Hot Topics in Machine Learning
07 – Neural Networks

Prof. Dr. Rainer Gemulla

Universität Mannheim

HWS 2017
Artificial Neural Network (ANN)

- A powerful family of models inspired by biological neural networks
  - Hope: should be good at what humans are good at
  - Many relationships to what we learned so far
- Studied to
  - Understand how the brain works (not covered here)
  - Build learning machines
- Used for a variety of learning tasks
  - Classification, regression, prediction, clustering, …
  - Feature generation / dimensionality reduction
  - Supervised, semi-supervised, unsupervised
  - Generative model, discriminative model, discriminative functions
- State-of-the-art results in areas such as
  - Computer vision
  - Natural language processing
  - Speech recognition
  - Statistical relational learning
  - …
Art
Handwriting recognition

<table>
<thead>
<tr>
<th>20M-40M-60M-80M-100M-120M-150N</th>
<th>0.35</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Image of handwritten digits]</td>
<td></td>
</tr>
</tbody>
</table>

35 errors

0.35% errors on validation
30 out of the 35 errors have correct second prediction
Traffic sign recognition

- Initial phase:
  - More than 10 teams
  - We won 1st place with 98.98%
- 40000 color images 15x15 to 250x250
- 43 different signs: 26640 training images, 12569 testing images

Input:
- Extract Region of Interest (ROI)
- Enhance contrast with four different methods
- Resize up/down to 48x48 pixels
- Great variation in details, contrast and illumination

43 different signs: 26640 training images, 12569 testing images

- Final phase: last Wednesday at IJCNN 2011
- Flexible, High Performance Convolutional Neural Networks for Image Classification; D. Ciresan et al.

99.43% recognition rate
(humans: 98.84%)
Word2vec

Takes a large corpus and maps words to vectors (unsupervised), capturing “semantic” similarity.

### Similar to “Sweden”?

<table>
<thead>
<tr>
<th>Word</th>
<th>Cosine distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>norway</td>
<td>0.760124</td>
</tr>
<tr>
<td>denmark</td>
<td>0.715460</td>
</tr>
<tr>
<td>finland</td>
<td>0.620022</td>
</tr>
<tr>
<td>switzerland</td>
<td>0.588132</td>
</tr>
<tr>
<td>belgium</td>
<td>0.585835</td>
</tr>
<tr>
<td>netherlands</td>
<td>0.574631</td>
</tr>
<tr>
<td>iceland</td>
<td>0.562368</td>
</tr>
<tr>
<td>estonia</td>
<td>0.547621</td>
</tr>
<tr>
<td>slovenia</td>
<td>0.531408</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A is to B like C is to ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rome-Italy + China=Beijing</td>
</tr>
<tr>
<td>king-queen + man=woman</td>
</tr>
<tr>
<td>house-roof + castle=dome</td>
</tr>
<tr>
<td>Trump-republican + Obama=Democratic</td>
</tr>
<tr>
<td>building-architect + software=programmer</td>
</tr>
<tr>
<td>monkey-human + dinosaur=fossil</td>
</tr>
</tbody>
</table>
Speech and natural language processing

What are hot topics in machine learning?

What are hot topics in machine learning?

Translate from ENGLISH (detected)

Translate into GERMAN

What are hot topics in machine learning?

Was sind aktuelle Themen des maschinenellen Lernens?

Named Entity Recognition:

1. Chase Manhattan and its merger partner J.P. Morgan and Citibank, which was involved in moving about $100 million

Basic dependencies:

1. Chase Manhattan and its merger partner J.P. Morgan and Citibank, which was involved in moving about $100 million for Raul Salinas de Gortari, brother of a former Mexican president, to banks in Switzerland, are also expected to sign on.
Google Smart Reply

**Turkey!**

Hi all,
We wanted to invite you to join us for an early Thanksgiving on November 22nd, beginning around 2PM. Please bring your favorite dish! RSVP by next week.

Dave

---

**Server issues**

Hi team,

The server appears to be dropping about 10% of requests (see attached dashboards). There hasn't been a new release since last night, so I'm not sure what's going on. Is anyone looking into this?

...
Overview

In this lecture; basics of

- Artificial neural networks
- Feedforward neural networks
- Autoencoders
- Parameter estimation
- The neural network zoo

Not in this lecture

- In-depth coverage
- Learning theory
- Connections to how the brain works
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo

7. Summary
Artificial neuron

• An artificial neuron (AN) is a function $f : \mathbb{R}^n \to \mathbb{R}$
  ▶ Inputs are vectors $x \in \mathbb{R}^n$
  ▶ Output is a value $y \in \mathbb{R}$

• $f$ is taken from a family of functions that is parameterized by
  ▶ A weight vector $w \in \mathbb{R}^n$ (one weight per input)
  ▶ A bias $b \in \mathbb{R}$
  ▶ A transfer function or activation function $\phi : \mathbb{R} \to \mathbb{R}$

• Basic structure: $y = \phi(b + \langle w, x \rangle)$
  ▶ Computes the weighted sum $a = b + \langle w, x \rangle$ of its inputs and bias
  ▶ $a$ is called activation
  ▶ Passes $a$ through the transfer function to obtain output $y$
  ▶ $\phi$ can be deterministic or stochastic

• As before: bias can be replaced by an additional input $x_0 = 1$ and corresponding weight $w_0 = b$
Types of artificial neurons

- The **type** of an AN is determined by its transfer function $\phi$
- An AN of a given type can represent the family of functions

$$F_\phi = \{ x \rightarrow \phi(b + \langle w, x \rangle) \mid b \in \mathbb{R}, w \in \mathbb{R}^n \}$$

- Each function in this family can be represented by its bias and weight vector
- We will see later that types are usually specified up-front, whereas weights are learned
- The simplest type of neuron is a **constant neuron**
  - No inputs; output fixed value $x \in \mathbb{R}$
  - Notation (from now on): $\bigcirc x$
Linear neuron

- Uses $\phi(a) = a$
- Notation: 

Simple but computationally limited

We almost always want non-linear transfer functions
Binary threshold neuron

- Also called **McCulloch-Pitts neurons**
- Outputs fixed “spike” if weighted sum exceeds a threshold (= negative bias)

\[ \phi(a) = l(a \geq 0) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{otherwise} \end{cases} \]

- Notation: \( \lceil \rfloor \) or without bias \( \geq 0, \geq 1, \ldots \)

- Note: this is a non-linear transfer function
- One interpretation: each input is the truth value of some proposition, output is truth value of another proposition
Rectified linear neuron (ReLU)

- Also called linear threshold neuron
- \( \phi(a) = \max\{0, a\} = \begin{cases} a & \text{if } a \geq 0 \\ 0 & \text{otherwise} \end{cases} \)
- Notation: \( \square \)

- Note: again, a non-linear transfer function
- Perhaps the most popular type of neuron in deep NNs
Logistic neuron

- Use logistic function $\phi(a) = \frac{1}{1+\exp(-a)}$
- Other sigmoids can also be used (e.g., tanh)
- Notation: 

\[
a = b + \langle x, w \rangle
\]

- Gives a real-valued output that is smooth and bounded
Stochastic binary neuron

- Also use logistic function
- But output of the logistic function is treated as a probability of producing a spike (1)

\[ \phi(a) = \begin{cases} 
1 & \text{with probability } \frac{1}{1 + \exp(-a)} \\
0 & \text{otherwise} 
\end{cases} \]

- Defines a probability distribution over output
What is an artificial neural network?

