Lecture 2: Learning with neural networks

Deep Learning @ UvA
Machine Learning Paradigm for Neural Networks
The Backpropagation algorithm for learning with a neural network
Neural Networks as modular architectures
Various Neural Network modules
How to implement and check your very own module
The Machine Learning Paradigm
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “forward propagation”
- Evaluate predictions

\[ h(x_i; \theta) \]

\[ \hat{y}_i \propto h(x_i; \theta) \]

\[ L(\theta; \hat{y}_i, h) = (y_i - \hat{y}_i)^2 \]
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “forward propagation”
- Evaluate predictions

\[ h(x_i; \theta) \]

\[ \hat{y}_i \propto h(x_i; \theta) \]

\[ L(\theta; \hat{y}_i, h) \]

\[ (y_i - \hat{y}_i)^2 \]
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “forward propagation”
- Evaluate predictions

Model

\[ h(x_i; \theta) \]

Score/Prediction/Output

\[ \hat{y}_i \propto h(x_i; \theta) \]

Objective/Loss/Cost/Energy

\[ L(\theta; \hat{y}_i, h) = (y_i - \hat{y}_i)^2 \]

Data

Input: X
Targets: Y
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “forward propagation”
- Evaluate predictions

\[ h(x_i; \theta) \]
\[ \hat{y}_i \propto h(x_i; \theta) \]
\[ L(\theta; \hat{y}_i, h) \]

\( Y \) targets
\( X \) input
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon "backpropagation"
- Evaluate predictions

\[ \mathcal{L}(\theta) = \frac{1}{2} \sum_{i} (y_i - \hat{y}_i)^2 \]
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “backpropagation”
- Evaluate predictions

\[
\begin{align*}
\text{Input: } X & \quad \text{Targets: } Y \\
\hat{Y}_i & = h(x_i; \theta) \\
\frac{\partial L(\theta; \hat{y}_i)}{\partial \hat{y}_i} & \\
\end{align*}
\]
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon "backpropagation"
- Evaluate predictions

\[
\begin{align*}
\text{Data} & \quad \text{Input: } X \\
\text{Targets: } Y & \quad \text{Model}
\end{align*}
\]

\[
\frac{\partial \hat{y}_i}{\partial h}
\]

\[
\frac{\partial L(\theta; \hat{y}_i)}{\partial \hat{y}_i}
\]
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon "backpropagation"
- Evaluate predictions

\[
\begin{align*}
\frac{\partial h(x_i)}{\partial \theta} & \quad \frac{\partial \hat{y}_i}{\partial h} \\
\frac{\partial \hat{y}_i}{\partial \theta} & \quad \frac{\partial L(\theta; \hat{y}_i)}{\partial \hat{y}_i}
\end{align*}
\]
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “backpropagation”
- Evaluate predictions

\[
\begin{align*}
\frac{\partial h(x_i)}{\partial \theta} & \quad \frac{\partial \hat{y}_i}{\partial h} \\
\frac{\partial \hat{y}_i}{\partial y_i} & \quad \frac{\partial \hat{y}_i}{\partial y_i} \\
\frac{\partial L(\theta; \hat{y}_i)}{\partial \hat{y}_i} & \quad \frac{\partial L(\theta; \hat{y}_i)}{\partial \hat{y}_i}
\end{align*}
\]
Optimization through Gradient Descent

- As with many model, we optimize our neural network with Gradient Descent
  \[ \theta(t+1) = \theta(t) - \eta_t \nabla_{\theta} \mathcal{L} \]

- The most important component in this formulation is the gradient

- Backpropagation to the rescue
  - The backward computations of network return the gradients
  - How to make the backward computations
What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.

- \[ a_L(x; \theta_1, \ldots, \theta_L) = h_L(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L) \]
  - \( x \) : input, \( \theta_l \) : parameters for layer \( l \), \( a_l = h_l(x, \theta_l) \) : (non-)linear function

- Given training corpus \( \{ X, Y \} \) find optimal parameters

\[
\theta^* \leftarrow \arg \min_\theta \sum_{(x,y) \subseteq (X,Y)} \ell(y, a_L(x; \theta_1, \ldots, \theta_L))
\]
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex

\[
\begin{align*}
&h_5(x_i; \theta) \quad h_4(x_i; \theta) \\
&h_4(x_i; \theta) \\
&h_3(x_i; \theta) \\
&h_2(x_i; \theta) \quad h_2(x_i; \theta) \\
&h_1(x_i; \theta) \\
&\text{Input}
\end{align*}
\]

Interweaved connections (Directed Acyclic Graph architecture - DAGNN)
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex

Loopy connections
(Recurrent architecture, special care needed)
A neural network model is a series of hierarchically connected functions.

This hierarchies can be very, very complex.
What is a module?

