Deep Learning (contd.)

CS771: Introduction to Machine Learning

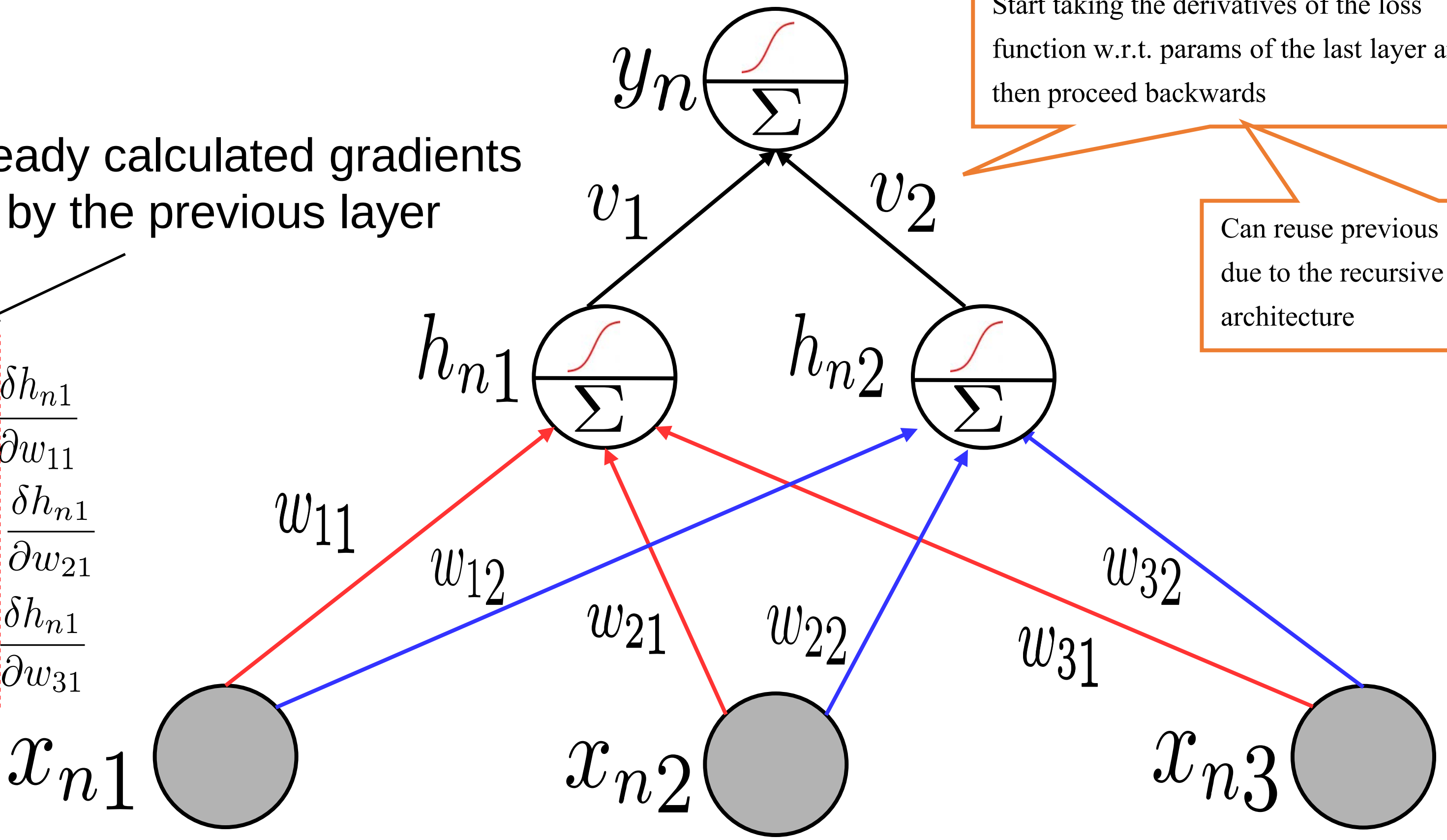
Nisheeth

Backpropagation

- Backpropagation = Gradient descent using chain rule of derivatives
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z}$

Reuse already calculated gradients computed by the previous layer

$$\frac{\delta \mathcal{L}}{\partial w_{11}} = \boxed{\frac{\delta \mathcal{L}}{\partial h_{n1}}} \frac{\delta h_{n1}}{\partial w_{11}}$$
$$\frac{\delta \mathcal{L}}{\partial w_{21}} = \boxed{\frac{\delta \mathcal{L}}{\partial h_{n1}}} \frac{\delta h_{n1}}{\partial w_{21}}$$
$$\frac{\delta \mathcal{L}}{\partial w_{31}} = \boxed{\frac{\delta \mathcal{L}}{\partial h_{n1}}} \frac{\delta h_{n1}}{\partial w_{31}}$$

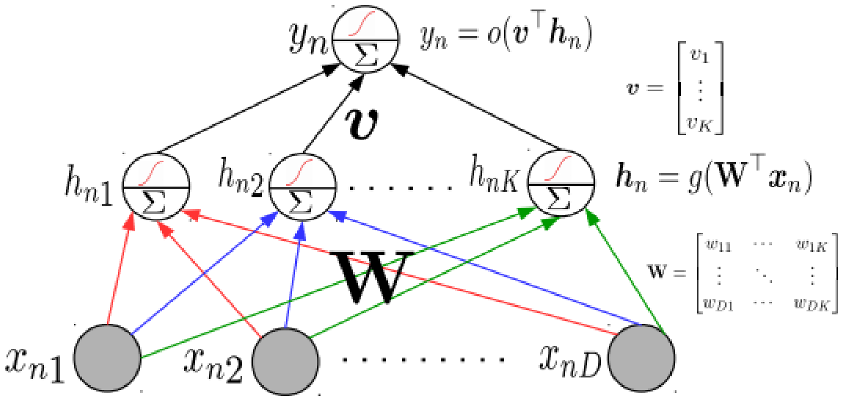


Start taking the derivatives of the loss function w.r.t. params of the last layer and then proceed backwards

