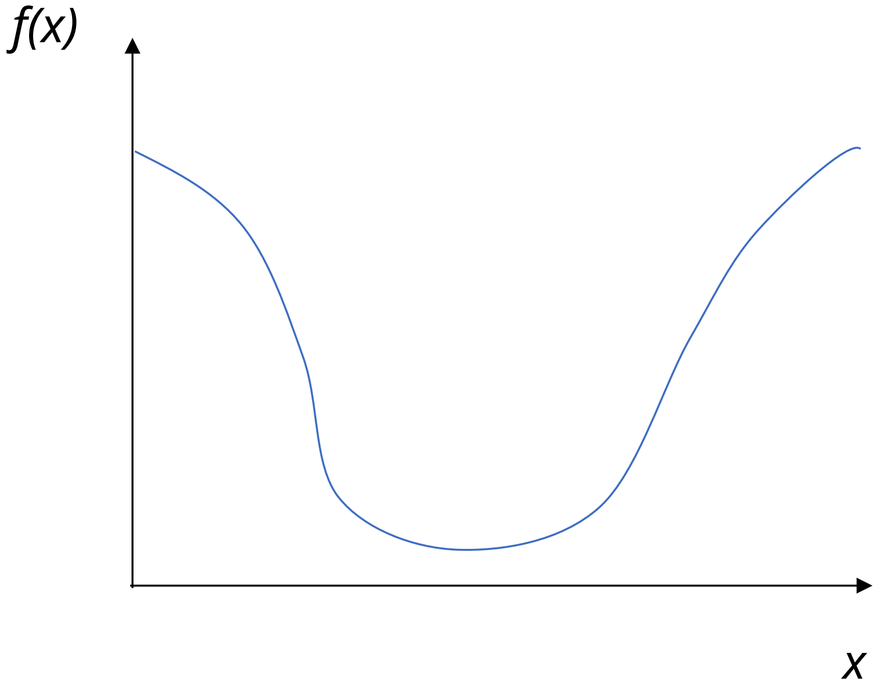
Training Neural Networks

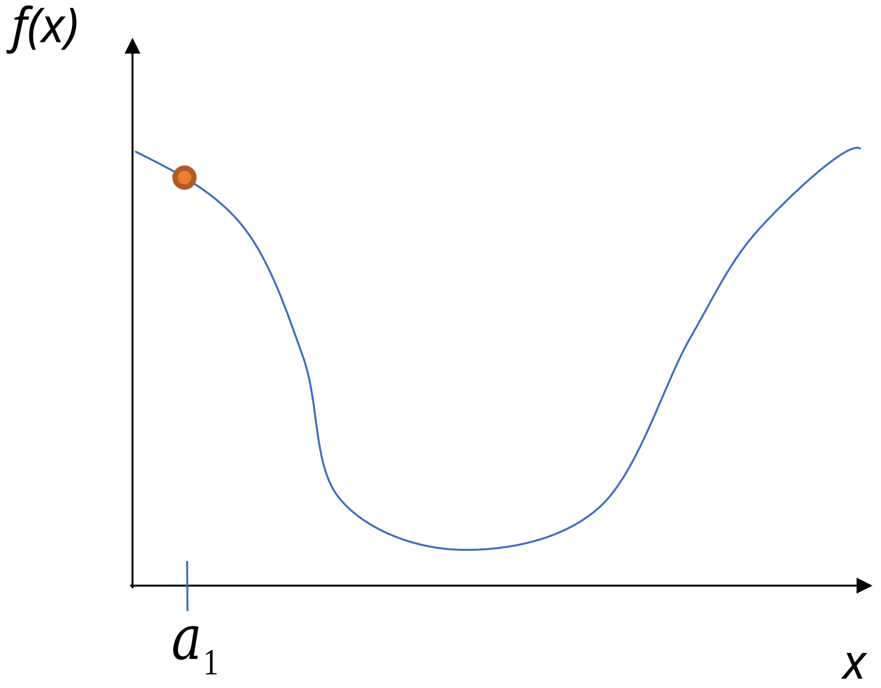
Learning the weight matrices ***W***

Gradient descent



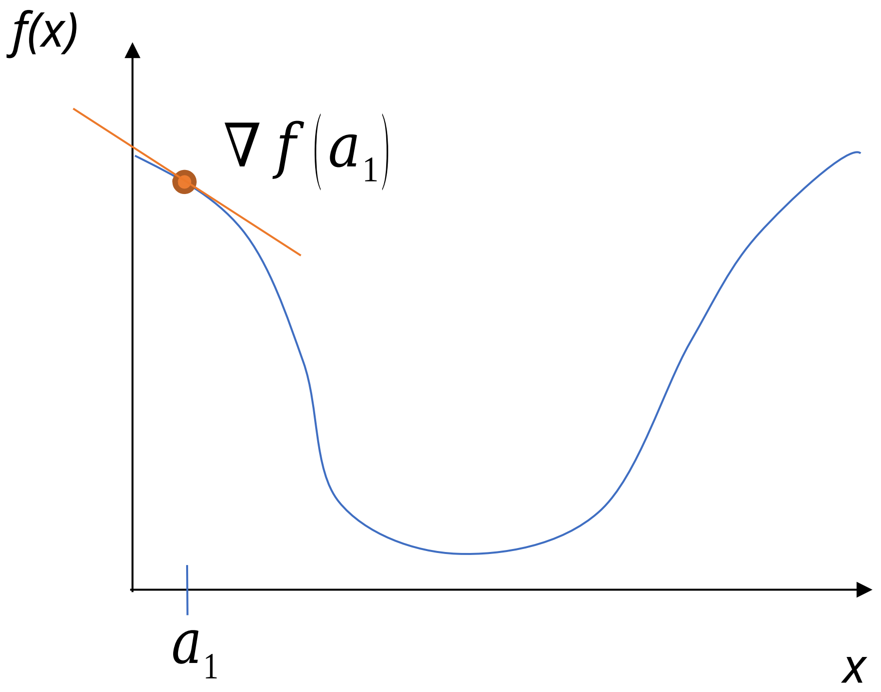
General approach

Pick random starting point.



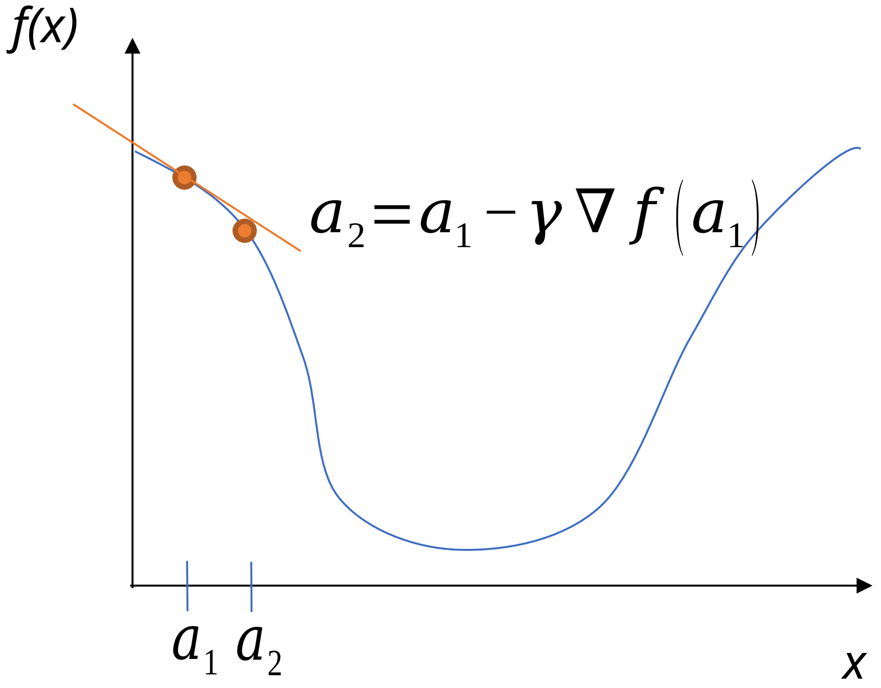
General approach

Compute gradient at point (analytically or by finite differences)



General approach

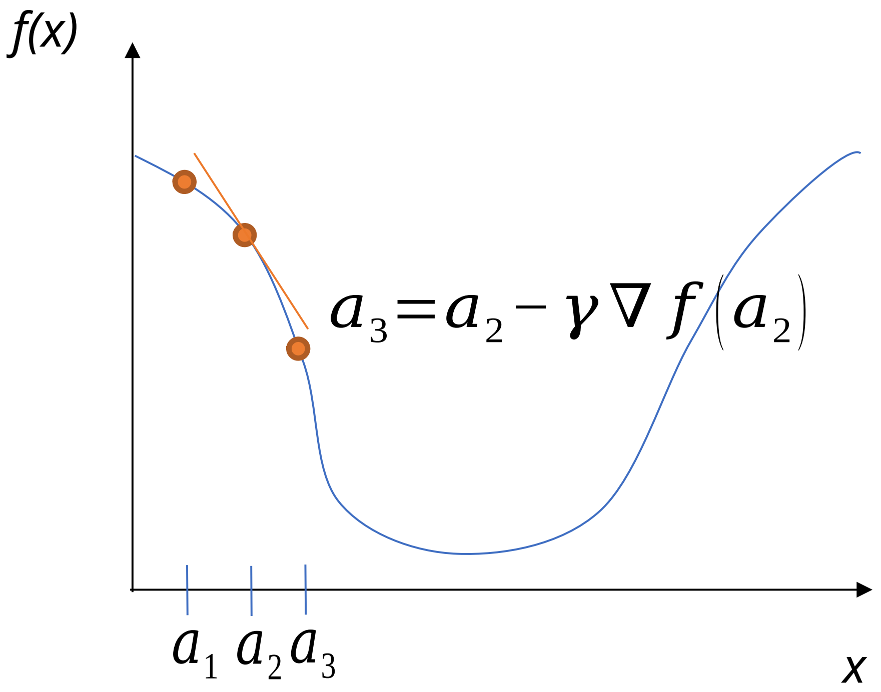
Move along parameter space in direction of negative gradient



