Intro to Deep Learning using TensorFlow

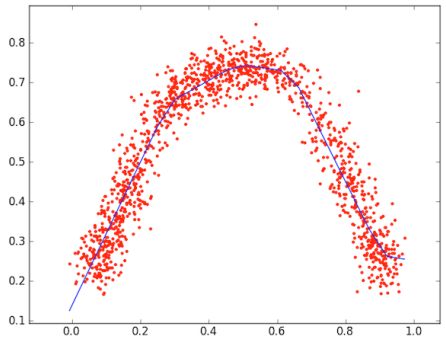
What is TensorFlow?

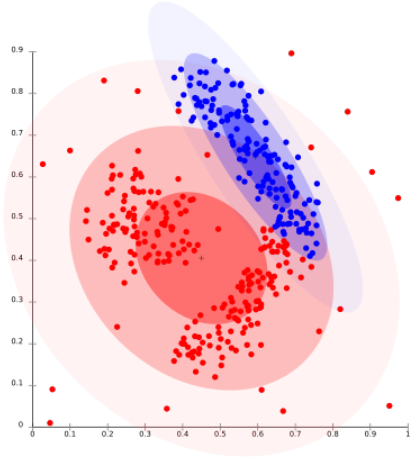
"TensorFlow™ is an open source software library for high performance numerical computation. […] it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains."

What Can I Use It For?

At its simplest, it can be used for optimization of almost any supervised or unsupervised learning algorithm.

- Supervised → We have inputs and outputs and want to find a model that describes them. [i.e. Regression/Classification]
- Unsupervised → We have only the outputs and want to make observations of the data. [i.e. Clustering]
- Use in reinforcement learning as QFA/VFA is also used, but that won't be covered here!

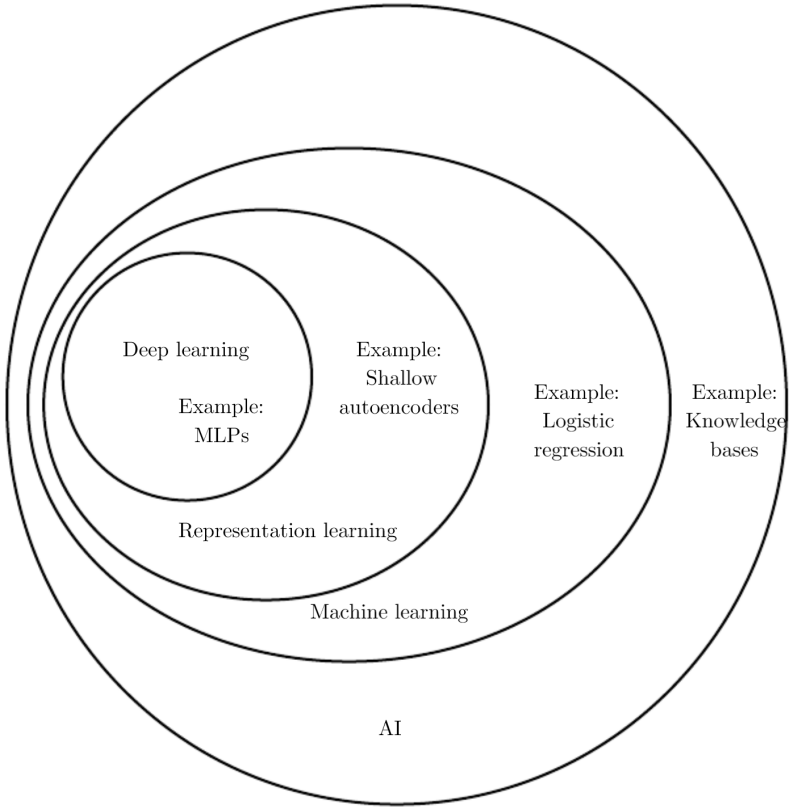




Deep Learning?

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

Is a form of representation learning, where an algorithm tries to represent something with high-level features as a collection of lower-level features.

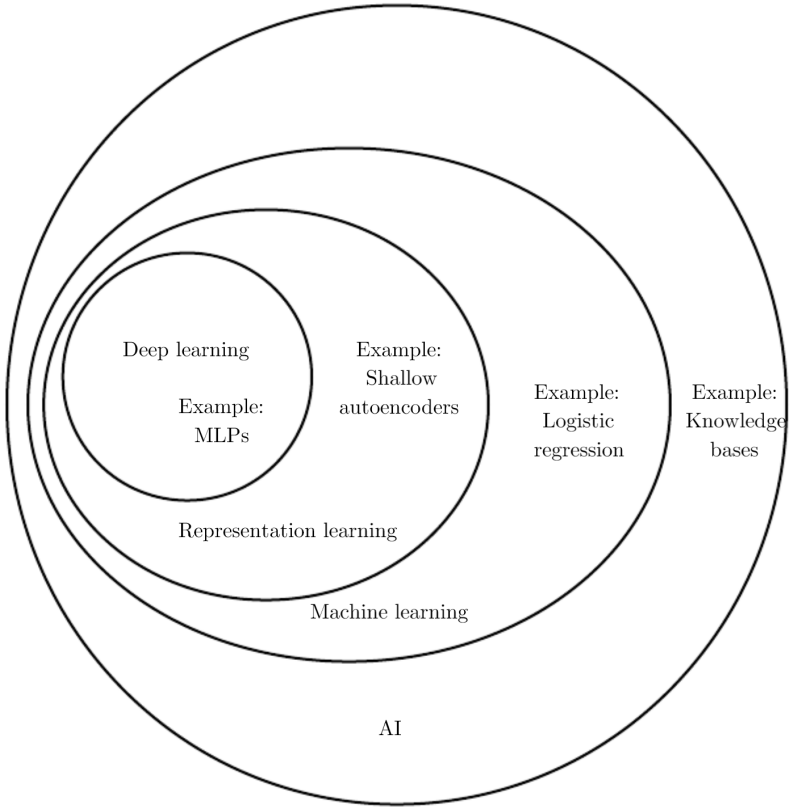


Deep Learning

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

- Essentially perform representation learning many times in a single model!

Learning is just a feature intrinsic to most ML models as they tend to make generalizable predictions given an incomplete dataset.



