Statistical Machine Learning (BE4M33SSU)  
Lecture 6: Artificial Neural Networks

Jan Drchal

Czech Technical University in Prague  
Faculty of Electrical Engineering  
Department of Computer Science
Outline

Topics covered in the lecture:

- Neuron types
- Layers
- Loss functions
- Computing loss gradients via backpropagation
- Gradient Descent overview
- Parameter initialization
- Regularization
Neural Networks Overview

- Training examples: \( \mathcal{T}^m = \{(x_i, y_i) \in (\mathcal{X} \times \mathcal{Y}) \mid i = 1, \ldots, m\} \), where \( \mathcal{X} \subseteq \mathbb{R}^n \) and \( \mathcal{Y} \subseteq \mathbb{R}^K \)

- Neural network is a composition of simple linear or non-linear functions (neurons) parametrized by weights and biases

- Here we consider \( \mathcal{H} \) a hypothesis space of neural networks having a fixed architecture

- Learning methods are based on Empirical Risk Minimization:

\[
R_{\mathcal{T}^m}(h_\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell(y_i, h_\theta(x_i)),
\]

where \( h_\theta \in \mathcal{H} \) denotes a neural network parametrized by \( \theta \)
McCulloch-Pitts Perceptron

\[ x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n \]  
input (feature vector)

\[ w = (w_1, w_2, \ldots, w_n)^T \in \mathbb{R}^n \]  
weights

\[ b \in \mathbb{R} \]  
bias (threshold)

\[ s = \langle w, x \rangle + b \in \mathbb{R} \]  
inner potential

\[ f(s) = \begin{cases} 
-1 & \text{if } s < 0 \\
1 & \text{else}
\end{cases} \]  
activation function

\[ \hat{y} = h_{(w,b)}(x) \in \{-1, 1\} \]  
output (activity)

\[ \hat{y} = f(s) = f \left( \sum_{i=1}^{n} w_i x_i + b \right) = f (\langle w, x \rangle + b) \]

- It is the linear classifier we have already seen.
Treat bias as an extra fixed input $x_0 = 1$ weighted $w_0 = b$:

$$
\hat{y} = f (\langle \mathbf{w}, \mathbf{x} \rangle + b) = f (\langle \mathbf{w}, \mathbf{x} \rangle + w_0 \cdot 1) = f (\langle \mathbf{w}', \mathbf{x}' \rangle)
$$

- $\mathbf{x}' = (1, x_1, \ldots, x_n)^T \in \mathbb{R}^{n+1}$
- $\mathbf{w}' = (w_0, w_1, \ldots, w_n)^T \in \mathbb{R}^{n+1}$
- Unless otherwise noted we will use $\mathbf{x}, \mathbf{w}$ instead of $\mathbf{x}', \mathbf{w}'$
### Activation Functions

- **Step Function**
- **Bipolar Step Function**
- **Linear**
- **Sigmoid**
- **Hyperbolic Tangent**
- **ReLU**

- **Logistic sigmoid:** \( \sigma(s) \triangleq \frac{1}{1 + e^{-s}} = \frac{e^s}{e^s + 1} \)

- **Note:** \( \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2 \sigma(s) - 1 \)
Linear Neuron

- Training examples: $\mathcal{T}^m = \{(x_i, y_i) \in (\mathbb{R}^{n+1} \times \mathbb{R}) \mid i = 1, \ldots, m\}$

- Single neuron with linear activation function $\equiv$ linear regression:

$$\hat{y} = s = \langle x, w \rangle, \quad \hat{y} \in \mathbb{R}$$

- Inputs: $X = \begin{pmatrix} 1 & x_{11} & \ldots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \ldots & x_{mn} \end{pmatrix} = \begin{pmatrix} x_1^T \\ \vdots \\ x_m^T \end{pmatrix}$

- Targets: $y = (y_1, \ldots, y_m)^T, \quad y_i \in \mathbb{R}$

- Outputs: $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_m)^T, \quad \hat{y}_i \in \mathbb{R}$

- For the whole dataset we get:

$$\hat{y} = Xw, \quad \hat{y} \in \mathbb{R}^m$$
Linear Neuron: Maximum Likelihood Estimation

- Assumption: data are Gaussian distributed with mean \( \langle x_i, w \rangle \) and variance \( \sigma^2 \):

\[
y_i \sim \mathcal{N} ( \langle x_i, w \rangle, \sigma^2 ) = \langle x_i, w \rangle + \mathcal{N} (0, \sigma^2)
\]

