INTRO TO DEEP LEARNING

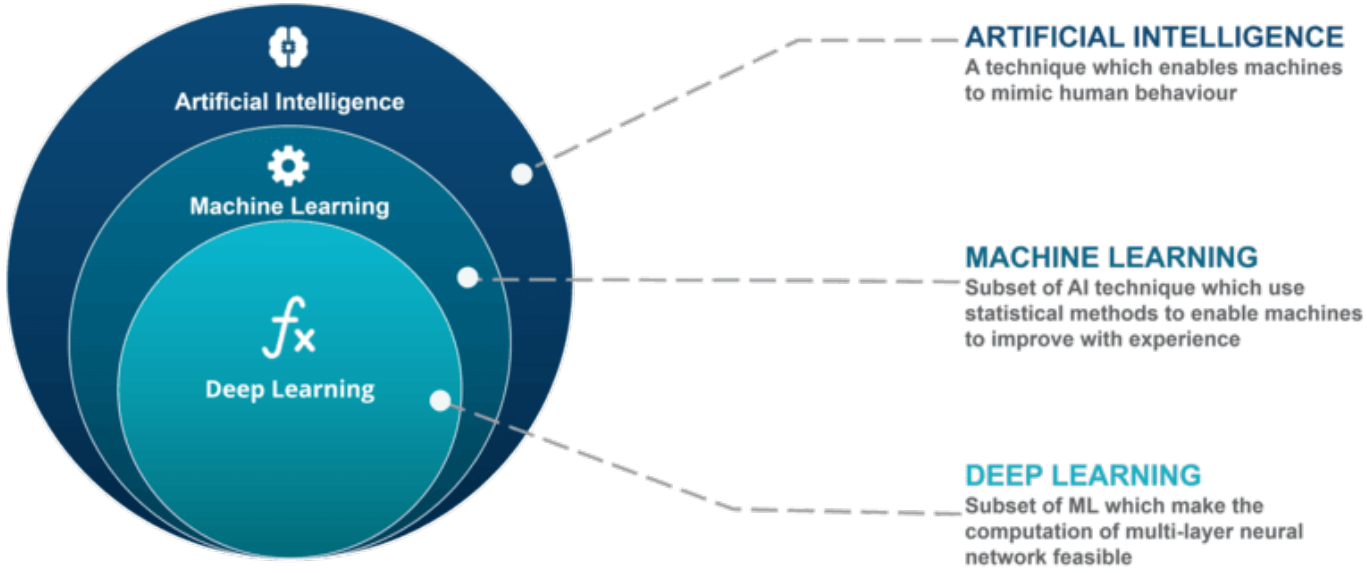
WHAT IS DEEP LEARNING?

Machine Learning?

Deep learning is a subset of machine learning, which is a subset of A.I.

Machine learning uses algorithms to parse data, learn from that data, and make informed decisions based on what it has learned

Deep learning structures algorithms in layers to create an "artificial neural network" that can learn and make intelligent decisions on its own

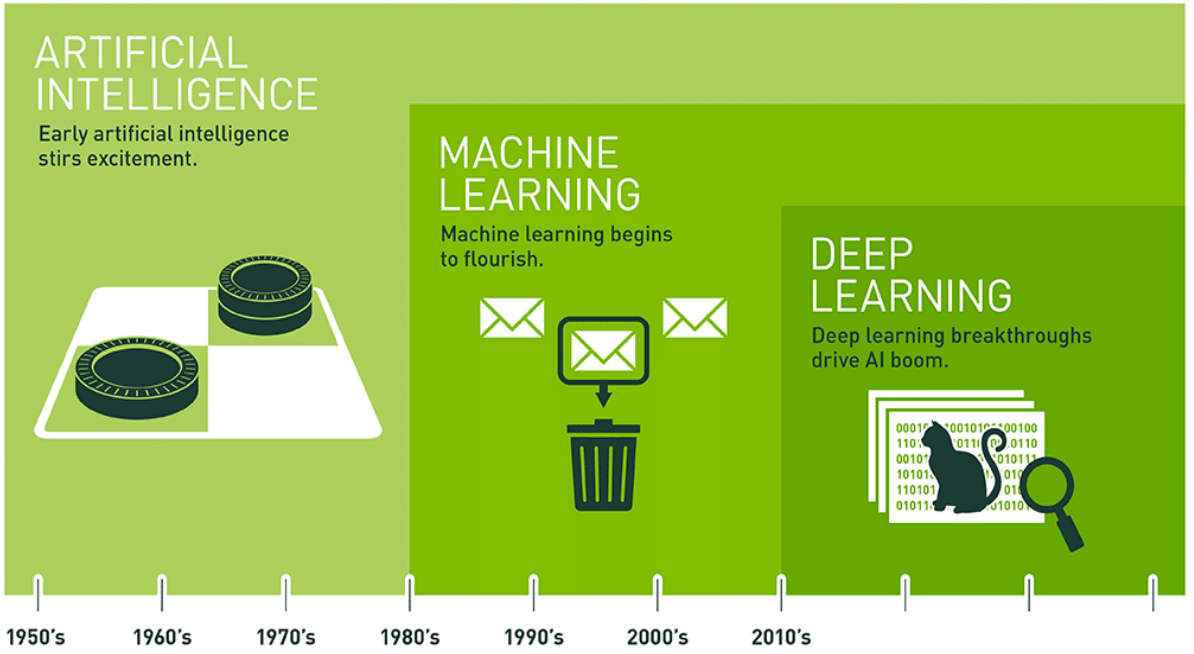


Deep Learning

Deep learning isn't new, but it has very gotten a lot of traction due to improvements in evaluating generic optimization problems.

Deep learning attempts to represent something with high-level features as a collection of lower-level features.

Deep learning resembles the neural connections that exist in the human brain.



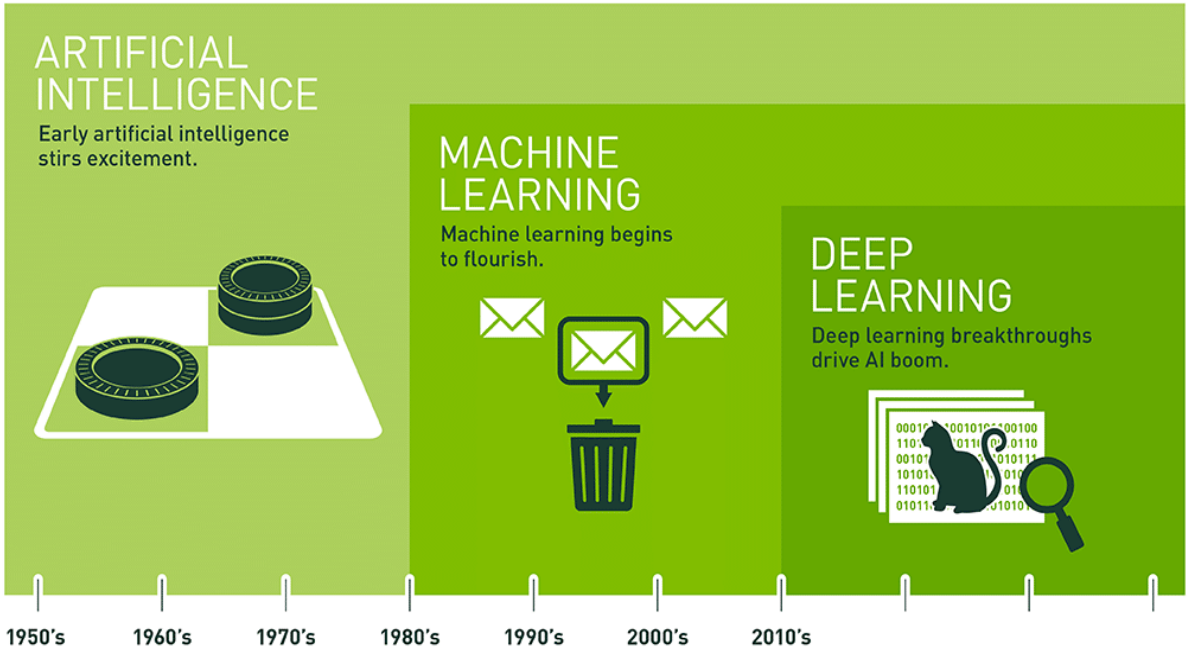
Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

Deep Learning

Deep learning is called as such because its algorithms are composed of several feature extracting transformations in series.

Examples:

- Recommendation
- Translation
- Autonomous vehicles
- Computer vision
- Text generation
- Style transfer
- And so many more applications……



Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

BASIC CONCEPTS

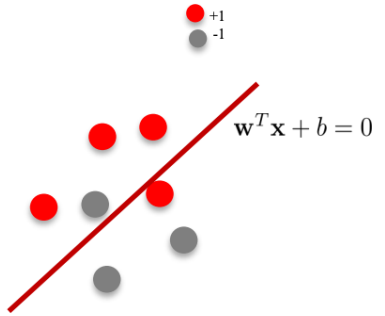
Perceptron

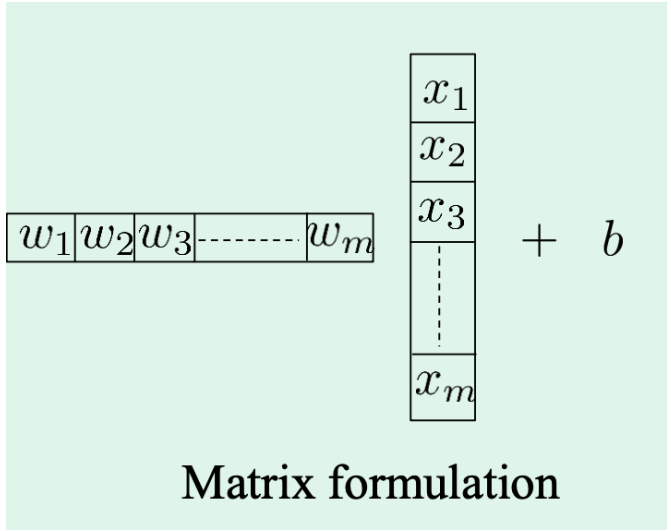
Perceptron is a linear classifier

Can be considered as a special "Neuron" with the sign function being the activation function of choice.

