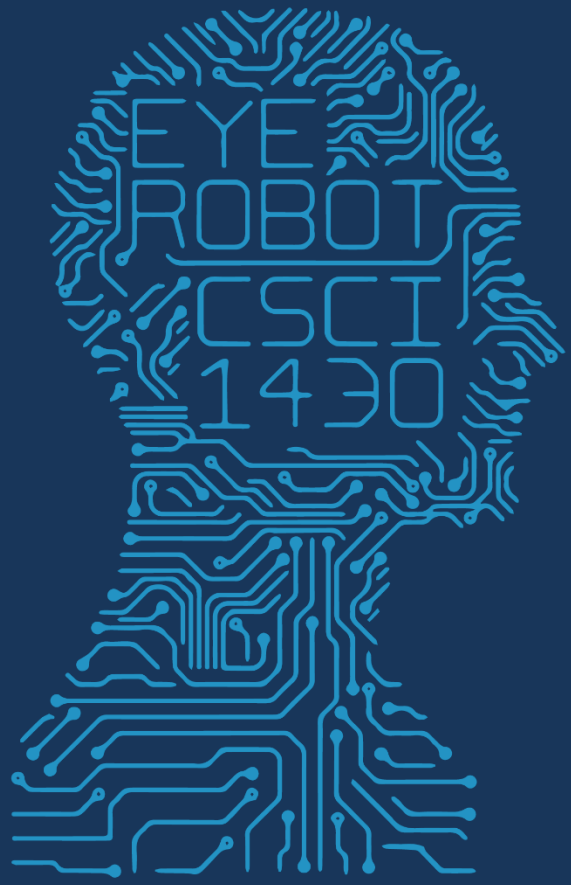
1950

Future Vision



2017 MWF 1pm 368

Computer Vision


Getty

Goals

- Build a classifier which is more powerful at representing complex functions *and* more suited to the learning problem.

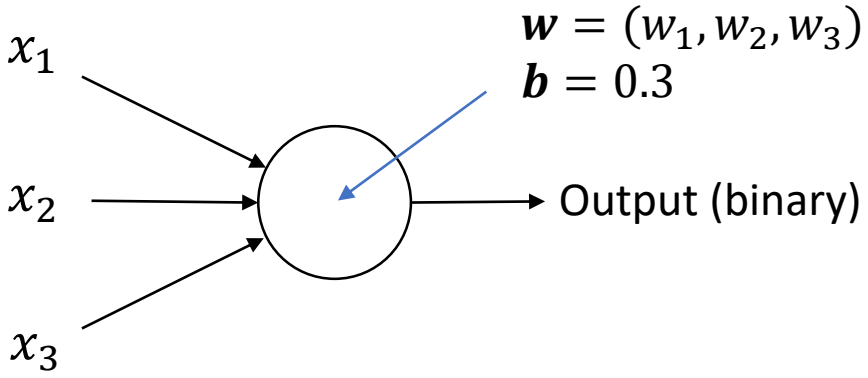
What does this mean?

1. Assume that the *underlying data generating function* relies on a composition of factors.

2. Learn a feature representation that is specific to the dataset.

Neural Networks

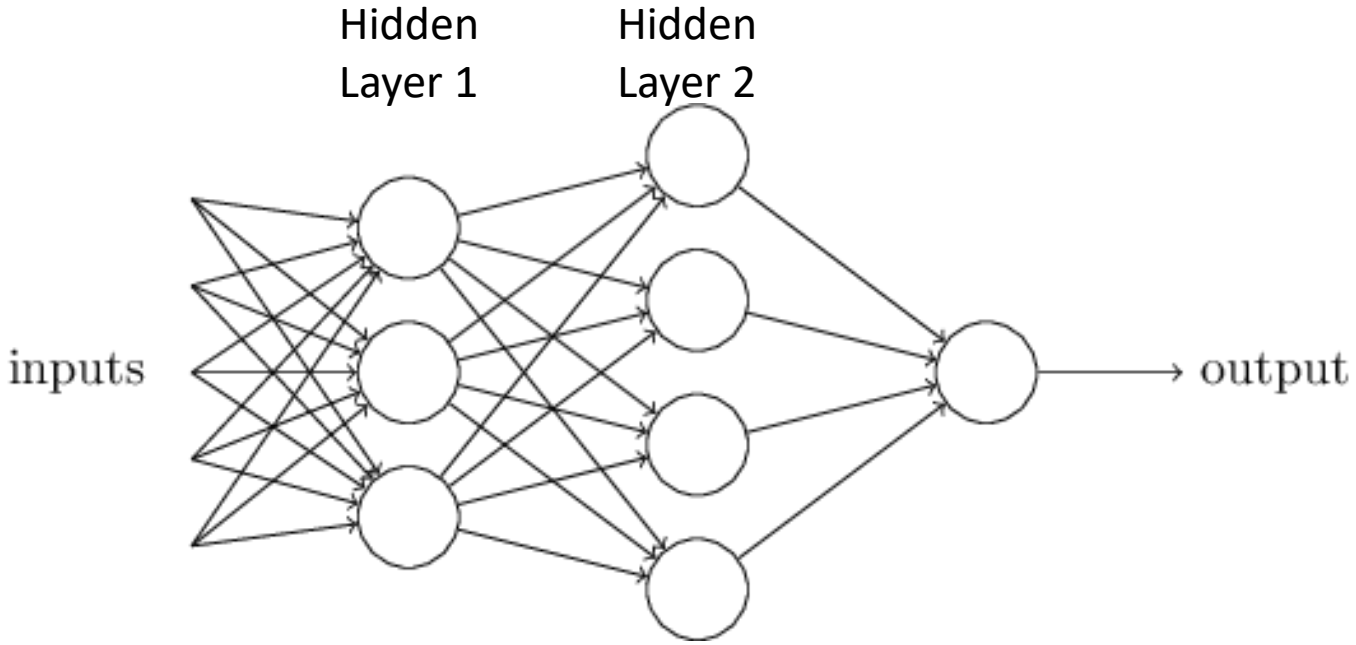
- Basic building block for composition is a *perceptron* (Rosenblatt c.1960)
- Linear classifier – vector of weights *w* and a 'bias' *b*



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

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

Composition

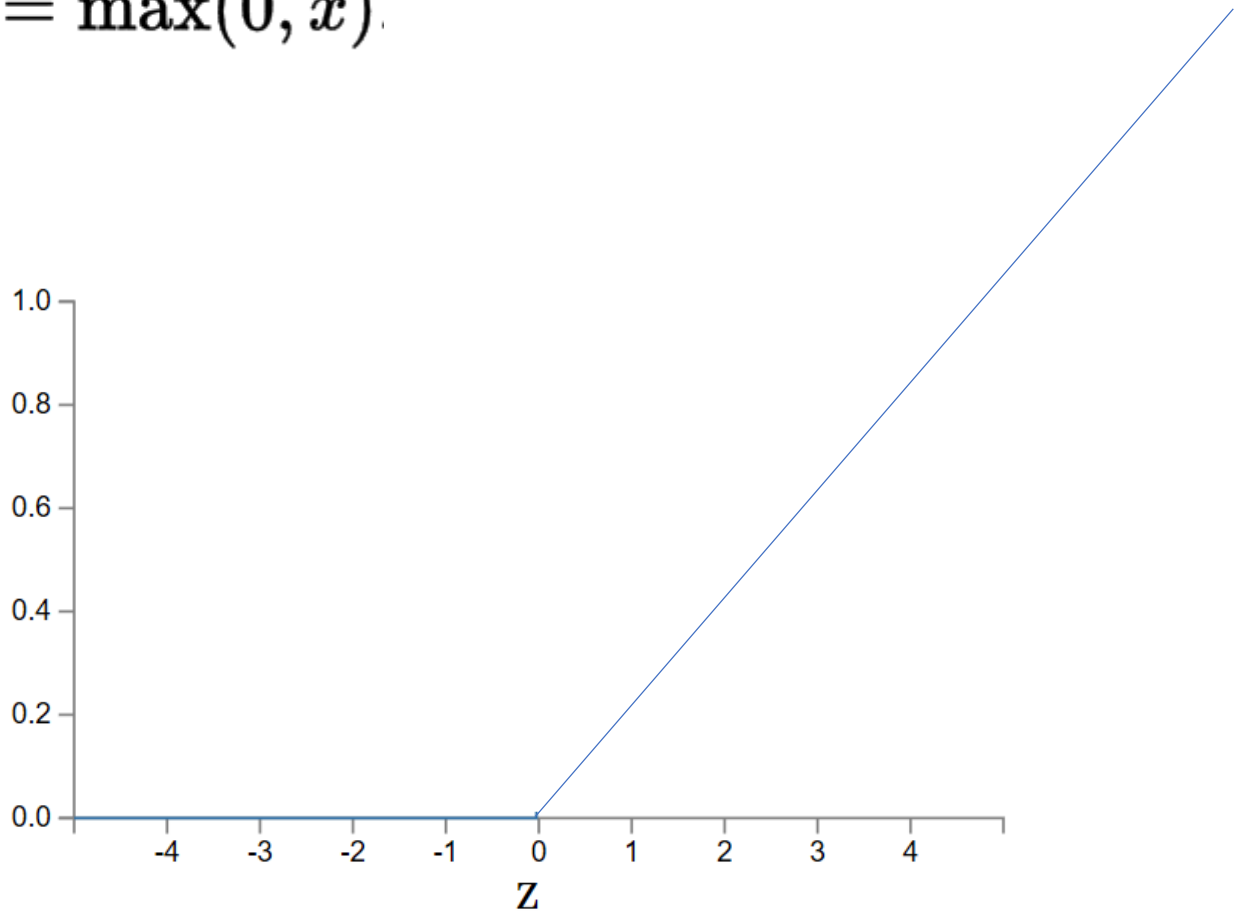


Layers that are in between the input and the output are called *hidden layers*, because we are going to *learn* their weights via an optimization process.

