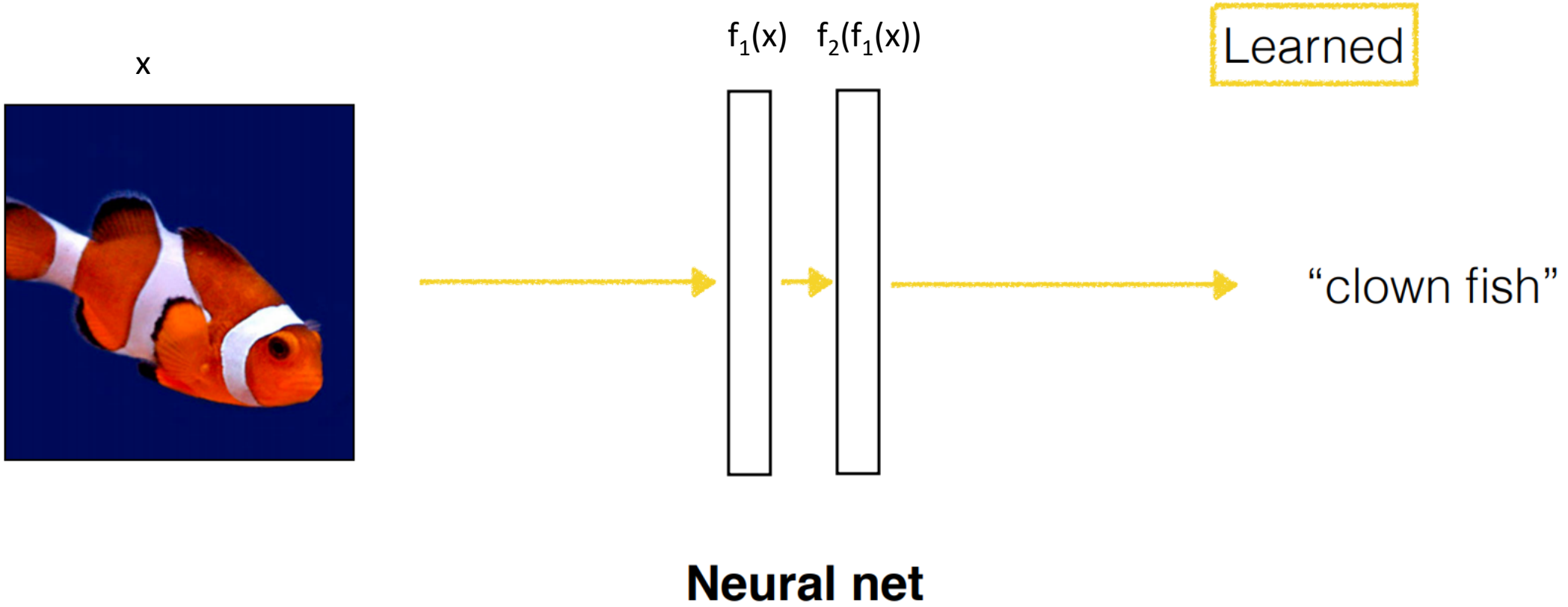
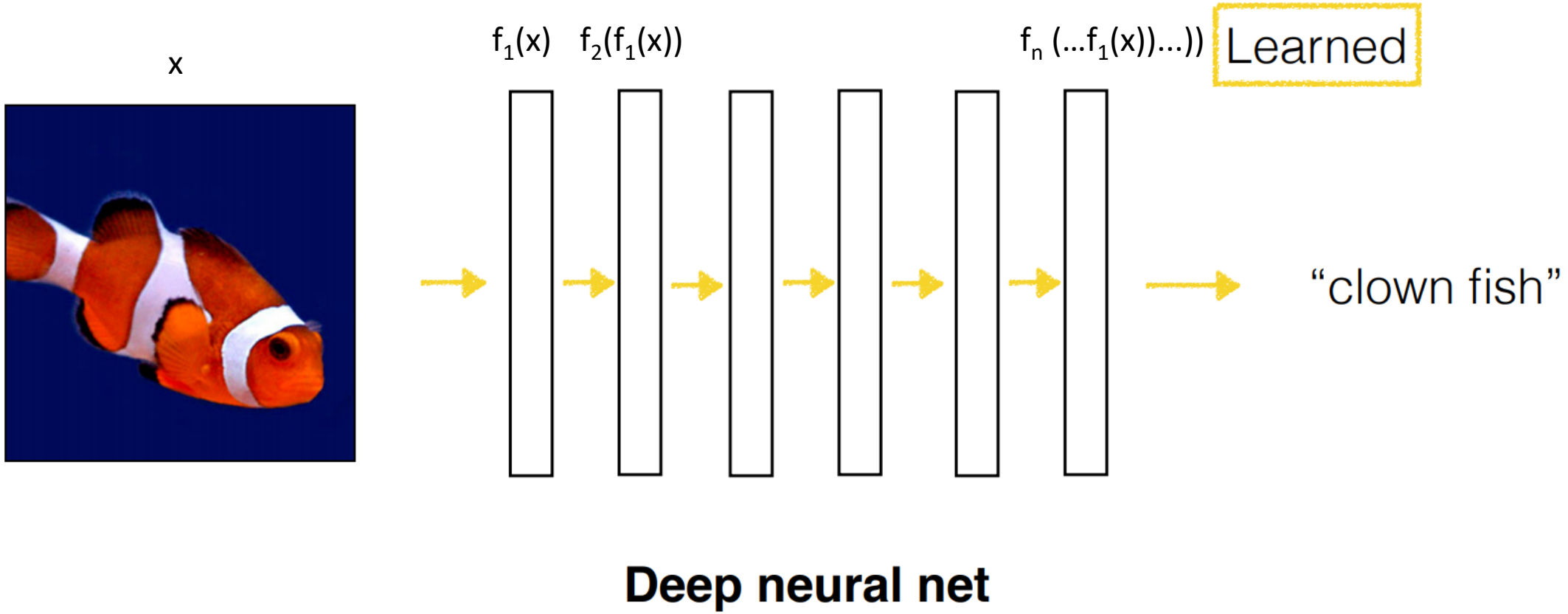
Neural Networks

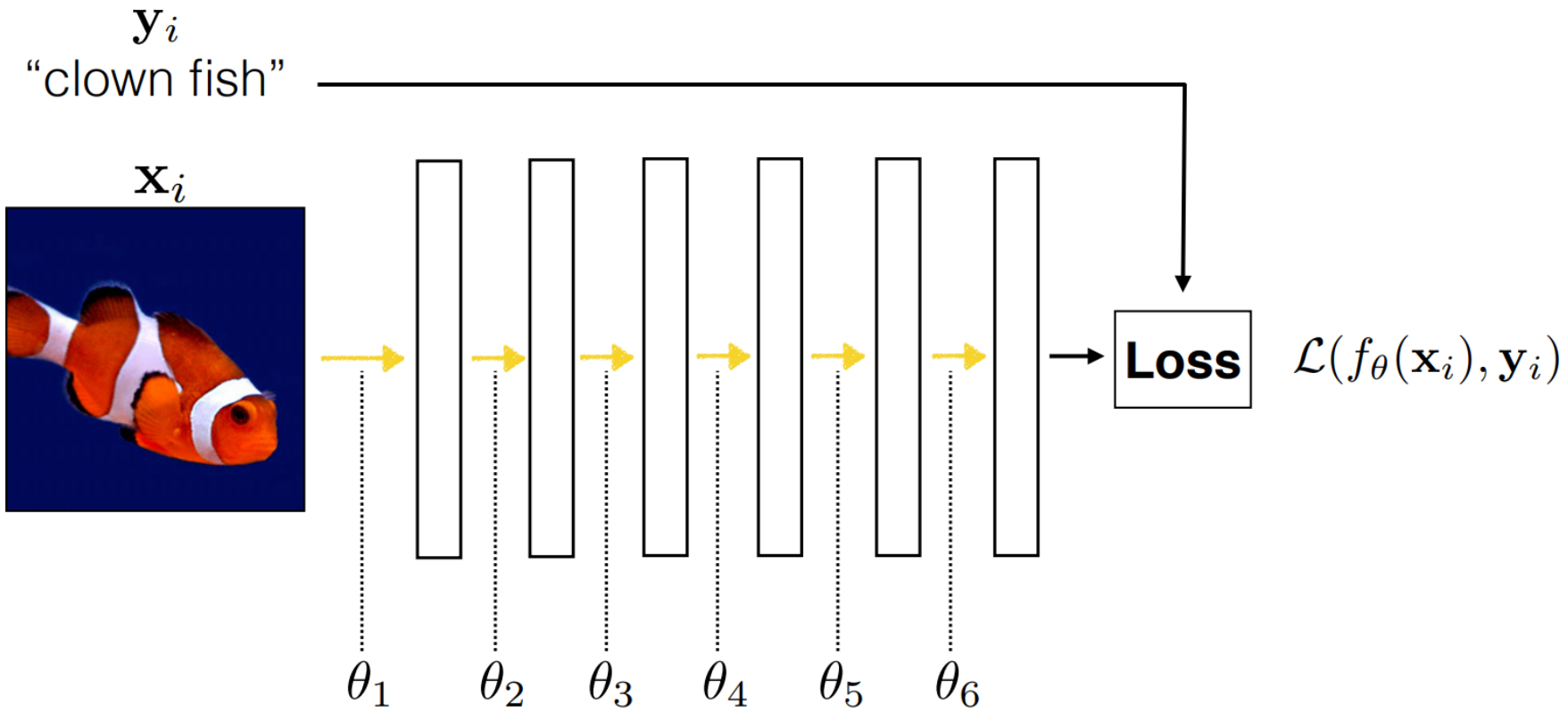
Object Recognition



Object Recognition



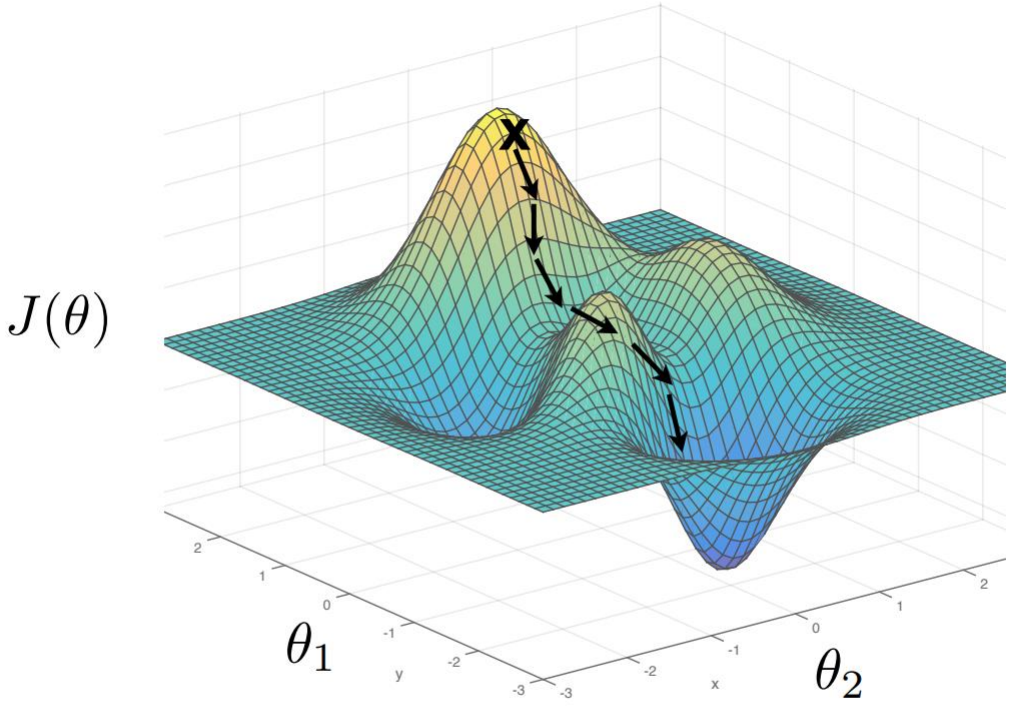
Object Recognition



$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Gradient Descent

- Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient



$$\theta^* = \arg\min_{\theta} J(\theta)$$

Gradient Descent

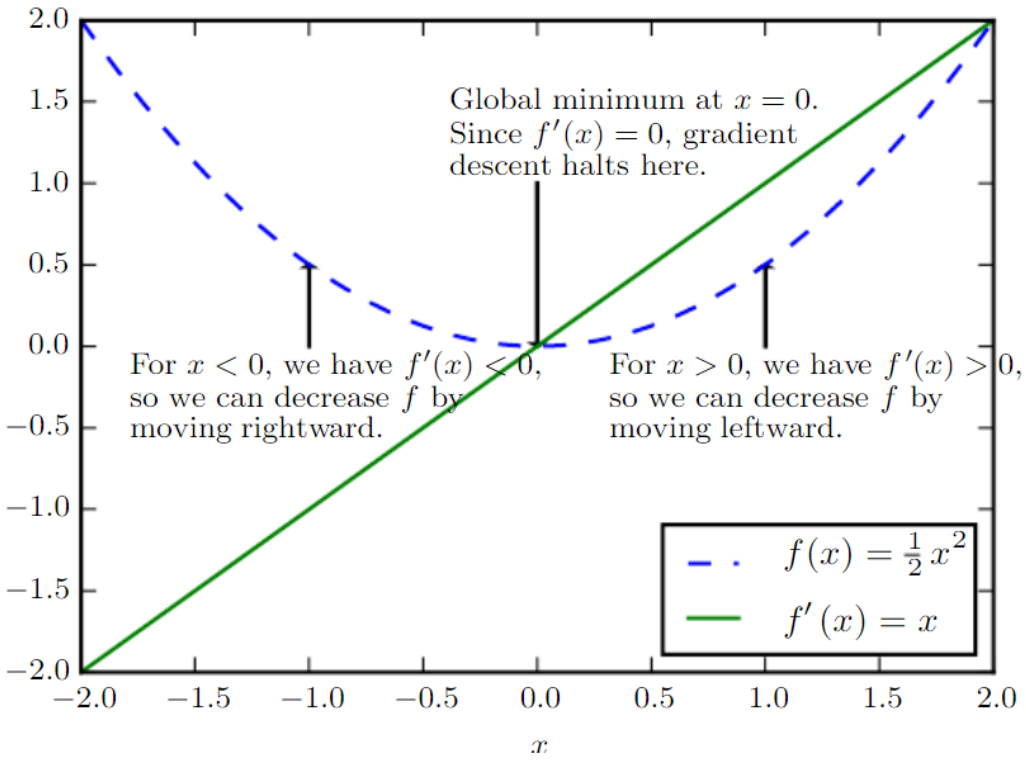


Figure 4.1: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

Gradient Descent

$$\theta^* = \arg\min_{\theta} \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \eta_t \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta^t}$$

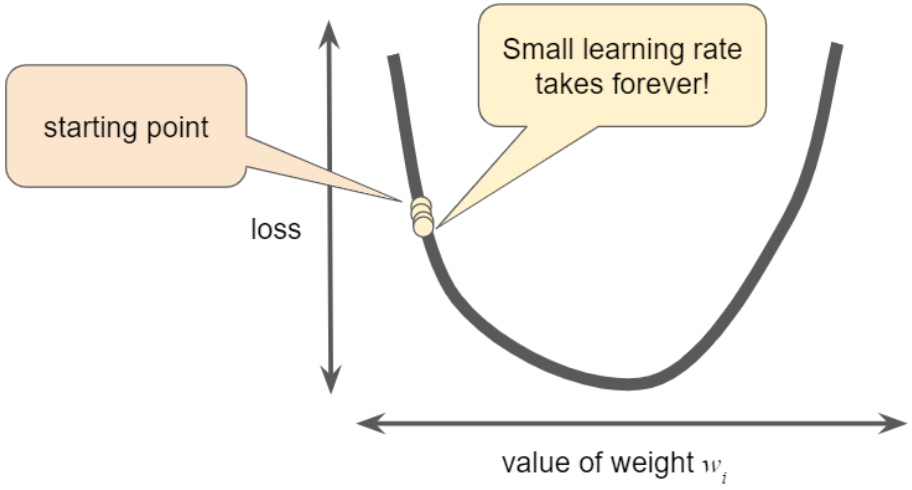
learning rate

Learning Rate

$$\theta^{t+1} = \theta^t - \eta_t \frac{\partial J(\theta)}{\partial \theta}\bigg|_{\theta=\theta^t}$$

learning rate

- The learning rate parameter controls how much the optimization algorithm learns at each iteration
- A very low learning algorithm will take a long time to converge

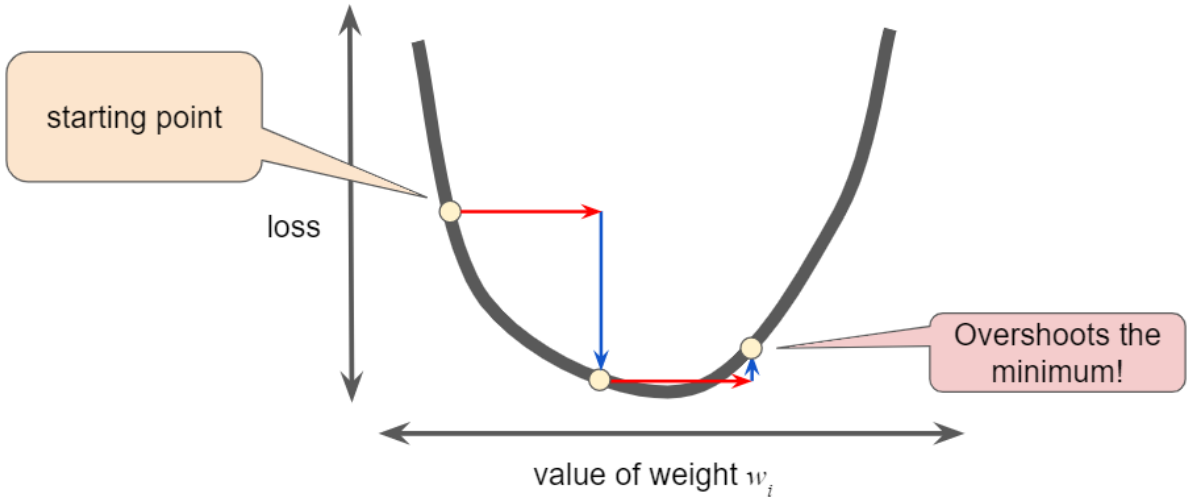


Learning Rate

$$\theta^{t+1} = \theta^t - \eta_t \frac{\partial J(\theta)}{\partial \theta}\bigg|_{\theta=\theta^t}$$

learning rate

- The learning rate parameter controls how much the optimization algorithm learns at each iteration
- A high learning rate will cause the algorithm to miss the optimal solution



Learning Rate

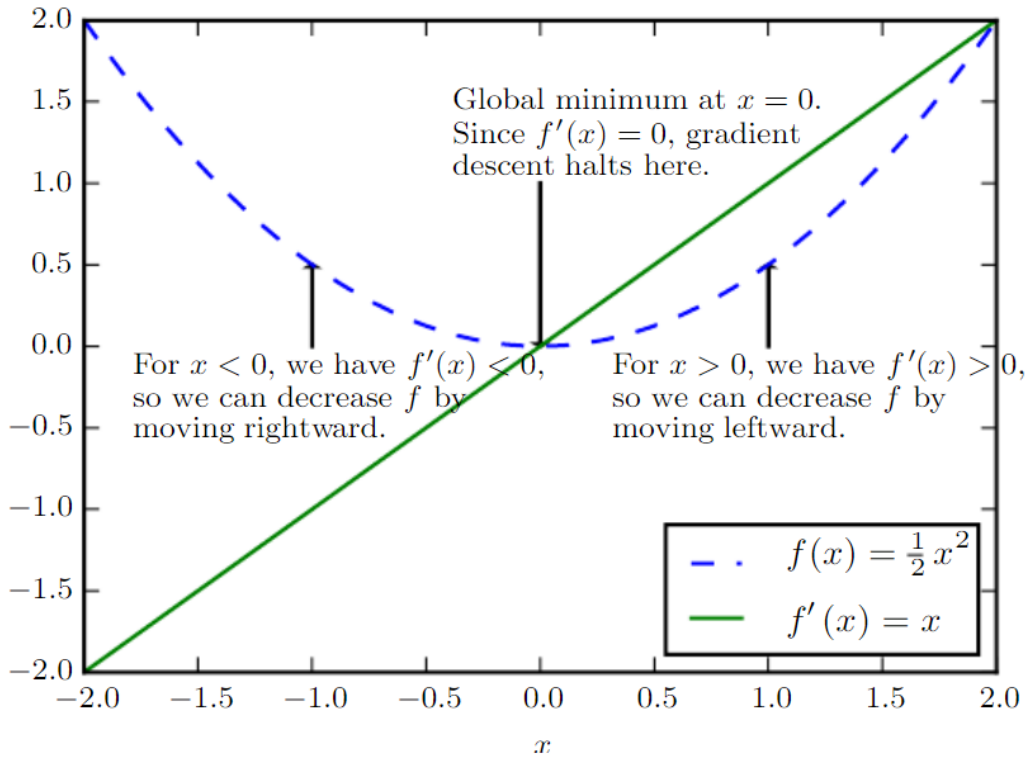


Figure 4.1: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

Learning rate

- Several gradient descent variants exist.
- These methods differ in the way the way they **update** the learning rate.
- A good comparison between these methods is provided in

  https://ruder.io/optimizing-gradient-descent/

  (run the animation)

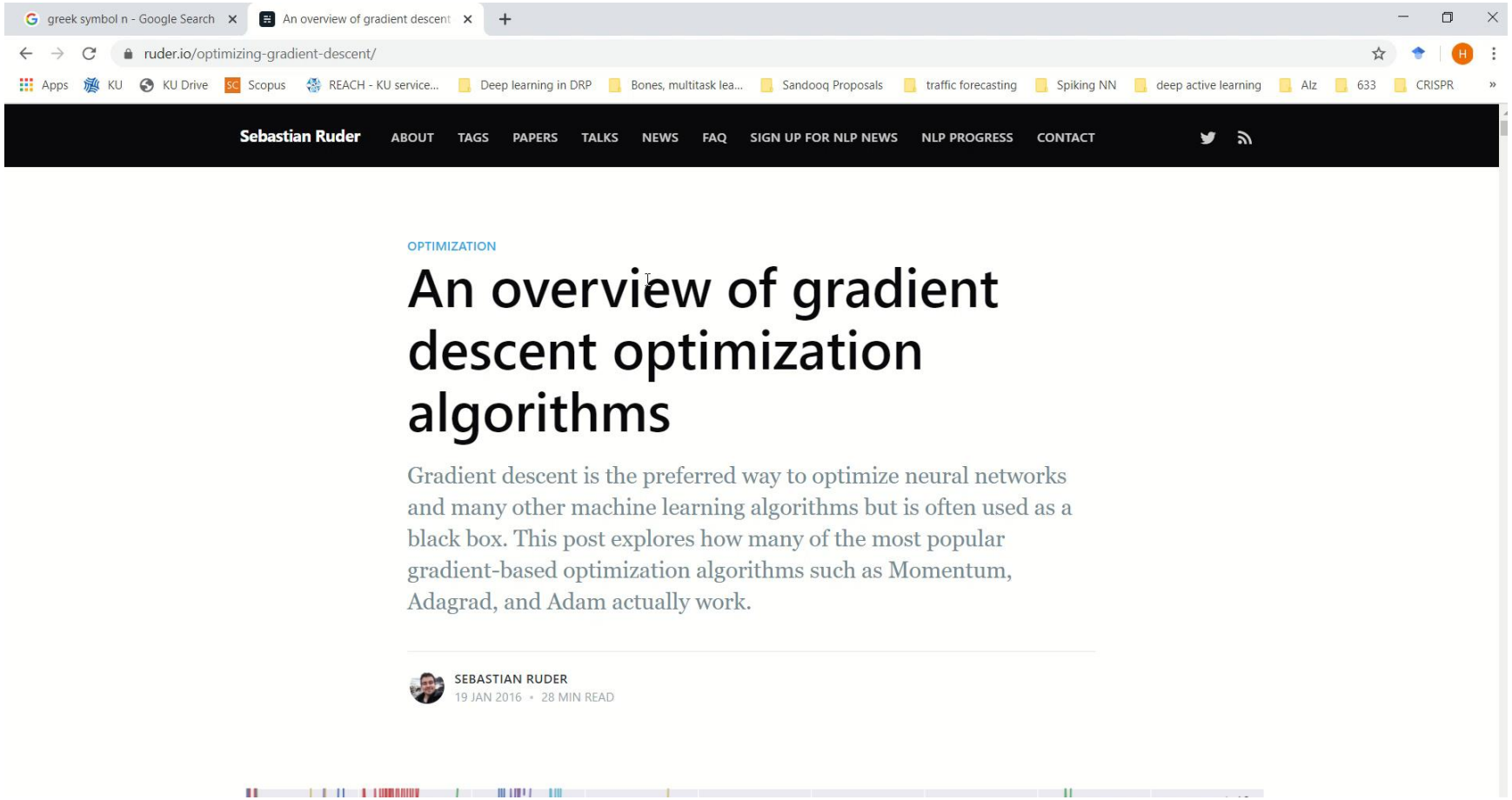
$$\theta^* = \arg\min_{\theta} \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