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters ($\theta$)
  - Receives as an argument an input $x$
  - And returns an output $a$ based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation $\rightarrow$ store module input $\rightarrow$
  - easy to get module output fast
  - easy to compute derivatives
A neural network is a composition of modules (building blocks)

- Any architecture works

- If the architecture is a feedforward cascade, no special care

- If acyclic, there is right order of computing the forward computations

- If there are loops, these form **recurrent** connections (revisited later)
Simply compute the activation of each module in the network

\[ a_l = h_l(x_l; \theta), \text{ where } a_l = x_{l+1} \text{ (or } x_l = a_{l-1}) \]

We need to know the precise function behind each module \( h_l(\ldots) \)

Recursive operations
- One module’s output is another’s input

Steps
- Visit modules one by one starting from the data input
- Some modules might have several inputs from multiple modules

Compute modules activations with the right order
- Make sure all the inputs computed at the right time
Simply compute the gradients of each module for our data
- We need to know the gradient formulation of each module
  \( \partial h_l(x_l; \theta_l) \) w.r.t. their inputs \( x_l \) and parameters \( \theta_l \)

We need the **forward computations first**
- Their result is the sum of losses for our input data

Then take the reverse network (reverse connections) and traverse it backwards

Instead of using the activation functions, we use their gradients

The whole process can be described very neatly and concisely with the **backpropagation algorithm**
Again, what is a neural network again?

- \( a_L(x; \theta_{1,...,L}) = h_L(h_{L-1}(...h_1(x, \theta_1), \theta_{L-1}), \theta_L) \)
  - \( x \): input, \( \theta_l \): parameters for layer \( l \), \( a_l = h_l(x, \theta_l) \): (non-)linear function

- Given training corpus \( \{X, Y\} \) find optimal parameters
  \[
  \theta^* \leftarrow \arg \min_{\theta} \sum_{(x, y) \in (X, Y)} \ell(y, a_L(x; \theta_{1,...,L}))
  \]

- To use any gradient descent based optimization (\( \theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\partial L}{\partial \theta^{(t)}} \)) we need the gradients

  \[
  \frac{\partial L}{\partial \theta_l}, l = 1, ..., L
  \]

- How to compute the gradients for such a complicated function enclosing other functions, like \( a_L(...) \)?
Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars $x, y, z$
  \[ \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \]
- When $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $z \in \mathbb{R}$
  \[ \frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow \text{gradients from all possible paths} \]
Chain rule

- Assume a nested function, \( z = f(y) \) and \( y = g(x) \)

- Chain Rule for scalars \( x, y, z \)
  \[
  \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}
  \]

- When \( x \in \mathbb{R}^m, y \in \mathbb{R}^n, z \in \mathbb{R} \)
  \[
  \frac{dz}{dx^i} = \sum_j \frac{dz}{dy^j} \frac{dy^j}{dx^i} \rightarrow \text{gradients from all possible paths}
  \]

\[
\frac{dz}{dx^1} = \frac{dz}{dy^1} \frac{dy^1}{dx^1} + \frac{dz}{dy^2} \frac{dy^2}{dx^1}
\]
Chain rule

- Assume a nested function, \( z = f(y) \) and \( y = g(x) \)
- Chain Rule for scalars \( x, y, z \)
  - \( \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \)
- When \( x \in \mathbb{R}^m, y \in \mathbb{R}^n, z \in \mathbb{R} \)
  - \( \frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \) → gradients from all possible paths

\[
\frac{dz}{dx^2} = \frac{dz}{dy^1} \frac{dy^1}{dx^2} + \frac{dz}{dy^2} \frac{dy^2}{dx^2}
\]
Chain rule

- Assume a nested function, \( z = f(y) \) and \( y = g(x) \)
- Chain Rule for scalars \( x, y, z \)
  - \[ \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \]
- When \( x \in \mathbb{R}^m, y \in \mathbb{R}^n, z \in \mathbb{R} \)
  - \[ \frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow \text{gradients from all possible paths} \]

\[
\frac{dz}{dx^3} = \frac{dz}{dy^1} \frac{dy^1}{dx^3} + \frac{dz}{dy^2} \frac{dy^2}{dx^3}
\]
Chain rule

- Assume a nested function, \( z = f(y) \) and \( y = g(x) \)

- Chain Rule for scalars \( x, y, z \)
  - \( \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \)

- When \( x \in \mathbb{R}^m, y \in \mathbb{R}^n, z \in \mathbb{R} \)
  - \( \frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow \) gradients from all possible paths
  - or in vector notation
    \[
    \frac{dz}{dx} = \left( \frac{dy}{dx} \right)^T \cdot \frac{dz}{dy}
    \]
  - \( \frac{dy}{dx} \) is the Jacobian
The Jacobian

