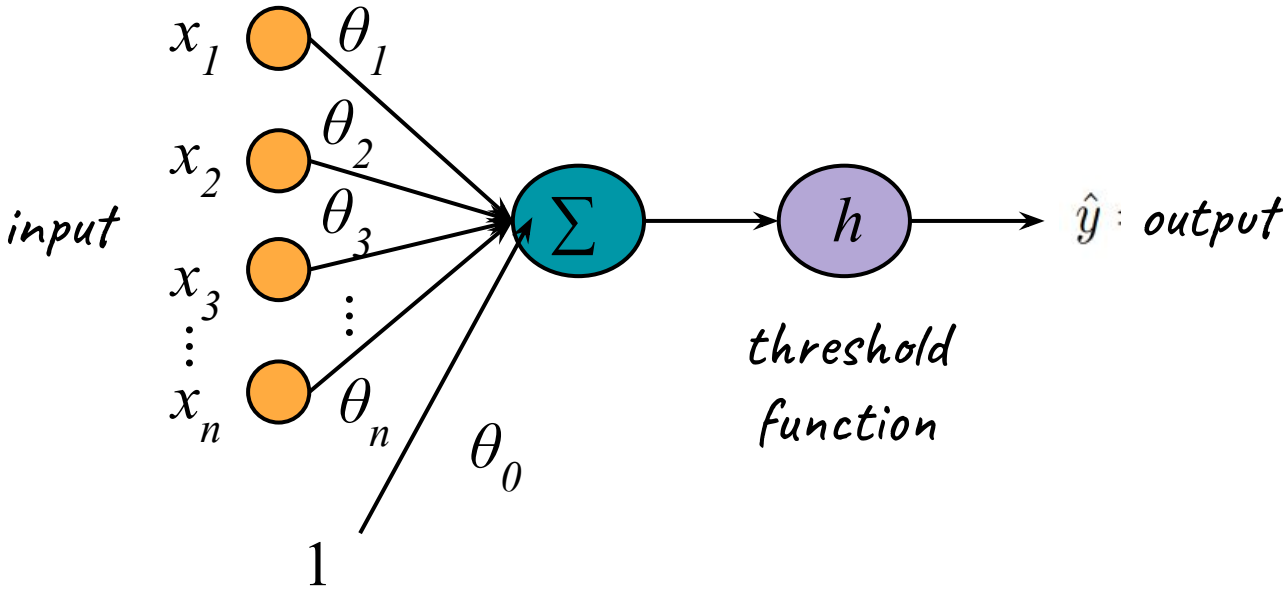
Deep Learning

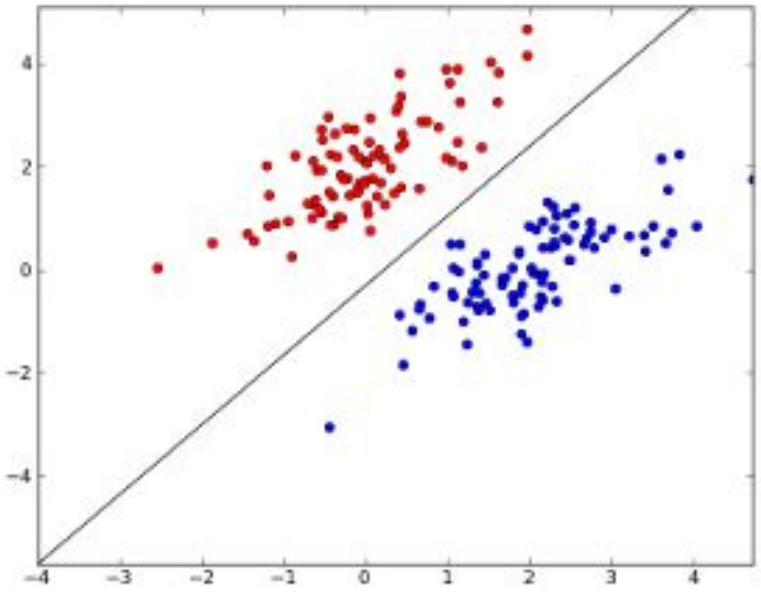
Backpropagation

What we know...

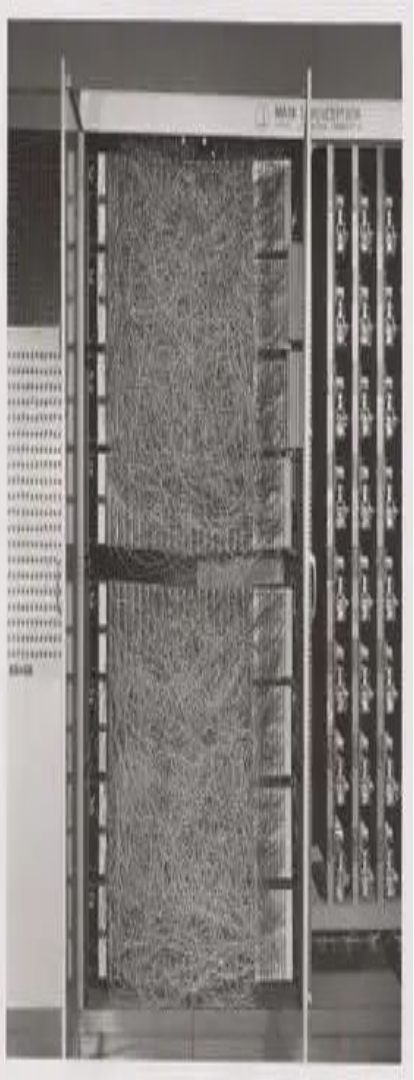
Perceptron


input
x_1
x_2
x_3
x_n
θ_1
θ_2
θ_3
θ_n
θ_0
1
Σ
h
threshold function
$\hat{y}$: output

Perceptron



Perceptron as a machine for linear classification



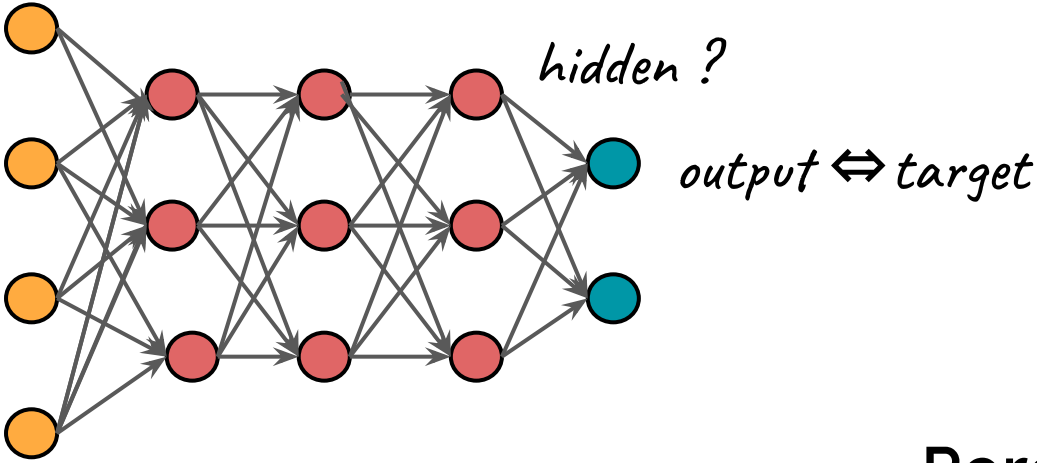
Perceptron Learning Rule

Given a training set $\mathcal{T} = \{(\mathbf{x}_1, y_1), .., (\mathbf{x}_N, y_N)\}$, $y_i \in \{0, 1\}$ and a learning rate $0 < \eta < 1$.

Algorithm

- Initialize weights θ_i and bias θ_0 to small random values.
- Iterate until the entire training set is classified correctly:
  - Select random sample $(\mathbf{x}_i, y_i)$ from $\mathcal{T}$.
  - Compute $\hat{y}_i = h(\boldsymbol{\theta}^T \mathbf{x}_i + \theta_0)$
  - Update weights and bias:
    $\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \eta(\hat{y}_i - y_i)\mathbf{x}_i$
    $\theta_0^{t+1} = \theta_0^t + \eta(\hat{y}_i - y_i)$

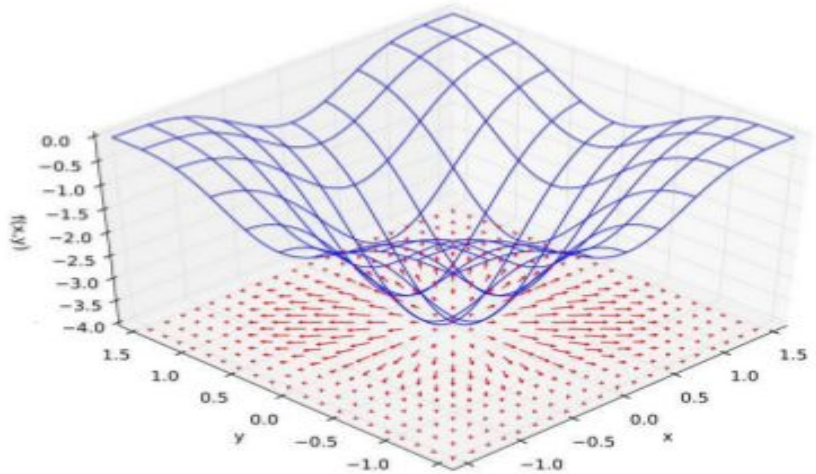
How do we learn weights?



Perceptron learning rule cannot be used

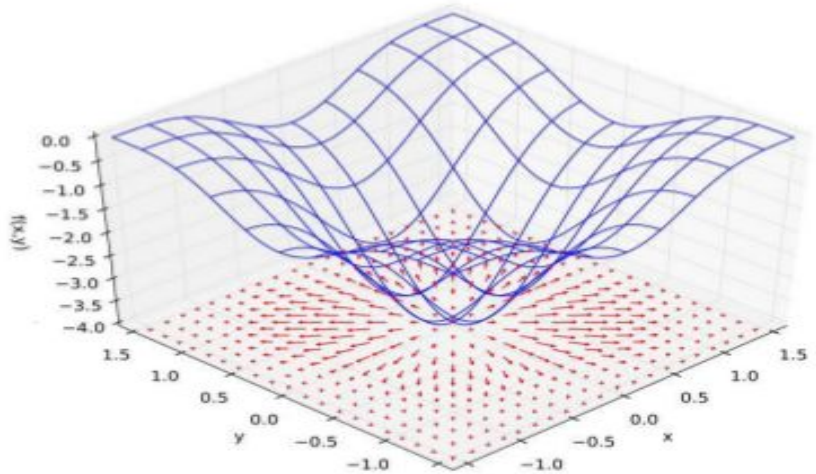
What do we need for training NNs?