γ= amount to move
= *learning rate*

General approach

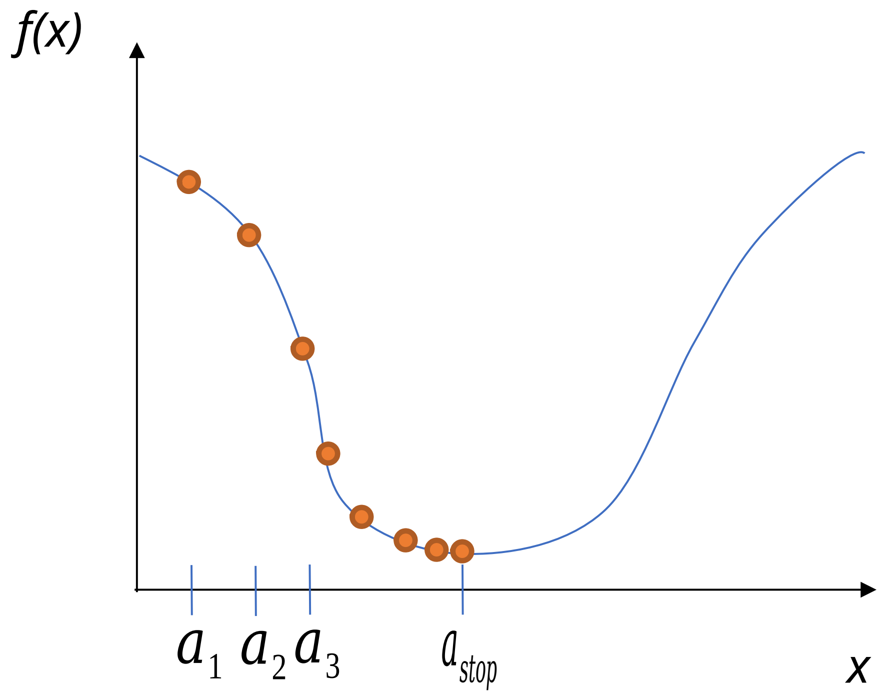
Move along parameter space in direction of negative gradient.



γ= amount to move
= *learning rate*

General approach

Stop when we don't move any more.

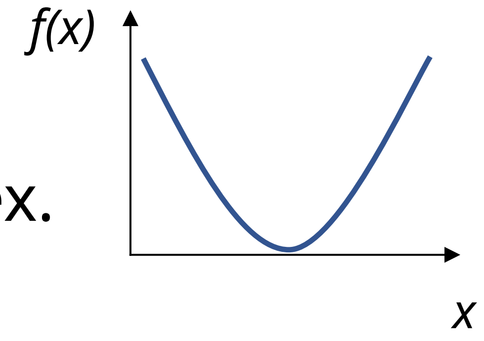


$$a_{stop}:$$
$$a_{n-1} - \gamma \nabla f(a_{n-1}) = 0$$

Gradient descent

1847!!

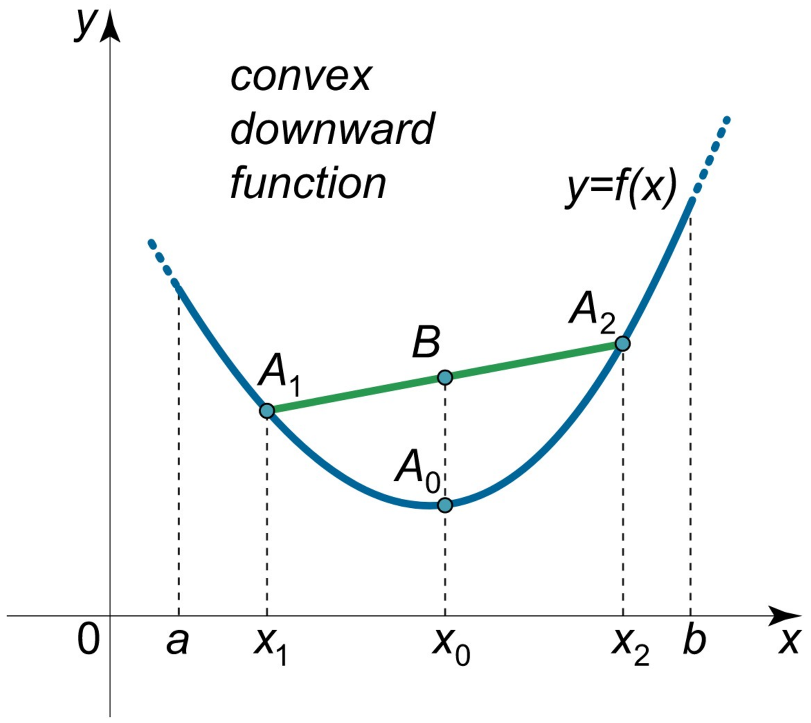
- Optimizer for functions.
- Guaranteed to find optimum for convex functions.
  - Non-convex = find *local* optimum.
  - Most vision problems aren't convex.

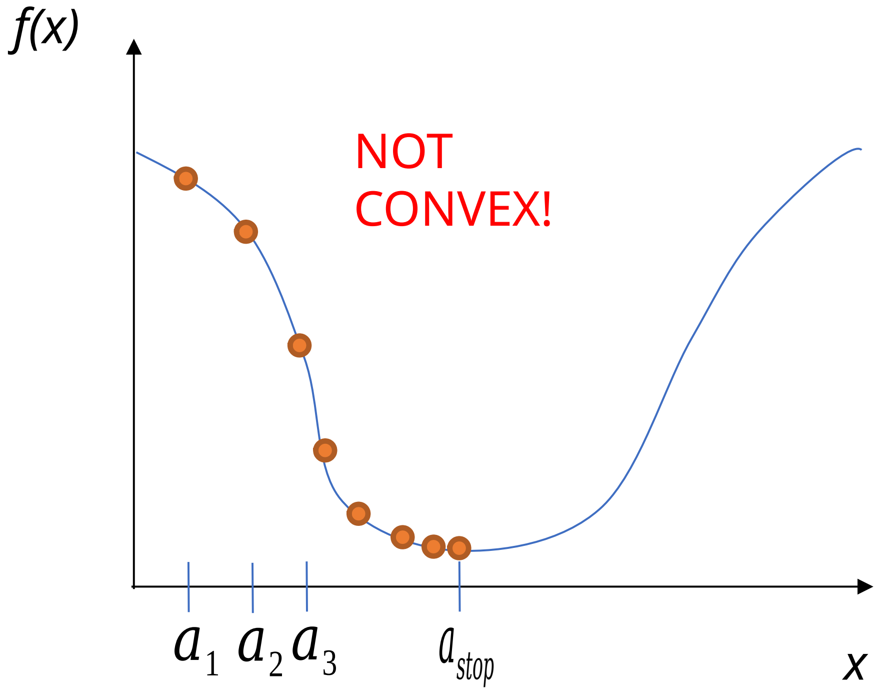


- Works for multi-variate functions.
  - Need to compute matrix of *partial derivatives ("Jacobian")*

Convex functions

f a function is convex downward , the midpoint B of each chord (A1,A2) lies above the corresponding point A0 of the graph of the function or coincides with this point.





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 $x = \left(A^T A\right)^{-1} A^T b$

You can, yes.