Tensors and Data Flow

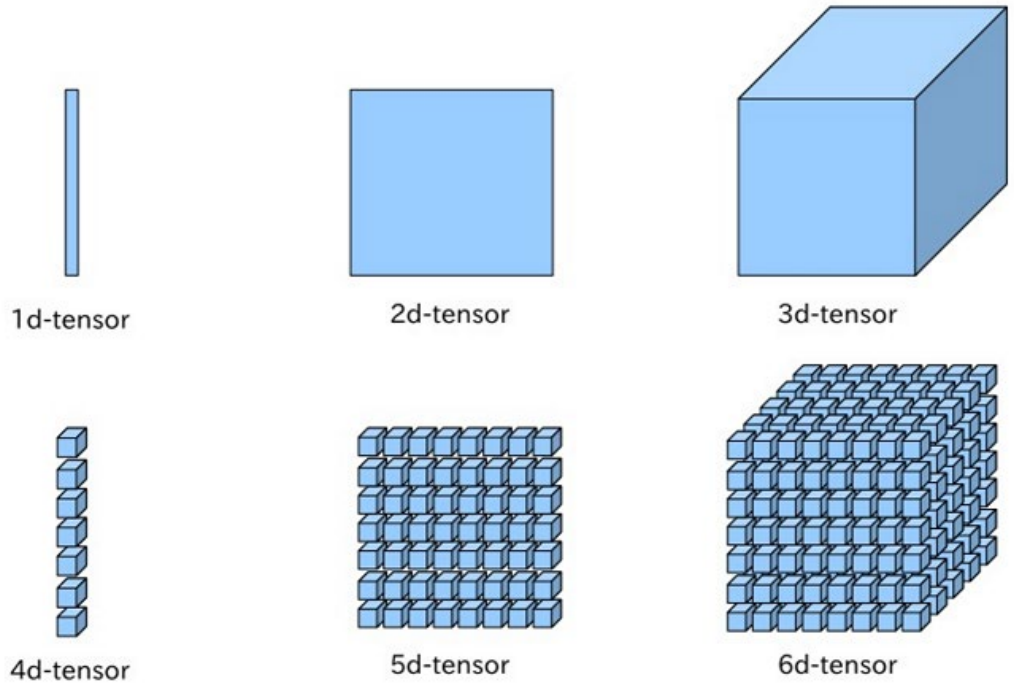
Tensors

An n-dimensional matrix of form…

- $X \in \mathbb{F}^{m_1 \times m_2 \times \cdots \times m_n}$

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

Most of these are familiar in the form of scalars, vectors, etc…



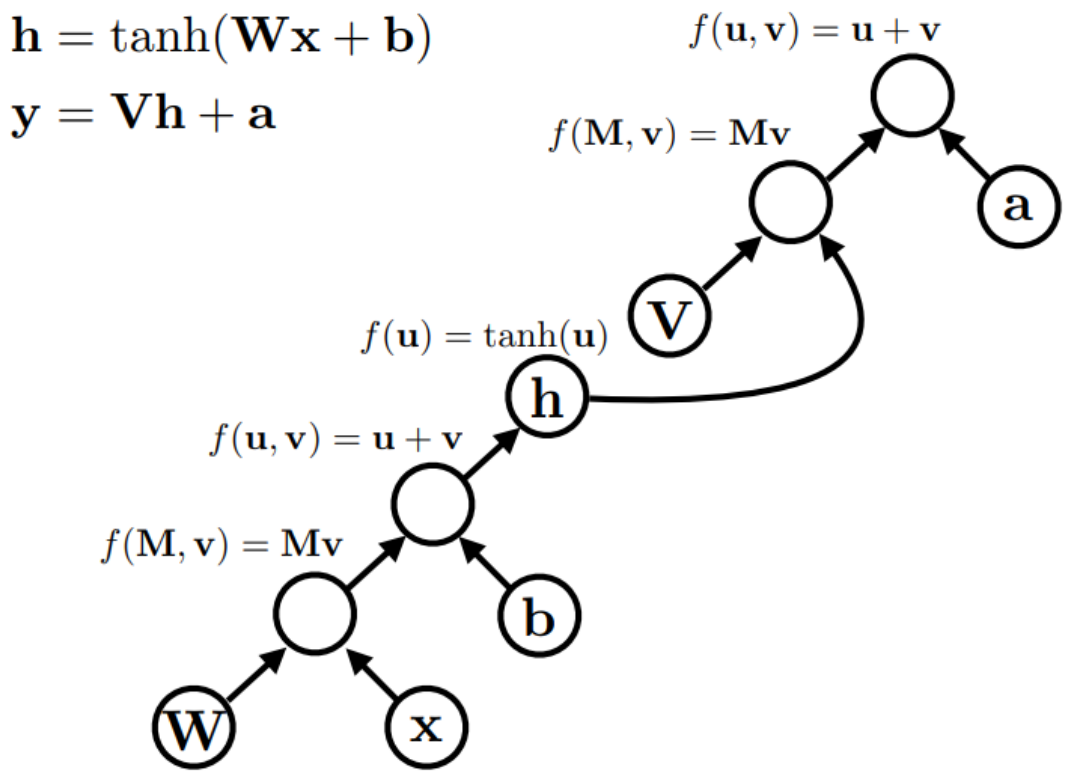
Computation Graphs

In TensorFlow, all computations are recorded in graphs that are executed only upon starting a session.

To be formal, this directed and acyclic graph (DAG) is designed such that the nodes represent operations/variables and the directed edges represent data transfer.

Why?

- Increased Parallelization
- Less Computational Waste



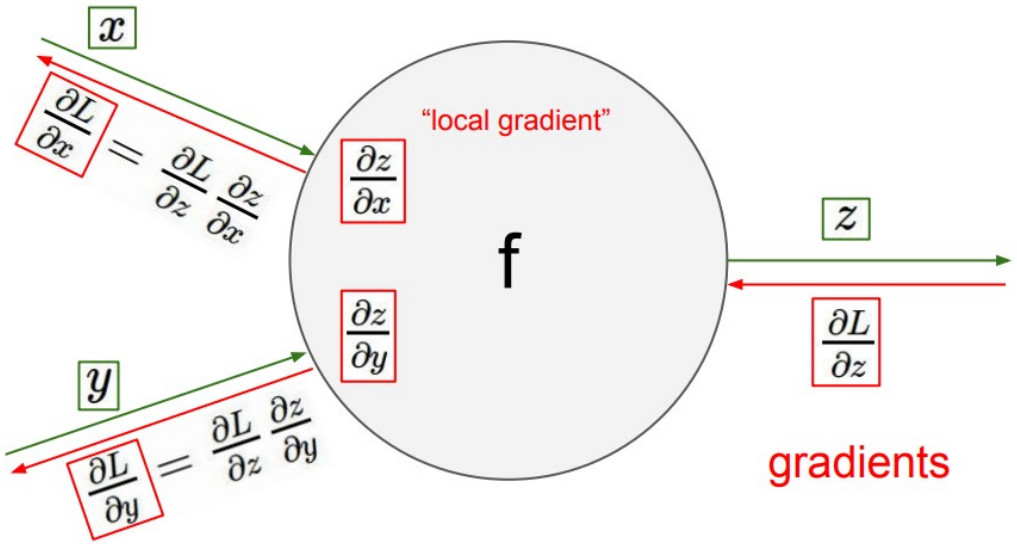
The Core of Backpropagation

Graph optimization performed via simple calculus.

- The difficulty is obscured from the end-user!

$$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx}$$ [Chain Rule]

Backpropagation is intuitive and can be optimized significantly using computational graphs.



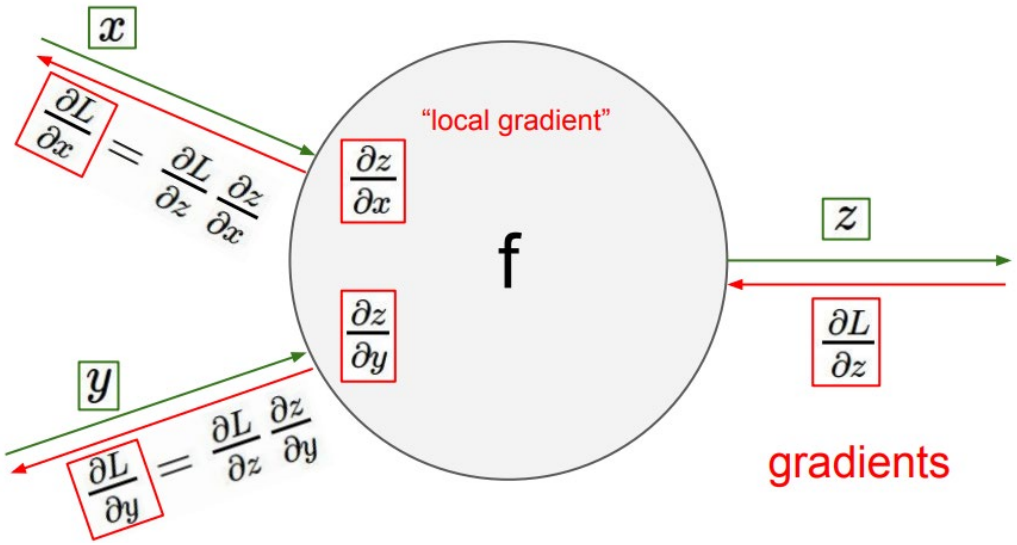
The Core of Backpropagation

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

- https://cs231n.github.io/

Do yourself a favor and implement backpropagation statically at least once. You will definitely not regret it and learn a lot at the same time!

Or take some courses offered here such as ECE 285, ECE 271B, CSE 190/253, CSE 15x/25x, COGS 181

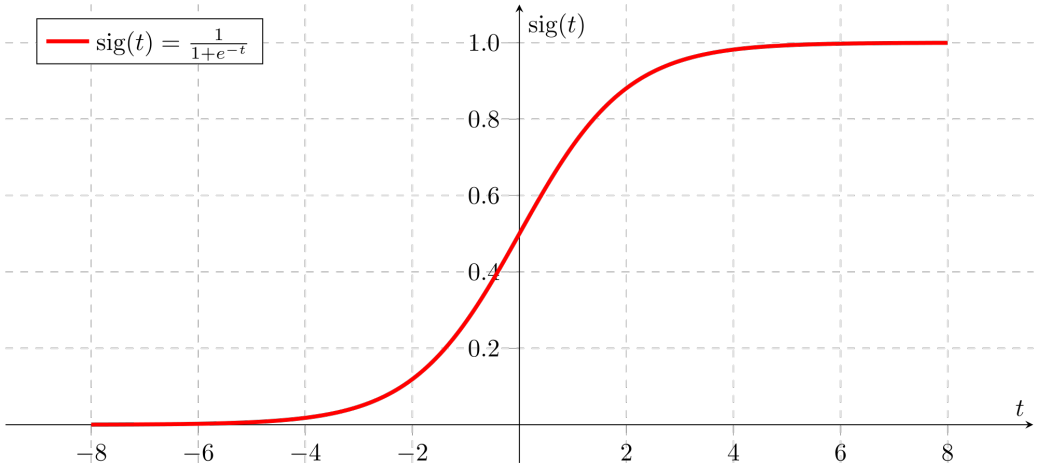


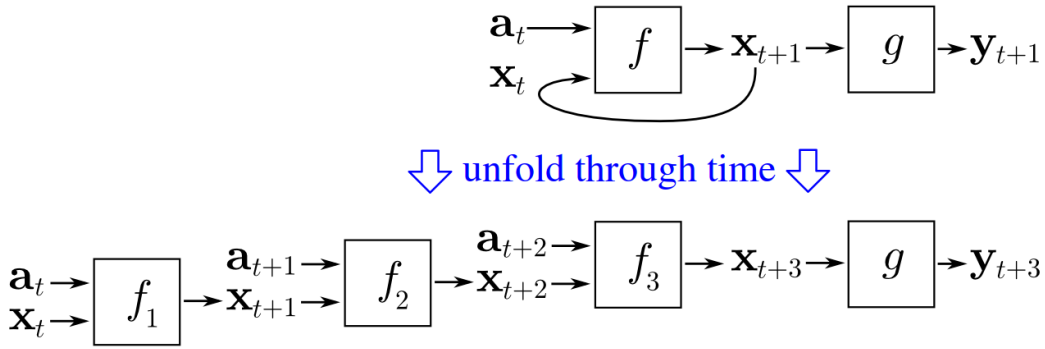
So Can I Optimize Anything?

Maybe! Keep in mind that derivatives aren't perfect.

- Vanishing gradients
- Exploding gradients

These are common problems that are usually addressed through proper initializations (Xavier/He) or architectural choices.





