Machine Learning

Lecture 8 & 9: Deep Learning

Prof. Dr. Stephan Günnemann

Data Mining and Analytics
Technische Universität München

10.12.2018 & 17.12.2018

Reading material

- Goodfellow, Deep Learning: chapters 6, 7, 11
- Bishop: chapters 5.1, 5.2, 5.3, 5.5

Acknowledgements

- Slides based on an earlier version by Patrick van der Smagt

Another look at Logistic Regression

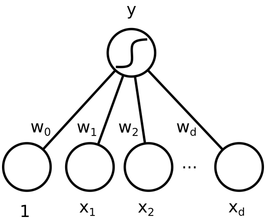
We had before:

$$y \mid \boldsymbol{x} \sim \mathrm{Bernoulli}\Big(\sigma\big(\boldsymbol{w}^T \boldsymbol{x} + w_0\big)\Big)$$

$$\boldsymbol{w}^T \boldsymbol{x} := w_0 + w_1 x_1 + \ldots + w_D x_D$$

We can represent this graphically
(the first node $1 = x_0$ is for the bias term):

- each node is a (scalar) input
- multiply the input with the weight on the edge: $x_i w_i$
- compute weighted sum of incoming edges: $a_0 = \sum_{i=0}^{D} x_i w_i$
- apply (activation) function: $y = \sigma(a_0)$



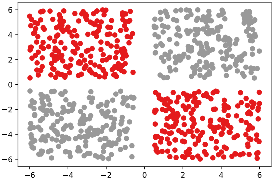
The XOR dataset

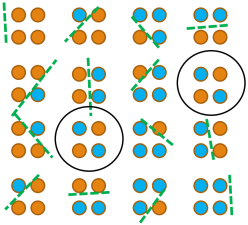
The XOR dataset is not linearly separable
$\rightarrow$ Logistic Regression will fail since it
learns a linear decision boundary

In general:

- $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} \in \mathbb{R}^d$
- Given n points there are 2^n dichotomies
- Only $2 \cdot \sum_{i=0}^{d} \binom{n-1}{i}$ are linearly separable[1]
- With $n > d$ the probability that $\mathcal{X}$ is linearly separable goes to 0

$O(N^D)$





How to handle non-linearity? Basis functions

We have input vectors $\boldsymbol{x}$ and associated output values y. We want to describe the underlying functional relation.

We can use the following simple model:

$$f(\boldsymbol{x}, \boldsymbol{w}) = \sigma(w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\boldsymbol{x})) = \sigma(\boldsymbol{w}^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x})) \tag{1}$$

where

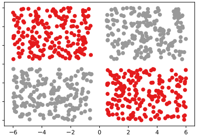
$\boldsymbol{\phi}$	basis function	—	many choices, can be nonlinear
w_0	bias	—	equivalent to defining $\phi_0 \equiv 1$; or adding constant 1 to every sample

Remember we are linear in $\boldsymbol{w}$!

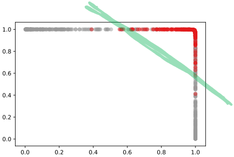
Example: Handling XOR dataset by custom basis functions

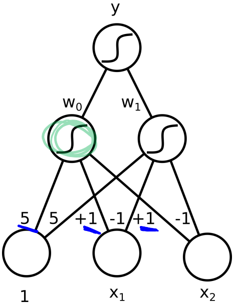
Apply a (nonlinear) transformation ϕ that maps samples to a space where they are linearly separable. For example:

$\mathbb{R}^2$ space



Transformed $\mathbb{R}^2$ space





Here we defined a custom basis function $\boldsymbol{\phi} : \mathbb{R}^3 \to \mathbb{R}^2$

$$\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{\phi}(1, x_1, x_2) = (\ \sigma(5 + x_1 + x_2)\ ,\ \sigma(5 - x_1 - x_2)\ )$$

Example: Handling XOR dataset by custom basis functions

Overall function that is modeled:

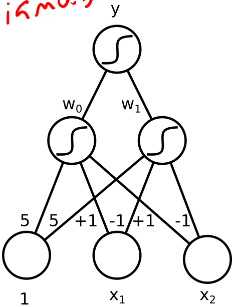
$$f(\boldsymbol{x},\boldsymbol{w}) = \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x})) = \sigma_1\Big(\begin{bmatrix} w_0 & w_1 \end{bmatrix} \cdot \sigma_0\big(\overbrace{\begin{bmatrix} 5 & 1 & 1 \\ 5 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}}^{\mathbb{R}^2} \big)\Big)$$

sigmoid E.G. sigmoid

How to find the parameters $\boldsymbol{w}$?