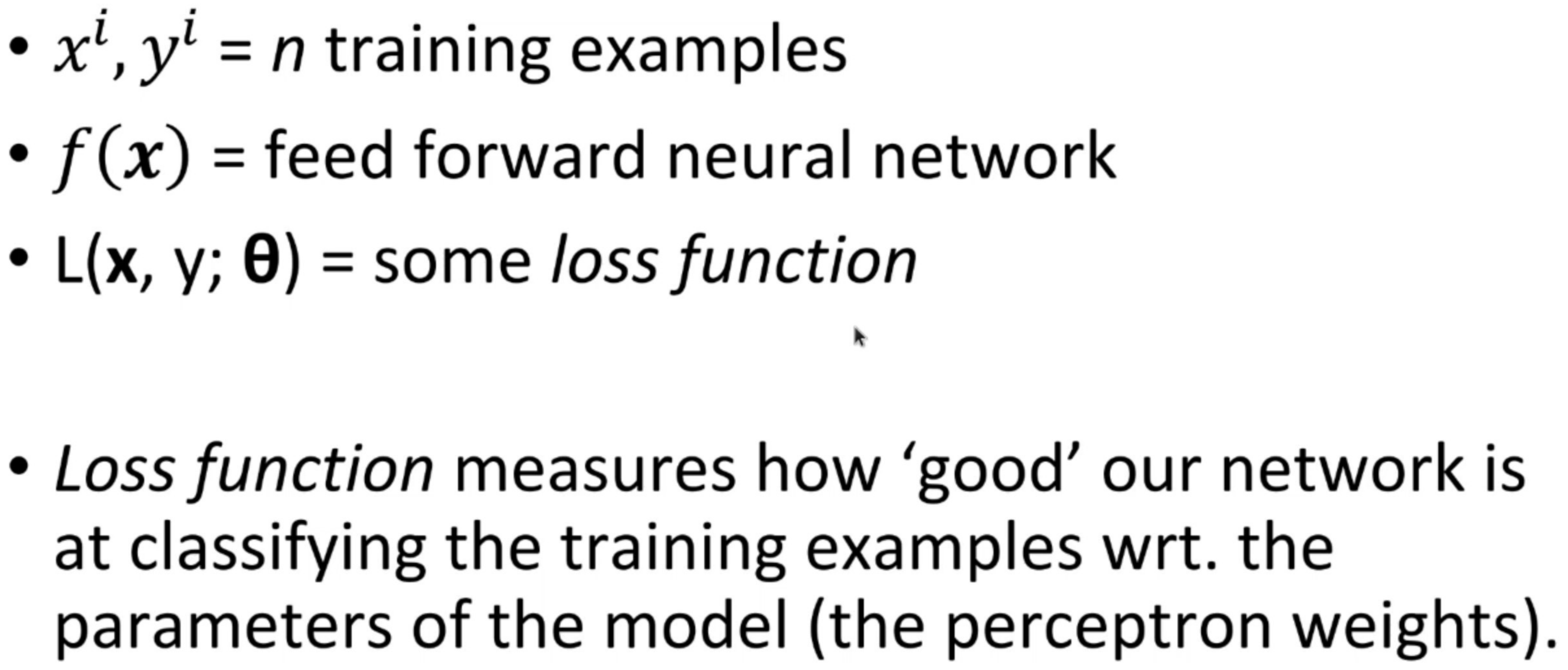
Why would I use this over Least Squares?

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

Matrices are ridiculously large. You will need to invert a (1 000 000 x 1 000 000) matrix (have fun).

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

Train NN with Gradient Descent

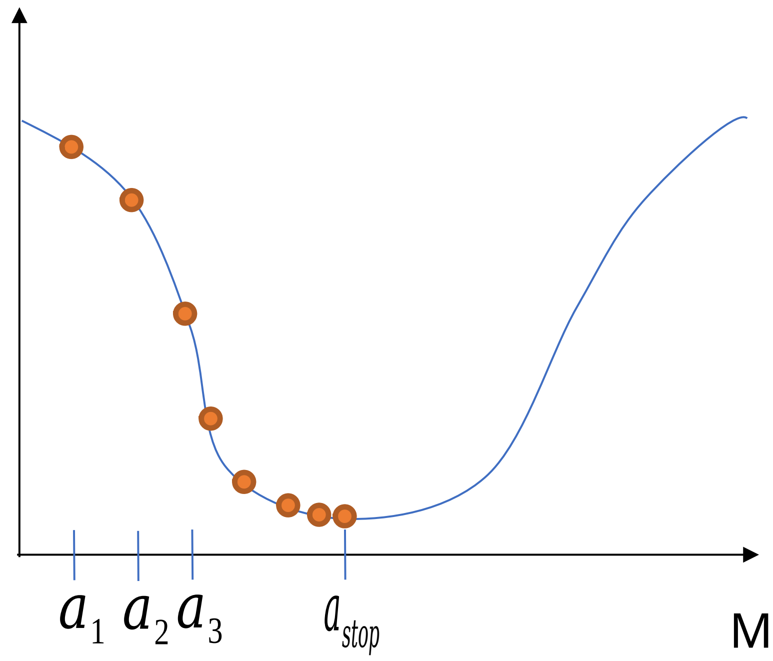


- x^i, y^i = *n* training examples
- $f(\boldsymbol{x})$ = feed forward neural network
- L(**x**, y; **θ**) = some *loss function*

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

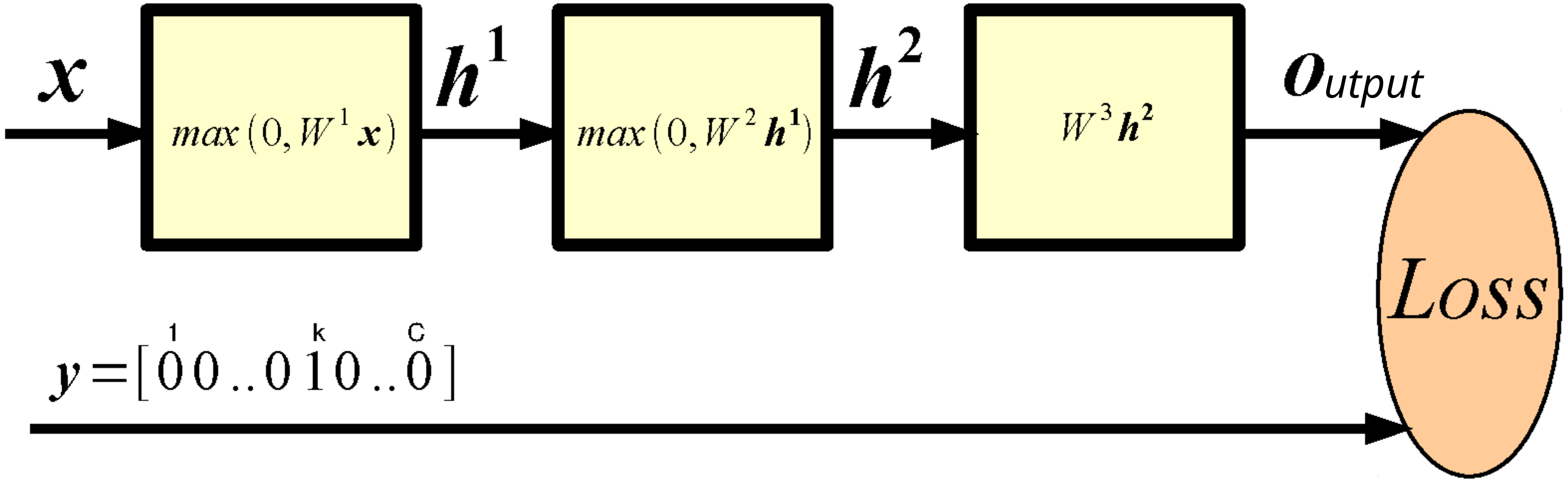
Train NN with Gradient Descent

Loss function (Evaluate NN on training data)



Model parameters (perceptron weights)

How Good is a Network?


$\boldsymbol{x}$
$\max(0, W^1 \boldsymbol{x})$
$\boldsymbol{h}^1$
$\max(0, W^2 \boldsymbol{h}^1)$
$\boldsymbol{h}^2$
$W^3 \boldsymbol{h}^2$
$\boldsymbol{o}$utput
$Loss$
$y = [\overset{1}{0}\, 0 \,..\, 0\, \overset{k}{1}\, 0 \,..\, \overset{C}{0}]$

What is an appropriate loss?

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

$$y = \textit{true label} \qquad \hat{y} = \textit{predicted label} \qquad L(\hat{y}, y) = I(\hat{y} \neq y),$$

- 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

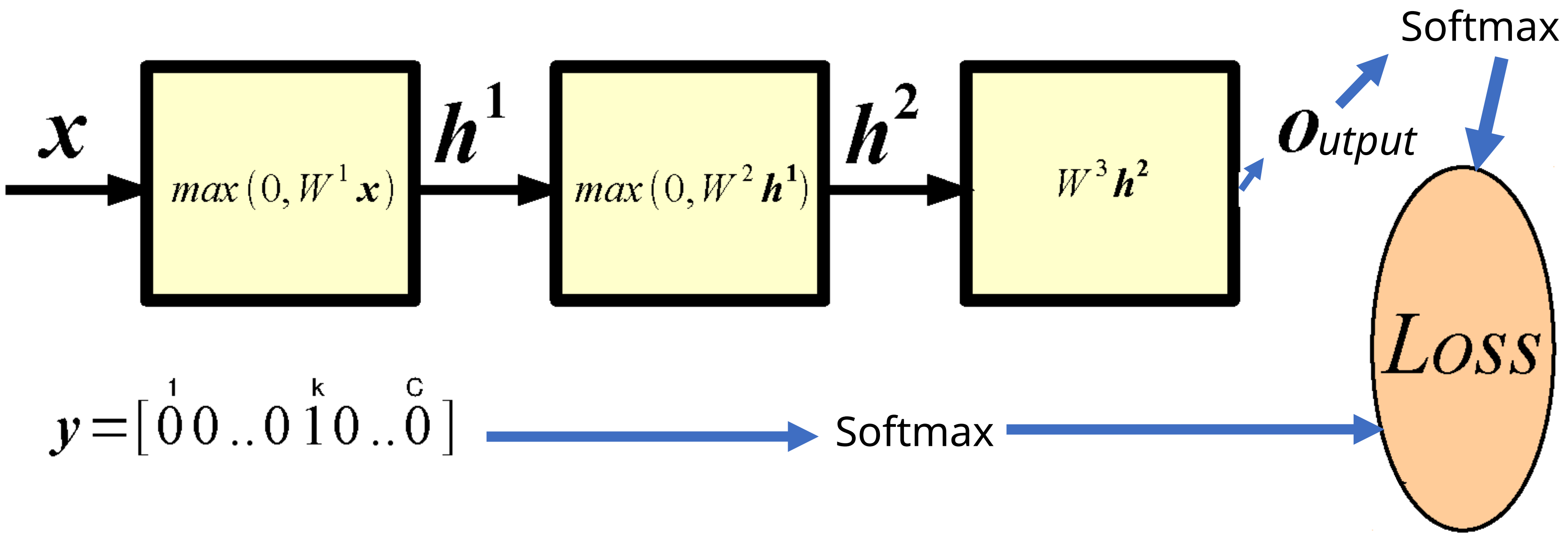
Special function on last layer - ‘S*oftmax*’:

“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):

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

Example: your model is classifying 20 classes, the final ouput vector will have a size 20x1. Softmax transforms the outputs to positive numbers with their sum = 1. Each row now corresponds to the probability of its associated class.

