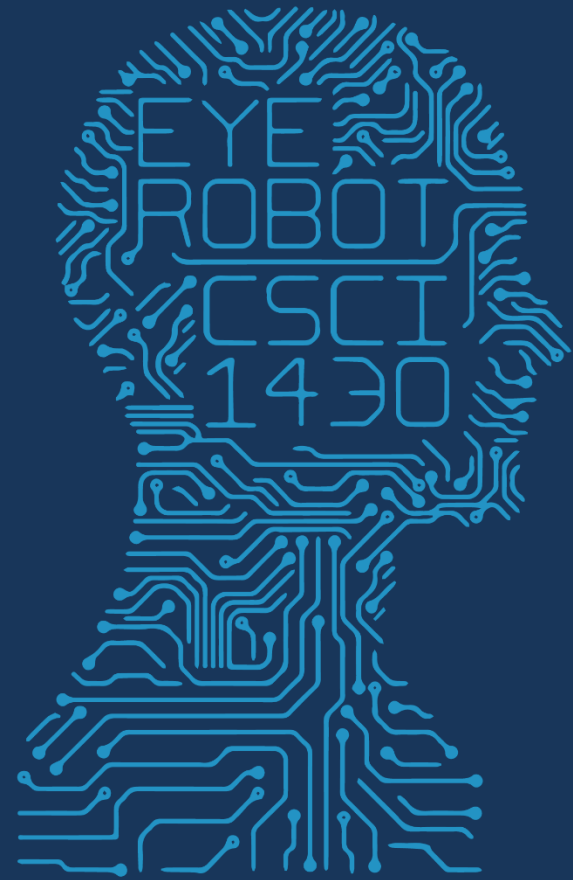




1950

FUTURE VISION



2020

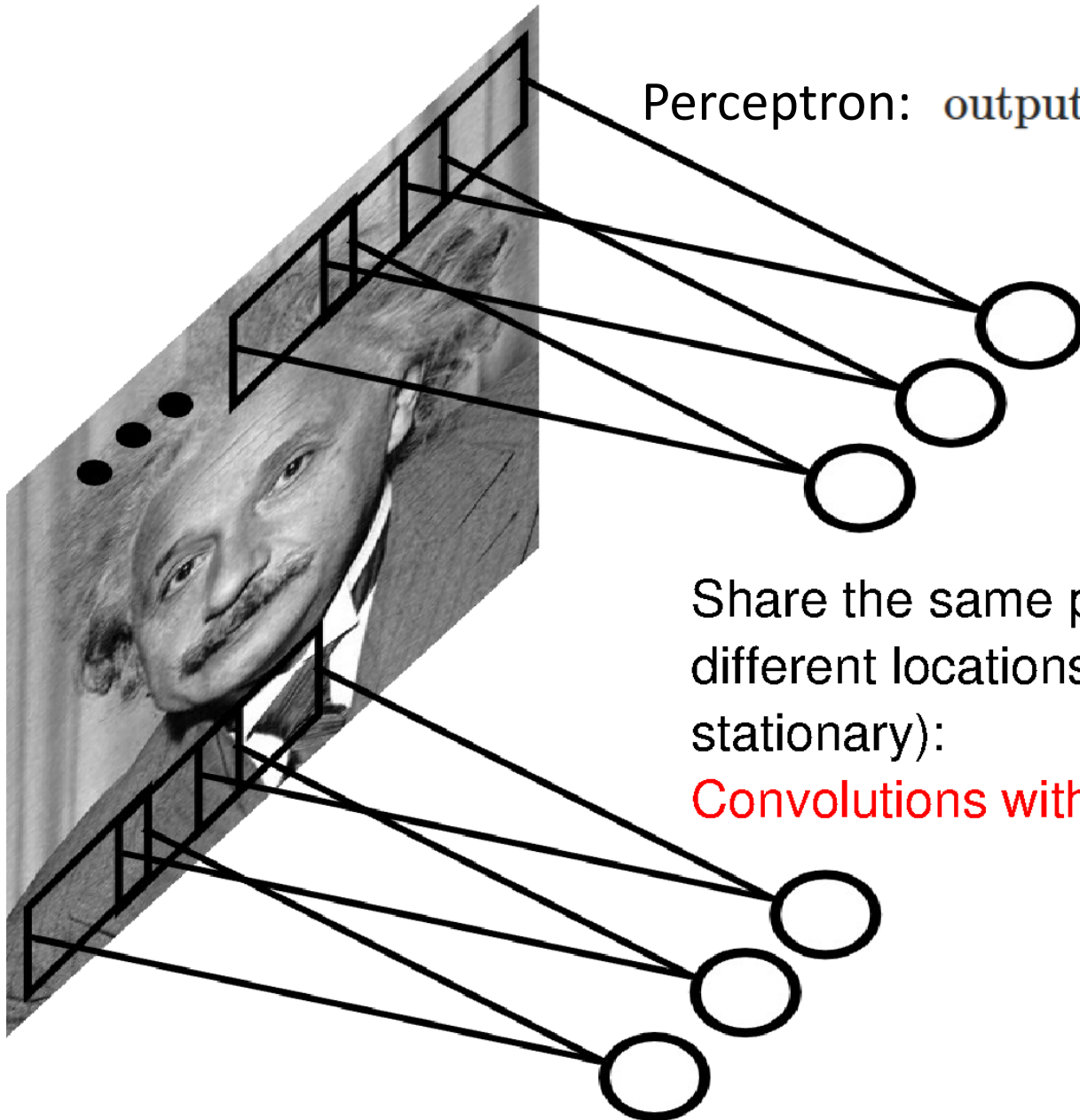
COMPUTER VISION







Convolutional Layer



$$\text{Perceptron: } \text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

$$w \cdot x \equiv \sum_j w_j x_j$$

This is convolution!

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

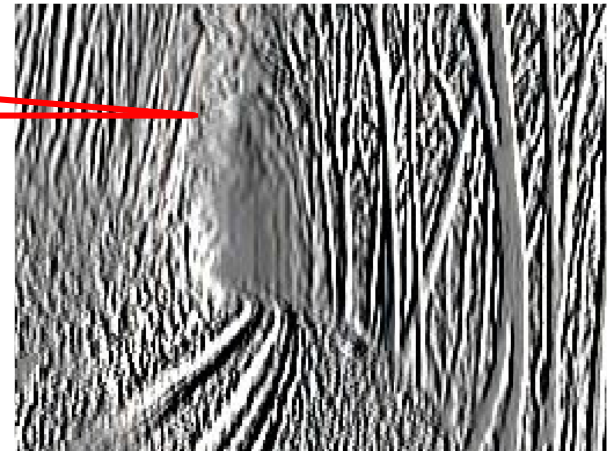
Convolutions with learned kernels

Convolutional Layer

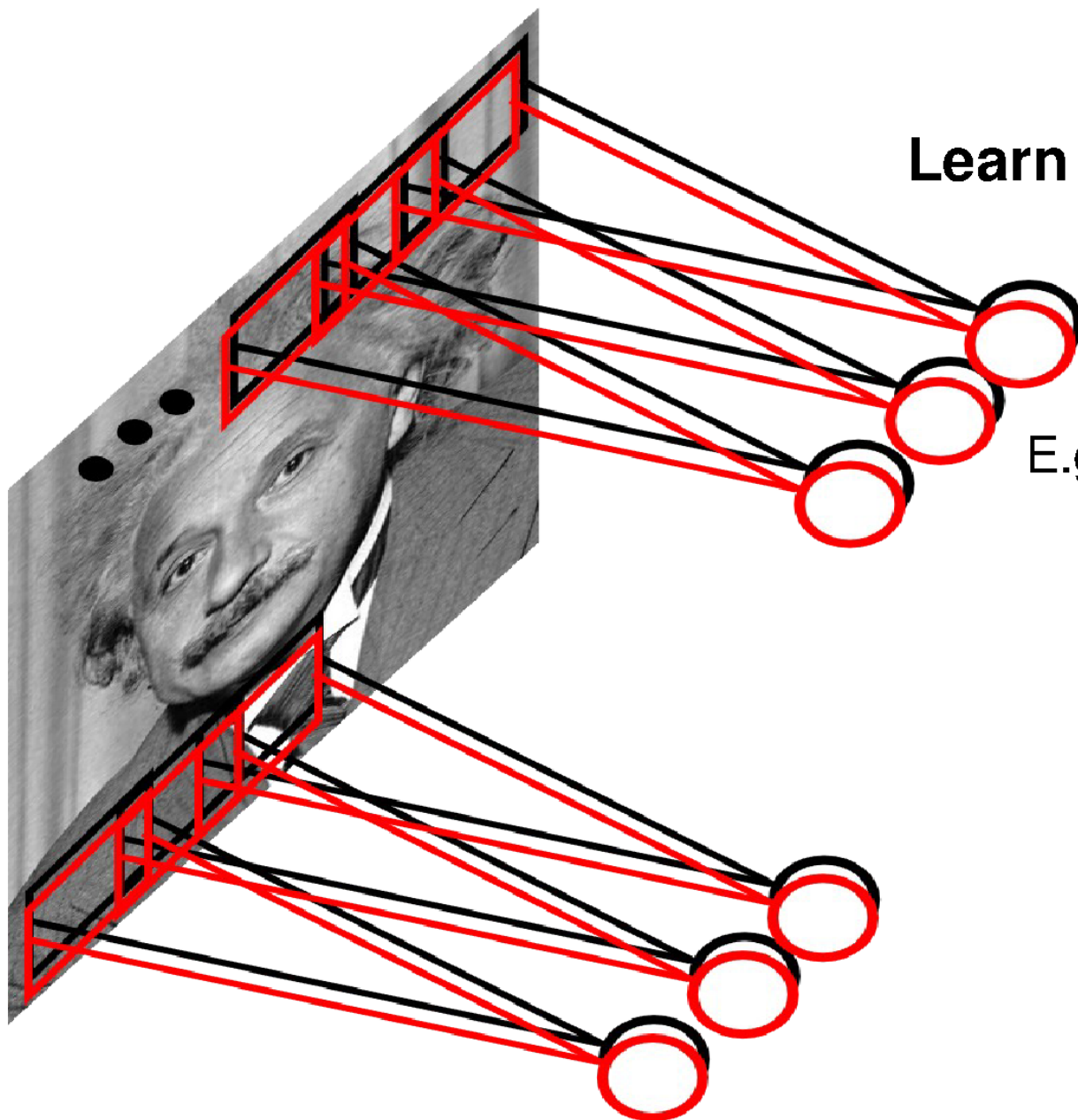


$$* \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} =$$

Shared weights



Convolutional Layer

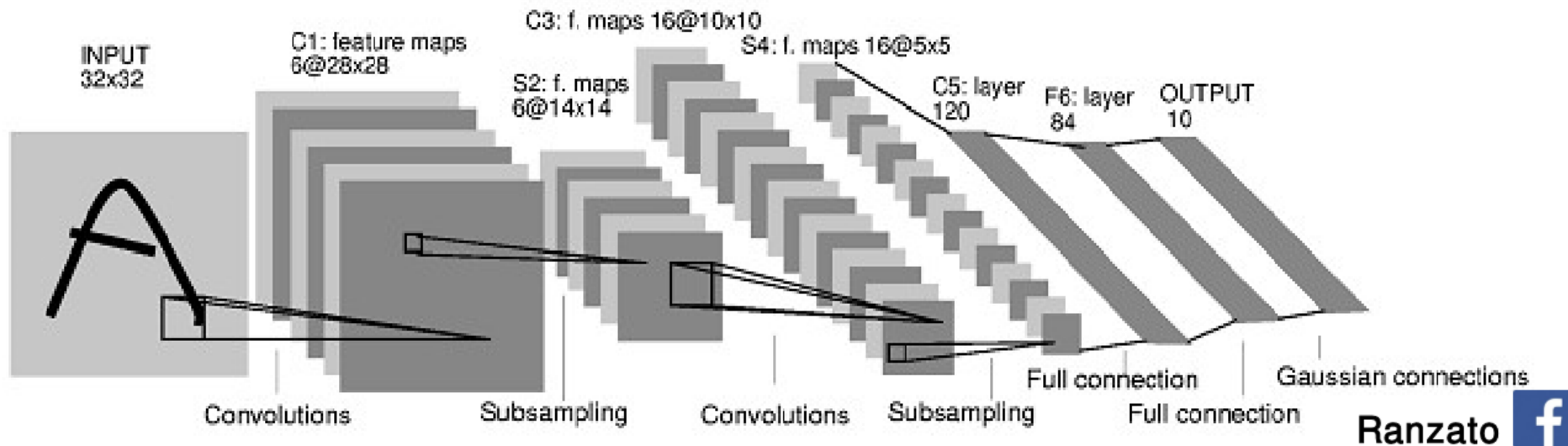


Learn **multiple filters.**

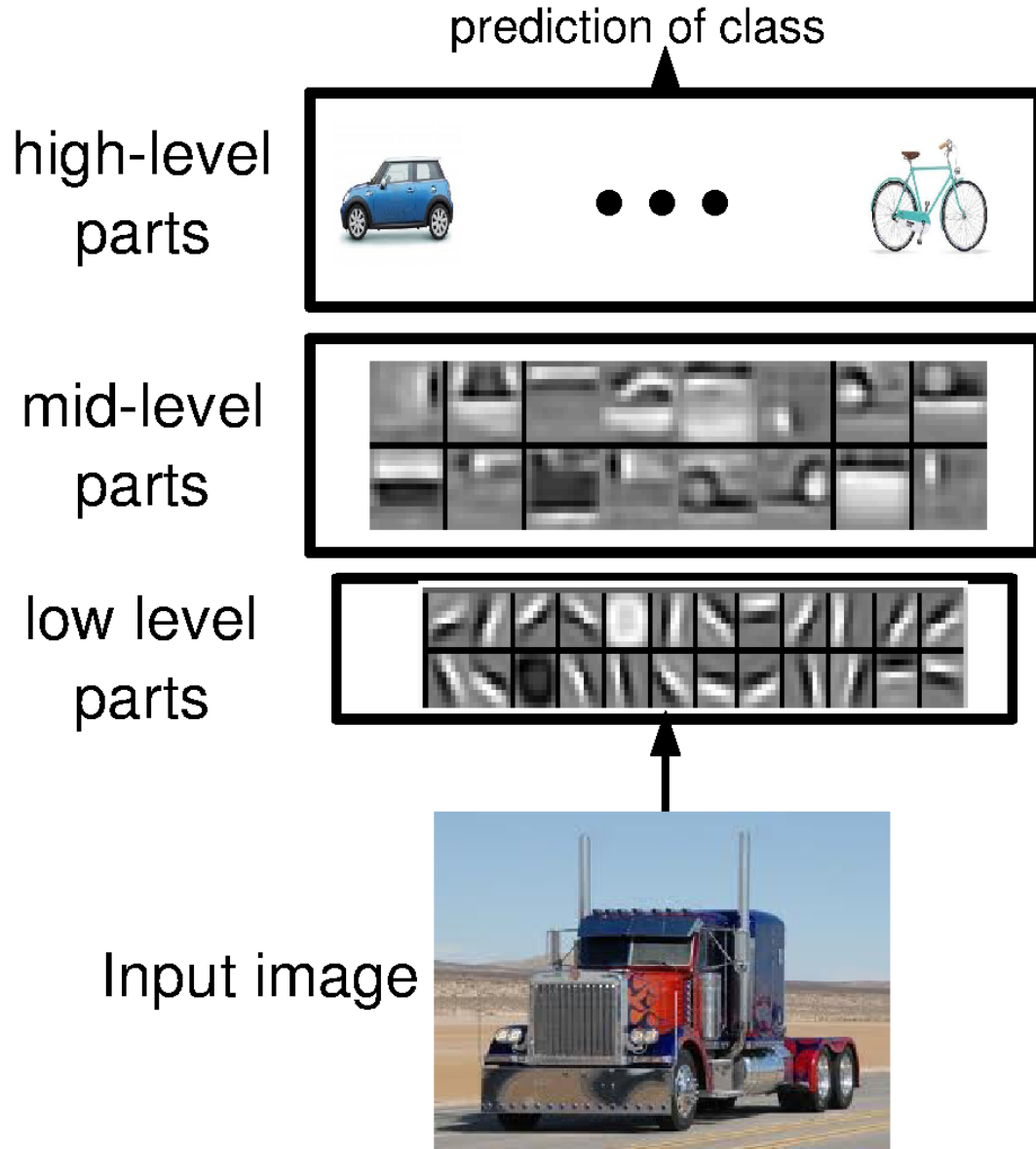
Filter = 'local' perceptron.
Also called *kernel*.

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

Yann LeCun's MNIST CNN architecture



Interpretation



- distributed representations
- feature sharing
- compositionality

DEMO

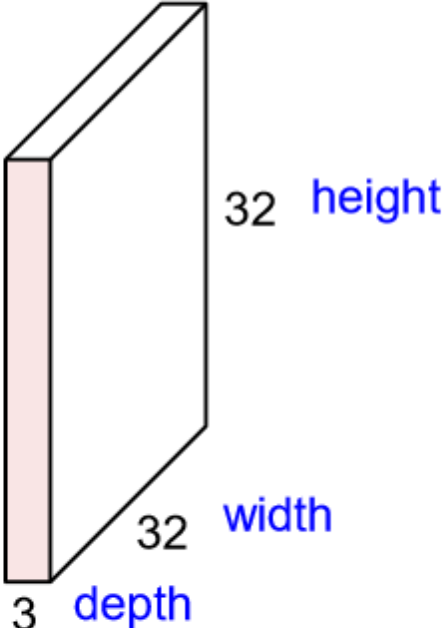
<http://scs.ryerson.ca/~aharley/vis/conv/>

Thanks to Adam Harley for making this.

