Object Classification and Detection with Deep Convolutional Neural Networks

Dr. PP

Research Scientist
AI Lab @ BAT, XiNiuEdu

September 1, 2017

Preliminary Background

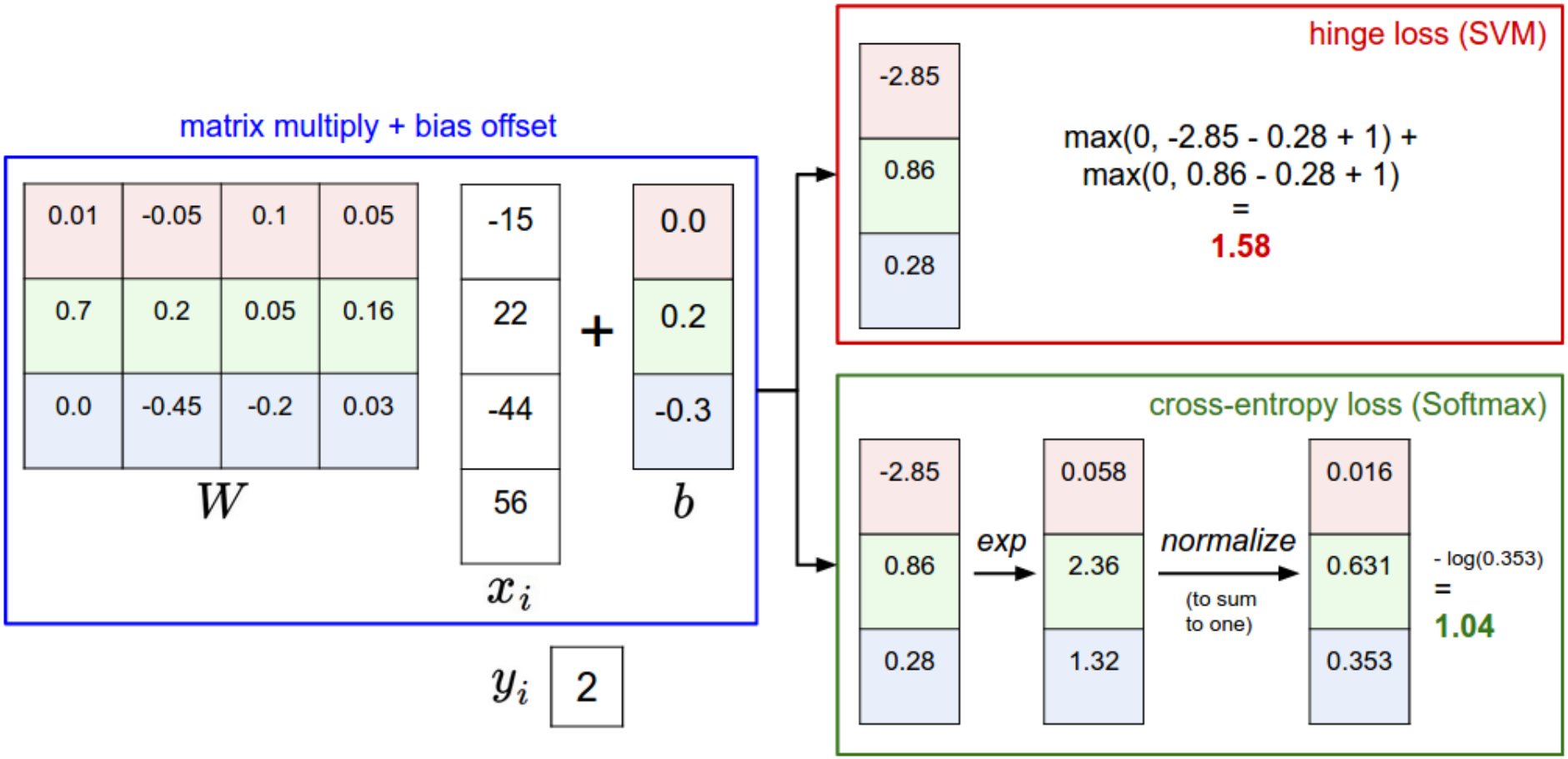
- Basic calculus, e.g. (partial) derivatives.
- Basic knowledge of machine leanring, e.g. linear classifier, loss functions, overfitting, underfitting.
- Basic optimization algorithm, e.g. SGD.
- Basic level of a programming language, e.g. Python.

In this lecture, you will learn

- how a neural network outputs a prediction given an input feature vector (Recap);
- how to train a neural network given training data, i.e. backpropagation (Recap);
- practical techniques for tuning parameters of neural networks;
- basic concepts of convolutional neural networks (ConvNets);
- how to apply ConvNets on a Kaggle competition for top 5%;
- how to apply ConvNets to detect region-of-interests (ROIs).

Overview

Recap: Cross-Entropy vs. Hinge



Recap: Compute the Gradient

For a single data point, the hinge loss is :

$$L_i = \sum_{j \neq y_i} [\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)] \tag{1}$$

We can differentiate the loss function with respect to the weights. Taking the gradient with respect to w_{y_i} we obtain:

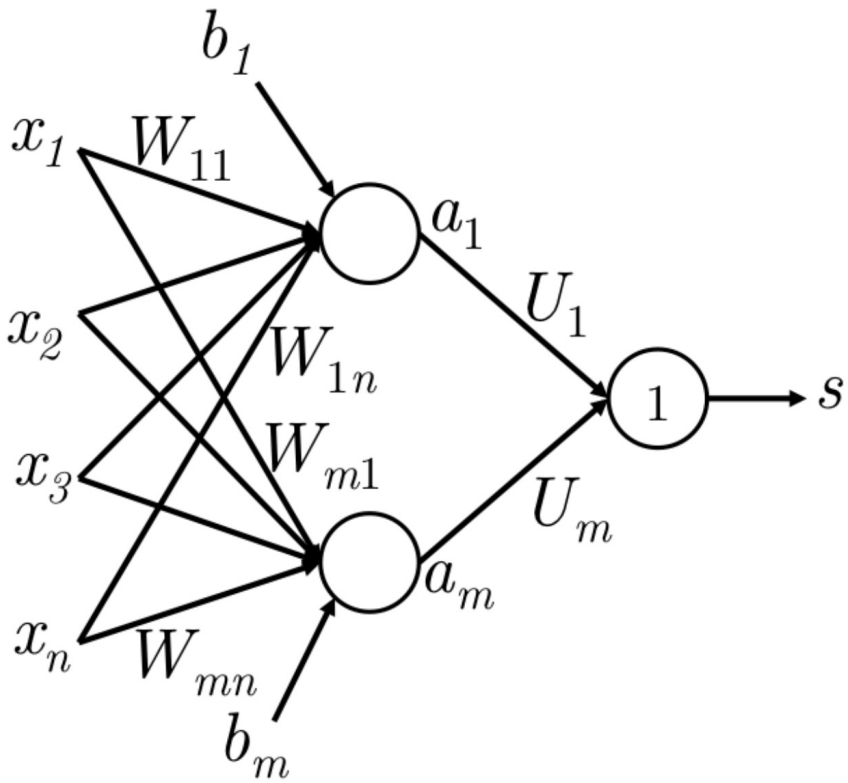
$$\frac{\partial L_i}{\partial w_{y_i}} = -[\sum_{j \neq y_i} \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)] x_i \tag{2}$$

For other rows where $j \neq y_i$ the gradient is:

$$\frac{\partial L_i}{\partial w_j} = \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i \tag{3}$$

Recap: A Single Hidden Layer Neural Network

A neural network is nothing but a stack of single neurons. One can think of activations as indicators of the presence of some weighted combination of features. We can then use a combination of these activations to perform classification tasks.



$$z = Wx + b$$
$$a = \sigma(z)$$
$$s = U^T a$$

Recap: Objective Funciton

Let's consider hinge loss as our objective function. If we call the score computed for "true" labeled data as s and the score computed for "false" labeled data as s_c. Then the optimization objective is:

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

where $s_c =$

Recap: Objective Funciton

Let's consider hinge loss as our objective function. If we call the score computed for "true" labeled data as s and the score computed for "false" labeled data as s_c. Then the optimization objective is:

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

where $s_c = U^T \sigma(W x_c + b)$ and $s =$

Recap: Objective Funciton

Let's consider hinge loss as our objective function. If we call the score computed for "true" labeled data as s and the score computed for "false" labeled data as s_c. Then the optimization objective is:

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

where $s_c = U^T\sigma(Wx_c + b)$ and $s = U^T\sigma(Wx + b)$.

Recap: Objective Funciton

Let's consider hinge loss as our objective function. If we call the score computed for "true" labeled data as s and the score computed for "false" labeled data as s_c. Then the optimization objective is:

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

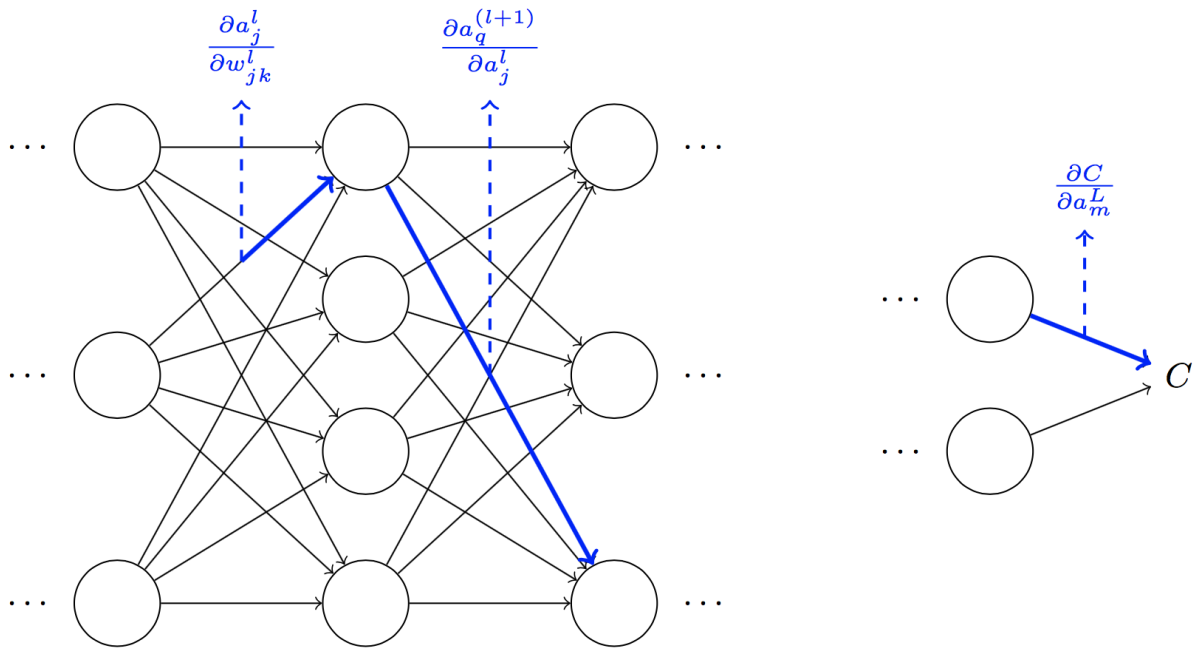
where $s_c = U^T \sigma(W x_c + b)$ and $s = U^T \sigma(W x + b)$.

Trick

We can scale this margin such that it is $\Delta = 1$ and let other parameters in the optimization problem adapt to this without any change in performance.

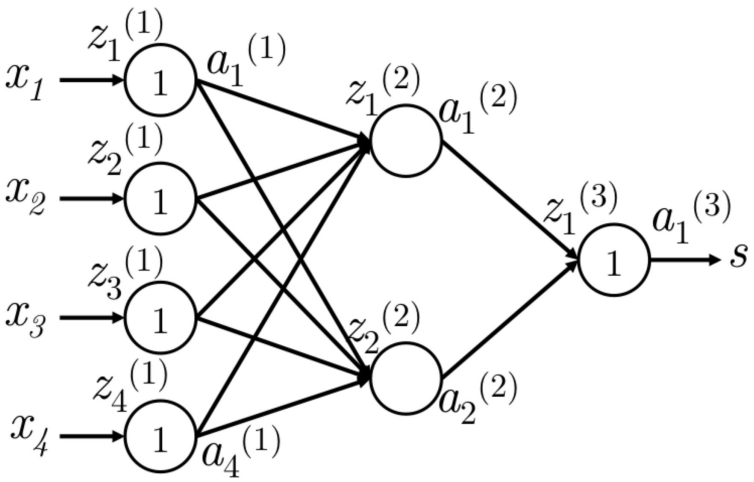
$$\text{minimize } J = \max(1 + s_c - s, 0) \tag{4}$$

Recap: The Big Picture of Backpropgation



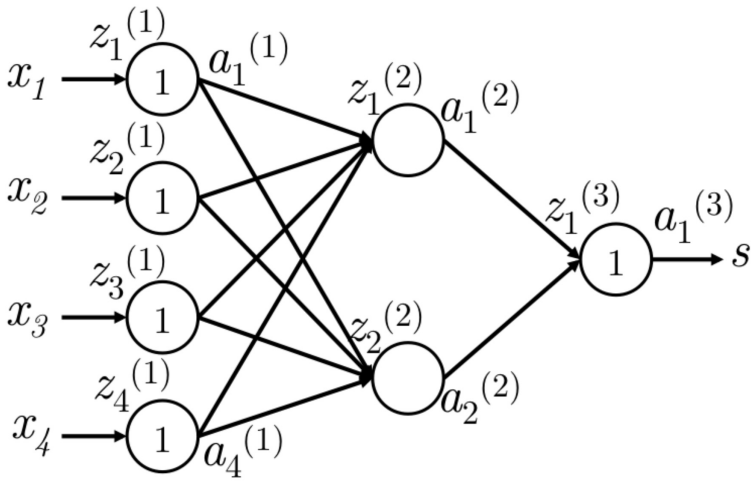
(1) Every edge between two neurons in the network is associated with a rate factor which is just the partial derivative of one neuron's activation with respect to the other; (2) The rate factor for a path is just the product of the rate factors along the path; (3) And the total rate of change $\frac{\partial C}{\partial w_{jk}^l}$ is just the sum of the rate factors of all possible paths.

Recap: Backpropagation Notation



- x_i is an input to the neural network.
- s is the output of the neural network.
- The j-th neuron of layer k receives the scalar input $z_j^{(k)}$ and produces the scalar activation output $a_j^{(k)}$.
- For the input layer, $x_j = z_j^{(1)} = a_j^{(1)}$.
- $W^{(k)}$ is the transfer/weights matrix that maps the output from the k-th layer to the input to the $(k+1)$-th.
- $\delta_j^{(k)}$ **is the backpropagated error calculated at** $z_j^{(k)}$**:** $\delta_j^{(k)} = \frac{\partial J}{\partial z_j^{(k)}}$**.**

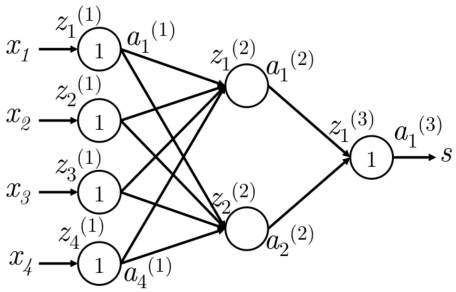
Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$



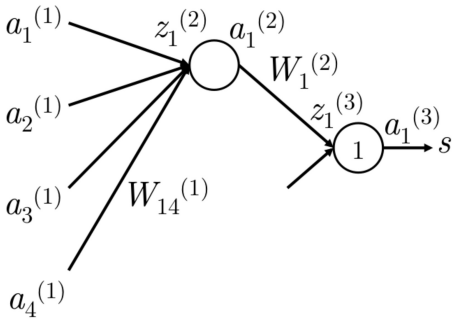
Suppose the cost $J = (1 + s_c - s)$ is positive and we want to perform the update of parameter $W_{14}^{(1)}$, we must realize that $W_{14}^{(1)}$ only contributes to $z_1^{(2)}$ and thus $a_1^{(2)}$. **Backpropagated gradients are only affected by values they contribute to.**

$$\frac{\partial J}{\partial s} = -1$$

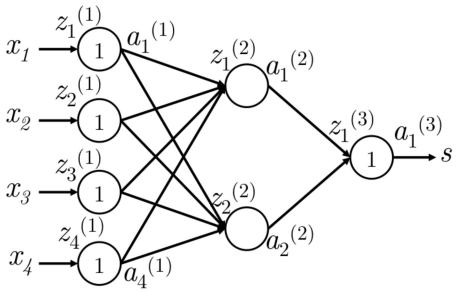
Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$



$$\frac{\partial s}{\partial W_{14}^{(1)}} =$$

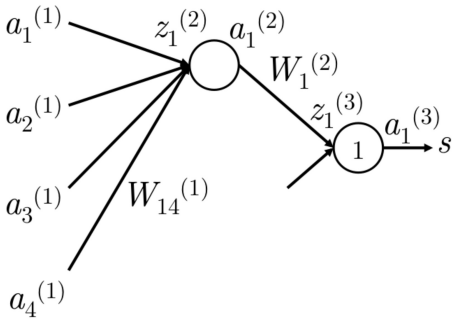


Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$

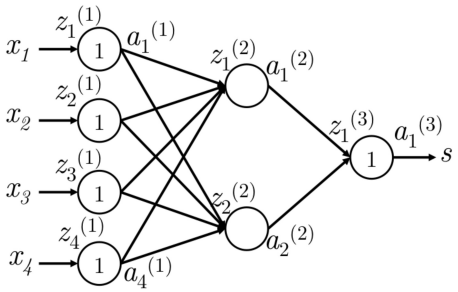


$$\frac{\partial s}{\partial W_{14}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{14}^{(1)}} = \frac{\partial W_1^{(2)} a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$\Rightarrow W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}} =$$



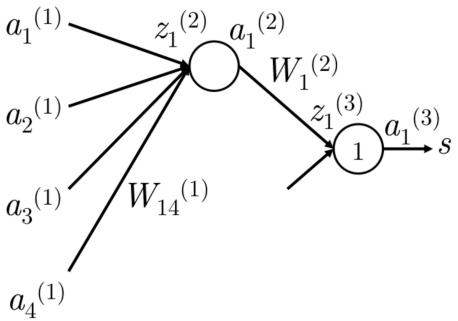
Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$



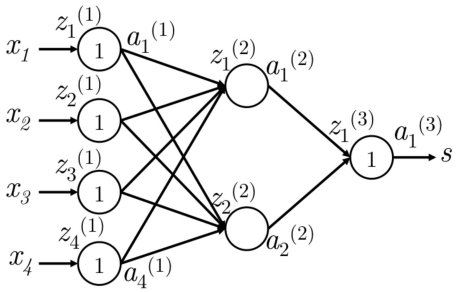
$$\frac{\partial s}{\partial W_{14}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{14}^{(1)}} = \frac{\partial W_1^{(2)} a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$\Rightarrow W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$=$$



Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$

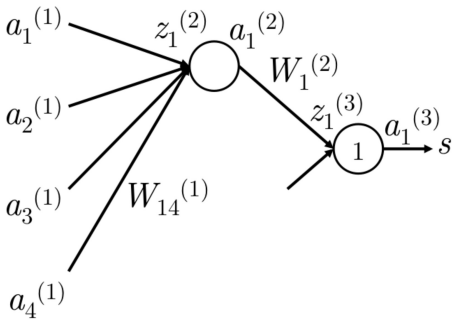


$$\frac{\partial s}{\partial W_{14}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{14}^{(1)}} = \frac{\partial W_1^{(2)} a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}}$$

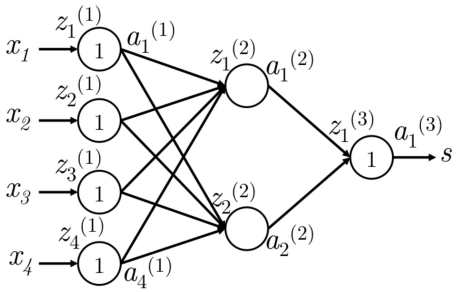
$$\Rightarrow W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$= W_1^{(2)} \sigma'(z_1^{(2)}) \frac{\partial}{\partial W_{14}^{(1)}} (b_1^{(1)} + \sum_k a_k^{(1)} W_{1k}^{(1)})$$

$$=$$



Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$



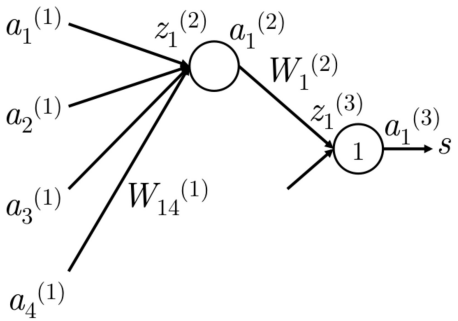
$$\frac{\partial s}{\partial W_{14}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{14}^{(1)}} = \frac{\partial W_1^{(2)} a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$\Rightarrow W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}$$

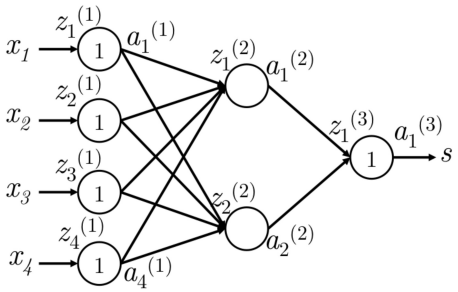
$$= W_1^{(2)} \sigma'(z_1^{(2)}) \frac{\partial}{\partial W_{14}^{(1)}} (b_1^{(1)} + \sum_k a_k^{(1)} W_{1k}^{(1)})$$

$$= W_1^{(2)} \sigma'(z_1^{(2)}) a_4^{(1)}$$

$$=$$



Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$



$$\frac{\partial s}{\partial W_{14}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{14}^{(1)}} = \frac{\partial W_1^{(2)} a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}}$$

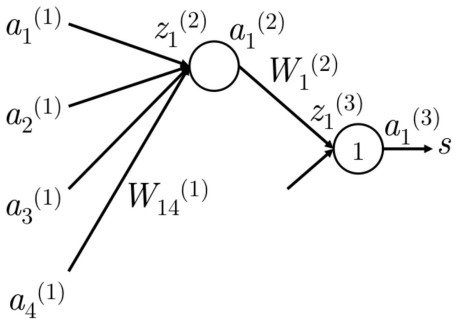
$$\Rightarrow W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$= W_1^{(2)} \sigma'(z_1^{(2)}) \frac{\partial}{\partial W_{14}^{(1)}} (b_1^{(1)} + \sum_k a_k^{(1)} W_{1k}^{(1)})$$

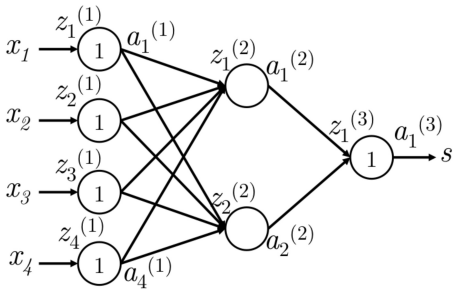
$$= W_1^{(2)} \sigma'(z_1^{(2)}) a_4^{(1)}$$

$$= \frac{\partial J}{\partial z_1^{(2)}} a_4^{(1)}$$

$$=$$



Recap: Compute $\frac{\partial J}{\partial W_{14}^{(1)}}$



$$\frac{\partial s}{\partial W_{14}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{14}^{(1)}} = \frac{\partial W_1^{(2)} a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}}$$

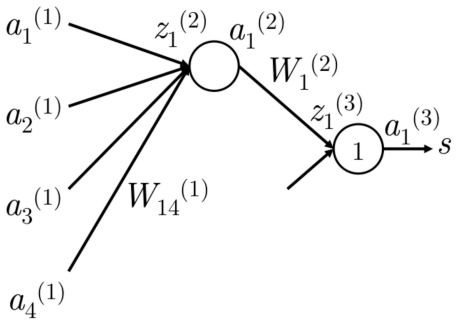
$$\Rightarrow W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial W_{14}^{(1)}} = W_1^{(2)} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}$$

$$= W_1^{(2)} \sigma'(z_1^{(2)}) \frac{\partial}{\partial W_{14}^{(1)}} (b_1^{(1)} + \sum_k a_k^{(1)} W_{1k}^{(1)})$$

$$= W_1^{(2)} \sigma'(z_1^{(2)}) a_4^{(1)}$$

$$= \frac{\partial J}{\partial z_1^{(2)}} a_4^{(1)}$$

$$= \delta_1^{(2)} a_4^{(1)}$$

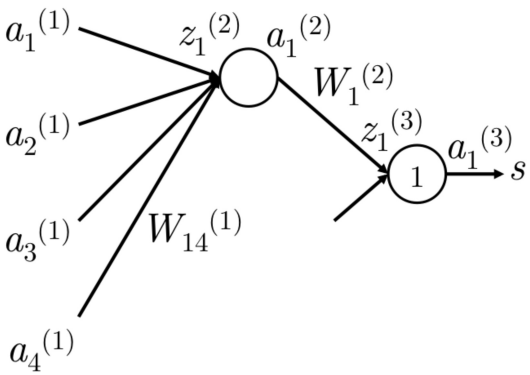


Homework 0

Derive the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation, i.e., find the gradients with respect to the softmax input vector θ, when the prediction is made by $\hat{y} = softmax(\theta)$. Remember the cross entropy function is:

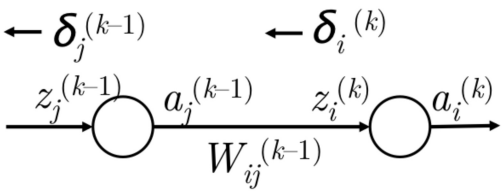
$$CE(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

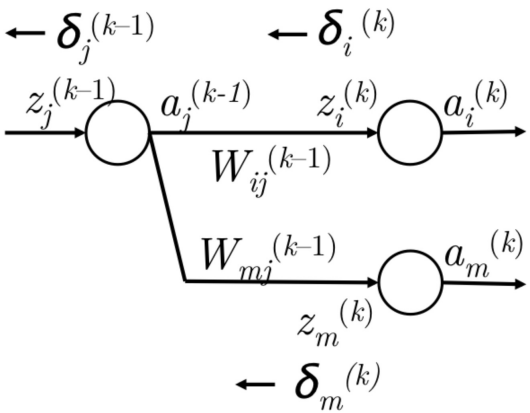
Error Distribution Interpretation of Backpropagation



1. We start with an error signal of 1 propagating backwards from $a_1^{(3)}$.
2. We then multiply this error by the local gradient of the neuron which maps $z_1^{(3)}$ to $a_1^{(3)}$. This happens to be 1 in this case and thus, the error is still 1. This is now known as $\delta_1^{(3)}$.
3. Then the error reaches to $a_1^{(2)}$ is the error at $z_1^{(3)}$ mutiplies $W_1^{(2)}$, which is $\delta_1^{(3)} W_1^{(2)} = W_1^{(2)}$.
4. As we did in step 2, we need to move the error across the neuron which maps $z_1^{(2)}$ to $a_1^{(2)}$. We do this by multiplying the error signal at $a_1^{(2)}$ by the local gradient of the neuron which happens to be $\sigma'(z_1^{(2)})$.
5. The error signal at $z_1^{(2)}$ is $W_1^{(2)}\sigma'(z_1^{(2)})$, which is known to be $\delta_1^{(2)}$.
6. Finally we need to distribute the "fair share" of the error to $W_{14}^{(1)}$ by simply multiplying it by the input it was responsible for forwarding, which happens to be $a_4^{(1)}$.
7. Thus, the gradient of loss with respect to $W_{14}^{(1)}$ is calculated to be $W_1^{(2)}\sigma'(z_1^{(2)})a_4^{(1)}$.