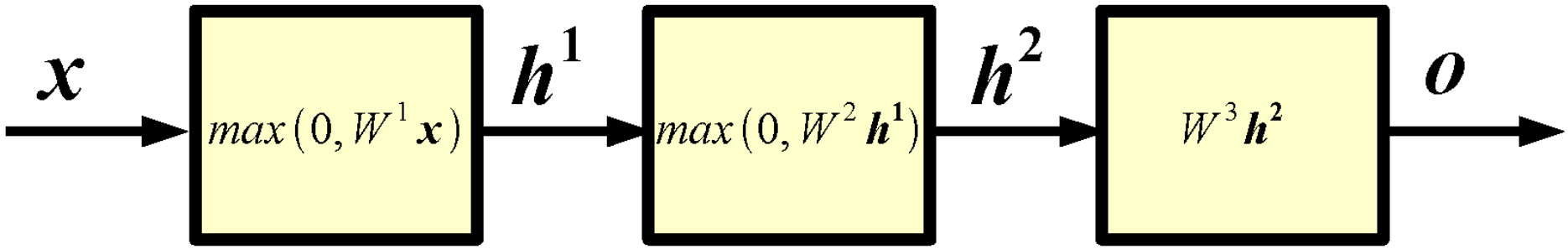
Nielson

Rectified Linear Unit

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



Neural Networks: example



x input

h^1 1-st layer hidden units

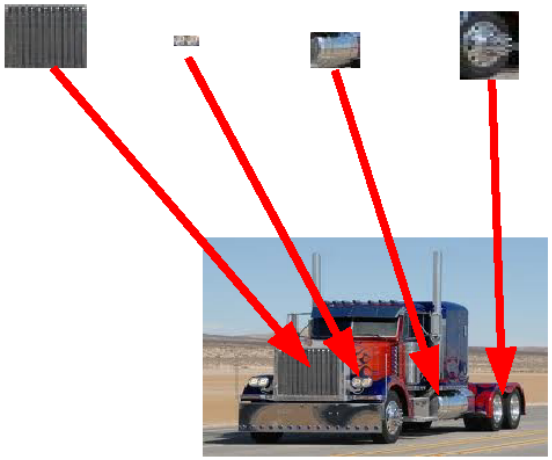
h^2 2-nd layer hidden units

o output

Example of a 2 hidden layer neural network (or 4 layer network, counting also input and output).

Interpretation of many layers

[0 0 **1** 0 0 0 0 **1** 0 0 **1** **1** 0 0 **1** 0 …] truck feature

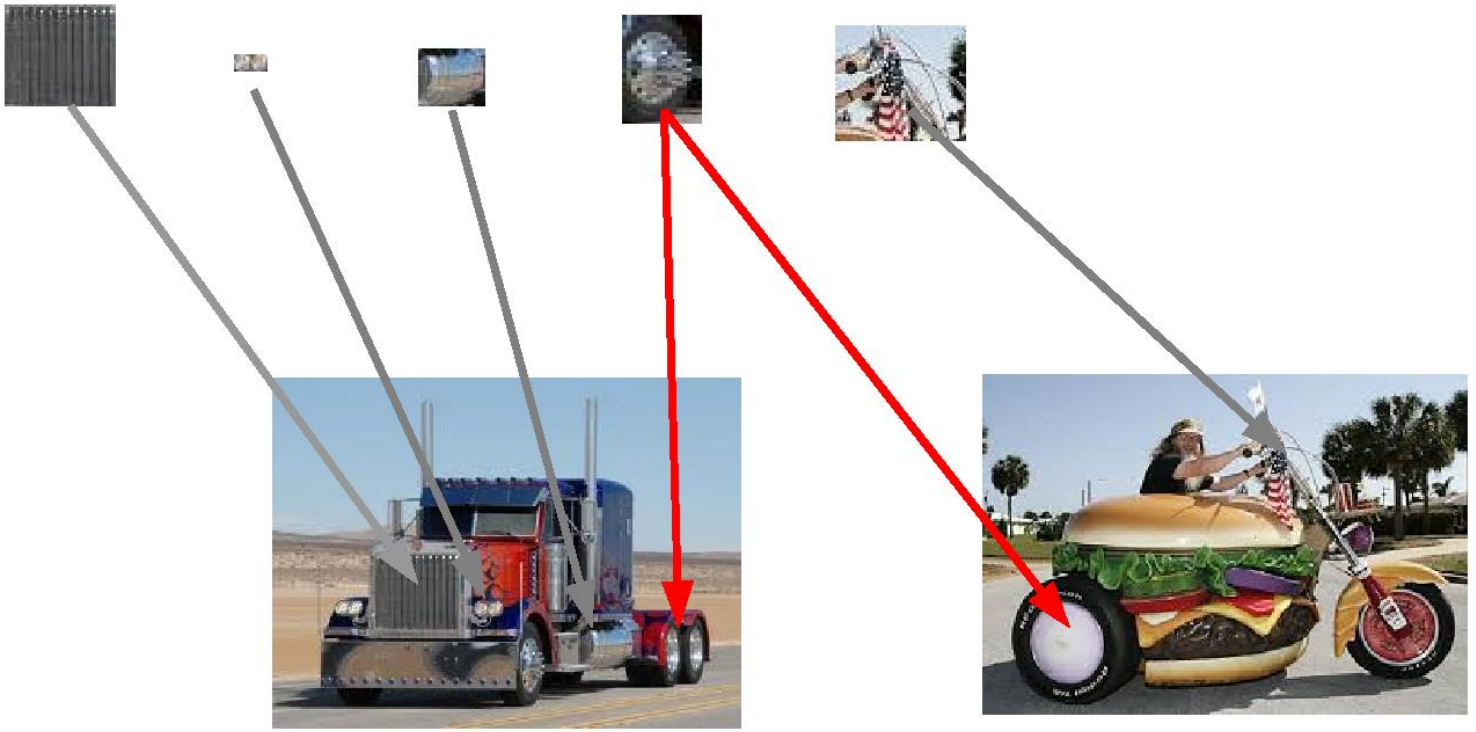


Exponentially more efficient than a
1-of-N representation (a la k-means)