More here: <http://scs.ryerson.ca/~aharley/vis>

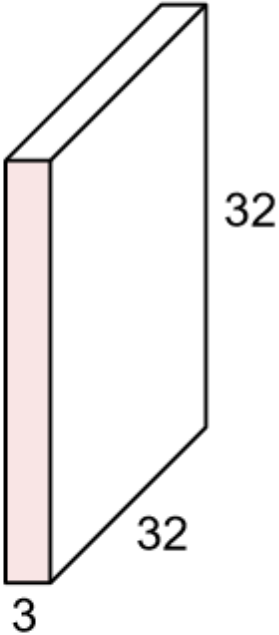
Convolutions: More detail

32x32x3 image



Convolutions: More detail

32x32x3 image

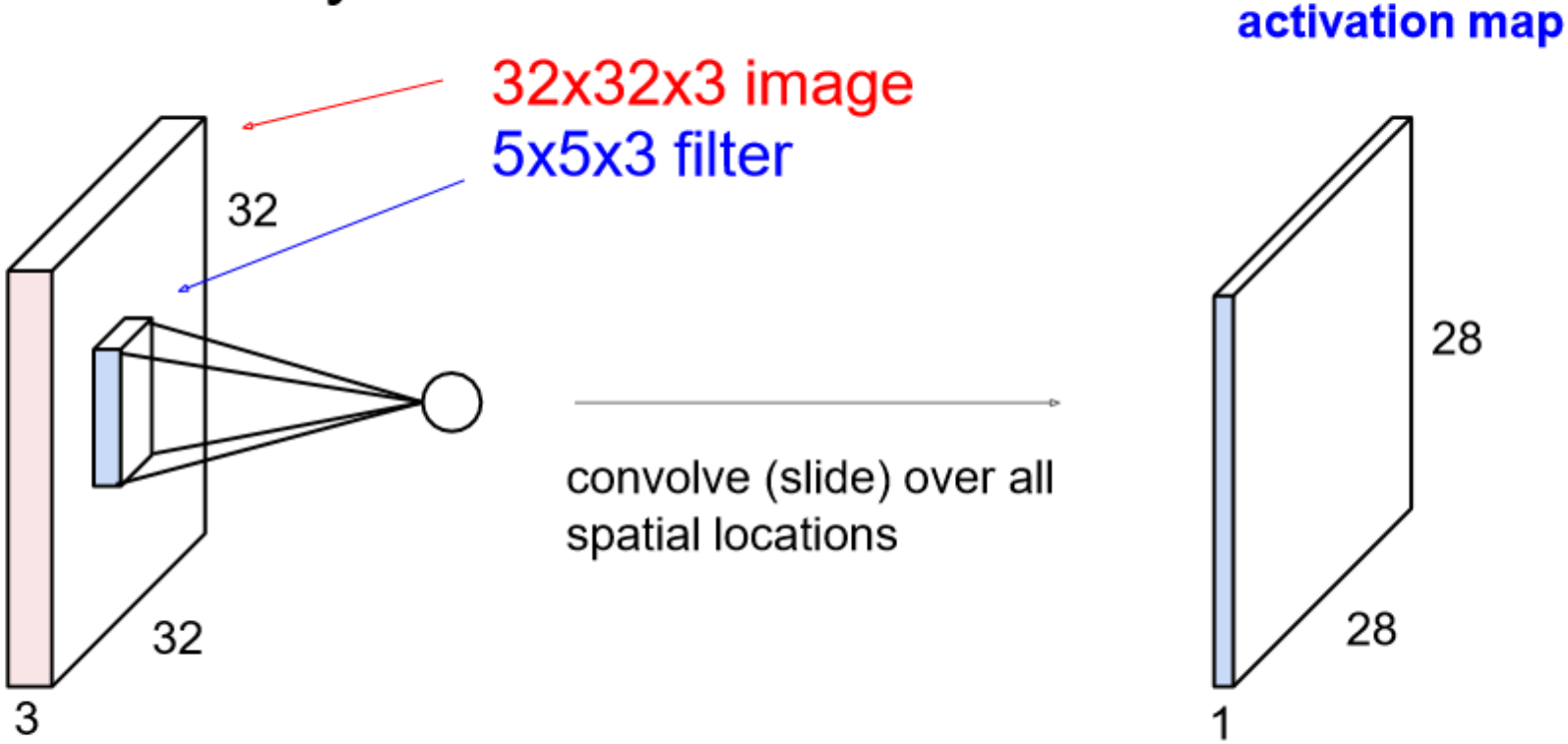


5x5x3 filter



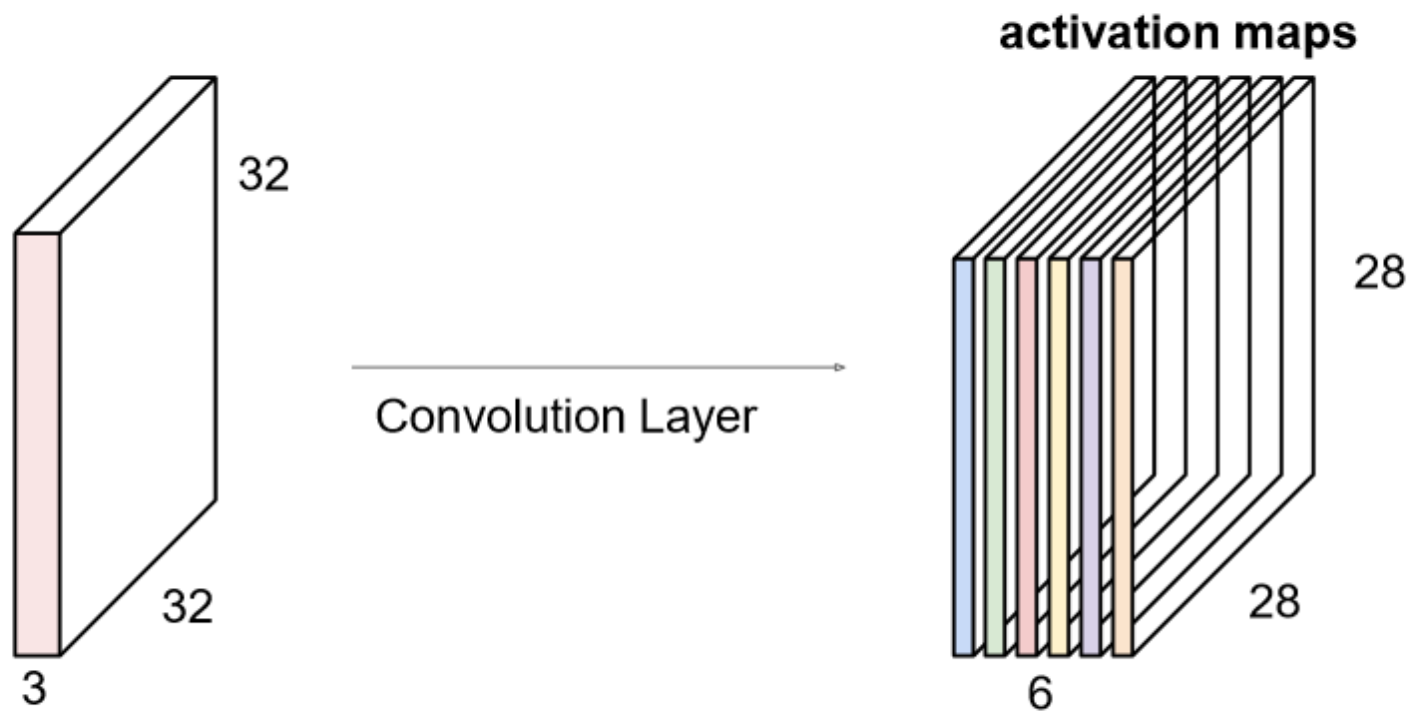
Convolutions: More detail

Convolution Layer



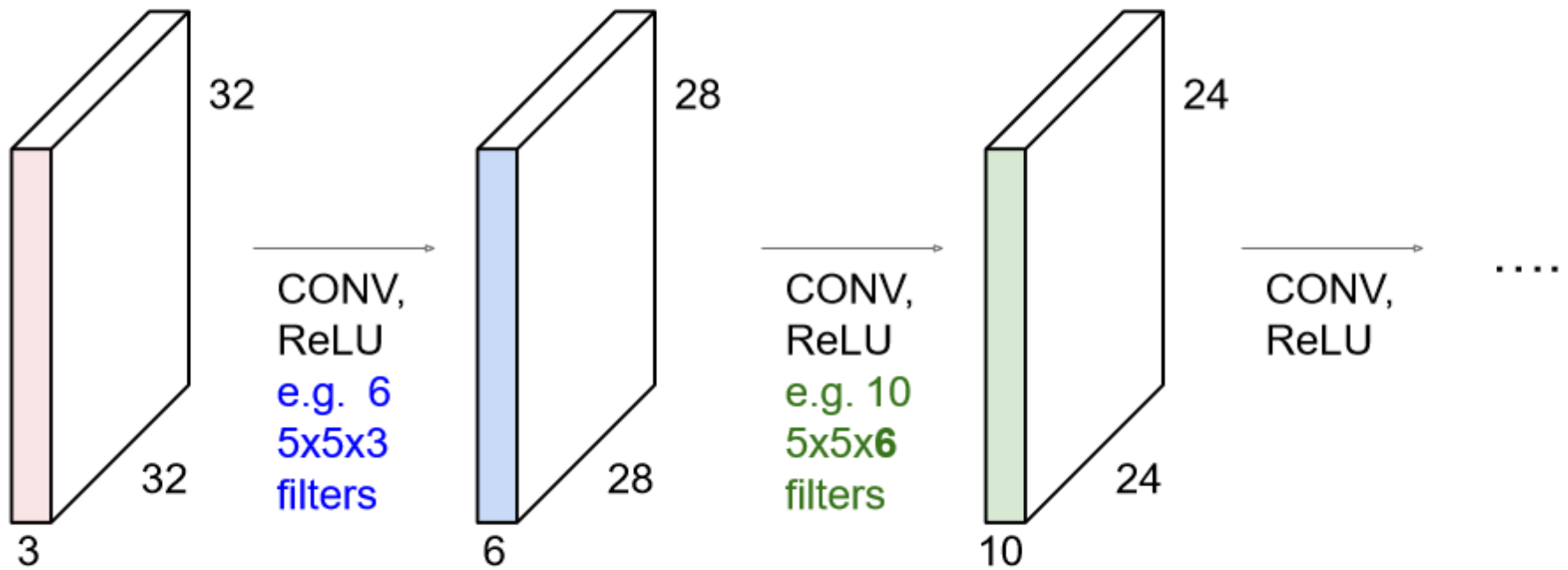
Convolutions: More detail

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!

Convolutions: More detail



Think-Pair-Share

Input size: $96 \times 96 \times 3$

Kernel size: $5 \times 5 \times 3$

Stride: 1

Max pooling layer: 4×4

Output feature map size?

a) 5×5

b) 22×22

c) 23×23

d) 24×24

e) 25×25

Input size: $96 \times 96 \times 3$

Kernel size: $3 \times 3 \times 3$

Stride: 3

Max pooling layer: 8×8

Output feature map size?

a) 2×2

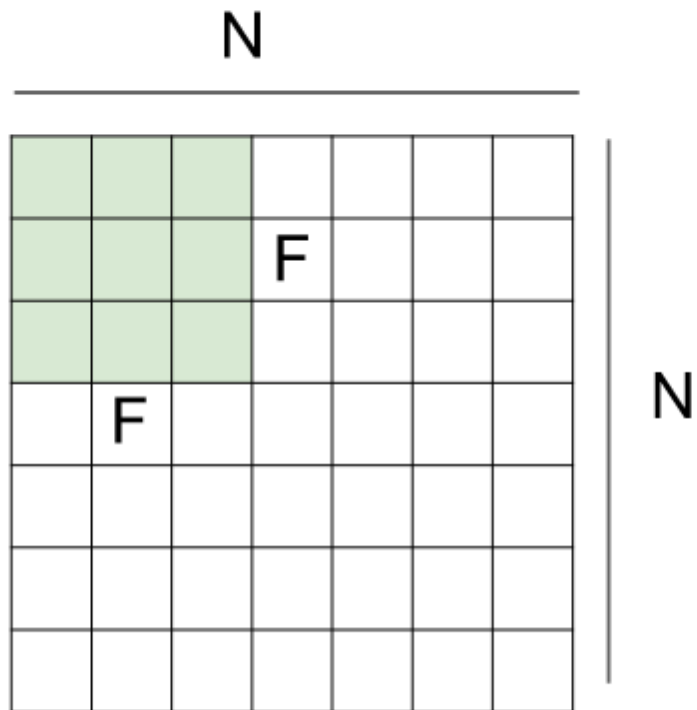
b) 3×3

c) 4×4

d) 5×5

e) 12×12

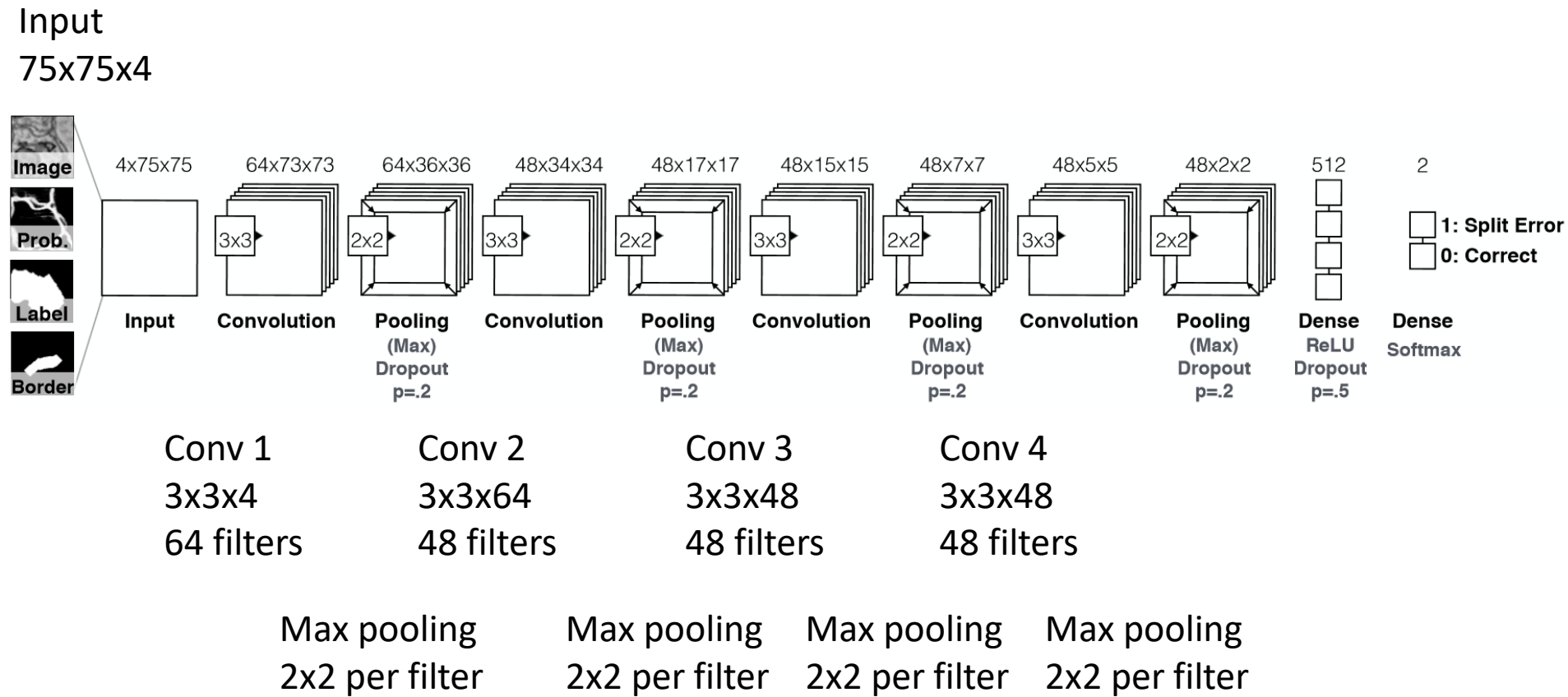
Convolutions: More detail



Output size:
 $(N - F) / \text{stride} + 1$

Our connectomics diagram

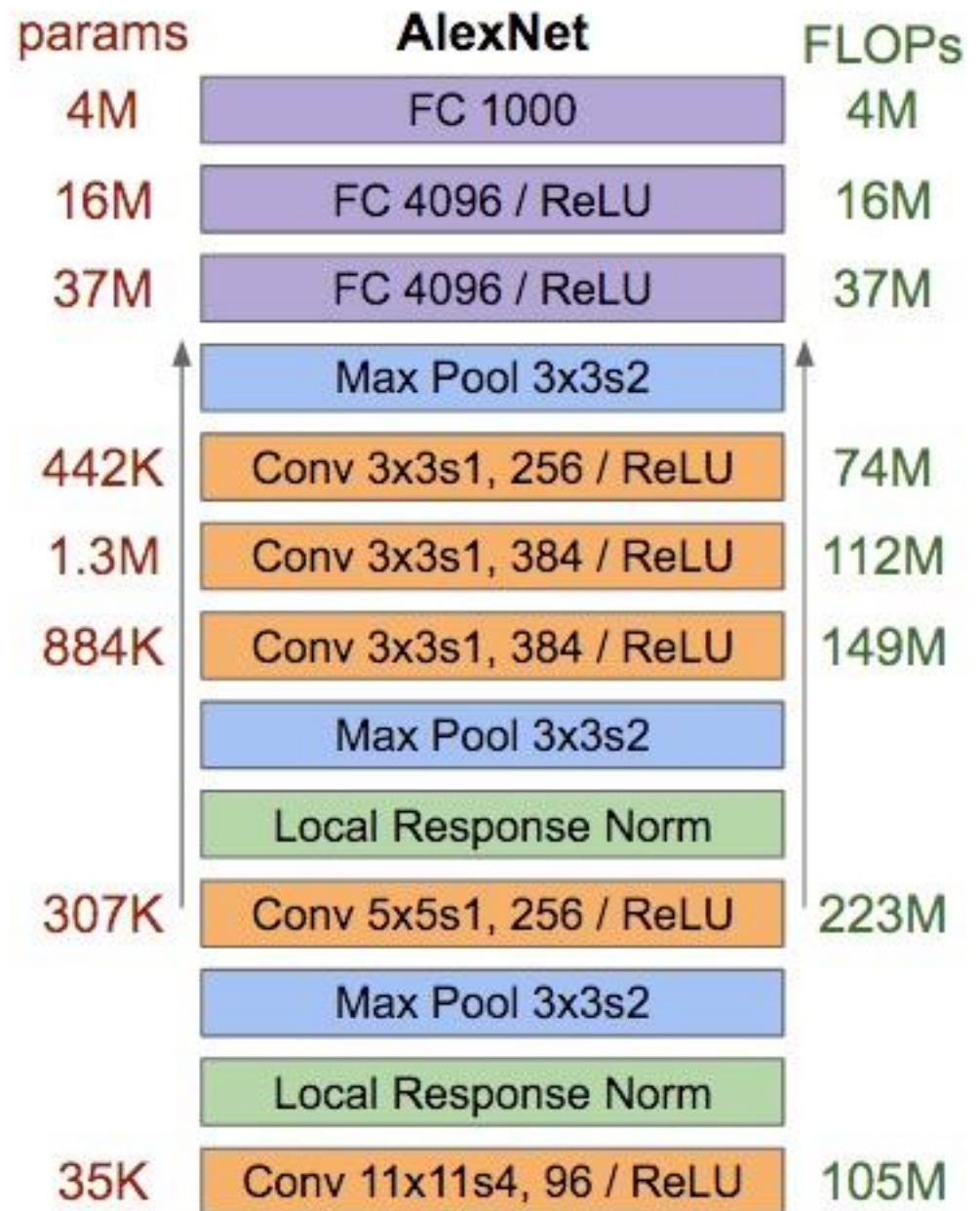
Auto-generated from network declaration by *nolearn* (for Lasagne / Theano)



Reading architecture diagrams

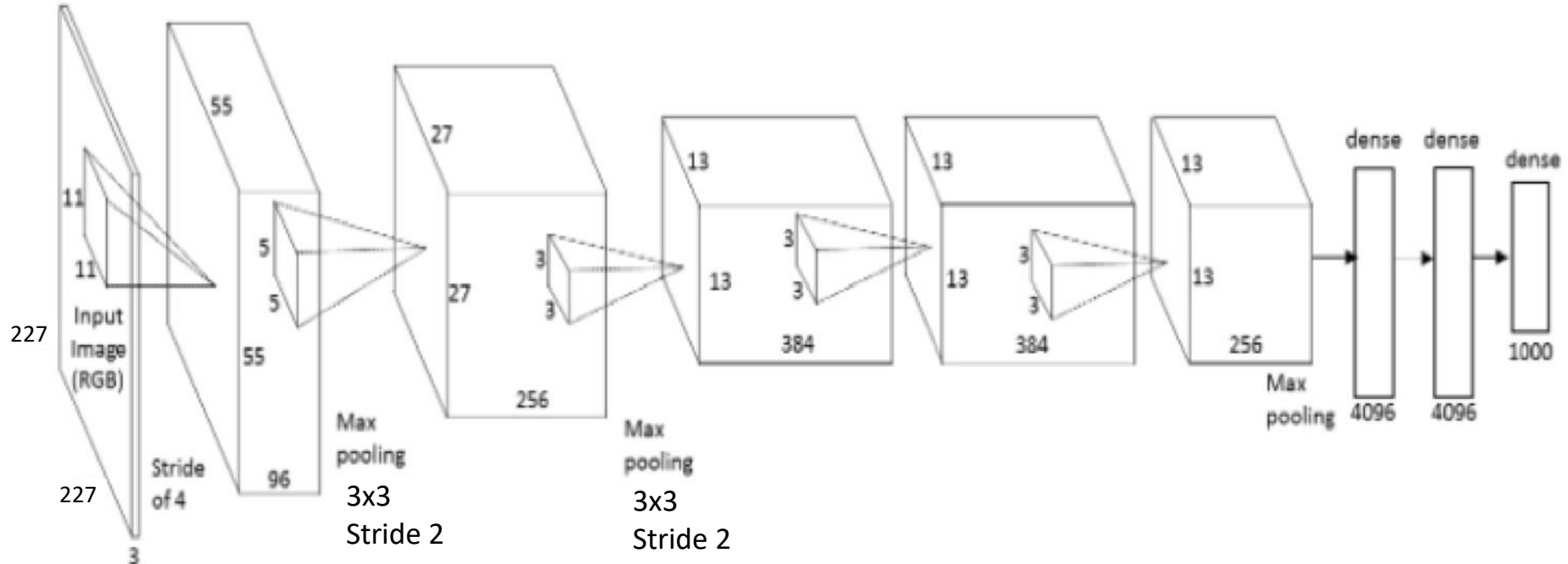
Layers

- Kernel sizes
- Strides
- # channels
- # kernels
- Max pooling



AlexNet diagram (simplified)

Input size
227 x 227 x 3



Conv 1

11 x 11 x 3
Stride 4
96 filters

Conv 2

5 x 5 x 96
Stride 1
256 filters

Conv 3

3 x 3 x 256
Stride 1
384 filters

Conv 4

3 x 3 x 192
Stride 1
384 filters

Conv 4

3 x 3 x 192
Stride 1
256 filters

Wait, why isn't it called a correlation neural network?

It could be.

Deep learning libraries implement correlation.

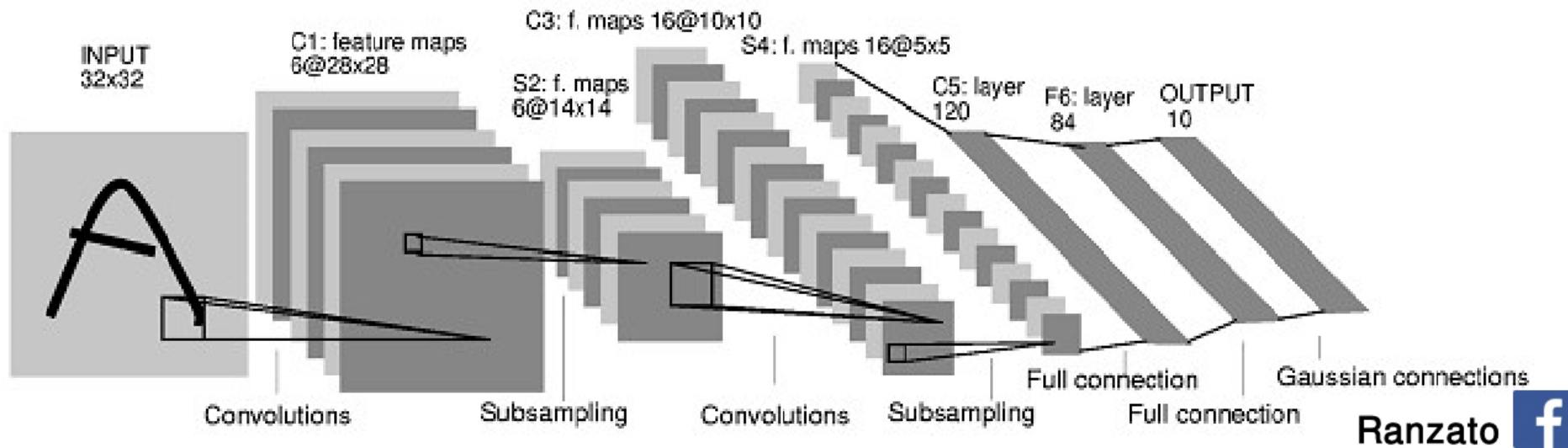
Correlation relates to convolution via a 180deg rotation of the kernel. When we *learn* kernels, we could easily learn them flipped.

Associative property of convolution ends up not being important to our application, so we just ignore it.

[p.323, Goodfellow]

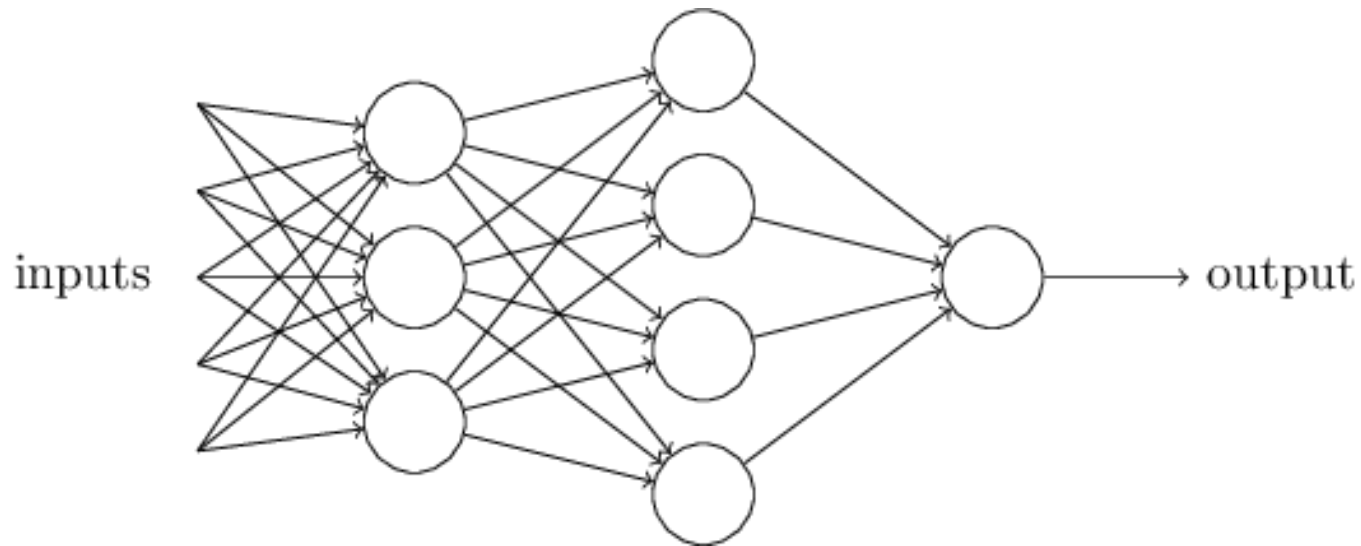
What does it mean to convolve over greater-than-first-layer hidden units?

