

NEURAL NETWORKS

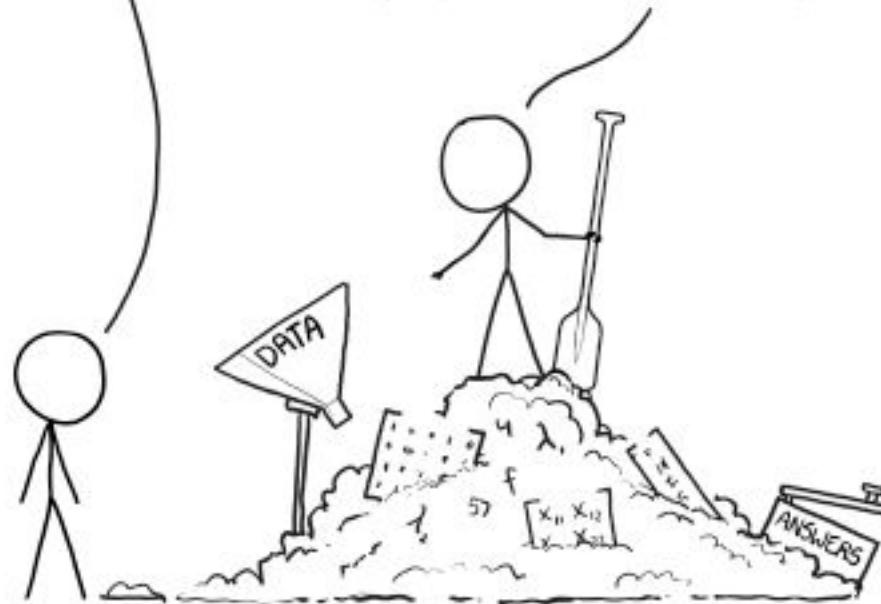
Deep learning = Deep neural networks =
neural networks

THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG
PILE OF LINEAR ALGEBRA, THEN COLLECT
THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

JUST STIR THE PILE UNTIL
THEY START LOOKING RIGHT.



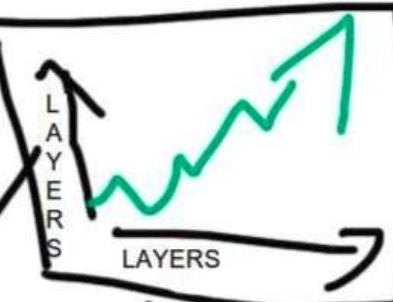
STATISTICAL LEARNING

Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high. Get some experts and minimize the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin



NEURAL NETWORKS

STACK
MORE
LAYERS



Hopefully this won't be all you get from this class

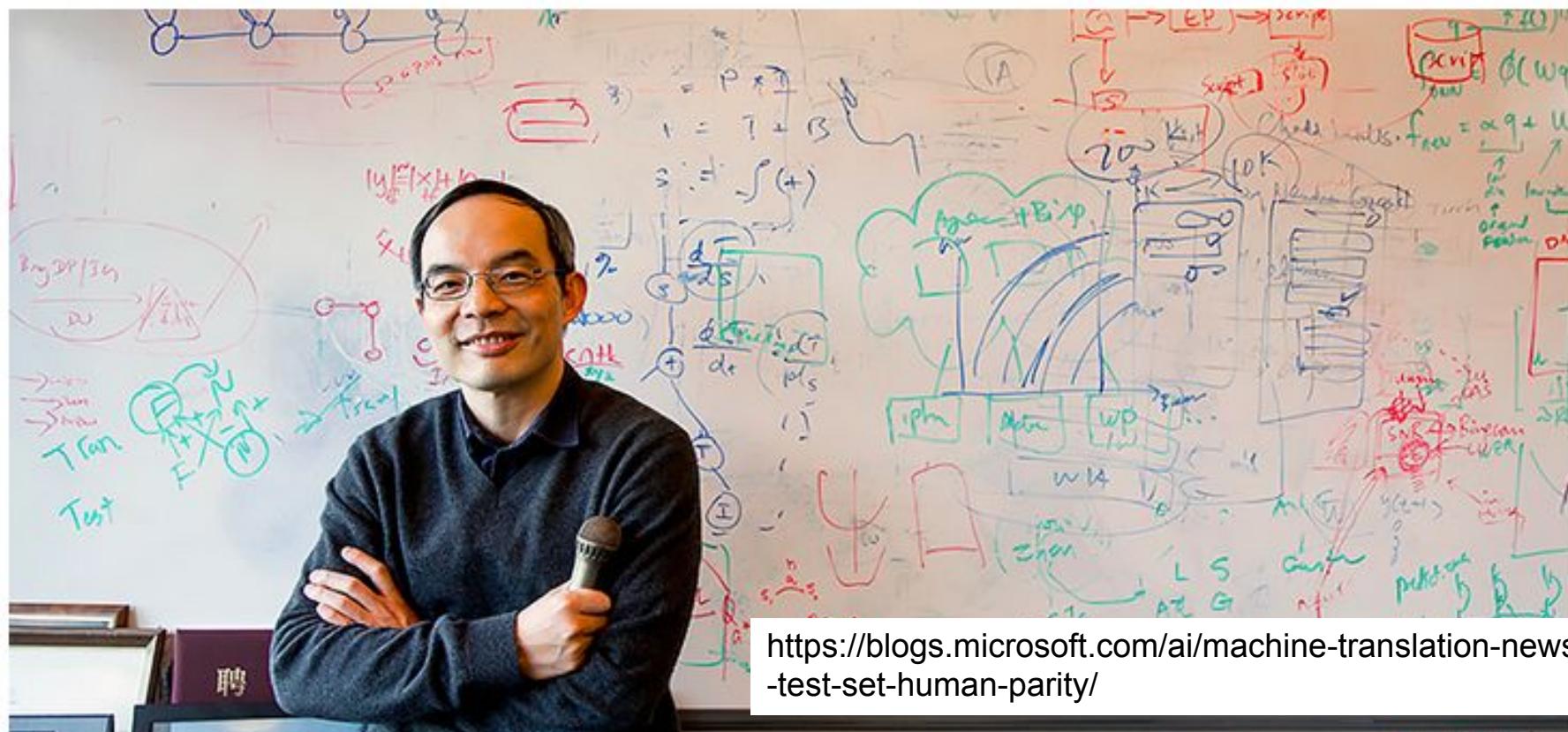
DNNs (Deep Neural Networks)

- Why deep learning?
- Greatly improved performance in ASR and other tasks (Computer Vision, Robotics, Machine Translation, NLP, etc.)
- Surpassed human performance in many tasks

| Task | Previous state-of-the-art | Deep learning (2012) | Deep learning (2017) |
|---------------------|---------------------------|----------------------|----------------------|
| TIMIT | 24.4% | 20.0% | 17.0% |
| Switchboard | 23.6% | 16.1% | 5.5% |
| Google voice search | 16.0% | 12.3% | 4.9% |

Microsoft reaches a historic milestone, using AI to match human performance in translating news from Chinese to English

Mar 14, 2018 | [Allison Linn](#)



<https://blogs.microsoft.com/ai/machine-translation-news-test-set-human-parity/>

Google's AlphaGo Defeats Chinese Go Master in Win for A.I.

点击查看本文中文版

By PAUL MOZUR MAY 23, 2017

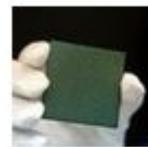


<https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html>

RELATED COVERAGE



A.I. Is
Replacing Us



China's
AI

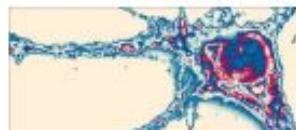


THE FUTURE
OF GO



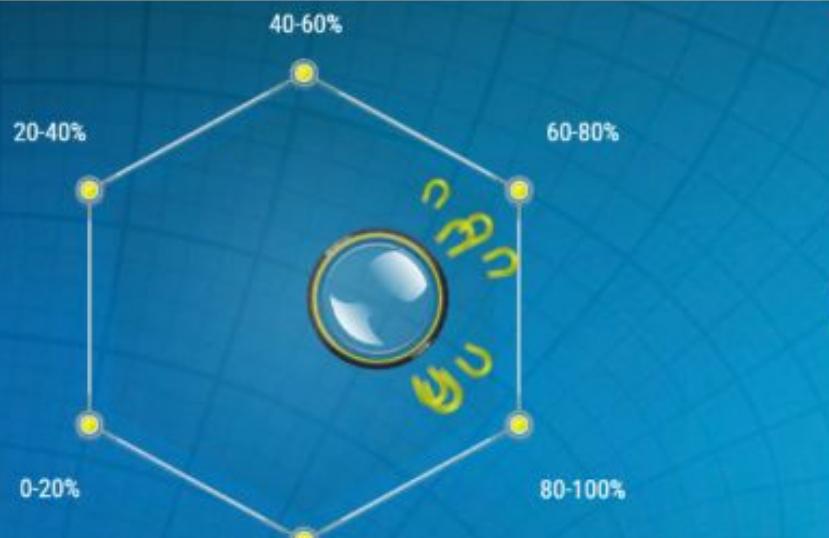
Master
vs.
Google

Artificial swarm intelligence diagnoses pneumonia better than individual computer or doctor



Hear from leading minds and find inspiration for your own research

by Fan Liu — September 27, 2018 [Comment](#) 0



Courtesy of Unanimous AI

[Bangkok to Tokyo](#)

THB 4,030

[BOOK NOW](#)

[Bangkok to Hangzhou](#)

THB 4,030

[BOOK NOW](#)

[ezoic](#)

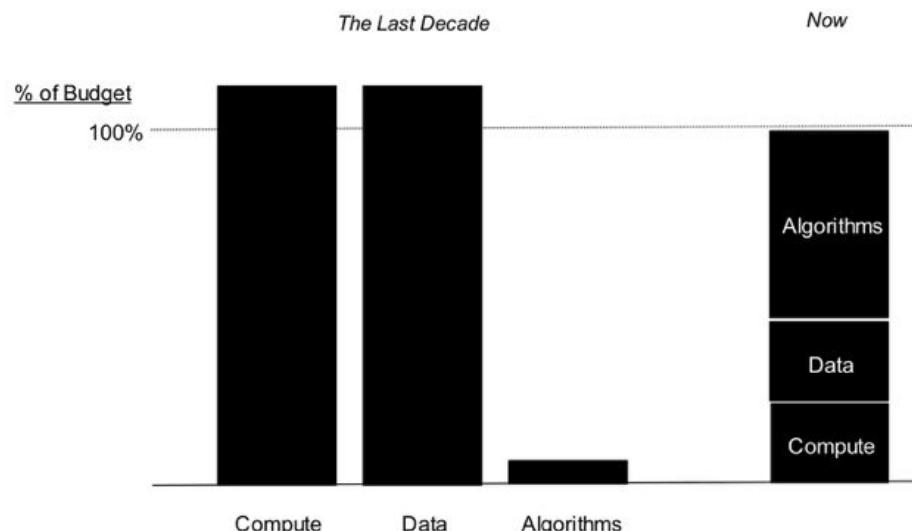
[report this](#)

[Popular Posts](#)

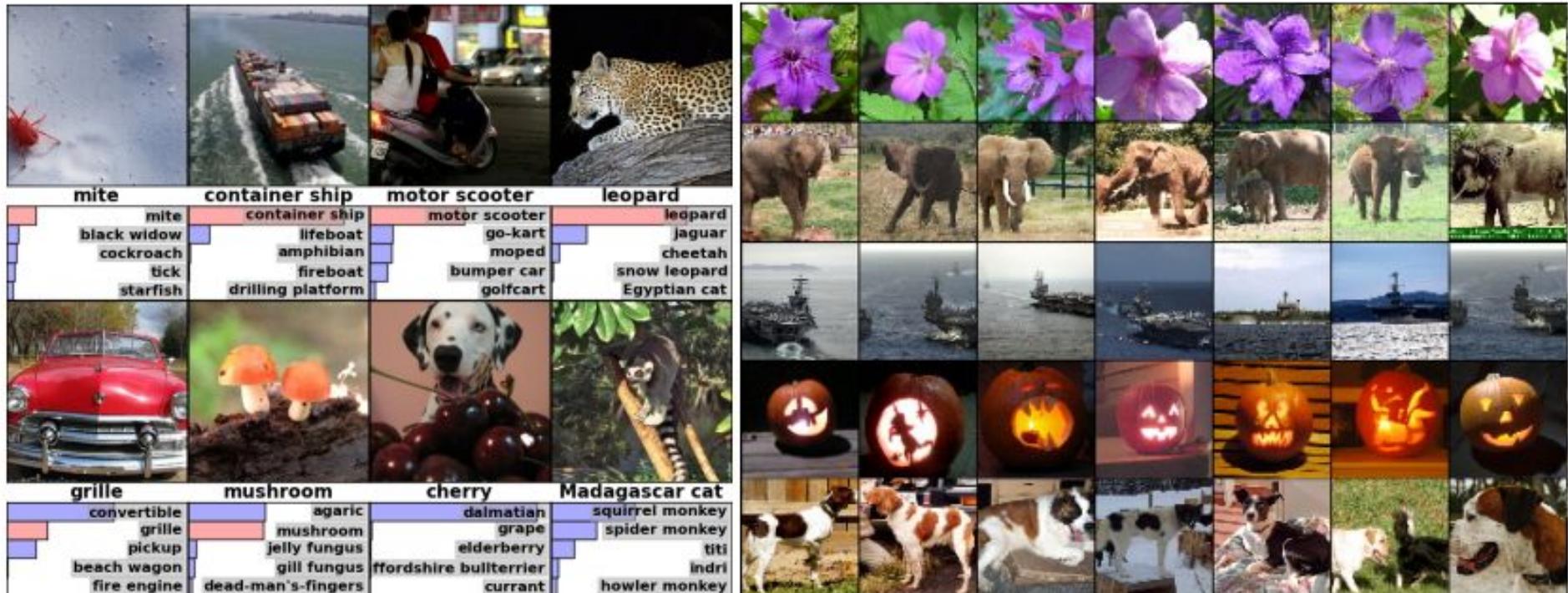
<https://www.stanforddaily.com/2018/09/27/artificial-swarm-intelligence-diagnoses-pneumonia-better-than-individual-computer-or-doctor/>

Why now

- Neural Networks has been around since 1990s
- **Big data** – DNN can take advantage of large amounts of data better than other models
- **GPU** – Enable training bigger models possible
- **Deep** – Easier to avoid bad local minima when the model is large



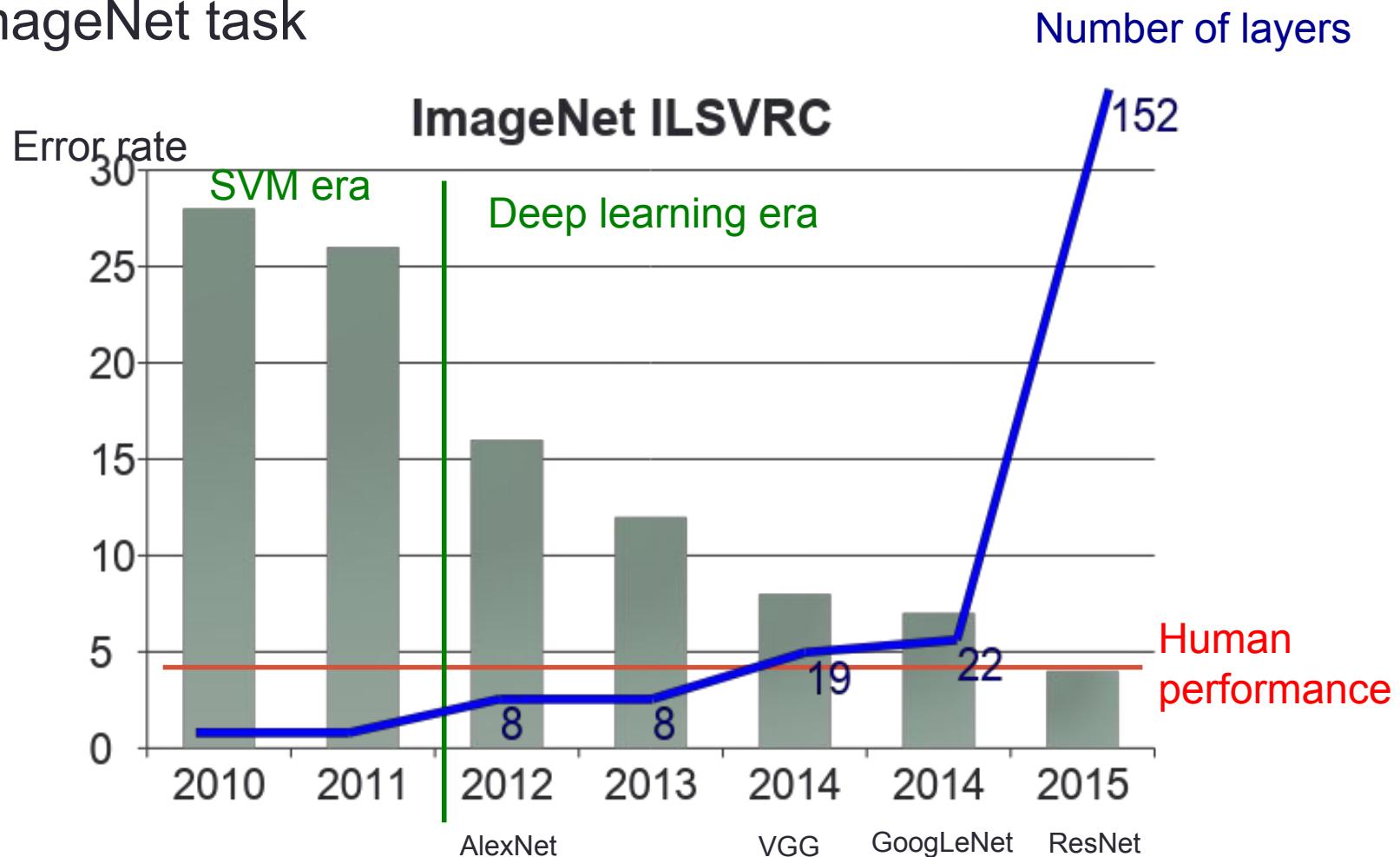
ImageNet - Object classification



Alex, Krizhevsky, Imagenet classification with deep convolutional neural networks, 2012

Wider and deeper networks

- ImageNet task





Statistics > Machine Learning

Dynamical Isometry and a Mean Field Theory of CNNs: How to Train 10,000-Layer Vanilla Convolutional Neural Networks

Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel S. Schoenholz, Jeffrey Pennington

(Submitted on 14 Jun 2018)

In recent years, state-of-the-art methods in computer vision have utilized increasingly deep convolutional neural network architectures (CNNs), with some of the most successful models employing hundreds or even thousands of layers. A variety of pathologies such as vanishing/exploding gradients make training such deep networks challenging. While residual connections and batch normalization do enable training at these depths, it has remained unclear whether such specialized architecture designs are truly necessary to train deep CNNs. In this work, we demonstrate that it is possible to train vanilla CNNs with ten thousand layers or more simply by using an appropriate initialization scheme. We derive this initialization scheme theoretically by developing a mean field theory for signal propagation and by characterizing the conditions for dynamical isometry, the equilibration of singular values of the input-output Jacobian matrix. These conditions require that the convolution operator be an orthogonal transformation in the sense that it is norm-preserving. We present an algorithm for generating such random initial orthogonal convolution kernels and demonstrate empirically that they enable efficient training of extremely deep architectures.

Comments: ICML 2018 Conference Proceedings

Subjects: Machine Learning (stat.ML); Machine Learning (cs.LG)

Cite as: arXiv:1806.05393 [stat.ML]

(or arXiv:1806.05393v1 [stat.ML] for this version)

Submission history

From: Samuel Schoenholz [view email]

[v1] Thu, 14 Jun 2018 07:04:15 GMT (6734kb,D)

Which authors of this paper are endorsers? | Disable MathJax (What is MathJax?)

Link back to: arXiv, form interface, contact.

Download:

- PDF
- Other formats

(license)

Current browse context:

stat.ML

< prev | next >

new | recent | 1806

Change to browse by:

cs

cs.LG

stat

References & Citations

- NASA ADS

Bookmark (what is this?)