- Likelihood for i.i.d. data:

\[
p ( y | w, X, \sigma ) = \prod_{i=1}^{m} p ( y_i | w, x_i, \sigma ) = \prod_{i=1}^{m} (2\pi\sigma^2)^{-\frac{1}{2}} e^{-\frac{1}{2\sigma^2} (y_i - \langle w, x_i \rangle)^2} = (2\pi\sigma^2)^{-\frac{m}{2}} e^{-\frac{1}{2\sigma^2} \sum_{i=1}^{m} (y_i - \langle w, x_i \rangle)^2} = (2\pi\sigma^2)^{-\frac{m}{2}} e^{-\frac{1}{2\sigma^2} (y - Xw)^T (y - Xw)}
\]

- Negative Log Likelihood (switching to minimization):

\[
\mathcal{L} (w) = \frac{m}{2} \log (2\pi\sigma^2) + \frac{1}{2\sigma^2} (y - Xw)^T (y - Xw)
\]
Linear Neuron: Maximum Likelihood Estimation (contd.)

- Note that

$$\sum_{i=1}^{m} \ell(y_i, \hat{y}_i) = (y - Xw)^T (y - Xw)$$

is the **sum-of-squares** or **squared error** (SE)

- Minimization of $\mathcal{L}(w) \equiv$ least squares estimation

  Solving $\frac{\partial \mathcal{L}}{\partial w} = 0$ we get $w^* = (X^TX)^{-1} X^T y$ (see seminar)

$$0.8x + 2 + \mathcal{N}(0, 1)$$
Logistic Sigmoid and Probability

- Denote: $\hat{y} = \sigma(s)$, $\hat{y} \in (0, 1)$

- Sigmoid output can represent a parameter of the Bernoulli distribution:

$$p(y|\hat{y}) = \text{Ber}(y|\hat{y}) = \hat{y}^y (1 - \hat{y})^{1-y} = \begin{cases} 
\hat{y} & \text{for } y = 1 \\
1 - \hat{y} & \text{for } y = 0 
\end{cases}$$

- Models confidence of the positive class $y = 1$

- Binary classifier:

$$h(\hat{y}) = \begin{cases} 
1 & \text{if } \hat{y} > \frac{1}{2} \\
0 & \text{else} 
\end{cases}$$
Logistic Regression

- MCP neuron using sigmoid activation function $\equiv$ **logistic regression**:
  \[ \hat{y} = \sigma(\langle w, x \rangle), \hat{y} \in (0, 1) \]

- **Inputs**: \( \mathbf{X} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \ldots & x_{mn} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_m^T \end{pmatrix} \)

- **Target class**: \( \mathbf{y} = (y_1, \ldots, y_m)^T, y_i \in \{0, 1\} \)

- **Output class**: \( \hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_m)^T, \hat{y}_i \in (0, 1) \)

- Note that the logistic regression (including the decision rule) solves actually a classification task
Logistic Regression MLE Leads to the Cross-Entropy

- Likelihood, for the logistic regression:

\[
p(y|w, X) = \prod_{i=1}^{m} \text{Ber}(y_i|\hat{y}_i) = \prod_{i=1}^{m} \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}
\]

- Negative Log Likelihood:

\[
\mathcal{L}(w) = \sum_{i=1}^{m} - [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)]
\]

- This loss function is called the cross-entropy

- The \( \ell(y_i, \hat{y}_i) \) is the negative log probability of the correct answer \( y_i \in \{0, 1\} \) given by the model output \( \hat{y}_i \in (0, 1) \)
Maximum Likelihood Estimation

- Maximum Likelihood Estimation: $w^* = \arg\min_w L(w)$

- Derivative of the loss w.r.t. to the sigmoid argument:

$$\frac{\partial L}{\partial s_i} = \hat{y}_i - y_i \quad \text{(see seminar)}$$

- Gradient w.r.t. logistic regression parameters:

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{m} \frac{\partial L}{\partial s_i} \cdot \frac{\partial s_i}{\partial w} = \sum_{i=1}^{m} x_i (\hat{y}_i - y_i) = X^T(\hat{y} - y)$$

- $\frac{\partial L}{\partial w} = 0$ has no analytical solution $\implies$ use numerical methods
**Linear Layer**