Cross-entropy loss function

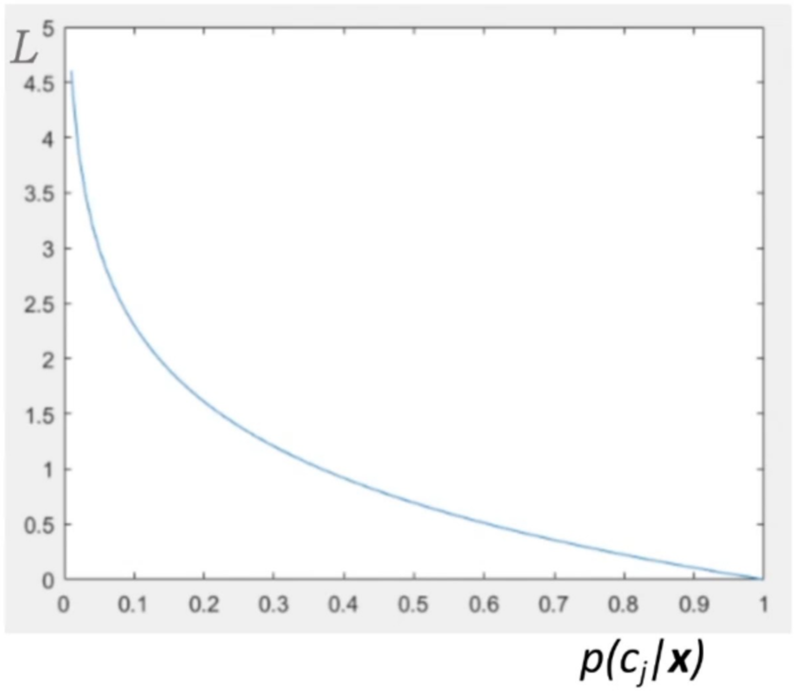
- Negative log-likelihood

y - binary indicator (0 or 1) if class label j is the correct classification for observation

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

- Minimizing this is similar to minimizing KL-divergence on predicted and target probability distributions

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



https://machinelearningmastery.com/cross-entropy-for-machine-learning/

Training

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

training set

model weights

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

sum over all training data

label associated with the training data point n

input vector associated with the training data point n

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

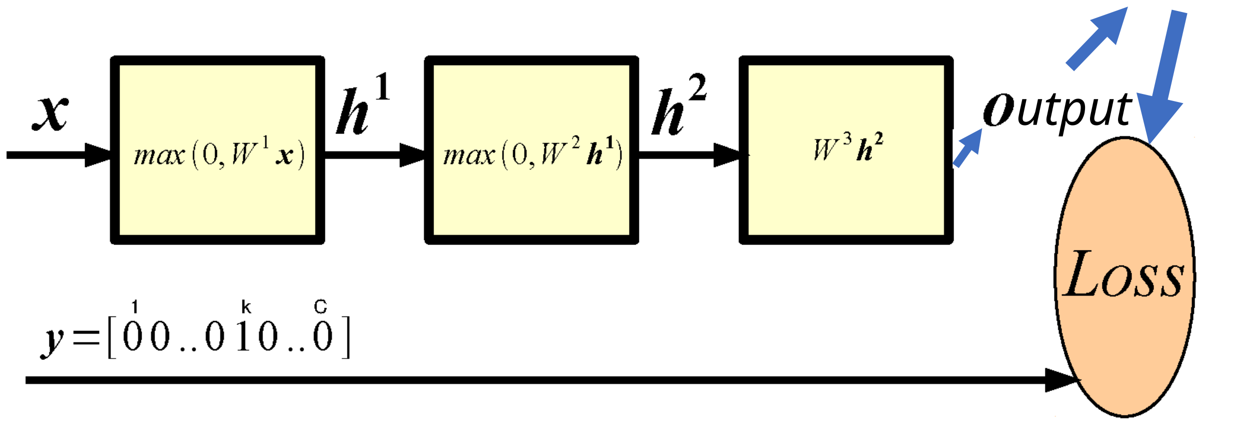
Derivative w.r.t. Input of Softmax

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

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

By substituting the first formula in the second, and taking the derivative wr.t. ***o*** we get:

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



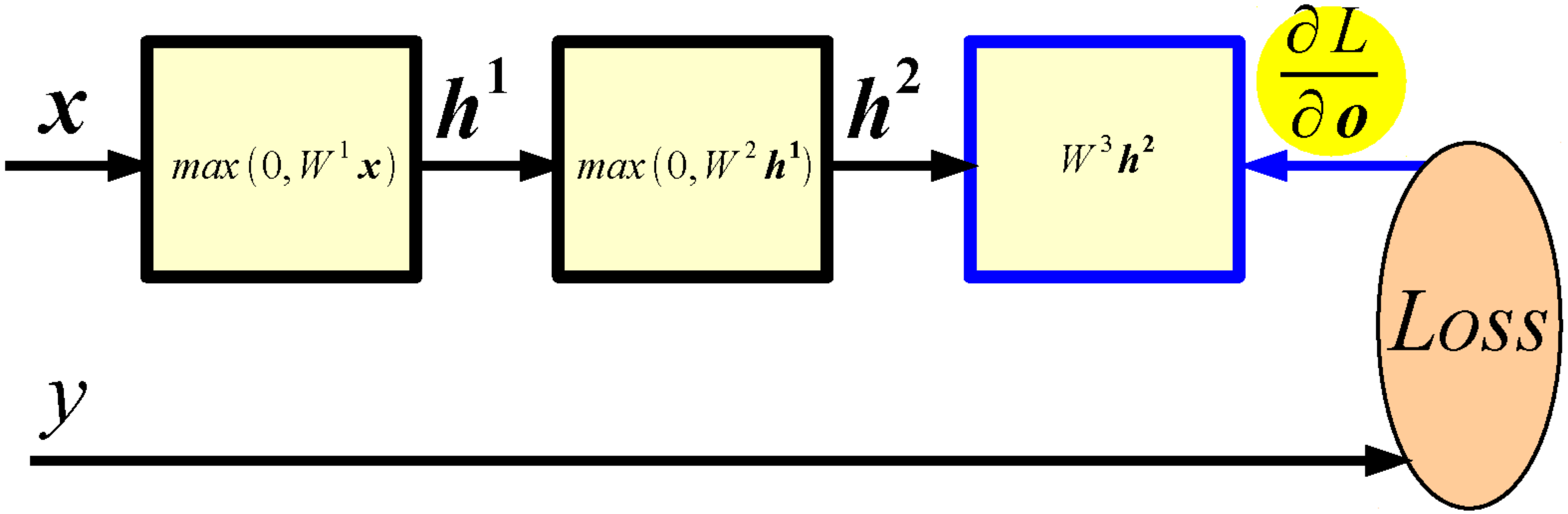
Softmax
$\boldsymbol{x}$
$max(0, W^1\boldsymbol{x})$
$\boldsymbol{h}^1$
$max(0, W^2\boldsymbol{h}^1)$
$\boldsymbol{h}^2$
$W^3\boldsymbol{h}^2$
$\boldsymbol{o}utput$
$Loss$
$\boldsymbol{y}=[\overset{1}{0}\,0\,..\,0\,\overset{k}{1}\,0\,..\,\overset{C}{0}]$

Ranzato

Proof:

$$
\begin{aligned}
\frac{\partial \xi}{\partial z_i} &= -\sum_{j=1}^{C} \frac{\partial t_j \log(y_j)}{\partial z_i} \\
&= -\sum_{j=1}^{C} t_j \frac{\partial \log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{1}{y_j} \frac{\partial y_j}{\partial z_i} \\
&= -\frac{t_i}{y_i} \frac{\partial y_i}{\partial z_i} - \sum_{j \neq i}^{C} \frac{t_j}{y_j} \frac{\partial y_j}{\partial z_i} = -\frac{t_i}{y_i} y_i(1-y_i) - \sum_{j \neq i}^{C} \frac{t_j}{y_j}(-y_j y_i) \\
&= -t_i + t_i y_i + \sum_{j \neq i}^{C} t_j y_i = -t_i + \sum_{j=1}^{C} t_j y_i = -t_i + y_i \sum_{j=1}^{C} t_j \\
&= y_i - t_i
\end{aligned}
$$