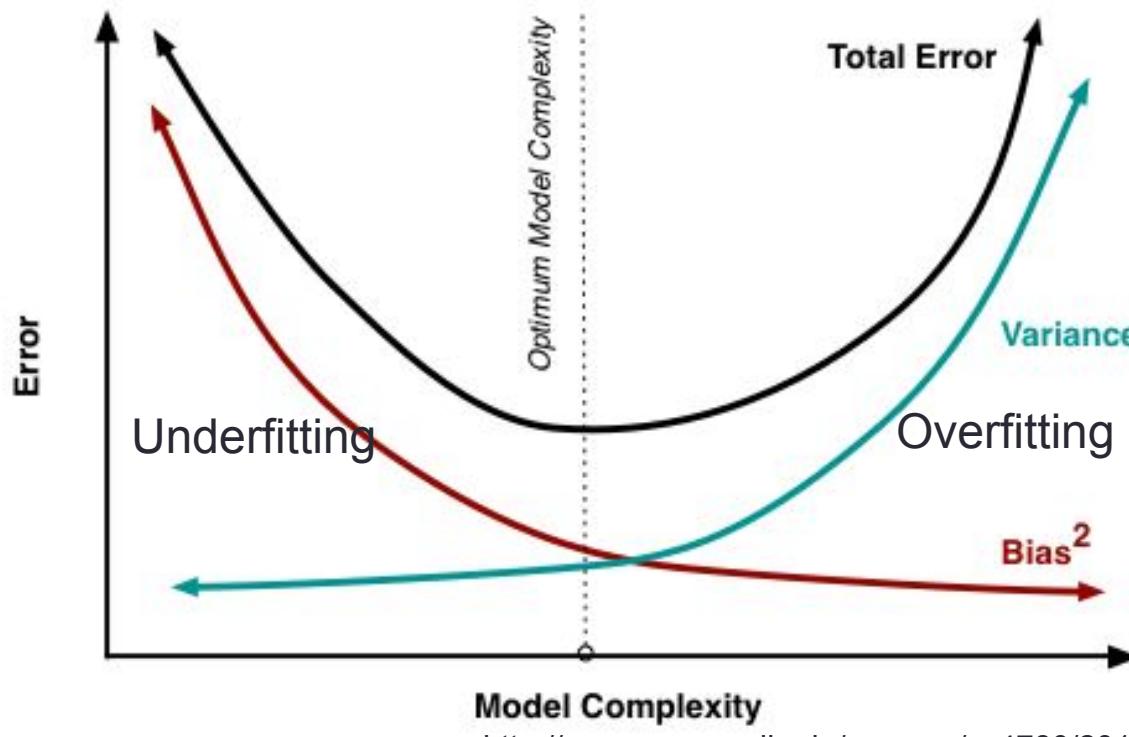


ScienceWISE

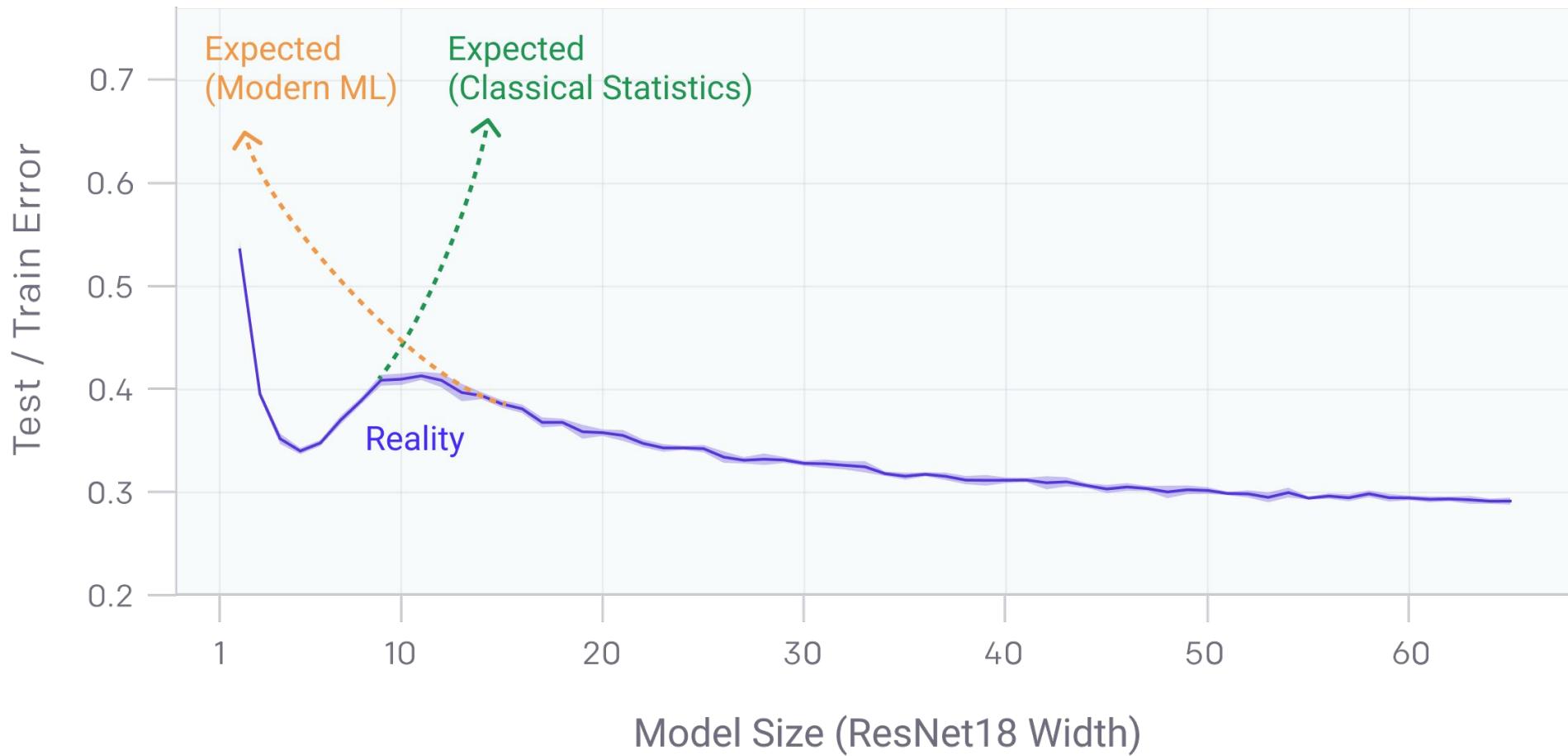


Bias-Variance Underfitting-Overfitting

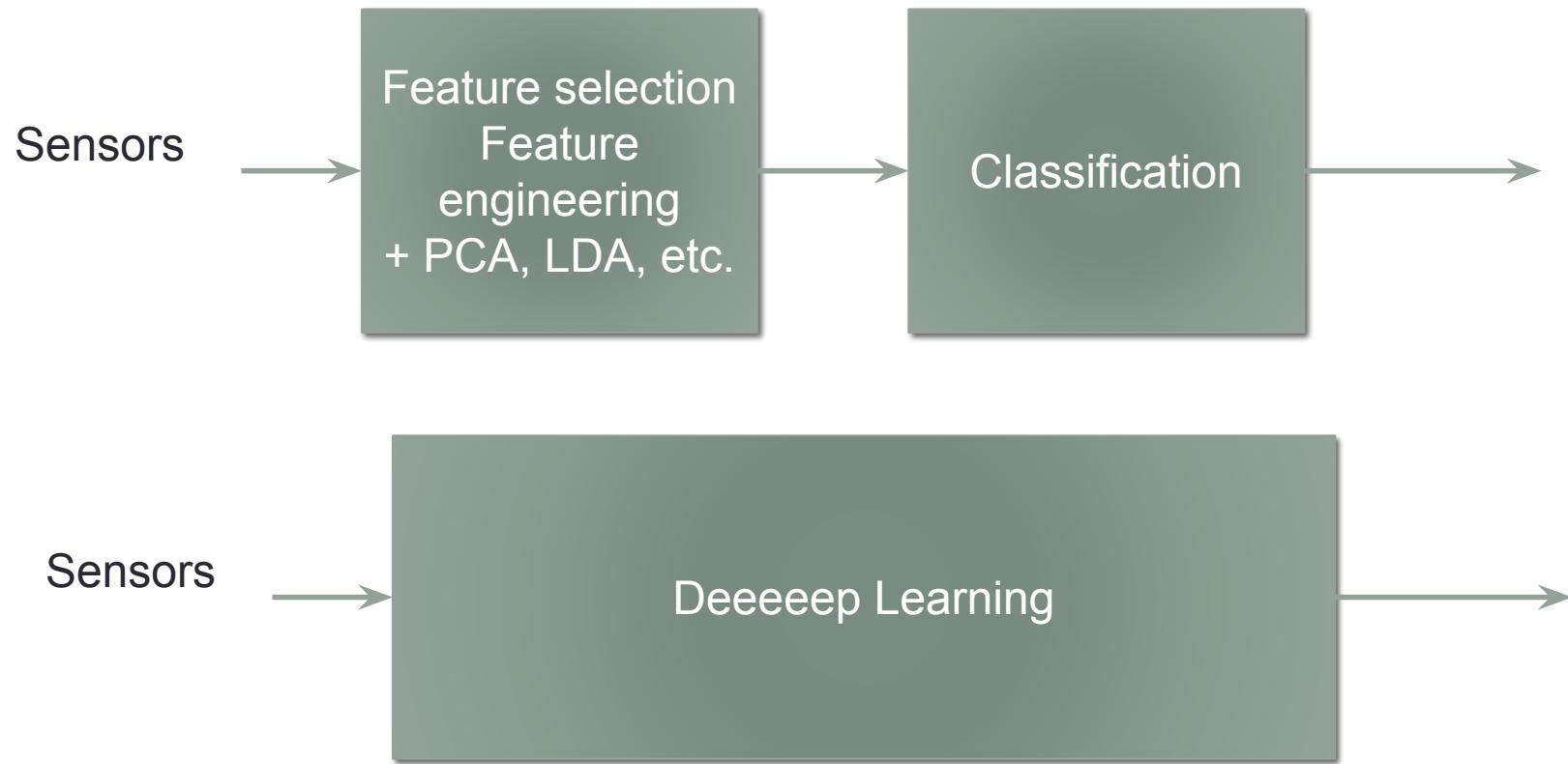
- Usually if you try to reduce the bias of your model, the variance will increase, and vice versa.
- Called the bias-variance trade-off



The double descent problem



Traditional VS Deep learning

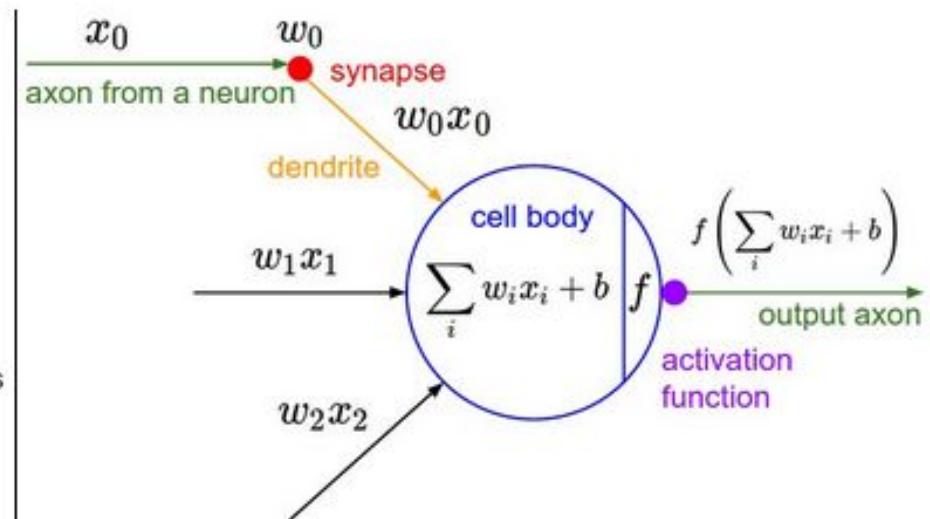
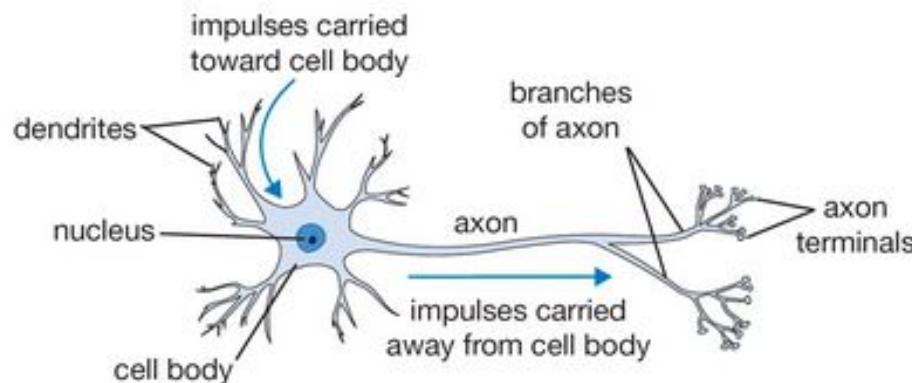


Neural networks

- Fully connected networks
 - Neuron
 - Non-linearity
 - Softmax layer
- DNN training
 - Loss function and regularization
 - SGD and backprop
 - Learning rate
 - Overfitting – dropout, batchnorm
- Demos
 - Tensorflow, Keras
- CNN, RNN, LSTM, GRU <- Next class

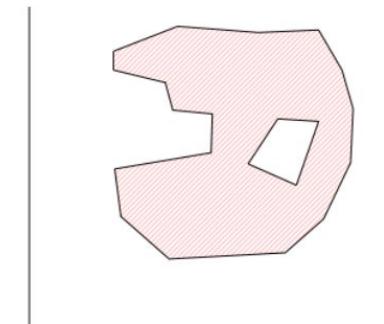
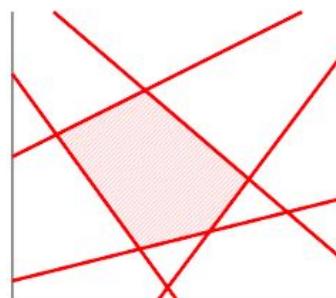
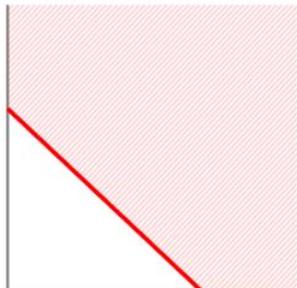
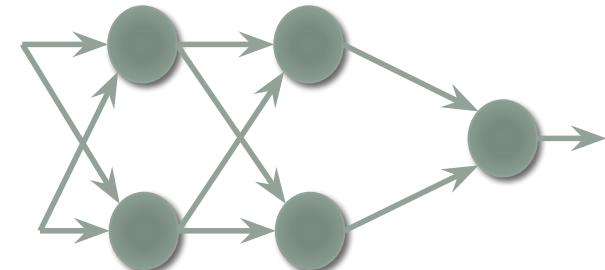
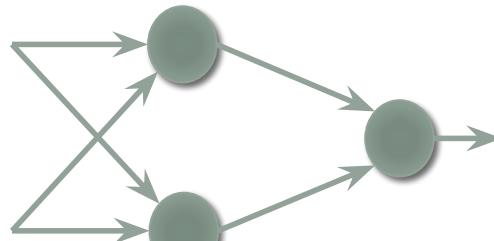
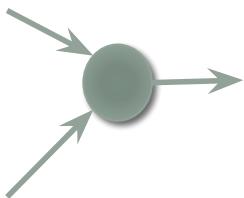
Fully connected networks

- Many names: feed forward networks or deep neural networks or multilayer perceptron or artificial neural networks
- Composed of multiple neurons



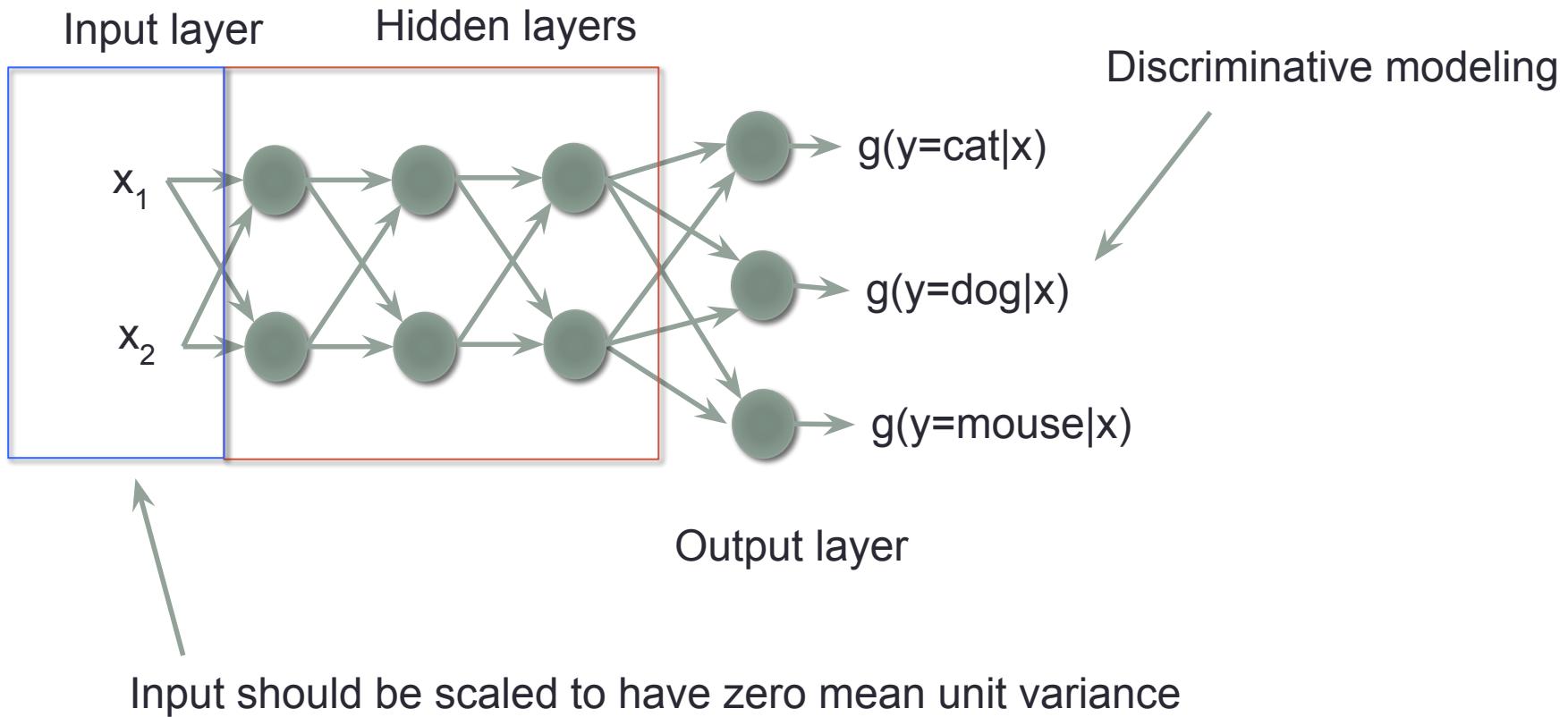
Combining neurons

- Each neuron splits the feature space with a hyperplane
- Stacking neuron creates more complicated decision boundaries
- More powerful but prone to overfitting



Terminology

Deep in Deep neural networks means many hidden layers



Matrices

- Inputs

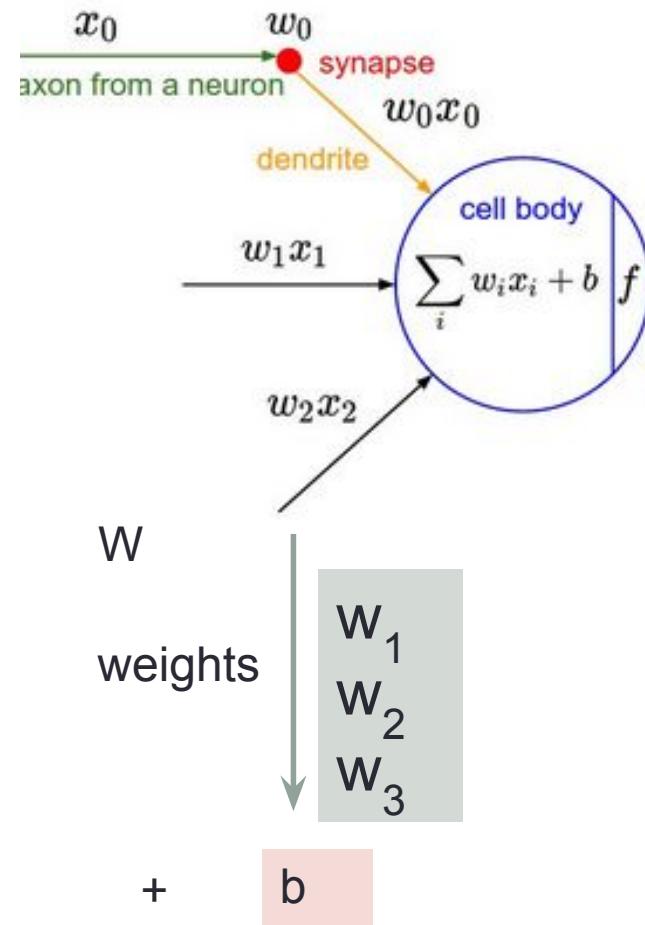
features

$$\downarrow \quad \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix}$$

X

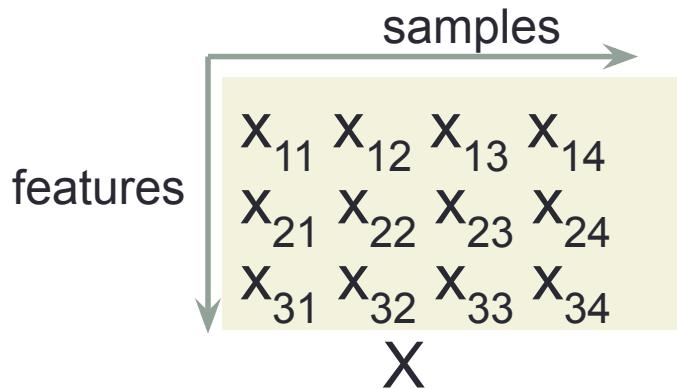
$$W^T X + b$$

$$\begin{matrix} W_1 & W_2 & W_3 \\ x_1 \\ x_2 \\ x_3 \end{matrix}$$



Matrices

- Inputs



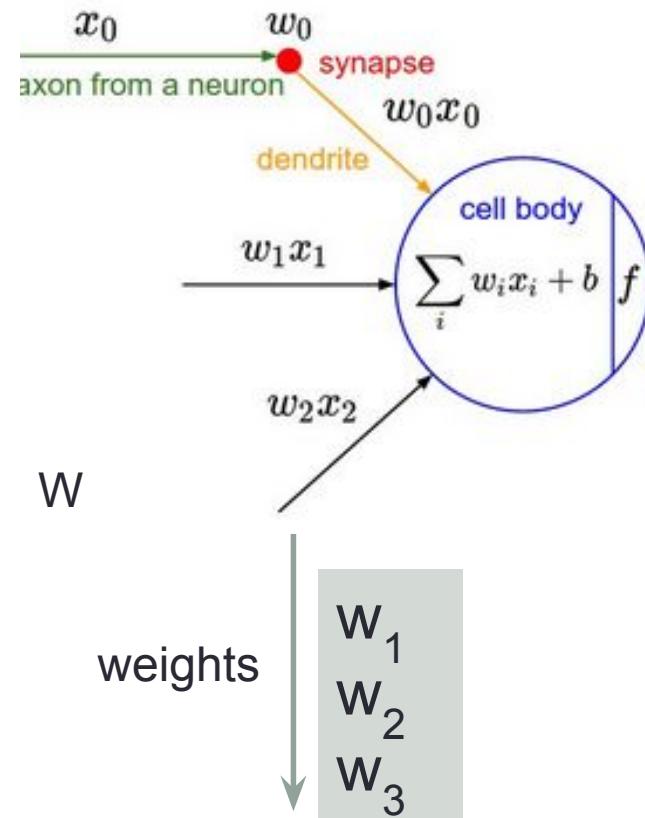
$$W^T X + b$$

$$\begin{matrix} W_1 & W_2 & W_3 \end{matrix}$$

$$\begin{matrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{matrix}$$

+

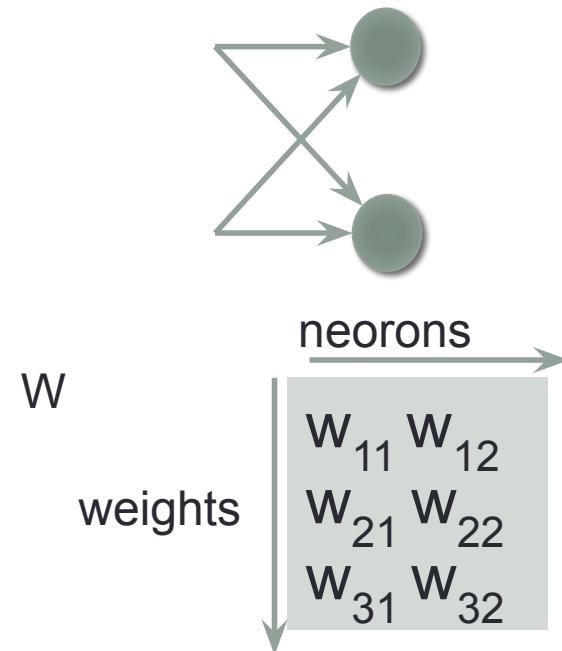
$$b$$



Matrices

- Inputs

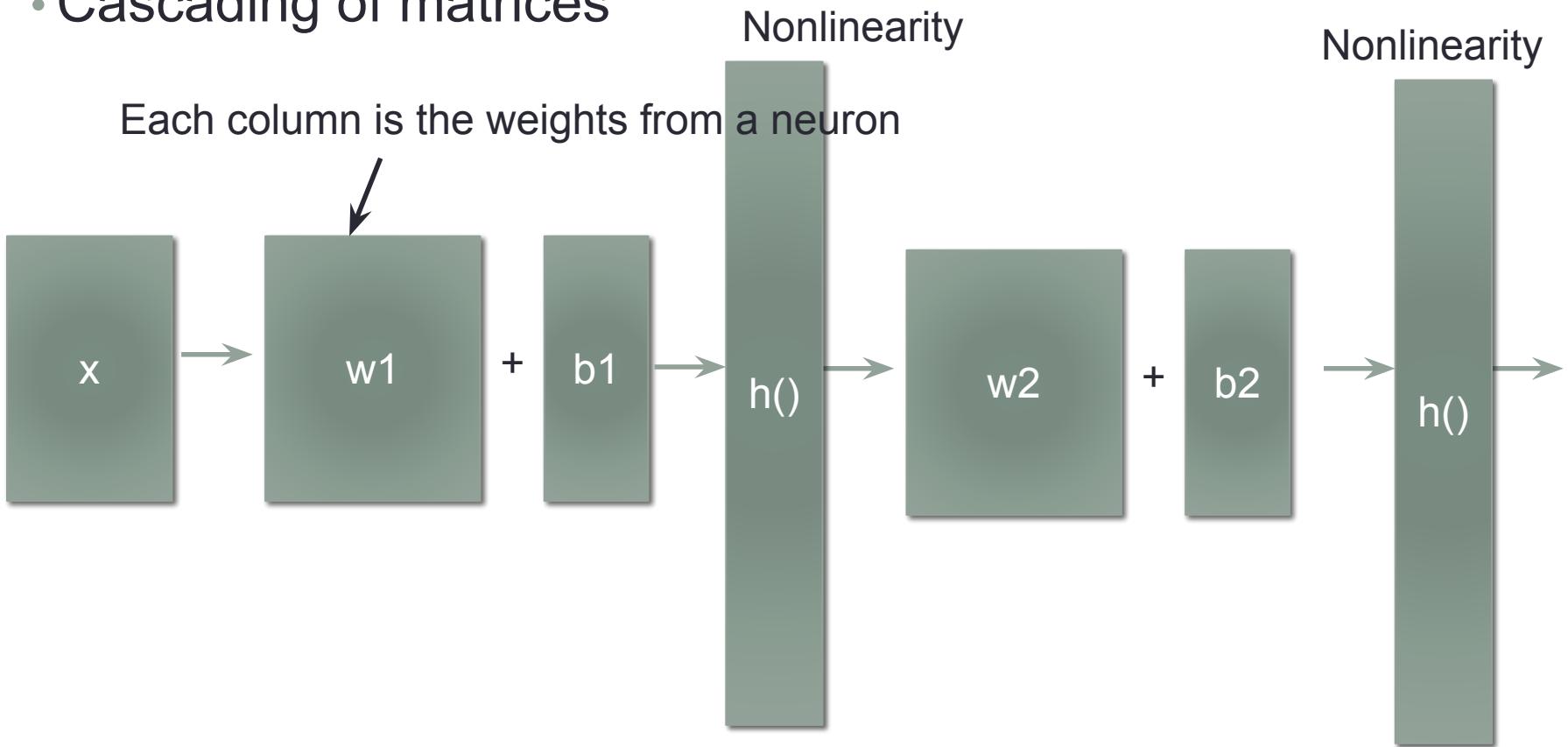
| | samples | | | |
|----------|----------|----------|----------|----------|
| features | x_{11} | x_{12} | x_{13} | x_{14} |
| | x_{21} | x_{22} | x_{23} | x_{24} |
| | x_{31} | x_{32} | x_{33} | x_{34} |
| X | | | | |



$$w^T x + b = \begin{matrix} w_{11} & w_{21} & w_{31} \\ w_{21} & w_{22} & w_{23} \end{matrix} \begin{matrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{matrix} + \begin{matrix} b_1 \\ b_2 \end{matrix}$$