Backpropagate $\delta^{(k)}$ to $\delta^{(k-1)}$

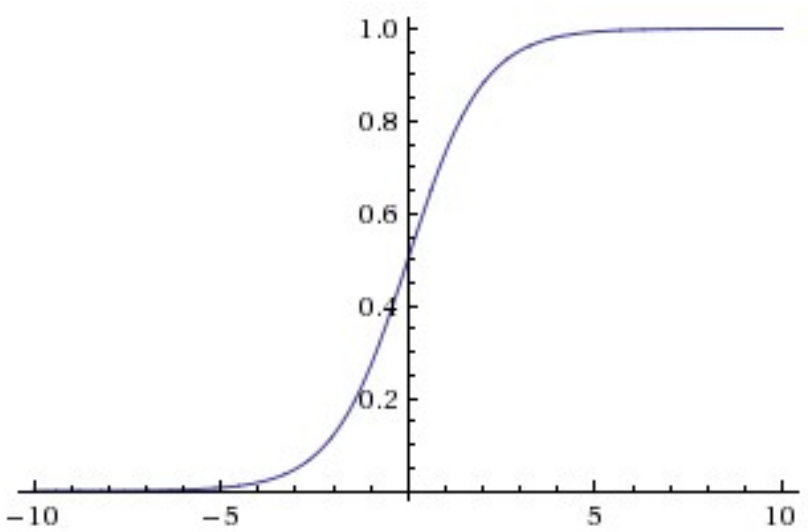


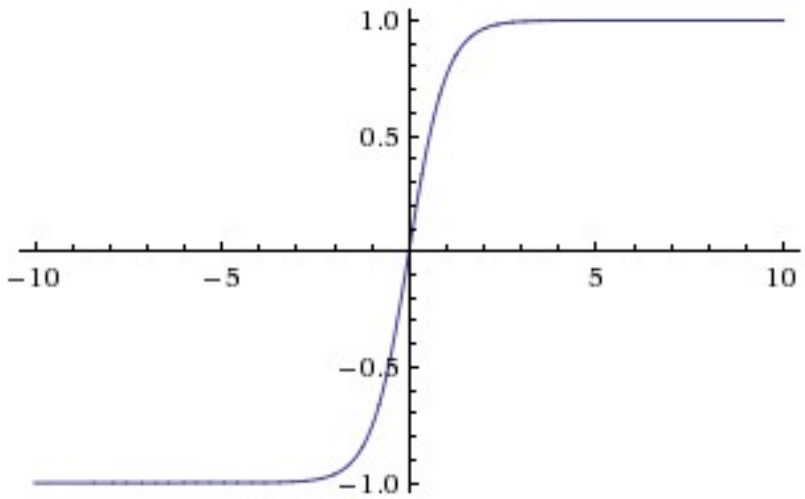


1. We have error $\delta_i^{(k)}$ propagating backwards from $z_j^{(k)}$, i.e. neuron i at layer k.
2. We propagate this error backwards to $a_j^{(k-1)}$ by multiplying $\delta_i^{(k)}$ by the path weight $W_{ij}^{(k-1)}$.
3. Thus, the error received at $a_j^{(k-1)}$ is $\delta_i^{(k)} W_{ij}^{(k-1)}$.
4. However, $a_j^{(k-1)}$ may have been forwarded to multiple nodes in the next layer(i.e. node m in layer k).
5. Thus, the total error received at $a_j^{(k-1)}$ is $\delta_i^{(k)} W_{ij}^{(k-1)} + \delta_m^{(k)} W_{mj}^{(k-1)}$.
6. In fact, we can generalize this to be $\sum_i \delta_i^{(k)} W_{ij}^{(k-1)}$.
7. Now, we can have the correct error at $a_j^{(k-1)}$, we move it across neuron j at layer $k-1$ by multiplying with the local gradient $\sigma'(z_j^{(k-1)})$.
8. Finally, the error that reaches at $z_j^{(k-1)}$, called $\delta_j^{(k-1)}$ is $\sigma'(z_j^{(k-1)}) \sum_i \delta_i^{(k)} W_{ij}^{(k-1)}$.

Activation Functions

- **Sigmoid.** $\sigma(x) = \frac{1}{1+\exp(-x)}$
- **Tanh.** $tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$



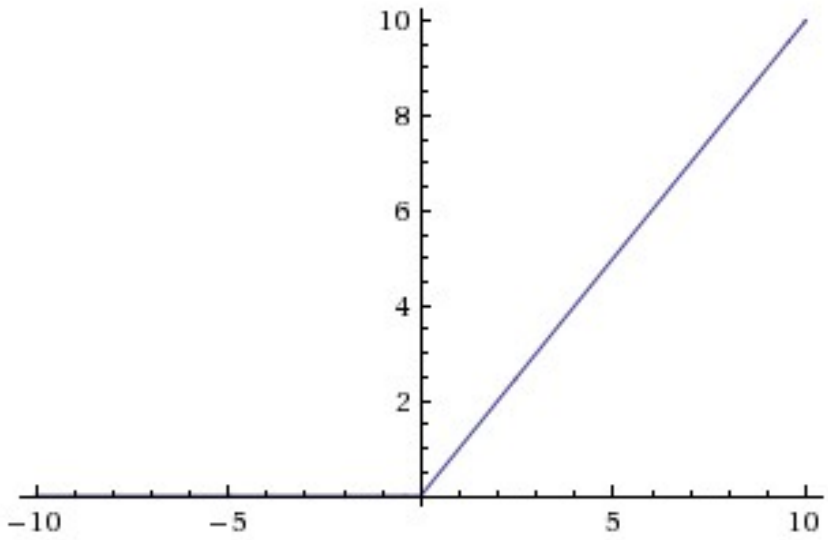


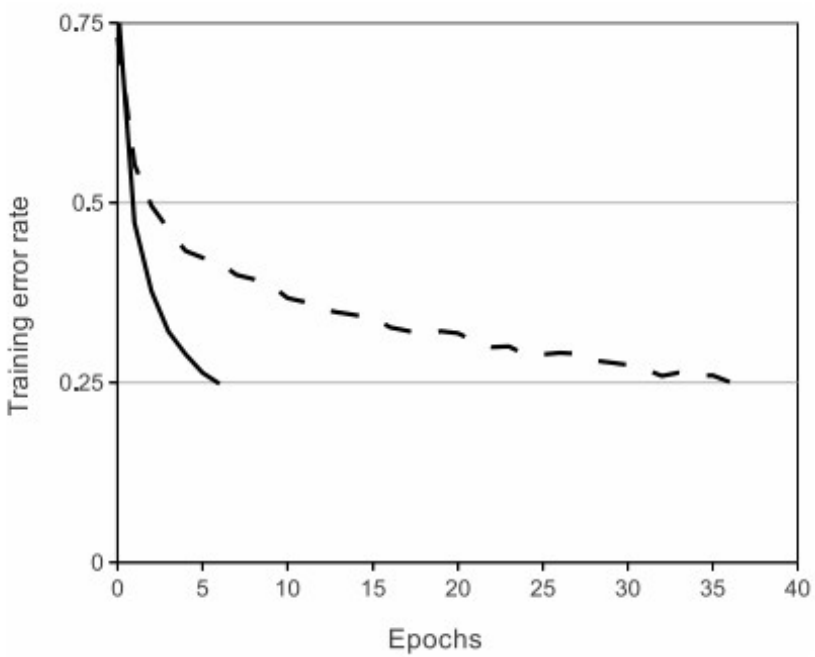
Sigmoids and Tanhs may saturate and kill gradients!

Preferred Activation Function - ReLU

Rectified Linear Units. $f(x) = \max(0, x)$

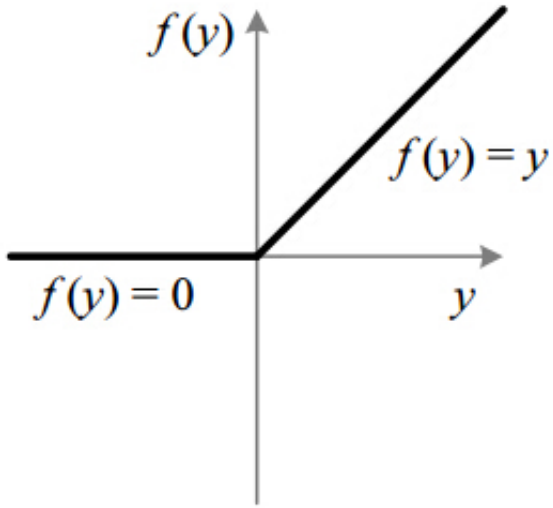
- $(+)$ Accelerate the convergence significantly($\times 6$).
- $(+)$ More efficient implementation compared with exponencials in Sigmoid/Tanh.
- $(-)$ ReLU units can be "dead" during training.

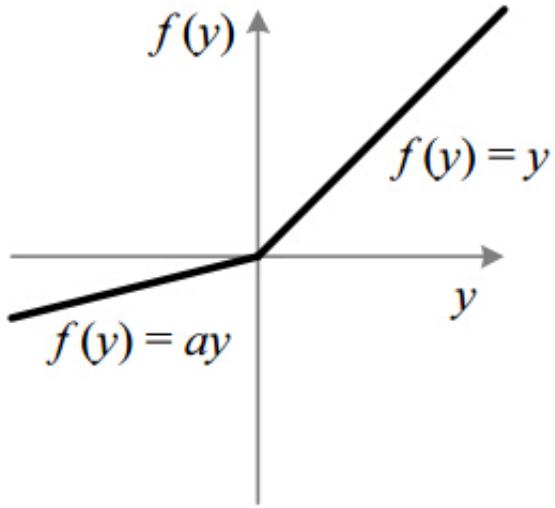




Leaky ReLU

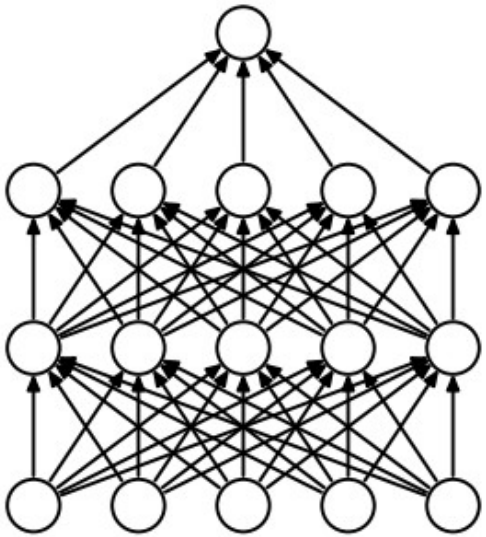
Leaky ReLU fixes the "dying ReLU" problem. $f(x) = \max(ax, x)$ e.g. $a = 0.3$



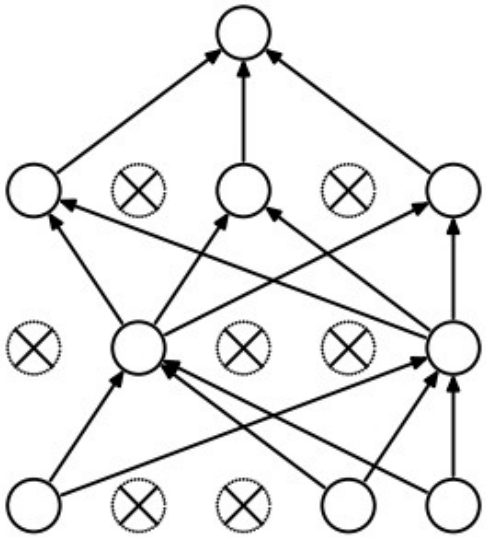


Regularization for Neural Networks - Dropout

Training: Sampling a sub-network within the full Neural Network.
Testing: Ensembles of all sub-networks(exponentially-sized) without dropout.



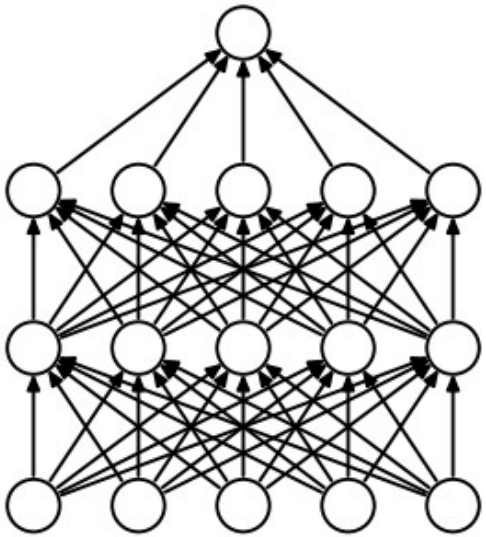
(a) Standard Neural Net



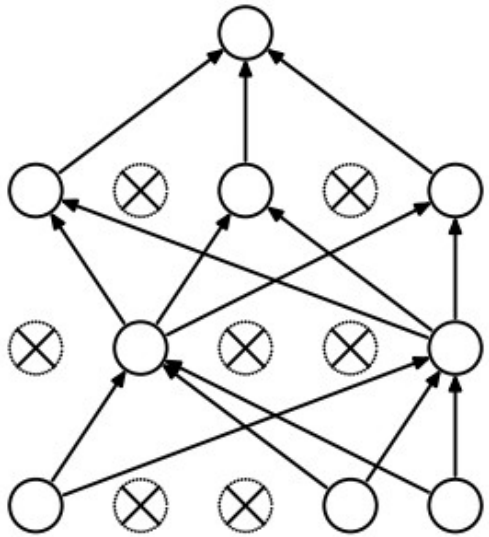
(b) After applying dropout.

Regularization for Neural Networks - Dropout

Training: Sampling a sub-network within the full Neural Network.
Testing: Ensembles of all sub-networks(exponentially-sized) without dropout.



(a) Standard Neural Net

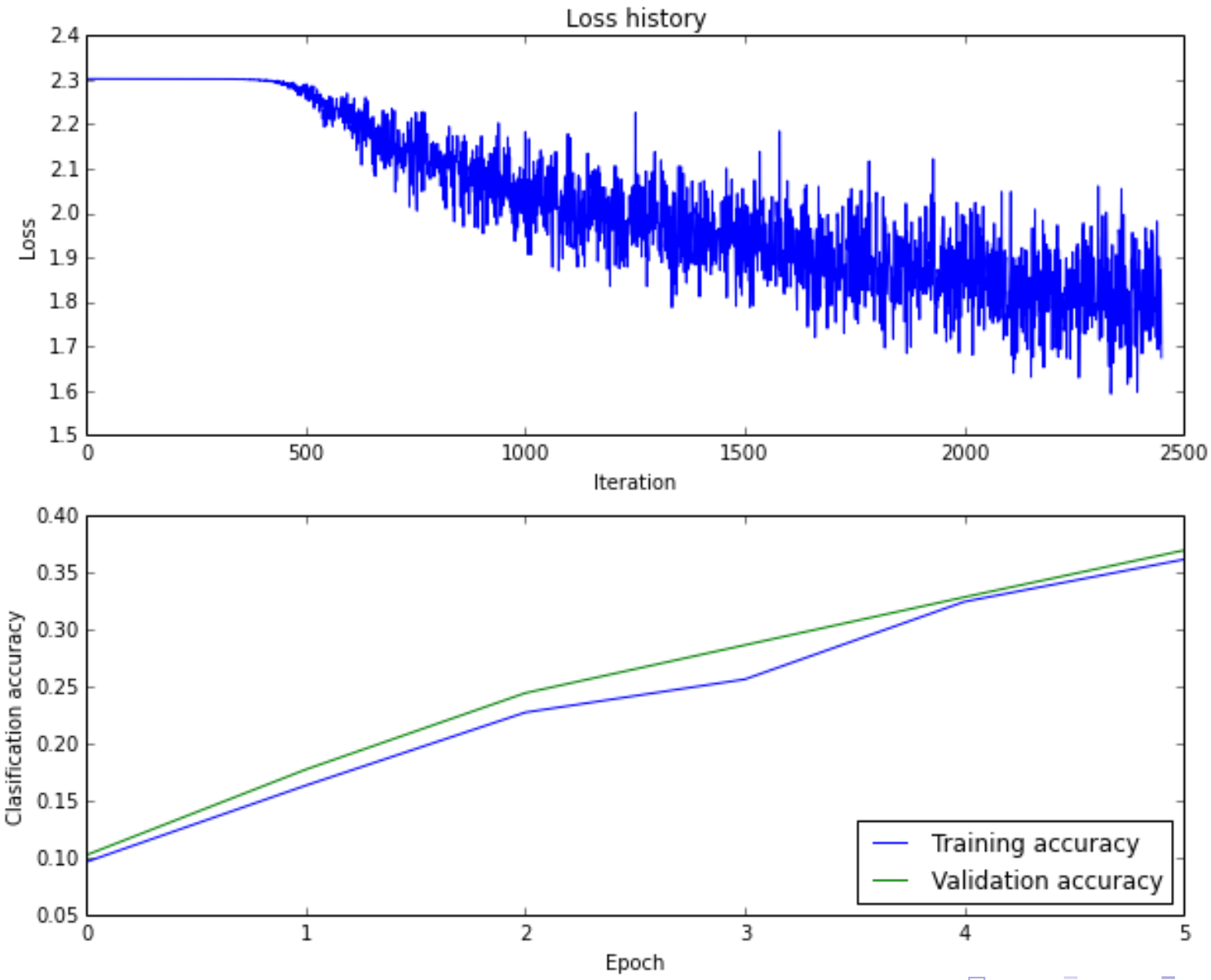


(b) After applying dropout.

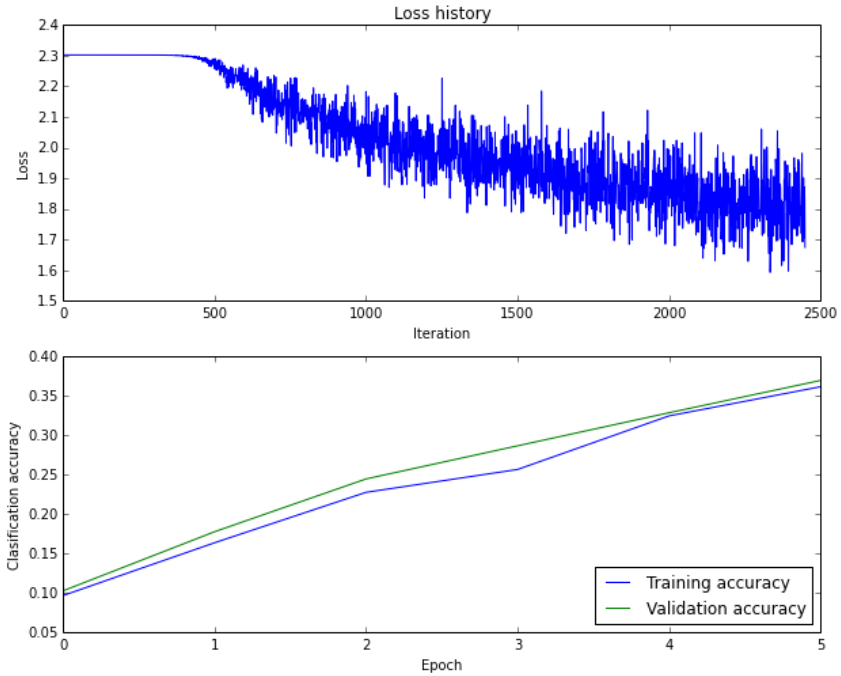
Example (Numpy Code)

```
H1 = np.maximum(0, np.dot(W1, X) + b1) # forward pass
U1 = np.random.rand(H1.shape) < p # dropout mask
H1 *= U1 # drop
```

Tuning Parameters for Neural Networks

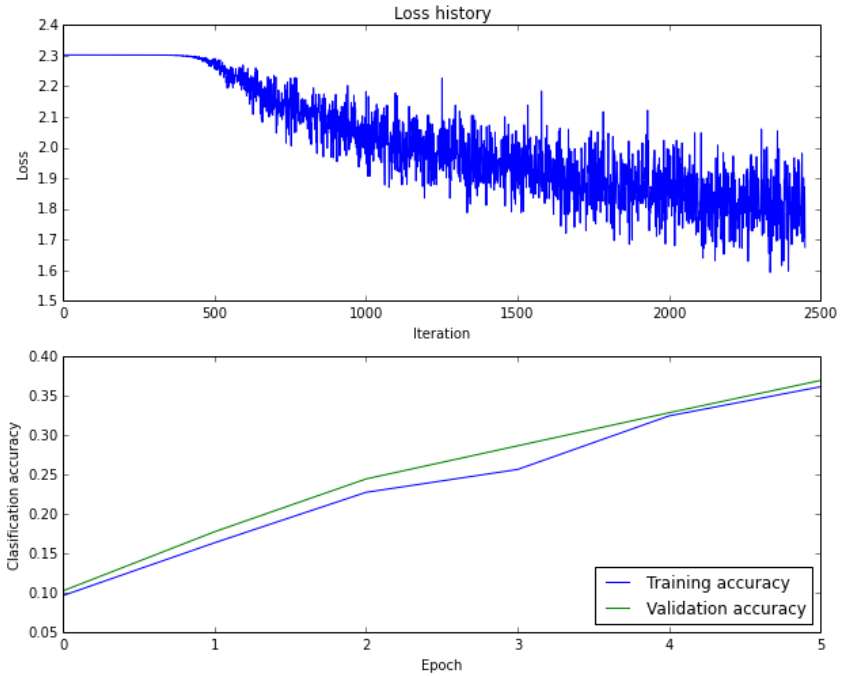


Tuning Parameters for Neural Networks



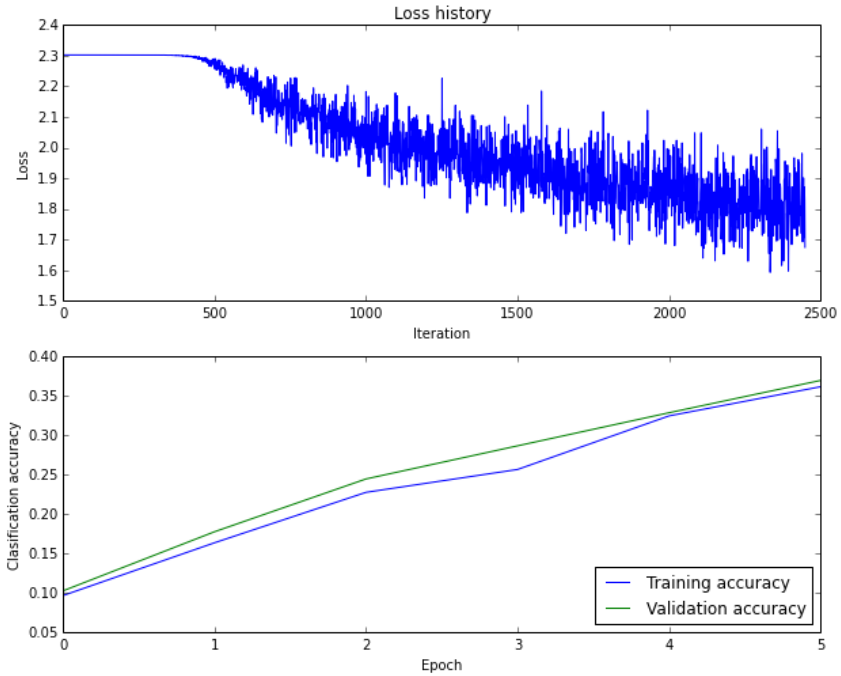
1. The loss is decreasing more or less linearly,

Tuning Parameters for Neural Networks



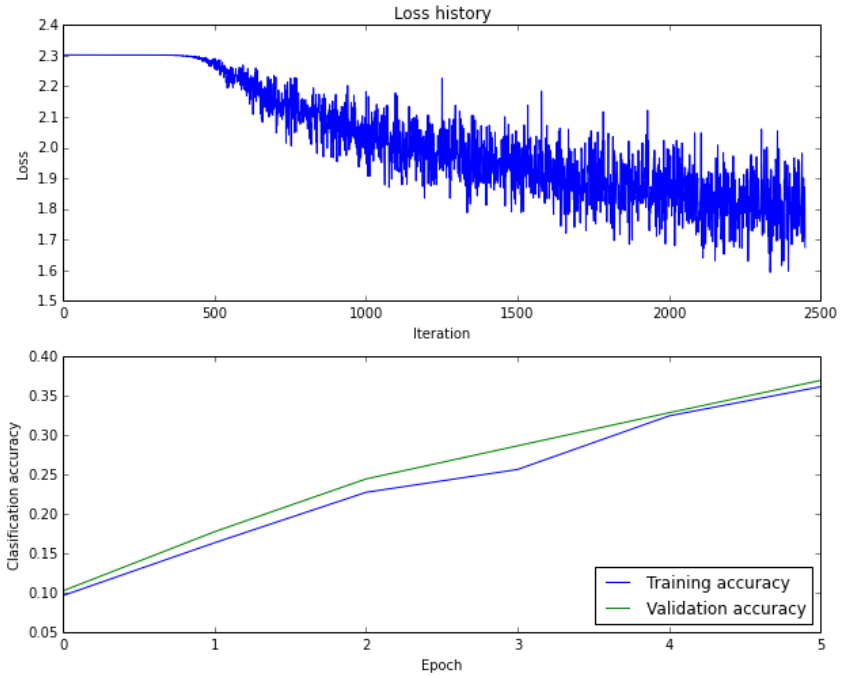
1. The loss is decreasing more or less linearly, indicating learning rate may be too low.