Can reuse previous derivative computations due to the recursive nature of the neural net architecture

Backpropagation through an example

Consider a single hidden layer MLP



Assuming regression (o = identity), the loss function for this model

$$\begin{aligned}
\mathcal{L} &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2
\end{aligned}$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

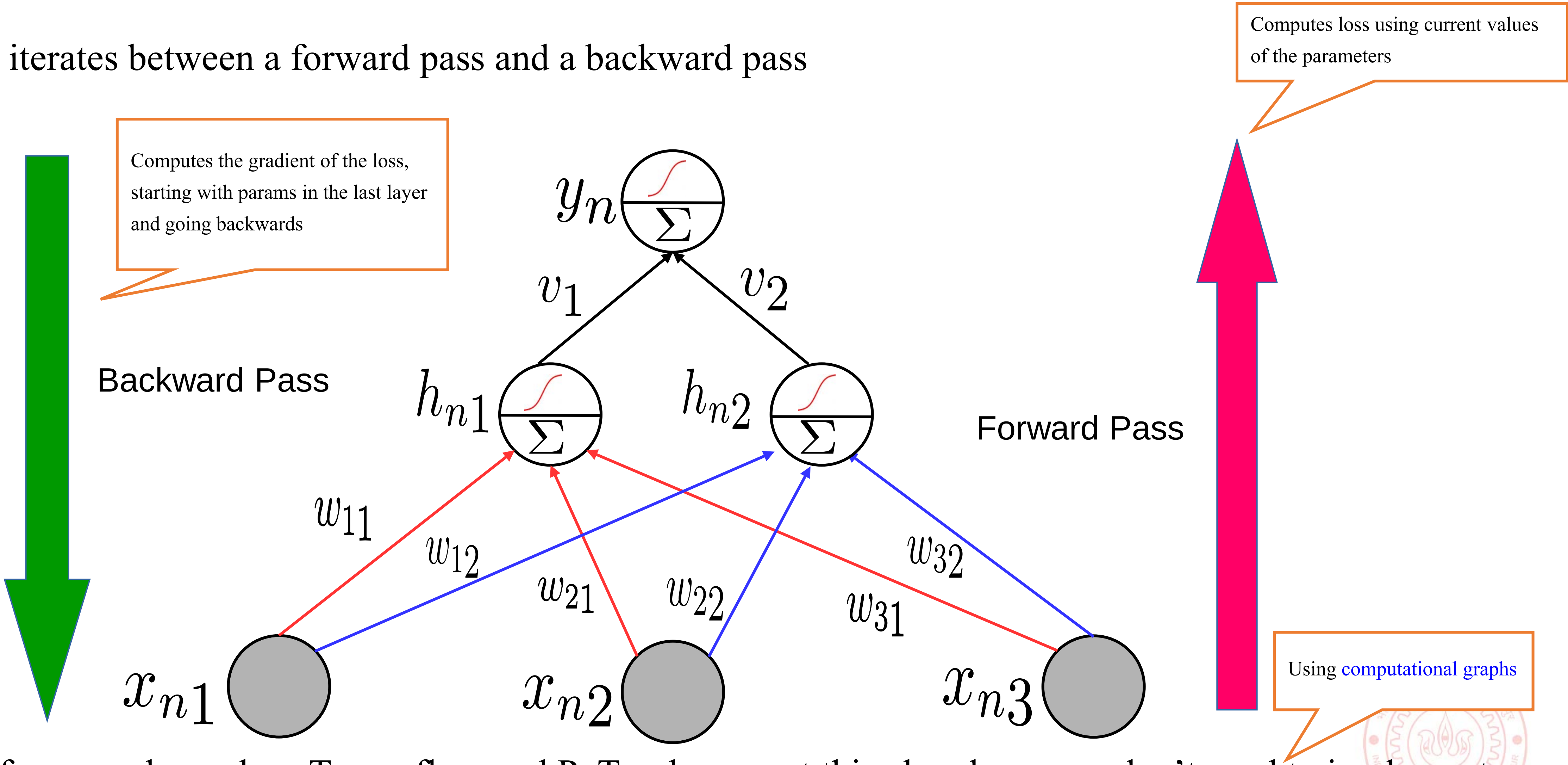
- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{dk}} &= \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}} \\
\frac{\partial \mathcal{L}}{\partial h_{nk}} &= -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))v_k = -\boldsymbol{e}_n v_k \\
\frac{\partial h_{nk}}{\partial w_{dk}} &= g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n) x_{nd} \qquad (\text{note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))
\end{aligned}$$

- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$.
  Backprop updates NN params $\mathbf{W}$, $\boldsymbol{v}$ using grad methods
- Backprop caches many of the calculations for reuse