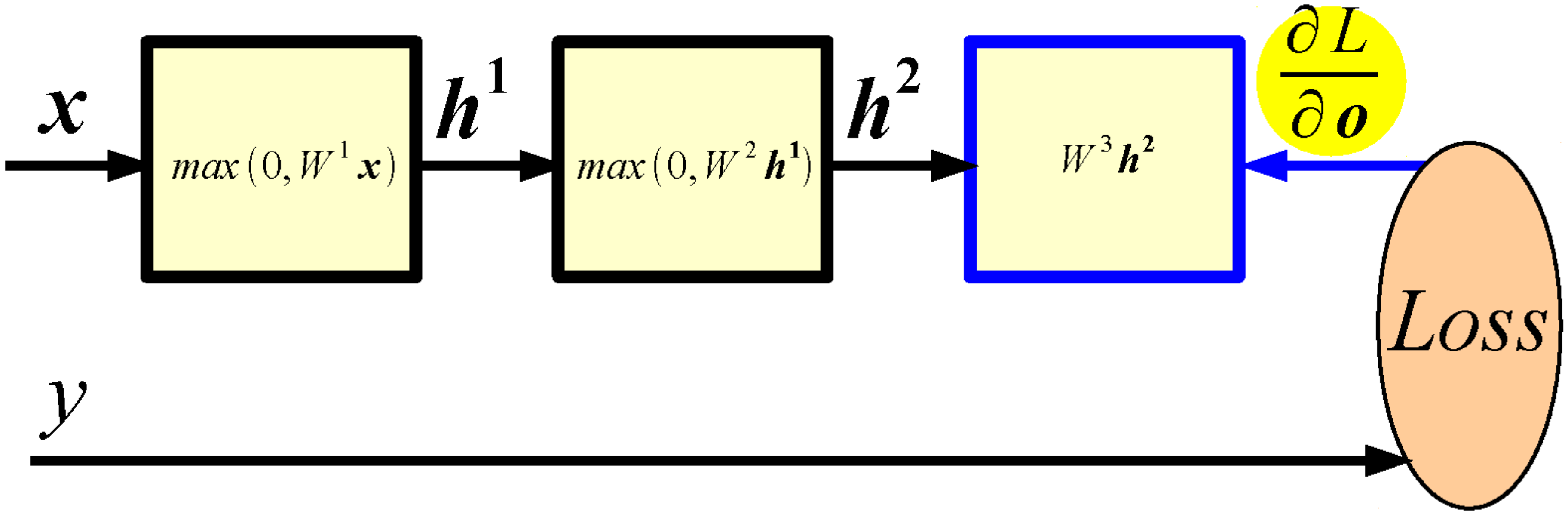
Backward Propagation


$\boldsymbol{x}$
$max(0, W^1 \boldsymbol{x})$
$\boldsymbol{h}^1$
$max(0, W^2 \boldsymbol{h}^1)$
$\boldsymbol{h}^2$
$W^3 \boldsymbol{h}^2$
$\frac{\partial L}{\partial \boldsymbol{o}}$
$Loss$
y

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

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

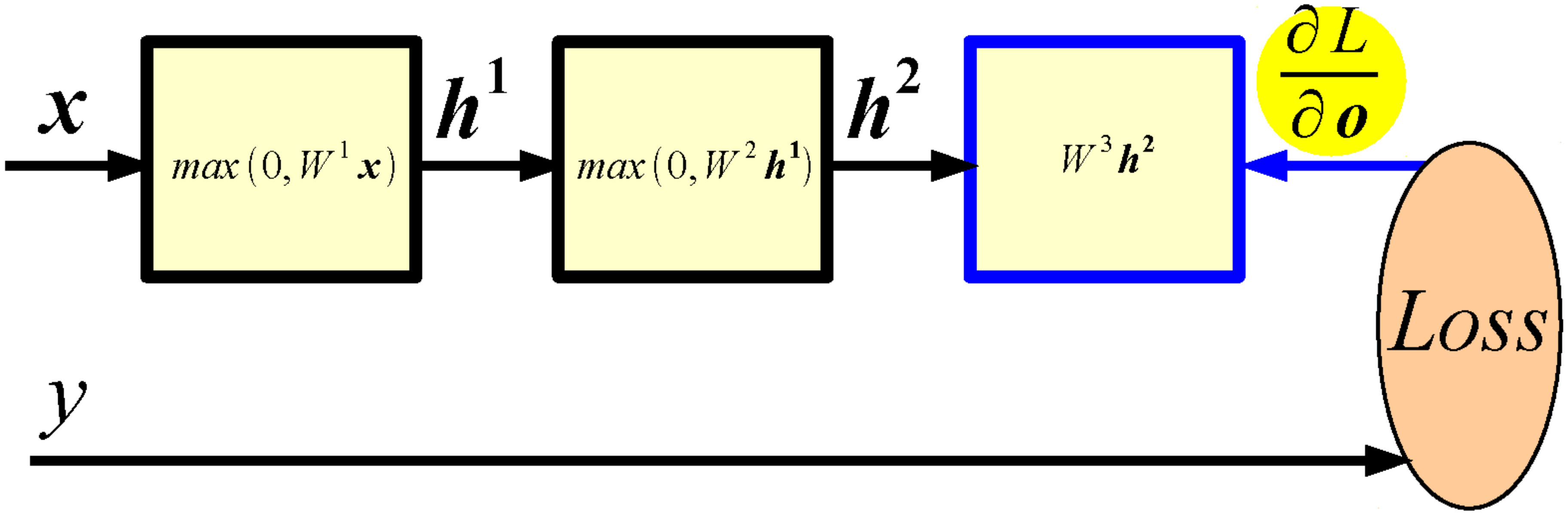
Backward Propagation



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



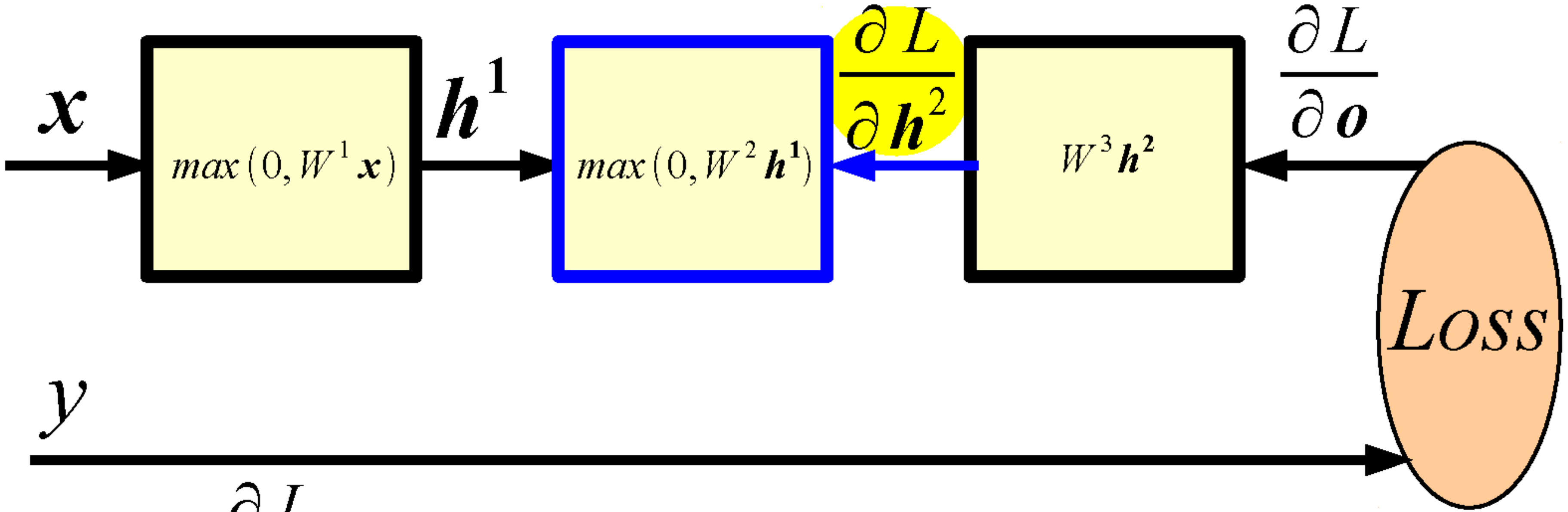
$\boldsymbol{x}$
$max(0, W^1 \boldsymbol{x})$
$\boldsymbol{h}^1$
$max(0, W^2 \boldsymbol{h}^1)$
$\boldsymbol{h}^2$
$W^3 \boldsymbol{h}^2$
$\frac{\partial L}{\partial \boldsymbol{o}}$
$Loss$
y