- (Stochastic) Gradient Descent in conjunction with Backpropagation
- Backpropagation is a method for computing gradients

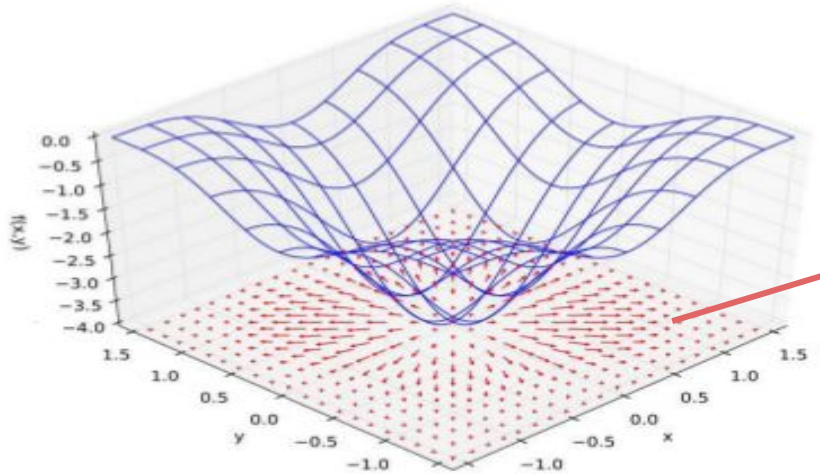


Gradient: vector of partial derivatives wrt to all the coordinates of the params

$$\nabla_{\mathbf{w}} L = \left[\frac{\partial L}{\partial w_1} \quad \frac{\partial L}{\partial w_2} \cdots \frac{\partial L}{\partial w_N} \right]$$

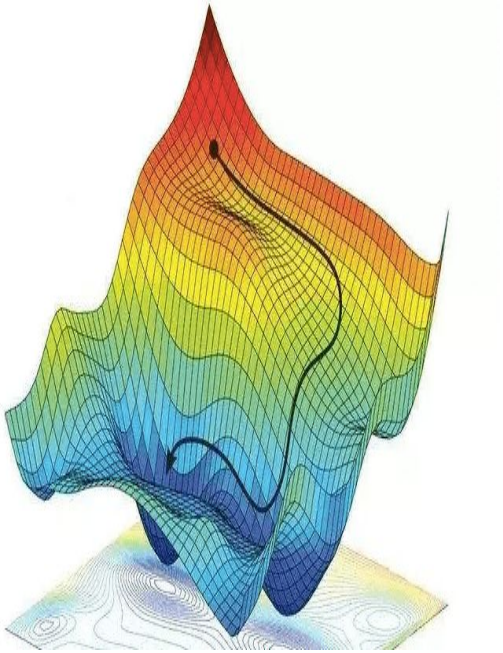


- Each partial derivative measures how fast the loss changes in one direction.
- When the gradient is zero, i.e. all the partials derivatives are zero, the loss is not changing in any direction.
- Note: the arrows (gradients) point out from a minimum



Note: the arrows (gradients) point out from a minimum

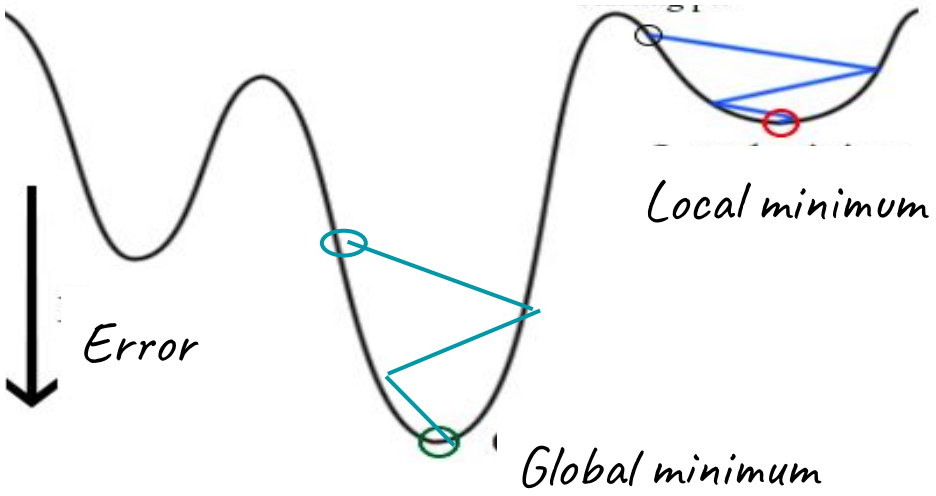
Batch Gradient Descent



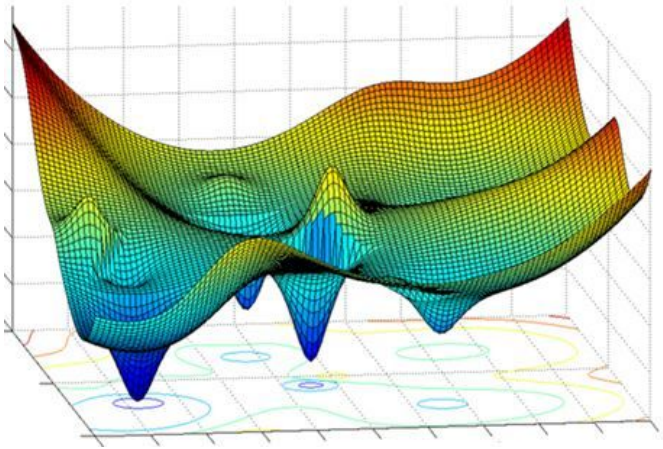
Algorithm

Require: Learning rate ϵ_k
Require: Initial Parameter θ
1: **while** stopping criteria not met **do**
2: Compute gradient estimate over N examples:
3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
5: **end while**

Considering only one direction



Local minima



➔ In neural networks the optimization problem is non-convex, it probably has local minima.

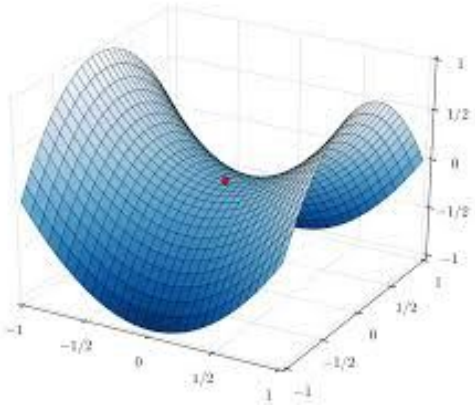
➔ This prevent people to use neural network for long time, in favour of methods that are guarantees to find the optimal solution.

But are local minima really a problem?

- Common view among practitioners: there are local minima, but they are probably still pretty good.

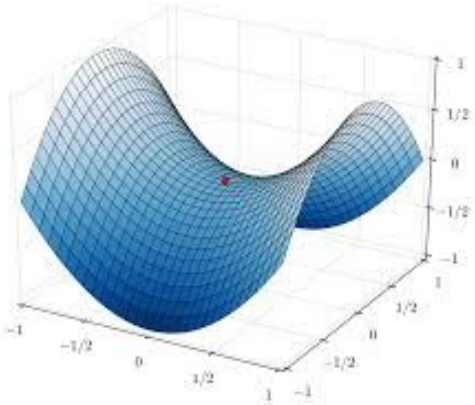
- Some other optimization-related issues are much more important.

Saddle Points



- ➔ Some directions curve upwards, and others curve downwards.
- ➔ At a saddle point, the gradient is 0 even if we are not at a minimum.
- ➔ Saddle points very common in high dimensions!

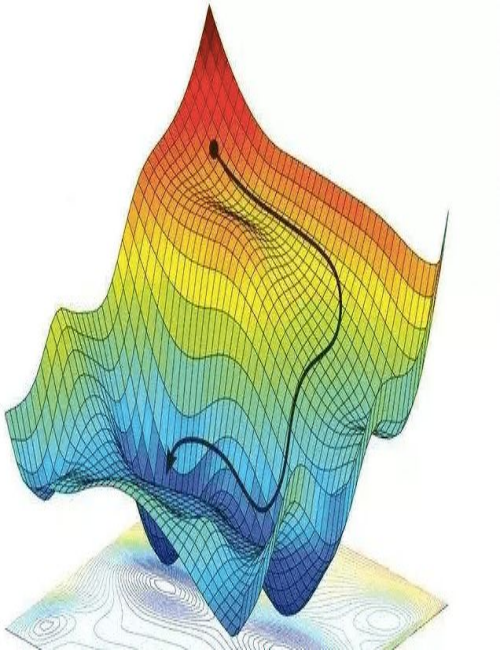
Saddle Points



When would saddle points be a problem?

- ➔ If we are exactly on the saddle point, then we are stuck.
- ➔ If we are slightly to the side, then we can get unstuck.

