Statistical Machine Learning (BE4M33SSU)
Lecture 7: Artificial Neural Networks

Jan Drchal

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

Topics covered in the lecture:

♦ Neuron types
♦ Layers
♦ Loss functions
♦ Computing loss gradients via backpropagation
♦ Learning neural networks
♦ Regularization
McCulloch-Pitts Perceptron

\[ \mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n \]  
\[ \hat{y} \in \{-1, 1\} \]  
\[ \mathbf{w} = (w_1, w_2, \ldots, w_n)^T \in \mathbb{R}^n \]  
\[ b \in \mathbb{R} \]  
\[ s \in \mathbb{R} \]  
\[ f(s) = \begin{cases} 
-1 & \text{if } s < 0 \\
1 & \text{else}
\end{cases} \]  
\[ \hat{y} = f(s) = f \left( \sum_{i=1}^{n} w_i x_i + b \right) = f (\mathbf{w} \cdot \mathbf{x} + b) \]
McCulloch-Pitts Perceptron: Treating Bias

- Treat bias as an extra fixed input $x_0 = 1$ and weighted $w_0 = b$:

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

- $x' = (x_0, x_1, \ldots, x_n)^T \in \mathbb{R}^{n+1}$
- $w' = (w_0, w_1, \ldots, w_n)^T \in \mathbb{R}^{n+1}$
- Unless otherwise noted we will use $x, w$ instead of $x', w'$
Activation Functions

- **Step Function**
- **Bipolar Step Function**
- **Linear**
- **Sigmoid**
  \[ \sigma(s) = \frac{1}{1 + e^{-s}} \]
- **Hyperbolic Tangent**
- **ReLU**

**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

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

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

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

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

- For the whole dataset we get:

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

Assumption: data are Gaussian distributed with mean \( x_i \cdot w \) and variance \( \sigma^2 \):

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

Likelihood for i.i.d. data:

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

Negative Log Likelihood:

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

- Note that \( \sum_{i=1}^{m} (y_i - w \cdot x_i)^2 = (y - Xw)^T (y - Xw) \) is the **sum-of-squares** or **squared error** (SE).

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

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

- Note \( \hat{X} = (X^T X)^{-1} X^T \) is called the **Moore-Penrose pseudo-inverse**: if \( X \) is square and invertible then \( \hat{X} = X^{-1} \) (use \( (AB)^{-1} = B^{-1} A^{-1} \)).

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

- Denote: $\hat{y} = \sigma(s)$, $\hat{y} \in (0, 1)$
- Sigmoid output can represent a parameter of the Bernoulli distribution:

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

- Motivation: log-odds linear model (see AE4B33RPZ)
- Binary classifier: $h(\hat{y}) = \begin{cases} 1 & \text{if } \hat{y} > \frac{1}{2} \\ 0 & \text{else} \end{cases}$

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$
Logistic Regression

- MCP neuron using sigmoid activation function $\equiv$ logistic regression:

$$\hat{y} = \sigma(w \cdot x), \hat{y} \in (0, 1)$$

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

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

- Output class: $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_m)^T, \hat{y}_i \in (0, 1)$
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
Maximum Likelihood Estimation

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

- Derivative of the loss w.r.t. to the sigmoid argument: \( \frac{\partial L}{\partial s_i} = \hat{y}_i - y_i \)

- 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

- Hessian:
  \[
  \frac{\partial^2 L}{\partial w^2} = \sum_{i=1}^{m} \hat{y}_i(1 - \hat{y}_i) x_i \cdot x_i = X^T S X,
  \]
  where \( S \triangleq \text{diag}(\hat{y}_i(1 - \hat{y}_i)) \)

- Hessian is positive definite, hence the loss is convex and has unique global minimum (see AE4B33RPZ)
Rectified Linear Unit (ReLU)

- Definition $f(s) = \max(0, s)$
- Fast to compute
- Helps with *vanishing gradients* problem: the gradient is constant for $s > 0$, while for sigmoid-like activations it becomes increasingly small
- Leads to sparse representations: $s < 0$ turns the neuron completely off
- Unbounded: use regularization to prevent numerical problems
- Might block gradient propagation $\rightarrow$ dead units $\rightarrow$ Leaky ReLU
- Satisfies the universal approximation property
Multilayer Perceptron (MLP)