- When $x \in \mathbb{R}^3, y \in \mathbb{R}^2$

$$J(y(x)) = \frac{dy}{dx} = \begin{bmatrix}
\frac{\partial y^{(1)}}{\partial x^{(1)}} & \frac{\partial y^{(1)}}{\partial x^{(2)}} & \frac{\partial y^{(1)}}{\partial x^{(3)}} \\
\frac{\partial y^{(2)}}{\partial x^{(1)}} & \frac{\partial y^{(2)}}{\partial x^{(2)}} & \frac{\partial y^{(2)}}{\partial x^{(3)}} \\
\frac{\partial y^{(1)}}{\partial x^{(1)}} & \frac{\partial y^{(1)}}{\partial x^{(2)}} & \frac{\partial y^{(1)}}{\partial x^{(3)}}
\end{bmatrix}$$
Chain rule in practice

- \( f(y) = \sin(y), y = g(x) = 0.5x^2 \)

\[
\frac{df}{dx} = \frac{d[\sin(y)]}{dg} \frac{d[0.5x^2]}{dx} = \cos(0.5x^2) \cdot x
\]
The loss function $\mathcal{L}(y, a_L)$ depends on $a_L$, which depends on $a_{L-1}$, ..., which depends on $a_2$: $a_L(x; \theta_1, \ldots, \theta_L) = h_L(h_{L-1}(\ldots h_1(x, \theta_1), \ldots, \theta_{L-1}), \theta_L)$

- Gradients of parameters of layer $l$ $\Rightarrow$ Chain rule

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial a_L} \cdot \frac{\partial a_L}{\partial a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial a_{L-2}} \cdot \ldots \cdot \frac{\partial a_l}{\partial \theta_l}$$

- When shortened, we need to two quantities

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \left( \frac{\partial a_l}{\partial \theta_l} \right)_T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$$

Gradient of a module w.r.t. its parameters

Gradient of loss w.r.t. the module output
For $\frac{\partial a_l}{\partial \theta_l}$ in $\frac{\partial L}{\partial \theta_l} = \left(\frac{\partial a_l}{\partial \theta_l}\right)^T \cdot \frac{\partial L}{\partial a_l}$ we only need the Jacobian of the $l$-th module output $a_l$ w.r.t. to the module’s parameters $\theta_l$

- Very local rule, every module looks for its own
- Since computations can be very local
  - graphs can be very complicated
  - modules can be complicated (as long as they are differentiable)
Backpropagation ⇔ Chain rule!!!

- For \( \frac{\partial L}{\partial a_l} \) in \( \frac{\partial L}{\partial \theta_l} = \left( \frac{\partial a_l}{\partial \theta_l} \right)^T \cdot \frac{\partial L}{\partial a_l} \) we apply chain rule again

\[
\frac{\partial L}{\partial a_l} = \left( \frac{\partial a_{l+1}}{\partial a_l} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}}
\]

- We can rewrite \( \frac{\partial a_{l+1}}{\partial a_l} \) as gradient of module w.r.t. to input

  - Remember, the output of a module is the input for the next one: \( a_l = x_{l+1} \)

Gradient w.r.t. the module input

\[
\frac{\partial L}{\partial a_l} = \left( \frac{\partial a_{l+1}}{\partial x_{l+1}} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}}
\]

Recursive rule (good for us)!!!
Multivariate functions $f(x)$

- Often module functions depend on multiple input variables
  - Softmax!
  - Each output dimension depends on multiple input dimensions

- For these cases for the $\frac{\partial a_l}{\partial x_l}$ (or $\frac{\partial a_l}{\partial \theta_l}$) we must compute Jacobian matrix as $a_l$ depends on multiple input $x_l$ (or $\theta_l$)
  - e.g. in softmax $a^2$ depends on all $e^{x_1}, e^{x_2}$ and $e^{x_3}$, not just on $e^{x_2}$

$$a^j = \frac{e^{x^j}}{e^{x_1} + e^{x_2} + e^{x_3}}, j = 1,2,3$$
Diagonal Jacobians

- Often in modules the output depends only in a single input
  - e.g. a sigmoid \( a = \sigma(x) \), or \( a = \tanh(x) \), or \( a = \exp(x) \)
    \[
    a(x) = \sigma(x) = \sigma \left( \begin{bmatrix} x^1 \\ x^2 \\ x^3 \end{bmatrix} \right) = \begin{bmatrix} \sigma(x^1) \\ \sigma(x^2) \\ \sigma(x^3) \end{bmatrix}
    \]

- Not need for full Jacobian, only the diagonal: anyways \( \frac{d \sigma_i}{dx_j} = 0, \forall i \neq j \)

\[
\frac{da}{dx} = \frac{d\sigma}{dx} = \begin{bmatrix}
\sigma(x^1)(1 - \sigma(x^1)) & 0 & 0 \\
0 & \sigma(x^2)(1 - \sigma(x^2)) & 0 \\
0 & 0 & \sigma(x^3)(1 - \sigma(x^3))
\end{bmatrix}
\sim \begin{bmatrix}
\sigma(x^1)(1 - \sigma(x^1)) \\
\sigma(x^2)(1 - \sigma(x^2)) \\
\sigma(x^3)(1 - \sigma(x^3))
\end{bmatrix}
\]

- Can rewrite equations as inner products to save computations
Dimension analysis