Deep Learning Primer (MLP Networks)

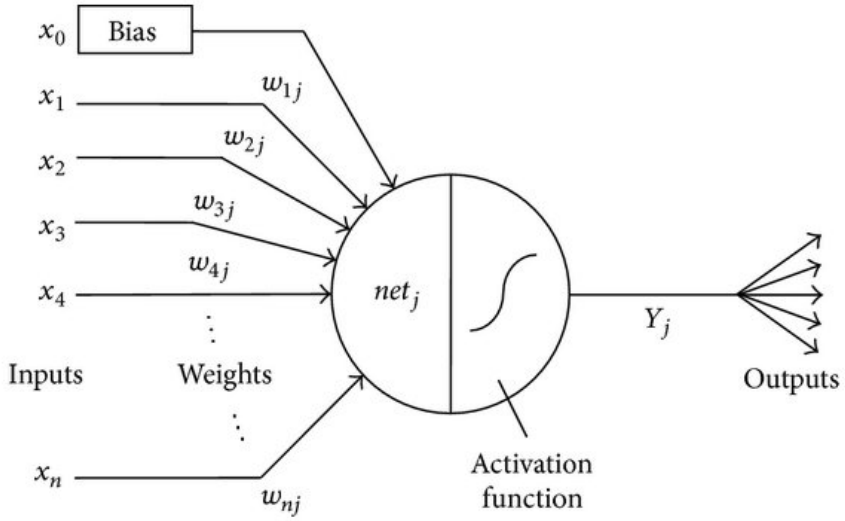
The Basic “Neuron”

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

These neurons are composed of:

- **Inputs**, labeled x_n [AKA **features**]
- **Weights** associated with each input, w_{nj}
- An **accumulation** of the product of each input and weight, $s = \sum_n x_n w_{nj} = \overrightarrow{w_j}\boldsymbol{x}$
- The **activation function**, $\phi(s)$
- The **output**, $y_j = \phi(s)$

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



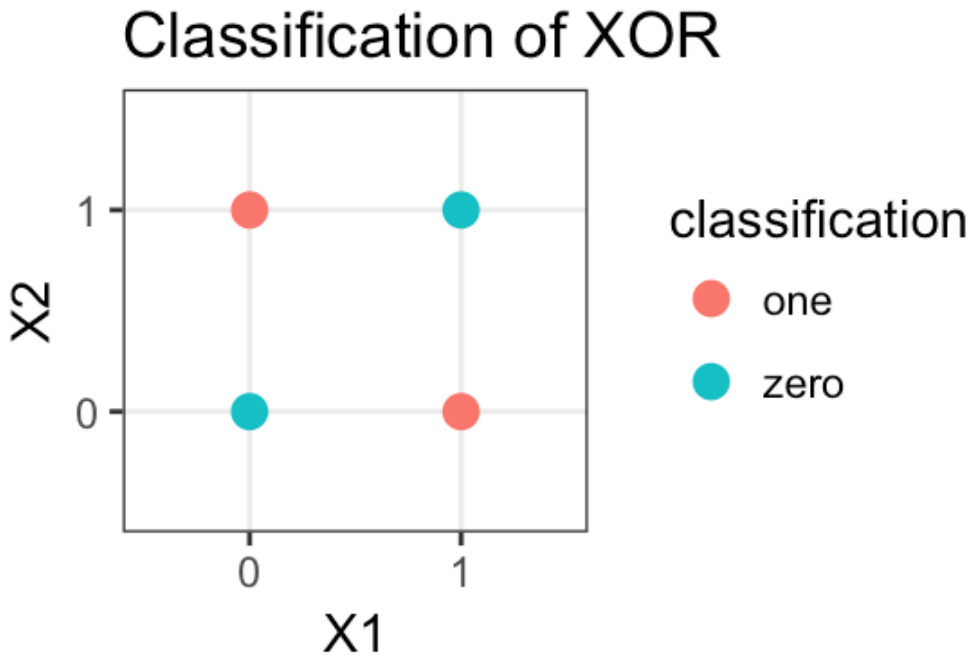
Why Non-Linear?

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.

We will be solving this example in the live demo…

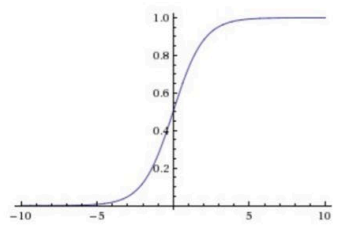


Activation Functions

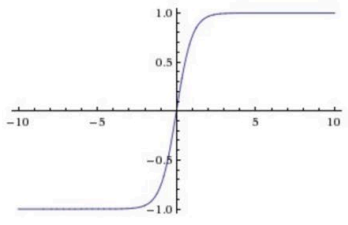
The activation functions in the output layer determine the function of the network. For example:

- Sigmoid →Conditional Probability, $P(y|x)$
- Tanh → Shifted Conditional Prob.
- ReLU → Strictly Positive Regression
- ReLU Alternatives
  - Useful mainly for preventing vanishing gradients (What happens if ReLU gradient taken on left?)

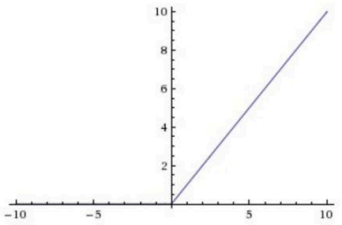
Softmax is generalization of sigmoid to n different outputs. [Useful for classification, $\arg\max_i P(y = i|x)$]



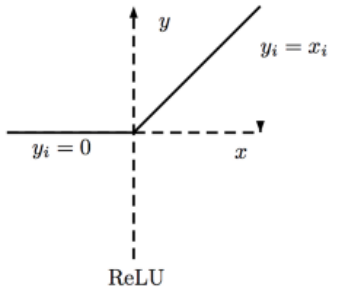
Sigmoid



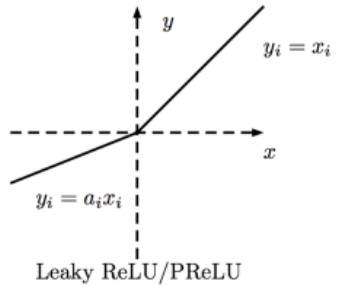
tanh



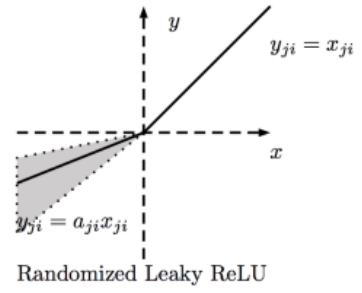
ReLU



ReLU



Leaky ReLU/PReLU



Randomized Leaky ReLU

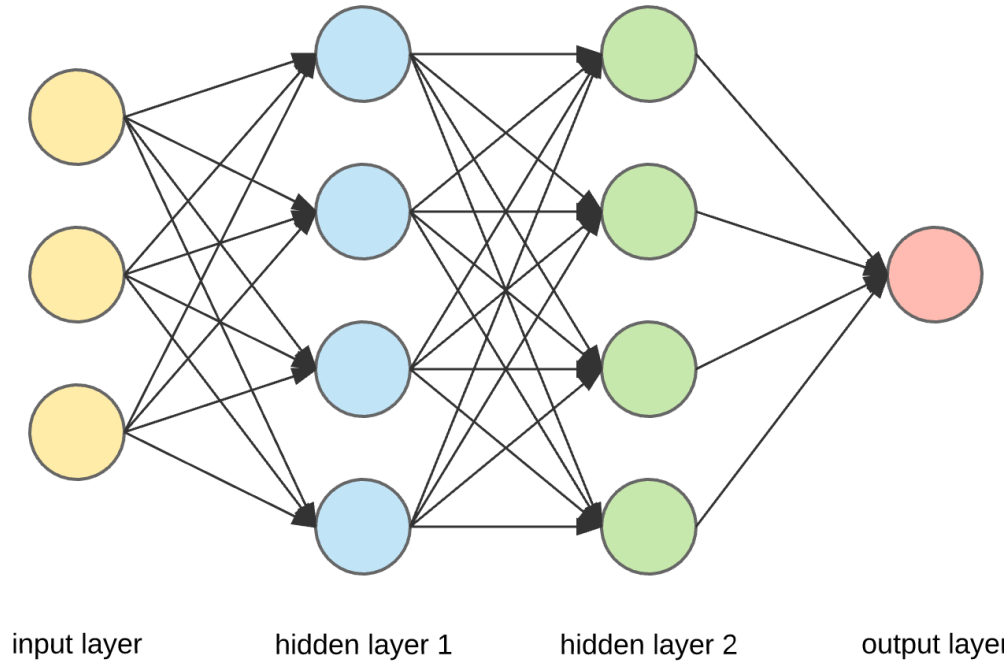
Multilayer Perceptron

MLP is created by arranging these neurons into layers and collecting neurons between layers.

- This is the most basic form of a “deep” neural network.
- Note that we can collect all edges into sets of weights (W_1, W_2, W_3)

Output:

$$y = \phi_3(W_3\phi_2(W_2\phi_1(W_1x + b_1) + b_2) + b_3)$$



Multilayer Perceptron

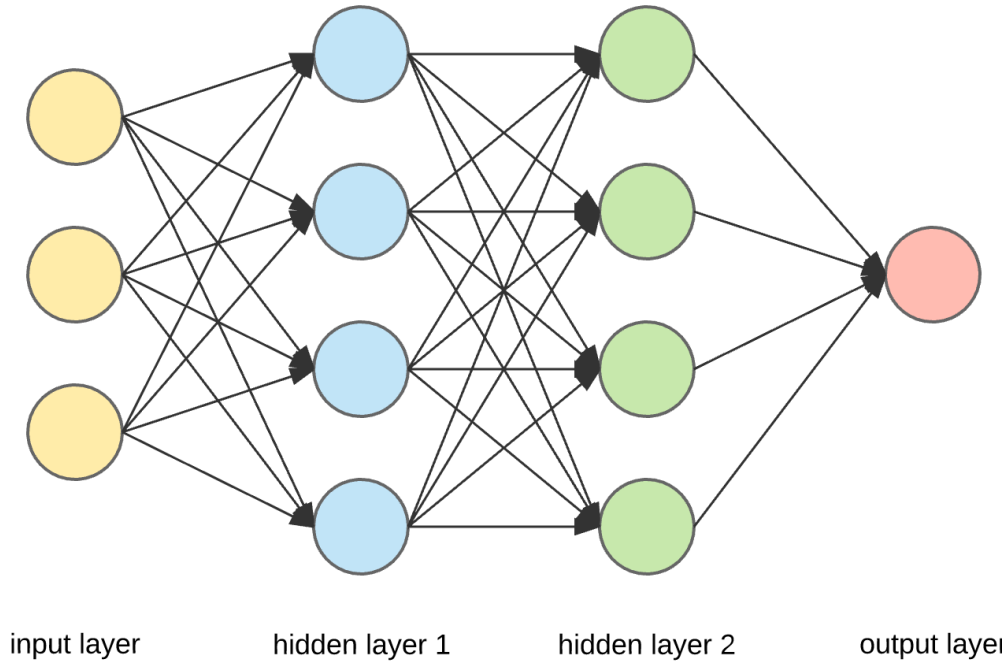
Assume that $\forall n,\ \phi_n(x) = x$ and $b_n = \mathbf{0}$

Notice that the equation before becomes:

- $y = (W_3 W_2 W_1)x = W_c x$

In other words, linear hidden layers completely defeat the point of the network, which is to learn complex functions!