Yann LeCun's MNIST CNN architecture



Multi-layer perceptron (MLP)

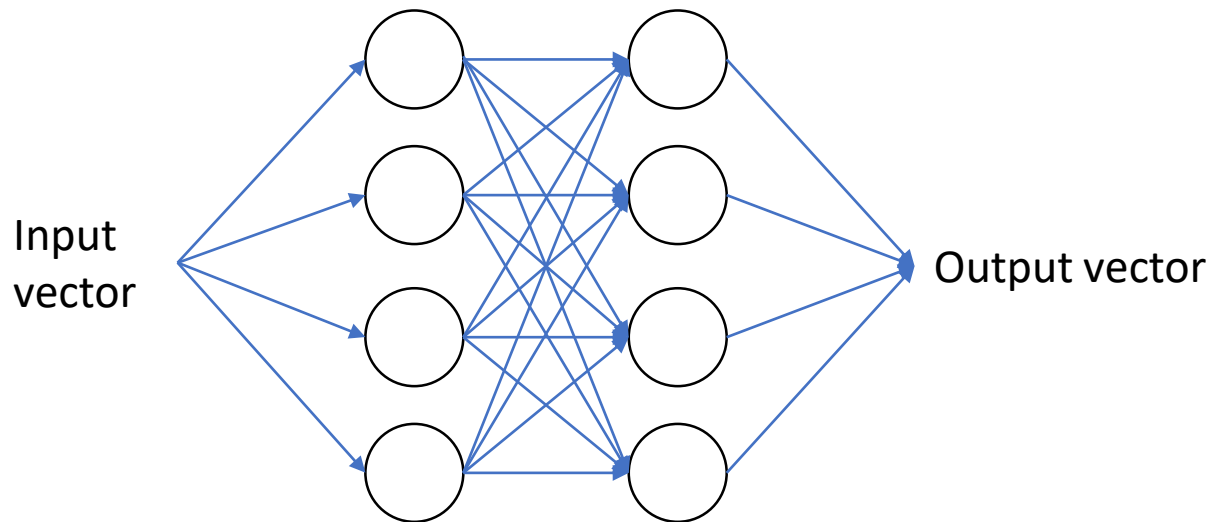
...is a *'fully connected'* neural network with non-linear activation functions.



'Feed-forward' neural network

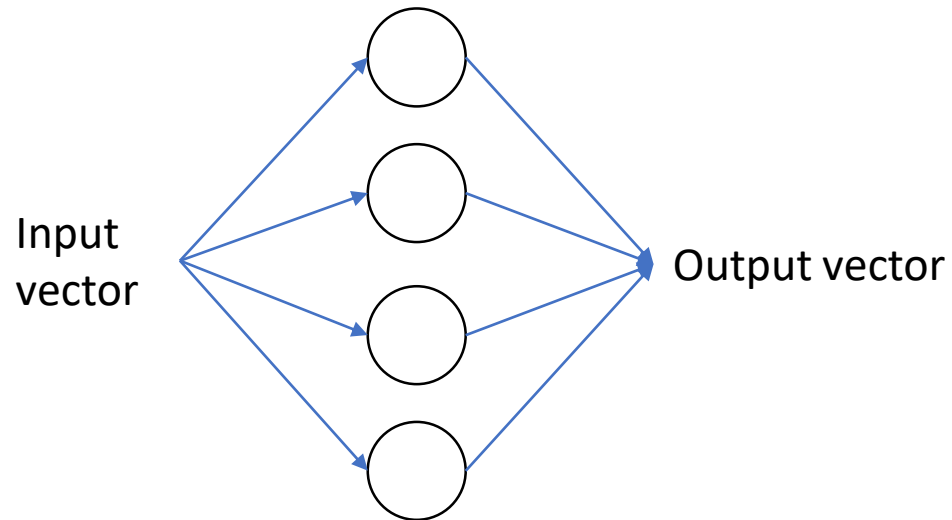
Does anyone pass along the weight without an activation function?

No – this is linear chaining.



Does anyone pass along the weight without an activation function?

No – this is linear chaining.



Are there other activation functions?

Yes, many.

As long as:

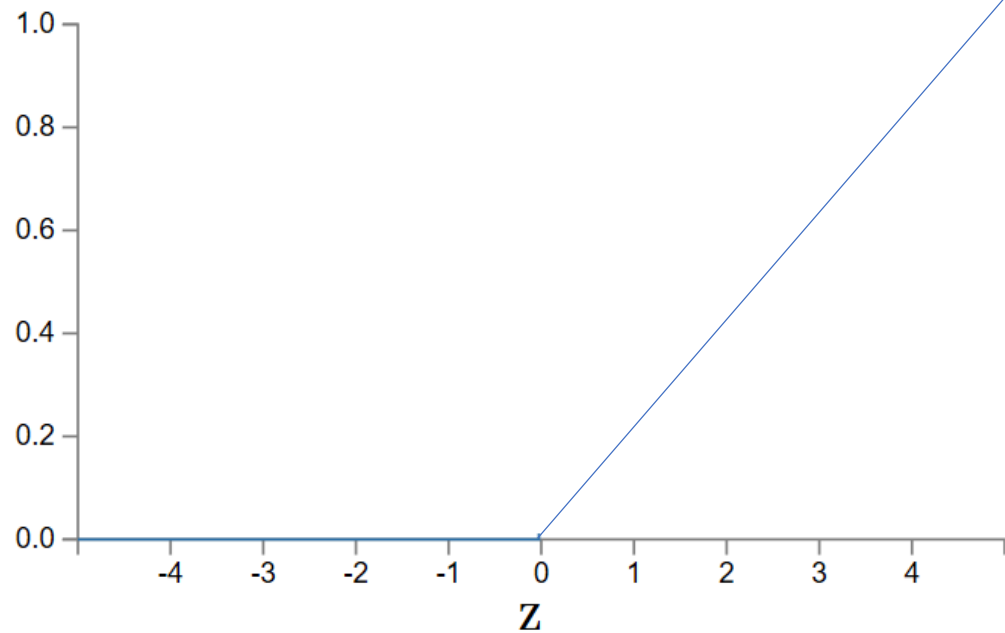
- Activation function $s(z)$ is well-defined as $z \rightarrow -\infty$ and $z \rightarrow \infty$
- These limits are different

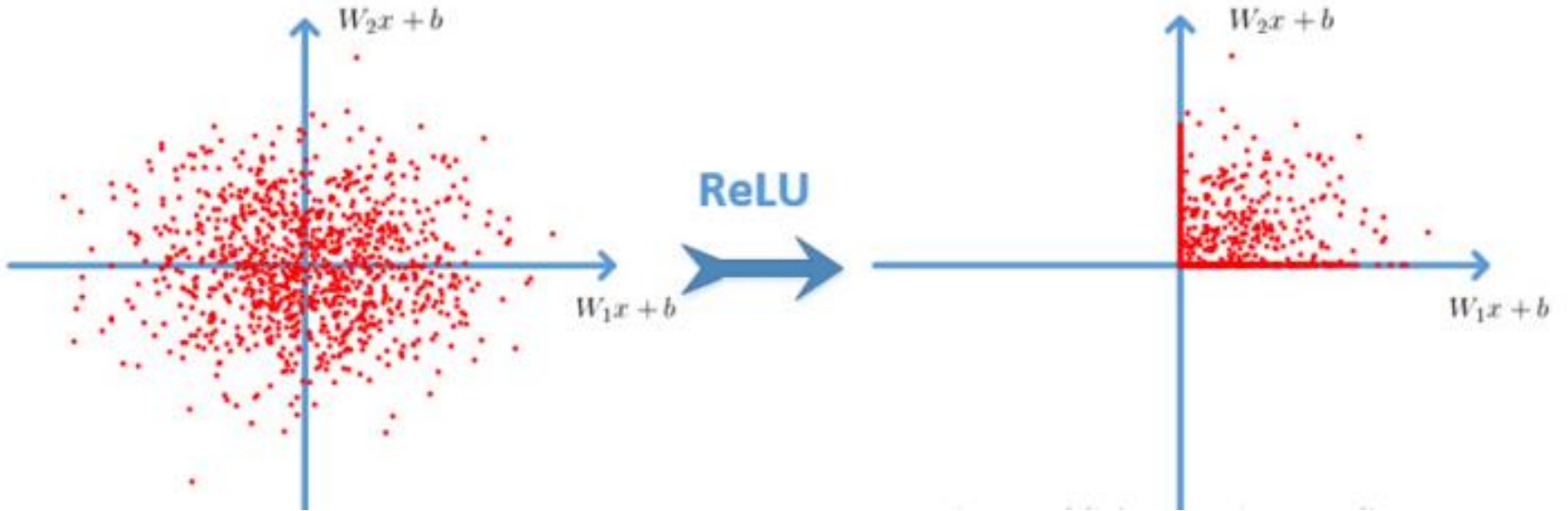
Then *we can make a step!* [Think visual proof]

It can be shown that it is universal for function approximation.

Activation functions: Rectified Linear Unit

- ReLU $f(x) = \max(0, x)$.

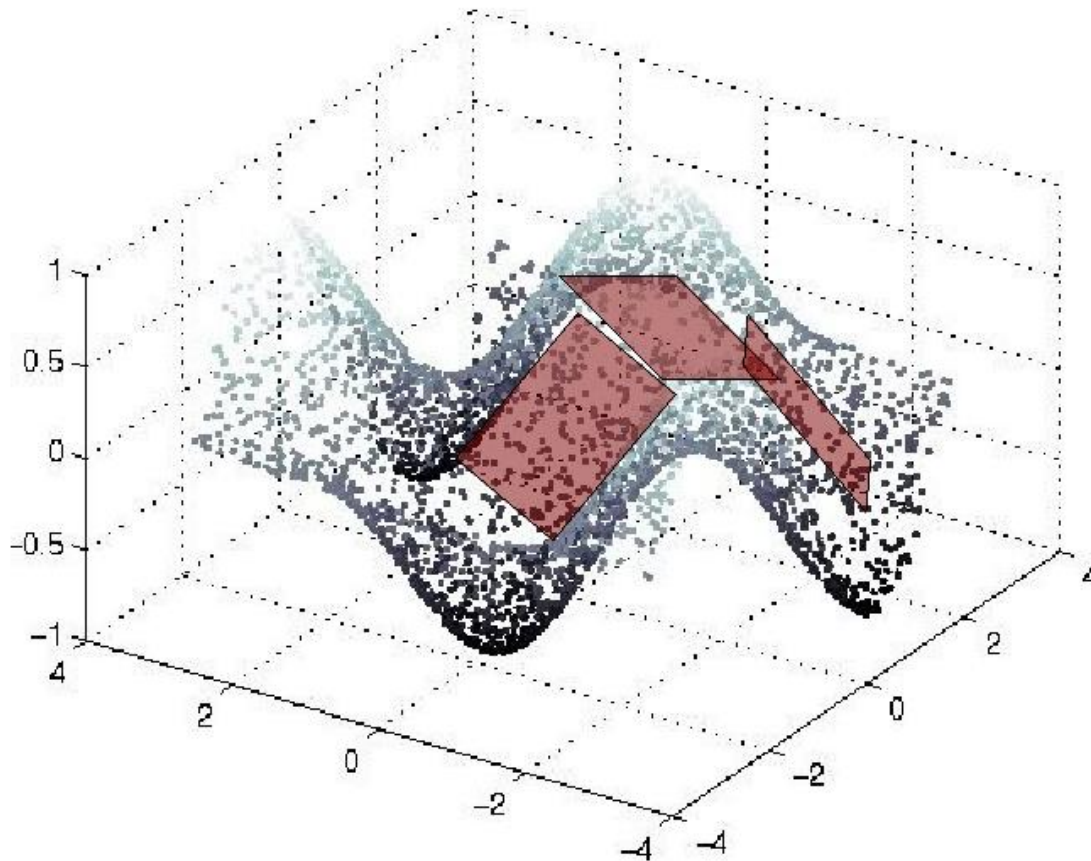




Rectified Linear Unit

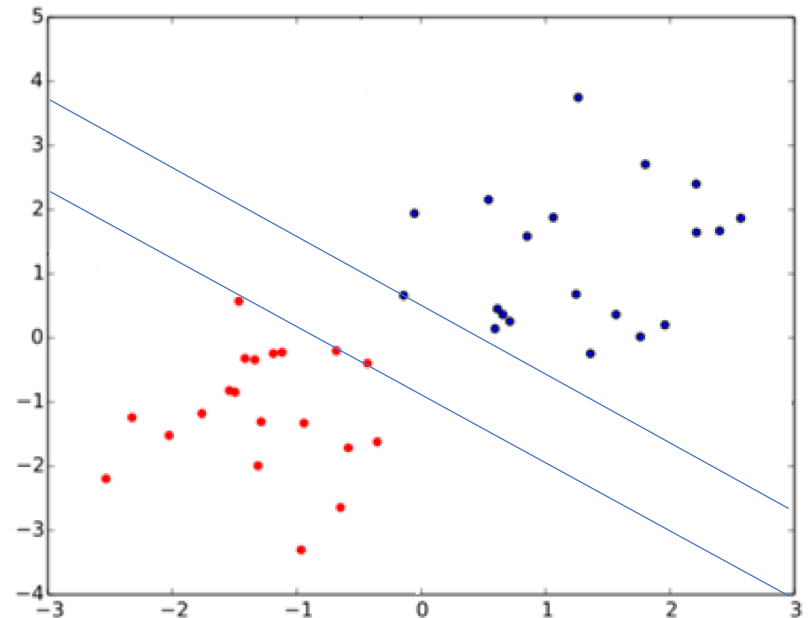
Question: What do ReLU layers accomplish?

Answer: Piece-wise linear tiling: mapping is locally linear.



What is the relationship between SVMs and perceptrons?

SVMs attempt to learn the support vectors which maximize the margin between classes.



What is the relationship between SVMs and perceptrons?

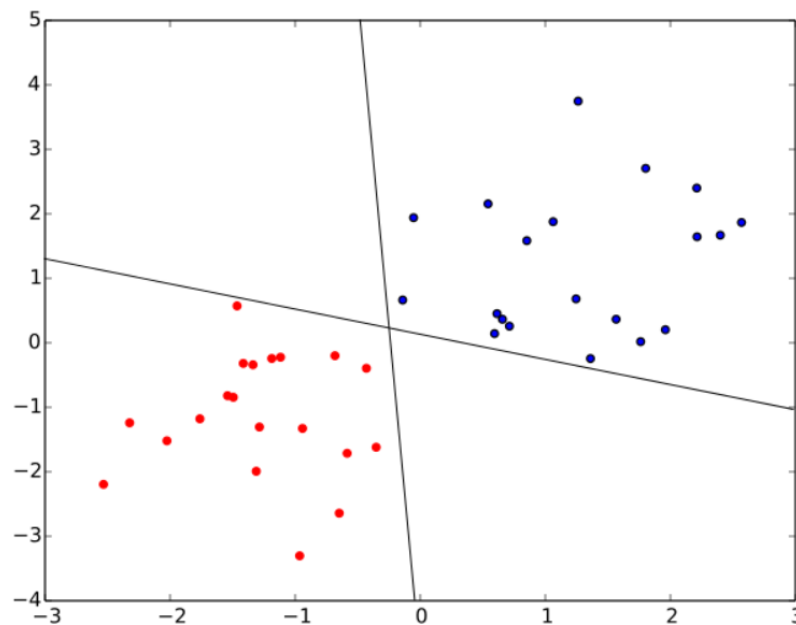
SVMs attempt to learn the support vectors which maximize the margin between classes.

A perceptron does not.

Both of these perceptron classifiers are equivalent.

‘Perceptron of optimal stability’ is used in SVM:

Perceptron
+ optimal stability
+ kernel trick
= *foundations of SVM*



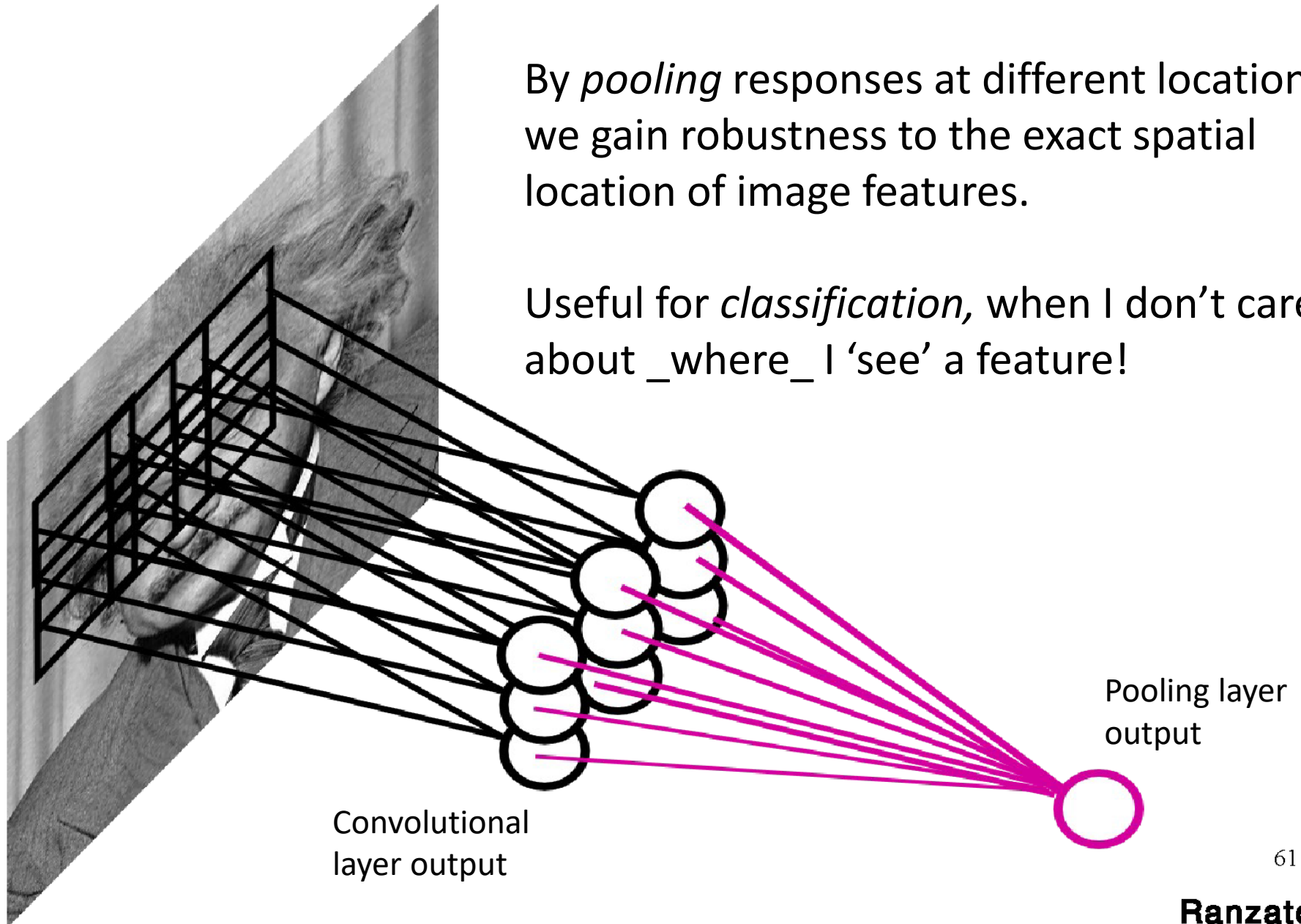
Why is pooling useful again?

What kinds of pooling operations might we consider?

Pooling Layer

By *pooling* responses at different locations, we gain robustness to the exact spatial location of image features.

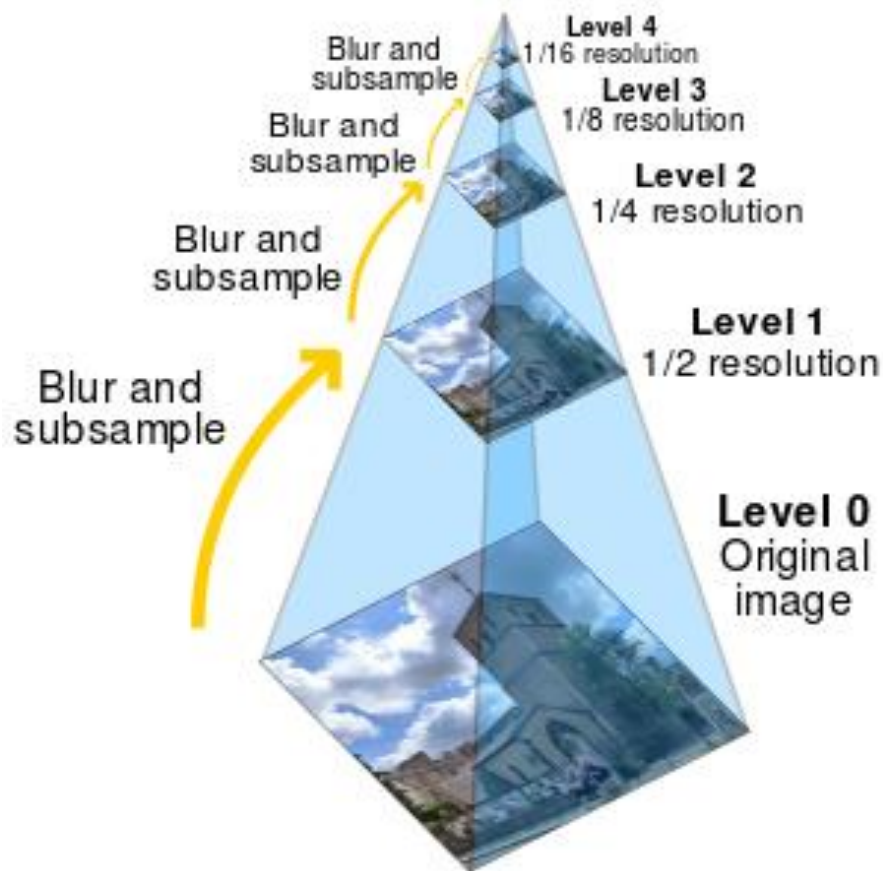
Useful for *classification*, when I don't care about *_where_* I 'see' a feature!