Note that the learning process is not strictly gradient descent but stochastic gradient descent.

$$f(\mathbf{x}; \mathbf{w}, b) = sign(\mathbf{w}^T\mathbf{x} + b)$$





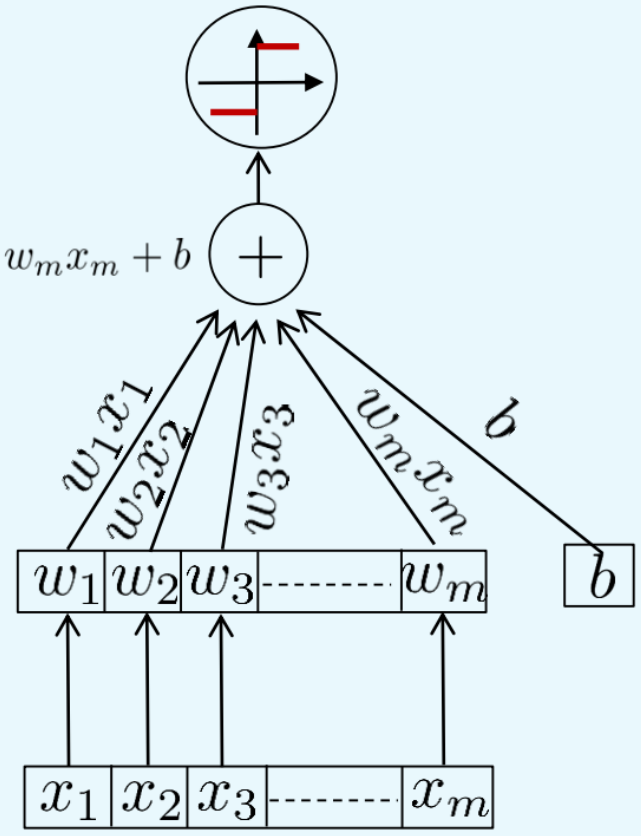
Matrix formulation

$sign(w_1x_1 + w_2x_2 + w_3x_3 + .. + w_mx_m + b)$

$w_1x_1 + w_2x_2 + w_3x_3 + .. + w_mx_m + b$

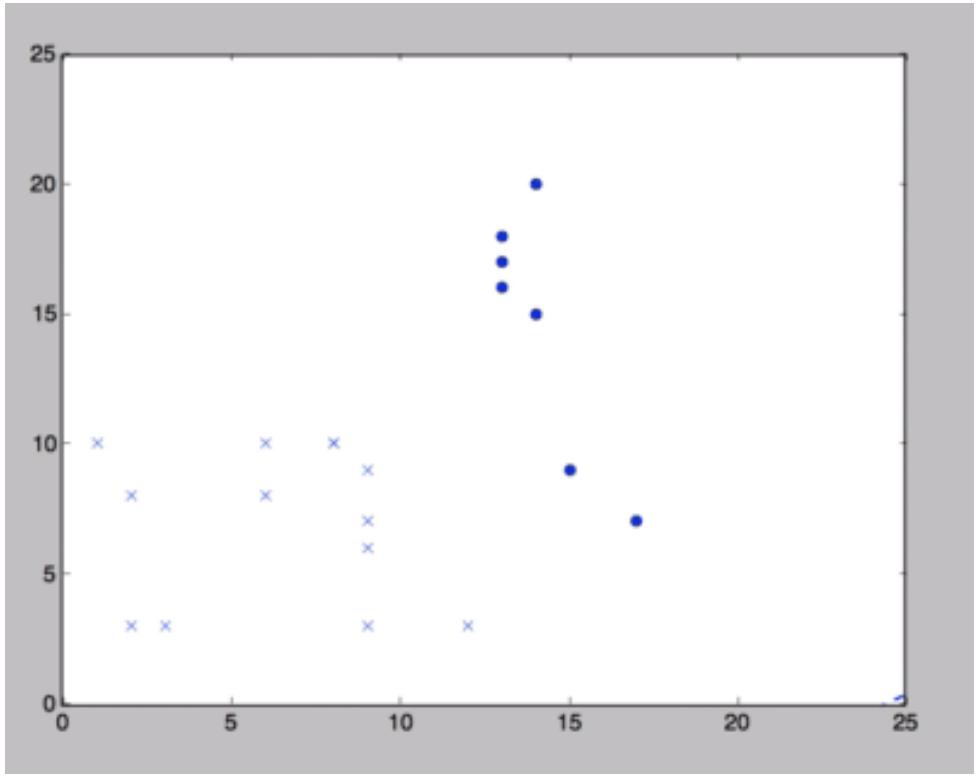
Model Parameters: $\mathbf{w} \in \mathbb{R}^m$

Input: $\mathbf{x} \in \mathbb{R}^m$



Schematic illustration

Perceptron



- Initialize the weights (however you choose)

  $\mathbf{w}^T\mathbf{x} + b$ (initialize $\mathbf{w}$, and b)
- Step 1: Choose a data point.
- Step 2: Compute the model output for the data point.
- Step 3: Compare model output to the target output.
  - If correct classification, go to Step 5!
  - If not, go to Step 4.
- Step 4: Update weights using perceptron learning rule. Start over on Step 1 with the first data point.

Or

$$\mathbf{w}_{t+1} = \mathbf{w}_t + (target_i - output_i)\mathbf{x}_i \qquad \mathbf{w}_{t+1} = \mathbf{w}_t + \lambda(target_i - output_i)\mathbf{x}_i$$
$$b_{t+1} = b_t + (target_i - output_i) \qquad b_{t+1} = b_t + \lambda(target_i - output_i)$$

- Step 5: Go to the next data point. If you have gone through them all, you have found the solution!

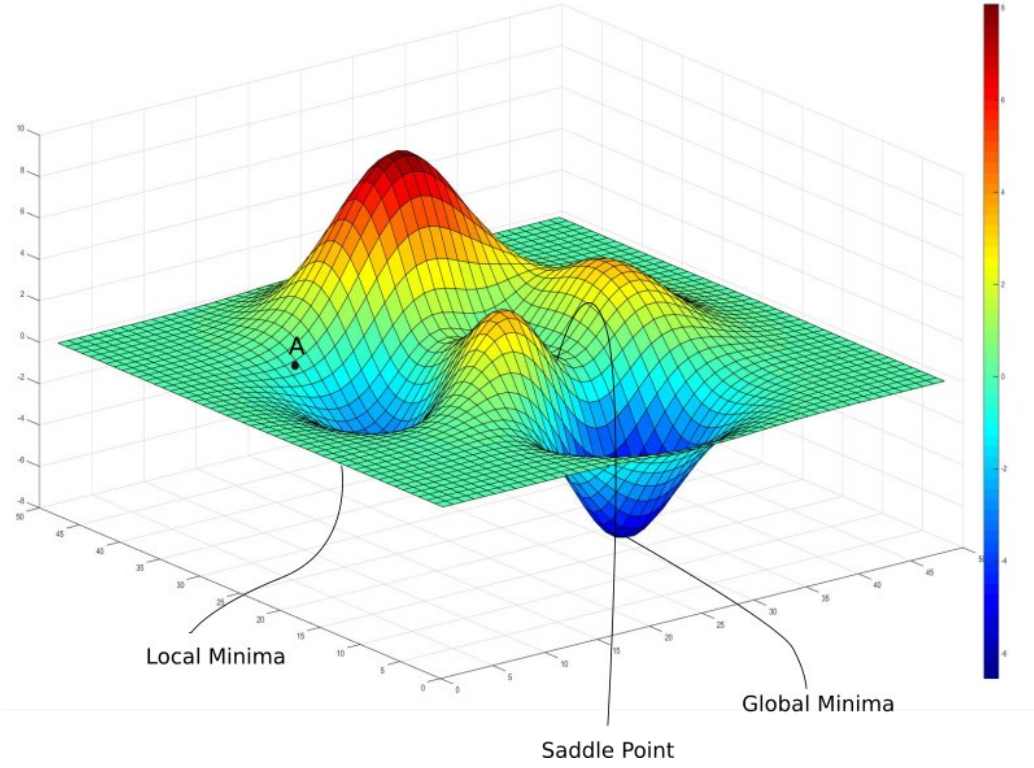
Gradient Descent

Gradient descent works for strictly convex functions, but neural networks are not strictly convex…

Stopping condition: gradient = 0

- Where have we seen this to be the case?
  - Local Minima
  - Global Minima
  - Saddle Points

What should our goal be?



Gradient Descent

Minimize network losses using stochastic gradient descent (SGD). The generic implementation for a neural network is known as backpropagation.

> [***SGD***] Randomly grab input, calculate the loss, and then backpropagate to update the model parameters.

Why do we choose SGD over GD?