- A useful pointer is to generally use ReLU for every hidden layer!



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!

Assuming $L(\boldsymbol{x})$ is our loss function:

- $\Psi_{\{W_n, b_n\}} = \arg\min_{W_n, b_n} L(\boldsymbol{x})$

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!

Universal Approximation Theorem

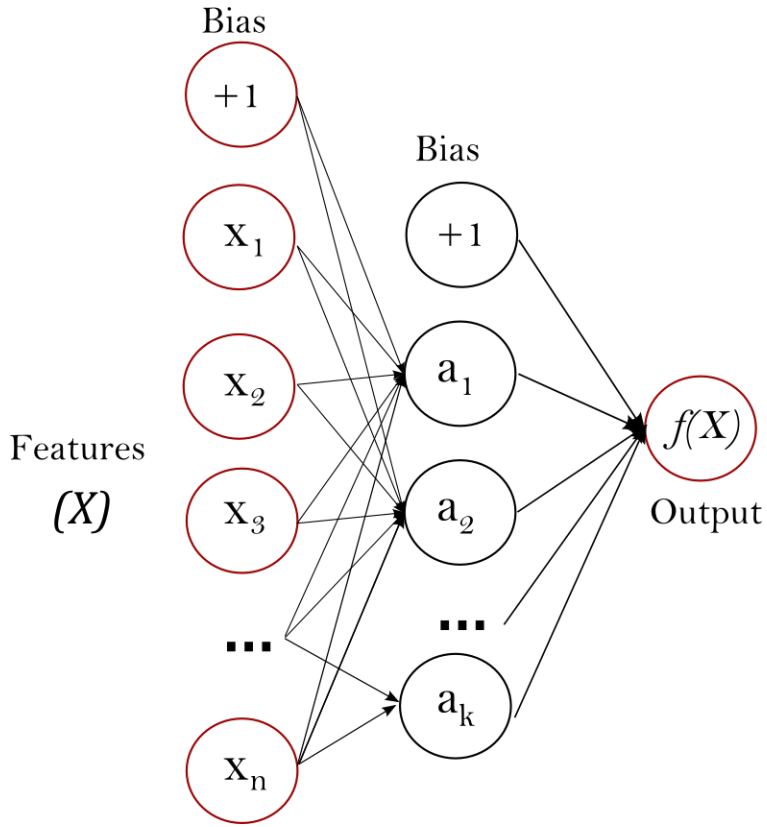
States that ***any bounded and continuous algorithm can be approximated by a neural network with only a single hidden layer consisting of n hidden neurons***.

- A perfect representation of the "No Free Lunch Theorem": ***a more complex model does not always correspond to better generalizations***.

How many neurons for the hidden layer?

- Unfortunately, no definitive answer...

No observation made on trainability of problem!



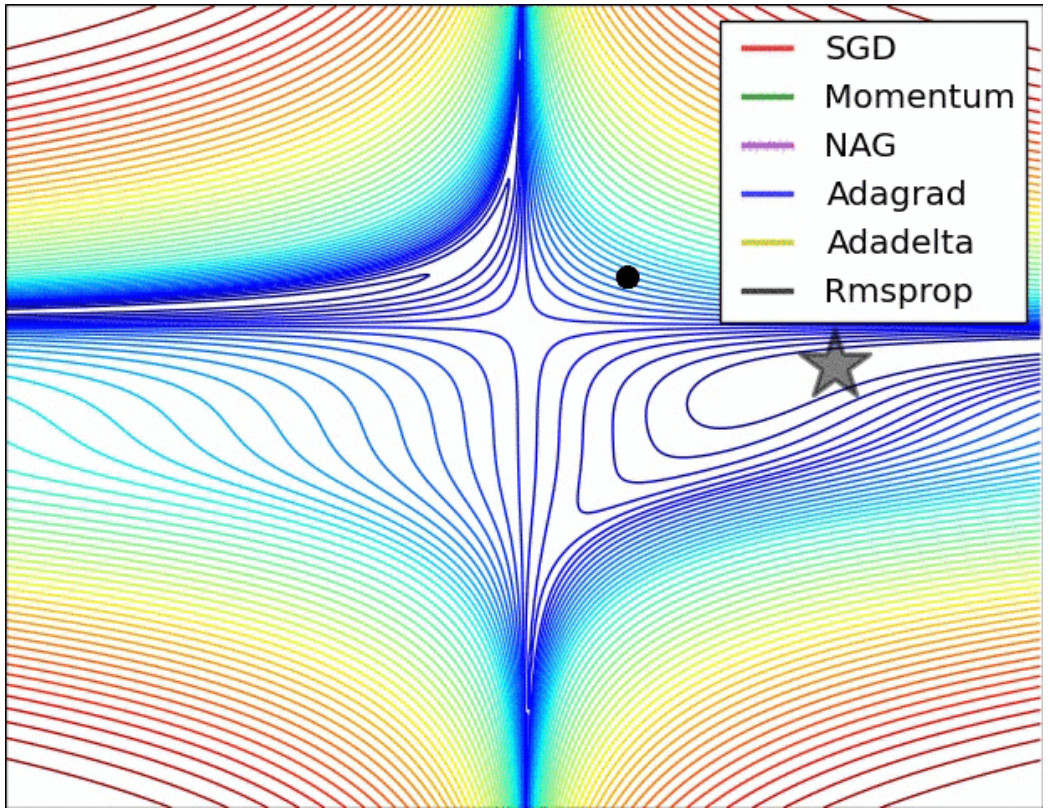
Training the Network

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.
- $p \leftarrow p - \alpha \nabla_p F(\boldsymbol{x})$ [General form of Update]
- Backpropagation ➔ Chain Rule Application
  - Note that calculating the chain rule from the right to left means the function looks as follows when evaluating…

$$p \rightarrow p - \alpha \frac{dL(\boldsymbol{x})}{dz_1} \frac{dz_1}{dz_2} \cdots \frac{dz_n}{dp}$$

where $L(\boldsymbol{x})$ is the loss function of the network.

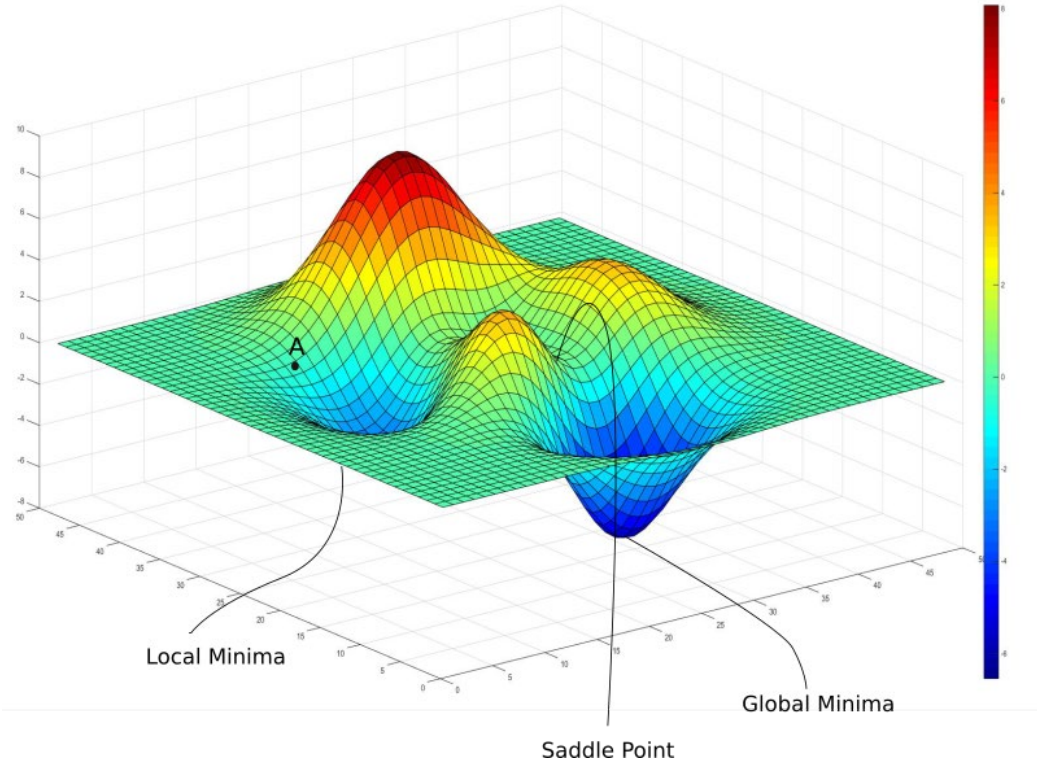


Potential Caveats?

