Research and Innovation Center of Excellence

MSc on Intelligent Critical Infrastructure Systems

Machine Learning
Lecture 6

Christos Kyrkou
Research Lecturer
KIOS Research and Innovation Center of Excellence
University of Cyprus

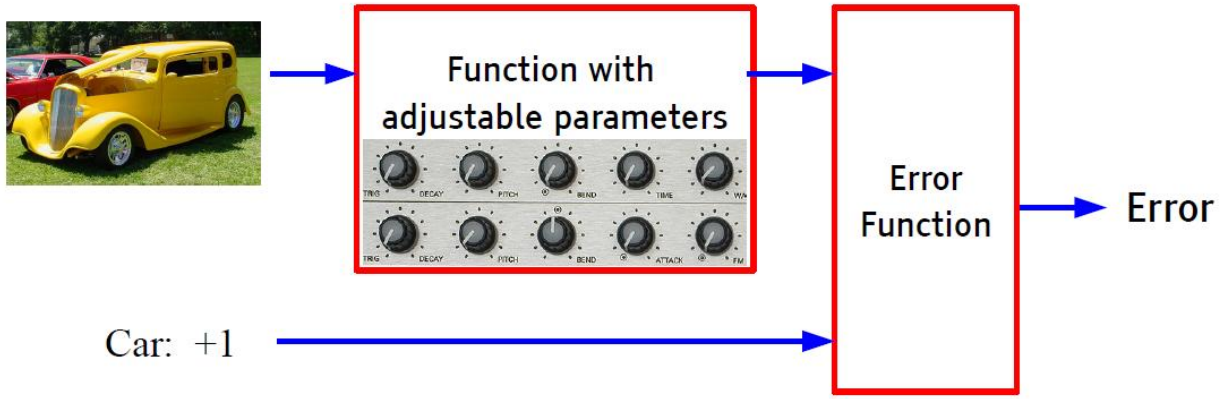
Imperial College London

University of Cyprus

funded by:

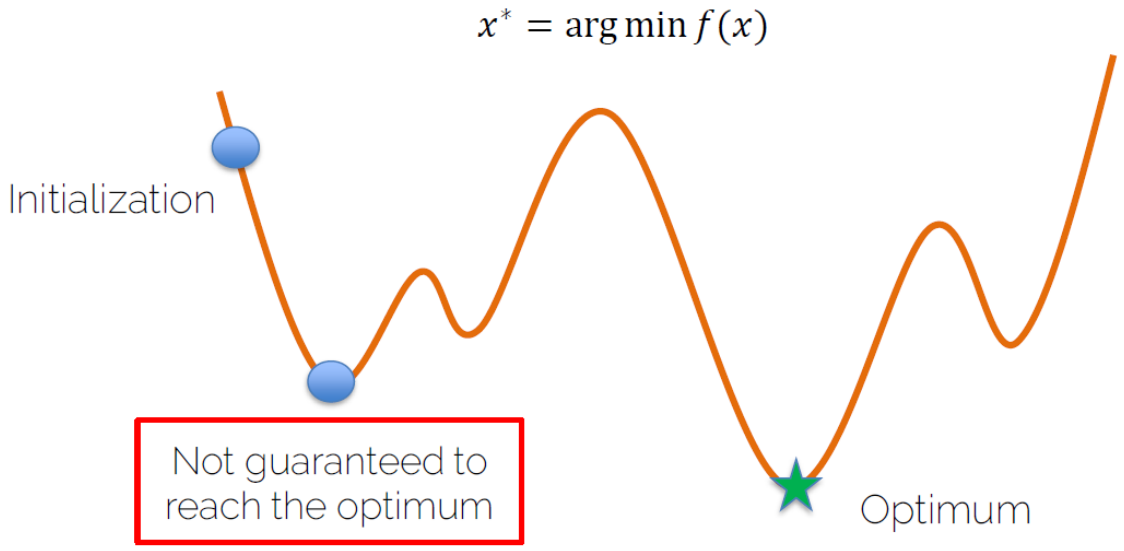
Training Neural Network

- Given a dataset with ground truth training pairs $[x_i; y_i]$ optimize a deep neural network with **non-linear activations**, to produce the behaviour.
- Find optimal **weights** W using **stochastic gradient decent**, such that the **loss function** is minimized
- Compute **gradients** with **backpropagation**



Initialization is Extremely Important

- The initialization step can be critical to the model's ultimate performance, and it requires the right method.
- How to initialize the weights?



All Weights Zero

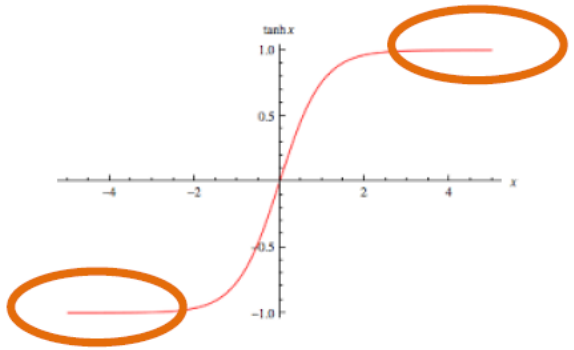
- If all the weights are initialized with 0, the derivative with respect to loss function is the same for every w in W, thus all weights have the same value in subsequent iterations.
- The hidden units are all doing to compute the same function, gradients are going to be the same
  - No symmetry breaking
- Assigning random values to weights is better than just 0 assignment.

Neural Network Initialization

- Large Values?
  - If weights are initialized with very high values the term $dot(W, X) + b$ becomes significantly higher
    - if an activation function like $sigmoid()$ is applied, the function maps its value near to 1 where the slope of gradient changes slowly and learning takes a lot of time. → **Vanishing Gradient**
    - Can also lead to large output values, weights can become so large and cause overflow. → **Exploding Gradient**
- Small Values?
  - If weights are initialized with very small values, some activation functions might map to zero, which is the same case as above. → **Vanishing Gradient**
  - If model is large is slower to converge

Neural Network Initialization

- Big Random Numbers
  - Gaussian with $zero$ mean and standard deviation 1
    - $N(\mu, \sigma) \sim \mu = 0, \sigma = 1$
- Impact on Tanh Activation:
  - Output saturated to -1 and 1. Gradient of the activation function becomes close to 0.

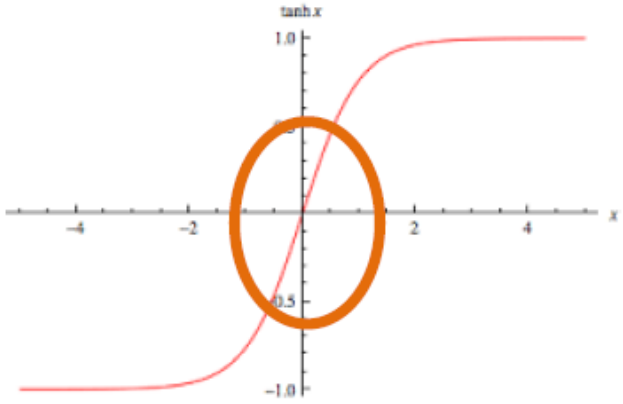


$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial s} \cdot \frac{\partial s}{\partial w_i} \approx 0$$

Vanishing gradient, caused by saturated activation function.

Neural Network Initialization

- Small Random Numbers
  - Gaussian with $zero$ mean and standard deviation 0.01
    - $N(\mu, \sigma) \sim \mu = 0, \sigma = 0.01$
  - Even activation function's gradient is ok, we still have vanishing gradient problem
  - Small outputs of layer l (input of layer $l+1$) cause small gradient w.r.t. to the weights of layer $l+1$



Small w_i^l cause small output for layer l:

$$f_l\left(\sum_i w_i^l x_i^l + b^l\right) \approx 0$$

$$\frac{\partial L}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot \frac{\partial f_{l+1}}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot x_i^{l+1} \approx 0$$

Vanishing gradient, caused by small output

Xavier Initialization

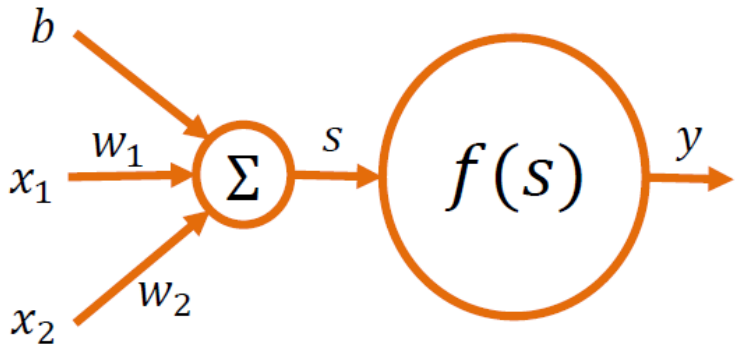
- To prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:
  - The variance of the activations should stay the same across every layer.
    - i.e., variance of the output is the same as the input
  - Calculate with respect to the number of neurons in the previous layer.
  - Xavier Initialization:
    - $N(\mu, \sigma) \sim \mu = 0, \sigma = \sqrt{Var}$
    - $Var = \frac{2}{n}$, where n is number of input neurons
  - Use ReLU and Xavier initialization

More on activation functions

- Activation functions are used to make to make the computation non-linear
- They have a profound effect on the neural network training
- Choice of activation function affects the propagation of gradient similarly to the weight initialization

Sigmoid Output not zero-centered

- We want to compute the gradient w.r.t. the weights



Assume we have all positive data:

$\boldsymbol{x} = (x_1, x_2)^T > 0$

either positive or negative

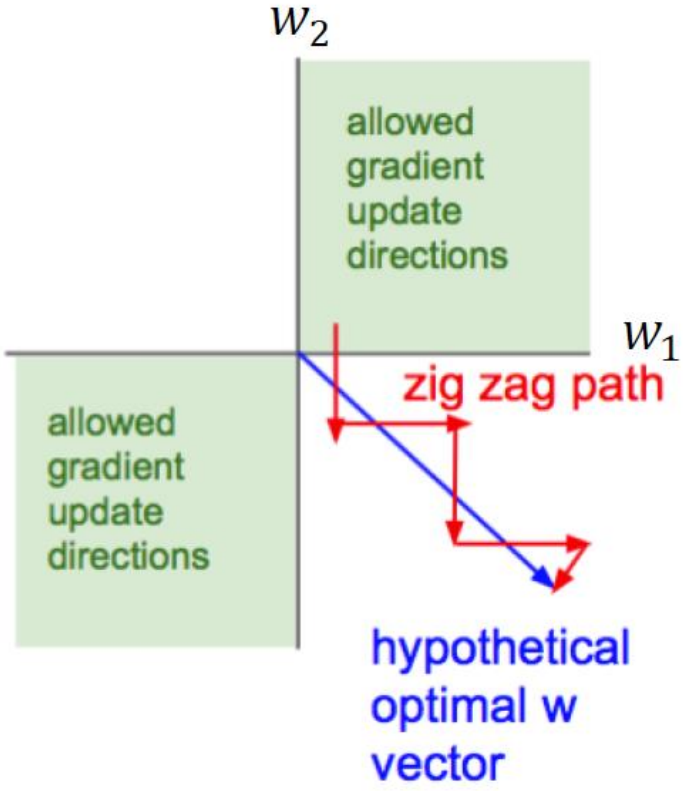
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial w_1} \qquad x_1 > 0$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial w_2} \qquad x_2 > 0$$

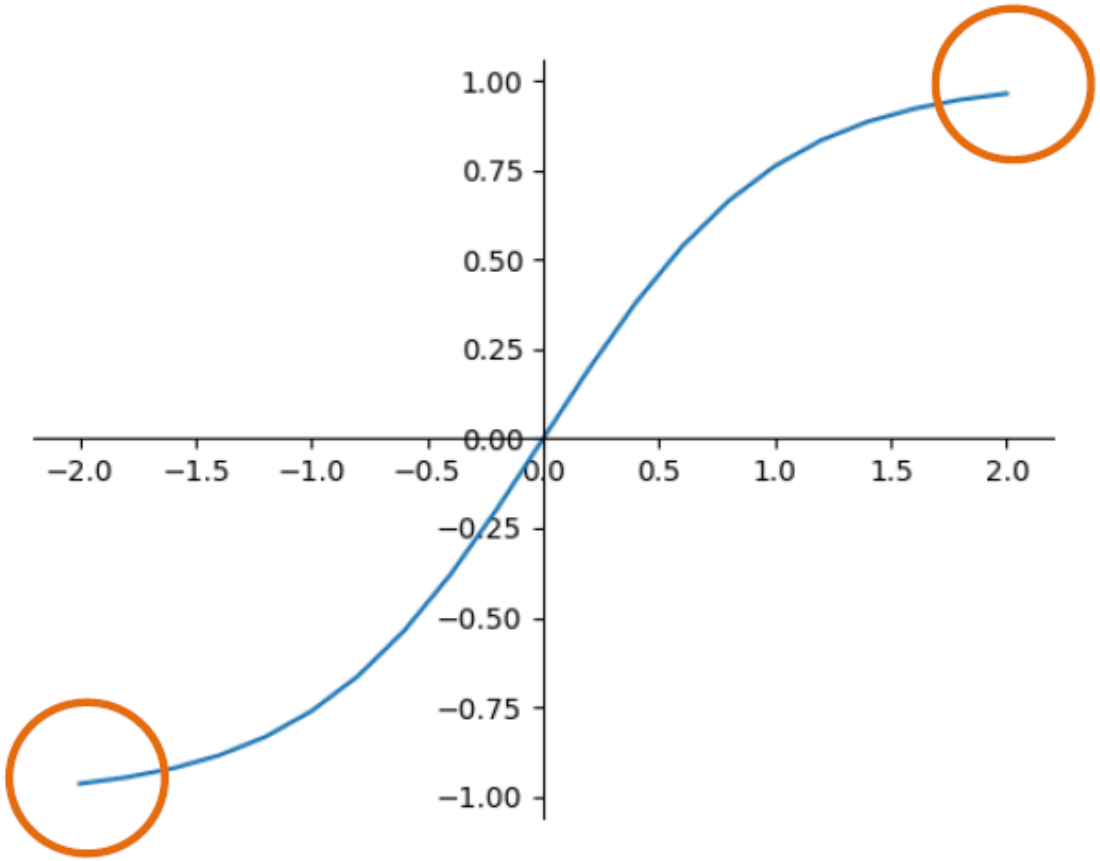
- It is going to be either positive or negative for all weights' update
  - Remember sigmoid goes between 0-1

Sigmoid Output not zero-centered

- w_1, w_2 can only be increased or decreased at the same time, which is not good for update
- That is also why you need zero-centered data



TanH Activation



✗ Still saturates

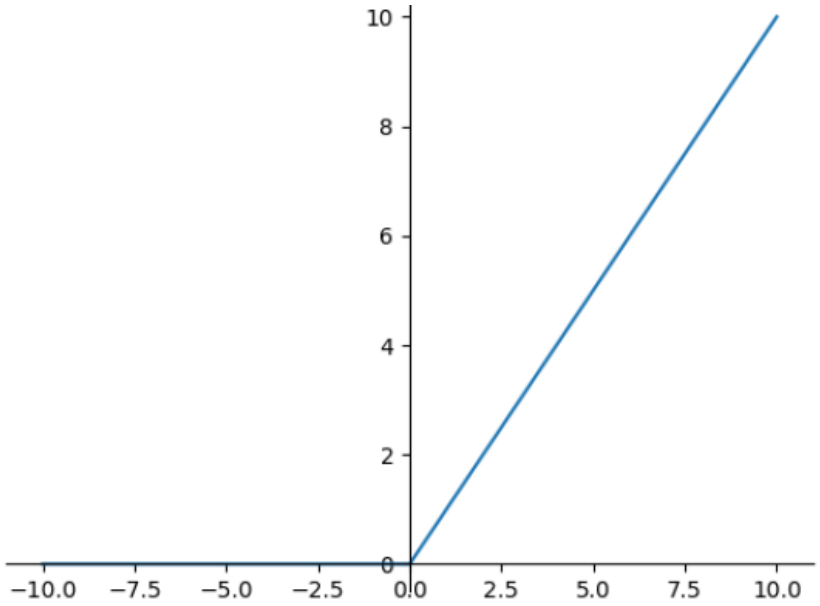
✓ Zero-centered

Rectified Linear Units (ReLU)

- $\sigma(x) = \max(0, x)$

✘ Dead ReLU

What happens if a ReLU outputs zero?