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

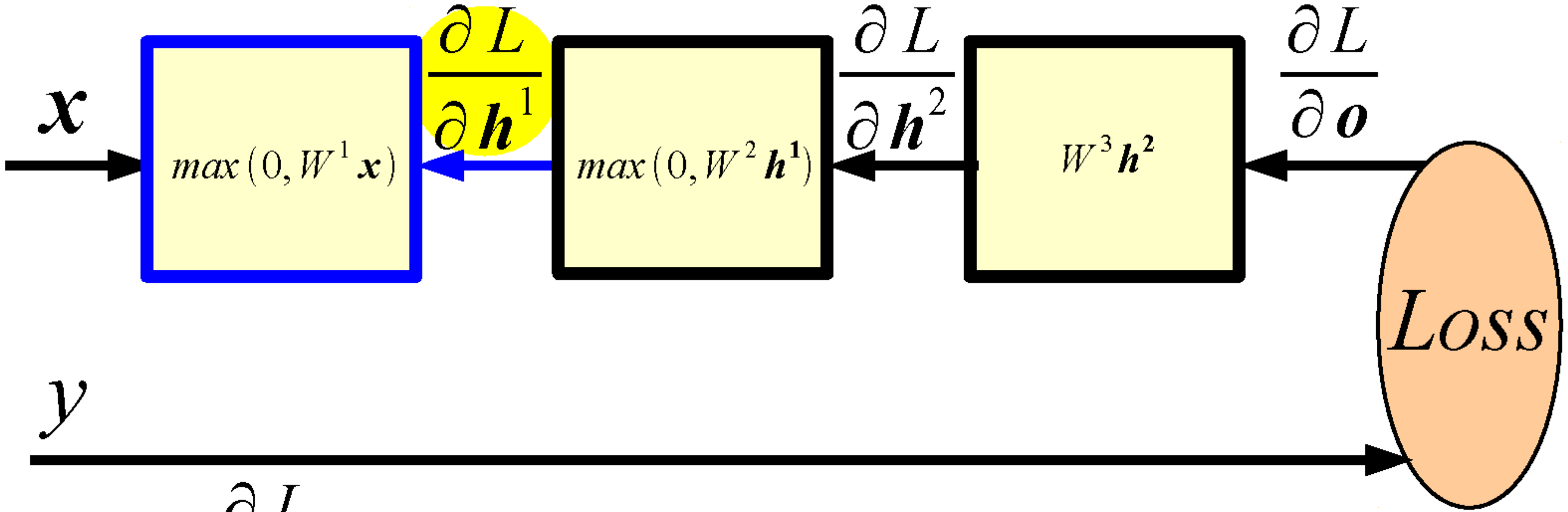


x
$\max(0, W^1 \boldsymbol{x})$
$\boldsymbol{h}^1$
$\max(0, W^2 \boldsymbol{h}^1)$
$\frac{\partial L}{\partial \boldsymbol{h}^2}$
$W^3 \boldsymbol{h}^2$
$\frac{\partial L}{\partial \boldsymbol{o}}$
$Loss$
y

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


$\boldsymbol{x}$
$max(0, W^1 \boldsymbol{x})$
$\frac{\partial L}{\partial \boldsymbol{h}^1}$
$max(0, W^2 \boldsymbol{h}^1)$
$\frac{\partial L}{\partial \boldsymbol{h}^2}$
$W^3 \boldsymbol{h}^2$
$\frac{\partial L}{\partial \boldsymbol{o}}$
$Loss$
y