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

Stopping condition: $\nabla_p F(\boldsymbol{x}) = 0$

- Where have we seen this to be the case?
  - Local Minima
  - Global Minima
  - Saddle Points
- Local perturbations solve saddle point issues, but how about reaching local minima? Shouldn't the goal be only the global minima?



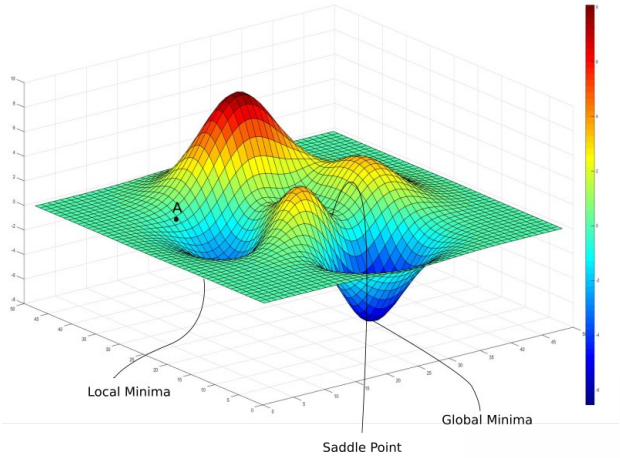
Potential Caveats?

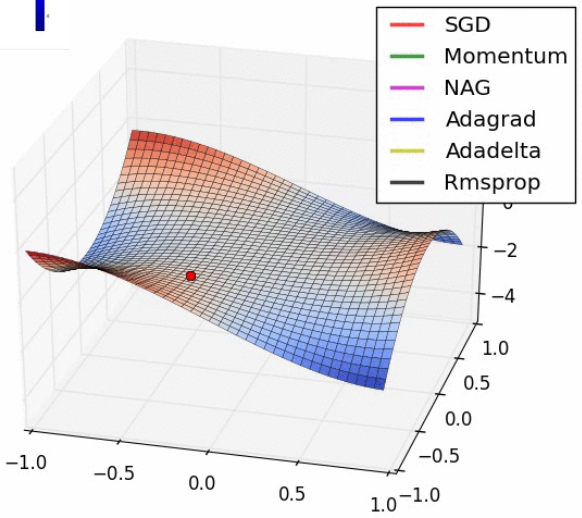
Local perturbations can solve saddle point issues, but how about reaching local minima? Shouldn't the goal be only the global minima?

- The global minima is not necessarily the best for the network!
  - Global Minima ➔ High Likelihood of Overfitting
- It is possible that "best" performing minima is a local minima instead of the global minima!

In Application:

- RMSProp and Adam generally understood to perform the best as optimizers.



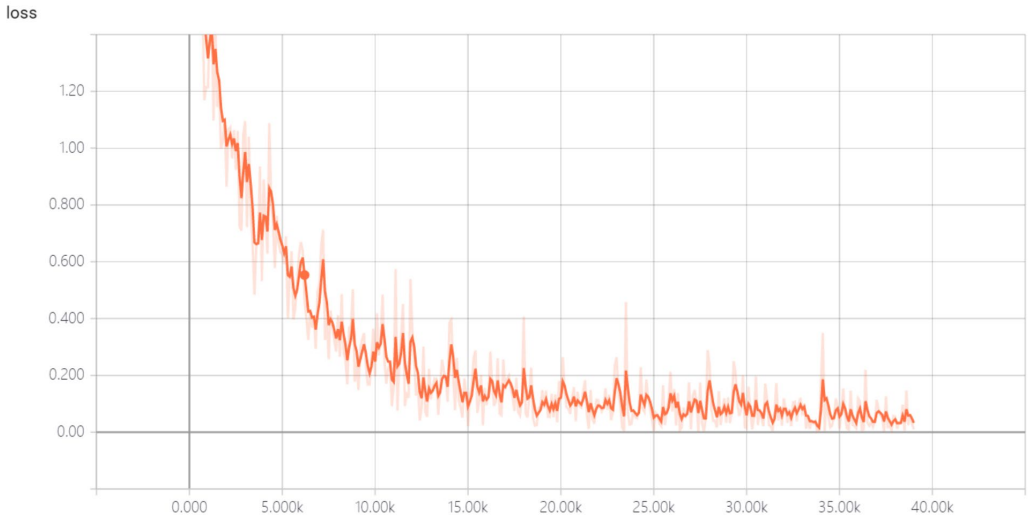


Evaluating Performance

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!



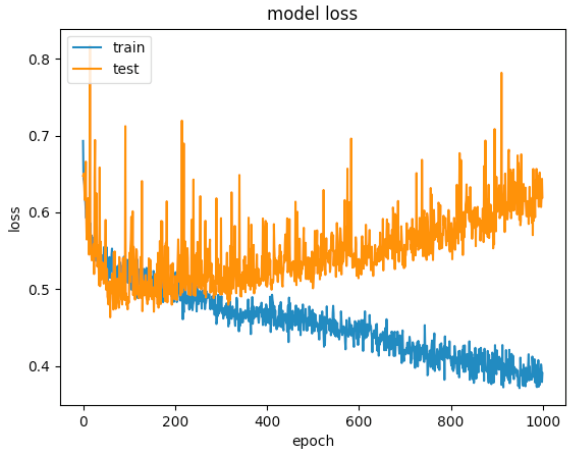
Evaluating Performance

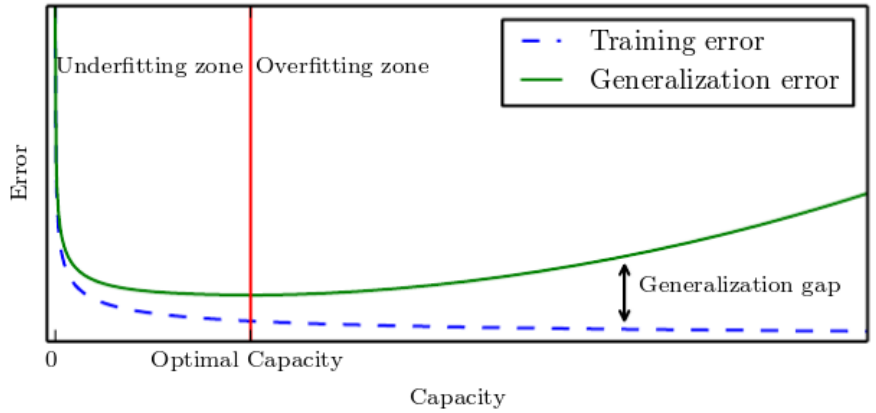
Loss gives information as to whether the model currently training is “learning” anything. It does not mean that what is learned is representative of the dataset.

- 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.





Convolutional Neural Networks

Images and Fully Connected Networks

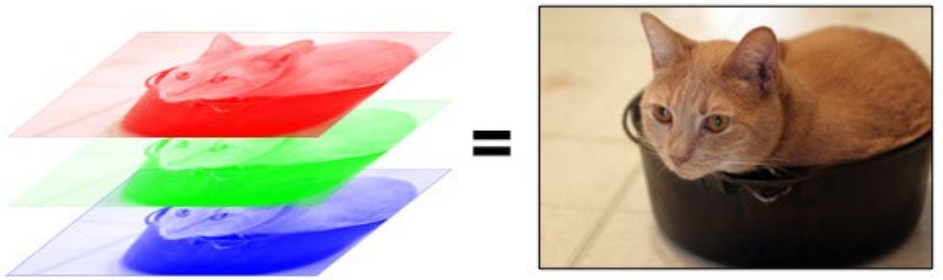
MLPs are great for applications where the number of features is somewhat small. Consider the dimensionality of a 1080p image:

- $1920 \times 1080 \times 3 = \mathbf{6\ 220\ 800}$ **inputs**

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…

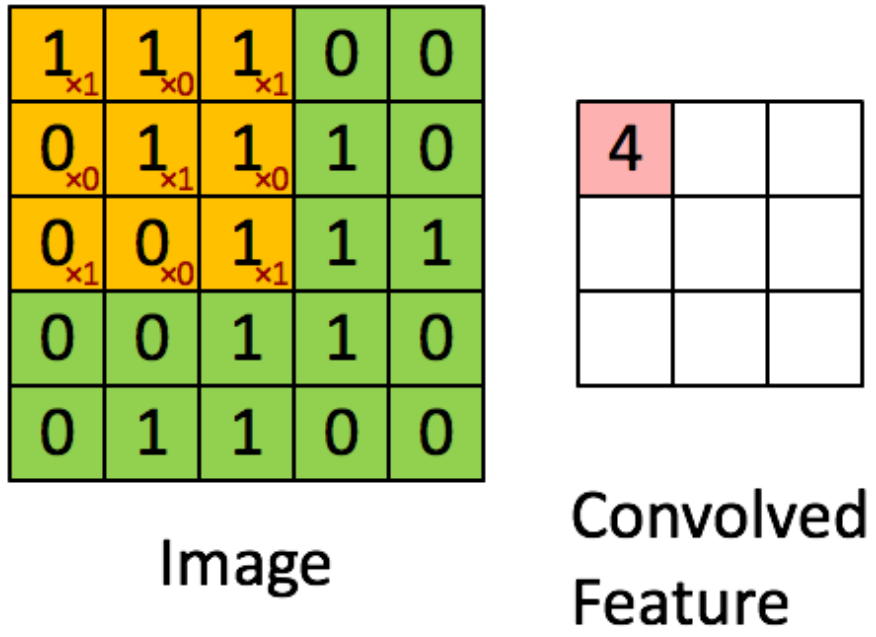


Convolution and Kernels

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.