Large and consistent gradients ✓

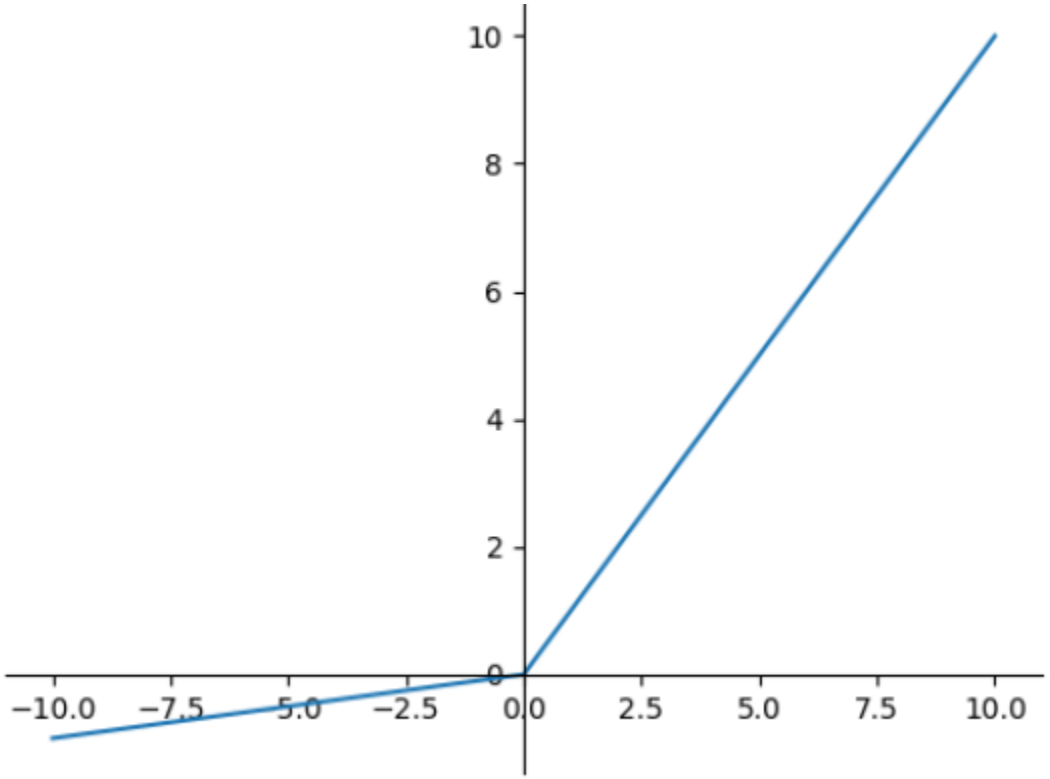
✓ Fast convergence

✓ Does not saturate

- Initializing ReLU neurons with slightly positive biases (0.01) makes it likely that they stay active for most inputs

Parametric Leaky ReLU

- $\sigma(x) = \max(\boldsymbol{a} \cdot x, x)$
  - a – one more parameter to backprop into
  - Does not die



Softmax function

- If we need to classify inputs into C different classes, we put C units in the last layer to produce C one-vs.-others scores $f_1, f_2, \ldots, f_C$
- Apply softmax function to convert these scores to probabilities:
  - $\text{softmax}(f_1, \ldots, f_c) = \left(\frac{\exp(f_1)}{\sum_j \exp(f_j)}, \ldots, \frac{\exp(f_C)}{\sum_j \exp(f_j)} \right)$
- If one of the inputs is much larger than the others, then the corresponding softmax value will be close to 1 and others will be close to 0
- Use log likelihood (categorical cross-entropy) loss:
  - $l(\mathbf{x}_i, y_i; \mathbf{w}) = -\sum_c \log P_{\mathbf{w}}(y_i == \text{c} \mid \mathbf{x}_i)$

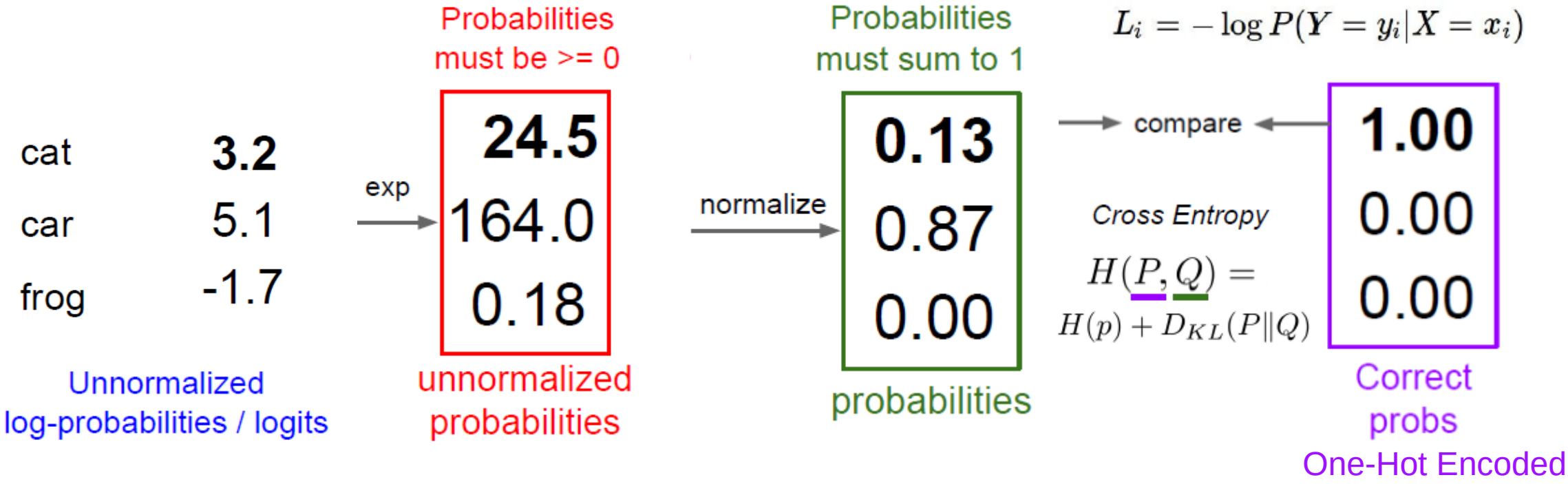
Softmax Classifier (Multinomial Logistic Regression)

- Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

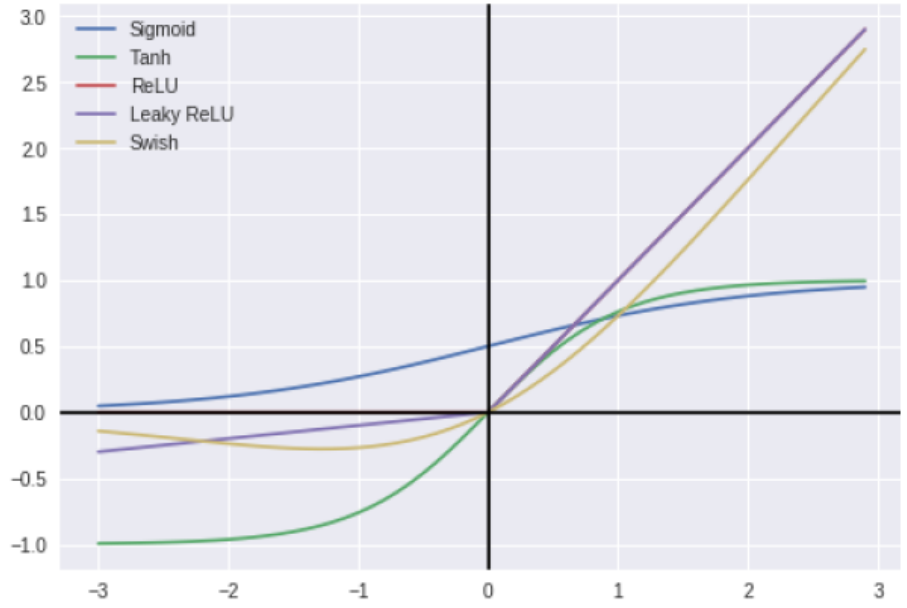
$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function



In a Nutshell

- Sigmoid is not really used.
- ReLU is the standard choice.
  - Second choice are the variants of ReLU or Maxout
- Recurrent nets will require TanH or similar.



ACTIVATION FUNCTION	EQUATION	RANGE
Linear Function	$f(x) = x$	$(-\infty, \infty)$
Step Function	$f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$	$\{0, 1\}$
Sigmoid Function	$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Hyperbolic Tanjant Function	$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$	$(-1, 1)$
ReLU	$f(x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU	$f(x) = \begin{cases} 0.01 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$	$(-\infty, \infty)$
Swish Function	$f(x) = 2x\sigma(\beta x) = \begin{cases} \beta = 0 \; for \; f(x) = x \\ \beta \to \infty \; for \; f(x) = 2\max(0, x) \end{cases}$	$(-\infty, \infty)$

Stochastic Gradient Descent (SGD)

- **Stochastic gradient descent** is an iterative method for optimizing an objective function.

- It can be regarded as a stochastic approximation of gradient descent optimization.
  - $\nabla_\theta L = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L_i$

m training samples in the current minibatch

Gradient for the k-th minibatch

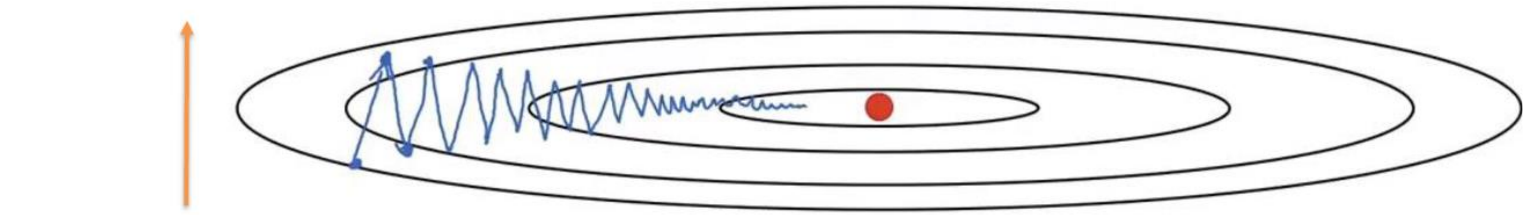
- It replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data)
  - Batch gradient descent: Use **all** examples in each iteration
  - Stochastic gradient descent: Use **1** example in each iteration
  - Minibatch gradient descent: Use **m** examples in each iteration

Minibatch Stochastic gradient descent

- The loss can be approximated with a subset of the data
  - Choose subset of trainset $m \ll n$
  - Minibatch size is another hyperparamter
    - Typically power of $2 \rightarrow 8,16,32,64,128, \ldots$
    - Smaller batch size means greater variance in the gradients
      - Noisy updates

Problems of SGD

- Gradient is scaled equally across all dimensions
  - i.e., cannot independently scale directions
  - Need to have conservative minimum learning rate to avoid divergence
  - Slower than ‘necessary’
- Finding good learning rate is an art by itself



Source: A. Ng

We're making many steps back and forth along this dimension. Would love to track that this is averaging out over time.

Would love to go faster here...
I.e., accumulated gradients over time

Gradient Descent with momentum

- Step will be largest when a sequence of gradients all point to the same direction
- Momentum can accelerate training and learning rate schedules can help to converge the optimization process.

$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^{k} - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{k})$$

accumulation rate ('friction', momentum)

velocity

learning rate

Gradient of current minibatch

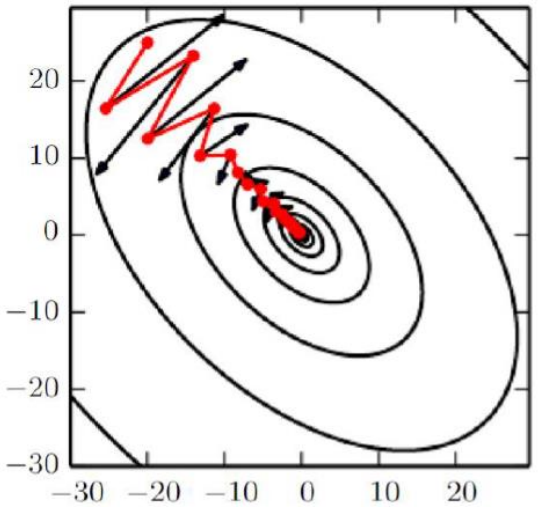
$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} + \boldsymbol{v}^{k+1}$$

weights of model

velocity

Exponentially-weighted average of gradient

Important: velocity $\boldsymbol{v}^{k}$ is vector-valued!