Backpropagation

- Backprop iterates between a forward pass and a backward pass

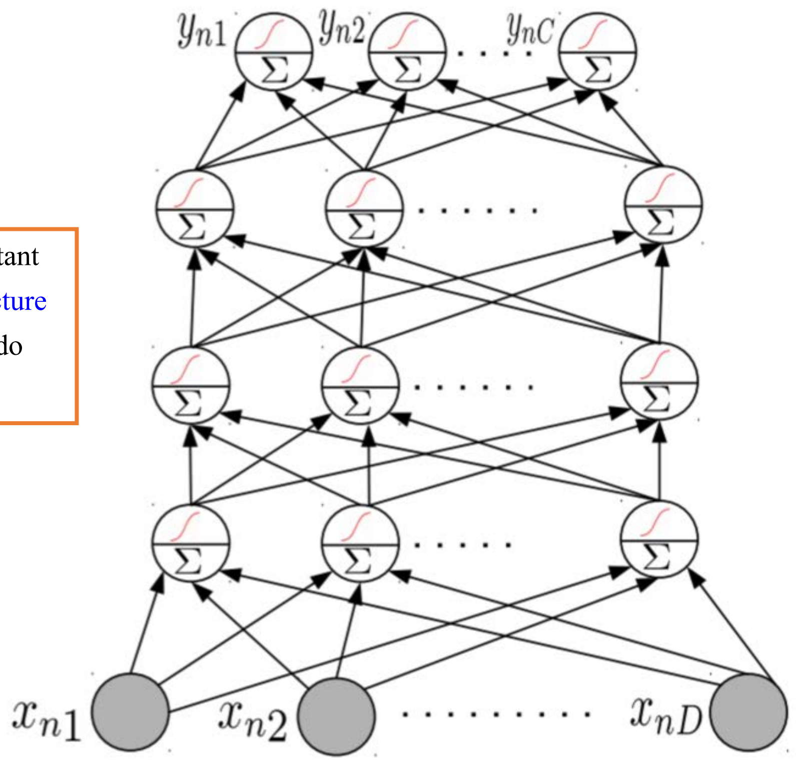


- Software frameworks such as Tensorflow and PyTorch support this already so you don't need to implement it by hand (so no worries of computing derivatives etc)

Neural Nets: Some Aspects

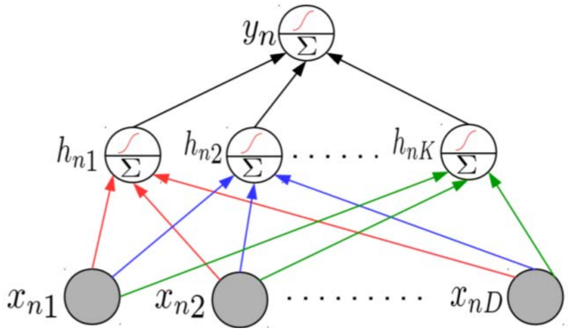
- Much of the magic lies in the hidden layers
- Hidden layers learn and detect good features
- Need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornik's universal function approximator theorem)?
  - Complex networks (several, very wide hidden layers) or simpler networks (few, moderately wide hidden layers)?
  - Aren't deep neural network prone to overfitting (since they contain a huge number of parameters)?

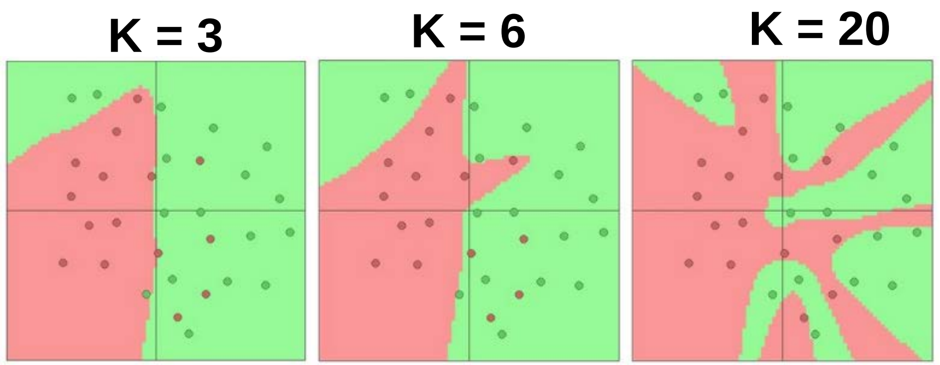
Choosing the right NN architecture is important and a research area in itself. Neural Architecture Search (NAS) is an automated technique to do this



Representational Power of Neural Nets

- Consider a single hidden layer neural net with K hidden nodes





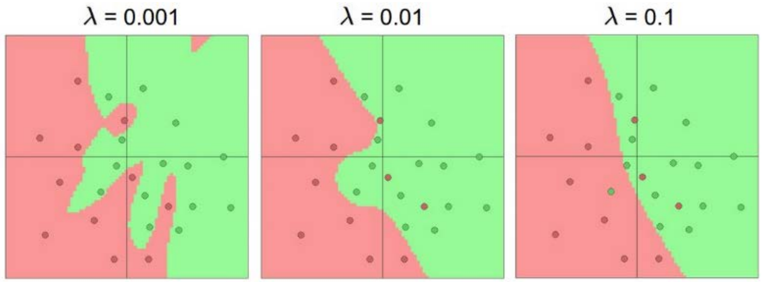
- Recall that each hidden unit "adds" a function to the overall function
- Increasing K (number of hidden units) will result in a more complex function
- Very large K seems to overfit (see above fig). Should we instead prefer small K?
- No! It is better to use large K and regularize well. Reason/justification:
  - Simple NN with small K will have a few local optima, some of which may be bad
  - Complex NN with large K will have many local optimal, all equally good (theoretical results on this)
- We can also use multiple hidden layers (each sufficiently large) and regularize well

Preventing Overfitting in Neural Nets

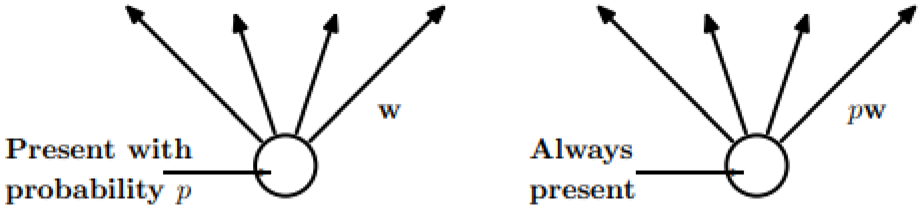
Various other tricks, such as weight sharing across different hidden units of the same layer (used in convolutional neural nets or CNN)

- Neural nets can overfit. Many ways to avoid overfitting, such as
  - Standard regularization on the weights, such as ℓ_2, ℓ_1, etc (ℓ_2 reg. is also called weight decay)