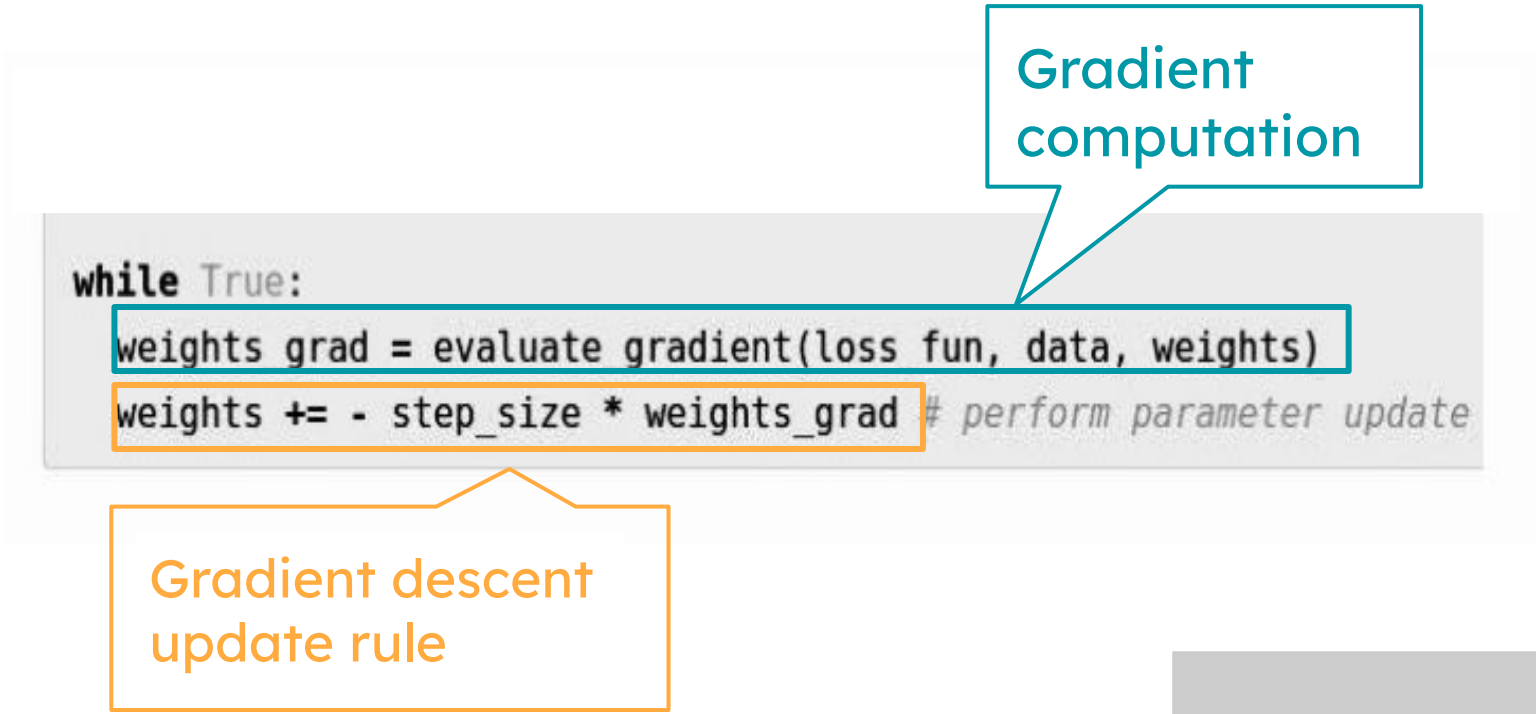
Batch Gradient Descent



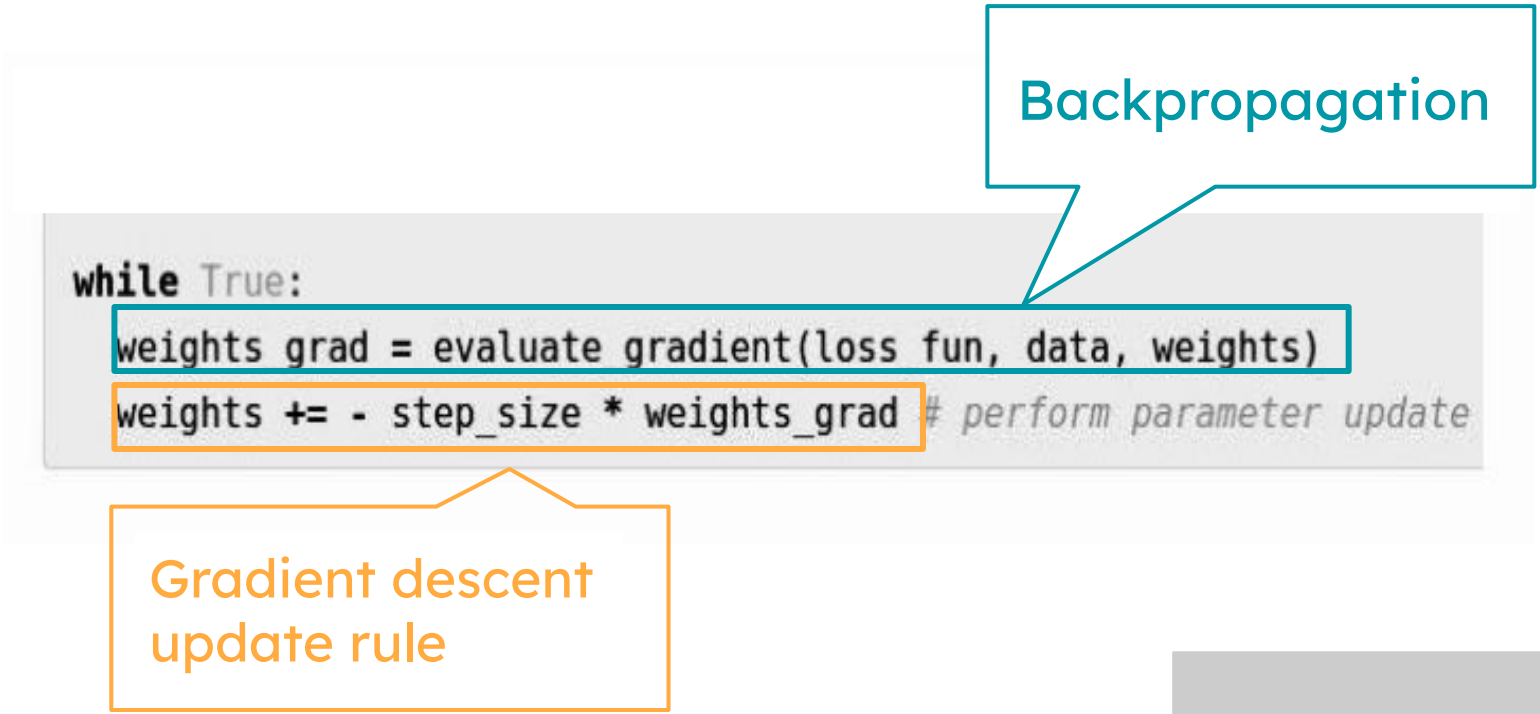
Algorithm

Require: Learning rate ϵ_k
Require: Initial Parameter θ
1: **while** stopping criteria not met **do**
2: Compute gradient estimate over N examples:
3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
5: **end while**

Batch Gradient Descent



Batch Gradient Descent



Brief History

Perceptron

Electronic Brain

XOR problem

MLP Backpropagation

1940 1950 1960 1970 1980 1990 2000 2010 2020

S. McCulloch - W. Pitts

F. Rosenblatt

M. Minsky - S. Papert

D. Rumelhart - G. Hinton - R. Wiliams

1986

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

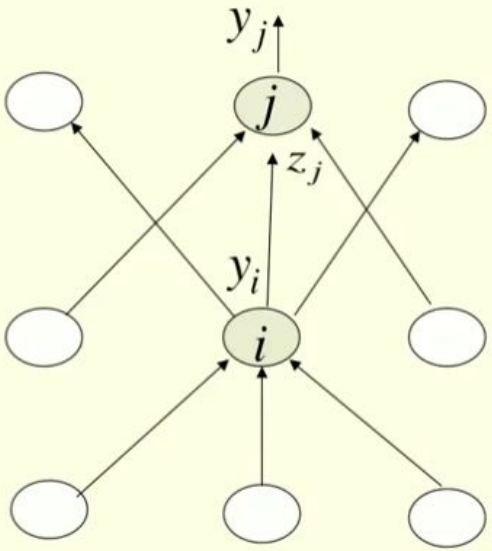
The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

Units can be given biases by introducing an extra input to each

1986

Backpropagating dE/dy



$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

How do we learn weights?

How do we learn weights?

Idea 1:

- ➔ Randomly perturb one weight, see if it improves performance, save the change
- ➔ Resemble evolutionary computations

Problems:

- ➔ Very inefficient: need to do many passes over a sample set for just one weight change
- ➔ Hard in the final part of the learning

How do we learn weights?

Idea 2:

➔ Perturb all the weights in parallel, and correlate the performance gain with weight changes

Problems:

➔ Also inefficient

➔ Very hard to implement

How do we learn weights?

Idea 3:

➔ Only perturb activations (they are fewer)

Problems:

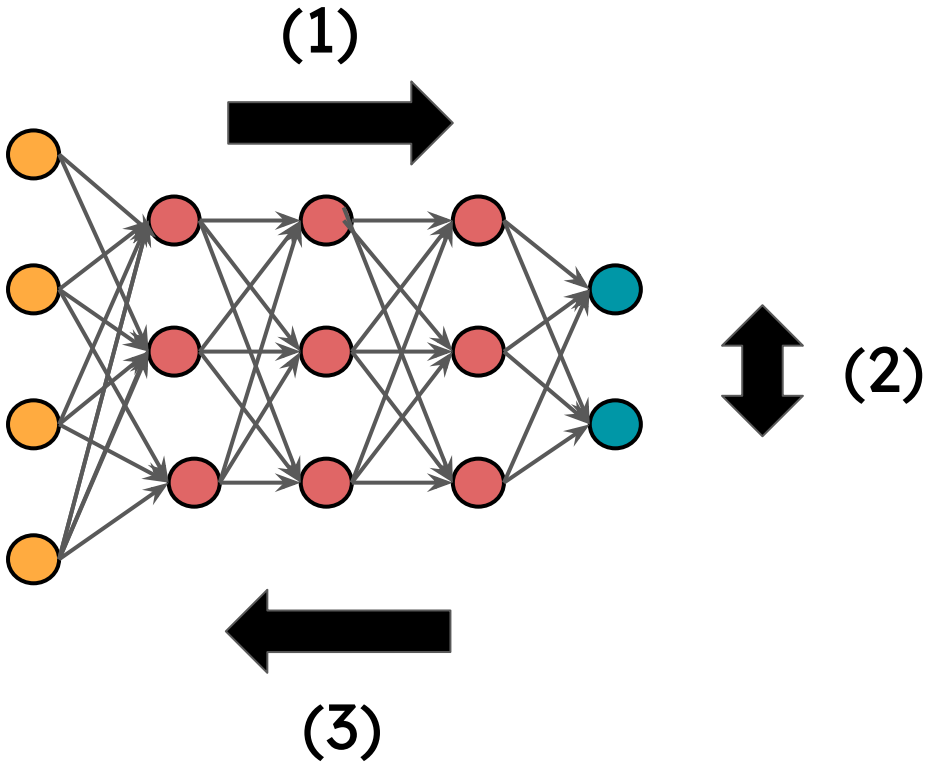
➔ Better but still inefficient

Can we do it better?