Just train the model by minimizing a corresponding loss function.

Here: binary cross-entropy since binary classification problem.



$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{N} -\Big(y_i \log f(\boldsymbol{x}_i,\boldsymbol{w}) + (1-y_i)\log\big[1 - f(\boldsymbol{x}_i,\boldsymbol{w})\big]\Big)$$

How to find the basis functions?

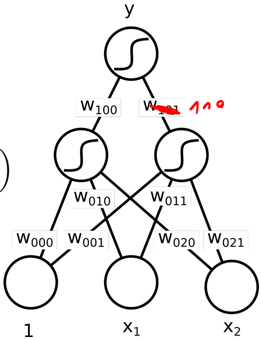
Different datasets require different transformations to become (almost) linearly separable.

Idea: learn the basis functions and the weights of the logistic regression **jointly** from the data (end-to-end learning)

- Previously: only learning w_{100} and w_{110}
- Now: Learn all w_{ijk} where i=layer, j = input node, k = output node

$$f(\boldsymbol{x}, \boldsymbol{W}) = \sigma_1\Bigg(\begin{bmatrix} w_{100} & w_{110} \end{bmatrix} \cdot \sigma_0\Bigg(\begin{bmatrix} w_{000} & w_{010} & w_{020} \\ w_{001} & w_{011} & w_{021} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}\Bigg)\Bigg)$$

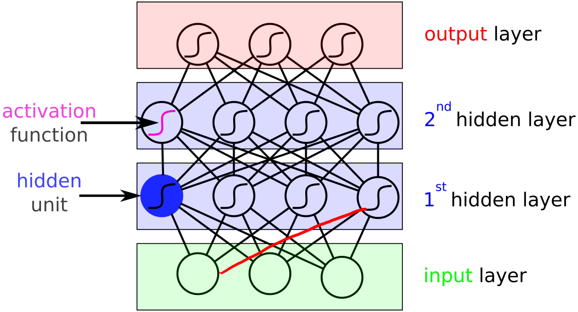
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\arg\min} \sum_{i=1}^{N} -\Big(y_i \log f(\boldsymbol{x}_i, \boldsymbol{W}) + (1 - y_i) \log\big[1 - f(\boldsymbol{x}_i, \boldsymbol{W})\big]\Big)$$



We get a simple Feed-Forward Neural Network (FFNN) with 1 hidden layer.
Note: σ_0 and σ_1 can be arbitrary activation functions.

Making the model more complicated

Each basis function can be a more complicated function of the feature vector $\boldsymbol{x}$ (a function of other basis functions rather than a function of $\boldsymbol{x}$).



output layer

2^{nd} hidden layer

1^{st} hidden layer

input layer

activation function

hidden unit

By adding more hidden layers we get a 'deep' neural network:

$$f(\boldsymbol{x}, \boldsymbol{W}) = \sigma_2\Big(\boldsymbol{W}_2^T \sigma_1\big(\boldsymbol{W}_1^\mathsf{T} \sigma_0(\boldsymbol{W}_0^T \boldsymbol{x})\big)\Big)$$

where $\boldsymbol{W} = \{\boldsymbol{W}_0, \boldsymbol{W}_1, \boldsymbol{W}_2\}$ are the weights to be learned. Above architecture is called a Multi-layered Perceptron (MLP) = fully-connected (feed-forward) Neural Network

Why use nonlinear activation functions?

$$\underbrace{w_1 \cdot w_2} \cdot x$$

$$w_3 \cdot x$$

Multiple linear layers:

$$\begin{aligned} f(\boldsymbol{x}, \boldsymbol{W}) &= \boldsymbol{W}_k \quad (\boldsymbol{W}_{k-1} \quad (\dots \quad (\boldsymbol{W}_0 \boldsymbol{x}) \dots)) \\ &= (\boldsymbol{W}_k \boldsymbol{W}_{k-1} \dots \boldsymbol{W}_1 \boldsymbol{W}_0) \boldsymbol{x} \\ &= \boldsymbol{W}' \boldsymbol{x} \end{aligned}$$

results in a linear transformation!