More linear algebra

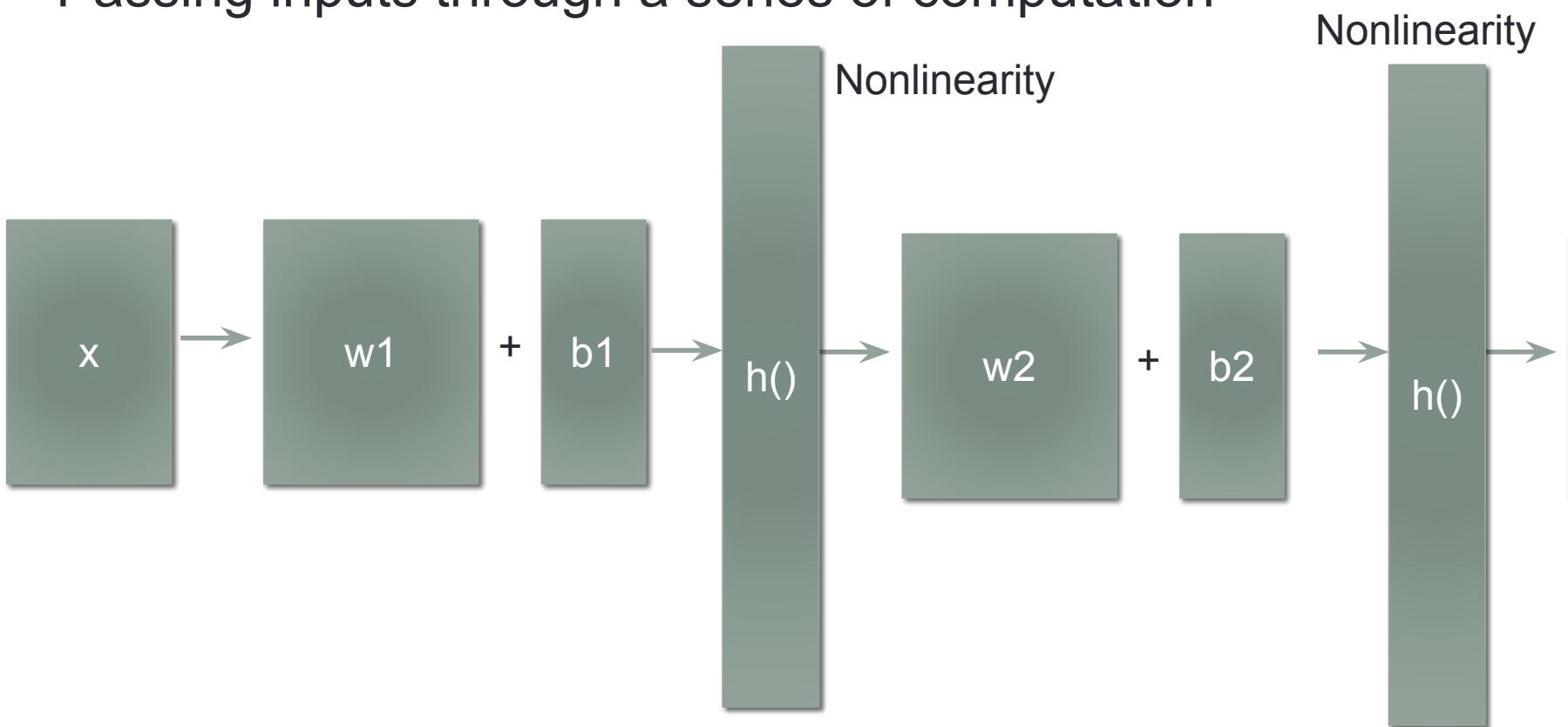
- Cascading of matrices



$$h(W_2^T h(W_1^T X + b_1) + b_2)$$

Computation graph

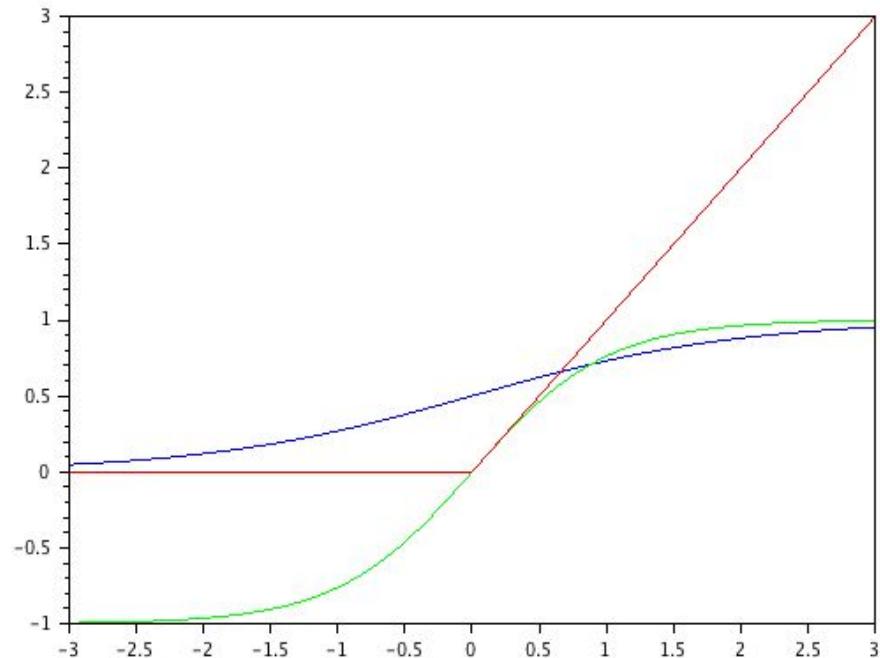
- Passing inputs through a series of computation



$$h(W_2^T h(W_1^T X + b_1) + b_2)$$

Non-linearity

- The Non-linearity is important in order to stack neurons
- Sigmoid or logistic function
- \tanh
- Rectified Linear Unit (ReLU)
- Swish (new!)
- Most popular is ReLU and its variants (Fast to train, and more stable)



Non-linearity

- Sigmoid

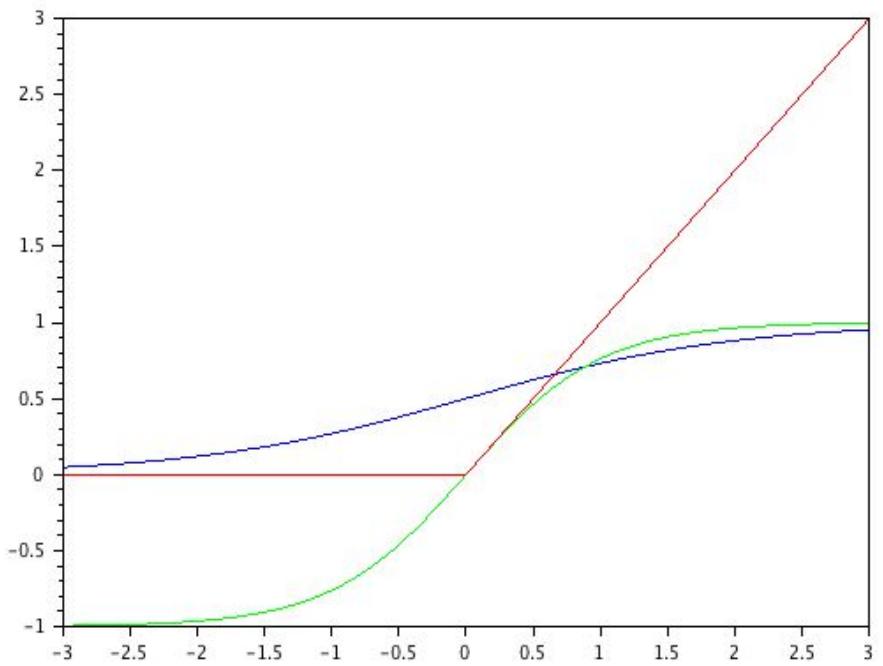
$$\frac{1}{1 + e^{-x}}$$

- tanh

$$\tanh(x)$$

- Rectified Linear Unit (ReLU)

$$\max(0, x)$$



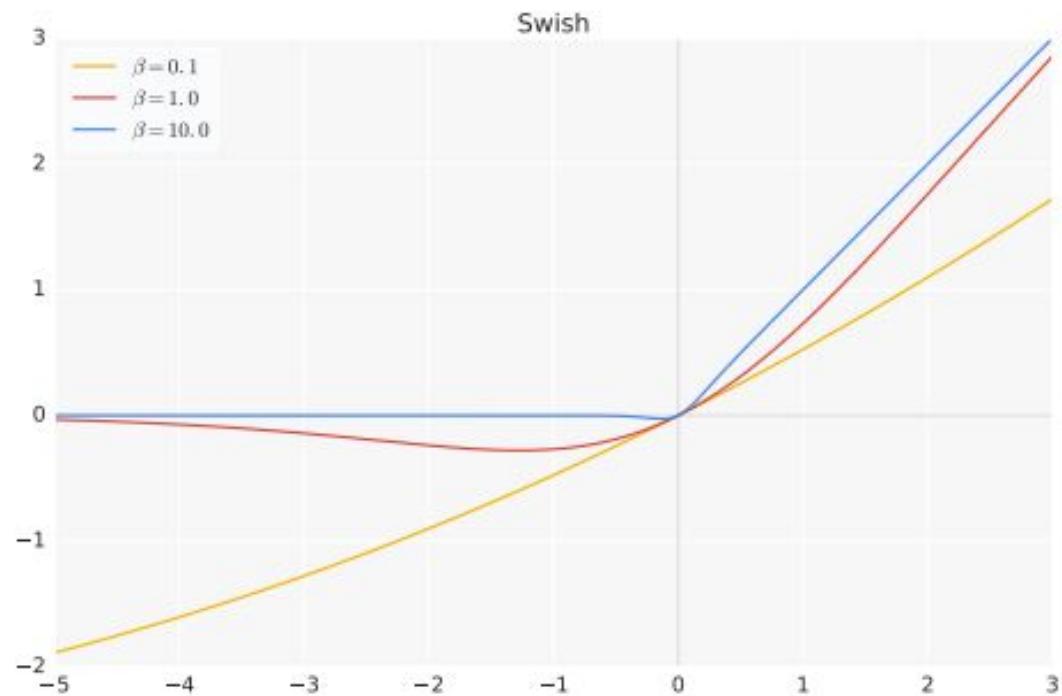
Swish

Found through reinforcement learning to be the best
general non-linearity

$$x \cdot \text{sig}(\beta x)$$

sig refers to a sigmoid function
Beta is a learnable parameter
or can be set to 1 for slightly
worse performance

Beta \rightarrow inf, then Swish \rightarrow ReLu



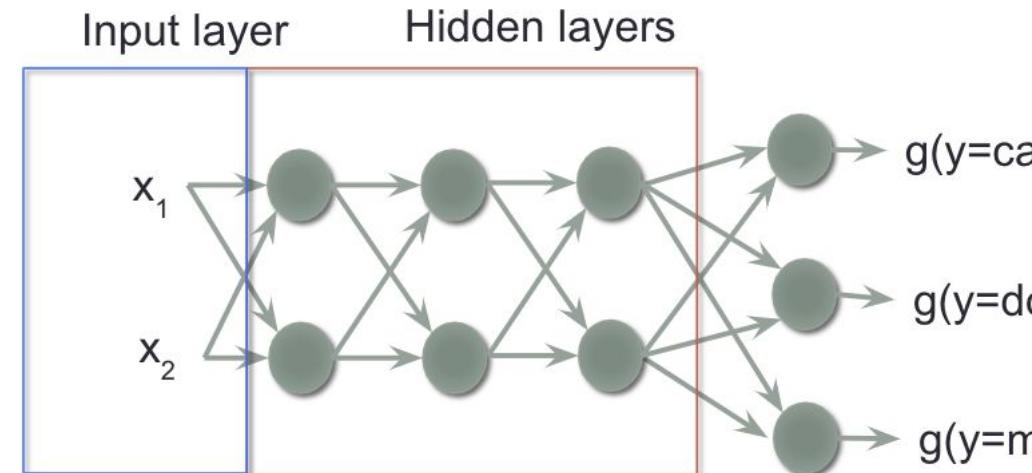
[Searching for Activation Functions - arXiv](#)

Proven theoretically to be optimal

[Expectation propagation: a probabilistic view of Deep Feed Forward Networks](#)

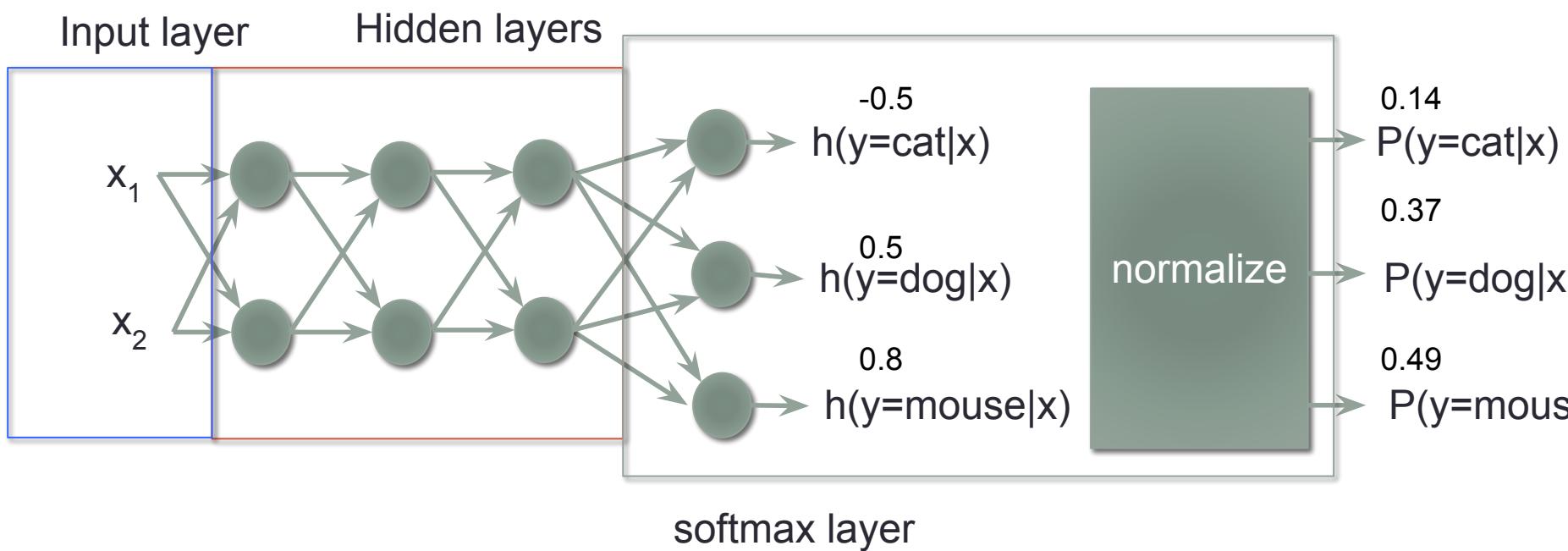
Output layer – Softmax layer

- We usually want the output to mimic a probability function ($0 \leq P \leq 1$, sums to 1)
- Current setup has no such constraint
- The current output should have highest value for the correct class.
 - Value can be positive or negative number
- Takes the exponent
- Add a normalization



Softmax layer

$$P(y = j|x) = \frac{e^{h(y=j|x)}}{\sum_y e^{h(y|x)}}$$

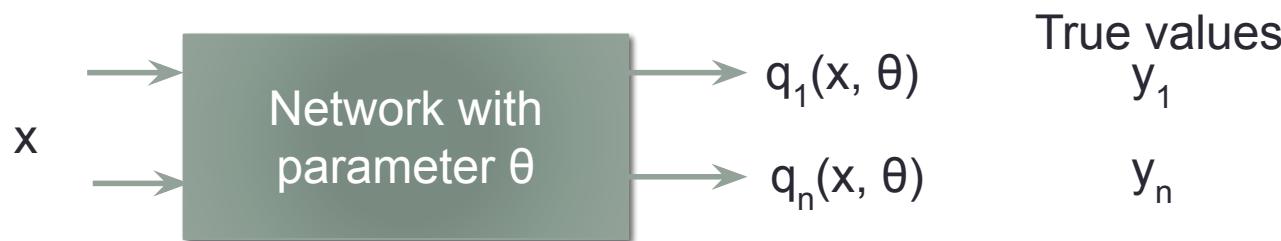


Neural networks

- Fully connected networks
 - Neuron
 - Non-linearity
 - Softmax layer
- DNN training
 - Loss function and regularization
 - SGD and backprop
 - Learning rate
 - Overfitting – dropout, batchnorm
- Demos
 - Tensorflow, Keras
- CNN
- RNN, LSTM, GRU <- Next class

Objective function (Loss function)

- Can be any function that summarizes the performance into a single number
- Cross entropy
- Sum of squared errors



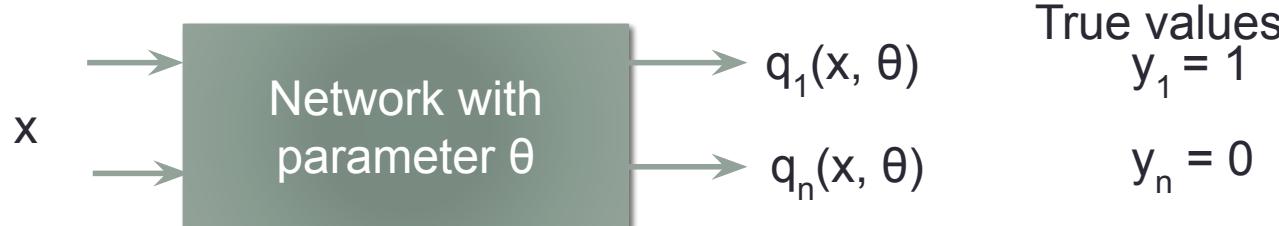
Cross entropy loss

- Used for softmax outputs (probabilities), or classification tasks

$$L = -\sum_n y_n \log q_n(x, \theta)$$

- Where y_n is 1 if data x comes from class n
0 otherwise

- L only has the term from the correct class
- L is non negative with highest value when the output matches the true values, a “loss” function

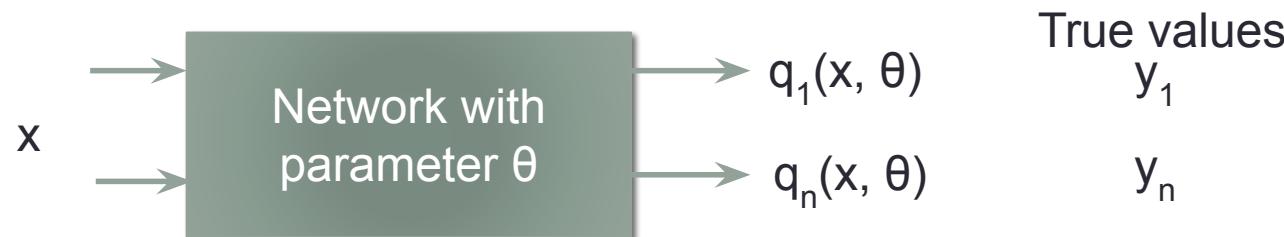


Sum of squared errors (MSE)

- Used for any real valued outputs such as regression

$$L = \frac{1}{2} \sum_n (y_n - q_n(x, \theta))^2$$

- Non negative, the better the lower the loss



Other losses (later)

Squared EMD loss

Classification tasks with ordering (dog -> elephant is better than dog -> car)

Adversarial loss

Learning distribution or when no single answer exist

Perceptual loss

Wants output to have certain properties (Computer vision)

Smoothed L1

$L1 + L2$

Regularization

There are two main approaches to regularize neural networks

- Explicit regularization
 - Deals with the loss function
- Implicit regularization
 - Deals with the network

Regularization in one slide

- What?
 - Regularization is a method to lower the model variance (and thereby increasing the model bias)
- Why?
 - Gives more generalizability (lower variance)
 - Better for lower amounts of data (reduce overfitting)
- How?
 - Introducing regularizing terms in the original loss function
 - Can be anything that make sense

$$\mathbf{w}^T \mathbf{w} + C \sum \epsilon_i$$

MAP estimate is MLE with regularization (the prior term)

Famous types of regularization

- L1 regularization: Regularizing term is a sum
 - $\mathbf{w}^T \mathbf{w} + C \sum \varepsilon_i$
- L2 regularization: Regularizing term is a sum of squares
 - $\mathbf{w}^T \mathbf{w} + C \sum \varepsilon_i^2$

| L2 regularization | L1 regularization |
|--|---|
| Computational efficient due to having analytical solutions | Computational inefficient on non-sparse cases |
| Non-sparse outputs | Sparse outputs |
| No feature selection | Built-in feature selection |

Regularization in neural networks

L2

- We want to improve generalization somehow.
- Observation, models are better when the weights are spread out (no peaky weights).
 - Try to use every part of the model.
- Add a cost if we put some value to the weights
- Regularized loss = Original loss + $0.5 C \sum w^2$
 - we sum the square of weights of the whole model
 - 0.5 is for prettiness when we take derivative
 - C is a hyperparameter weighting the regularization term

Regularization in neural networks

- We want to improve generalization somehow.
- Observation, models behave better when we force the weights to be sparse.
 - Sparse means many weights are zero or close to zero
 - Force the model to focus on only important parts
 - Less prone to noise
- Add a cost if we put some value to the weights
- Regularized loss = Original loss + $0.5 C \sum |w|$

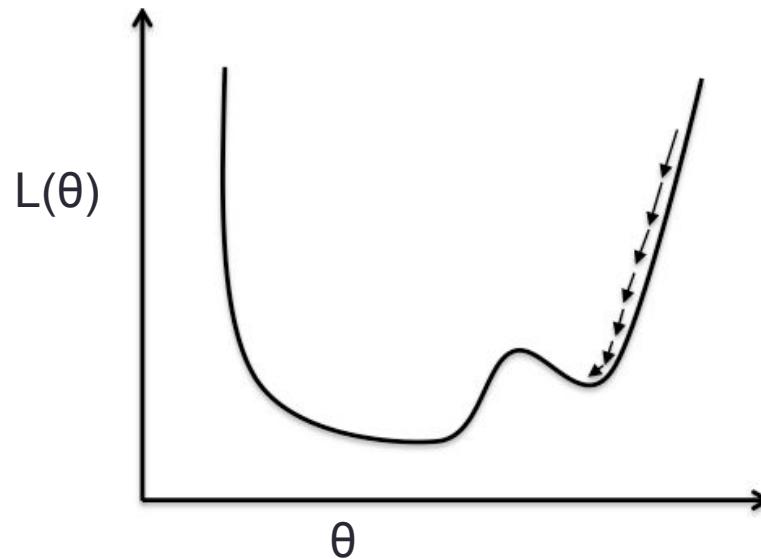
- we sum the absolute weights of the whole model
- 0.5 is for prettiness when we take derivative
- C is a hyperparameter weighting the regularization term

L1 L2 regularization notes

- Can use both at the same time
 - People claim L2 is superior (called weight decay by some community)
- Mostly ignored nowadays
- Other regularization methods exist (we will go over these later)

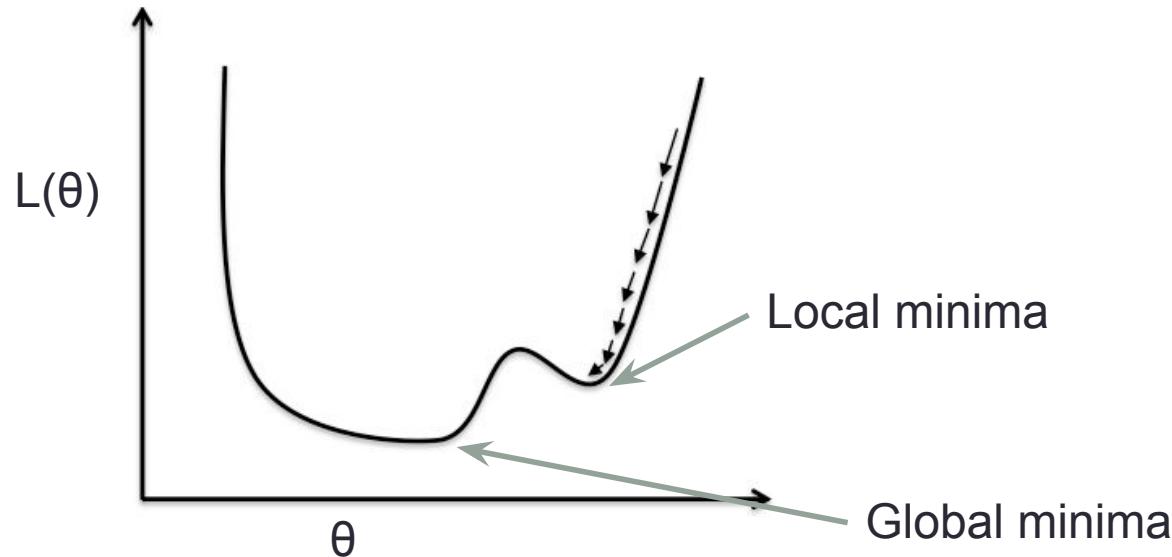
Minimization using gradient descent

- We want to minimize L with respect to θ (weights and biases)
 - Differentiate with respect to θ
 - Gradients passes through the network by Back Propagation



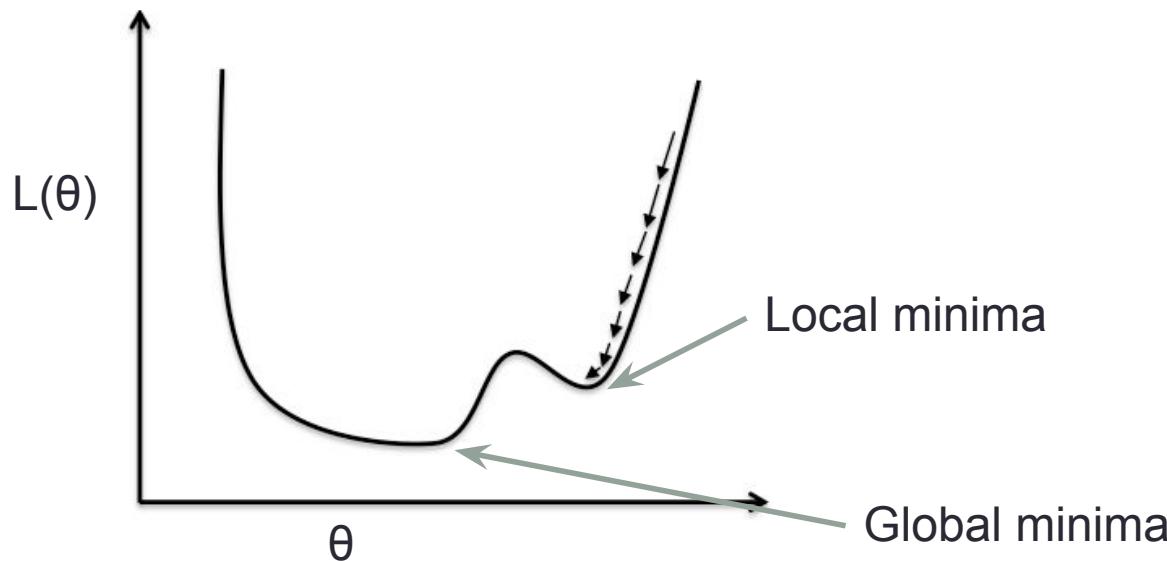
Deep vs Shallow

- The loss function of neural network is non-convex (and non-concave)
 - Local minimas can be avoided with convexity
 - Linear regression, SVM are convex optimization
 - Convexity gives easier training
 - Does not imply anything about the generalization of the model
 - The loss is optimized by the training set



Deep vs Shallow

- If deep, most local minimas are the global minimas!
 - Always a way to lower the loss in the network with millions of parameters
 - Enough parameters to remember every training examples
 - Does not imply anything about generalization



Differentiating a neural network model

- We want to minimize loss by gradient descent
- A model is very complex and have many layers! How do we differentiate this!!?