Tuning Parameters for Neural Networks



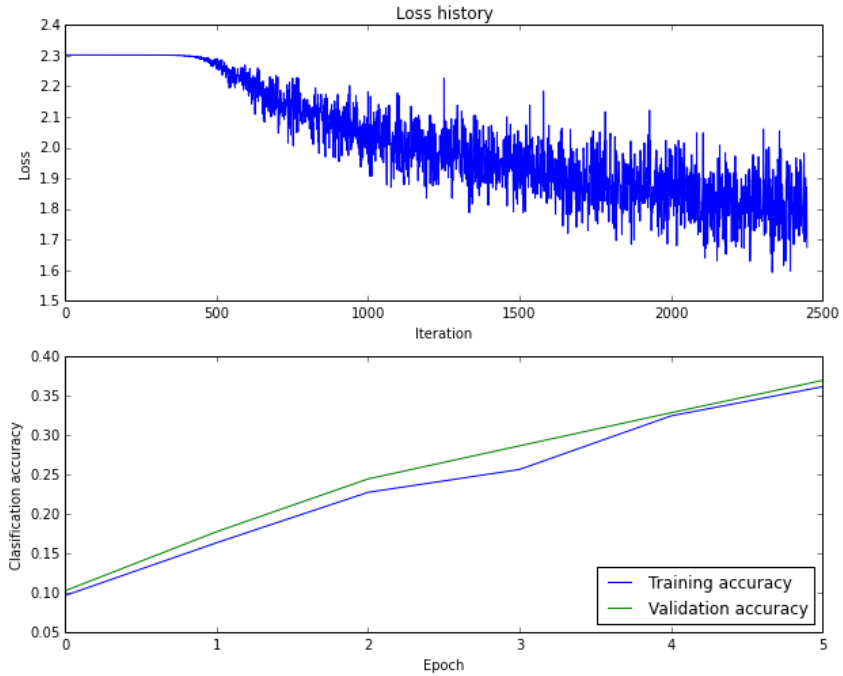
1. The loss is decreasing more or less linearly, indicating learning rate may be too low.
2. The loss fluctuates a lot,

Tuning Parameters for Neural Networks



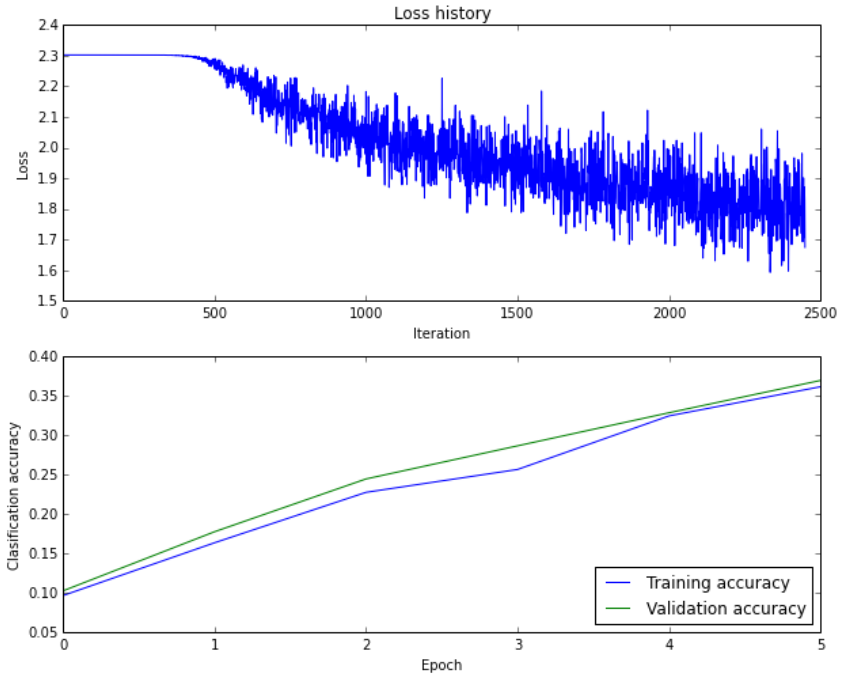
1. The loss is decreasing more or less linearly, indicating learning rate may be too low.
2. The loss fluctuates a lot, suggesting batch size may be too small.

Tuning Parameters for Neural Networks



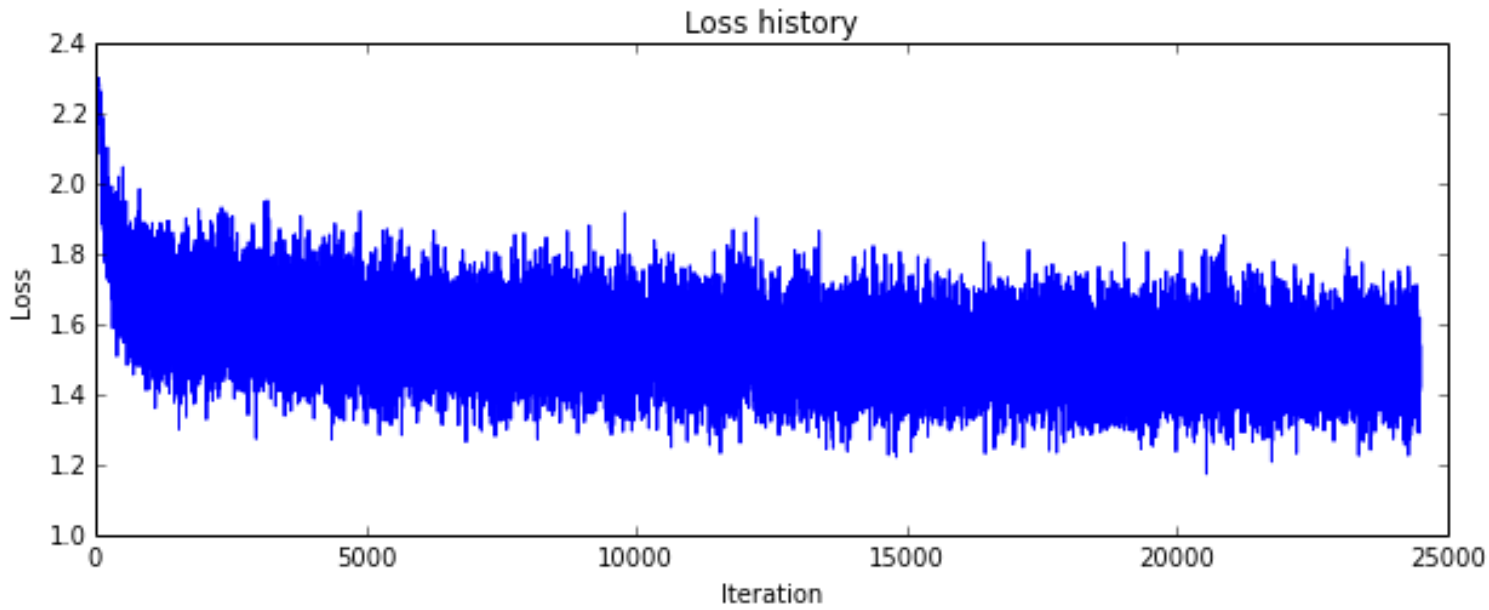
1. The loss is decreasing more or less linearly, indicating learning rate may be too low.
2. The loss fluctuates a lot, suggesting batch size may be too small.
3. There is no gap between the training and validation accuracy(Overfitting?),

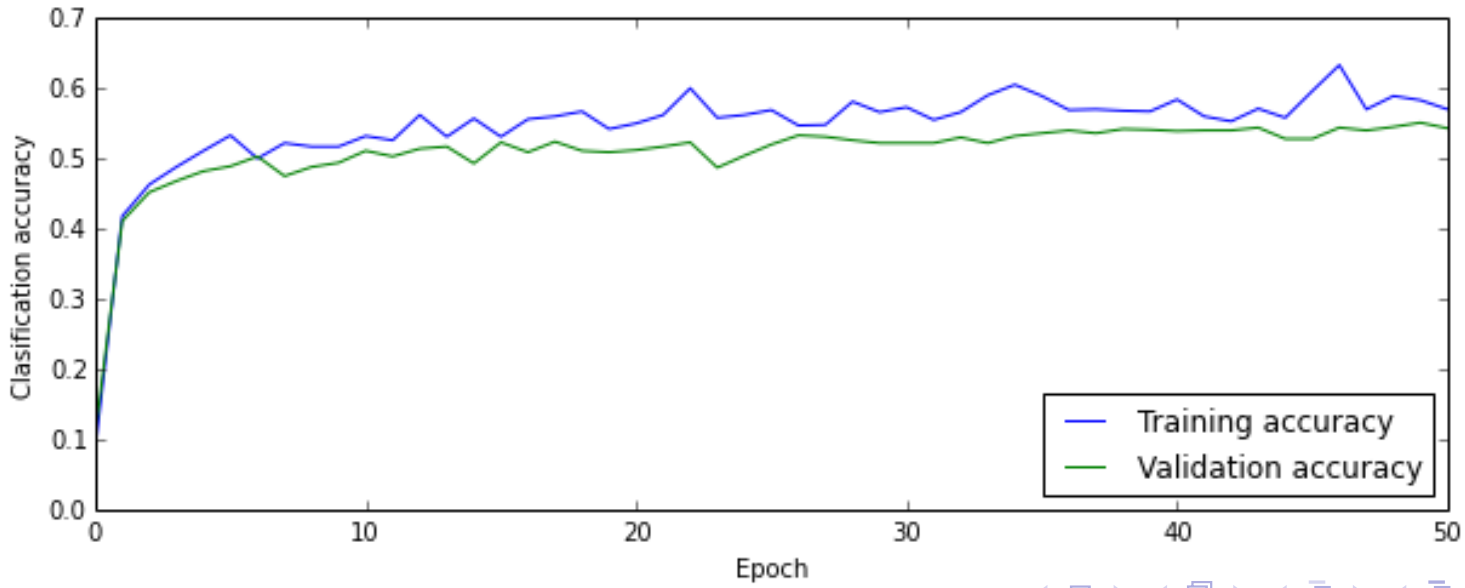
Tuning Parameters for Neural Networks



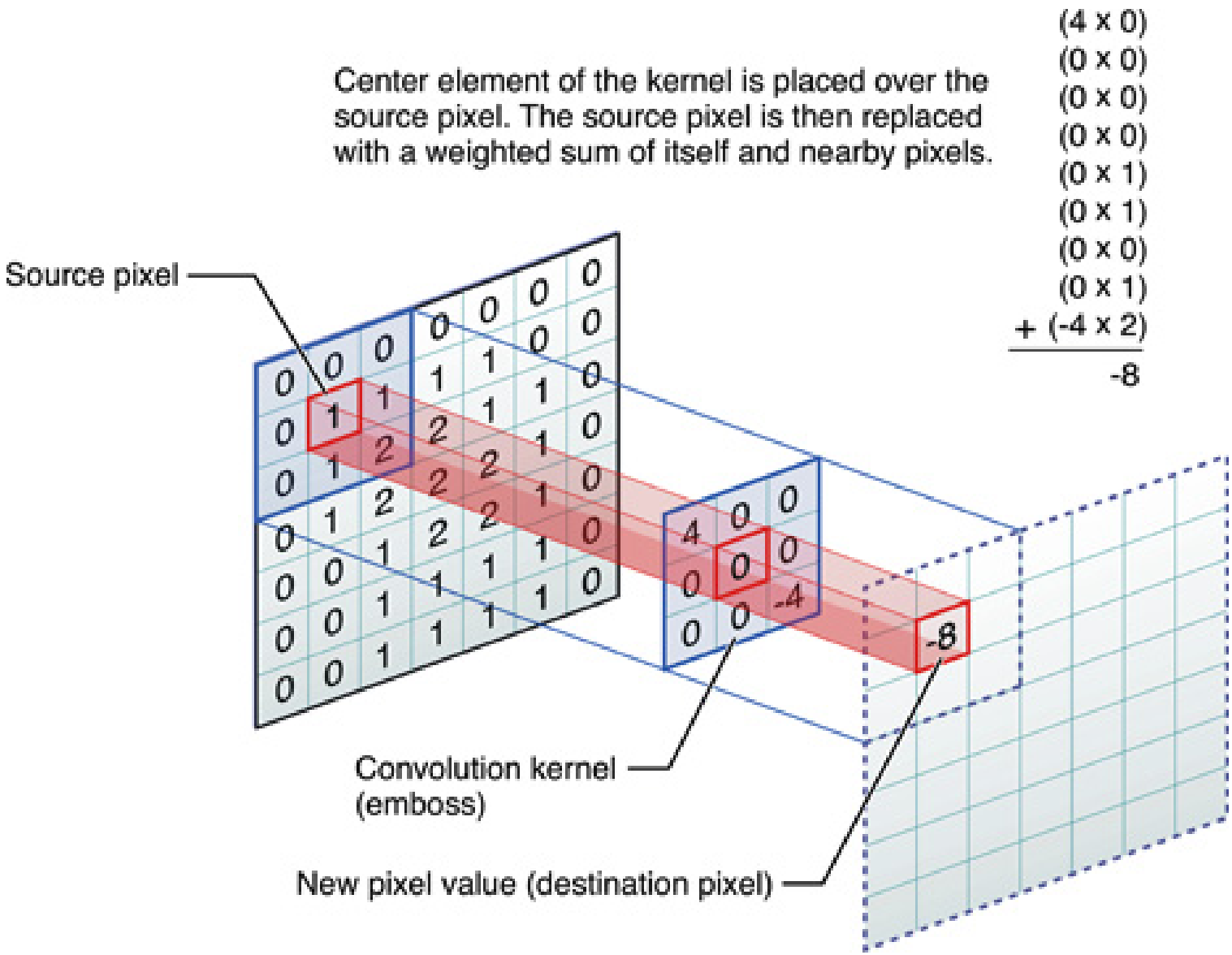
1. The loss is decreasing more or less linearly, indicating learning rate may be too low.
2. The loss fluctuates a lot, suggesting batch size may be too small.
3. There is no gap between the training and validation accuracy(~~Overfitting?~~low compacity), and we should increase training size(what if size is limited?).

Tuning Parameters for Neural Networks

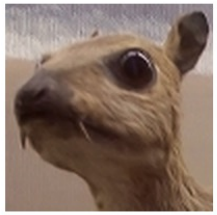
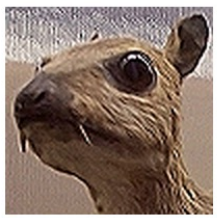




Convolution Operator

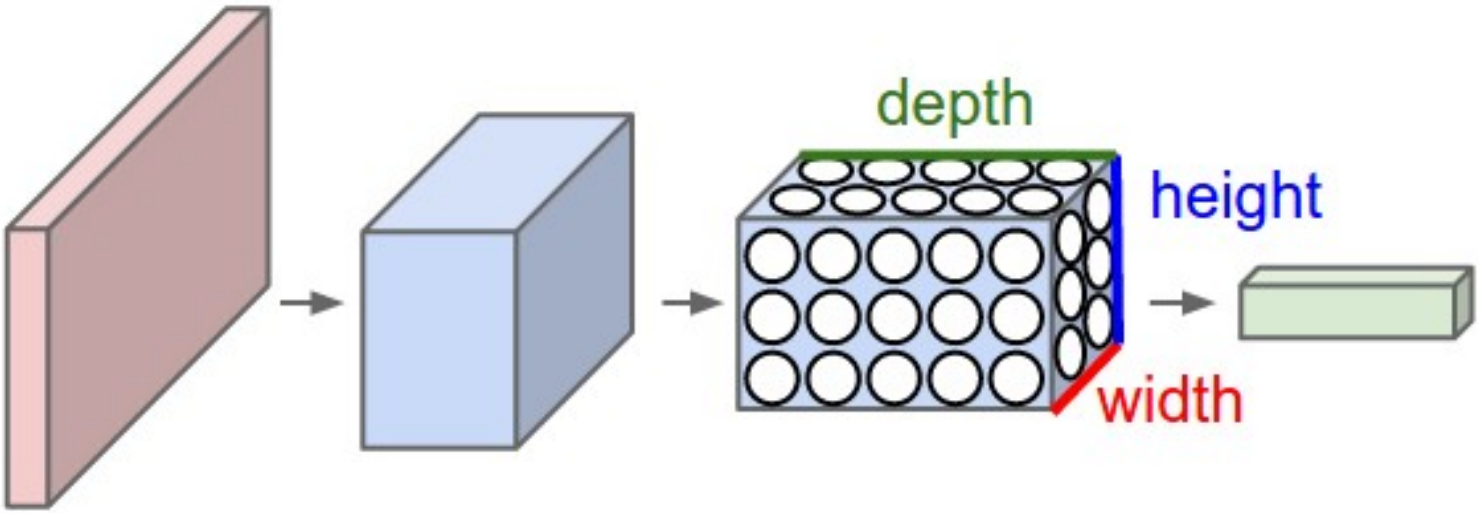


Convolutional Kernels

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

ConvNet's Input Are Images

ConvNet architectures make the explicit assumption that the inputs are images(e.g. 32x32x3), which allows us to encode certain properties into the architecture. Unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth.** Note that the final output layer has dimensions: 1x1xK.

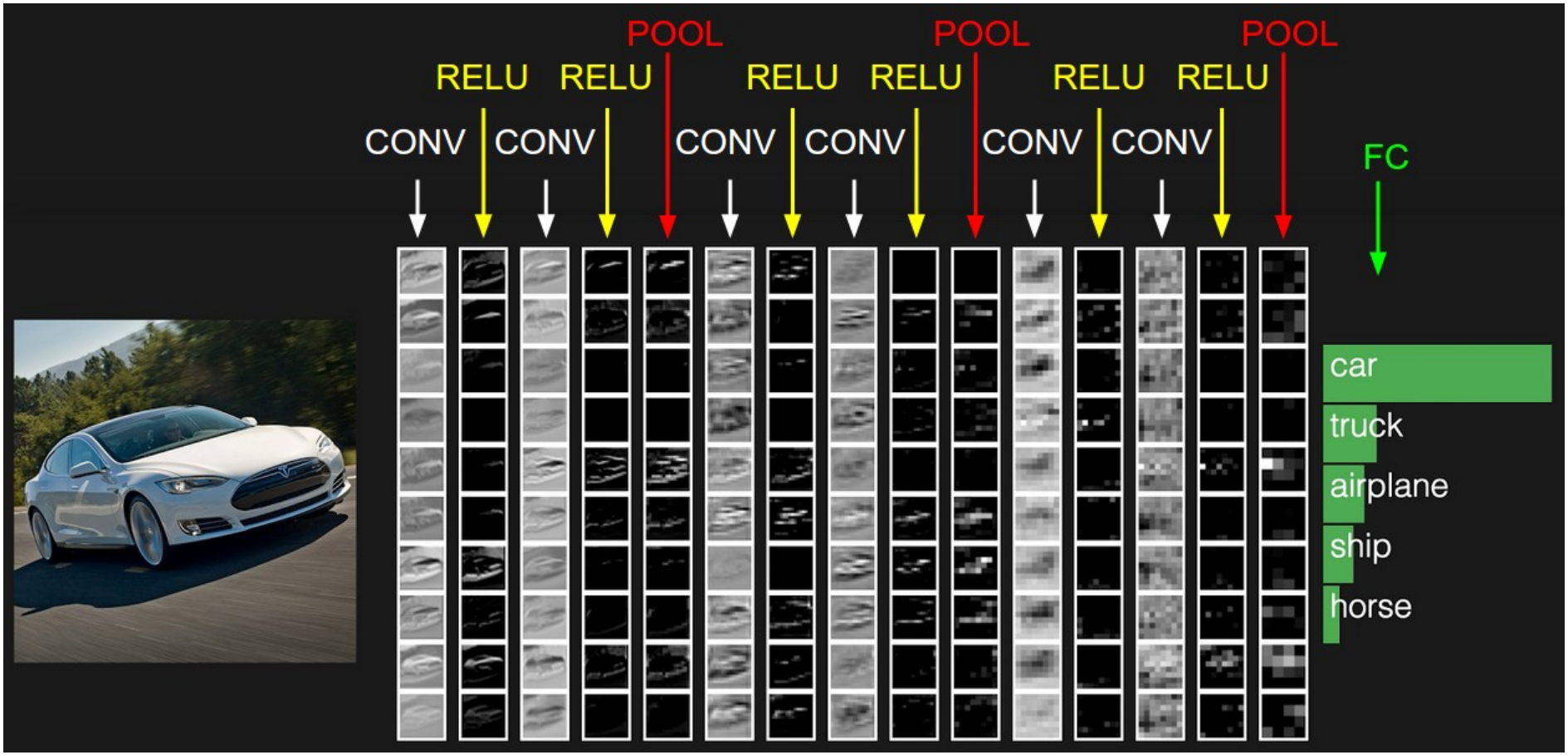


ConvNets Architecture Overview

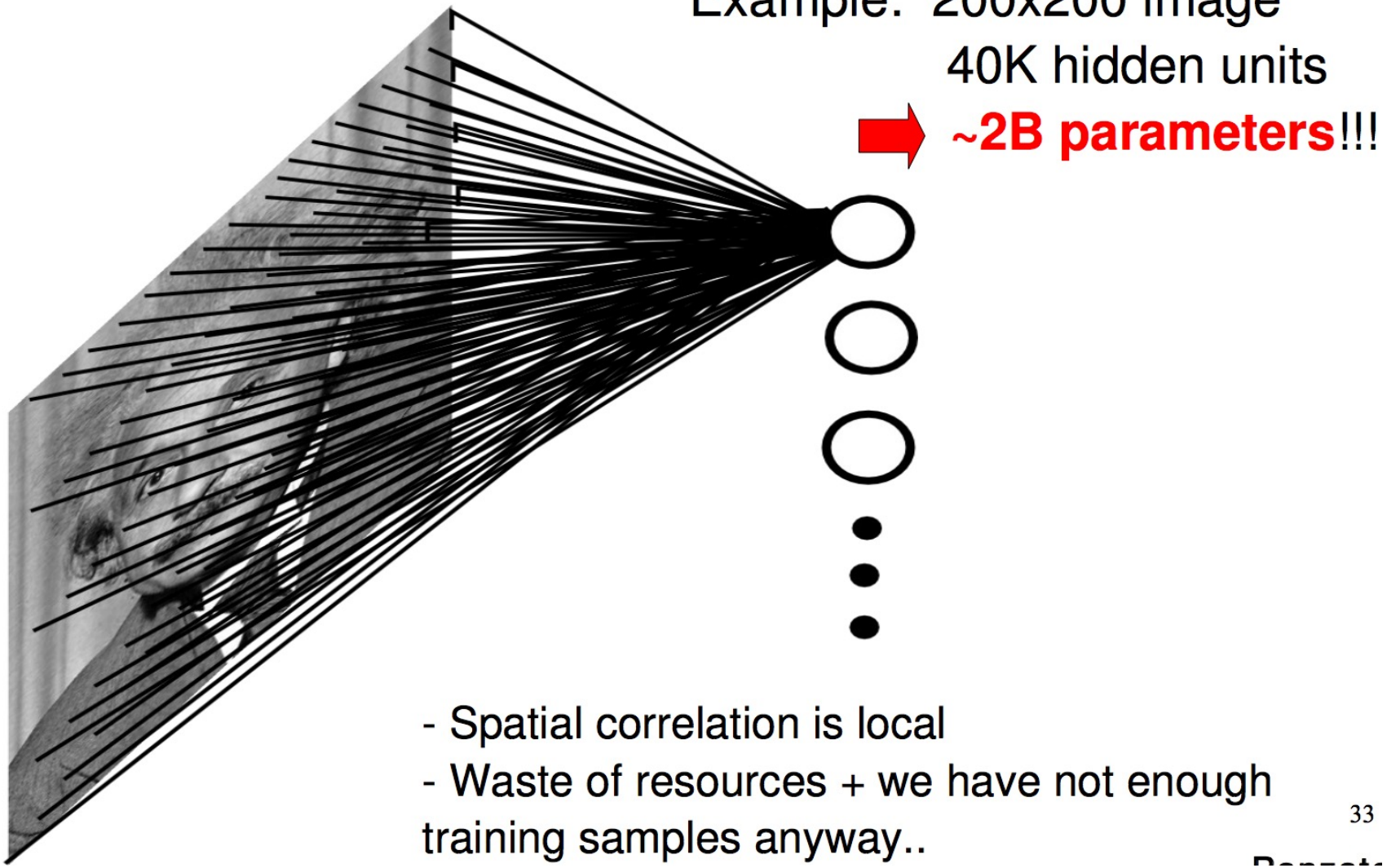
A simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- **INPUT** [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- **CONV** layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and the region they are connected to in the input volume. This may result in volume such as [32x32x12].
- **RELU** layer will apply an elementwise activation function, such as the **max(0,x)**. This leaves the size of the volume unchanged ([32x32x12]).
- **POOL** layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- **FC** (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10.