Pooling is similar to downsampling

...but on feature maps,
not the input!

...except sometimes we
don't want to blur,
as other functions might
be better for classification.



Pooling Layer: Examples

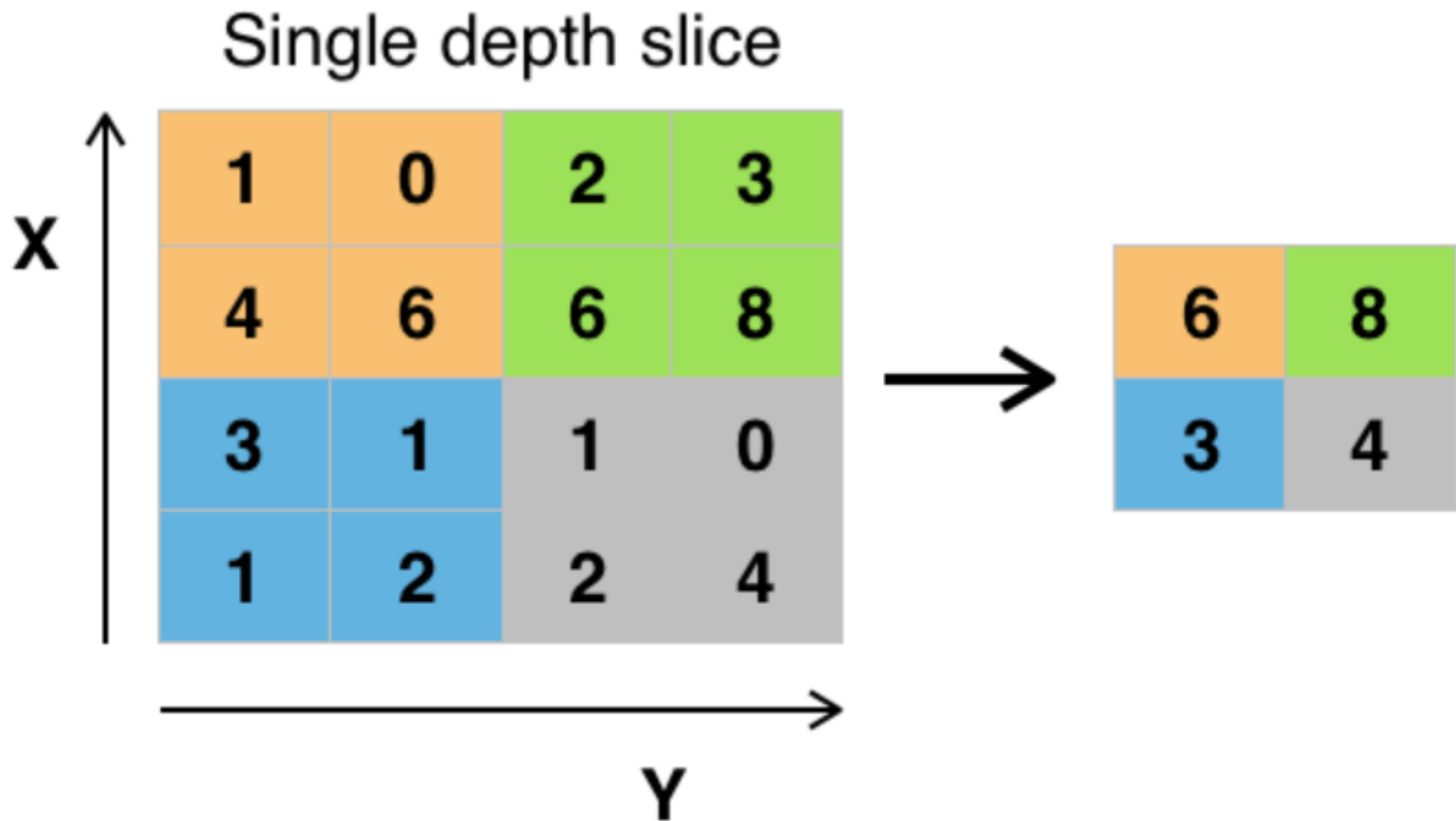
Max-pooling:

$$h_j^n(x, y) = \max_{\bar{x} \in N(x), \bar{y} \in N(y)} h_j^{n-1}(\bar{x}, \bar{y})$$

Average-pooling:

$$h_j^n(x, y) = 1/K \sum_{\bar{x} \in N(x), \bar{y} \in N(y)} h_j^{n-1}(\bar{x}, \bar{y})$$

Max pooling



OK, so what about invariances?

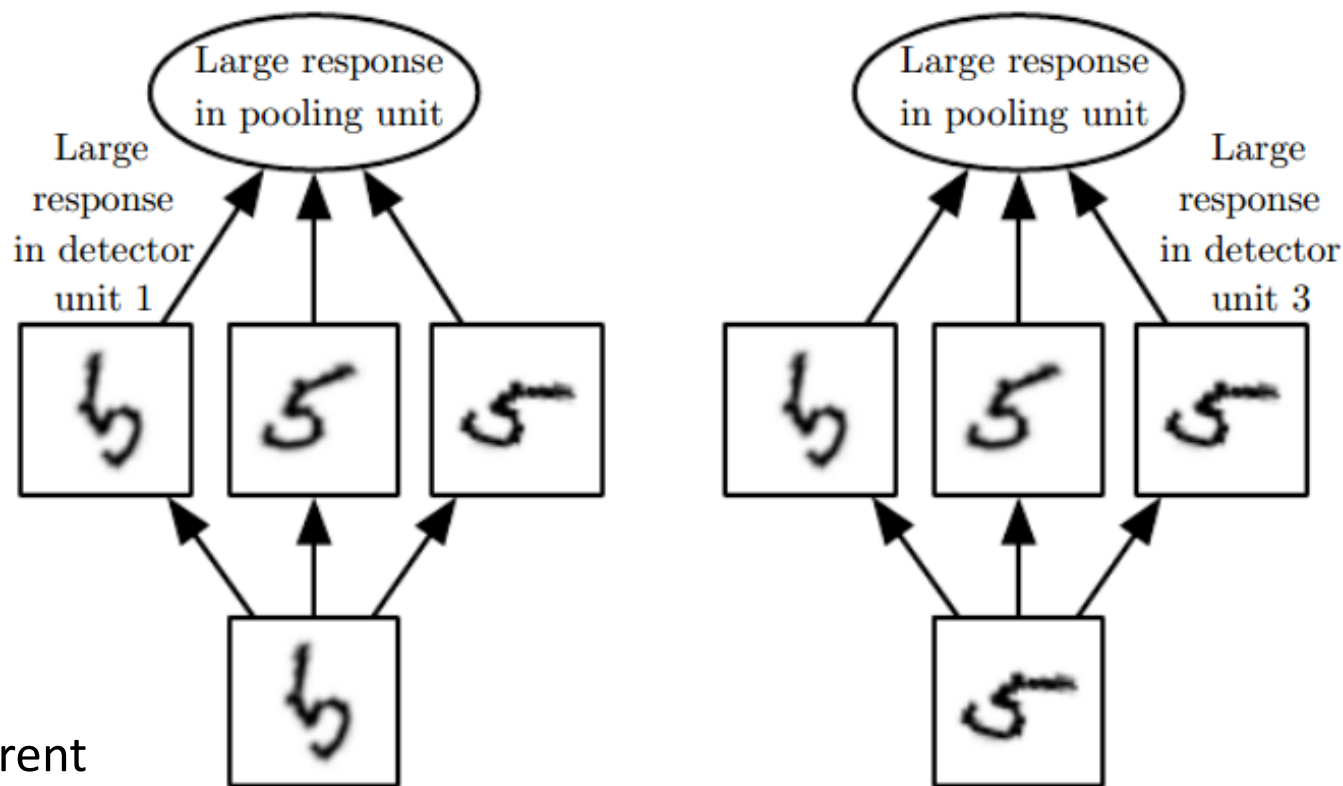
What about translation, scale, rotation?

Convolution is translation *equivariant* ('shift-equivariant') – we could shift the image and the kernel would give us a corresponding ('equal') shift in the feature map.

But! If we rotated or scaled the input, the same kernel would give a different response.

Pooling lets us aggregate (avg) or pick from (max) responses, but the kernels themselves must be trained and so learn to activate on scaled or rotated instances of the object.

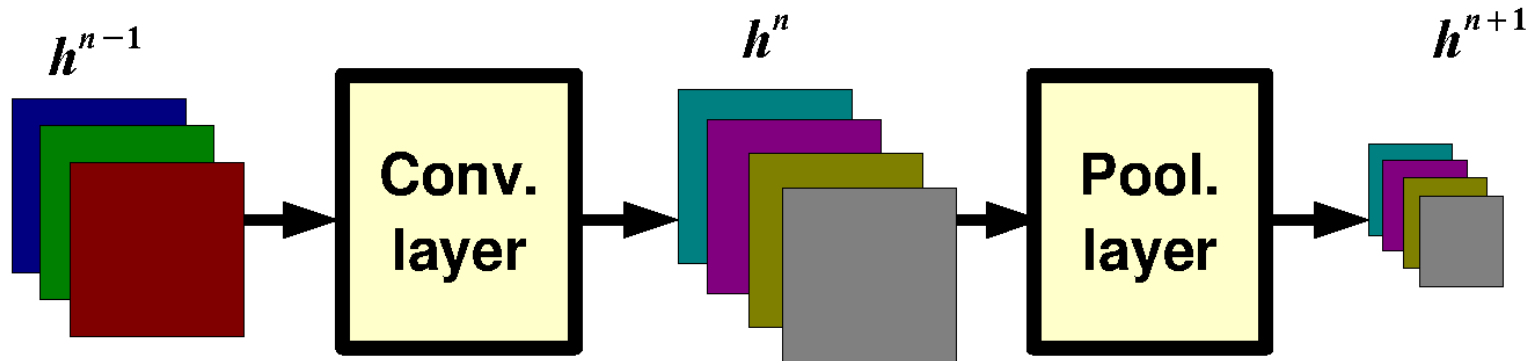
If we max pooled over depth (# kernels)...



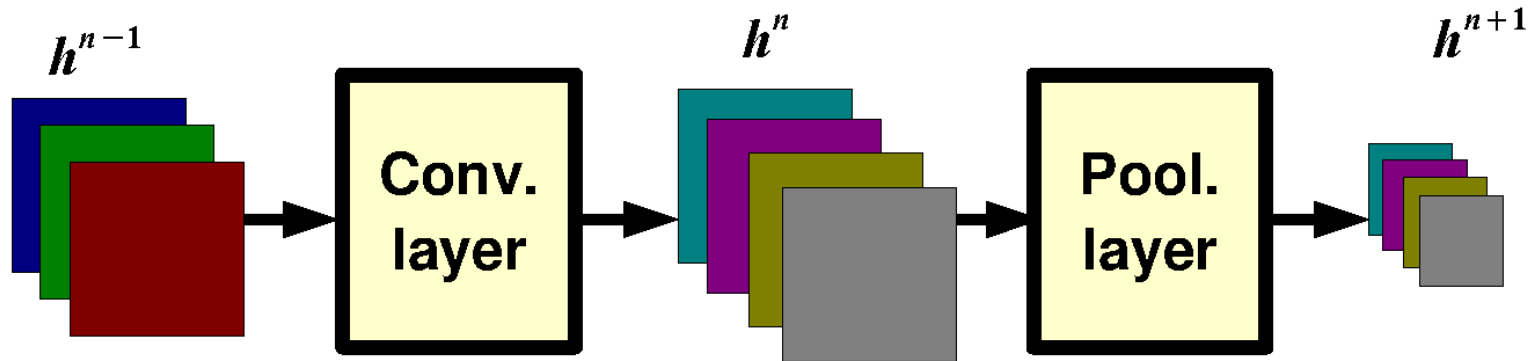
Three different kernels trained to fire on different rotations of '5'.

Fig 9.9, Goodfellow et al. [the book]

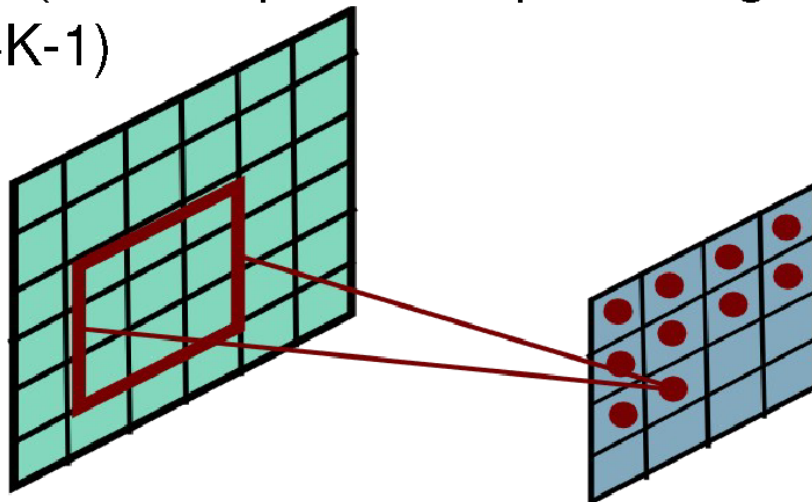
Pooling Layer: Receptive Field Size



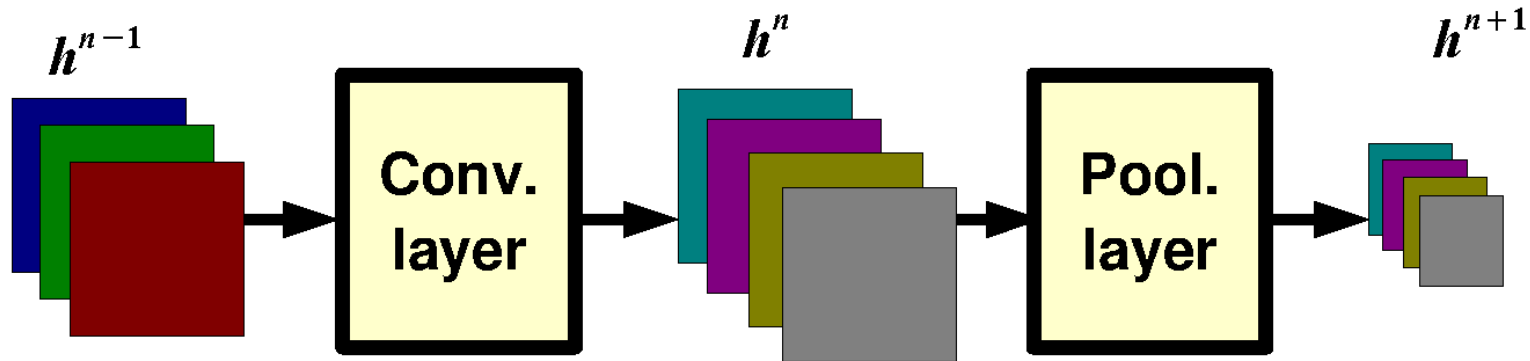
Pooling Layer: Receptive Field Size



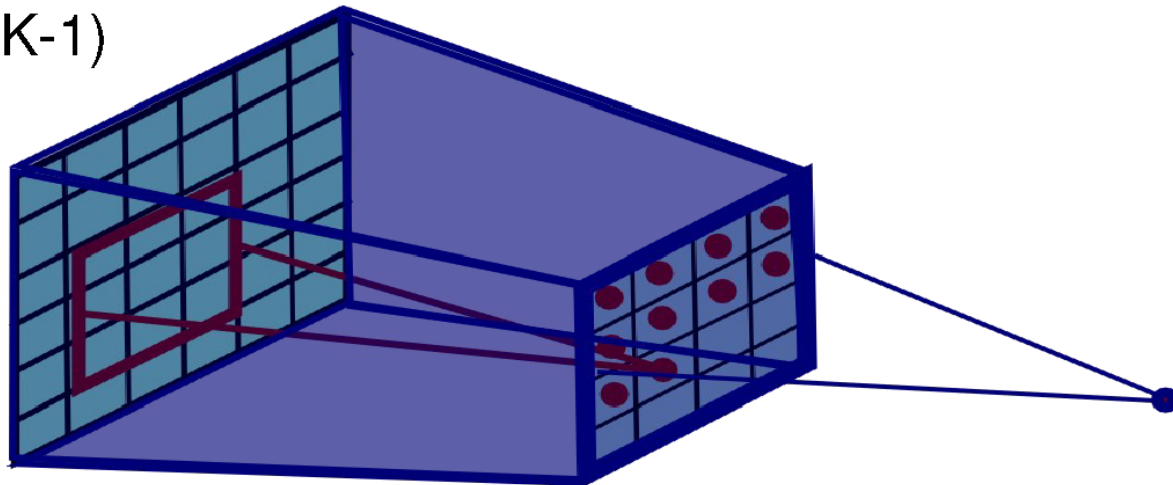
If convolutional filters have size $K \times K$ and stride 1, and pooling layer has pools of size $P \times P$, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: $(P+K-1) \times (P+K-1)$



Pooling Layer: Receptive Field Size



If convolutional filters have size $K \times K$ and stride 1, and pooling layer has pools of size $P \times P$, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: $(P+K-1) \times (P+K-1)$



I've heard about many more terms of jargon!

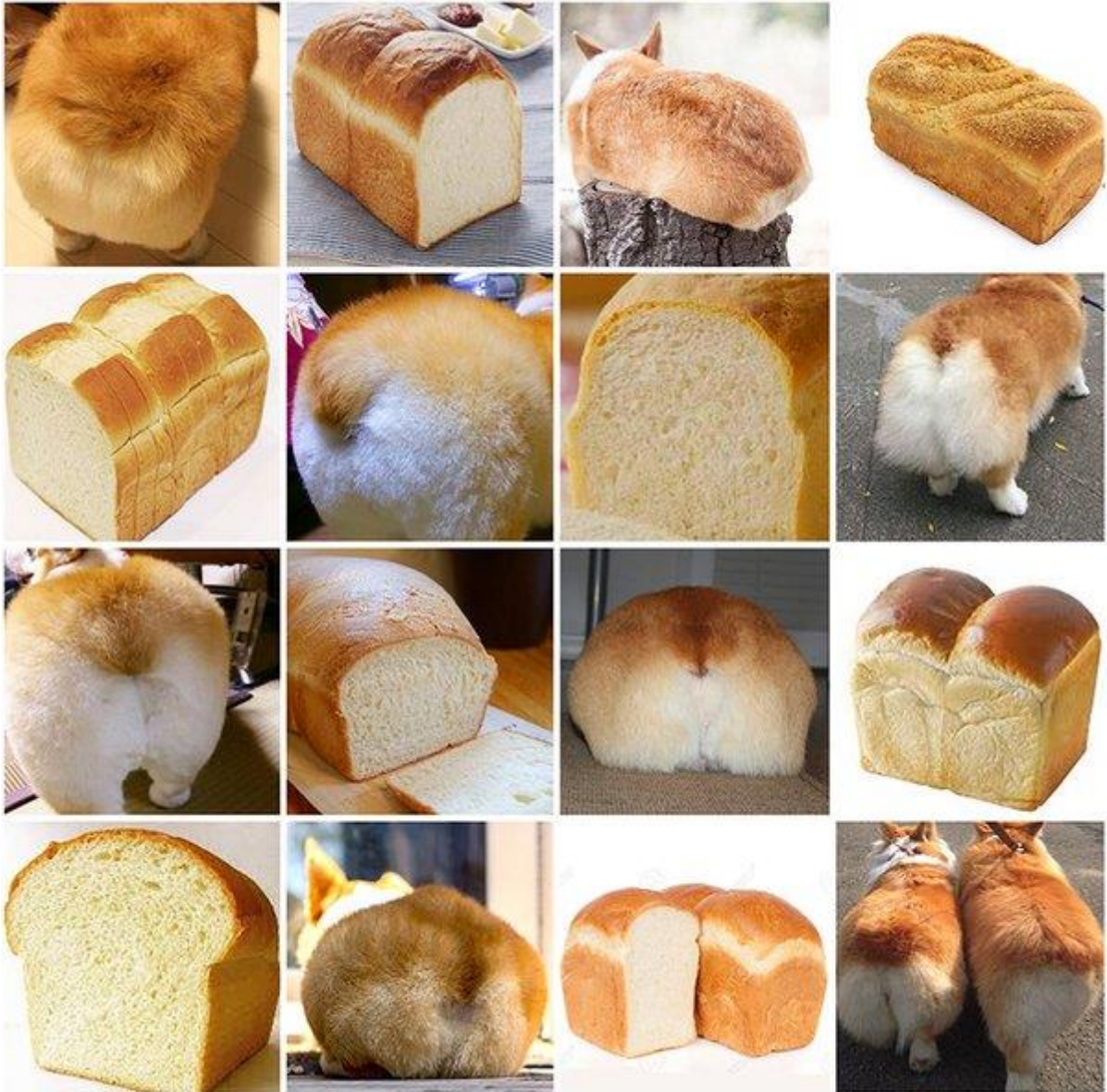
Skip connections

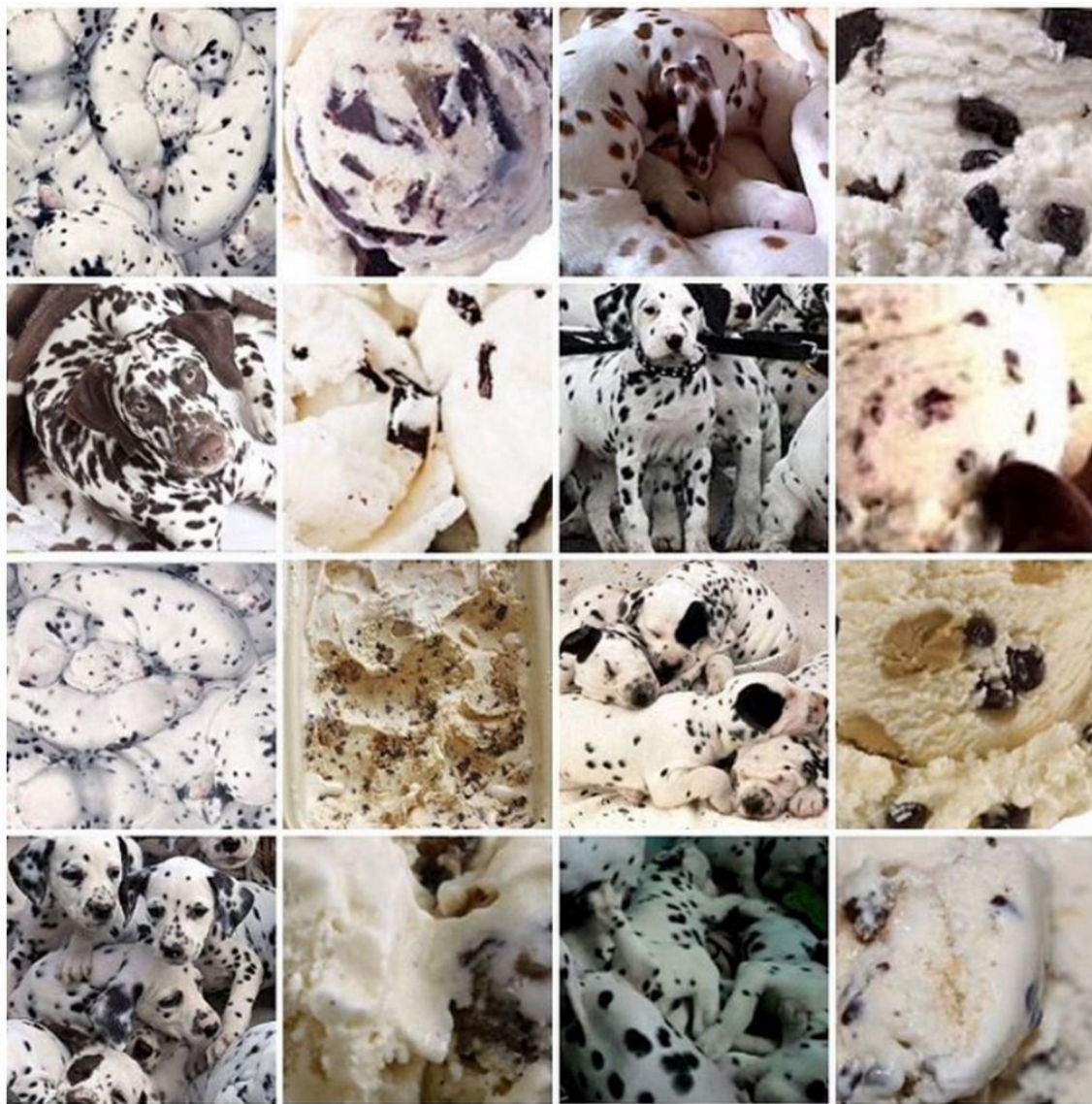
Residual connections

Batch normalization

...we'll get to these in a little while.