- Multiple linear neurons: linear (dense, fully-connected, affine) layer
- Output \( k \): \( \hat{y}_k = \langle x, w_k \rangle, \ k = 1, 2, \ldots, K \)
- All outputs using **weight matrix** \( \mathbf{W} \): \( \hat{y} = x^T \mathbf{W} \)
- Multiple samples: \( \hat{\mathbf{Y}} = \mathbf{X} \mathbf{W} \)

\[
\mathbf{W} = \begin{pmatrix} w_1 \ldots w_K \end{pmatrix} = \begin{pmatrix} w_{01} & \cdots & w_{0K} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nK} \end{pmatrix}
\]

\[
\mathbf{X} = \begin{pmatrix} x_1^T \\ \vdots \\ x_m^T \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \cdots & x_{mn} \end{pmatrix}
\]

\[
\hat{\mathbf{Y}} = \begin{pmatrix} \hat{y}_1^T \\ \vdots \\ \hat{y}_m^T \end{pmatrix} = \begin{pmatrix} \hat{y}_{11} & \cdots & \hat{y}_{1K} \\ \vdots & \ddots & \vdots \\ \hat{y}_{m1} & \cdots & \hat{y}_{mK} \end{pmatrix}
\]
Multinomial classification, $K$ mutually exclusive classes

Definition: $\sigma_k(s) \triangleq \frac{e^{s_k}}{\sum_{c=1}^{K} e^{s_c}}$, where $K$ is the number of classes

Softmax represents a categorical probability distribution: $\sigma_k \in (0, 1)$ for $k \in \{1 \ldots K\}$ and $\sum_{k=1}^{K} \sigma_k = 1$

Describes class membership probabilities: $p(y = k|s) = \sigma_k(s)$
Softmax Layer MLE

- **Target:** \( \mathbf{y} = (y_1 \ldots y_m)^T, y_i \in \{1, 2, \ldots, K\} \)

- **One-hot encoding for sample \( i \) and class \( k \):** let \( y_{ik} \triangleq [y_i = k] \)

- **Likelihood:**
  \[
  p(\mathbf{y}|\mathbf{w}, \mathbf{X}) = \prod_{i=1}^{m} \prod_{c=1}^{K} \hat{y}_{ic}^{y_{ic}}
  \]

- **Negative Log Likelihood:**
  \[
  \mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{m} \sum_{c=1}^{K} y_{ic} \log(\hat{y}_{ic})
  \]

  Again the **cross-entropy**

- **See seminar for the gradient**
Multinominal Logistic Regression

- \( \text{linear layer} + \text{softmax layer} = \text{multinominal logistic regression:} \)

\[
\hat{y}_k = \sigma_k(x^T W)
\]

- Classifier: \( h(x, W) = \arg\max_k \hat{y}_k \)
## Loss Functions: Summary

<table>
<thead>
<tr>
<th>problem</th>
<th>output</th>
<th>suggested loss function</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary classification</td>
<td>sigmoid neuron</td>
<td>cross-entropy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-\frac{1}{m} \sum_{i=1}^{m} [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)])</td>
</tr>
<tr>
<td>multinominal classification</td>
<td>softmax</td>
<td>multinominal cross-entropy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{K} y_{ic} \log(\hat{y}_{ic}))</td>
</tr>
<tr>
<td>regression</td>
<td>linear neuron</td>
<td>mean squared error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2)</td>
</tr>
<tr>
<td>multi-output regression</td>
<td>linear layer</td>
<td>mean squared error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{K} (y_{ic} - \hat{y}_{ic})^2)</td>
</tr>
</tbody>
</table>

*These exactly correspond to the empirical risk* \(R_{\mathcal{T}^m}(h)\)
- Feed-forward ANN
- Fully-connected layers
- MLP for regression would typically use linear output layer
Recurrent Neural Network (RNN)

- Fully-Connected Recurrent Neural Network (FRNN)
- Both inputs and outputs are sequences
- Feedback connections $\rightarrow$ memory (similarly to sequential circuitry)
Modular and Hierarchical Architectures

- Directed Acyclic Graphs (DAGs)
- Layers can be organized in *modules*
- Hierarchies of modules, module reuse
Backpropagation Overview

- A method to compute a gradient of the *loss function* with respect to its parameters: $\nabla \mathcal{L}(\mathbf{w})$

- $\nabla \mathcal{L}(\mathbf{w})$ is in turn used by optimization methods like gradient descent