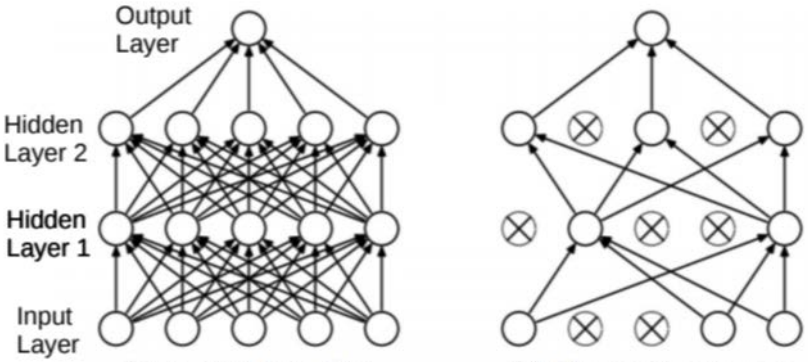
Single Hidden Layer NN with K = 20 hidden units and L2 regularization



  - Early stopping (traditionally used): Stop when validation error starts increasing
  - Dropout: Randomly remove units (with some probability $p \in (0,1)$) during training



(a) At training time (b) At test time

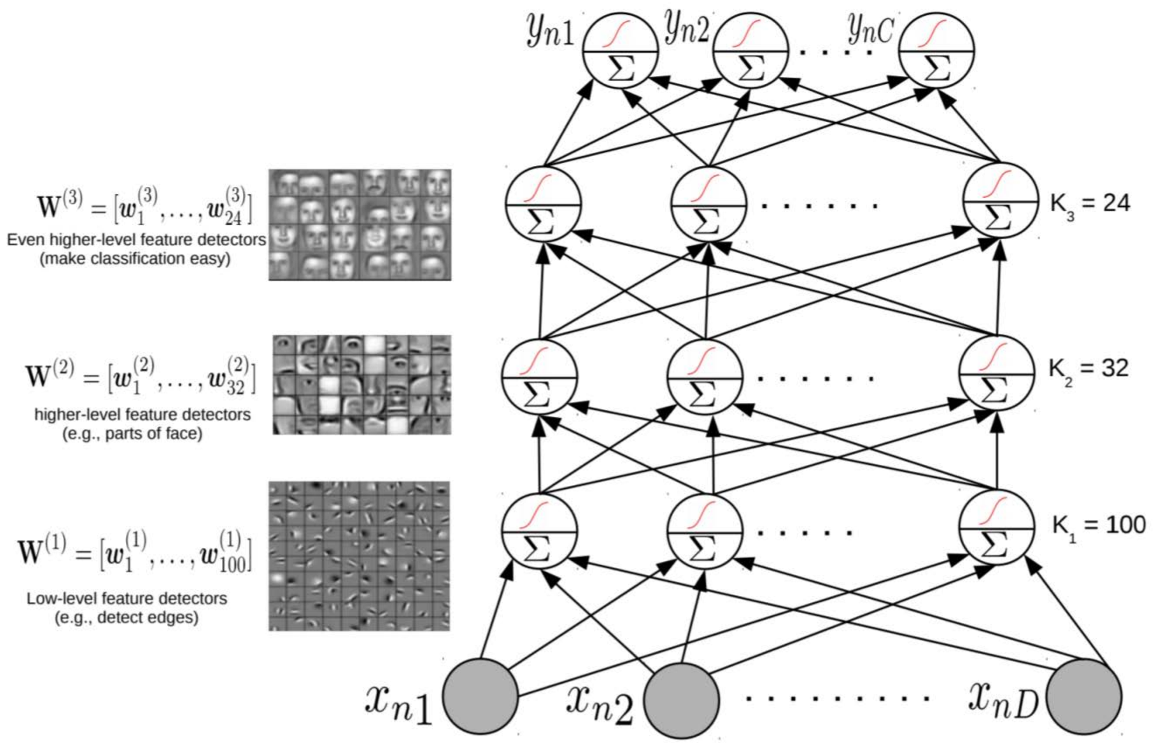


(a) Standard Neural Net (b) After applying dropout.

Fig courtesy: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Srivastava et al, 2014)

Wide or Deep?

- While very wide single hidden layer can approx. any function, often we prefer many, less wide, hidden layers



- Higher layers help learn more directly useful/interpretable features (also useful for compressing data using a small number of features)