One iteration of gradient descent:

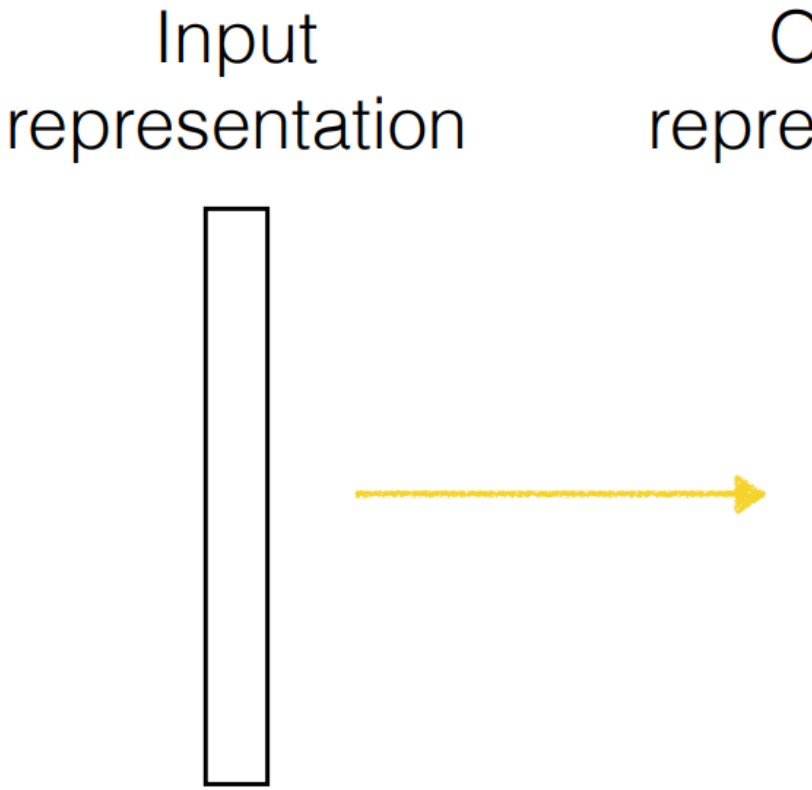
$$\theta^{t+1} = \theta^t - \eta_t \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta^t}$$

learning rate

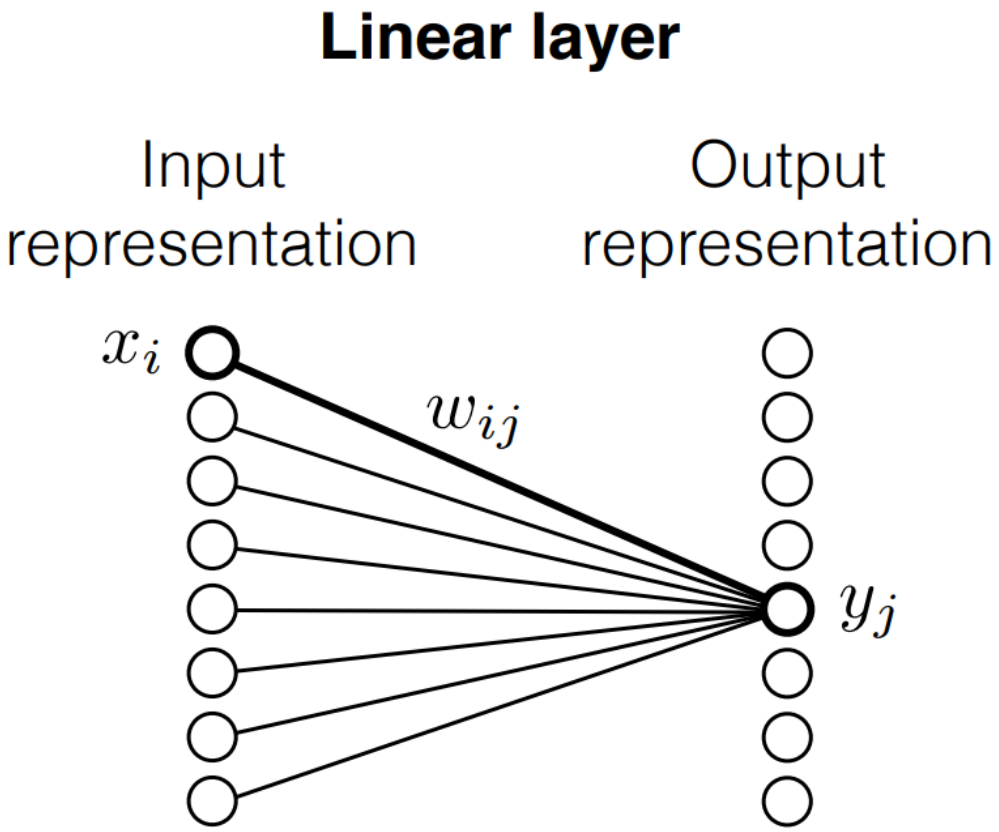
Learning rate



Computation in a neural net

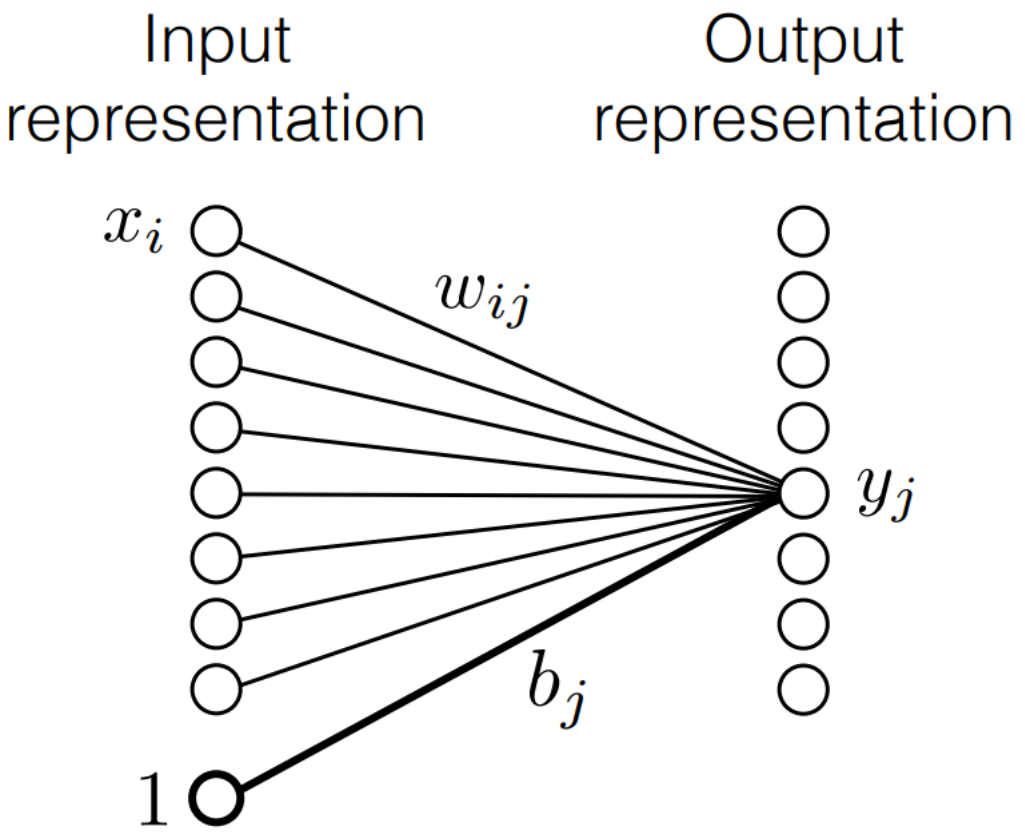


Computation in a neural net



$$y_j = \sum_i w_{ij} x_i$$

Computation in a neural net

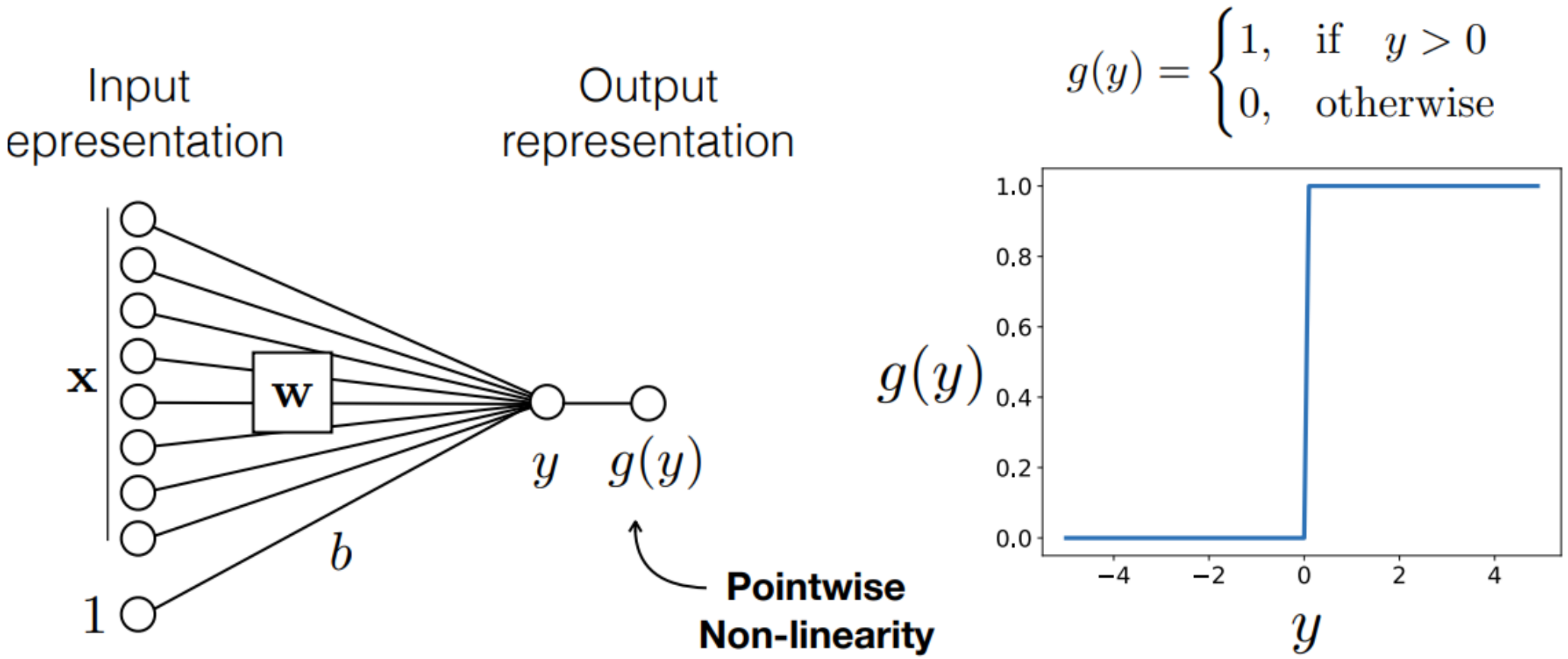


$$y_j = \sum_i w_{ij} x_i + b_j$$

weights

bias

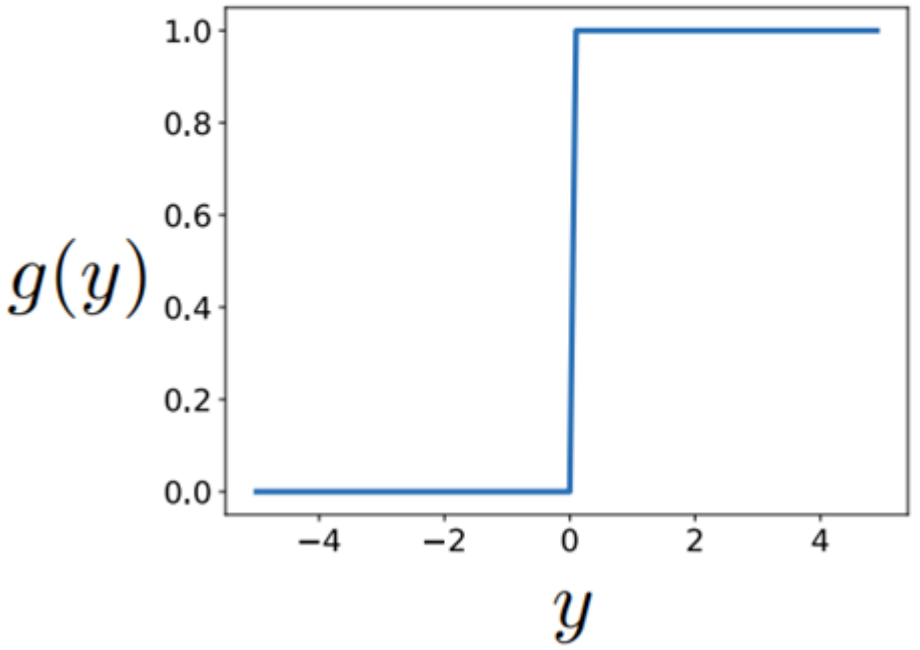
Using a Neural Network for Classification



Using Neural Networks for Classification

- $g(y)$ is non-differentiable function, smooth transition functions are easier to optimize and provide better results

$$g(y) = \begin{cases} 1, & \text{if} \quad y > 0 \\ 0, & \text{otherwise} \end{cases}$$

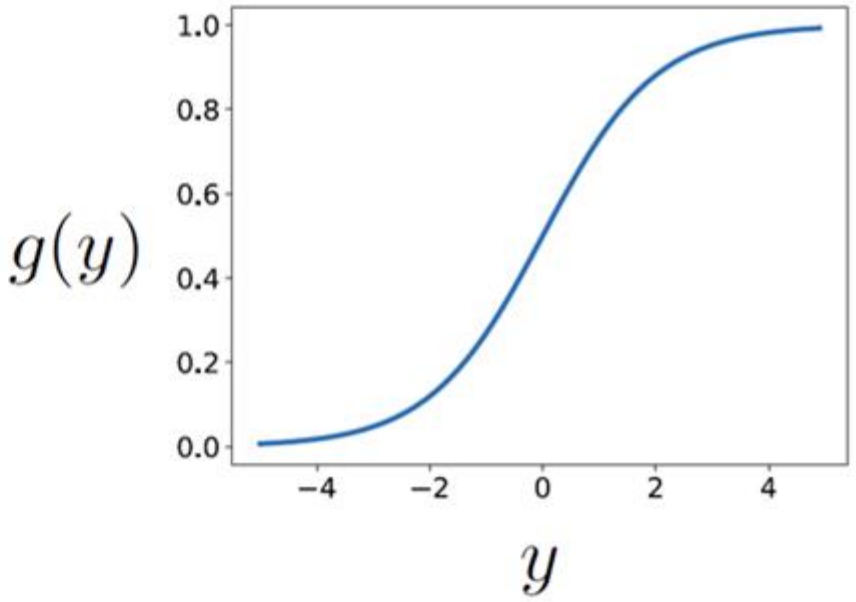


Using Neural Networks for Classification

- Interpretation as firing rate of neuron
- Bounded between [0,1]
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0.5

Sigmoid

$$g(y) = \frac{1}{1 + e^{-y}}$$



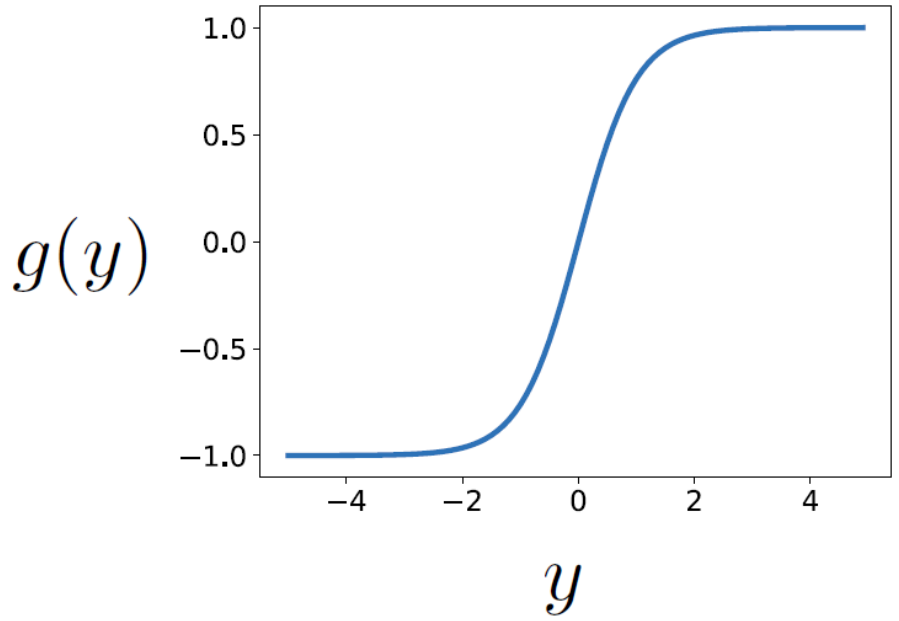
Using Neural Networks for Classification

- Bounded between [-1,+1]
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0
- Preferable to sigmoid

tanh(x) = 2 sigmoid(2x) –1

Tanh

$$g(y) = \frac{e^{y} - e^{-y}}{e^{y} + e^{-y}}$$

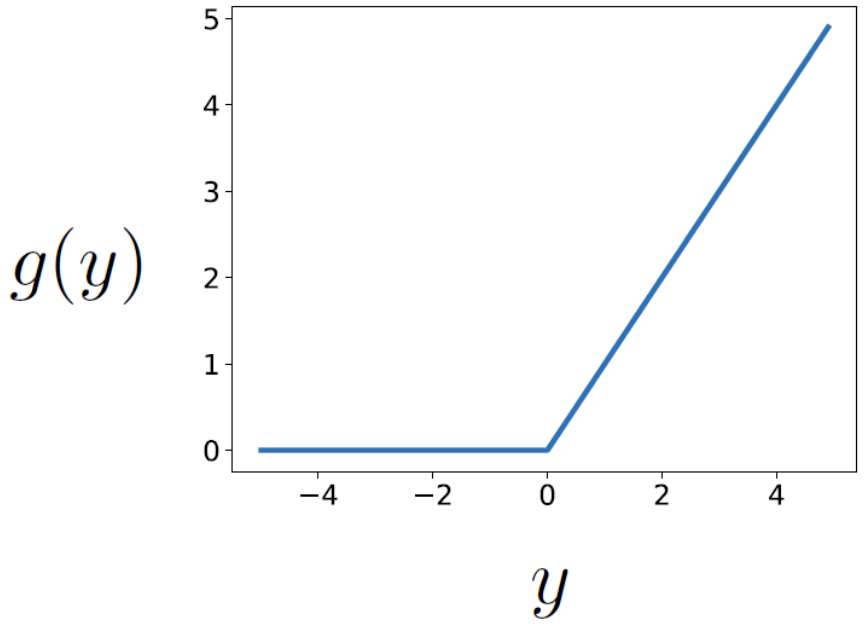


Using Neural Networks for Classification

- Unbounded output (on positive side)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} 0, & \text{if} \quad y < 0 \\ 1, & \text{if} \quad y \geq 0 \end{cases}$
- Also seems to help convergence (see 6x speedup vs tanh in [Krizhevsky et al.])
- Drawback: if strongly in negative region, unit is dead forever (no gradient).
- Default choice: widely used in current models.

Rectified linear unit (ReLU)

$$g(y) = \max(0, y)$$

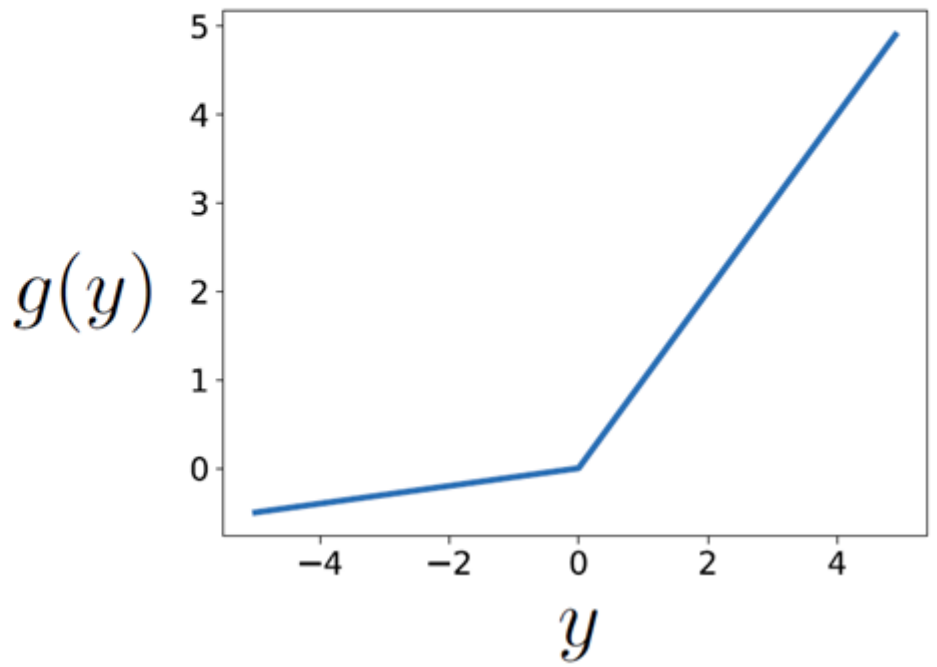


Using Neural Networks for Classification

- where α is small (e.g. 0.02)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} a, & \text{if} \quad y < 0 \\ 1, & \text{if} \quad y \geq 0 \end{cases}$
- Also known as probabilistic ReLU (PReLU)
- Has non-zero gradients everywhere (unlike ReLU)
- α can also be learned (see Kaiming He et al. 2015).

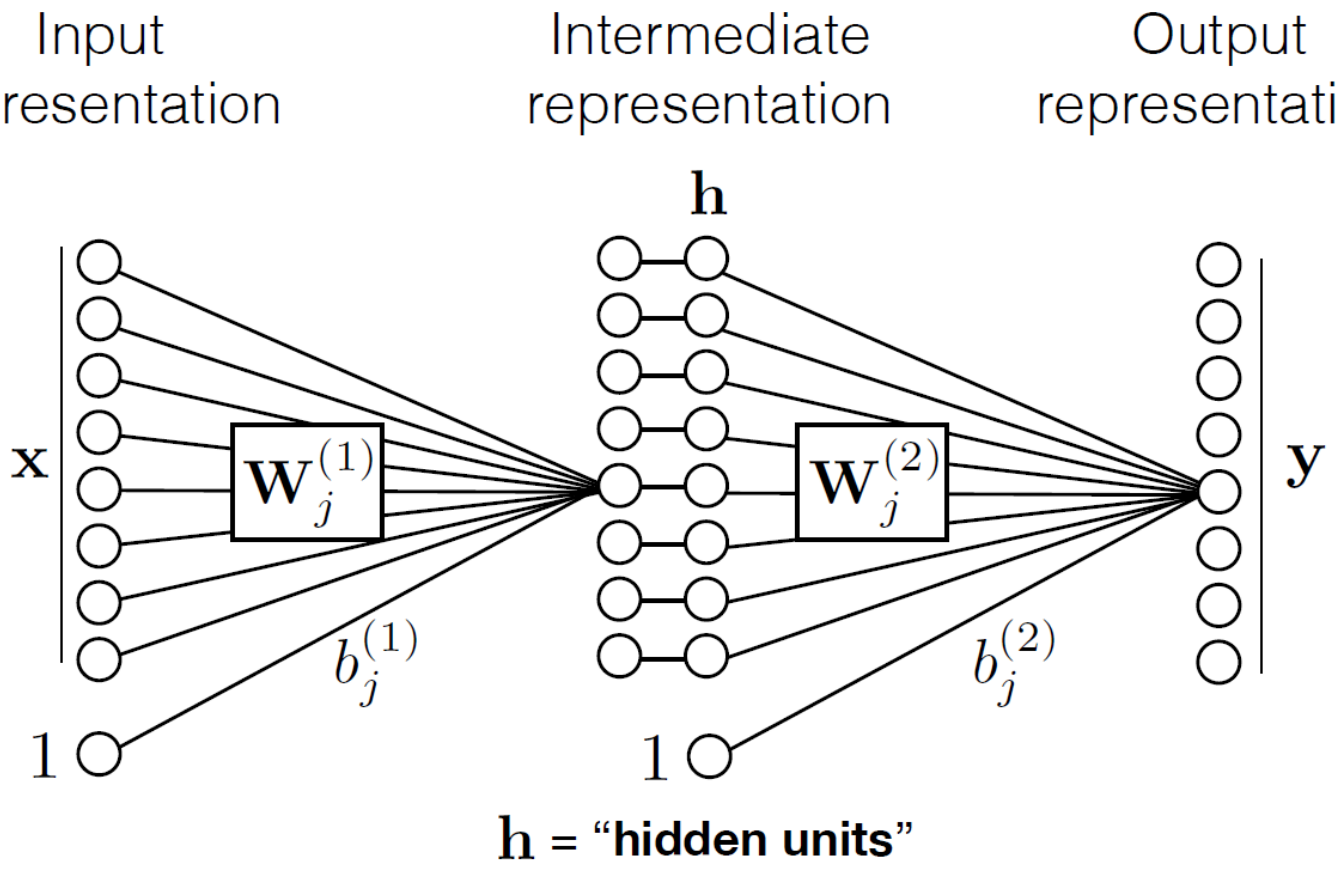
Leaky ReLU

$$g(y) = \begin{cases} \max(0, y), & \text{if} \quad y \geq 0 \\ a \min(0, y), & \text{if} \quad y < 0 \end{cases}$$



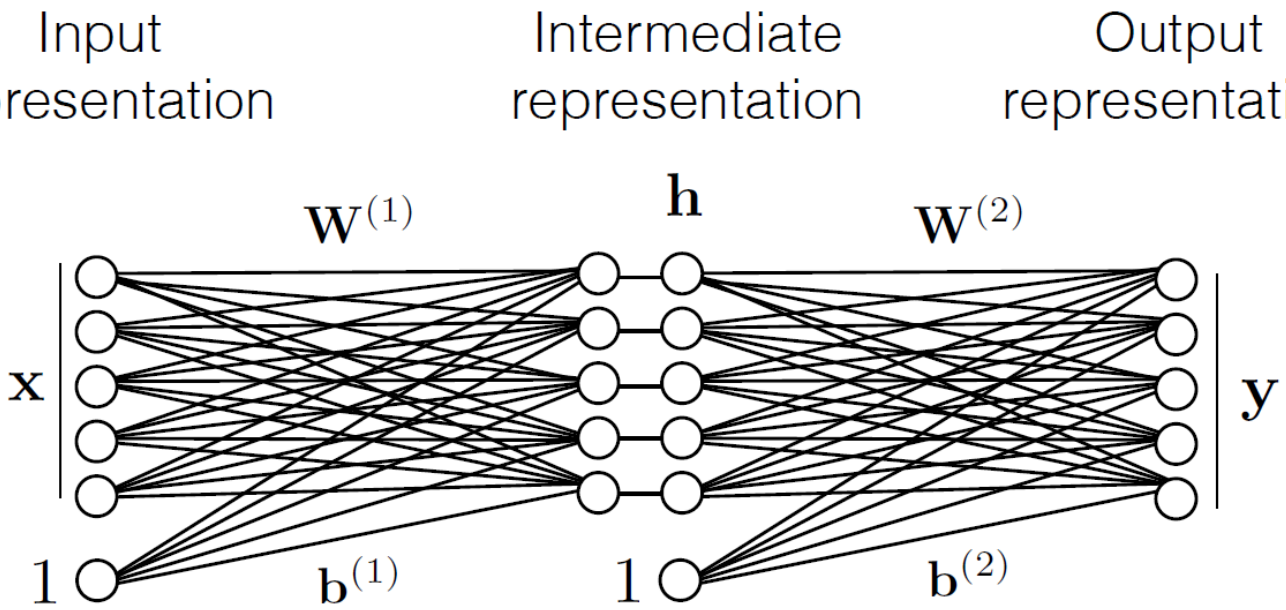
Stacking Layers

By stacking layers, neural networks can provide solutions that goes beyond linear problems.