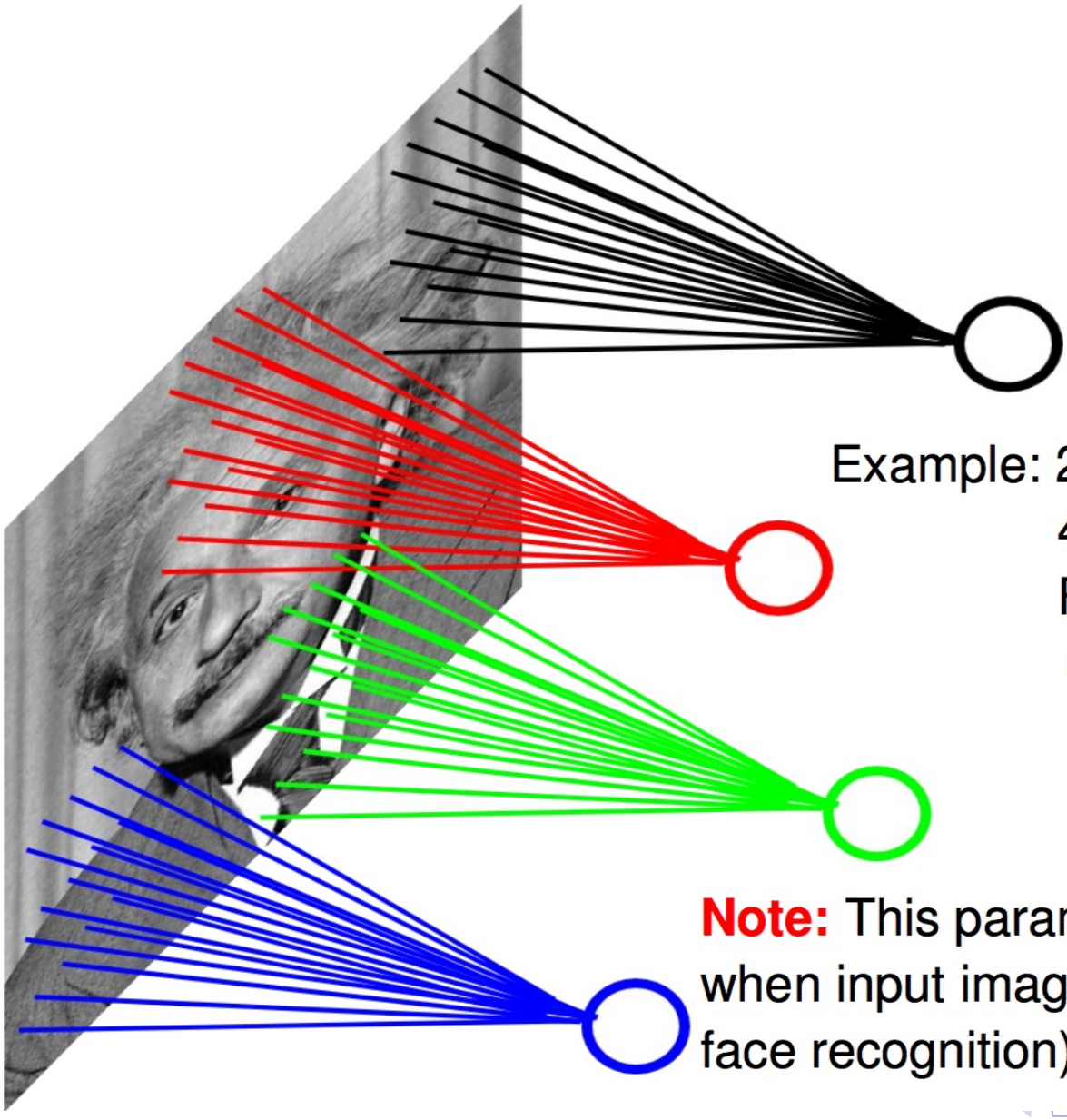
Architecture Visualization



Motivation of Convolutional Layer



Local Connectivity

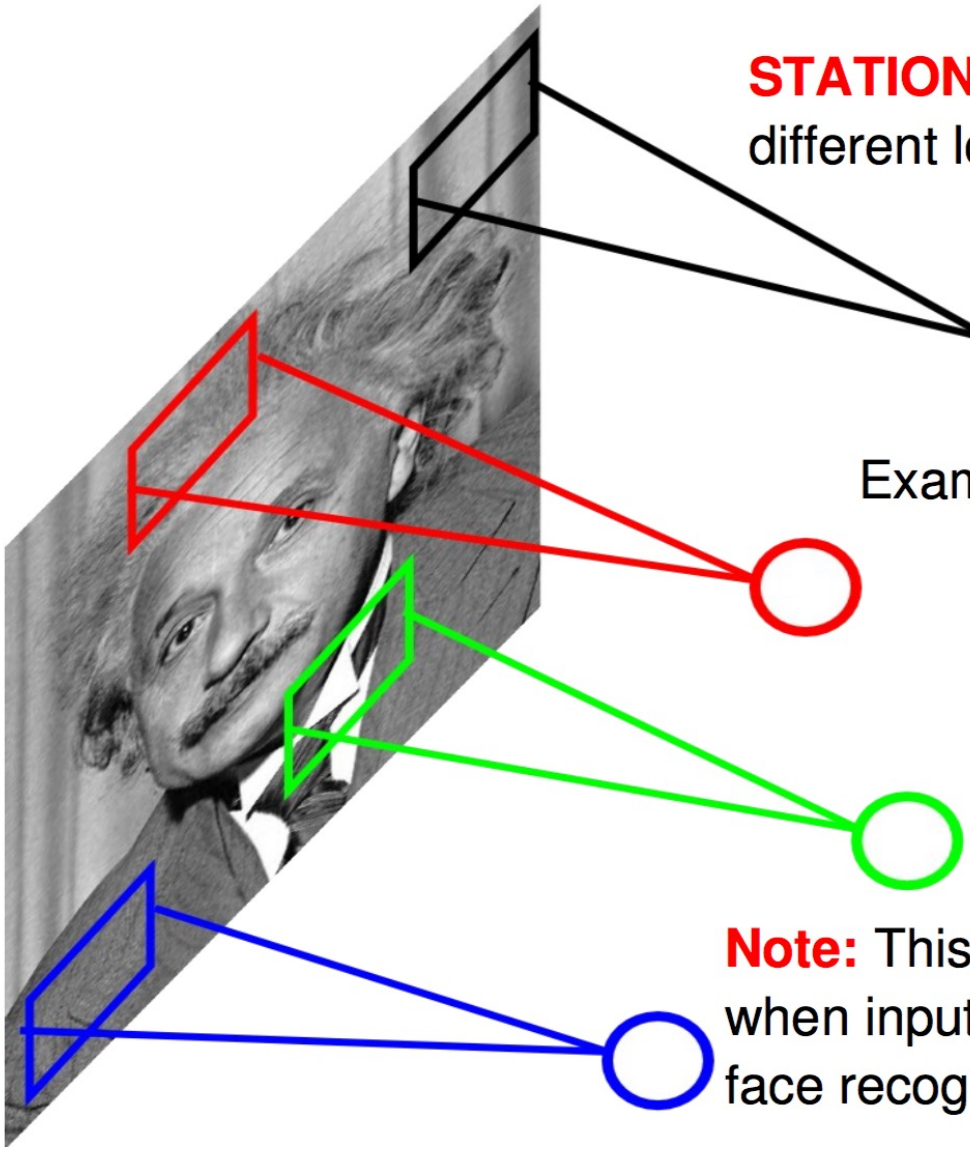


Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

Ranza

Local Connectivity



STATIONARITY? Statistics is similar at different locations

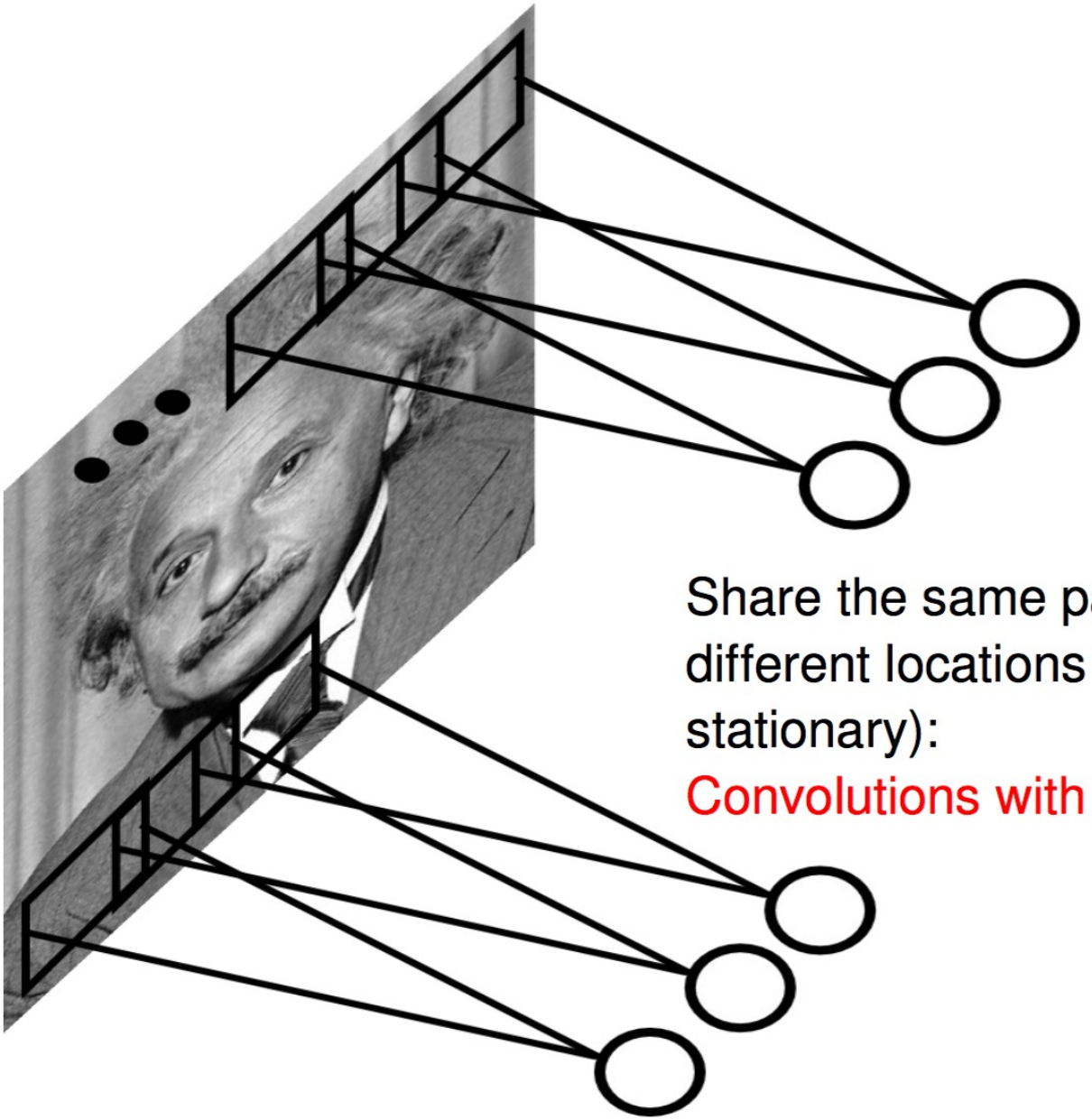
Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

35

Ranzato

Parameters Sharing

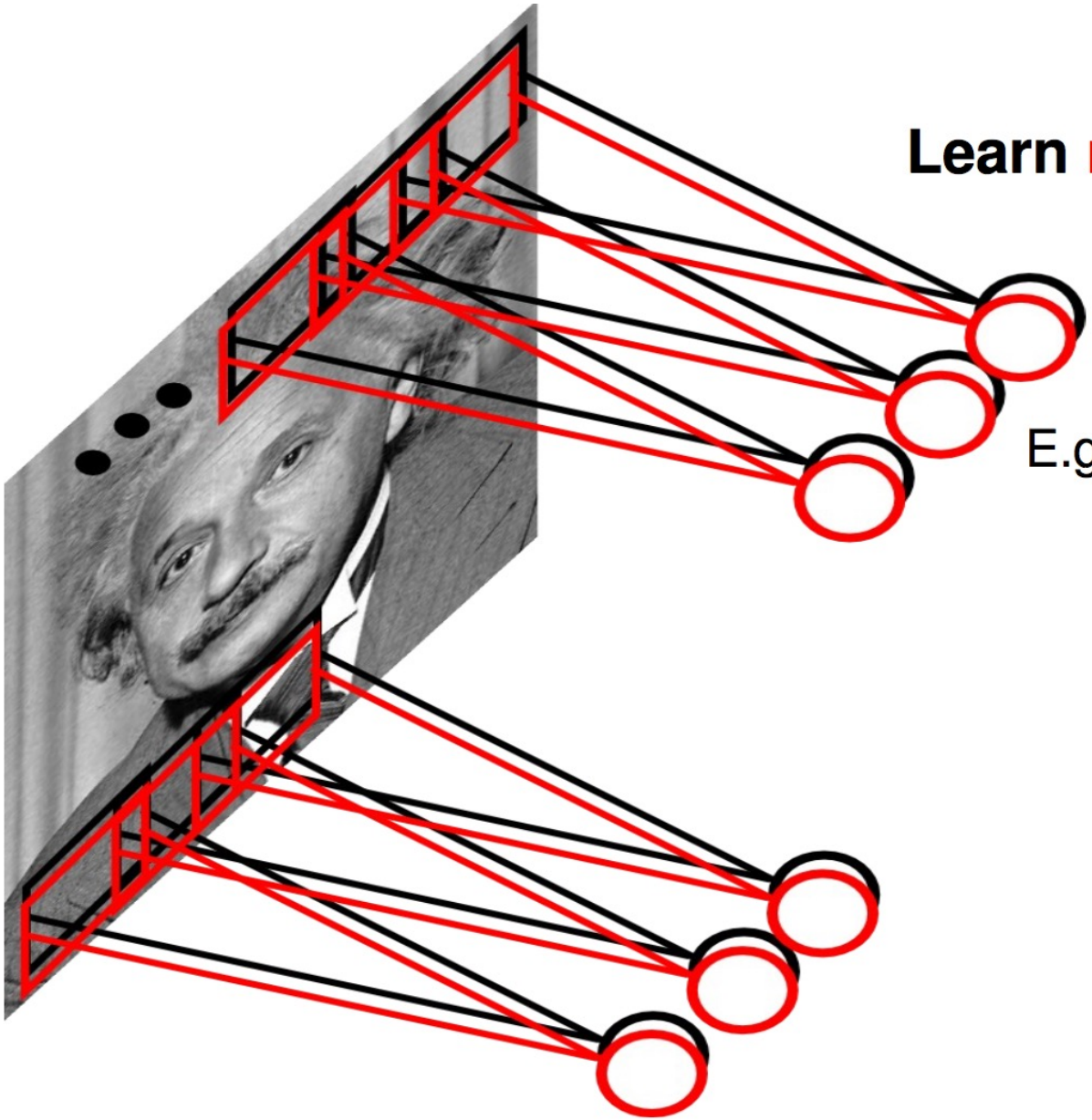


Share the same parameters across different locations (assuming input is stationary):

Convolutions with learned kernels

Ranz

Learning Multiple Features/Filters



Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters

Ranz

Quiz for Convolutional Layer

Example (1)

For example, suppose that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field is of size 5x5, how many parameters in the first Conv Layer(Assume only one filter)?

Quiz for Convolutional Layer

Example (1)

For example, suppose that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field is of size 5x5, how many parameters in the first Conv Layer(Assume only one filter)?
*Each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights. Notice that the the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.*

Example (2)

Suppose an input volume had size [16x16x20]. If the receptive field size of 3x3 and we would like to learn 10 feature maps, how many parameters in this Conv Layer?

Quiz for Convolutional Layer

Example (1)

For example, suppose that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field is of size 5x5, how many parameters in the first Conv Layer(Assume only one filter)?

*Each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights. Notice that the the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.*

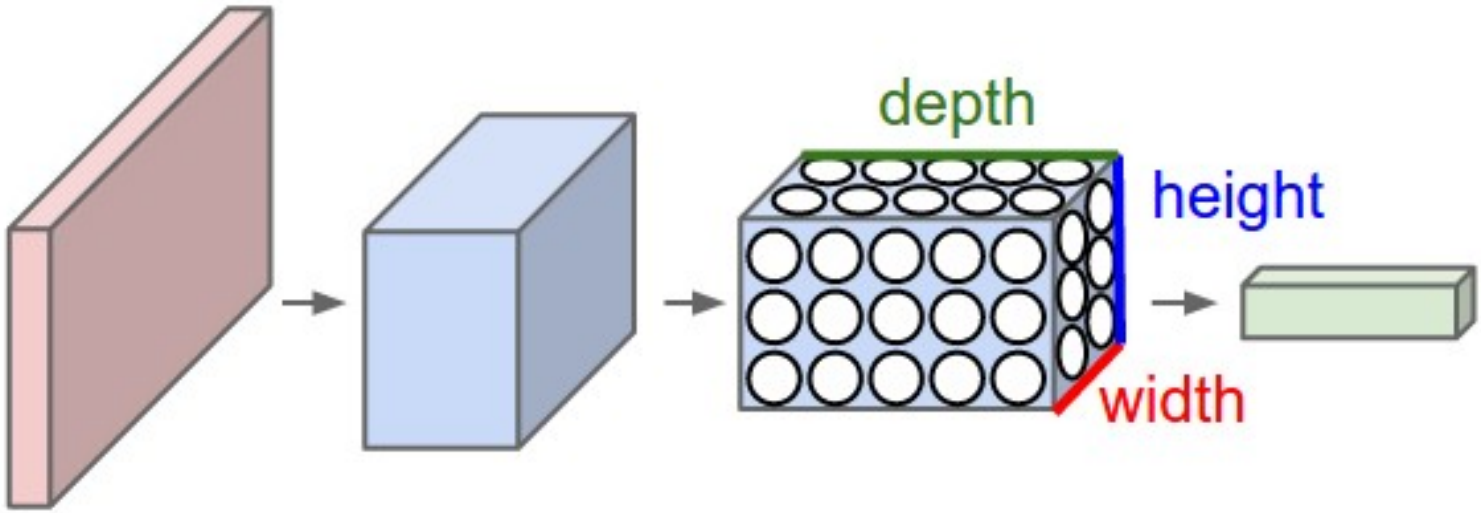
Example (2)

Suppose an input volume had size [16x16x20]. If the receptive field size of 3x3 and we would like to learn 10 feature maps, how many parameters in this Conv Layer?

*Every neuron in one filter in the Conv Layer would now have a total of 3*3*20 = 180 connections to the input volume. Since we have 10 such filters, then the total parameters would be 10*180=1800.*

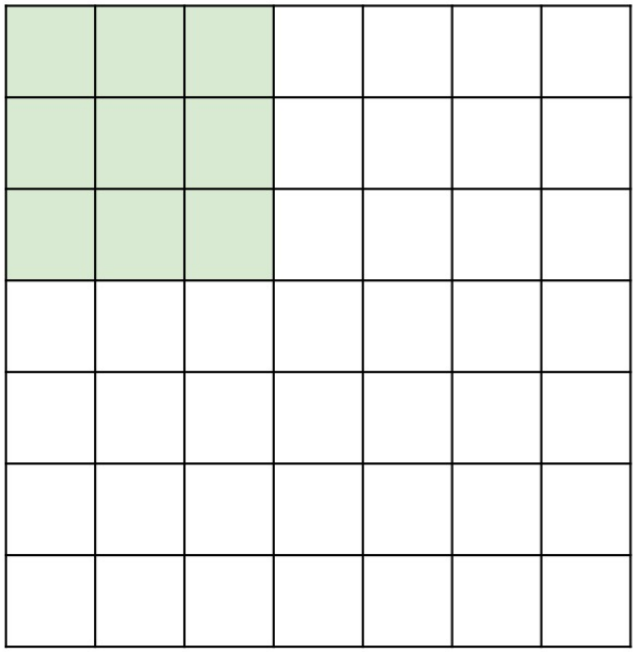
Conv Layer Parameters - Depth

First, the **depth** of the output volume is a hyperparameter that we can pick; It controls the number of neurons in the Conv layer that connect to the same region of the input volume. This is analogous to a regular Neural Network, where we had multiple neurons in a hidden layer all looking at the exact same input.



Conv Layer Parameters - Stride

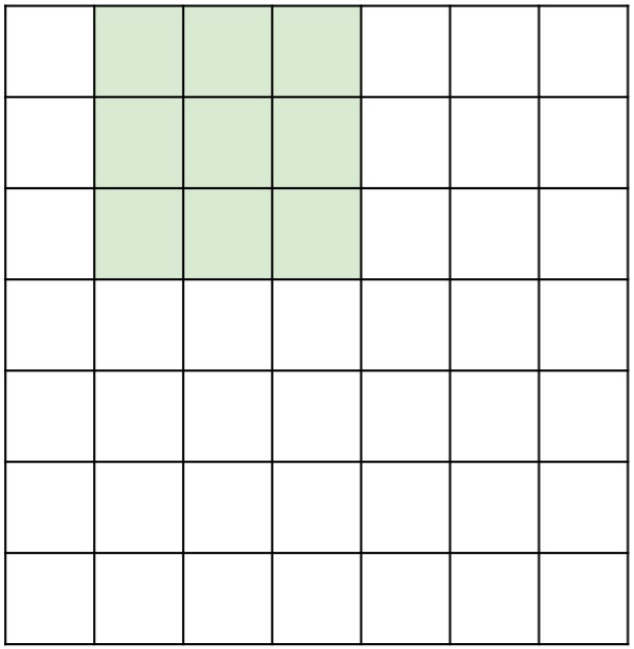
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1

Conv Layer Parameters - Stride

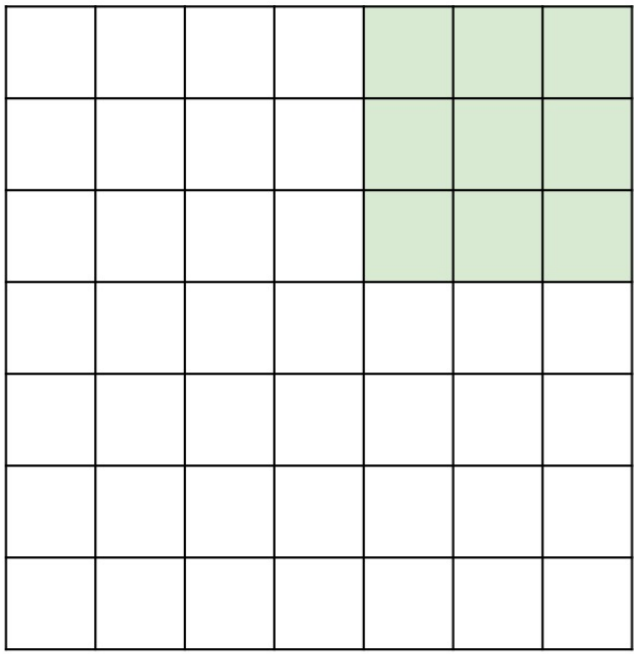
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1

Conv Layer Parameters - Stride

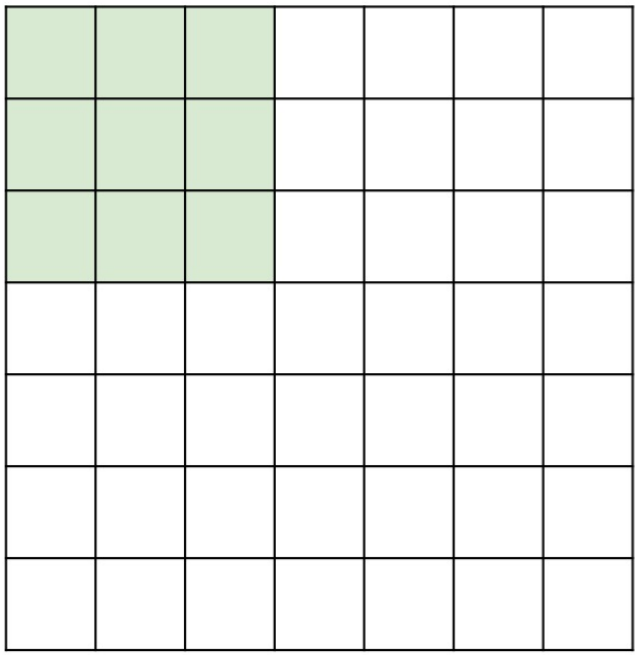
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

Conv Layer Parameters - Stride

Replicate this column of hidden neurons across space, with some **stride.**

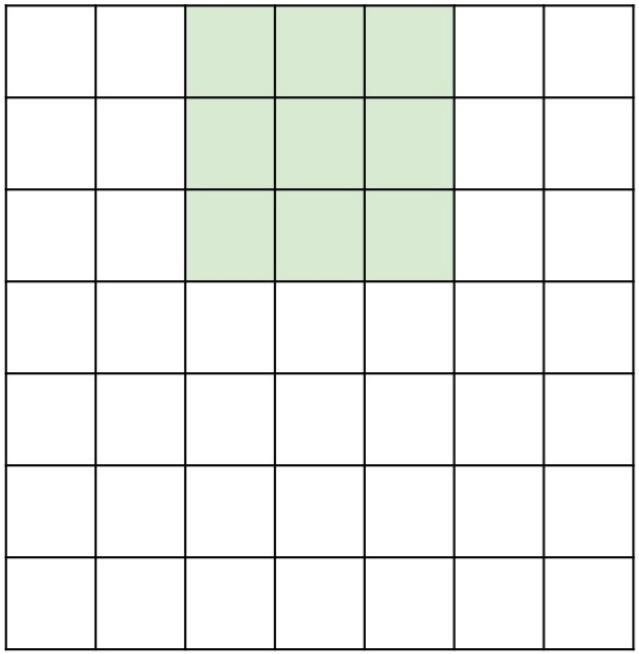


7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?

Conv Layer Parameters - Stride

Replicate this column of hidden neurons across space, with some **stride.**

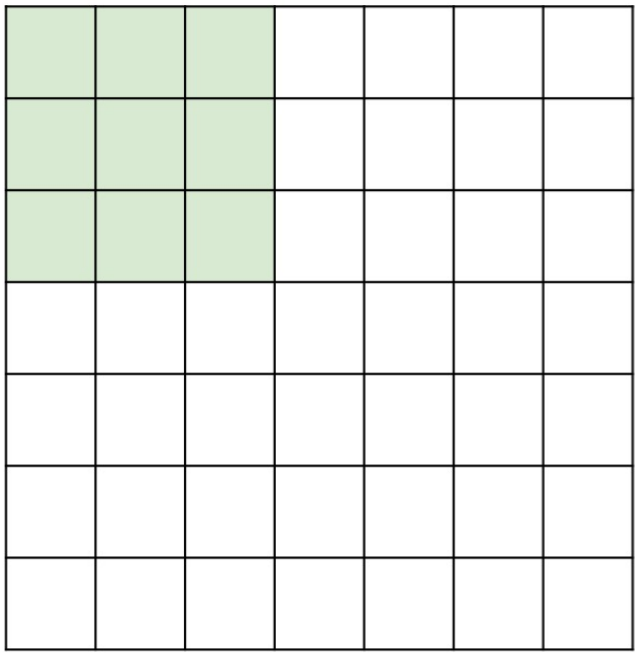


7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?