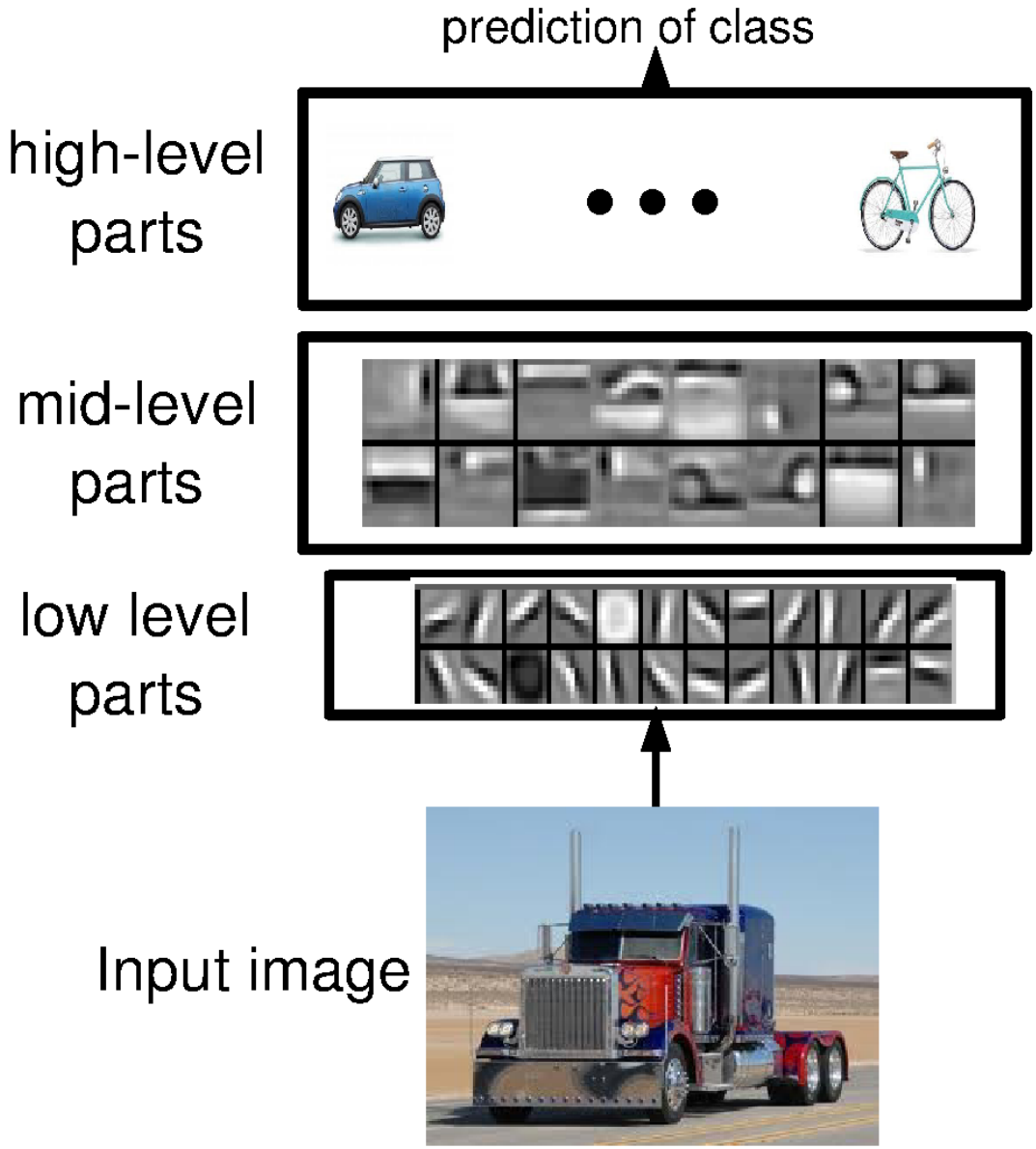
Interpretation

[1 1 0 0 0 1 0 **1** 0 0 0 0 1 1 0 1…] motorbike

[0 0 1 0 0 0 0 **1** 0 0 1 1 0 0 1 0 …] truck

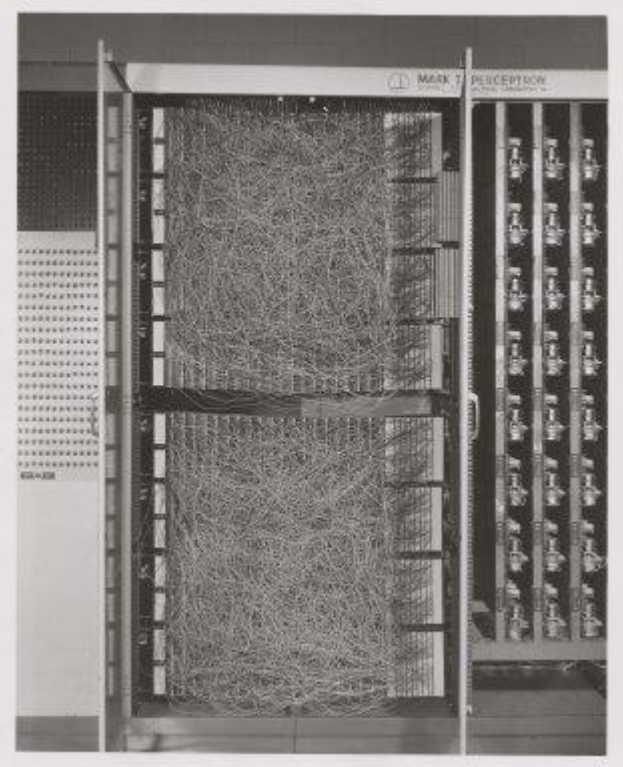


Interpretation



- distributed representations
- feature sharing
- compositionality

Lee et al. “Convolutional DBN's ...” ICML 2009



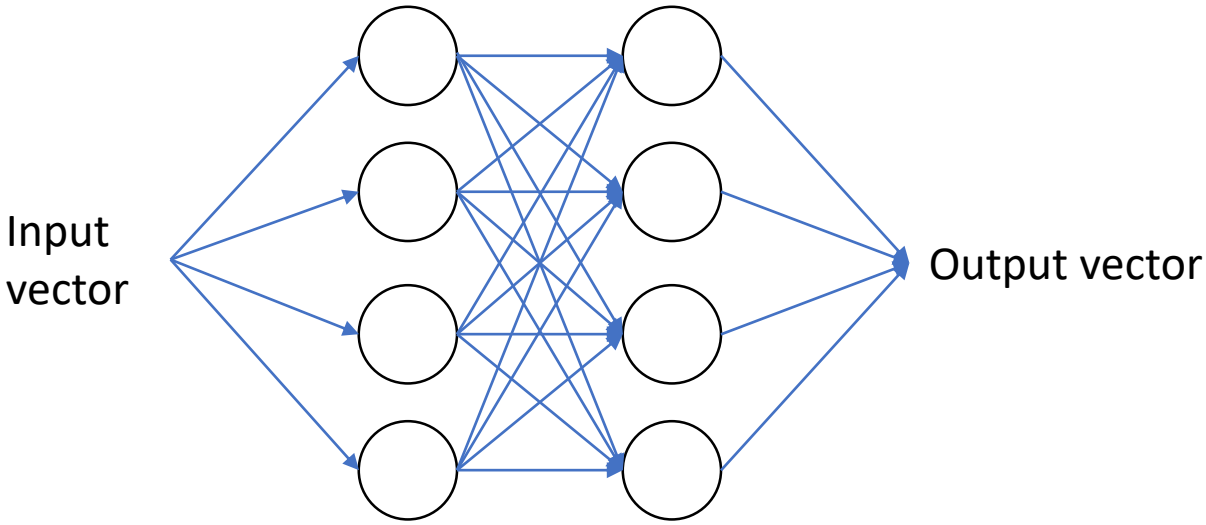
Mark 1 Perceptron
c.1960

20x20 pixel
camera feed

Wikipedia

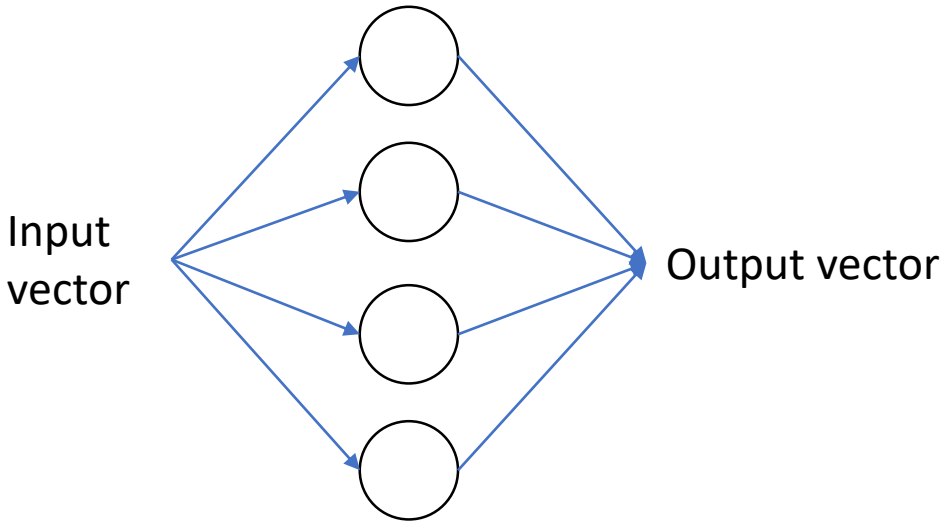
Does anyone pass along the weight unthresholded? I.E., a real output instead of binary?

No – this is linear chaining.



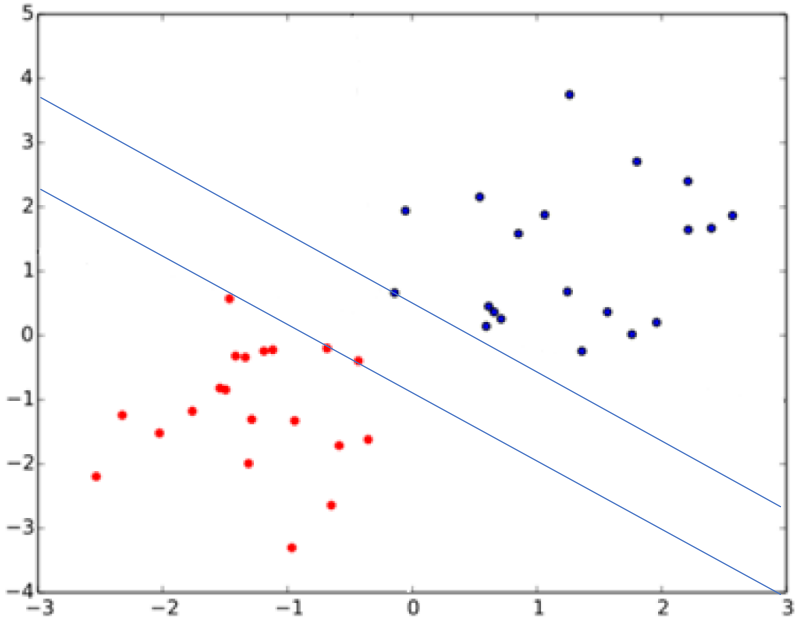
Does anyone pass along the weight unthresholded? I.E., a real output instead of binary?

No – this is linear chaining.



What is the relationship between SVMs and perceptrons?

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

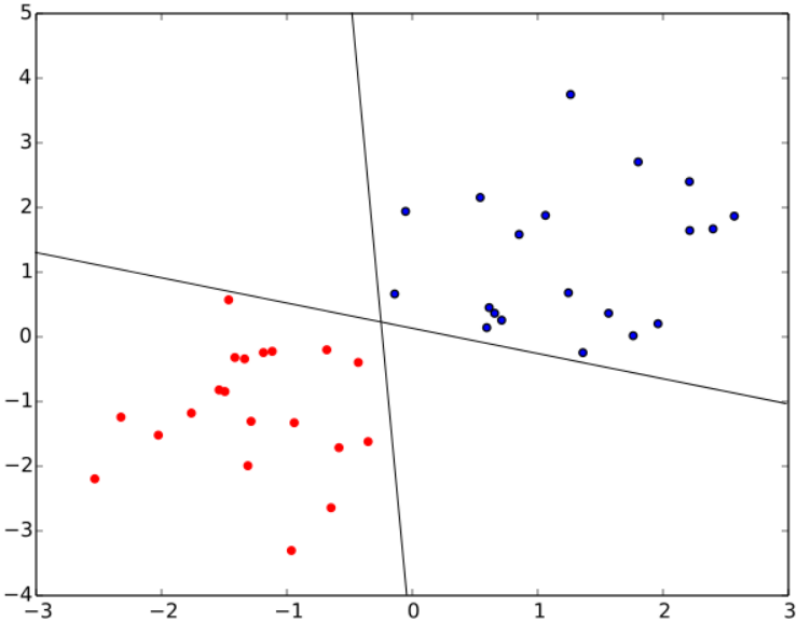


What is the relationship between SVMs and perceptrons?

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

- A perceptron does not.
E.G., both of these perceptron classifiers are equivalent.

- ‘Perceptron of optimal stability’ is used in SVM.



What is the relationship between SVMs and perceptrons?

-SVMs apply the kernel trick to compute distances via the dot product in a higher-dimensional space.

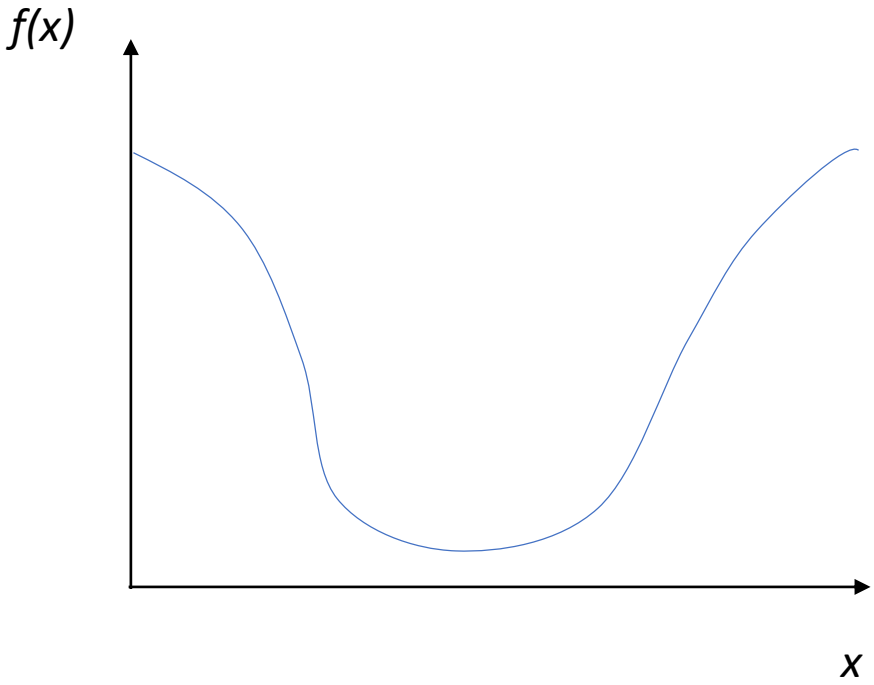
- A perceptron does not.