Stacking Layers

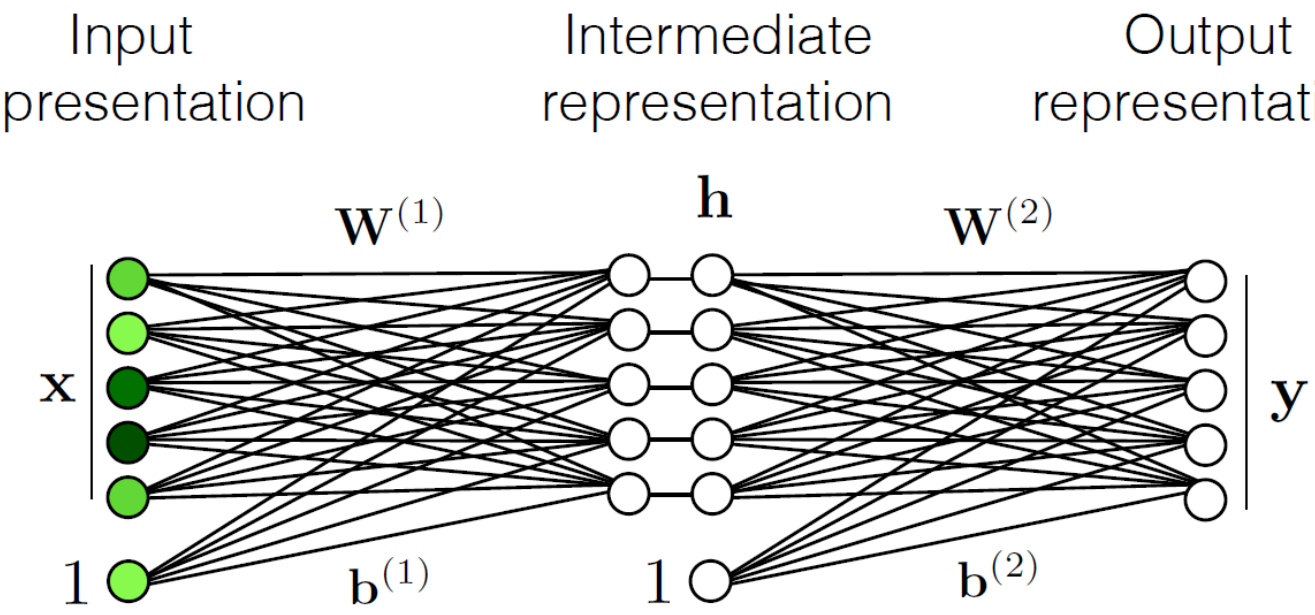
By stacking layers, neural networks can provide solutions that goes beyond linear problems.



$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Stacking Layers



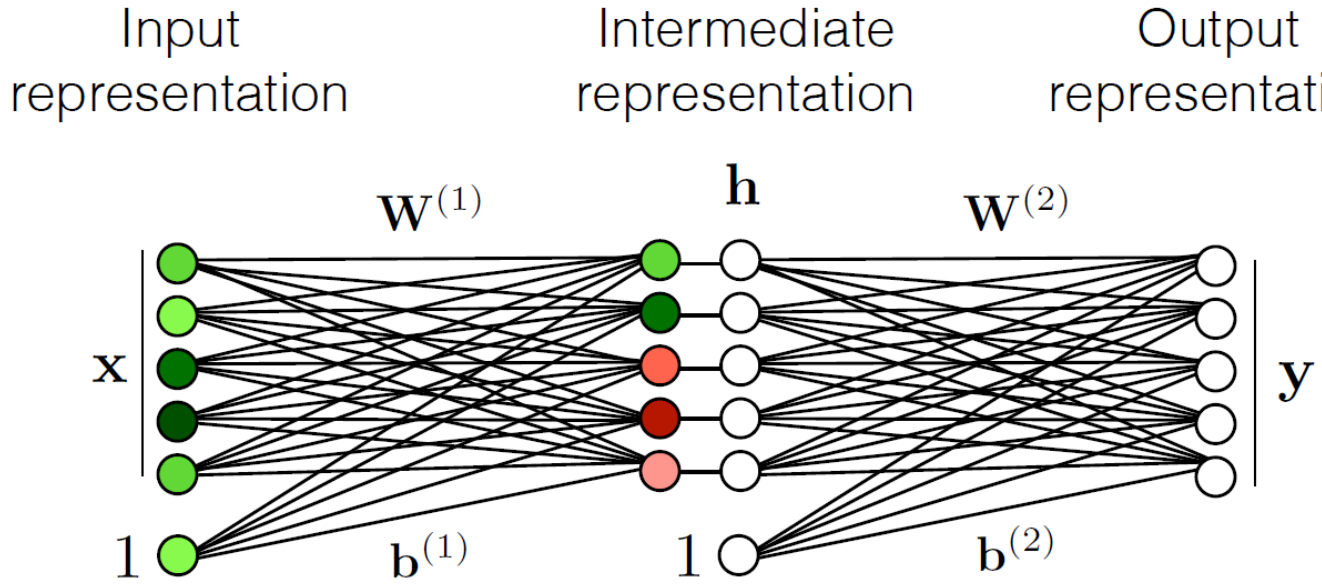
positive

negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Stacking Layers



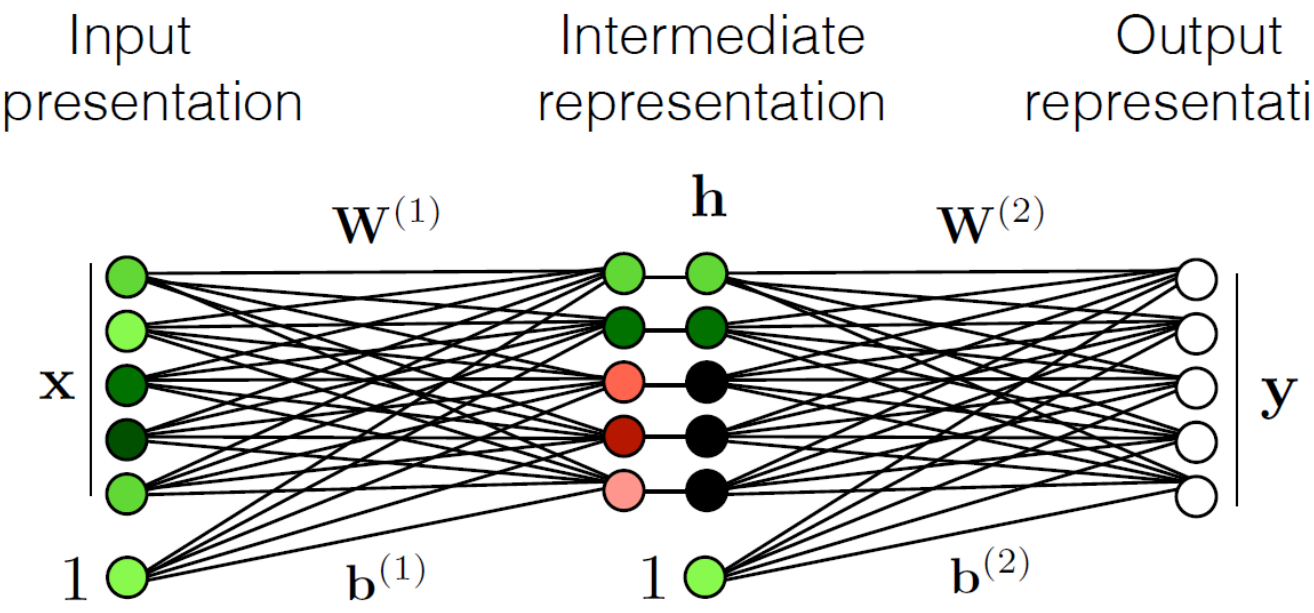
positive

negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Stacking Layers



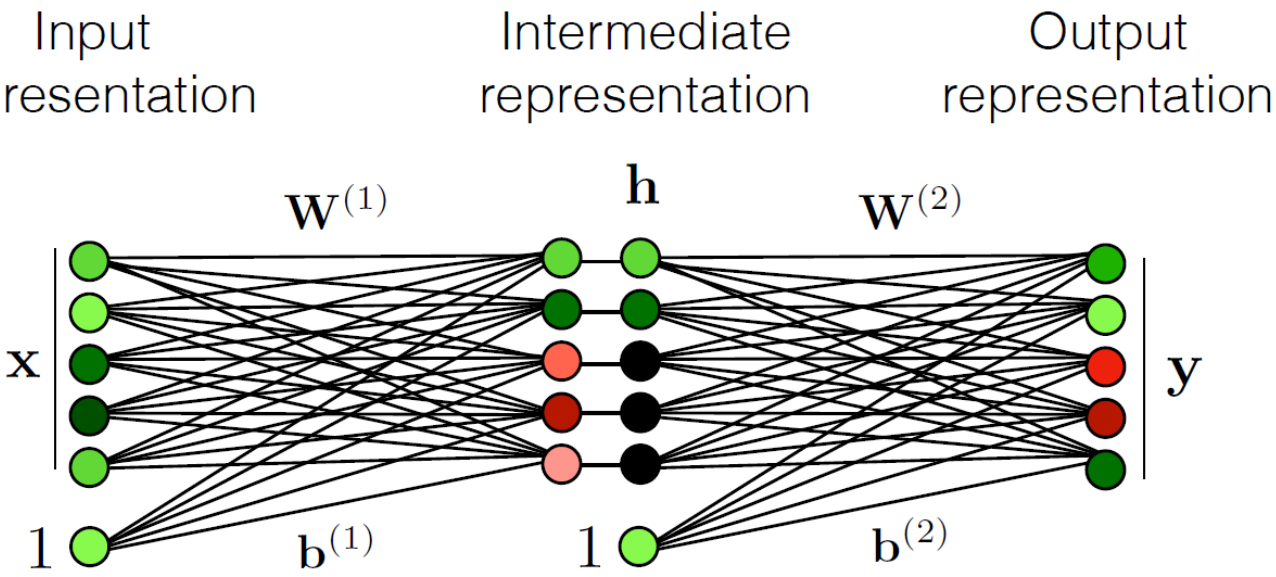
positive

negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Stacking Layers



positive

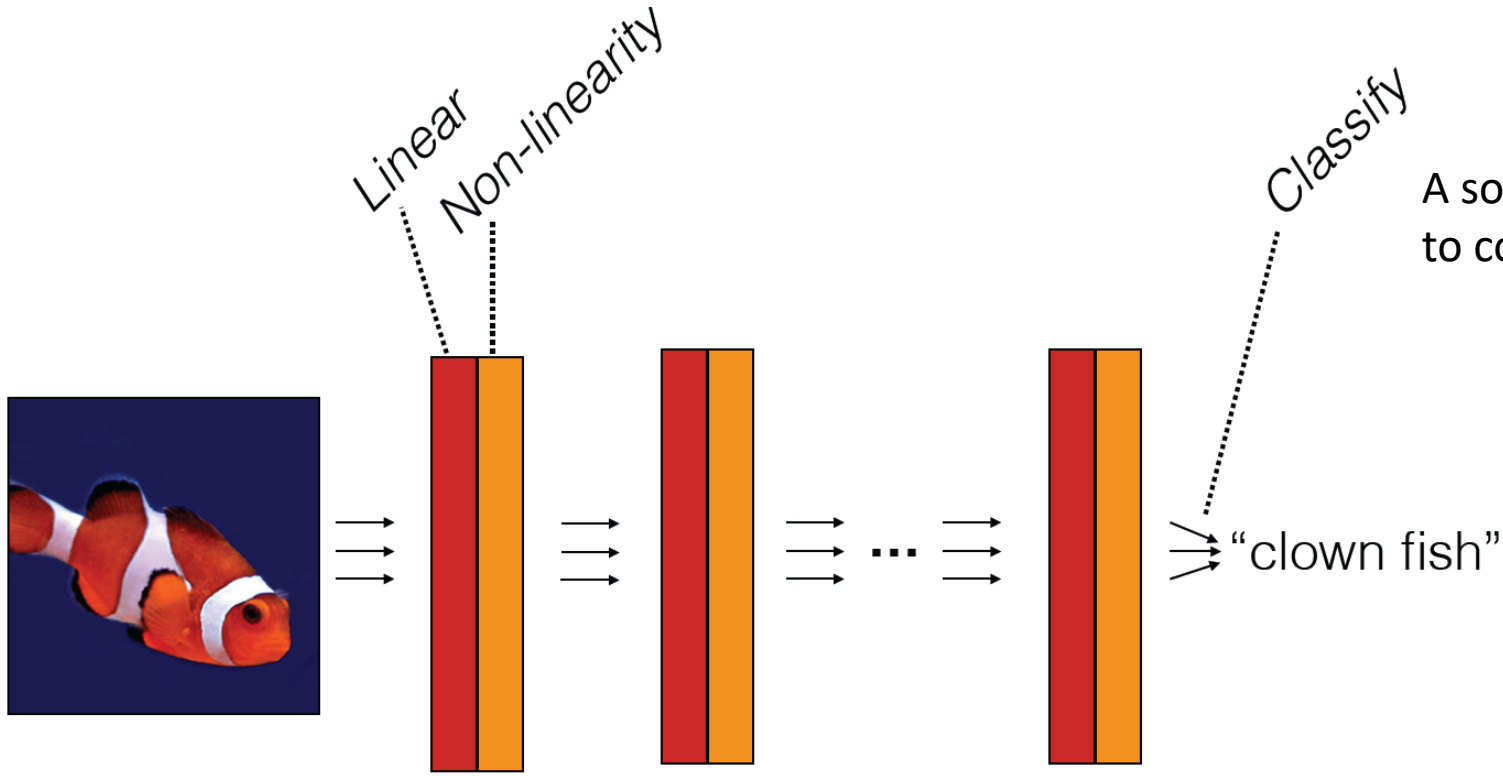
negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

These layers are called **Fully Connected (FC)** layers

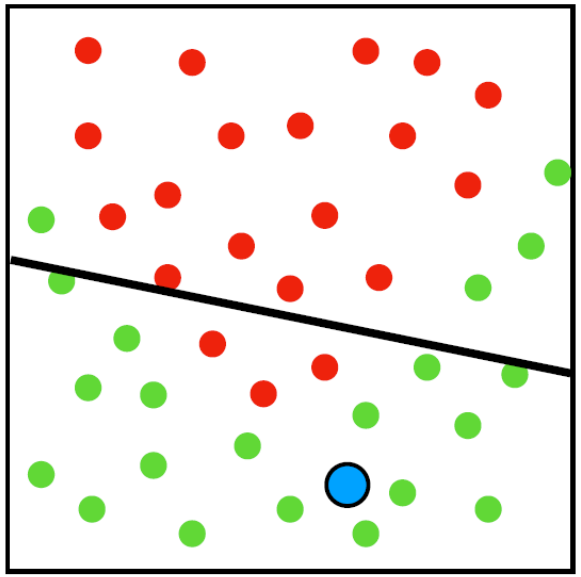
Deep Networks



$$f(\mathbf{x}) = f_L(\ldots f_2(f_1(\mathbf{x})))$$

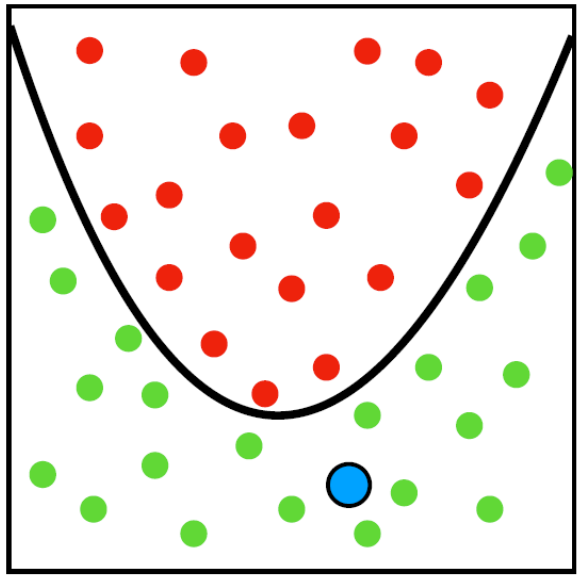
Model Order: Correct number of layers

What class is ● ?