- To make sure everything is done correctly $\Rightarrow$ “Dimension analysis”
- The dimensions of the gradient w.r.t. $\theta_l$ must be equal to the dimensions of the respective weight $\theta_l$

$$\dim \left( \frac{\partial L}{\partial a_l} \right) = \dim(a_l)$$

$$\dim \left( \frac{\partial L}{\partial \theta_l} \right) = \dim(\theta_l)$$
Dimension analysis

- For $\frac{\partial L}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \frac{\partial L}{\partial a_{l+1}}$

\[
[d_l \times 1] = [d_{l+1} \times d_l]^T \cdot [d_{l+1} \times 1]
\]

- For $\frac{\partial L}{\partial \theta_l} = \frac{\partial \alpha_l}{\partial \theta_l} \cdot \left(\frac{\partial L}{\partial \alpha_l}\right)^T$

\[
[d_{l-1} \times d_l] = [d_{l-1} \times 1] \cdot [1 \times d_l]
\]

\[\begin{align*}
\text{dim}(a_l) &= d_l \\
\text{dim}(\theta_l) &= d_{l-1} \times d_l
\end{align*}\]
Backpropagation: Recursive chain rule

- **Step 1.** Compute forward propagations for all layers recursively
  \[ a_l = h_i(x_l) \text{ and } x_{l+1} = a_l \]

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations
    \[
    \frac{\partial L}{\partial a_l} = \left( \frac{\partial a_{l+1}}{\partial x_{l+1}} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}} \quad \text{and} \quad \frac{\partial L}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left( \frac{\partial L}{\partial a_l} \right)^T
    \]

- **Step 3.** Use the gradients \( \frac{\partial L}{\partial \theta_l} \) with Stochastic Gradient Descend to train
Backpropagation: Recursive chain rule

- **Step 1.** Compute forward propagations for all layers recursively
  \[ a_l = h_l(x_l) \text{ and } x_{l+1} = a_l \]

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

\[
\begin{align*}
\frac{\partial L}{\partial a_l} &= \left( \frac{\partial a_{l+1}}{\partial x_{l+1}} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}} \\
\frac{\partial L}{\partial \theta_l} &= \frac{\partial a_l}{\partial \theta_l} \cdot \left( \frac{\partial L}{\partial a_l} \right)^T
\end{align*}
\]

- **Step 3.** Use the gradients \( \frac{\partial L}{\partial \theta_l} \) with Stochastic Gradient Descend to train
  - Vector with dimensions \([d_{l-1} \times 1]\)
  - Vector with dimensions \([d_l \times 1]\)
  - Jacobian matrix with dimensions \([d_{l+1} \times d_l]^T\)
  - Matrix with dimensions \([d_{l-1} \times d_l]\)
  - Vector with dimensions \([1 \times d_l]\)
Dimensionality analysis: An Example

- \( d_{l-1} = 15 \) (15 neurons), \( d_{l} = 10 \) (10 neurons), \( d_{l+1} = 5 \) (5 neurons)
- Let’s say \( a_{l} = \theta_{l}^{T} x_{l} \) and \( a_{l+1} = \theta_{l+1}^{T} x_{l+1} \)
- Forward computations
  - \( a_{l-1} : [15 \times 1] \), \( a_{l} : [10 \times 1] \), \( a_{l+1} : [5 \times 1] \)
  - \( x_{l} : [15 \times 1] \), \( x_{l+1} : [10 \times 1] \)
  - \( \theta_{l} : [15 \times 10] \)
- Gradients
  - \( \frac{\partial L}{\partial a_{l}} : [5 \times 10]^{T} \cdot [5 \times 1] = [10 \times 1] \)
  - \( \frac{\partial L}{\partial \theta_{l}} : [15 \times 1] \cdot [10 \times 1]^{T} = [15 \times 10] \)
Intuitive Backpropagation
Backpropagation in practice

- Things are dead simple, just compute per module

\[
\frac{\partial a(x; \theta)}{\partial x} \quad \text{and} \quad \frac{\partial a(x; \theta)}{\partial \theta}
\]

- Then follow iterative procedure

\[
\frac{\partial L}{\partial a_l} = \left( \frac{\partial a_{l+1}}{\partial x_{l+1}} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}}
\]

\[
\frac{\partial L}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left( \frac{\partial L}{\partial a_l} \right)^T
\]
Backpropagation in practice

- Things are dead simple, just compute per module

\[
\frac{\partial a(x; \theta)}{\partial x} \quad \frac{\partial a(x; \theta)}{\partial \theta}
\]

- Then follow iterative procedure [remember: \( a_l = x_{l+1} \)]

\[
\frac{\partial L}{\partial a_l} = \left( \frac{\partial a_{l+1}}{\partial x_{l+1}} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}}
\]

\[
\frac{\partial L}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left( \frac{\partial L}{\partial a_l} \right)^T
\]
Forward propagation