Source: I. Goodfellow

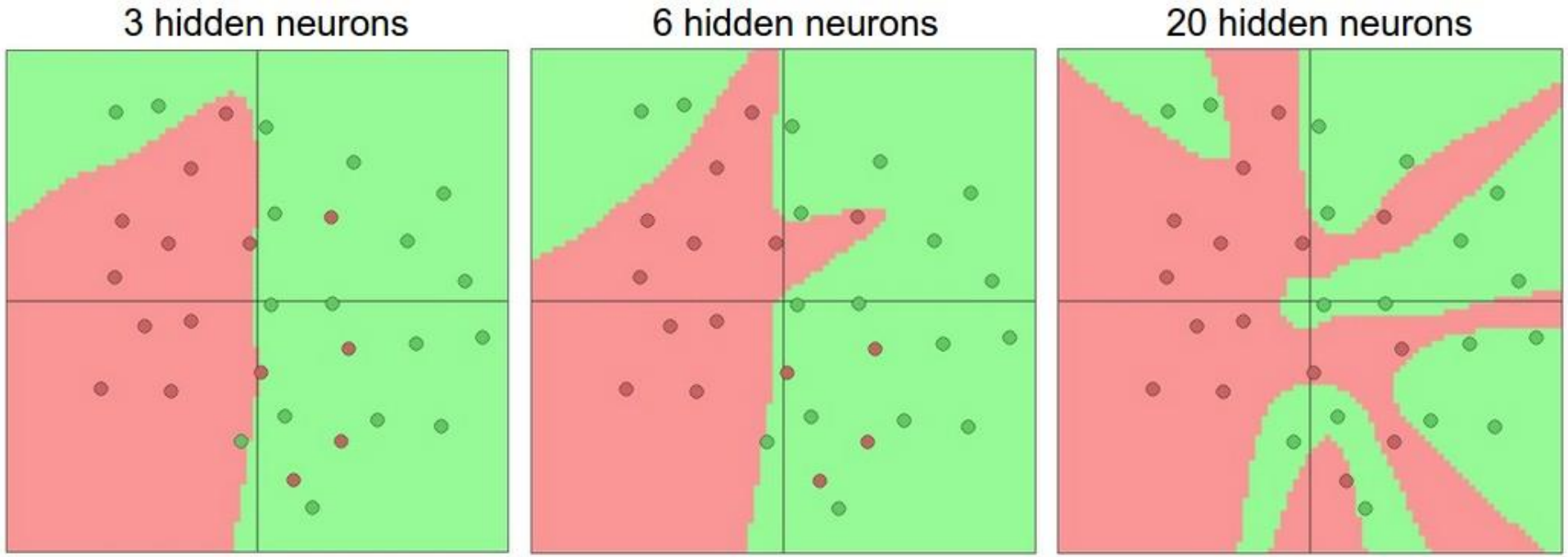
https://visualstudiomagazine.com/articles/2017/08/01/neural-network-momentum.aspx

There are a few optimizers

- ‘Vanilla’ SGD
- **SGD with Momentum**
- **RMSProp**
- **Adam**
- Adagrad
- Adadelta

Setting number of layers and their sizes

- Larger Neural Networks can represent more complicated functions
- Hidden layer size and network capacity:



Source: http://cs231n.github.io/neural-networks-1/

Regularization

- **Regularization**: Prevent the model from doing *too* well on training data

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

- L2-regularization (weight decay): (regularization parameter)

$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2}W^2$$

Enforces similar values for weights

λ = regularization strength (hyperparameter)

- L1-regularization:

$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2}|W|$$

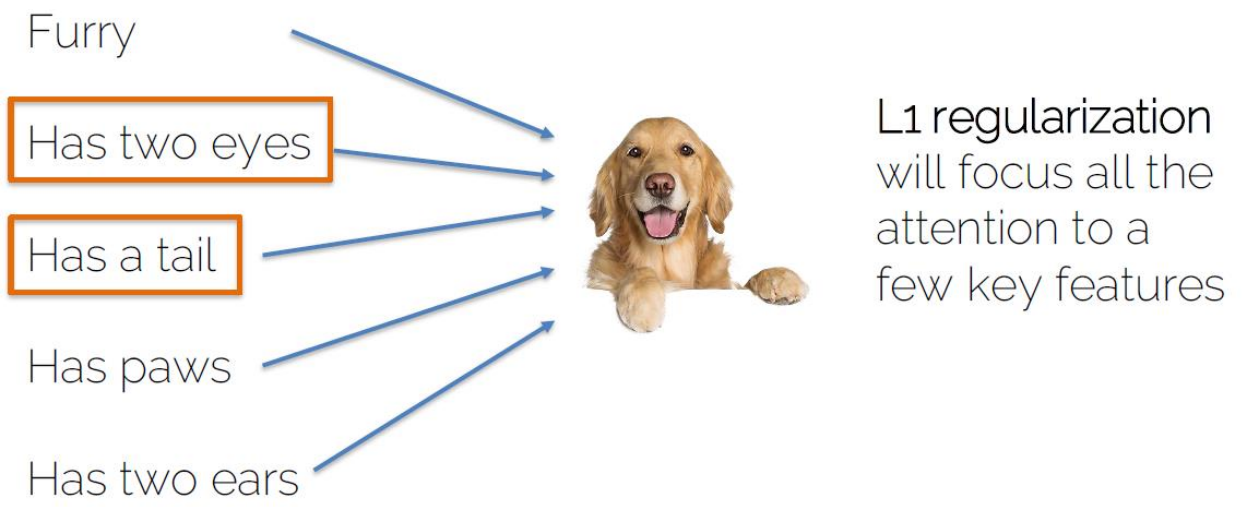
Enforces Sparsity

Regularization Effect

- L1 regularization:

$$R(W) = \sum_{i=1}^{n} |w_i|$$

$$W_1 = [0, 0.55, 0.45, 0, 0]$$

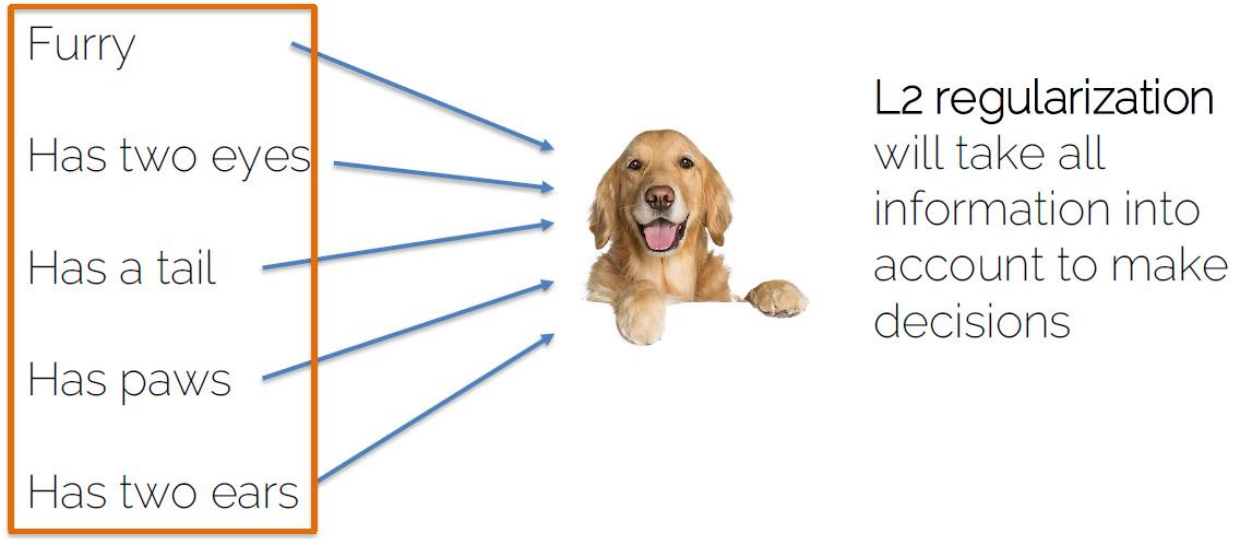


L1 regularization will focus all the attention to a few key features

- L2 regularization:

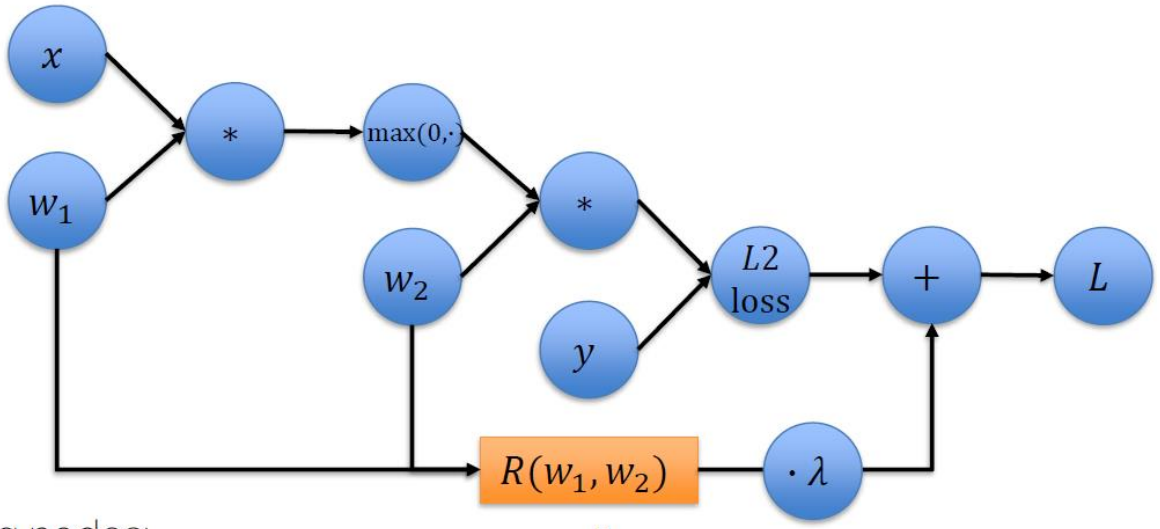
$$R(W) = \sum_{i=1}^{n} w_i^2$$

$$W_2 = [0.15, 0.25\ 0.25, 0.15, 0.2]$$



L2 regularization will take all information into account to make decisions

Regularization in the compute graph



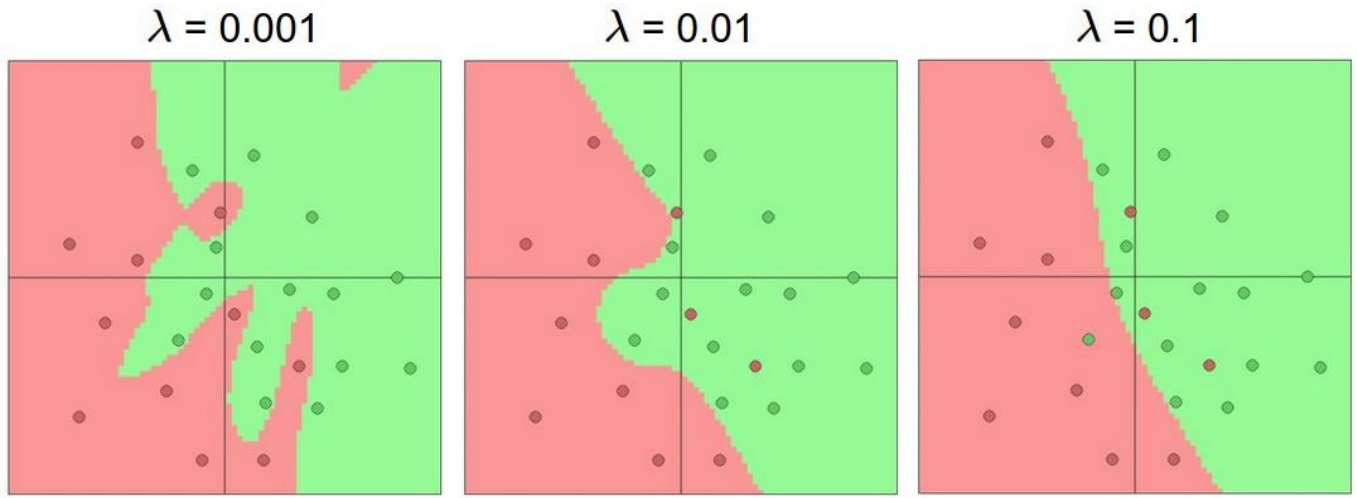
Combining nodes:
Network output + L2-loss + regularization

$$\sum_{i=1}^{n} \|w_2 \max(0, w_1 x_i) - y_i\|_2^2 + \lambda(w_1^2 + w_2^2)$$

Regularization Example

- It is common to add a penalty (e.g., $L2$) on weight magnitudes to the objective function:
- $E(\mathbf{w}) = \sum_i \mathrm{L}(\mathrm{f}(\mathbf{x}_i, W), y_i) + \lambda \|W\|^2$
  - Quadratic ($L2$) penalty encourages network to use all of its inputs "a little" rather than a few inputs "a lot"

Decision Boundary

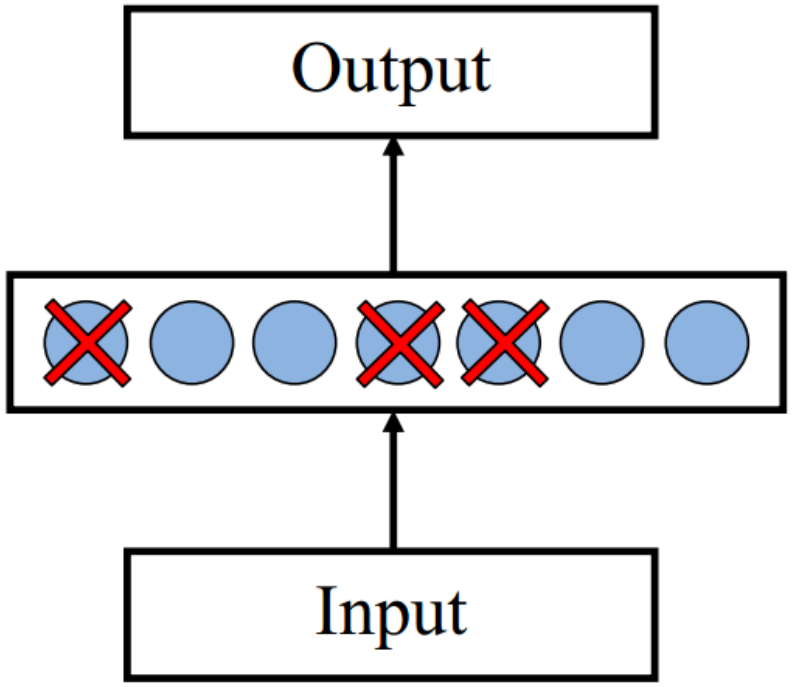


Notice that the training error is reduced

Source: http://cs231n.github.io/neural-networks-1/

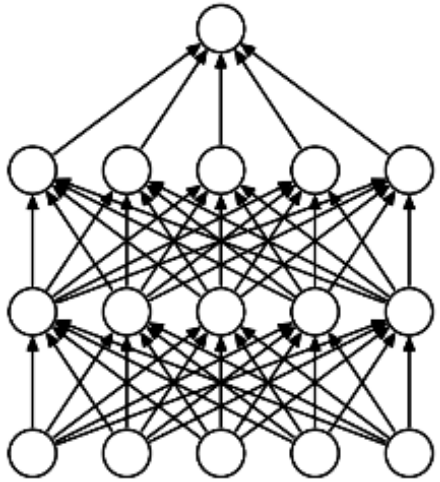
Regularization: Dropout

- Proposed by (Hinton et al, 2012)
- Randomly turn off some neurons
- Allows individual neurons to independently be responsible for performance.
- Implement as another layer that we can use.
  - Zeros activations

