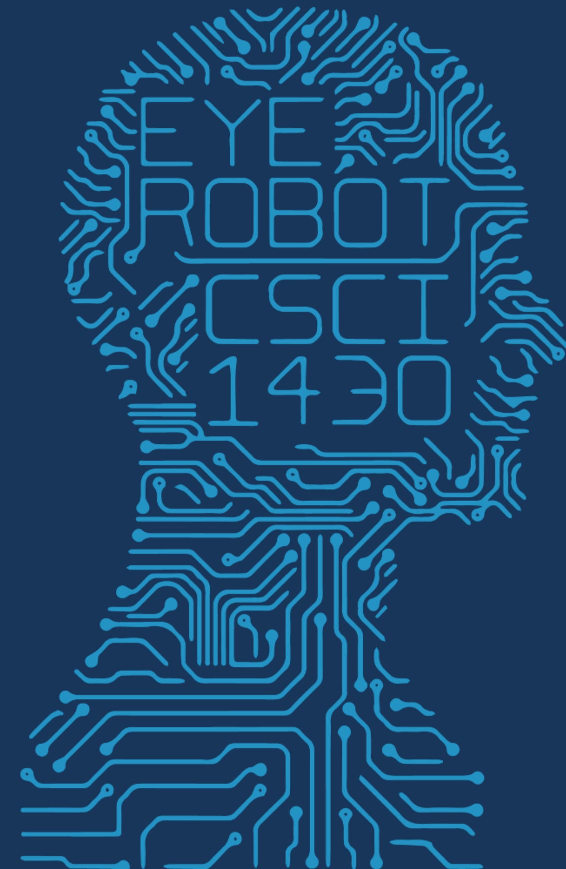
1950

Future Vision



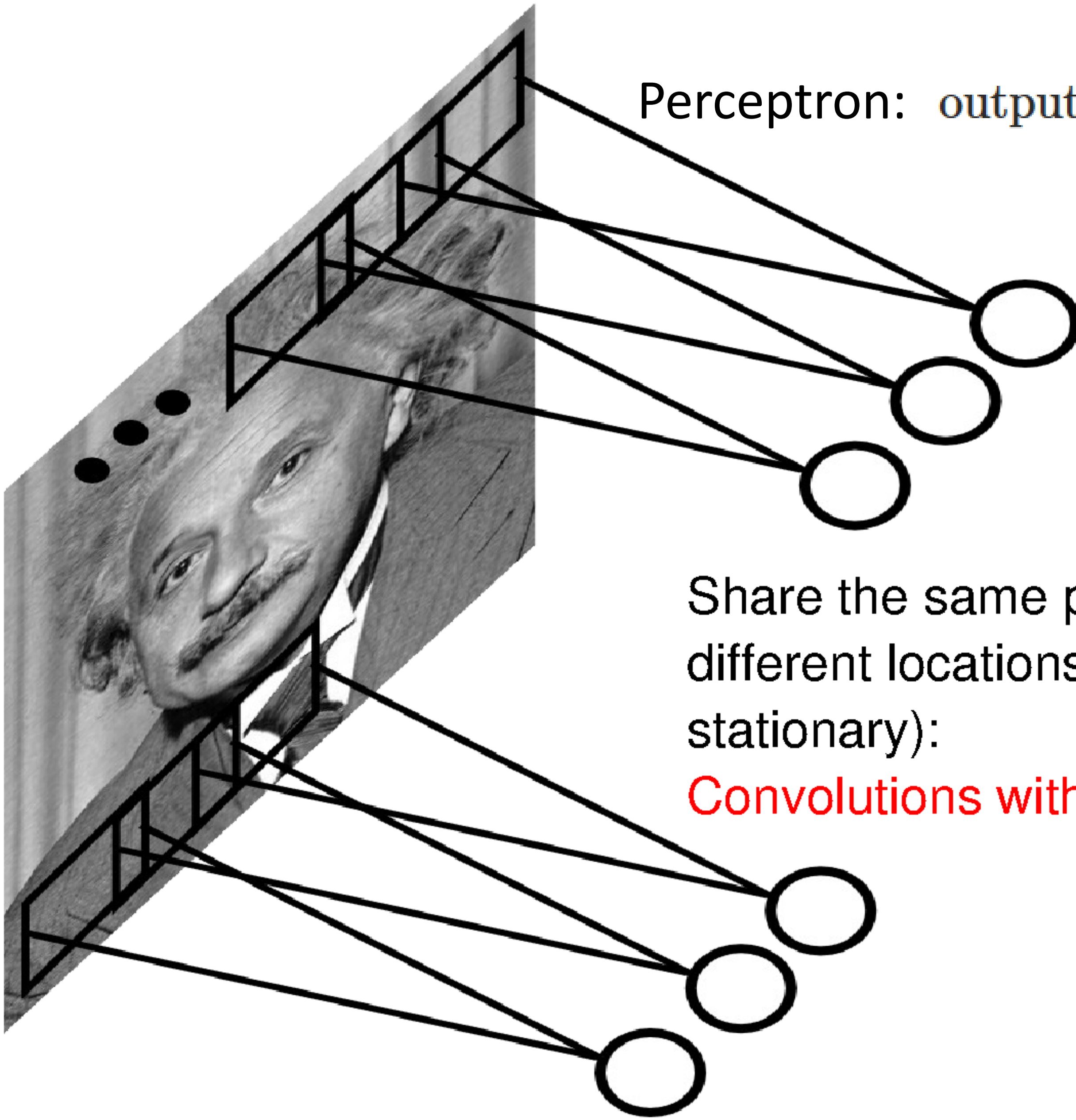
2017 MWF 1pm

Computer Vision


@teenybiscuit


@teenybiscuit

Convolutional Layer



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!

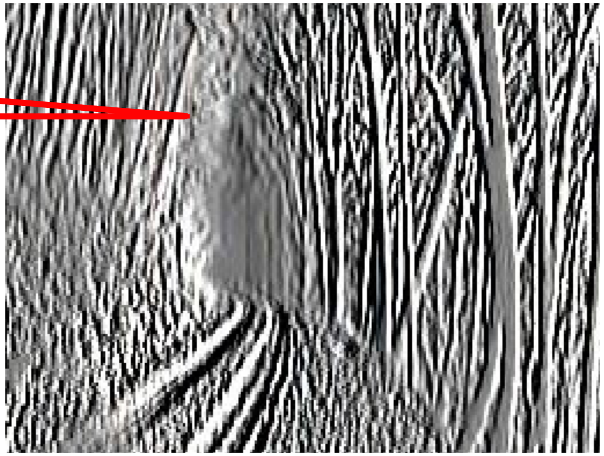
Share the same parameters across different locations (assuming input is stationary):
Convolutions with learned kernels