- A network of artificial neurons
  - A set of (artificial) neurons
  - Connections between neurons (directed or undirected)
- Many different architectures
  - How many neurons?
  - Of which type?
  - Are there output neurons?
  - Are there hidden neurons (neither input nor output)?
  - Which neurons are connected?
  - Are connections directed or undirected?
  - Are there cycles?
- Picking the right architecture for the problem at hand is important and requires skill/thought/compute power → Architecture engineering
- Can represent a wide range of functions (universal approximation theorem)
- We look at some types of ANNs in this lecture
MNIST, best performer, architecture

Deep convolutional neural network (no preprocessing)

Figure 1: Architecture of a convolutional neural network with 3 GPU implementation.
MNIST, best performer, result

0.35% errors on validation
30 out of the 35 errors have correct second prediction
Learning

- Once we settled on an architecture, we need to learn connection strengths (weights)
- Requires a suitable notion of model performance (e.g., error or likelihood)
- Generally need to learn all weights jointly
  → As before: optimization, MLE, MAP, Bayesian
- Values of hidden units can be thought of as features, but which features are good is unknown and needs to be learned (more later)
- This makes learning hard
Neural network everything

- Deep neural networks produce great results in some areas
  - Use them for “everything”?

- Consider a supervised learning task:

<table>
<thead>
<tr>
<th>Available data</th>
<th>Affordable model complexity</th>
<th>Need for domain knowledge</th>
<th>Training cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

- Deep neural networks are complex models
  - Millions of parameters
  - Need sufficient data
  - Need sufficient computational resources
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo

7. Summary
Feedforward neural networks

- A **feedforward neural network (FNN)** is an ANN in which
  - All connections are directed, and
  - There are no cycles (i.e., forms a DAG)
- Neurons can be grouped in **layers**
  - Input neurons: no incoming edges (first layer)
  - Output neurons: no outgoing edges (last layer)
  - Hidden neurons: all others (layer = maximum distance from input)
  - Layers do not need to be fully connected
  - Often: edges only between subsequent layers
    (but: edges that skip layers are allowed, too)
- Example: an FNN with one hidden layer (omitting bias inputs)
Supervised learning with FNNs

- **Supervised learning**
  - Given a labeled set $D = \{ (x_i, y_i) \}_{i=1}^{N}$ of input-output pairs,
  - Learn a mapping from inputs $x_i$ to outputs $y_i$
  - In FNNs: for input $x_i$, we want output $\hat{y}_i$ close to $y_i$
  - Learning means adjusting the weights such that the FNN does this

- **FNNs are discriminative**
  - Given an input $x$, they compute an output $\hat{y}$
  - But they don’t allow going from outputs to inputs

- **Hidden layers are inputs of the next layer**
  - We may also think of hidden layers as features for the next layer
  - These features are not provided upfront, but learned

- **FNNs without hidden layers correspond to generalized linear models**
Linear neural networks

- In a **linear neural network**, all neurons are linear neurons
- Suppose no hidden layer and one output
  - Then $\hat{y}_i = b + \langle w, x_i \rangle$
  - Suppose we want to minimize the **squared error** ($= \text{sum of squared errors the network makes over all examples and outputs}$)
  - Objective function: $\text{argmin}_{b,w} \sum_i (y_i - \hat{y}_i)^2$
  - That’s the same objective as linear regression
- Linear neural networks (w/o hidden layer) $\equiv$ linear regression
  - To determine bias and weights, we can use any suitable linear regression library
  - If we have multiple outputs, we get multivariate linear regression
- Outputs remain linear even with hidden layers (why?)
  - That’s why we often want non-linear transfer functions
Representing weights

- Consider a linear NN with $n$ inputs, $m$ outputs, and no biases

\[ x_1 \quad w_1 \quad \hat{y}_1 \]
\[ x_2 \quad w_1 \quad \hat{y}_2 \]
\[ x_3 \quad w_2 \]
\[ x_4 \quad w_2 \]

- We have $\hat{y}_j = \langle w_j, x \rangle$, where $w_j \equiv$ weights of $j$’s output neuron
- Recall matrix products: $[AB]_{ij} = \sum_k a_{ik} b_{kj} = \langle a_i, b_j \rangle = a_i^T b_j$, where $a_i$ is the $i$-th row of $A$ and $b_j$ the $j$-th column of $B$
- Let $W$ be an $n \times m$ weight matrix in which the $j$-th column equals $w_j$; then $\hat{y} = W^T x$
- For our example, $W = \begin{pmatrix} w_1 & w_2 \end{pmatrix}$ and

\[
W^T x = \begin{pmatrix} w_1^T \\ w_2^T \end{pmatrix} x = \begin{pmatrix} \langle w_1, x \rangle \\ \langle w_2, x \rangle \end{pmatrix} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \end{pmatrix} = \hat{y}
\]

- The “action” of a layer of a linear NN corresponds to a vector-matrix product between inputs and weights
**Autoencoders**

- Linear NNs can do more than what we may expect at first glance
- Turns out FNNs are useful for unsupervised learning as well
  - We are given an unlabeled dataset $D = \{ x_i \}_{i=1}^N$
  - We don’t have outputs
  - We want to find structure, or patterns, or reduce dimensionality
  - Idea: use input $x_i$ as its own output, i.e., set $y_i = x_i$
  - The resulting FNN is called a feedforward autoencoder
- Two simple (and useless) linear feedforward autoencoders
  - Can you figure out optimal weight matrices (such that $\hat{x}_{ij} = x_{ij}$)?
Autoencoders and bottlenecks

- Consider a linear feedforward autoencoder with one hidden layer with $r < n$ hidden neurons
- What are the optimal weight matrices for this network?

- A layer with few neurons is referred to as a **bottleneck**
  - Since the network needs to **reconstruct** all inputs well, the optimal choice of weight matrices depends on all inputs
  - Since the hidden layer has less values than the input, we may view its values $h_i$ as a compressed representation of input $x_i$
  - Alternatively: the $h_i$ can be seen as suitable “features” that capture the “essence” of the data (= dimensionality reduction)
  - We refer to $h_i$ as a **distributed representation** of $x_i$
Obtaining optimal weights

- We have \( h_i = W_1^T x \) and \( \hat{x}_i = W_2^T h_i = W_2^T W_1^T x \)
- For squared error, we search \( \arg\min_{W_1, W_2} \left[ \sum_i \sum_j (x_{ij} - \hat{x}_{ij})^2 \right] \)
- The solution can be read off the so-called singular value decomposition (SVD) of \( X \) (lecture data mining and matrices)
  - Let \( X \) be the design matrix and \( U_r \Sigma_r V_r^T \) its rank-\( r \) truncated SVD
  - \( U_r \) is an \( N \times r \) matrix with the first \( r \) left-singular vectors of \( X \)
  - \( V_r \) is an \( n \times r \) matrix with the first \( r \) right-singular vectors of \( X \)
  - \( \Sigma_r \) is an \( r \times r \) matrix with the first \( r \) singular values of \( X \)
  - An optimal solution is \( W_1 = V_r \) and \( W_2 = V_r^T \)
  - For this solution, \( h_i = [U_r]_{i*} \Sigma_r \)
- This is closely related to principal component analysis (PCA)
  - Main difference: PCA centers the data so that each feature has mean 0 (often: also normalizes each feature)
  - Then \( W_1 \) contains the first \( r \) principal components as its columns
  - And \( h_i \) contains the \( r \) scores for \( x_i \) that PCA produces
**Example: Weather data**