- Feed-forward ANN
- Fully-connected layers
- MLP for regression would typically use linear output layer
Recurrence Neural Network (RNN)

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

- Layers can be organized in *modules*
- Hierarchies of modules
- Module reuse
Linear Layer

- Output \( k \): \( \hat{y}_k = x \cdot w_k, \; k = 0, 1, \ldots, K \)
- All outputs using weight matrix \( W \): \( \hat{y} = x^T W \)
- Multiple samples: \( \hat{Y} = XW \)

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

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

\[
\hat{Y} = \begin{pmatrix} \hat{y}_1^T \\ \vdots \\ \hat{y}_m^T \end{pmatrix} = \begin{pmatrix} \hat{y}_{11} & \cdots & \hat{y}_{1K} \\ \vdots & \ddots & \vdots \\ \hat{y}_{m1} & \cdots & \hat{y}_{mK} \end{pmatrix}
\]
Softmax Layer

- Multinominal classification
- 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 probability distribution: \( 0 \leq \sigma_k \leq 1 \) for \( k \in \{1 \ldots K\} \) and \( \sum_{c=1}^{K} \sigma_c = 1 \)
- Describes class membership probabilities: \( p(y = k|s) = \sigma_k(s) \)
Softmax Layer (contd.)

- **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 \( y_{ik} = \mathbb{I}\{y_i = k\} \) and \( \hat{y}_{ik} = \mathbb{I}\{\hat{y}_i = k\} \)

- **Likelihood:**

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

- **Negative Log Likelihood:**

\[
\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{m} \sum_{c=1}^{K} \mathbb{I}\{y_i = c\} \log(\hat{y}_{ic})
\]

Again the **cross-entropy**

- See seminar for the gradient
Multinominal Logistic Regression

- **linear layer + softmax layer = multinominal logistic regression:**

\[ \hat{y}_k = \sigma_k(\mathbf{w}_k \cdot \mathbf{x}) \]

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

<table>
<thead>
<tr>
<th>problem</th>
<th>suggested loss function</th>
</tr>
</thead>
</table>
| binary classification | cross-entropy  \[
- \sum_{i=1}^{m} [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)] \] |
| multinominal classification | multinominal cross-entropy \[
- \sum_{i=1}^{m} \sum_{c=1}^{K} \mathbb{I}\{y_i = c\} \log(\hat{y}_{ic}) \] |
| regression         | squared error \[
\sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \]                             |
| multi-output regression | squared error  \[
\sum_{i=1}^{m} \sum_{c=1}^{K} (y_{ic} - \hat{y}_{ic})^2 \] |

- Mean w.r.t. to \( m \) is often used
Backpropagation Overview

- A method to compute gradient of the *loss function* with respect to its parameters

- 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:

\[
\mathcal{L}(\mathbf{w}) = - \sum_{i=1}^{m} \sum_{c=1}^{K} \mathbb{1}\{y_i = c\} \log \left( \frac{\exp(\mathbf{x}_i \cdot \mathbf{w}_c)}{\sum_{k=1}^{K} \exp(\mathbf{x}_i \cdot \mathbf{w}_k)} \right)
\]
Backpropagation Based on Modules

- Computation of $\nabla L(w)$ 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 hierarchies).
- 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 output for layer $l$, then:

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

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

```
z^1 = x_1 
z^2 = f_1(z^1) 
z^3 = f_2(z^2) 
\mathcal{L} = z^4 = f_3(z^3)
```

```
\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} = \frac{\partial L}{\partial \mathcal{L}} = 1
```
Backpropagation: Parameters

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

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

- Derivatives of module outputs w.r.t. to the parameters are all we need
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): \( \frac{\partial z^{l+1}}{\partial w^l} \)

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:**  
  \[ \frac{\partial z_j^{l+1}}{\partial w_{ik}} = \mathbb{I}\{j = k\} z_i^l \]
Example: Squared Error

- **forward:** \( z^{l+1} = \frac{1}{2} \sum_{i=1}^{K} (y_i - z^l_i)^2, \quad i \in \{1, \ldots, n\} \)

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

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

\[
\theta^* = \underset{\theta}{\text{argmin}} \mathcal{L}(\theta)
\]

where \( \theta \) is a set of all parameters defining the ANN (e.g., all weight matrices)

- Gradient descent:

\[
\theta(t+1) = \theta(t) - \eta(t) \nabla \mathcal{L}(\theta(t))
\]

where \( \eta(t) > 0 \) is the learning rate or step size at iteration \( t \)
Batch, Online and Mini-Batch Learning

When to update weights?