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

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
```

Ranzato

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*”

Psuedo code:

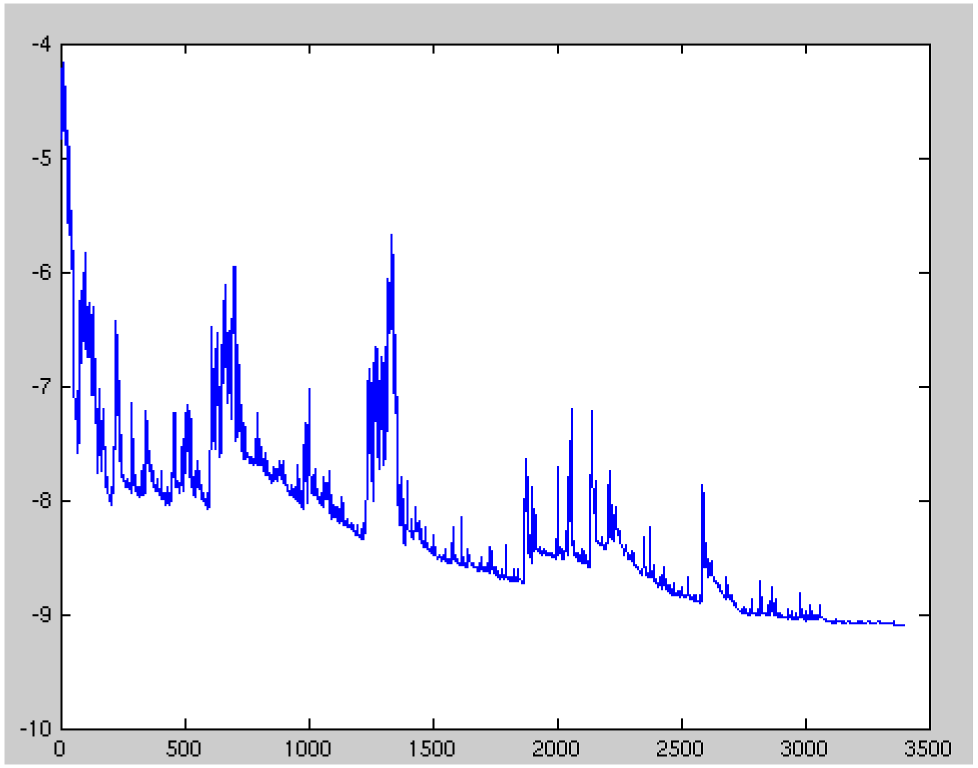
```
Pick starting W  and learning rate $\gamma$
While not at minimum:
    • Shuffle training set
    • For each data point i=1…n  (maybe as mini-batch)
        • Gradient descent
```

“Epoch“

Stochastic Gradient Descent

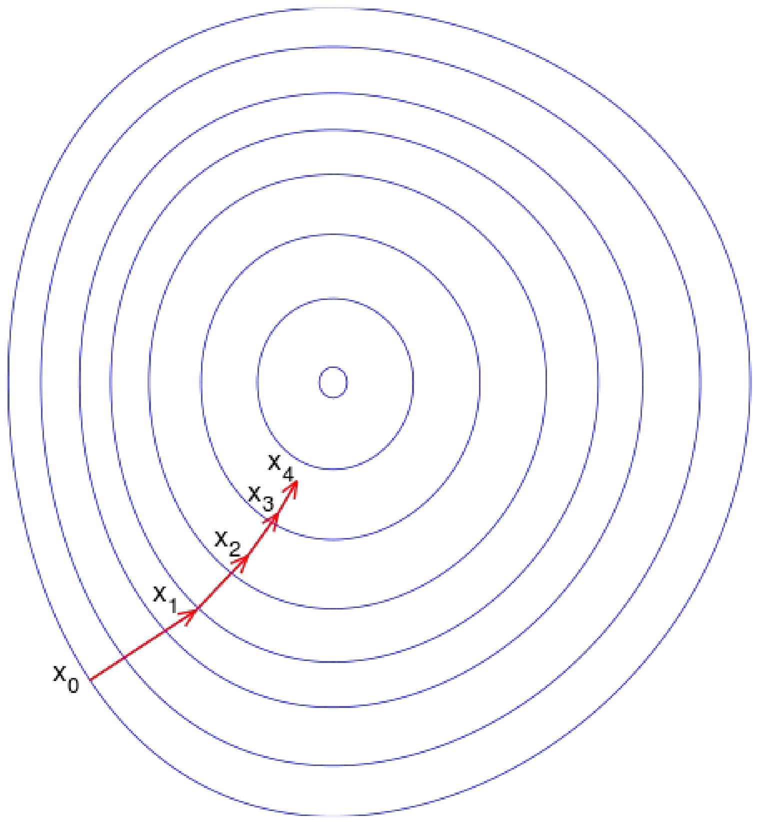
Loss will not always decrease (locally) as training data point is random.

Still converges over time.



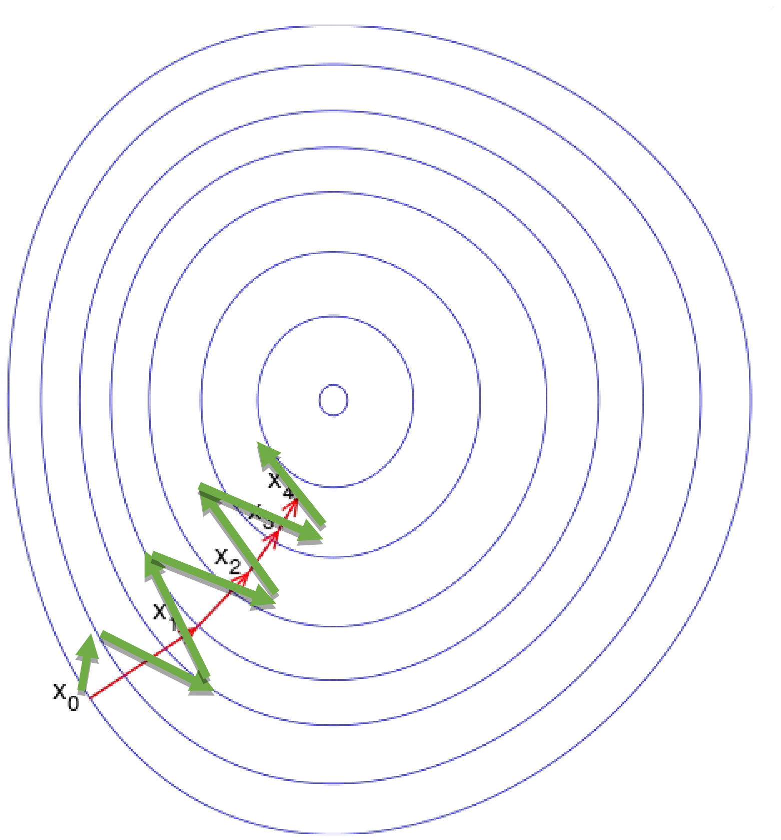
Wikipedia

Gradient descent oscillations



Wikipedia

Gradient descent oscillations



Slow to converge to the (local) optimum

Wikipedia

Momentum

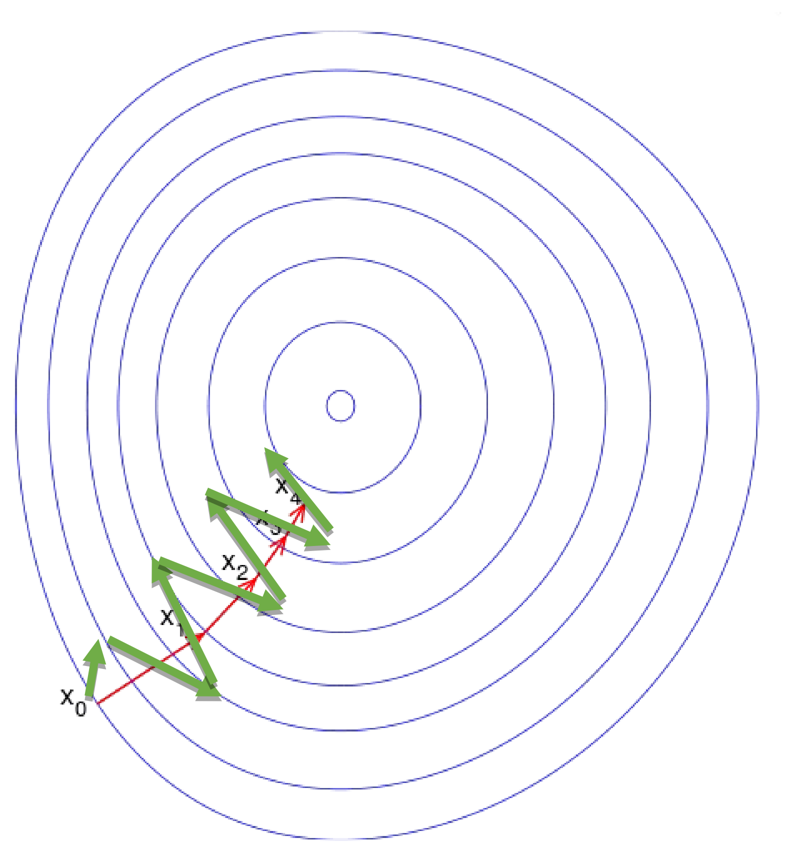
- Adjust the gradient by a weighted sum of the previous amount plus the current amount.

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

- With momentum (new α parameter):

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

Lowering the learning rate = smaller steps in SGD

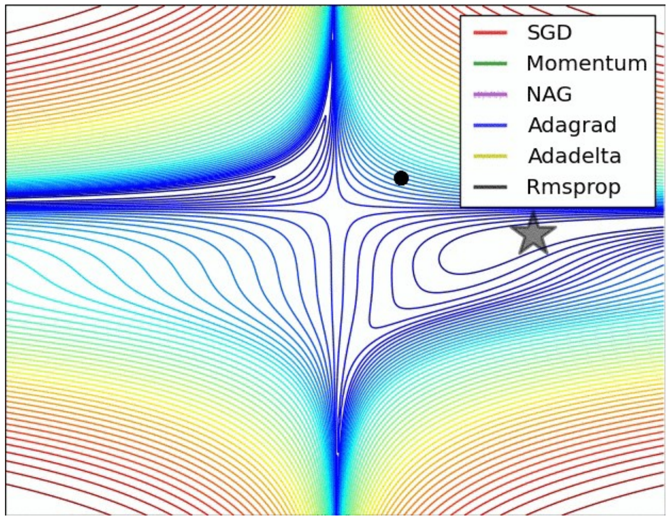


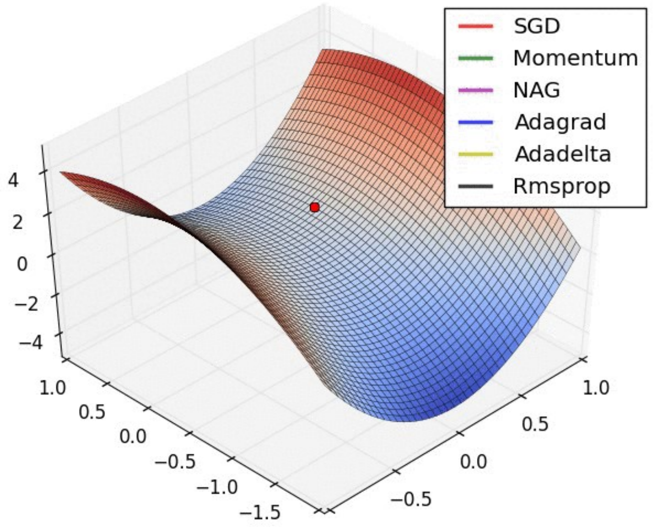
-Less ‘ping pong’

-Takes longer to get to the optimum

Wikipedia

Low learning rate may get stuck in flat regions in energy landscape





Large learning rate can inadvertently increase rather than decrease the training error (can jump over the sought minima).

have fun tuning :)

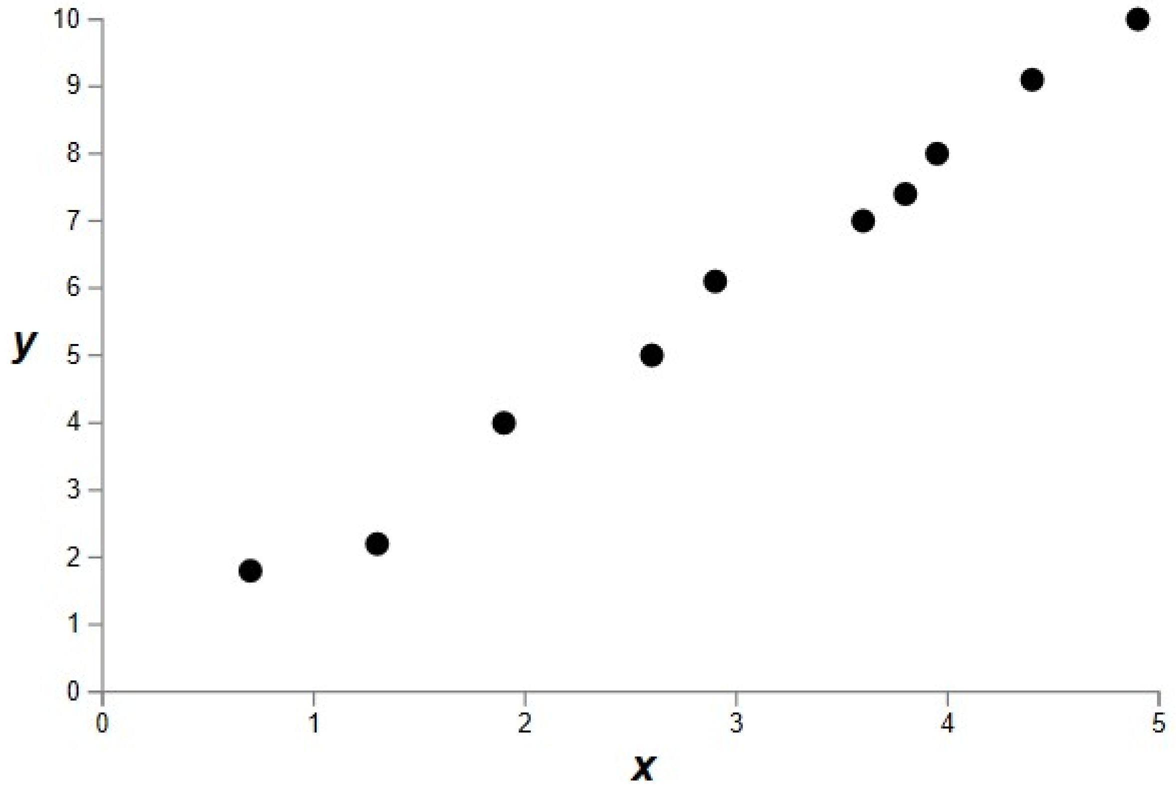
Problem of fitting

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

- More data = less chance to overfit

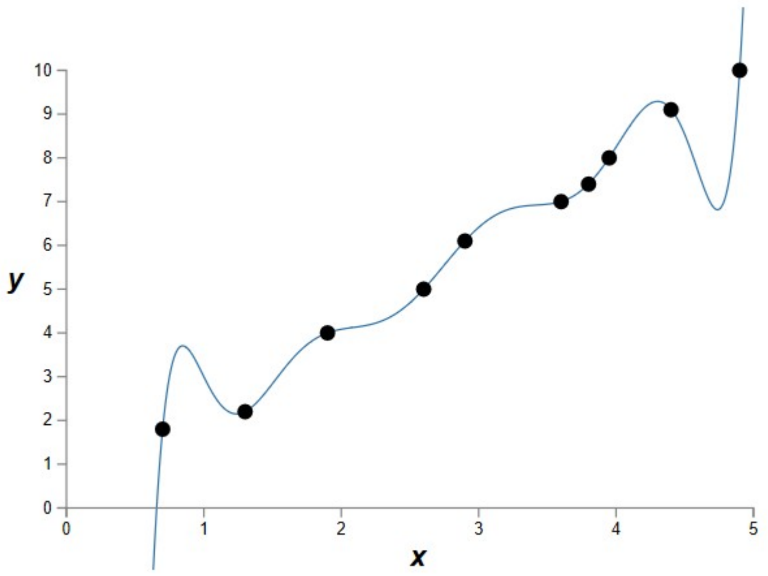
- How do we know what is required?