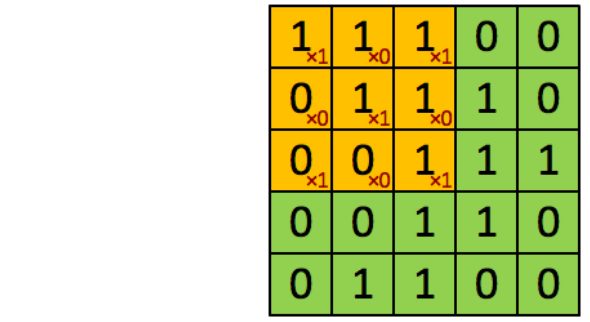
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

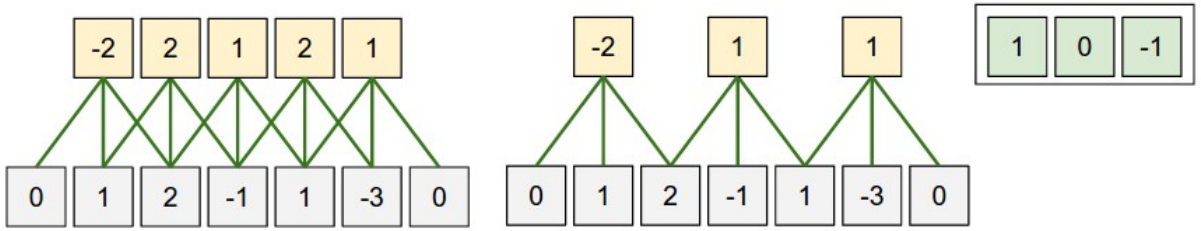
Convolution and Kernels

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!



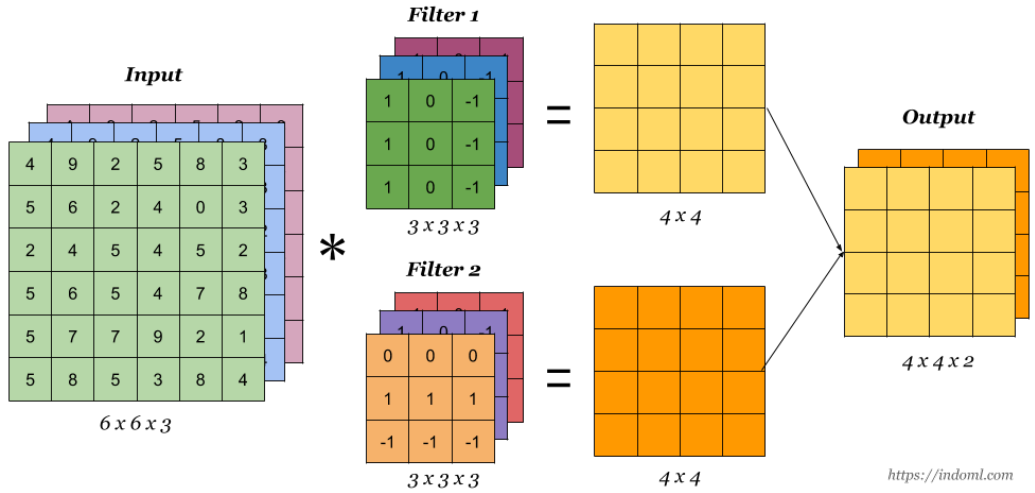


$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

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.



$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

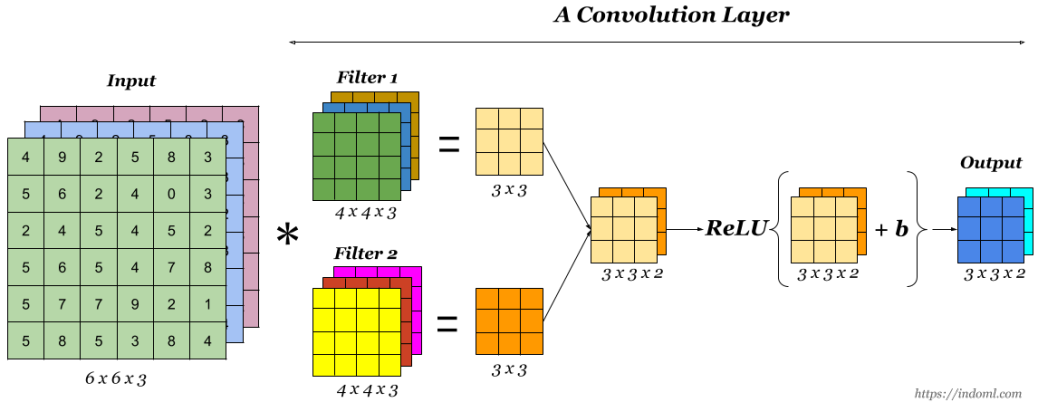
Convolutional Layers

Still missing the non-linearity, however…

- After generating the channels, we may choose to use an activation function on the output and learn some bias, $\boldsymbol{b}$, along with the kernel, K.

Convolutional layer consists of k kernels with size $(N \times N)$ and channel count C_1 intrinsic to the input image $(W_1 \times H_1)$.

- Total of $(k \times N \times N \times C_1)$ parameters vs $\big((W_1 \times H_1 \times C_1) \times (k \times N \times N)\big)$ parameters with a MLP.



$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

Convolutional Layers (Attributes)

Stride (S)

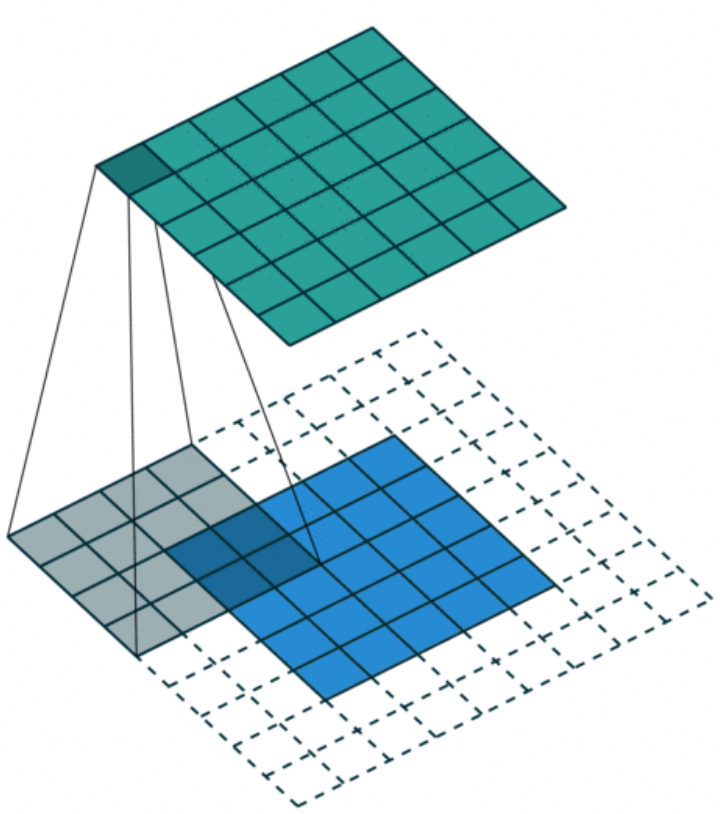
- Amount by which window jumps as it passes through array.

Kernel Size ($N \times N$) and quantity (k)

- 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 (P)

- Provides padding to reproduce certain size at output.



Output Size: $\left(\frac{W_1 - N + 2P}{S} + 1\right) \times \left(\frac{H_1 - N + 2P}{S} + 1\right) \times k$

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

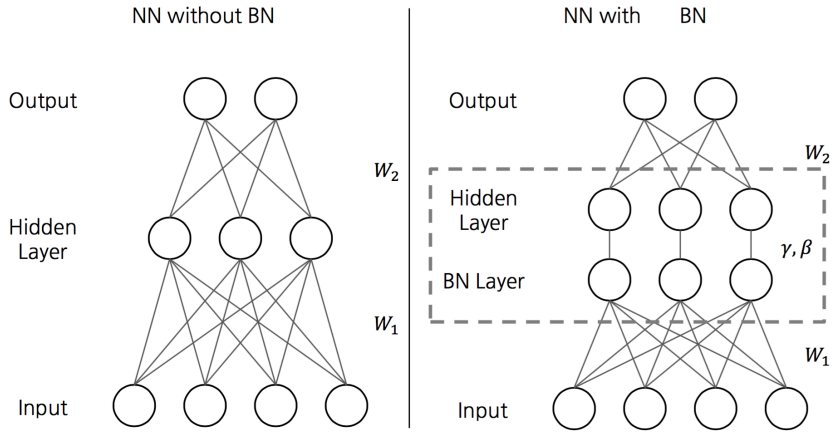
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

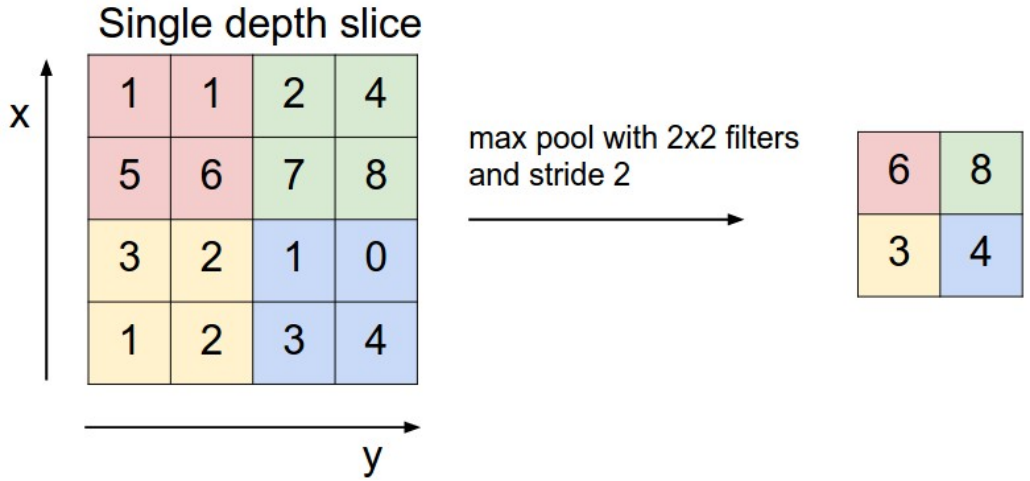


Pooling Layers

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

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

But why?