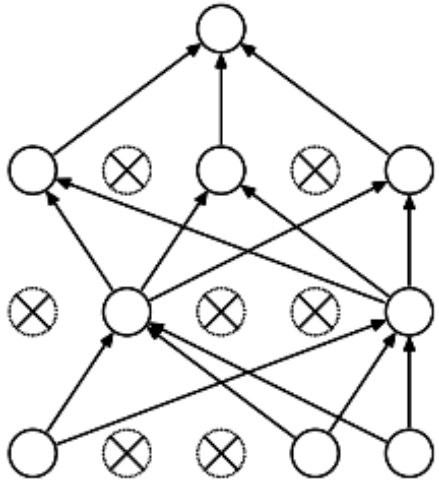


Dropout training

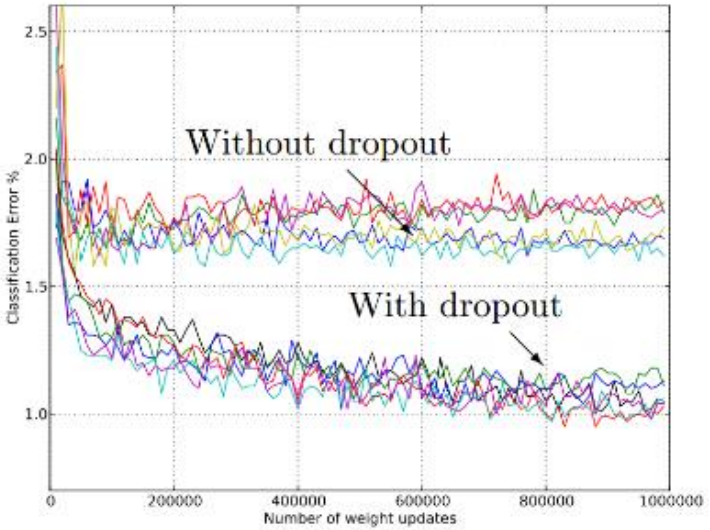
- We can specify the drop rate (or keep rate) per layer, e.g., 0.5
- Each time decide whether to delete one hidden unit with some probability p



(a) Standard Neural Net



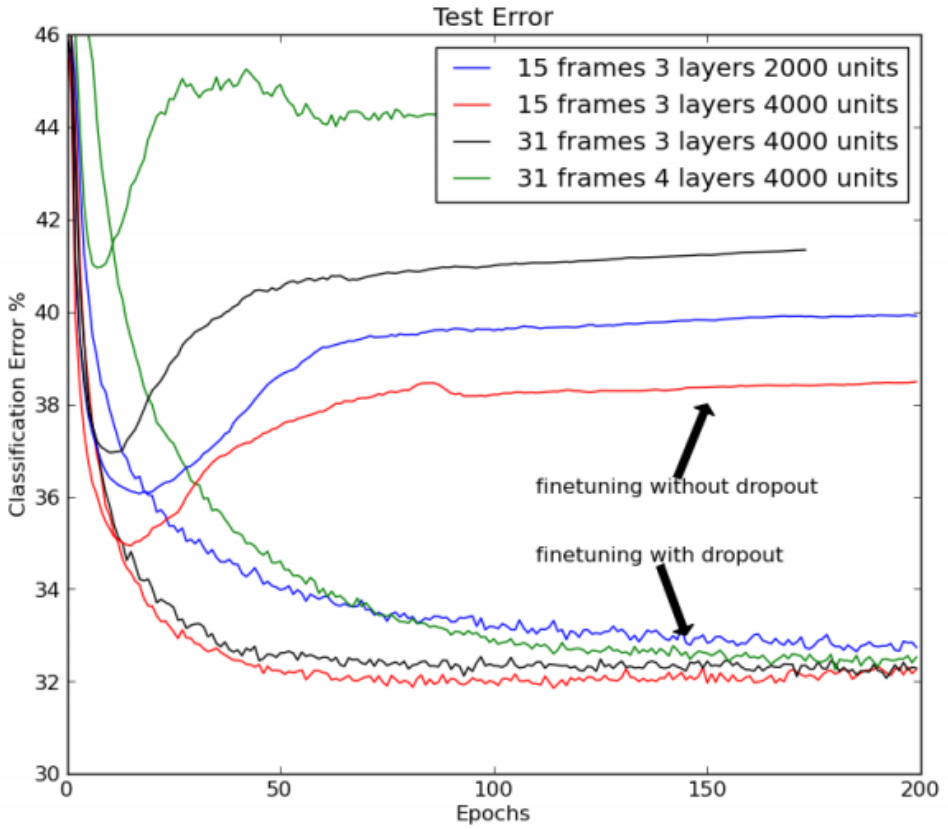
(b) After applying dropout.



Dropout: A simple way to prevent neural networks from overfitting [Srivastava JMLR 2014]

Adapted from Jia-bin Huang

Dropout training

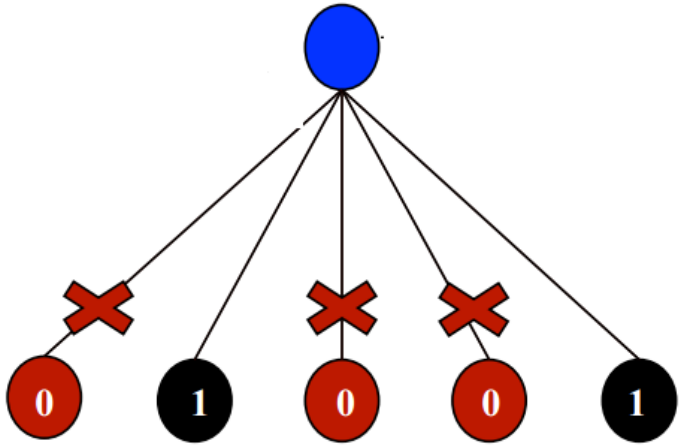


- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

Dropout training

- Model averaging effect
  - Among 2^H models, with shared parameters
    - H: number of units in the network
  - Only a few get trained
  - Much stronger than the known regularizer

- What about the input space?
  - Do the same thing!



Batch Normalization

- Batch Normalization (BN) is a normalization method/layer for neural networks
- Usually, inputs to neural networks are normalized to either the range of $[0,1]$ or $[-1,1]$ or to $mean = 0$ and $variance = -1$
- BN essentially performs *Whitening* to the intermediate layers of the networks.
  - An input distribution is considered white for a vector x if it has:
    - Zero mean
    - Unit variance
    - Input is Decorrelated (Correlation matrix is I)

Batch Normalization

- It was motivated by the problem of internal covariance shift, where changes in the distribution of the inputs of each layer affects the affect the learning rate of the network.
- It tries to make the output of a layer have a mean of 0 and unit variance by normalizing the output based on statistics of the batch earning rate of the network.

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

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

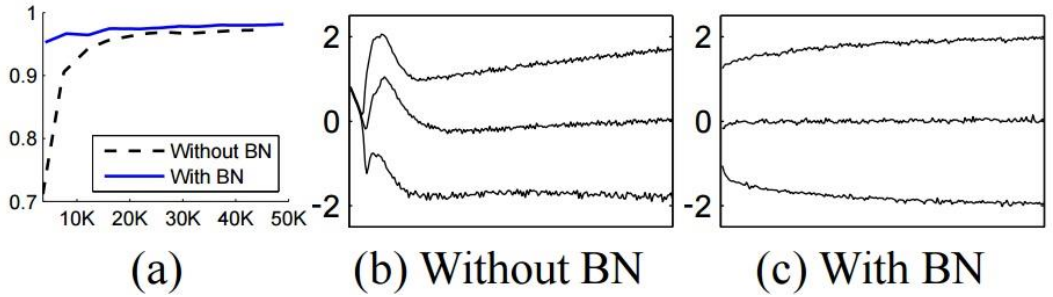
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

Used internally at that layer

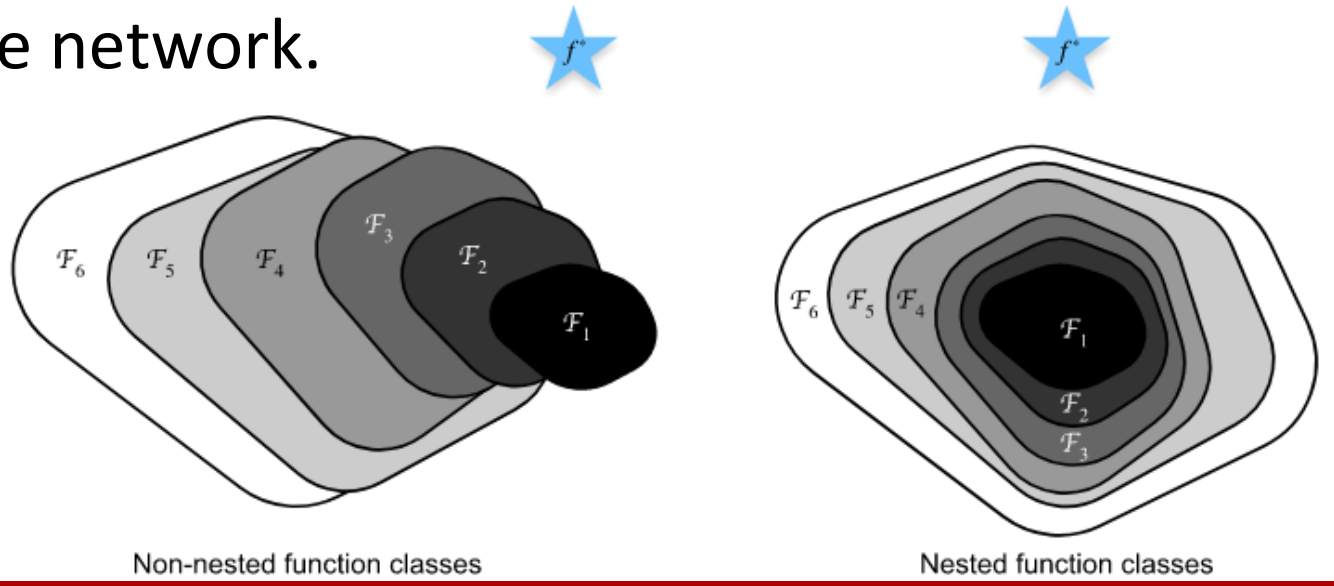
Propagated to Next layer



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

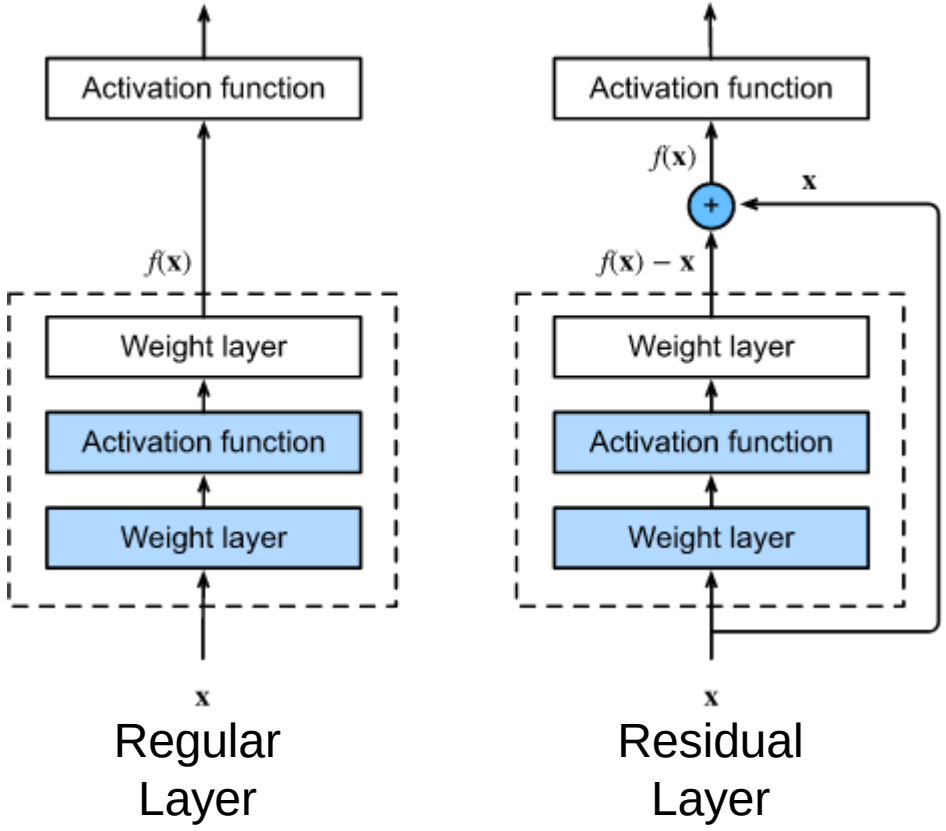
What is the impact of adding a layer?

- Consider F, the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach.
- Thus, only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network.