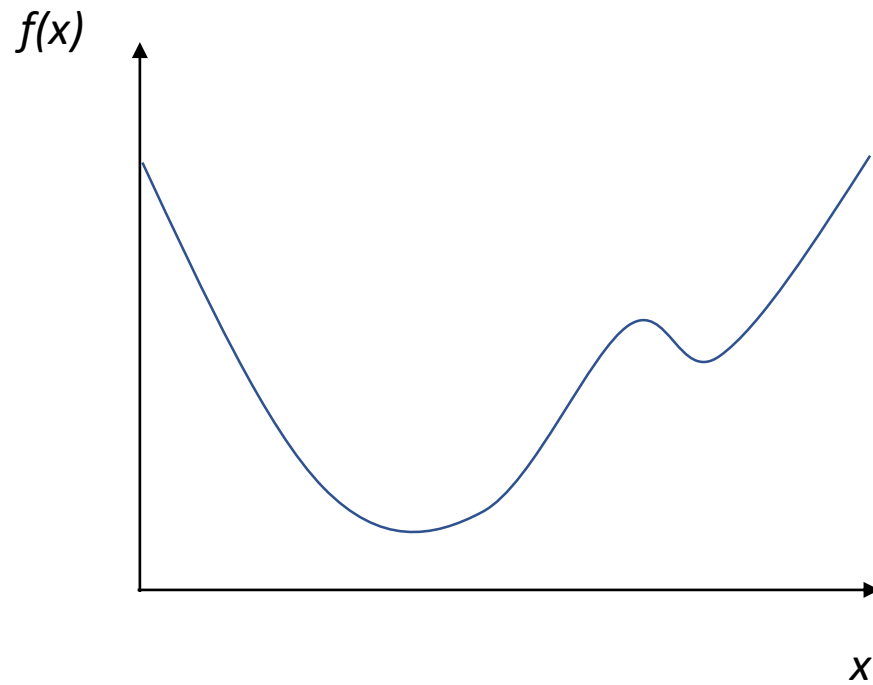




Training Neural Networks

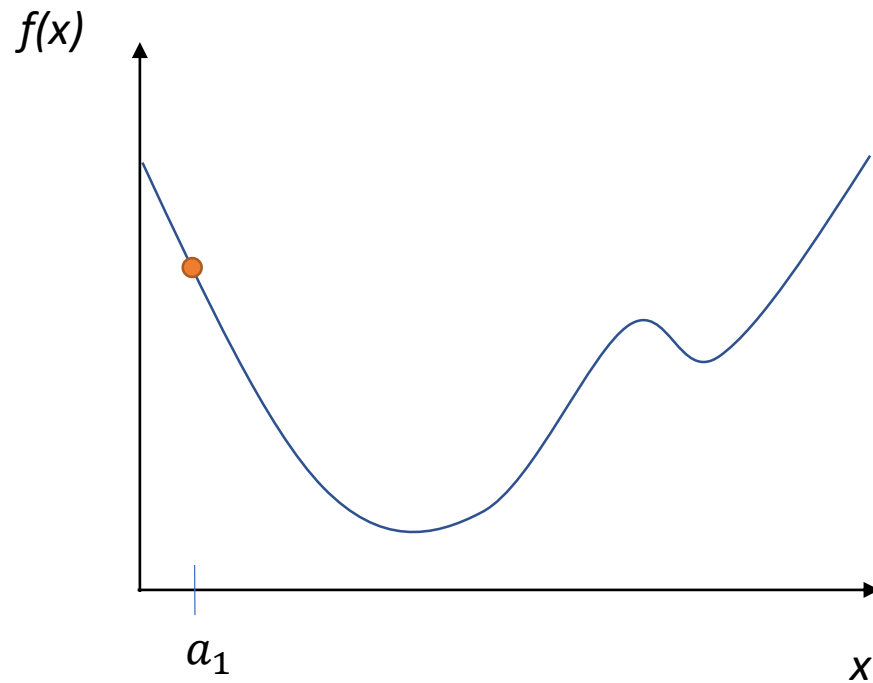
Learning the weight matrices W

Gradient descent



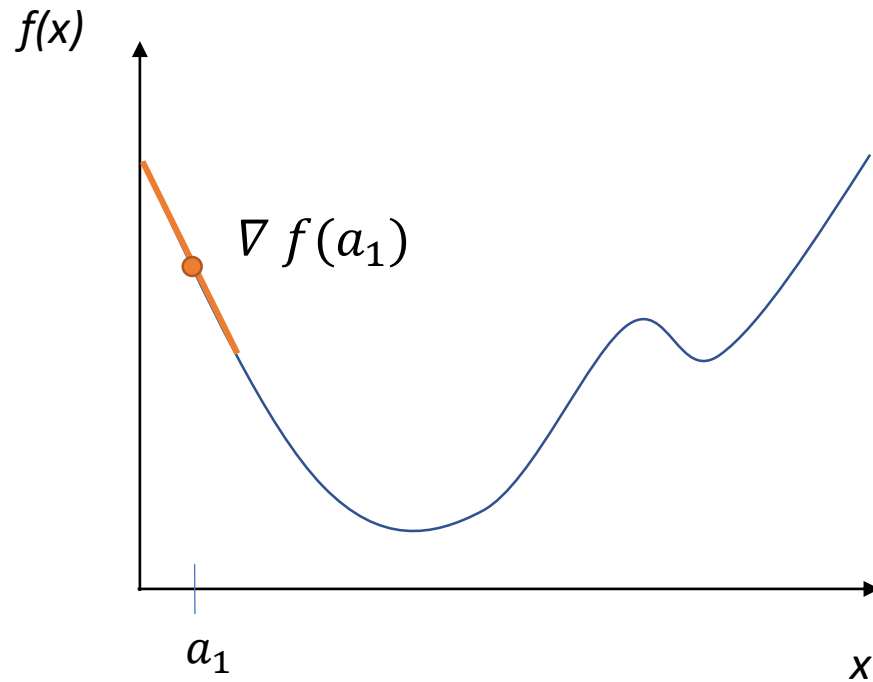
General approach

Pick random starting point.



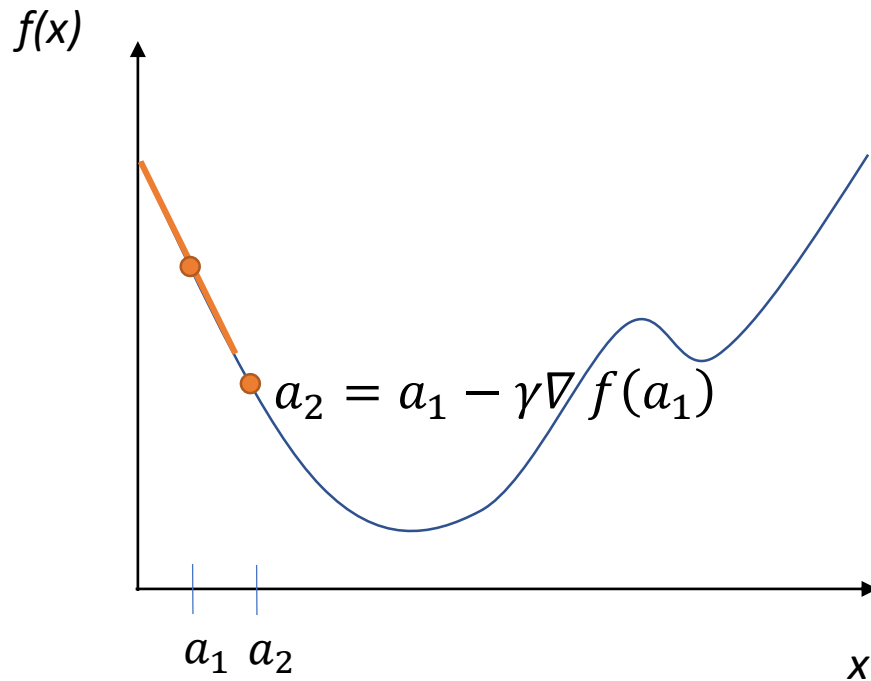
General approach

Compute gradient at point (analytically or by finite differences)



General approach

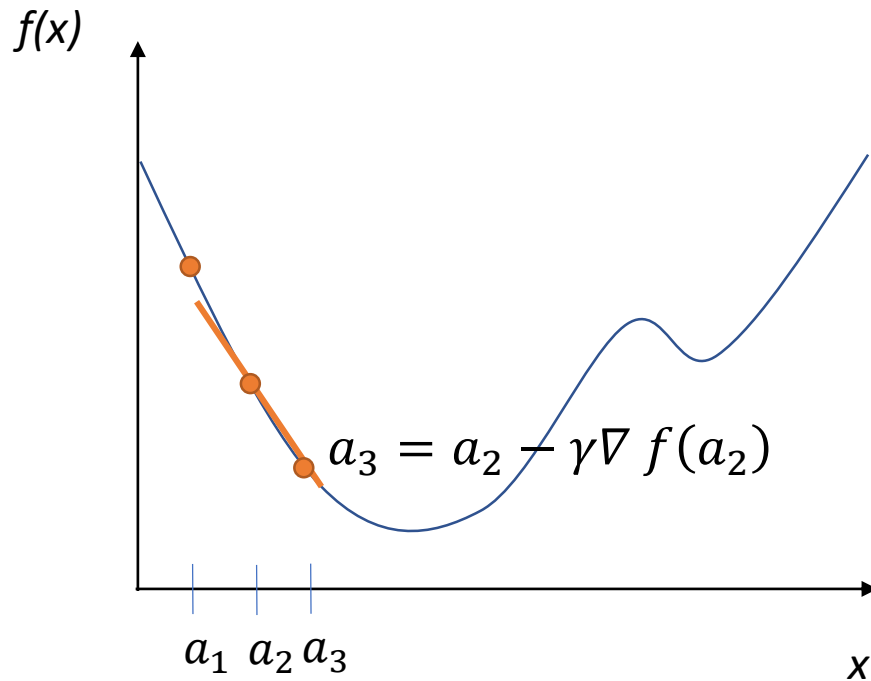
Move along parameter space in direction of negative gradient



γ = amount to move
= *learning rate*

General approach

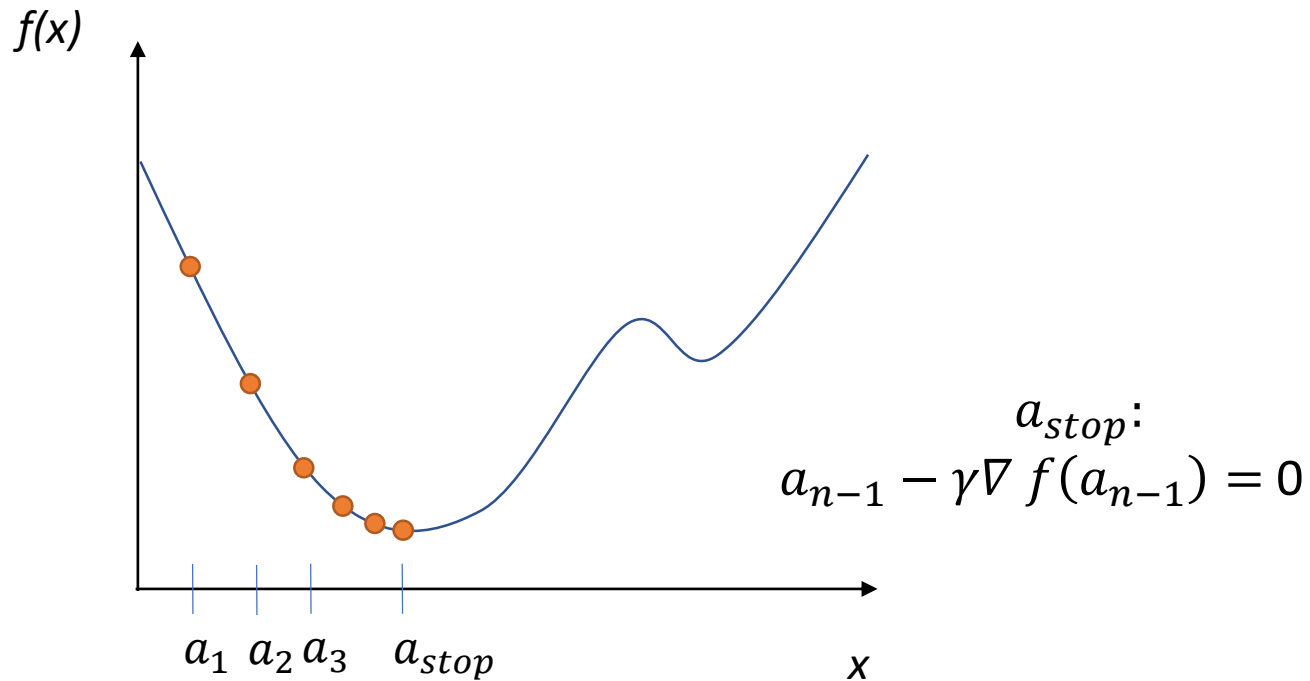
Move along parameter space in direction of negative gradient.



γ = amount to move
= *learning rate*

General approach

Stop when we don't move any more.

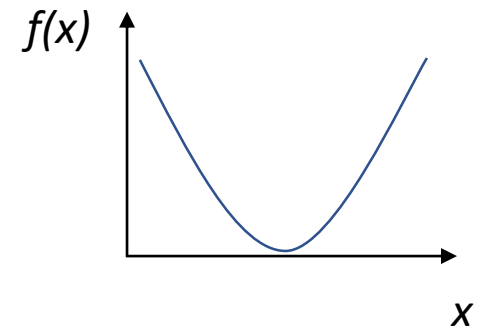


Gradient descent

Optimizer for functions.

Guaranteed to find optimum for convex functions.

- Non-convex = find *local* optimum.
- Most vision problems aren't convex.



Works for multi-variate functions.

- Need to compute matrix of *partial derivatives* (“Jacobian”)

Why would I use this over Least Squares?

If my function is convex,
why can't I just use linear least squares?

$$A\mathbf{x} - \mathbf{b} = 0$$

$$F(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2.$$

$$\nabla F(\mathbf{x}) = 2A^T (A\mathbf{x} - \mathbf{b}).$$

Analytic solution = normal equations $x = (A^T A)^{-1} A^T b$

You can, yes.

Why would I use this over Least Squares?

But now imagine that I have 1,000,000 data points.

Matrices are huge.

Even for convex functions, gradient descent allows me to iteratively solve the solution without requiring very large matrices.

We'll see how.

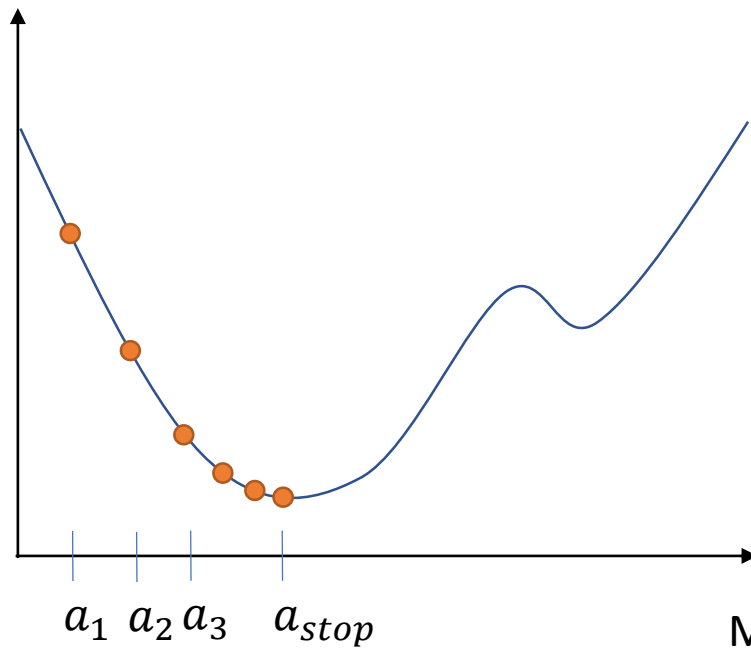
Train NN with Gradient Descent

- $x^i, y^i = n$ training examples
- $f(\mathbf{x})$ = feed forward neural network
- $L(\mathbf{x}, y; \boldsymbol{\theta})$ = some *loss function*

Loss function measures how ‘good’ our network is at classifying the training examples wrt. the parameters of the model (the perceptron weights).

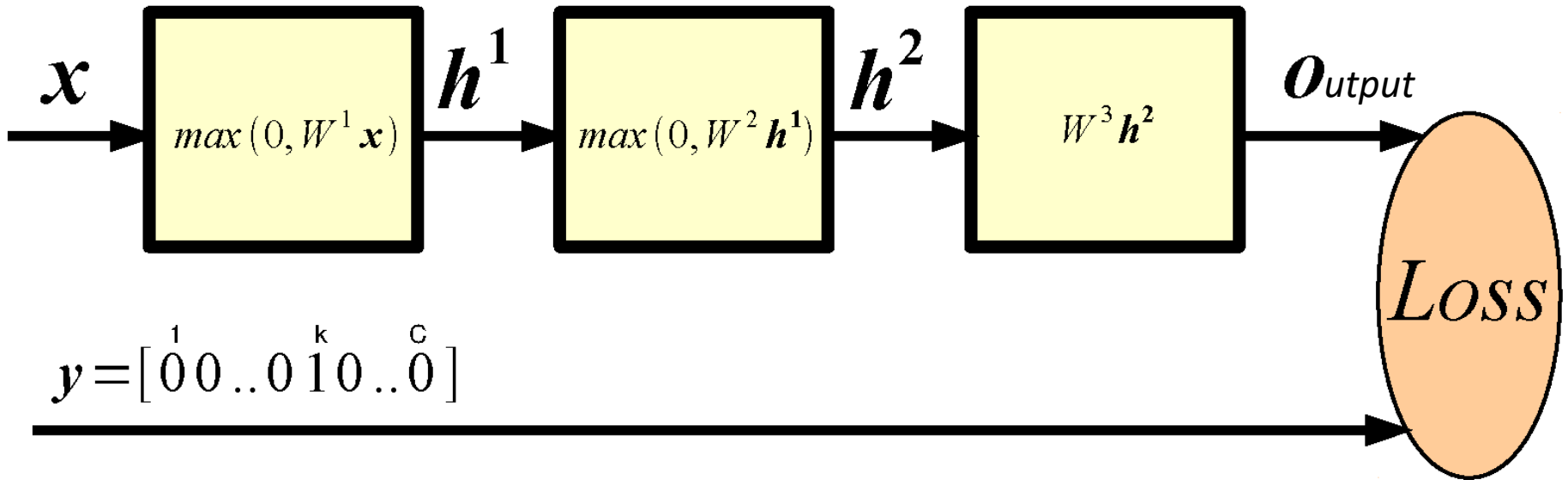
Train NN with Gradient Descent

Loss function
(Evaluate NN
on training data)



Model parameters
(perceptron weights)

How Good is a Network?



What is an appropriate loss?

What if we define an output threshold on detection?

- Classification: compare training class to output class
- Zero-one loss L (per class)

$y = \text{true label}$

$\hat{y} = \text{predicted label}$

$$L(\hat{y}, y) = I(\hat{y} \neq y),$$

Is it good?

- Nope – it's a step function.
- I need to compute the *gradient* of the loss.
- This loss is not differentiable, and 'flips' easily.

Classification as probability

Special function on last layer - 'Softmax' $\sigma()$:

"Squashes" a C -dimensional vector \mathbf{O} of arbitrary real values to a C -dimensional vector $\sigma(\mathbf{O})$ of real values in the range $(0, 1)$ that add up to 1.

Turns the output into a probability distribution on classes.

$$p(c_k = 1 | \mathbf{x}) = \frac{e^{o_k}}{\sum_{j=1}^C e^{o_j}}$$

Classification as probability

Softmax example:

“Squashes” a C -dimensional vector \mathbf{O} of arbitrary real values to a C -dimensional vector $\sigma(\mathbf{O})$ of real values in the range $(0, 1)$ that add up to 1.

Turns the output into a probability distribution on classes.

