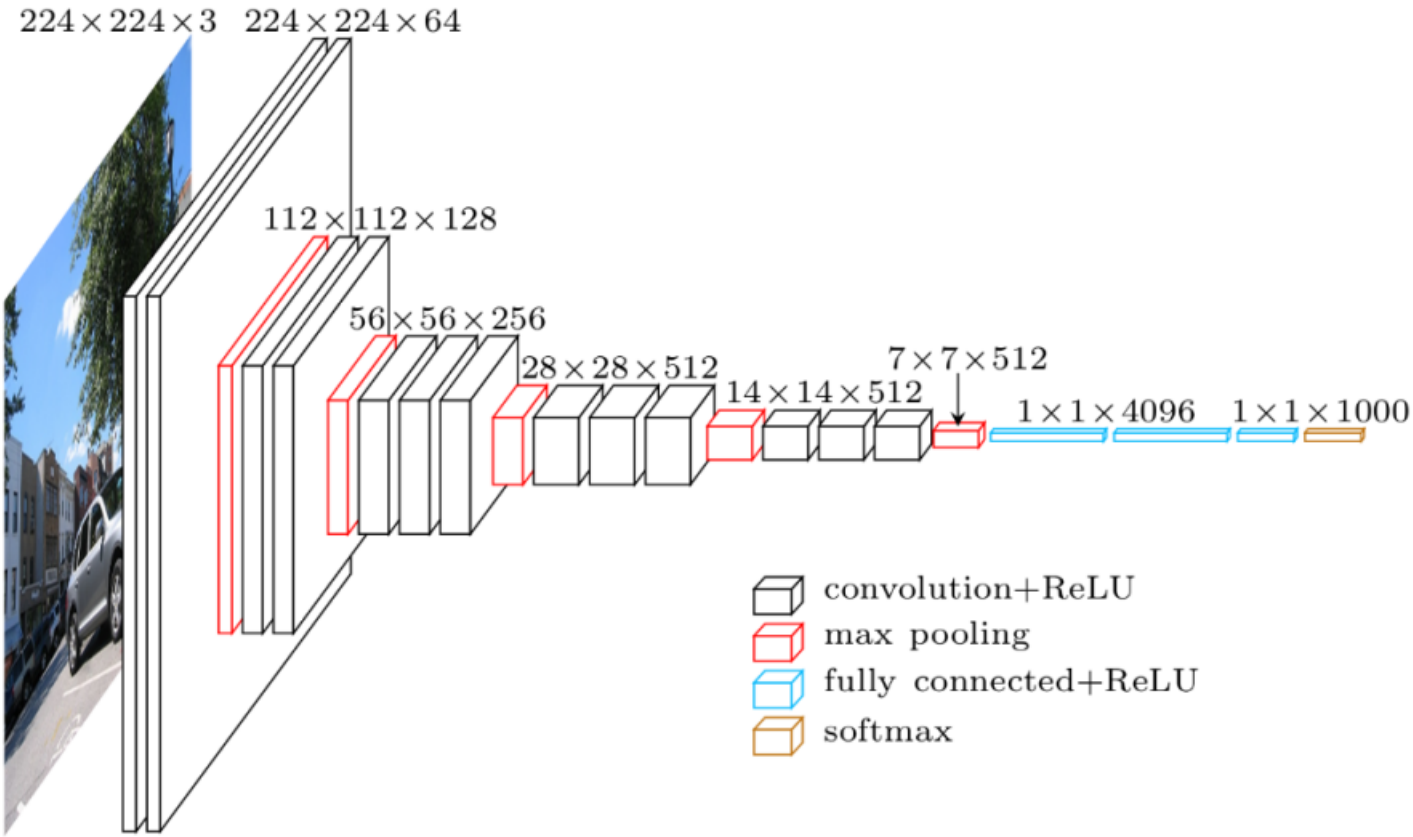
Filters Over Several Input Layers



- Convolutional layers are typically built up one after the other
- The idea is that features get *built up*, at low levels, you might have edge detectors, later layers use these edges to build up structure, and by high levels recognizable objects are being searched for
  - These images are made by doing reverse gradient descent on the network, i.e. updating the image pixels themselves, trying to make the image "light up" (set node output high) a particular node
- Networks these days can contain *hundreds* of these layers
  - This is the meaning of *deep* in deep learning

Image from `https://twopointseven.github.io/2017-10-29/cnn/`

Realistic Networks



- Example of a real network used for image classification, VGG-16
- Typically, networks consist of several convolution layers following by max pooling layers (take the max from a 2x2 square)

Extensions You Can (and Should) Study

- VGG16 has long been surpassed, but this is the basis for modern image classification
- Some examples of extending the basic CNN idea
- Structural Changes
  - ResNet: A powerful idea which greatly helps the classification ability of the network: simply add the output from one layer to a "residual" several layers down
    - This allows the network to effectively "skip over" several layers, so early training can set a few layer layers (avoiding the Vanishing Gradient Problem), then subsequent training "refines" the result with the residual layers
  - Inception layers: Have multiple filter sizes from one layer, sum the outputs
- Regularization
  - BatchNorm and related techniques
- Transfer Learning: Start with a network trained on one problem, refine the weights to solve a different problem
  - The idea being a general network should have good, early layer filters, and just need to refine the output layers for a new problem