Back propagation

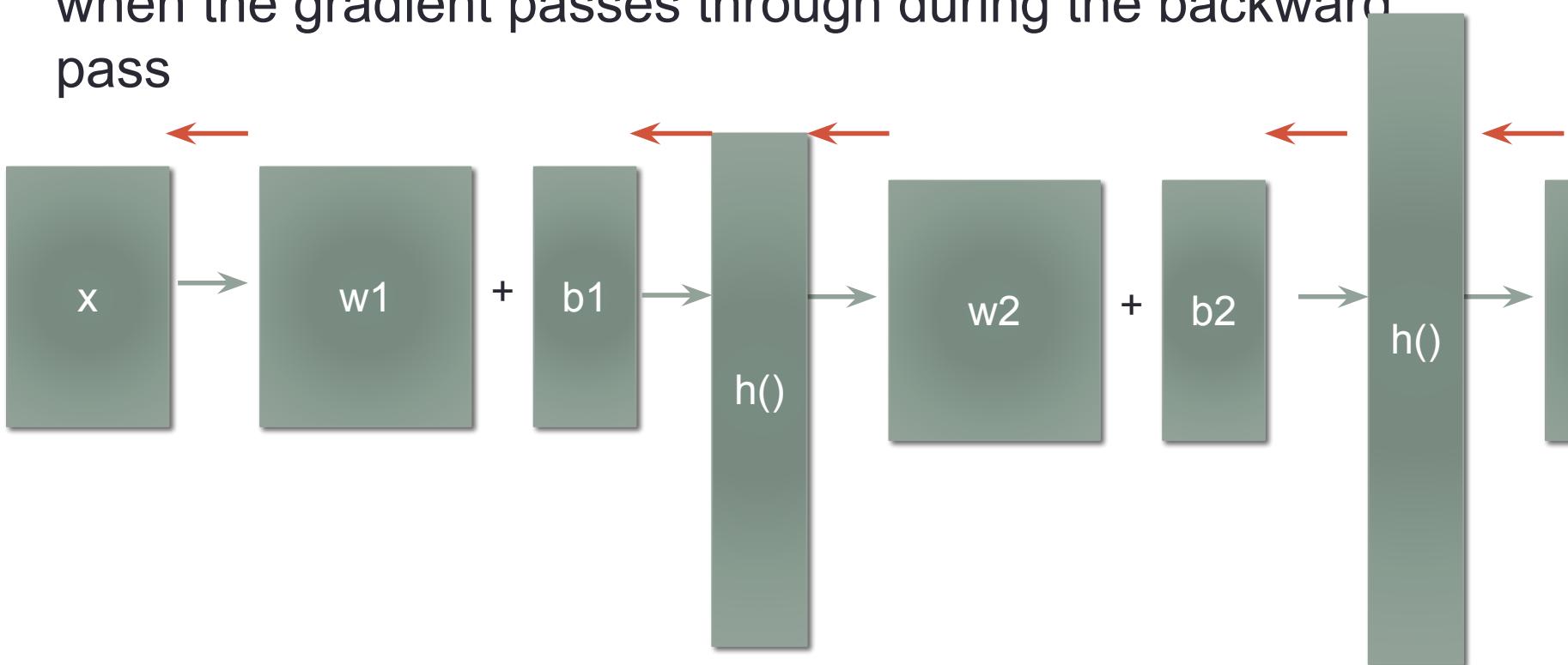
- Forward pass
 - Pass the value of the input until the end of the network
- Backward pass
 - Compute the gradient starting from the end and passing down gradients using chain rule

Back propagation

- Regularization terms only appears at the particular weights when doing the derivative
- What about cross entropy?

Backprop and computation graph

- We can also define what happens to a computing graph when the gradient passes through during the backward pass



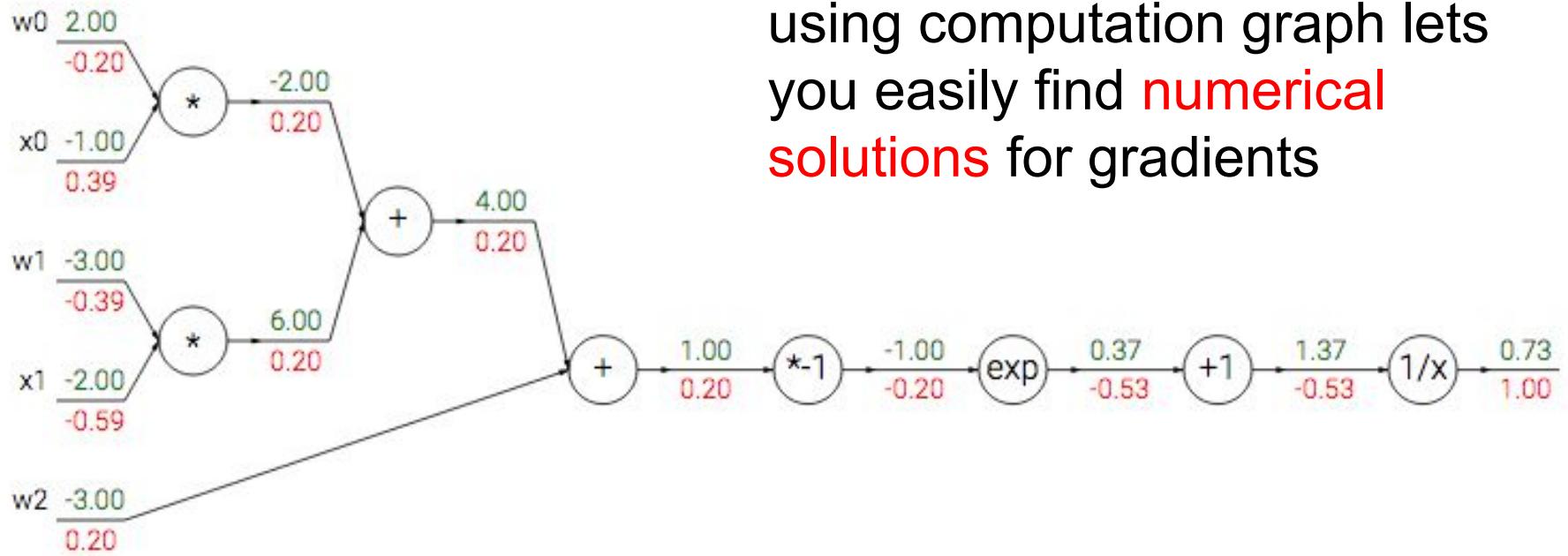
This lets us to build any neural networks without having to redo all the derivation as long as we define a forward and backward computation for the block.

Numerical gradient flow

- Let's find the gradient of

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

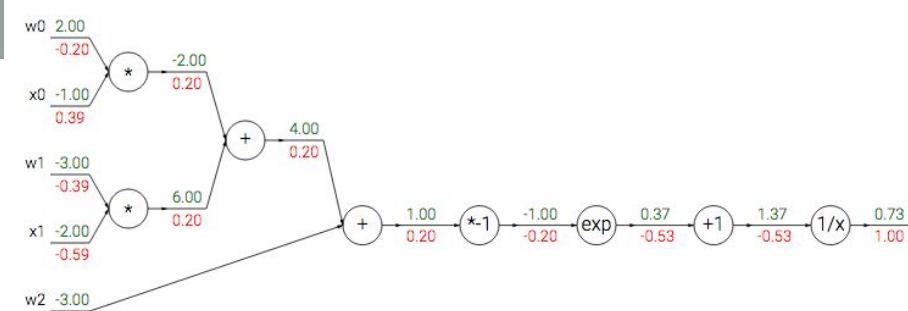
Computation graph



Doing backprop (chain rule) by using computation graph lets you easily find **numerical solutions** for gradients

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

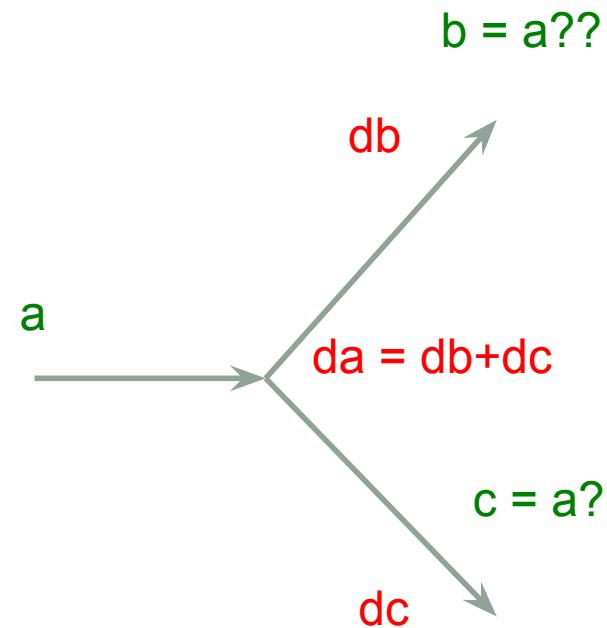
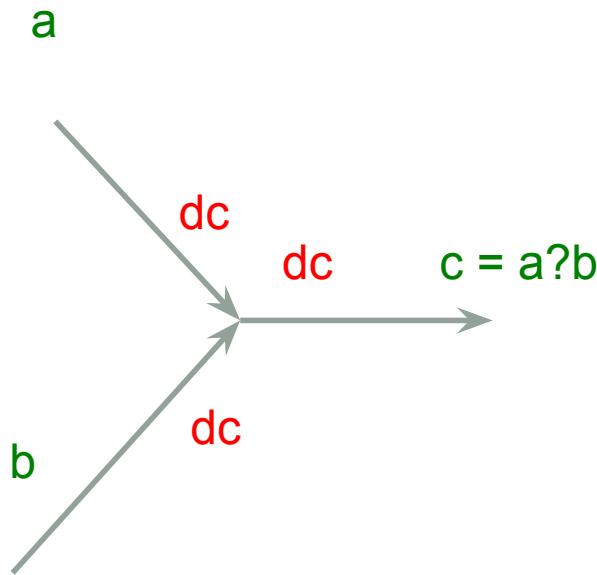
- $w = [0, -3, -3]$
- $x = [-1, -2]$
- $t_0 = w[0]*x[0]$
- $t_1 = w[1]*x[1]$
- $t_{01} = t_0 + t_1$
- $t_{012} = t_{01} + w[2]$
- $n_t = -t_{012}$
- $e = \exp(n_t)$
- $\text{denom} = e + 1$
- $f = 1/\text{denom}$



- $\text{ddenom} = -1/\text{denom}/\text{denom}$
- $de = 1 * \text{ddenom}$
- $dn_t = \exp(n_t) * de$
- $dt_{012} = -dn_t$
- $dw_2 = 1 * dt_{012}$
- $dt_{01} = 1 * dt_{012}$
- $dt_0 = 1 * dt_{01}$
- $dt_1 = 1 * dt_{01}$
- $dw_1 = x[1]dt_1$
- $dx_1 = w[1]dt_1$
- $dw_0 = x[0]dt_0$; $dx_0 = w[0]dt_0$

Perform backward pass in reverse order. No need to explicitly find overall derivative

Gradient flow at forks



Forward and backward pass acts differently at forks

Gradient and non-linearities

We can now talk about how good a non-linearity is by looking at the gradients.

We want

Something that is **differentiable numerically**

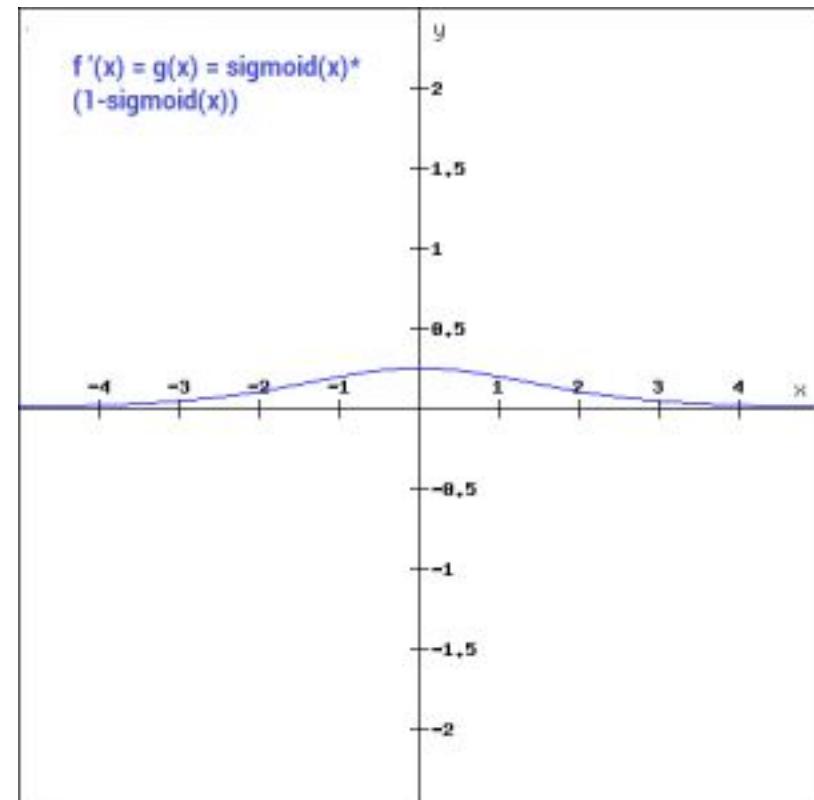
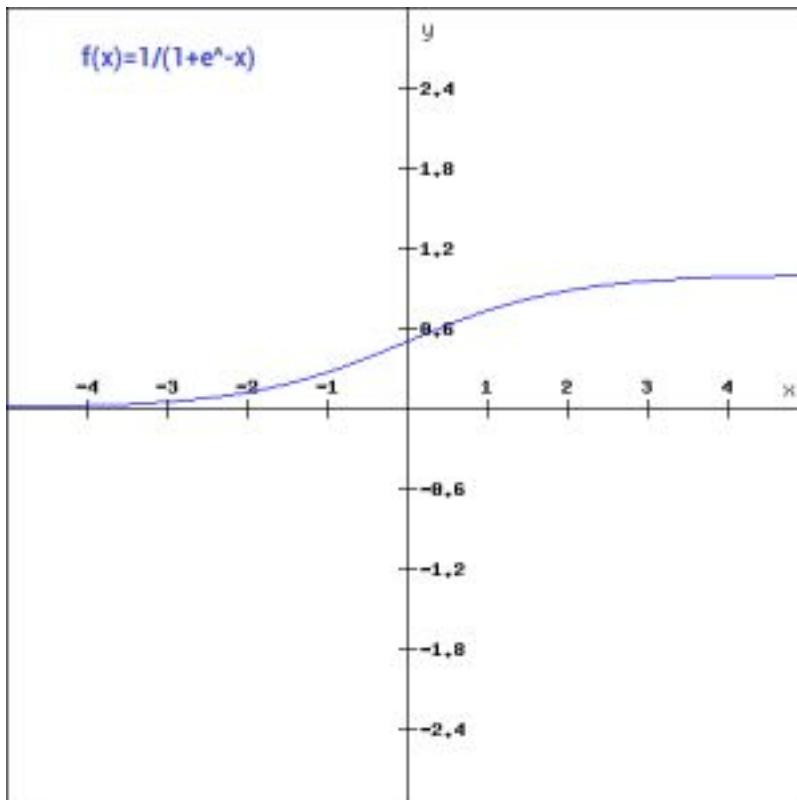
Cheap to compute

Big gradients at every point

Notes on non-linearity

- Sigmoid

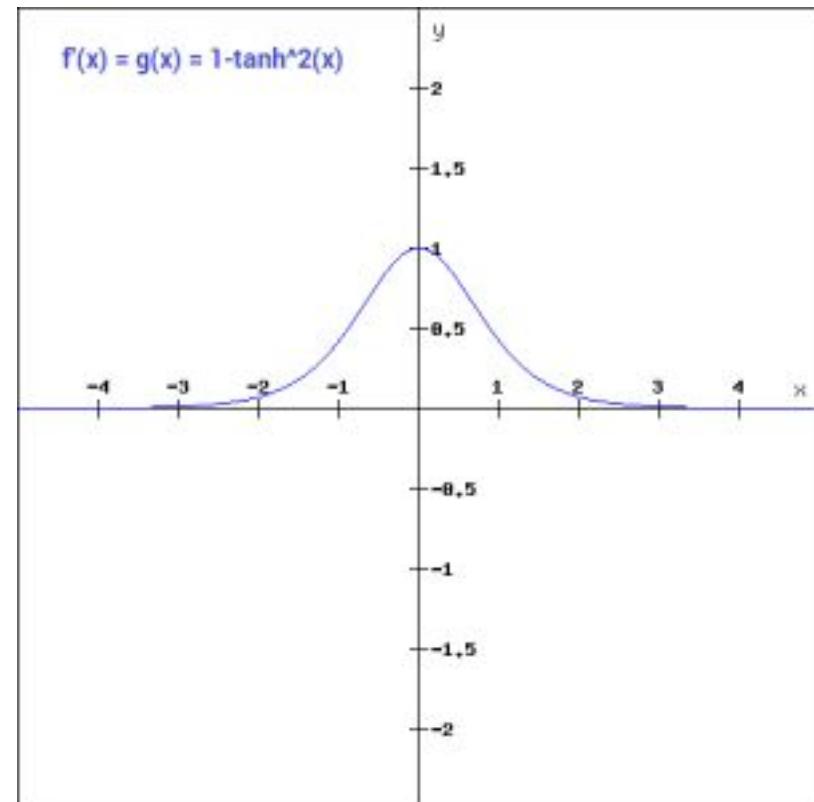
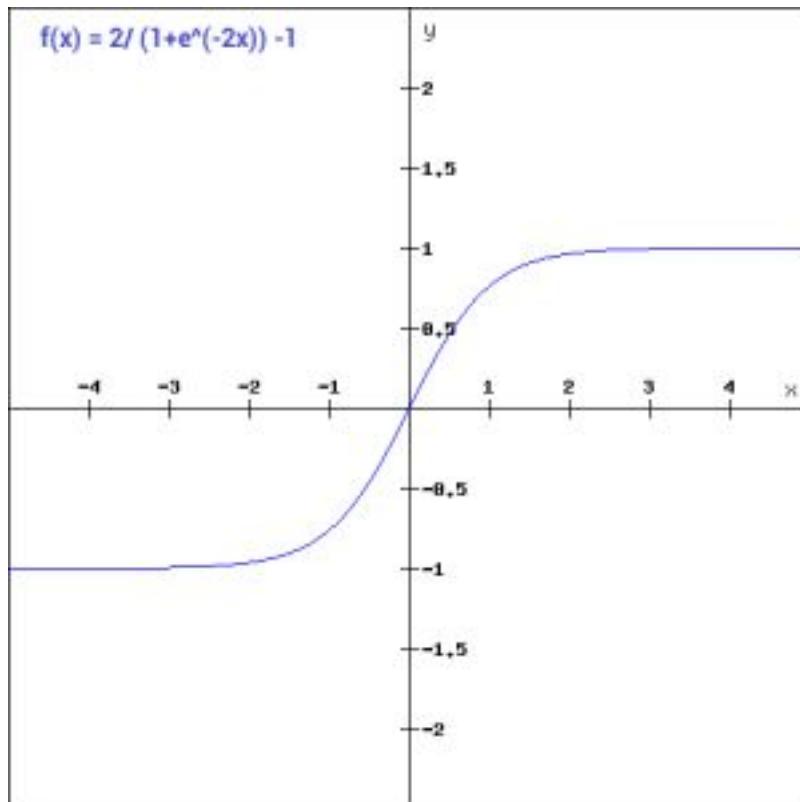
Models get stuck if fall go far away from 0. Output always positive



Notes on non-linearity

- Tanh

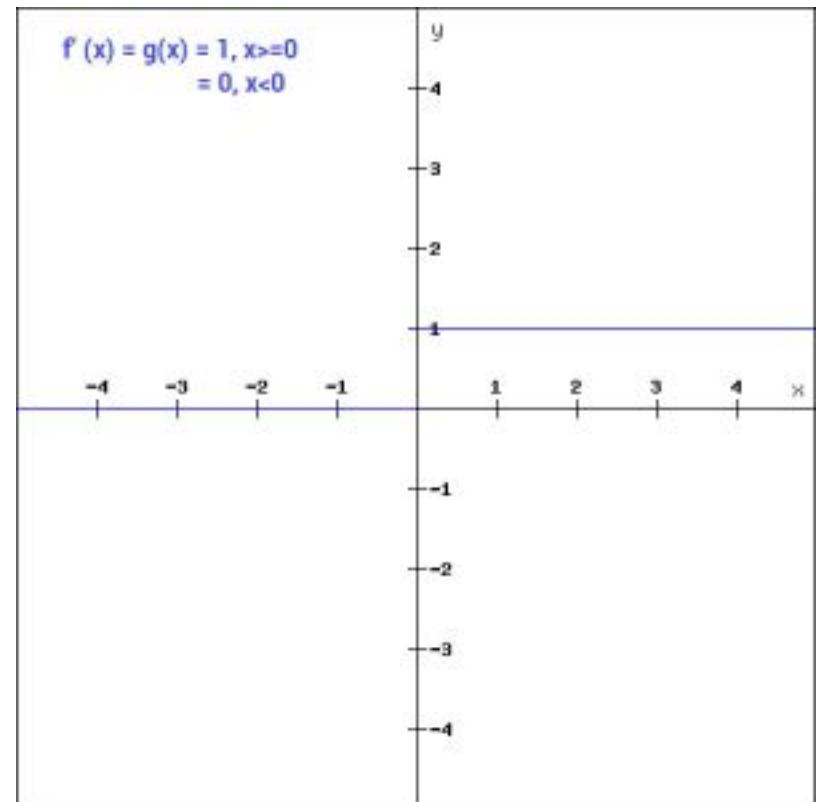
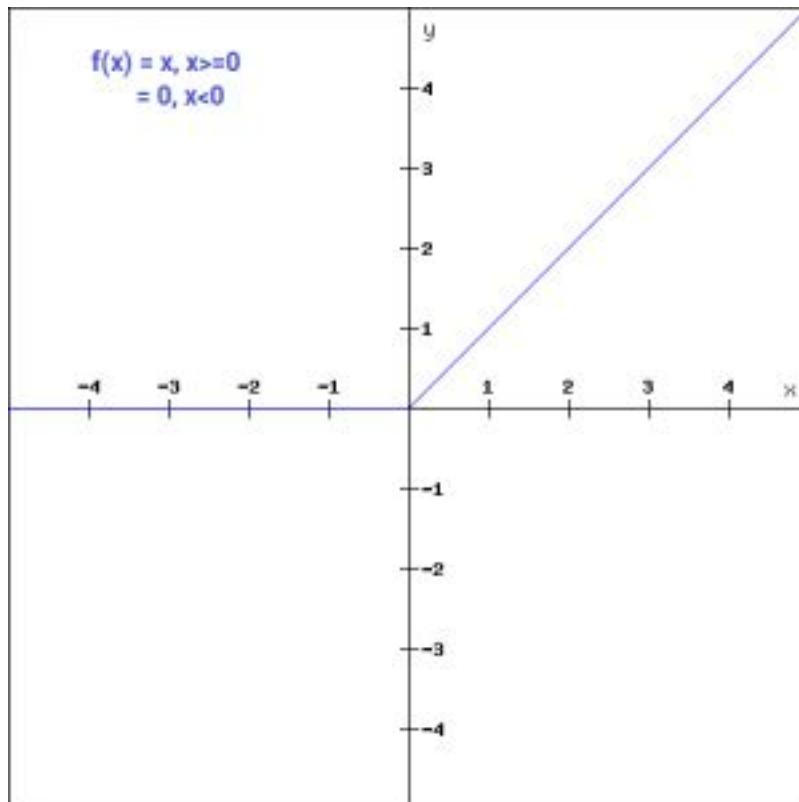
Output can be +- . Models get stuck if far away from 0