- For instance, let’s consider our module is the function $\cos(\theta x) + \pi$
- The forward computation is simply

```python
import numpy as np
def forward(x):
    return np.cos(self.theta*x)+np.pi
```
The backpropagation for the function $\cos(x) + \pi$

```python
import numpy as np
def forward_dx(x):
    return -self.theta*np.sin(self.theta*x)

def backward_dx(x):
    return -self.theta*np.sin(self.theta*x)
```

```python
import numpy as np
def forward_dtheta(x):
    return -x*np.sin(self.theta*x)

def backward_dtheta(x):
    return -x*np.sin(self.theta*x)
```
Backpropagation: An example
\[ L = 3, a_3 = h_3(x_3) \]
\[ \theta_3 = \emptyset \]
\[ a_2 = h_2(x_2, \theta_2) \]
\[ \theta_2 \]
\[ a_1 = h_1(x_1, \theta_1) \]
\[ \theta_1 \]
\[ x_1 \]
\[ x_1^1 \; x_1^2 \; x_1^3 \; x_1^4 \]
Forward propagations

Compute and store $a_1 = h_1(x_1)$

$\ell_{a_3} = h_3(x_3)$

$\theta_3 = \emptyset$

$\ell_{a_2} = h_2(x_2, \theta_2)$

$\theta_2$

$\ell_{a_1} = h_1(x_1, \theta_1)$

$\theta_1$

$x_1$

Example

$a_1 = \sigma(\theta_1 x_1)$

Store!!!
Backpropagation visualization at epoch \((t)\)

**Forward propagations**

Compute and store \(a_2 = h_2(x_2)\)

\[
L = 3, a_3 = h_3(x_3) \\
\theta_3 = \emptyset \\
a_2 = h_2(x_2, \theta_2) \\
a_1 = h_1(x_1, \theta_1) \\
x_1
\]

Example

\[
a_1 = \sigma(\theta_1 x_1) \\
a_2 = \sigma(\theta_2 x_2)
\]

Store!!!
Backpropagation visualization at epoch \( (t) \)

**Forward propagations**

Compute and store \( a_3 = h_3(x_3) \)

\[
\begin{align*}
L &= 3, \\
a_3 &= h_3(x_3) \\
a_2 &= h_2(x_2, \theta_2) \\
a_1 &= h_1(x_1, \theta_1) \\
\theta_3 &= \emptyset \\
\theta_2 \\
\theta_1 \\
x_1 \\
\end{align*}
\]

Example

\[
\begin{align*}
a_1 &= \sigma(\theta_1 x_1) \\
a_2 &= \sigma(\theta_2 x_2) \\
a_3 &= \| y - x_3 \|^2
\end{align*}
\]

Store!!!
Backpropagation visualization at epoch \( (t) \)

**Backpropagation**

\[
\frac{\partial \mathcal{L}}{\partial a_3} = \ldots \leftarrow \text{Direct computation}
\]

\[
\frac{\partial \mathcal{L}}{\partial \theta_3} = \not\exists
\]

Example

\[
a_3 = \mathcal{L}(y, x_3) = h_3(x_3) = 0.5 \|y - x_3\|^2
\]
Backpropagation visualization at epoch \((t)\)

### Backpropagation

\[
\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2} \\
\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}
\]

**Example**

\[
\mathcal{L}(y, x_3) = 0.5 \|y - x_3\|^2 \\
x_3 = a_2 \\
a_2 = \sigma(\theta_2 x_2) \\
\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3) \\
\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) \\
\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2)) = x_2 a_2(1 - a_2) \\
\frac{\partial \mathcal{L}}{\partial a_2} = -(y - x_3) \\
\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2(1 - a_2)
\]
Backpropagation visualization at epoch \((t)\)

Backpropagation

\[
\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}
\]

\[
\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}
\]

\[
a_3 = h_3(x_3)
\]

\[
\theta_3 = \emptyset
\]

\[
a_2 = h_2(x_2, \theta_2)
\]

\[
\theta_2
\]

\[
a_1 = h_1(x_1, \theta_1)
\]

\[
\theta_1
\]

\[
x_1
\]

Example

\[
\mathcal{L}(y, a_3) = 0.5 \| y - a_3 \|^2
\]

\[
a_2 = \sigma(\theta_2 x_2)
\]

\[
x_2 = a_1
\]

\[
a_1 = \sigma(\theta_1 x_1)
\]

\[
\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)
\]

\[
\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)
\]

\[
\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)
\]

\[
\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)
\]

Computed from the exact previous backpropagation step (Remember, recursive rule)

\[
\mathcal{L}
\]

\[
\sum
\]

\[
\sum
\]

\[
\sum
\]

\[
\sum
\]
Backpropagation visualization at epoch \((t + 1)\)

Forward propagations

Compute and store \(a_1 = h_1(x_1)\)

\[
\begin{align*}
L &= 3, \ a_3 = h_3(x_3) \\
\theta_3 &= \emptyset \\
a_2 &= h_2(x_2, \theta_2) \\
\theta_2 &\quad \downarrow \\
a_1 &= h_1(x_1, \theta_1) \\
\theta_1 &\quad \downarrow \\
x_1 &\quad \downarrow
\end{align*}
\]