A. It finds the global optimal solution.

B. It is computationally more efficient.

C. It is less memory demanding.

D. It avoids bad local optimal.

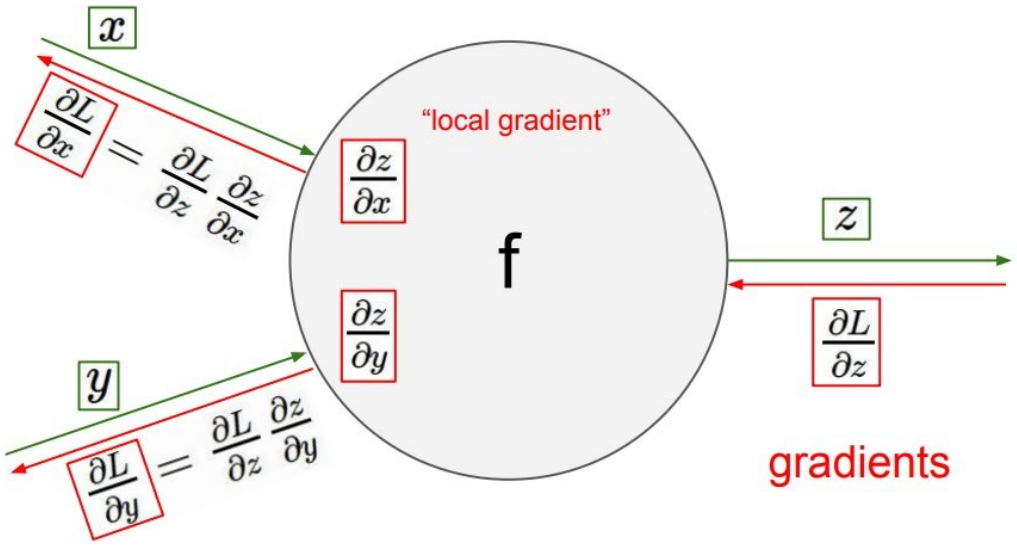
E. All of the above.

Back Propagation

Check out how backpropagation works more in depth with the CS231N material from Stanford!

Or take some courses offered here such as ECE 285, ECE 271B, CSE 190/253, COGS 181

Essentially finding the gradient using the chain rule and updating your equations.

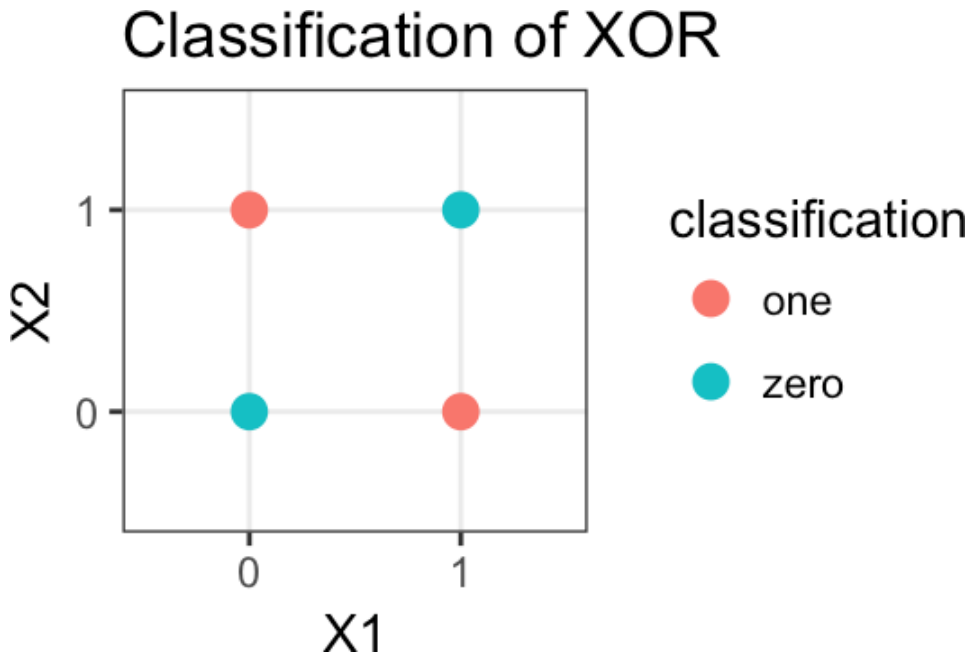


Why not perceptron?

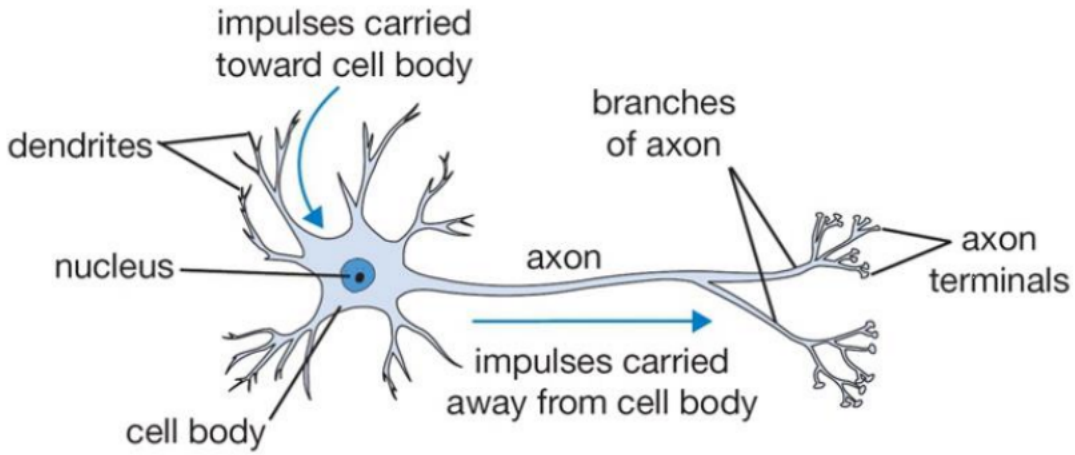
Can you find a straight line that can be drawn across the graph that can divide the two classes on the right?

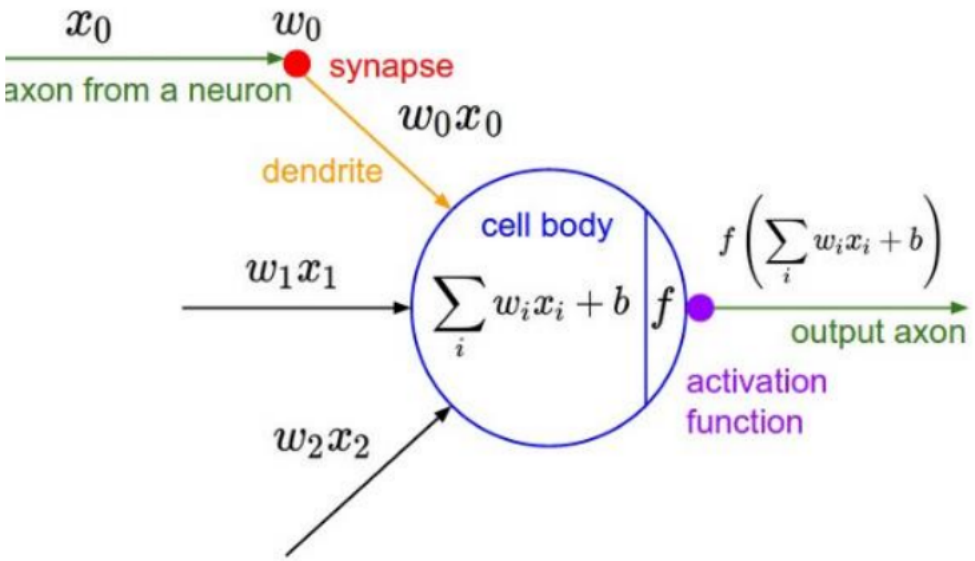
Obvious answer: No

So we have a reason to introduce non-linearities in the network. Without these non-linearities, it becomes impossible to model non-linear classification/regression problems such as the XOR classifier.



Neurons


impulses carried
toward cell body
dendrites
branches
of axon
nucleus
axon
axon
terminals
impulses carried
away from cell body
cell body


x_0
w_0
axon from a neuron
synapse
$w_0 x_0$
dendrite
cell body
$w_1 x_1$
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
output axon
activation
function
$w_2 x_2$

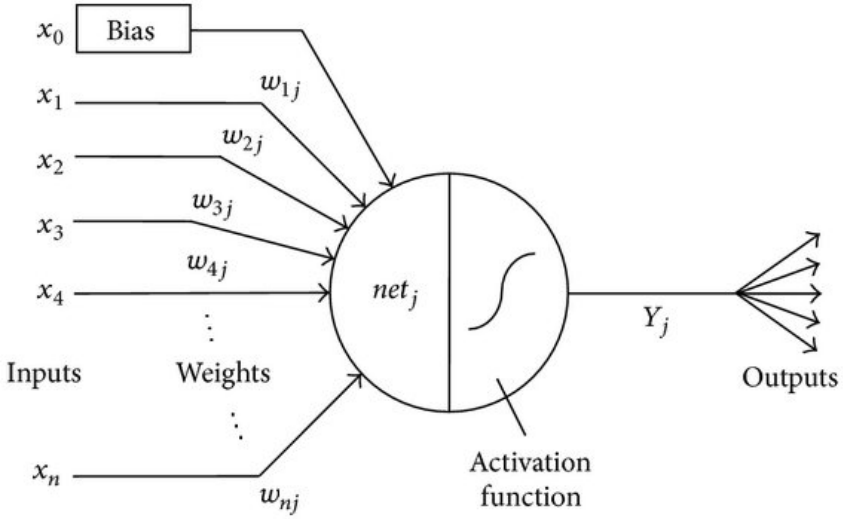
Neurons

The most fundamental network is comprised of “layers” of neurons.

These neurons are composed of:

- **Inputs**, labeled [AKA **features**]
- **Weights** associated with each input
- An **accumulation** of the product of each input and weight
- The **activation function**
- The **output** or the input for the next neuron.

Note that the activation function determines whether the neuron is non-linear.