Ranzato

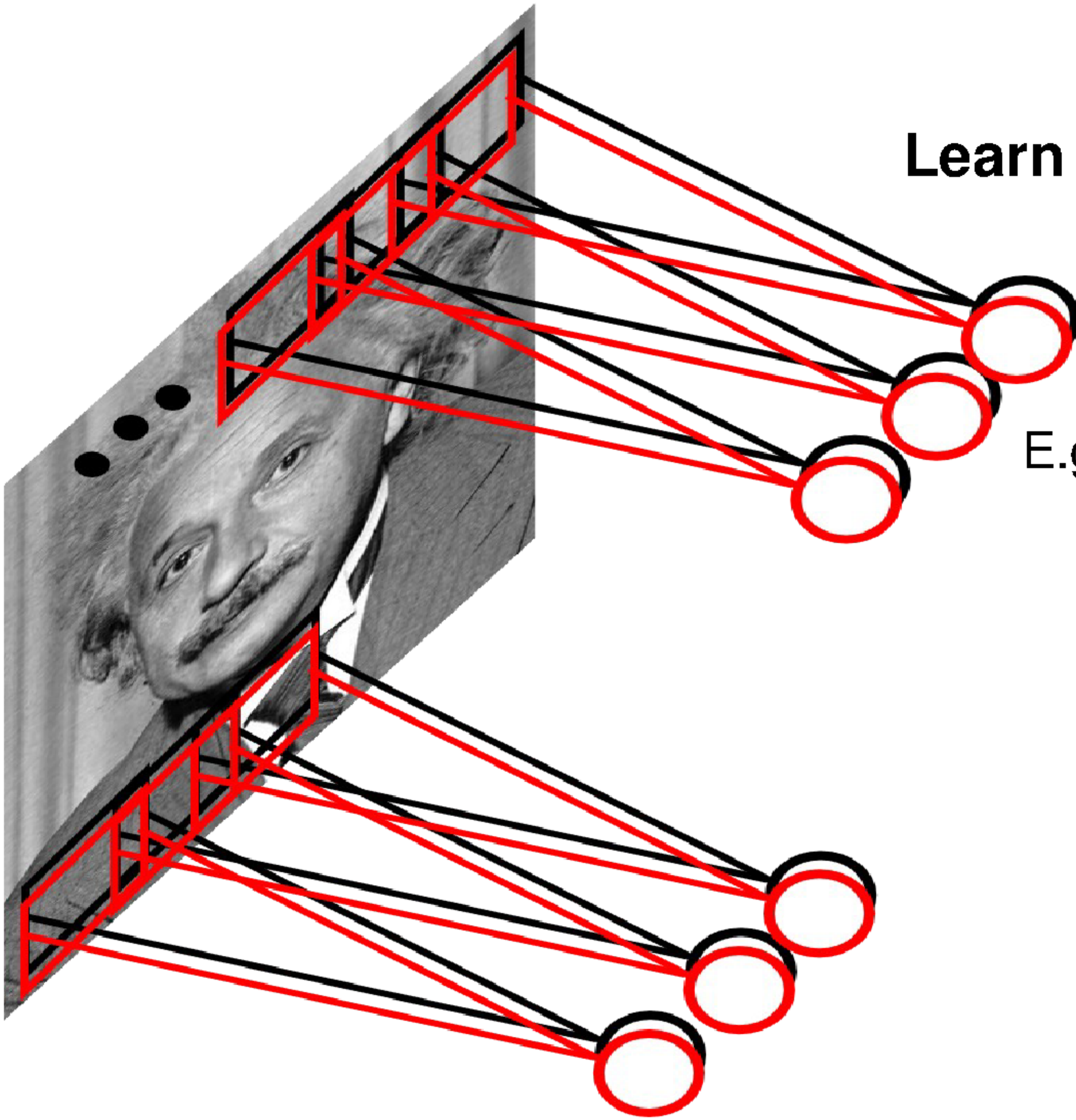
Convolutional Layer

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

Shared weights



Convolutional Layer

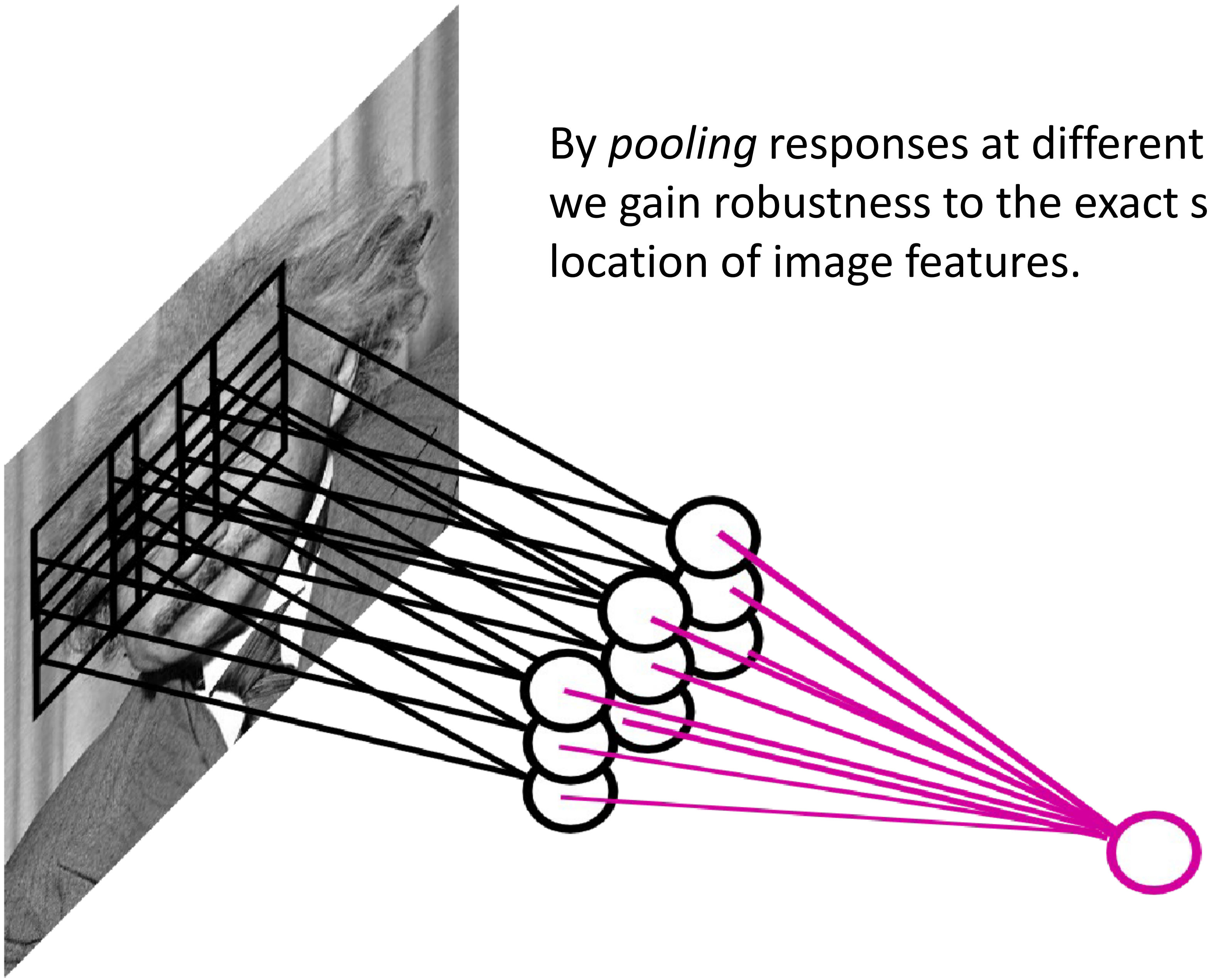


Learn multiple filters.

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

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

Pooling Layer



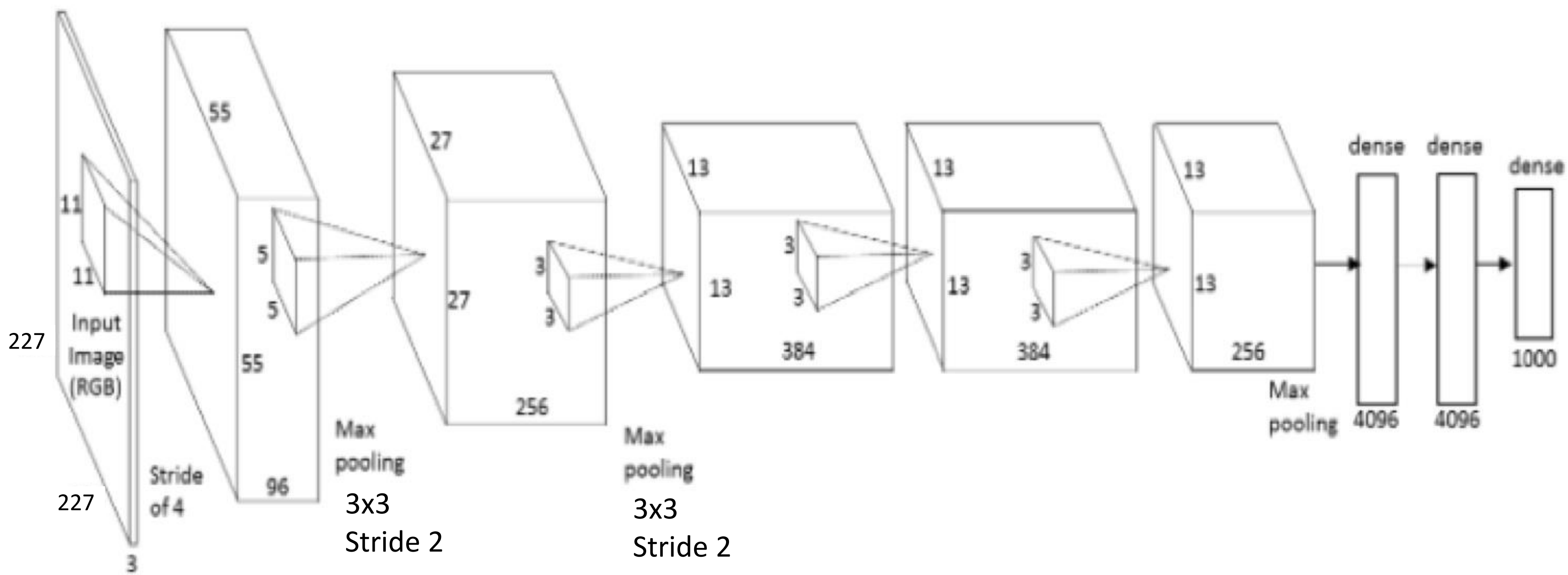
By *pooling* responses at different locations, we gain robustness to the exact spatial location of image features.

Ranzato

AlexNet diagram (simplified)

[Krizhevsky et al. 2012]

Input size
227 x 227 x 3

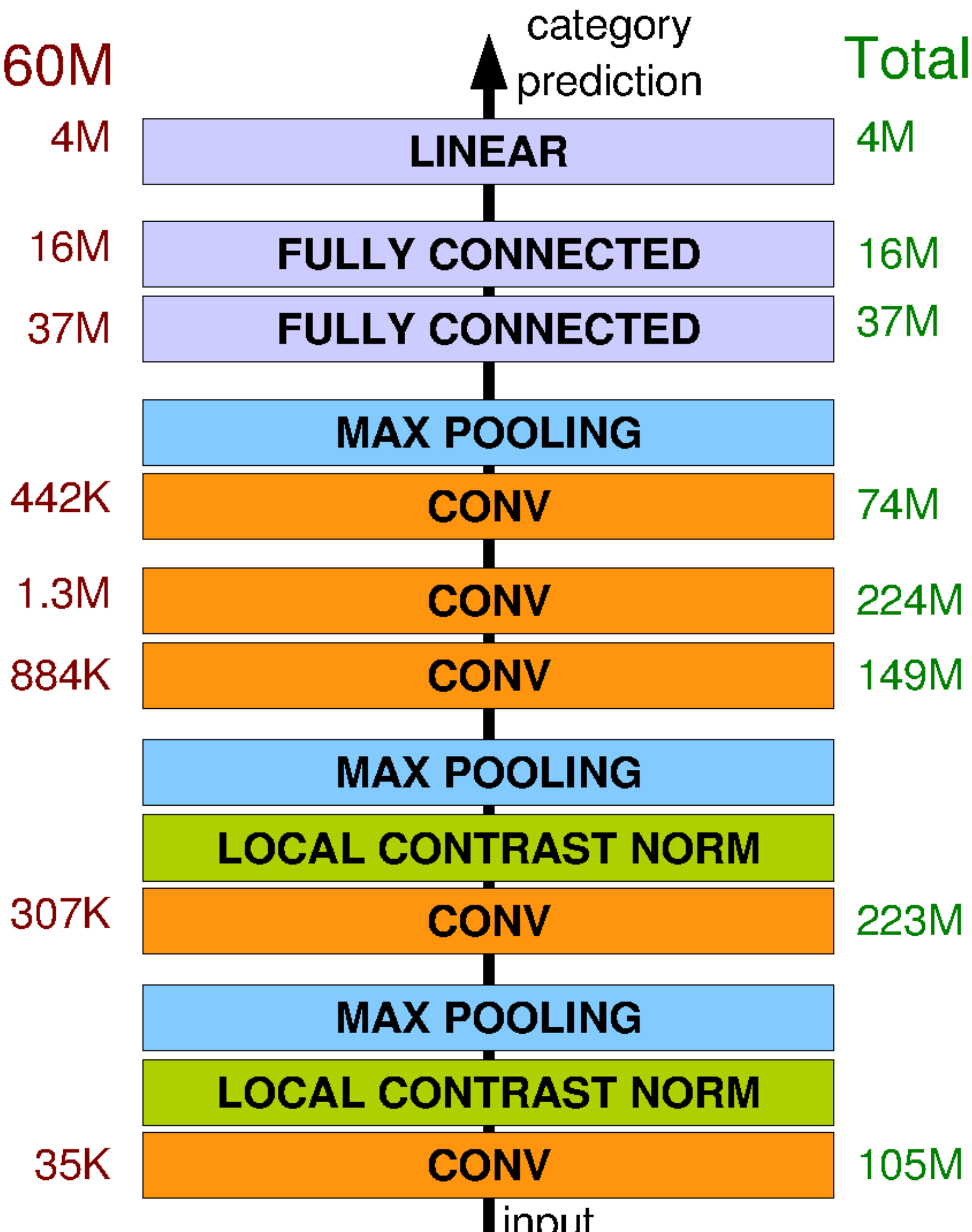


Conv 1	**Conv 2**	**Conv 3**	**Conv 4**	**Conv 4**
11 x 11 x 3	5 x 5 x 96	3 x 3 x 256	3 x 3 x 192	3 x 3 x 192
Stride 4	Stride 1	Stride 1	Stride 1	Stride 1
96 filters	256 filters	384 filters	384 filters	256 filters

Architecture for Classification

Total nr. params: 60M

Total nr. flops: 832M



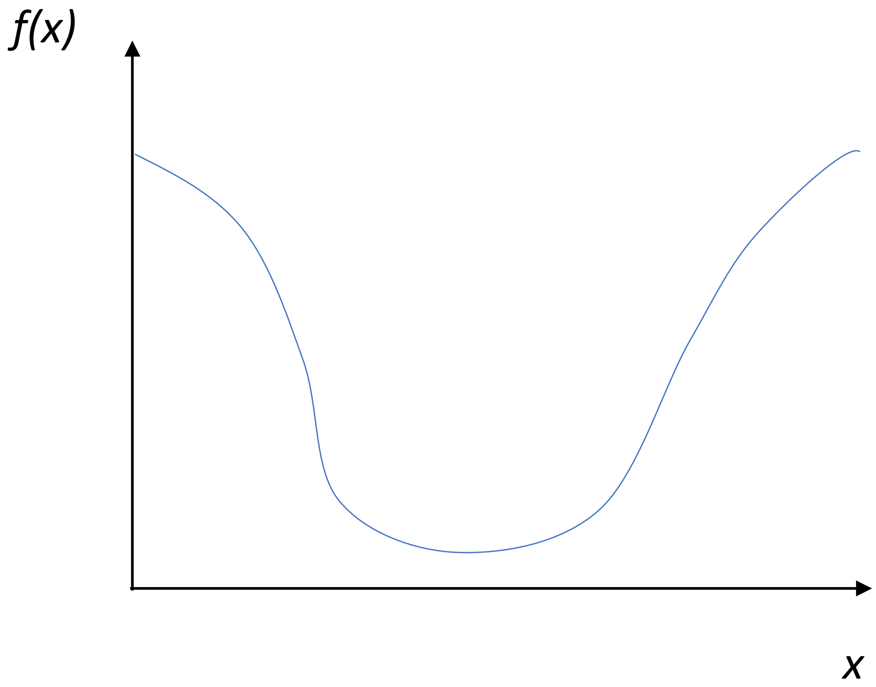
Krizhevsky et al. “ImageNet Classification with deep CNNs” NIPS 2012