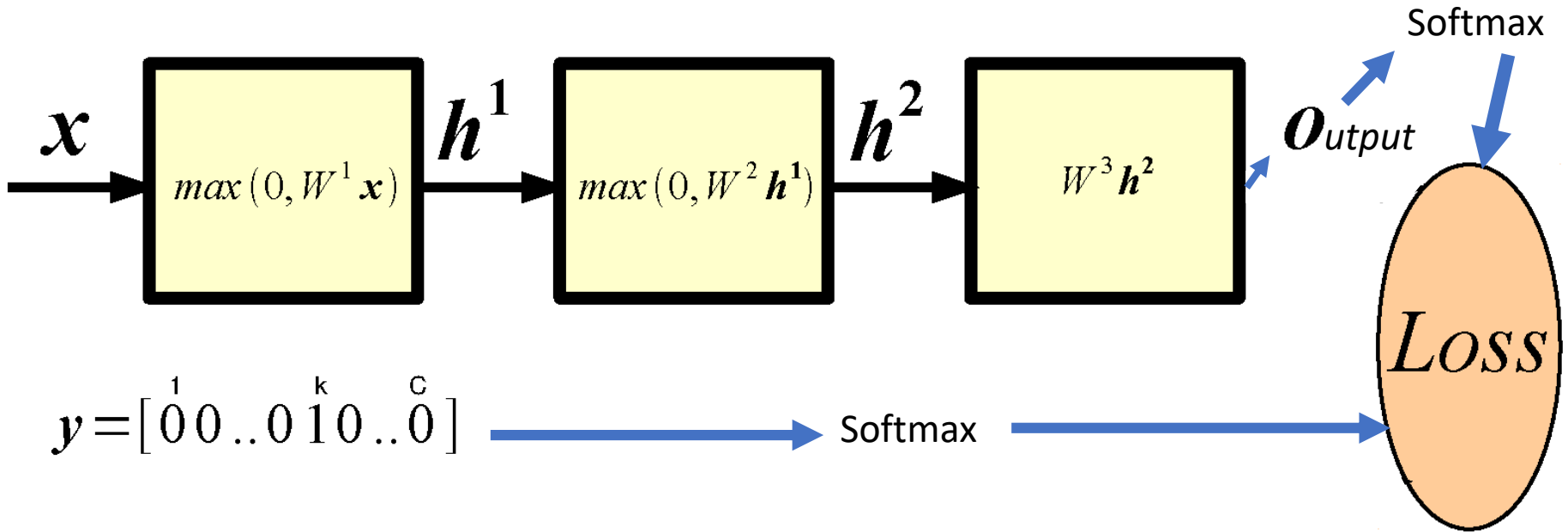
Output from perceptron layer
'distance from class boundary'

$$\mathbf{O} = [2.0, 0.7, 0.2, -0.3, -0.6, -2.5]$$

$$p(c_k=1|\mathbf{x}) = \frac{e^{o_k}}{\sum_{j=1}^C e^{o_j}}$$

$$\sigma(\mathbf{O}) = [0.616, 0.168, 0.102, 0.061, 0.046, 0.007]$$

How Good is a Network?



Probability of class k given input (softmax):

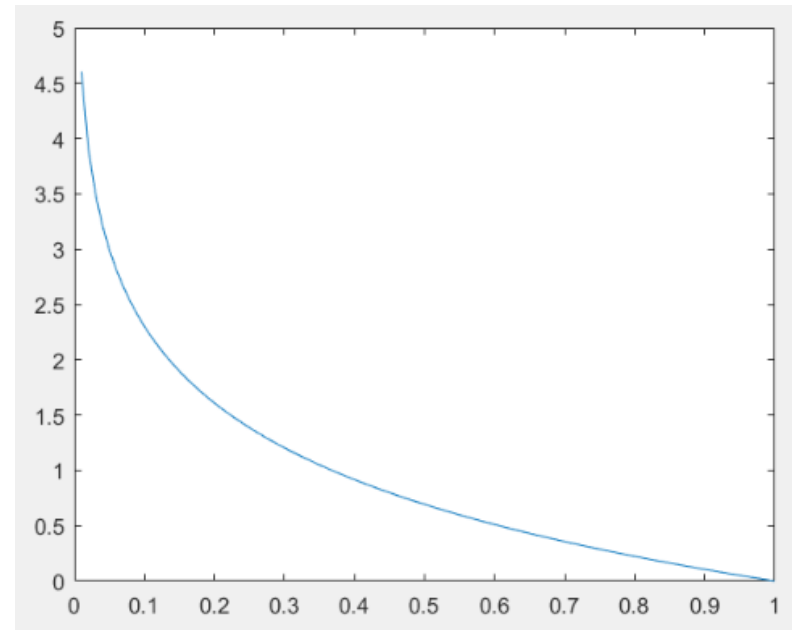
$$p(c_k=1|\mathbf{x}) = \frac{e^{o_k}}{\sum_{j=1}^C e^{o_j}}$$

Cross-entropy loss function

Negative log-likelihood

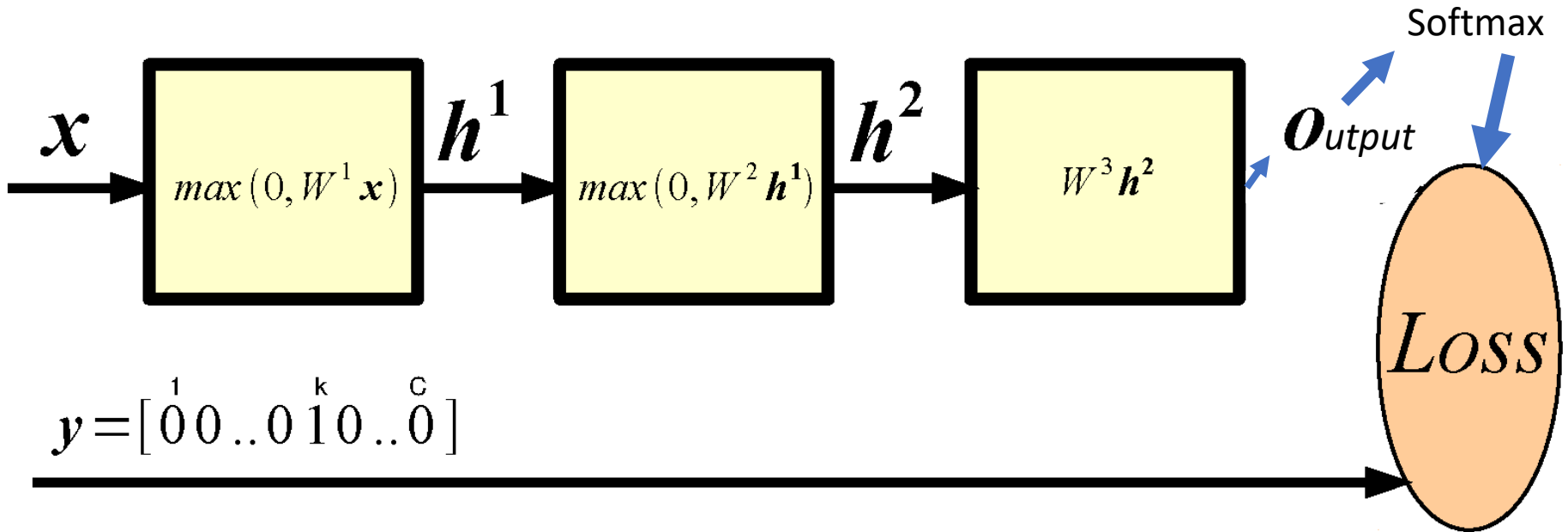
$$L(\mathbf{x}, y; \boldsymbol{\theta}) = - \sum_j y_j \log p(c_j | \mathbf{x})$$

- Measures difference between predicted and training probability distributions (see Project 4 for more details)
- Is it a good loss?
 - Differentiable
 - Cost decreases as probability increases



$p(c_j | \mathbf{x})$

How Good is a Network?



Probability of class k given input (softmax):

$$p(c_k = 1 | \mathbf{x}) = \frac{e^{o_k}}{\sum_{j=1}^C e^{o_j}}$$

(Per-sample) **Loss**; e.g., negative log-likelihood (good for classification of small number of classes):

$$L(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}) = - \sum_j y_j \log p(c_j | \mathbf{x})$$

Training

Learning consists of minimizing the loss wrt. parameters over the whole training set.

$$\boldsymbol{\theta}^* = \mathit{arg\ min}_{\boldsymbol{\theta}} \sum_{n=1}^P L(\mathbf{x}^n, y^n; \boldsymbol{\theta})$$

Training

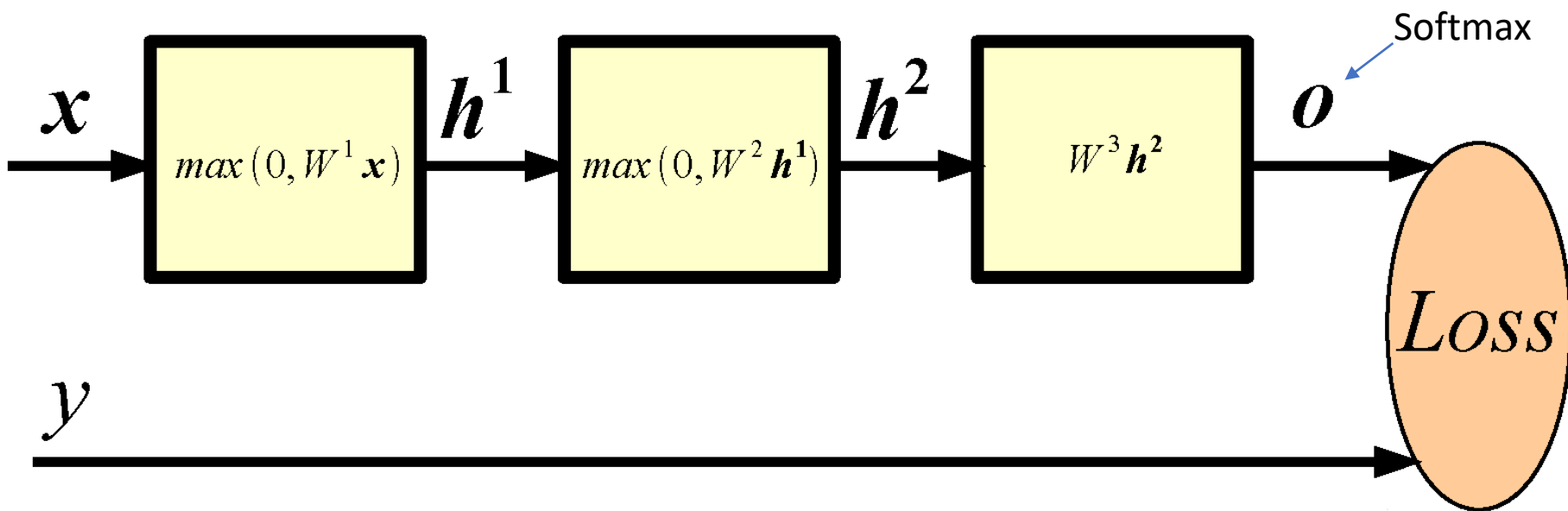
Learning consists of minimizing the loss wrt. parameters over the whole training set.

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{n=1}^P L(\mathbf{x}^n, y^n; \boldsymbol{\theta})$$

Question: How to minimize a complicated function of the parameters?

Answer: Chain rule, a.k.a. **Backpropagation!** That is the procedure to compute gradients of the loss w.r.t. parameters in a multi-layer neural network.

Key Idea: Wiggle To Decrease Loss

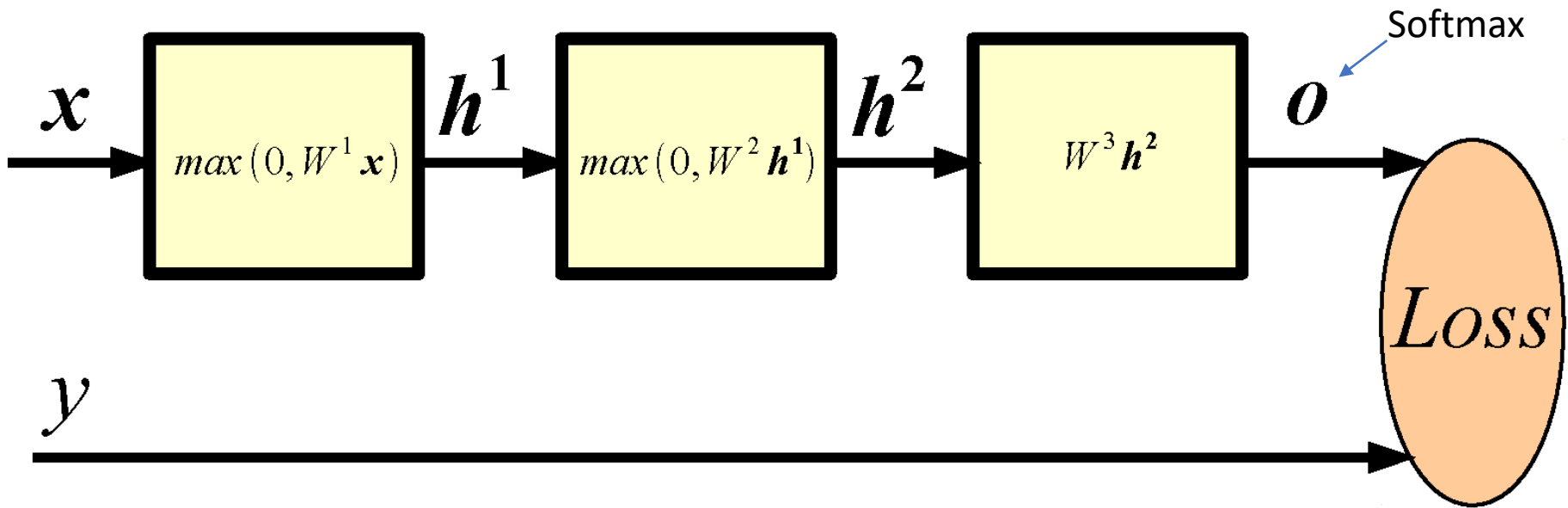


Let's say we want to decrease the loss by adjusting $W_{i,j}^1$.
We could consider a very small $\epsilon = 1e-6$ and compute:

$$L(\mathbf{x}, y; \boldsymbol{\theta})$$

$$L(\mathbf{x}, y; \boldsymbol{\theta} \setminus W_{i,j}^1, W_{i,j}^1 + \epsilon)$$

Key Idea: Wiggle To Decrease Loss



Let's say we want to decrease the loss by adjusting $W_{i,j}^1$.
We could consider a very small $\epsilon = 1e-6$ and compute:

$$L(\mathbf{x}, y; \boldsymbol{\theta})$$

$$L(\mathbf{x}, y; \boldsymbol{\theta} \setminus W_{i,j}^1, W_{i,j}^1 + \epsilon)$$

Then, update:

$$W_{i,j}^1 \leftarrow W_{i,j}^1 + \epsilon \operatorname{sgn}(L(\mathbf{x}, y; \boldsymbol{\theta}) - L(\mathbf{x}, y; \boldsymbol{\theta} \setminus W_{i,j}^1, W_{i,j}^1 + \epsilon))$$

Derivative of loss wrt. softmax

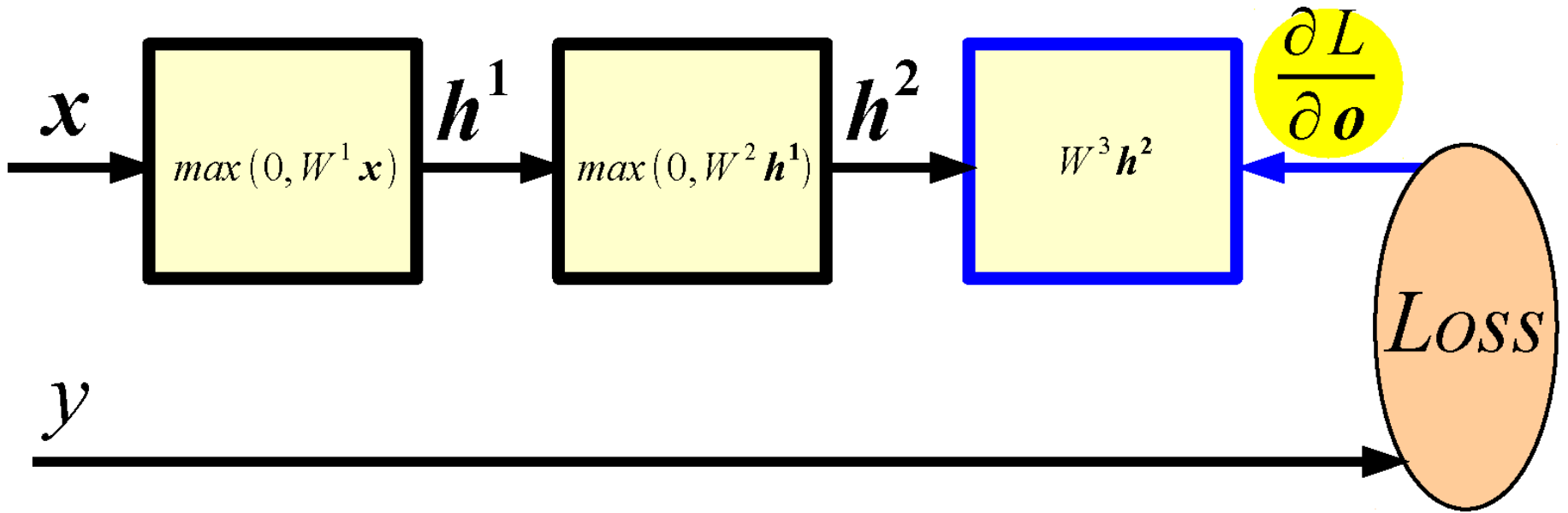
$$p(c_k = 1 | \mathbf{x}) = \frac{e^{o_k}}{\sum_j e^{o_j}}$$

$$L(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}) = - \sum_j y_j \log p(c_j | \mathbf{x}) \quad \mathbf{y} = [0 \overset{1}{0} \dots 0 \overset{k}{1} 0 \dots 0 \overset{c}{0}]$$

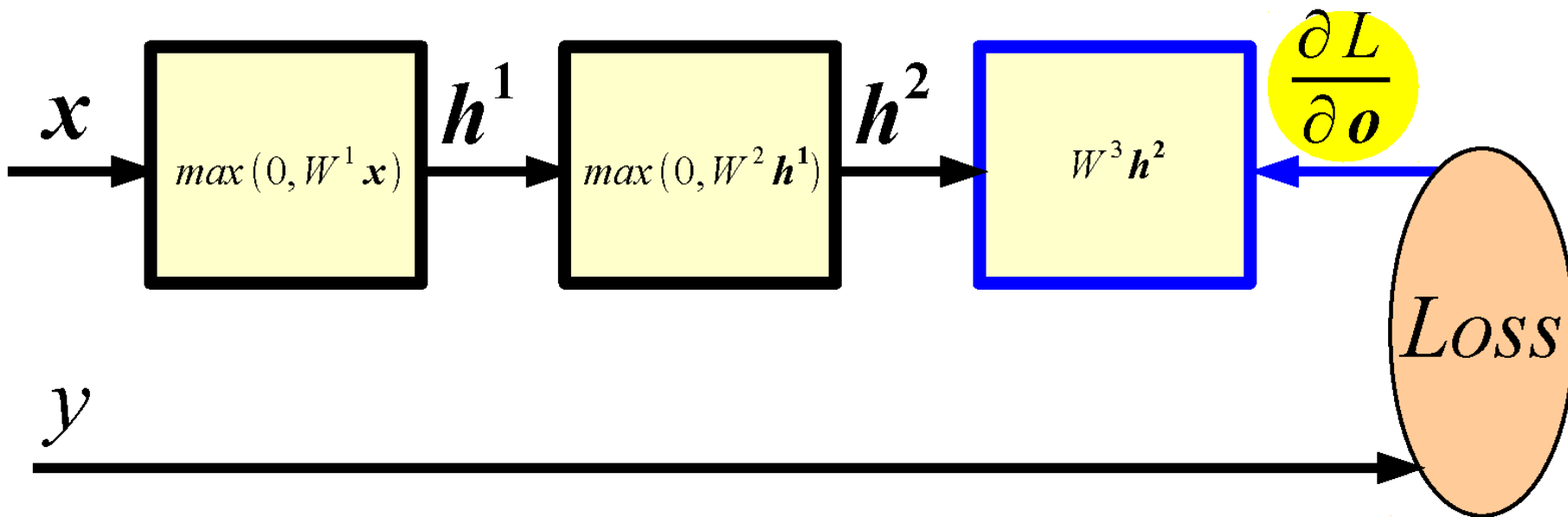
By substituting the first formula in the second, and taking the derivative w.r.t. \boldsymbol{o} we get:

$$\frac{\partial L}{\partial \boldsymbol{o}} = p(c | \mathbf{x}) - \mathbf{y}$$

Backward Propagation



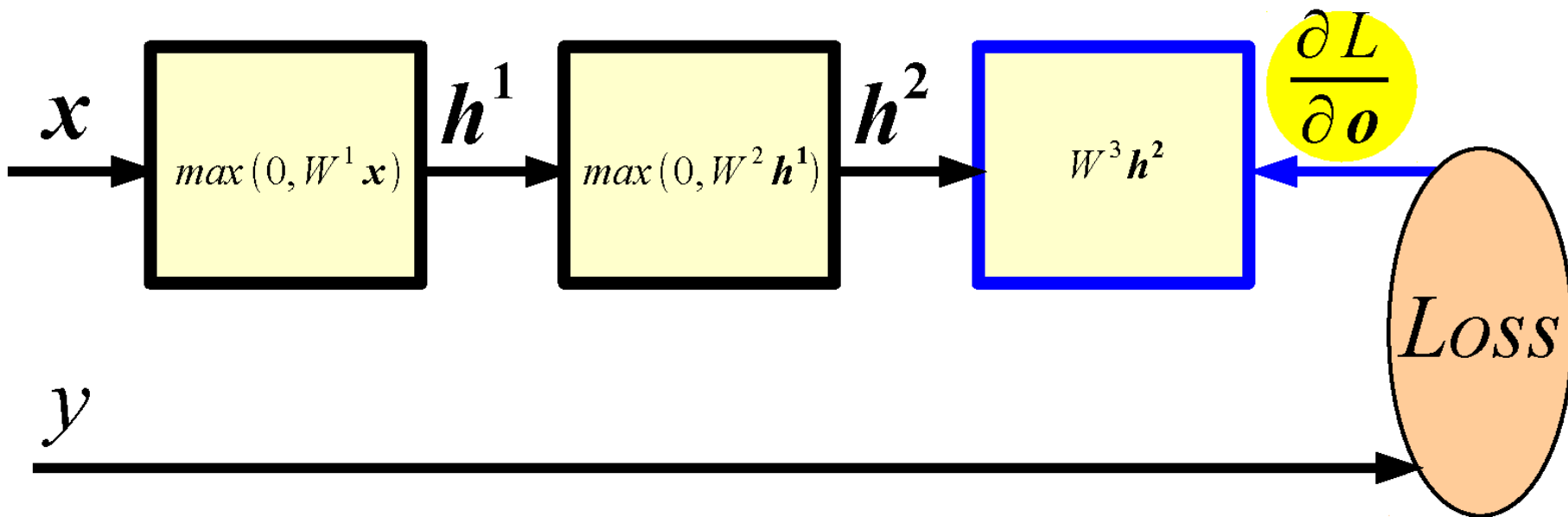
Backward Propagation



Given $\frac{\partial L}{\partial \mathbf{o}}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial W^3} \quad \leftarrow \text{Chain rule from calculus}$$