Notes on non-linearity

- ReLU

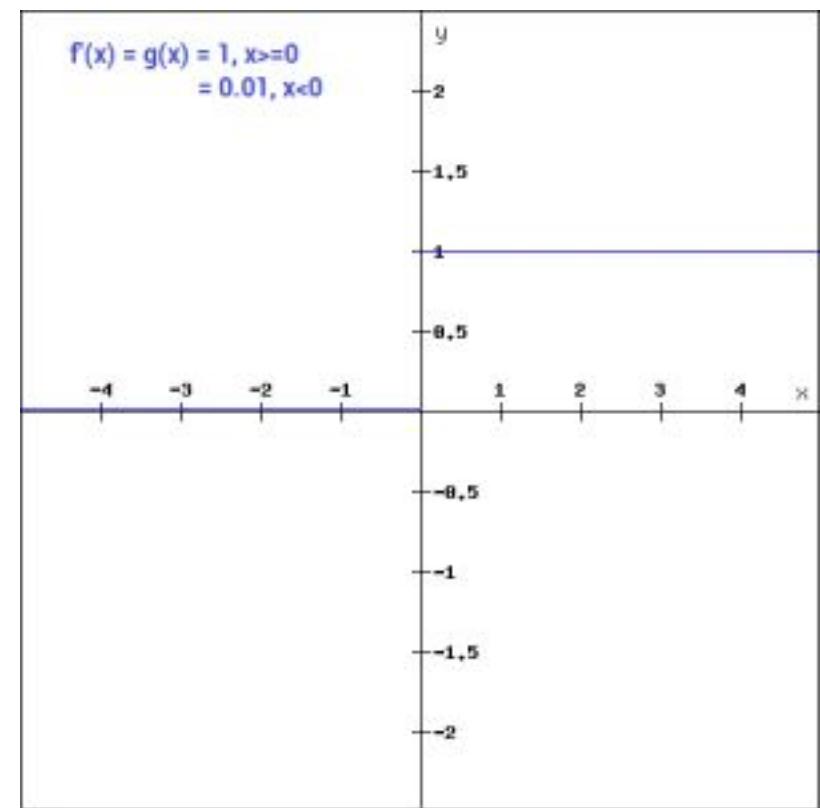
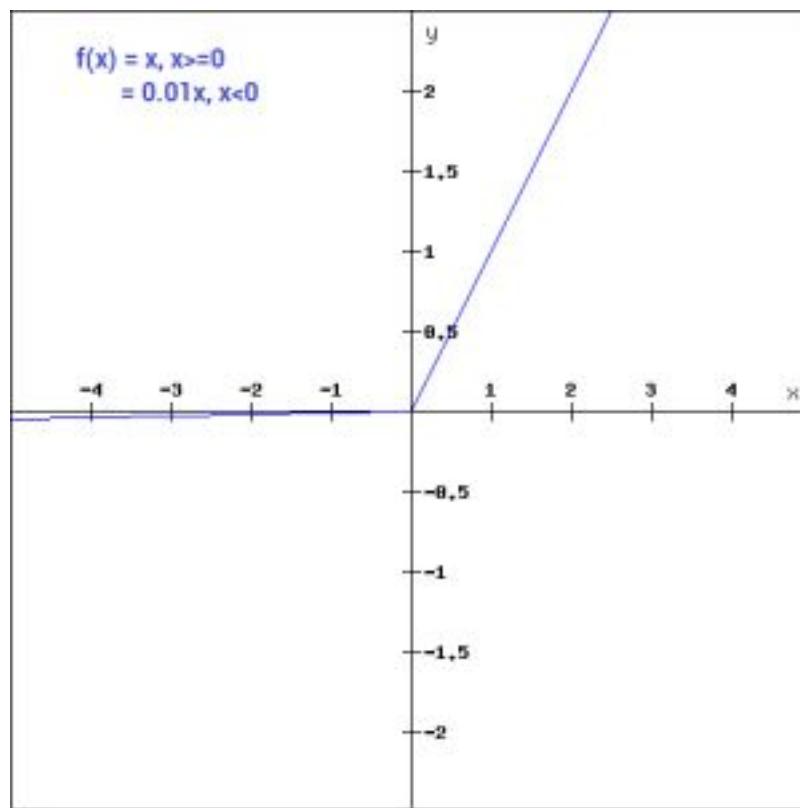
High gradient in positive. Fast compute. Gradient doesn't move in negative



Notes on non-linearity

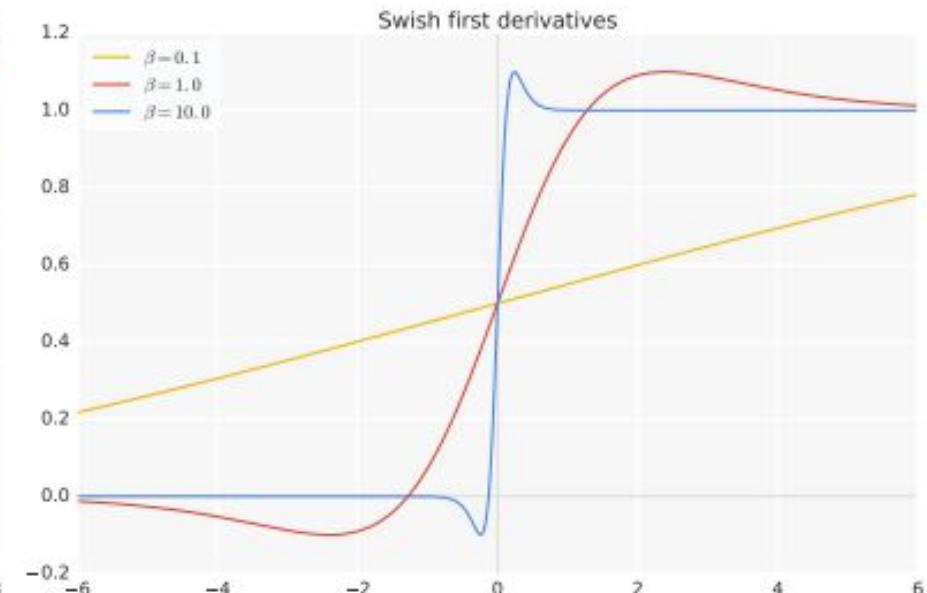
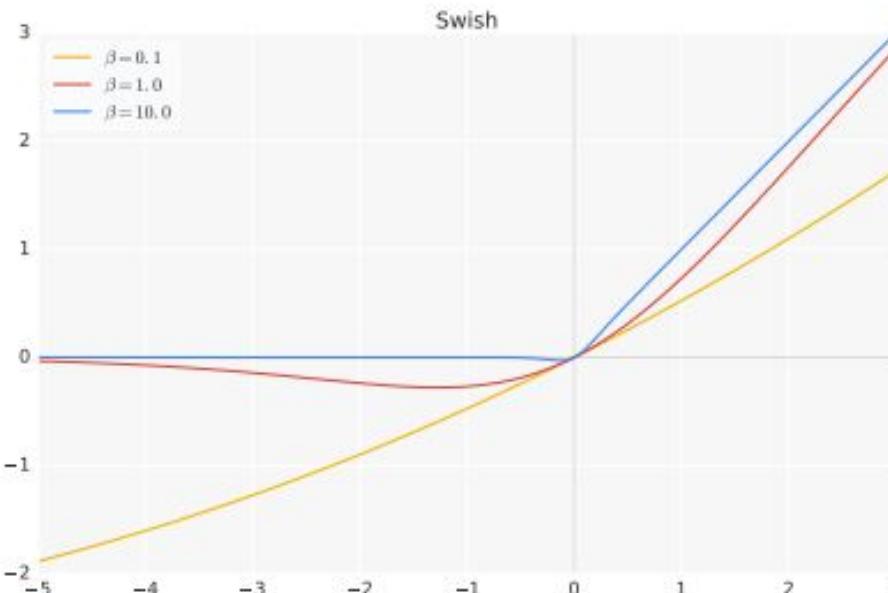
- Leaky ReLU

Negative part now have some gradient. Small improvements depending on tasks



Notes on non-linearity

- Swish
Nonnegative everywhere. Not monotonic.



Initialization

- The starting point of your descent
- Important due to local minimas
- Not as important with large networks AND big data
- Now usually initialized randomly
 - One strategy

$$W \sim \text{Uniform}(0, \frac{1}{\sqrt{\text{FanIn} + \text{FanOut}}})$$

- For ReLUs
$$w = np.random.randn(n) * \sqrt{2.0/n}$$
- Or use a pre-trained network as initialization

Stochastic gradient descent (SGD)

- Consider you have one million training examples
 - Gradient descent computes the objective function of **all** samples, then decide direction of descent
 - Takes too long
 - SGD computes the objective function on **subsets** of samples
 - The subset should not be biased and properly randomized to ensure no correlation between samples
- The subset is called a mini-batch
- Size of the mini-batch determines the training speed and accuracy
 - Usually somewhere between 32-1024 samples per mini-batch
- Definition: 1 batch vs 1 epoch

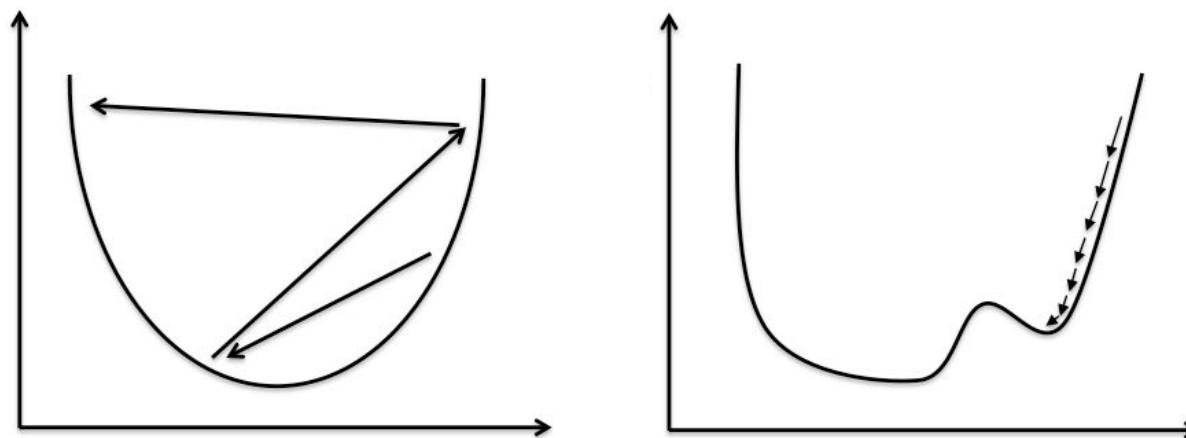
Self regularizing property of SGD

- SGD by its randomized nature does not overfit (as fast)
 - Considered as an implicit regularization (no change in the loss)

<https://cbmm.mit.edu/sites/default/files/publications/CBMM-Memo-067-v3.pdf>

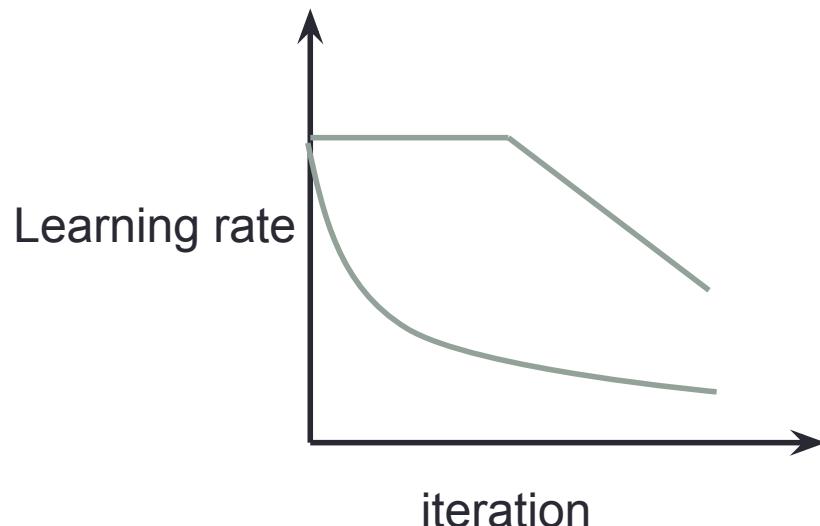
Learning rate

- How fast to go along the gradient direction is controlled by the learning rate
- Too large models diverge
- Too small the model get stuck in local minimas and takes too long to train



Learning rate scheduling

- Usually starts with a large learning rate then gets smaller later
- Depends on your task
- Automatic ways to adjust the learning rate : Adagrad, Adam, etc. (still need scheduling still)



Learning rate strategies (annealing)

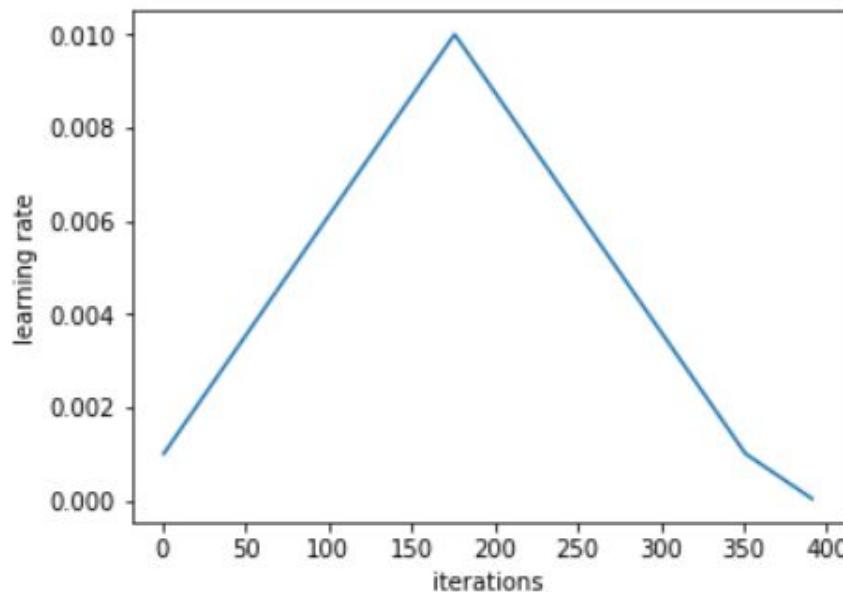
- Step decay: reduce learning rate by x after y epochs
- New bob method: half learning rate every time the validation error goes up. Only plausible in larger tasks
- Exponential decay: multiplies the learning rate by $\exp(-\text{rate} * \text{epoch number})$

Learning rate warm up

Initial point of the network can be at a bad spot.

Try not to go to fast - has a warm up period.

Useful for large datasets, or adaption (transfer learning)



Potentially leads to faster convergence and better accuracy

See links below for methods to select the shape of the triangle

<https://sgugger.github.io/the-1cycle-policy.html#the-1cycle-policy>

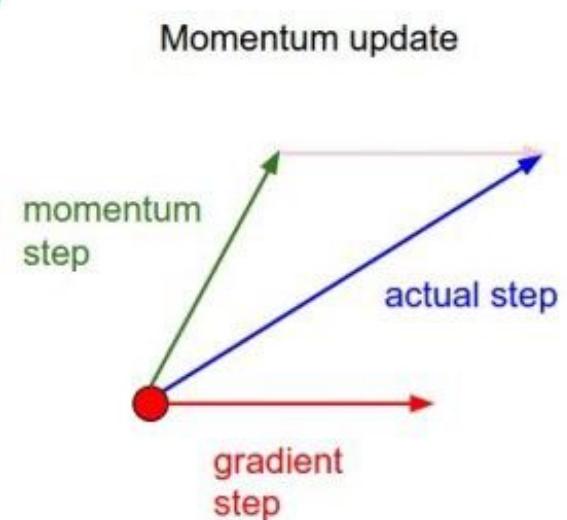
[Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour](#)

[Cyclical Learning Rates for Training Neural Networks](#)

Momentum

- Gradient descent can get stuck on small local minimas
 - Or slow down at saddle points
- Have concept of speed

$$\begin{aligned} \text{speed} & \quad \text{Momentum rate} & \text{gradient} \\ V_t &= \underline{\beta} V_{t-1} + (1 - \underline{\beta}) \underline{\nabla_w L(W, X, y)} \\ W &= W - \underline{\alpha} V_t & \text{learning rate} \end{aligned}$$

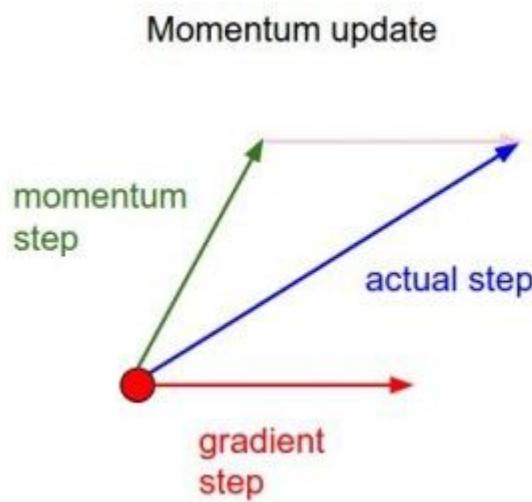


Nesterov Momentum

- Momentum with look ahead.
 - Compute gradient as if we took an additional step

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W - \underline{\beta V_{t-1}}, X, y)$$

$$W = W - V_t \quad \text{gradient is computed as if we took a step}$$

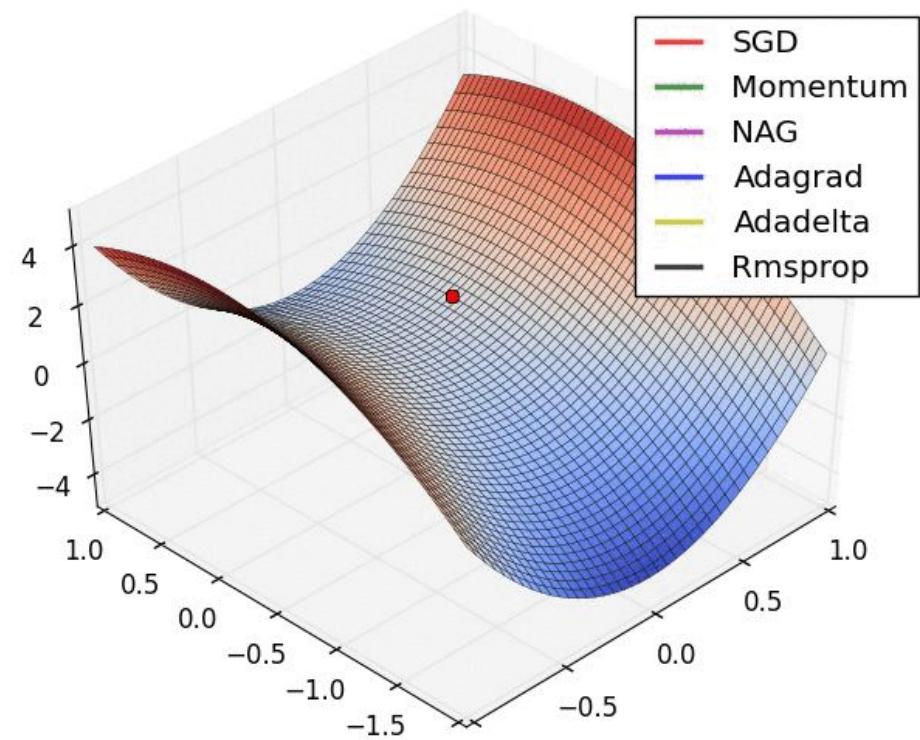
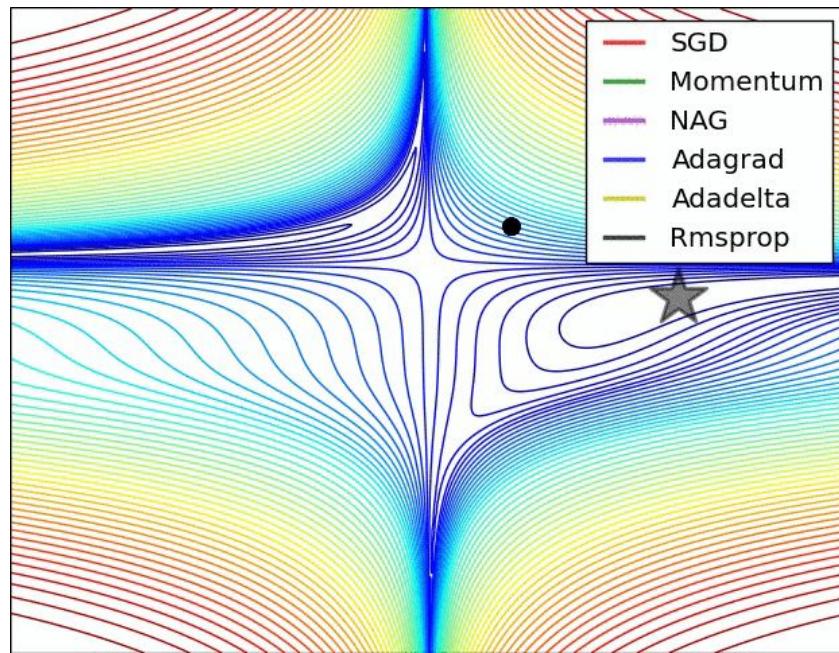


Optimizers

- Besides learning rate scheduling (coarse grain) we can do finer (and automatic) control of the learning rate via optimizers
- RMSprop
 - Faster than SGD but slower than ADAM
 - More stable than ADAM
- ADAM + variants
 - Most popular for its ease of use

People find simple SGD with momentum and decay to perform better (with proper tuning)

Optimization method and speed

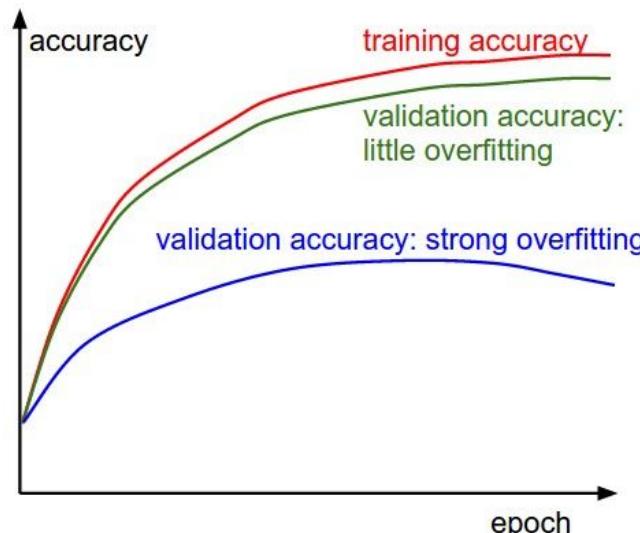


Learning rate tricks

- At least decay the learning rate
 - Monitor validation set performance
- If the loss never goes down -> decrease the learning rate (by factor of 10)
- Start with ADAM. Also try RMSprop and SGD with Nesterov Momentum if you have time

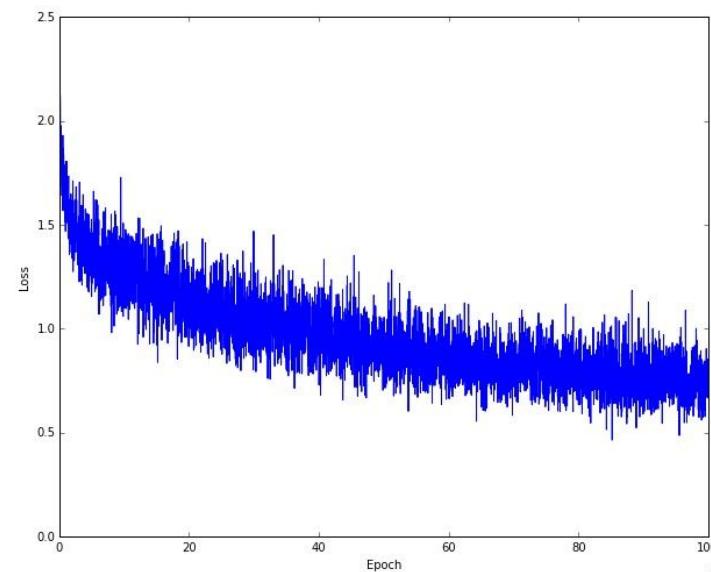
Overfitting

- You can keep doing back propagation forever!
- The training loss will always go down
- But it overfits
- Need to monitor performance on a held out set
- Stop or decrease learning rate when overfit happens