Ranzato

Training Neural Networks

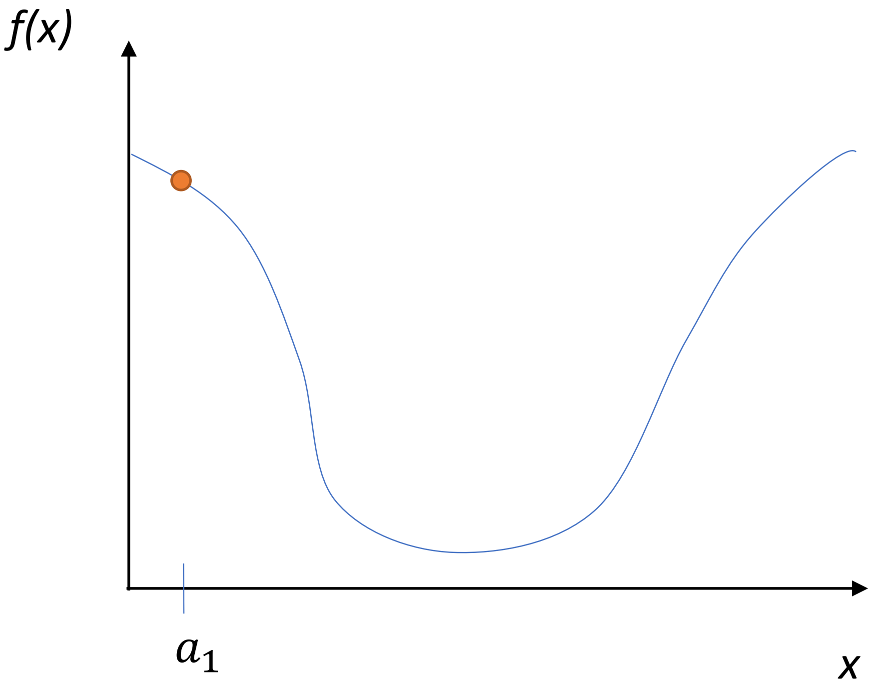
Learning the weight matrices $\boldsymbol{W}$

Gradient descent



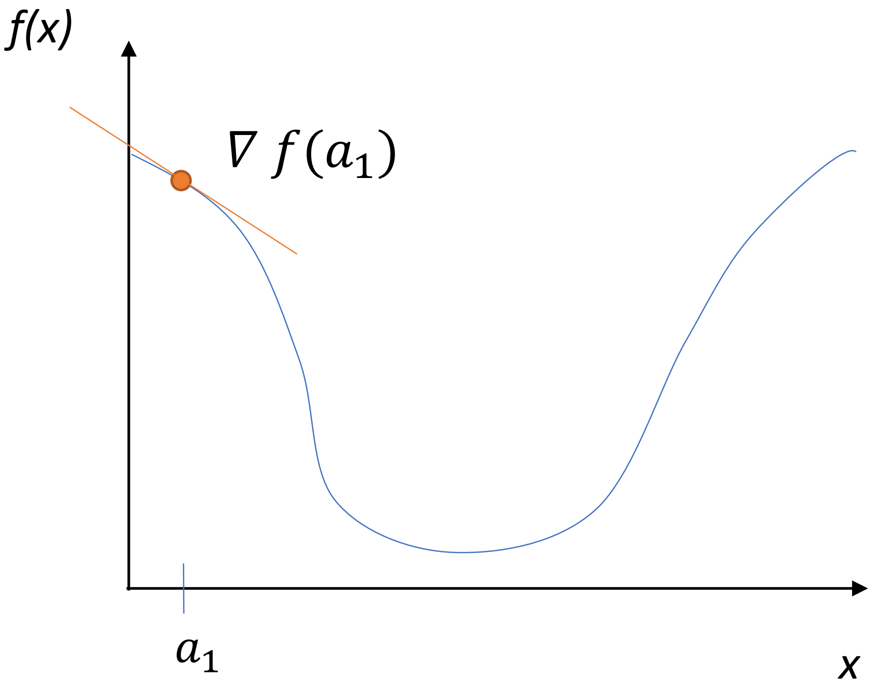
General approach

Pick random starting point.



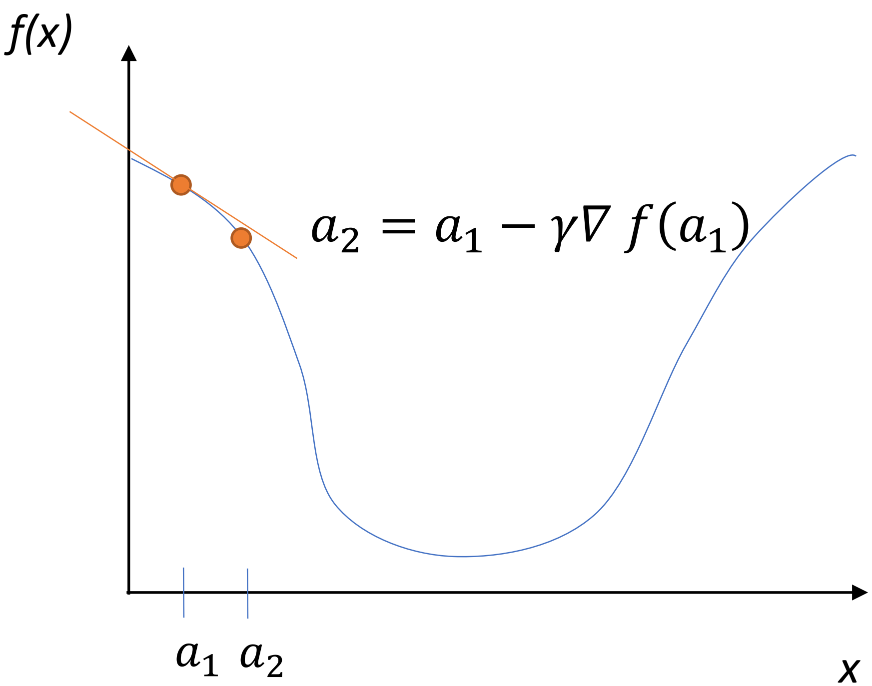
General approach

Compute gradient at point (analytically or by finite differences)



General approach

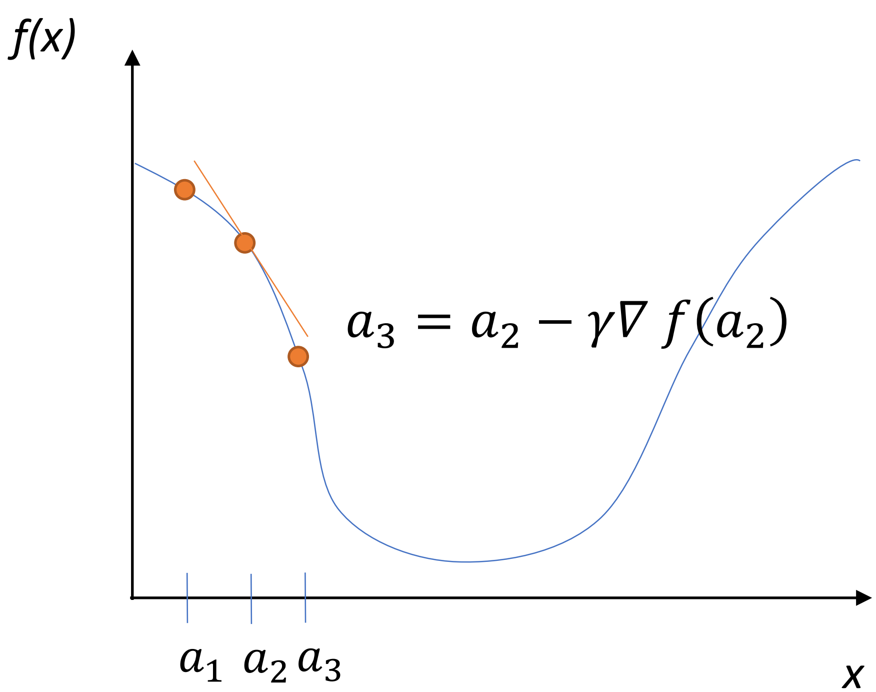
Move along parameter space in direction of negative gradient



γ = amount to move
= *learning rate*

General approach

Move along parameter space in direction of negative gradient.

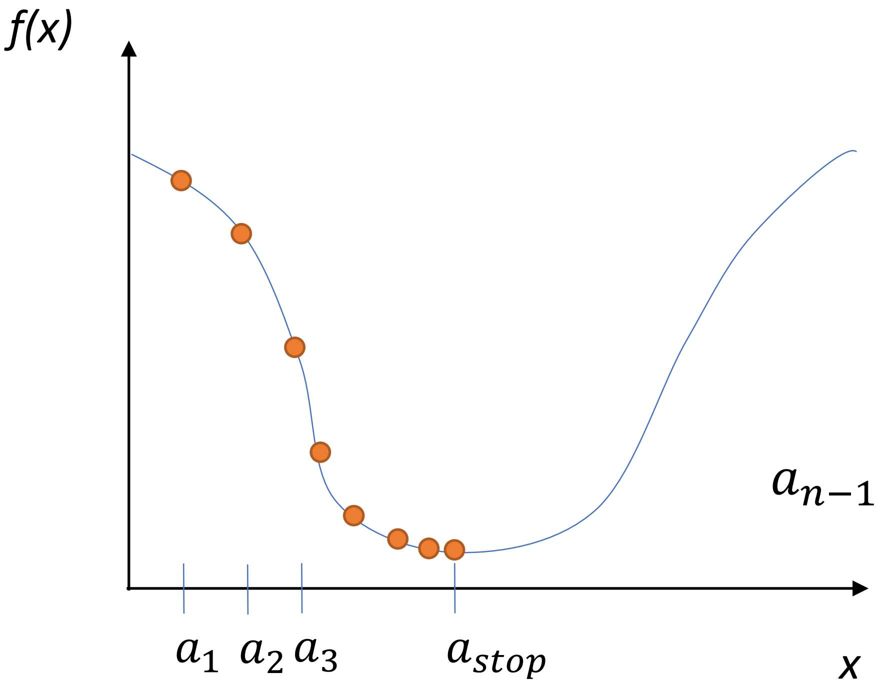


$$a_3 = a_2 - \gamma \nabla f(a_2)$$

γ = amount to move
= *learning rate*

General approach

Stop when we don't move any more.

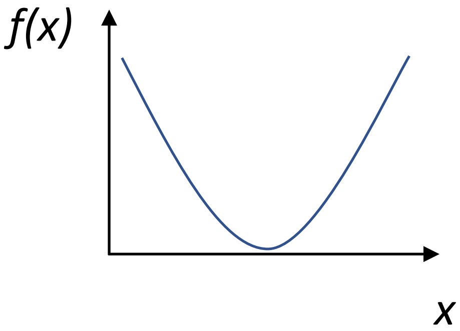


a_{stop}:

$$a_{n-1} - \gamma \nabla f(a_{n-1}) = 0$$

Gradient descent

- Optimizer for functions.
- Guaranteed to find optimum for convex functions.
  - Non-convex = find *local* optimum.
  - Most vision problems aren't convex.