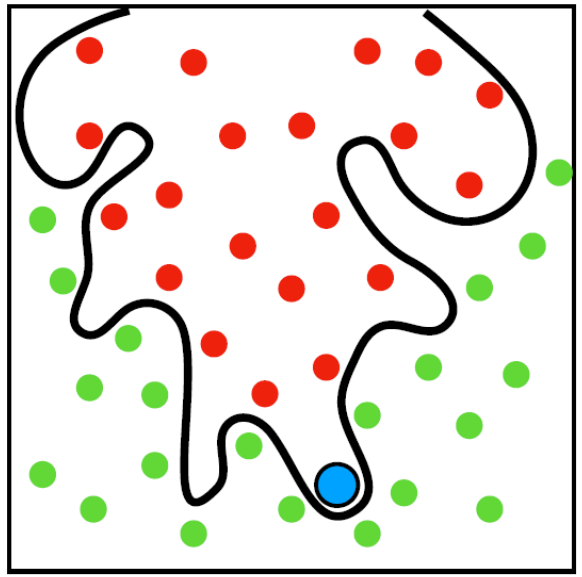
Answer: ●

Underfitting



Answer: ●

Appropriate model



Answer: ●

Overfitting

Classifier layer

Image Classification Using Neural Networks

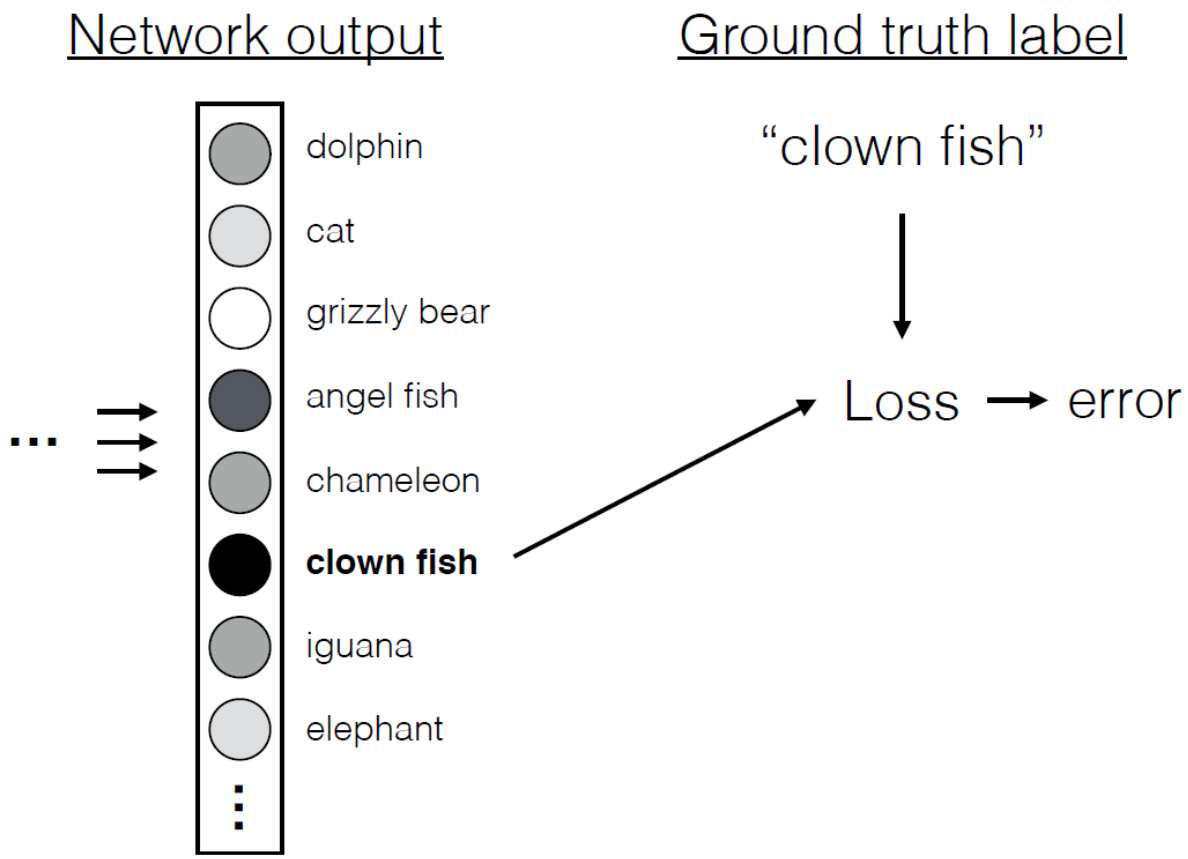


Image Classification Using Neural Networks

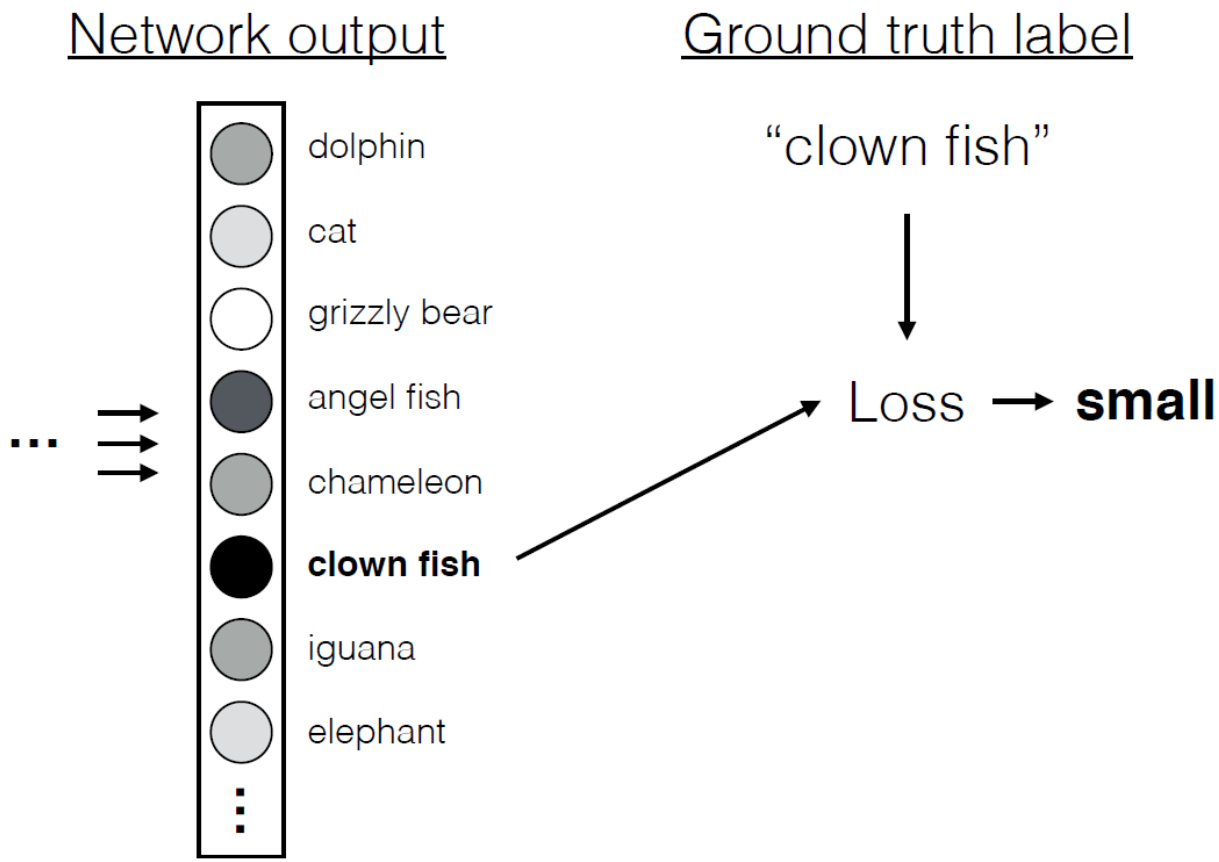


Image Classification Using Neural Networks

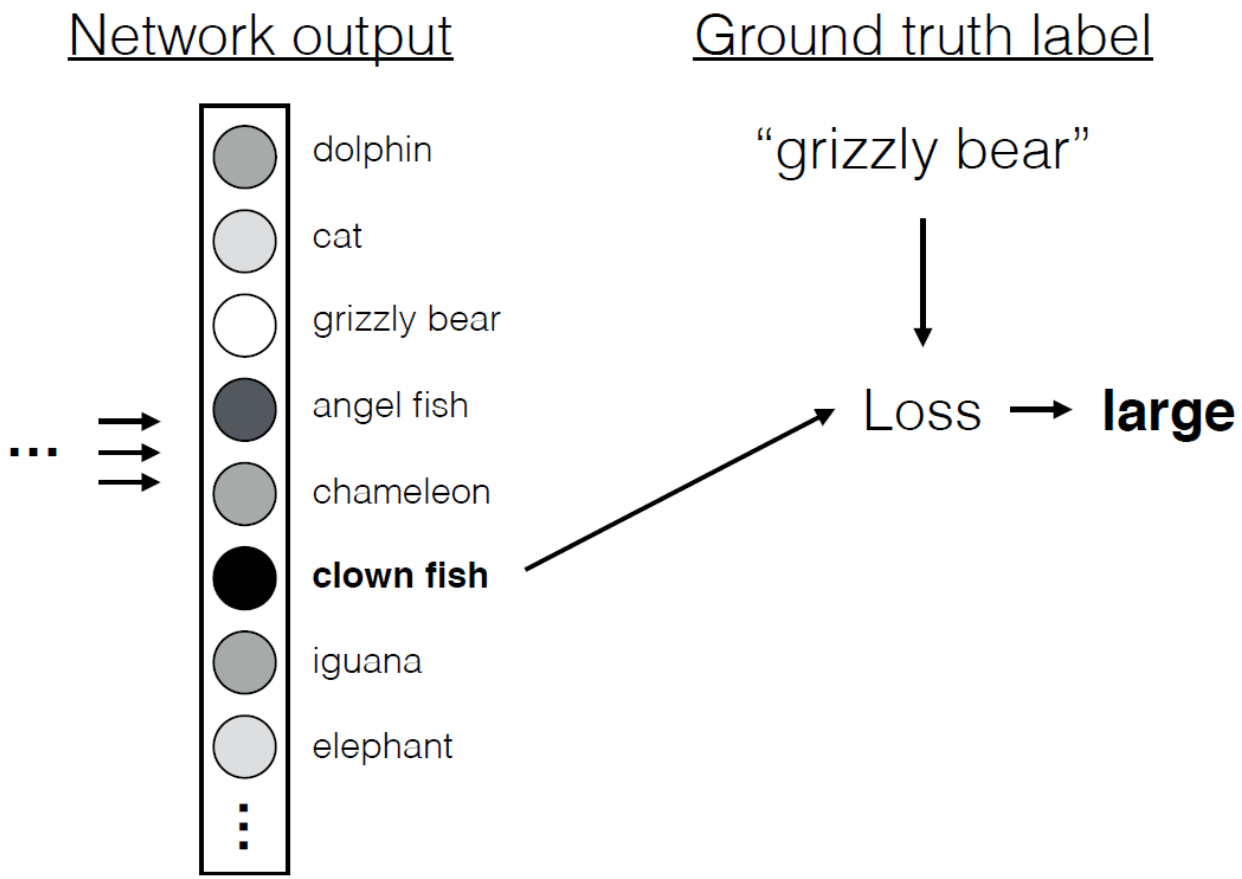


Image Classification Using Neural Networks

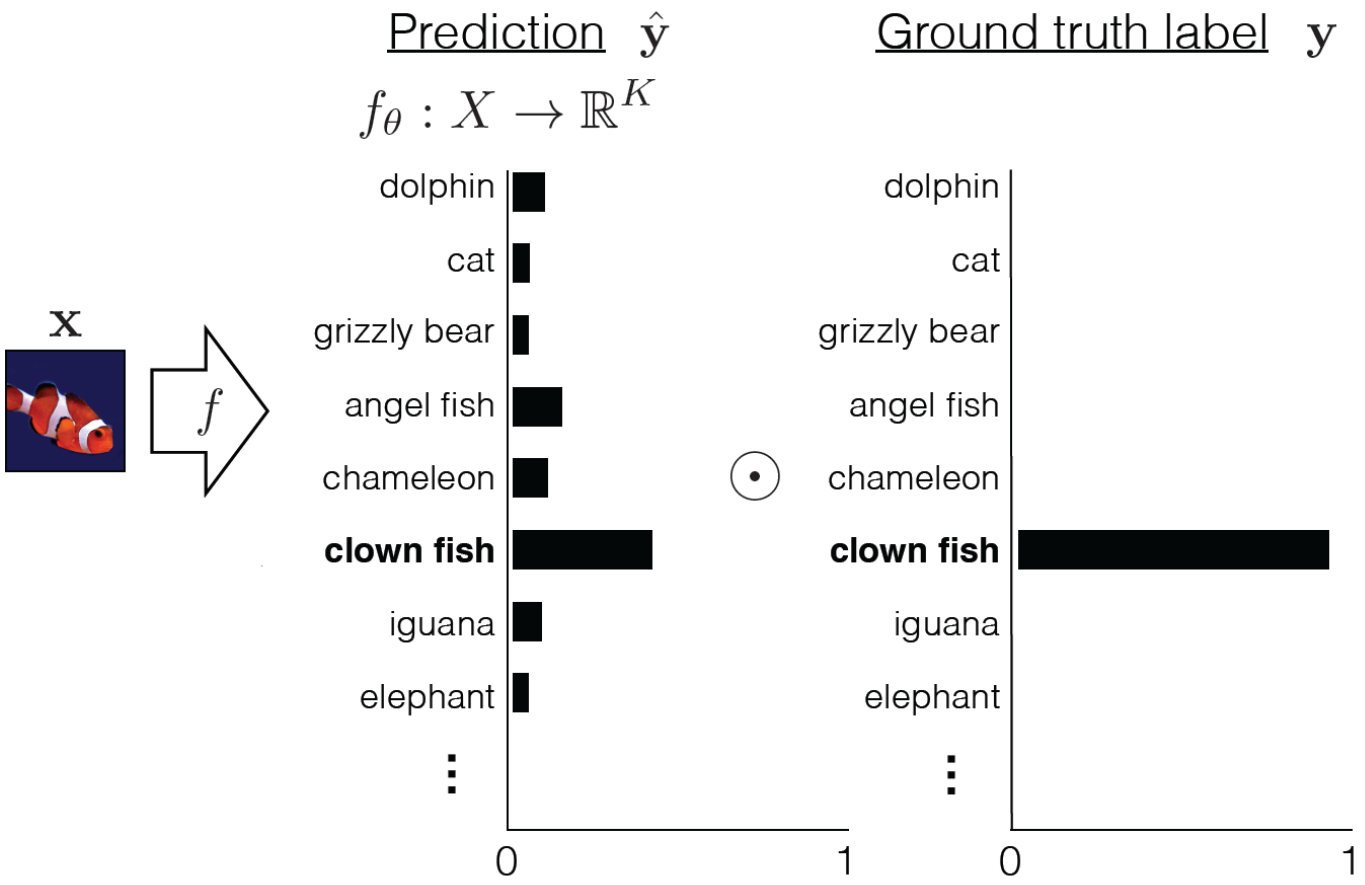
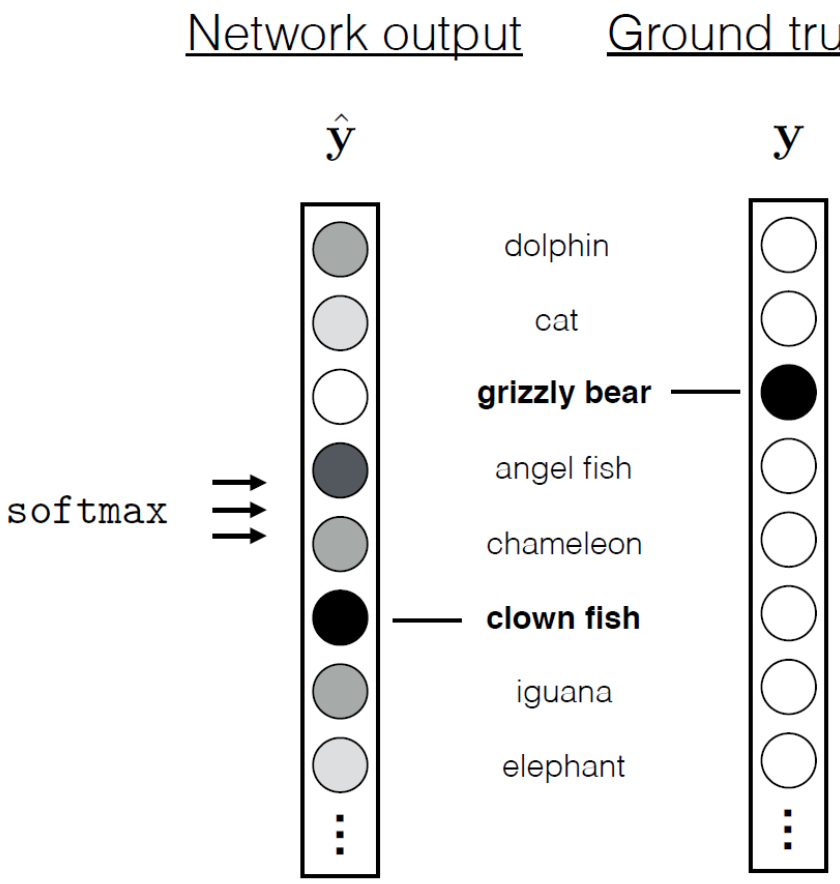


Image Classification Using Neural Networks

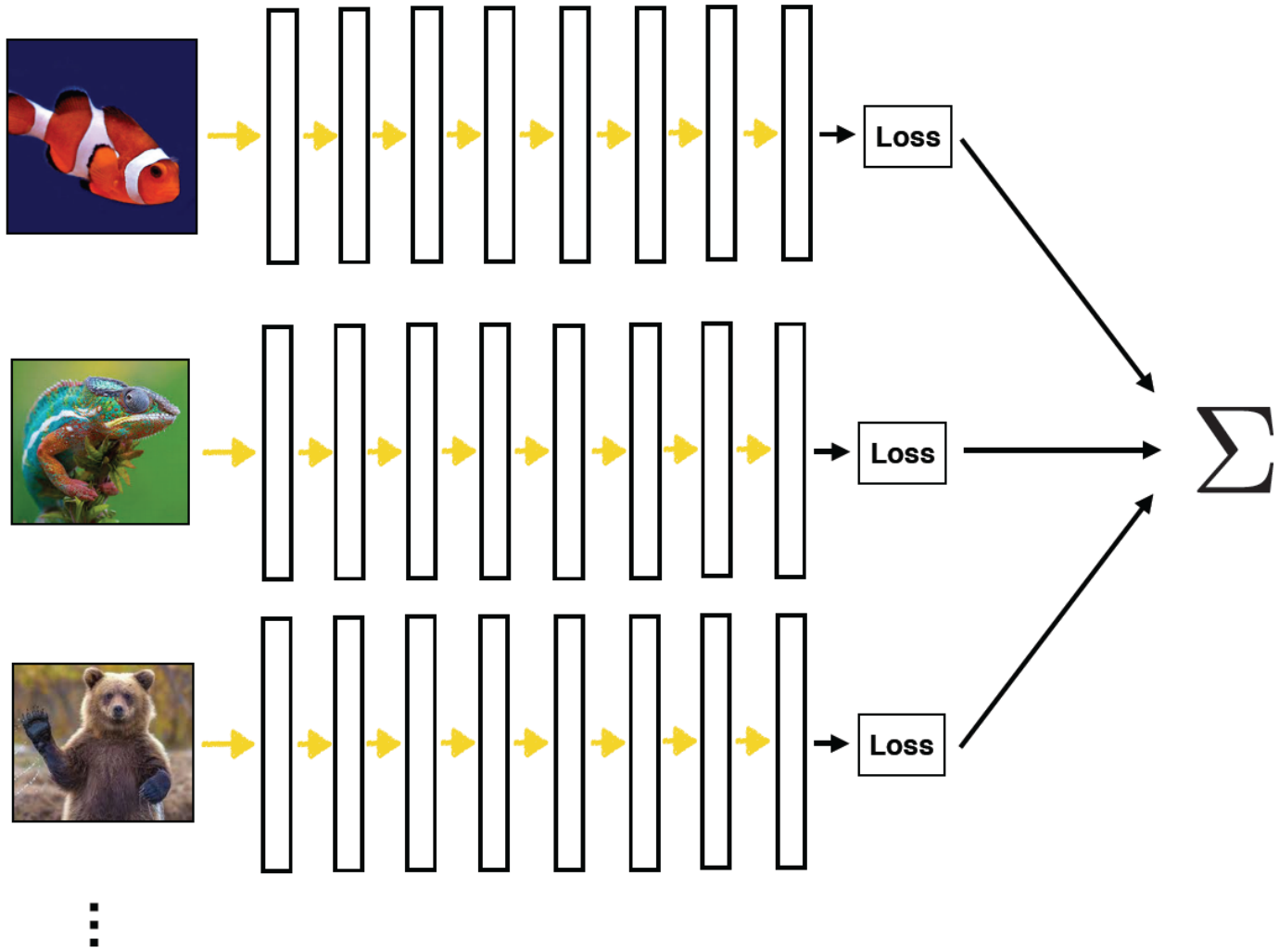


Probability of the observed data under the model

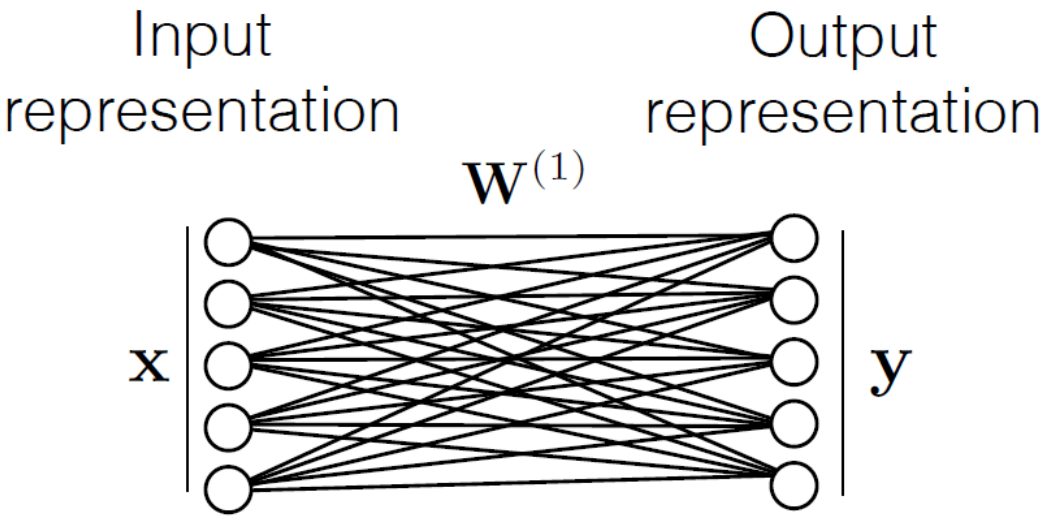
$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

In this example, a very high loss is incurred due to misclassification

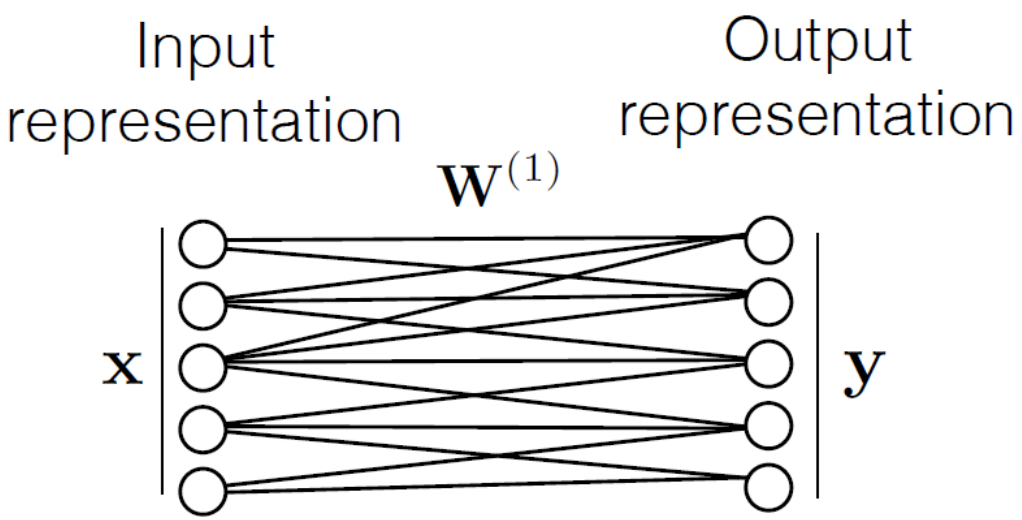
Parallel Processing



Connectivity patterns



Fully connected layer



Locally connected layer (Sparse W)

Convolutional Layer

-1	0	+1
-2	0	+2
-1	0	+1

x filter

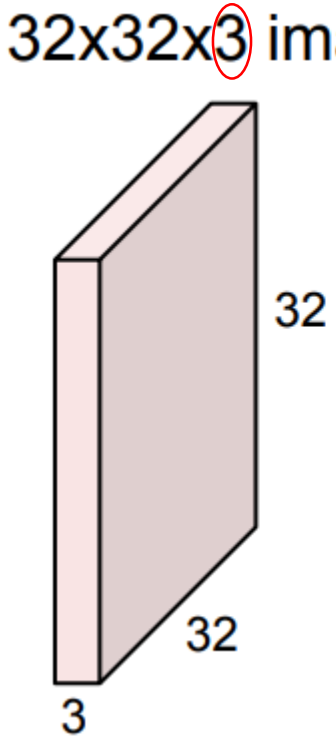
+1	+2	+1
0	0	0
-1	-2	-1

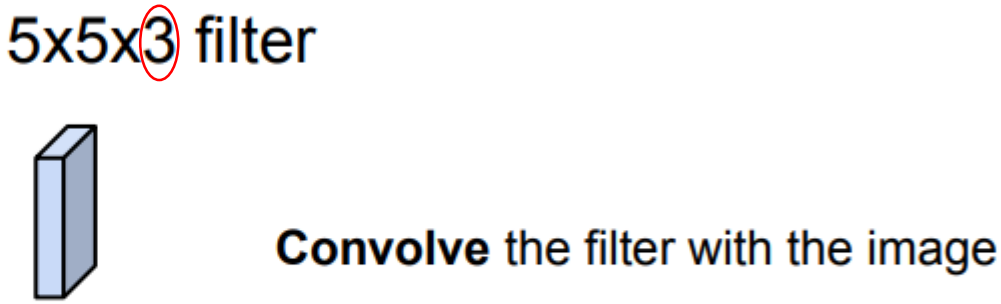
y filter

Sobel X and Y filters

Traditionally, filter weights were designed. Convolutional layers learn filter weights from training data

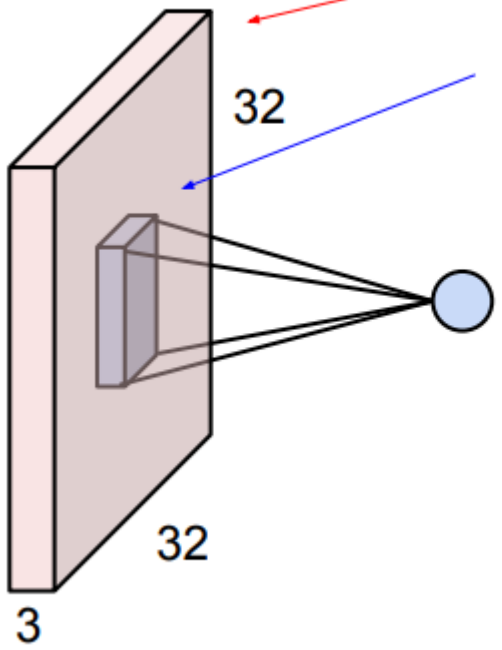
Convolutional Layer





What if the image was grayscale?