Conv nets basics

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image patch

1	0	1
0	1	0
1	0	1

Filter

$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

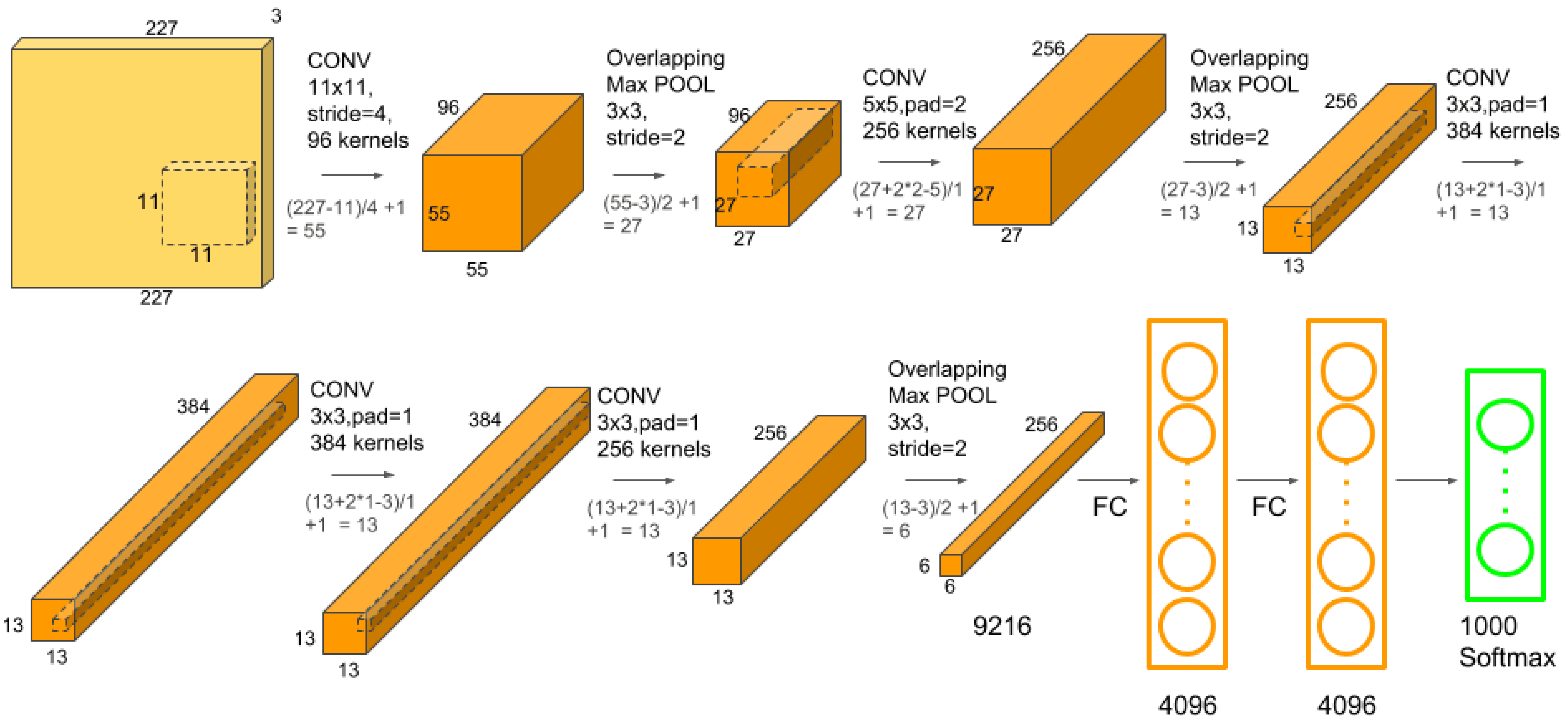
Convolution