Multiple nonlinear layers:

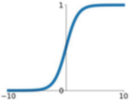
$$f(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W}_k \, \sigma_k(\boldsymbol{W}_{k-1} \, \sigma_{k-1}(\dots \sigma_0(\boldsymbol{W}_0 \boldsymbol{x}) \dots))$$

For non-linear functions we can not (in general) simplify it.

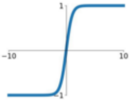
Activation functions

$\sigma\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}\sigma(1)\\\sigma(2)\end{bmatrix}$

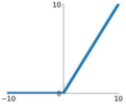
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



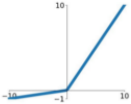
tanh
$\tanh(x)$



✓ **ReLU**
$\max(0, x)$

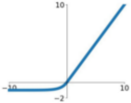


✗ **Leaky ReLU**
$\max(0.1x, x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

✗ **ELU**
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Most activiation functions are applied/operate *element-wise* when given a *multi-dimensional* input (e.g. all from above, except maxout).

Softmax activation is a notable exception!

Neural networks are universal approximators

Universal approximation theorem
An MLP with a linear output layer and one hidden layer can approximate any continuous function defined over a closed and bounded subset of $\mathbb{R}^D$, under mild assumptions on the activation function ('squashing' activation functions; e.g. sigmoid) and given the number of hidden units is large enough.
[Cybenko 1989; Funahashi 1989; Hornik et al 1989, 1991; Hartman et al 1990].

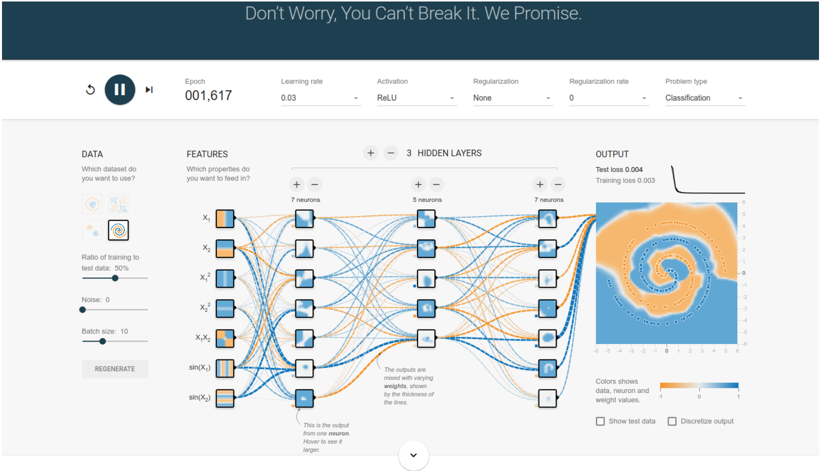
Also in the discrete case: a neural network can approximate any function from a discrete space to another.

Good news: Regardless of the function we want to learn, there exists an MLP that can represent that function.

Bad news: The learning algorithm is not guaranteed to find the true parameters.

- overfitting
- picking a wrong function with 'bad' training loss

Demo: http://playground.tensorflow.org



Don't Worry, You Can't Break It. We Promise.

Epoch 001,617

Learning rate 0.03

Activation ReLU

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

3 HIDDEN LAYERS

OUTPUT

Test loss 0.004

Training loss 0.003

Colors shows data, neuron and weight values.

☐ Show test data ☐ Discretize output

This is the output from one ***neuron****. Hover to see it larger.*

The outputs are mixed with varying ***weights****, shown by the thickness of the lines.*

Multiple hidden layers

According to the universal approximation theorem, a two-layer feed-forward network can represent any function.

Why do we add more layers?

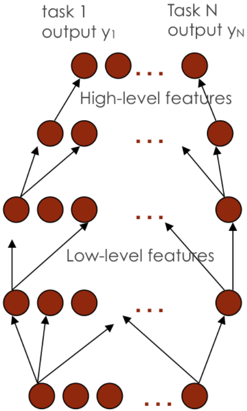
- The required hidden units might be exponential in the number of samples, in either discrete or continuous case.
- The issue of using the 2-layer network is easy to see in the discrete case. The number of possible functions $f : \{0, 1\}^D \rightarrow \{0, 1\}$ is 2^{2^D}. Representing all these functions requires $\mathcal{O}(2^D)$ degrees of freedom.
- For some families of functions, if we use *a few* layers we would need a large number of hidden units (and therefore parameters). But we can get the same representation power by adding *more layers*, fewer hidden units, and fewer parameters.