Backpropagation

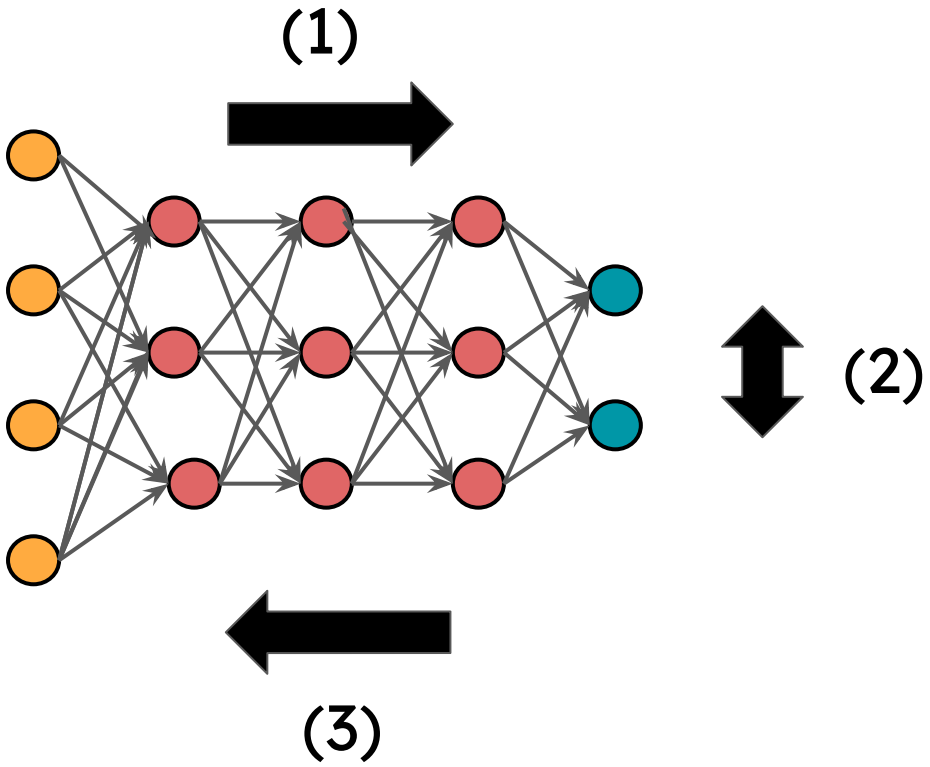
BP in a nutshell

Three steps



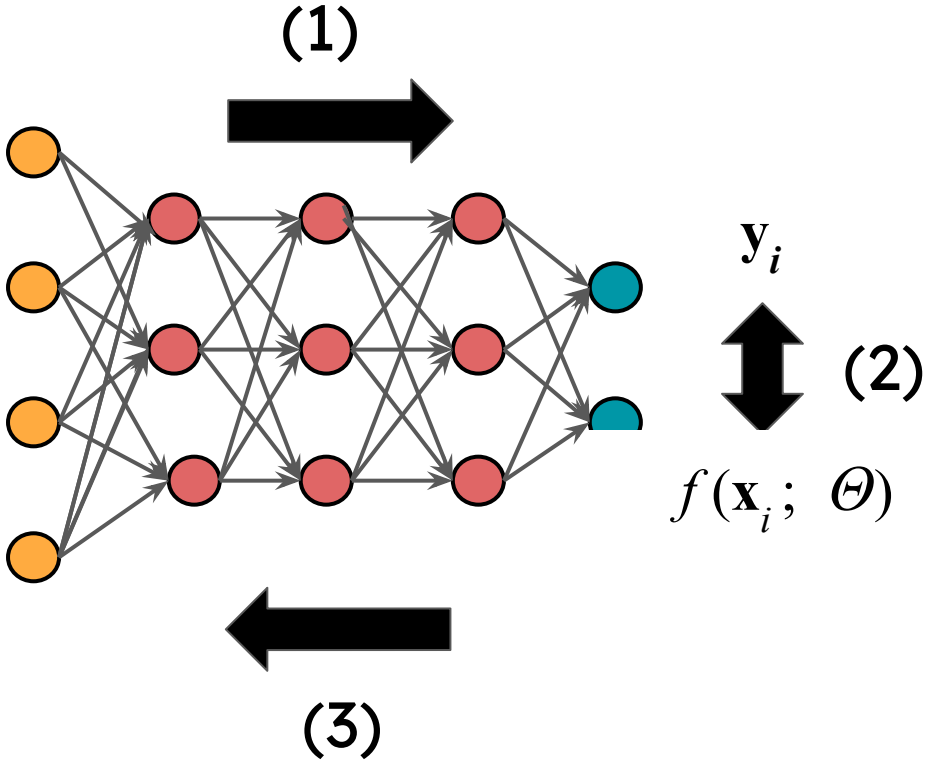
BP in a nutshell

(1) **Forward propagation**: sum inputs, produce activations, feed-forward



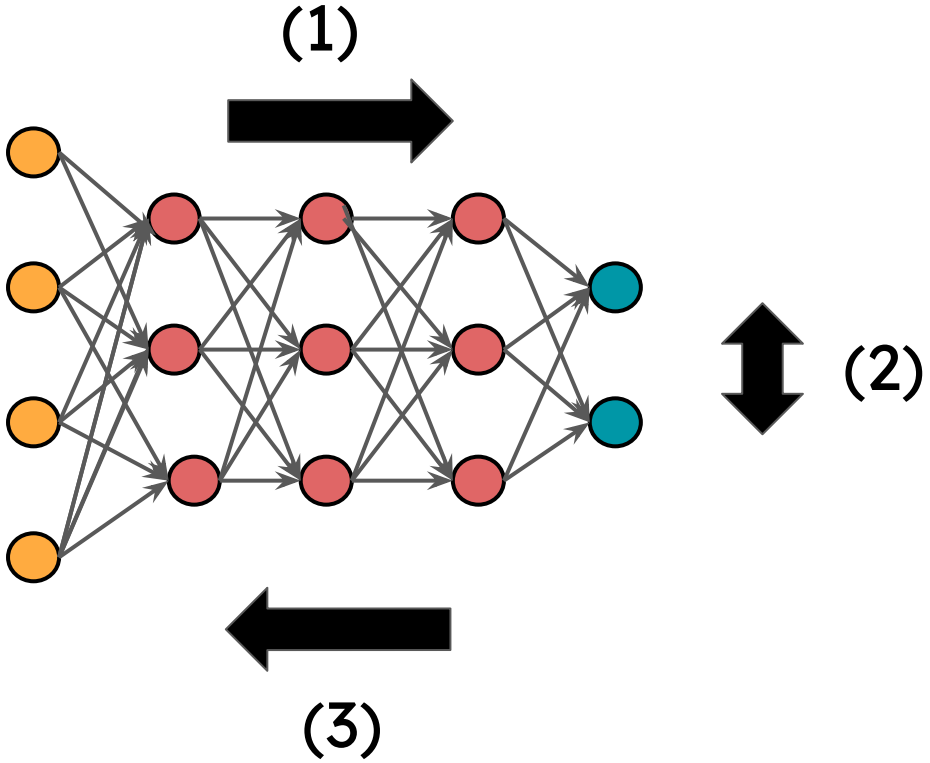
BP in a nutshell

(2) **Error estimation:** compare labels with predictions



BP in a nutshell

(3) Back propagate the error signal and used it to update weights by computing gradients



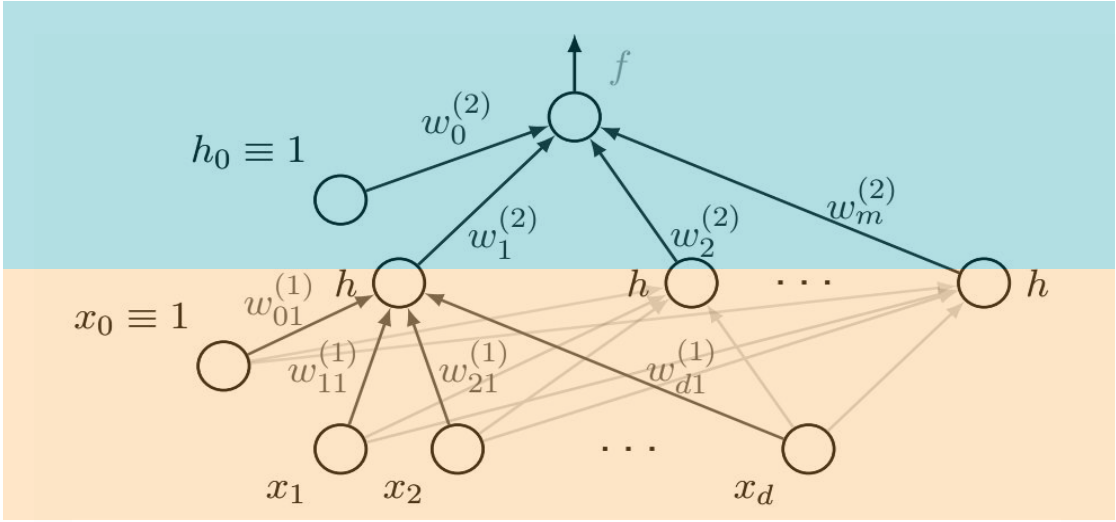
Key Ideas

- From the training data we do not know what the hidden units should do.
- But, we can compute how fast the error changes as we change a hidden activity
- Use error derivatives w.r.t. hidden activities

Key Ideas

- Each hidden unit can affect many output units and have separate effects on error. We combine them
- We can compute error derivatives for hidden units efficiently.
- Once we have error derivatives for hidden activities, easy to get error derivatives for weights going in.

Step 1: Forward



$$\hat{y}(\mathbf{x};\mathbf{w}) = f\left(\sum_{j=1}^{m} w_j^{(2)} h\left(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_0^{(2)}\right)$$

Step 2: Error

Loss function

$$L(X;\mathbf{w}) = \sum_{i=1}^{N} \frac{1}{2} \left(y_i - \hat{y}(\mathbf{x}_i;\mathbf{w})\right)^2$$

➔ No closed-form solution ⇒ gradient descent.

➔ Need to evaluate derivative of loss on a single example.

Step 2: Error

Loss function

$$L(X;\mathbf{w}) = \sum_{i=1}^{N} \frac{1}{2} \left(y_i - \hat{y}(\mathbf{x}_i;\mathbf{w})\right)^2$$

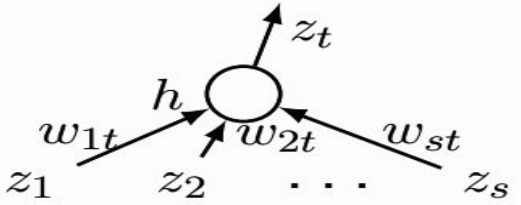
➔ We can consider a simple linear model for output $\hat{y} = \sum_j w_j x_{ij}$:

$$\frac{\partial L(\mathbf{x}_i)}{\partial w_j} = \underbrace{(\hat{y}_i - y_i)}_{\text{error}} x_{ij}.$$

Step 3: Backprop

General unit activation

$$z_t = h\left(\underbrace{\sum_j w_{jt} z_j}_{a_t}\right)$$



We can compute

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t}\frac{\partial a_t}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j$$

Step 3: Backprop

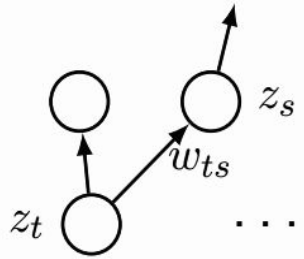
$$\frac{\partial L}{\partial w_{jt}} = \underbrace{\frac{\partial L}{\partial a_t}}_{\delta_t} z_j$$

➔ Output unit:

$$\delta_t = \hat{y} - y$$

➔ Hidden unit:

$$\delta_t = \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t} = h'(a_t) \sum_{s \in S} w_{ts} \delta_s$$

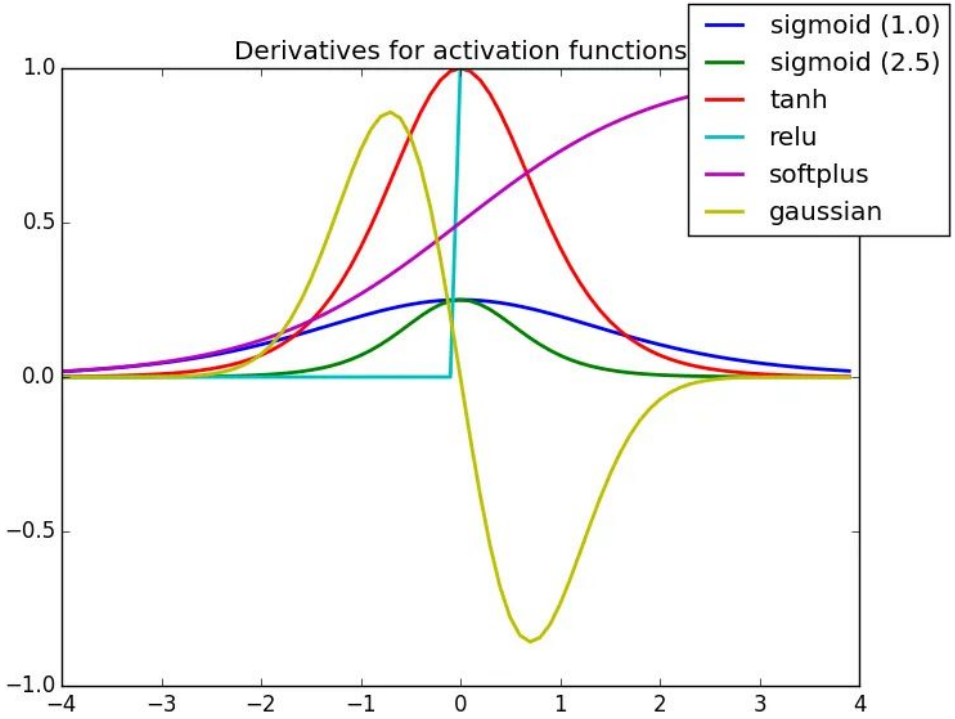


$$a_s = \sum_{j:j \to s} w_{js} h(a_j)$$

Derivatives for Activation Functions

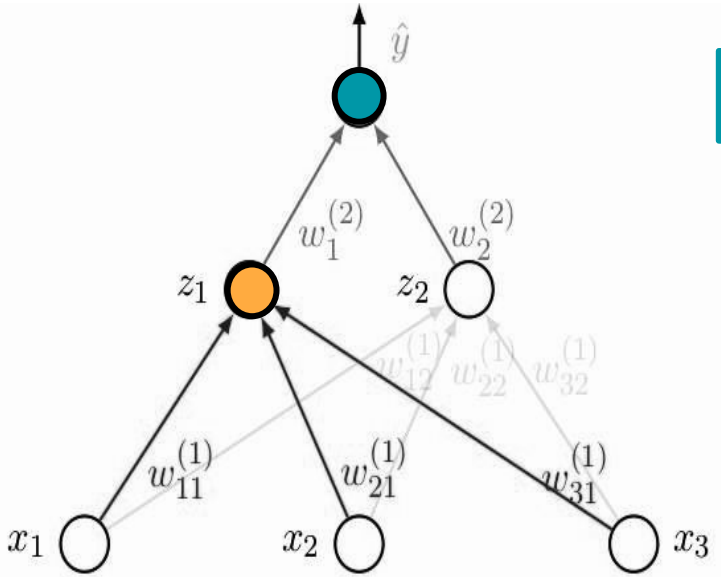
$$\delta_t = \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t}$$

$$= h'(a_t) \sum_{s \in S} w_{ts} \delta_s$$



Let’s see an example…

Forward...