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

Training Neural Networks

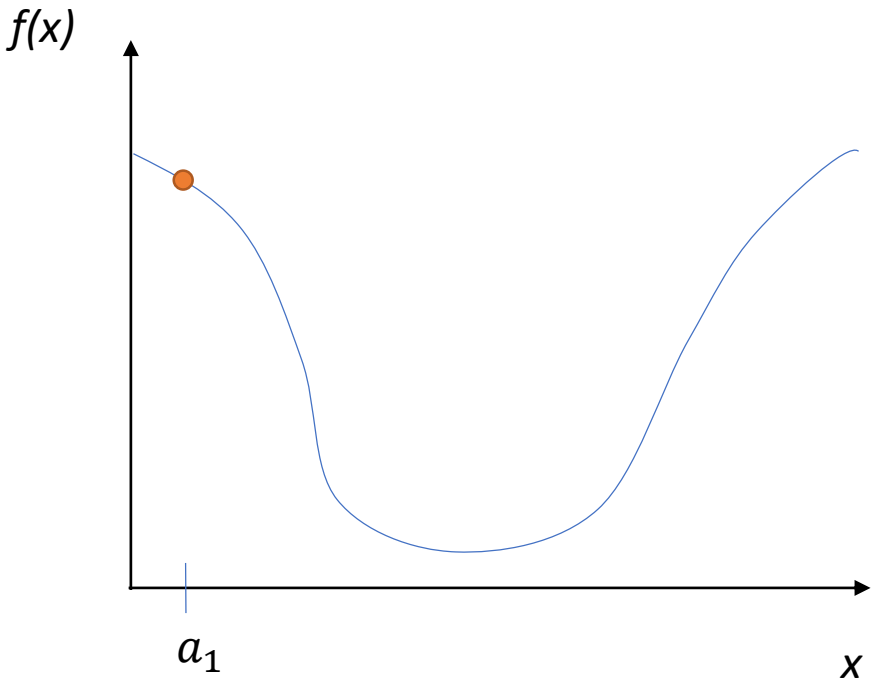
Learning the weight matrices ***W***

Gradient descent



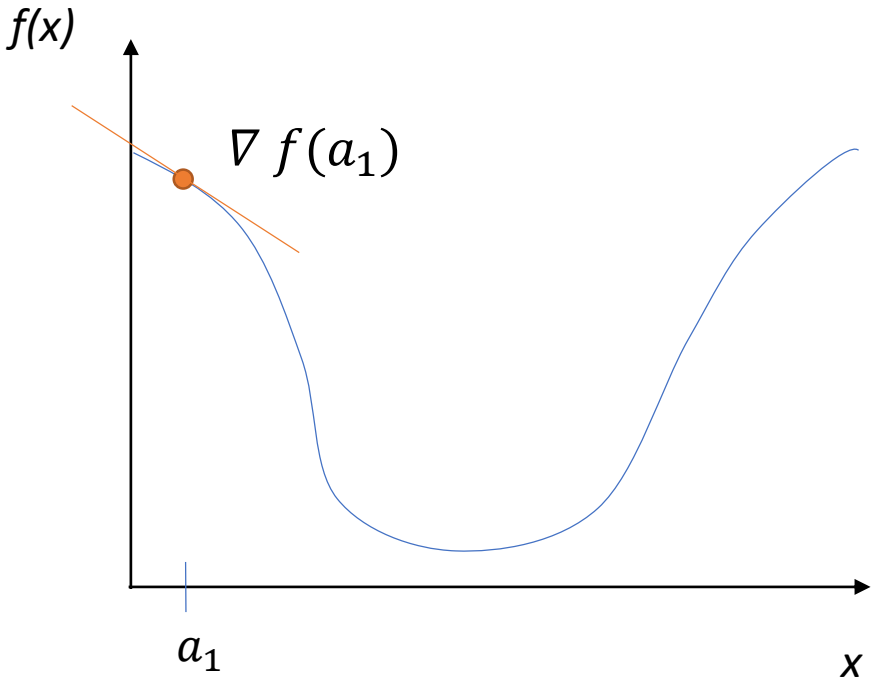
General approach

Pick random starting point.



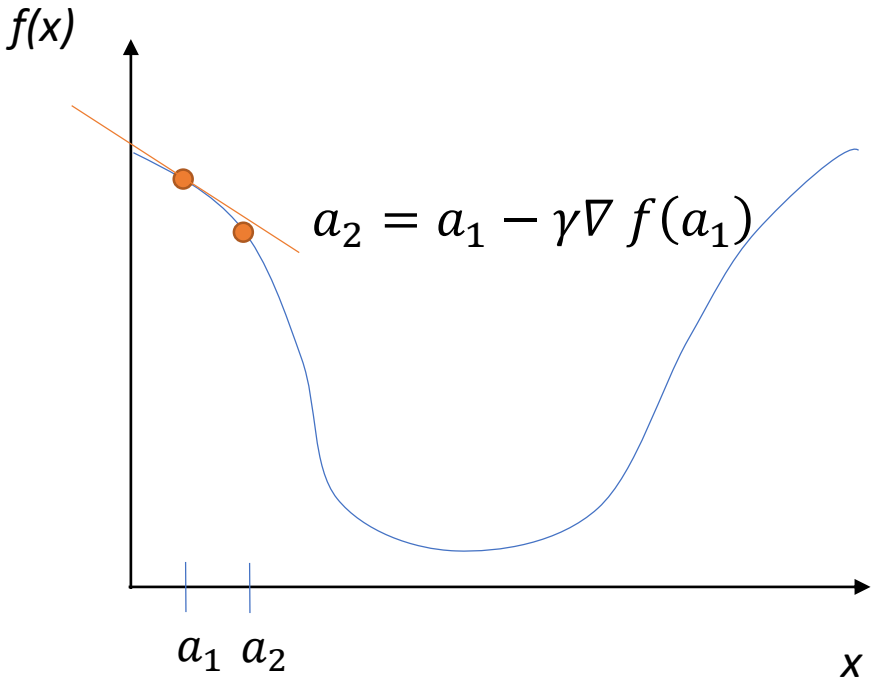
General approach

Compute gradient at point (analytically or by finite differences)



General approach

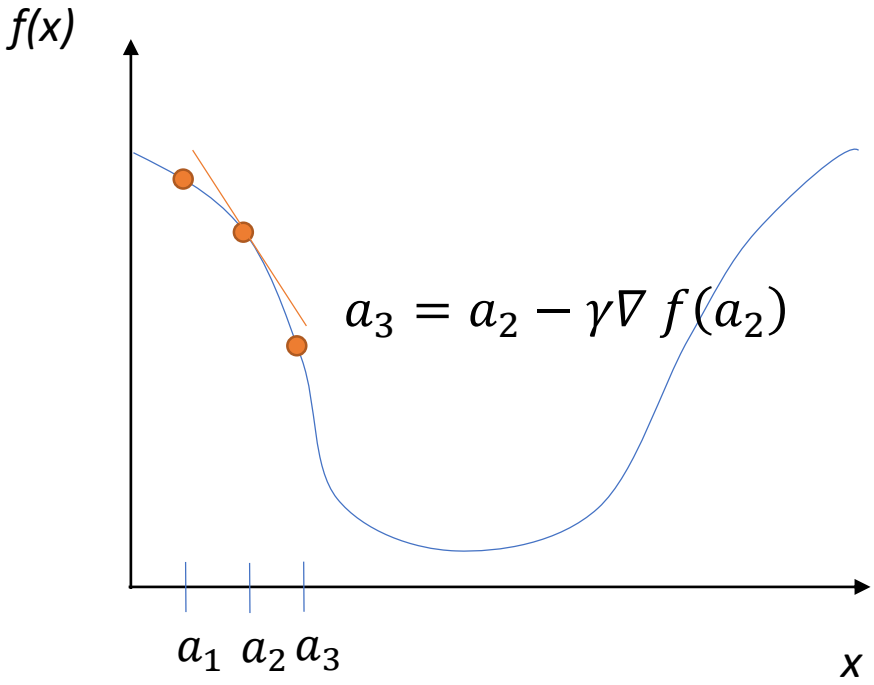
Move along parameter space in direction of negative gradient



γ = amount to move
= *learning rate*

General approach

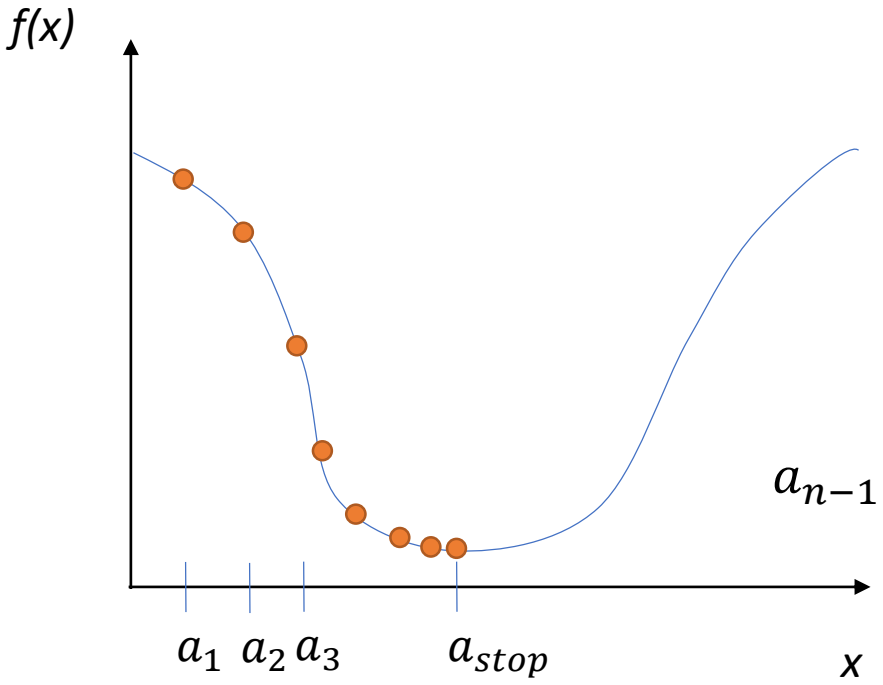
Move along parameter space in direction of negative gradient.



γ = amount to move
= *learning rate*

General approach

Stop when we don't move any more.



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.