Data fitting problem

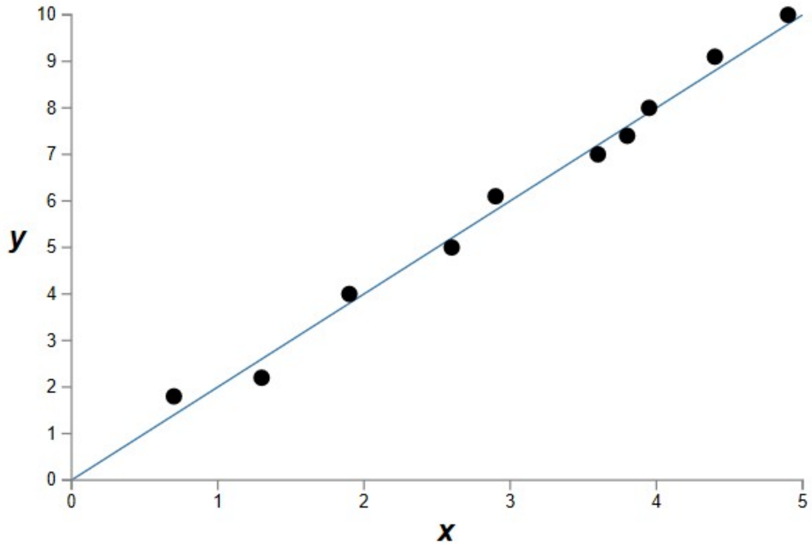


[Nielson

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



9th order polynomial



1st order polynomial

[Nielson]

Both can describe the data…

- …but one is simpler.

- Occam's razor (c. 1287-1347):

  "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 => generalizes better.

Regularization

- Idea:
  Penalize the use of parameters to prefer small weights.

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

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

$$C = C_0 + \lambda \sum_{W} W^2$$

Regularization

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

$$C = C_0 + \lambda \sum_{W} W^2$$

$$\mathrm{C} = \underbrace{-\frac{1}{n}\sum_{j} y_j \log p(c_j|\boldsymbol{x})}_{\text{Normal cross-entropy loss}} + \underbrace{\lambda \sum_{\mathrm{W}} \mathrm{W}^2}_{\text{Regularization term}}$$

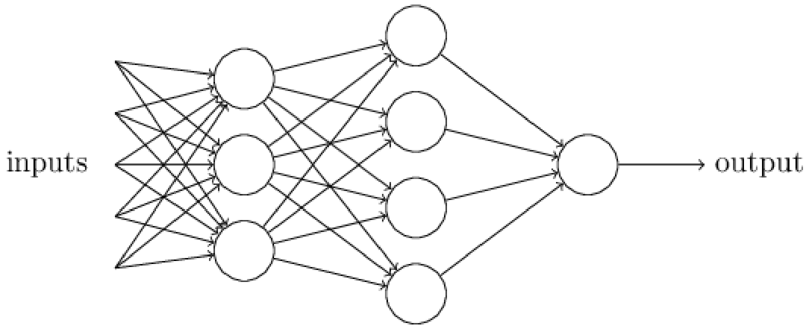
averaged over number of training data

[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.



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

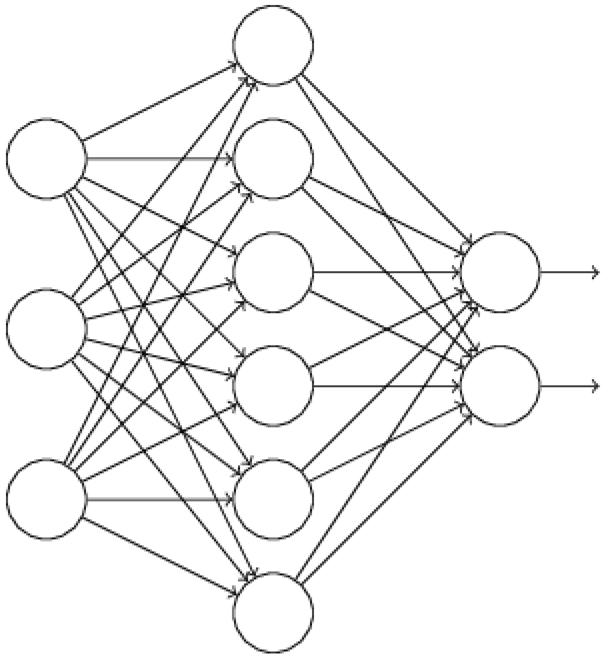
[Nielson]

Regularization

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

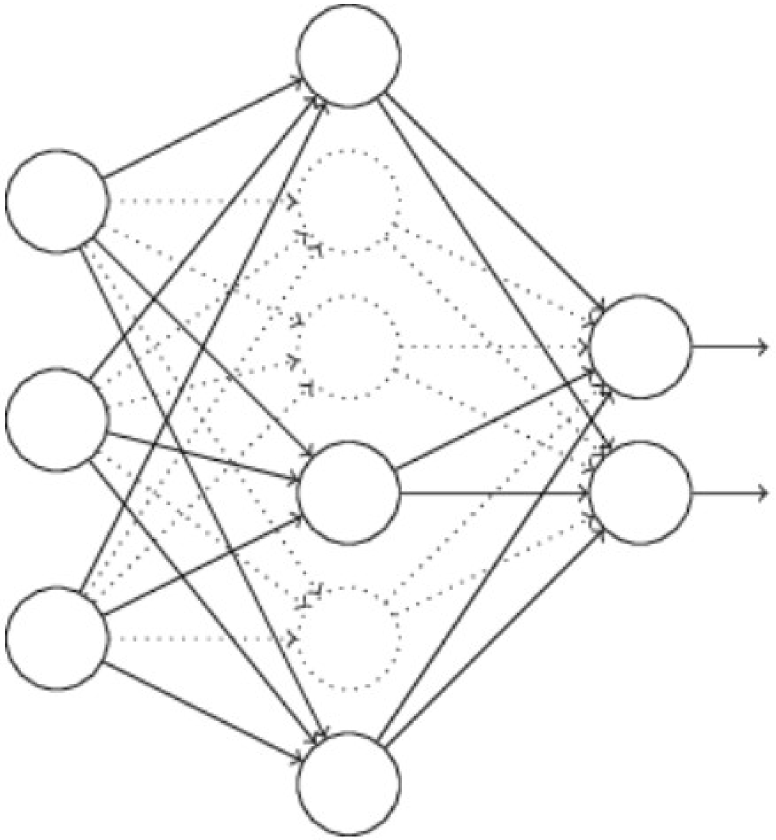
- Why not train 5 different networks with random starts and vote on their outcome?
  - Works fine!
  - Helps generalization because error is averaged.
  - BUT you need to run 5 different networks ! (slow)

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.

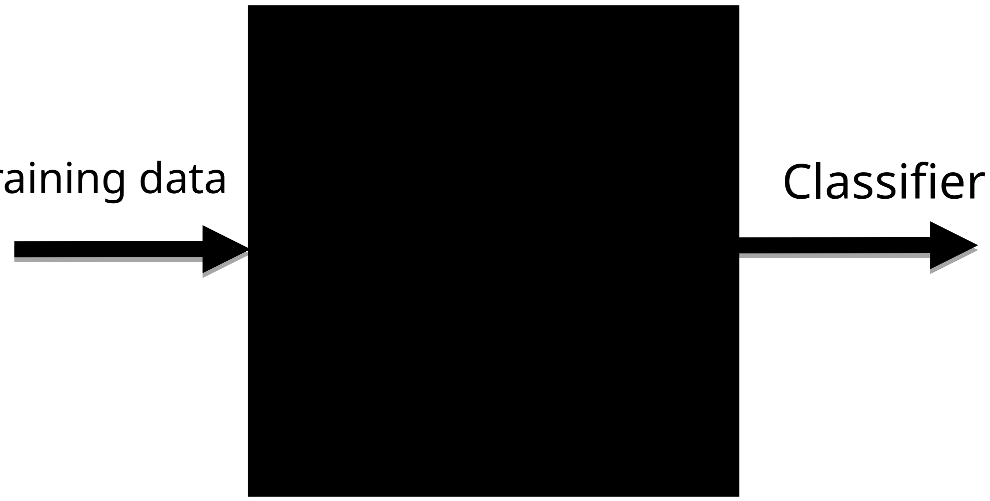
[Nielson

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

...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*.



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.