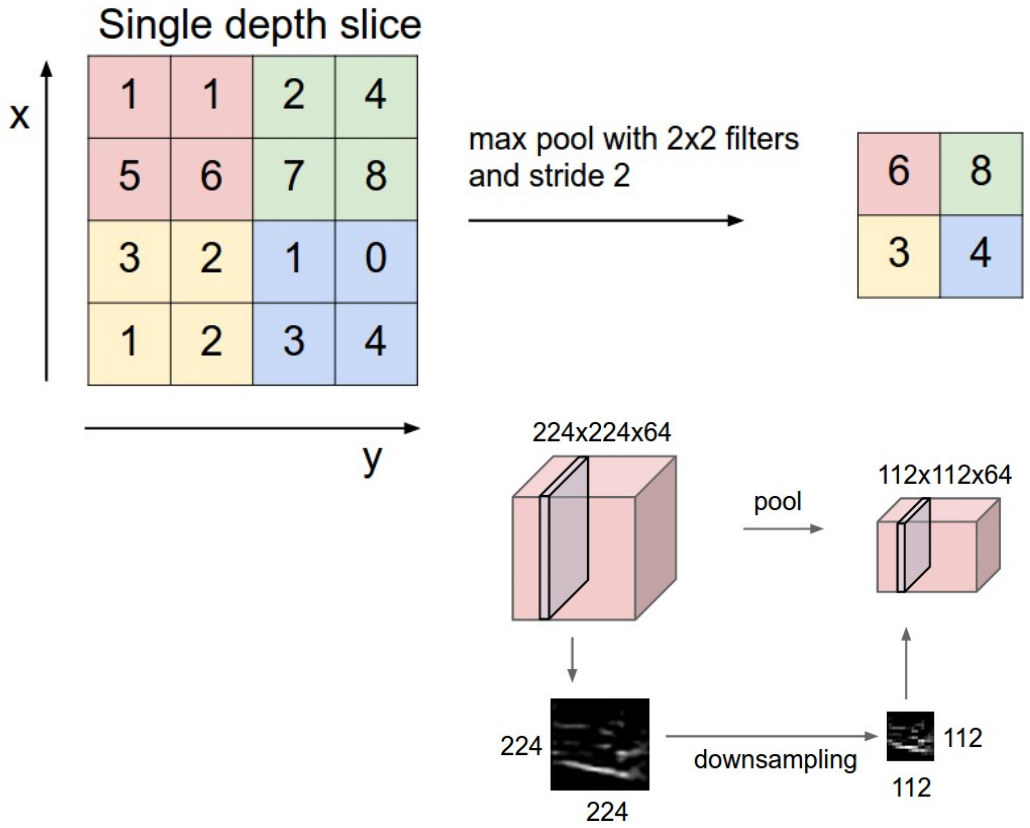


Why Pooling Layers?

But why?

- 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!



Output Size: $\left(\frac{W_1-N}{S}+1\right)\times\left(\frac{H_1-N}{S}+1\right)\times C_1$

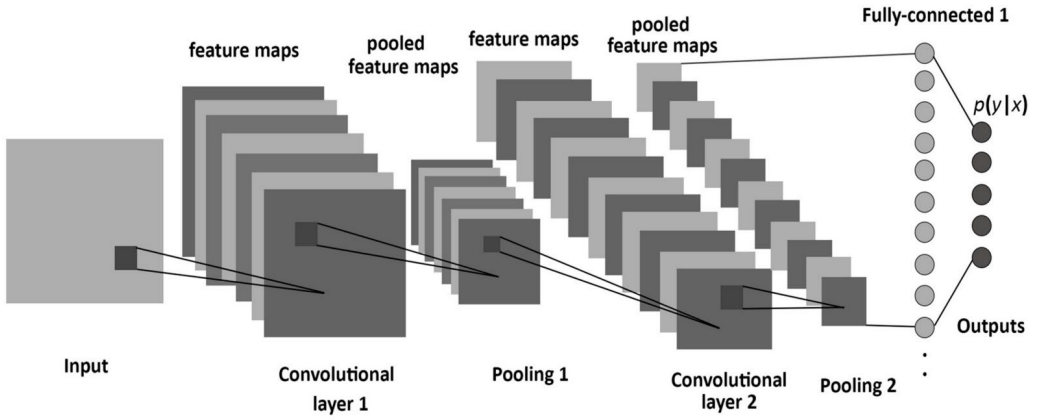
Fully Connected Layers

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

The fully-connected layer is simply a MLP 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.
- Sometimes called the flatten “layer” but it isn’t technically a layer.

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



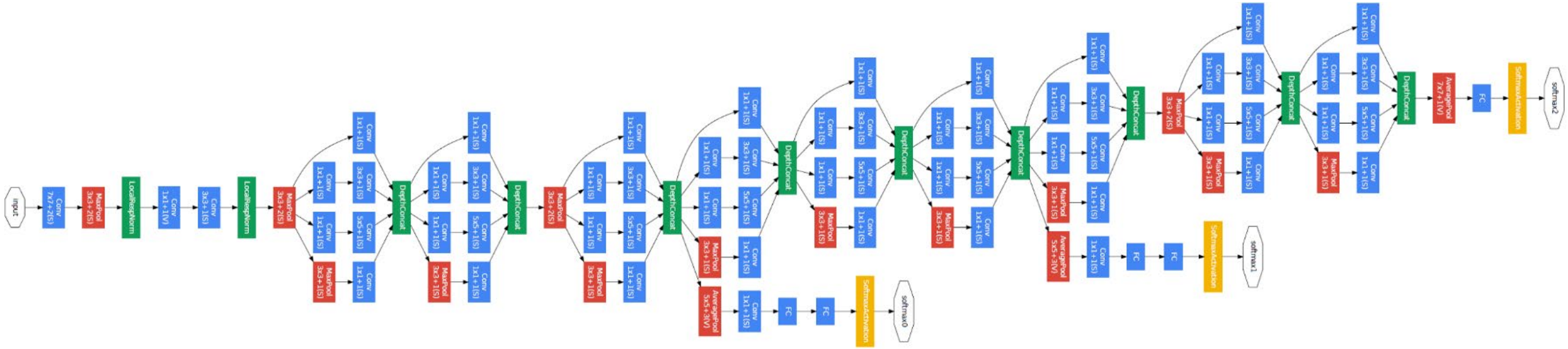
Convolutional Neural Networks

With all of these tools in hand, we are ready to discuss convolutional neural networks! The bottom is an example network, Inception V3.

Concatenates blocks of pooling and convolutional layers with batch norm applied.

Consists of “Inception” modules:

- Tries to learn small and large representations and combines them through concatenation.
- 1x1 convolutions serve to control the depth of the feature maps as they move through the network.



Convolutional Neural Networks

ResNet introduced residual connections for convolutional networks.

Recall that learning in extremely deep networks is hard due to vanishing gradients!

- ResNet addresses this by allowing a copy of an older layer to be summed with the results of a series of convolutional layers.
- Learning with residuals: $\mathrm{y}(\mathrm{x}) = F(x) + x$

Allowed ResNet to train with 1000+ convolutional layers! A true embodiment of the "just go deeper" ideal of CNNs.

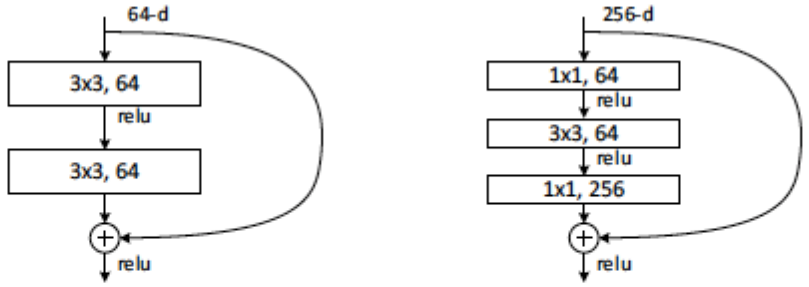


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

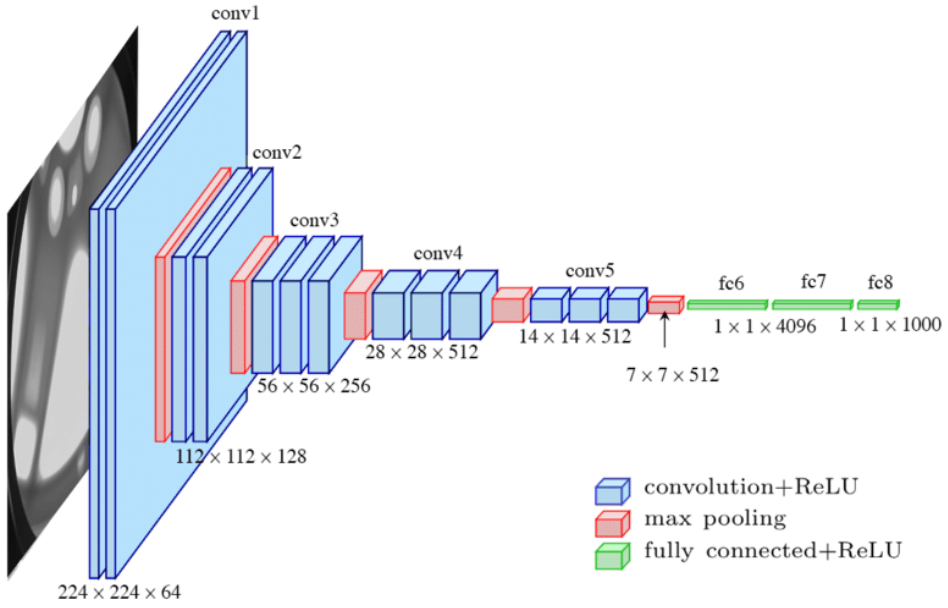
Convolutional Neural Networks

Finally, the VGG family of networks are convolutional networks that consist of nothing by convolutional layers and pooling layers.