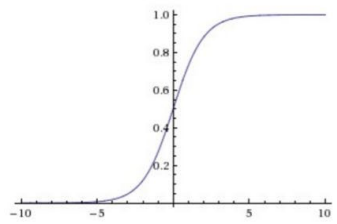
Activation Functions

The activation functions in the output layer determine non-linearity of the network. Typical examples:

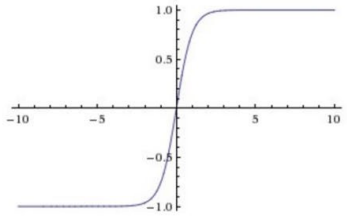
- Sigmoid, Tanh, ReLU, ReLU Alternatives
- Softmax is generalization of sigmoid

In practice:

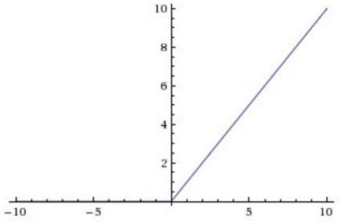
- Use ReLU, careful with learning rate
- Try out Leaky ReLU and other alternatives
- Try out tanh but don't expect much
- Don't use sigmoid



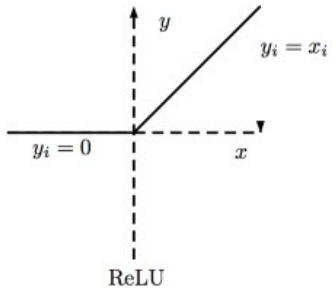
Sigmoid



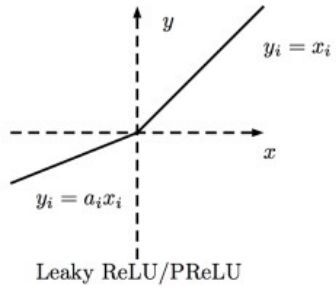
tanh



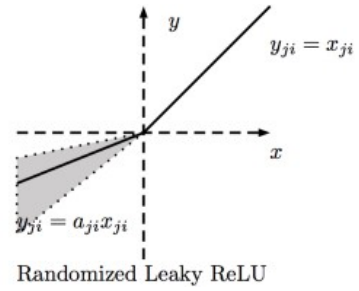
ReLU



ReLU



Leaky ReLU/PReLU



Randomized Leaky ReLU

Loss Functions

Gives a measure of how "far away" the current predictions of the network are from the target for a specific task.

Be careful when choosing the loss function!

> *Example:* Outliers with L1/L2 Loss. L2 overemphasizes outliers!
>
> Many don't even make sense in the context of certain problems!

Loss Functions

- Cross-Entropy Loss [Binary Classification]
- Multiclass Cross-Entropy [Multiclass Classification]
- L1/L2 Loss [Usually Regression]
- KL/etc. Divergences [Density Estimation]
- And many others depending on application…

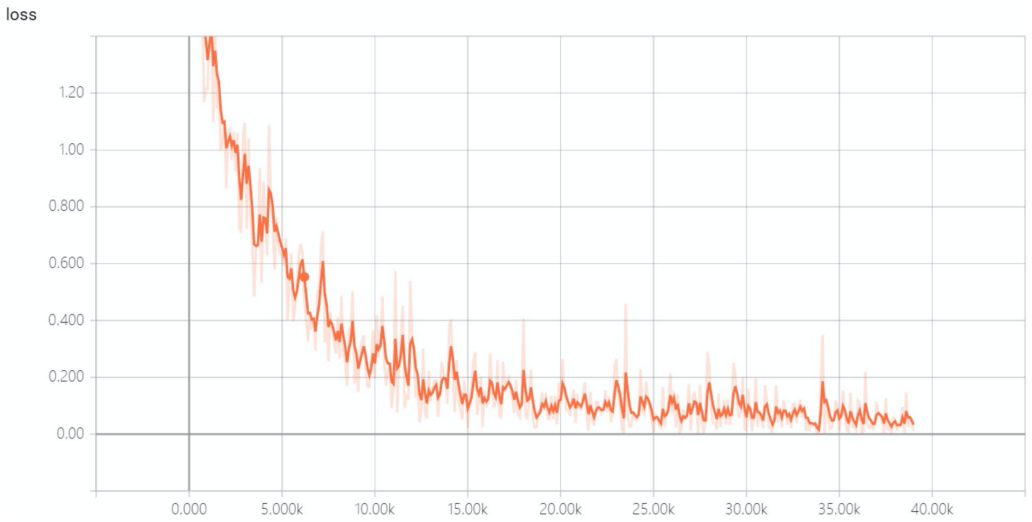
> You will see that many usually create their own loss function as sums of these defined above!

Loss Functions

So given a set of updated weights and our inputs, we can calculate our output by simply following the network through a forward propagation.

How can we measure how well our network is training?

> Loss is what we minimize on, so clearly that should be all that we can look at, right?
>
> > No! Need to look at the results of our network on some data that it has not seen to make sure it is generalizing well!



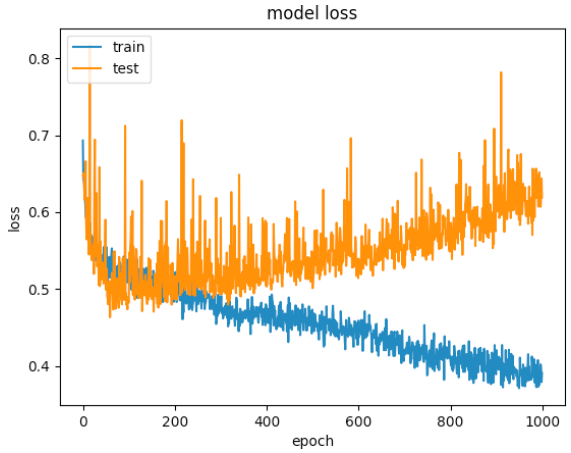
Loss Functions

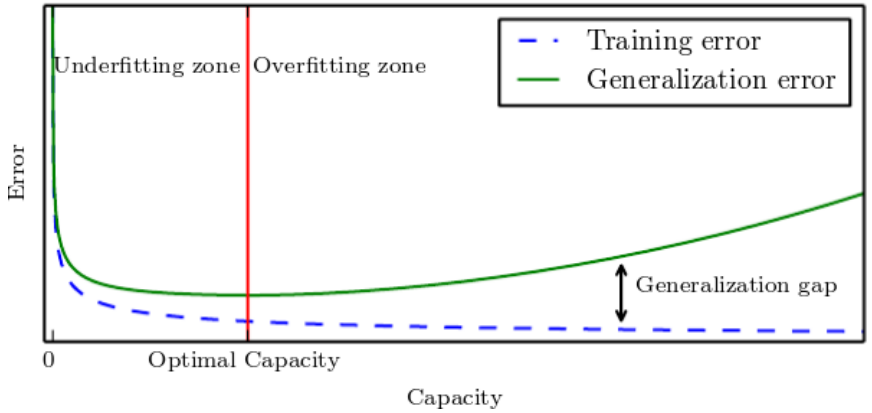
Loss gives information as to whether the model currently training is “learning” anything. It does not mean that what is learned is representative of the dataset.

- Always look at the training/validation error or accuracy as a metric of overfitting! Large separations indicate overfitting.

This means that we always have training/validation/testing split for our dataset.

- With a small dataset, try data augmentation or a different training technique.





3Blue1Brown

He explains many difficult mathematical concepts very intuitively so please check out his videos on deep learning if these concepts don't really make sense yet.

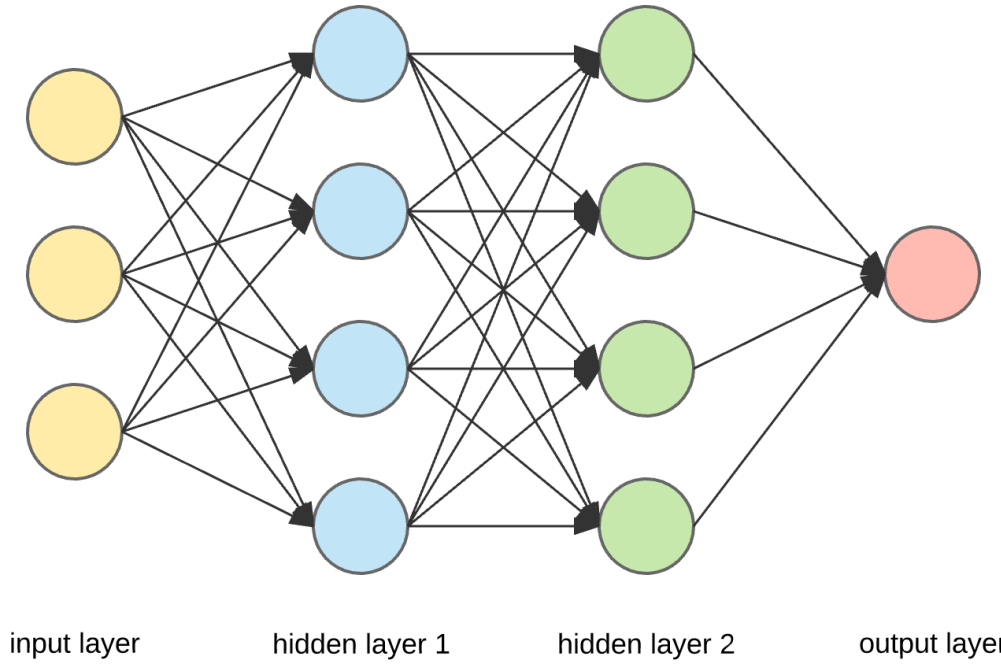
FEED FORWARD NEURAL NETWORK

Multilayer Neurons

Sometimes Feed Forward Neural Network is loosely referred to Multilayer Perceptron (MLP)