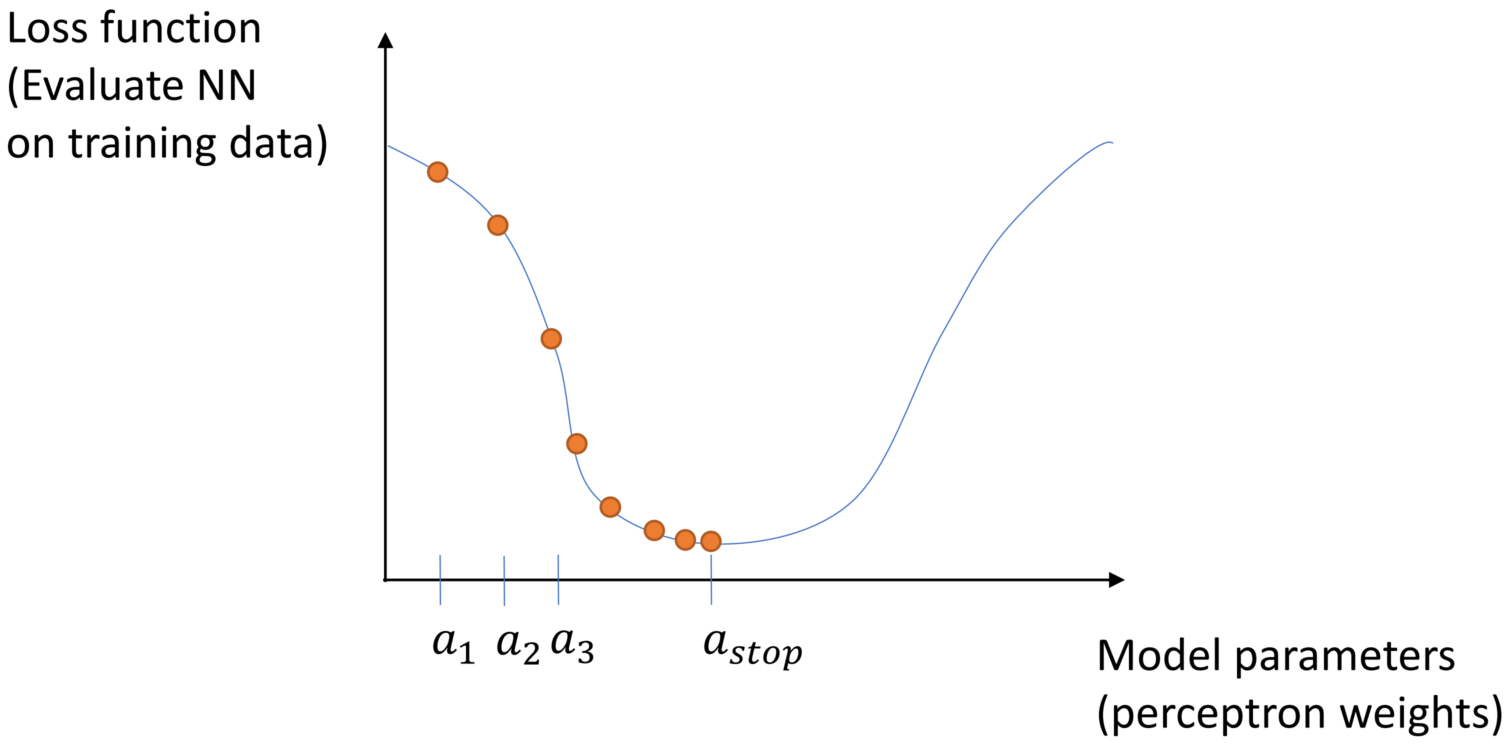
- Works for multi-variate functions.
  - Need to compute matrix of *partial derivatives ("Jacobian")*

Train NN with Gradient Descent

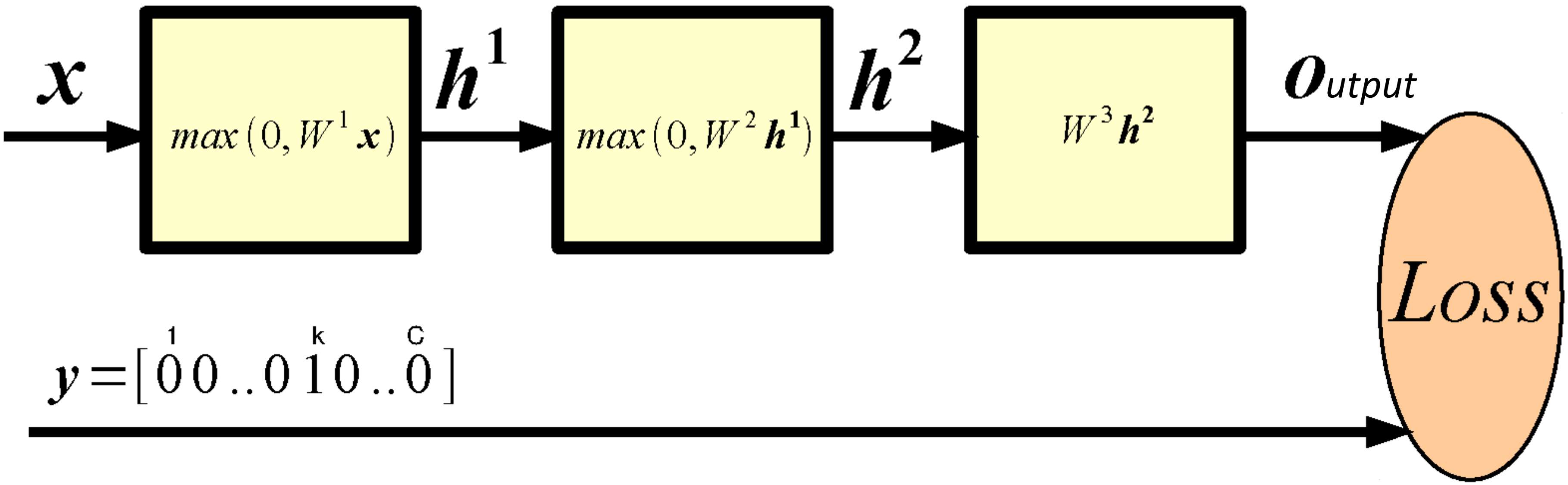
- x^i, y^i = *n* training examples
- $f(\boldsymbol{x})$ = feed forward neural network
- L(**x**, y; **θ**) = some *loss function*

- *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



How Good is a Network?



What is an appropriate loss?

- Define some output threshold on detection
- Classification: compare training class to output class
- Zero-one loss L (per class)

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

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

- 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 has binary outputs

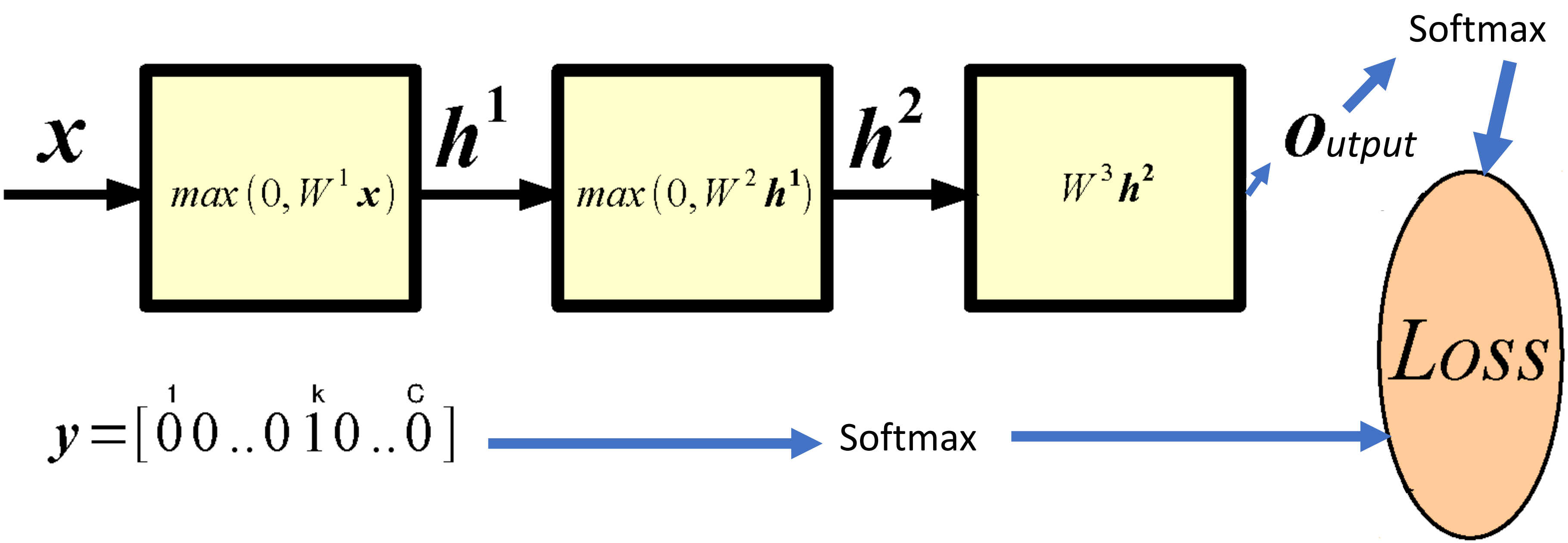
Special function on last layer - '*Softmax*':

- "squashes" a *C*-dimensional vector ***O*** of arbitrary real values to a *C*-dimensional vector σ(***O***) of real values in the range (0, 1) that add up to 1.

- Turns the output into a probability distribution on classes.

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

How Good is a Network?



Probability of class k given input (softmax):

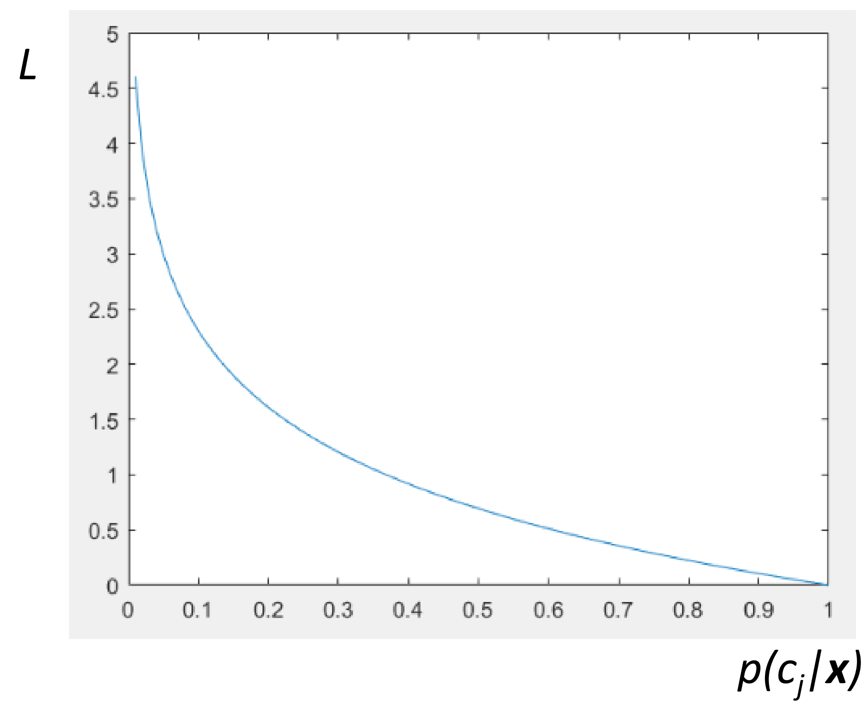
$$p(c_k = 1 | \boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

Cross-entropy loss function

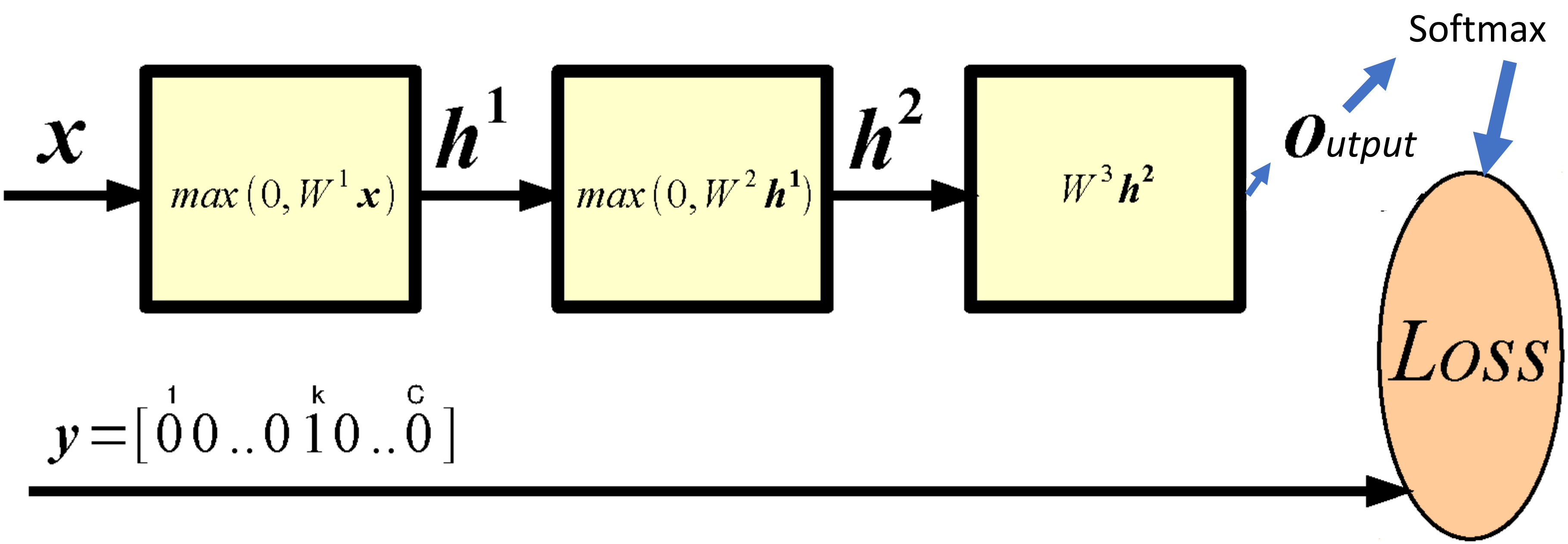
- Negative log-likelihood

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

- Is it a good loss?
  - Differentiable
  - Cost decreases as probability increases



How Good is a Network?