Monitoring performance

- Monitor performance on a dev/validation set
 - This is NOT the test set
- Can monitor many criterions
 - Loss function
 - Classification accuracy
- Sometimes these disagree
- Actual performance can be noisy, need to see the trend



Dropout

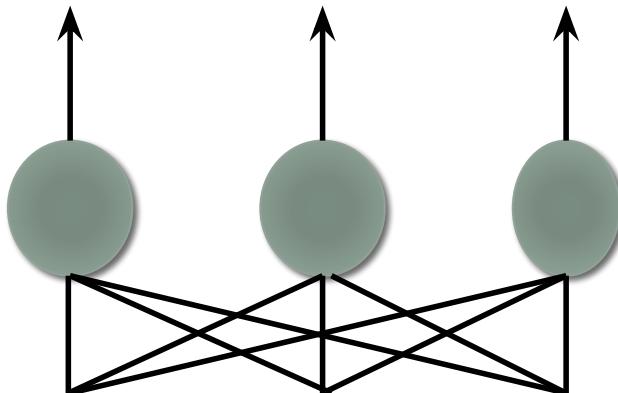
A **implicit regularization** technique for reducing overfitting

Randomly turn off different subset of neurons during training

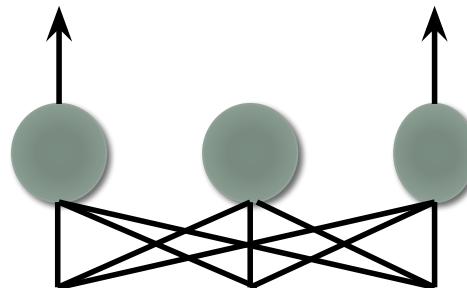
Network no longer depend on any particular neuron

Force the model to have redundancy – robust to any corruption in input data

A form of performing model averaging (ensemble of experts)



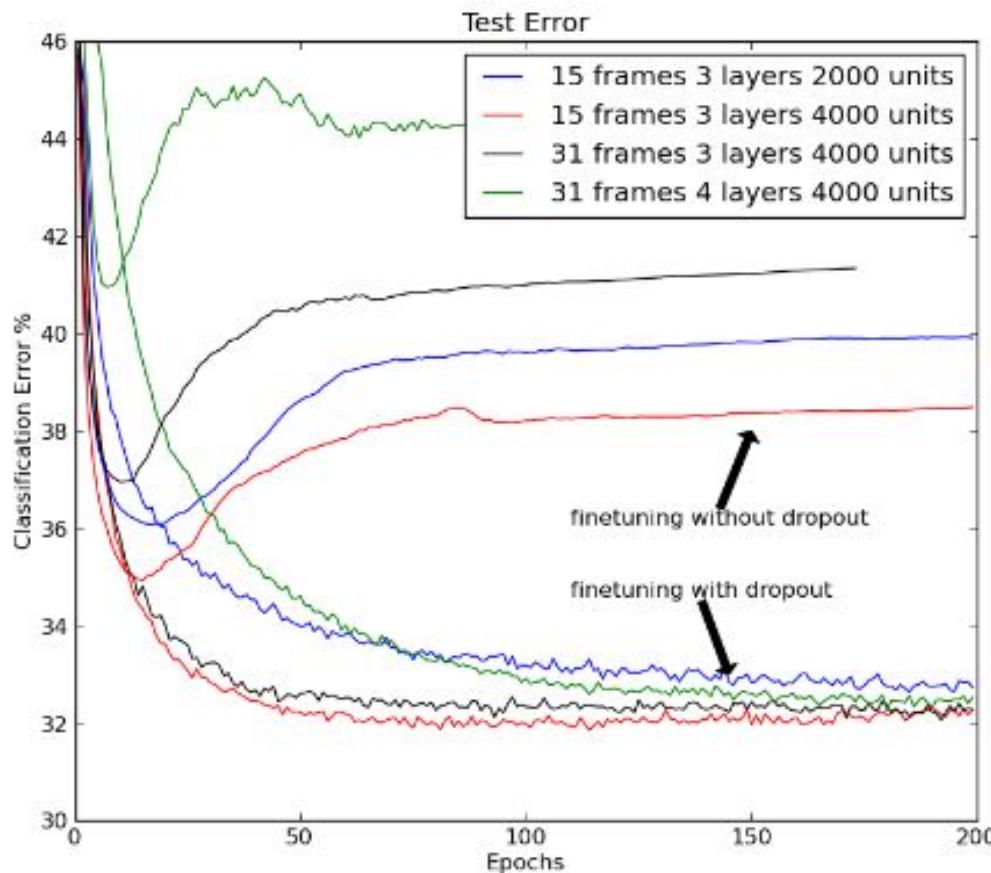
Model



Dropout rate of 0.33

Dropout on TIMIT

- A phoneme recognition task



Batch normalization

- Recent technique for (implicit) regularization
- **Normalize every mini-batch** at various batch norm layers to standard Gaussian (different from global normalization of the inputs)
- Place batch norm layers before non-linearities
- Faster training and better generalizations

For each mini-batch that goes through
batch norm

1. Normalize by the mean and variance of the mini-batch for each dimension
2. Shift and scale by learnable parameters

$$\hat{x} = \frac{x - \mu_b}{\sigma_b}$$
$$y = \alpha \hat{x} + \beta$$

Replaces dropout in some networks

<https://arxiv.org/abs/1502.03167>

Dropout vs batchnorm

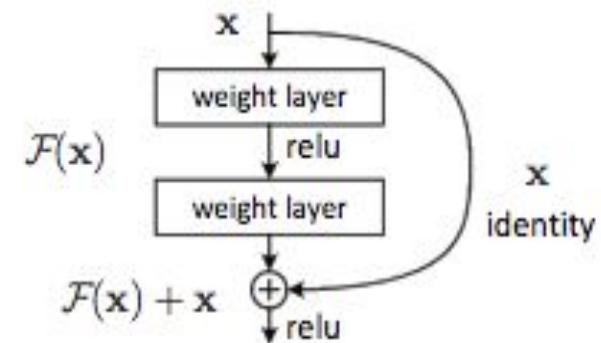
- You can add dropout in the hidden layers (0-0.5)
- Or input layers (0-0.2 is typical)
 - “Noising” the inputs, data augmentation
- Dropout in computer vision
 - use batchnorm instead (convolution layers leak output when using dropout)
- Dropout in NLP
 - Usually works better than Batchnorm for NLP in simple architectures
 - Drop full words at the embedding
 - Recurrent dropout <http://arxiv.org/abs/1512.05287> (bugged in TF2)
 - Recent works (transformers) find batchnorm to be better than dropout
 - For seq2seq models, layer norm is popular

Vanishing/Exploding gradient

- Backprop introduces many multiplications down chain
- The gradient value gets smaller and smaller
 - The deeper the network the smaller the gradient in the lower layers
 - Lower layers changes too slowly (or not at all)
 - Hard to train very deep networks (>6 layers)
- The opposite can also be true. The gradient explodes from repeated multiplication
 - Put a maximum value for the gradient (Gradient clipping)

- How to deal with this?
 - Residual connection

<https://arxiv.org/pdf/1512.03385.pdf>

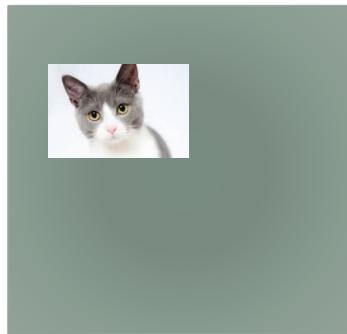
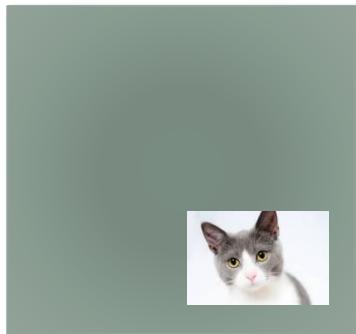


Neural networks

- Fully connected networks
 - Neuron
 - Non-linearity
 - Softmax layer
- DNN training
 - Loss function and regularization
 - SGD and backprop
 - Learning rate
 - Overfitting – dropout, batchnorm
- Demos
 - Tensorflow, Keras
- CNN
- RNN, LSTM, GRU <- Next class

Convolutional Neural Networks (CNNs)

- Consider an image of a cat. DNNs need different neurons to learn every possible location a cat can be

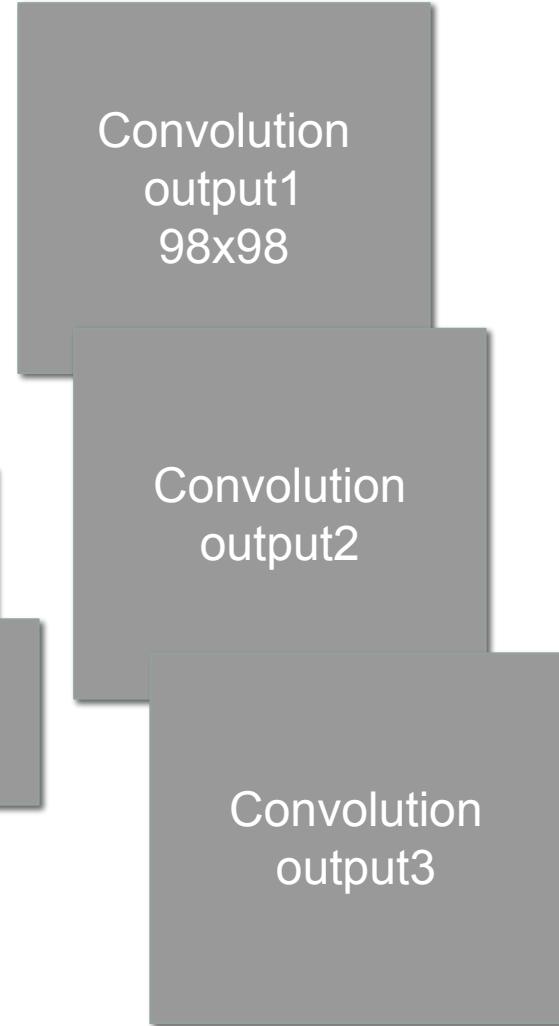
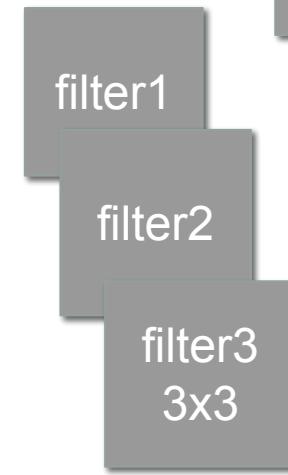
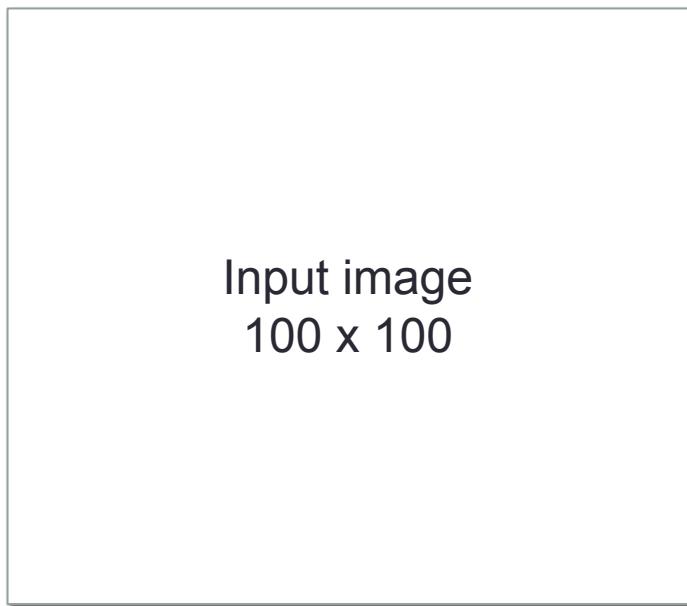


- Can we use the same parameters to learn that a cat exists regardless of location?
- 2 parts: convolutional layer and pooling layer

Convolutional filters

Multiply inputs with filter values

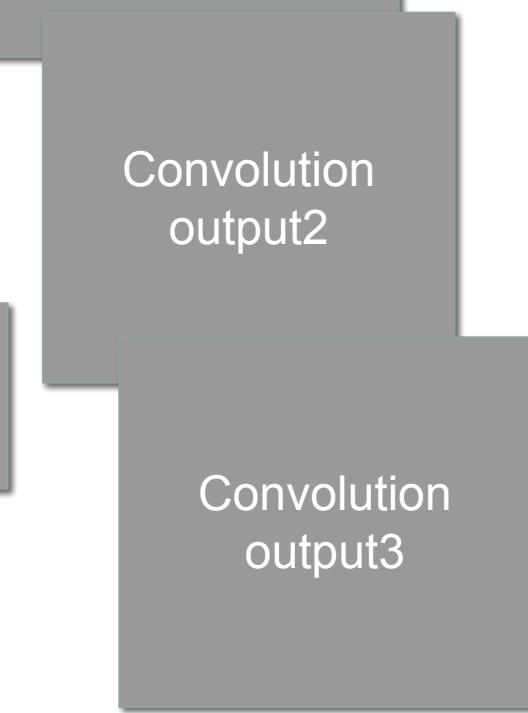
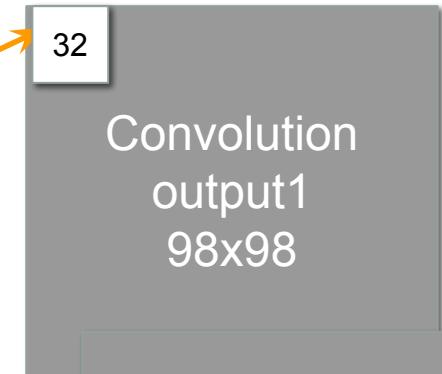
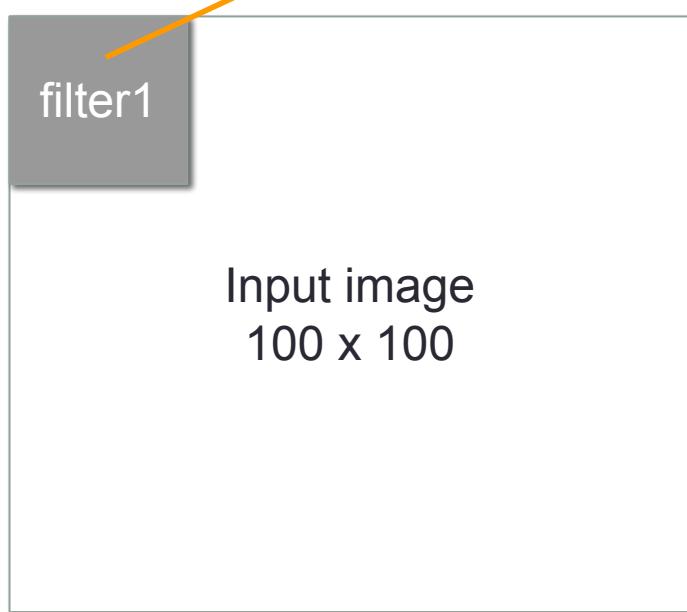
Output one feature map per filter



Convolutional filters

| | | |
|---|---|----|
| 0 | 1 | -1 |
| 1 | 0 | 1 |
| 1 | 2 | 0 |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

$$1*2 + -1*3 + 1*4 + 1*6 + 1*7 + 2*8 = 32$$

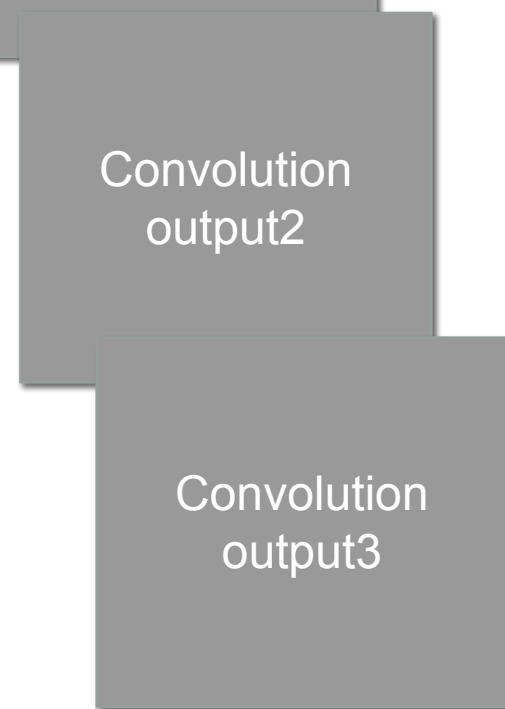
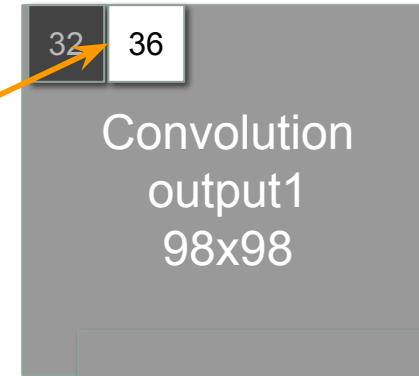
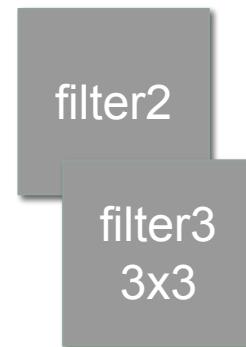
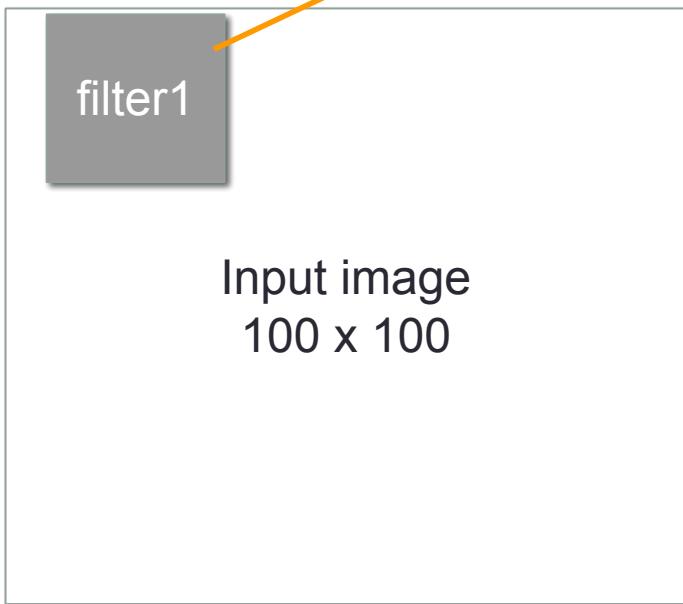


Convolutional filters

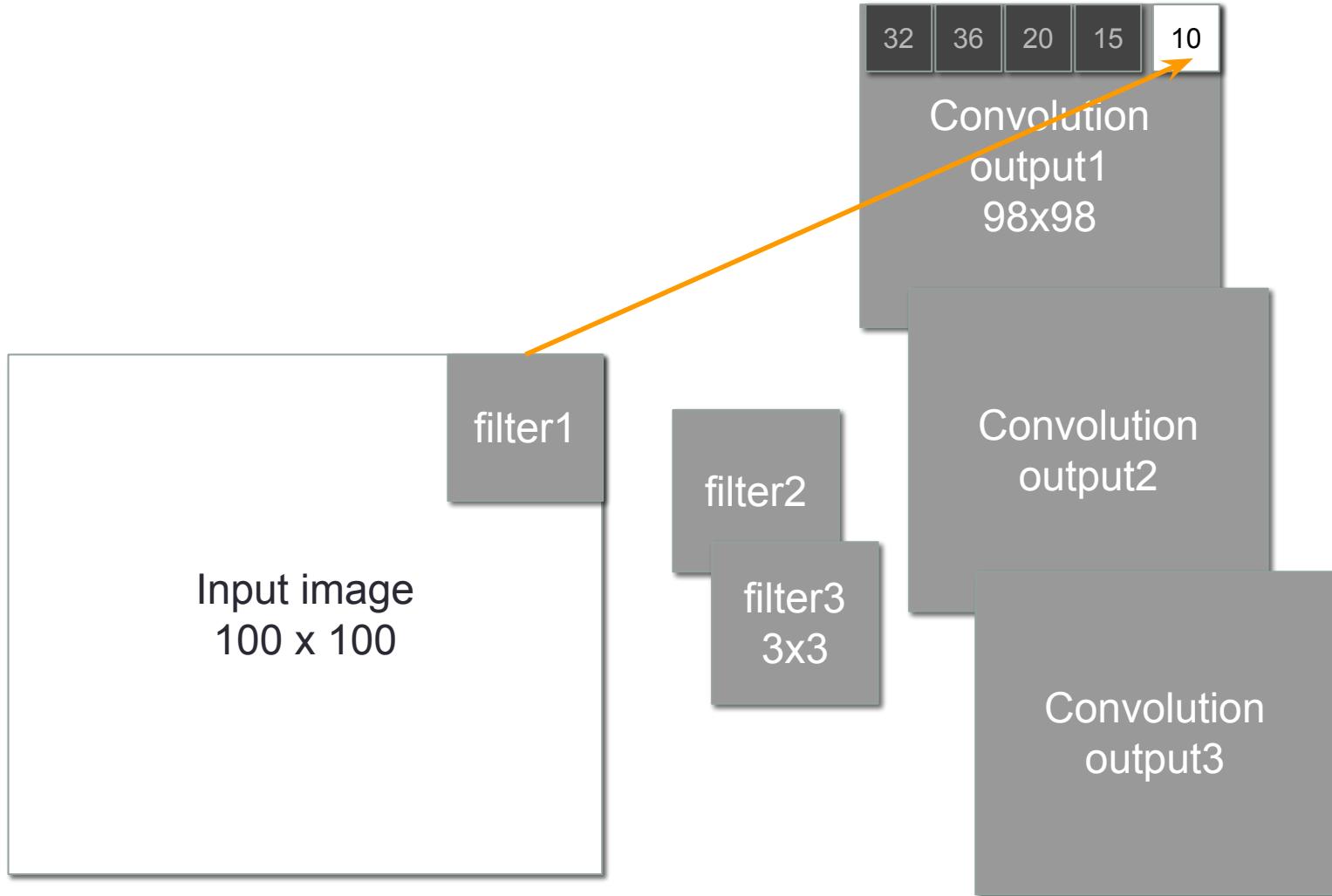
Stride of 1

| | | |
|---|---|----|
| 0 | 1 | -1 |
| 1 | 0 | 1 |
| 1 | 2 | 0 |
| 2 | 3 | 1 |
| 5 | 6 | 3 |
| 8 | 9 | 8 |

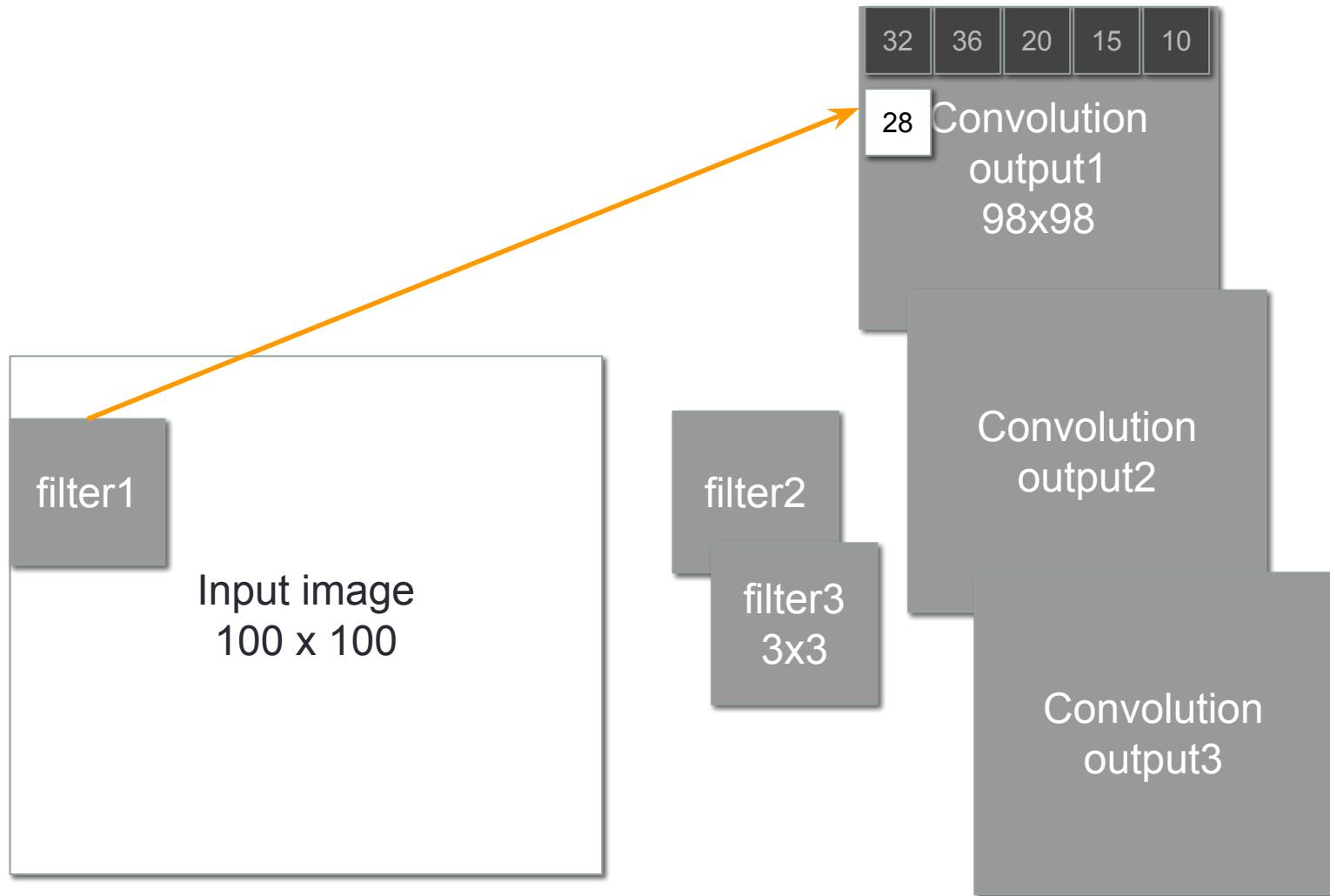
$$1*3 + -1*1 + 1*5 + 1*3 + 1*8 + 2*9 = 36$$