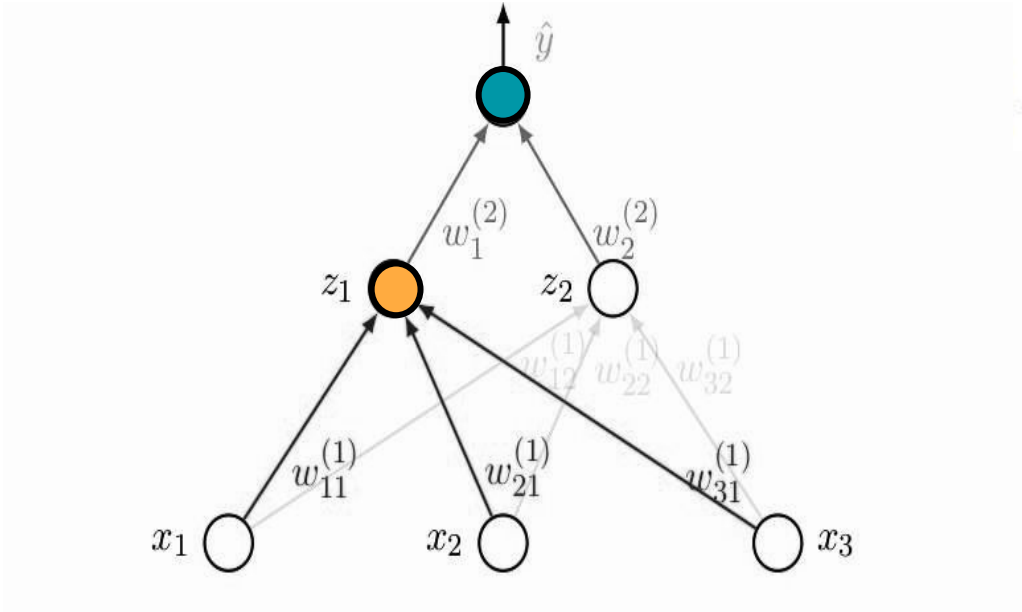


$$\hat{y} = w_1^{(2)} z_1 + w_2^{(2)} z_2$$

$$z_1 = \tanh(a_1)$$

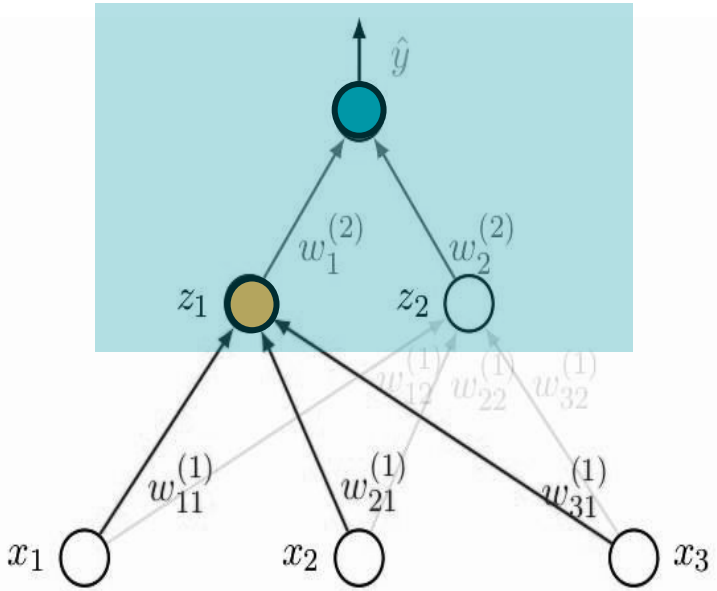
$$a_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3$$

Loss



$$L = (\hat{y} - y)^2$$

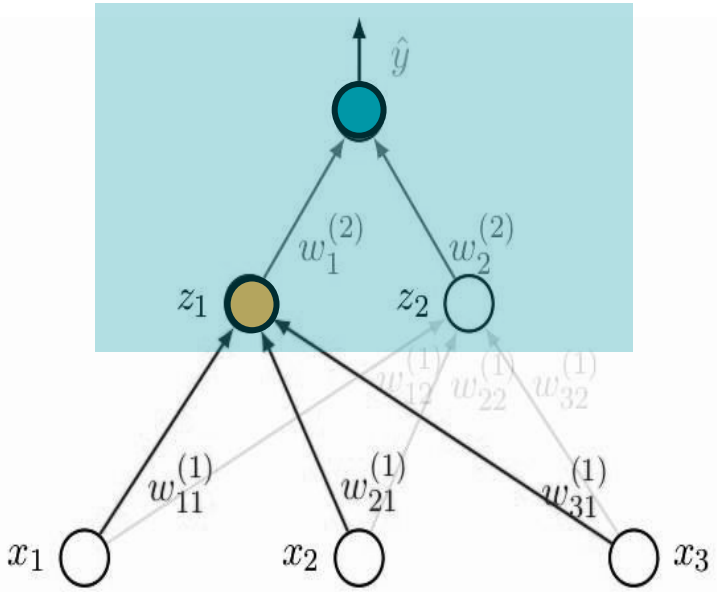
Want to assign credit for the loss to each weight



Top Layer

Gradient

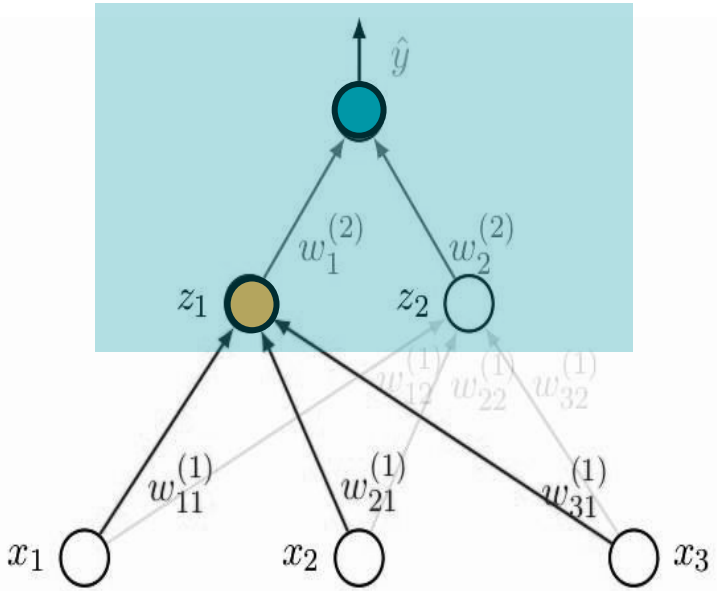
$$\frac{\partial L}{\partial w_1^{(2)}} = \frac{\partial(\hat{y}-y)^2}{\partial w_1^{(2)}} = 2(\hat{y}-y)\frac{\partial(w_1^{(2)}z_1 + w_2^{(2)}z_2)}{\partial w_1^{(2)}} = 2(\hat{y}-y)z_1$$



Top Layer

Update

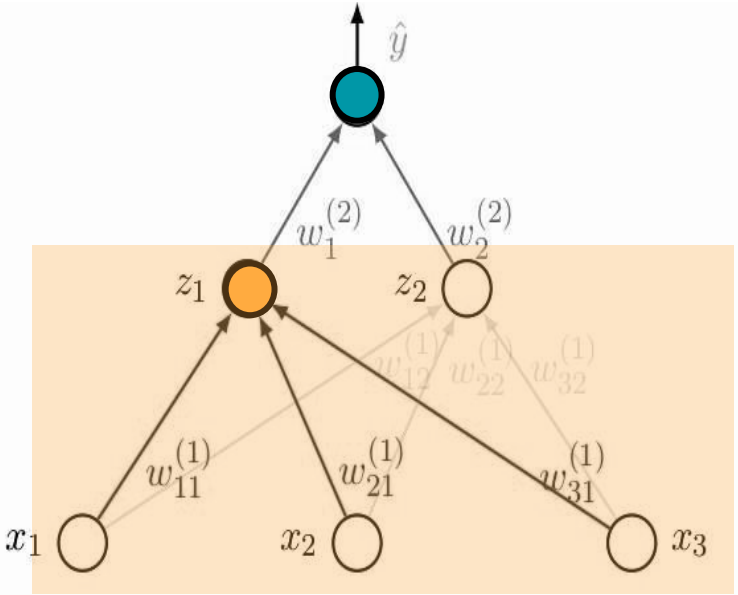
$$w_1^{(2)} := w_1^{(2)} - \eta \frac{\partial L}{\partial w_1^{(2)}} = w_1^{(2)} - \eta \delta z_1 \text{ with } \delta = (\hat{y} - y)$$