Probability of class k given input (softmax):

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

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

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

Training

Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

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

Training

Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

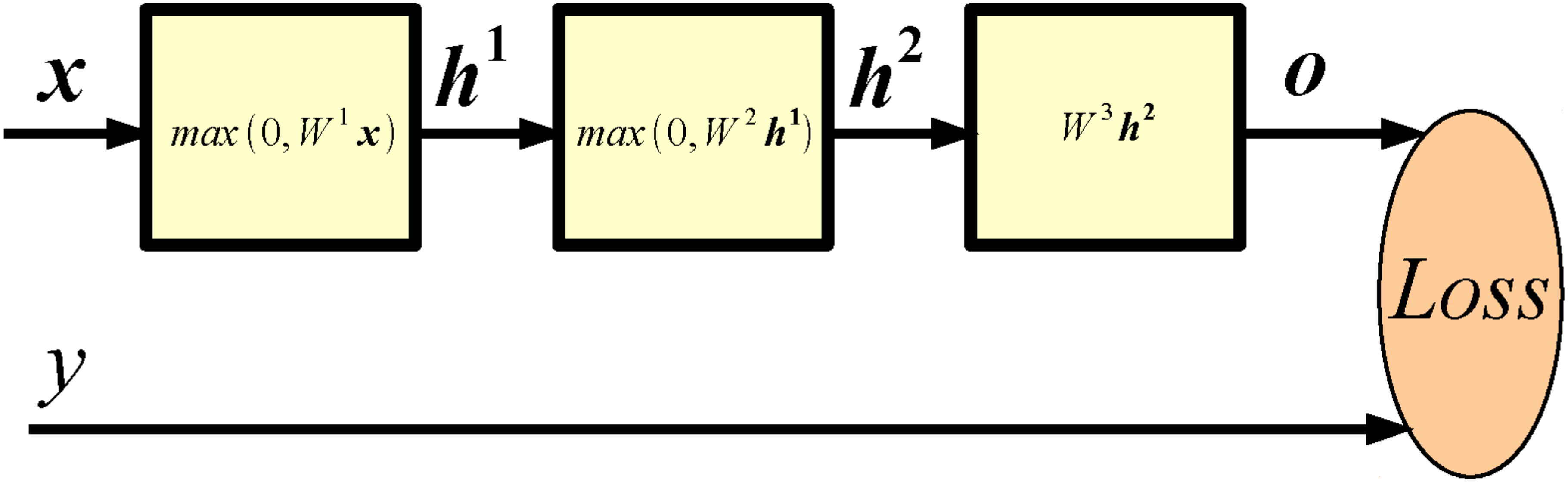
$$\boldsymbol{\theta}^* = arg\ min_{\boldsymbol{\theta}} \sum_{n=1}^{P} L(\boldsymbol{x}^n, y^n; \boldsymbol{\theta})$$

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.

Rumelhart et al. "Learning internal representations by back-propagating.." Nature 1986

Key Idea: Wiggle To Decrease Loss

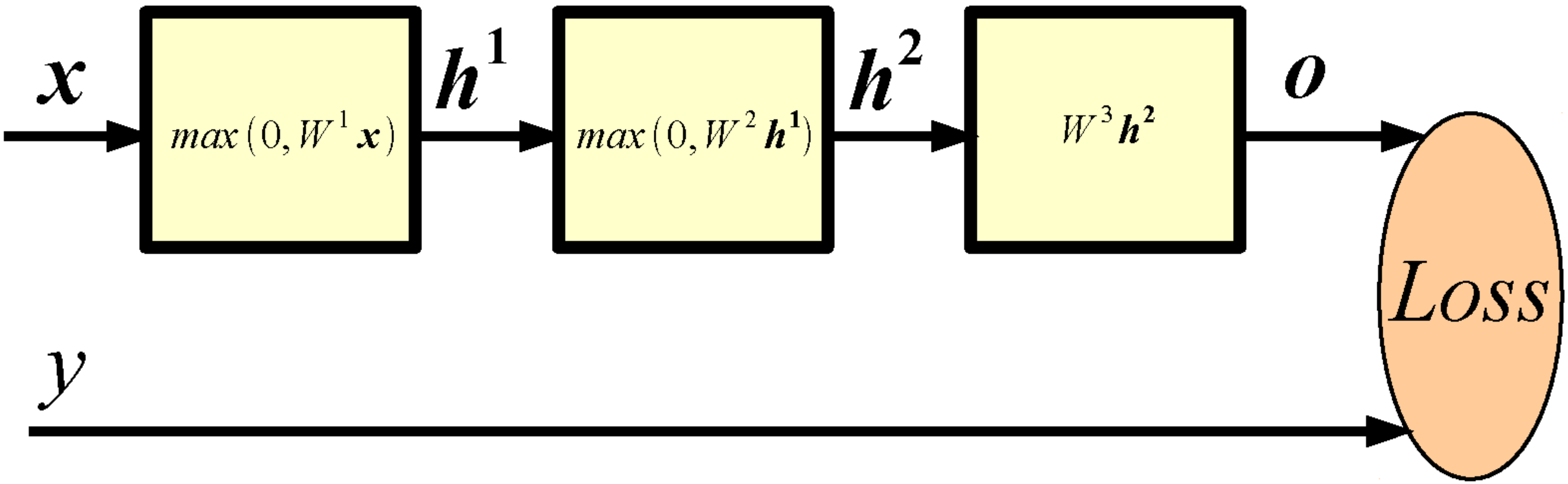


Let's say we want to decrease the loss by adjusting $W^1_{i,j}$.
We could consider a very small $\epsilon = 1\text{e-}6$ and compute:

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

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

Key Idea: Wiggle To Decrease Loss



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

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

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

Then, update:

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

Ranzato

Derivative w.r.t. Input of Softmax

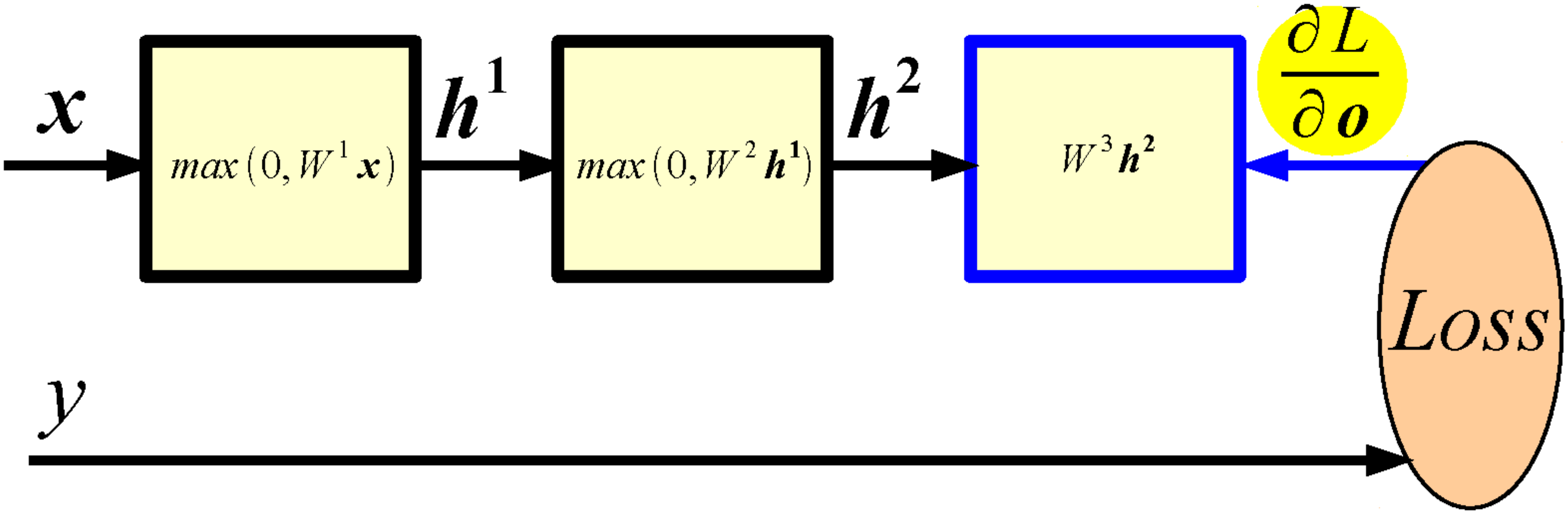
$$p(c_k=1|\boldsymbol{x})=\frac{e^{o_k}}{\sum_j e^{o_j}}$$

$$L(\boldsymbol{x},y;\boldsymbol{\theta})=-\sum_j y_j \log p(c_j|\boldsymbol{x}) \qquad \boldsymbol{y}=[\overset{1}{0}\,0\,..\,0\,\overset{k}{1}\,0\,..\,\overset{C}{0}]$$