<table>
<thead>
<tr>
<th>City</th>
<th>Jan</th>
<th>Apr</th>
<th>Jul</th>
<th>Oct</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stockholm</td>
<td>-0.70</td>
<td>8.60</td>
<td>21.90</td>
<td>9.90</td>
<td>10.00</td>
</tr>
<tr>
<td>Minsk</td>
<td>-2.10</td>
<td>12.20</td>
<td>23.60</td>
<td>10.20</td>
<td>10.60</td>
</tr>
<tr>
<td>London</td>
<td>7.90</td>
<td>13.30</td>
<td>22.80</td>
<td>15.20</td>
<td>14.80</td>
</tr>
<tr>
<td>Budapest</td>
<td>1.20</td>
<td>16.30</td>
<td>26.50</td>
<td>16.10</td>
<td>15.00</td>
</tr>
<tr>
<td>Paris</td>
<td>6.90</td>
<td>14.70</td>
<td>24.40</td>
<td>15.80</td>
<td>15.50</td>
</tr>
<tr>
<td>Bucharests</td>
<td>1.50</td>
<td>18.00</td>
<td>28.80</td>
<td>18.00</td>
<td>16.50</td>
</tr>
<tr>
<td>Barcelona</td>
<td>12.40</td>
<td>17.60</td>
<td>27.50</td>
<td>21.50</td>
<td>20.00</td>
</tr>
<tr>
<td>Rome</td>
<td>11.90</td>
<td>17.70</td>
<td>30.30</td>
<td>21.40</td>
<td>20.40</td>
</tr>
<tr>
<td>Lisbon</td>
<td>14.80</td>
<td>19.80</td>
<td>27.90</td>
<td>22.50</td>
<td>21.50</td>
</tr>
<tr>
<td>Athens</td>
<td>12.90</td>
<td>20.30</td>
<td>32.60</td>
<td>23.10</td>
<td>22.30</td>
</tr>
<tr>
<td>Valencia</td>
<td>16.10</td>
<td>20.20</td>
<td>29.10</td>
<td>23.60</td>
<td>22.30</td>
</tr>
<tr>
<td>Malta</td>
<td>16.10</td>
<td>20.00</td>
<td>31.50</td>
<td>25.20</td>
<td>23.20</td>
</tr>
</tbody>
</table>
Example: Weights and representation

<table>
<thead>
<tr>
<th>$W_1$</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>0.22</td>
<td>-0.85</td>
</tr>
<tr>
<td>Apr</td>
<td>0.40</td>
<td>0.06</td>
</tr>
<tr>
<td>Jul</td>
<td>0.64</td>
<td>0.47</td>
</tr>
<tr>
<td>Oct</td>
<td>0.45</td>
<td>-0.18</td>
</tr>
<tr>
<td>Year</td>
<td>0.43</td>
<td>-0.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$W_2$</th>
<th>Jan</th>
<th>Apr</th>
<th>Jul</th>
<th>Oct</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.22</td>
<td>0.40</td>
<td>0.64</td>
<td>0.45</td>
<td>0.43</td>
</tr>
<tr>
<td>2</td>
<td>-0.85</td>
<td>0.06</td>
<td>0.47</td>
<td>-0.18</td>
<td>-0.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$H$</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stockholm</td>
<td>26.02</td>
<td>8.25</td>
</tr>
<tr>
<td>Minsk</td>
<td>28.63</td>
<td>10.30</td>
</tr>
<tr>
<td>London</td>
<td>34.76</td>
<td>0.00</td>
</tr>
<tr>
<td>Budapest</td>
<td>37.36</td>
<td>7.42</td>
</tr>
<tr>
<td>Paris</td>
<td>36.69</td>
<td>1.48</td>
</tr>
<tr>
<td>Bucharest</td>
<td>41.07</td>
<td>7.79</td>
</tr>
<tr>
<td>Barcelona</td>
<td>45.51</td>
<td>-3.22</td>
</tr>
<tr>
<td>Rome</td>
<td>47.36</td>
<td>-1.50</td>
</tr>
<tr>
<td>Lisbon</td>
<td>48.25</td>
<td>-5.34</td>
</tr>
<tr>
<td>Athens</td>
<td>51.66</td>
<td>-1.69</td>
</tr>
<tr>
<td>Valencia</td>
<td>50.30</td>
<td>-6.16</td>
</tr>
<tr>
<td>Malta</td>
<td>52.86</td>
<td>-5.45</td>
</tr>
</tbody>
</table>
Bottlenecks of two neurons can be useful for visualization.
Example: Representing documents

804414 newswire stories, inputs = per-document probabilities of 2000 most common word stems, autoencoder = logistic hidden units + linear output units
Discussion: Bottlenecks and autoencoders

- Bottlenecks force network to capture what is most important
  - Useful for supervised, semi-supervised, and unsupervised learning
- Bottlenecks are useful for
  - Dimensionality reduction and feature generation
  - Smoothing or denoising
  - Enhancing clustering properties of the data
  - Visualization
- Autoencoders use bottlenecks to find distributed representations
  - E.g., for semi-supervised learning: train autoencoder on all inputs (labeled and unlabeled), use representations for subsequent learner
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo

7. Summary
Perceptron

- Invented 1957 by Frank Rosenblatt at the Cornell Aeronautical Laboratory
- Corresponds to an FNN without hidden layers and binary threshold units for outputs (single-layer perceptron)

\[ x_1 \]
\[ x_2 \]
\[ x_3 \]
\[ \hat{y} \]

- This is our first NN with a non-linearity
- Many hopes and much controversy about what it can do at the time (see reference below for history)

Olazaran, 1996
Decision boundary of perceptrons

- Consider a perceptron with a single output, weights $\mathbf{w}$, and bias $b$
- We can view it as a **binary classifier** with classification rule

\[
\hat{y} = \begin{cases} 
0 & b + \langle \mathbf{w}, \mathbf{x} \rangle < 0 \\
1 & b + \langle \mathbf{w}, \mathbf{x} \rangle \geq 0 
\end{cases}
\]

- The **decision boundary** is thus

\[
\langle \mathbf{w}, \mathbf{x} \rangle = -b
\]

- What does this mean?
  - We have touched on this multiple times, let’s review
Inner product (geometric interpretation)

The geometric interpretation of the inner product of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ is given by

$$\langle \mathbf{u}, \mathbf{v} \rangle = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta,$$

where $-\pi \leq \theta \leq \pi$ denotes the angle between $\mathbf{u}$ and $\mathbf{v}$. 
Understanding perceptrons (1)

- Consider the case $b = 0$ and the classification rule

$$
\hat{y} = \begin{cases} 
0 & \langle w, x \rangle < 0 \\
1 & \langle w, x \rangle \geq 0 
\end{cases} = \begin{cases} 
0 & \|w\| \|x\| \cos \angle(w, x) < 0 \\
1 & \|w\| \|x\| \cos \angle(w, x) \geq 0 
\end{cases}
$$

- Observe
  - Negative instances have angle $|\angle(w, x)| > 90^\circ$
  - The decision boundary has angle $|\angle(w, x)| = 90^\circ$ (includes origin)
  - Positive instances have angle $|\angle(w, x)| \leq 90^\circ$

- The decision boundary defines a hyperplane
  - Subspace of dimensionality $n - 1$ with normal vector $w$
  - Set of points in $\mathbb{R}^n$ orthogonal to $w$
  - E.g., for two dimensions we have the line $w_1 x_1 + w_2 x_2 = 0$
    (and consequently $x_2 = -\frac{w_1}{w_2} x_1$)

- If we add a bias term $b \neq 0$, we obtain an affine hyperplane
  - I.e., does not go through origin
  - For two dimensions, intercept is $-b/w_2$
Understanding perceptrons (2)

Intercept \(= -\frac{b}{w_2}\)
What can perceptrons learn?

- Perceptrons can classify perfectly if there exists an affine hyperplane that separates the classes
  - We then say the data is **linearly separable**
- Otherwise, the perceptron must make errors on some inputs
- This is quite limited; e.g., perceptrons cannot learn the XOR function

- We will come back to this later
The perceptron learning algorithm (1)

- Assume $b = 0$ (but: can have bias feature $x_0 = 1$)
- How can we find a corresponding weight vector?
- Let $\tilde{y}_i = \begin{cases} -1 & y_i = 0 \\ 1 & y_i = 1 \end{cases}$
- Perceptron makes an error whenever $\text{sgn}(\langle w, x_i \rangle) \neq \tilde{y}_i$
- Rosenblatt developed a simple perceptron learning algorithm:
  1: $w \leftarrow 0$ \hspace{1cm} // initialize (optionally: random)
  2: while $\exists i : \text{sgn}(\langle w, x_i \rangle) \neq \tilde{y}_i$ \hspace{1cm} // i.e., an error on $(x_i, y_i)$
  3: $w \leftarrow w + \tilde{y}_i x_i$ \hspace{1cm} // add/substract data
- We add when error on pos. instance ($\langle w, x_i \rangle \leq 0, y_i = 1$)
- We substract when error on neg. instance ($\langle w, x_i \rangle \geq 0, y_i = 0$)
The perceptron learning algorithm (2)

Here: only positive instances.