Beyond Classification

- The idea of Machine Learning and Deep Learning, in particular, is that the model is not just learning to classify but capturing essential features of the inputs: the **latent variables** of the dataset
  - Not every pixel is created equal, in a dataset of cats, there are far fewer ways to draw cat images than there are possible values for the pixels
- We can train a model to find and fit the underlying latent variables which captures the essential features and then can be used in ways other than simply classifying the input
- For example, we could use it to do:
  - Denoising: Given a noisy or degraded sample, recover the original (e.g. remove dirt or recolorize an old film image)
  - Missing value imputation: fill in some missing data from a sample (e.g. given an image where a black square was put over part of the image, fill in what was in the square)
  - Sampling: randomly generate a sample input drawn from the latent variable probability distribution (starting from some random noise, generate a random image that fits in the dataset)

Sampling

- We have a **generator**, which we think of as a black box for now
- The generator should be fed a vector from the latent variable space
  - We can posit that the latent variables are gaussian, so pick out a multi-dimensional gaussian, and give any arbitrary dimension
  - Today, we won't give the latent variables explicit meaning, but for our cat e.g. it could be things like what direction the cat is facing, the color of the fur (along some spectrum), length of fur, shape of face, etc. etc.
- The generator should spit out sample that is indistinguishable from a sample pulled from the dataset
- Say we take MNIST, and posit a latent space of dimension 10
- If we have an MNIST generator, we sample a gaussian 10 times, feed that 10D vector into the generator and it should spit out a 28x28 image of a hand-drawn number
- This drawing should be indistinguishable from the original MNIST set (given two images, one drawn from MNIST and one from the generator, it should be impossible to tell which is which (assuming you haven't just memorized the entire MNIST set))

How to Build a Generator?

- There are several ways we can take our basic CNN DL setup and use it to build a generator G
- Thinking about the last point of the previous page: what if we had a network that tries to distinguish real MNIST images from images from the generator?
  - A *discriminator* network D, outputs 0 for generated images, 1 for real images
- We could use the discriminator output like a loss function:
  - Use G to generate an image from a random latent vector and pass through D
  - Make small changes to the parameters of G and see how D changes
  - Update the parameters of G so that the D output **increases**
  - I.e. we try to change G so that D gives values closer to 1 (the output for a real MNIST image)
  - Eventually, G should only output images that give D output 1

Adversaries

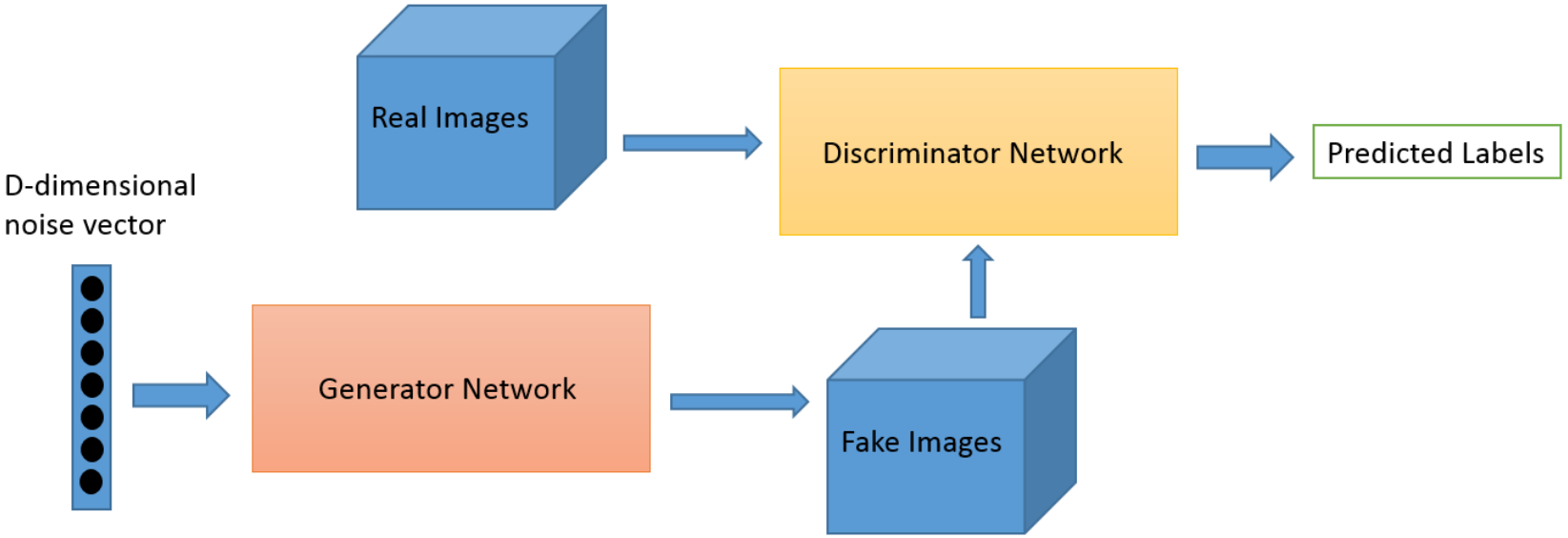
- But now D is useless: it can't tell real from fake images
- But what if we now train D? I.e. pass real images and update D so it gives outputs closer to 1, pass fake images and update D so it gives outputs closer to 0
  - Whatever criteria D used before to tell real from fake images no longer works, so we retrain D to find new criteria
- For a fixed network G, if we update D enough, images from G will be given output 0 and images from MNIST will have output 1
- With the updated D, we can start training G again to fool the new D
- We can keep playing this game as long as we like
  - Train G to fool D
  - Train D to figure out which images are from G
  - Ad infinitum