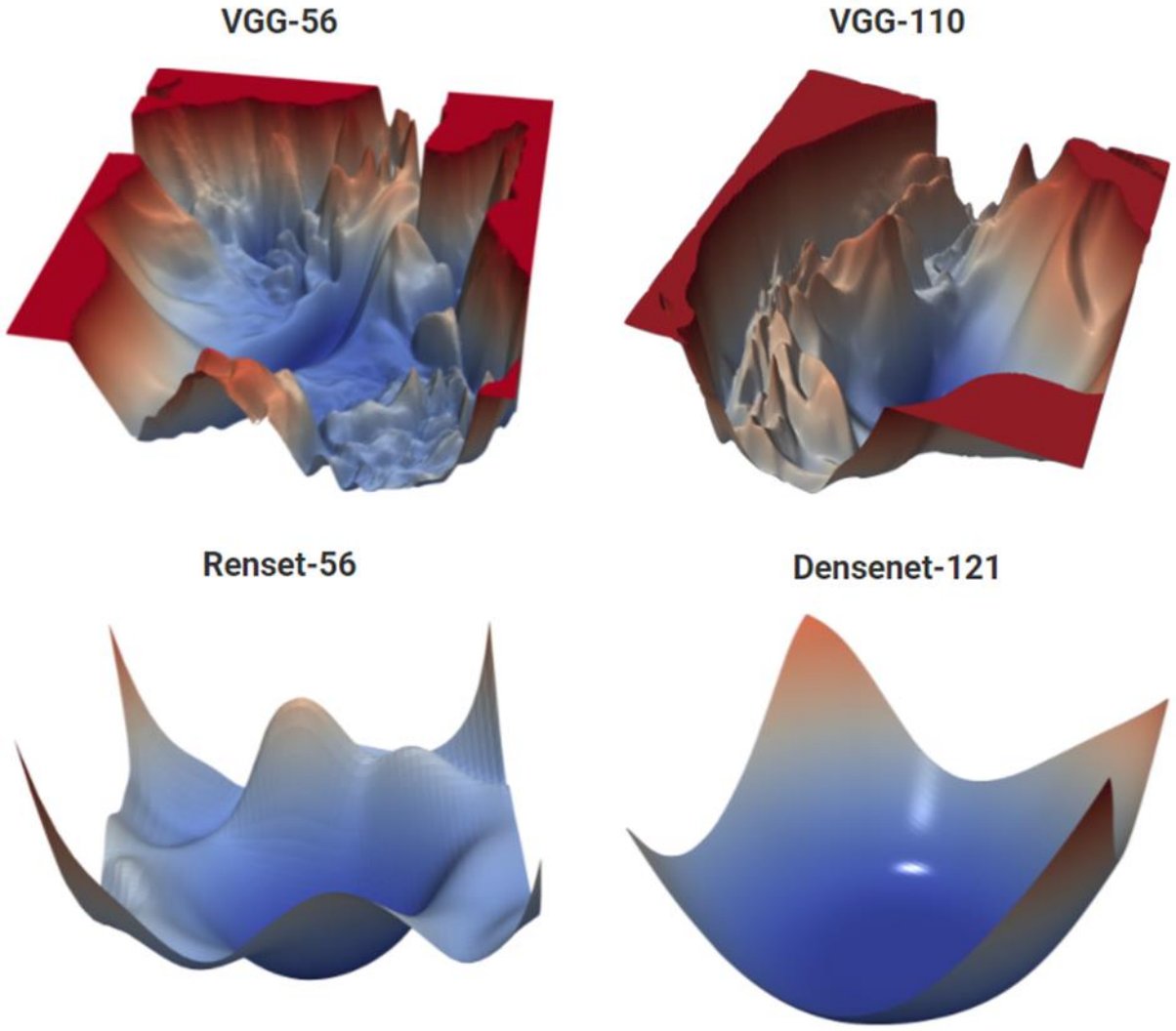
Residual Learning

- The residual mapping can learn the identity function more easily, such as pushing parameters in the weight layer to zero.
- We can train an effective deep neural network by having residual blocks. Inputs can forward propagate faster through the residual connections across layers.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.



He et al., 2016

Residual loss landscape



- Shallow networks have smooth landscapes populated by wide, convex regions.
- However, as networks become deeper, landscapes spontaneously become “chaotic” and highly non-convex, leading to poor training behaviour.
- Certain network architecture designs, like skip connections and wide networks (many filters per layer), undergo this chaotic transition at much more extreme depths, enabling the training the deep networks.

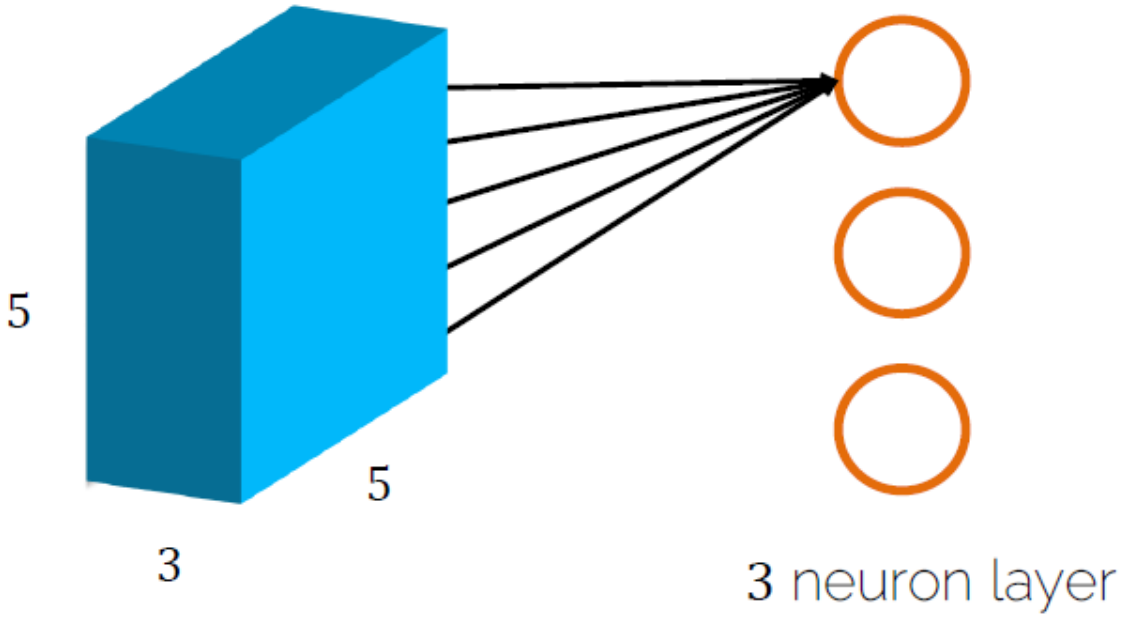
What is wrong with dense layers?

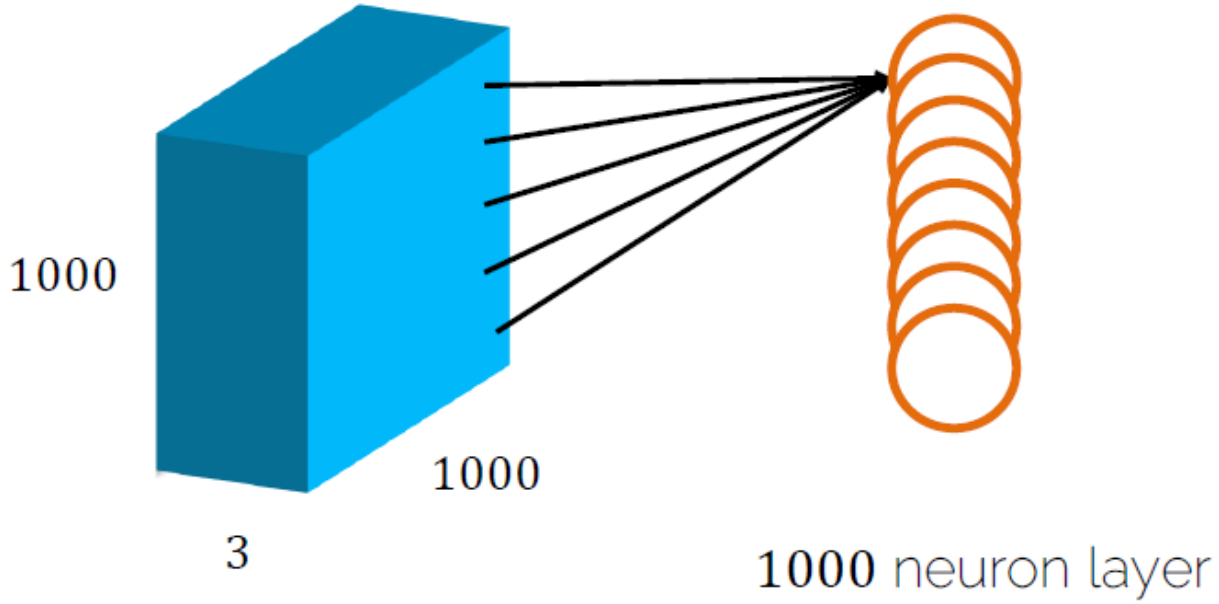
- We cannot make networks arbitrary complex

- Why don’t just go deeper to get better?
  - No Structure!!
  - It is just brute force.
  - Optimization becomes hard
  - Performance plateaus/drops!

- We want to restrict the degrees of freedom
  - **We want a layer with structure**
  - **Weight sharing** → using the same weights for the same image

What is wrong with dense layers?



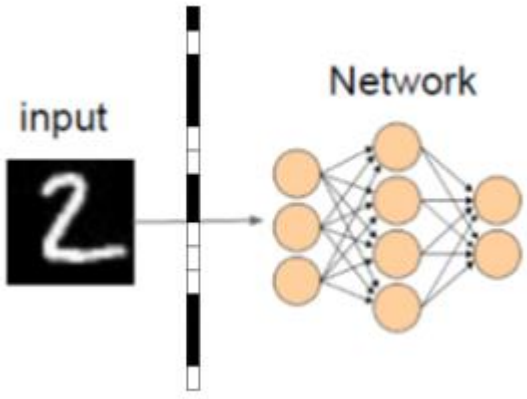


- 75 weights for the whole 5×5 tensor on the 3 channels
- 3×75 for all three neurons$\rightarrow$ 225

- $3 \times 10^9 \rightarrow 3$ billion weights!
- **IMPRACTICAL!**

What is wrong with dense layers?

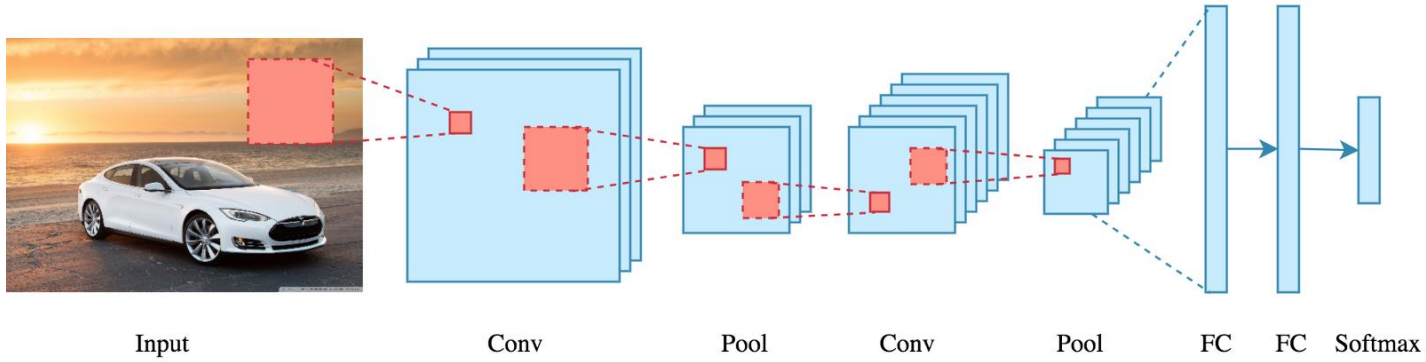
- The models that we have discussed so far are appropriate options when we are dealing with data consist of rows corresponding to examples and columns corresponding to features.
  - There might be interactions among the features, but we do not assume any structure a priori concerning how the features interact.



An image of the “Where’s Waldo” game.

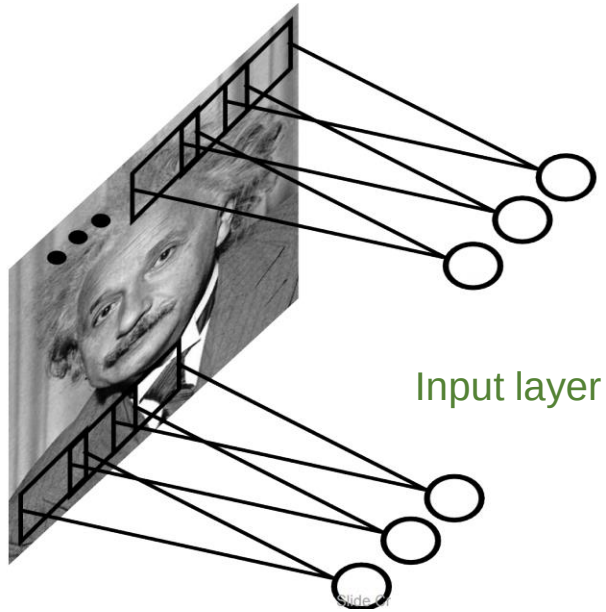
Convolutional Neural Networks (CNN)

- CNN is a class of deep neural networks
- Typically, used for image processing, signal processing, sound recognition.
- Avoids the fully connectness between layers
- Avoids the overfitting problem and requires significantly fewer adjustable parameters compared to the full-connected multilayer neural network



Convolutional Layer

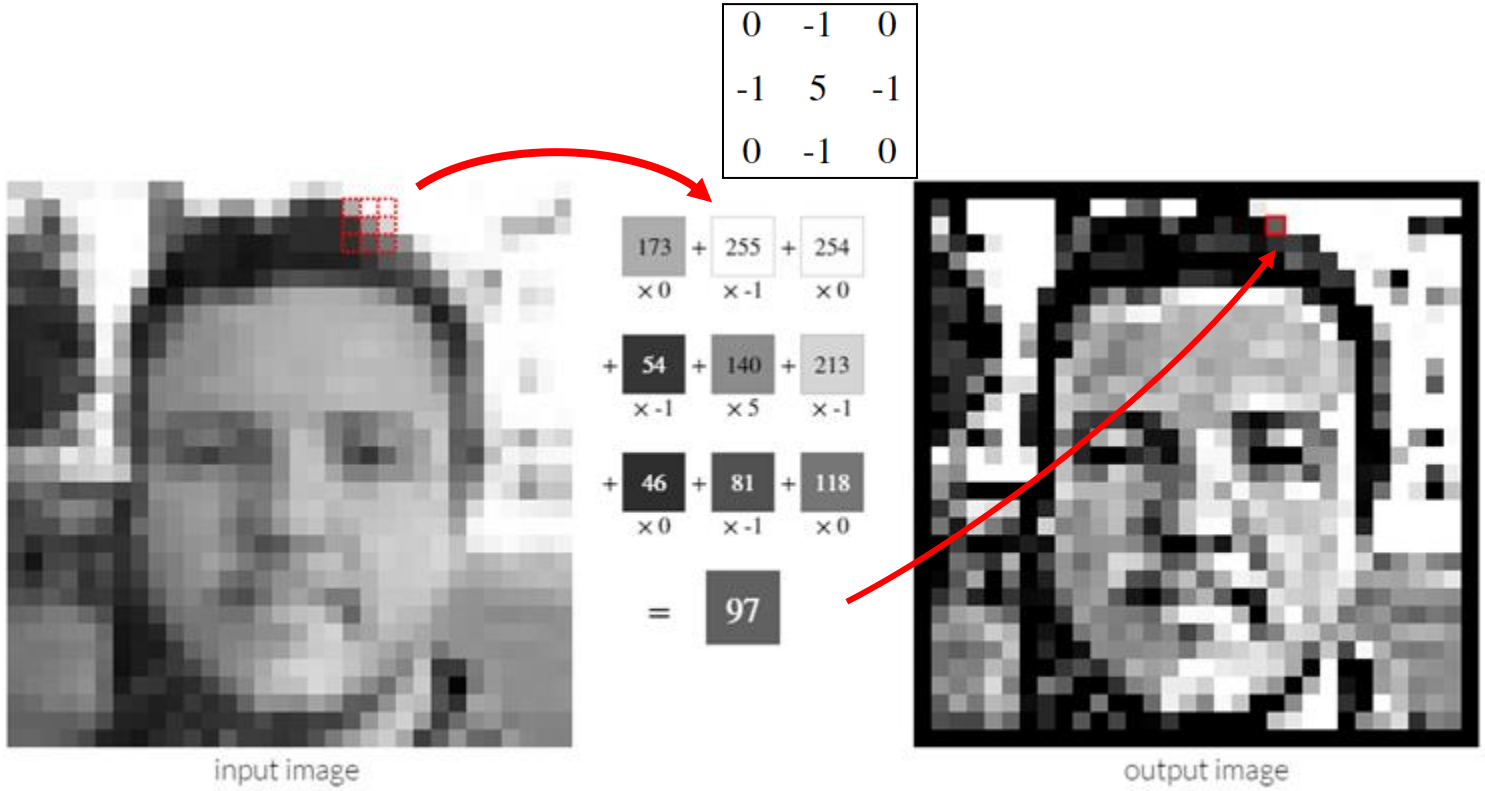
- Filters to capture different patterns in the input space.
  - Share parameters across different locations (assuming input is stationary)
  - Convolutions with learned filters
- Filters will be learned during training.
- The issue of variable-sized inputs will be resolved with a pooling layer.