1) Initial configuration

2) After correction with $x_1$

3) After correction with $x_3$

4) After correction with $x_1$
The perceptron learning algorithm (3)

- To get additional intuition, suppose we make an error on $x_i$
- We then “rotate” $w$
  - Towards $x_i$ when positive class $\rightarrow$ smaller angle $\angle(w, x_i)$
  - Away from $x_i$ when negative class $\rightarrow$ larger angle $\angle(w, x_i)$
- Now suppose we test $x_i$ again
  - $\langle w + \tilde{y}_i x_i, x_i \rangle = \langle w, x_i \rangle + \langle \tilde{y}_i x_i, x_i \rangle = \langle w, x_i \rangle + \tilde{y}_i \|x_i\|^2 \geq 0$
    - If we misclassified a pos. instance, inner product now larger
    - If we misclassified a neg. instance, inner product now smaller
    - i.e., we either won’t make an error again or are closer to decision boundary
  - Note: $\|x_i\|^2 > 0$ when we include bias feature ($x_0 = 1$)
- How well does this algorithm work?
The perceptron learning algorithm (4)

- When data is linearly separable
  - Perceptron learning algorithm will converge after a finite number of steps (more in exercise)
  - May perform exponentially many weight updates
  - Produces an arbitrary solution that separates the classes
  - Improvement: perceptron of optimal stability $\equiv$ linear support vector machines

- When the data is not linearly separable
  - We may want to minimize the misclassification rate
  - But the perceptron learning algorithm won’t terminate
  - To fix this, one option is to use the pocket algorithm
    - Run the perceptron learning algorithm using random examples (with replacement)
    - If example incorrectly classified, update weight vectors as before
    - Keep track of (and ultimately output) the weight vector for which so far the largest number of random examples was correctly classified in a row
  - Pocket algorithm converges to optimum solution with probability 1
  - But it may take a long time find a good weight vector
Complexity of perceptron learning

- If the data is linearly separable
  - In P
  - E.g., solve the linear program

\[
\begin{align*}
\text{minimize} & \quad 0 \\
\text{subject to} & \quad \langle x_i, w \rangle \geq 0 \quad \text{for all } x_i \text{ in pos. class } (y_i = 1) \\
& \quad \langle x_i, w \rangle < 0 \quad \text{for all } x_i \text{ in neg. class } (y_i = 0)
\end{align*}
\]

- If the data is not linearly separable and we want to minimize misclassification rate
  - Finding an optimal weight vector is NP-hard (when dimensionality \( n \) is part of the input)
  - Remains NP-hard even when weights restricted to \( \{-1, 1\} \)
  - NP-hard to approximate even when weights restricted to \( \{-1, 1\} \)
  - Fortunately, we are often able to nevertheless find sufficiently good weights in practice
Perceptrons with multiple output units

Consider a perceptron with $m$ binary outputs for classification tasks.

1. **Multi-label classification**
   - Each input is associated with $m$ binary class labels
   - Goal is to predict each of them
   - E.g.: height (small/tall), hair color (light/dark), ...

2. **Multi-class classification** (first option)
   - Each input is associated with one out of $2^m$ class labels
   - We associate each label with one output vector of the perceptron
   - Problem: Which label with which output vector? (choice matters)

3. **Multi-class classification** (second option)
   - Each input is associated with one out of $m$ class labels
   - We associate each label with its indicator vector (one-hot encoding)
   - Problem: What if the network outputs less/more than a single 1?

More later.
An autoencoder with a binary threshold unit

- Consider the following feedforward autoencoder (with biases)

  ![Autoencoder Diagram]

  - $x_i^1, b_1, w_1$
  - $x_i^2, b_2, w_2$

- Assume that we want to minimize squared error

- What does this autoencoder compute?
Interpreting the weights (1)

- Suppose that we are given $b_1$ and $w_1$
- The binary threshold unit then acts as a linear classifier
Interpreting the weights (2)

Let’s now look at $\mathbf{b}_2$ and $\mathbf{w}_2$

- All points in class 0 are mapped to $c_0 = \mathbf{b}_2$
- All points in class 1 are mapped to $c_1 = \mathbf{b}_2 + \mathbf{w}_2$

Given $b$ and $\mathbf{w}_1$, what are the optimal choices of $c_0$ and $c_1$?

- Denote by $h_i$ the class of input $\mathbf{x}_i$
- Squared error is $\sum_i \sum_j (x_{ij} - \hat{x}_{ij})^2 = \sum_i \| \mathbf{x}_i - \mathbf{c}_{h_i} \|^2$
- Alternatively: $\sum_{i:h_i=0} \| \mathbf{x}_i - \mathbf{c}_0 \|^2 + \sum_{i:h_i=1} \| \mathbf{x}_i - \mathbf{c}_1 \|^2$
- For each class $k$, our goal is to minimize the squared Euclidean distance between the $\mathbf{x}_i$’s of the class and its representative $\mathbf{c}_k$
- Optimum solution is the mean of the examples of the class

$$\mathbf{c}_k = \frac{1}{\sum_{i:h_i=k} 1} \sum_{i:h_i=k} \mathbf{x}_i$$
Interpreting the weights (3)

- The overall optimum solution is

- Can you already see what our autoencoder does?
Interpreting the weights (4)

- Optimum solution agrees with $K$-means for $K = 2$
- $K$-means objective is to minimize the sum of squared distances

$$\arg\min_{C} \sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2,$$

where $\mu_k$ is the mean of the points in cluster $C_k$

- Given an optimal $K$-means clustering for $K = 2$
  - Each data point is associated to cluster of closest representative
  - We set $c_k = \mu_k$ (and thus obtain $b_2$ and $w_2$)
  - We set $b_1$ and $w_1$ such that the decision boundary is the set of points with equal distance to $\mu_1$ and $\mu_2$ (see previous slide)
  - The binary threshold unit then associates each point with its correct cluster
An autoencoder with multiple binary threshold units

- What happens if we have multiple binary threshold units?

- This autoencoder also “clusters” the data
  - Associates each data point with a “binary code” (00, 01, 10, 11)
  - Each codeword can be seen as a cluster ($2^r$ in total)

- For $r > 1$ binary threshold units, the optimum solution does not correspond to $K$-means anymore (with $K = 2^r$)
  - Can you see why?
  - More (perhaps) in exercise
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo

7. Summary
An FNN with a single logistic unit

- If we replace the binary threshold unit of a perceptron by a logistic unit, we obtain an FNN similar to a perceptron

![Diagram of a neural network with a single logistic unit and three inputs](Image)

- What’s the difference?
  - Fix some weight vector $\mathbf{w}$ (and ignore bias)
  - Above neural network outputs $\hat{y} \in [0, 1]$ with

$$
\hat{y} = \frac{1}{1 + \exp(-\langle \mathbf{w}, \mathbf{x} \rangle)} \begin{cases} 
< 0.5 & \langle \mathbf{w}, \mathbf{x} \rangle < 0 \\
\geq 0.5 & \langle \mathbf{w}, \mathbf{x} \rangle \geq 0
\end{cases}
$$

- Thus if we round the output of the logistic unit to the closest integer, we obtain output of the corresponding perceptron
- Logistic unit can be seen as a “smooth” version of a binary threshold unit
Smoothing

If we scale the weights by some constant $c > 0$, we change the degree of smoothing.
Binary classification

- Suppose we use the network for a binary classification task
  - Given a labeled set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ of input-output pairs
- We can minimize the misclassification rate of rounding
  \[ \sum_{i} |y_i - \text{round}(\hat{y}_i)| \]
  - Equivalent to perceptron
  - Ok for separable data
- Or maximize the log-likelihood of the provided labels
  \[ \ln \mathcal{L} = \sum_{i} [y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)] \]
  - Equivalent to logistic regression
  - Input $\langle w, x \rangle$ to logistic transfer function interpreted as estimate of log odds of positive class
  - Output $\hat{y}_i$ interpreted as confidence for positive class
  - Useful for non-separable case
Multi-class classification

Consider $C$ logistic outputs neurons for classification tasks.