GAN

- GAN: Generative Adversarial Network
- This is the name for the setup we just described:
  - Two networks are randomly initialized
    - D takes in an image and outputs a number from 0 to 1
    - G takes in a vector and outputs an image (so its output can be fed into D)
    - The internal structure of the networks can be anything: D could be a feed-forward neural network, or a CNN, or anything else
    - We'll go through how to structure G, but a simple feed forward network can also be used
  - The networks are trained successively so the D distinguishes real from G images, then G is trained to fool the current D into believing its images are real
  - And so on and on

Original GAN paper: `https://arxiv.org/abs/1406.2661`

GAN

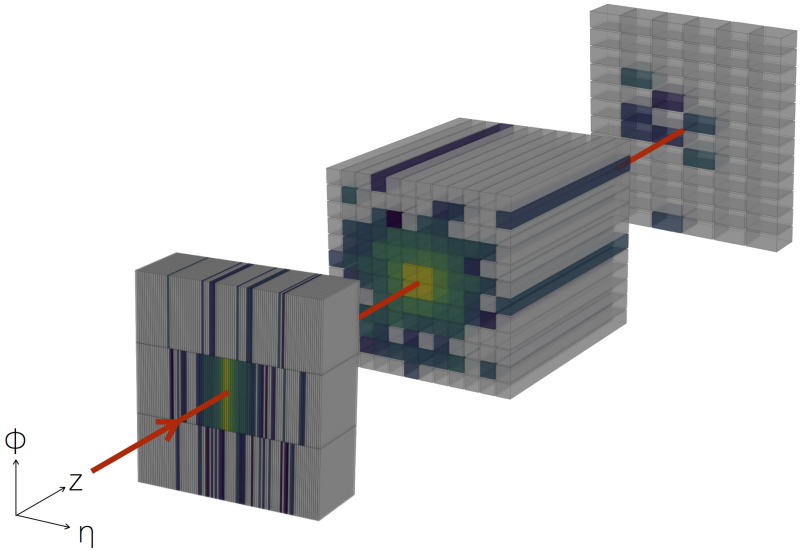


- Feed noise (random point in latent space) into G
- Train G to fool D, train D to catch out G,
  its a competition between the two networks
- Alternate gradient descent on batches between G and D

This Person Does Not Exist

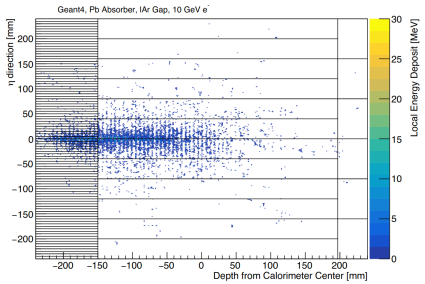
- This is the first image I got when I visited this person does not exist
- They've trained a GAN (technically, StyleGAN, an extension of the GAN idea) on human portrait photos
  - StyleGAN also allows them to control specific aspects of the latent space (hair color, skin color, gender, etc.)
  - Can see how powerful the results can be
- Highly recommend watching the videos from nVidia on this (accessible through the link)

CaloGAN

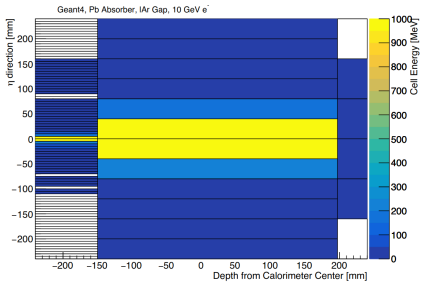


How does an electron look in liquid argon?

2D slice



We simulate exact (x, y, z)



What we can read out is this

- In Particle Physics, applications to simulation
- CaloGAN is training a GAN to simulate the calorimeter response to particles
  - This is an extremely time consuming step in simulation and important for HL-LHC which will have huge pile-up
- See also: JongSuk's project ;)

`https://github.com/hep-lbdl/CaloGAN`
`https://indico.cern.ch/event/567550/contributions/2629438/attachments/1510662/2355700/ACAT_GAN.pdf`