- **(Full) Batch learning**: after all patterns are used (epoch)
  - inefficient for redundant datasets
- **Online learning**: after each training pattern
  - noise can help overcome local minima but can also harm the convergence in the final stages while fine-tuning
    - Stochastic Gradient Descent (SGD) does this
      - convergence *almost surely* to local minimum when $\eta^{(t)}$ decreases *appropriately* in time
- **Mini-batch learning**: after a small sample of training patterns
Momentum

- Simulate inertia to overcome plateaus in the error landscape:

\[
\mathbf{v}^{(t+1)} = \mu \mathbf{v}^{(t)} - \eta^{(t)} \nabla \mathcal{L}(\theta^{(t)})
\]

\[
\theta^{(t+1)} = \theta^{(t)} + \mathbf{v}^{(t+1)}
\]

where \( \mu \in [0, 1] \) is the momentum parameter

- Momentum damps oscillations in directions of high curvature

- It builds velocity in directions with consistent (possibly small) gradient
**Nesterov’s Momentum**

- Different approach by Nesterov (1983, convex optimization):

\[
\mathbf{v}^{(t+1)} = \mu \mathbf{v}^{(t)} - \eta^{(t)} \nabla \mathcal{L}(\mathbf{\theta}^{(t)}) + \mu \mathbf{v}^{(t)}
\]

\[
\mathbf{\theta}^{(t+1)} = \mathbf{\theta}^{(t)} + \mathbf{v}^{(t+1)}
\]

- While classic momentum *corrects* the velocity using gradient at \( \mathbf{\theta}^{(t)} \), Nesterov uses \( \mathbf{\theta}^{(t)} + \mu \mathbf{v}^{(t)} \) which is similar to \( \mathbf{\theta}^{(t+1)} \)

- Stronger theoretical convergence guarantees for convex functions

- Slightly better in practice

- For more details see Sutskever et al.: *On the importance of initialization and momentum in deep learning*, 2013
Annealing the Learning Rate

- Decrease the learning rate in the course of optimization.

- **Step decay**: reduce the learning rate by a factor (e.g., $\frac{1}{2}$) every few iterations

- **Exponential decay**: set $\eta_t = \eta_0 e^{-kt}$ for the iteration $t$

- **Hyperbolic decay**: set $\eta_t = \frac{\eta_0}{1+kt}$
Resilient Propagation (Rprop)

- Motivation: a magnitude of gradient differs a lot for different Parameters
- Rprop (Riedmiller and Braun, 1992) does not use gradient value - the step size for each weight is adapted using its sign, only

1. for each $\theta_i$
2. \[ \text{if } \frac{\partial L(t-1)}{\partial \theta_i} \cdot \frac{\partial L(t)}{\partial \theta_i} > 0 \]
3. \[ \Delta_i^{(t)} = \min \left( \Delta_i^{(t-1)} \cdot \eta^+, \Delta_{max} \right) \]
4. \[ \text{elseif } \frac{\partial L(t-1)}{\partial \theta_i} \cdot \frac{\partial L(t)}{\partial \theta_i} < 0 \]
5. \[ \Delta_i^{(t)} = \max \left( \Delta_i^{(t-1)} \cdot \eta^-, \Delta_{min} \right) \]
6. \[ \theta_i^{(t+1)} = \theta_i^{(t)} - \text{sign} \left( \frac{\partial L(t)}{\partial \theta_i} \right) \cdot \Delta_i^{(t)} \]

where the step size $\Delta_i^{(t)} \in [\Delta_{min}, \Delta_{max}]$ and $0 < \eta^- < 1 < \eta^+$

- Typically order of magnitude faster than basic Gradient Descent
- Does not work well for mini-batches
- Igel and Hüsken: *Improving the Rprop Learning Algorithm*, 2000
Adagrad