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

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

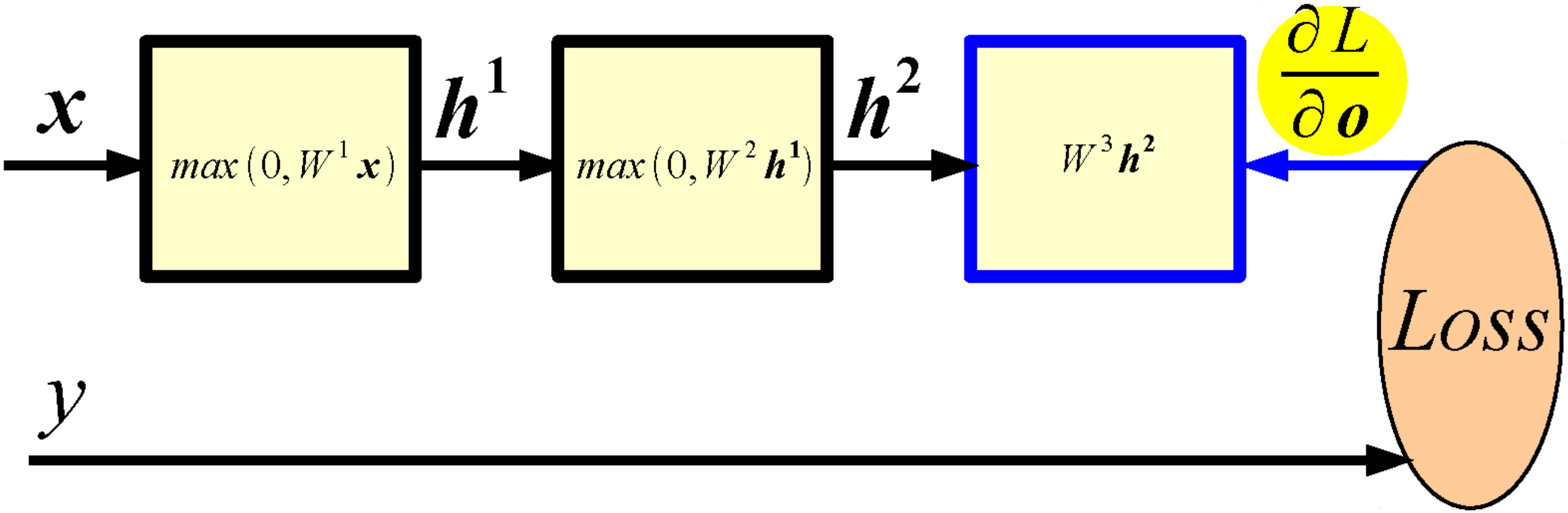
Backward Propagation



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

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

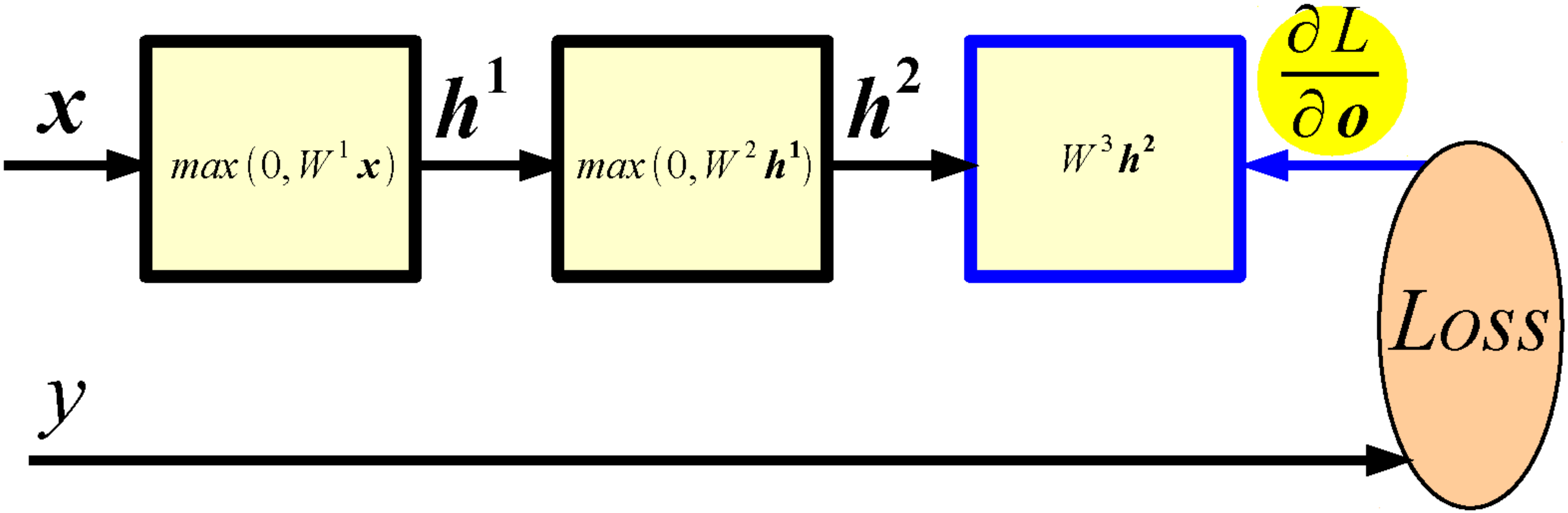
Backward Propagation



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

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

Backward Propagation

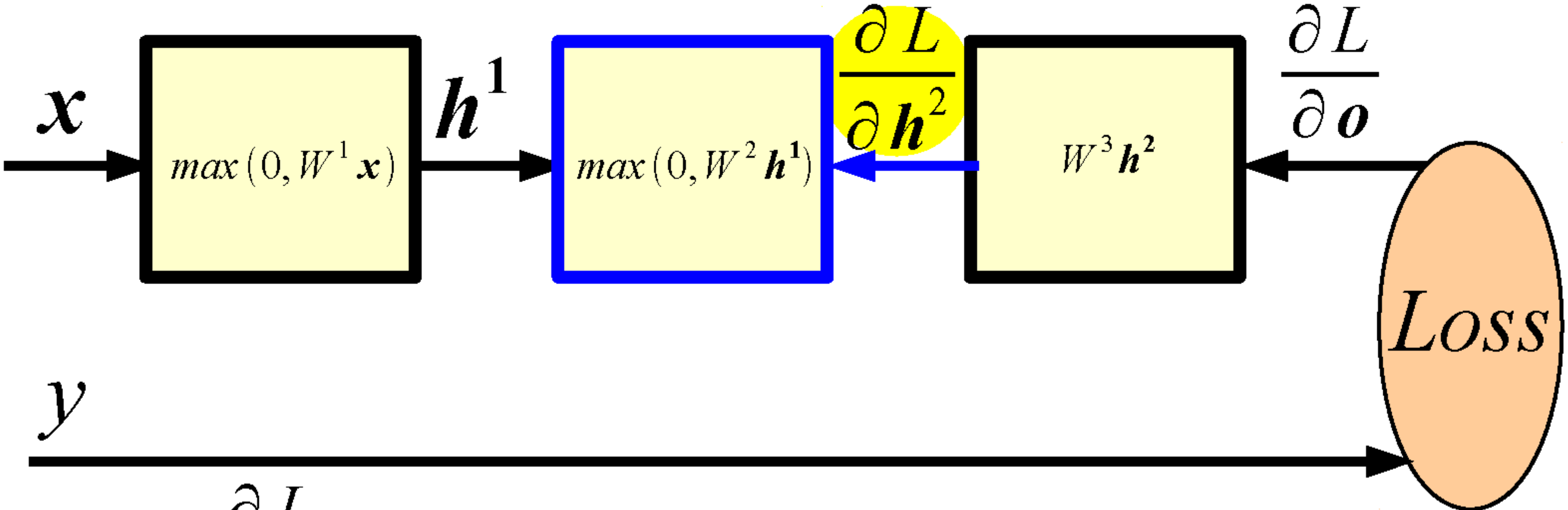


Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

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

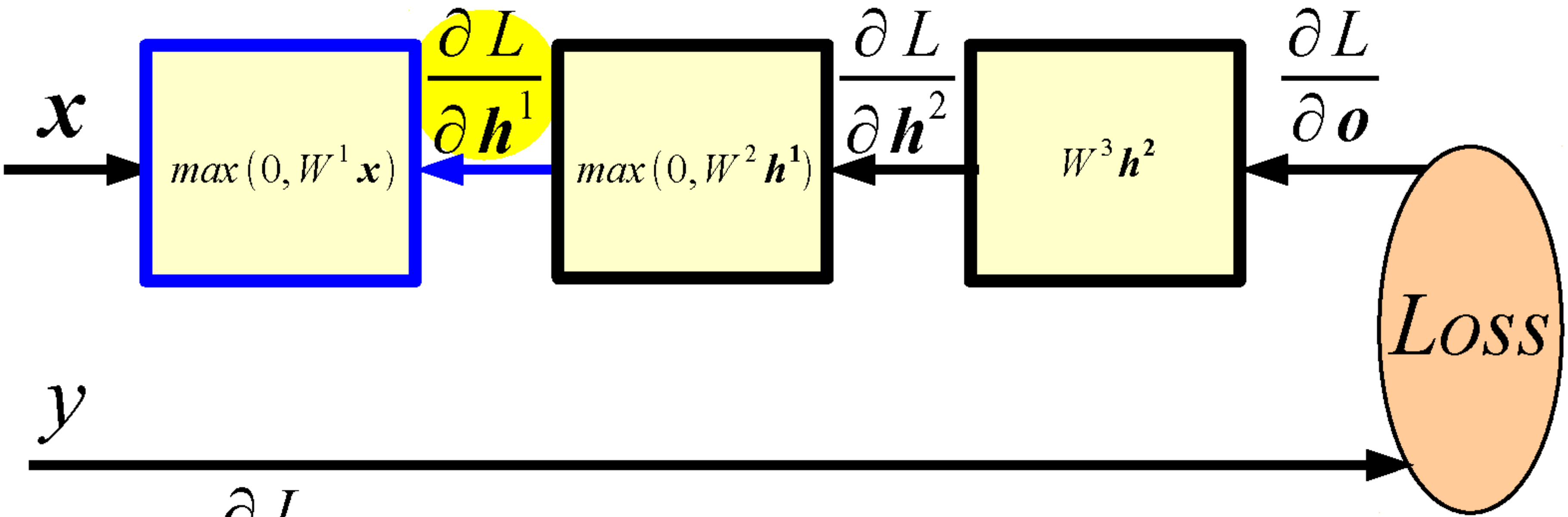
Backward Propagation



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

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

Backward Propagation



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

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

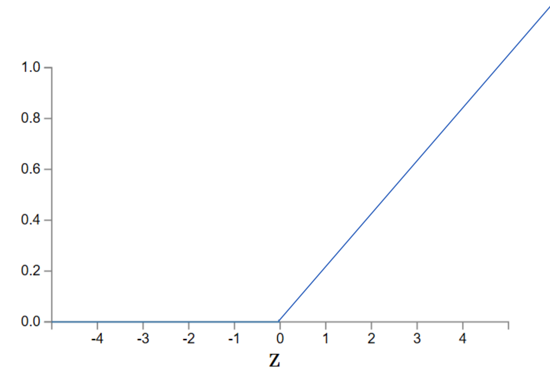
Ranzato

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

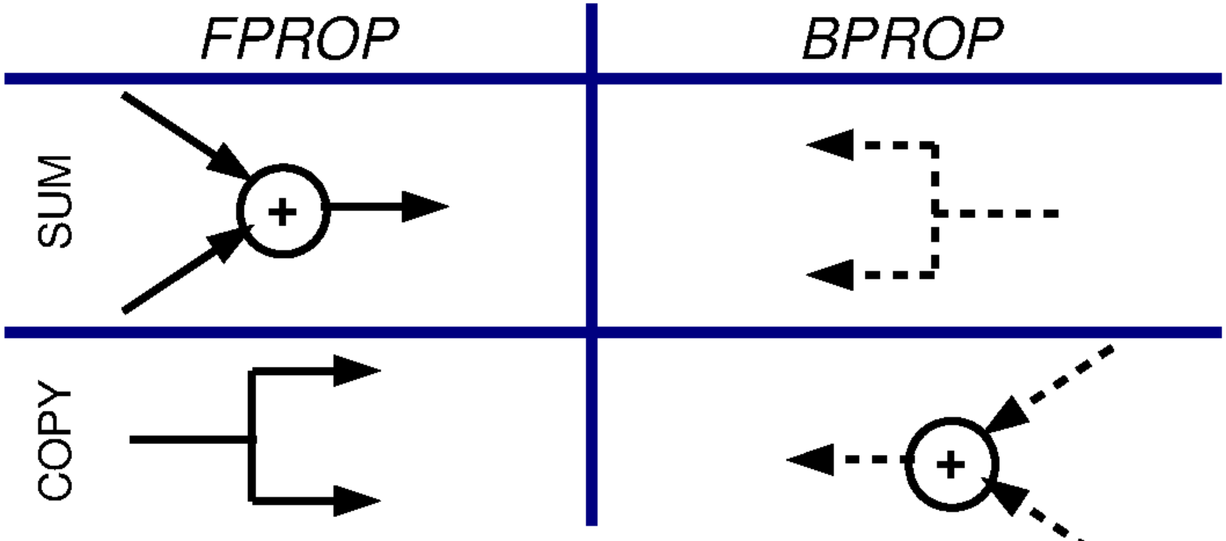
Question: Does BPROP work with ReLU layers only?

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

Question: What's the computational cost of BPROP?

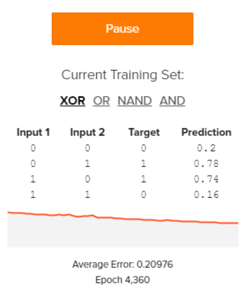
Answer: About twice FPROP (need to compute gradients w.r.t. input and parameters at every layer).