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

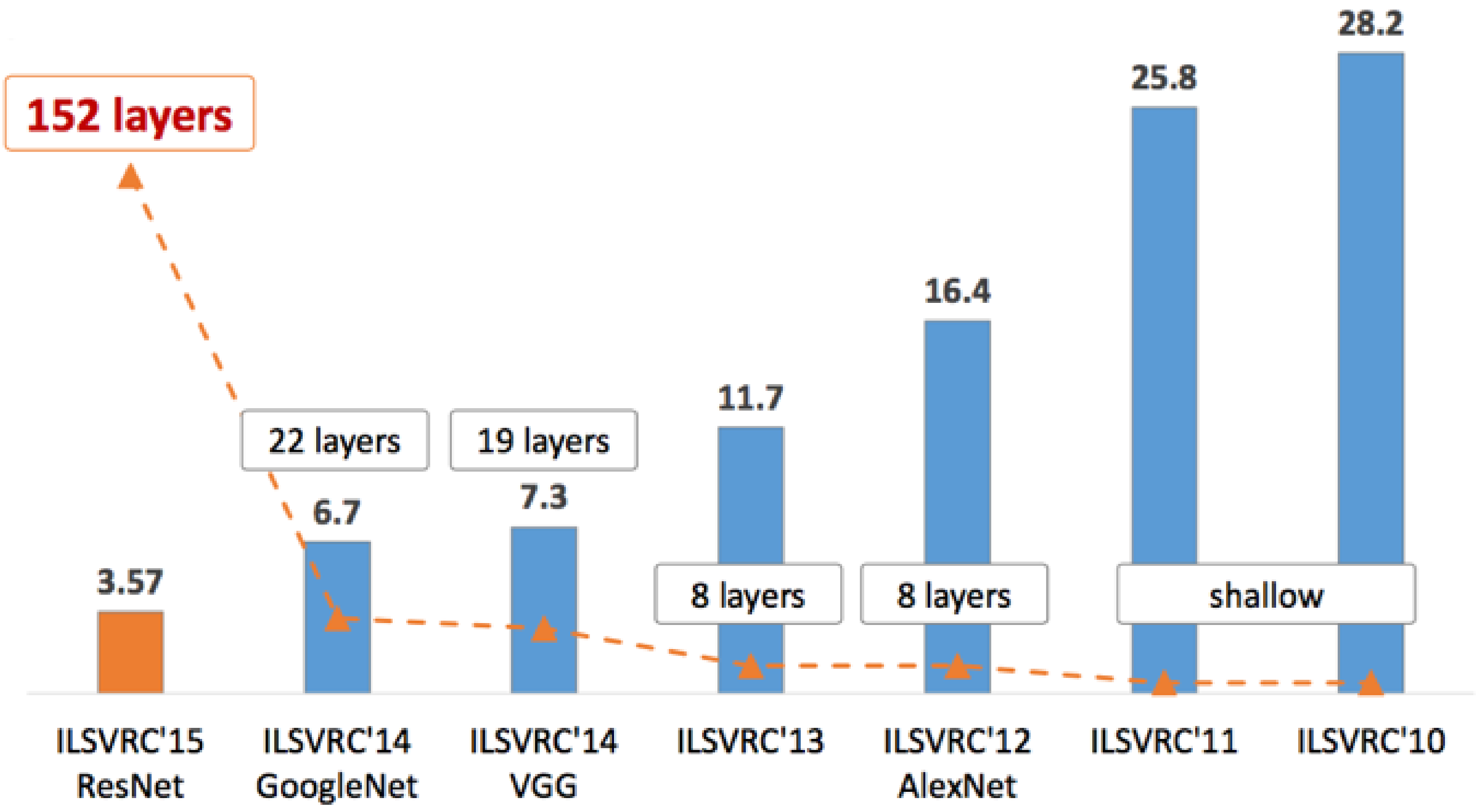
Discriminability from diverse filtering

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

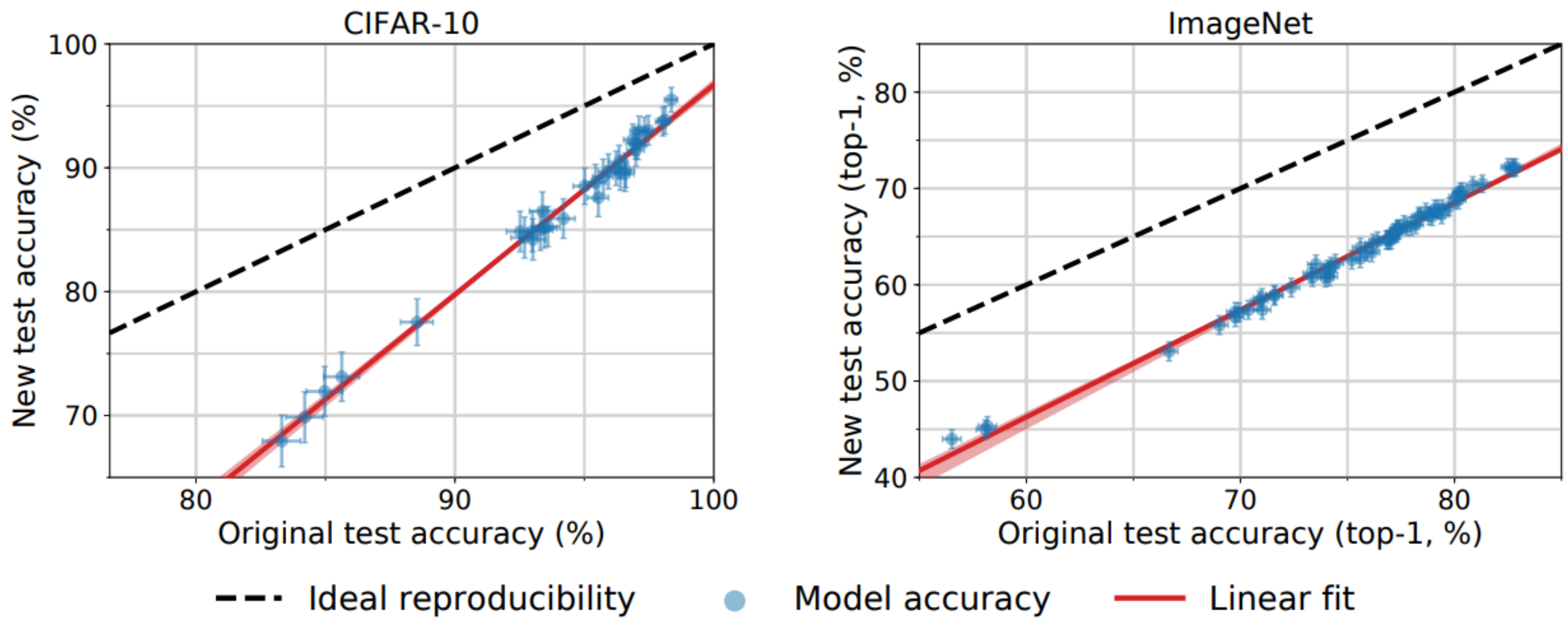