Example

\[a_1 = \sigma(\theta_1 x_1)\]

Store!!!
Backpropagation visualization at epoch \((t + 1)\)

**Forward propagations**

Compute and store \(a_2 = h_2(x_2)\)

\[
L = 3, a_3 = h_3(x_3) \\
\theta_3 = \emptyset \\
a_2 = h_2(x_2, \theta_2) \\
a_1 = h_1(x_1, \theta_1) \\
x_1
\]

**Example**

\[
a_1 = \sigma(\theta_1 x_1) \\
a_2 = \sigma(\theta_2 x_2)
\]

Store!!!
Backpropagation visualization at epoch \((t + 1)\)

Forward propagations

Compute and store \(a_3 = h_3(x_3)\)

\[
\begin{align*}
L & = 3, a_3 = h_3(x_3) \\
\theta_3 & = \emptyset \\
a_2 & = h_2(x_2, \theta_2) \\
a_1 & = h_1(x_1, \theta_1) \\
x_1 & = \sigma(\theta_1 x_1) \\
a_2 & = \sigma(\theta_2 x_2) \\
a_3 & = \|y - x_3\|^2
\end{align*}
\]
Backpropagation visualization at epoch \((t + 1)\)

\[
\begin{align*}
\frac{\partial L}{\partial a_3} &= \ldots \leftarrow \text{Direct computation} \\
\frac{\partial L}{\partial \theta_3} &= 0 \\
a_3 &= h_3(x_3) \\
\theta_3 &= \emptyset \\
a_2 &= h_2(x_2, \theta_2) \\
\theta_2 &= \emptyset \\
a_1 &= h_1(x_1, \theta_1) \\
\theta_1 &= \emptyset \\
x_1 &= \text{Input}
\end{align*}
\]

Example

\[
\frac{\partial L}{\partial x_3} = -(y - x_3)
\]

\[
a_3 = L(y, x_3) = h_3(x_3) = 0.5 \|y - x_3\|^2
\]
Backpropagation visualization at epoch \((t + 1)\)

\[
\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}
\]

\[
\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}
\]

\[
a_3 = h_3(x_3)
\]

\[
\theta_3 = \emptyset
\]

\[
a_2 = h_2(x_2, \theta_2)
\]

\[
\theta_2
\]

\[
a_1 = h_1(x_1, \theta_1)
\]

\[
\theta_1
\]

\[
x_1
\]

\[
\text{Example}
\]

\[
L(y, x_3) = 0.5 \|y - x_3\|^2
\]

\[
x_3 = a_2
\]

\[
a_2 = \sigma(\theta_2 x_2)
\]

\[
\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial x_3} = -(y - x_3)
\]

\[
\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))
\]

\[
\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2)) = x_2 a_2(1 - a_2)
\]

\[
\frac{\partial L}{\partial a_2} = -(y - x_3)
\]

\[
\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial a_2} x_2 a_2(1 - a_2)
\]

Stored during forward computations
Backpropagation visualization at epoch \((t + 1)\)

\[
\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}
\]

\[
\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}
\]

\[
a_3 = h_3(x_3)
\]

\[
a_2 = h_2(x_2, \theta_2)
\]

\[
a_1 = h_1(x_1, \theta_1)
\]

\[
x_1
\]

\[
\theta_3 = \emptyset
\]

\[
\theta_2
\]

\[
\theta_1
\]

\[
L(y, a_3) = 0.5 \| y - a_3 \|^2
\]

\[
a_2 = \sigma(\theta_2 x_2)
\]

\[
x_2 = a_1
\]

\[
a_1 = \sigma(\theta_1 x_1)
\]

\[
\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2(1 - a_2)
\]

\[
\frac{\partial a_1}{\partial \theta_1} = x_1 a_1(1 - a_1)
\]

\[
\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial a_2} \cdot \theta_2 a_2(1 - a_2)
\]

\[
\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial a_1} \cdot x_1 a_1(1 - a_1)
\]

Example

Computed from the exact previous backpropagation step (Remember, recursive rule)
Some practical tricks of the trade

- For classification use cross-entropy loss
- Use Stochastic Gradient Descent on mini-batches
- Shuffle training examples at each new epoch
- Normalize input variables
  - $(\mu, \sigma^2) = (0, 1)$
  - $\mu = 0$
Everything is a module
Neural network models

- A neural network model is a series of hierarchically connected functions
- These hierarchies can be very, very complex
Linear module

- Activation function $a = \theta x$
- Gradient with respect to the input $\frac{\partial a}{\partial x} = \theta$
- Gradient with respect to the parameters $\frac{\partial a}{\partial \theta} = x$
Sigmoid module