Convolutional Layer



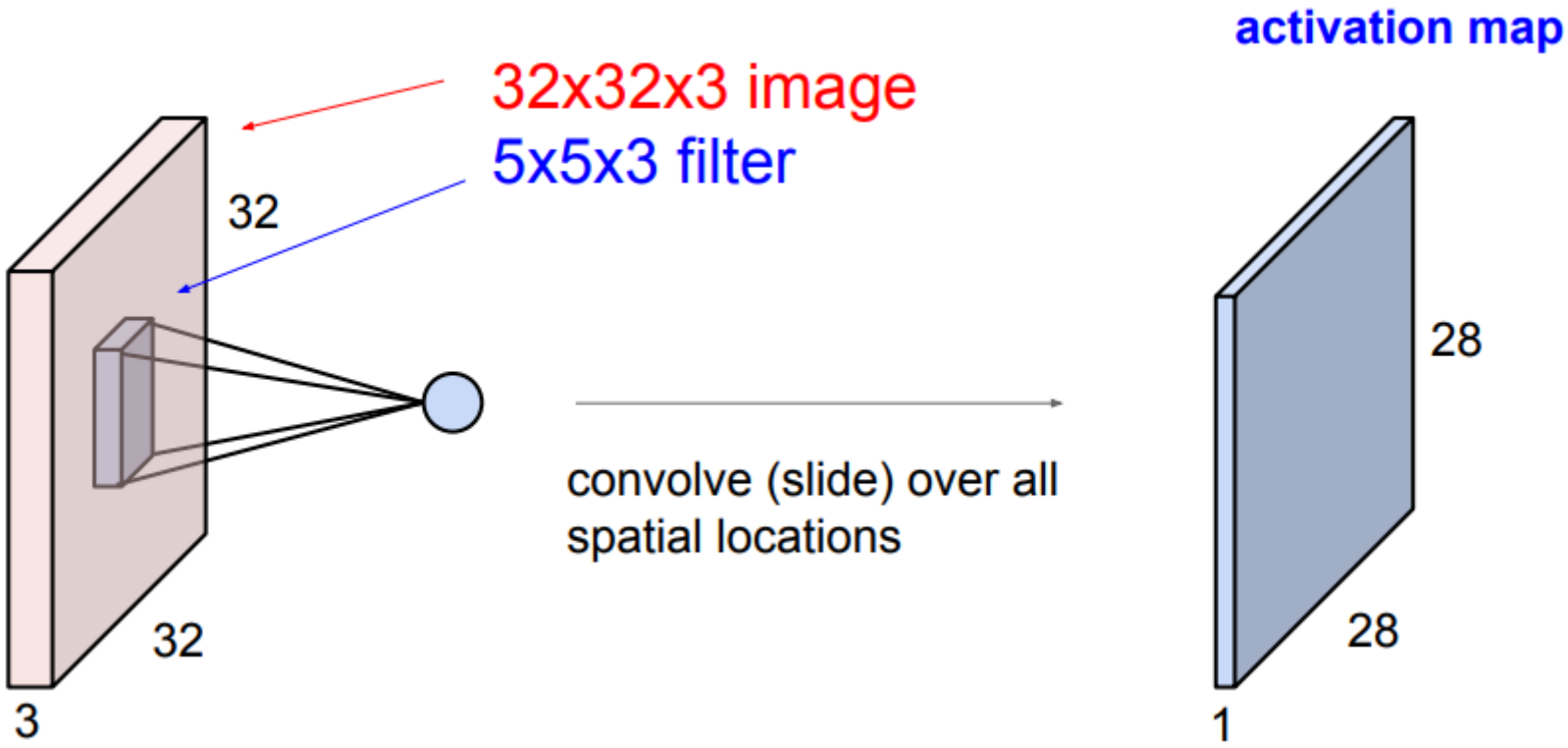
32x32x3 image

5x5x3 filter w

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

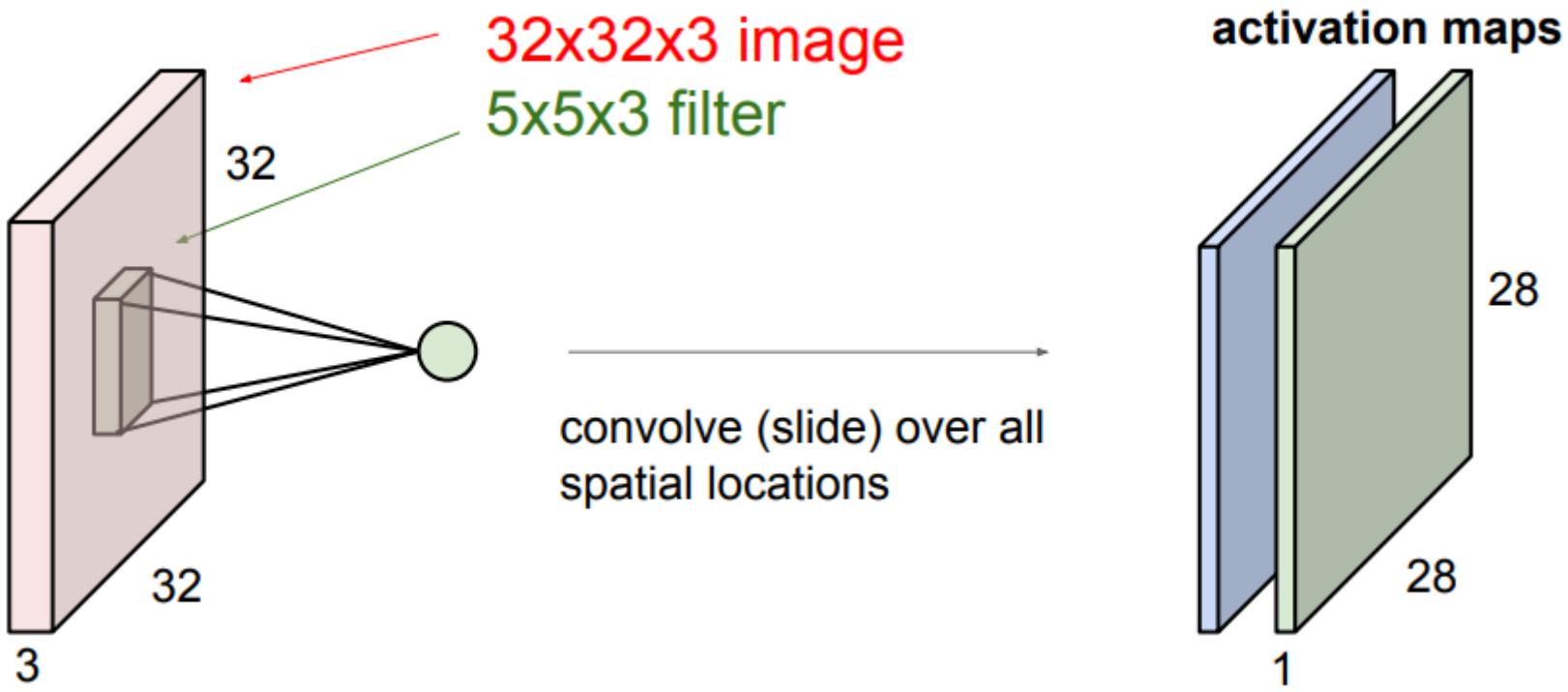
$$w^T x + b$$

Convolutional Layer



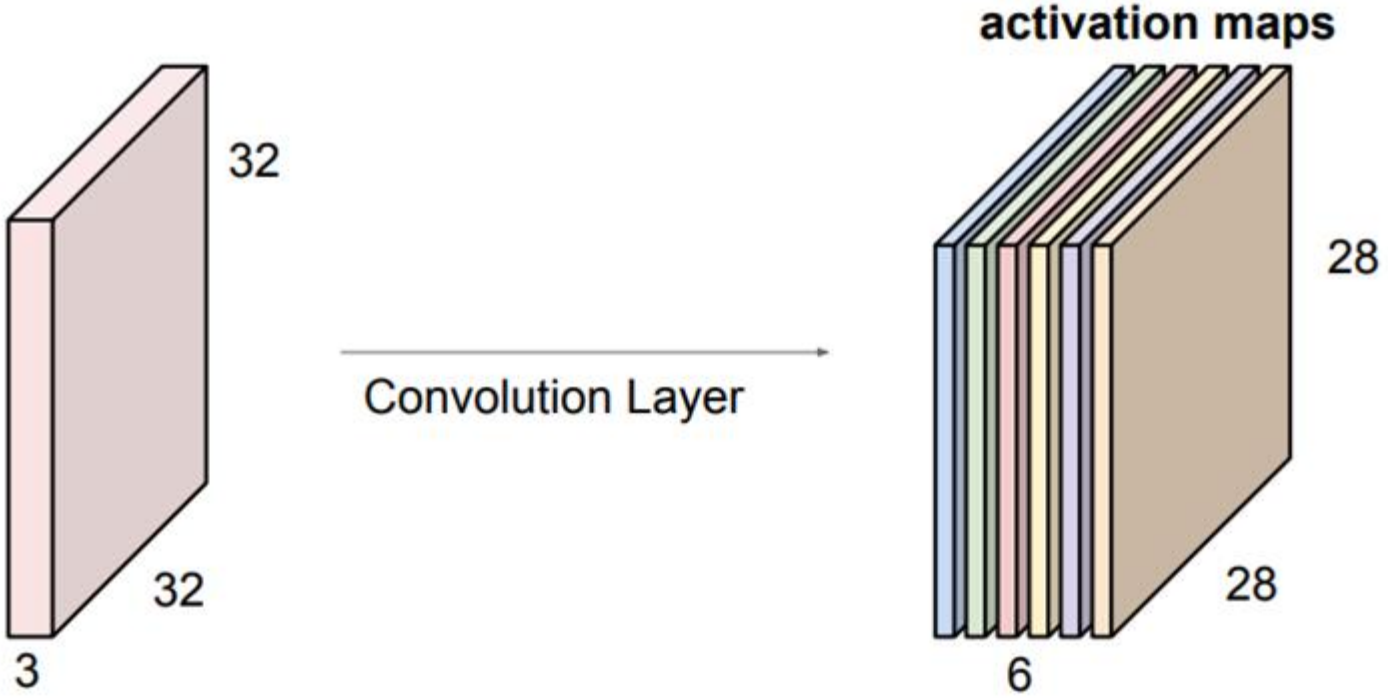
Zero-padding (optional) was not used
so output is smaller in size

Convolutional Layer



Convolutional Layer

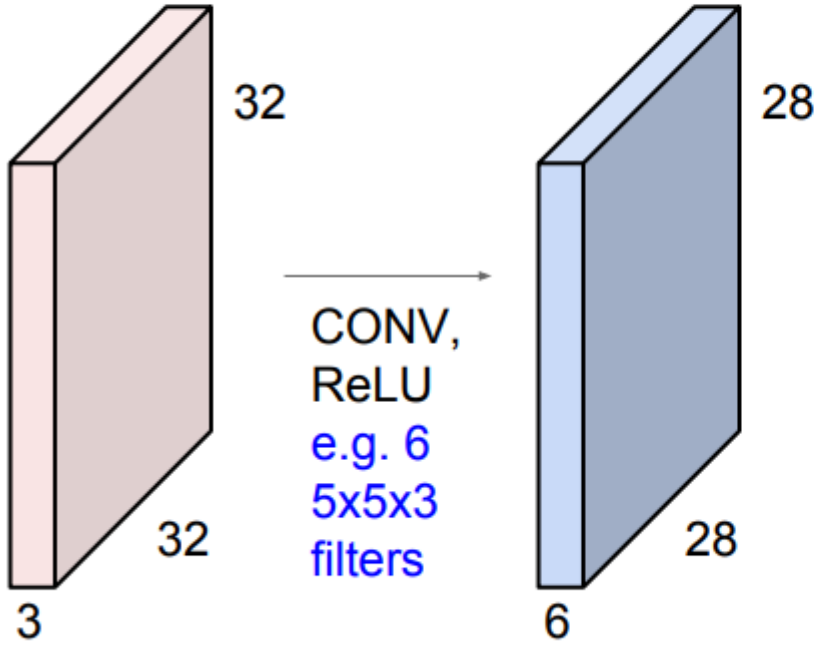
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6

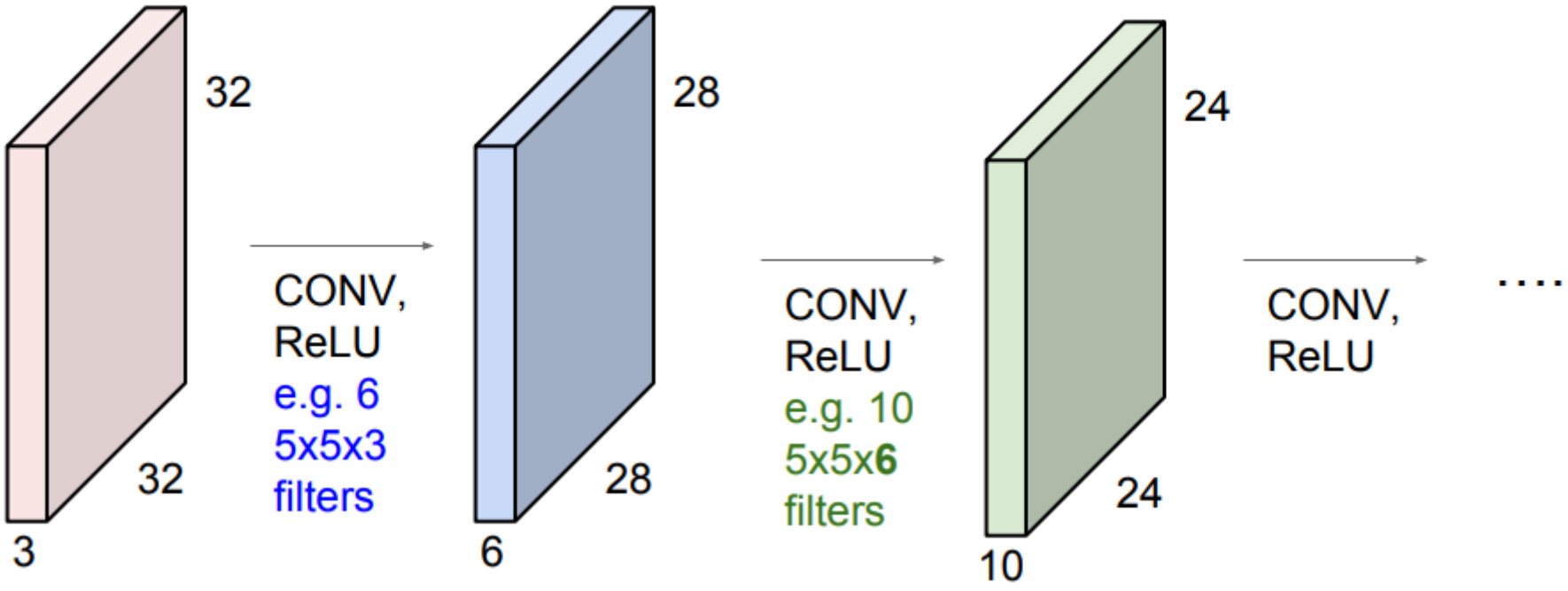
Convolutional Layer

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



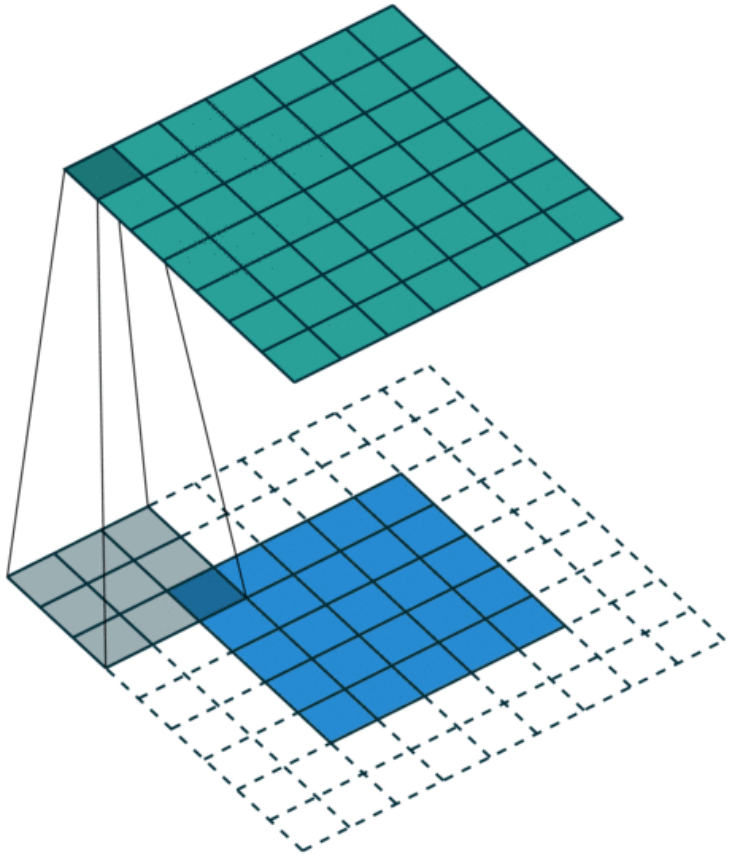
Convolutional Layer

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Other Types of Convolutions
Strided Convolutions

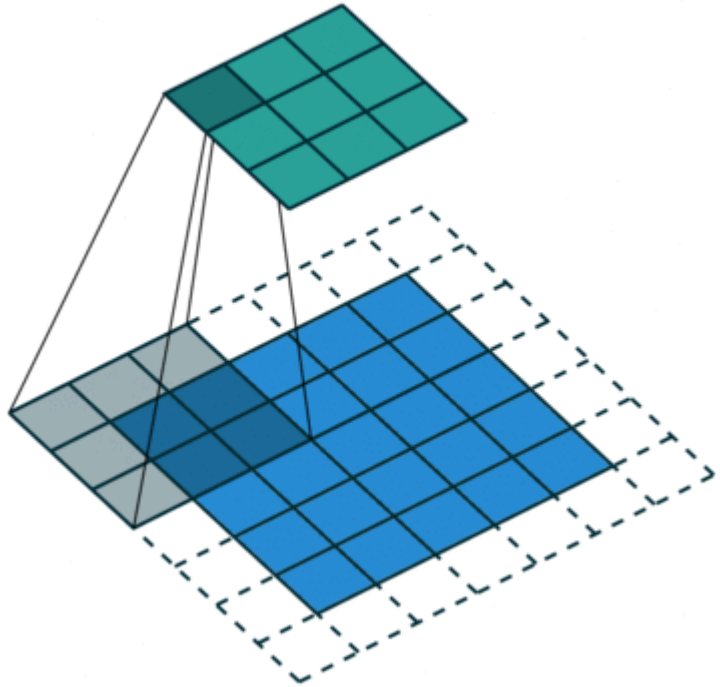
A 3x3 convolution with a stride =1 (default)



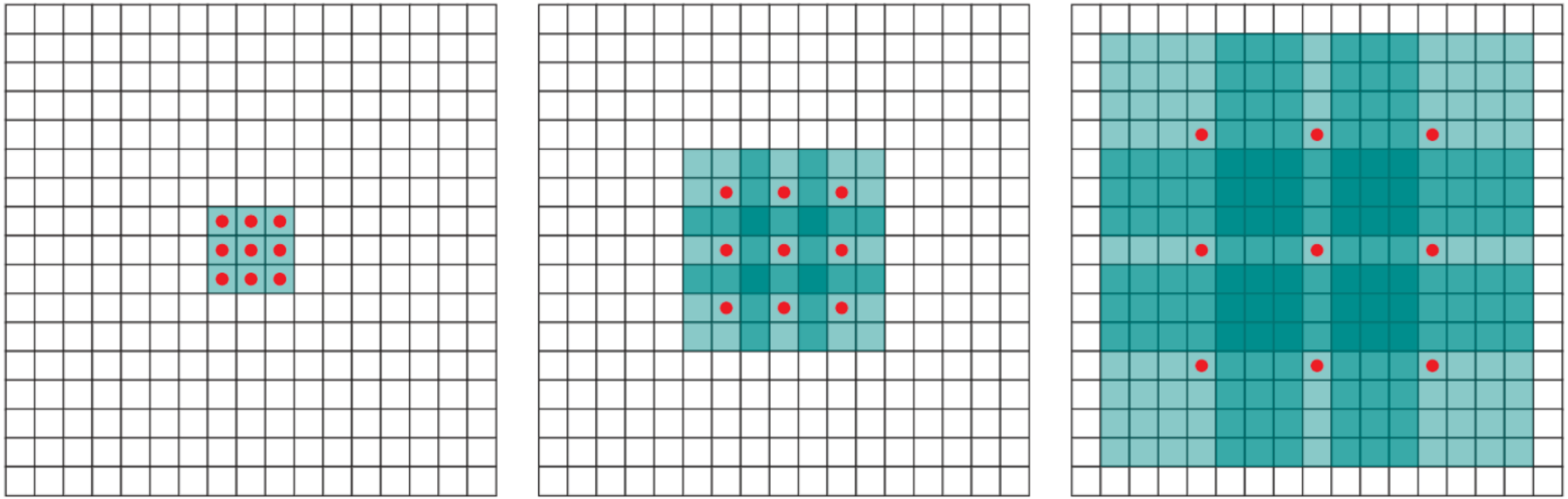
Other Types of Convolutions
Strided Convolutions

A 3x3 convolution with a stride =2

A way to summarize information, forces the network to focus on the big picture



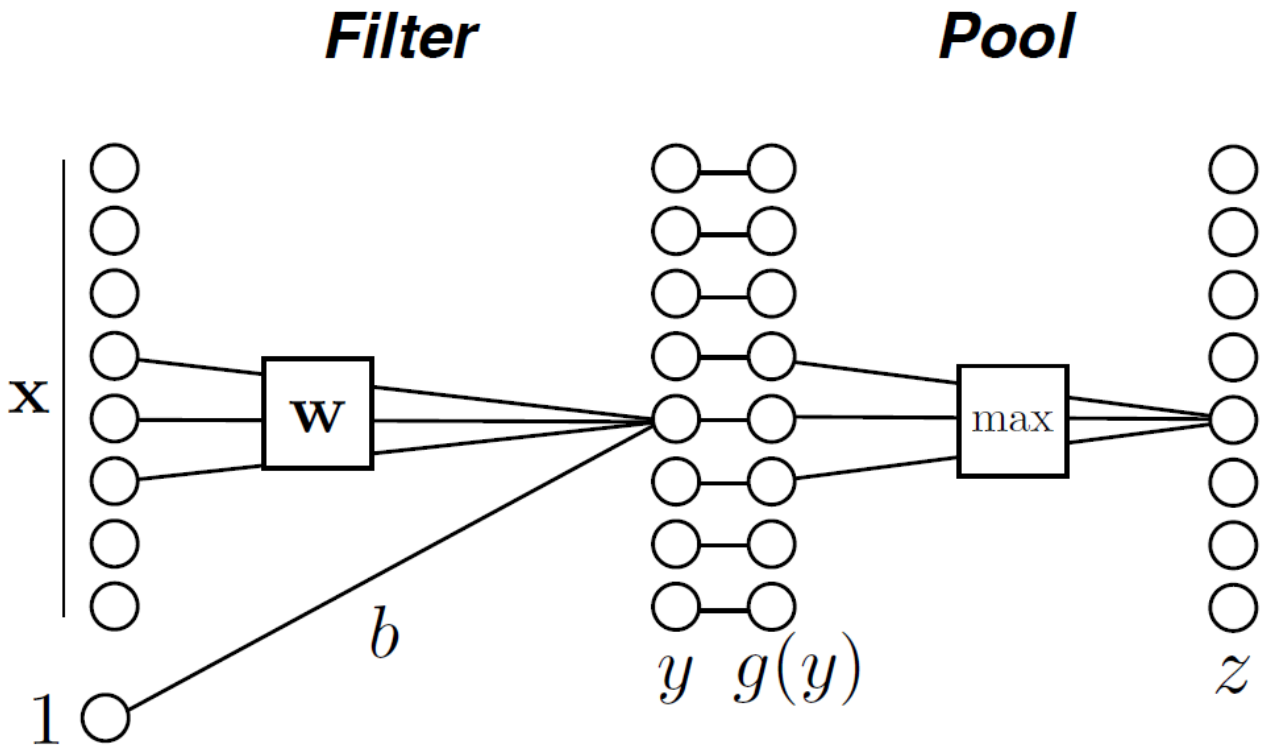
Other Types of Convolutions
Dilated Convolutions



(a) Default convolution (b) Convolution with dilation factor 2 (c) Convolution with dilation factor 4

This is equivalent to adding 0 in the filters for all places where the input value should be ignored (locations with on red dots)

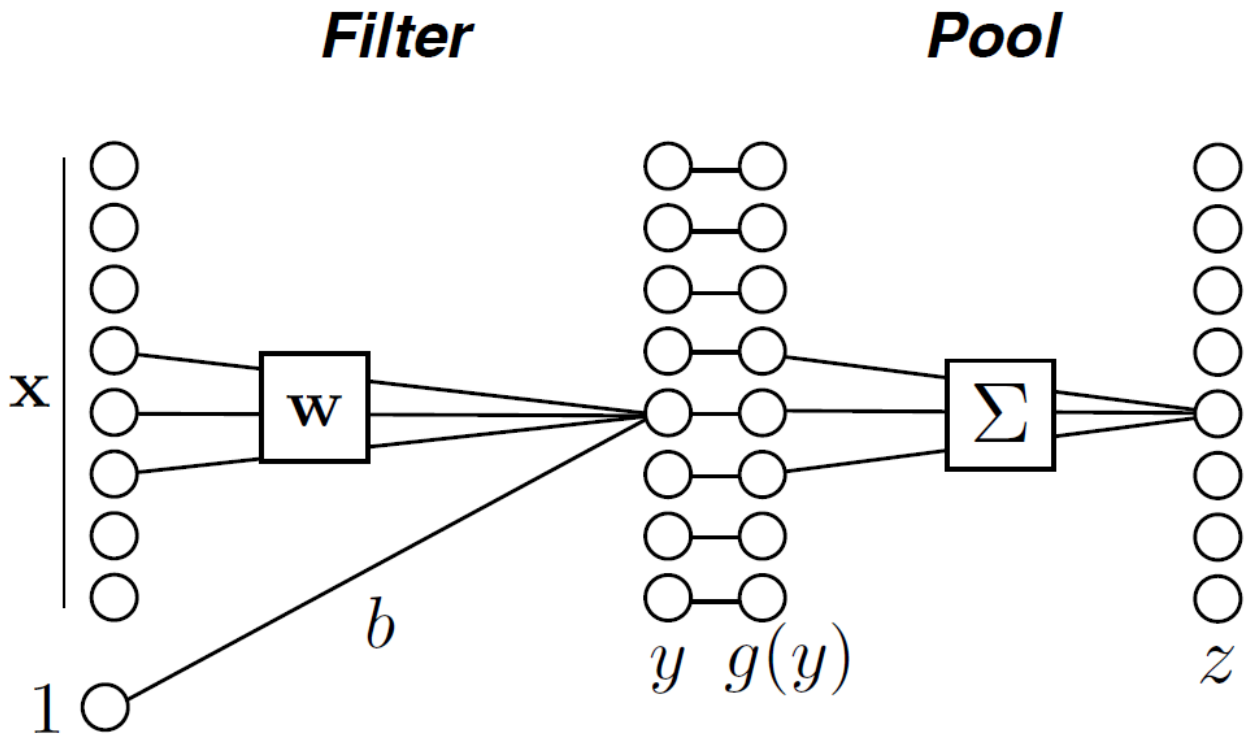
Max Pooling



Max pooling

$$z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

Average Pooling



Max pooling

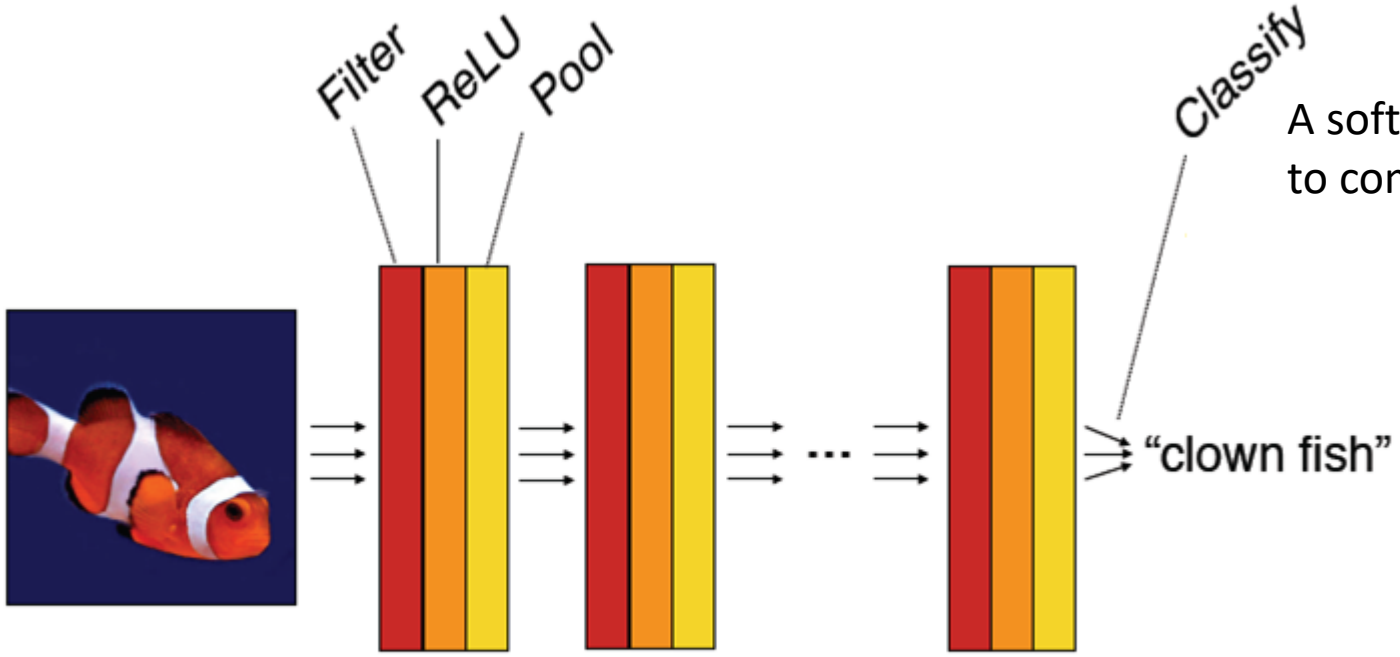
$$z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