Conv Layer Parameters - Stride

Replicate this column of hidden neurons across space, with some **stride.**



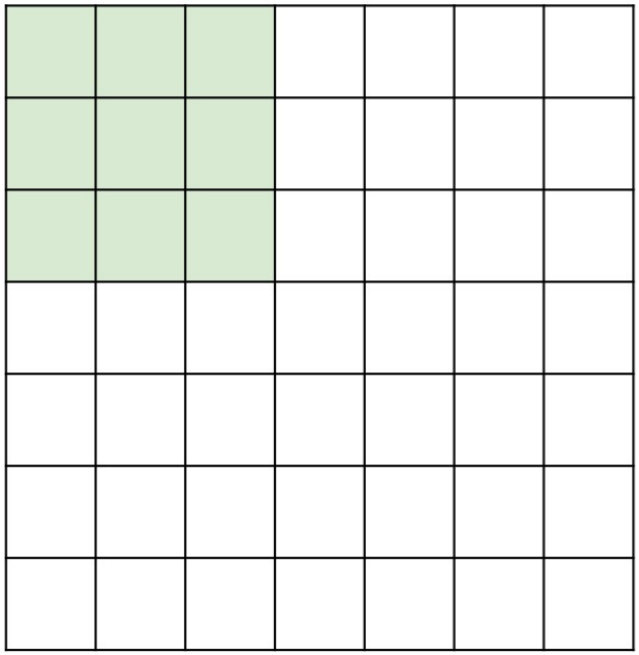
7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?
=> **3x3 output**

what about stride 3?

Conv Layer Parameters - Stride

Replicate this column of hidden neurons across space, with some **stride.**

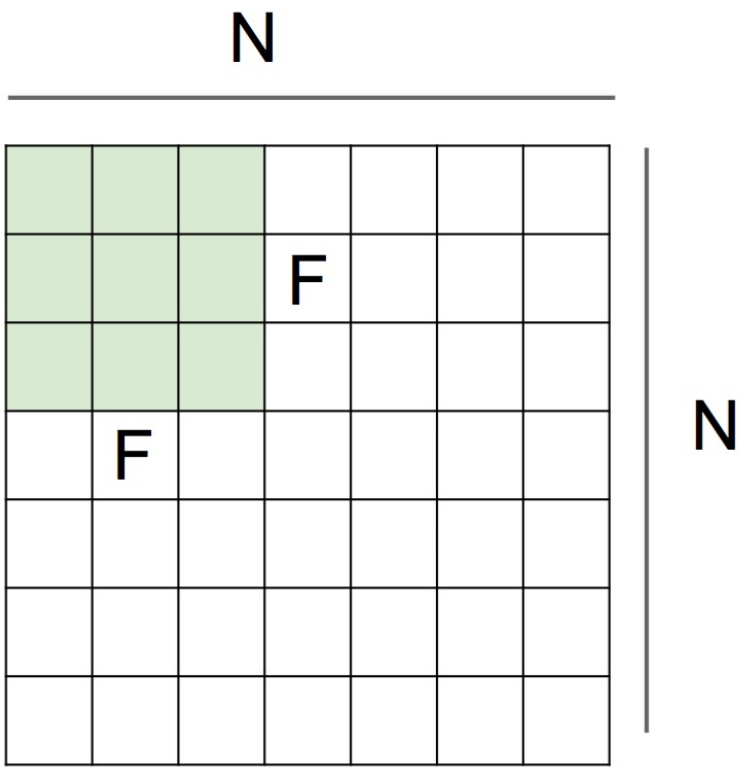


7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?
=> **3x3 output**

what about stride 3? Cannot.

Conv Layer Parameters - Stride



Output size:
(N - F) / stride + 1

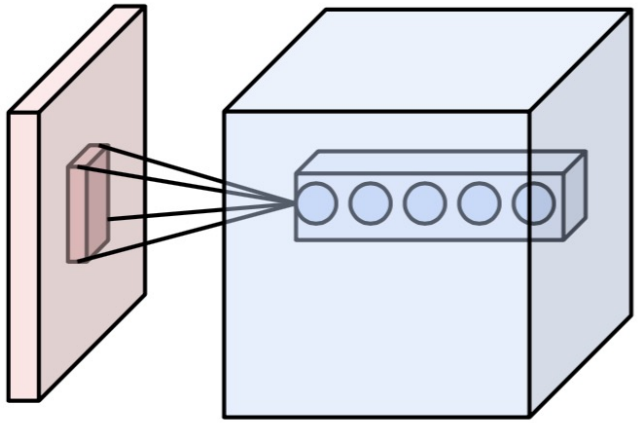
e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = … :\

Examples Time

Examples time:

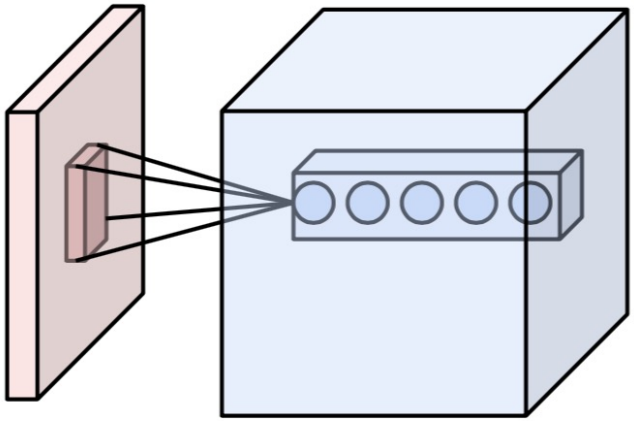
Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 1**
Number of neurons: **5**

Output volume: ?



Examples Time

Examples time:

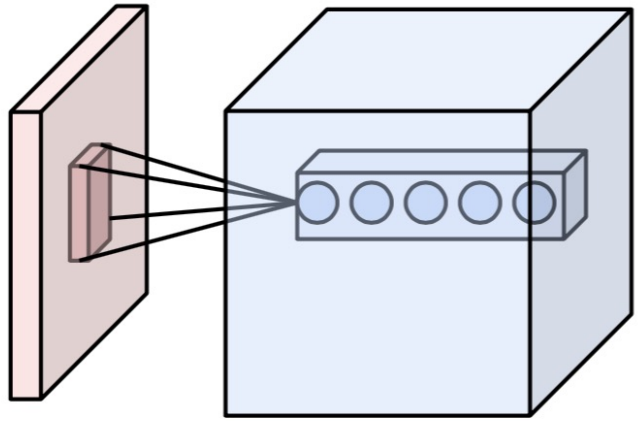


Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 1**
Number of neurons: **5**

Output volume: (32 - 5) / 1 + 1 = 28, so: **28x28x5**
How many weights for each of the 28x28x5 neurons?

Examples Time

Examples time:

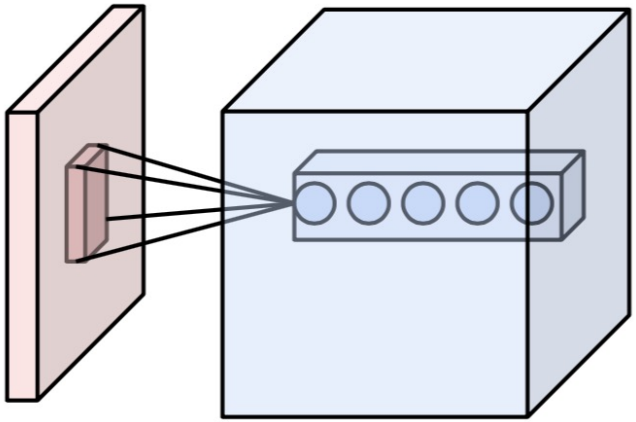


Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 1**
Number of neurons: **5**

Output volume: (32 - 5) / 1 + 1 = 28, so: **28x28x5**
How many weights for each of the 28x28x5 neurons? **5x5x3 = 75**

Examples Time

Examples time:

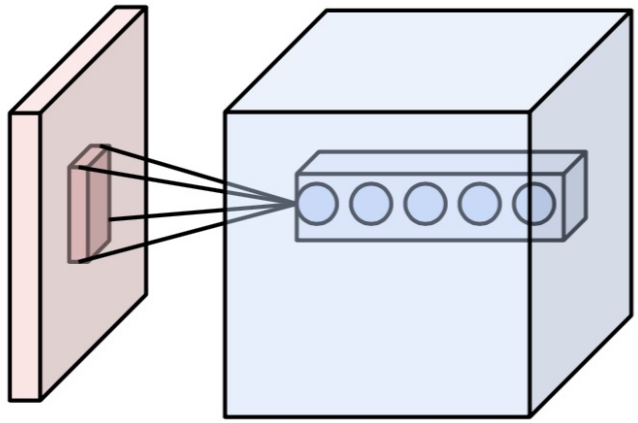


Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 2**
Number of neurons: **5**

Output volume: ?

Examples Time

Examples time:

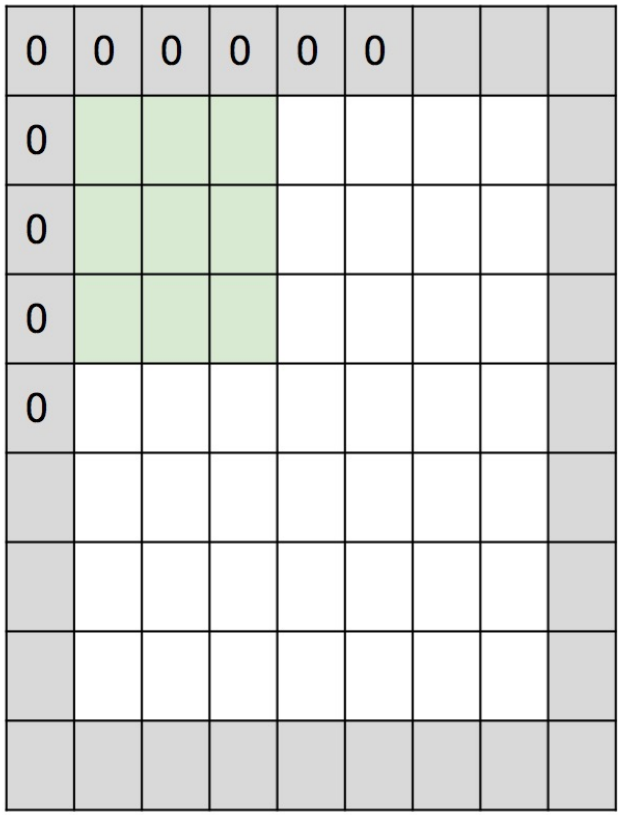


Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 2**
Number of neurons: **5**

Output volume: ? Cannot: (32-5)/2 + 1 = 14.5 :\

Conv Layer Parameters - Padding

In practice: Common to zero pad the border

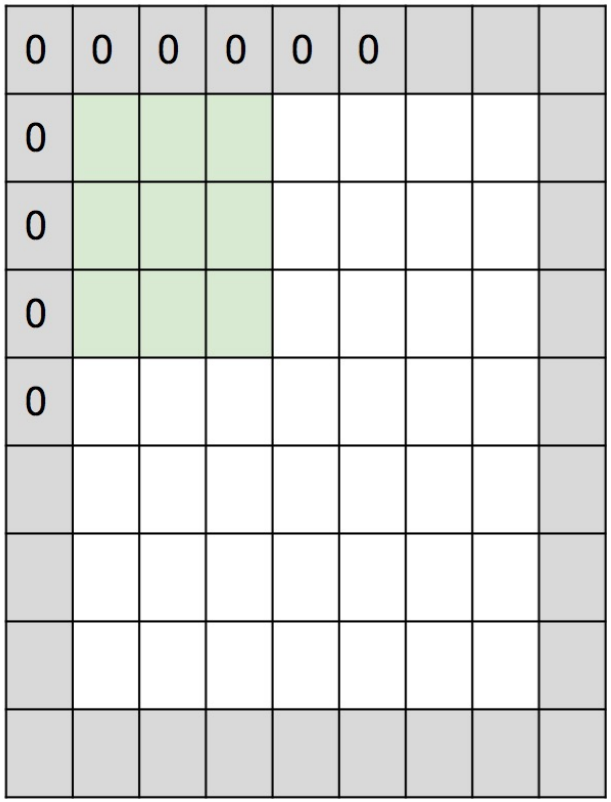


(in each channel)

e.g. input 7x7
neuron with receptive field 3x3, stride 1
pad with 1 pixel border => what is the output?

Conv Layer Parameters - Padding

In practice: Common to zero pad the border



(in each channel)

e.g. input 7x7
neuron with receptive field 3x3, stride 1
pad with 1 pixel border => what is the output?

7x7 => preserved size!

in general, common to see stride 1, size F, and zero-padding with (F-1)/2.
(Will preserve input size spatially)

Summary of Stride Constraints

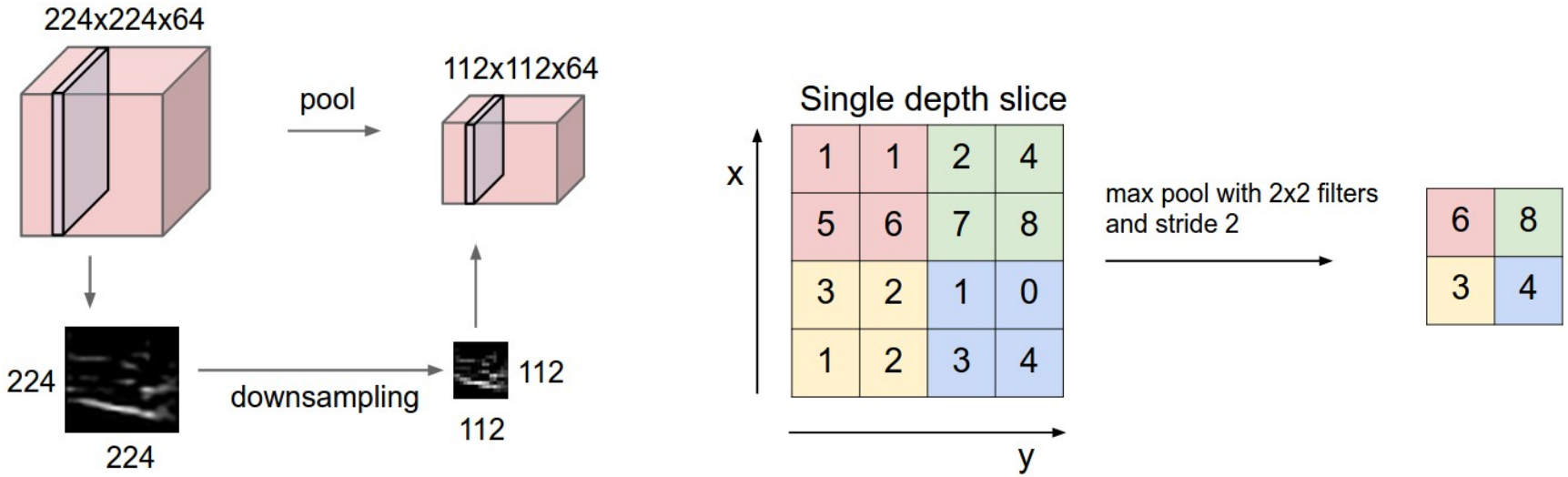
We can compute the spatial size of the output volume as a function of the input volume size (**W**), the receptive field size of the Conv Layer neurons (**F**), the stride with which they are applied (**S**), and the amount of zero padding used (**P**) on the border. You can convince yourself that the correct formula for calculating how many neurons "fit" is given by $(\mathbf{W} - \mathbf{F} + \mathbf{2P})/\mathbf{S} + \mathbf{1}$. If this number is not an integer, then the strides are set incorrectly and the neurons cannot be tiled so that they "fit" across the input volume neatly, in a symmetric way.

Example (ImageNet 2012)

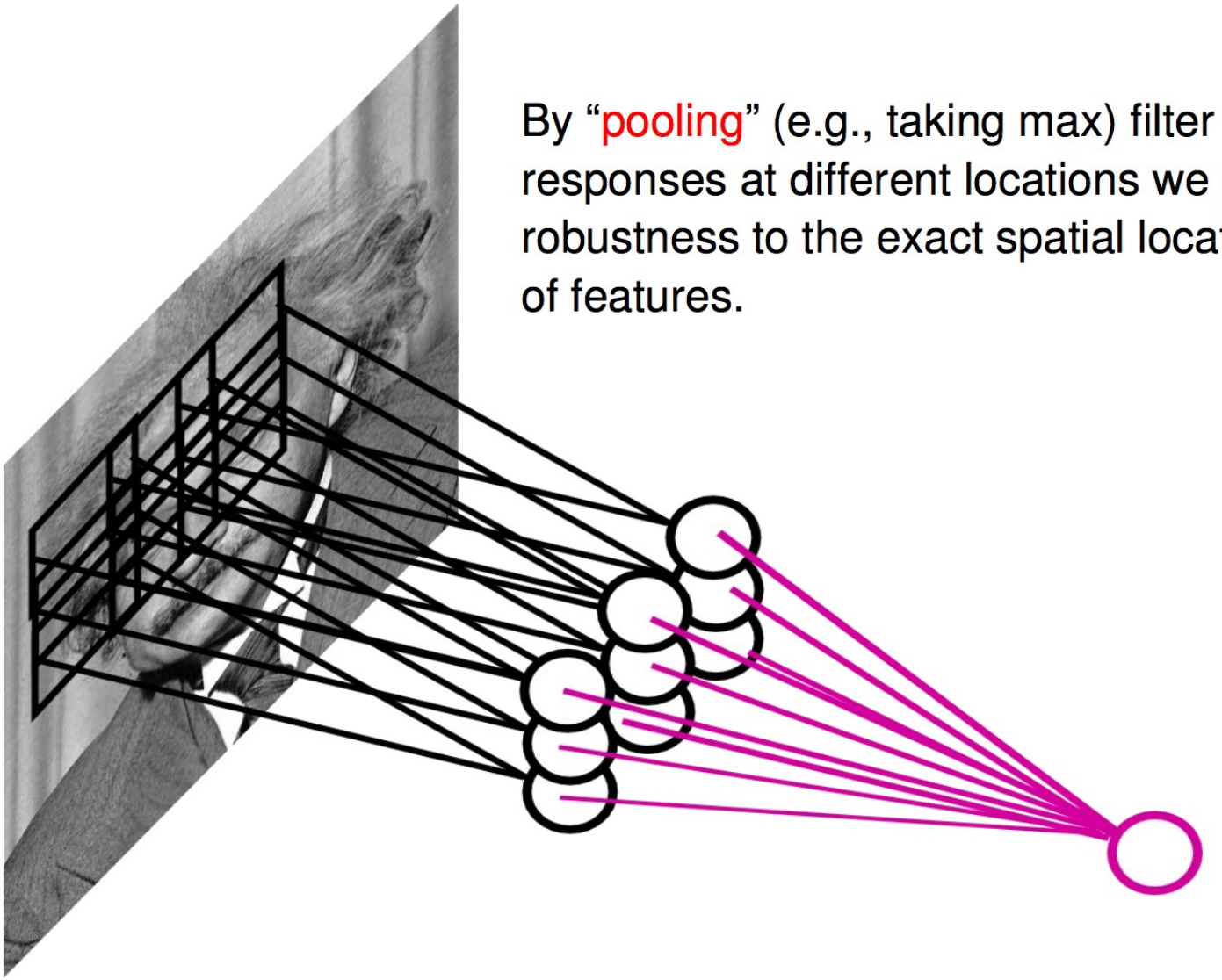
The Krizhevsky et al. architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size $\mathbf{F} = \mathbf{11}$, stride $\mathbf{S} = \mathbf{4}$ and no zero padding $\mathbf{P} = \mathbf{0}$. Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of $\mathbf{K} = \mathbf{96}$, the Conv layer output volume had size [55x55x96]. Each of the 55*55*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but of course with different weights.

Pooling Operation

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation.



MaxPooling Layer



By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

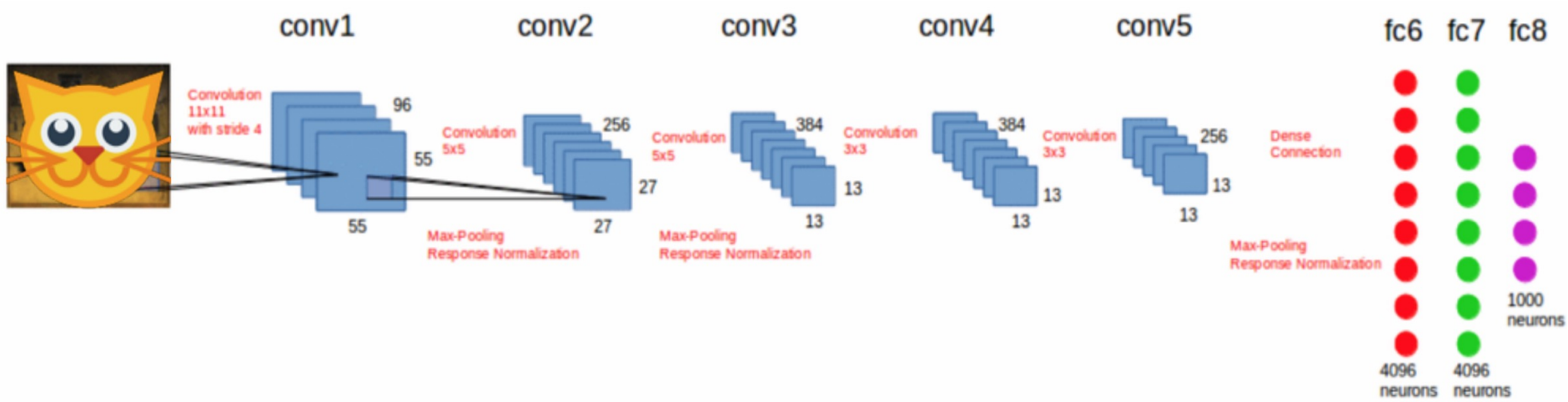
Ran

Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size.

ConvNet Architecture

$INPUT \rightarrow [[CONV \rightarrow RELU] * N \rightarrow POOL?] * M \rightarrow [FC \rightarrow RELU] * K \rightarrow FC$
where the $*$ indicates repetition, and the $POOL?$ indicates an optional pooling layer.



Modern ConvNets

- **AlexNet**. Winner of ImageNet 2012. It significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error).
- **ZF-Net**. Winner of ImageNet 2013. It was an improvement on AlexNet by tweaking the architecture hyperparameters.
- **GoogLeNet**. Winner of ImageNet 2014. Its main contribution was the development of an **Inception Module** that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M).
- **VGGNet**. Runner-up of ImageNet 2014. Its main contribution was in showing that the depth of the network is a critical component for good performance. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M).
- **ResNet**. Runner-up of ImageNet 2015. It features special skip connections and a heavy use of batch normalization.
- **Inception-V4**. Inception+ResNet.