- Activation function $a = \sigma(x) = \frac{1}{1+e^{-x}}$

- Gradient wrt the input $\frac{\partial a}{\partial x} = \sigma(x)(1 - \sigma(x))$

- Gradient wrt the input $\frac{\partial \sigma(\theta x)}{\partial x} = \theta \cdot \sigma(\theta x)(1 - \sigma(\theta x))$

- Gradient wrt the parameters $\frac{\partial \sigma(\theta x)}{\partial \theta} = x \cdot \sigma(\theta x)(1 - \sigma(\theta x))$
Sigmoid module – Pros and Cons

+ Output can be interpreted as probability
+ Output bounded in \([0, 1]\) \(\rightarrow\) network cannot overshoot
- Always multiply with \(< 1\) \(\rightarrow\) Gradients can be small in deep networks
- The gradients at the tails flat to 0 \(\rightarrow\) no serious SGD updates
  - Overconfident, but not necessarily “correct”
  - Neurons get stuck
Tanh module

- Activation function $a = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

- Gradient with respect to the input $\frac{\partial a}{\partial x} = 1 - tanh^2(x)$

- Similar to sigmoid, but with different output range
  - $[-1, +1]$ instead of $[0, +1]$
  - Stronger gradients, because data is centered around 0 (not 0.5)
  - Less bias to hidden layer neurons as now outputs can be both positive and negative (more likely to have zero mean in the end)
Rectified Linear Unit (ReLU) module (Alexnet)

- Activation function \( a = h(x) = \max(0, x) \)
- Gradient wrt the input \( \frac{\partial a}{\partial x} = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases} \)
- Very popular in computer vision and speech recognition
- Much faster computations, gradients
  - No vanishing or exploding problems, only comparison, addition, multiplication
- People claim biological plausibility
- Sparse activations
- No saturation
- Non-symmetric
- Non-differentiable at 0
- A large gradient during training can cause a neuron to “die”. Higher learning rates mitigate the problem
ReLU convergence rate

![Graph showing the convergence rate of ReLU and Tanh functions.](image)
Other ReLUs

- Soft approximation (softplus): \( a = h(x) = \ln(1 + e^x) \)
- Noisy ReLU: \( a = h(x) = \max(0, x + \varepsilon), \varepsilon \sim \mathcal{N}(0, \sigma(x)) \)
- Leaky ReLU: \( a = h(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \)
- Parametric ReLu: \( a = h(x) = \begin{cases} x, & \text{if } x > 0 \\ \beta x & \text{otherwise} \end{cases} \) (parameter \( \beta \) is trainable)
Softmax module

- Activation function \( a^{(k)} = \text{softmax}(x^{(k)}) = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}} \)
  - Outputs probability distribution, \( \sum_{k=1}^K a^{(k)} = 1 \) for \( K \) classes
- Because \( e^{a+b} = e^a e^b \), we usually compute
  \[
  a^{(k)} = \frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}}, \quad \mu = \max_k x^{(k)}
  \]
  hence
  \[
  \frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}} = \frac{e^{\mu}e^{x^{(k)}}}{e^{\mu} \sum_j e^{x^{(j)}}} = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}
  \]
- Avoid exponentiating large numbers \( \rightarrow \) better stability
Euclidean loss module

- Activation function $a(x) = 0.5 \|y - x\|^2$
  - Mostly used to measure the loss in regression tasks

- Gradient with respect to the input $\frac{\partial a}{\partial x} = x - y$
Cross-entropy loss (log-loss or log-likelihood) module

- Activation function \( a(x) = - \sum_{k=1}^{K} y^{(k)} \log x^{(k)}, \ y^{(k)} = \{0, 1\} \)

- Gradient with respect to the input \( \frac{\partial a}{\partial x^{(k)}} = - \frac{1}{x^{(k)}} \)

- The cross-entropy loss is the most popular classification losses for classifiers that output probabilities (not SVM)

- Cross-entropy loss couples well softmax/sigmoid module
  - Often the modules are combined and joint gradients are computed

- Generalization of logistic regression for more than 2 outputs
Many, many more modules out there ...

- Regularization modules
  - Dropout

- Normalization modules
  - $\ell_2$-normalization, $\ell_1$-normalization

**Question:** When is a normalization module needed?

**Answer:** Possibly when combining different modalities/networks (e.g. in Siamese or multiple-branch networks)

- Loss modules
  - Hinge loss

- and others, which we are going to discuss later in the course
Composite modules

or ...