A typical convnet: AlexNet



Superhuman object recognition in images

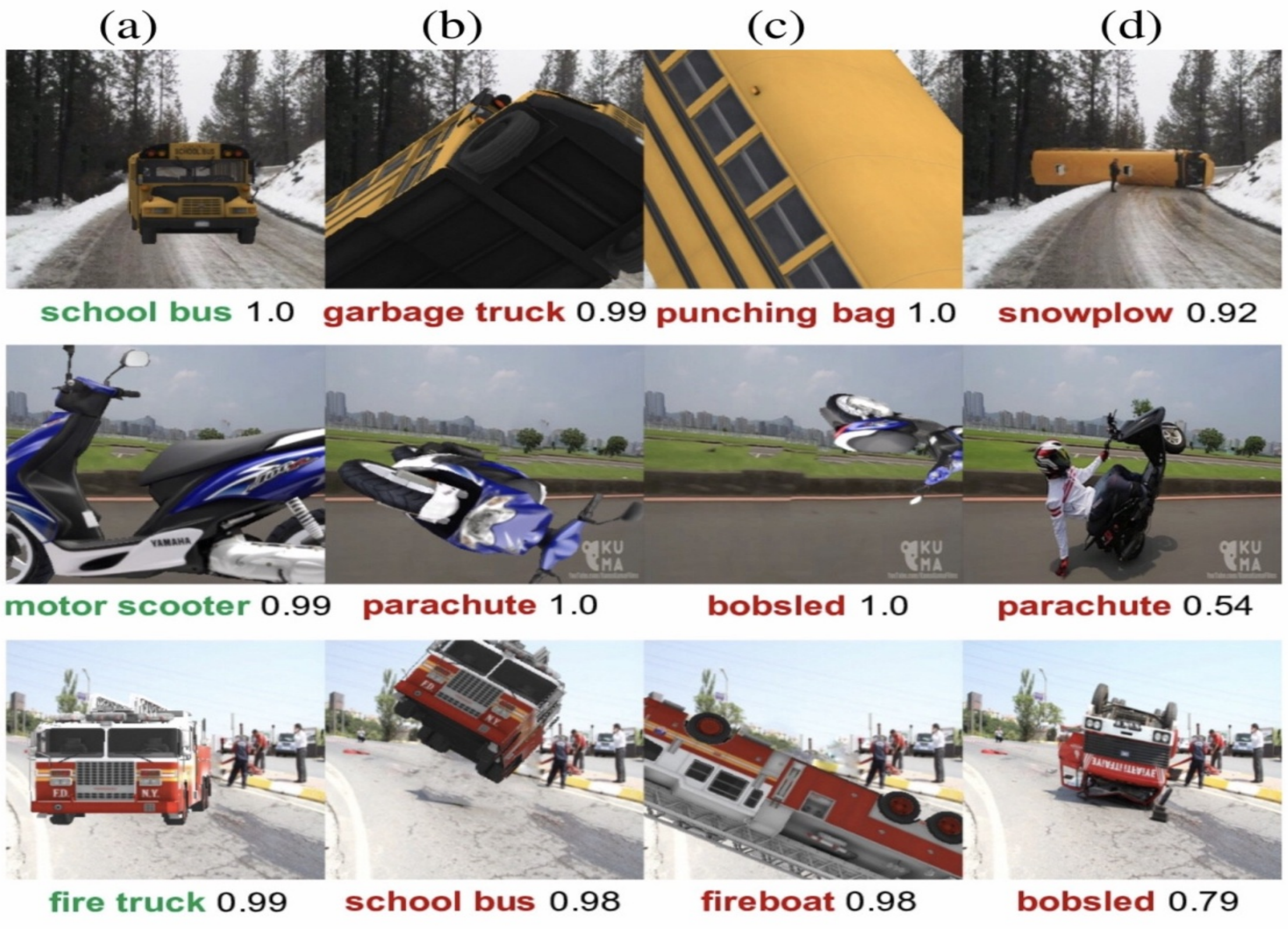


But deep networks are fickle



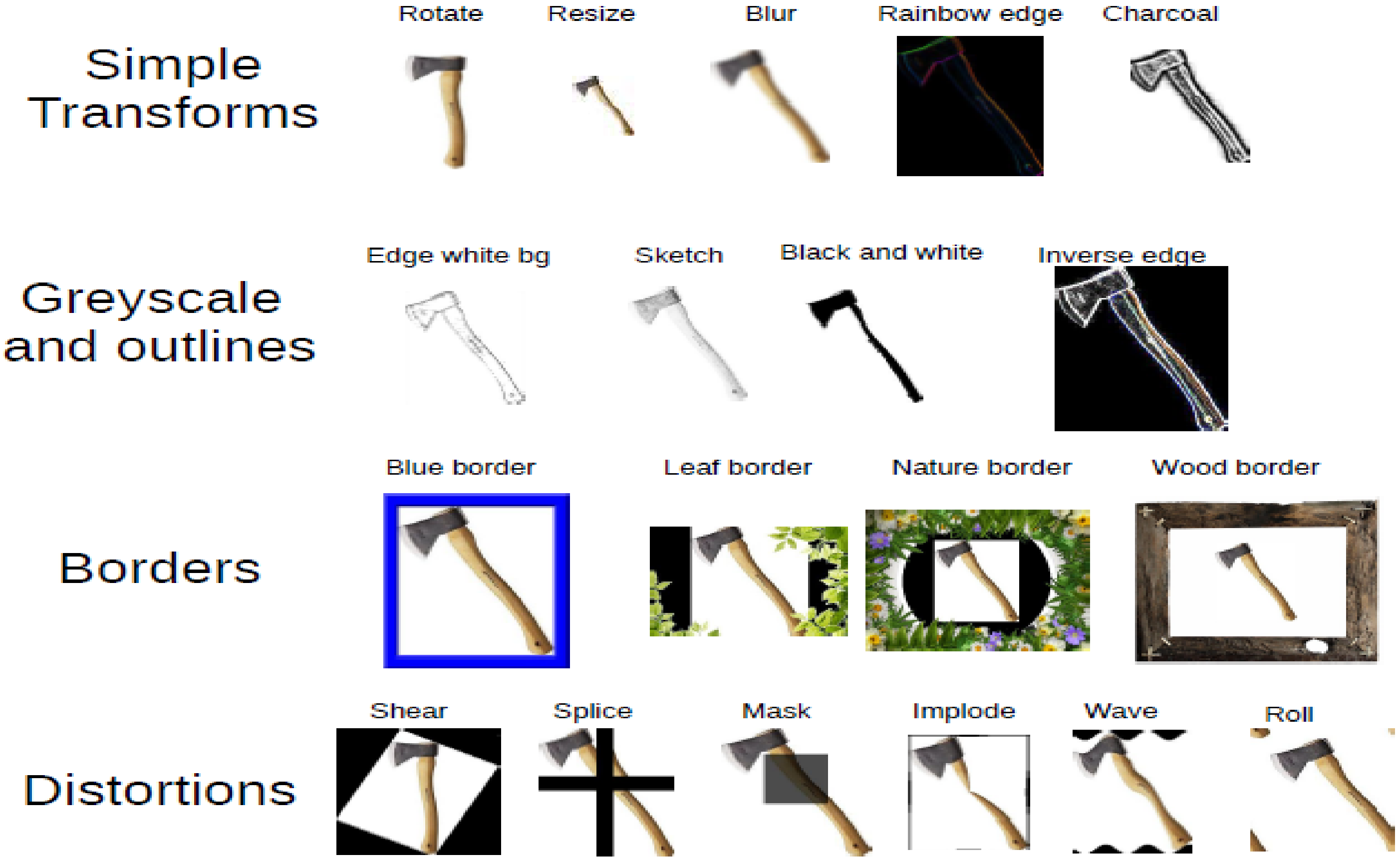
Do imagenet classifiers generalize to imagenet? ([link]())