- Here, we present the "modular" backpropagation (see Nando de Freitas’ Machine Learning course: [https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/](https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/))

- Let us use multinominal logistic regression as an example
Backpropagation: the Loss Function

- The loss function is the multinominal cross-entropy in this case:

$$L(w) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{K} [y_i = c] \log \left( \frac{\exp (\langle x_i, w_c \rangle)}{\sum_{k=1}^{K} \exp (\langle x_i, w_k \rangle)} \right)$$
Backpropagation Based on Modules

- Computation of $\nabla \mathcal{L}(w)$ involves repetitive use of the chain rule
- We can make things simpler by divide and conquer approach
- Divide to simplest possible modules (these can be later combined into complex networks)
- Represent even the loss function as a module
- Passing messages
Backpropagation: Backward Pass Message

Let $\delta^l = \frac{\partial L}{\partial z^l}$ be the sensitivity of the loss to the module input for layer $l$, then:

$$
\delta^l_i = \frac{\partial L}{\partial z^l_i} = \sum_j \frac{\partial L}{\partial z^{l+1}_j} \cdot \frac{\partial z^{l+1}_j}{\partial z^l_i} = \sum_j \delta^{l+1}_j \frac{\partial z^{l+1}_j}{\partial z^l_i}
$$

We need to know how to compute derivatives of outputs w.r.t. inputs only!
Backpropagation: Parameters

- Similarly if the module has parameters we want to know how the loss changes w.r.t. them:

  \[
  \frac{\partial L}{\partial w^l_i} = \sum_j \frac{\partial L}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial w^l_i} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial w^l_i}
  \]

- Derivatives of module outputs w.r.t. to the parameters are all we need

\[
\begin{align*}
  z^1 &= x_1 \\
  z^2 &= f_1(z^1) \\
  z^3 &= f_2(z^2) \\
  \mathcal{L} &= z^4 = f_3(z^3)
\end{align*}
\]
So for each module we need only to specify these three messages:

- **forward**: $z^{l+1} = f(z^l)$
- **backward**: $\frac{\partial z^{l+1}}{\partial z^l}$
- **parameter** (optional): $\frac{\partial z^{l+1}}{\partial w^l}$

### Backpropagation: Steps

1. **Forward Pass**
   - $z^1 = x_1$
   - $z^2 = f_1(z^1)$
   - $z^3 = f_2(z^2)$
   - $\mathcal{L} = z^4 = f_3(z^3)$

2. **Backward Pass** (Backpropagation)
   - $\delta^1 = \frac{\partial \mathcal{L}}{\partial z^1}$
   - $\delta^2 = \frac{\partial \mathcal{L}}{\partial z^2}$
   - $\delta^3 = \frac{\partial \mathcal{L}}{\partial z^3}$
   - $\delta^4 = \frac{\partial \mathcal{L}}{\partial z^4}$

3. **Parameters**
   - $\frac{\partial \mathcal{L}}{\partial w^1}$
Example: Linear Layer

- **forward:** \( z_{j}^{l+1} = \sum_{i=0}^{n} w_{ij}z_{i}^{l}, \quad j = 1, \ldots, K \)

- **backward:** \( \frac{\partial z_{j}^{l+1}}{\partial z_{i}^{l}} = w_{ij}, \quad i = 0, \ldots, n, \quad j = 1, \ldots, K \)

- **parameter:** \( \frac{\partial z_{j}^{l+1}}{\partial w_{ik}} = [j = k]z_{i}^{l} \)
Example: Squared Error

- **forward**: \( z^{l+1} = (y - z^l)^2 \)

- **backward**: \( \frac{\partial z^{l+1}}{\partial z^l} = 2(z^l - y), \quad i \in \{1, \ldots, n\} \)
Gradient Descent (GD)

- **Task**: find parameters which minimize loss over the training dataset:

\[
\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)
\]

where \( \theta \) is a set of all parameters defining the ANN

- Gradient descent: \( \theta_{k+1} = \theta_k - \alpha_k \nabla \mathcal{L}(\theta_k) \)

  where \( \alpha_k > 0 \) is the **learning rate** or **stepsize** at iteration \( k \)

- More about (Stochastic) Gradient Descent the next time . . .