Backward Propagation

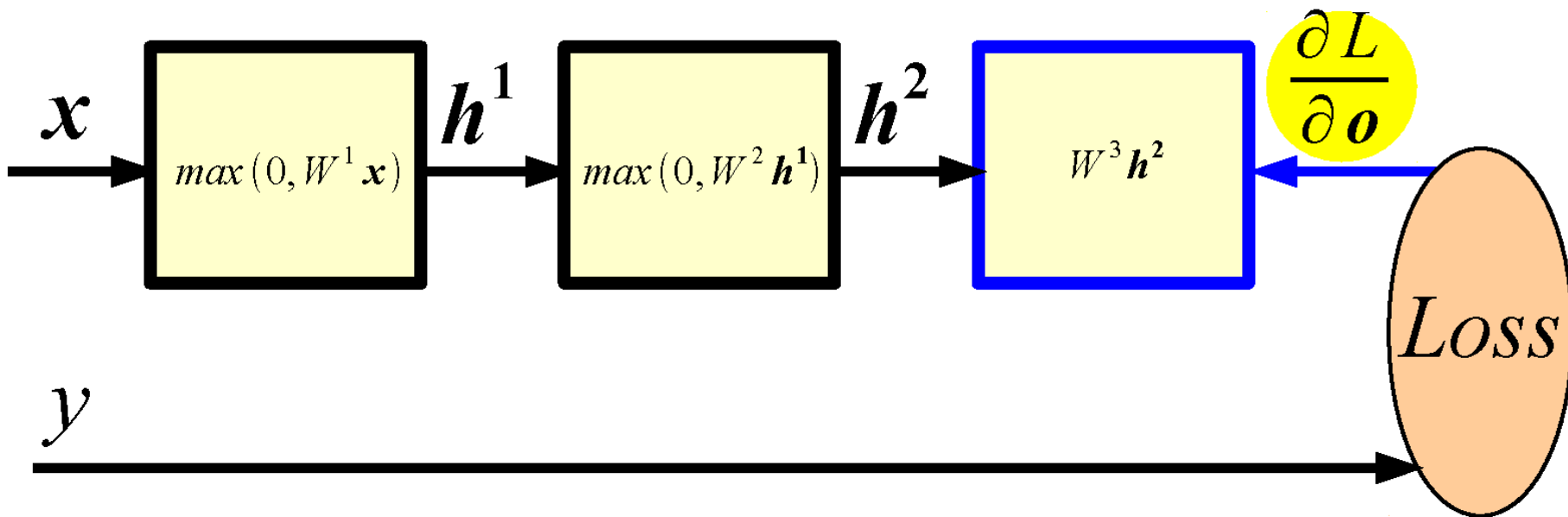


Given $\frac{\partial L}{\partial o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial o} \frac{\partial o}{\partial W^3}$$

$$\frac{\partial L}{\partial h^2} = \frac{\partial L}{\partial o} \frac{\partial o}{\partial h^2}$$

Backward Propagation



Given $\frac{\partial L}{\partial \mathbf{o}}$ and assuming we can easily compute the Jacobian of each module, we have:

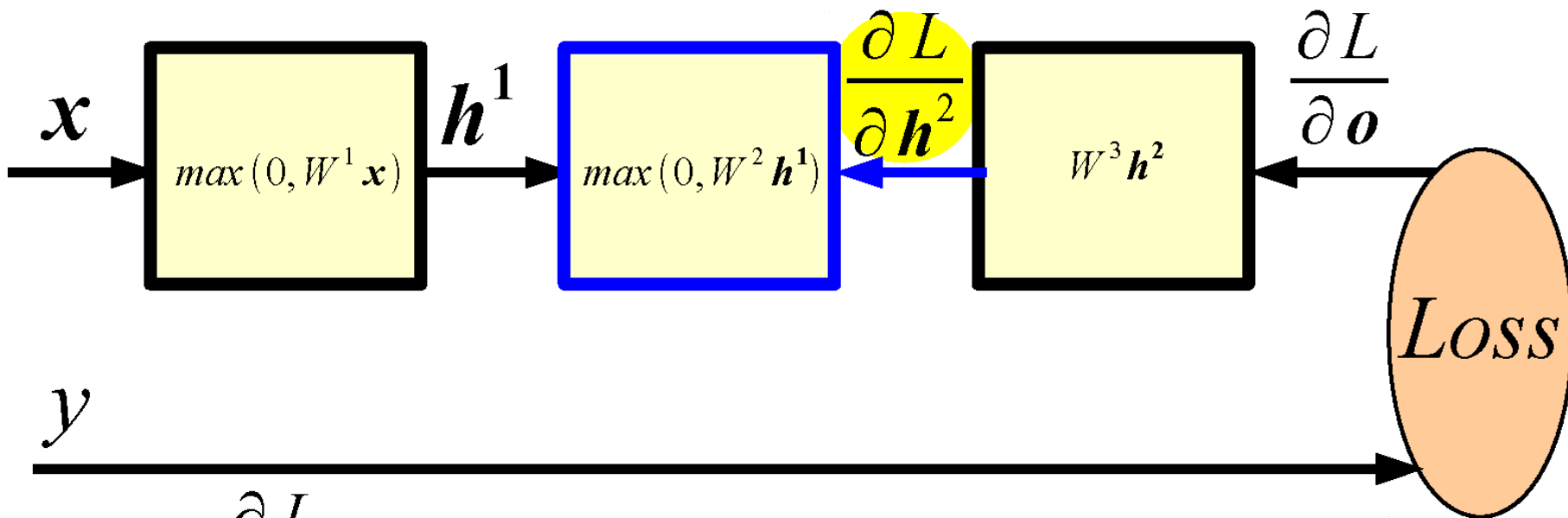
$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial W^3}$$

$$\frac{\partial L}{\partial \mathbf{h}^2} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}^2}$$

$$\frac{\partial L}{\partial W^3} = (p(c|\mathbf{x}) - \mathbf{y}) \mathbf{h}^{2T}$$

$$\frac{\partial L}{\partial \mathbf{h}^2} = W^{3T} (p(c|\mathbf{x}) - \mathbf{y})$$

Backward Propagation

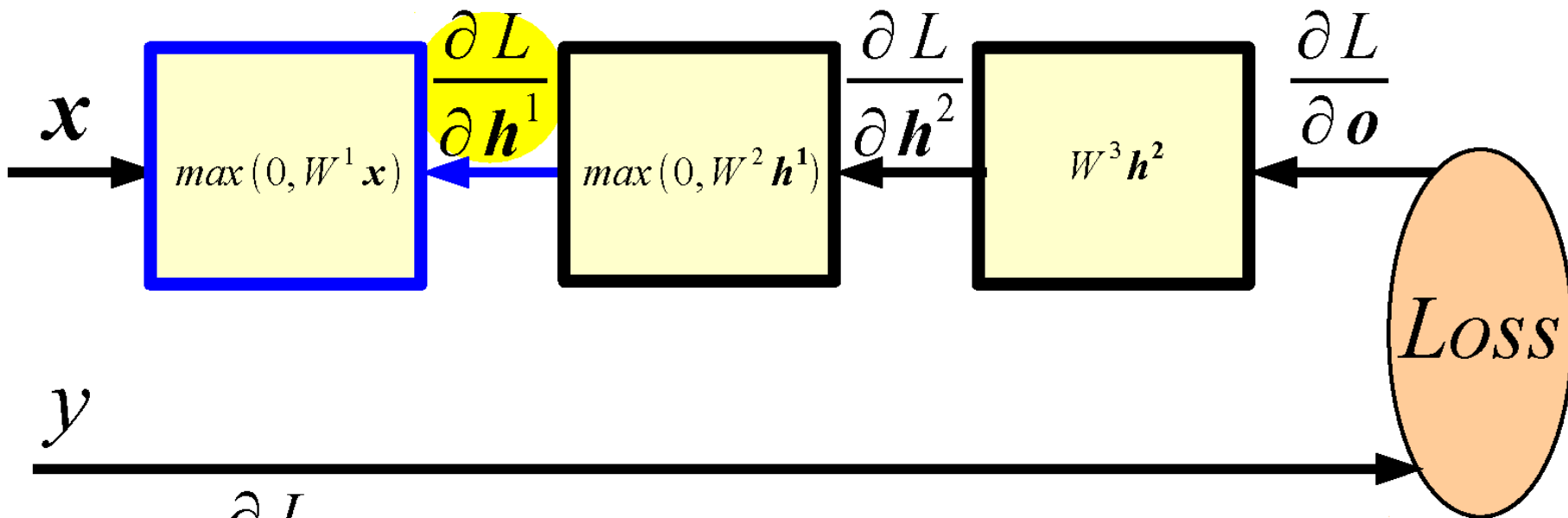


Given $\frac{\partial L}{\partial h^2}$ we can compute now:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial h^2} \frac{\partial h^2}{\partial W^2}$$

$$\frac{\partial L}{\partial h^1} = \frac{\partial L}{\partial h^2} \frac{\partial h^2}{\partial h^1}$$

Backward Propagation



Given $\frac{\partial L}{\partial h^1}$ we can compute now:

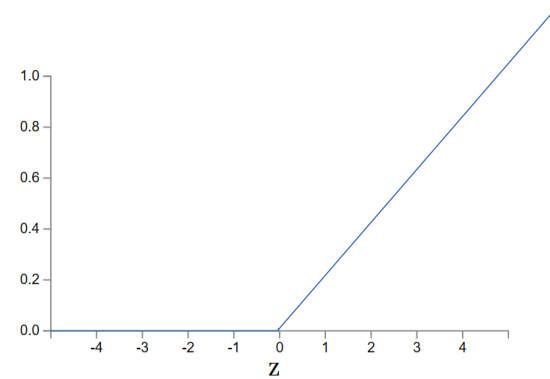
$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial h^1} \frac{\partial h^1}{\partial W^1}$$

Backward Propagation

Question: Does BPROP work with ReLU layers only?

Answer: Nope, any a.e. differentiable transformation works.

But the ReLU is not differentiable at 0!



Right. Fudge!

- '0' is the best place for this to occur, because we don't care about the result (it is no activation).
- 'Dead' perceptrons
- ReLU has unbounded positive response:
 - Potential faster convergence / overstep

Backward Propagation

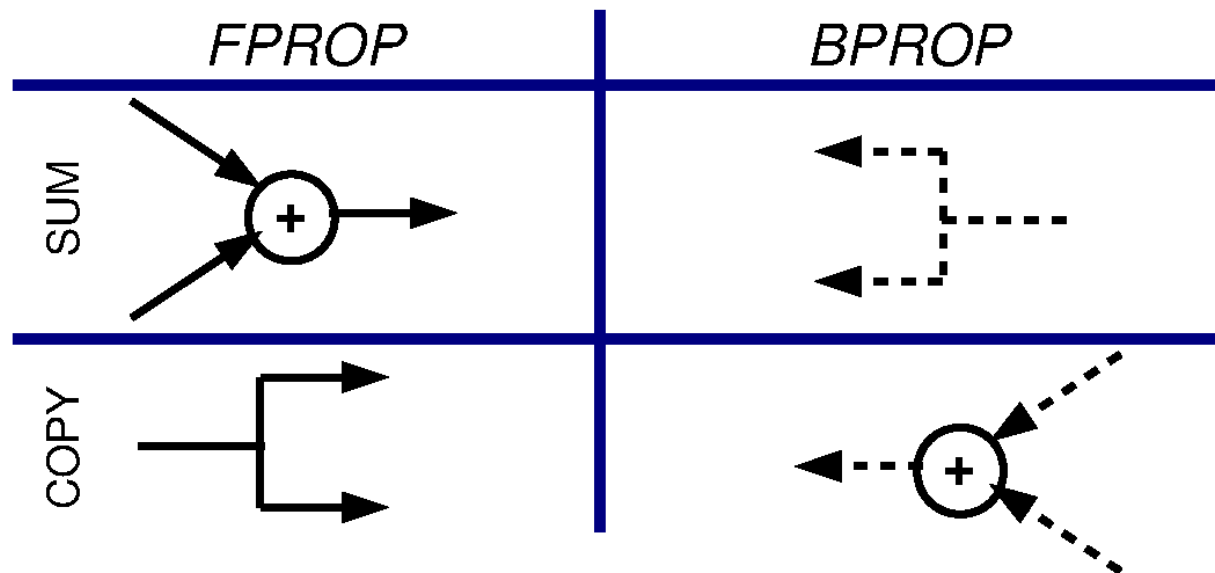
Question: Does BPROP work with ReLU layers only?

Answer: Nope, any a.e. differentiable transformation works.

Question: What's the computational cost of BPROP?

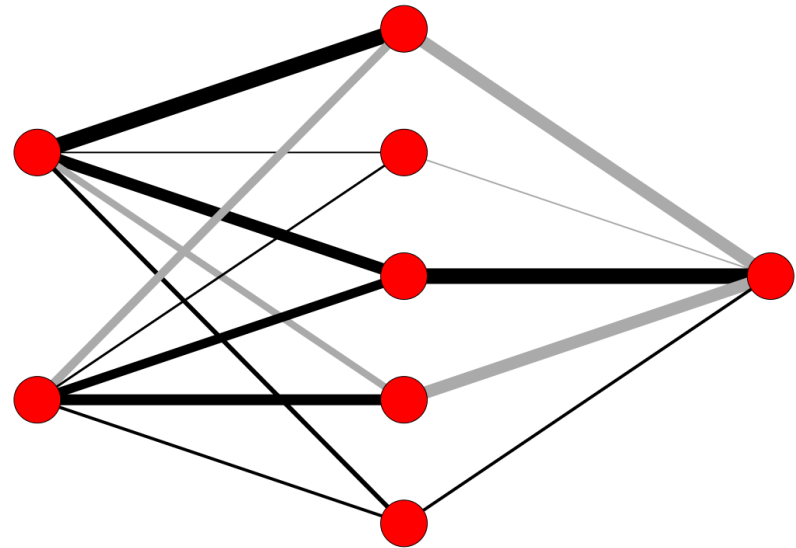
Answer: About twice FPROP (need to compute gradients w.r.t. input and parameters at every layer).

Note: FPROP and BPROP are dual of each other. E.g.,:



Optimization demo

- <http://www.emergentmind.com/neural-network>
- Thank you Matt Mazur



Toy Code (Matlab): Neural Net Trainer

```
% F-PROP
for i = 1 : nr_layers - 1
    [h{i}  jac{i}] = nonlinearity(W{i} * h{i-1} + b{i});
end
h{nr_layers-1} = W{nr_layers-1} * h{nr_layers-2} + b{nr_layers-1};
prediction = softmax(h{l-1});

% CROSS ENTROPY LOSS
loss = - sum(sum(log(prediction) .* target)) / batch_size;

% B-PROP
dh{l-1} = prediction - target;
for i = nr_layers - 1 : -1 : 1
    Wgrad{i} = dh{i} * h{i-1}';
    bgrad{i} = sum(dh{i}, 2);
    dh{i-1} = (W{i}' * dh{i}) .* jac{i-1};
end

% UPDATE
for i = 1 : nr_layers - 1
    W{i} = W{i} - (lr / batch_size) * Wgrad{i};
    b{i} = b{i} - (lr / batch_size) * bgrad{i};
end
```



Wow



what class

so misclassified

false positives

no good filtr

cool kernel

Stochastic Gradient Descent

- Dataset can be too large to strictly apply gradient descent wrt. all data points.
- Instead, randomly sample a data point, perform gradient descent per point, and iterate.
 - True gradient is approximated only
 - Picking a subset of points: “*mini-batch*”

Randomly initialize starting W and pick learning rate γ

While not at minimum:

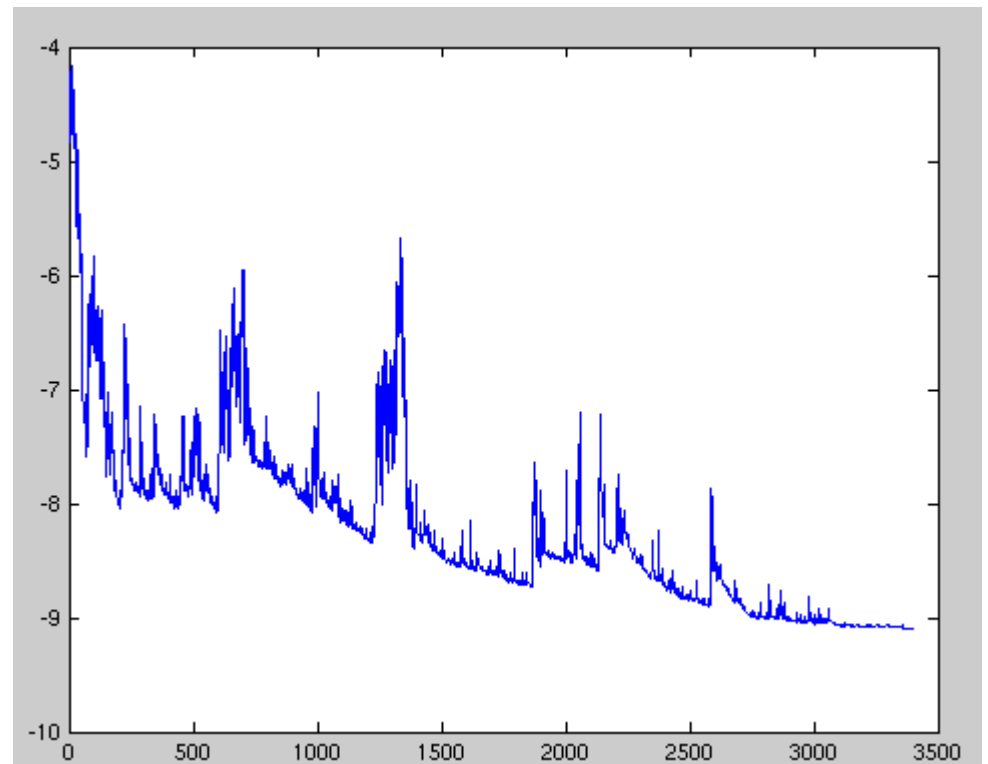
- Shuffle training set
- For each data point $i=1\dots n$ (*maybe as mini-batch*)
 - *Gradient descent*

} “Epoch”

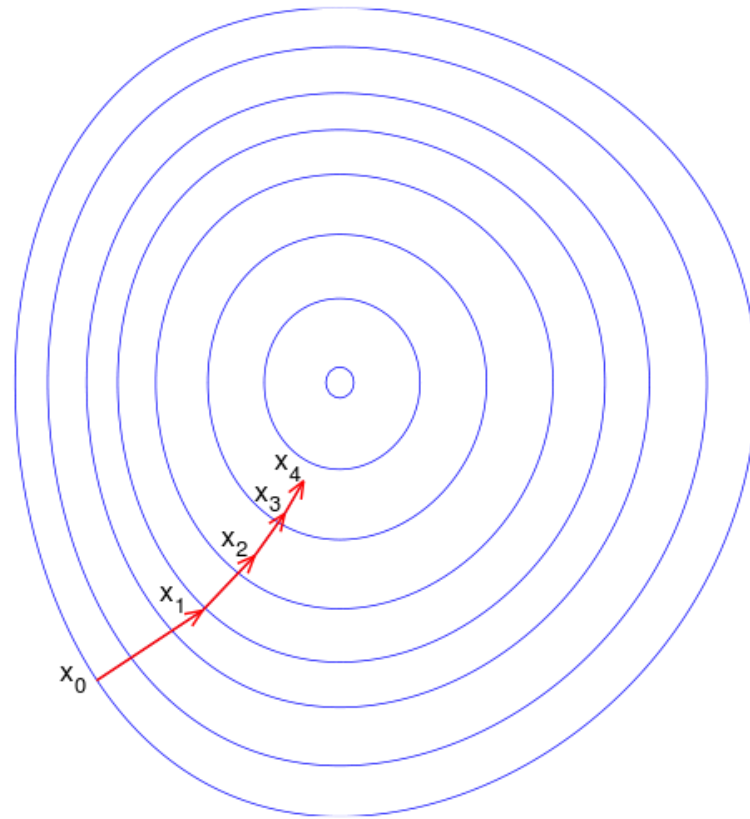
Stochastic Gradient Descent

Loss will not always decrease (locally) as training data point is random.

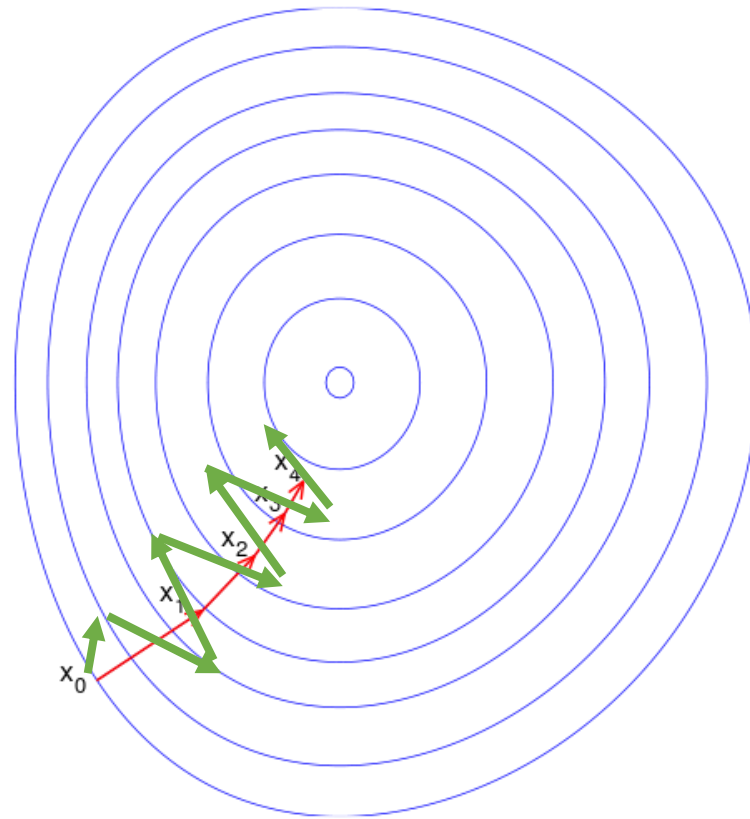
Still converges over time.