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

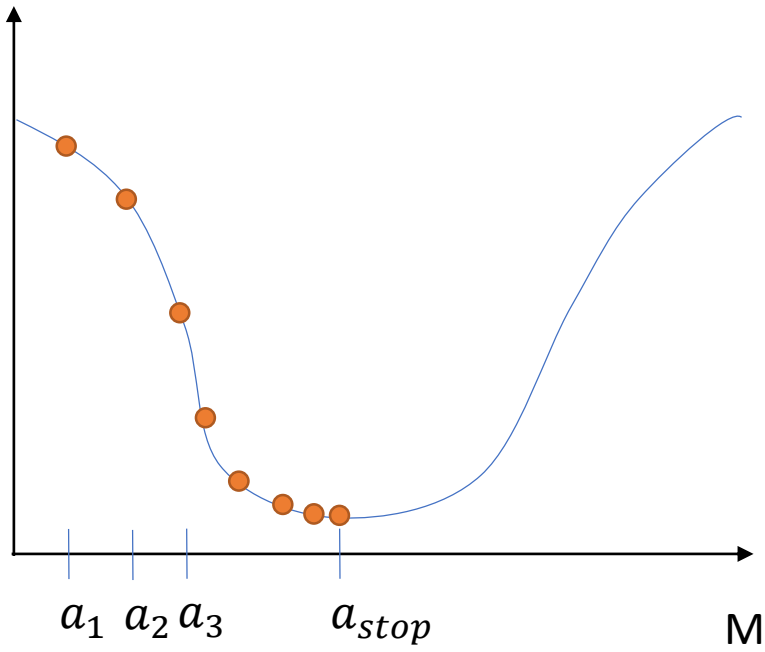
Train NN with Gradient Descent

- 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).

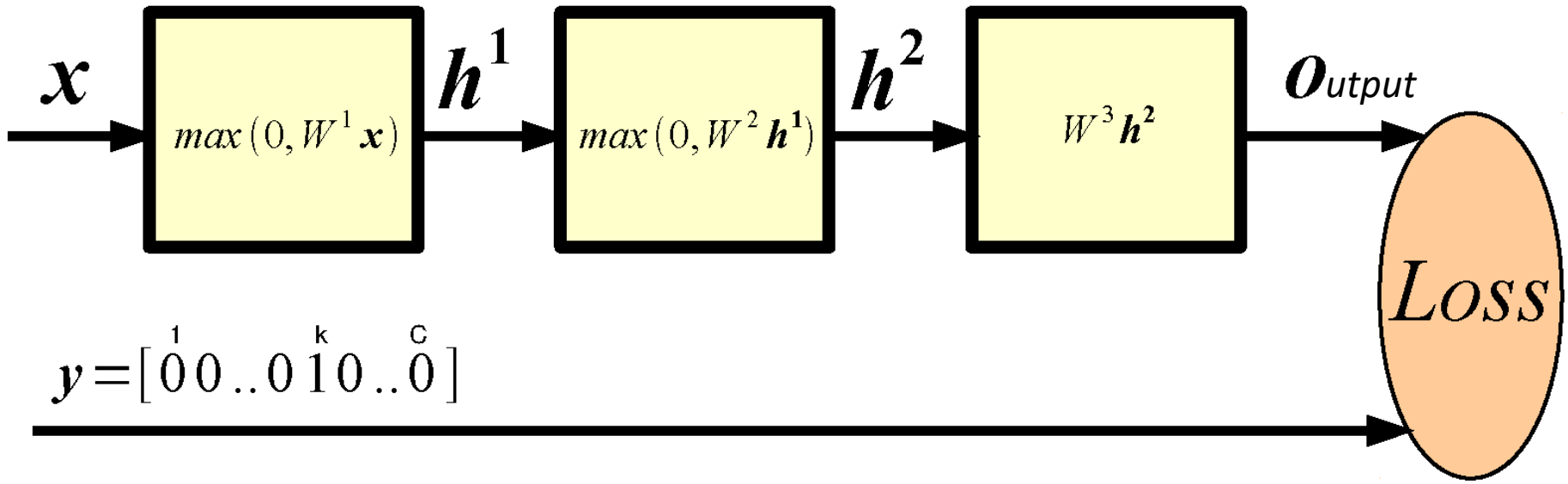
Train NN with Gradient Descent

Loss function
(Evaluate NN
on training data)



Model parameters
(perceptron weights)

How Good is a Network?



What is an appropriate loss?

- Compare training class to output class
- Zero-one loss (per class)

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

- Is it good?

What is an appropriate loss?

- Compare training class to output class
- Zero-one loss (per class)

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

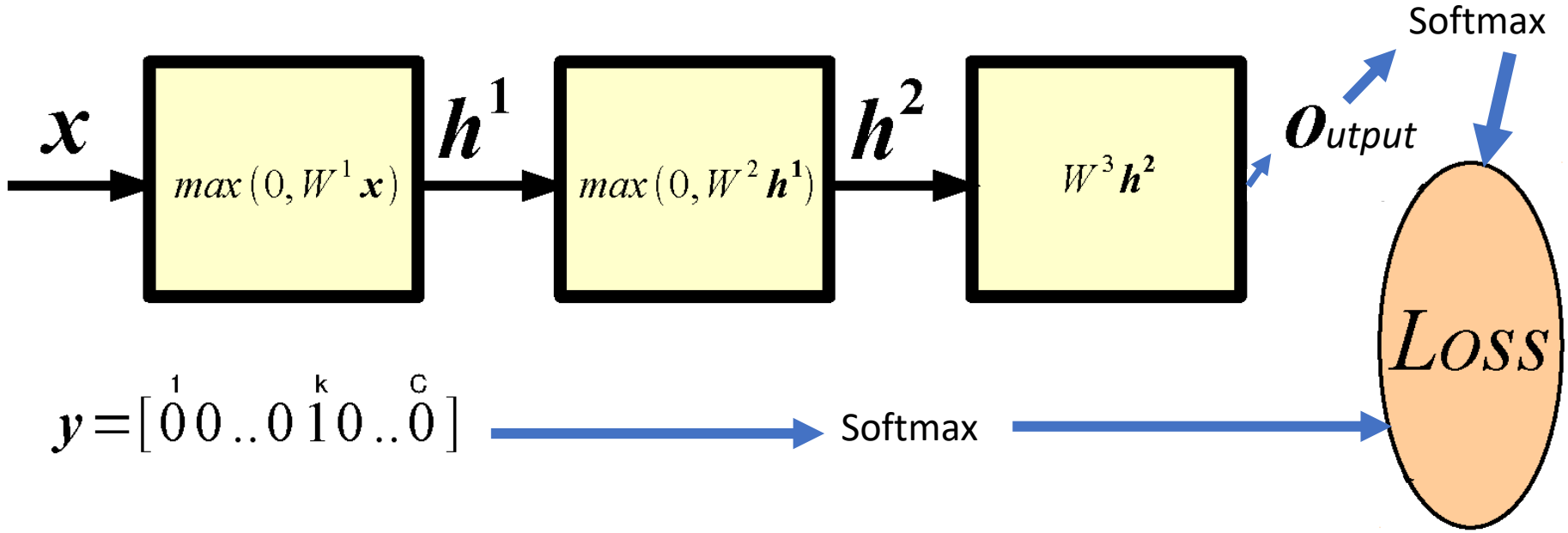
- Is it good?
  - Nope – it's a step function.
  - I need to compute the gradient of the loss.
  - This loss is not differentiable, and 'flips' easily.

But we have binary outputs

- Special function on last layer for classification
- '*Softmax*':
  - "squashes" a *K*-dimensional vector **z** of arbitrary real values to a *K*-dimensional vector σ (**z**) of real values in the range (0, 1) that add up to 1.
  - I.E., 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):

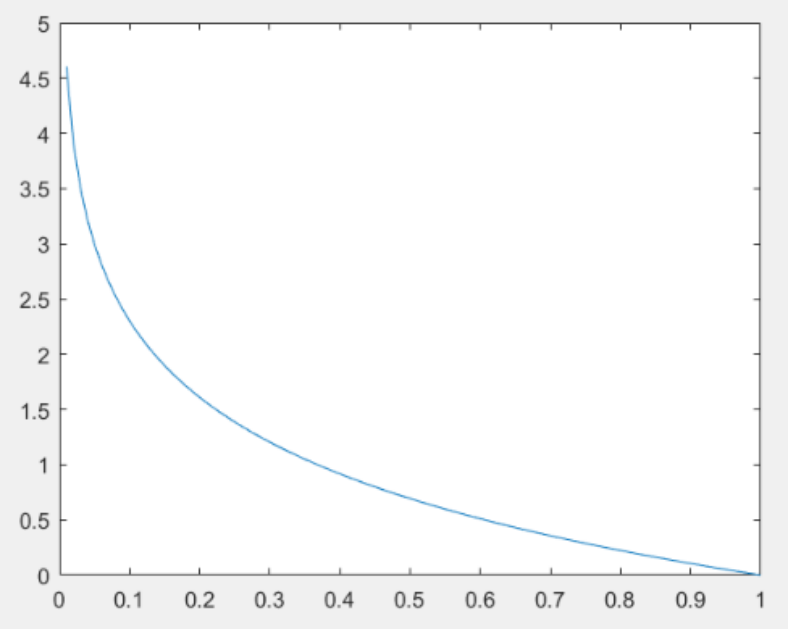
$$p(c_k = 1 | \boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