“Make your own module”
Backpropagation again

- **Step 1.** Compute forward propagations for all layers recursively

  \[ a_l = h_l(x_l) \text{ and } x_{l+1} = a_l \]

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

\[
\frac{\partial L}{\partial a_l} = \left( \frac{\partial a_{l+1}}{\partial x_{l+1}} \right)^T \cdot \frac{\partial L}{\partial a_{l+1}} \quad \frac{\partial L}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left( \frac{\partial L}{\partial a_l} \right)^T
\]

- **Step 3.** Use the gradients \( \frac{\partial L}{\partial \theta_l} \) with Stochastic Gradient Descent to train
New modules

- Everything can be a module, given some ground rules
- How to make our own module?
  - Write a function that follows the ground rules
- Needs to be (at least) first-order differentiable (almost) everywhere
- Hence, we need to be able to compute the
  \[ \frac{\partial a(x;\theta)}{\partial x} \text{ and } \frac{\partial a(x;\theta)}{\partial \theta} \]
A module of modules

- As everything can be a module, a module of modules could also be a module
- We can therefore make new building blocks as we please, if we expect them to be used frequently
- Of course, the same rules for the eligibility of modules still apply
1 sigmoid == 2 modules?

- Assume the sigmoid $\sigma(...)$ operating on top of $\theta x$
  
  $a = \sigma(\theta x)$

- Directly computing it $\Rightarrow$ complicated backpropagation equations

- Since everything is a module, we can decompose this to 2 modules

$$a_1 = \theta x \rightarrow a_2 = \sigma(a_1)$$
1 sigmoid == 2 modules?

- Two backpropagation steps instead of one

+ But now our gradients are simpler
  ◦ Algorithmic way of computing gradients
  ◦ We avoid taking more gradients than needed in a (complex) non-linearity

\[
a_1 = \theta x \quad \rightarrow \quad a_2 = \sigma(a_1)
\]
Network-in-network [Lin et al., arXiv 2013]

Figure 1: Comparison of linear convolution layer and mlpconv layer. The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (we choose the multilayer perceptron in this paper). Both layers map the local receptive field to a confidence value of the latent concept.
ResNet [He et al., CVPR 2016]

Figure 2. Residual learning: a building block.
Radial Basis Function (RBF) Network module

- RBF module
  \[ a = \sum_j u_j \exp(-\beta_j (x - w_j)^2) \]

- Decompose into cascade of modules
  \[ a_1 = (x - w)^2 \]
  \[ a_2 = \exp(-\beta a_1) \]
  \[ a_3 = u a_2 \]
  \[ a_4 = \text{plus}(..., a_3^{(j)}, ...) \]
Radial Basis Function (RBF) Network module

- An RBF module is good for regression problems, in which cases it is followed by a Euclidean loss module.
- The Gaussian centers $w_j$ can be initialized externally, e.g. with k-means.

\[
\begin{align*}
  a_1 &= (x - w)^2 \\
  a_2 &= \exp(-\beta a_1) \\
  a_3 &= ua_2 \\
  a_4 &= plus(..., a_3^{(j)}, ...) \\
\end{align*}
\]
An RBF visually

\[ a_1 = (x - w)^2 \rightarrow a_2 = \exp(-\beta a_1) \rightarrow a_3 = ua_2 \rightarrow a_4 = \text{plus}(..., a_3^{(j)},...) \]

\[ a_5 = ||y - a_4||^2 \]
Unit tests
Unit test

○ Always check your implementations
  ◦ Not only for Deep Learning

○ Does my implementation of the $\sin$ function return the correct values?
  ◦ If I execute $\sin(\pi/2)$ does it return 1 as it should

○ Even more important for gradient functions
  ◦ not only our implementation can be wrong, but also our math

○ Slightest sign of malfunction $\Rightarrow$ ALWAYS RECHECK
  ◦ Ignoring problems never solved problems
Gradient check

- Most dangerous part for new modules $\rightarrow$ get gradients wrong
- Compute gradient analytically
- Compute gradient computationally
- Compare

$$\Delta(\theta^{(i)}) = \left\| \frac{\partial a(x; \theta^{(i)})}{\partial \theta^{(i)}} - g(\theta^{(i)}) \right\|^2$$

- Is difference in $(10^{-4}, 10^{-7})$ $\rightarrow$ then gradients are good

Original gradient definition: $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
Gradient check

- Perturb one parameter $\theta^{(i)}$ at a time with $\theta^{(i)} + \varepsilon$
- Then check $\Delta(\theta^{(i)})$ for that one parameter only
- **Do not** perturb the whole parameter vector $\theta + \varepsilon$
  - This will give **wrong results** (simple geometry)
- Sample dimensions of the gradient vector
  - If you get a few dimensions of an gradient vector good, all is good
  - Sample function and bias gradients equally, otherwise you might get your bias wrong
Numerical gradients

- Can we replace analytical gradients with numerical gradients?
  - In theory, yes!
  - In practice, no!
    - Too slow
Be creative!

- What about trigonometric modules?
- Or polynomial modules?
- Or new loss modules?
Summary

- Machine learning paradigm for neural networks
- Backpropagation algorithm, backbone for training neural networks
- Neural network == modular architecture
- Visited different modules, saw how to implement and check them
Reading material & references

- [http://www.deeplearningbook.org/](http://www.deeplearningbook.org/)
  - Part I: Chapter 2-5
  - Part II: Chapter 6
Next lecture

- Optimizing deep networks
- Which loss functions per machine learning task
- Advanced modules
- Deep Learning theory