… and brittle



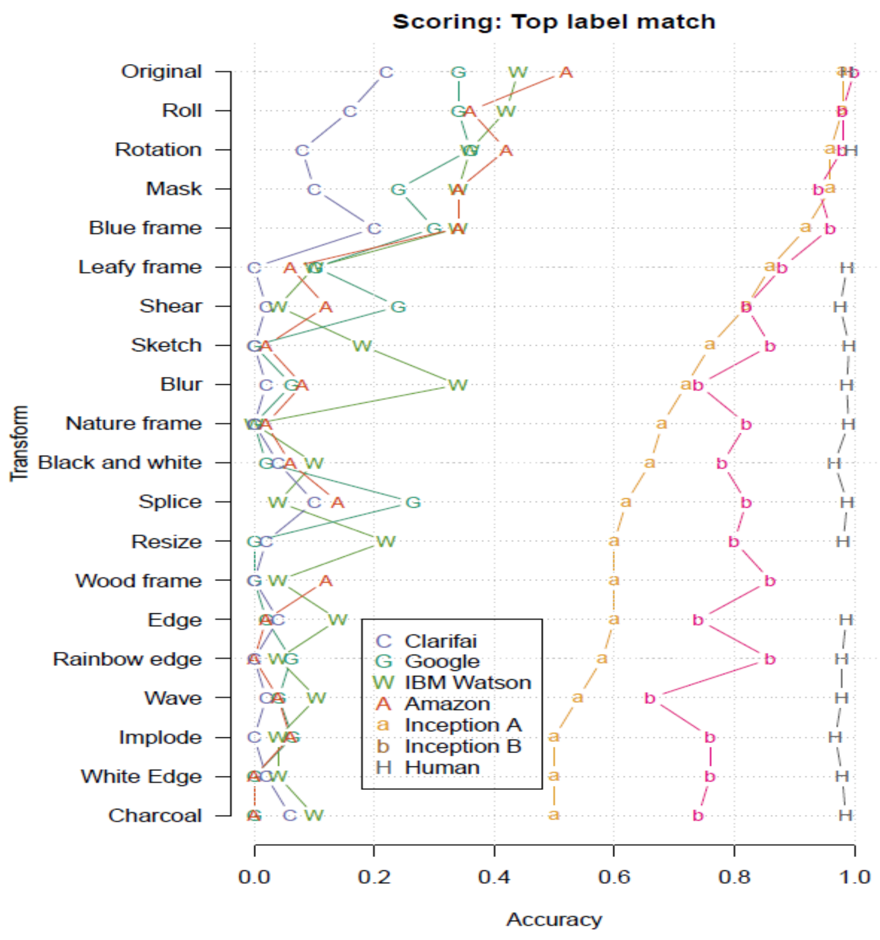
https://arxiv.org/pdf/1811.11553.pdf

… and stupid


Simple Transforms
Rotate
Resize
Blur
Rainbow edge
Charcoal
Greyscale and outlines
Edge white bg
Sketch
Black and white
Inverse edge
Borders
Blue border
Leaf border
Nature border
Wood border
Distortions
Shear
Splice
Mask
Implode
Wave
Roll

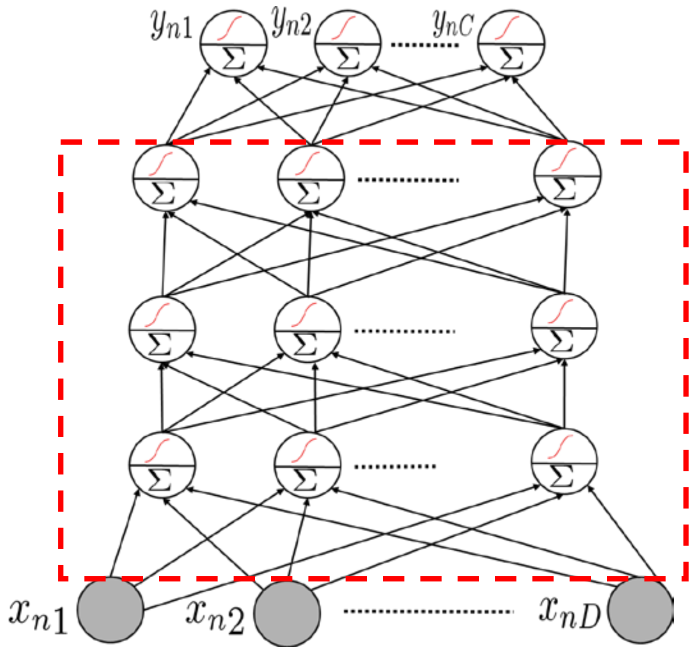
… beyond belief

- Untransformed images are classified with 98% and 100% accuracy
- Transformed image accuracy drops enormously
- Human performance is unaffected
- Humans know when they are going to have trouble



Using a Pre-trained Network

- A deep NN already trained in some "generic" data can be useful for other tasks, e.g.,
  - Feature extraction: Use a pre-trained net, remove the output layer, and use the rest of the network as a feature extractor for a related dataset



This part of a pre-trained net can be used as a feature extractor on some new task

Many packages, like Tensorflow and PyTorch provide such pre-trained module ready to be used

Sometimes also known as "transfer learning" in the context of neural nets

  - Fine-tuning: Use a pre-trained net, use its weights as initialization to train a deep net for a new but related task (useful when we don't have much training data for the new task)

Deep Neural Nets: Some Comments

- Highly effective in learning good feature rep. from data in an "end-to-end" manner

- The objective functions of these models are highly non-convex
  - But fast and robust non-convex opt algos exist for learning such deep networks

- Training these models is computationally very expensive
  - But GPUs can help to speed up many of the computations

- Also useful for unsupervised learning problems (will see some examples)
  - Autoencoders for dimensionality reduction
  - Deep generative models for generating data and (unsupervisedly) learning features – examples include generative adversarial networks (GAN) and variational auto-encoders (VAE)