Cross-entropy loss function

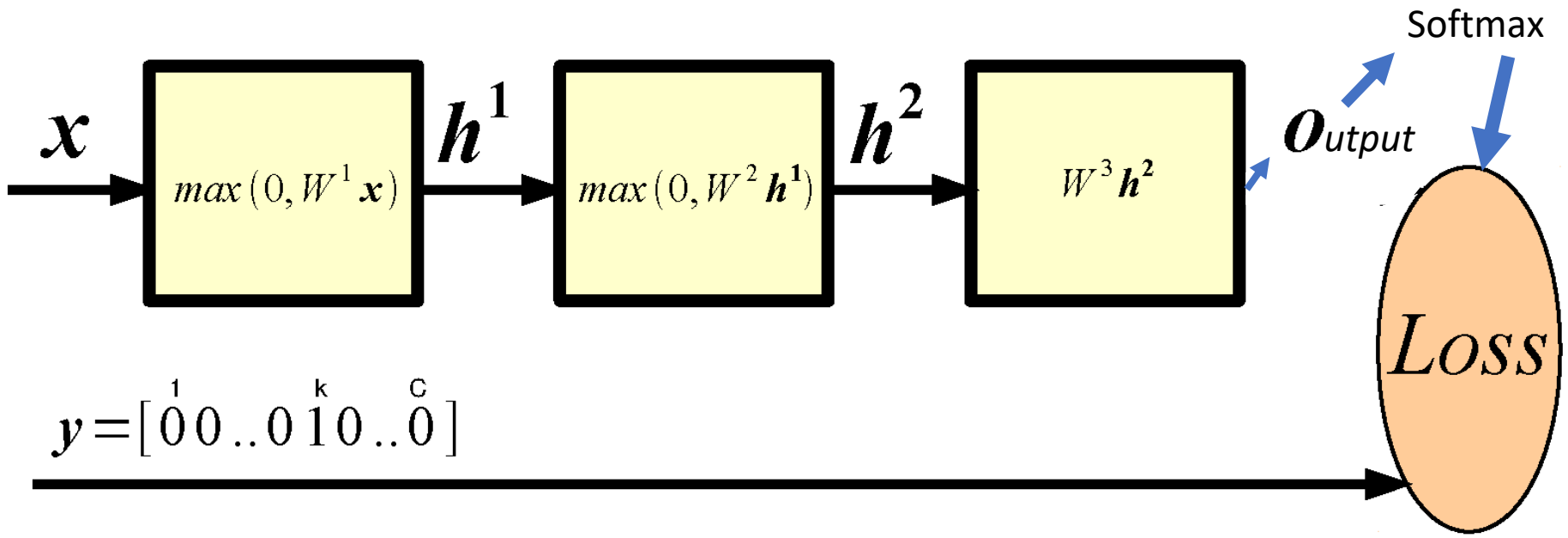
- Negative log-likelihood

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

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



How Good is a Network?



Probability of class k given input (softmax):

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

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

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


Getty

Training

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

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

Training

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

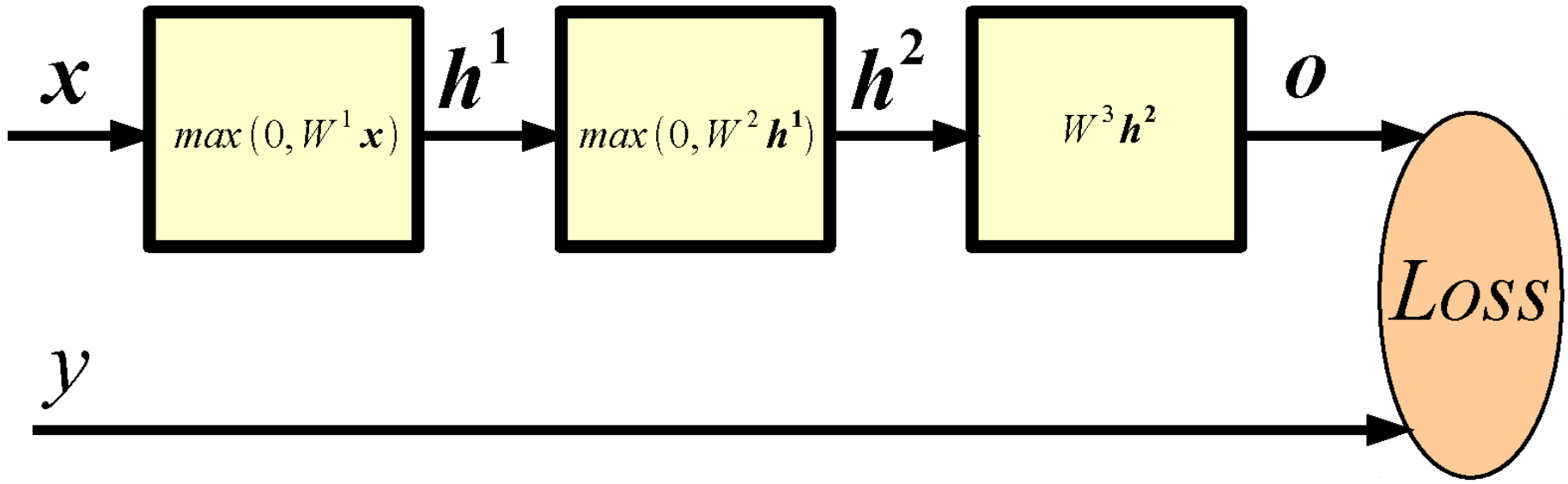
$$\boldsymbol{\theta}^* = arg\,min_{\boldsymbol{\theta}} \sum_{n=1}^{P} L(\boldsymbol{x}^n, y^n; \boldsymbol{\theta})$$

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

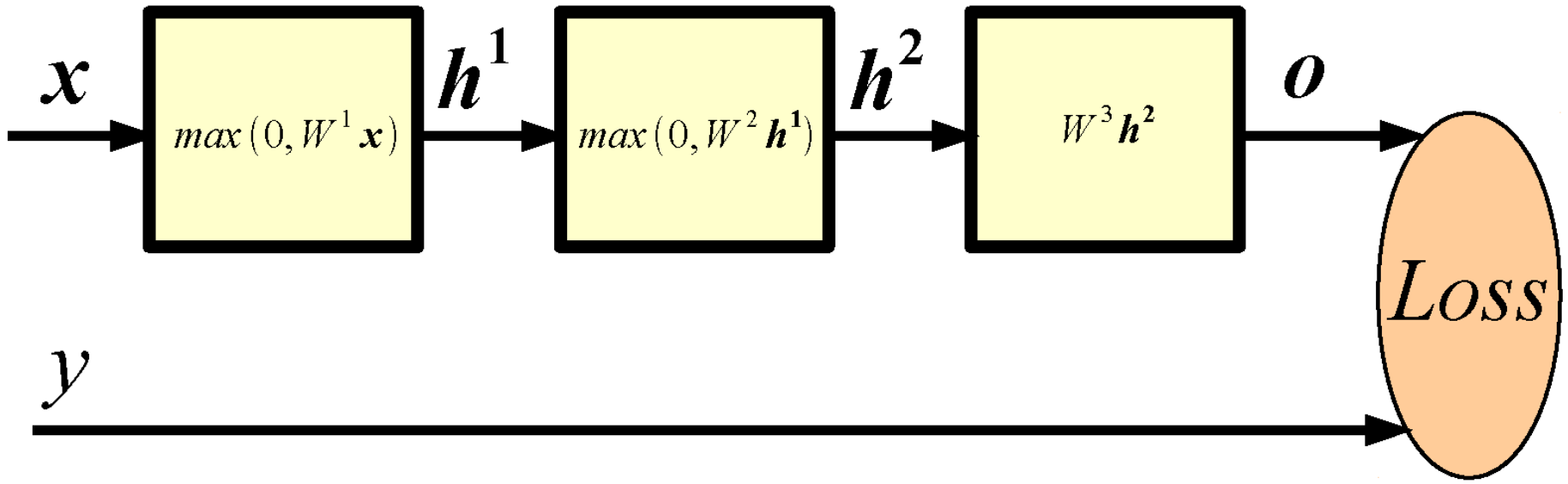


Let's say we want to decrease the loss by adjusting $W^1_{i,j}$.
We could consider a very small $\epsilon = 1\text{e-}6$ and compute:

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

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

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:

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

Derivative w.r.t. Input of Softmax

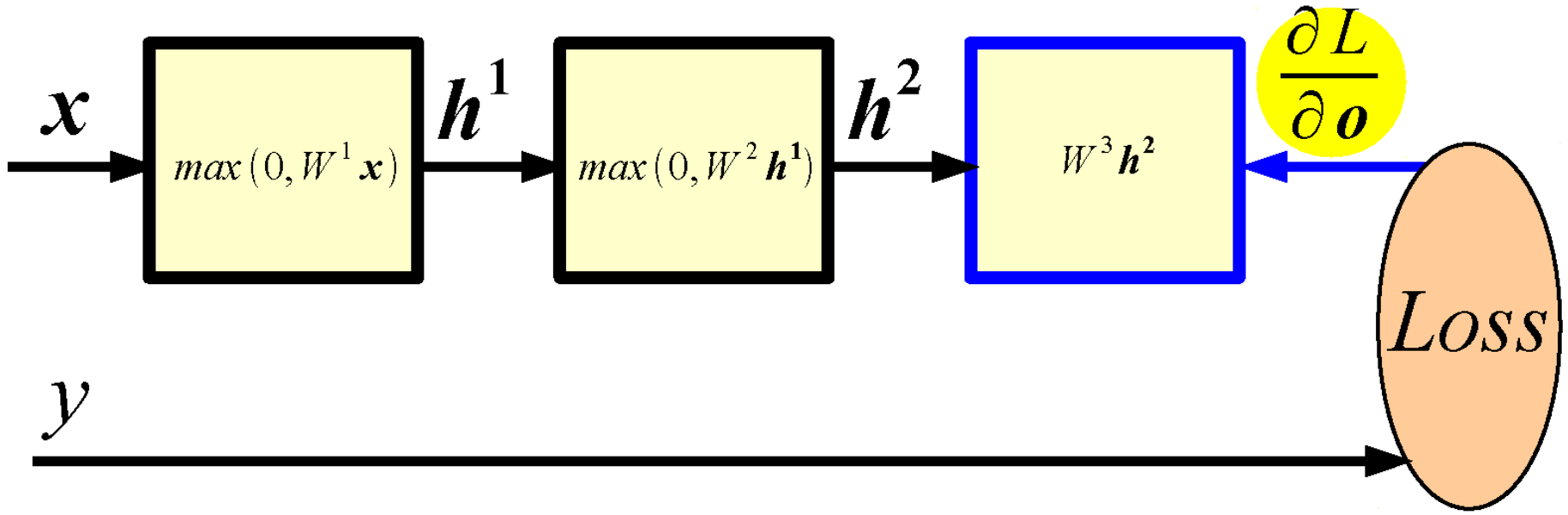
$$p(c_k=1|\boldsymbol{x})=\frac{e^{o_k}}{\sum_j e^{o_j}}$$