Top Layer

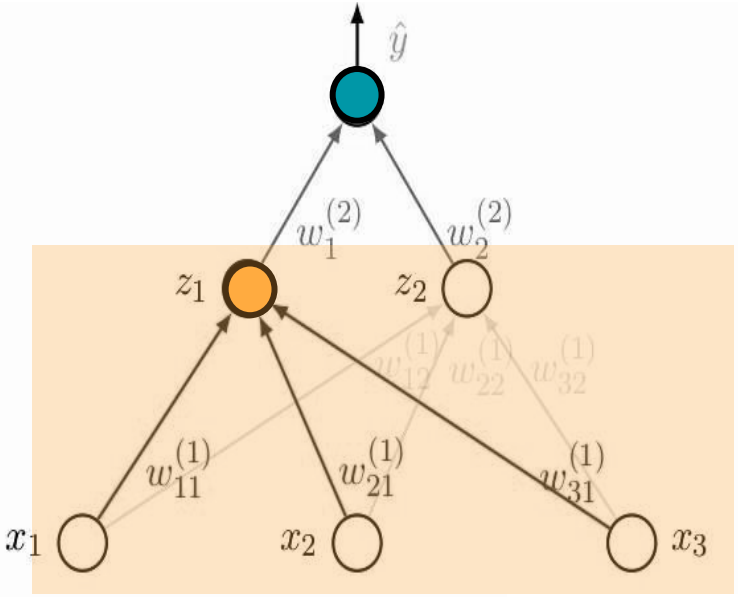
Same for other weights

$$w_2^{(2)} := w_2^{(2)} - \eta\delta z_2$$



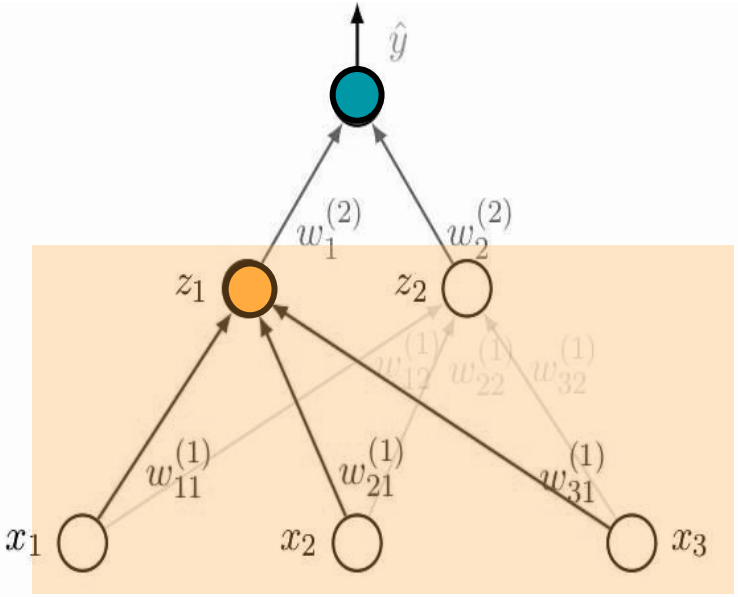
First Layer

Six weights to update



First Layer

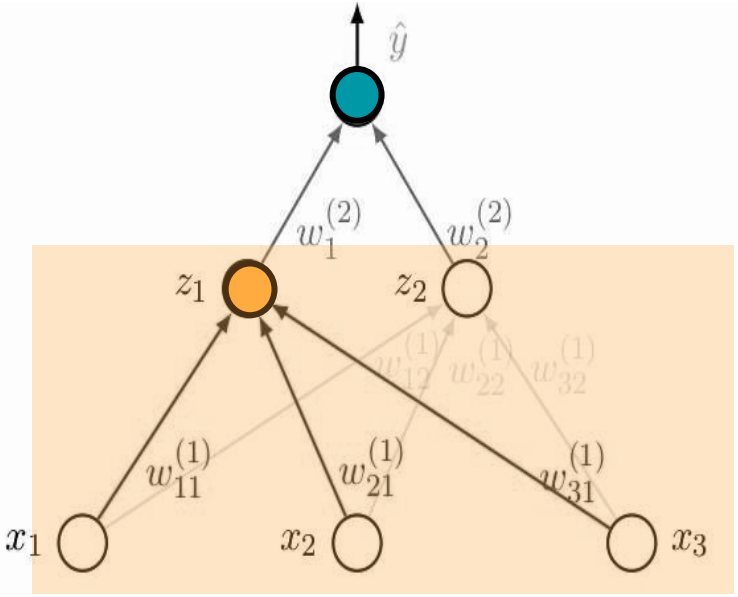
$$\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{\partial(\hat{y}-y)^2}{\partial w_{11}^{(1)}} = 2(\hat{y}-y)\frac{\partial(w_1^{(2)}z_1+w_2^{(2)}z_2)}{\partial w_{11}^{(21)}}$$



First Layer

We choose hyperbolic tangent as activation function

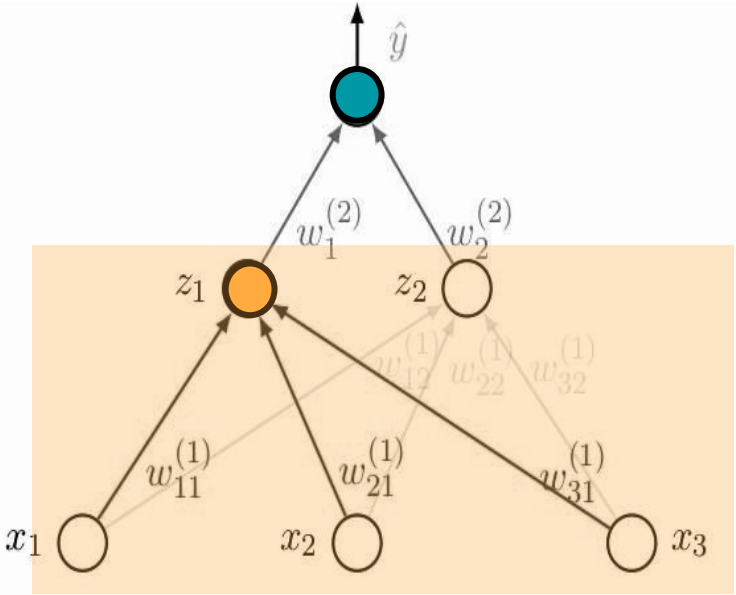
$$\frac{\partial(w_1^{(2)} z_1 + w_2^{(2)} z_2)}{\partial w_{11}^{(1)}} = w_1^{(2)} \frac{\partial(\tanh(w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3))}{\partial w_{11}^{(1)}} + 0$$



First Layer

We choose hyperbolic tangent as activation function

$$\frac{\partial L}{\partial w_{11}^{(1)}} = 2(\hat{y} - y) w_1^{(2)} (1 - \tanh^2(a_1)) x_1$$

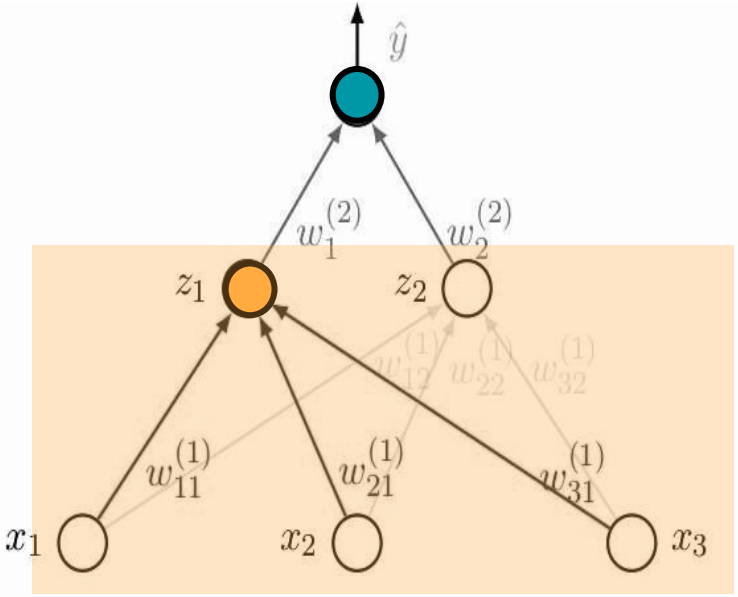


First Layer

Update

$$w_{11}^{(1)} := w_{11}^{(1)} - \eta \frac{\partial L}{\partial w_{11}^{(1)}}$$

$$\frac{\partial L}{\partial w_{11}^{(1)}} = 2(\hat{y} - y) w_1^{(2)} (1 - \tanh^2(a_1)) x_1$$



First Layer

Similarly

$$w_{22}^{(1)} := w_{22}^{(1)} - \eta \frac{\partial L}{\partial w_{22}^{(1)}}$$

$$\frac{\partial L}{\partial w_{22}^{(1)}} = 2(\hat{y} - y) w_2^{(2)} (1 - \tanh^2(a_2)) x_2$$

Summary

Top Layer

$$\frac{\partial L}{\partial w_i^{(2)}} = (\hat{y} - y) z_i = \delta z_i$$

local error

local input

Summary

Bottom Layer

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \overbrace{(\hat{y} - y) w_j^{(2)} (1 - \tanh^2(a_j))}^{\delta_j} x_i$$

local error

local input

$$\delta_j = (\hat{y} - y) w_j^{(2)} (1 - \tanh^2(a_j)) = \delta w_j^{(2)} (1 - \tanh^2(a_j))$$

Summary

Recursive computation

$$\delta_j = (\hat{y} - y) w_j^{(2)} (1 - \tanh^2(a_j)) = \delta w_j^{(2)} (1 - \tanh^2(a_j))$$