Slide Credit: Marc'Aurelio Ranzato

Convolution Operator

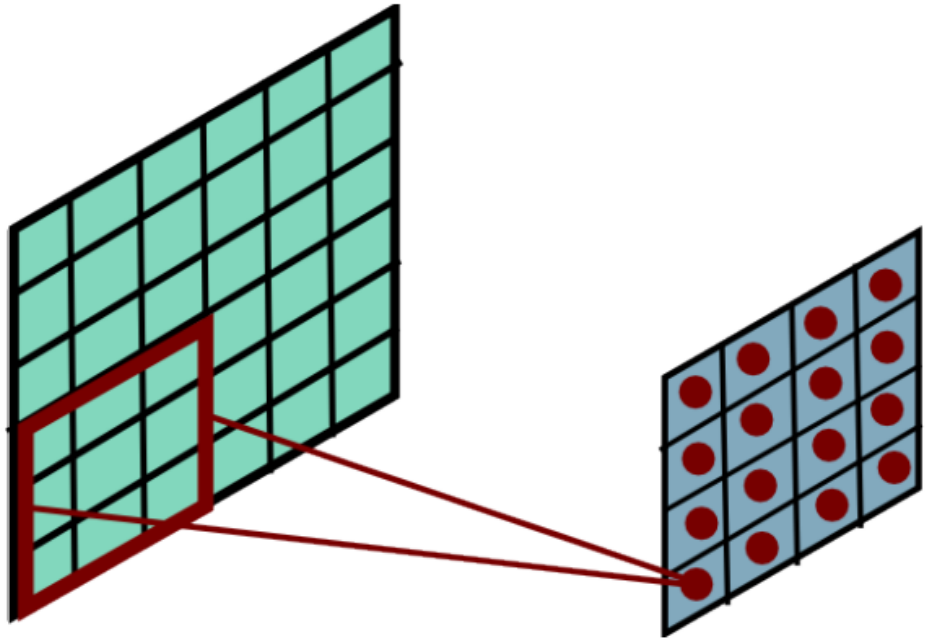
- Convolution in two dimension:
  - Example: Sharpen kernel:



Try other kernels: http://setosa.io/ev/image-kernels/

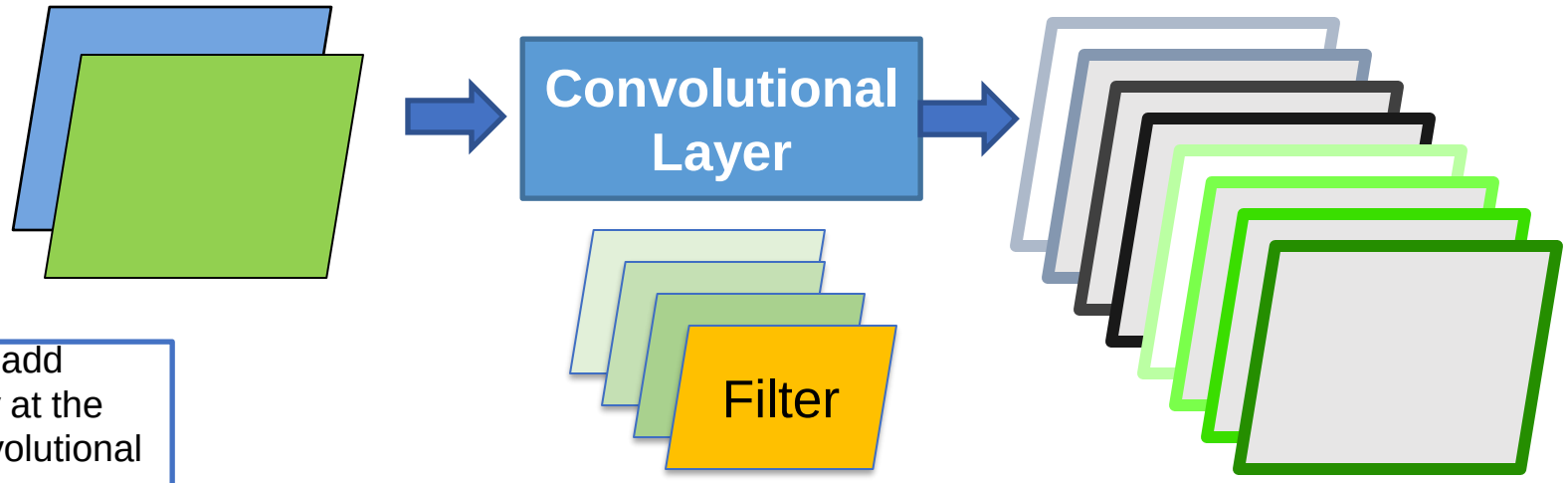
Convolution Operator (2)

- Convolution in two dimension:
  - Convolve a filter matrix across the image matrix



Convolutional Layer

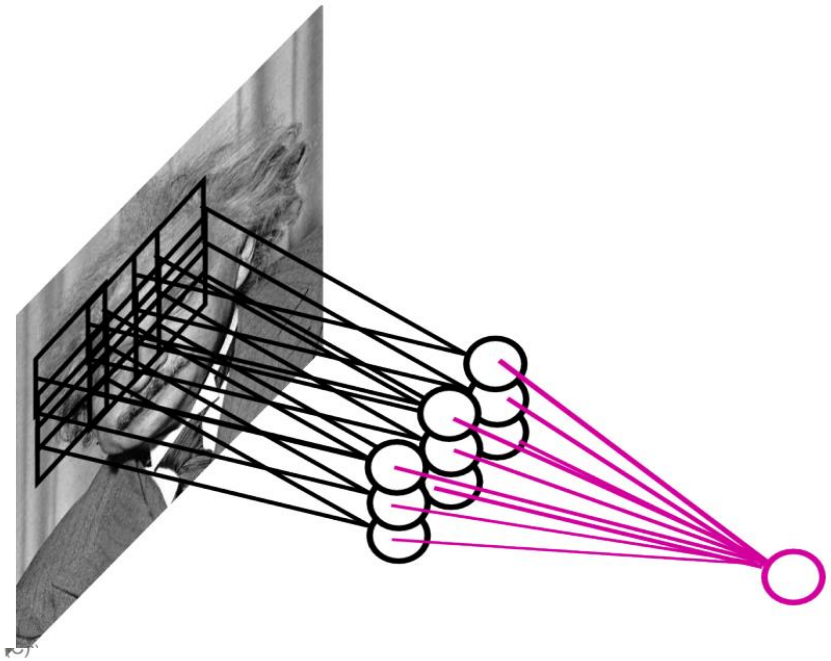
- The convolution of the input (vector/matrix) with weights (vector/matrix) results in a response vector/matrix.
- We can have multiple filters in each convolutional layer, each producing an output.
- If it is an intermediate layer, it can have multiple inputs!



One can add nonlinearity at the output of convolutional layer

Pooling Layer

- Reduce the dimension of the data by combining the outputs of a cluster of neurons to a single neuron
  - A layer which reduces inputs of different size, to a fixed size.
  - Pooling



Slide Credit: Marc'Aurelio Ranzato

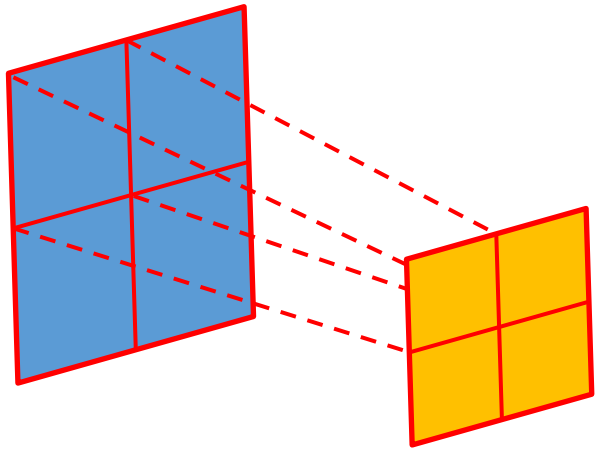
Pooling Layer

- How to handle variable sized inputs?
  - A layer which reduces inputs of different size, to a fixed size.
  - **Pooling**
  - Different variations
    - Max pooling

$$h_i[n] = \max_{i \in N(n)} \tilde{h}\,[i]$$

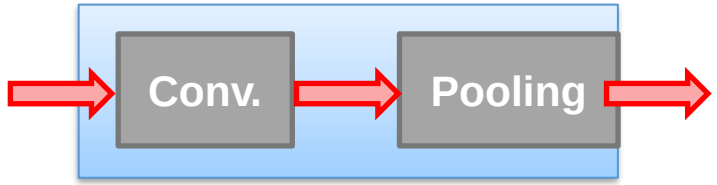
    - Average pooling

$$h_i[n] = \frac{1}{n} \sum_{i \in N(n)} \tilde{h}\,[i]$$

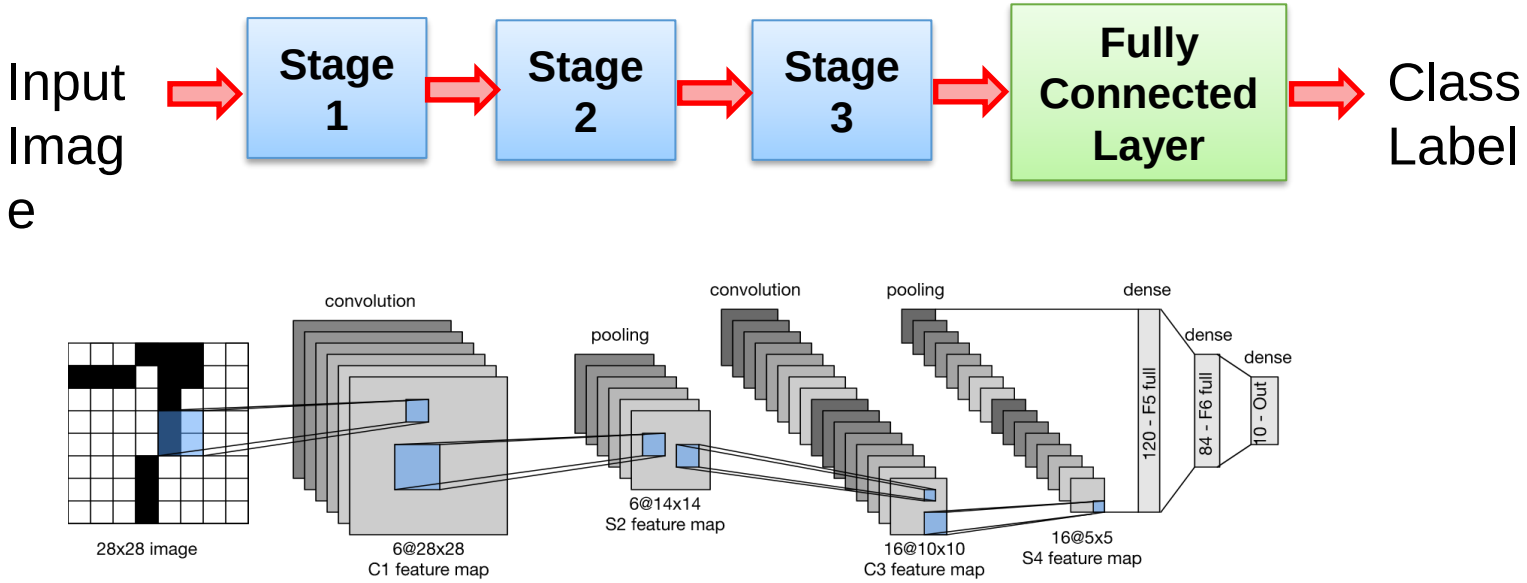


Convolutional Nets

- One stage structure:

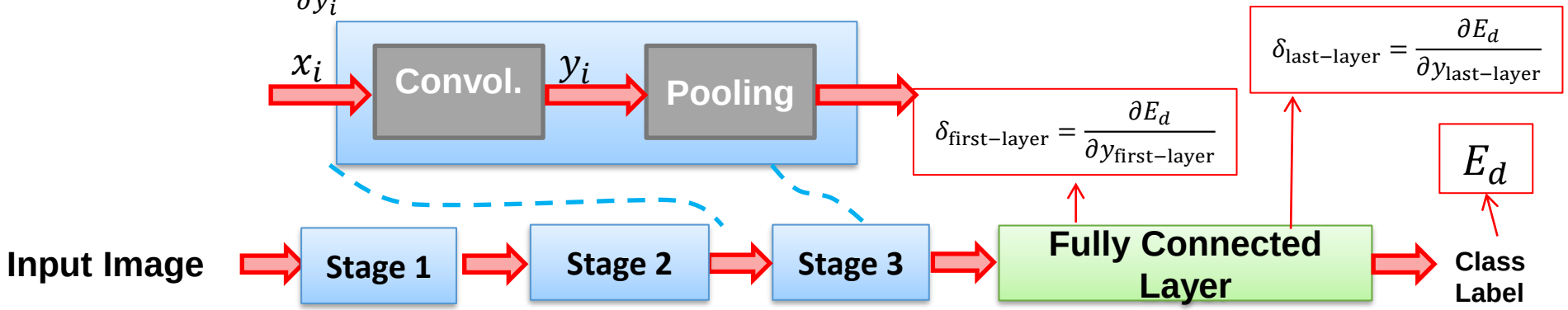


- Whole system:

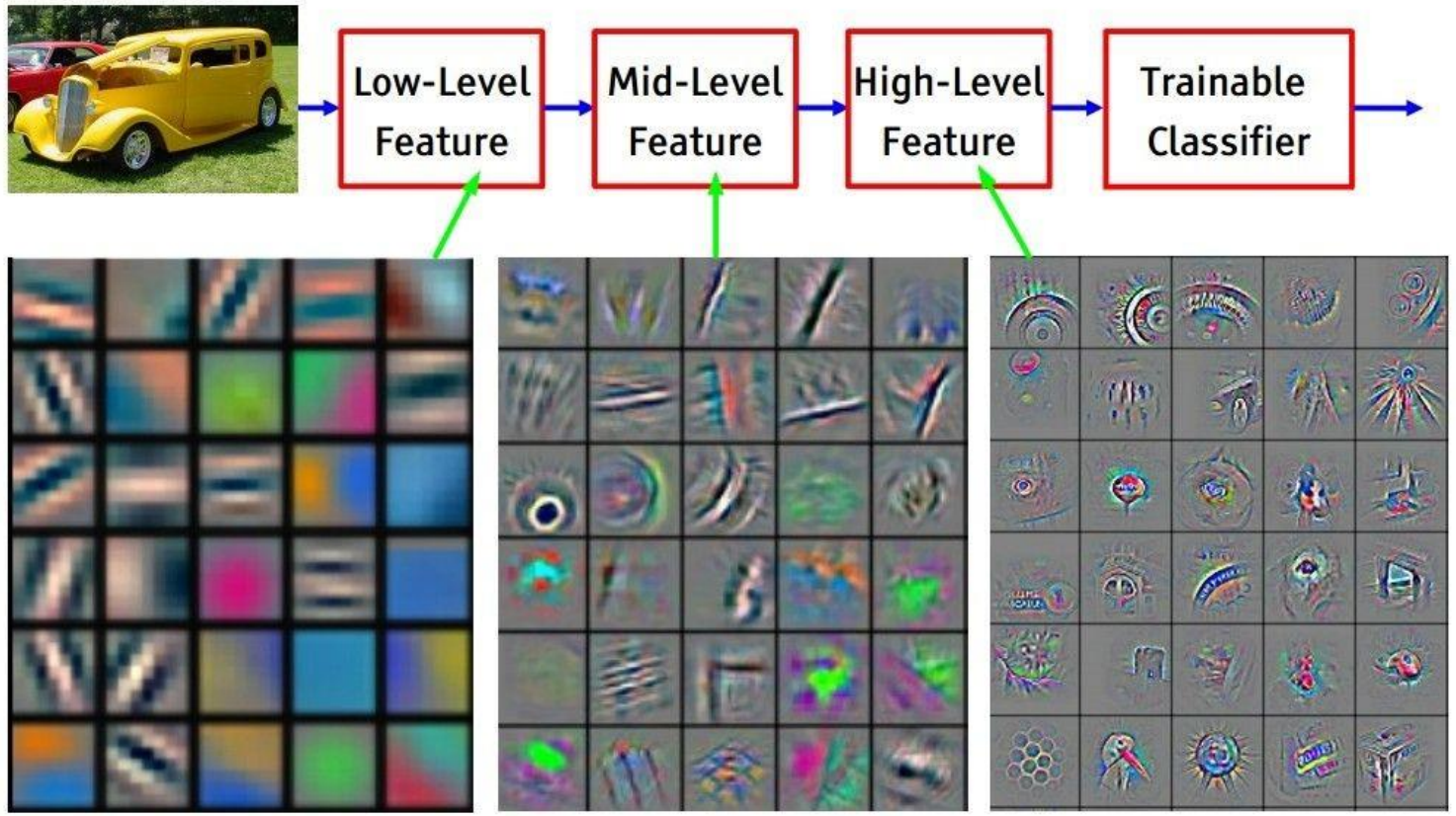


Training a ConvNet

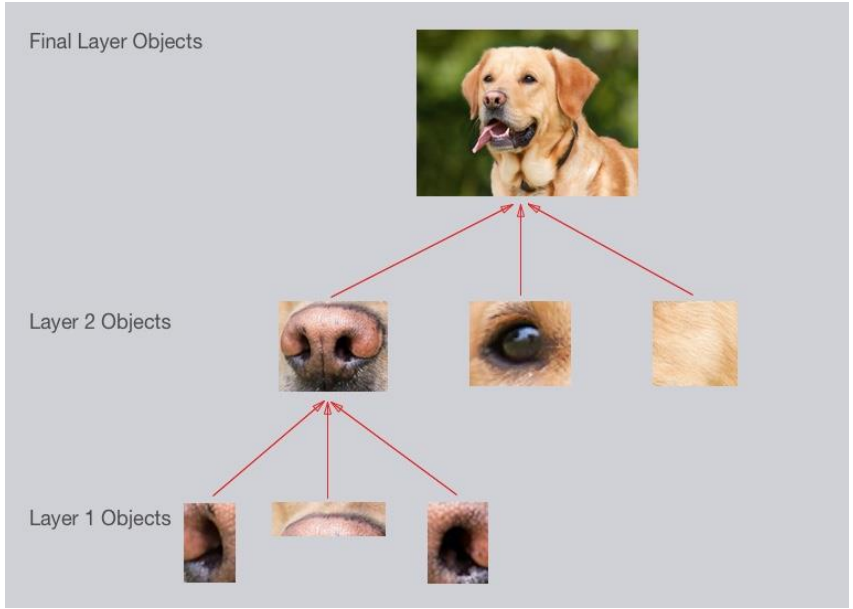
- The same procedure from Back-propagation applies here.
  - Remember in backprop we started from the error terms in the last stage, and passed them back to the previous layers, one by one.
- Back-prop for the pooling layer:
  - Consider, for example, the case of “max” pooling.
  - This layer only routes the gradient to the input that has the highest value in the forward pass.
  - Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called *the switches*) so that gradient routing is efficient during backpropagation.
  - Therefore we have: $\delta = \frac{\partial E_d}{\partial y_i}$



Convolutional Network



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

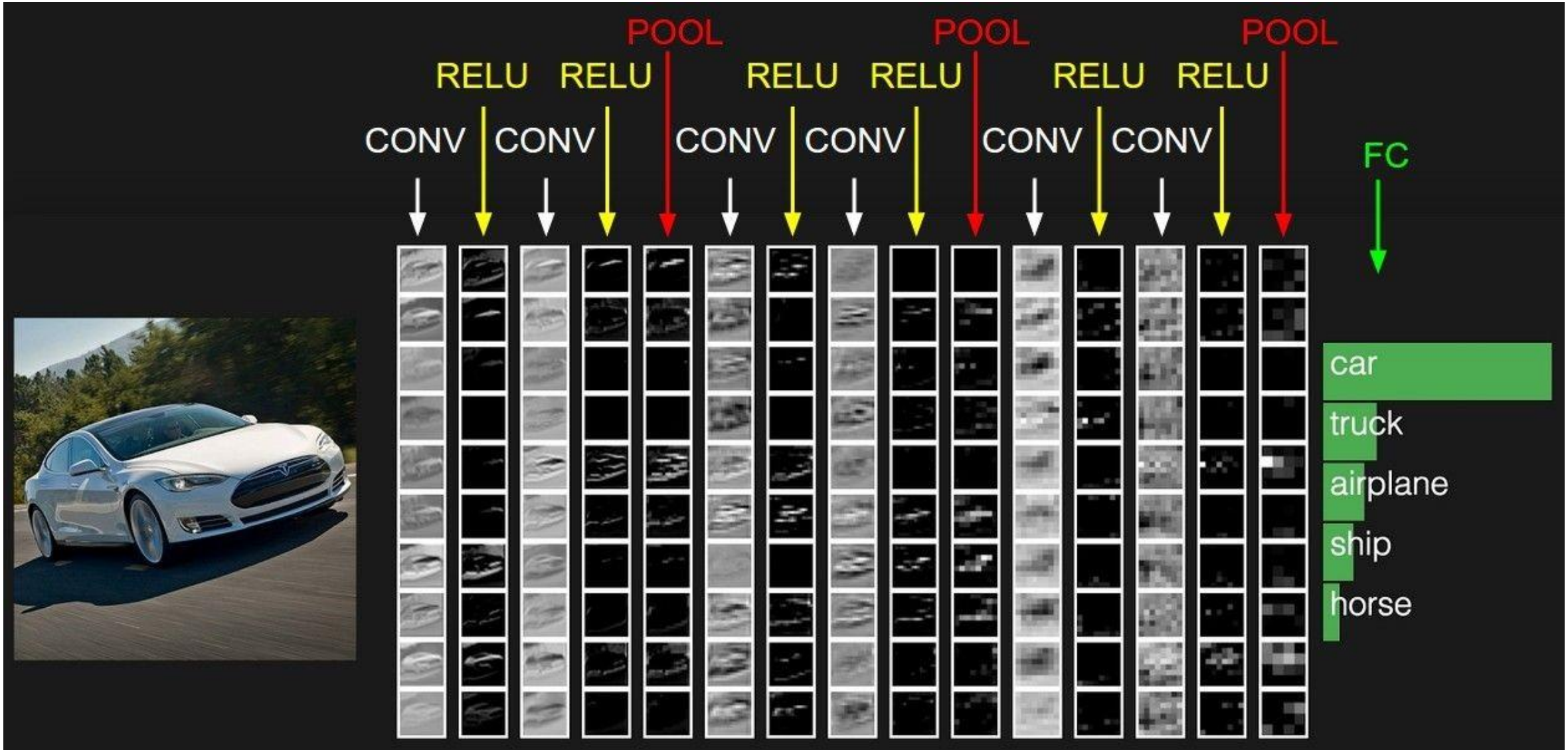


[From recent Yann LeCun slides]

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Credit: Andrej Karpathy

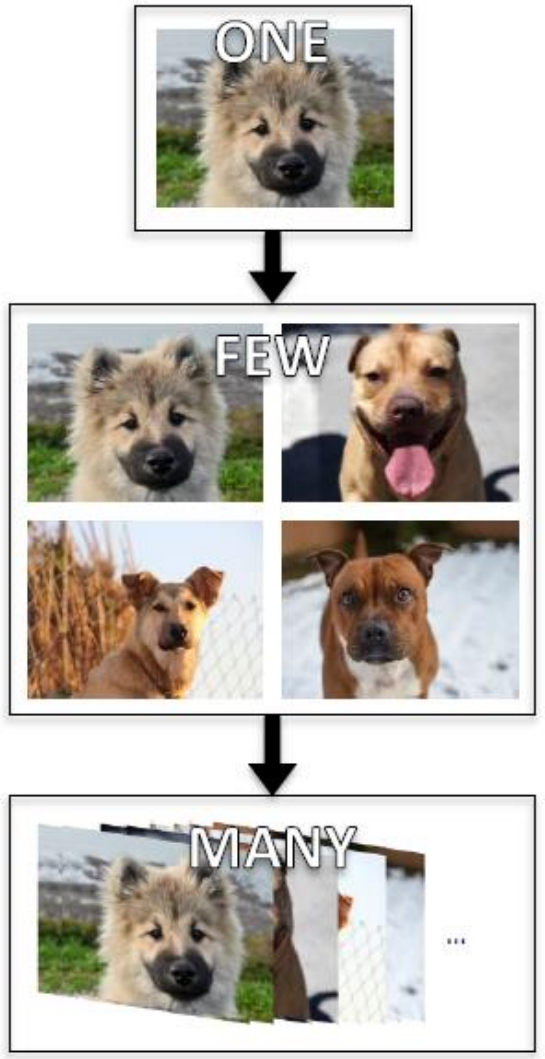
Convolutional Network



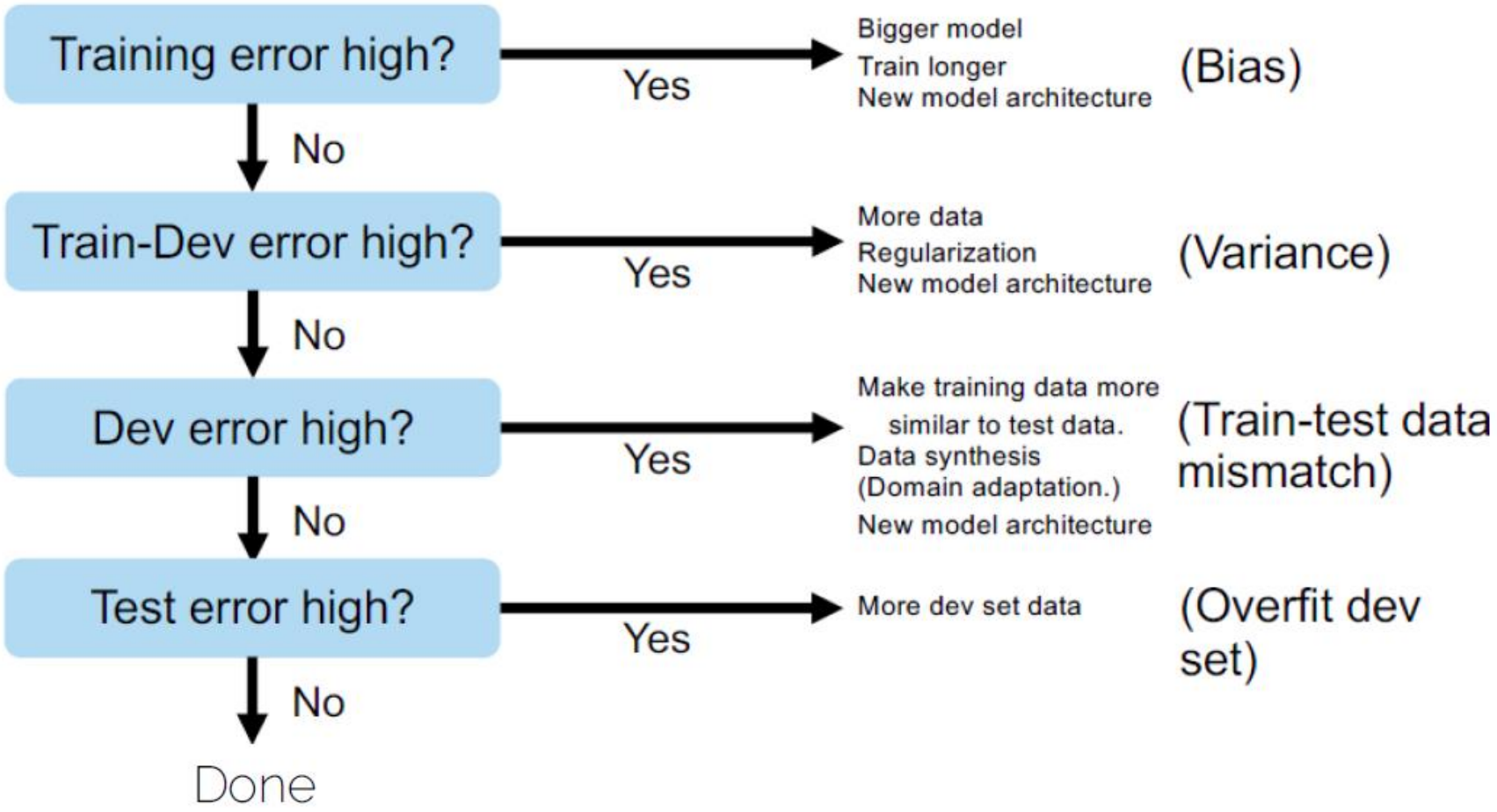
Andrej Karpathy

How to Start

- Start with single training sample
  - Check if output correct
  - Overfit → accuracy should be 100% because input just memorized
- Increase to handful of samples (e.g. 4)
  - Check if input is handled correctly
- Move from overfitting to more samples
  - $5, 10, 100, 1000, \ldots$
- At some point, you should see generalization