![Diagram](https://via.placeholder.com/150)

- **Multi-class classification**
  - Each input is associated with one out of $C$ class labels
  - We associate each label with its indicator vector (one-hot encoding)
  - We may interpret output $\hat{y}_{ic}$ as confidence in label $c$ and predict the label with the largest confidence

- **Problems with this approach**
  - Outputs $\hat{y}_{ic}$ may not sum to one $\rightarrow \hat{y}_i$ is not a probability vector
  - There are redundant parameters $\rightarrow$ overparameterized model

- **Solution**: tie the output neurons appropriately
  $\rightarrow$ **softmax regression** $\equiv$ **multinomial logistic regression**
The softmax function \( S(\eta) \)

- The **softmax function** \( S(\eta) \)
  - Takes a real vector \( \eta = (\eta_1, \ldots, \eta_C)^T \in \mathbb{R}^C \)
  - And transforms it into an \( C \)-dimensional probability vector \( S(\eta) \)
    \[
    S(\eta)_c = \frac{\exp(\eta_c)}{\sum_{c'=1}^{C} \exp(\eta_{c'})}
    \]
  - Called this way because it exaggerates differences and acts somewhat like the max function (approximates indicator function of largest coefficient)

![Graphs showing softmax distribution at different temperatures](softmaxDemo2)

**Figure 4.4** Softmax distribution \( S(\eta/T) \), where \( \eta = (3, 0, 1) \), at different temperatures \( T \). When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is “spiky”, with all its mass on the largest element. Figure generated by softmaxDemo2.
The softmax function (2)

Here is a plot of $S(3, x_2, 1)_2$.

- When we fix all but one argument and look at the corresponding output, we obtain a shifted and scaled logistic function.
Logistic regression and softmax

- Recall logistic regression model

\[ P(Y_i = 0 \mid x_i, w) = \frac{1}{1 + \exp(\langle w, x \rangle)} \]
\[ P(Y_i = 1 \mid x_i, w) = \frac{1}{1 + \exp(-\langle w, x \rangle)} = \frac{\exp(\langle w, x \rangle)}{1 + \exp(\langle w, x \rangle)} \]

- We can express this with the softmax function

\[ P(Y_i = 0 \mid x_i, w) = S(0, \langle w, x \rangle)_1 \]
\[ P(Y_i = 1 \mid x_i, w) = S(0, \langle w, x \rangle)_2 \]

→ Can be seen as a generalization of the logistic function
Softmax regression

- For $C$ classes, softmax regression uses the model
  \[ P(Y = c \mid x, W) = S(\langle w_1, x \rangle, \ldots, \langle w_C, x \rangle)_c = S(W^T x)_c \]

- Set $\hat{y}_i = S(W^T x_i)$ to the so-obtained predicted probabilities

- Learning the parameters
  - The weight vectors are redundant (why?)
  - We get non-redundant parameters if we set $w_C = 0$
  - One option: maximum likelihood estimate
    - Equivalent: minimize cross-entropy loss $H(Y, \hat{Y})$
    - Equivalent: minimize KL divergence $D_{KL}(Y \| \hat{Y})$
  - Cond. likelihood is given by
    \[ L(y \mid X, W) = \prod_{i=1}^{N} P(Y_i = y_i \mid X, W) = \prod_{i=1}^{N} S(W^T x_i)_{y_i} \]
  - Gradient of cond. log-likelihood ($1 \leq c \leq C - 1$)
    \[ \nabla_{w_c^T} f(W) = \sum_i (\hat{y}_{ic} - I(y_i = c)) x_i^T \]
Multi-class classification (revisited)

- Using softmax regression for multi-class classification
  - Each input is associated with one out of $C$ class labels
  - We associate each label with its indicator vector (one-hot encoding)
  - We use softmax regression in the output layer
  - In a neural network, such a layer is called a **softmax layer**
  - We interpret output $\hat{y}_{ic}$ as confidence in label $c$
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo

7. Summary
Multi-layer FNNs

- We mostly looked at FNNs without hidden layers
  - These networks are limited in what they can do
  - Linear regression, logistic regression, ...
  - We can improve performance by engineering better features

- In neural networks, we generally want hidden layers
  - Recall: can interpret hidden layers as features for the next layer
  - By including hidden layers, we aim to let the network “do” the feature engineering
  - If there is at least one hidden layer, the network is called a **multi-layer FNN** (and sometimes also **multi-layer perceptron**)
  - If more than one, called **deep FNN**
How powerful are multi-layer FNNs?

- Template architecture of multi-layer FNNs
  - Hidden layers are sigmoidal (e.g., logistic neuron)
  - Output layer is linear (for regression) or sigmoidal (for classification)
  - Number and sizes of hidden layers are hyperparameters

- How powerful are such networks?
- Consider a basic multi-layer FNN with
  - $n$ inputs
  - One hidden layer with $r$ sigmoidal neurons
  - One linear output neuron

- **Universal approximation theorem**: This FNN can approximate any continuous function on $[0, 1]^n$ arbitrarily well, given sufficiently (but finitely) many hidden neurons [Cybenko, 1989]
Wide or deep?

- Universal approximation means that we can represent “any” function

- But that doesn’t mean that we can learn that function
  - Training methods may fail to find good parameterization
  - Overfitting may occur
  - Number of required units can be exponential in the input dimensionality

- Deep models often show better generalization performance
  - Encode belief that function to be learned involves a composition of several simpler functions
  - Interpretation: hidden unit output = factor of variation, which in turn depends on other factors of variation
  - Interpretation: hidden unit output = auxiliary intermediate values in a multi-step computation
Task: transcribe multi-digit numbers from photographs

Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow et al. (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million.
Training multi-layer FNNs

- How to train? (supervised)
- Need a suitable measure of error = loss function
  - Negative log-likelihood
  - Negative log-posterior
  - Squared error
  - Hinge loss
  - Misclassification rate
  - ... 
  - Also: incorporate priors / other measures to prevent overfitting (briefly discussed later)

- For point estimates: find weights that minimize loss
- Basic option: use gradient-based optimization method
  - Need to compute the gradient with respect to all weights, for both hidden and output units
  - This can be done via backpropagation (“backprop”)
  - Roughly: chain rule + reuse of computations
  - Once we have the gradients, can run GD, SGD, Adam, ...
Backpropagation (1)

- Let’s look at a simple case

![Diagram showing a simple neural network with one neuron in each layer: x → f(a1) → h1 → f(a2) → h2 → f(a3) → y]

- Every layer has just one neuron
  - 2 hidden layers, activation function \( f \) in each layer
  - Input to \( k \)-th layer: \( a_k \) (= activations)
  - Output of \( k \)-th layer: \( h_k = f(a_k) \) (= distributed representation)
  - Parameters: \( w_1, w_2, w_3 \)

- What does this network compute?

\[
\hat{y} = f(a_3) = f(w_3 h_2) = f(w_3 f(a_2)) = f(w_3 f(w_2 h_1)) = f(w_3 f(w_2 f(a_1))) = f(w_3 f(w_2 f(w_1 x)))
\]
Backpropagation (2)