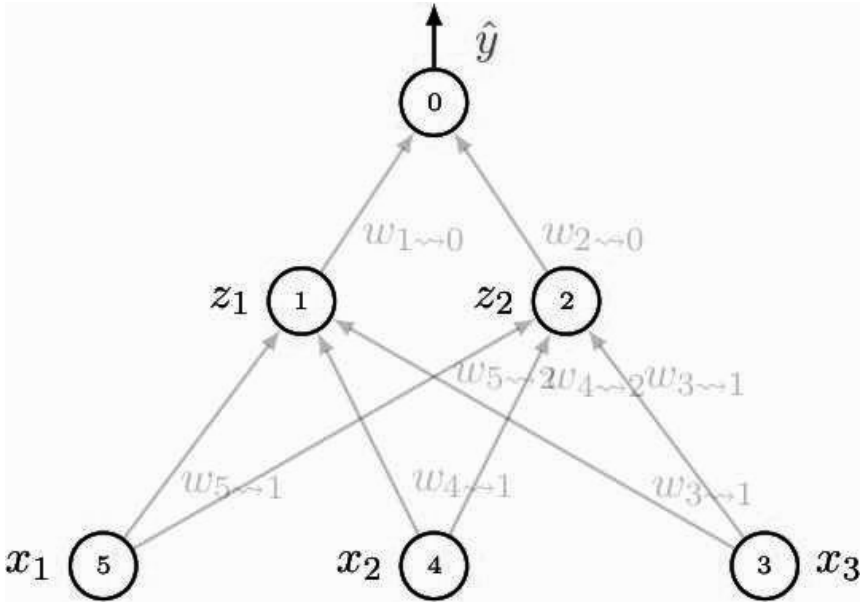
General Formulation

$$\frac{\partial L}{\partial w_{i \rightsquigarrow j}} = \delta_j z_i$$

weights FROM node i TO node j

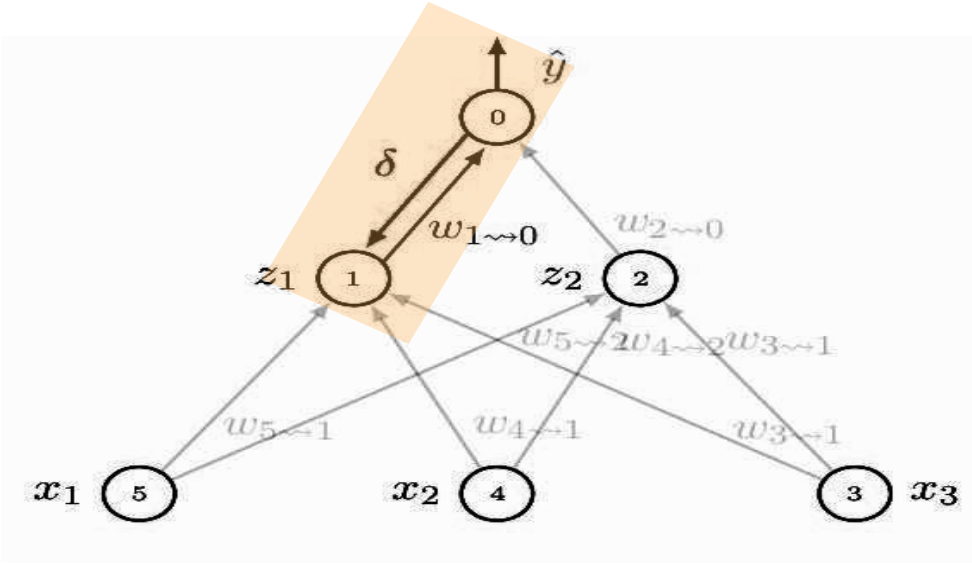
local error

local input



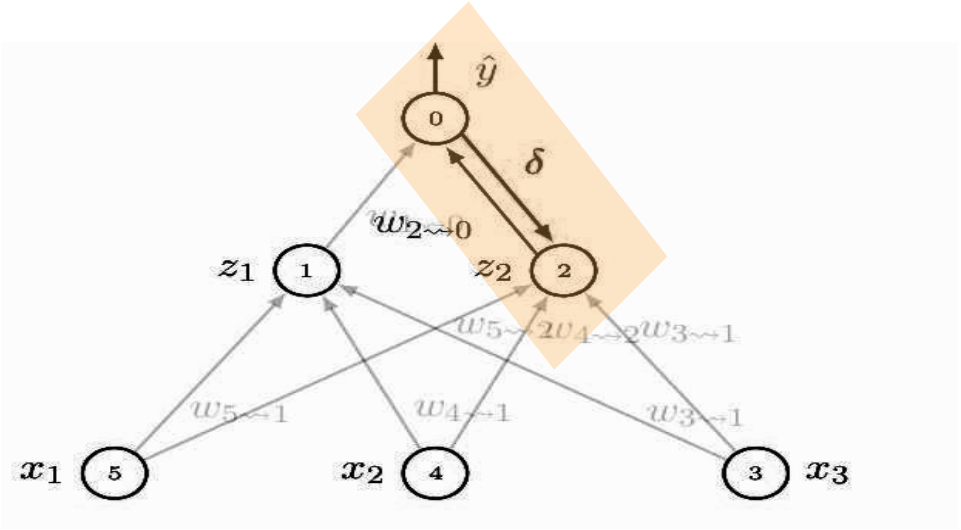
$$\frac{\partial L}{\partial w_{1\rightsquigarrow 0}} = \delta z_1$$

$$w_{1\rightsquigarrow 0} := w_{1\rightsquigarrow 0} - \eta \delta z_1$$



$$\frac{\partial L}{\partial w_{2 \rightsquigarrow 0}} = \delta z_2$$

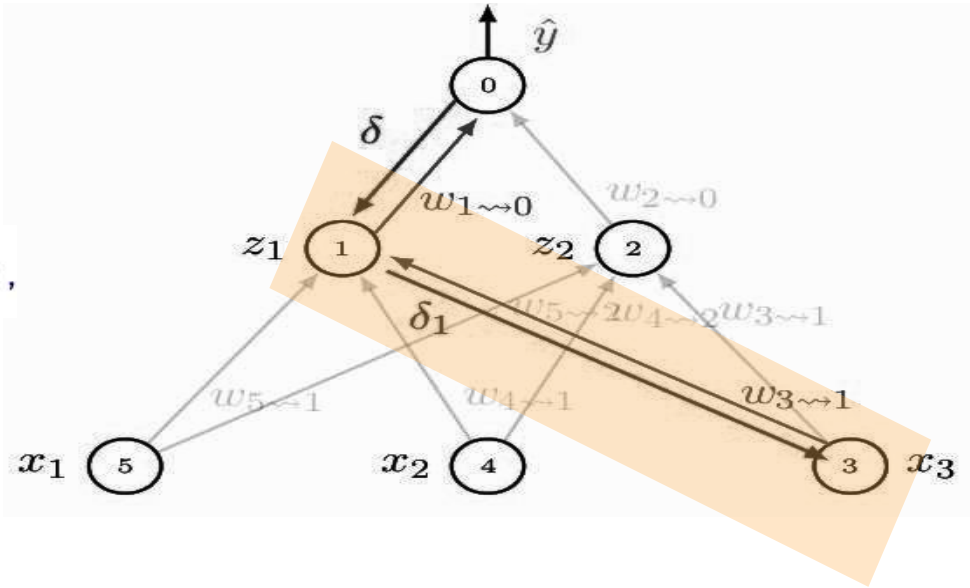
$$w_{2 \rightsquigarrow 0} := w_{2 \rightsquigarrow 0} - \eta \delta z_2$$



$$\frac{\partial L}{\partial w_{3 \rightsquigarrow 1}} = \delta_1 x_3$$

$$\delta_1 = (\delta)(w_{1 \rightsquigarrow 0})(1 - \tanh^2(a_1)),$$

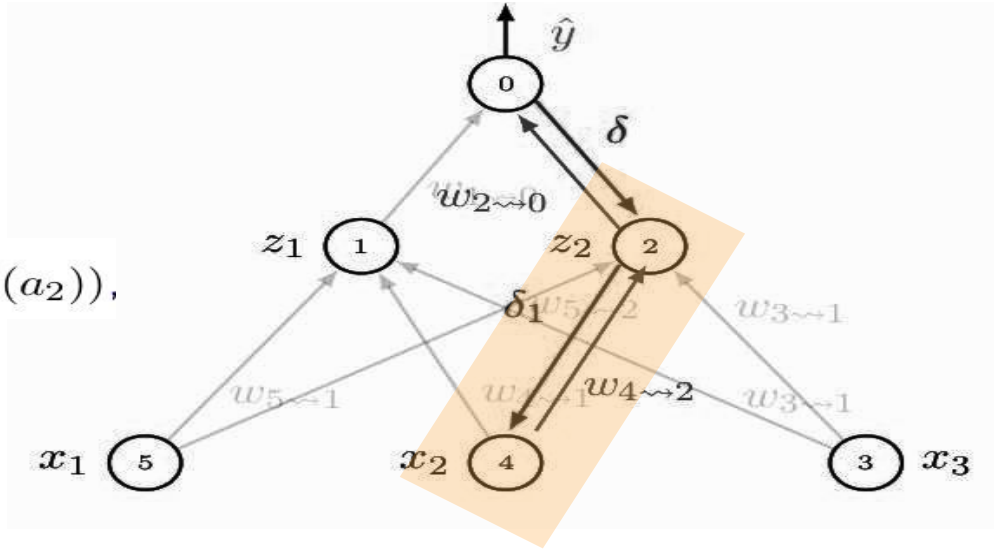
$$w_{3 \rightsquigarrow 1} := w_{3 \rightsquigarrow 1} - \eta \delta_1 x_3$$



$$\frac{\partial L}{\partial w_{4\rightsquigarrow 2}} = \delta_2 x_2$$

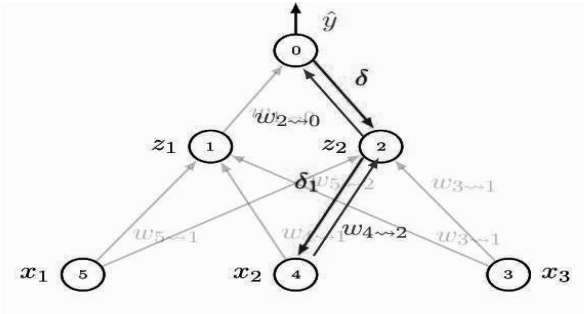
$$\delta_2 = (\delta)(w_{2\rightsquigarrow 0})(1 - \tanh^2(a_2)),$$

$$w_{4\rightsquigarrow 2} := w_{4\rightsquigarrow 2} - \eta \delta_2 x_2$$



Vector notation

$$W^{(2)} = \begin{bmatrix} w_{1 \rightsquigarrow 0} \\ w_{2 \rightsquigarrow 0} \end{bmatrix}$$



$$W^{(1)} = \begin{bmatrix} w_{5 \rightsquigarrow 1} & w_{5 \rightsquigarrow 2} \\ w_{4 \rightsquigarrow 1} & w_{4 \rightsquigarrow 2} \\ w_{3 \rightsquigarrow 1} & w_{3 \rightsquigarrow 2} \end{bmatrix}$$

$$Z^{(1)} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \text{ and } Z^{(2)} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

Forward

1. Compute $A^{(1)} = {Z^{(1)}}^T W^{(1)}$
2. Applying element-wise non-linearity $Z^{(2)} = \tanh A^{(1)}$
3. Compute Output $\hat{y} = {Z^{(2)}}^T W^{(2)}$
4. Compute Loss on example $(\hat{y} - y)^2$

Backward

1. Top: Compute δ
2. Gradient w.r.t $W^{(2)} = \delta Z^{(2)}$
3. Compute $\delta_1 = ({W^{(2)}}^T \delta) \odot (1 - \tanh(A^{(1)})^2)$
   Notes: (a): $\odot$ is Hadamard product. (b) have written ${W^{(2)}}^T \delta$ as δ can be a vector when there are multiple outputs
4. Gradient w.r.t $W^{(1)} = \delta_1 Z^{(1)}$
5. Update $W^{(2)} := W^{(2)} - \eta \delta Z^{(2)}$
6. Update $W^{(1)} := W^{(1)} - \eta \delta_1 Z^{(1)}$

Summary

- Backpropagation is all about assigning credit (or blame!) for error incurred to the weights
- We follow the path from the output (where we have an error signal) to the edge we want to consider
- We find the δs from the top to the edge concerned by using the chain rule

Summary

- Once we have the partial derivative, we can write the update rule for that weight
- As we go down the network we need previous δs
- Need to book-keep derivatives as we go down the network and reuse them

Research continues

2022

The Forward-Forward Algorithm: Some Preliminary Investigations

Geoffrey Hinton
Google Brain
geoffhinton@google.com

Abstract

The aim of this paper is to introduce a new learning procedure for neural networks and to demonstrate that it works well enough on a few small problems to be worth serious investigation. The Forward-Forward algorithm replaces the forward and backward passes of backpropagation by two forward passes, one with positive (*i.e.* real) data and the other with negative data which could be generated by the network itself. Each layer has its own objective function which is simply to have high goodness for positive data and low goodness for negative data. The sum of the squared activities in a layer can be used as the goodness but there are many other possibilities, including minus the sum of the squared activities. If the positive and negative passes can be separated in time, the negative passes can be done offline, which makes the learning much simpler in the positive pass and allows video to be pipelined through the network without ever storing activities or stopping to propagate derivatives.

1 What is wrong with backpropagation

The astonishing success of deep learning over the last decade has established the effectiveness of performing stochastic gradient descent with a large number of parameters and a lot of data. The gradients are usually computed using backpropagation (Rumelhart et al., 1986), and this has led to a lot of interest in whether the brain implements backpropagation or whether it has some other way of getting the gradients needed to adjust the weights on connections.

As a model of how cortex learns, backpropagation remains implausible despite considerable effort to invent ways in which it could be implemented by real neurons (Lillicrap et al., 2020; Richards and Lillicrap, 2019; Guerguiev et al., 2017; Scellier and Bengio, 2017). There is no convincing evidence that cortex explicitly propagates error derivatives or stores neural activities for use in a subsequent backward pass. The top-down connections from one cortical area to an area that is earlier in the visual pathway do not mirror the bottom-up connections as would be expected if backpropagation was being used in the visual system. Instead, they form loops in which neural activity goes through about half a dozen cortical layers in the two areas before arriving back where it started.