![Graphs showing optimization process]
Parameter Initialization

- Is it a good idea to set initially all weights to a constant?
  - **No.** All neurons would behave the same: the same $\delta$s are backpropagated. We need to *break the symmetry*

- Use small random numbers, e.g., sample from a Gaussian distribution with zero mean:
  - works well for shallow networks,
  - for deep networks we might get into trouble
Gaussian Initialization Example

- MLP, 10 \texttt{tanh} layers, 500 units each. Each input fed with $\mathcal{N}(0, 1)$
- Weights initialized to $\mathcal{N}(0, \sigma^2)$; showing layer output distributions
  \[
  \sigma = 1
  \]

\[
\sigma = 0.01
\]
Vanishing Gradient

- For large $\sigma$ (saturation) the $\tanh$ derivative is almost zero.

- For small $\sigma$ (output close to zero):
  - the derivative is at most 1,
  - the weights are very small and $\frac{\partial z_{j}^{l+1}}{\partial z_{i}^{l}} = w_{ij}$ holds for the preceding linear layer.

- In both cases: $\delta \to 0$ as the number of layers increases.
Rectified Linear Unit (ReLU)

- \( f(s) = \max(0, s) \)

- Helps with the *vanishing gradients* problem: the gradient is constant for \( s > 0 \), while for sigmoid-like activations it becomes increasingly small.

- Fast to compute.

- Leads to sparse representations: \( s < 0 \) turns the neuron completely off, which blocks the gradient propagation → dead units → Leaky ReLU.

- Unbounded: use regularization to prevent numerical problems.
Xavier Initialization

- Glorot and Bengio: *Understanding the difficulty of training deep feedforward neural networks*, 2010
- For the linear neuron $s = \sum_i w_i x_i$, let $w_i$ and $x_i$ be independent random variables, $w_i$ and $x_i$ are i.i.d., $\mathbb{E}(x_i) = \mathbb{E}(w_i) = 0$:

\[
\text{Var}(s) = \text{Var} \left( \sum_i w_i x_i \right) = \sum_i \text{Var}(w_i x_i) = \\
= \sum_i \mathbb{E} \left( [w_i x_i - \mathbb{E}(w_i x_i)]^2 \right) = \sum_i \mathbb{E} \left( [w_i x_i - \mathbb{E}(w_i) \mathbb{E}(x_i)]^2 \right) = \\
= \sum_i \mathbb{E}(w_i^2 x_i^2) = \sum_i \mathbb{E}(w_i^2) \mathbb{E}(x_i^2) = \\
= \sum_i \mathbb{E}([w_i - \mathbb{E}(w_i)]^2) \mathbb{E}([x_i - \mathbb{E}(x_i)]^2) = \\
= \sum_i \text{Var}(x_i) \text{Var}(w_i) = n_{in} \text{Var}(x) \text{Var}(w)
\]
Xavier Initialization (contd.)

- We have $\text{Var}(s) = n_{in} \text{Var}(x) \text{Var}(w)$
- We want $\text{Var}(s) = \text{Var}(x)$
- Choose $\text{Var}(w) = \frac{1}{n_{in}}$
- Works well for $\text{tanh}$ as it is linear near zero
- Do not forget to standardize ANN input data
Regularization

How to deal with overfitting?

- get more data
- find simpler model, search for optimal architecture, e.g., number, type and size of layers
- constrain model by regularization

Most types of regularization are based on constraining the parameter space

Bayesian point of view: introduce prior distribution on model parameters
L2 Regularization (Weight Decay): Motivation

- Limit hypothesis space by limiting the size of the weight vector
- For L2 regularization we will push down $w^T w = \|w\|_2^2$
- You already know this from SVMs!

**Intuition:** sigmoid-like neurons kept near zero potential (via small weights) behave similarly to linear neurons

- L2 regularization (weight decay): zero mean Gaussian prior
Example: L2 Regularization for Linear Regression

- Recall the linear regression likelihood:

\[
p(y|w, X) = \left(2\pi\sigma^2\right)^{-\frac{m}{2}} e^{-\frac{1}{2\sigma^2}(y-Xw)^T(y-Xw)}
\]

- Define a Gaussian prior with zero mean and variance \(\sigma_0^2\) for the parameters:

\[
p(w) = \left(2\pi\sigma_0^2\right)^{-\frac{1}{2}} e^{-\frac{1}{2\sigma_0^2}w^Tw}
\]

- Then the posterior is:

\[
p(w|y, X) = \frac{p(y|w, X) \cdot p(w)}{p(y|X)}
\]