- Let’s look at the gradient of the weight of the last layer

\[
\frac{\partial}{\partial w_3} \hat{y} = \frac{\partial}{\partial w_3} f(a_3) \quad \text{(chain rule)}
\]

\[
= \frac{\partial f(a_3)}{\partial a_3} \frac{\partial a_3}{\partial w_3}
\]

\[
= f'(a_3) \frac{\partial}{\partial w_3} w_3 h_2
\]

\[
= f'(a_3) h_2
\]
Let’s look at the gradient of the weight of the last-but-one layer

\[
\frac{\partial}{\partial w_2} \hat{y} = \frac{\partial}{\partial w_2} f(a_3) \\
= f'(a_3) \frac{\partial}{\partial w_2} w_3 h_2 \\
= f'(a_3) w_3 \frac{\partial}{\partial w_2} f(a_2) \quad \text{(chain rule)} \\
= f'(a_3) w_3 f'(a_2) \frac{\partial}{\partial w_2} w_2 h_1 \\
= f'(a_3) w_3 f'(a_2) h_1
\]
Let’s look at the gradient of the weight of the second-last layer

$$\frac{\partial}{\partial w_1} \hat{y} = \frac{\partial}{\partial w_1} f(a_3)$$

(chain rule)

$$= f'(a_3) \frac{\partial}{\partial w_1} w_3 h_2$$

$$= f'(a_3) w_3 \frac{\partial}{\partial w_1} f(a_2)$$

(chain rule)

$$= f'(a_3) w_3 f'(a_2) \frac{\partial}{\partial w_1} w_2 h_1$$

$$= f'(a_3) w_3 f'(a_2) w_2 \frac{\partial}{\partial w_1} f(a_1)$$

(chain rule)

$$= f'(a_3) w_3 f'(a_2) w_2 f'(a_1) \frac{\partial}{\partial w_1} a_1$$

$$= f'(a_3) w_3 f'(a_2) w_2 f'(a_1) x$$
Backpropagation (5)

- All together
  \[
  \frac{\partial}{\partial w_3} \hat{y} = f'(a_3) h_2 = \delta_3 h_2
  \]
  \[
  \frac{\partial}{\partial w_2} \hat{y} = f'(a_3) w_3 f'(a_2) h_1 = \delta_3 w_3 f'(a_2) h_1 = \delta_2 h_1
  \]
  \[
  \frac{\partial}{\partial w_1} \hat{y} = f'(a_3) w_3 f'(a_2) w_2 f'(a_1) x = \delta_2 w_2 f'(a_1) = \delta_1 x
  \]

- Observe: for layer $k$ (maps $h_{k-1} \xrightarrow{w_k} a_k \xrightarrow{f} h_k$)
  1. Take the $\delta_{k+1}$ value of subsequent layer $k+1$
  2. Multiply with the weights $w_{k+1}$ of the subsequent layer
  3. Multiply with the gradient of the activation function w.r.t. current input $a_k$ to obtain the $\delta_k$ value of current layer $k$
  4. Multiply $\delta_k$ with the outputs $h_{k-1}$ of previous layer $k-1$ to obtain the gradient w.r.t. $w_k$
Backpropagation (6)

- Backpropagation in a nutshell
  1. Forward pass to compute all outputs (forward propagation)
  2. Backward pass to compute all gradients
- The computation in both passes is local
  ▶ Only accesses information from the previous/current/next layer
  ▶ To do so, computed results are reused
- When we want the gradient of some loss function $L(\hat{y}, y)$
  ▶ Similar, one more application of the chain rule in the beginning
- When layers have more than one neuron
  ▶ Similar, but “multiplications” become matrix-vector products (involving Jacobians/weight matrices/outputs)
- When we have bias terms
  ▶ Similar, gradient of bias can also be computed from $\delta_{k+1}$
- When we have regularization terms
  ▶ As before, adapt gradients accordingly (e.g., weight decay)
Practical considerations

• Choice of activation function matters
  ▶ Sigmoidal units $\bigcirc$: gradient saturates when activation large
    $\Rightarrow$ Problem of **vanishing gradients**, esp. in early layers
  ▶ tanh units often preferable to logistic units
    $\Rightarrow$ Behave like a linear function for small activation
  ▶ Gradients of ReLU units $\bigcirc$ don’t saturate
    $\Rightarrow$ Often better performance
  ▶ Generally, a trial-and-error process

• Pretraining
  ▶ Try to start with useful weights (instead of random ones)
  ▶ Then use backprop for fine-tuning
  ▶ Example: **stacked autoencoders**: train an autoencoder greedily
    layer by layer ($= \text{learn } P(X)$), then use weights to initialize
    supervised learning ($= \text{learn } P(Y \mid X)$)
Overfitting

- Multi-layer FNNs are flexible → **overfitting** is serious concern
  - Universal approximation theorem tells us: with sufficiently many hidden neurons, FNN can perform arbitrarily well on the *training* set

- Some strategies against overfitting
  - Reduce the size of the neural network
  - Use regularization or max-norm constraints
  - Use **early stopping** during training (i.e., don’t train until convergence)
  - Deep over wide
  - Careful architecture engineering (e.g., non-fully connected layers)
  - Randomly perturb inputs to avoid learning “noise”
  - Randomly disable some units during training (**dropout**)

- Generally,
  - Need experience as guidance
  - Need computational resources for trial-and-error process
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo
   6.1 Distributed Word Representations
   6.2 Recurrent Neural Networks
   6.3 Convolutional Neural Networks

7. Summary
Overview

- The Neural Network Zoo is
  - Large
  - Diverse
  - Rapidly growing

- We do a short, high-level tour, focusing on popular architectures
  - Goal is to introduce you to some useful terms & techniques
  - Feedforward networks only

- Architecture engineering involves, for example,
  - Use of template models and parameter tying
  - Selection of which neurons are connected to each other
  - Selection/creation of (specialized types of) neurons
  - Dynamic restructuring of the network
  - Suitable feature representation of inputs
  - Training and data preprocessing/generation “tricks”
  - Number/size of hidden layers, priors, . . .
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo
   6.1 Distributed Word Representations
   6.2 Recurrent Neural Networks
   6.3 Convolutional Neural Networks

7. Summary
Distributed word representations (1)

- Consider a natural-language processing task
  - E.g., POS tagging, chunking, parsing, sentiment detection, spam detection, information extraction, ...
  - Suppose we operate at the word level
  - I.e., we use a vocabulary of $V$ distinct words

- Basic approach is to encode words using 1-of-$V$ encoding
  - I.e., one binary feature for every word in the vocabulary
  - $V$ features per input word $\rightarrow V$ parameters
  - Large number of parameters is problematic for learning when little training data available

- **Distributed word representations** (*word vectors*) map words to continuous representations
  - E.g., each word represented by an $R$-dimensional real-valued vector
  - Representation aims to provide similar/different representations for words that appear in similar/different contexts
  - Why helpful?: **distributional hypothesis** in linguistics states that words that occur in similar contexts tend to have similar meanings
  - Under this hypothesis: similar representation $\rightarrow$ similar meaning
Distributed word representations (2)

Top-30 closest word vectors to “God”, trained on the Bible
Distributed word representations (3)

- Distributed word representations for supervised learning
  - Use “meaning” of words, not words themselves
  - Trained models can generalize to unseen words
  - Significantly less features ($R \ll V$, typically $R=50–200$)

- Trained on large text corpora in unsupervised fashion
  - Many approaches: LSI, LDA, feedforward nets, recurrent nets, CBOW, skip gram, ...
  - Partly different objectives

- Here we look at one example: the continuous bag-of-words (CBOW) model
  - Hyperparameter $R$: size of word vector
  - Hyperparameter $W$: size of context window
**CBOW (1)**

Key idea: predict current word $w_t$ given its surrounding words (= context).