Mean pooling

$$z_k = \frac{1}{|\mathcal{N}|} \sum_{j \in \mathcal{N}(j)} g(y_j)$$

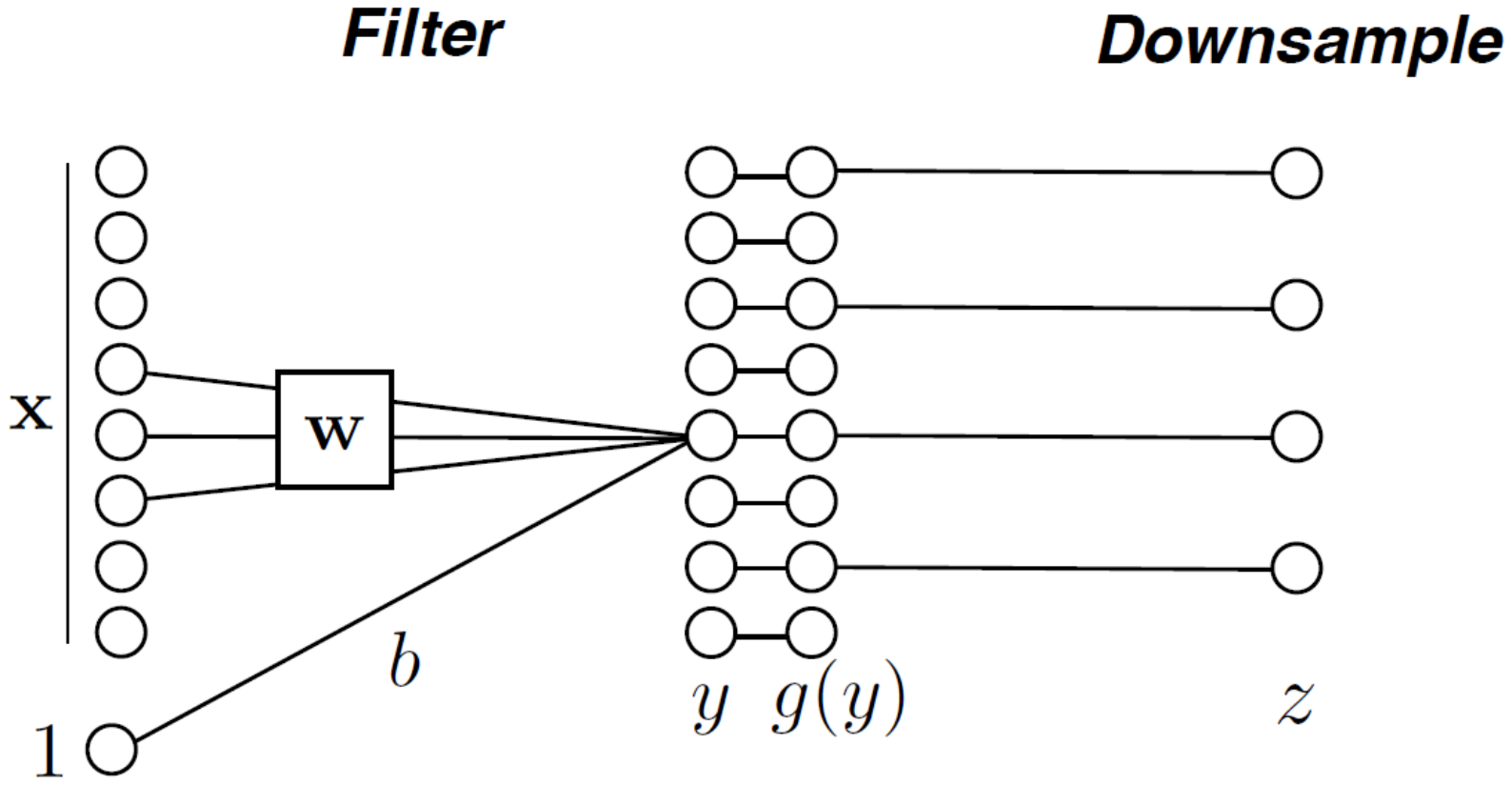
Pooling achieves stability to deformations

Implementation in NN



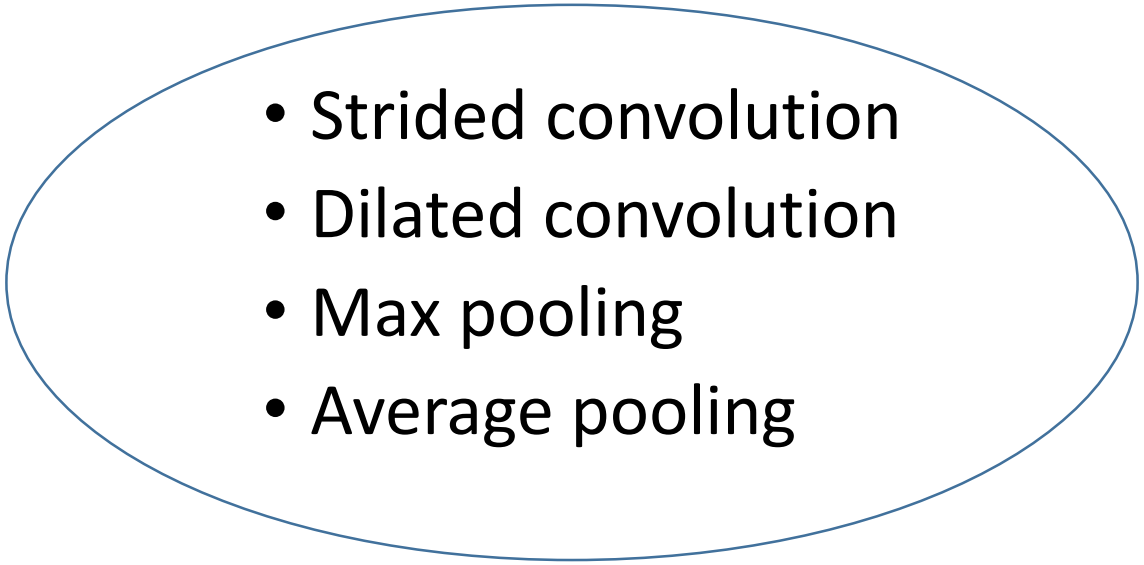
$$f(\mathbf{x}) = f_L(\ldots f_2(f_1(\mathbf{x})))$$

Downsampling



Using a stride parameter larger than 1 has the effect of downsampling

Summarizing Data



Summarizes data in a neural network

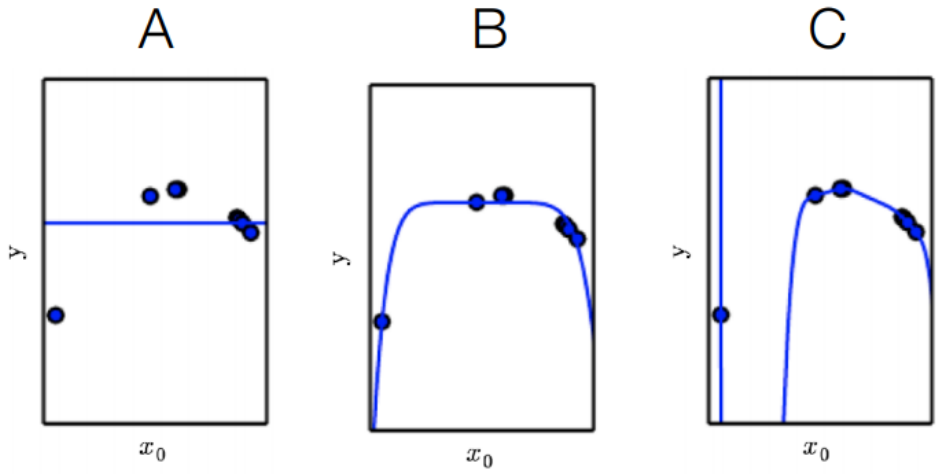
Regularizing Deep Nets

- Deep nets use millions of parameters organized in a hierarchical manner.
- On many datasets, it is easy to overfit — we may have more free parameters than data points to constrain them.
- One solution is to increase the size of training data, this is often expensive and time consuming
- Another solution is the use of regularizes
- How can we regularize to prevent the network from overfitting?
    1. Fewer neurons, fewer layers
    2. Weight decay
    3. Dropout
    4. Normalization layers

Recall: regularized least squares

$$f_\theta(x) = \sum_{k=0}^{K} \theta_k x^k$$

$$\theta^* = \arg\min_\theta \sum_{I=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i) + \lambda \|\theta\|_2^2$$



Low λ — C

Medium λ — B

High λ — A

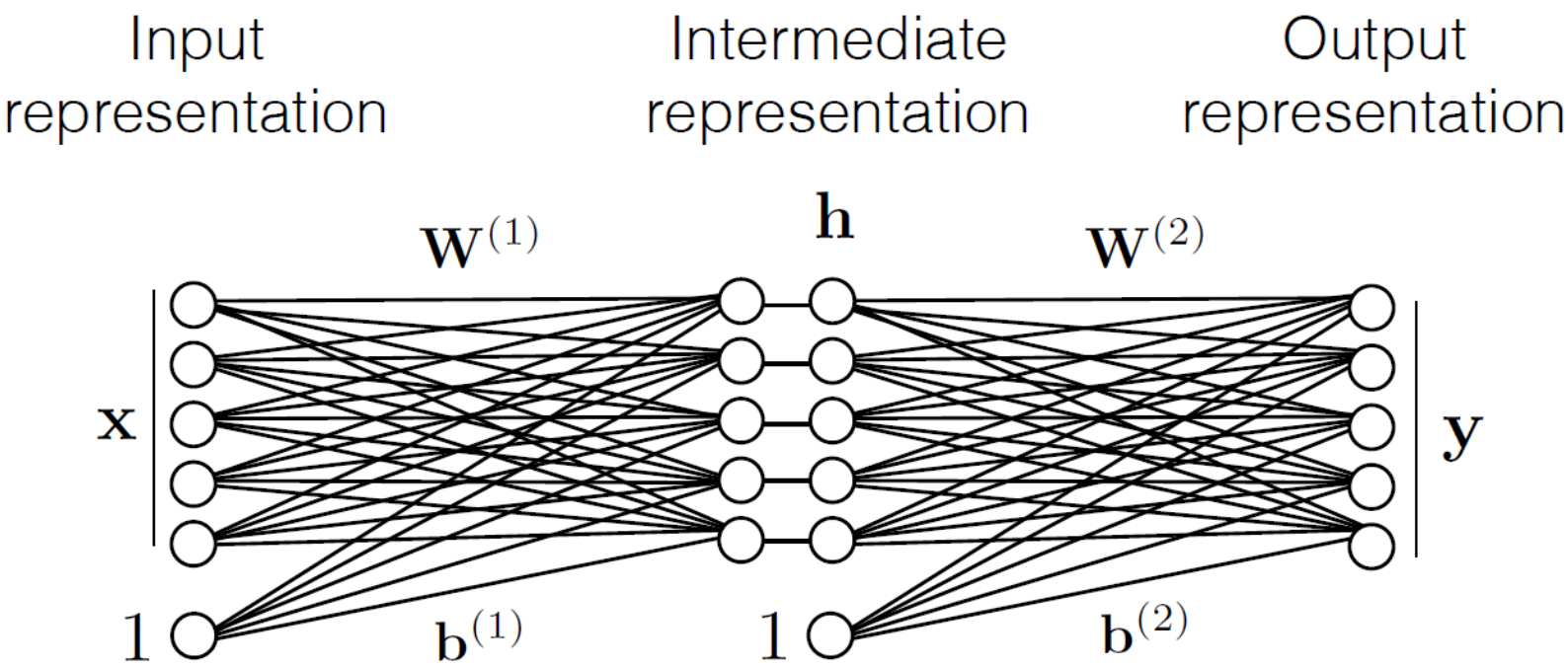
[Adapted from "Deep Learning", Goodfellow et al.]

Regularizing weights in NN

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i) + R(\theta)$$

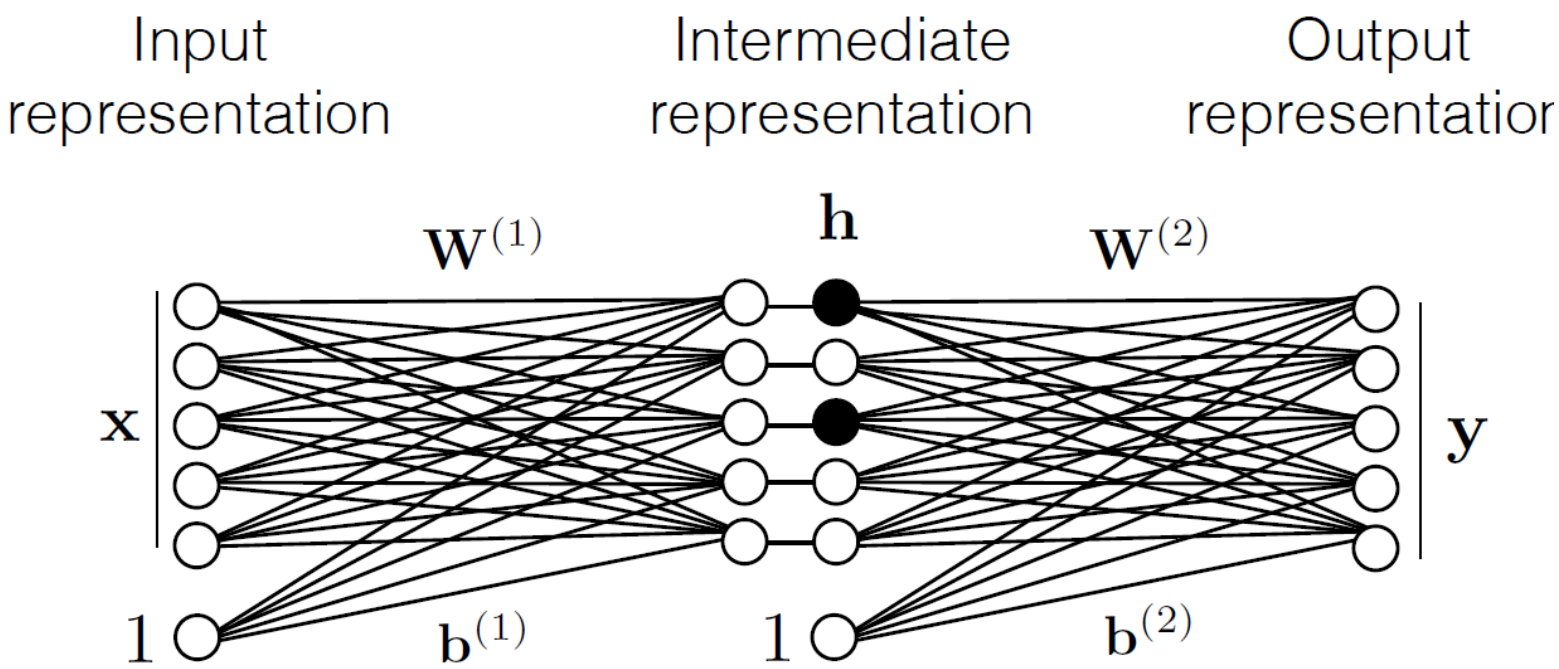
$$R(\mathbf{W}) = \lambda \|\mathbf{W}\|_2^2$$ ⟵ **weight decay**

Dropout



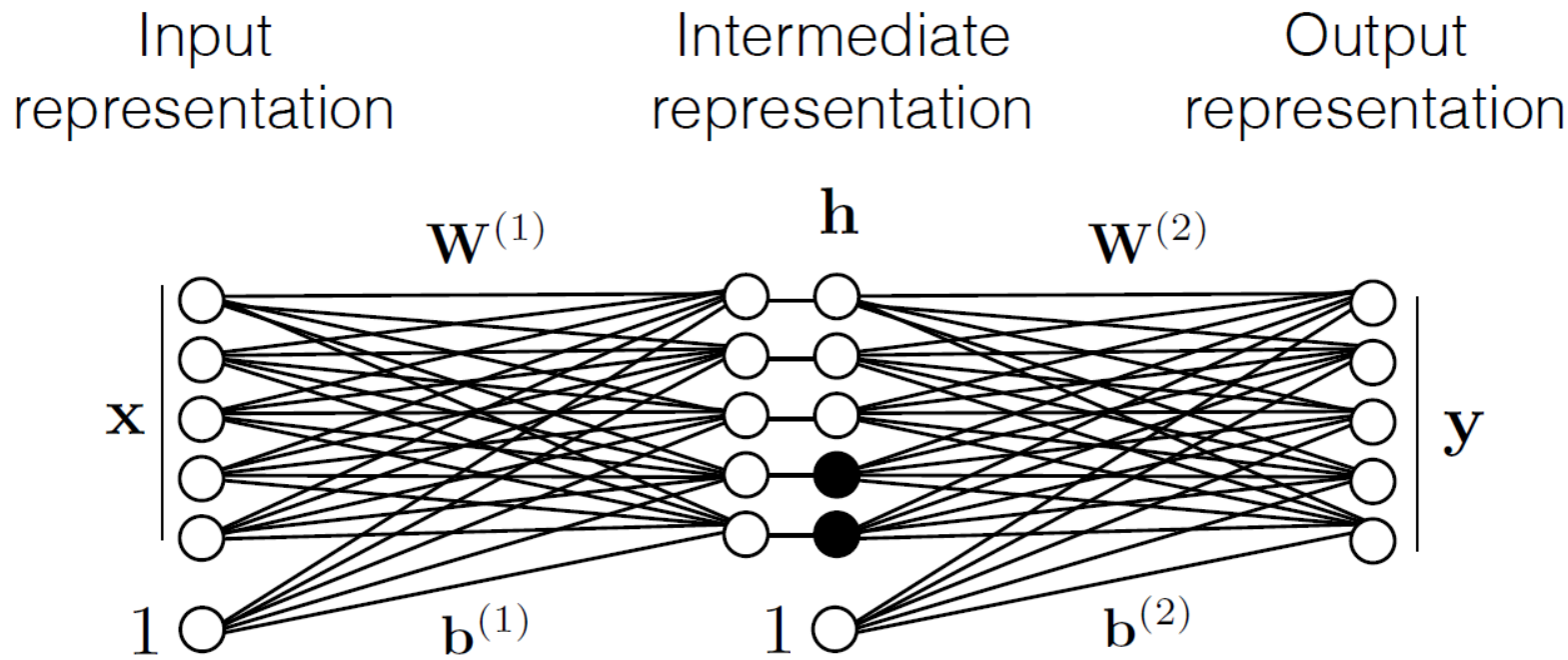
$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Dropout



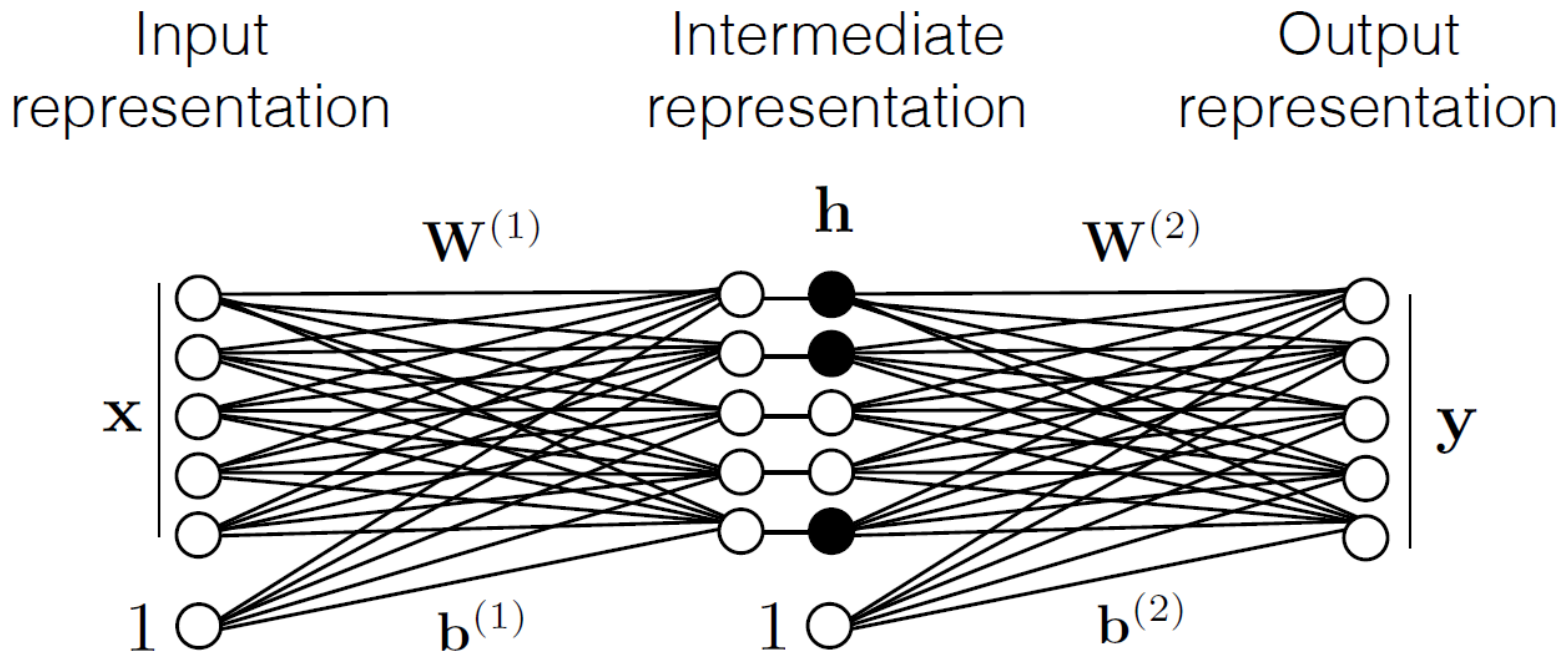
$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Dropout



$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

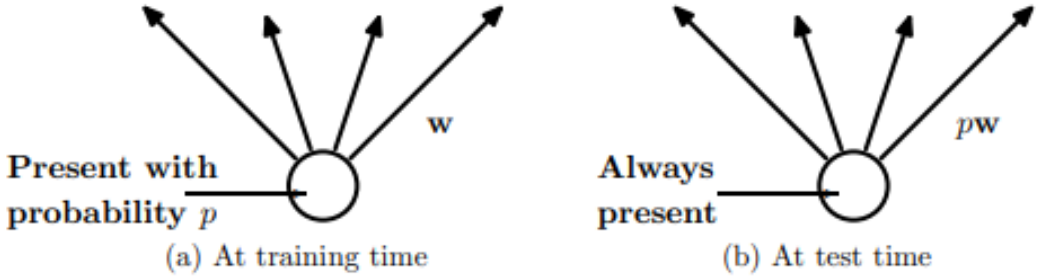
Dropout



$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Dropout

- The idea is to randomly drop hidden units (along with their connections) from the neural network during training