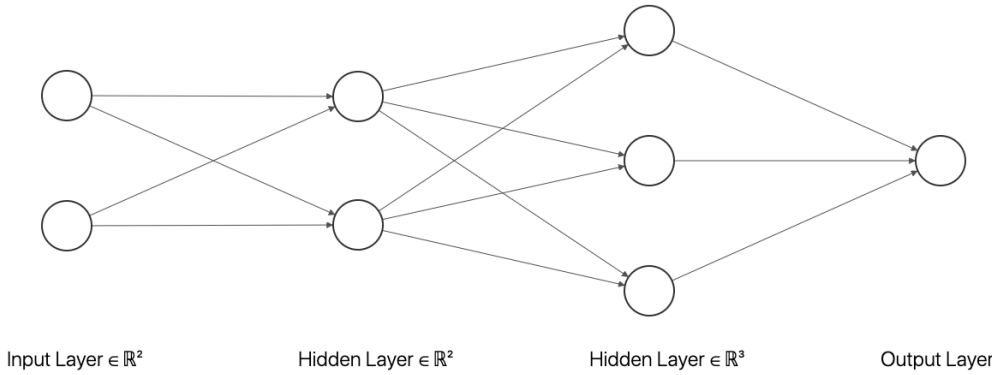
Created by arranging neurons into layers and collecting signals from the neurons between the layers

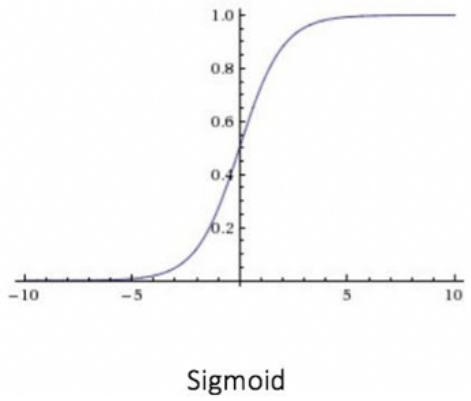
This is the most basic form of a “deep” Artificial Neural Network



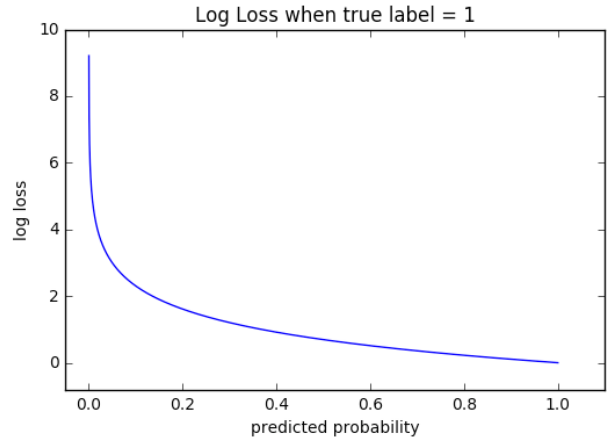
Algorithm

- Determine your network structure, activation function, and loss function
- Initialize the weights (however you choose but make sure the dimensions match up!)
- For n number of iterations:
  - Step1: Get the loss of your neural network using the initialized weights and find the gradient
  - Step2: Update the weights using the gradient and learning rate
  - Step3: Rinse and repeat until last iteration





Sigmoid



CONVOLUTIONAL NEURAL NETWORK

Images and Fully Connected Networks

FFNNs are great for applications where the number of features is somewhat small.

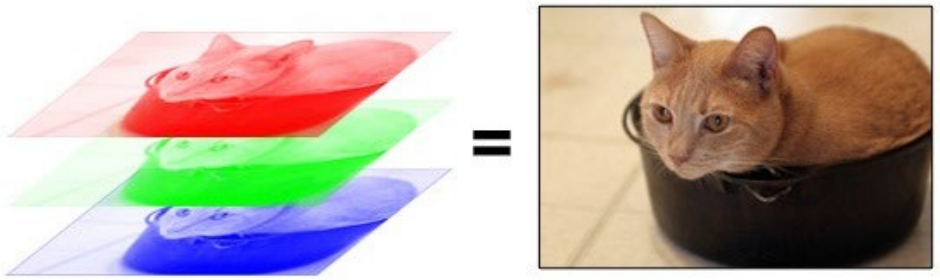
Consider the dimensionality of a 1080p image:

1920 x 1080 x 3 = 6220800 features

Larger the number of parameters to learn, the likelier it is for the model to overfit and the harder it is to learn!

Images change… Need a network that is stable against changes in the input image.

A cat can move, a camera's quality can change, the environment never stays the same, etc…

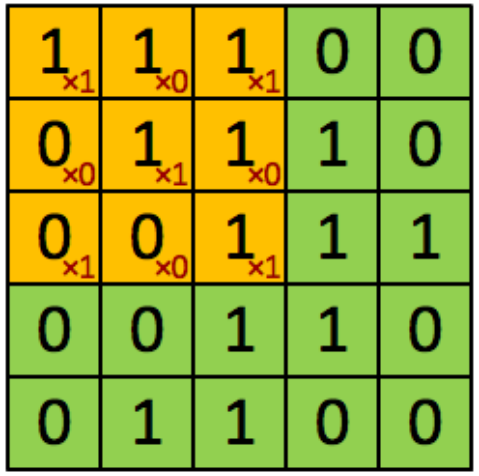


Convolution and Kernels

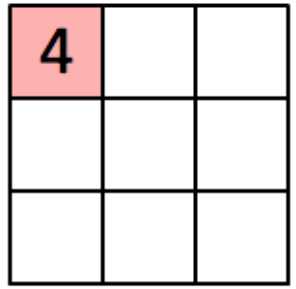
A starting point for CNNs is to understand how convolution works within a convolutional layer. Each convolutional layer contains a learned filter or kernel (seen in yellow on the right).

> A kernel can act as a detector for edges, curves, or more when it is slid and multiplied with image pixels.

By extracting desirable features from an image, dimensionality is reduced and a less sparse representation of the image is created.



Image



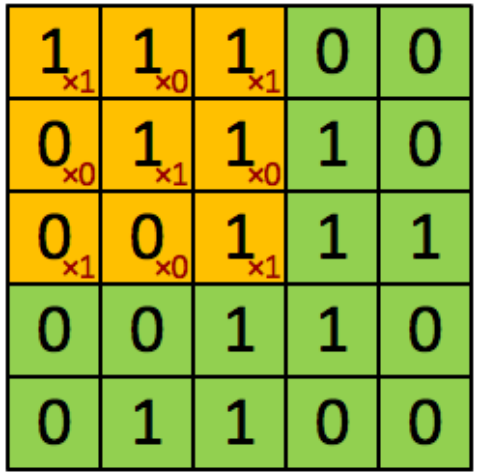
Convolved Feature

Convolution and Kernels

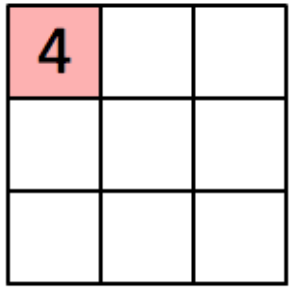
Convolutions are essentially linear transformations with parameter sharing occurring in the transformation matrix (not proven here).

Pipeline for a convolutional neural network can be summarized as a **dimensionality reduction with convolutional layers** and then **classification using fully connected layers**!

> Combine that with the shared kernel weights and you solve the two issues with using images as inputs!



Image

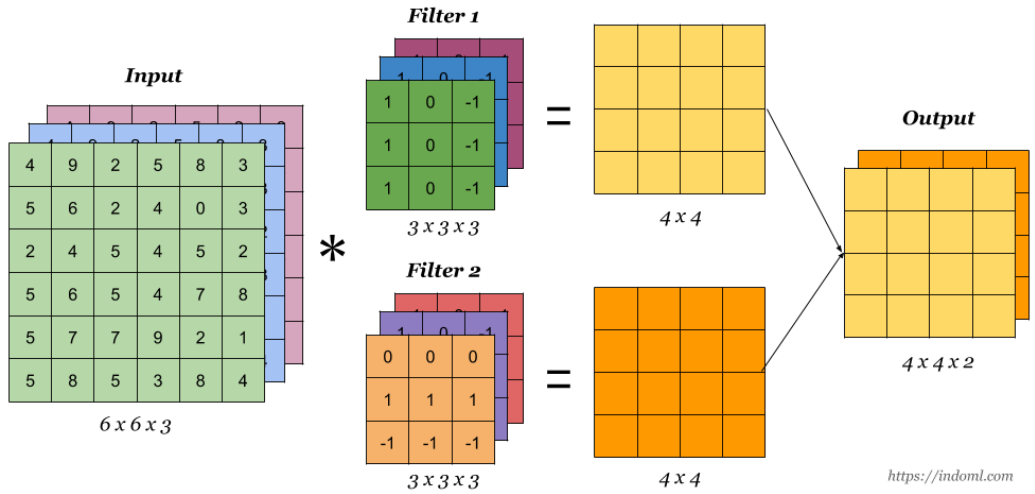


Convolved Feature

Convolutional Layers

While a single kernel might learn one particular attribute of an image, we generally want to learn several at a time to gain a better understanding of the image.

Convolutional layers consist of several kernels which are then convolved with the original image to generate the output channels.



Convolutional Layers (Attributes)

Stride

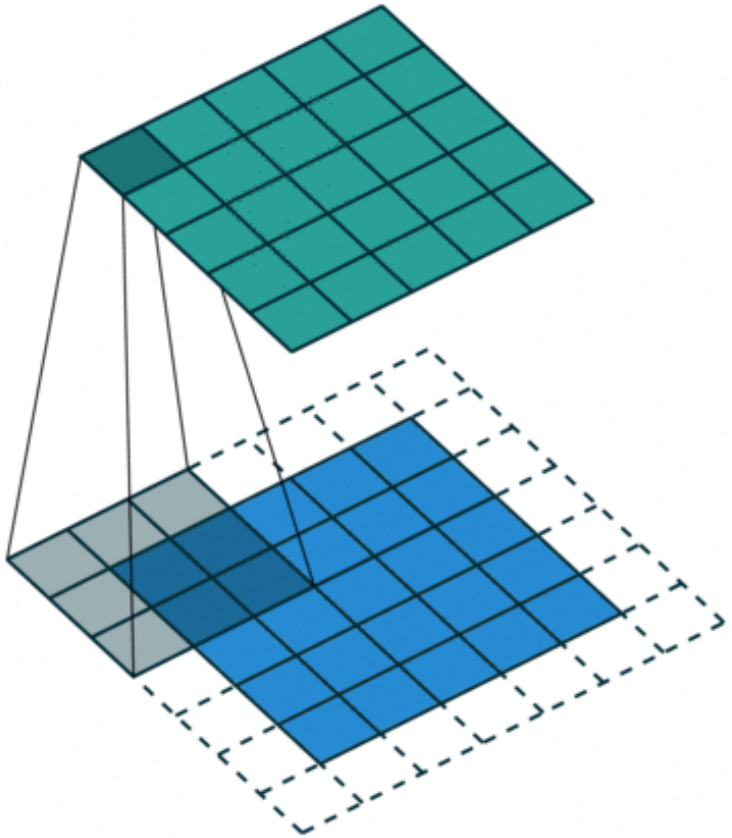
Amount by which window jumps as it passes through array.