![Diagram of CBOW model](image-url)
CBOW (2)

- How to read this architecture?
- Input layer
  - $2W$ context words, each encoded using 1-of-$V$ encoding
  - Output: $\mathbf{w}_{t-W}, \mathbf{w}_{t-1}, \ldots, \mathbf{w}_{t+1}, \mathbf{w}_{t+W} \in \{0, 1\}^V$
- Projection layer
  - $R$ linear units per context word ($2WR$ in total)
  - Weight matrix $\mathbf{W} \in \mathbb{R}^{N \times R}$ shared across words $\rightarrow$ parameter tying
  - Outputs: $\mathbf{v}_{t-W}, \mathbf{v}_{t-1}, \ldots, \mathbf{v}_{t+1}, \mathbf{v}_{t+W} \in \mathbb{R}^R$, where $\mathbf{v}_i = \mathbf{W}^T \mathbf{w}_i = \text{row of } \mathbf{W} = \text{word vector of word } \mathbf{w}_i$
- Sum layer
  - $R$ linear units; weight matrix $\text{fixed}$ such that layer outputs $\mathbf{s} = \sum_i \mathbf{v}_i$
  - Note: $\sum_i \mathbf{w}_i = \text{word counts} \rightarrow \text{bag-of-words}$
  - Note: $\sum_i \mathbf{v}_i = \text{word vector sums} \rightarrow \text{“continuous” bag-of-words}$
  - Projection + sum layer thus output sum of word vectors of context words
- Output layer
  - Softmax ($V$ classes), predict $\mathbf{w}_t$
  - Parameterized by another weight matrix $\mathbf{W}' \in \mathbb{R}^{R \times V}$
Discussion

- **Intuition**
  - Network contains a bottleneck: $2WV$ inputs, $R$ units in sum layer
  - Bottleneck forces “compression” → can’t predict perfectly
  - Model performs well when context representation $s$ helps to predict current word $w_t$
  - According to the distributional hypothesis, similar words also occur with context $s \rightarrow s$ also helps to predict similar words
  - Since $s$ is composed of word vectors, the word vectors implicitly capture word meaning

- **In practice**, skip-gram model tends to work better
  - The other way around: given current word, predict context words
  - Order now matters

- **Training**
  - Slide window over text corpora, optimize loss (e.g., MLE)
  - Optionally: scale down weights for more distant context
  - Cheap, if it weren’t for the softmax layer
    → CBOW actually uses a “hierarchical softmax” layer
Softmax with many classes

- Consider a softmax layer with \( C \) outputs and input \( h \) from the previous layer. We have

\[
\hat{y}_c = S(W^T h + b)_c = \frac{\exp(\langle w_c, h \rangle + b_c)}{\sum_{c'} \exp(\langle w_{c'}, h \rangle + b_{c'})}.
\]

- During training
  - Need to compute \( P(Y = y \mid x, \theta) = \hat{y}_y \) (where \( y \) is true label)
  - To do so, we need all terms \( \exp(\langle w_{c'}, h \rangle + b_{c'}) \)
  - Gradient in backpropagation generally non-zero for all parameters

  - If \( C \) is small \( \rightarrow \) fine
  - If \( C \) is very large \( \rightarrow \) high cost
    - E.g., for word vectors \( C = V \) (e.g., Google News, \( V = 3M \))
    - When \( W = 2 \): Add 4 vectors to get hidden representation \( s \), compute 3M inner products to get softmax output
    - Common approach: use “approximate” softmax layer
Hierarchical softmax (1)

- **Hierarchical softmax** layer: arrange classes in a “decision tree”
  - Input is $h$
  - Leaves are classes
  - Interior vertices are decision points
  - Each possible choice associated with a probability
  - Probability of class = probability of corresponding path

- Example: two alternative trees for $C = 16$
Hierarchical softmax (2)

- To model the probability distribution over child nodes of each interior vertex, use softmax
  - One weight matrix and bias vector per interior node
  - Probability distribution over child nodes of \( v_i \) is \( S(\mathbf{W}_i^T \mathbf{h} + \mathbf{b}_i) \)

- For binary trees
  - Weight vector and (scalar) bias
  - Balanced tree with \( C \) leaves
    \( \rightarrow \) \( C - 1 \) interior nodes / weight vectors / bias terms
  - More when tree is unbalanced
Hierarchical softmax (3)

- Flat tree → Output is $S(W^T h + b)_c \equiv \text{softmax}$

  - Output depends on entire $W$ and $b$
    → $C$ weight vectors and bias terms

- Balanced tree
  - Probability of, say, leaf $y = 0101$ is

\[
S(0, w^T h + b)_0 \cdot S(0, w_0^T h + b_0)_1 \cdot S(0, w_{01}^T h + b_{01})_0 \cdot S(0, w_{010}^T h + b_{010})_1
\]

  - Only depends on 4 weight vectors and bias terms
  - In general: $C$ classes → $\log_2 C$ weight vectors / bias terms
  - E.g., for $C = 3M$ → $\log_2 C \approx 22$
  - Much more efficient to compute & all other weight vectors / bias terms have gradient 0
Hierarchical softmax (4)

- Choice of tree matters for prediction performance
  - Generally, regular softmax gives better predictions
  - Hierarchical softmax is able to produce good predictions if the classes in the “right” subtree are easy to discriminate from the classes of the “wrong” subtrees
  - E.g., for words: cluster words and recursively partition them into two clusters → hierarchical softmax can achieve similar prediction performance as regular softmax

- Choice of tree matters for training speed
  - Flat: as slow as softmax
  - Balanced: logarithmic cost
  - Ideal: Huffman tree based on class frequencies
    → Minimize expected path lengths (frequent classes → short path)

- Note: No/limited runtime improvement during prediction
  - Still need to compute all probabilities to get distribution over labels
  - MAP prediction can sometimes be made faster
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo
   6.1 Distributed Word Representations
   6.2 Recurrent Neural Networks
   6.3 Convolutional Neural Networks

7. Summary
Sequence labeling

- Recall the sequence labeling task
  - Input variables form a sequence
  - One output variable per input → also form a sequence

\[
\begin{array}{cccccc}
  i & 1 & 2 & 3 & 4 & 5 \\
  Y_i & DT & NN & VBD & DT & NN \\
  X_i & \text{The dog ate the cat} \\
\end{array}
\]

- How to solve this task with neural networks?
  - Can proceed similarly as in linear-chain CRFs
  - Key ingredients: template model + parameter tying
  - For neural networks, such models are called recurrent neural networks (RNNs)
Let’s try (1)

- Approach 1: fully connected, layered, deep FNN
  - From now on: each vertex corresponds to a set of neurons
  - Inputs encoded using 1-of-\( V \) encoding or word vectors
  - One or more hidden layers
  - Each hidden layer usually has multiple units per word
  - Output layer is softmax (one softmax unit per output label)

- Observe: Outputs not independent given \( x \) (but given \( h \))

- Problem
  - Extremely large number of parameters → infeasible to train
  - How to handle different input lengths?
Let’s try (2)

**RNN**

\[
\begin{array}{c}
y_t \\
W_2, b_2 \\
h_t \\
W_1, b_1 \\
x_t
\end{array}
\]

**Unfolded RNN**

\[
\begin{array}{c}
y \\
W_2, b_2 \\
h \\
W_1, b_1 \\
x
\end{array}
\]

\[
\begin{array}{c}
\text{The} \\
dog \\
ate \\
the \\
cat
\end{array}
\]

- Much simpler model
- Can handle multiple input lengths with *parameter tying*
  - Idea: share parameters for every “position”
  - That’s a *template model*: network is constructed by repeating the same template for every position
- Example of an RNN

**Problem**

- Outputs independent of each other (given input)
- Outputs only depend on their corresponding word \( \rightarrow \) looses context
Let’s try (3)

Let’s connect the hidden layer horizontally
- Idea: hidden units capture information about previous words and current word \(\rightarrow\) outputs become dependent
- Which information? \(\rightarrow\) let the network figure out during training
- Again, we use parameter tying

That’s a unidirectional RNN
- Problem: outputs during prediction independent of subsequent words (the “future”)
- Note: in linear-chain CRFs, where we connected outputs, this issue does not occur (why not?)
Let’s try (4)

- Let’s add a second hidden layer, but connect backwards
  - Idea: new hidden units capture information about subsequent words
- That’s a **bidirectional RNN**
  - Information about “past” and “future” captured in hidden units
Discussion