Multiple hidden layers

Functions that can be compactly represented with k layers may require exponentially many hidden units when using only $k-1$ layers.

Multiple levels of latent variables allow combinatorial sharing of statistical strength.

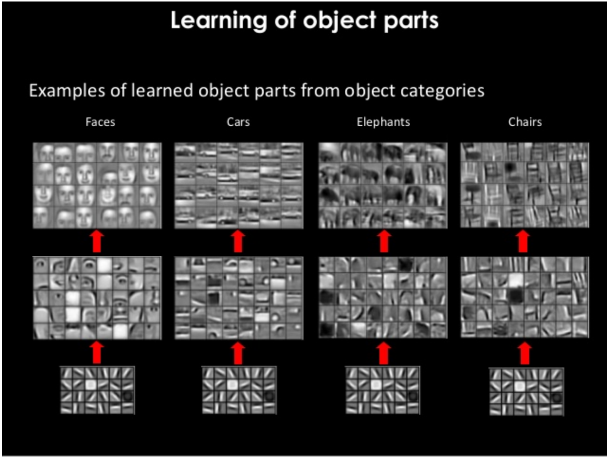
Different high-level features share lower-level features.



from: *Understanding and Improving Deep Learning Algorithms*, Yoshua Bengio, ML Google Distinguished Lecture, 2010

Multiple hidden layers

We learn "features" of "features". This allows for *better generalization*. (In contrast: A "wide" network tends to memorize data.)



Learning of object parts

Examples of learned object parts from object categories

Parameter Learning

Loss function

The choice of the loss function/cost function and the activation function of the *last* layer depend on the dataset being used and the distribution of the target variable.

Some common choices:

Output type	**Output distribution**	**Output Layer**	**Cost function**
Binary	Bernoulli	Sigmoid	Binary crossentropy
Discrete	Multinomial	Softmax	Crossentropy
Continuous	Gaussian	Linear	Gaussian crossentropy (Mean Squared Error)
Continuous	Arbitrary	GAN, VAE, ...	Various

Example 1: Binary classification

Given a set of labeled data $\{\boldsymbol{x}_i, y_i\}_{i=1}^{N}$, where $y_i \in \{0, 1\}$.
Usually one takes an NN with sigmoidal outputs

$$y = \sigma(a) = \frac{1}{1 + \exp(-a)} = f(\boldsymbol{x}, \boldsymbol{W})$$

In this case, we would use the binary cross-entropy between $f(\boldsymbol{x}_i, \boldsymbol{W})$ and y_i as the cost function

$$E(\boldsymbol{W}) = -\sum_{i=1}^{N} \Big(y_i \log f(\boldsymbol{x}_i, \boldsymbol{W}) + (1 - y_i) \log\big[1 - f(\boldsymbol{x}_i, \boldsymbol{W})\big] \Big)$$

$\min_{W} E(W)$

Example 2: Standard multi-class classification

Take a set of labeled data $\{\boldsymbol{x}_i, y_i\}_{i=1}^{N}$, where $y_i \in \{0,1\}^K$ (i.e. 1-of-K coding).
For this, we usually take an NN with Softmax outputs

$$y_k = \frac{exp(a_k)}{\sum_j exp(a_j)} = f_k(\boldsymbol{x}, \boldsymbol{W})$$

In this case, we would use cross-entropy between $f(\boldsymbol{x}_i, \boldsymbol{W})$ and y_i as the cost function

$$E(\boldsymbol{W}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} \Big(y_{nk} \log f_k(\boldsymbol{x}_n, \boldsymbol{W}) \Big)$$

Minimizing the cost function

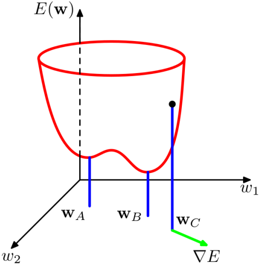
In practice $E(\boldsymbol{W})$ is often non-convex $\Longrightarrow$ optimization is tricky

- a local minimum is not necessarily a global minimum.
- potentially there exist several local minima, many of which can be equivalent (see next tutorial session)
- often it is not possible to find a global minimum nor is it useful.

We may find a few local minima, and pick the one with higher performance on a validation set.