Kernel Size

It is advised to not make them large as it has been shown that smaller kernels can still extract larger features as convolutional layers are applied in series.

Padding

Provides padding to reproduce certain size at output.



Batch Normalization (Layer)

Partially addresses the issue of (internal) covariate shift.

- As the network is trained, the layers can each experience a type of "covariate shift"

Remember that normalization on the inputs allows for easier training of certain algorithms.

- The same logic should apply for the activations in between layers! Remember that different batches generate different layer output distributions.

Can act as regularization due to noisy batch mean/variances

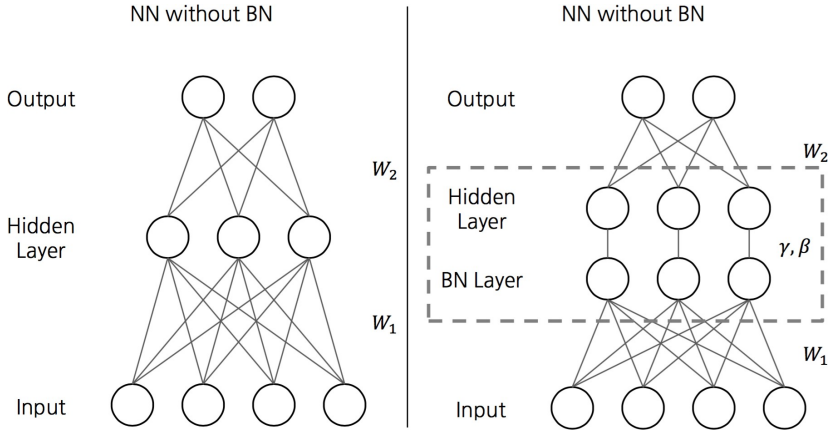
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

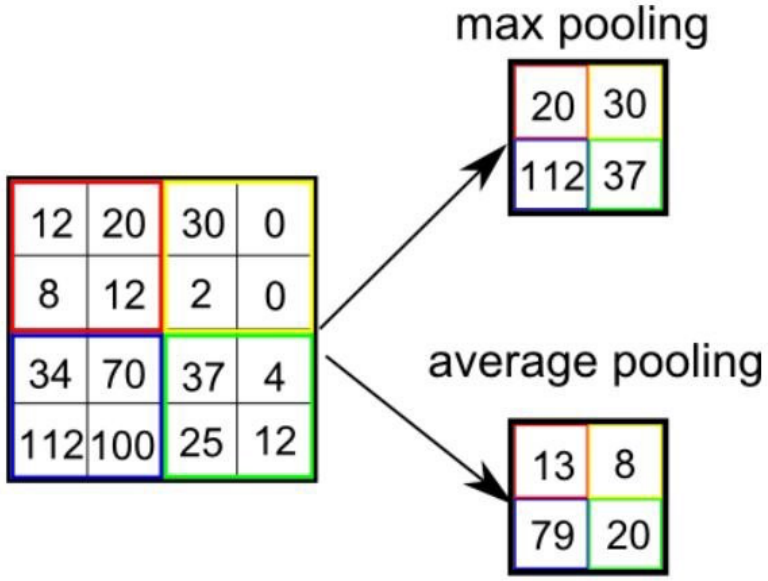


Pooling Layers

Pool the values in every $NN \times NN$ slice of the channels of the input and spit out either the average, maximum, or minimum of the slices with stride SS.

- Effectively a sliding window performing an avg/min/max operation over it's area.
- IE: The right represents two types of 2×2 pooling layer with a stride of 2.

But why?



Pooling Layers

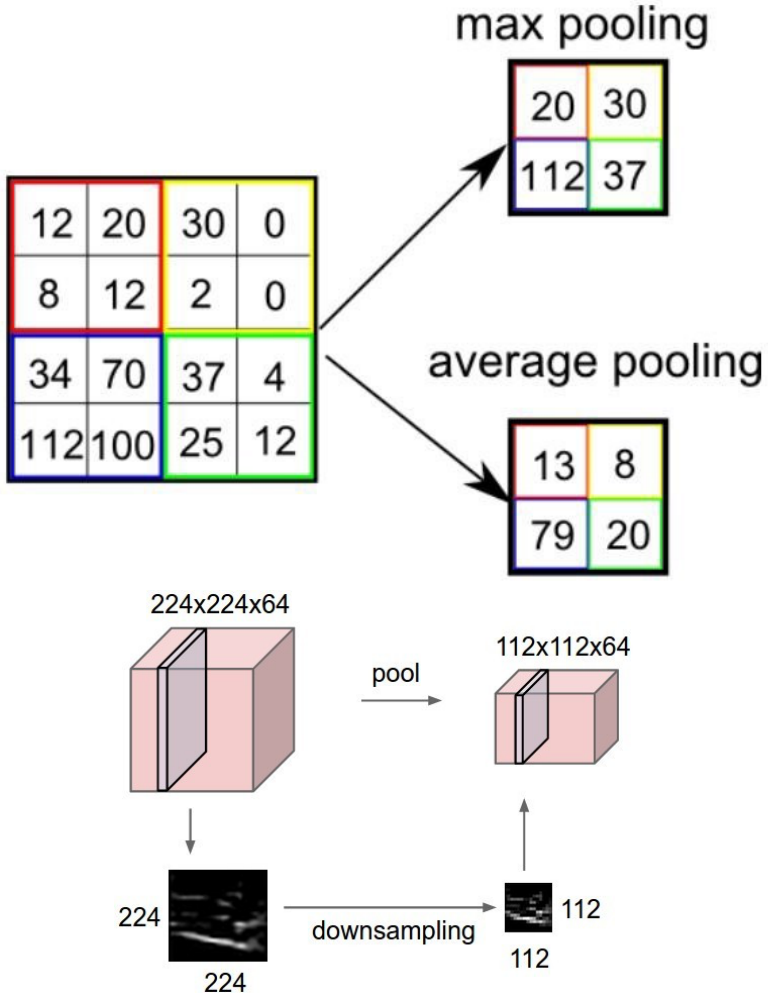
Dimensionality Reduction

This is entirely in line with everything discussed for convolutional layers.

Positional / Rotational Invariance

If we suppose the activation maps for a given convolutional layer are affected by a change in the location of the subject in the input, then pooling these activations should lessen the impact of the change!

Do not make them too large or you lose too much information!



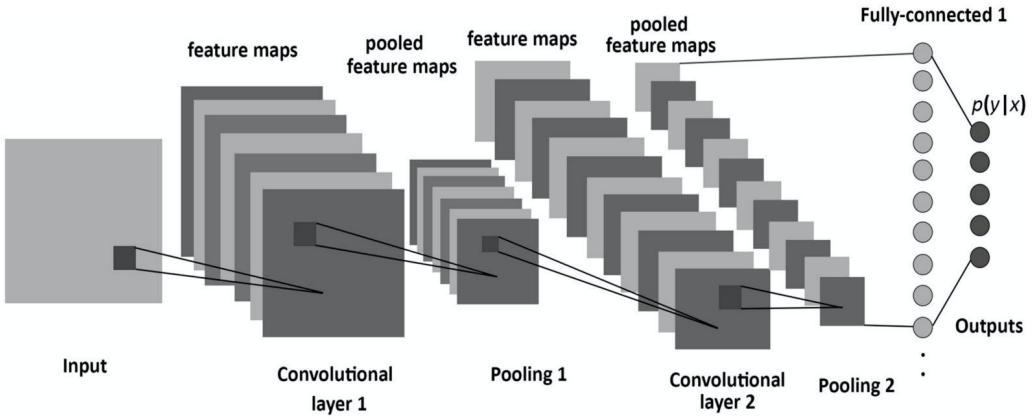
Fully Connected Layers

Luckily, convolutional and pooling layers make up most of what makes convolutional neural networks unique!

The fully-connected layer is simply neurons with flattened feature maps attained from the last convolution as the input.

> Flattened, meaning, all values are concatenated into a long vector across rows and then channels.

No particular way of building CNNs exists either. Use experience to guide your model building…



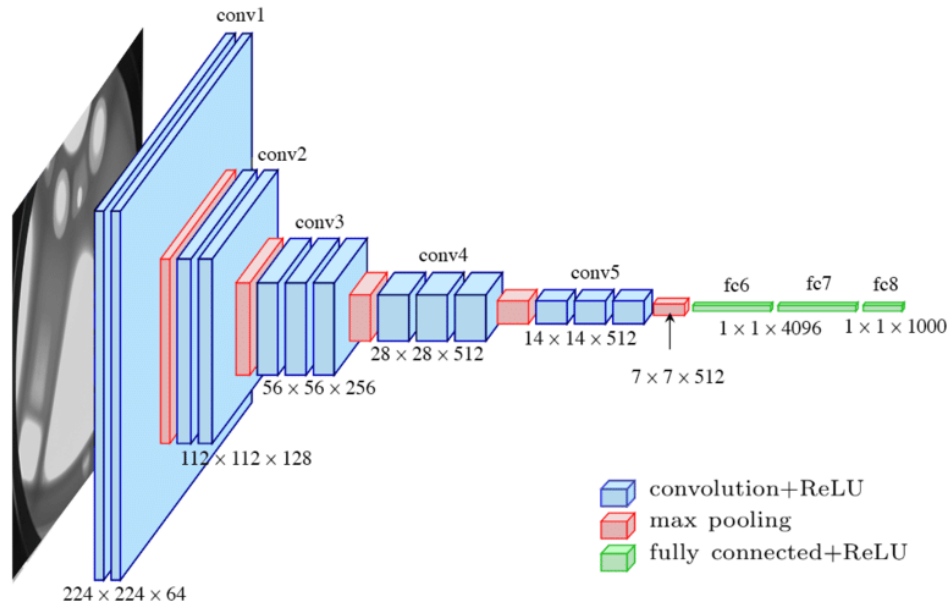
Convolutional Neural Networks

With all of these tools in hand, we are ready to discuss convolutional neural networks!