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

- This allows units not to rely too much on other units.
- Neurons learn to perform well in a wide variety of different contexts provided by the other hidden units

Normalization

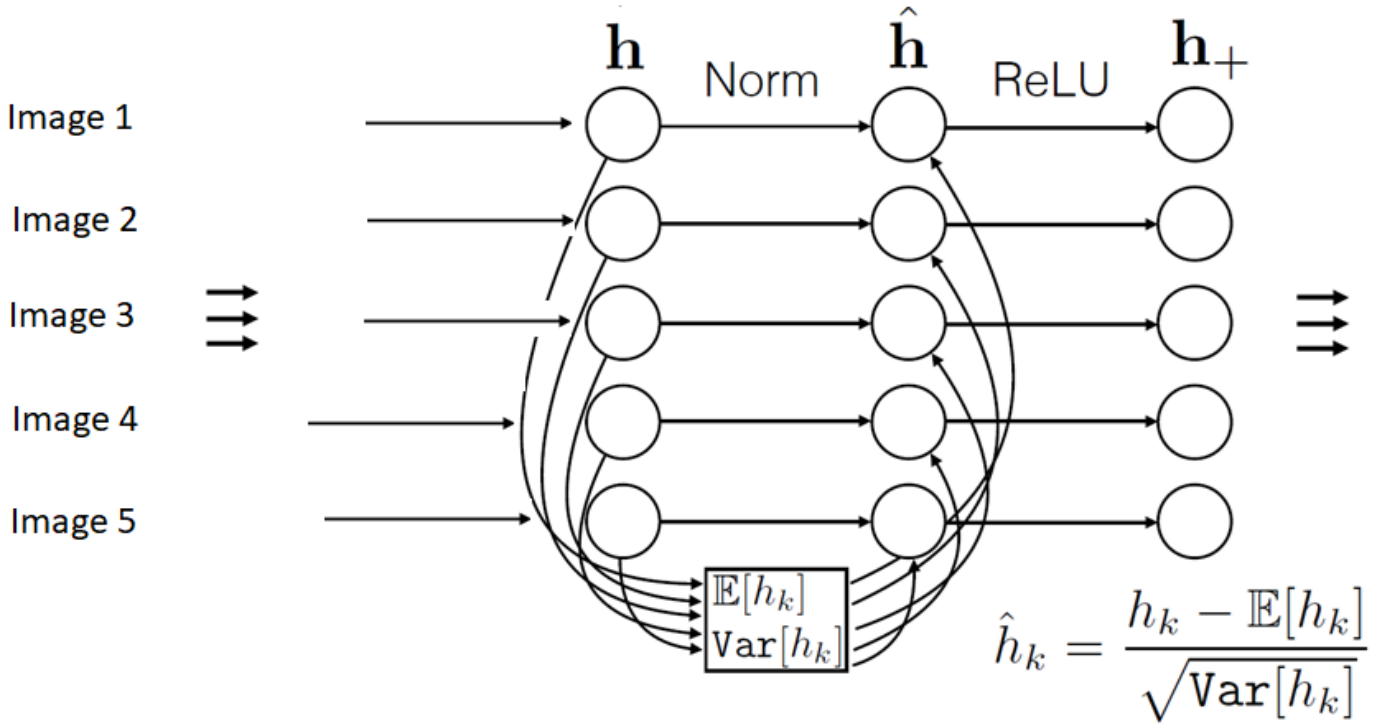
- Normalization is a regularization technique that can help improve the performance of NNs.
- Subtract the mean and divide by standard deviation

$$z = \frac{x - \mu}{\sigma}$$

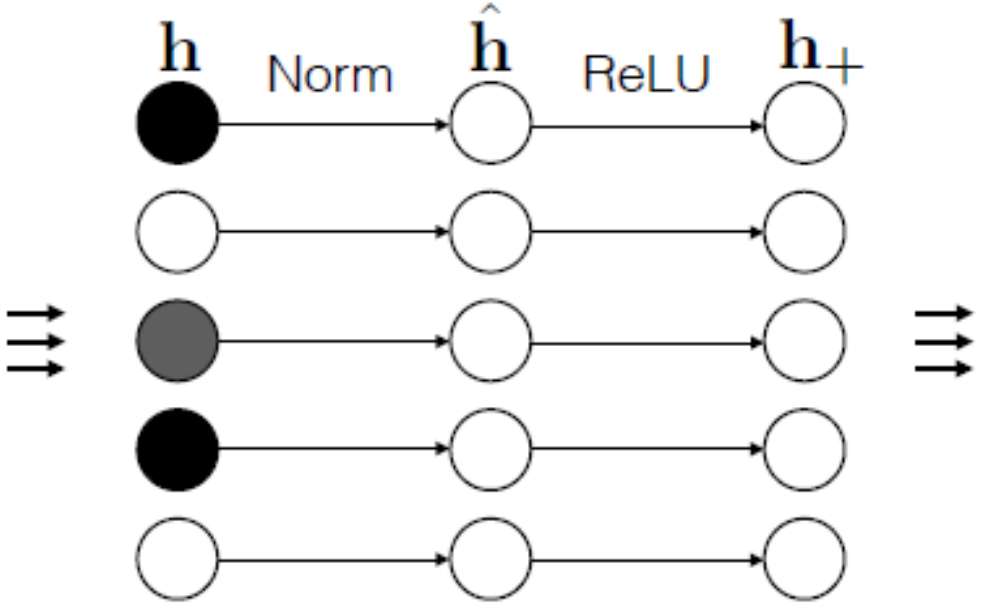
- The new mean is 0 and standard deviation of 1.
- Necessary for comparing numbers from different scales (e.g. TOEFL vs IELTS).
- Input data is often normalized before feeding them to a neural network. In addition normalization layers normalizes the weights and biases in the network.

Normalization layers

Assume that training is performed in batches, where each “minibatch” is composed of 5 data samples.

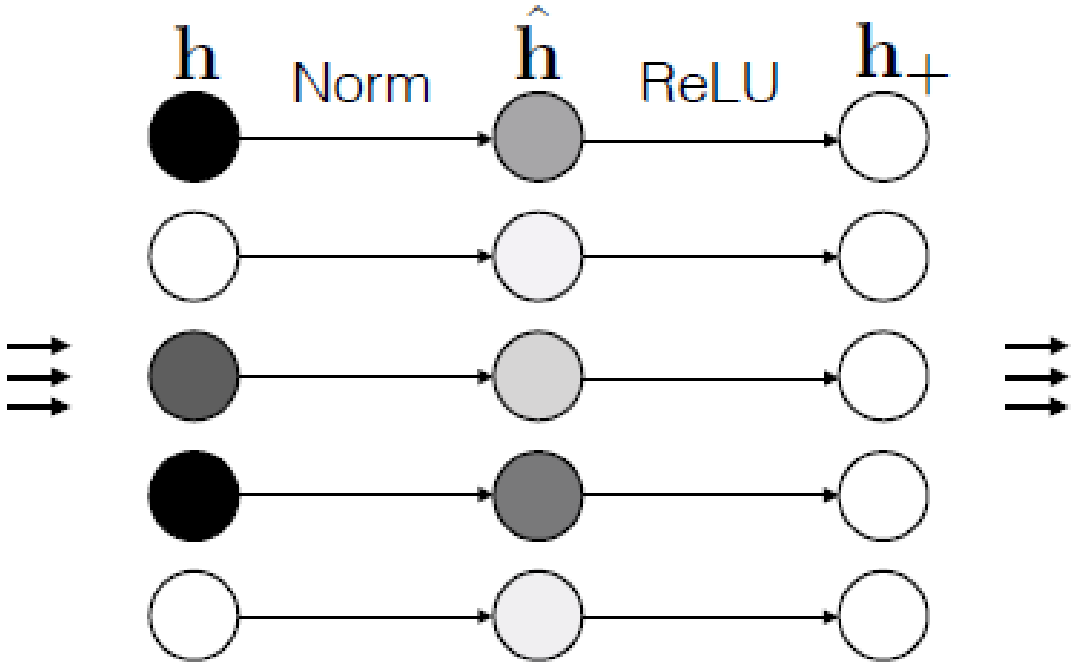


Normalization layers (Batch Normalization)



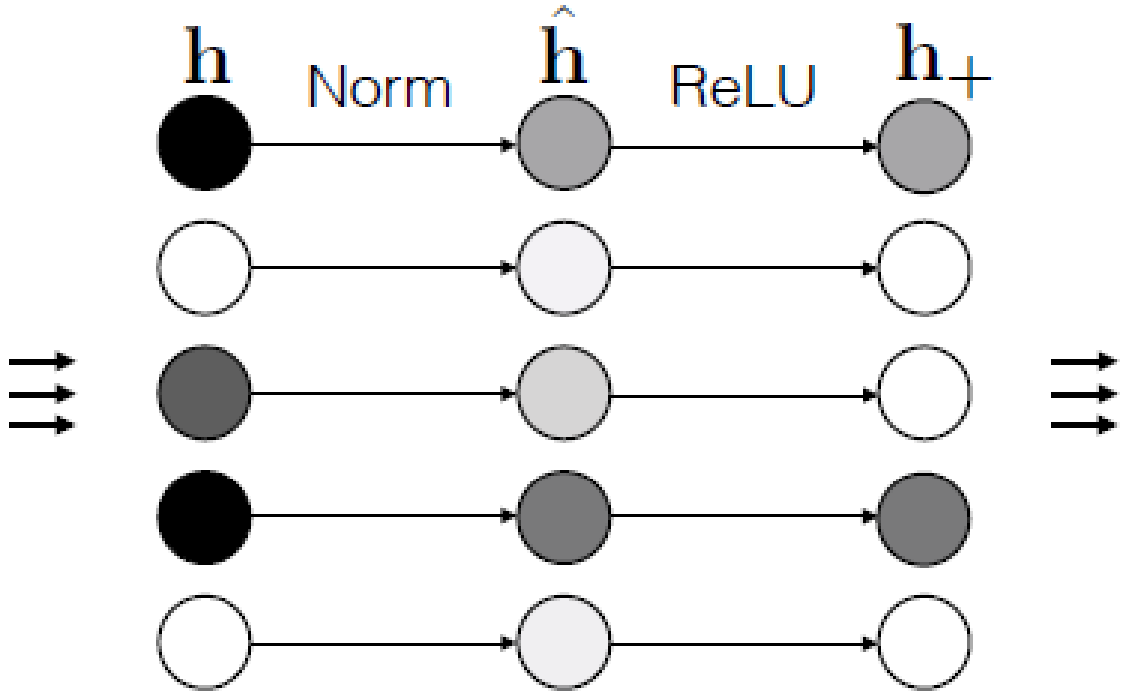
$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\mathtt{Var}[h_k]}}$$

Normalization layers



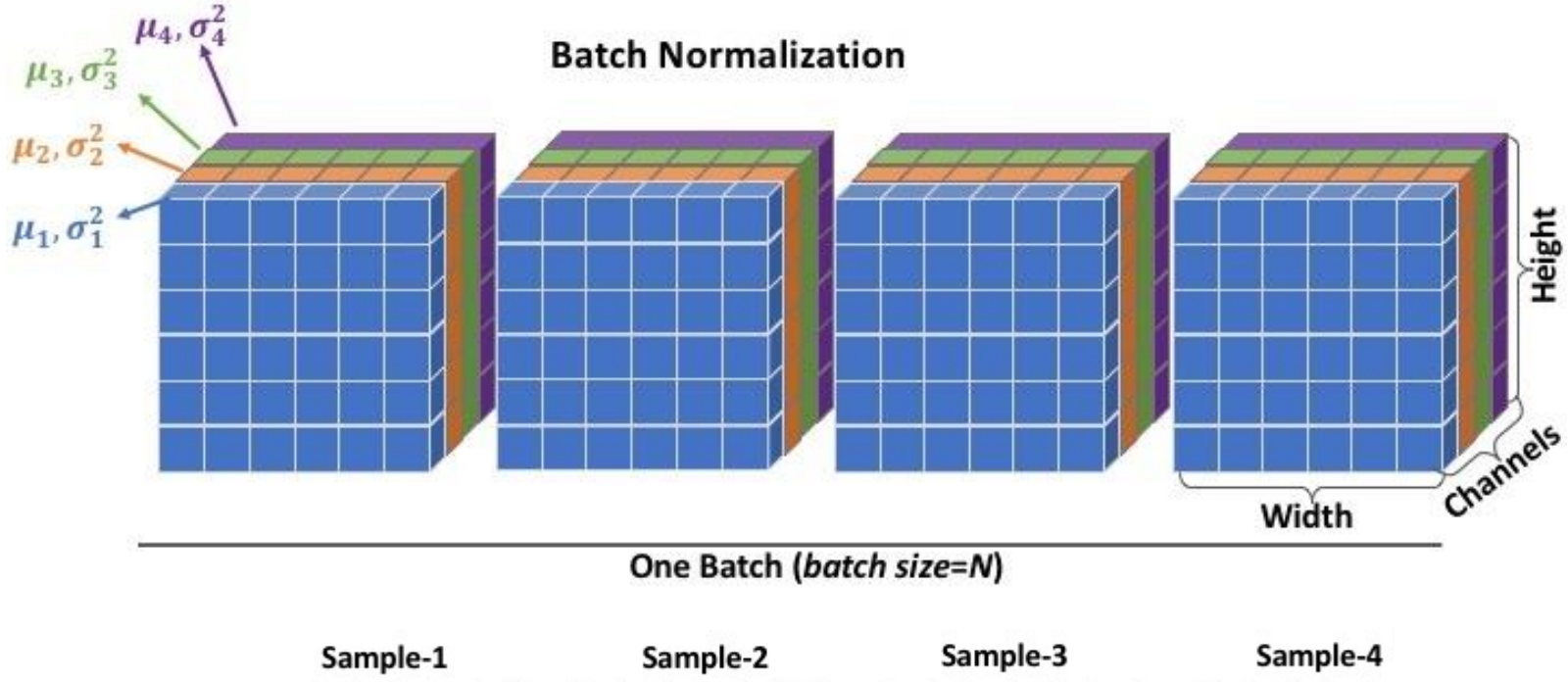
$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\mathtt{Var}[h_k]}}$$

Normalization layers



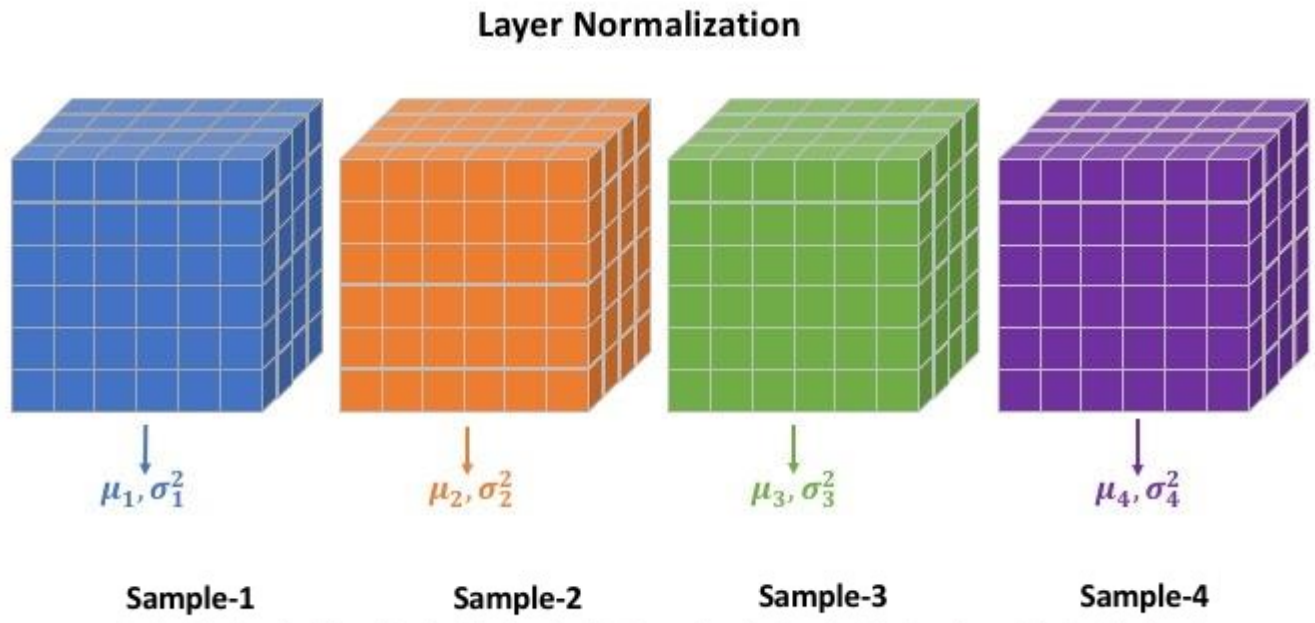
$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\mathtt{Var}[h_k]}}$$

Main types of normalization:
1. Batch Normalization



Normalization is applied for each uniformly colored region in the figure

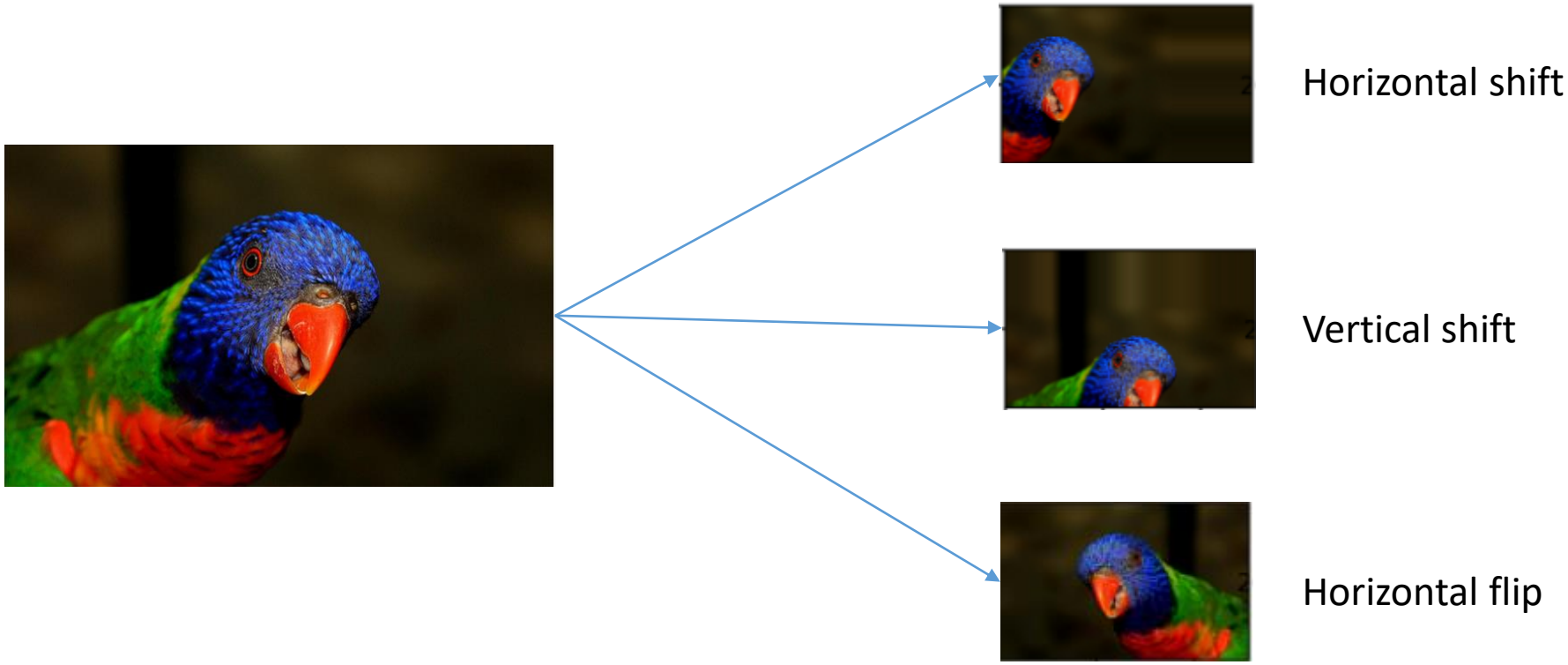
2. Layer Normalization



Normalization is applied for each uniformly colored region in the figure

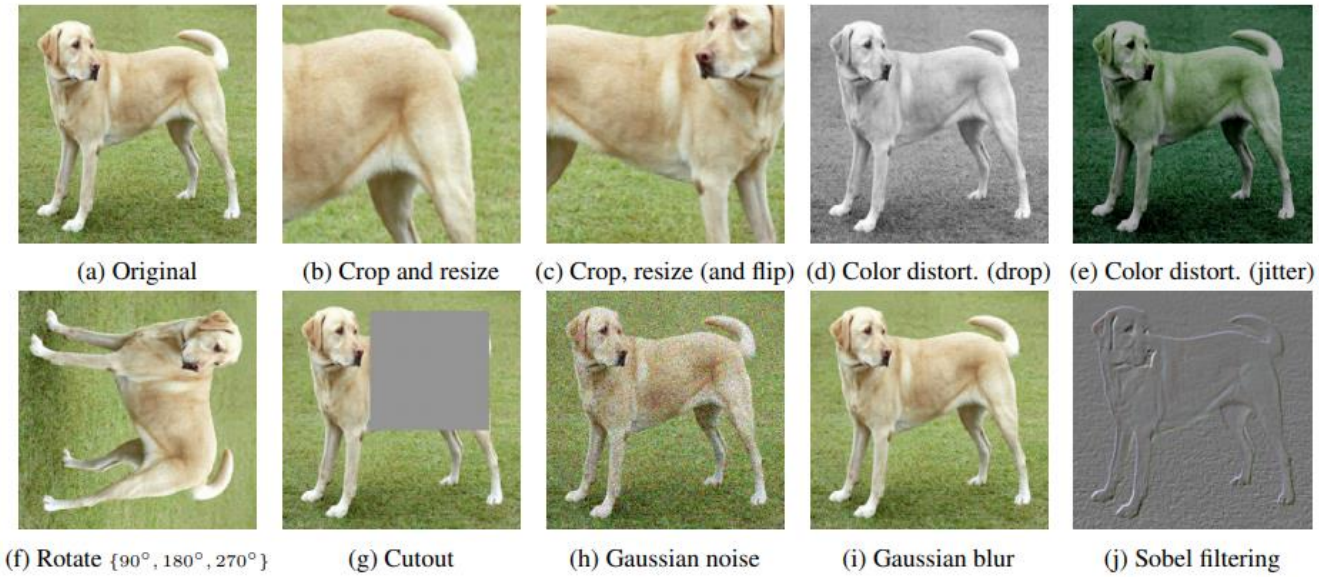
Data Augmentation

- Data augmentation provides a simple and effective way to add more images to your dataset



Data Augmentation

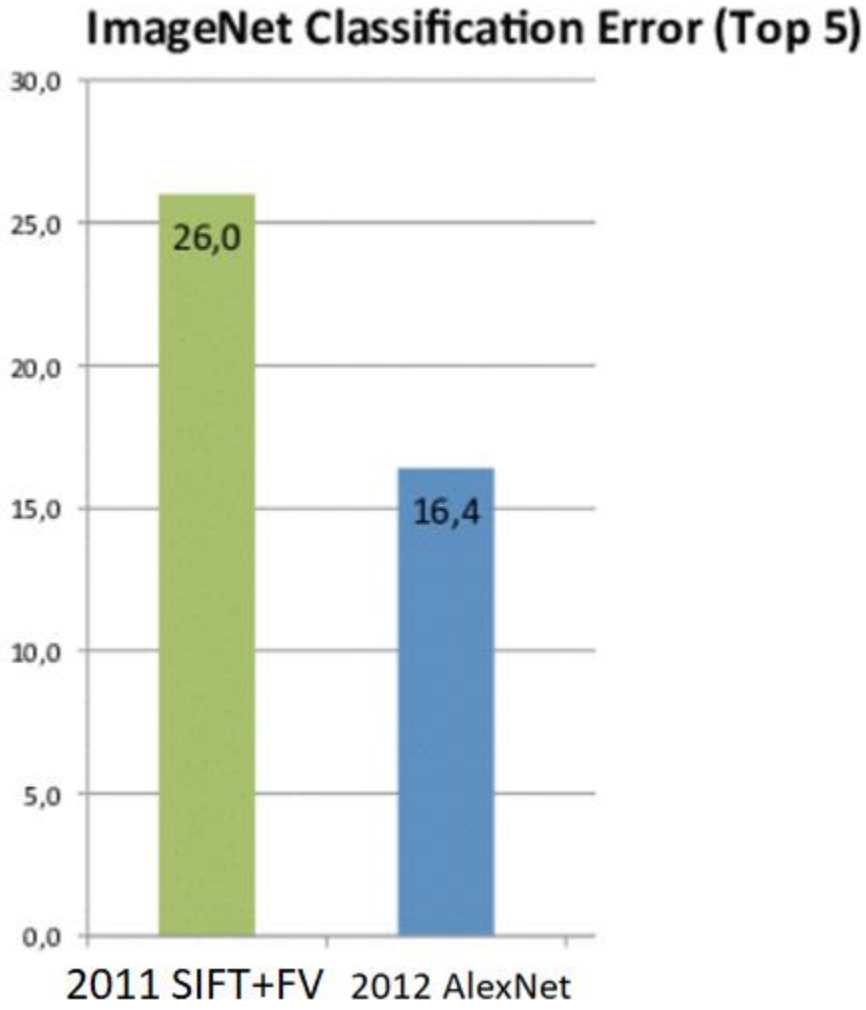
- Other possible operations
  - Rotations
  - Color distortions
  - Gaussian smoothing
  - Gaussian noise
  - Sobel filtering
  - Random crop + resize.
  - ….