Default approach:
find a local minimum by
using gradient descent

$$\boldsymbol{W}^{(t+1)} = \boldsymbol{W}^{(t)} - \alpha \nabla_{\boldsymbol{W}} E(\boldsymbol{W}^{(t)})$$



How can we compute the gradient?

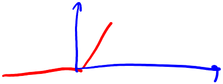
1) By hand: manually working out $\nabla_{\boldsymbol{W}} E$ and coding it is tricky and cumbersome (furthermore: see point 3).

2) Numeric: Can be done as

$$\frac{\partial E_n}{\partial w_{ij}} = \frac{E_n(w_{ij} + \epsilon) - E_n(w_{ij})}{\epsilon} + \mathcal{O}(\epsilon)$$

- Each evaluation of the above equation roughly requires $\mathcal{O}\big(|W|\big)$ operations, where $|W|$ is the dimensionality of weight space.
- The evaluation has to be done for each parameter independently. Therefore computing $\nabla_{\boldsymbol{W}} E$ requires $\mathcal{O}\big(|W|^2\big)$ operations!

How can we compute the gradient?



3) Symbolic differentiation: Automates essentially how you would compute the gradient function by hand.
But: Explicitly 'writing down' (and computing) the gradient function for every parameter is very expensive

- potentially exponentially many different cases (e.g. when having multiple layers with ReLUs)
- many terms reappear in the gradient computation for *different* parameters (since the function f is hierarchically constructed); these terms could be re-used to make computation faster; however symbolic differentiation does not exploit this insight

$$\frac{\delta f}{\delta w_{i,j,k}}(..) = \left\{ \vdots \right.$$

4) Automatic differentiation: e.g. **backpropagation** for neural networks

- Computes $(\nabla_{\boldsymbol{W}} E)$ automatically and efficiently
- Evaluation in $\mathcal{O}(|W|)$ (every 'neuron' is visited only twice)

Recap: Chain Rule

Recall: Chain rule of calculus

- for scalars $x, y, \cancel{n} \in \mathbb{R}$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- for vectors $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n, z \in \mathbb{R}$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

or with a more compact notation (using Jacobian matrix):

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

$$\text{where } \left[\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right]_{ij} = \frac{\partial y_i}{\partial x_i}$$

Helpful Concept: Computational Graph

$\sum_{i,k} y_{ik} \cdot \sigma(U^{(2)})_{ki}$

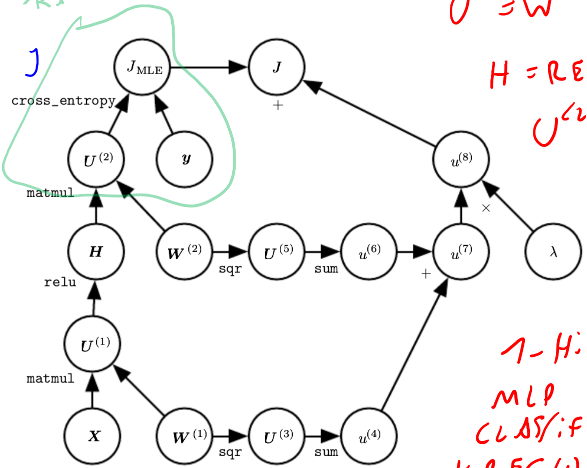
LOG(

J

$U^{(1)} = W^{(1)} \cdot X$

$H = \mathrm{RELU}(U^{(1)})$

$U^{(2)} = W^{(2)} \cdot H$



1-HIDDEN LAYER MLP FOR CLASSIFICATION + REGULARIZATION

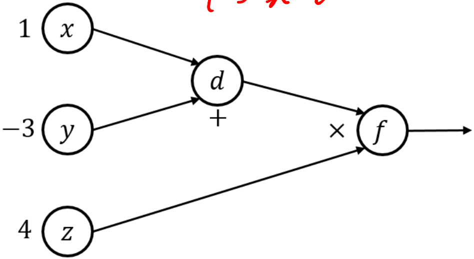
Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

Backpropagation: Toy Example

Example: $f = (x + y) \cdot z$. Find $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?