Analysis of VGGNet

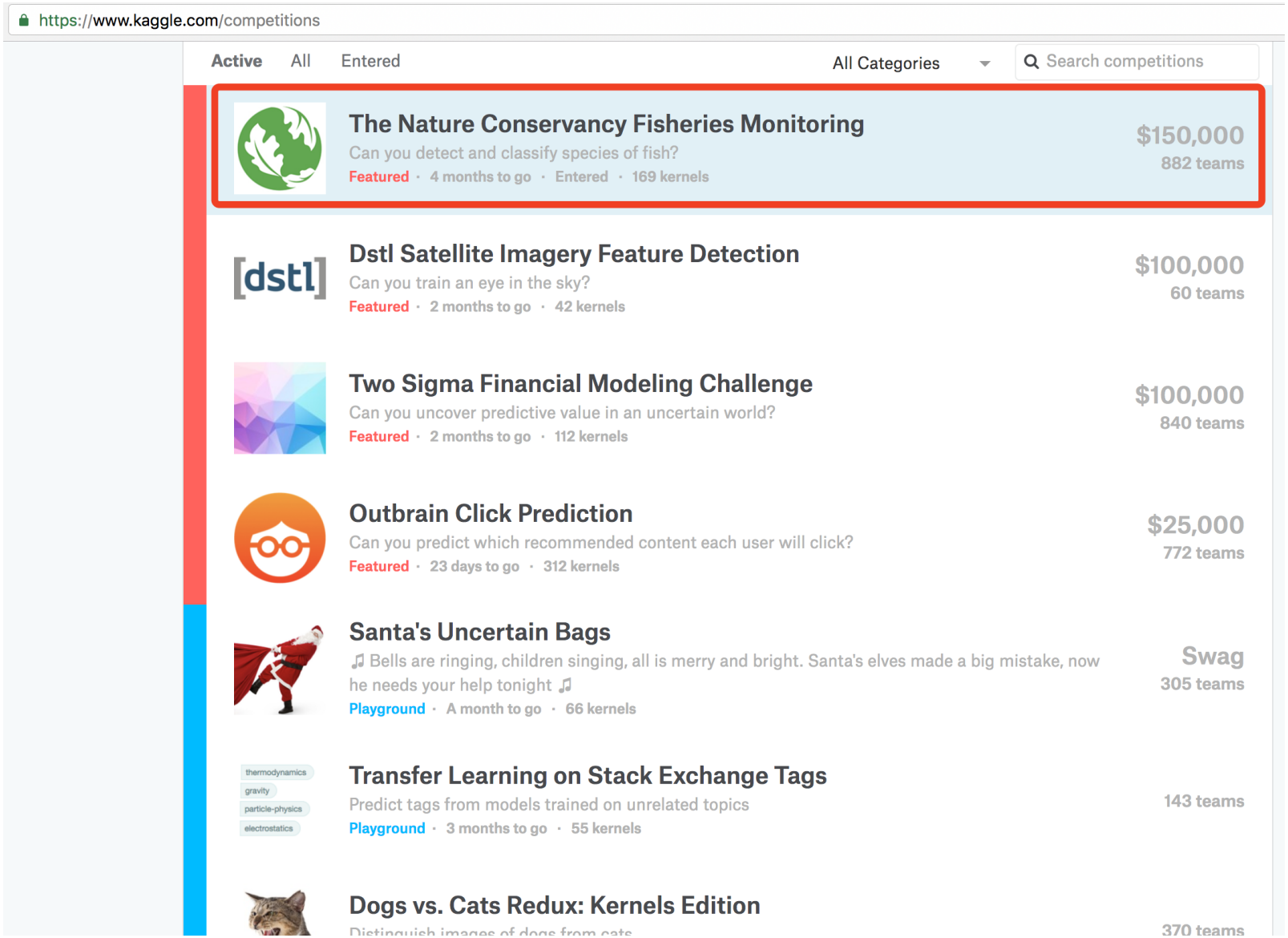
```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
```

Homework 1: MNIST Classification with ConvNets

In the homework of lecture 1, you have already trained a MLP/DNN to classify MNIST data. In this homework, you are required to train a deep ConvNet based on TensorFlow/Keras. Keywords: one-hot encoding/vectors, cross-entropy, convolutional layers/kernels, SGD, data augmentation.

```
# For a single-input model with 10 classes (categorical classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

Kaggle: NCFM – Marine Fish Classification



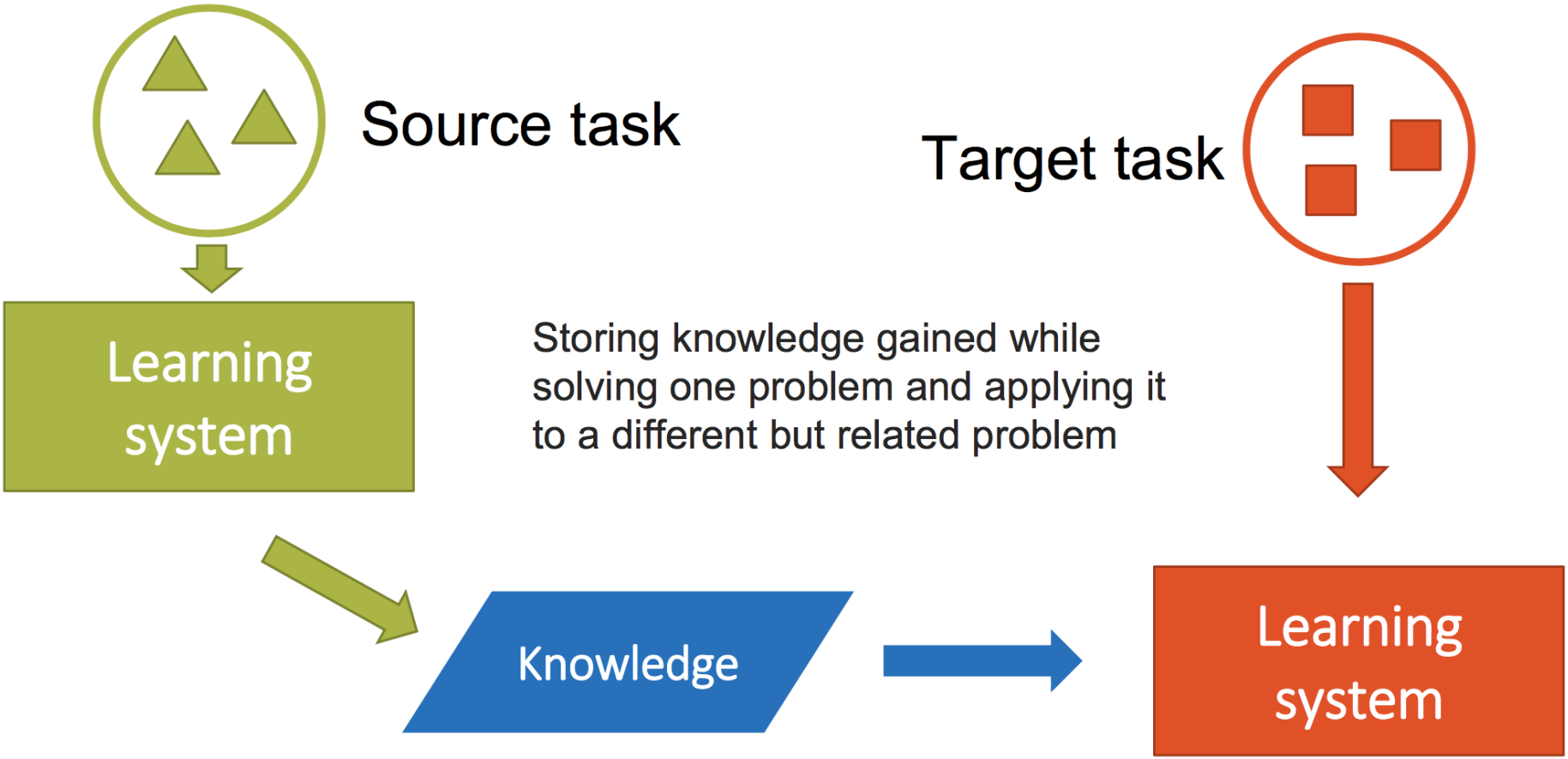
Problem Definition

- A typical **single-label image classification** problem covering 8 classes (7 for fish and 1 for No-fish).
- Training set is rather **small**, only 3777 images. Extra 1000 for testing.
- Images are challenging since noise/background dominates in the whole picture.

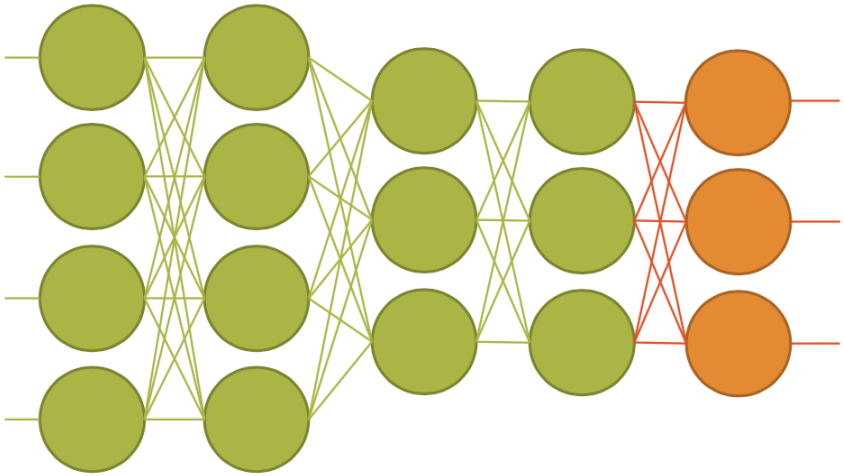
Problem Analysis

- Image classification $\Rightarrow$ ConvNets
- Small size of training set $\Rightarrow$ Risk of **overfitting** $\Rightarrow$ Transfer Learning / **Fine-tune** the weights of pretrained networks
  - **Pretrained** models trained across multiple GPUs on ImageNet.
  - ConvNet features are more **generic** in early layers and more **original-dataset-specific** in later layers.
  - Use a **smaller** learning rate to fine-tune.
  - Usually fine-tuning begins with later layers.
- Challenging images $\Rightarrow$ STOA ConvNets and Object Detection

Tansfer Learning

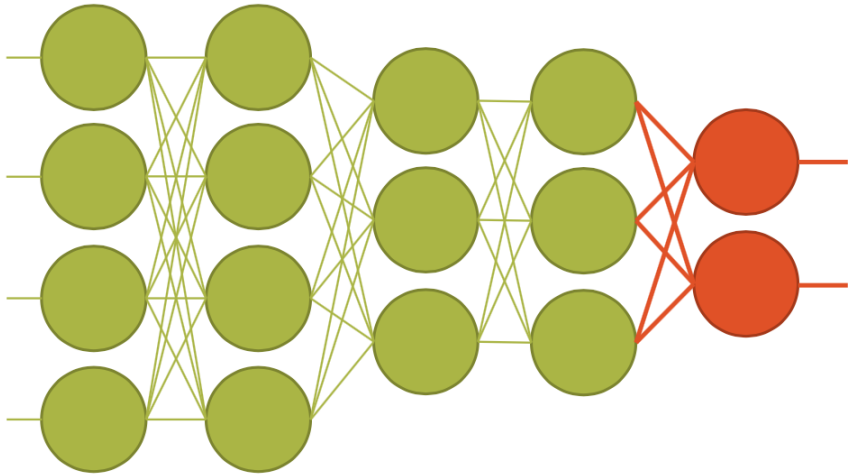


Tansfer Learning for Neural Networks



1) Train on Source Task

2) Remove last layers



3) Add new last layers

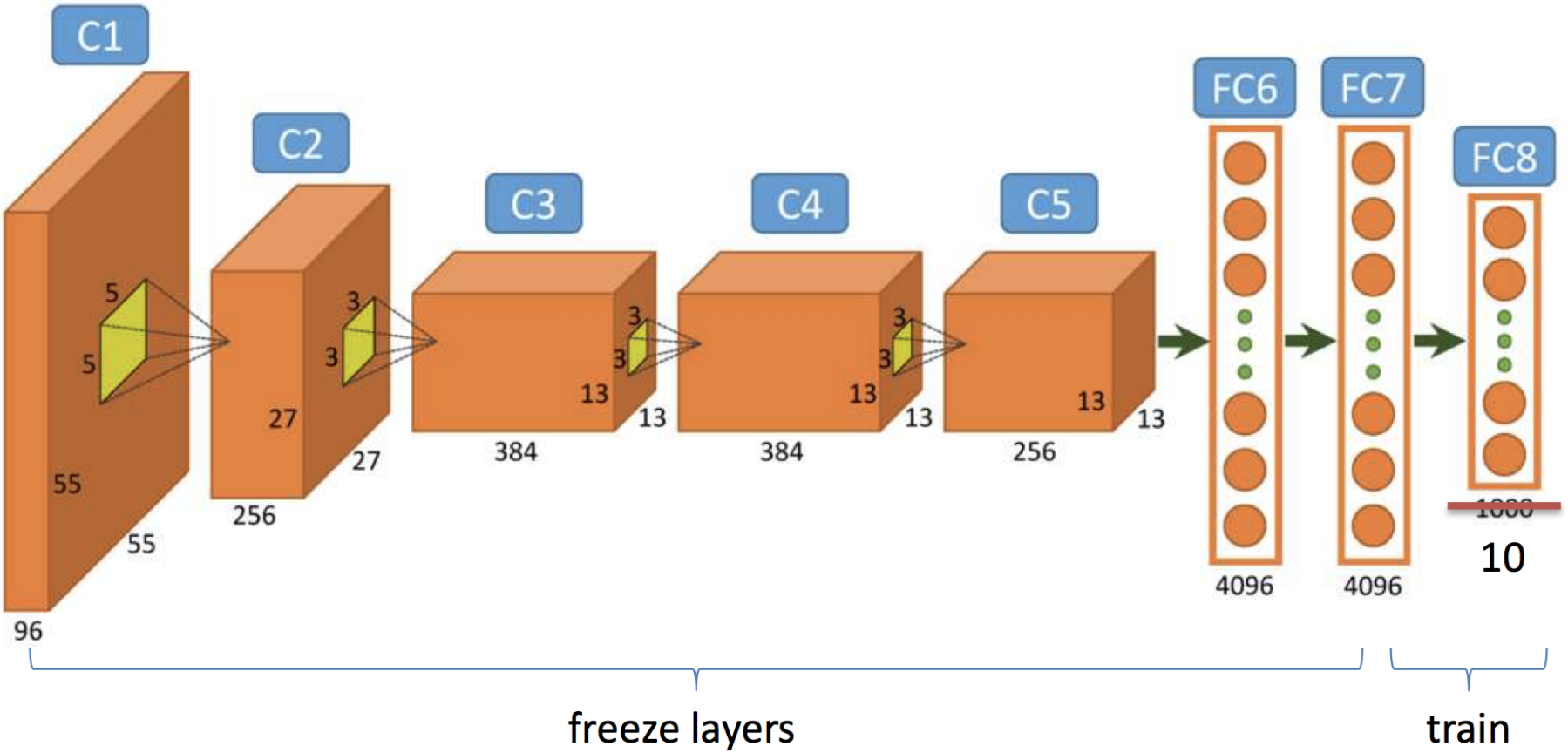
4) Train net for Target Task

When to Use Transer Learning

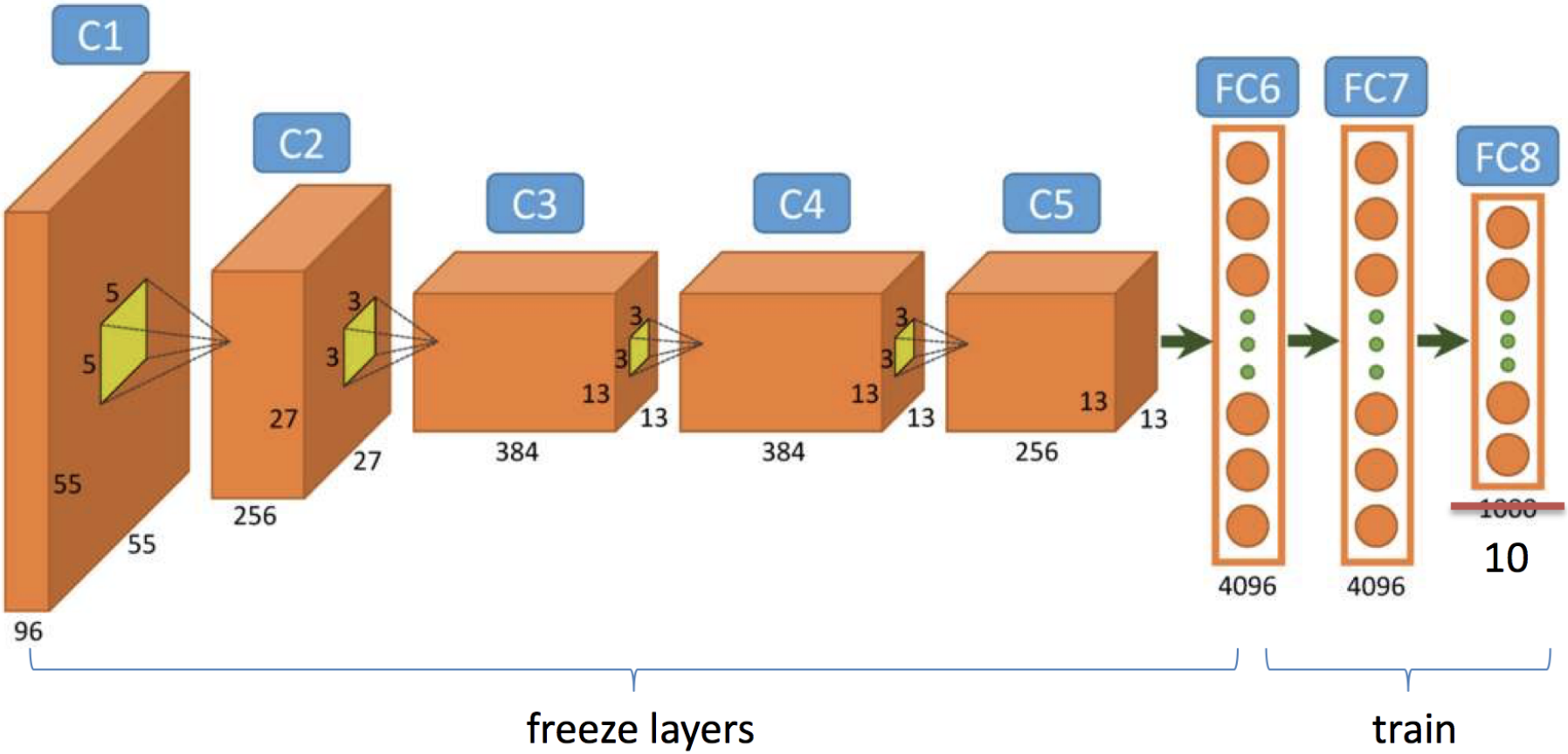
Low amount of data

Source and Target tasks are similar

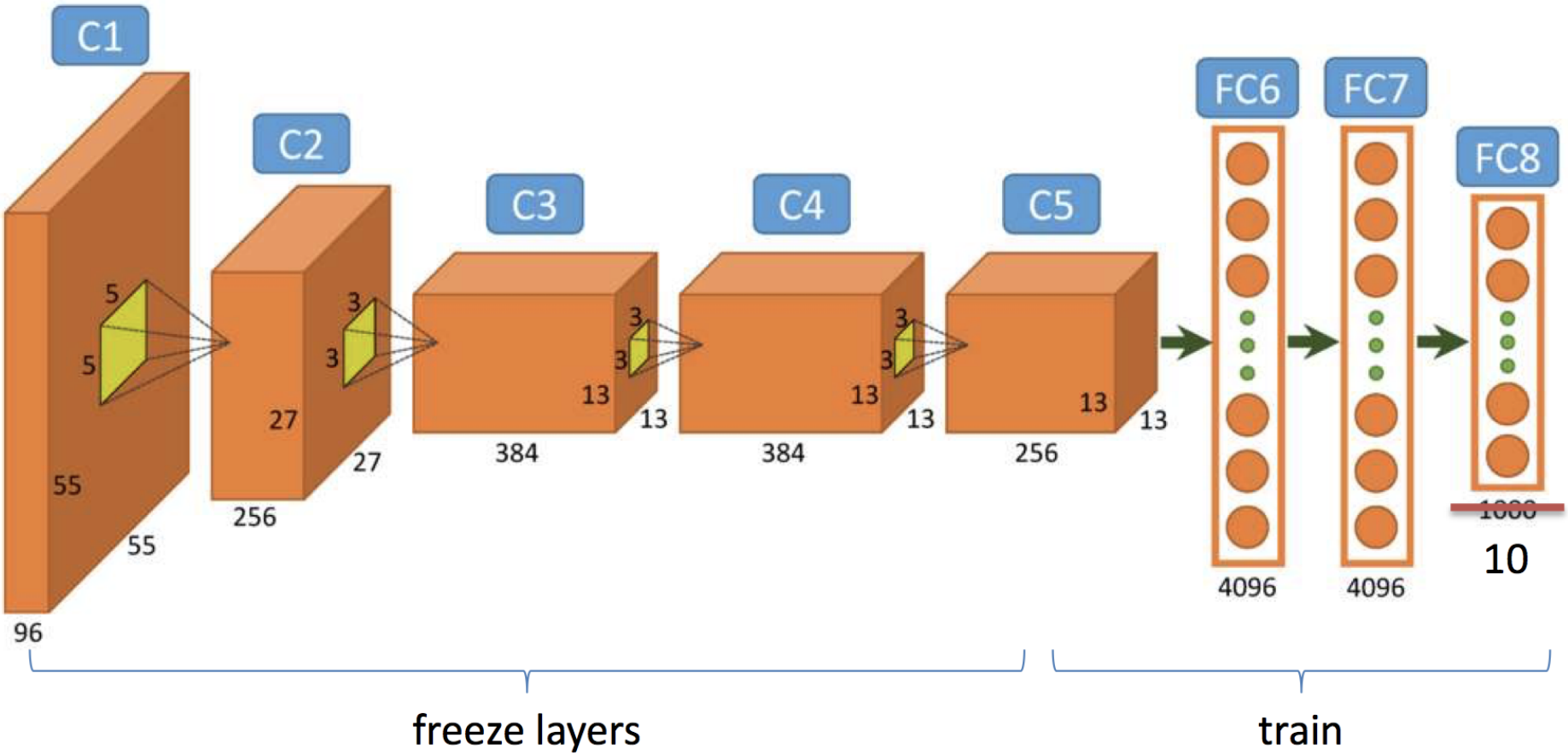
Fine-tuning Pretrained Network



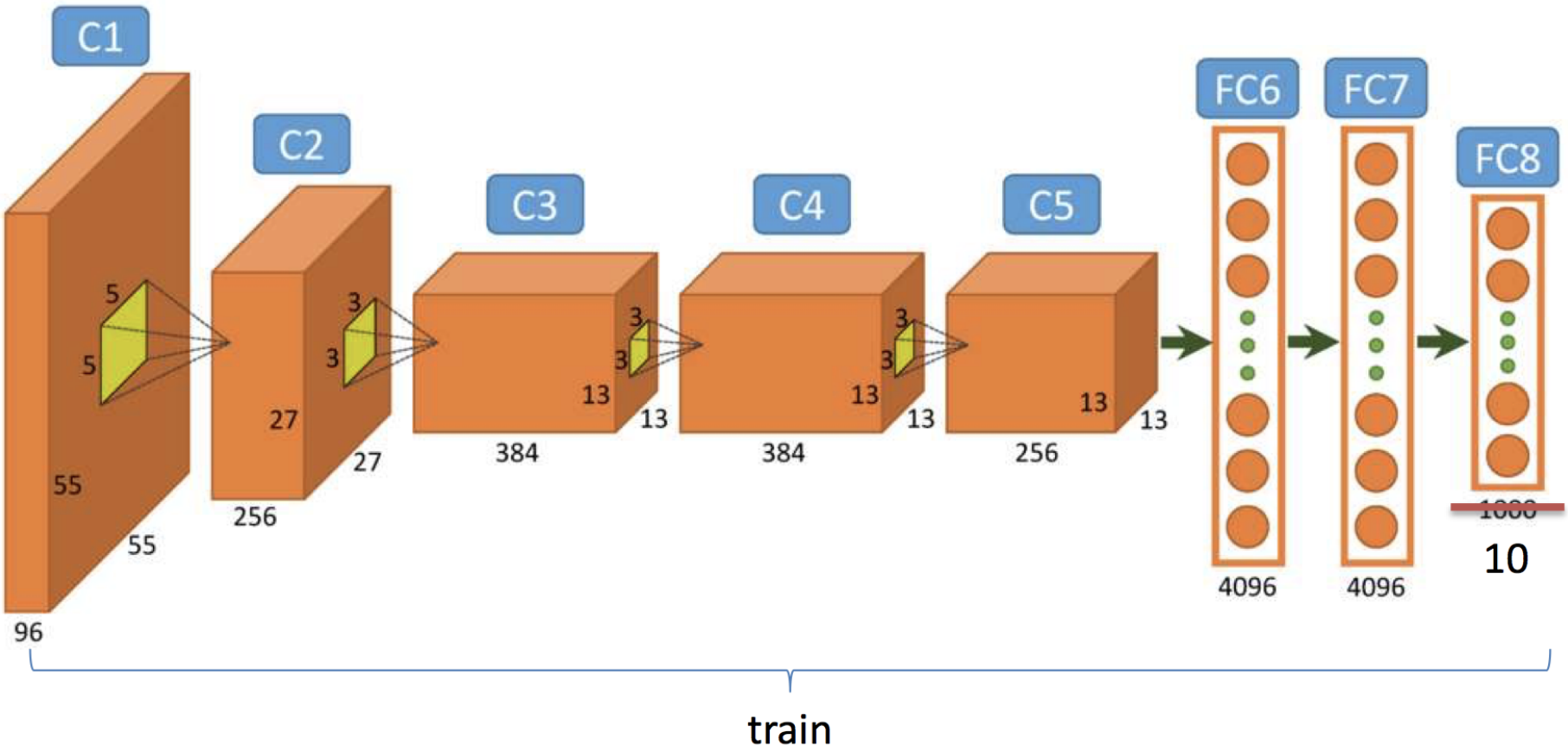
Fine-tuning Pretrained Network



Fine-tuning Pretrained Network

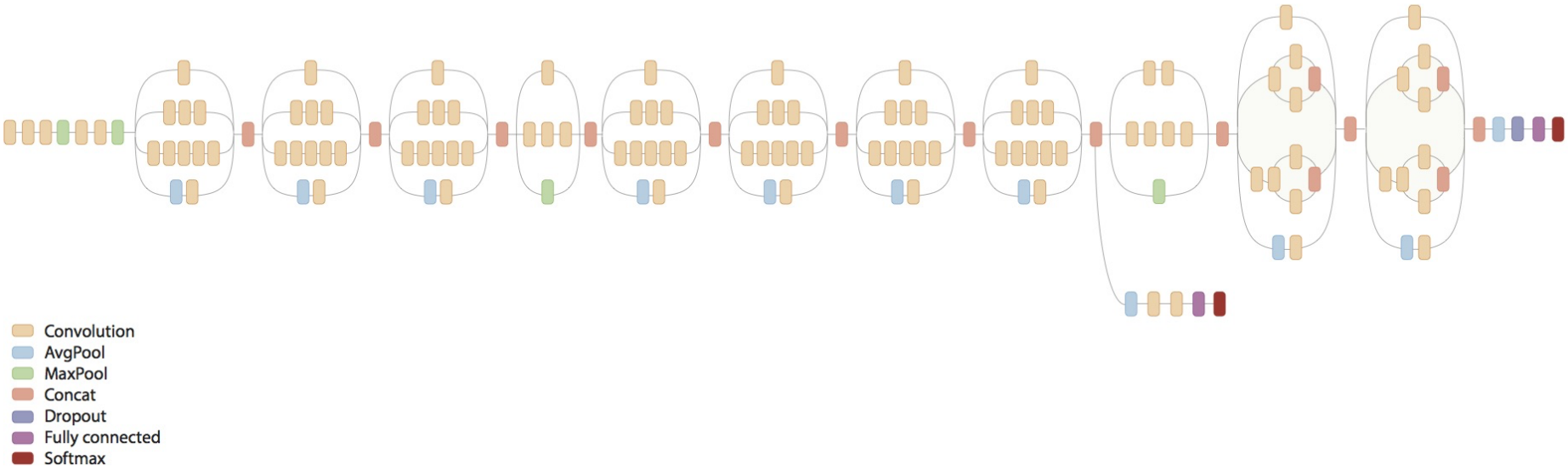


Fine-tuning Pretrained Network



Pretrained Model: Inception-V3

- 21.2% top-1 and 5.6% top-5 error for single frame evaluation.
- Less than 25M parameters.
- Factorization into smaller convolutions.
- Spatial Factorization into Asymmetric Convolutions.
- Pretrained weights off-the-shelf in most of deep learning frameworks.



Code Snippet 1: Import *