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

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

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

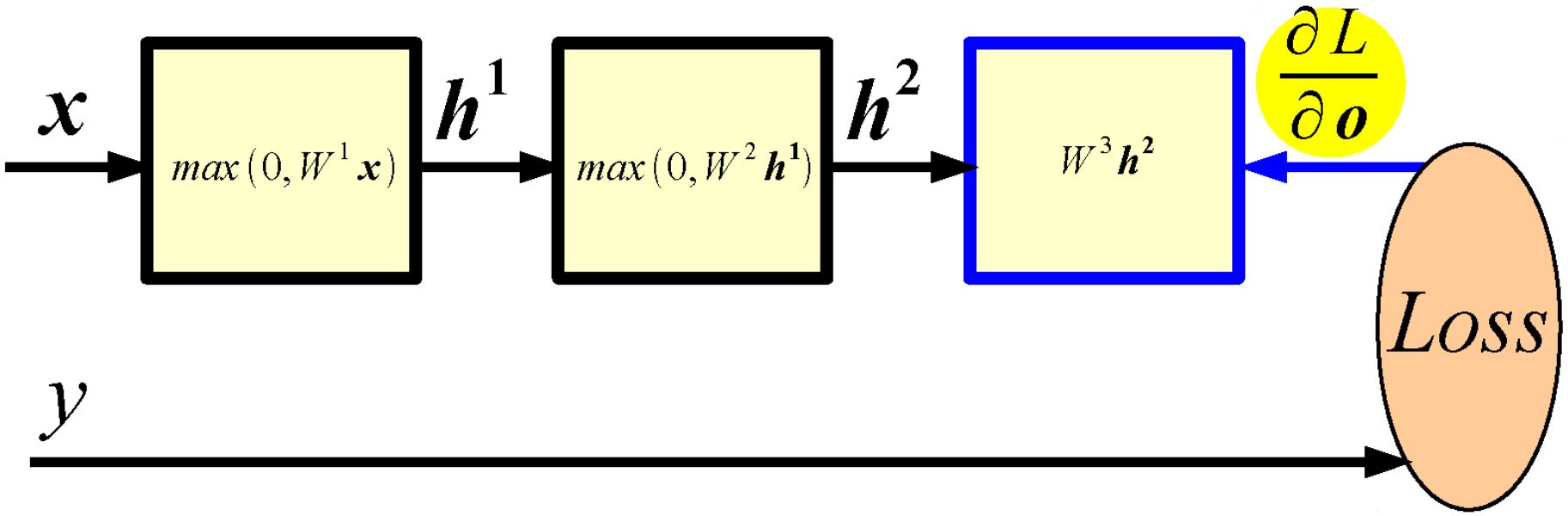
Backward Propagation



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

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

Backward Propagation

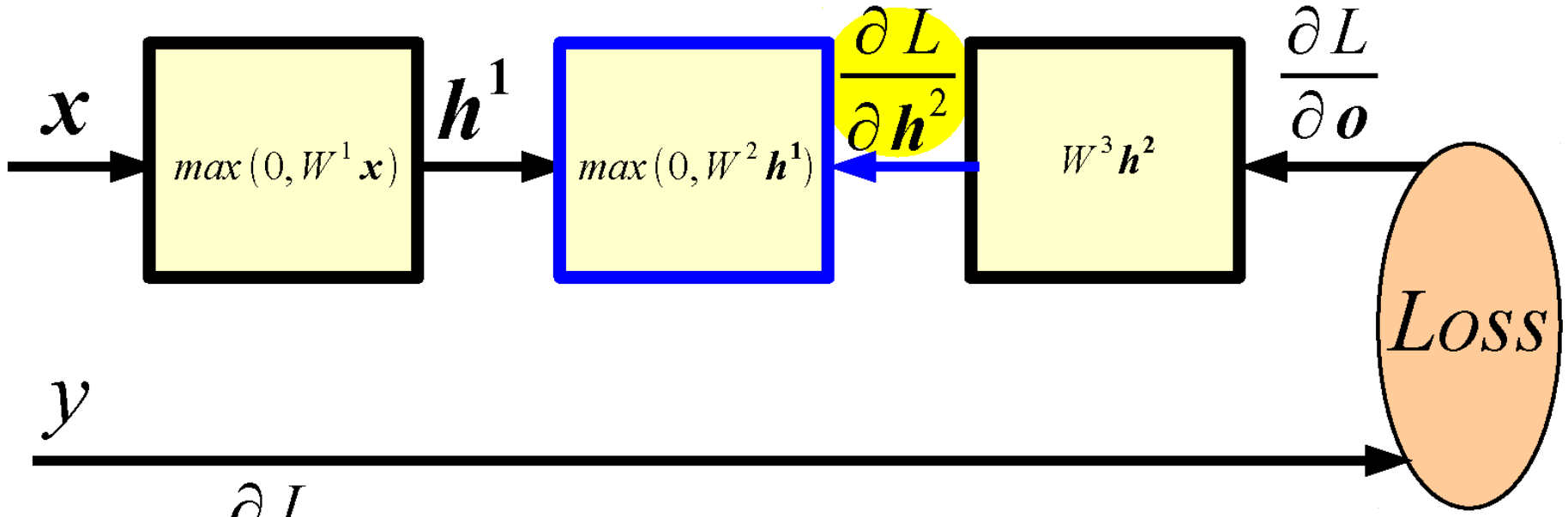


Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

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

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

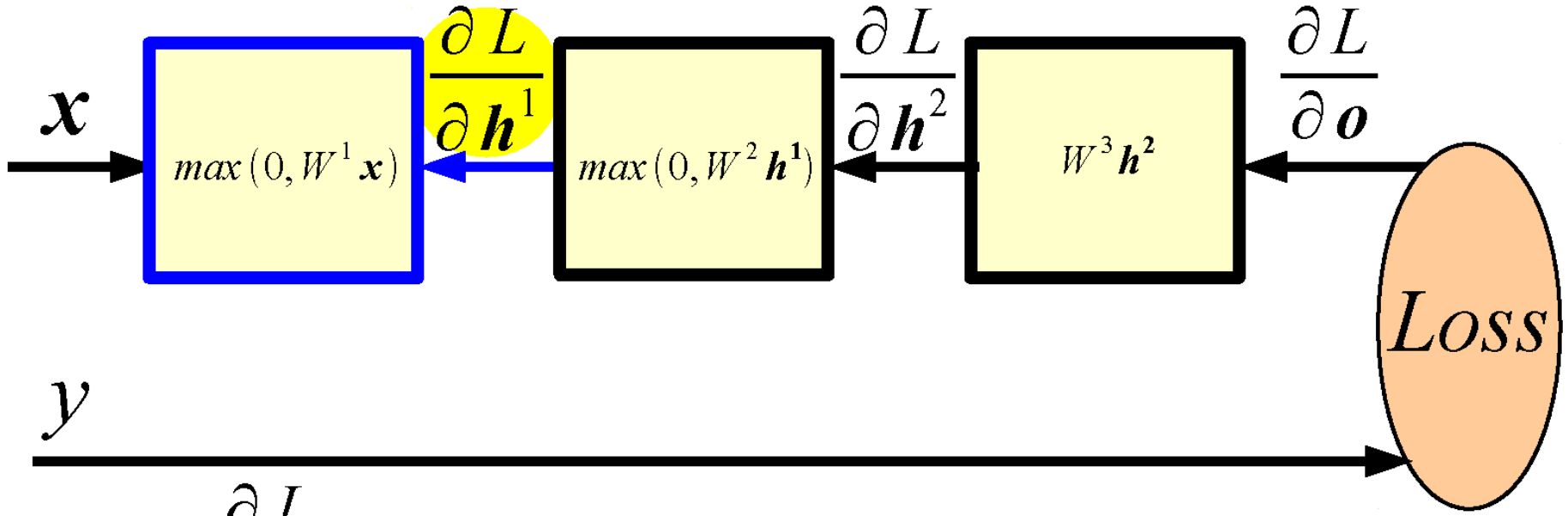
Backward Propagation



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

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

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:
  - Faster convergence
  - But can cause your gradient response to ‘explode’.

Backward Propagation

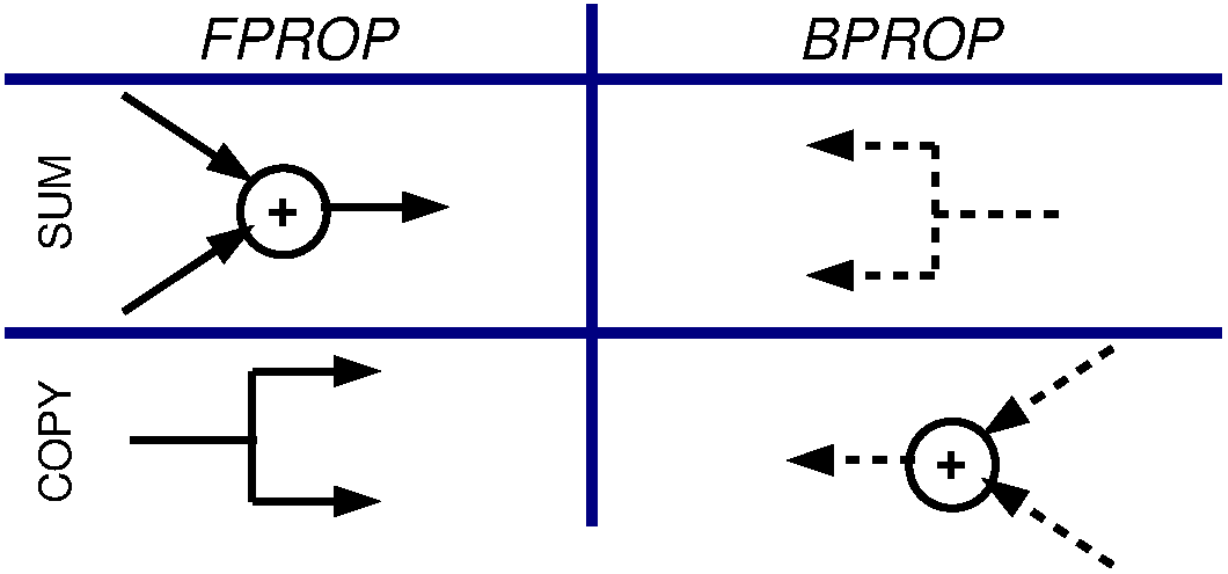
Question: Does BPROP work with ReLU layers only?