$d = x + y$

$f = d \cdot z$



Backpropagation: Toy Example

$d = x + y$

$f = d \cdot z$

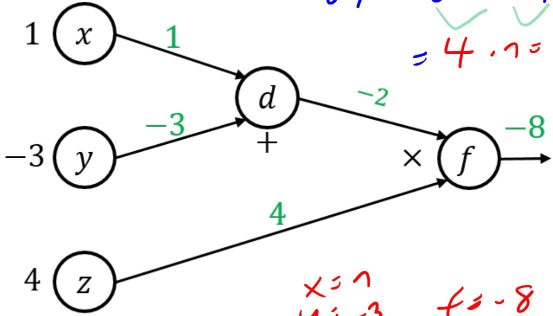
Example: $f = (x + y) \cdot z$. Find $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?

$$\frac{\delta f}{\delta y} = \underset{\checkmark}{\frac{\delta f}{\delta d}} \cdot \underset{\checkmark}{\frac{\delta d}{\delta y}}$$

$$= 4 \cdot 1 = 4$$

$$\frac{\delta f}{\delta z} = d = -2$$

$$\frac{\delta f}{\delta d} = z = 4$$



$$\frac{\delta f}{\delta x} = \underset{\checkmark}{\frac{\delta f}{\delta d}} \cdot \frac{\delta d}{\delta x}$$

$$4 \cdot 1$$

$$= 4$$

$x = 1$

$y = -3$

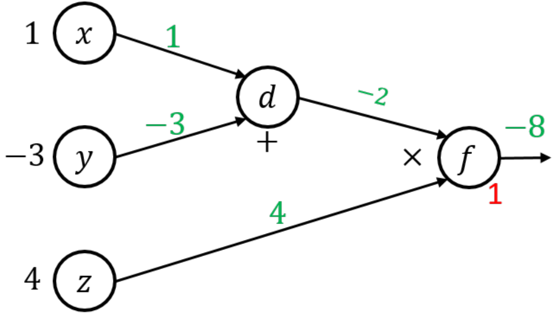
$z = 4$

$d = -2$

$f = -8$

Backpropagation: Toy Example

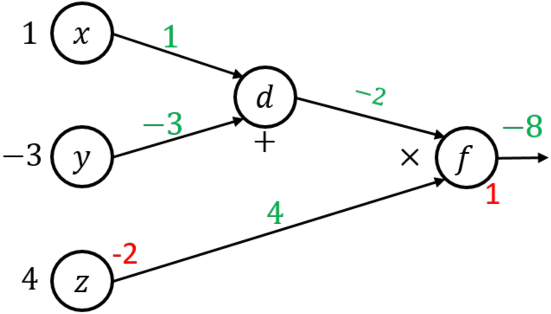
Example: $f = (x + y) \cdot z$. Find $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?



$\frac{\delta f}{\delta f} = 1$

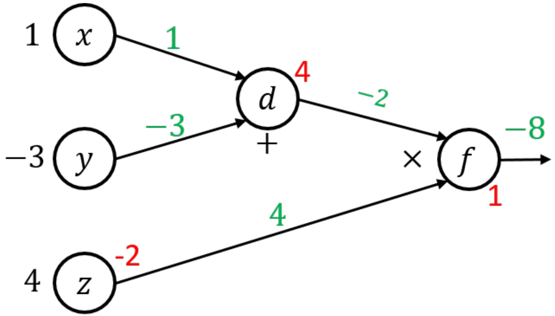
Backpropagation: Toy Example

Example: $f = (x + y) \cdot z$. Find $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?



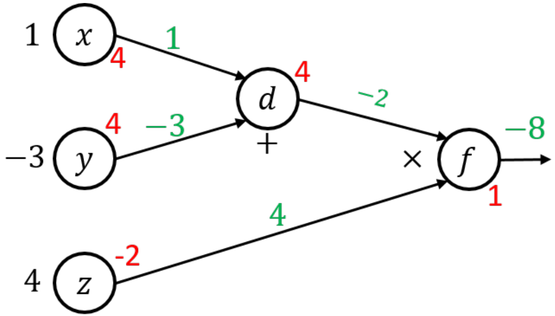
Backpropagation: Toy Example

Example: $f = (x + y) \cdot z$. Find $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?



Backpropagation: Toy Example

Example: $f = (x + y) \cdot z$. Find $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?



Backpropagation

In practice, for most of the cost functions we have that

$$E(\boldsymbol{W}) = \sum_{n=1}^{N} E_n(\boldsymbol{W})$$

The specific structure of the feed-forward neural networks allows us to efficiently compute $\nabla_{\boldsymbol{W}} E_n(\boldsymbol{W})$ for each instance in the training set.