```
from inception_v3 import InceptionV3, preprocess_input
import os
from keras.layers import Flatten, Dense, AveragePooling2D
from keras.models import Model
from keras.optimizers import RMSprop, SGD
from keras.callbacks import ModelCheckpoint
from keras.preprocessing.image import ImageDataGenerator

learning_rate = 0.0001
img_width = 299
img_height = 299
nbr_train_samples = 3019
nbr_validation_samples = 758
nbr_epochs = 50
batch_size = 32
```

Code Snippet 2: Build Up Network

```
print('Loading InceptionV3 Weights ...')
InceptionV3_notop = InceptionV3(include_top=False, weights='imagenet',
                    input_tensor=None, input_shape=(299, 299, 3))
# Note that the preprocessing of InceptionV3 is:
# (x / 255 - 0.5) x 2

print('Adding Average Pooling Layer and Softmax Output Layer ...')
output = InceptionV3_notop.get_layer(index = -1).output  # Shape: (8, 8, 2048)
output = AveragePooling2D((8, 8), strides=(8, 8), name='avg_pool')(output)
output = Flatten(name='flatten')(output)
output = Dense(8, activation='softmax', name='predictions')(output)

InceptionV3_model = Model(InceptionV3_notop.input, output)
#InceptionV3_model.summary()

optimizer = SGD(lr = learning_rate, momentum = 0.9, decay = 0.0, nesterov = True)
InceptionV3_model.compile(loss='categorical_crossentropy', optimizer = optimizer, metrics = ['accuracy'])

# autosave best Model
best_model_file = "./weights.h5"
best_model = ModelCheckpoint(best_model_file, monitor='val_acc', verbose = 1, save_best_only = True)
```

Code Snippet 3: Load training data with augmentation

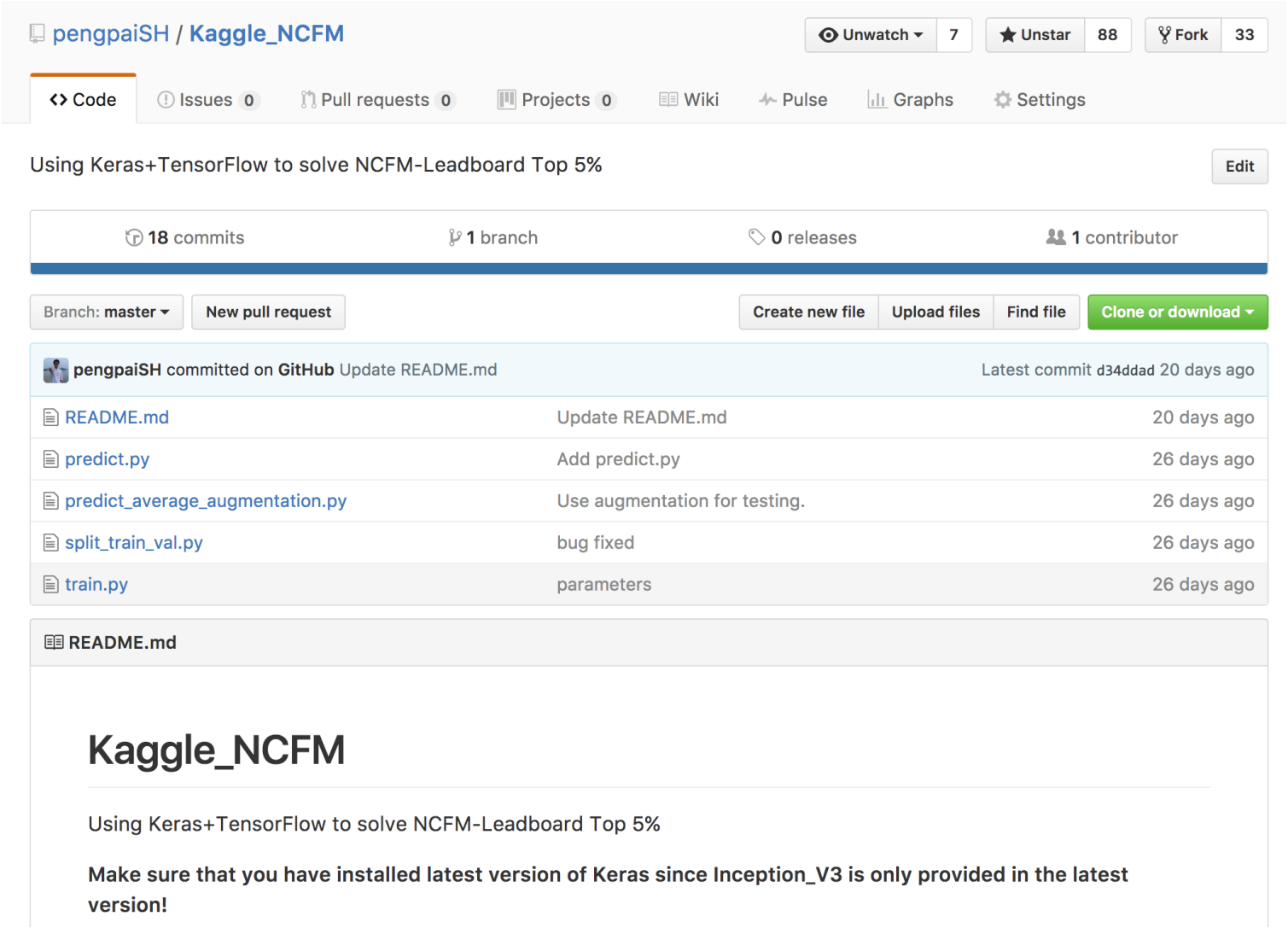
```
# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.1,
        zoom_range=0.1,
        rotation_range=10.,
        width_shift_range=0.1,
        height_shift_range=0.1,
        horizontal_flip=True)

# this is the augmentation configuration we will use for validation:
# only rescaling
val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size = (img_width, img_height),
        batch_size = batch_size,
        shuffle = True,
        classes = FishNames,
        class_mode = 'categorical')

validation_generator = val_datagen.flow_from_directory(
        val_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        shuffle = True,
        classes = FishNames,
        class_mode = 'categorical')
```

Code Snippet 4: Network Training

```
InceptionV3_model.fit_generator(
        train_generator,
        samples_per_epoch = nbr_train_samples,
        nb_epoch = nbr_epochs,
        validation_data = validation_generator,
        nb_val_samples = nbr_validation_samples,
        callbacks = [best_model])
```

Open Source in Github

https://github.com/pengpaiSH/Kaggle_NCFM



Submit the Result

Public Score: 1.3, Ranking: 50%

Trick 1: Average Multiple Testing Results with a Single Model

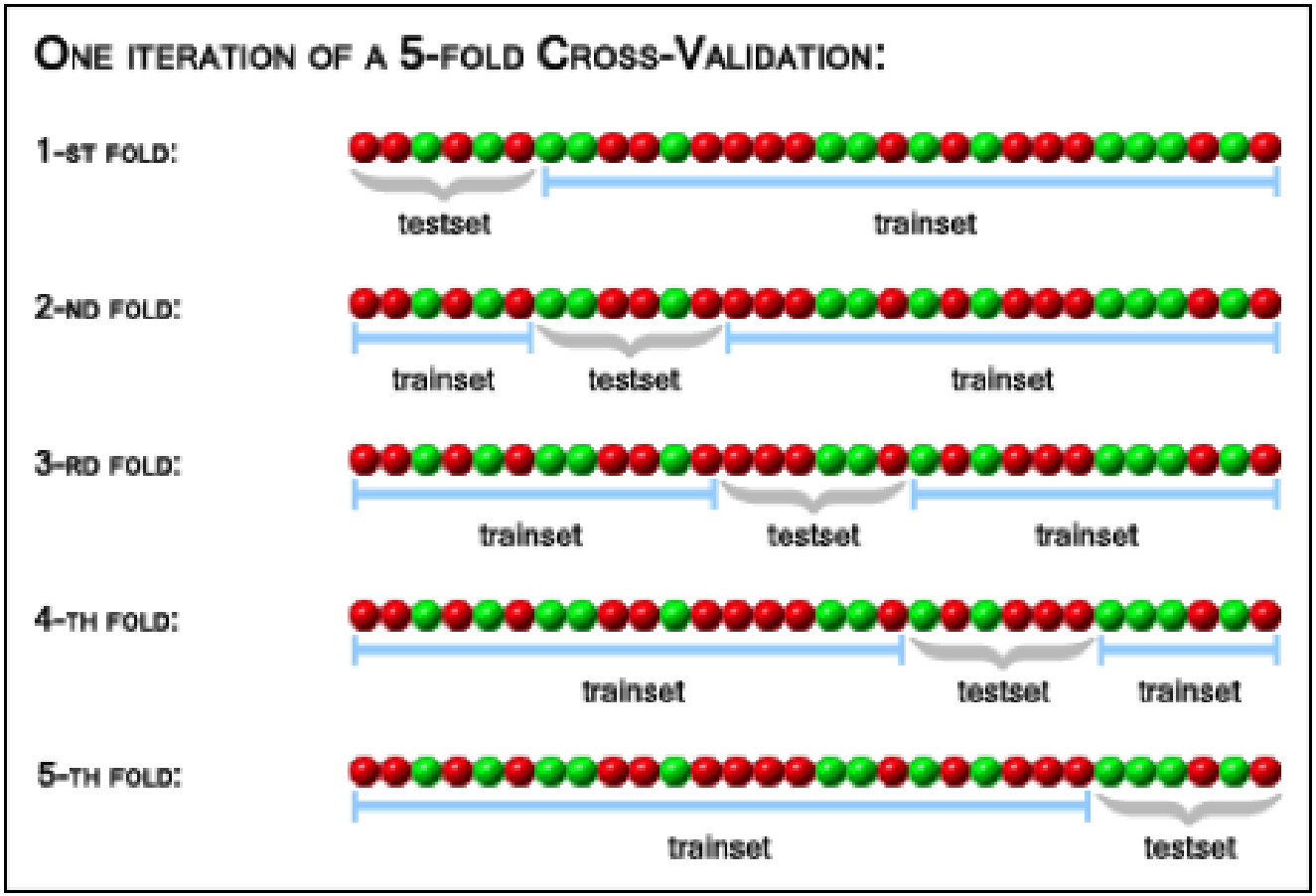
```
print('Loading model and weights from training process ...')
InceptionV3_model = load_model(weights_path)

for idx in range(nbr_augmentation):
    print('{}th augmentation for testing ...'.format(idx))
    random_seed = np.random.random_integers(0, 100000)