Gradient descent oscillations



Gradient descent oscillations



Slow to
converge to
the (local)
optimum

Momentum

- Adjust the gradient by a weighted sum of the previous amount plus the current amount.

- Without momentum: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \frac{\partial L}{\partial \boldsymbol{\theta}}$

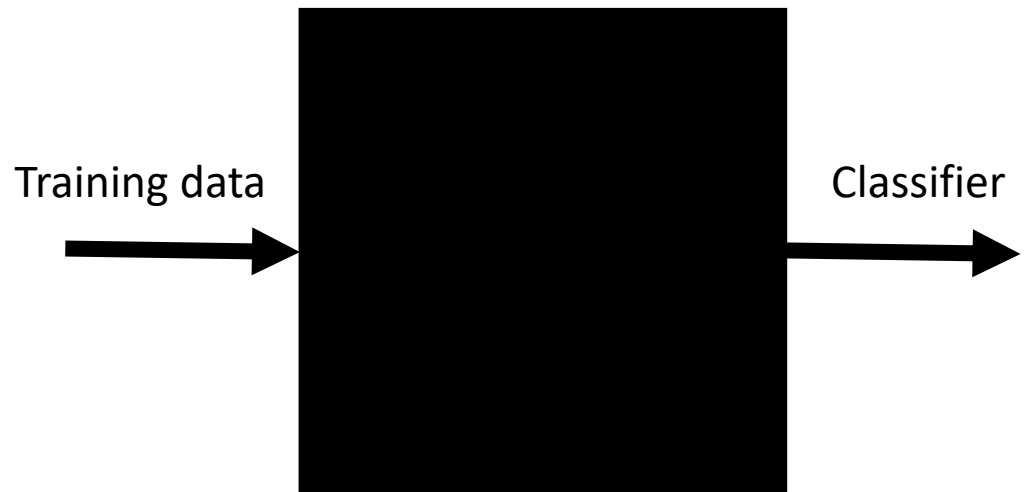
- With momentum (new α parameter):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \left(\alpha \left[\frac{\partial L}{\partial \boldsymbol{\theta}} \right]_{t-1} + \left[\frac{\partial L}{\partial \boldsymbol{\theta}} \right]_t \right)$$

But James...

...I thought we were going to treat machine learning like a black box? I like black boxes.

Deep learning is:
- a black box



But James...

...I thought we were going to treat machine learning like a black box? I like black boxes.

Deep learning is:

- a black box
- *also a black art.*



But James...

...I thought we were going to treat machine learning like a black box? I like black boxes.

Many approaches and hyperparameters:

Activation functions, learning rate, mini-batch size, momentum...

Often these need tweaking, and you need to know what they do to change them intelligently.

Nailing hyperparameters + trade-offs



agokasla 6:56 PM

uploaded and commented on this image: [image.png](#) ▾

“ WOOT! Nailed the hyperparameters. 4 generator updates per discriminator update. Wait extra long before you initiate the switch.



jamestompkin 6:57 PM

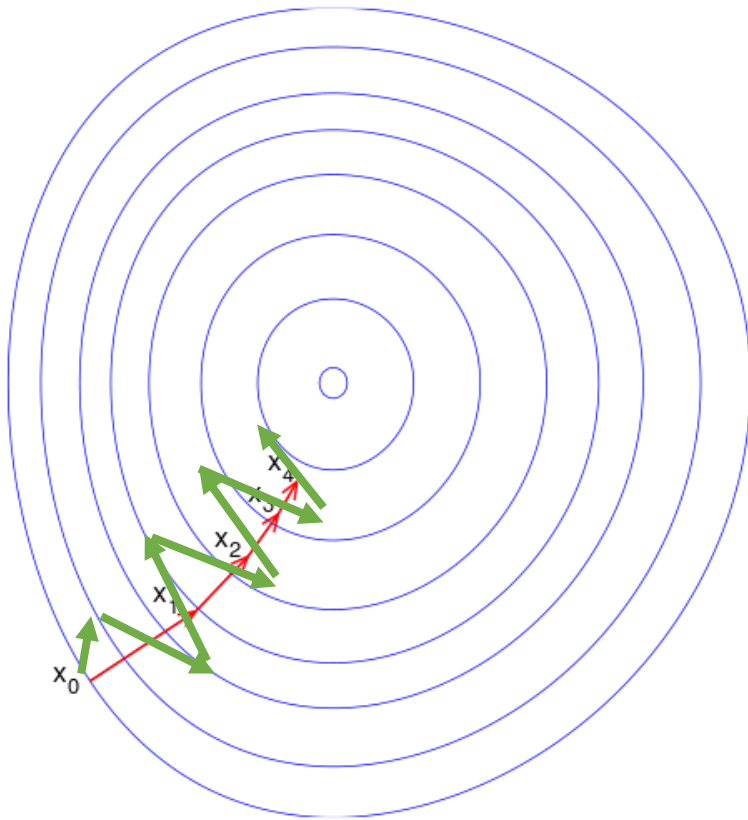
Well done - I wonder if we can turn hyperparameter nailing into the next e-Sport?



agokasla 4:30 AM

I am starting to think that the numeric instability of the model is starting to become a real issue. Lowering the learning rate could make it more stable, but it would require lowering it by two orders of magnitude which would make it take 100x longer to train right? 😞

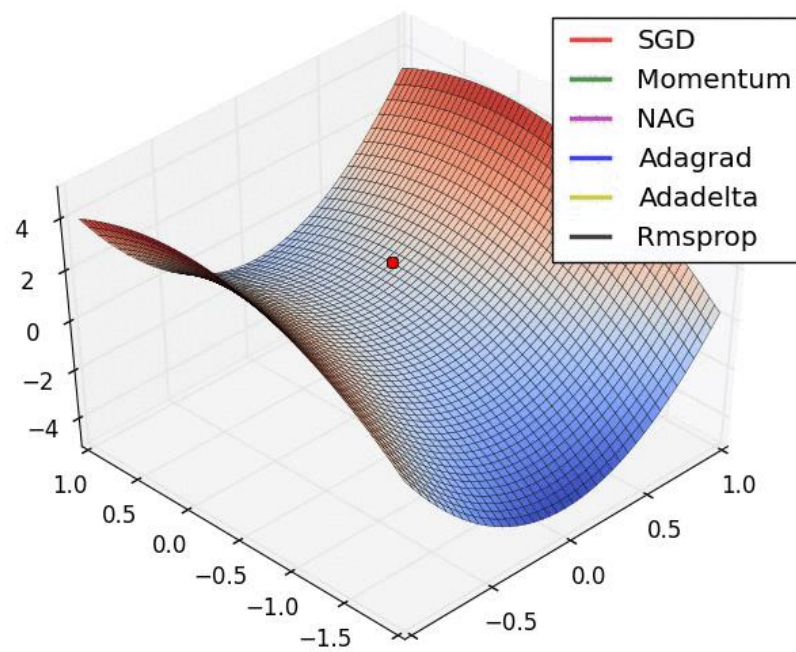
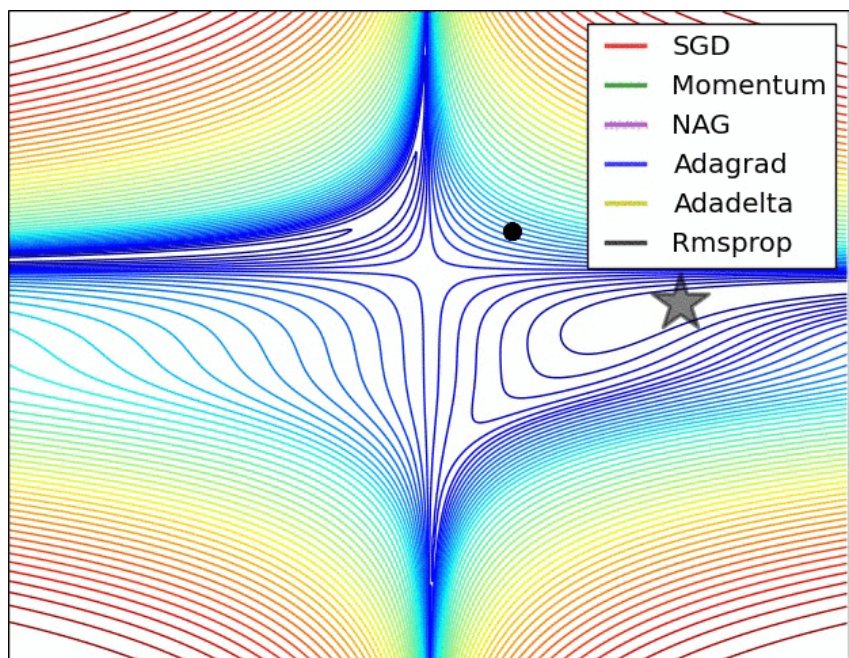
Lowering the learning rate = smaller steps in SGD



-Less 'ping pong'

-Takes longer to get to the optimum

Flat regions in energy landscape





@teenybiscuit



©teenybiscuit



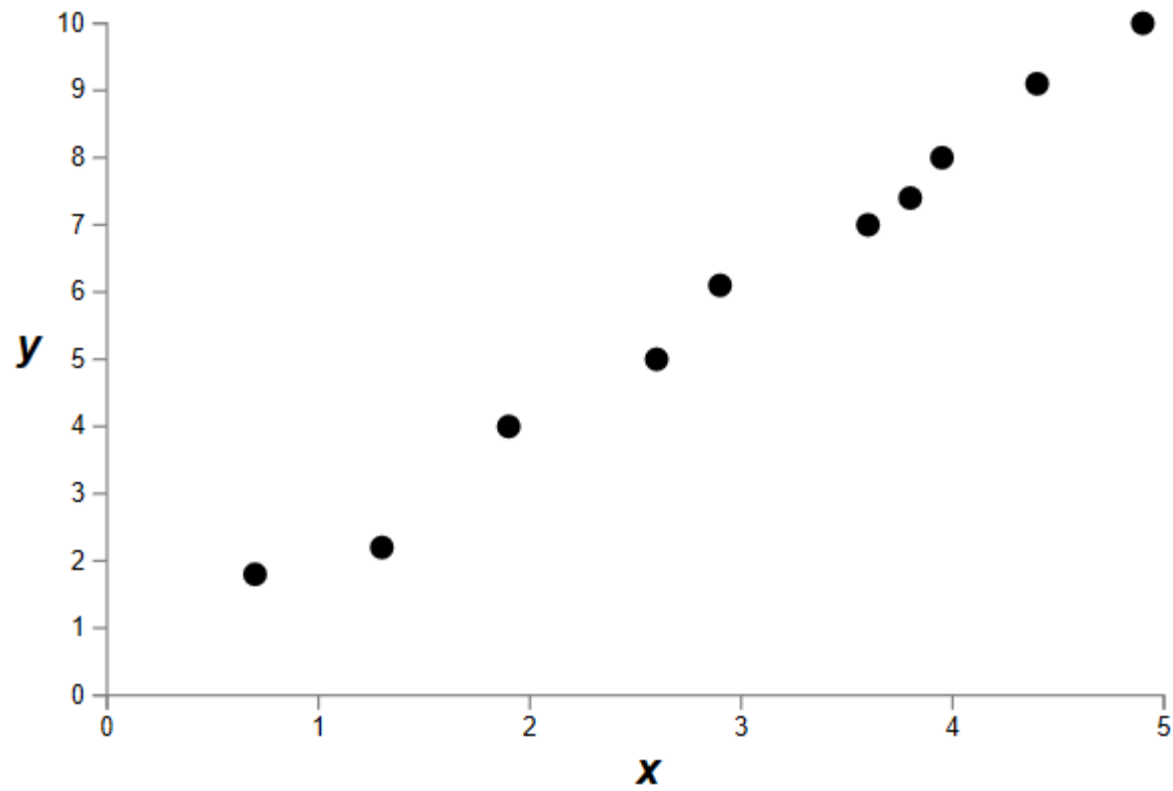
Problem of fitting

- Too many parameters = overfitting
- Not enough parameters = underfitting
- More data = less chance to overfit
- How do we know what is required?

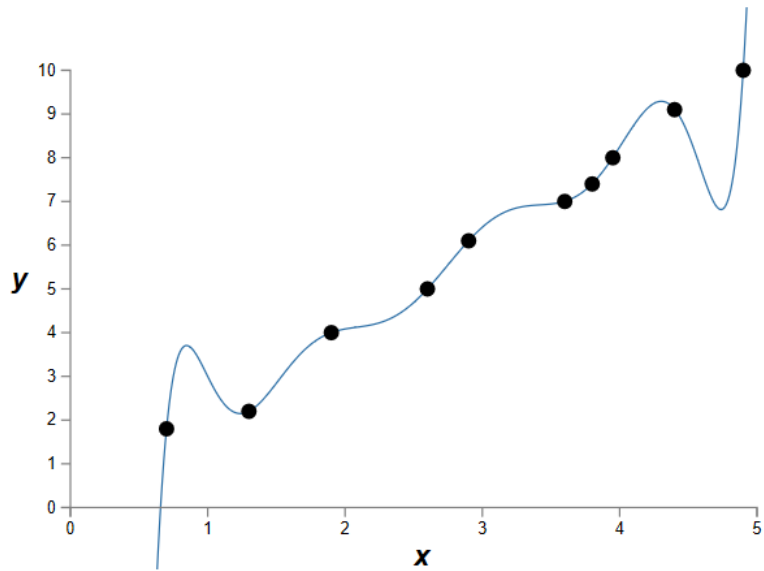
Regularization

- Attempt to guide solution to *not overfit*
- But still give freedom with many parameters

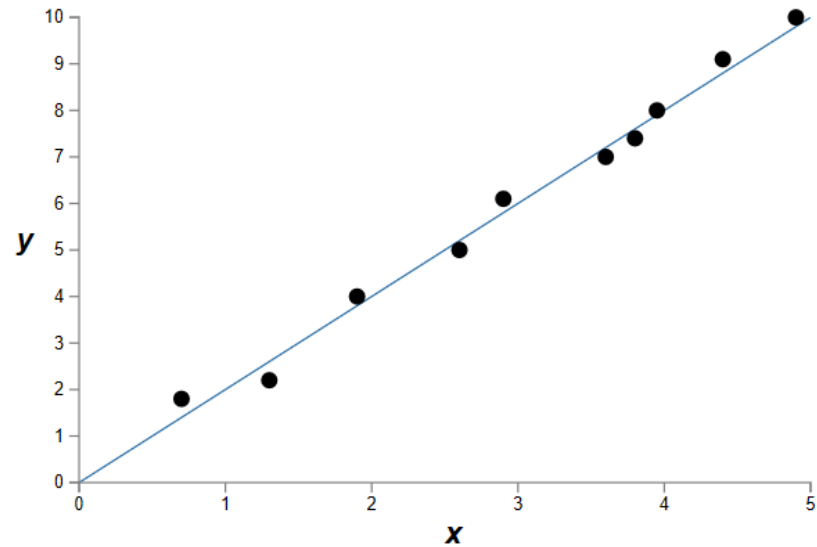
Data fitting problem



Which is better?
Which is better *a priori*?



9th order polynomial



1st order polynomial

Regularization

- Attempt to guide solution to *not overfit*
- But still give freedom with many parameters
- Idea:
Penalize the use of parameters to prefer small weights.

Regularization:

- Idea: add a cost to having high weights
- λ = regularization parameter

$$C = C_0 + \lambda \sum_w w^2,$$

Both can describe the data...

- ...but one is simpler.
- Occam's razor:
"Among competing hypotheses, the one with the fewest assumptions should be selected"

For us:

Large weights cause large changes in behaviour in response to small changes in the input.

Simpler models (or smaller changes) are more robust to noise.

Regularization

- Idea: add a cost to having high weights
- λ = regularization parameter

$$C = C_0 + \lambda \sum_w w^2,$$

$$C = \underbrace{-\frac{1}{n} \sum_{x_j} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]}_{\text{Normal cross-entropy loss (binary classes)}} + \underbrace{\lambda \sum_w w^2}_{\text{Regularization term}}.$$

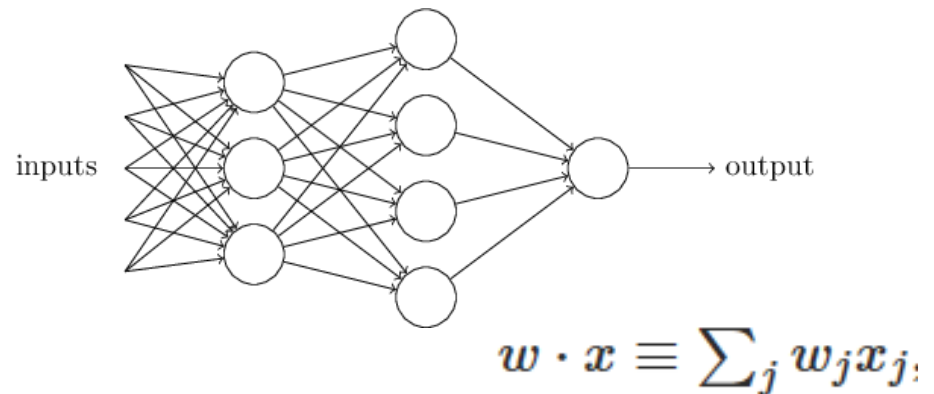
Normal cross-entropy
loss (binary classes)

Regularization term

Regularization: Dropout

Our networks typically start with random weights.
Every time we train = slightly different outcome.

- Why random weights?
- If weights are all equal, response across filters will be equivalent.
 - Network doesn't train.

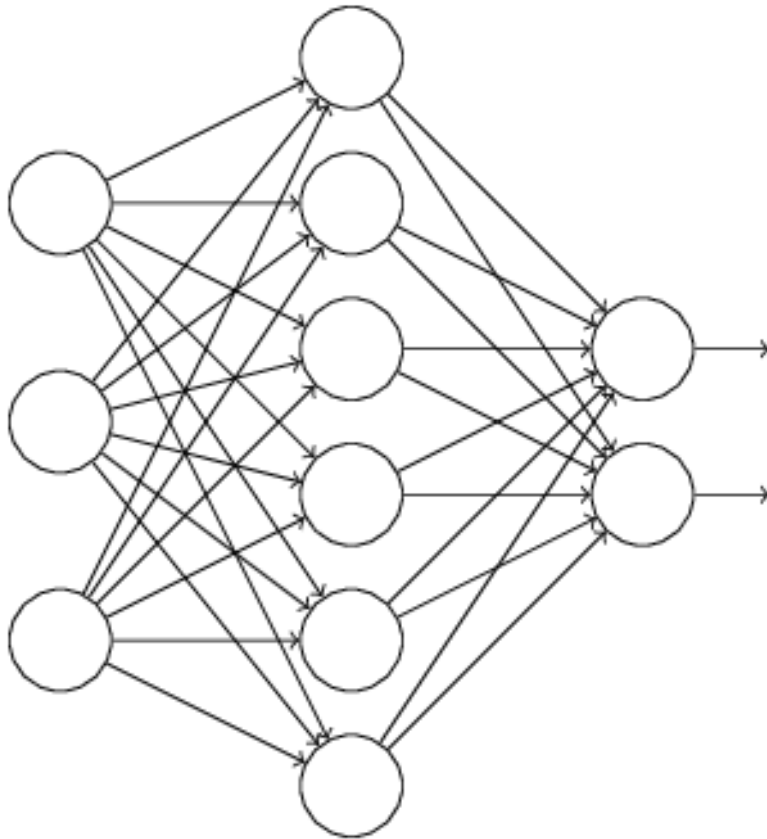


Regularization

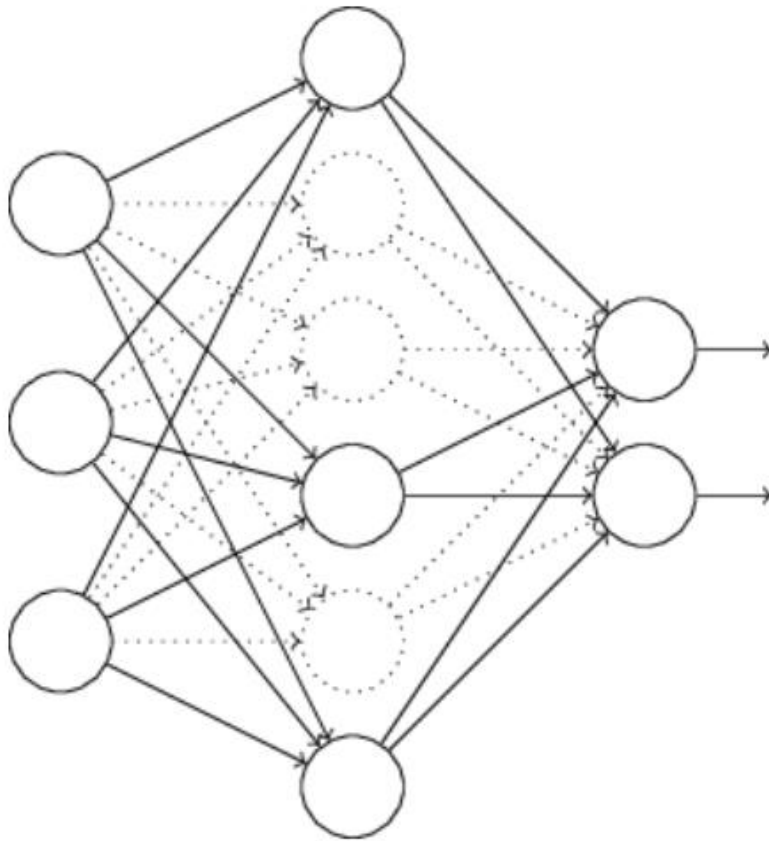
Our networks typically start with random weights.
Every time we train = slightly different outcome.

- Why not train 5 different networks with random starts and vote on their outcome?
 - Works fine!
 - Helps generalization because error due to overfitting is averaged; reduces variance.

Regularization: Dropout



Regularization: Dropout



At each mini-batch:

- Randomly select a subset of neurons.
- Ignore them.

On test: half weights outgoing to compensate for training on half neurons.

Effect:

- Neurons become less dependent on output of connected neurons.
- Forces network to learn more robust features that are useful to more subsets of neurons.
- Like averaging over many different trained networks with different random initializations.
- Except cheaper to train.

Many forms of 'regularization'

- Adding more data is a kind of regularization
- Pooling is a kind of regularization
- Data augmentation is a kind of regularization