For simplicity, let's consider only the element-wise activation functions (let's ignore softmax for now).

Notation:

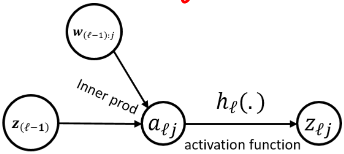
- $w_{\ell ij}$: a weight where ℓ=layer, i = input node and j = output node
- $z_{\ell i}^{(n)}$: the value of a neuron, ℓ=layer, i = node index, n = instance
- $a_{\ell i}^{(n)}$: the value of a logit, ℓ=layer and i = node index, n = instance; $\mathbf{a}_{\ell}^{(n)} = \boldsymbol{W}_{\ell-1} \cdot \mathbf{z}_{\ell-1}^{(n)}$
- $h_{\ell}(.)$: the activation function of the ℓ-th layer; $z_{\ell j}^{(n)} = h_{\ell}(a_{\ell j}^{(n)})$

Backpropagation

$a = z . w$

$\frac{\delta a}{\delta w} = z$

$$\frac{\partial E_n}{\partial w_{(\ell-1)ij}} = \underbrace{\boxed{\frac{\partial E_n}{\partial a_{\ell j}^{(n)}}}}_{\delta} \underbrace{\boxed{\frac{\partial a_{\ell j}^{(n)}}{\partial w_{(\ell-1)ij}}}}_{x}$$



- $\delta_{\ell j}^{(n)} \equiv \frac{\partial E_n}{\partial a_{\ell j}^{(n)}}$, is called the error of the j-th neuron at the ℓ-th layer

- The term $\frac{\partial a_{\ell j}^{(n)}}{\partial w_{(\ell-1)ij}}$ is simply equal to $z_{(\ell-1)i}^{(n)}$

Algorithm 1: Backpropagation

```
For each instance $\mathbf{x}_n$ in the training set:
    Forward pass: compute the values of $\mathbf{z}_1^{(n)}$, $\mathbf{z}_2^{(n)}$, ...., $\mathbf{z}_L^{(n)}$ (and $\mathbf{a}_l^{(n)}$)
    Compute the errors recursively as $\boldsymbol{\delta}_L^{(n)}$, $\boldsymbol{\delta}_{(L-1)}^{(n)}$, ... , $\boldsymbol{\delta}_2^{(n)}$
    Compute $\nabla_{\boldsymbol{W}} E_n$ using the above equation.
Compute the gradient as $\nabla_{\boldsymbol{W}} E = \sum_n (\nabla_{\boldsymbol{W}} E_n)$
```

Forward pass

The forward pass is trivial. Just evaluate the function.

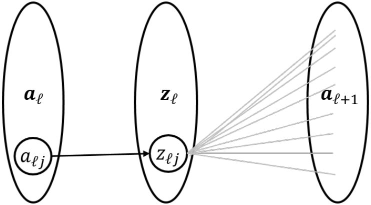
$$
\begin{aligned}
\mathbf{z}_0^{(n)} &= \mathbf{x}^{(n)} \\
\mathbf{a}_\ell^{(n)} &= \boldsymbol{W}_{\ell-1}\, \mathbf{z}_{\ell-1}^{(n)} \\
\mathbf{z}_\ell^{(n)} &= \mathbf{h}_\ell\Big(\mathbf{a}_\ell^{(n)}\Big)
\end{aligned}
$$

Backward pass

$z = h(a)$

Computing the error recursively:

$\delta_{\ell j}^{(n)}$ $\delta_{\ell+1, j}^{(n)}$



$$\frac{\partial E_n}{\partial a_{\ell j}^{(n)}} = \frac{\partial E_n}{\partial a_{(\ell+1)j}^{(n)}} \frac{\partial a_{(\ell+1)j}^{(n)}}{\partial z_{\ell j}^{(n)}} \frac{\partial z_{\ell j}^{(n)}}{\partial a_{\ell j}^{(n)}} \Rightarrow \delta_{\ell j}^{(n)} = \left(\boldsymbol{\delta}_{\ell+1}^{(n)}\right)^T \boldsymbol{W}_{\ell j:}\, h'_\ell(a_{\ell j}^{(n)})$$

Derivative of the activation function is usually known.
For example, for the sigmoid function: $\sigma'(a) = \sigma(a)\big(1 - \sigma(a)\big)$