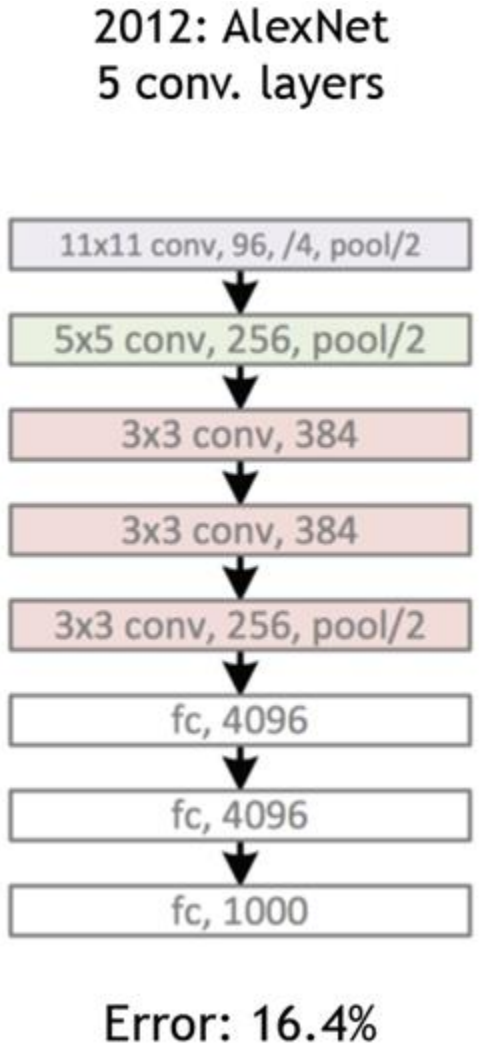


Examples of known CNN networks

- Alexnet
- VGG-Net
- ResNet: Residual Neural Networks
- DenseNet: Dense Neural Networks

Alexnet

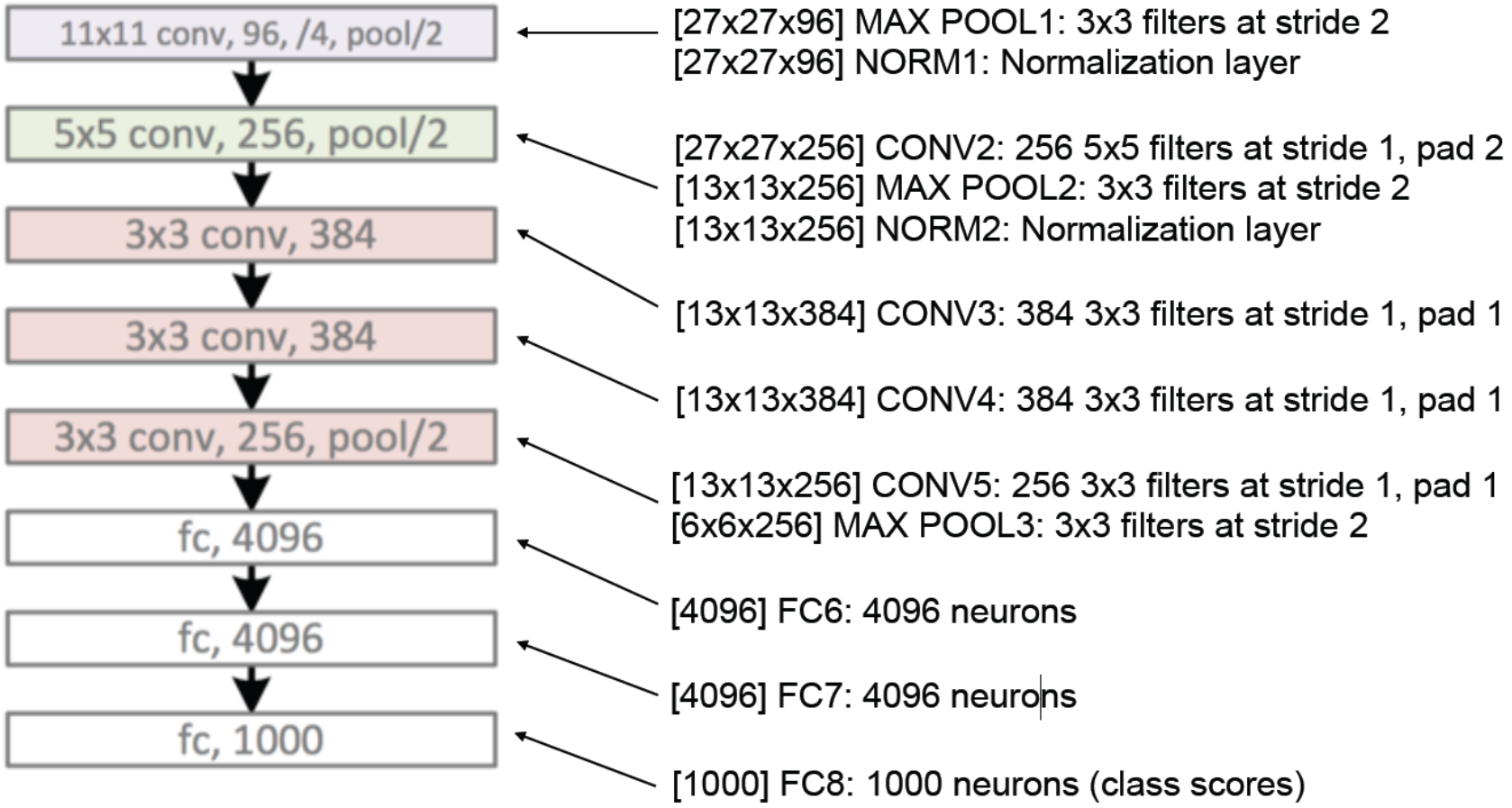




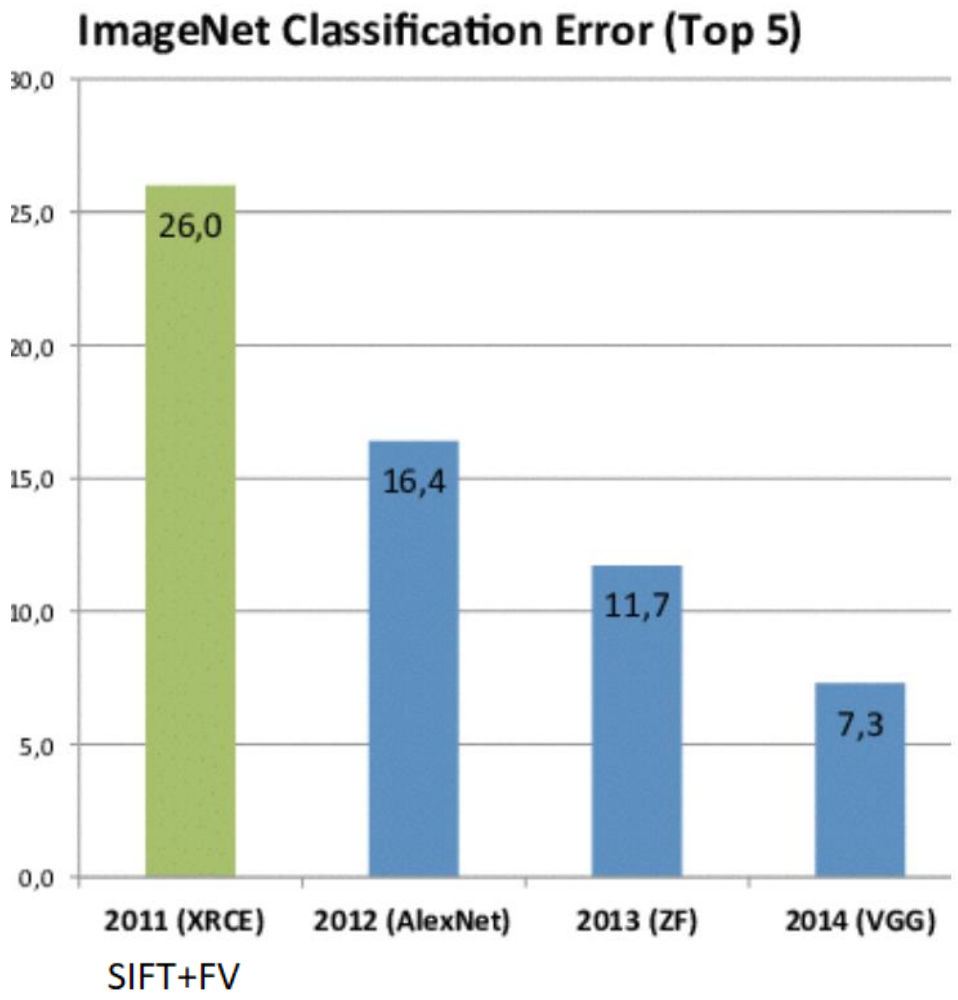
Identifying the 1000 classes in the dataset

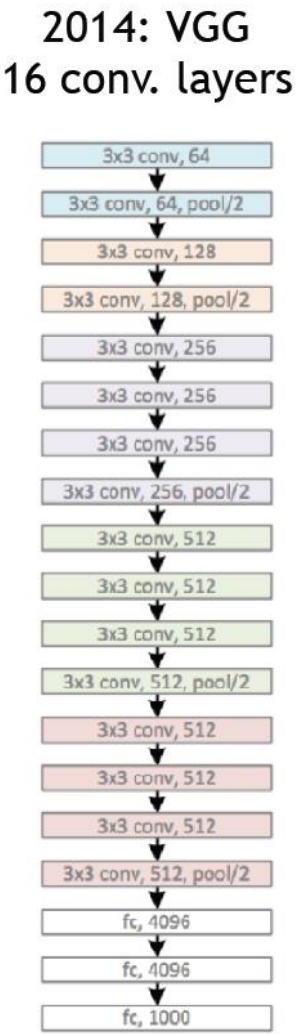
ImageNet 2012 challenge object classes: http://image-net.org/challenges/LSVRC/2012/browse-synsets

Alexnet


[227x227x3] INPUT
11x11 conv, 96, /4, pool/2
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
5x5 conv, 256, pool/2
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
3x3 conv, 384
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
3x3 conv, 384
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
3x3 conv, 256, pool/2
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
fc, 4096
[4096] FC6: 4096 neurons
fc, 4096
[4096] FC7: 4096 neurons
fc, 1000
[1000] FC8: 1000 neurons (class scores)

VGG-Net


ImageNet Classification Error (Top 5)
30,0
25,0
20,0
15,0
10,0
5,0
0,0
26,0
16,4
11,7
7,3
2011 (XRCE)
2012 (AlexNet)
2013 (ZF)
2014 (VGG)
SIFT+FV

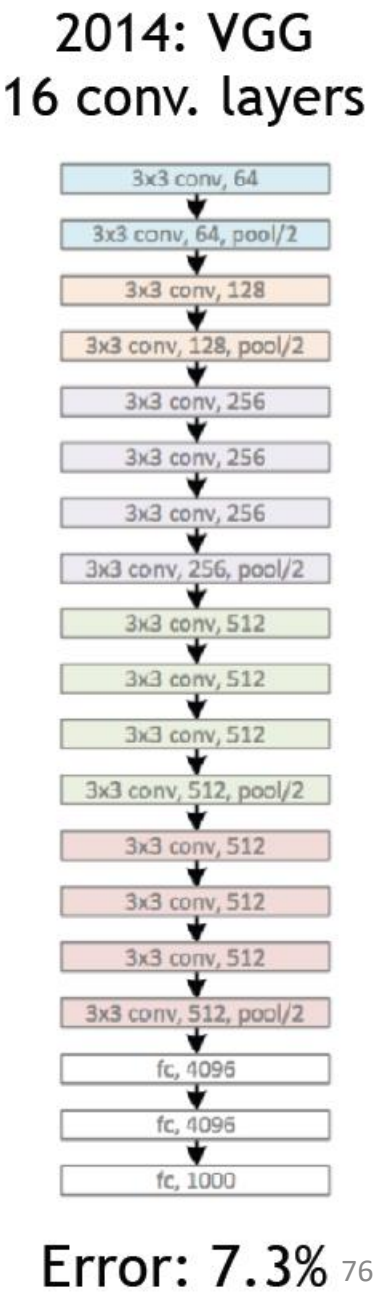

2014: VGG
16 conv. layers
3x3 conv, 64
3x3 conv, 64, pool/2
3x3 conv, 128
3x3 conv, 128, pool/2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256, pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512, pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512, pool/2
fc, 4096
fc, 4096
fc, 1000
Error: 7.3%

VGG-Net

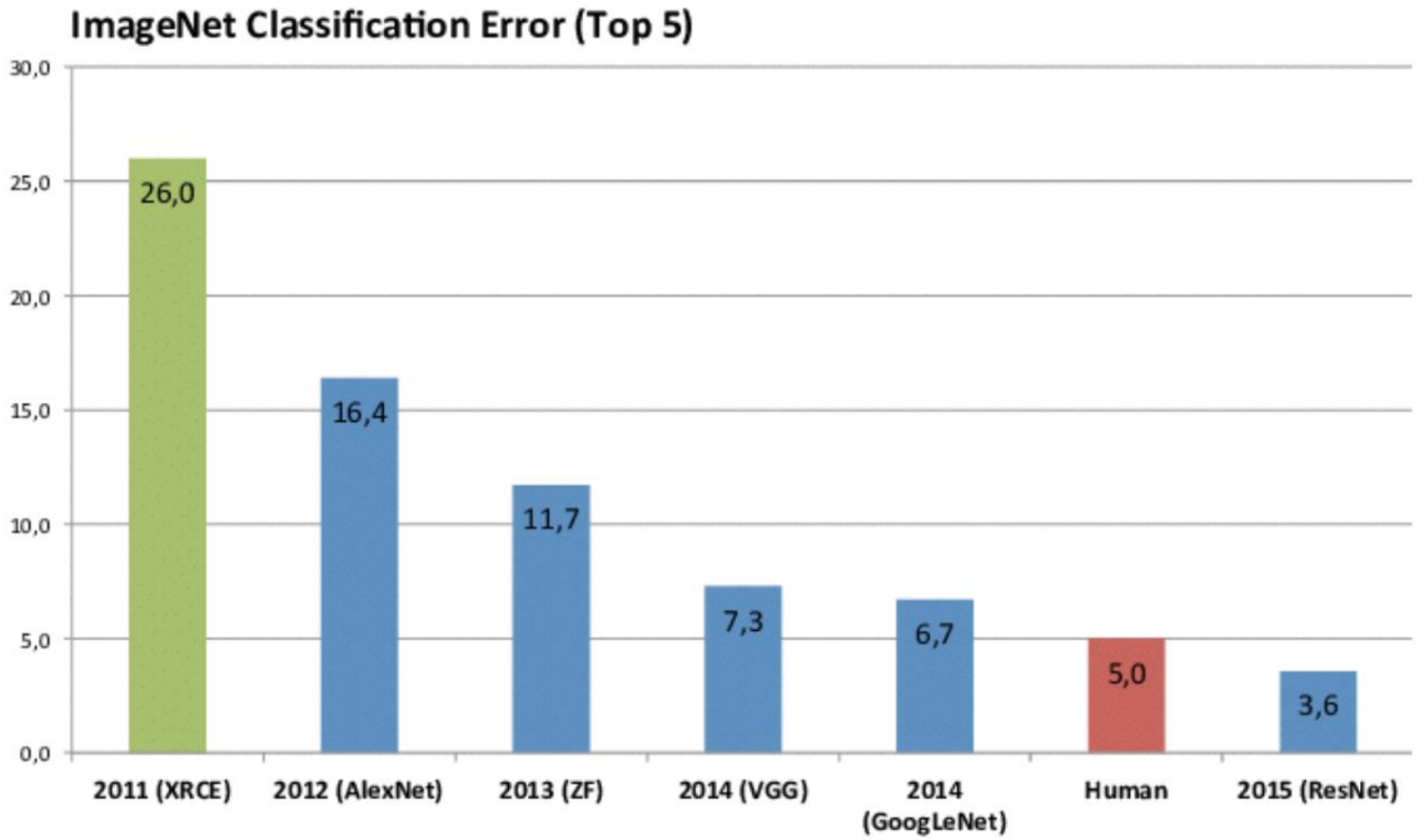
- Important Features: Fixed size convolutions 3 x 3
- Increased depth,

AlexNet: 5 Conv. layers + 3 FC

VGG-Net: 16 Conv. layers + 3 FC



ResNet



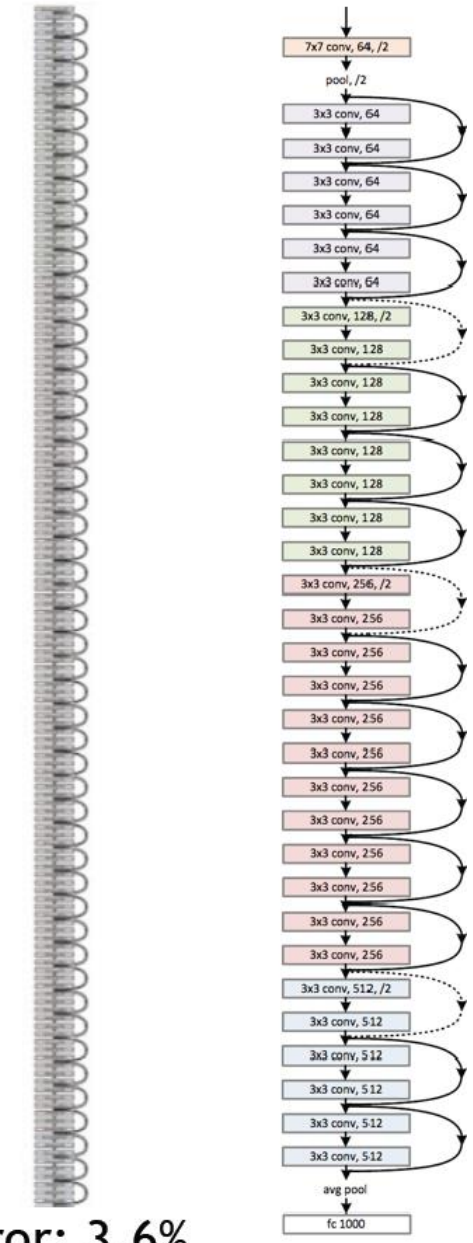
2016: ResNet
>100 conv. layers

Error: 3.6%

Is the network better than humans?

ResNet

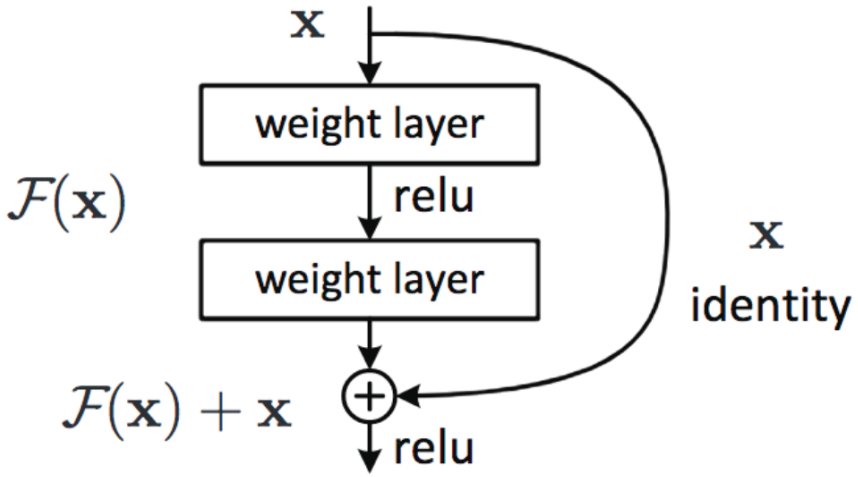
2016: ResNet
>100 conv. layers



Error: 3.6%

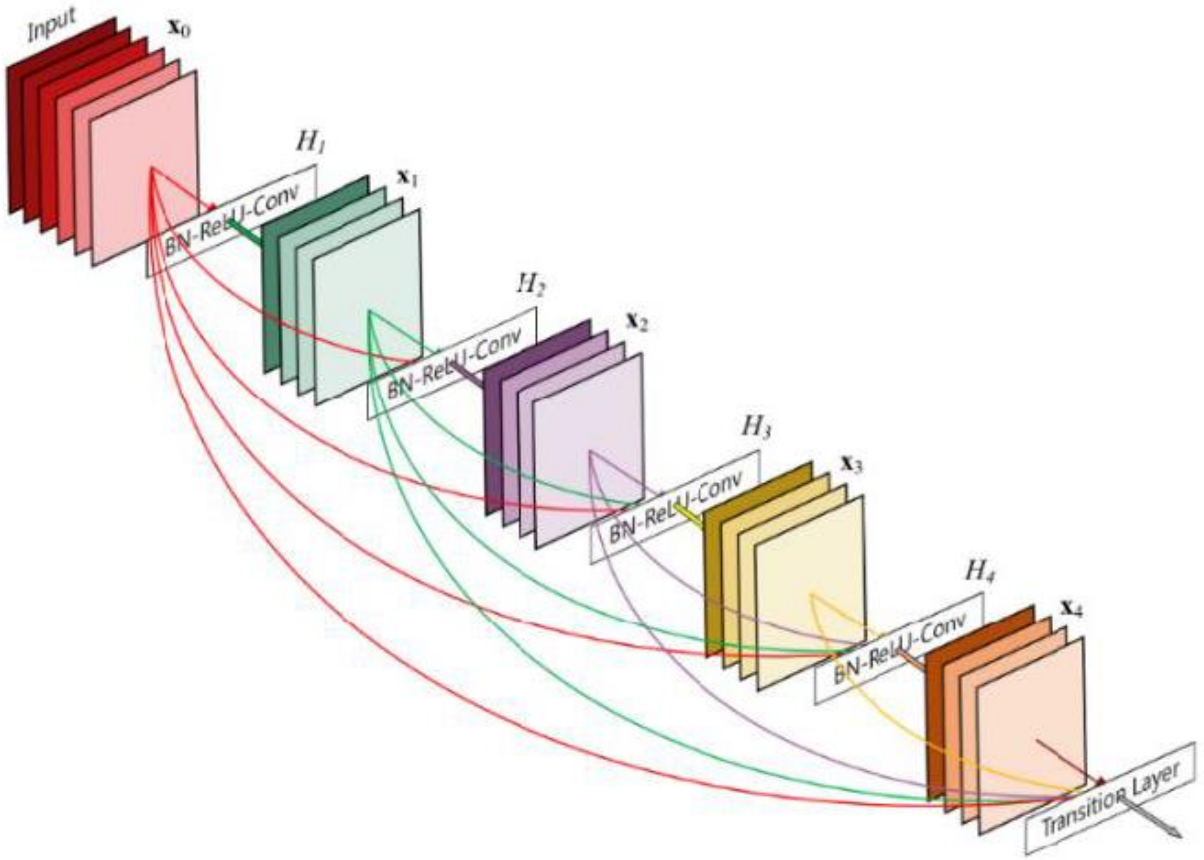
Main developments

- Increased depth possible through residual blocks



Dense Nets

- Taking the idea of ResNet to the extreme



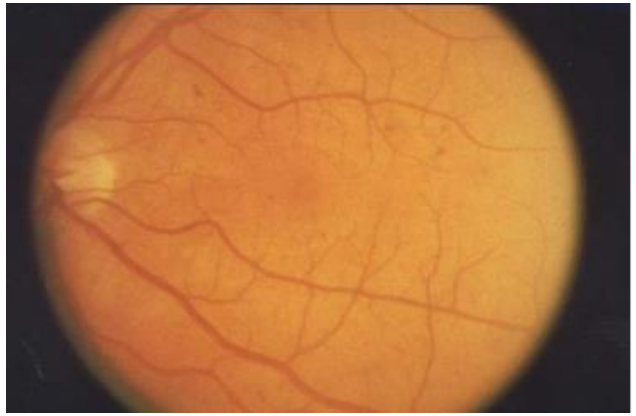
Transfer Learning

- The concept of taking an existing, trained model and training it further, either as is or as part of a new model. Also, known as fine tuning.
- Example 1: Assume your goal is to build a neural network that can classify the different types of cars.
- Instead of training the network from scratch, you start with a pertained model on ImageNet and fine-tune the model to fit your new dataset (i.e. initialize the network with ImageNet weights).
- Its common to freeze early layers during training with a new dataset.

Transfer Learning

- Example 2
- The goal is to build a binary detector for injured retinas.

  Start with a network trained with ImageNet or a similar dataset. Freeze the weights in initial layers. Change the output layer to binary and train the network



- This approach can use existing knowledge in the network before adapting to new problems.