Basic Recipe for Machine Learning



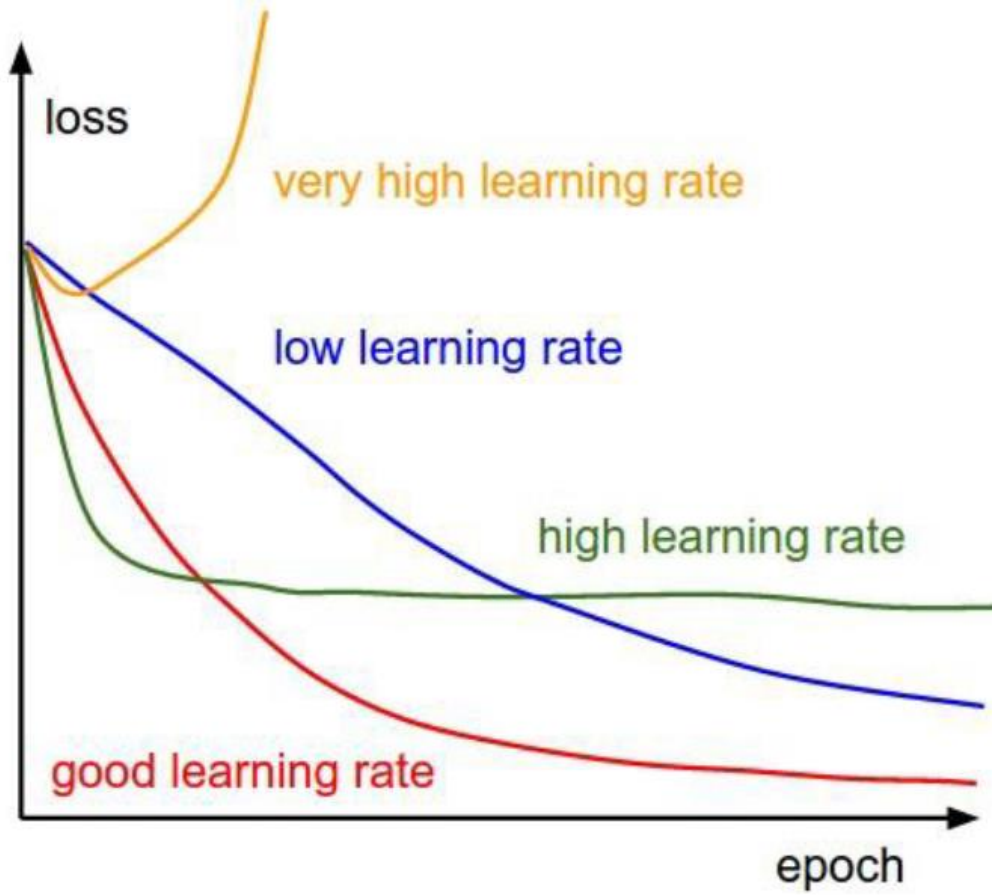
Credit: Andrew Ng

Find a Good Learning Rate

- Use all training data with small weight decay
- Perform initial loss sanity check e.g., $\log(C)$ for softmax with C classes
- Find a learning rate that makes the loss drop significantly (exponentially) within 100 iterations
- Good learning rates to try: $1e-1, 1e-2, 1e-3, 1e-4$
  - Choice also depends on optimizer

Learning Rate: Implications

- What if too high?
- What if too low?



http://cs231n.github.io/neural-networks-3/

Hyperparameters

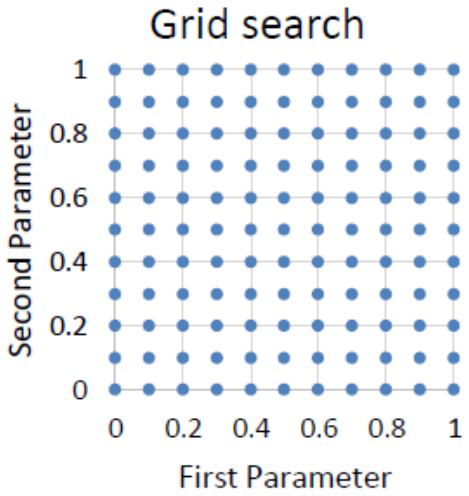
- Network architecture (e.g., numlayers, #weights)
- Number of iterations
- Learning rate(s)(i.e., solver parameters, decay, etc.)
- Regularization (more later next lecture)
- Batch size

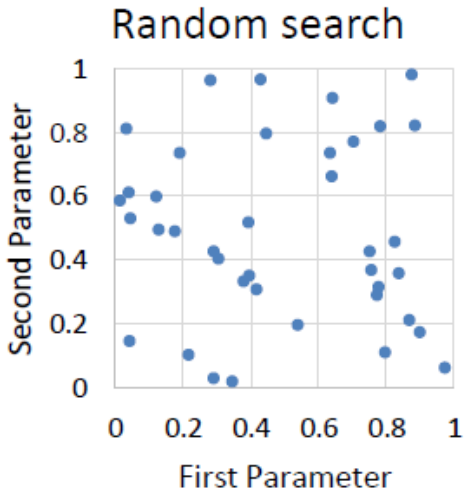
…

- Overall: learning setup + optimization = hyperparameters

Hyperparameter Tuning

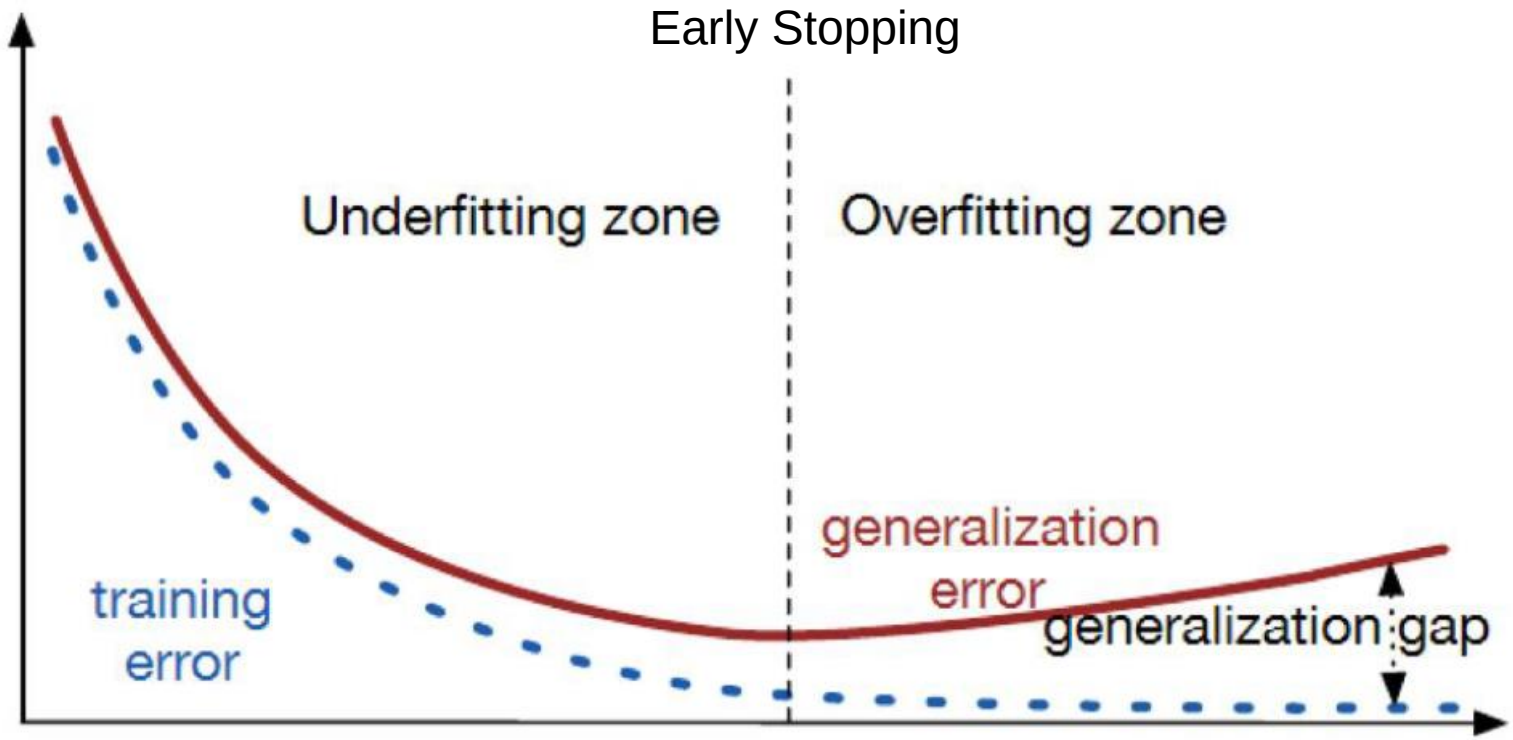
- Methods:
  - **Manual** Search
    - Most common ☺
  - **Grid** search (structured, for 'real' applications)
    - Define ranges for all parameters spaces and select points
    - Usually pseudo-uniformly distributed
    - Iterate over all possible configurations
  - **Random** Search:
    - Like grid search but one picks points at random in the predefined ranges
  - Stand on the shoulder of giants:
    - Take what is found to work and adapt from there





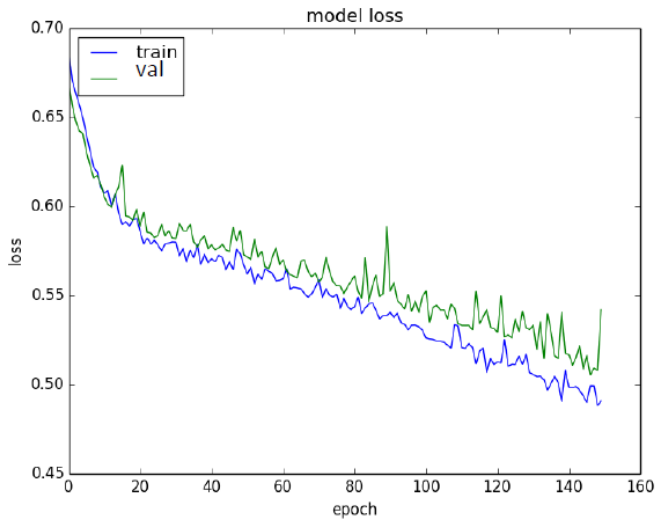
Over- and under- fitting

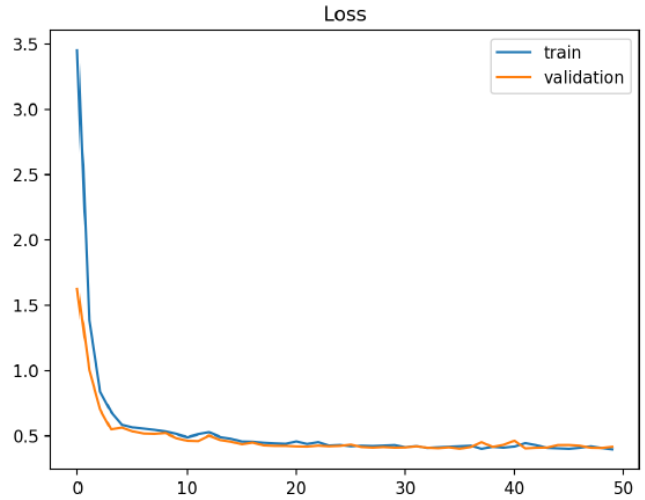
- Monitor Loss



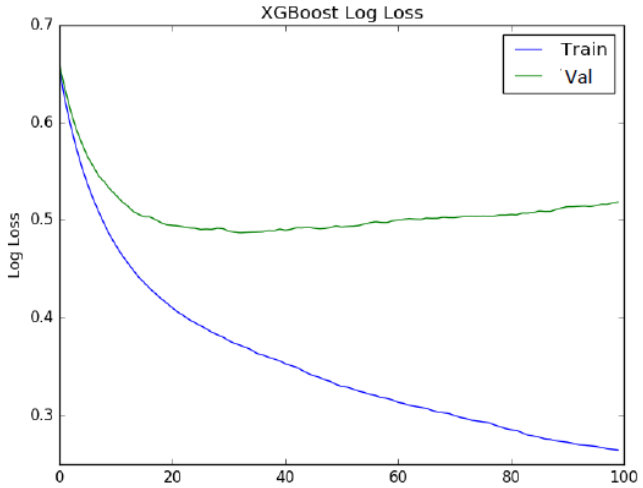
Learning Curves

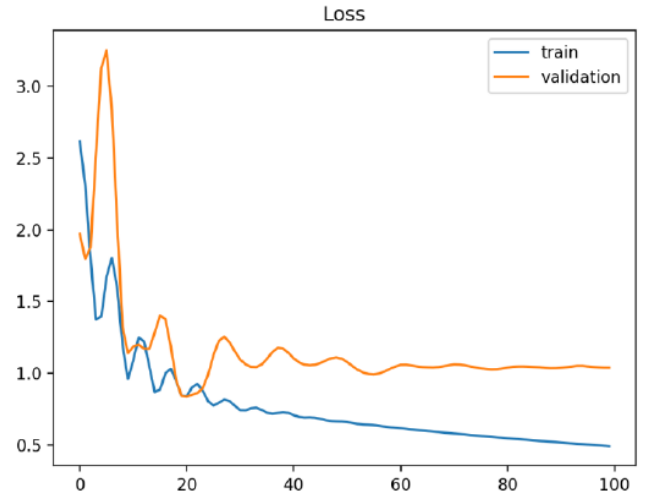
- Good Learning Curves





- Overfitting Curves





Common Mistakes in Practice

- Did not overfit to single batch first
  - Did not catch easy mistakes early on
- Forgot to toggle train/eval mode for network during inference
- Used different pre-processing for training/validation or forgot to apply during test time.
- Passed softmaxed outputs to a loss functions that expects logits

Summary

- Advances in initialization/normalization/architecture can lead to performance improvements.
- A CNN is a network that employs convolutional layers.
  - In a CNN, we interleave convolutions, nonlinearities, and (often) pooling operations.
  - In a CNN, convolutional layers are typically arranged so that they gradually decrease the spatial resolution of the representations, while increasing the number of channels.
  - In traditional CNNs, the representations encoded by the convolutional blocks are processed by one or more fully-connected layers prior to emitting output.
- Catch easy mistakes early on with sanity-check experiments and small scale data subset