Computing δ for the last layer

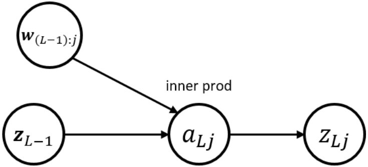
To make it simple, let's assume the following:

1. the last layer is linear, i.e., $h_L(a) = a$
2. we've used the mean squared error (we are solving, e.g., a regression problem)

$$\delta_{Lj}^{(n)} = \frac{\partial E_n}{\partial a_{Lj}^{(n)}} = \frac{\partial E_n}{\partial z_{Lj}^{(n)}} \frac{\partial z_{Lj}^{(n)}}{\partial a_{Lj}^{(n)}}$$

$$= \frac{\partial}{\partial z_{Lj}^{(n)}} \left(\frac{1}{2} \left(z_{Lj}^{(n)} - y_j^{(n)} \right)^2 \right) \times 1$$

$$= z_{Lj}^{(n)} - y_j^{(n)} \Rightarrow \text{error!}$$



We can follow a similar procedure in the case of logistic sigmoid activation function along with binary cross entropy, or softmax activation function along with cross entropy.

Backpropagation: Summary

$$\frac{\delta f}{\delta w_{ijk}}$$

- Allows to compute the gradient very efficiently
- Only two passes through the computational graph required (forward, backward)
- ! We only need to know the 'local' gradient function per node/activation function in the computational graph
  - above, we assumed to know $h'_\ell(a_{\ell j}^{(n)})$
  - as well as the gradient of the loss/cost function
- If you use any state-of-the-art deep learning framework, for every operation/activation/loss function, the corresponding 'local' gradient is implemented as well
  - given this, you can construct arbitrary complex functions and the gradient can be computed automatically

Overall, we can now easily apply our well-known gradient descent to learn the optimal parameters.

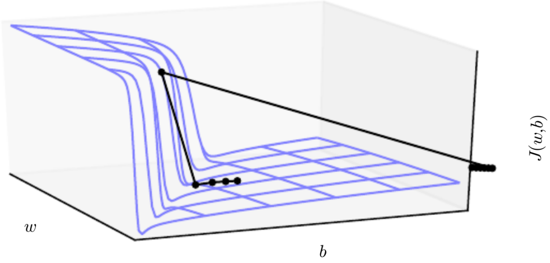
Will we reach a local minimum?

Cliffs: In some points, the norm of the gradient might become to too large.

- The next step will catapult the parameters very far.

To alleviate this issue, there is a simple yet effective trick: Gradient clipping

- We can clip the elements of the gradient to an interval $[-c, +c]$



source: Goodfellow

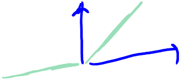
Will we reach a local minimum?

The cost function might be constant over wide flat regions of the weight space.

- Gradient descent actually fails, because it cannot find the steepest direction.
- Prevalent when using sigmoid activation function: its input values might be too large or too small ⇒ zero gradient for incoming weights (**vanishing gradient**)

ReLU activation function alleviates the above problem, because the gradient is always 1 at least on positive numbers.

Dead ReLU units: Assume that the input of a ReLU unit becomes negative for all data instances because of, e.g., a large negative bias. The gradient w.r.t. the incoming weights becomes zero ⇒ The unit will remain at this state forever.



Will we reach a local minimum?

$\cdots(w \cdot (\text{RELU}(w \cdot x)))$

Repetition of a parameter might also result in vanishing gradient problem (prevalent in RNNs; see later).
Consider the following simple case:

- The operation is multiplying the input vector by the matrix $\boldsymbol{W}$, t times

$w^t \cdot x$

- To put it simple:
  $\boldsymbol{W} = \mathbf{V}\ diag(\mathbf{D})\ \mathbf{V}^{-1} \Rightarrow \boldsymbol{W}^t = \mathbf{V}\big(diag(\mathbf{D})\big)^t\mathbf{V}^{-1}$.

In this simple case, the gradient w.r.t. the elements of $\mathbf{D}$ most probably would either vanish or explode:

- if $\mathbf{D}_{ii} < 1.0$, then $\mathbf{D}_{ii}^t$ would be near zero, the effect of the gradient would be faded.
- if $\mathbf{D}_{ii} > 1.0$, then $\mathbf{D}_{ii}^t$ would explode and makes the computations unstable.