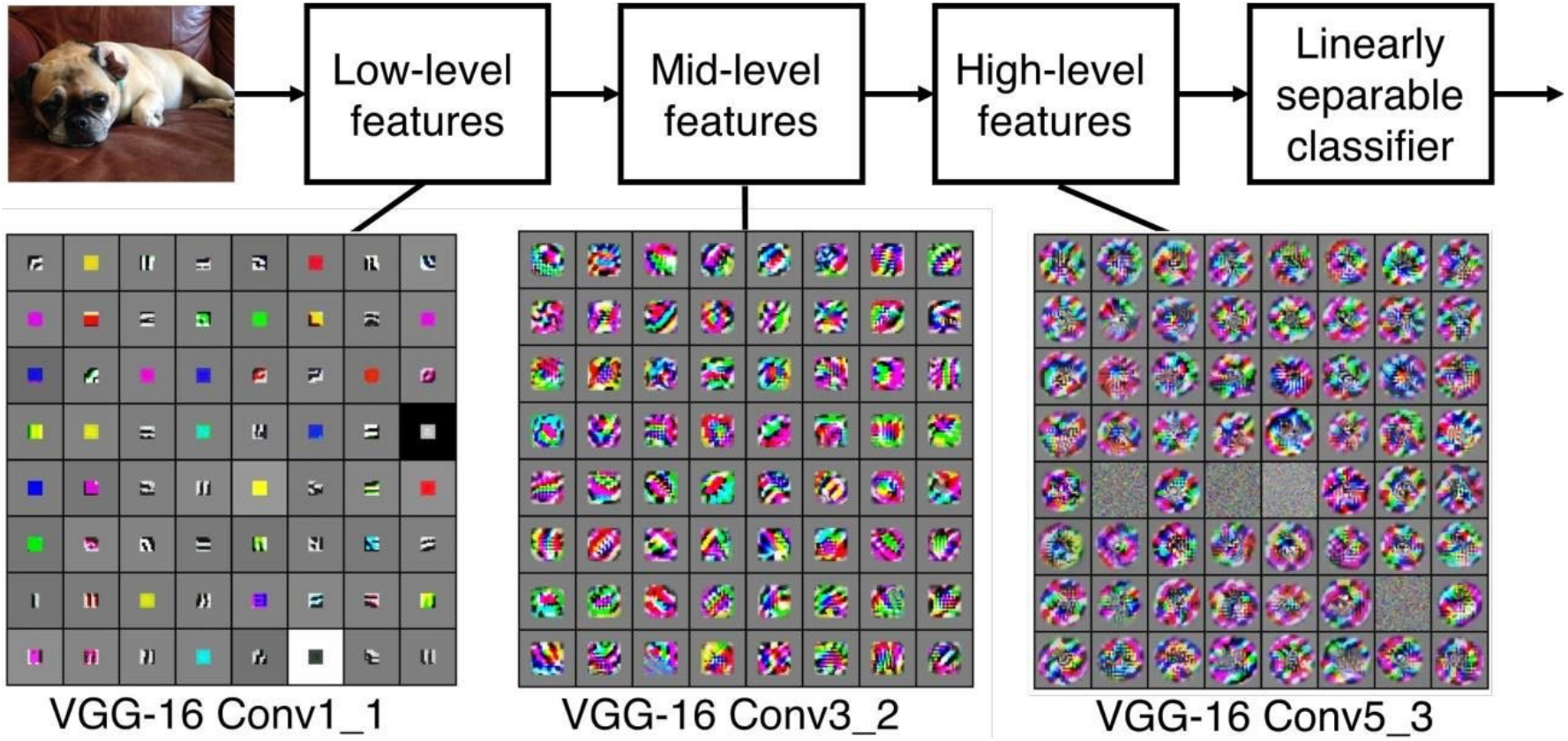
VGGNet is a CNN architecture with networks that consist of just convolutional layers and pooling layers

These make good study cases for people interested in convolutional neural networks. Since there is no complicated structure, it tends to be quite straightforward and used for quick training and inference!

Perfect example of how features become higher level as the inputs propagate through the network.



What Does A CNN Learn?



Common CNN Applications

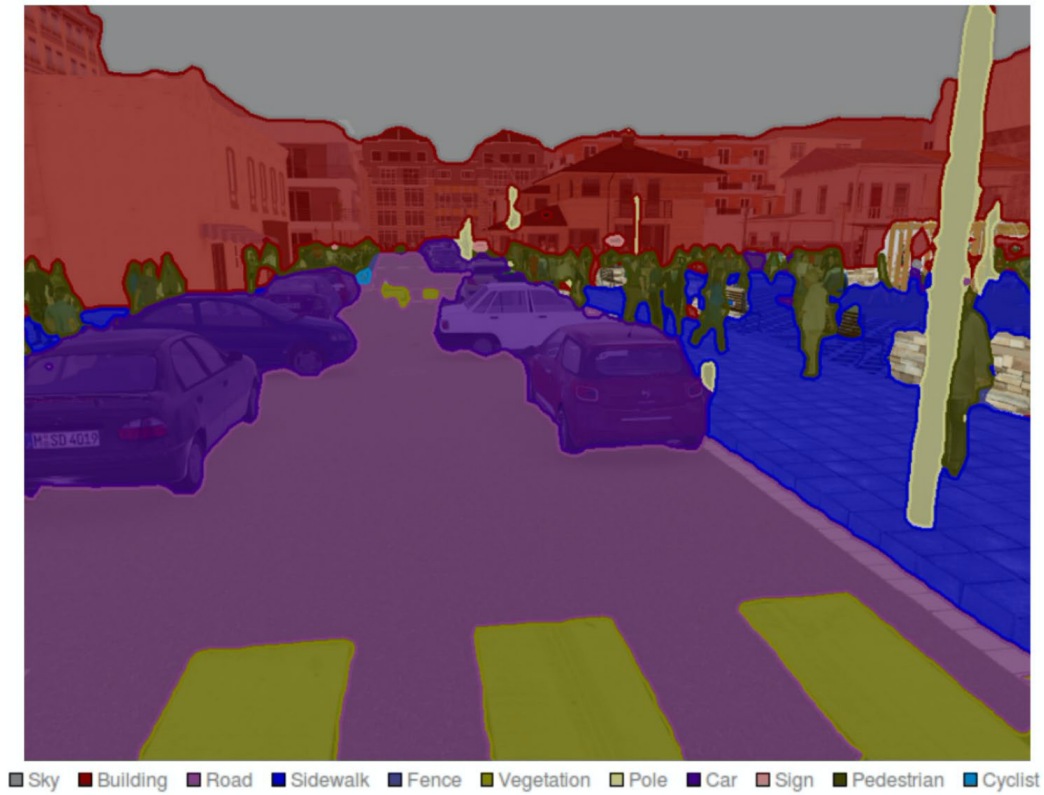
Feature Extraction

Image Classification

Scene Labeling (Segmentation)

Image Generation

Image Analysis



More on CNN

Read more on CNN architectures (advanced):

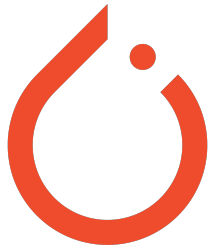
- LeNet
- AlexNet
- ZFNet
- GoogLeNet
- VGGNet

PyTorch

It's a Python-based scientific computing package

A replacement for NumPy to use the power of GPUs

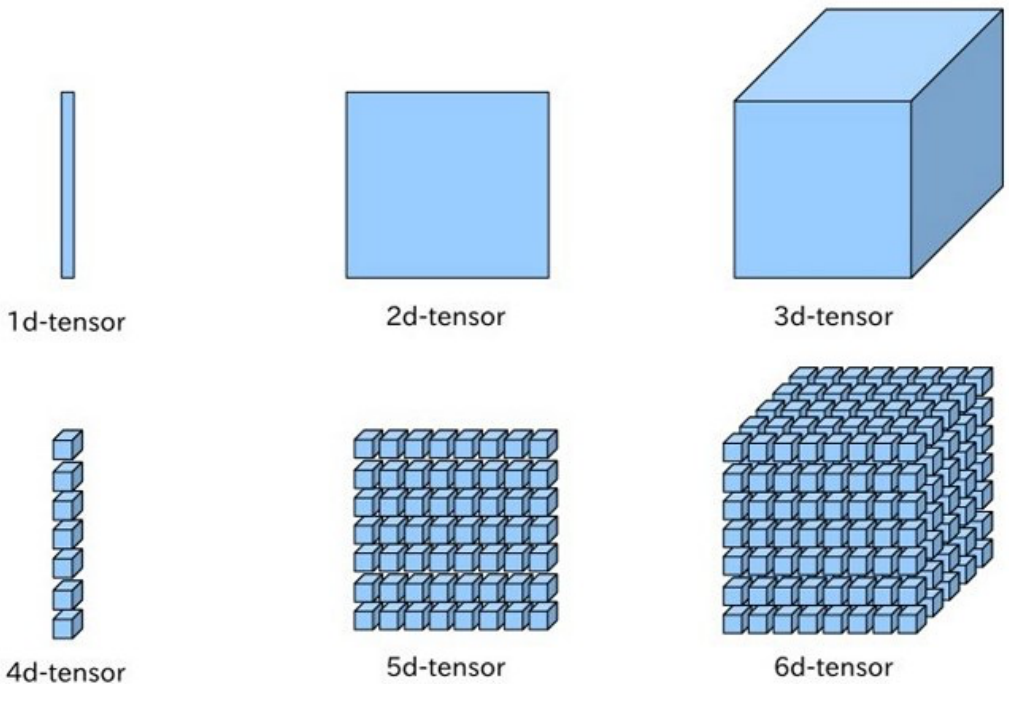
A deep learning research platform that provides maximum flexibility and speed



Tensors

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing

Essentially a representation of data AND linear transformations, both of which form the core of most learning algorithms.