Convolutional filters

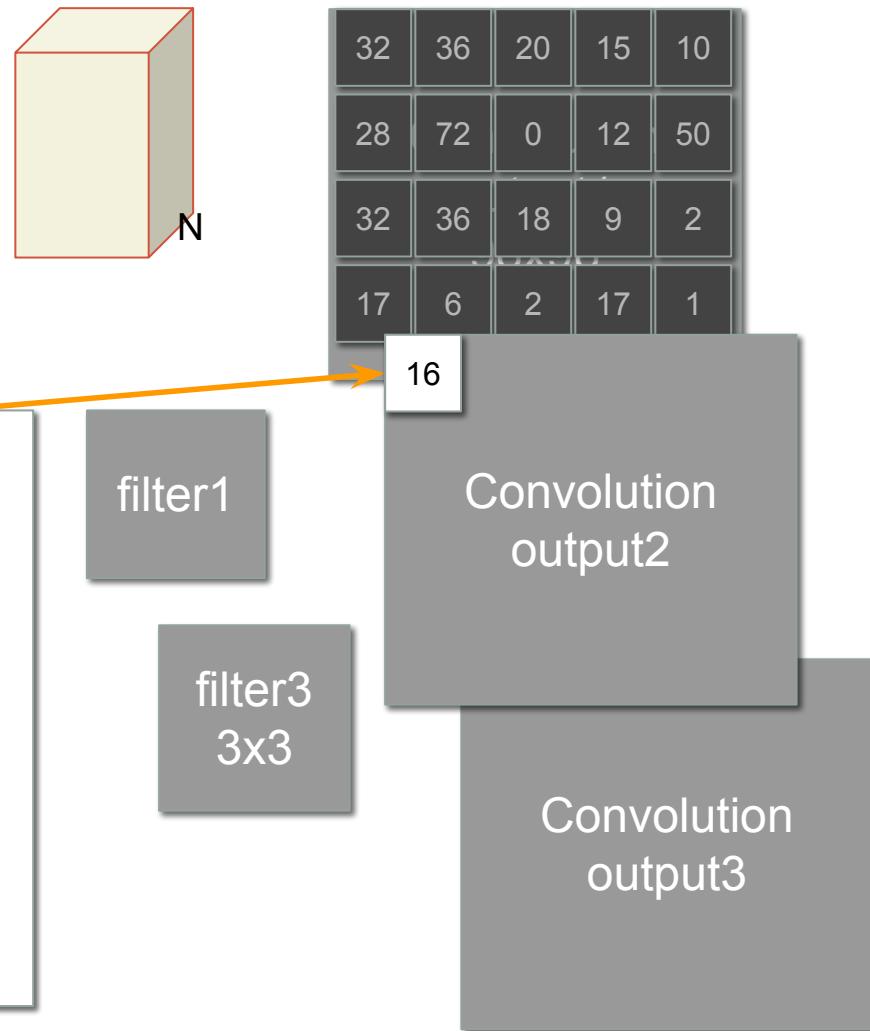


Convolutional filters



Convolutional filters

N filters means N feature maps
You get a 3 dimensional output



Pooling/subsampling

Reduce dimension of the feature maps



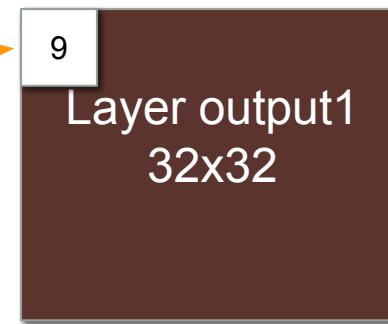
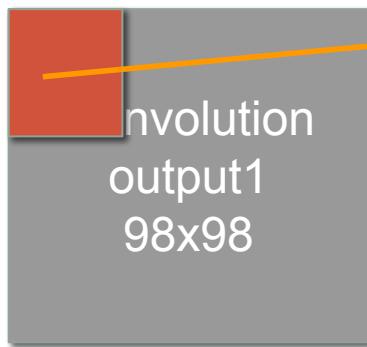
3x3 Max filter
with no overlap



Pooling/subsampling

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Max = 9

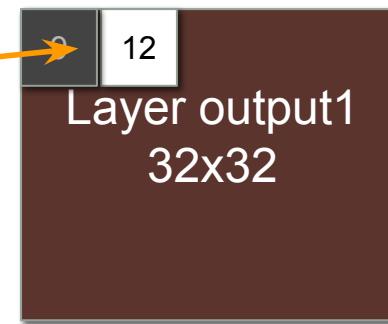
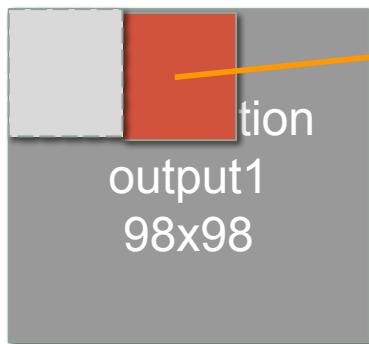


Pooling/subsampling

| | | |
|---|---|----|
| 5 | 2 | 1 |
| 5 | 7 | 1 |
| 9 | 5 | 12 |

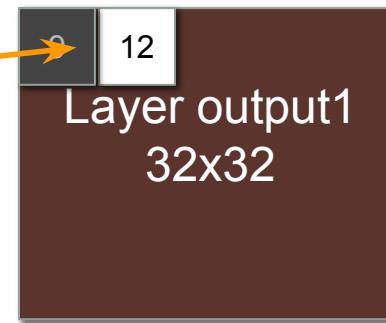
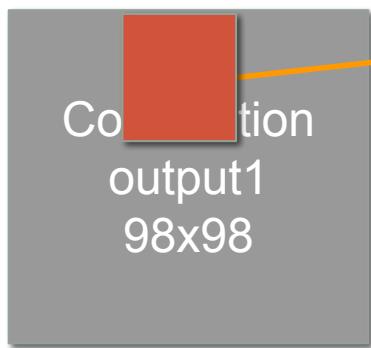
Max = 12

Stride = 3



Pooling/subsampling

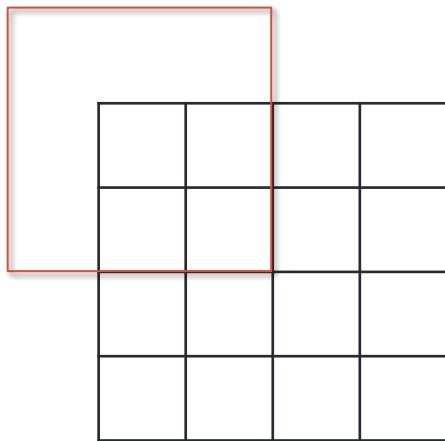
Can use other functions besides max
Example, average



Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

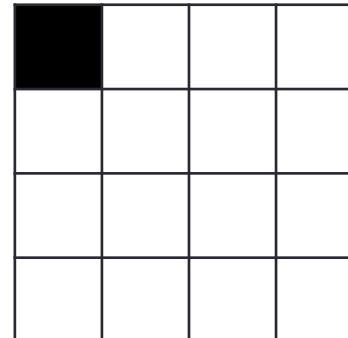
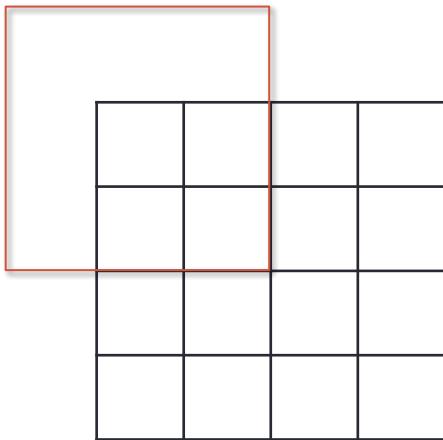
What is the output size?



Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

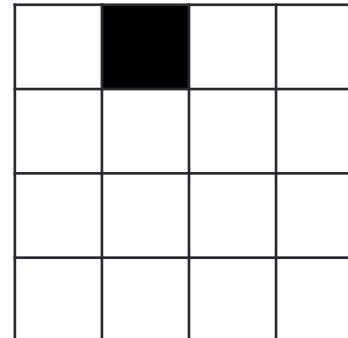
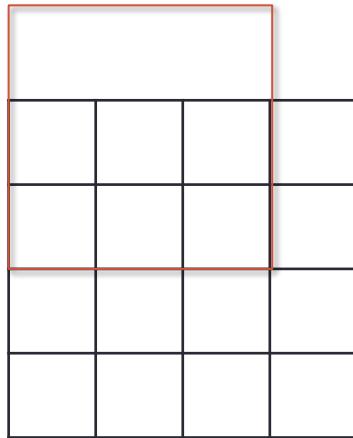
What is the output size?



Convolution puzzle

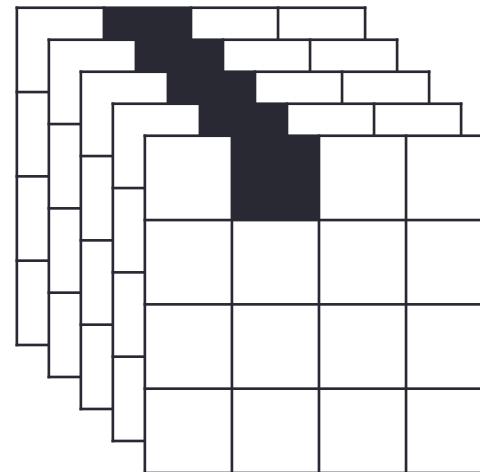
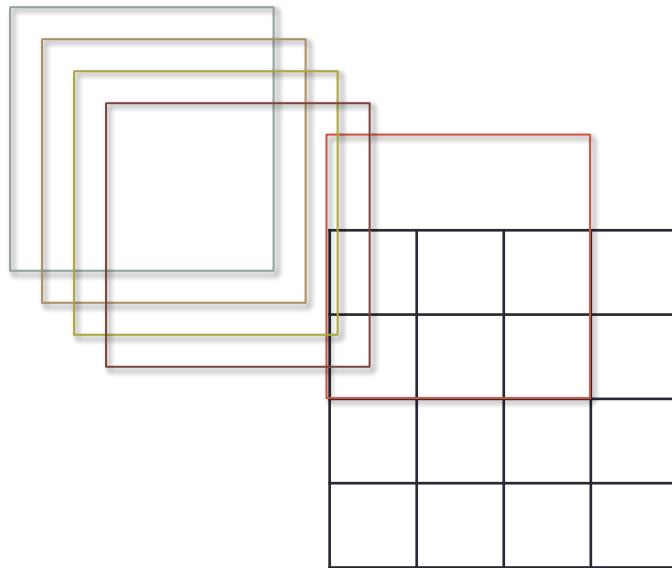
5 filters 3x3 filter pad, stride 1, pad 1

What is the output size?



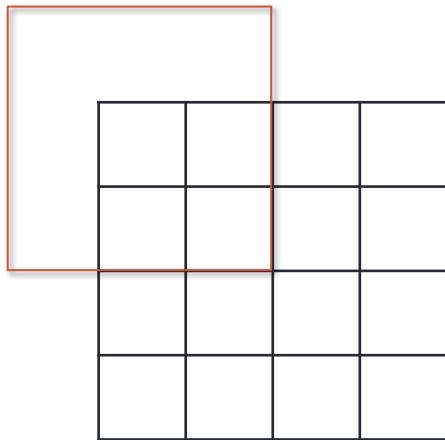
Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1



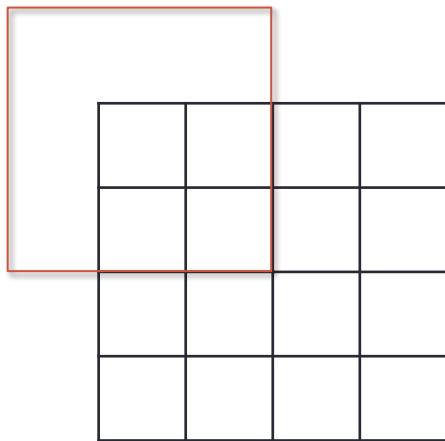
Convolution puzzle

3x3 filter pad, stride 2, pad 1



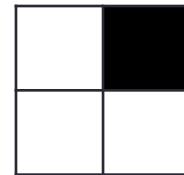
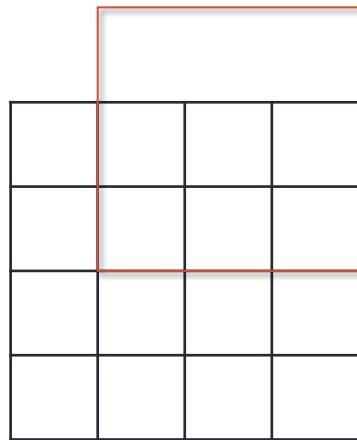
Convolution puzzle

3x3 filter pad, stride 2, pad 1



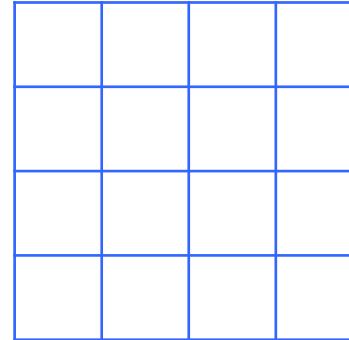
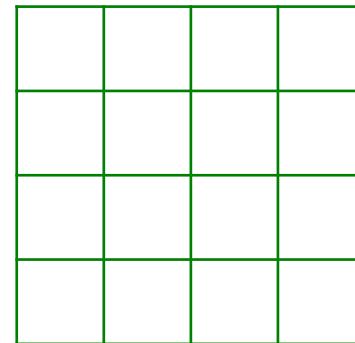
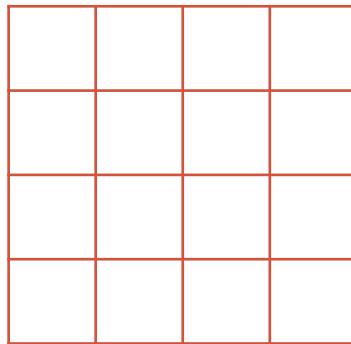
Convolution puzzle

3x3 filter pad, stride 2, pad 1



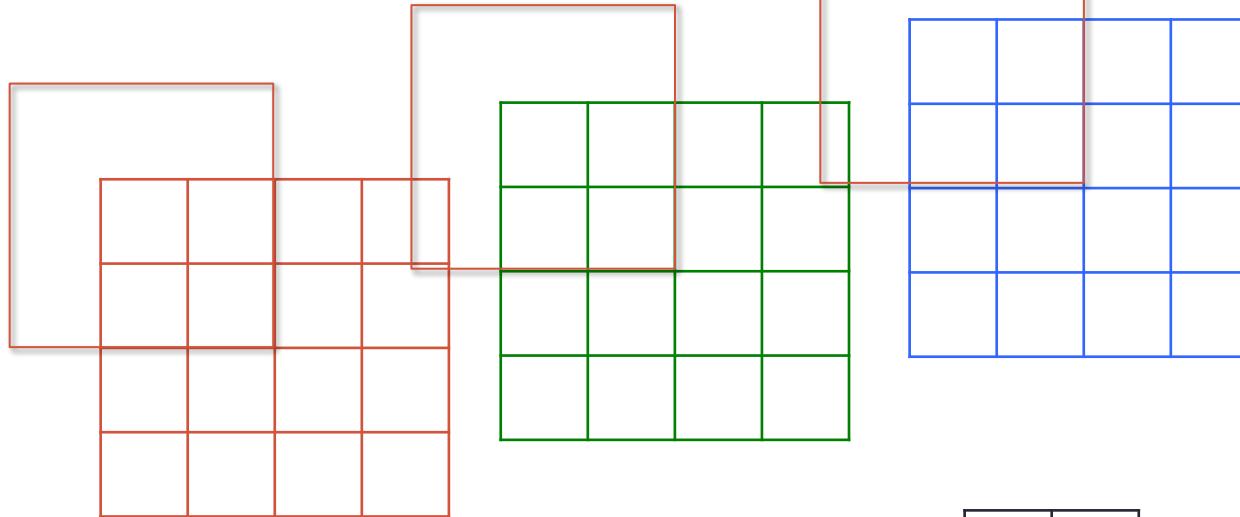
Convolution puzzle

RGB input (3 channels) 5 filters 3x3 filter pad, stride 2, pad 1

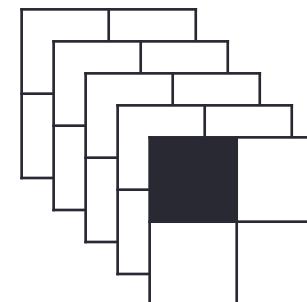


Convolution puzzle

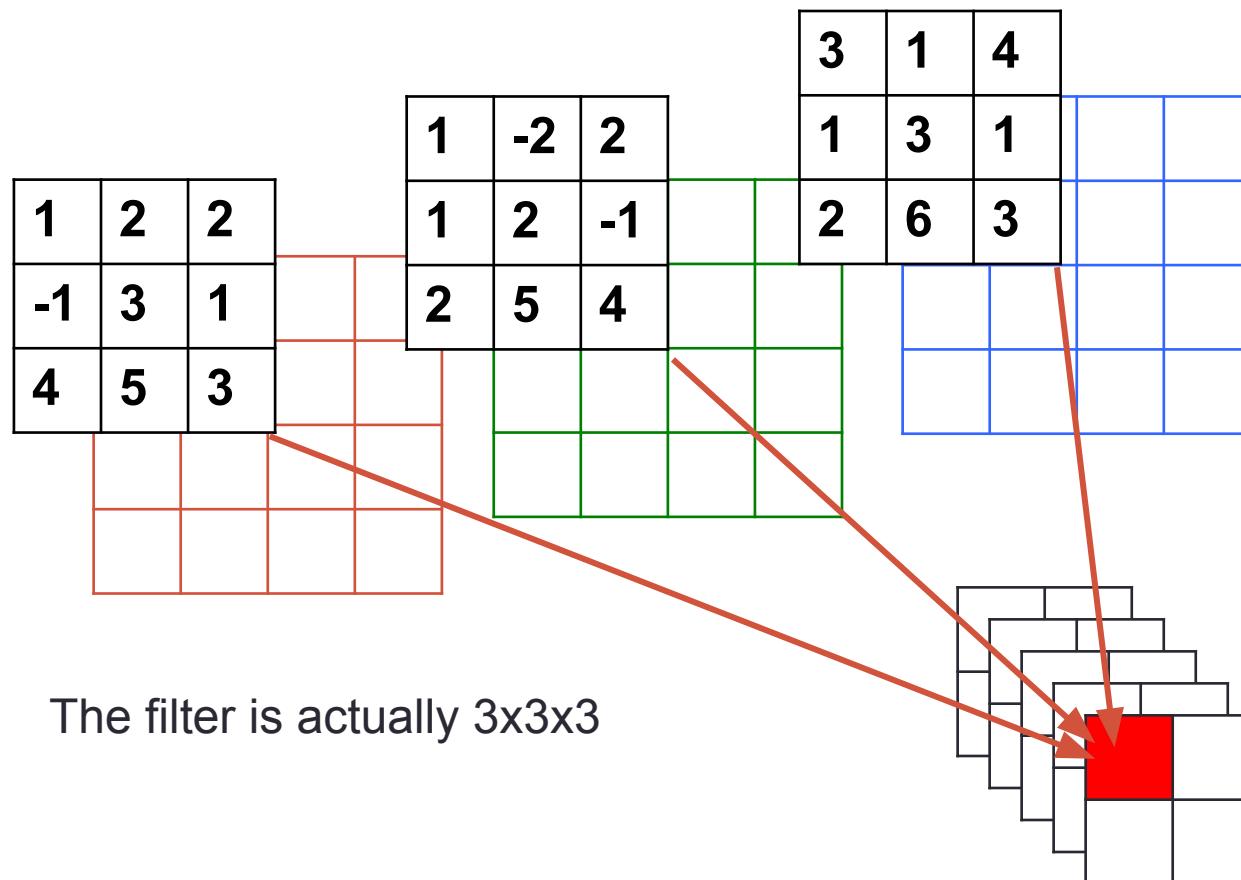
RGB input (3 channels) 5 filters 3x3 filter pad, stride 2, pad 1



The filter is actually 3x3x3

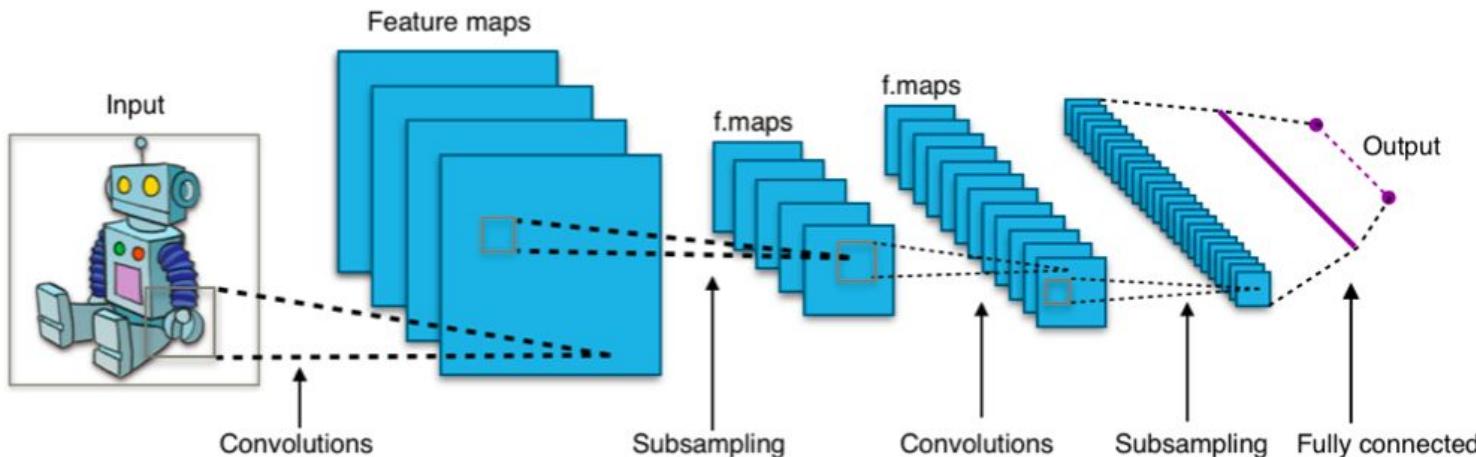


Convolution puzzle



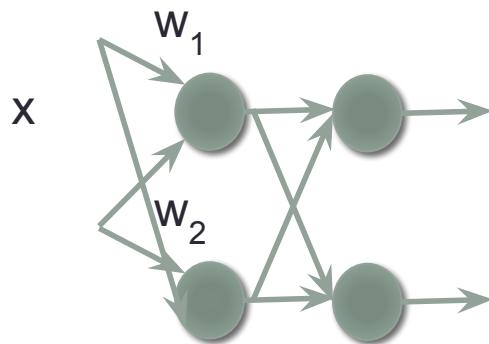
CNN overview

- Filter size, number of filters, filter shifts, and pooling rate are all parameters
- Usually followed by a fully connected network at the end
 - CNN is good at learning low level features
 - DNN combines the features into high level features and classify