- Adaptive Gradient method
- Idea: reduce learning rates for parameters having high values of gradient

\[
g_i^{(t+1)} = g_i^{(t)} + \left( \frac{\partial L}{\partial \theta_i^{(t)}} \right)^2
\]

\[
\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t+1)}} + \epsilon} \cdot \frac{\partial L}{\partial \theta_i^{(t)}}
\]

- \( g_i \) accumulates squared partial derivatives w.r.t. to the parameter \( \theta_i \)
- \( \epsilon \) is a small positive number to prevent division by zero
- Weakness: ever increasing \( g_i \) leads to slow convergence eventually
RMSProp

- Similar to Adagrad but employs a moving average:

\[
g_i^{(t+1)} = \gamma g_i^{(t)} + (1 - \gamma) \left( \frac{\partial L}{\partial \theta_i^{(t)}} \right)^2
\]

- \(\gamma\) is a *decay* parameter (typical value \(\gamma = 0.9\))

- Unlike for Adagrad updates do not get infinitesimally small
Adam (Adaptive Moment Estimation)


\[
\begin{align*}
    m_i^{(t+1)} &= \beta_1 m_i^{(t)} + (1 - \beta_1) \frac{\partial L}{\partial \theta_i^{(t)}} \\
    \hat{m}_i^{(t+1)} &= \frac{m_i^{(t+1)}}{1 - \beta_1 m_i^{(t)}} \\
    v_i^{(t+1)} &= \beta_2 v_i^{(t)} + (1 - \beta_2) \left( \frac{\partial L}{\partial \theta_i^{(t)}} \right)^2 \\
    \hat{v}_i^{(t+1)} &= \frac{v_i^{(t+1)}}{1 - \beta_2 v_i^{(t)}} \\
    \theta_i^{(t+1)} &= \theta_i^{(t)} - \eta \frac{\hat{m}_i^{(t+1)}}{\sqrt{\hat{v}_i^{(t+1)}} + \epsilon}
\end{align*}
\]

- \(m_i\) and \(v_i\) are first and second raw moment estimates of gradient (mean and uncentered variance)
- \(\hat{m}_i\) and \(\hat{v}_i\) are corrections of \(m_i\) and \(v_i\) as these are biased to zero due to initialization \(m_i = v_i = 0\)
- \(\beta_1\) and \(\beta_2\) are *decay* parameters
Second Order Methods

- Newton’s method for optimization:

\[
\theta^{(t+1)} = \theta^{(t)} - \left( \nabla^2 \mathcal{L}(\theta^{(t)}) \right)^{-1} \nabla \mathcal{L}(\theta^{(t)})
\]

- No parameters needed
- Need to compute the Hessian matrix and invert it!
- Approximations, e.g., L-BFGS are not suitable for mini-batches
Regularization

- How to deal with overfitting?
  - get more data
  - find simpler model, search for optimal architecture, e.g., number, type and size of layers
  - use regularization

- Most types of regularization are based on penalties for model complexity

- Bayesian point of view: introduce prior distribution on model parameters
L2 Regularization

- Recall the solution for the linear regression \( w^* = (X^T X)^{-1} X^T y \)
- What if \( X^T X \) has no inverse?
- We can modify the solution by adding a small element to the diagonal:
  \[
  w^* = (X^T X + \lambda I)^{-1} X^T y, \quad \lambda > 0
  \]
- It turns out that this approach no only helps with inverting \( X^T X \) but it also improves model generalization
- It is the solution of the regularized loss function:
  \[
  \mathcal{L}(w) = (y - Xw)^T (y - Xw) + \lambda w^T w
  \]
  this one is called the L2 regularization, see seminar for the derivation
- The term \( \lambda w^T w = \lambda \|w\|^2 \) minimizes the size of the weight vector
- Note that we omit bias in \( \lambda w^T w \)
L2 regularization (contd.)

- L2 regularization is also called the ridge regression or the **weight decay**.
Recall the 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 we already know:

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

- Having zero mean Gaussian prior keeps the weights smaller
- Weight decay is widely used for most types of layers in ANNs
- Intuition: sigmoid-like neurons kept near zero potential (via small weights) behave similarly to linear neurons
- The same works for other models, e.g., polynomial regression
- $\lambda$ is usually set using cross validation
Dropout

- Idea: average many neural networks, share weights to make this computationally feasible
- For each training example omit a unit with probability $p$ (often $p = 0.5$)
- This is like sampling from $2^U$ networks where $U$ is the number of units
- Typically only a small amount of $2^U$ networks is actually sampled
- Prevents coadaptation of feature detectors

Figure 1: Dropout Neural Net Model.
Left: A standard neural net with 2 hidden layers.
Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.
Dropout (contd.)