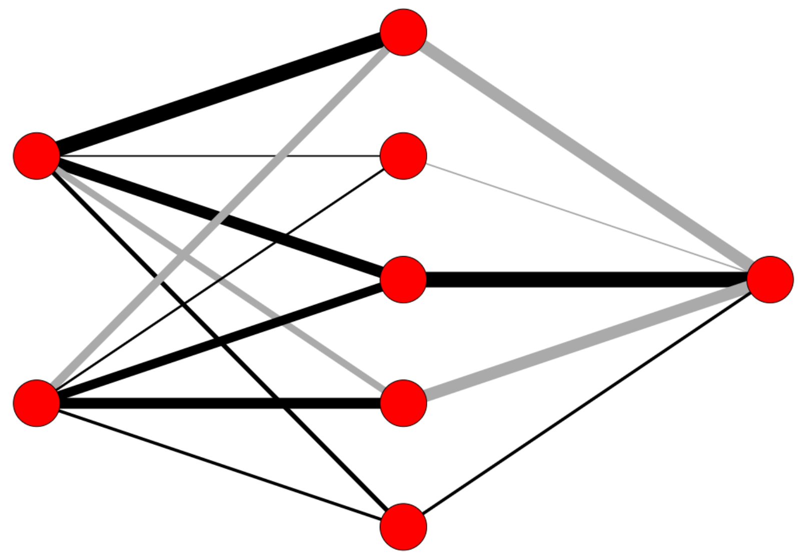
Note: FPROP and BPROP are dual of each other. E.g.,:



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
```

Stochastic Gradient Descent

- Dataset can be too large to strictly apply gradient descent.
- 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*”

Pick starting W and learning rate γ

While not at minimum:

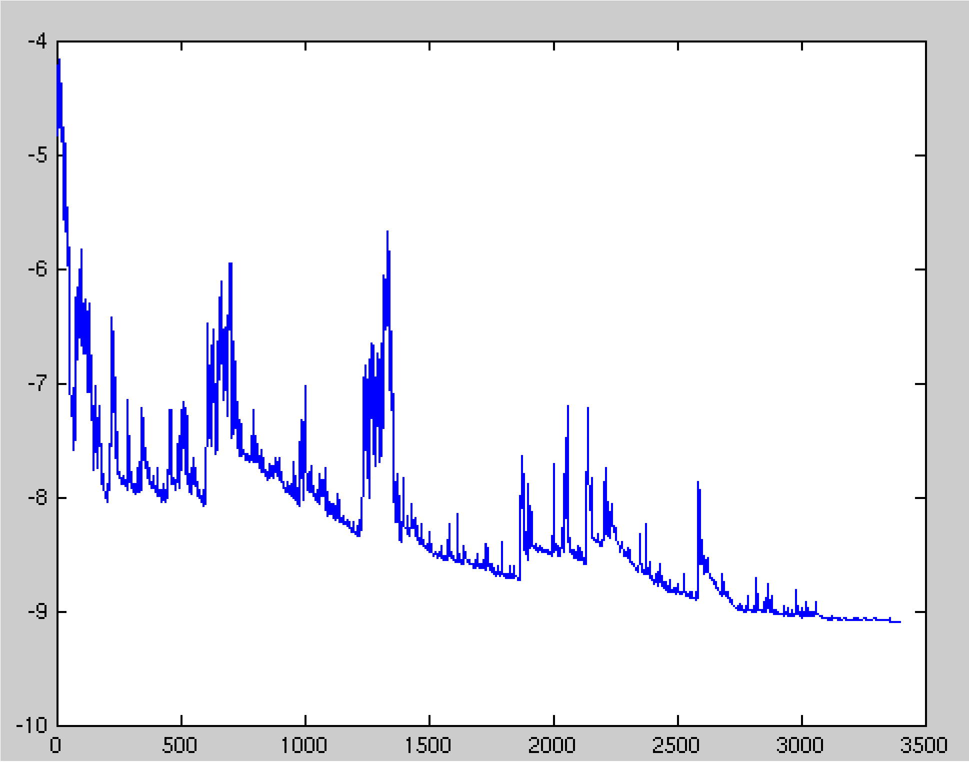
  - Shuffle training set
  - For each data point *i=1…n (maybe as mini-batch)*
    - *Gradient descent*

“Epoch“

Stochastic Gradient Descent

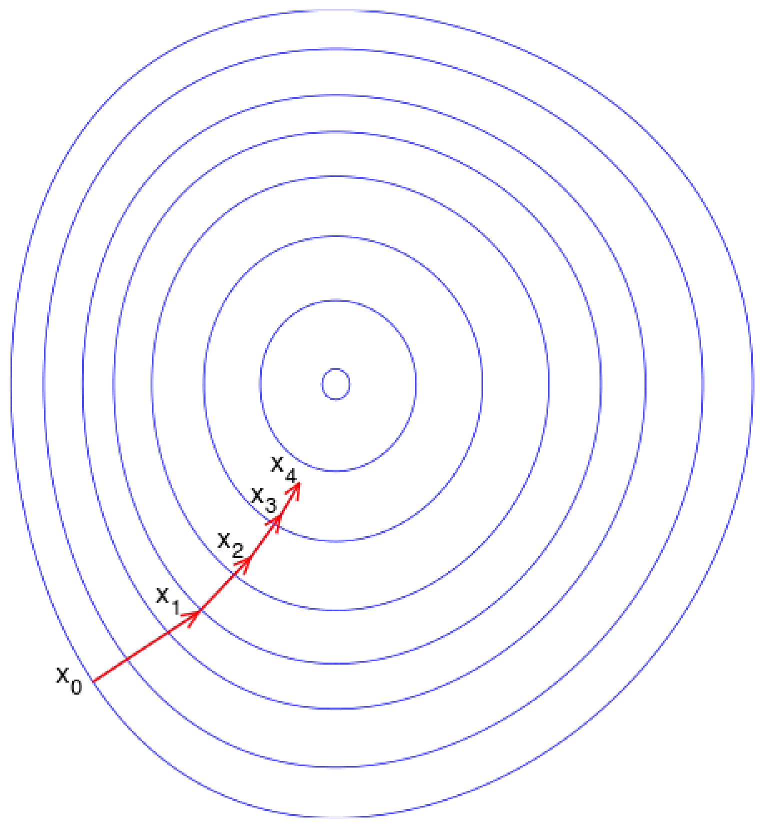
Loss will not always decrease (locally) as training data point is random.

Still converges over time.



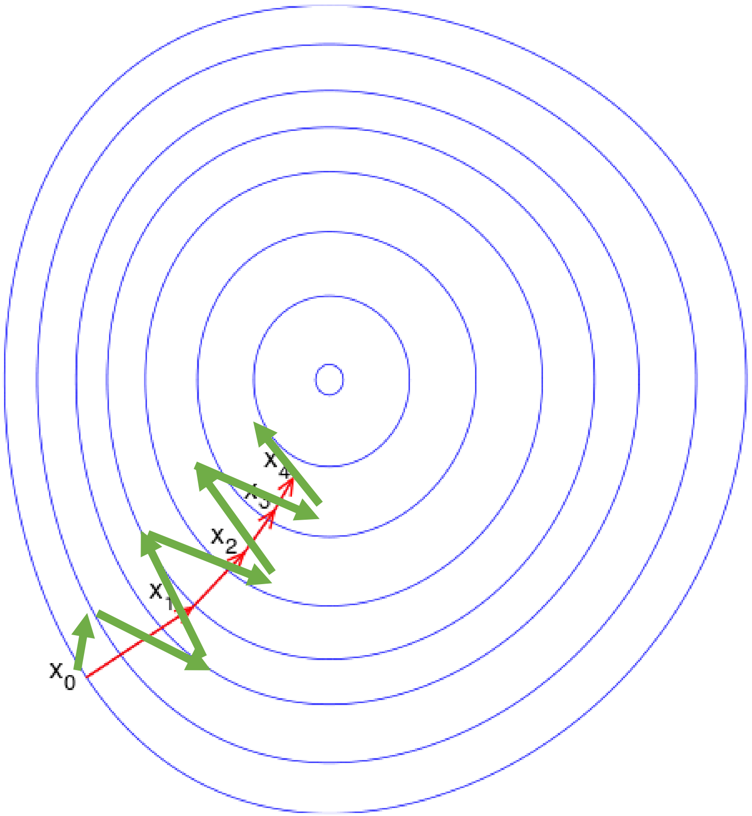
Wikipedia

Gradient descent oscillations



Wikipedia

Gradient descent oscillations



Slow to converge to the (local) optimum

Wikipedia

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*.

http://www.isrtv.com/

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

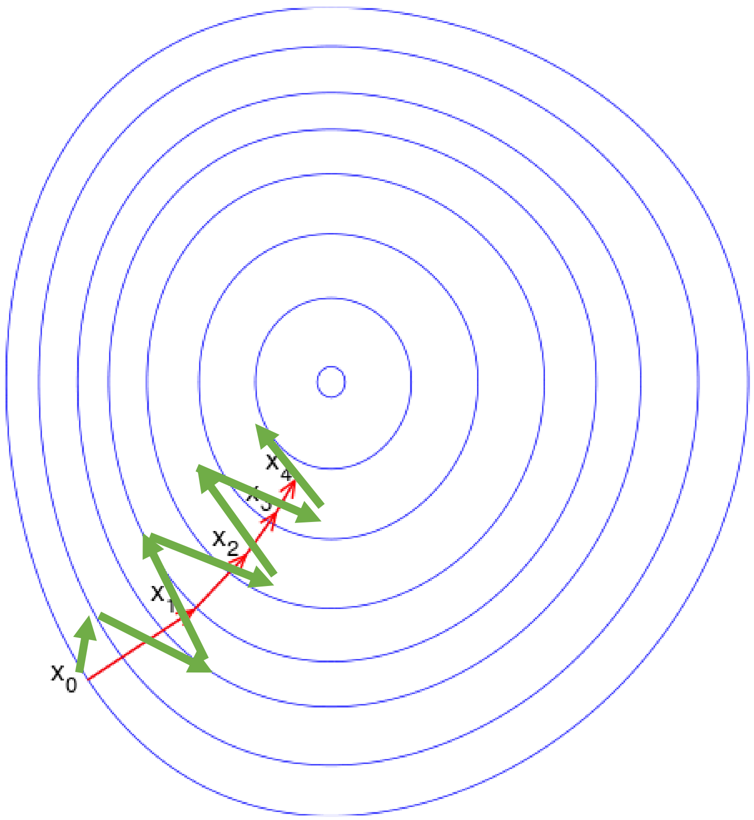
agokasla 6:56 PM
uploaded and commented on this image: image.png

> 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

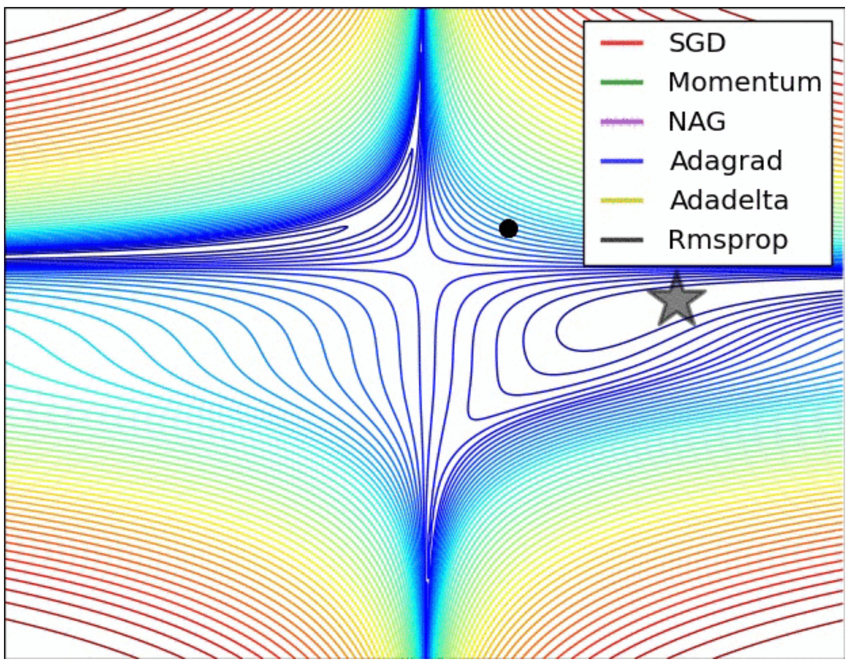


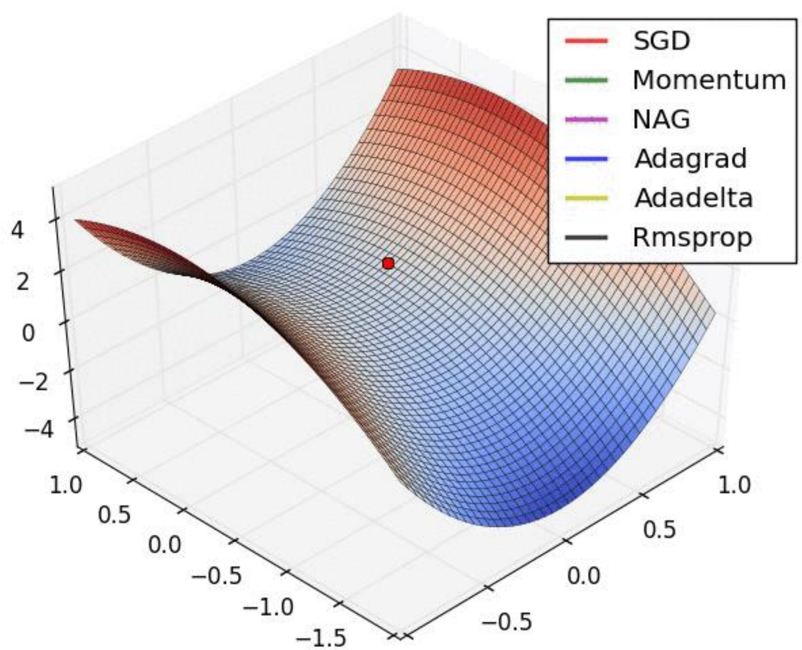
-Less 'ping pong'

-Takes longer to get to the optimum

Wikipedia

Flat regions in energy landscape





Problem of fitting

- Too many parameters = overfitting
- Not enough parameters = underfitting

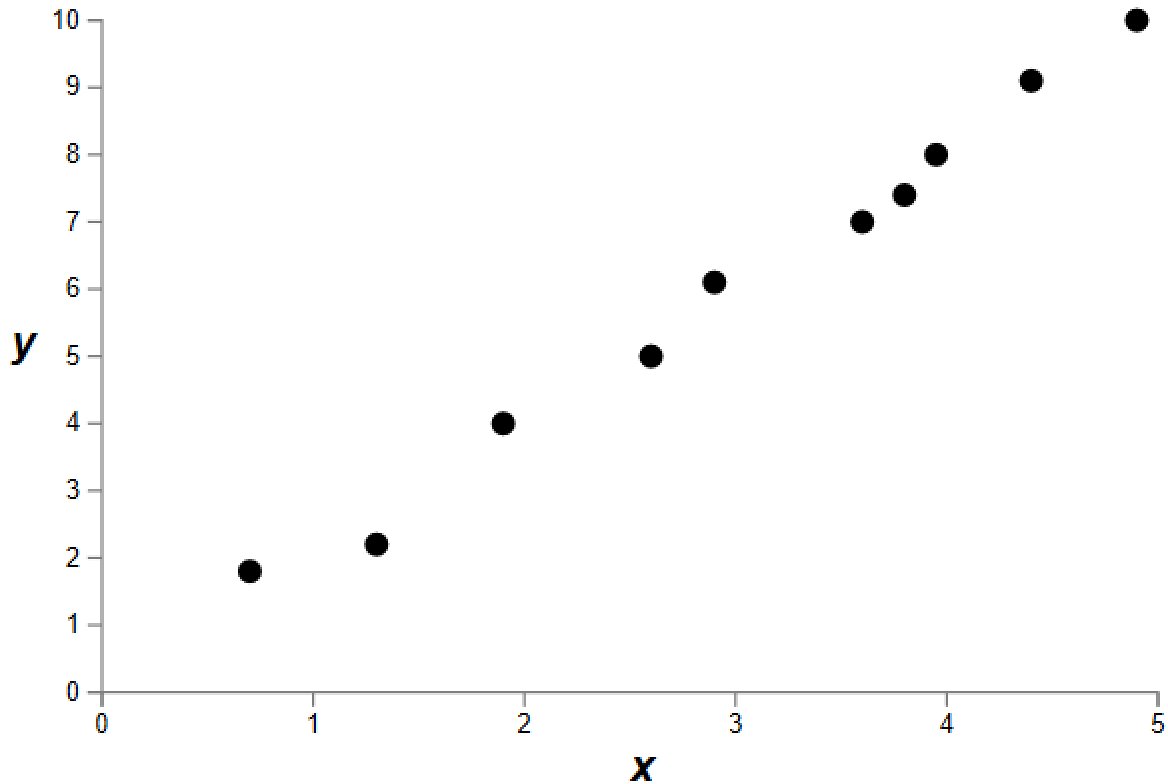
- More data = less chance to overfit

- How do we know what is required?

Regularization

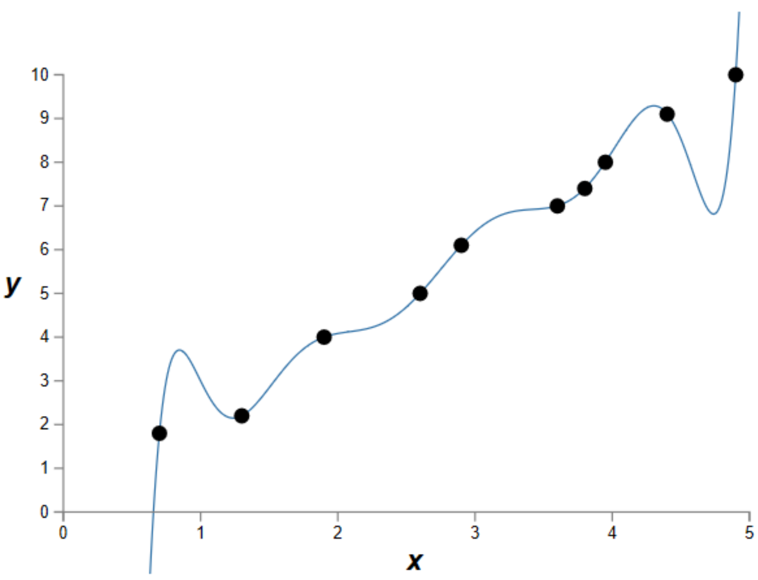
- Attempt to guide solution to *not overfit*
- But still give freedom with many parameters

Data fitting problem

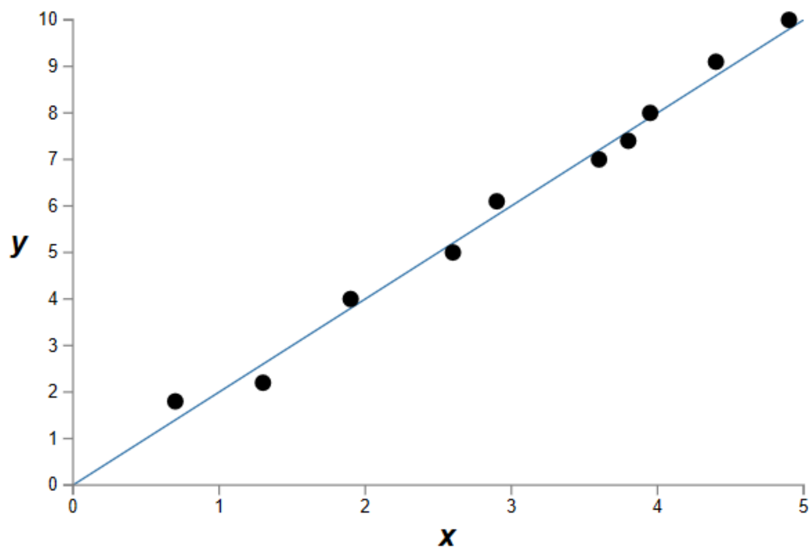


[Nielson]

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



9th order polynomial



1st order polynomial

[Nielson]

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

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

[Nielson]

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

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

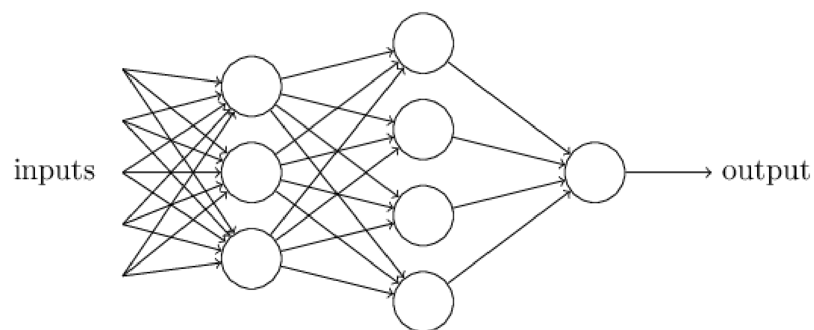
$$C = \underbrace{-\frac{1}{n} \sum_{xj} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]}_{\text{Normal cross-entropy loss (binary classes)}} + \underbrace{\lambda \sum_w w^2}_{\text{Regularization term}}.$$

[Nielson]

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.



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

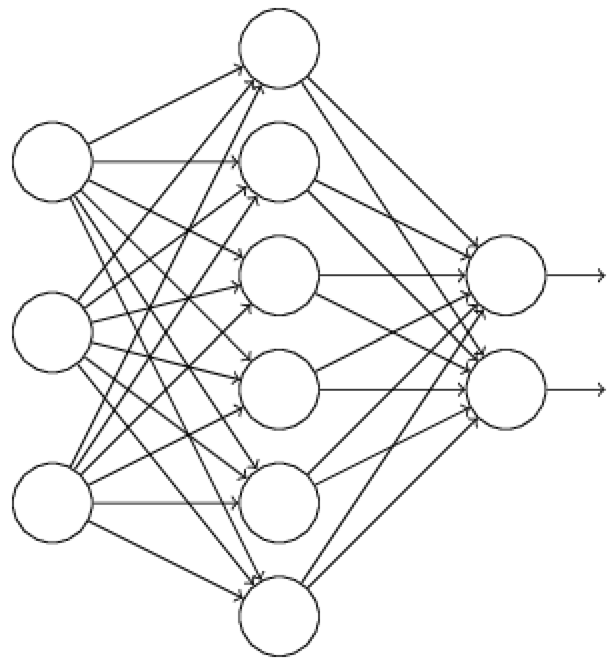
[Nielson]

Regularization

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

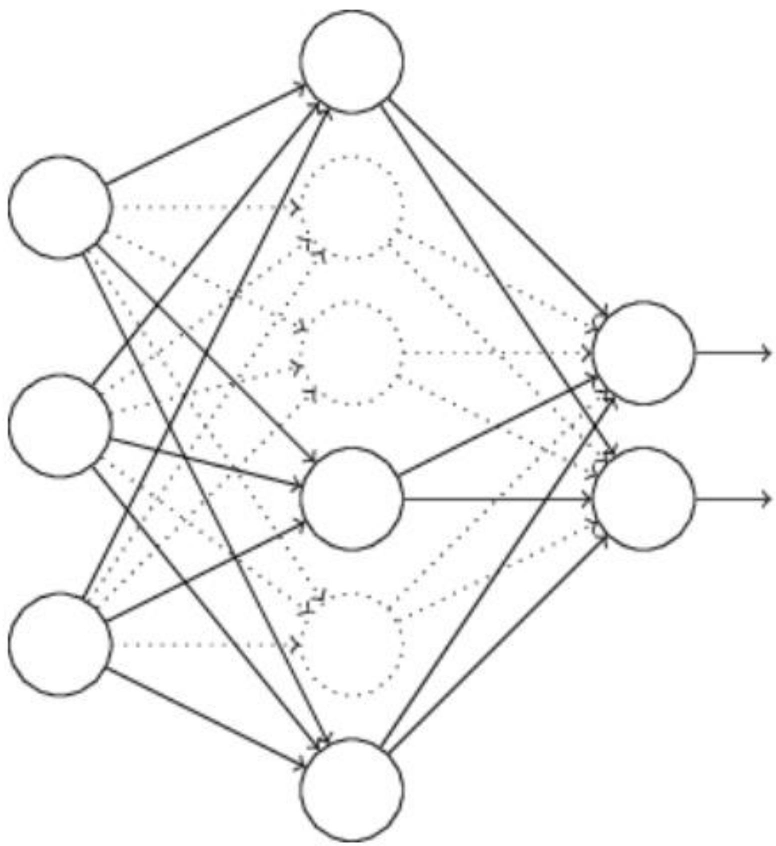
- Why not train 5 different networks with random starts and vote on their outcome?
  - Works fine!
  - Helps generalization because error is averaged.

Regularization: Dropout



[Nielson]

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.

[Nielson]

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