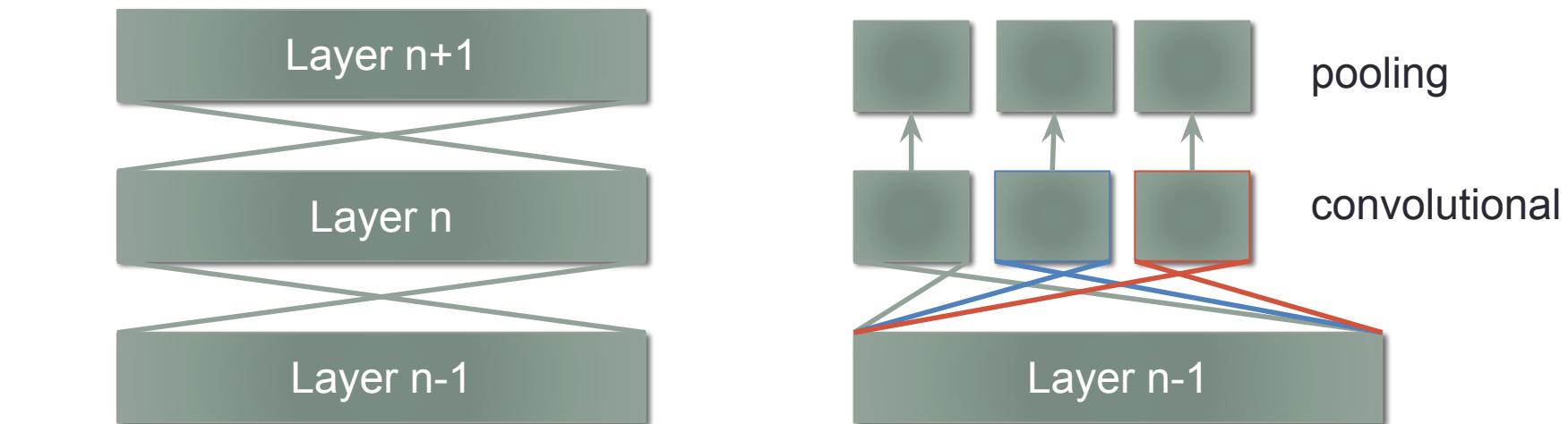


Parameter sharing in convolution neural networks

- $W^T x$

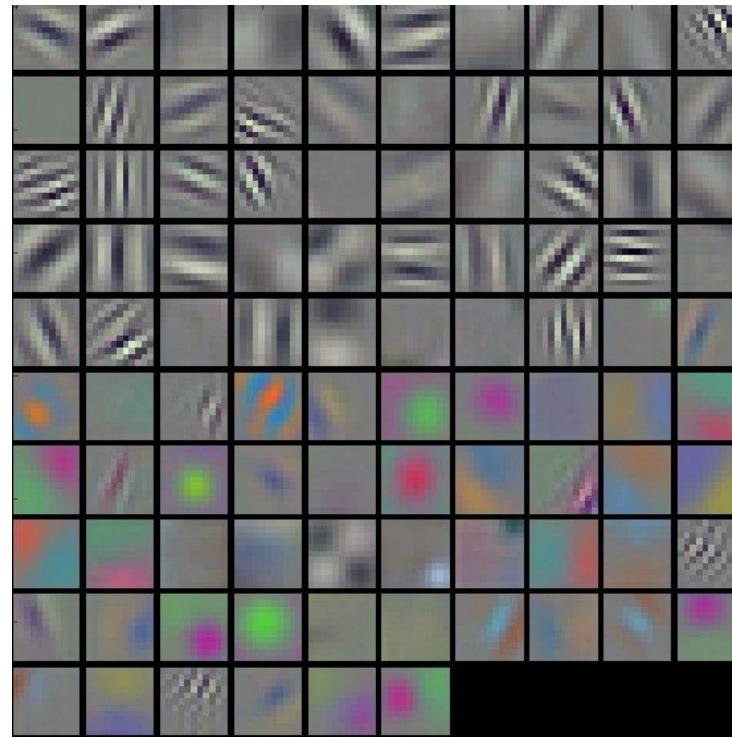


- Cats at different location might need two neurons for different locations in fully connect NNs.
- CNN shares the parameters in 1 filter
- The network is no longer fully connected



Visualizing convolutional layers

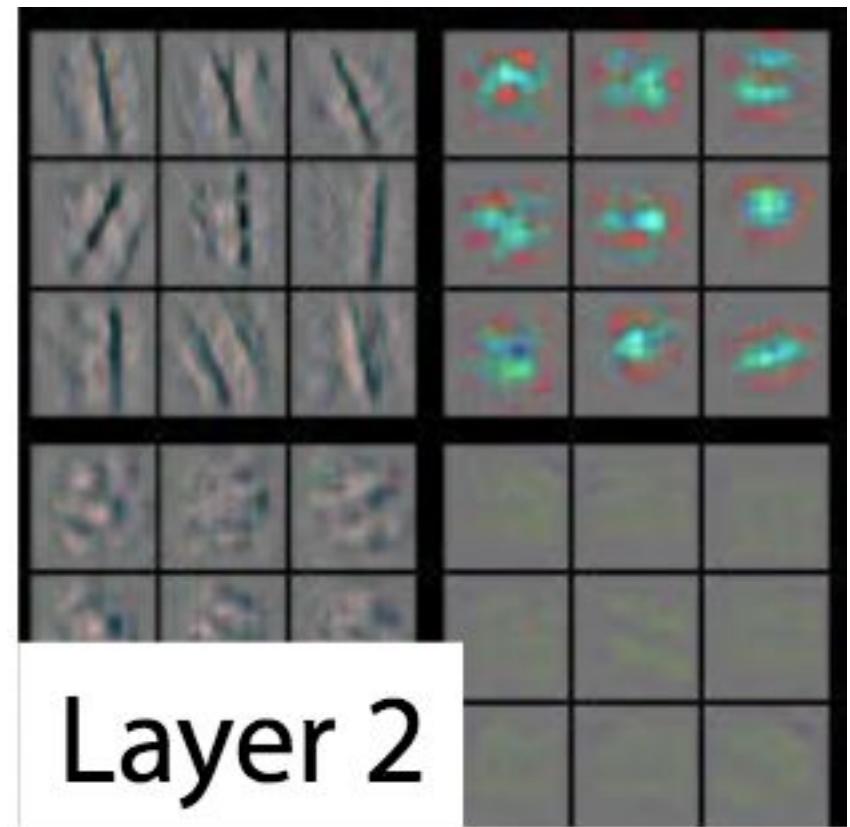
- We can visualize the weights of the filter
- “Matched filters”

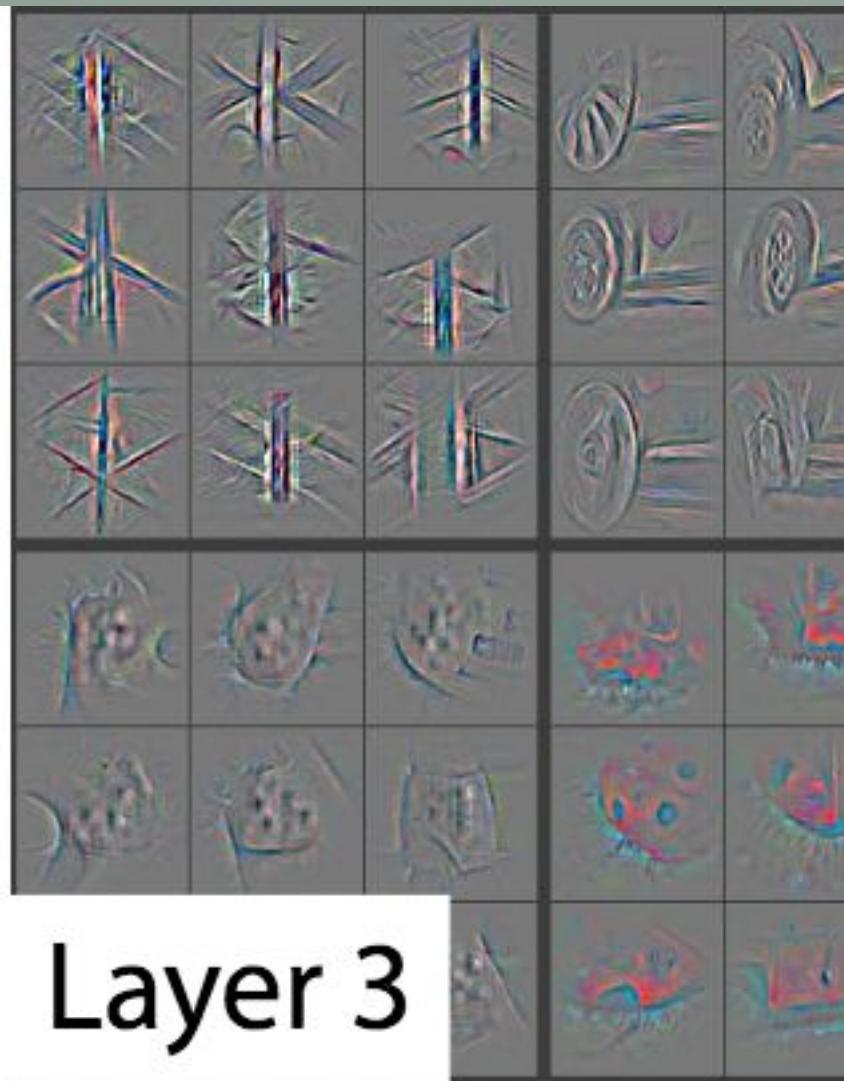


Higher layer captures higher-level concepts

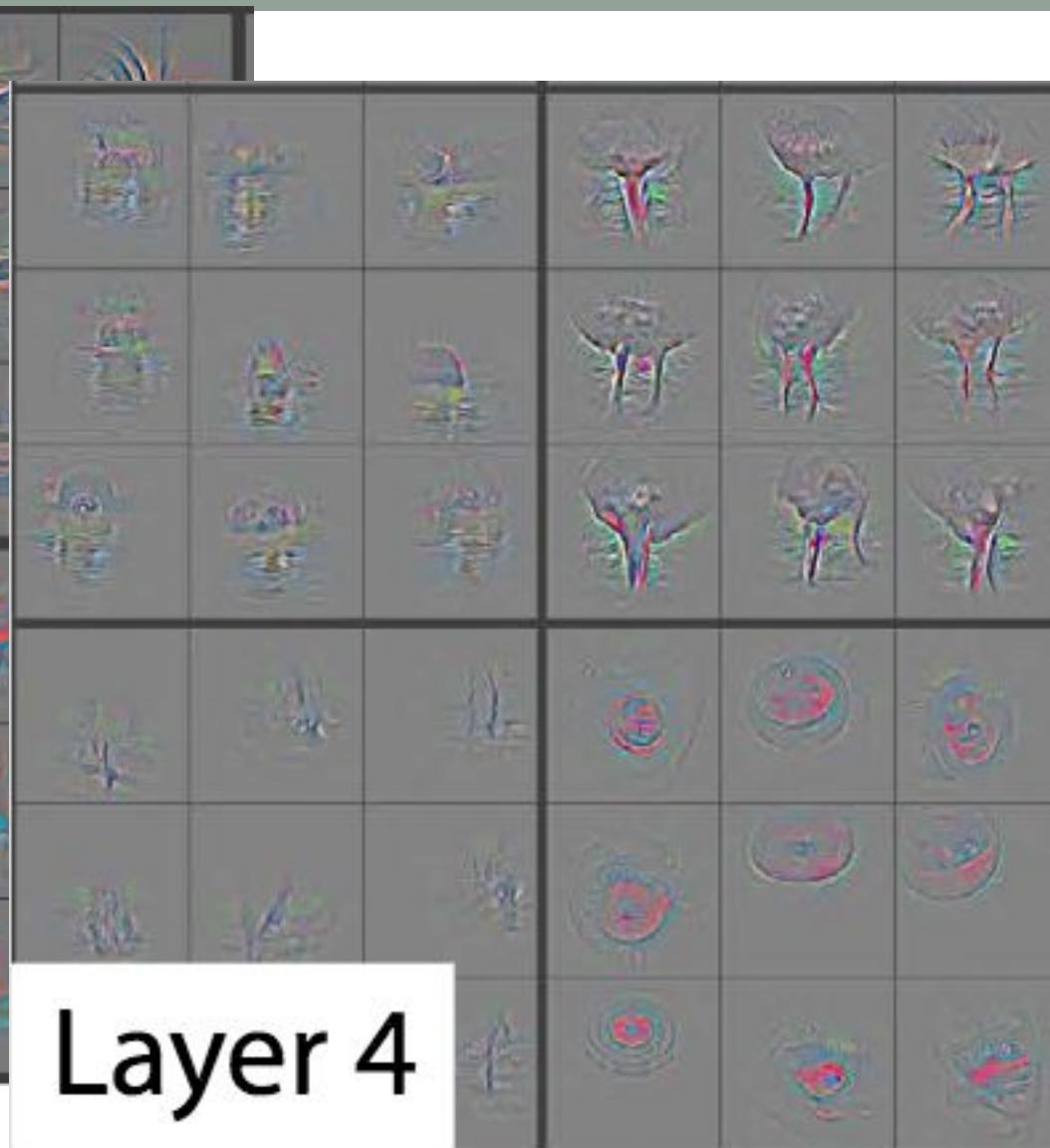


Layer 1





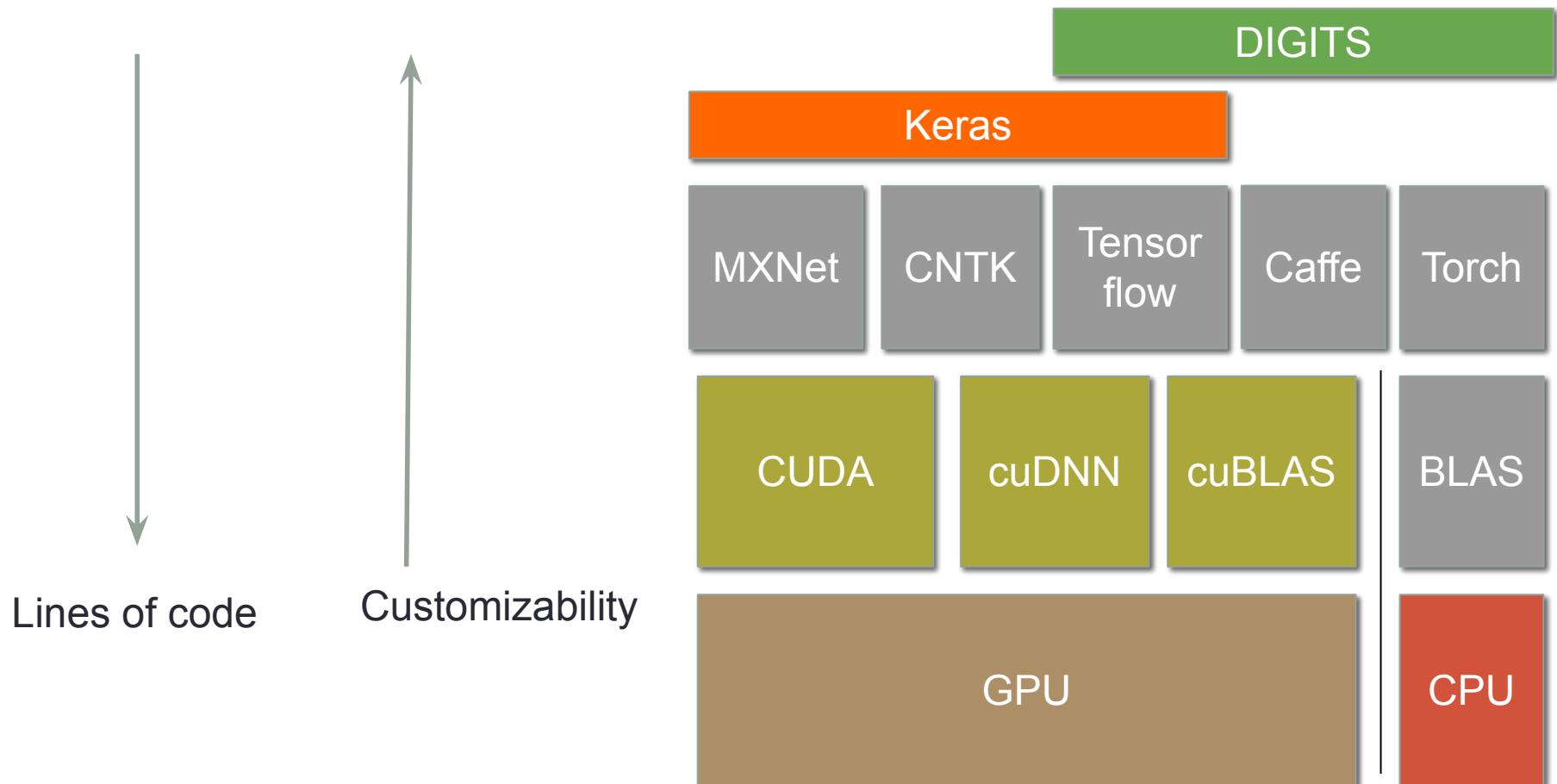
Layer 3



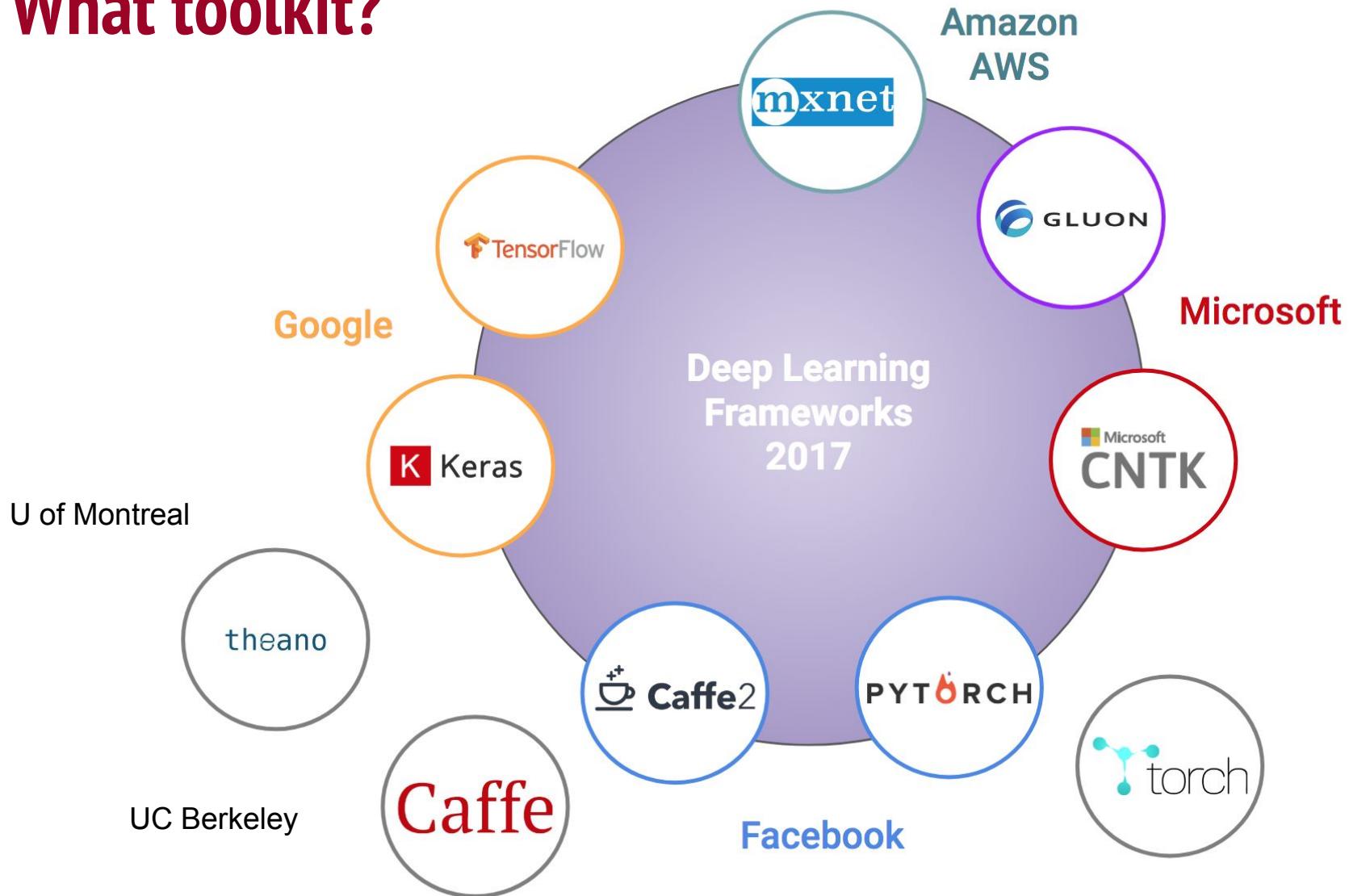
Layer 4

What toolkit

Tradeoff between customizability and ease of use

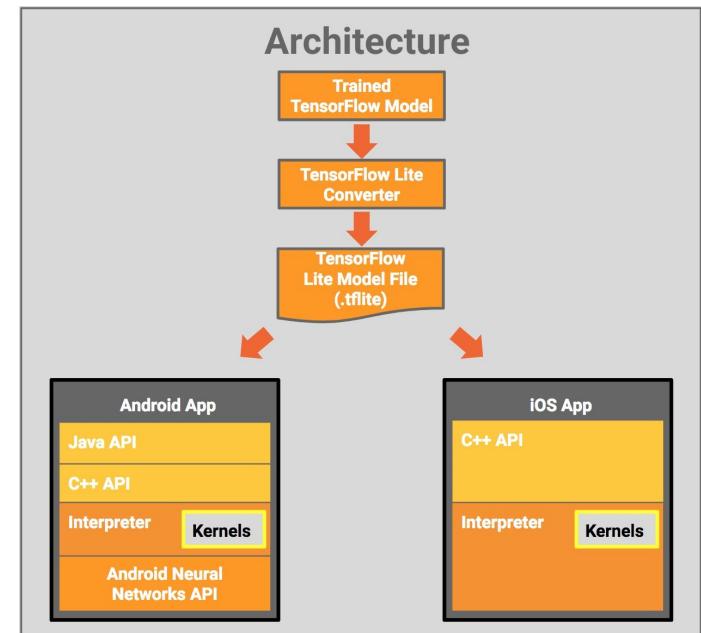


What toolkit?



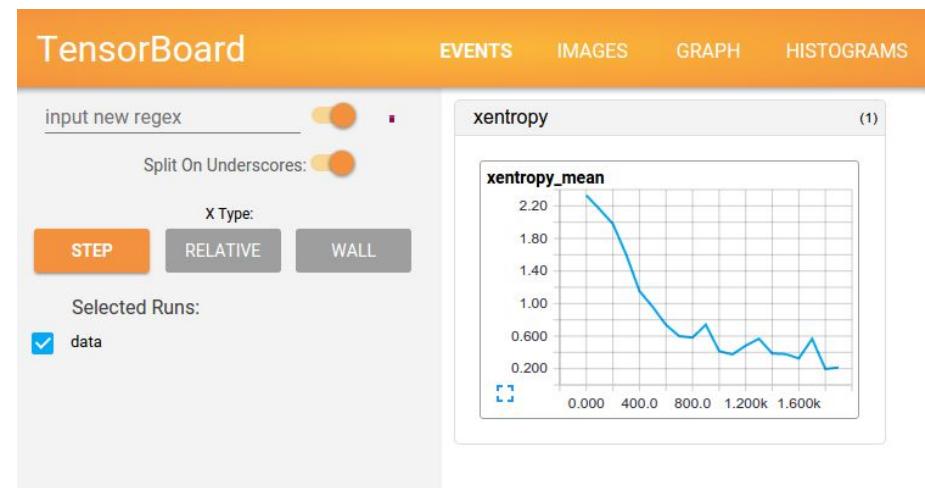
Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: pytorch
- Easiest to deploy: tensorflow
 - Tensorflow lite for mobile
 - TensorRT support
- Best tools: TensorFlow
 - Tensorboard



Which?

- Easiest to use and play with deep learning: Keras
- Easiest to use and tweak: pytorch
- Easiest to deploy: tensorflow
 - Tensorflow lite for mobile
 - TensorRT support
- Best tools: TensorFlow
 - Tensorboard
- Community: TensorFlow



Keras steps

- Define the network
 - Compile the network
 - Fit the network

```
def get_feedforward_nn():
    input1 = Input(shape=(21,))
    x = Dense(100, activation='relu')(input1)
    x = Dense(100, activation='relu')(x)
    x = Dense(100, activation='relu')(x)
    out = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=input1, outputs=out)
    model.compile(optimizer=Adam(),
                  loss='binary_crossentropy',
                  metrics=['acc'])

    return model

    , y_train, epochs=epochs, batch_size=batch_size, verbose=verbose)

    callbacks_list_feedforward_nn,
    data=(x_val, y_val))
```

Keras is easy!

Dense

[source]

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zero')
```

Just your regular densely-connected NN layer.

`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`).

- **Note:** if the input to the layer has a rank greater than 2, then it is flattened prior to the initial dot product with `kernel`.

Example

```
# as first layer in a sequential model:  
model = Sequential()  
model.add(Dense(32, input_shape=(16,)))  
# now the model will take as input arrays of shape (*, 16)  
# and output arrays of shape (*, 32)
```

```
# after the first layer, you don't need to specify  
# the size of the input anymore:  
model.add(Dense(32))
```

Dropout

[source]

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

Applies Dropout to the input.

Dropout consists in randomly setting a fraction `rate` of input units to 0 at each update during training time, which helps prevent overfitting.

Arguments

- `rate`: float between 0 and 1. Fraction of the input units to drop.
- `noise_shape`: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input.
For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
- `seed`: A Python integer to use as random seed.