The denominator does not depend on the parameters \(w\):

\[
p(w|y, X) \propto p(y|w, X) \cdot p(w)
\]
MAP Estimate

- Maximizing $p(w|y, X)$ gives us the Maximum a posteriori (MAP) estimate:

$$w_{MAP} = \arg\max_w p(w|y, X) = \arg\min_w (-\log p(w|y, X))$$

where

$$-\log p(w|y, X) = \frac{1}{2\sigma^2} (y - Xw)^T (y - Xw) + \frac{1}{2\sigma_0^2} w^T w + C$$

- We can omit $C$, define $\lambda = \frac{\sigma^2}{\sigma_0^2}$ and minimize the loss function:

$$\mathcal{L}(w) = (y - Xw)^T (y - Xw) + \lambda w^T w$$

- The term $\lambda w^T w = \lambda \|w\|_2^2$ minimizes the size of the weight vector

- Note that we omit bias in $\lambda w^T w$
L2 Regularization Improves Numerical Stability

- Recall the solution for the linear regression $\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$
- What if $\mathbf{X}^T \mathbf{X}$ has no inverse?
- We can modify the solution by adding a small element to the diagonal:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}, \quad \lambda > 0$$

- It turns out that the solution is the minimizer of our regularized loss function:

$$\mathcal{L}(\mathbf{w}) = (\mathbf{y} - \mathbf{Xw})^T (\mathbf{y} - \mathbf{Xw}) + \lambda \mathbf{w}^T \mathbf{w},$$

see seminar for the derivation
Other Regularization Related Approaches

- L1 regularization: sum absolute values, i.e., use $\lambda \|w\|_1$
- Randomize inputs: same as the weight decay for linear neurons
- Dataset augmentation
- Early stopping: start with small weights, stop when validation loss starts to grow, often used for limited time-budget
- Weight sharing and sparse connectivity: Convolutional Neural Networks (next lecture)
- Model averaging (see lectures on Ensembling)
- Dropout and DropConnect
Next Lecture

- Stochastic Gradient Descent
- Deep Neural Networks
- Convolutional Neural Networks
- Transfer learning
\[ \hat{y} = \sum_{i=1}^{n} w_i x_i + b \]
$w_0 = b$

$x_1$

$x_2$

$\vdots$

$x_n$

$w_1$

$w_2$

$w_n$

$\hat{y}$
$0.8x + 2 + \mathcal{N}(0, 1)$
\[ \sigma(s) = \frac{1}{1 + e^{-s}} \]
\[
\begin{align*}
  z^1 &= x \\
  z^2 &= s = \sum_{i=1}^{K} x_i s_i \\
  z^3 &= \hat{y} = \text{softmax}(s) \\
  z^4 &= \mathcal{L} = \text{loss}(y, \hat{y})
\end{align*}
\]
\[ \frac{\partial L}{\partial \omega^l} \]
\( z^1 = x_1 \)

\( z^2 = f_1(z^1) \)

\( z^3 = f_2(z^2) \)

\( z^4 = f_3(z^3) \)

\( L = \frac{\partial L}{\partial z^4} = 1 \)

\( \delta^1 = \frac{\partial L}{\partial z^1} \)

\( \delta^2 = \frac{\partial L}{\partial z^2} \)

\( \delta^3 = \frac{\partial L}{\partial z^3} \)
\[ L = z^4 = f_3(z^3) \]

\[ z^3 = f_2(z^2) \]

\[ z^2 = f_1(z^1) \]

\[ z^1 = x_1 \]
1. Forward Pass

\[ z^1 = x_1 \]
\[ z^2 = f_1(z^1) \]
\[ z^3 = f_2(z^2) \]
\[ L = z^4 = f_3(z^3) \]

2. Backward Pass (Backpropagation)

\[ \delta^1 = \frac{\partial L}{\partial z^1} \]
\[ \delta^2 = \frac{\partial L}{\partial z^2} \]
\[ \delta^3 = \frac{\partial L}{\partial z^3} \]
\[ \delta^4 = \frac{\partial L}{\partial z^4} \]

3. Parameters

\[ \frac{\partial L}{\partial w^1} \]
\[ \tanh(x) \]
$\tanh'(x)$
\[ f(s) = \max(0, s) \]
\[ \sigma(s) = \frac{1}{1 + e^{-s}} \]