- No need for input/output at every position
- Think of hidden state as the “memory” of the network
- Widely used variant: LSTMs
  - Long Short Term Memory networks
  - RNNs that uses carefully designed structure to connect hidden units so that the can better remember the more distant past
  - Many variants have been proposed
- Feature engineering can be useful
  - E.g., for per-day time series: whether current day is a holiday, which weekday/month/season, ...
Example: RNNs for language modeling

- Language model = distribution over sequences of words
  - Can use RNN trained to predict next word
  - Can use RNN trained to predict next character

- **Fun results** with RNNs; e.g., write like Shakespeare:

  Second Senator:
  
  *They are away this miseries, produced upon my soul,*
  
  *Breaking and strongly should be buried, when I perish*
  
  *The earth and thoughts of many states.*

  DUKE VINCENTIO:
  
  *Well, your wit is in the care of side and that.*

  Second Lord:
  
  *They would be ruled after this chamber, and*
  
  *my fair nues begun out of the fact, to be conveyed,*
  
  *Whose noble souls I’ll have the heart of the wars.*
Example: Google Smart Reply (1)

**Turkey!**

dcorrado

to me

Hi all,
We wanted to invite you to join us for an early Thanksgiving on November 22nd, beginning around 2PM. Please bring your favorite dish! RSVP by next week.

Dave

**Server issues**

Dan Mané

to me

Hi team,
The server appears to be dropping about 10% of requests (see attached dashboards). There hasn't been a new release since last night, so I'm not sure what's going on. Is anyone looking into this?

...
Example: Google Smart Reply (2)

- Underlying architecture is an LSTM network
- Network first “reads” the input
  - Process is called **encoding**
  - Encodes the sentence into a continuous vector representation, known as the **thought vector**
- Network then “generates” the output
  - Process is called **decoding**
  - Note: outputs fed as input into next position
- Practical issues
  - Network tends to produce similar responses → filter those out
  - Network tends to generic responses → weigh those down
Attention

- An promising alternative to the encoder-decoder architecture is attention
  - Don't just look at final thought vector, but all hidden states
  - Key idea: let network focus it's “attention” on what’s important for its current output
  - E.g., use weighted combination of hidden states
Example: attention

Translation

Image captioning

A woman is throwing a frisbee in a park.
Outline

1. Introduction

2. Feedforward Neural Networks & Autoencoders

3. Perceptrons

4. Softmax regression

5. Multi-Layer FNNs

6. The Neural Network Zoo
   6.1 Distributed Word Representations
   6.2 Recurrent Neural Networks
   6.3 Convolutional Neural Networks

7. Summary
Convolutional Neural Networks

- Key applications
  - Computer vision (e.g., image recognition and classification such as identifying faces, objects, traffic signs)
  - Natural-language processing (e.g., sentence classification)

- Fully-connected networks not well-suited
  1. Too many features
  2. Don’t make use of spatial locality (e.g., must individually learn to detect an eye at different positions)

- Convolutional neural networks
  1. Fewer features (via sparse connections)
  2. Use spatial locality (via translation invariance of feature detectors)
Convolution (1)

- Consider data that is spatially arranged in a meaningful way
  - E.g., in a sequence (time series, sentences)
  - E.g., in a 2D grid (gray-scale images)
  - E.g., in a 3D grid (color images, 3rd dimension is color channel)
- Suppose we look at some local region of the data
  - E.g., a contiguous subsequence (3 days)
  - E.g., a small rectangle
  - Called receptive field
- And apply a linear operation (called filter) to it
  - Take all data points in the region, output one value
  - E.g., the average
- Convolution = apply operation to many local regions
  - Region is “shifted” over the data
  - E.g., a sliding window for sequences (shifted left/right)
  - E.g., a sliding rectangle for 2D grids (shifted left/right/up/down)
  - Filter always stays the same
  - Outputs can be arranged in the same shape as the data
Convolution (2)

- Consider a sliding window of size 3 over a sequence
  - Filter then takes a linear combination of 3 values
  - Can be represented by a $3 \times 1$ kernel (or basis) matrix containing the weights; e.g.,
    \[ K = \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \]
    corresponds to $(x_{t-1}, x_t, x_{t+1}) \rightarrow x_{t-1} - x_{t+1}$
  - In general, kernel matrix has same shape as receptive field

- To apply the filter, we shift the sliding window over the sequence and arrange the outputs corresponding to the window position

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>-2</td>
<td>-3</td>
<td>-6</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - Use zero padding at the borders
  - Here window shifted by one element $\rightarrow$ output size $=$ input size
  - In general, can be shifted multiple elements (stride)
    $\rightarrow$ output size $<$ input size
Convolution (3)

Identity

\[
\begin{pmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]

Box blur

\[
\frac{1}{9} \begin{pmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1
\end{pmatrix}
\]

Sharpen

\[
\begin{pmatrix}
0 & -1 & 0 \\
-1 & 5 & -1 \\
0 & -1 & 0
\end{pmatrix}
\]

Edge detection

\[
\begin{pmatrix}
0 & -1 & 0 \\
-1 & 4 & -1 \\
0 & -1 & 0
\end{pmatrix}
\]
Convolution layer

- We can represent a convolution as an FNN

- Weights are shared across all neurons (=kernel)
- Such a layer is called a **convolution layer**
  - We can think of the kernel as “feature detector”
  - Large output when (1) inputs where weights are large positive are large and (2) inputs where weights are large negative are small
  - The kernel is *learnt* → we learn a feature detector
  - Neurons are arranged in same shape as input

- Convolutional layers use non-linear neurons (tanh, ReLU)
- Note: parameter tying, sparse connections
  - Can be seen as a template
LeNet5

- **LeNet5** is a well-known early architecture (1998)
- Used for digit/letter classification

![LeNet5 architecture diagram](image)

**Subsampling layer**
- Reduces dimensionality by averaging (plus non-linearity)
- Convolution with non-overlapping receptive fields
  (here: $2 \times 2 \rightarrow$ size reduced by factor 4)
- (Scaled) tanh units almost everywhere
- Output computes distance to manually-crafted image (one per class) $\rightarrow$ radial basis function (RBF)

**Fig. 2.** Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.
Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are not samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1). Best viewed in electronic form.
Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are not samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1). Best viewed in electronic form.
Revolution of Depth

ImageNet Classification top-5 error (%)
Evolution (2)

- CNNs (for object detection) get deeper
  - But adding depth alone is insufficient and may actually hurt

- Many new techniques
  - Different ways to reduce/increase resolution (e.g., max-pooling, $1 \times 1$ convolutions)
  - Small networks process output of each convolution
  - GoogLeNet: Multiple different parallel paths (e.g., different sizes of receptive fields)
  - ResNet: Use skip edges & only compute residuals

- Preprocessing matters
  - Use multiple transformations of each example (e.g., crop/zoom/shift/flip/rotate)
    - “More” training data, higher robustness
  - Sometimes: also during testing
Outline

1. Introduction
2. Feedforward Neural Networks & Autoencoders
3. Perceptrons
4. Softmax regression
5. Multi-Layer FNNs
6. The Neural Network Zoo
7. Summary
Lessons learned

- **Artificial neural networks**
  - Useful for a variety of learning tasks, great results in some areas
  - Complex models, need data + compute + experience

- **Feedforward neural networks**
  - Discriminative models, directed flow from input to output
  - Hidden layers enable high representation power
  - Outputs of hidden layers can be seen as learned features (distributed representation)
  - Train with backprop + tricks

- **Autoencoder**
  - An ANN that predicts its own input
  - Useful in unsupervised, semi-supervised, and supervised learning (e.g., dimensionality reduction, clustering, pretraining)

- **Example ANNs**
  - Perceptron, softmax, CBOW, RNNs, CNNs
Suggested reading

- I. Goodfellow, Y. Bengio, A. Courville
  *Deep Learning* (Ch. 6)