- How to make predictions with networks using dropout?
- Scale outputs (output weights) of all affected units by $p$
- For a linear unit taking inputs from the dropout layer we have:

$$s = \sum_{i=1}^{n} w_i \delta_i x_i, \quad p(\delta_i | p) = \text{Ber}(\delta_i | p)$$

$$\mathbb{E}(s) = \sum_{i=1}^{n} w_i \mathbb{E}(\delta_i) x_i = p \sum_{i=1}^{n} w_i x_i$$

where the expectation is computed over all $2^n$ configurations

- For general neural networks we still get a good approximation of the expectation when scaling by $p$

- See Baldi and Sadowski: *The Dropout Learning Algorithm*, 2014

- When used for inputs higher values of $p$ are suggested
Other Regularization Approaches

- **L1 regularization**: sum absolute values, i.e., use $\lambda \| w \|_1$
- **Early stopping**: start with small weights, stop when validation loss starts to grow
- **Randomize inputs**: same as the weight decay for linear neurons
- **Noisy weights**
- **DropConnect**: connection-based dropout
- **Other weight sharing approaches**: Convolutional Neural Networks
- **Model averaging**
Next Lecture

- Deep Neural Networks
- Convolutional Neural Networks
- Autoencoders
- Transfer learning
$w_0 = b$

$\sum w_i x_i$
**Step Function**

**Bipolar Step Function**

**Linear**

\[ f(s) = s \]

**Sigmoid**

\[ \sigma(s) = \frac{1}{1 + e^{-s}} \]

**Hyperbolic Tangent**

**ReLU**

\[ f(s) = \max(0, s) \]
The equation $0.8x + 2 + \mathcal{N}(0, 1)$ represents a linear model with Gaussian noise. The graph shows a scatter plot of data points and a straight line representing the model, with a dashed line indicating the trend of the data.
\[ \sigma(s) = \frac{1}{1 + e^{-s}} \]
$f(s) = \max(0, s)$
softmax
1
2
3
K
s_1
s_2
s_3
s_K
\hat{y}_1
\hat{y}_2
\hat{y}_3
\hat{y}_K
\[ z^1 = x_1 \]

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

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

\[ L = z^4 = f_3(z^3) \]

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

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

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

\[ \delta^4 = \frac{\partial L}{\partial z^4} = \frac{\partial L}{\partial L} = 1 \]
\[
\begin{align*}
  z^1 &= x_1 \\
  \delta^1 &= \frac{\partial \mathcal{L}}{\partial z^1} \\
  \frac{\partial \mathcal{L}}{\partial w^1} \\
  z^2 &= f_1(z^1) \\
  \delta^2 &= \frac{\partial \mathcal{L}}{\partial z^2} \\
  z^3 &= f_2(z^2) \\
  \delta^3 &= \frac{\partial \mathcal{L}}{\partial z^3} \\
  \mathcal{L} &= z^4 = f_3(z^3) \\
  \delta^4 &= \frac{\partial \mathcal{L}}{\partial z^4} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1
\end{align*}
\]
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

\[ \frac{\partial \mathcal{L}}{\partial w^1} \]
$N = 15$, $\eta = 0.1$

$N = 15$, $\eta = 0.4$

$N = 15$, $\eta = 0.55$
$N = 15, \mu = 0., \eta = 0.1$

$N = 15, \mu = 0.7, \eta = 0.1$
Nesterov classic

\[
\n \theta^{(t)} - \eta^{(t)} \nabla \mathcal{L}(\theta^{(t)}) + (\theta^{(t)} + \mu v^{(t)}) \nabla \mathcal{L}(\theta^{(t)} + \mu v^{(t)})
\]

classic

\[
\n \theta^{(t)} + \mu v^{(t)}
\]

Nesterov
$N = 1$
\[ \sigma(s) = \frac{1}{1 + e^{-s}} \]
Dropout is a technique that addresses both these issues. It prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Figure 1. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability $p$ independent of other units, where $p$ can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.