Pipeline

PyTorch provides the elegantly designed modules and classes, including torch.nn, to help you create and train neural networks.

An nn.Module contains layers, and a method forward(input) that returns the output.

Follow this tutorial.

WHAT ELSE?

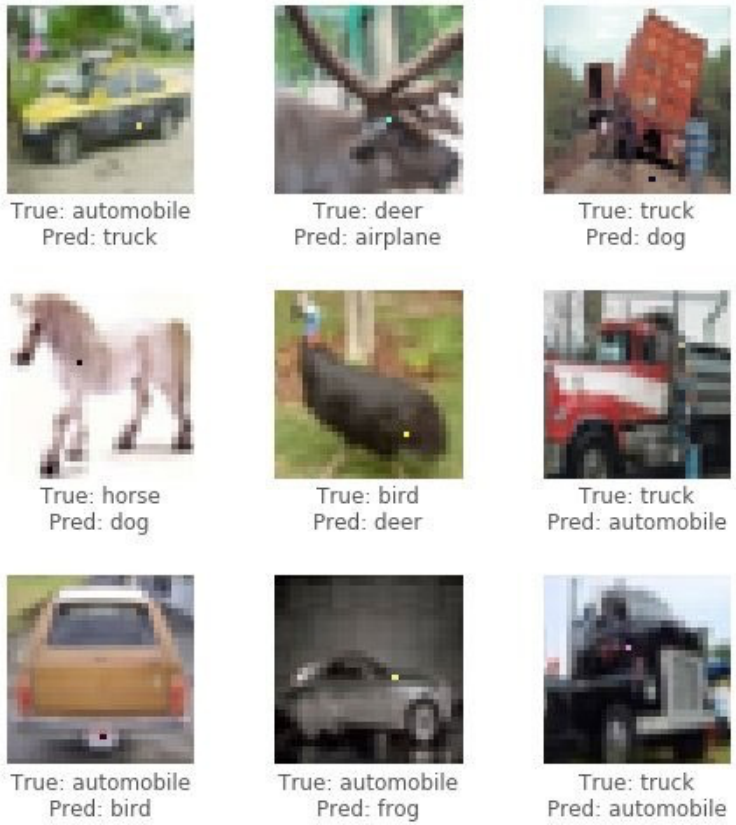
Dataset Considerations

Insufficient Intra/Inter – Class Variation

- Symptoms:
  - Good Training and Testing Accuracy
  - Bad Generalization when Dataset Resampled
- Remedies:
  - Re-sampling Data / Data Augmentation
  - Constraining Problem

Dataset Poisoning (Incorrect Distribution)

- Symptoms:
  - Good Training / Bad Testing Accuracy
- Remedies:
  - Inspecting & Removing Incorrectly Labeled Samples



Dataset Considerations

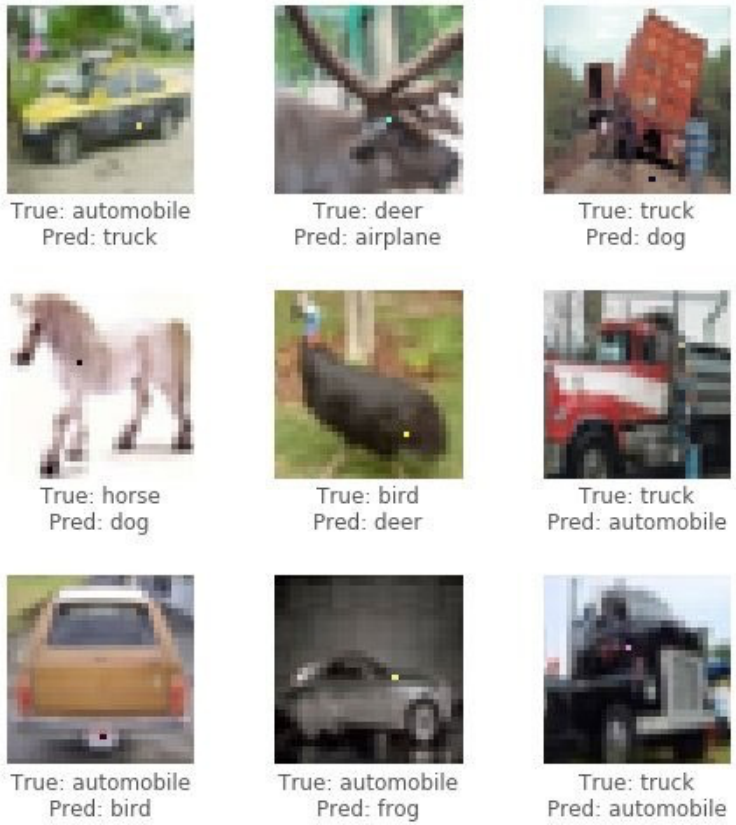
Class Imbalances

Symptoms:

- Good Overall Accuracy / Terrible Class-wise Accuracy

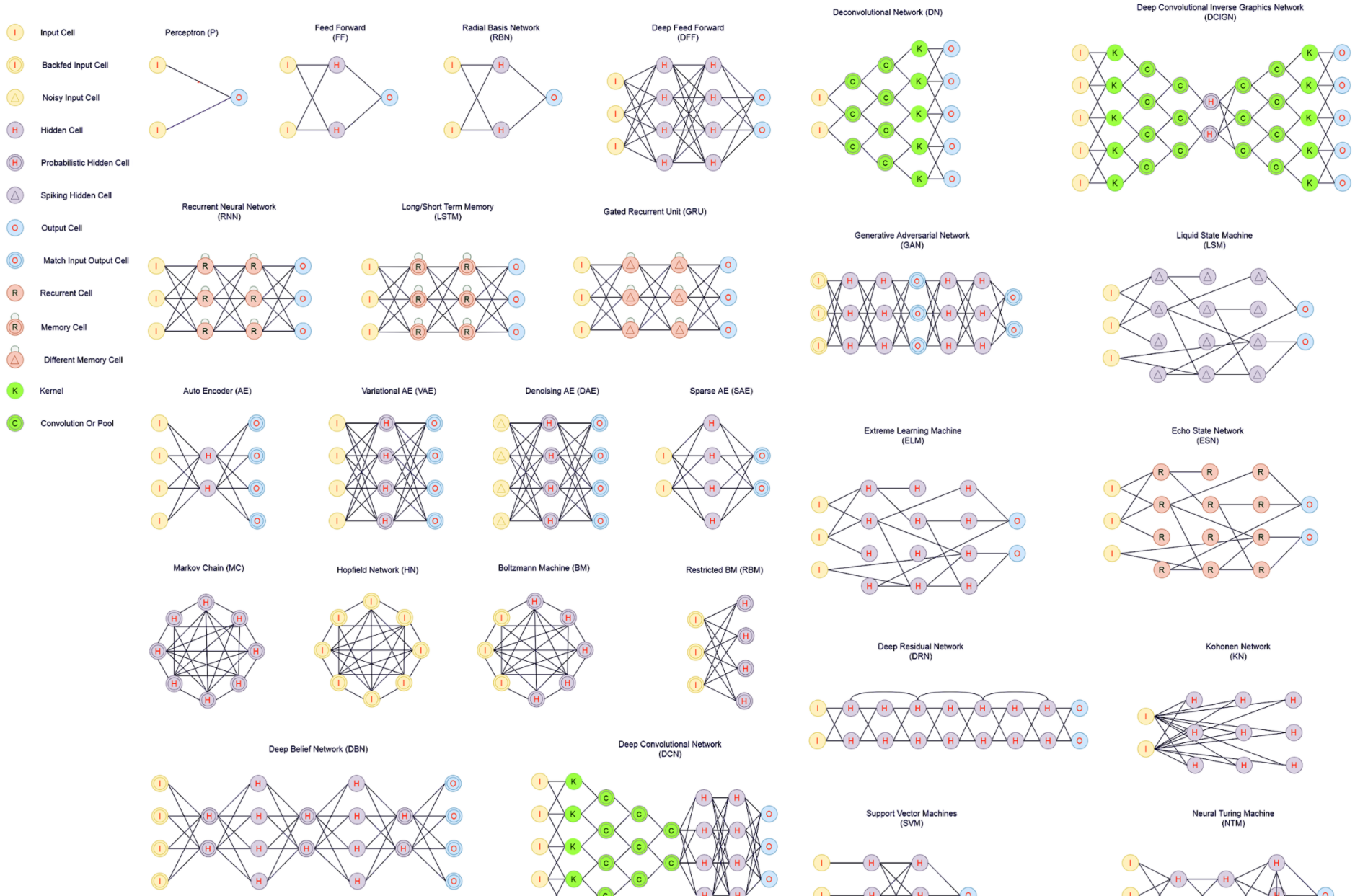
Possible Remedies:

- Weighed Loss Function
- Data Augmentation
- Distribution Estimation (Data Creation)



Types of NN and their applications

Check out this site if you'd like to implement deep learning for a problem but don't know where to start.



Main Types of Neural Networks

QUESTIONS