    test_generator = test_datagen.flow_from_directory(
            test_data_dir,
            target_size=(img_width, img_height),
            batch_size=batch_size,
            shuffle = False, # Important !!!
            seed = random_seed,
            classes = None,
            class_mode = None)
```

Result of Trick 1

21	↓3	FishingBR	1.03750	18	Mon, 28 Nov 2016 08:26:41 (-5.3d)
22	—	Bojan Tunguz	1.03861	30	Tue, 29 Nov 2016 15:49:45
23	↓4	YetiMan	1.04922	30	Thu, 24 Nov 2016 04:00:00 (-2d)
24	↓4	JeffMomo	1.05807	8	Thu, 17 Nov 2016 23:05:01 (-21.7h)
25	↓4	Patrick Chan	1.05952	13	Tue, 22 Nov 2016 17:11:20 (-12.4h)
26	↓3	北京的冬天 飘着白雪	1.06705	25	Tue, 22 Nov 2016 03:44:19 (-3.2d)
27	↓3	Hugo Pinto | Humberto Brandão	1.06867	37	Tue, 29 Nov 2016 11:04:33 (-3.4d)
28	↑213	EhabAlbadawy	1.07230	14	Mon, 28 Nov 2016 01:09:21 (-0.3h)
29	↓4	clustifier	1.07630	18	Thu, 24 Nov 2016 21:19:38 (-0.7h)
30	↓4	(><)	1.08436	18	Fri, 25 Nov 2016 11:11:56 (-7d)
31	↓4	loweew	1.08548	7	Sun, 20 Nov 2016 00:35:48
32	new	Emanuel Sena	1.08673	13	Tue, 29 Nov 2016 03:02:06
33	↓5	Igor Krashenyi	1.08724	30	Mon, 28 Nov 2016 05:41:28 (-4.7d)
34	↓5	Luis Andre Dutra e Silva	1.08932	39	Wed, 23 Nov 2016 20:38:00 (-44.6h)
35	↑8	gstieger	1.09093	44	Tue, 29 Nov 2016 14:16:36 (-37.8h)
36	new	**PengPai**	**1.09217**	**5**	**Tue, 29 Nov 2016 15:40:13 (-7.5h)**
37	↓7	apgeorg	1.09376	42	Fri, 25 Nov 2016 10:15:50 (-23.7h)
38	↓7	Roman Ring	1.09618	13	Sun, 20 Nov 2016 14:12:10 (-24.8h)
39	↓7	Veovis	1.09892	19	Wed, 23 Nov 2016 09:42:37 (-3.9d)
40	↓7	Marcelo Bastos	1.09902	18	Thu, 24 Nov 2016 07:04:34

3d improvement: none

Trick 2: Ensemble Models by Cross-Validation



Result of Trick 2

12	—	Andrey Ostapets	0.90055	10	Sun, 27 Nov 2016 20:06:37
13	↓2	Zshiney	0.91655	10	Wed, 16 Nov 2016 22:18:31 (-21.6h)
14	new	njm	0.93451	15	Thu, 01 Dec 2016 15:09:45 (-1.5h)
15	new	XuleiYang	0.93669	7	Thu, 01 Dec 2016 03:55:51 (-1.2h)
16	↓3	wlamond	0.94289	17	Mon, 21 Nov 2016 12:51:27 (-4.3d)
17	↓3	Evgeny Nizhibitsky	0.94963	29	Wed, 23 Nov 2016 09:47:04
18	↑36	Jeremy Howard	0.97162	22	Thu, 01 Dec 2016 04:18:17
19	↑227	Nam	0.97587	22	Thu, 01 Dec 2016 01:58:52 (-1.3h)
20	↓5	Andrew Beam	0.98727	9	Fri, 18 Nov 2016 12:42:05 (-19.3h)
21	↓3	DrorSh	1.00216	35	Tue, 29 Nov 2016 11:03:55 (-40h)
22	new	Anish Shah	1.00728	9	Wed, 30 Nov 2016 12:46:13 (-30.5h)
23	↓7	PhoenixLin	1.01032	10	Tue, 22 Nov 2016 01:59:50 (-0.7h)
24	↓7	Damodar	1.02252	14	Fri, 25 Nov 2016 02:50:54 (-24.3h)
25	new	**PengPai**	**1.02463**	**6**	**Thu, 01 Dec 2016 16:30:55**
		Your Best Entry ↑ You improved on your best score by 0.06754. You just moved up 17 positions on the leaderboard. Tweet this!			
26	↓7	FishingBR	1.03750	18	Mon, 28 Nov 2016 08:26:41 (-5.3d)
27	↓4	Bojan Tunguz	1.03797	37	Wed, 30 Nov 2016 21:29:40
28	↓2	YetiMan	1.04033	30	Thu, 24 Nov 2016 04:08:08 (-3.3d)

Result of Trick 3

Trick 3: With Different Training Parameters, e.g. Data Augmentation, Initialization, etc.

1	—	Paulo Pinto *	0.71038	6	Tue, 29 Nov 2016 14:16:51 (-4.7d)
2	—	冯唐易老，李广难封 *	0.74868	32	Thu, 24 Nov 2016 22:07:15 (-5.6h)
3	—	DataGeek|Breuker|Inversion *	0.79608	19	Thu, 24 Nov 2016 02:15:51
4	↑2	hitszzy *	0.79638	37	Fri, 02 Dec 2016 06:54:26 (-0.3h)
5	↓1	Alec Karfonta *	0.80733	21	Tue, 06 Dec 2016 00:01:46 (-5.9d)
6	↓1	John Seamons	0.80744	22	Mon, 05 Dec 2016 07:38:13 (-14.2d)
7	↑214	Florian Muellerklein	0.81247	5	Tue, 06 Dec 2016 03:32:07 (-36.2h)
8	↑32	Igor Krashenyi	0.82096	43	Sat, 03 Dec 2016 14:05:26 (-27h)
9	↓2	¸.·´¯`·.·´¯`·.¸¸.·´¯`·.¸. ><(((º>	0.85661	33	Fri, 25 Nov 2016 18:56:54
10	↓2	trangle	0.86197	12	Fri, 25 Nov 2016 15:07:25 (-24.4h)
11	↓2	VictorGarcia	0.86260	24	Sat, 26 Nov 2016 09:14:55 (-0h)
12	↓2	ZFTurbo	0.86698	29	Fri, 25 Nov 2016 13:17:42 (-5.1d)
13	**↑29**	**PengPai**	**0.88741**	**7**	**Tue, 06 Dec 2016 13:40:05**
14	↓3	穷且益坚，不坠青云之志	0.90025	17	Tue, 22 Nov 2016 02:21:48
15	↓3	Andrey Ostapets	0.90055	10	Sun, 27 Nov 2016 20:06:37
16	↓3	Zshiney	0.91655	10	Wed, 16 Nov 2016 22:18:31 (-21.6h)
17	↓3	njm	0.92599	31	Mon, 05 Dec 2016 12:31:32 (-3d)
18	↓3	XuleiYang	0.92936	9	Fri, 02 Dec 2016 04:52:56 (-0h)
19	↓3	wlamond	0.94289	17	Mon, 21 Nov 2016 12:51:27 (-4.3d)
20	new	sirnodin	0.94683	5	Tue, 06 Dec 2016 07:46:23

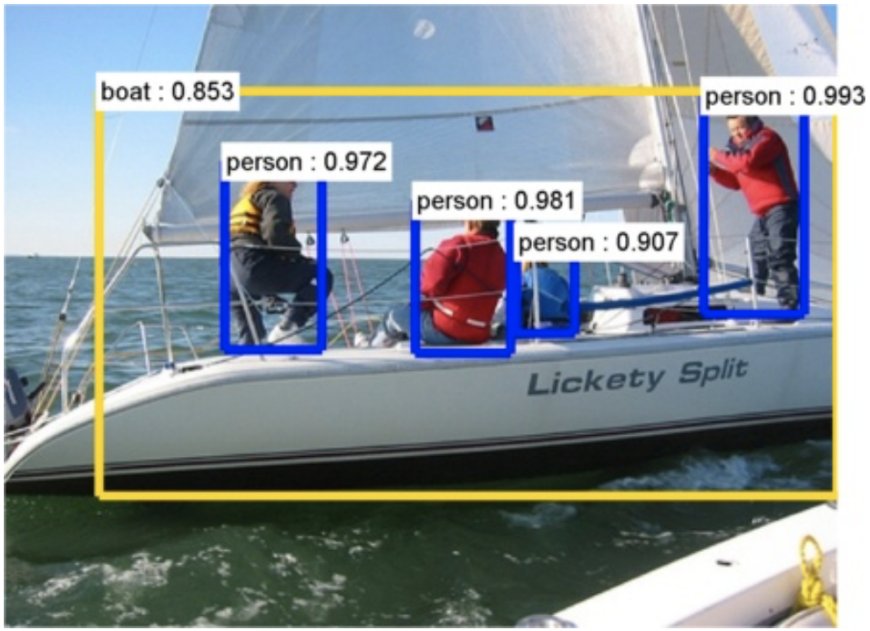
Homework 2: ConvNets for NCFM

https://www.kaggle.com/c/the-nature-conservancy-fisheries-monitoring Train a ConvNet for NCFM without start codes. You could still submit your predictions after deadline. Some hints which may help:

- Pick up a STOA ConvNet architecture, e.g. InceptionV3, ResNet, Inception-ResNet, DenseNet, etc.
- Use pre-trained weights on ImageNet speeds up convergence.
- Finetune with a small learning rate.
- Use data augmentation to reduce overfitting.
- Split train and local validation.
- Ensemble methods always help to further improve.
- https://www.kaggle.com/c/the-nature-conservancy-fisheries-monitoring/discussion/25902
- https://github.com/yhenon/keras-frcnn (more on this later).

Image Classification and Object Detection

Image classification
(what? *I don't care where*)



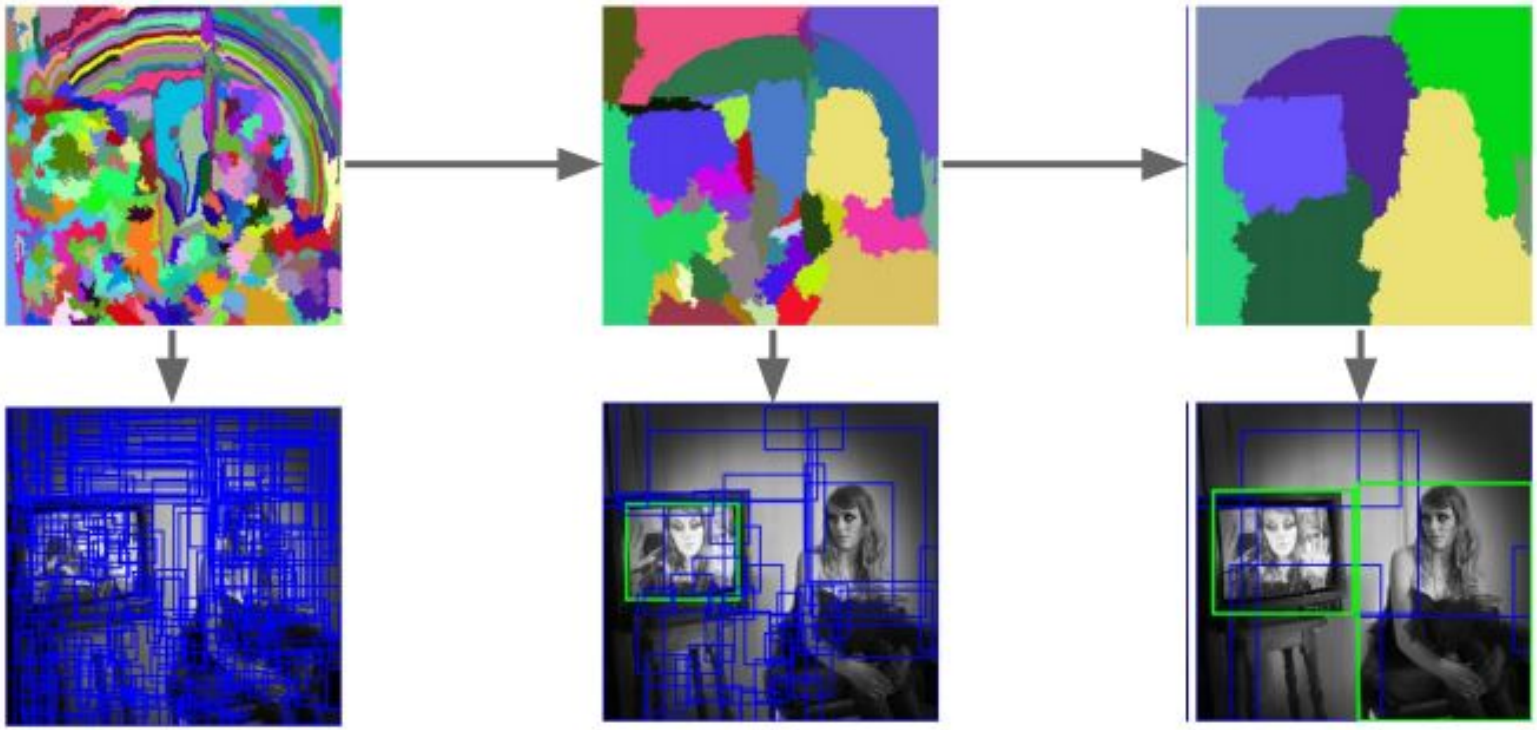
Object detection
(what + where?)

Selective Search: Discovering Region Proposals

Region Proposals: Selective Search

Bottom-up segmentation, merging regions at multiple scales

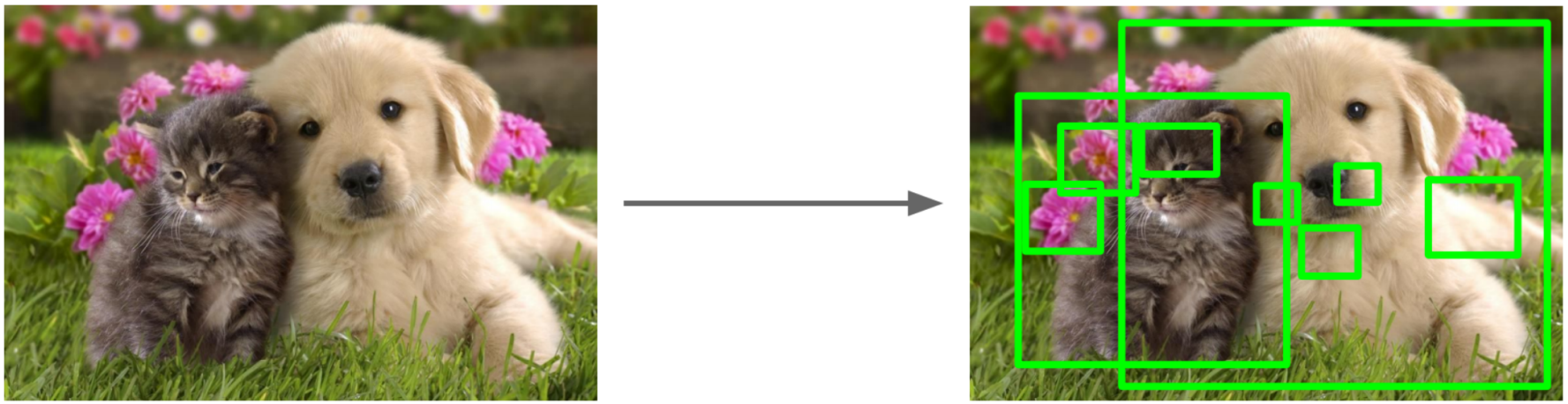
Convert regions to boxes



Uijlings et al, "Selective Search for Object Recognition", IJCV 2013

Region Proposals

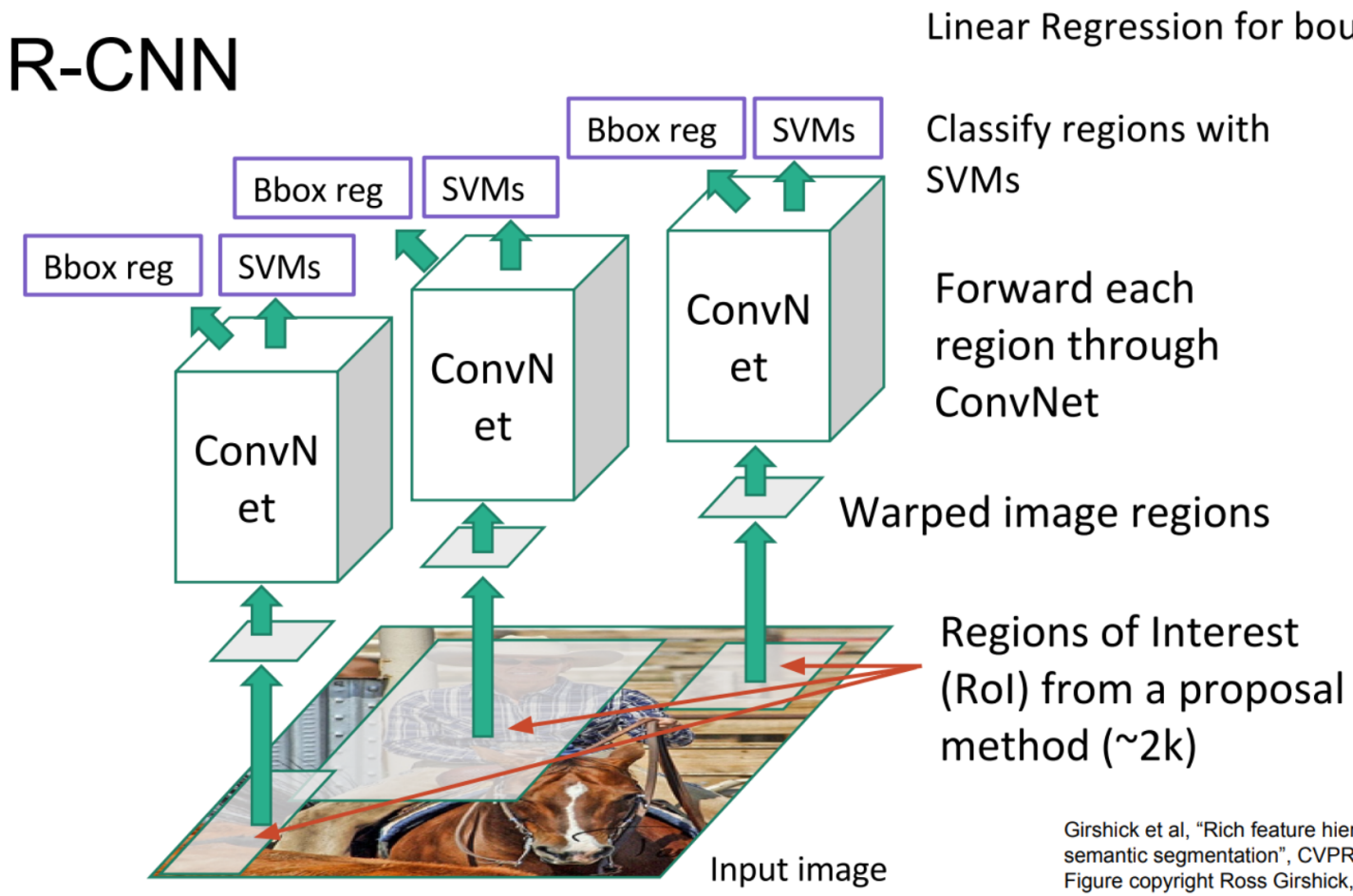
- Find “blobby” image regions that are likely to contain objects
- Relatively fast to run; e.g. Selective Search gives 1000 region proposals in a few seconds on CPU



Alexe et al, “Measuring the objectness of image windows”, TPAMI 2012
Uijlings et al, “Selective Search for Object Recognition”, IJCV 2013
Cheng et al, “BING: Binarized normed gradients for objectness estimation at 300fps”, CVPR 2014
Zitnick and Dollar, “Edge boxes: Locating object proposals from edges”, ECCV 2014

R-CNN

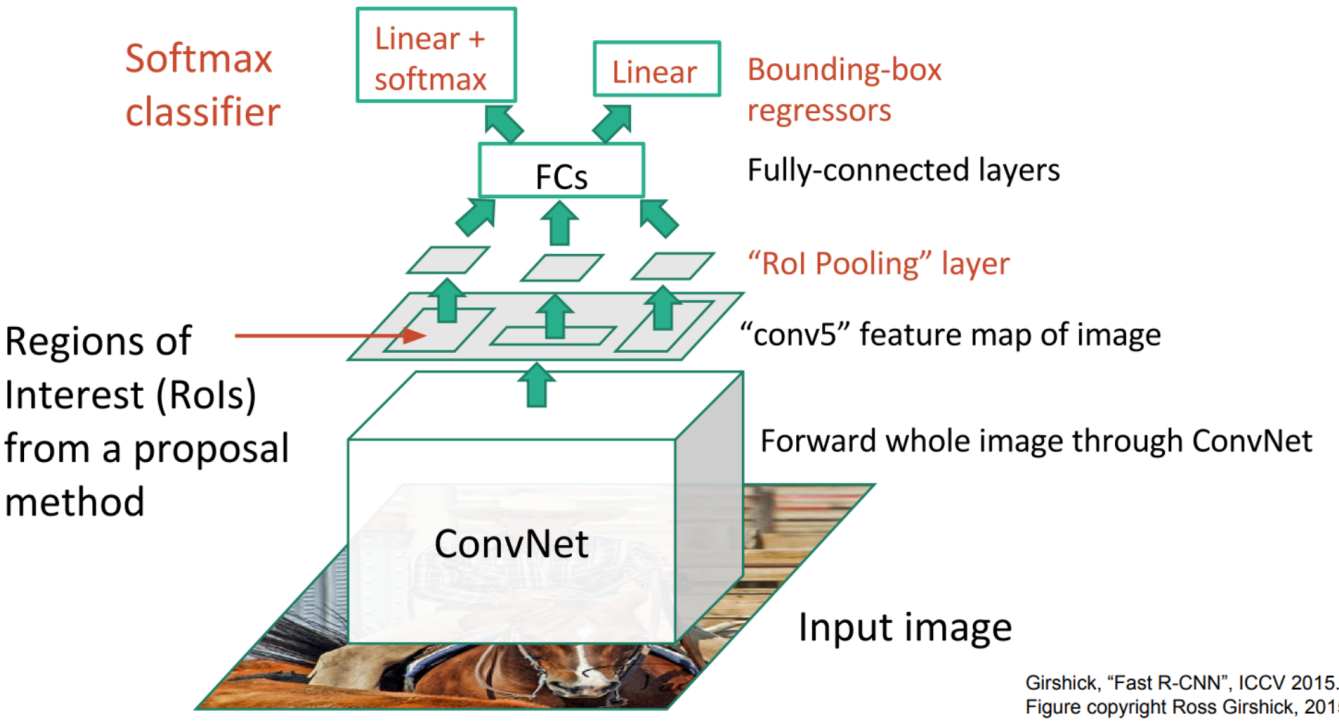
- Training is slow (84 hours), takes a lot of disk space (saving proposals and features).
- Inference (detection) is slow, 47 seconds / image with VGG16.



Girshick et al, "Rich feature hierarchies for accurate object detection and semantic segmentation", CVPR 2014.
Figure copyright Ross Girshick, 2015; source. Reproduced with permission.

Fast R-CNN

- ConvNets weights are shared when extracting features instead of feed-forward for each region proposal feature extraction.
- Visual features are extracted online instead of saving features on disk before feeding into SVM.
- Use softmax classification in the output layer instead of SVM, i.e. multi-task leanring.
- ROI pooling layer is proposed to accomodate fully-connected layer.

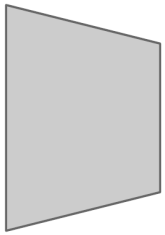


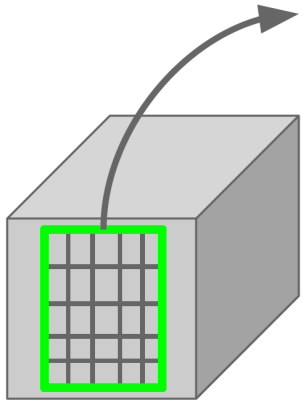
Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; source. Reproduced with permission.

ROI Pooling

Hi-res input image:
3 x 800 x 600
with region
proposal

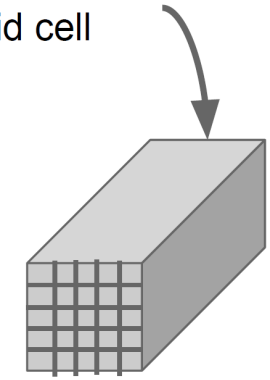
Convolution
and Pooling





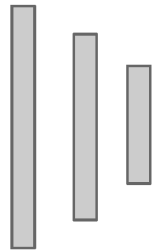
Hi-res conv features:
C x H x W
with region proposal

Max-pool within
each grid cell



RoI conv features:
C x h x w
for region proposal

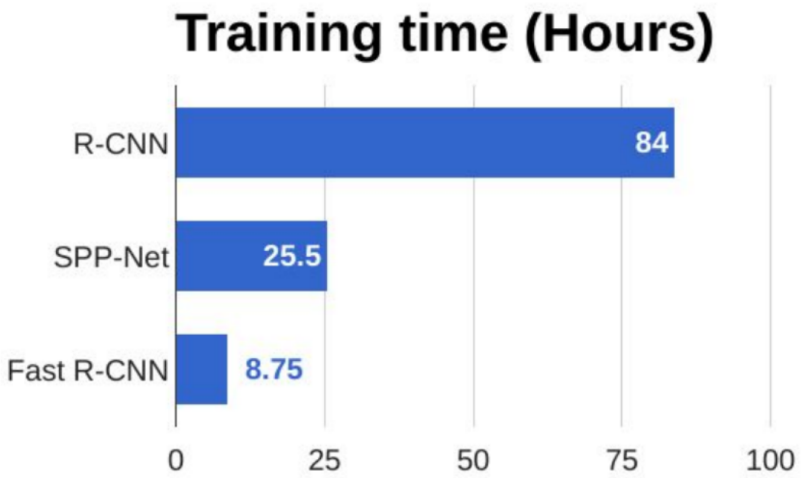
Fully-connected
layers

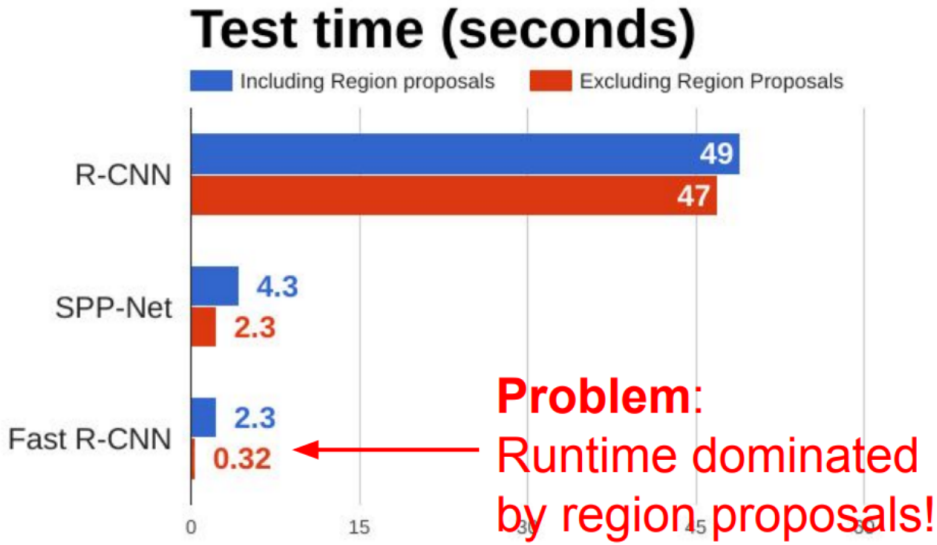


Fully-connected layers expect
low-res conv features:
C x h x w

R-CNN VS. Fast R-CNN

R-CNN vs SPP vs Fast R-CNN





Girshick et al, "Rich feature hierarchies for accurate object detection and semantic segmentation", CVPR 2014.
He et al, "Spatial pyramid pooling in deep convolutional networks for visual recognition", ECCV 2014
Girshick, "Fast R-CNN", ICCV 2015

Faster R-CNN

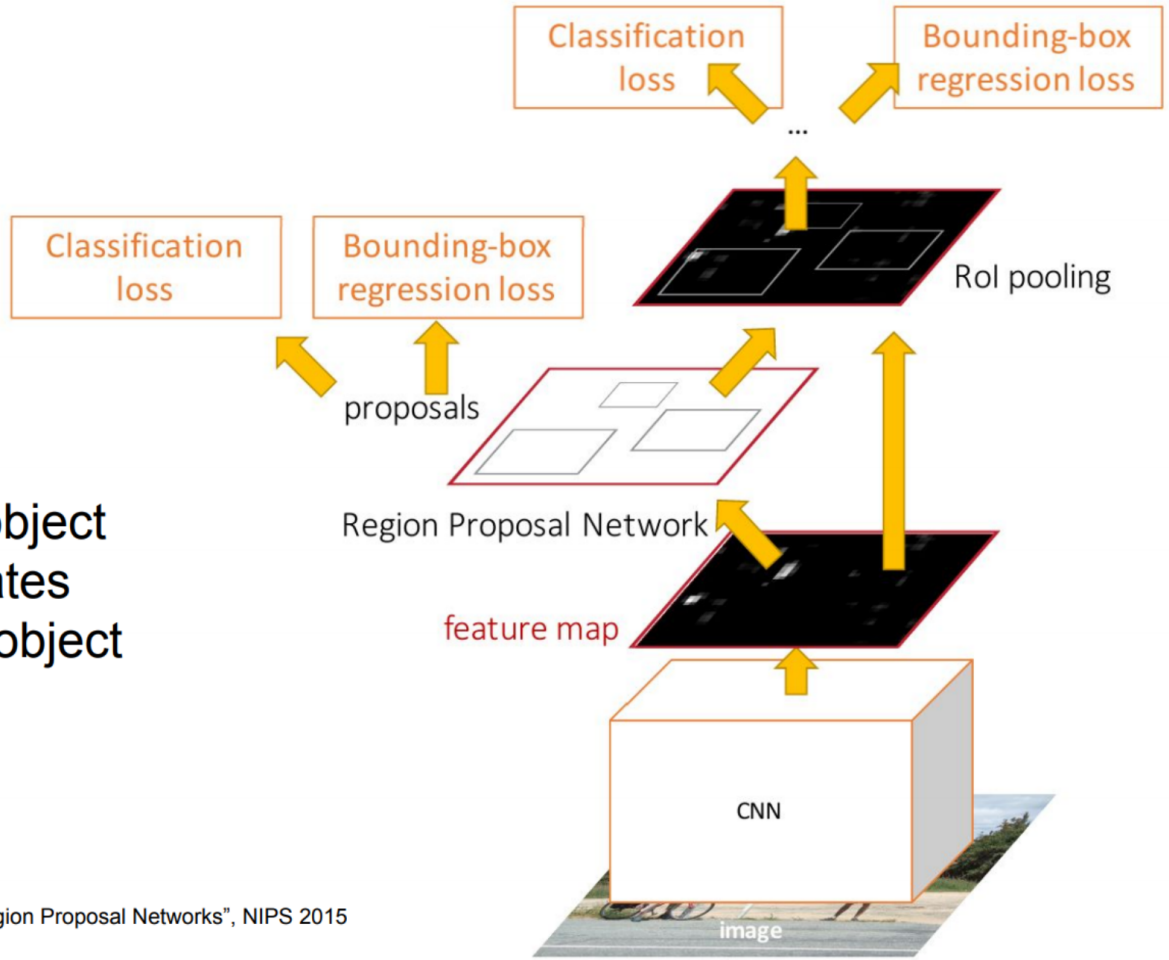
Faster R-CNN:

Make CNN do proposals!

Insert **Region Proposal Network (RPN)** to predict proposals from features

Jointly train with 4 losses:

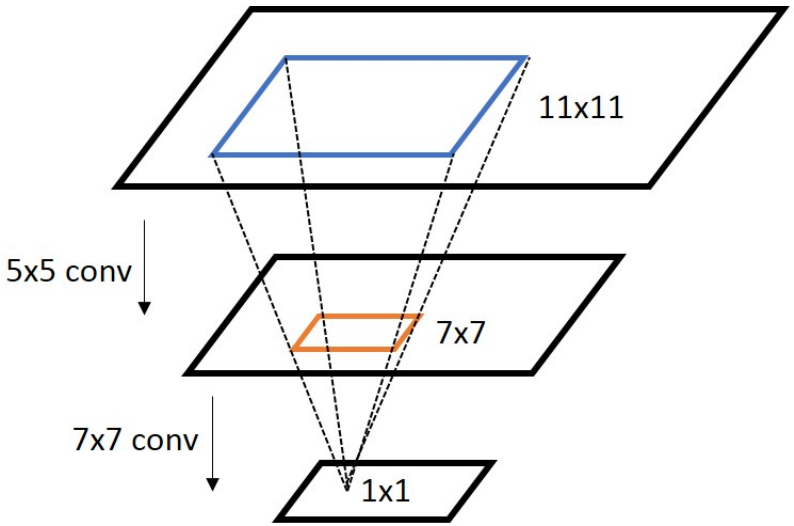
1. RPN classify object / not object
2. RPN regress box coordinates
3. Final classification score (object classes)
4. Final box coordinates

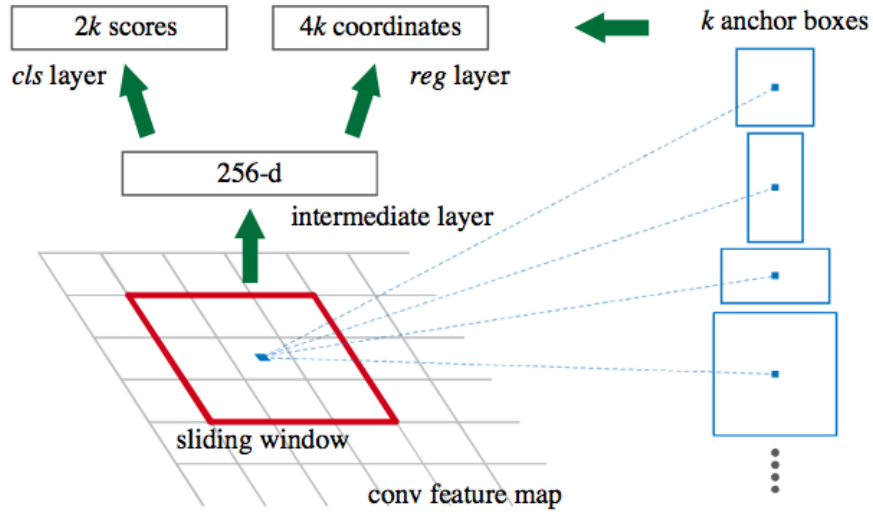


Ren et al, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", NIPS 2015
Figure copyright 2015, Ross Girshick; reproduced with permission

Receptive Field and Feature Mapping

- $output\_field\_size = (input\_field\_size - kernel\_size + 2 * padding)/stride + 1$
- $input\_field\_size = output\_field\_size - 1) * stride - 2 * padding + kernel\_size$
- Project each anchor in the feature map back to the original input image. Each anchor corresponds 3 differentscales: $\{128^2, 256^2, 512^2\}$ and 3 different aspect ratios: $\{1:1, 1:2, 2:1\}$.

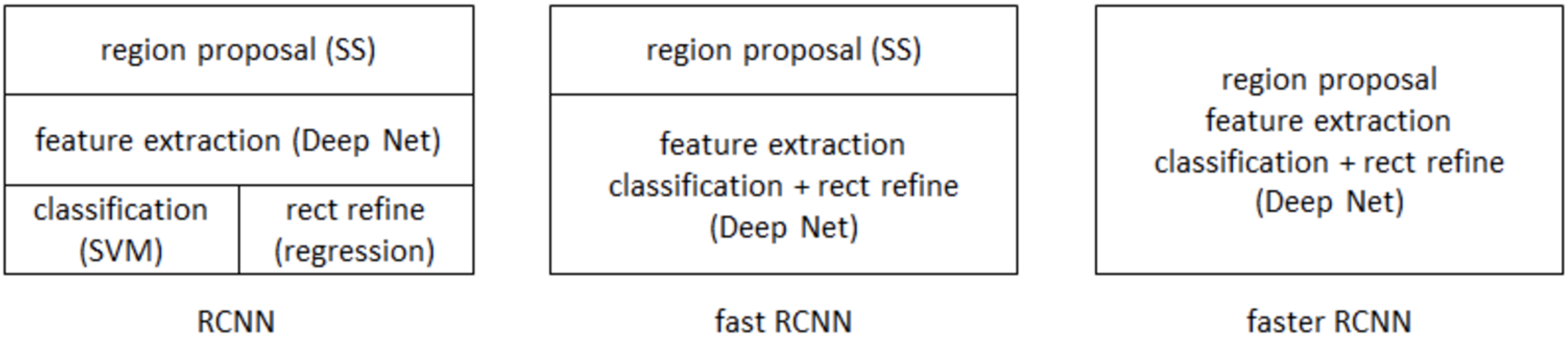




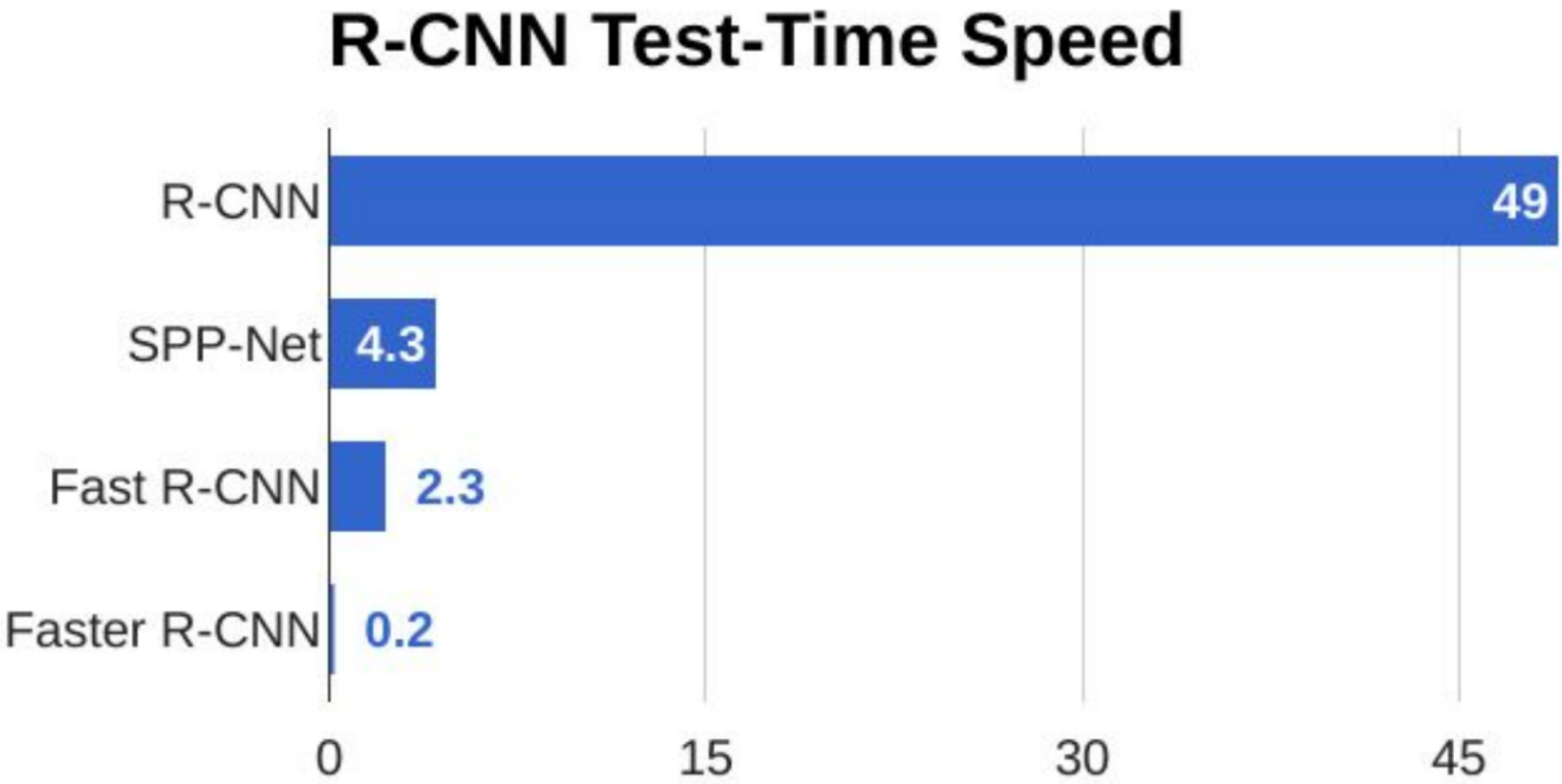
R-CNN VS. Fast R-CNN VS. Faster R-CNN

Roadmap: R-CNN $\Rightarrow$ Fast R-CNN $\Rightarrow$ Faster R-CNN

- Deep ConvNets are awesome! Why not use deep visual features?!
- Feedford from scratch for each region proposal feature extraction is unnecessary. Why not share the ConvNets weights?!
- Selective search for region proposals becomes the bottleneck. Why not predict region proposals inside the ConvNet (RPN, region proposal network)?!



Faster R-CNN Inference Time



Summary

- Recap neural networks basics: loss function, computing gradients.
- Backpropation intuition and deriviation in details.
- Tuning parameters for neural networks in practice.
- Two important features in covolutional layers design:

Summary

- Recap neural networks basics: loss function, computing gradients.
- Backpropation intuition and deriviation in details.
- Tuning parameters for neural networks in practice.
- Two important features in covolutional layers design: local connectivity and parameters sharing.
- How to compute output volume of a convolutional layer.
- Transfer learning in neural networks: fine-tune.
- Kaggle in practice: applying STOA ConvNets on a Kaggle competition.
- ConvNets for object detection: R-CNN, Fast-RCNN, Faster-RCNN, YOLO / SSD.