Backpropagation through time as a way of learning sequences is especially implausible. To deal with the stream of sensory input without taking frequent time-outs, the brain needs to pipeline sensory data through different stages of sensory processing and it needs a learning procedure that can learn on the fly. The representations in later stages of the pipeline may provide top-down information that influences the representations in earlier stages of the pipeline *at a later time step*, but the perceptual system needs to perform inference and learning in real time without stopping to perform backpropagation.

Preprint. Under review.

Our brains does not perform backprop

1 What is wrong with backpropagation

The astonishing success of deep learning over the last decade has established the effectiveness of performing stochastic gradient descent with a large number of parameters and a lot of data. The gradients are usually computed using backpropagation (Rumelhart et al., 1986), and this has led to a lot of interest in whether the brain implements backpropagation or whether it has some other way of getting the gradients needed to adjust the weights on connections.

As a model of how cortex learns, backpropagation remains implausible despite considerable effort to invent ways in which it could be implemented by real neurons (Lillicrap et al., 2020; Richards and Lillicrap, 2019; Guerguiev et al., 2017; Scellier and Bengio, 2017). There is no convincing evidence that cortex explicitly propagates error derivatives or stores neural activities for use in a subsequent backward pass. The top-down connections from one cortical area to an area that is earlier in the visual pathway do not mirror the bottom-up connections as would be expected if backpropagation was being used in the visual system. Instead, they form loops in which neural activity goes through about half a dozen cortical layers in the two areas before arriving back where it started.

Backpropagation through time as a way of learning sequences is especially implausible. To deal with the stream of sensory input without taking frequent time-outs, the brain needs to pipeline sensory data through different stages of sensory processing and it needs a learning procedure that can learn on the fly. The representations in later stages of the pipeline may provide top-down information that influences the representations in earlier stages of the pipeline *at a later time step*, but the perceptual system needs to perform inference and learning in real time without stopping to perform backpropagation.

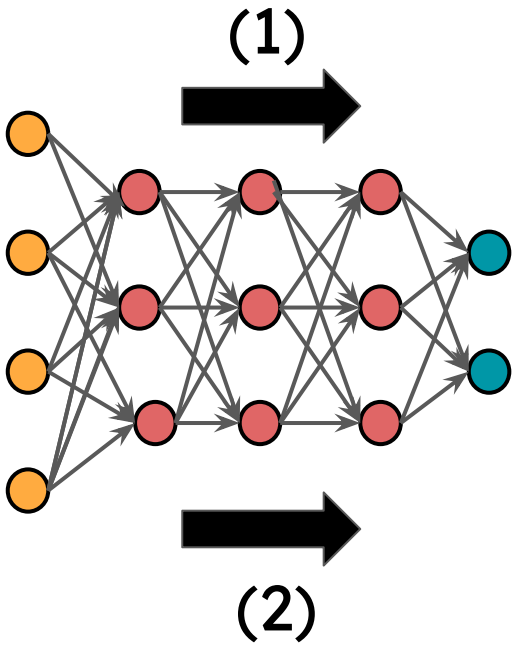
Backprop

- In the backprop step, the error is propagated back through the layers and the weights are updated.
- Backpropagation uses the partial derivatives and the chain rule to calculate gradients and adjust each weights.

Problem with Backprop

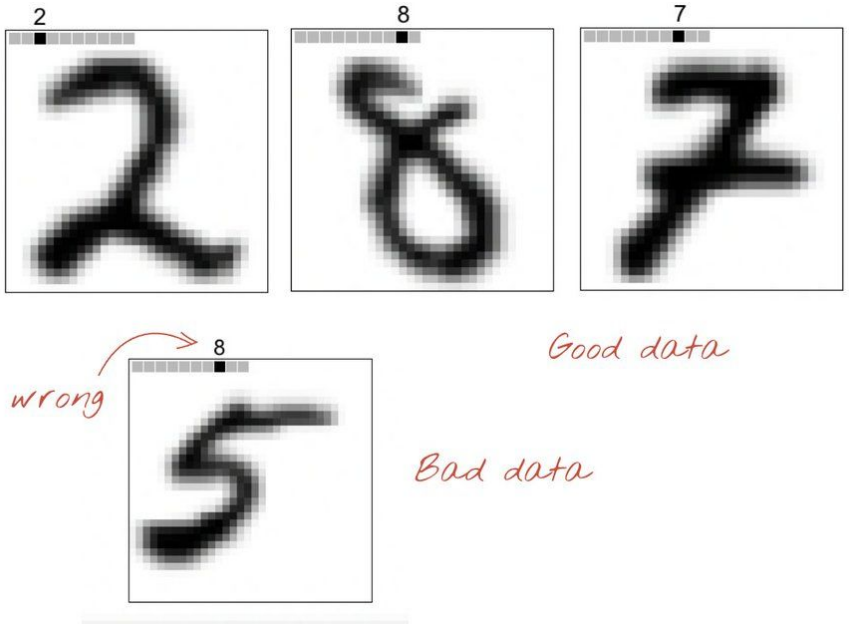
- **Problem 1:** detachment of learning and inference. To adjust the weights of a neural network, the training algorithm must stop inference to perform backpropagation. In the real world our brain receives a constant stream of information.

- **Problem 2:** Backpropagation does not work if the computation done in the forward pass is not differentiable.



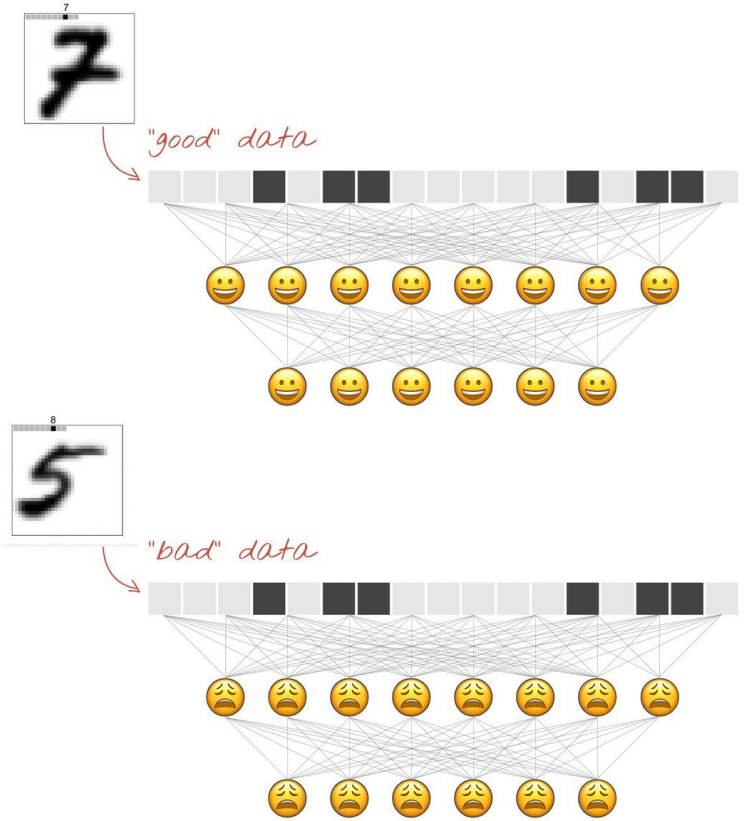
Replace the forward and backward passes with **two forward passes**.

The two passes are similar, but they work on **different data** and have **opposite objectives.**



Positive pass: real data, adjusts the weights of to increase the “goodness” of each layer.

Negative pass: negative data and adjusts the weights to reduce “goodness”.



“Goodness”: sum of squared neural activities and the negative sum of squared activities.

probability that an input vector is positive (*i. e.* real) is given by applying the logistic function, σ to the goodness, minus some threshold, θ

$$p(positive) = \sigma\left(\sum_j y_j^2 - \theta\right) \quad (1)$$

where y_j is the activity of hidden unit j before layer normalization. The negative data may be predicted by the neural net using top-down connections, or it may be supplied externally.

Advantages

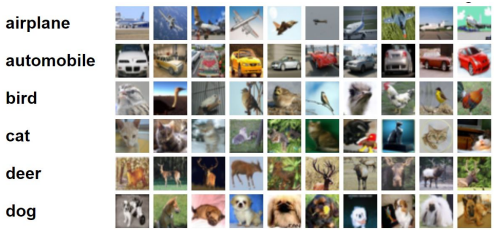
- Benefit from self-supervised learning that can make the NNs more robust while reducing the need for manually labeled examples. Masking techniques can create negative examples to train the model.

- FF algorithm does not require differentiable functions, it can still tune its trainable parameters without knowing the inner workings of every layer in the model.

Results

learning procedure	testing procedure	number of hidden layers	training % error rate	test % error rate
BP		2	0	37
FF min ssq	compute goodness for every label	2	20	41
FF min ssq	one-pass softmax	2	31	45
FF max ssq	compute goodness for every label	2	25	44
FF max ssq	one-pass softmax	2	33	46
BP		3	2	39
FF min ssq	compute goodness for every label	3	24	41
FF min ssq	one-pass softmax	3	32	44
FF max ssq	compute goodness for every label	3	21	44
FF max ssq	one-pass softmax	3	31	46

Table 1: A comparison of backpropagation and FF on CIFAR-10 using non-convolutional nets with local receptive fields of size 11x11 and 2 or 3 hidden layers. One version of FF is trained to maximize the sum of the squared activities on positive cases. The other version is trained to minimize it and this gives slightly better test performance. After training with FF, a single forward pass through the net is a quick but sub-optimal way to classify an image. It is better to run the net with a particular label as the top-level input and record the average goodness over the middle iterations. After doing this for each label separately, the label with the highest goodness is chosen. For a large number of labels, a single forward pass could be used to get a candidate list of which labels to evaluate more thoroughly.

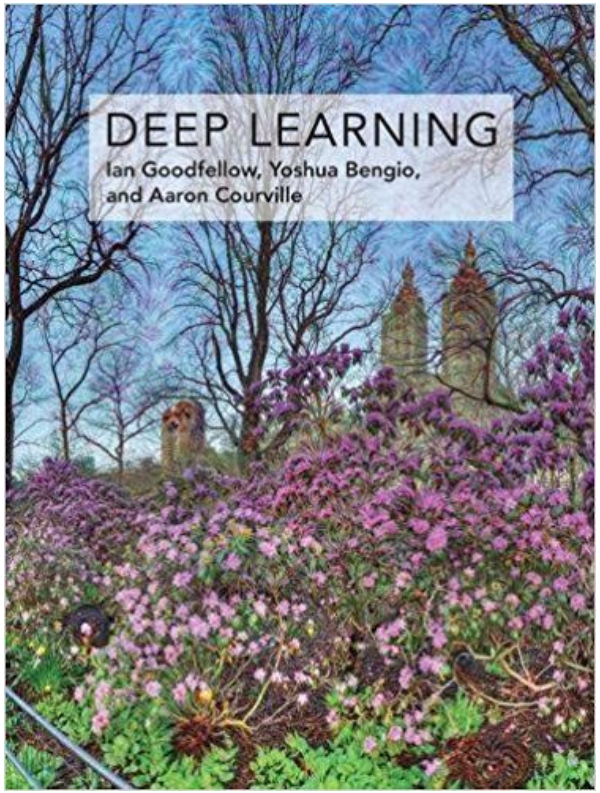


What’s next?

NEURAL INFORMATION PROCESSING SYSTEMS

Readings

★ Deep Learning Book - Chapter 6.



Questions?