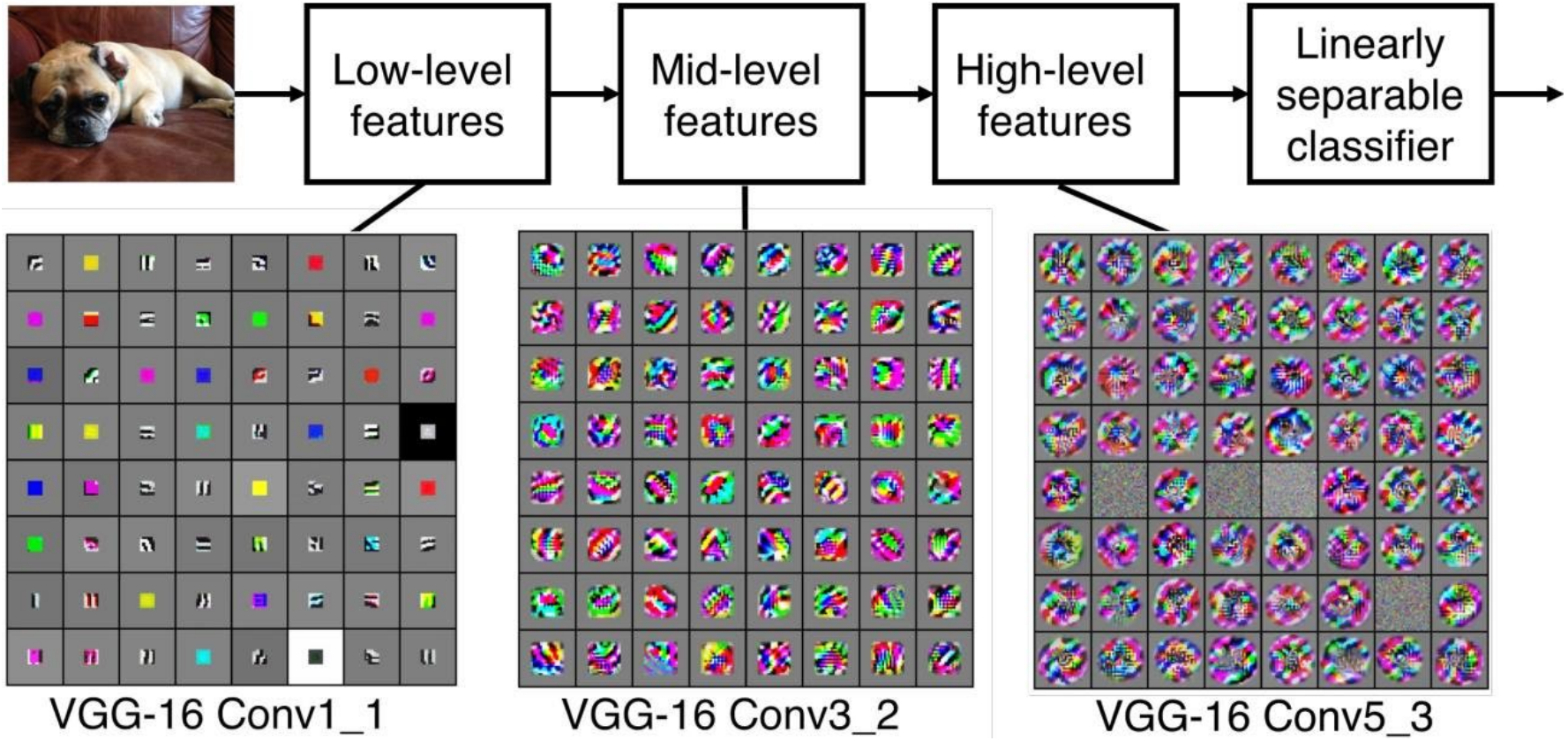
- The classic network of use for quick training and inference.

These make perfect study cases for people interested in convolutional neural networks. Since there is no complicated structure, the viewing of the activation maps tends to be quite straightforward!

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



What Does A CNN Learn?



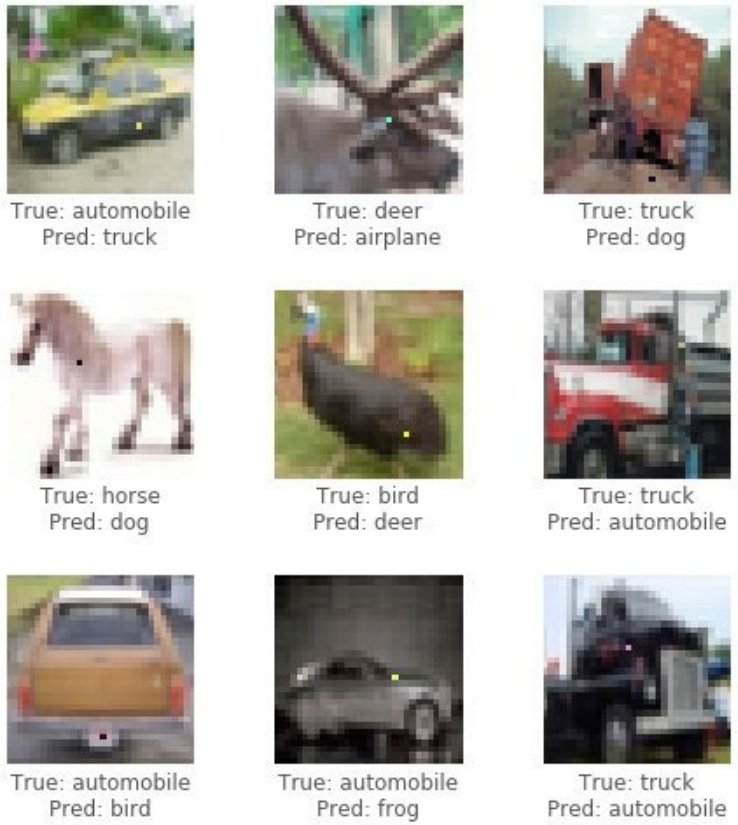
Dataset Considerations

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)

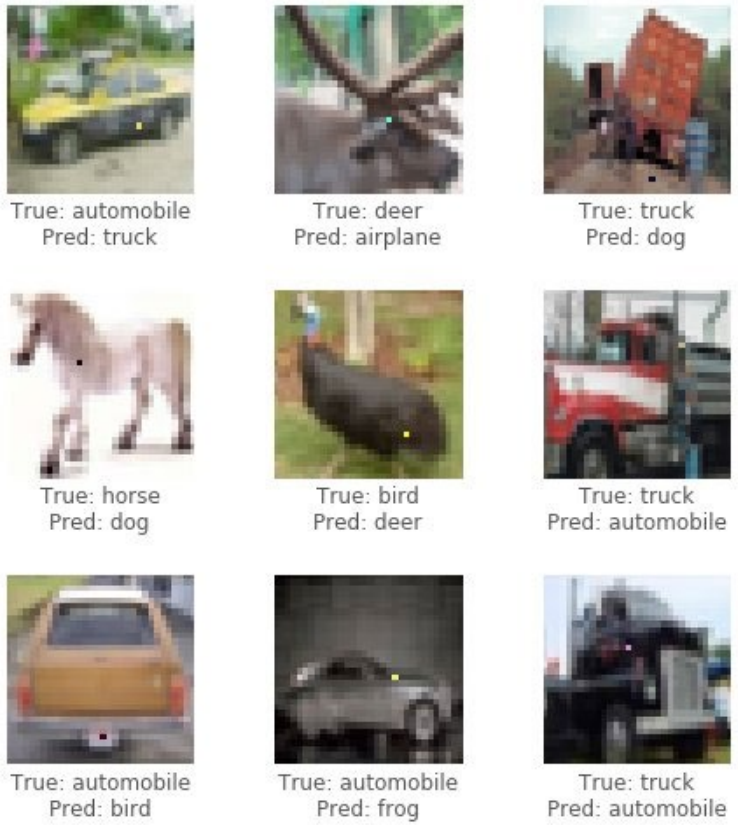
- Symptoms:
  - Good Training / Bad Testing Accuracy
- Remedies:
  - Inspecting & Removing Incorrectly Labeled Samples



Dataset Considerations

Class Imbalances

- Symptoms:
  - Good Overall Accuracy / Terrible Class-wise Accuracy
- Possible Remedies:
  - Weighed Loss Function
  - Data Augmentation
  - Distribution Estimation (Data Creation)



Selected Applications of CNNs

Feature Extraction

Image Classification

Scene Labeling [Segmentation]

- Note that these can use transposed convolutional layers (conv layers with fractional strides)

Image Generation

- Also include transposed convolutions

Image Analysis