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.,:



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:

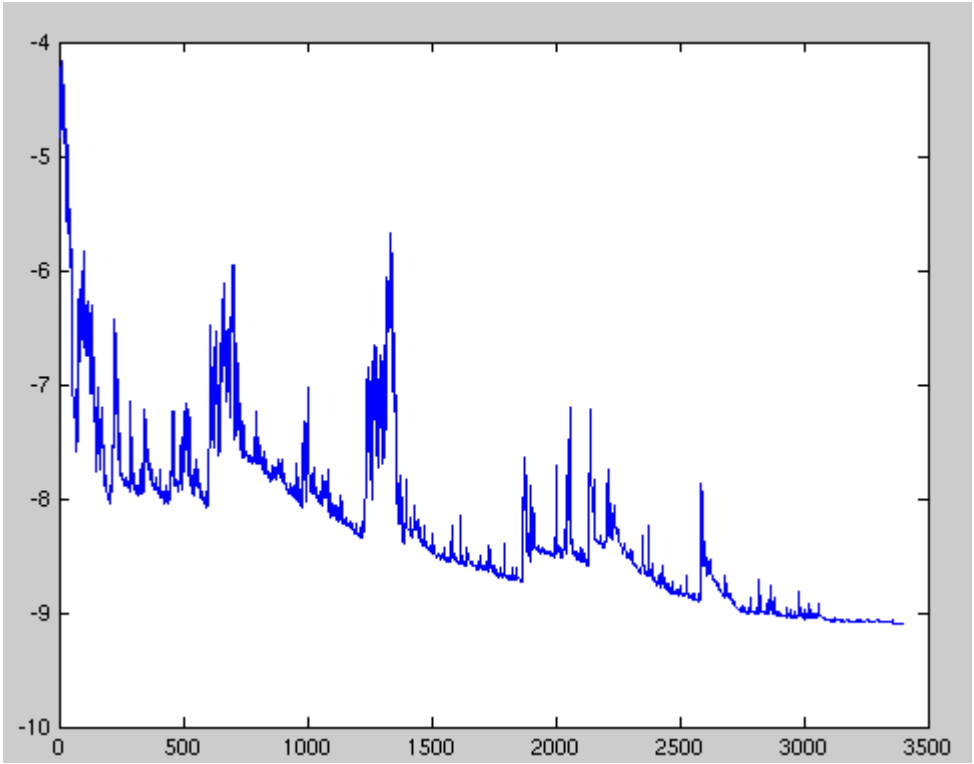
- Shuffle training set
- For each data point *i=1…n (maybe as mini-batch)*
  - *Gradient descent*

"Epoch"

Stochastic Gradient Descent

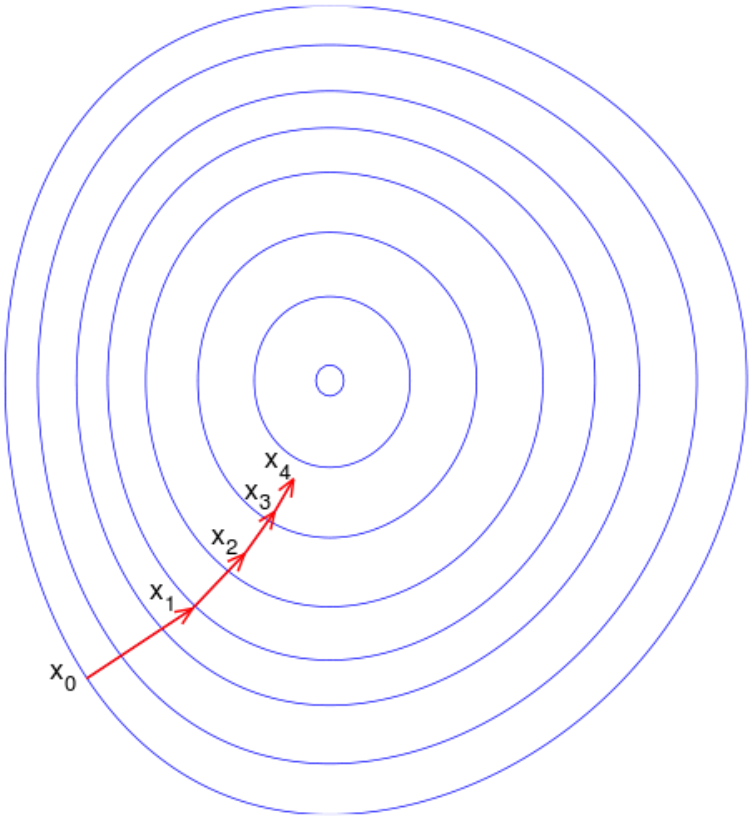
Loss will not always decrease (locally) as training data point is random.

Still converges over time.



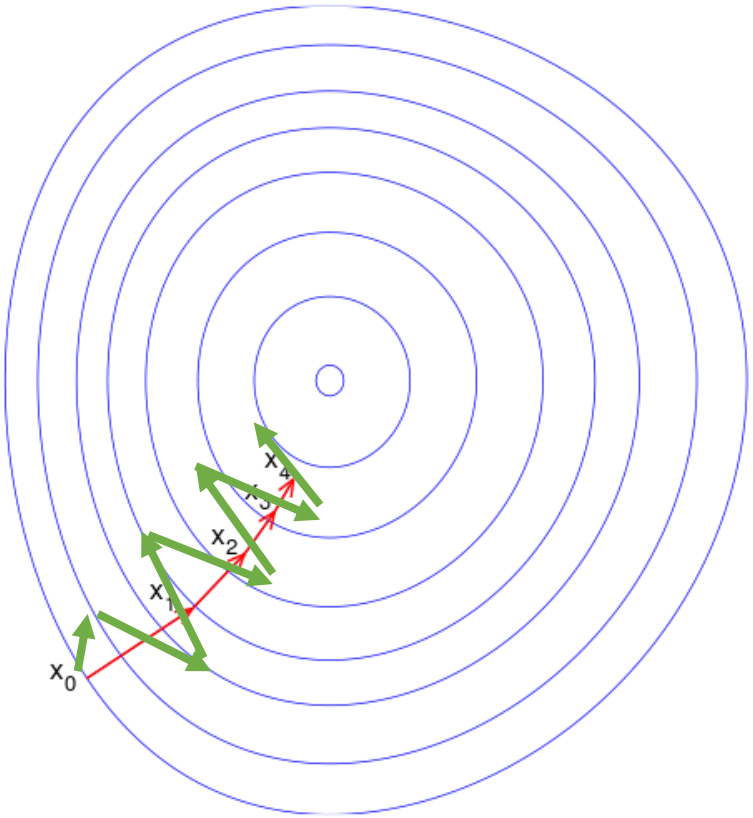
Wikipedia

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: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \frac{\partial L}{\partial \boldsymbol{\theta}}$

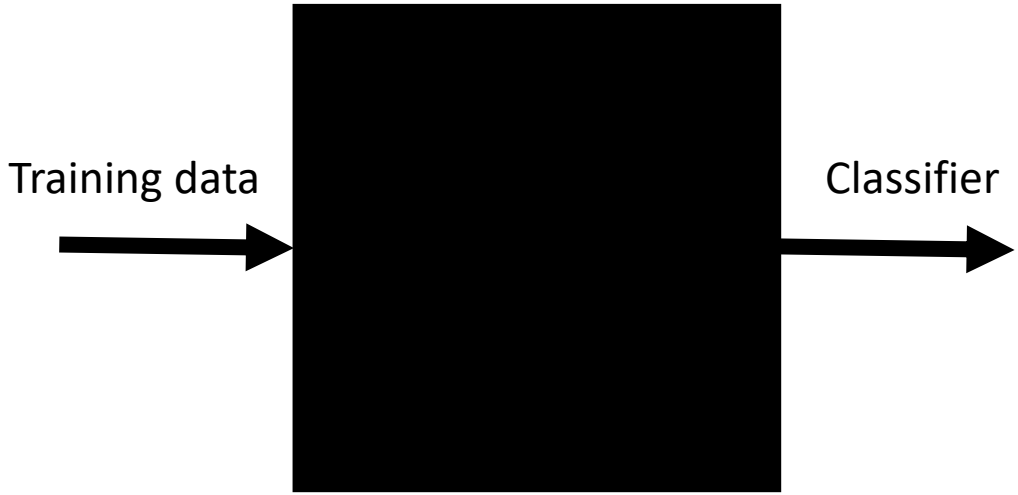
- With momentum (new α parameter):

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

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

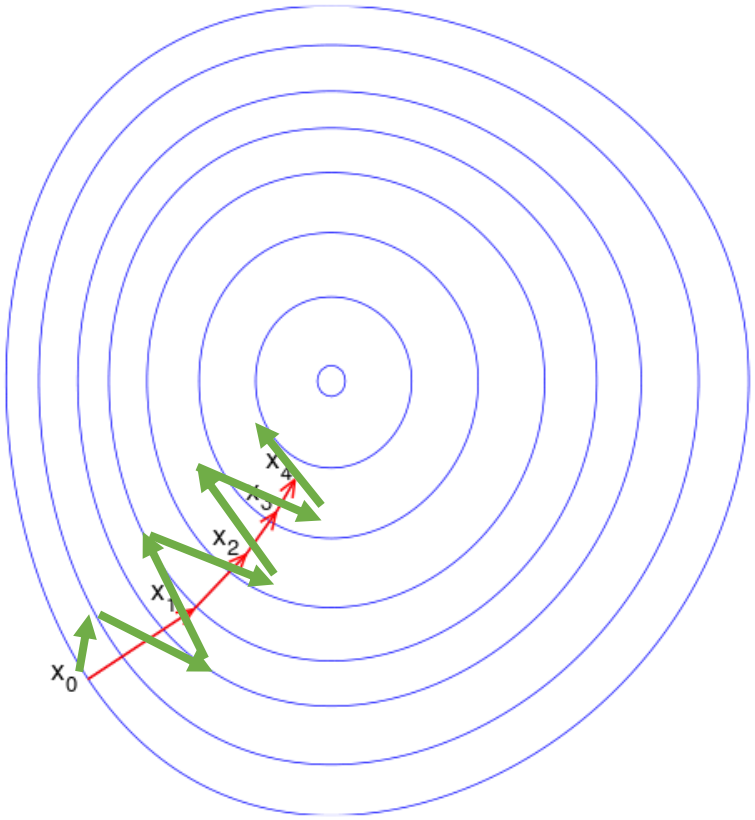
agokasla 6:56 PM
uploaded and commented on this image: **image.png**

> 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

Universality

- A single-layer network can learn any function:
  - So long as it is differentiable
  - To some approximation;
    More perceptrons = a better approximation

- Visual proof (Michael Nielson):

http://neuralnetworksanddeeplearning.com/chap4.html

If a single-layer network can learn any function…

- …given enough parameters…
- …then why do we go deeper?

Intuitively, composition is efficient because it allows reuse.

Empirically, deep networks do a better job than shallow networks at learning such hierarchies of knowledge.

On Wednesday

- Convolutional Neural Networks
- Regularization