Photogrammetry & Robotics Lab

Intro to Neural Networks
Part 2: Learning

Cyrill Stachniss

The slides have been created by Cyrill Stachniss.
In Part 1, We Discussed

- What are neurons and neural networks
- Activations, weights, biases
- Multi-layer perceptron (MLP)
- MLP for simple image classification
Part 2
Learning the Parameters
How to Make the Network Compute What We Want?

- Neural network is a recipe for performing a set of computations
- Structure and parameters are the design choices
- **How to set them?**
Network Parameters

Given a network structure, weights and biases tell the network what to do.

\[
\begin{align*}
\sigma \left( W^{(1)} a^{(0)} + b^{(1)} \right) &= a^{(1)} \\
\sigma \left( W^{(2)} a^{(1)} + b^{(2)} \right) &= a^{(2)} \\
\sigma \left( W^{(k)} a^{(k-1)} + b^{(k)} \right) &= a^{(k)}
\end{align*}
\]

\[\text{params} = \text{weights} \& \text{biases} \]

\[W^{(1)} b^{(1)} \ldots W^{(k)} b^{(k)}\]
What Means “Learning”?  

- NN are functions  
- The weights and biases determine what this function computes  
- Learning = determining parameters so that the network does what we want  
- Parameters are estimated by providing labeled examples (training data)
Our Handwritten Digit Network

- Input Layer: 784 raw pixels
- Hidden Layer 1: 128 simple patterns (relu)
- Hidden Layer 2: 64 combined patterns (relu)
- Output Layer: 10 patterns to digits (softmax)

[Image courtesy: Nielsen]
Compared to several other classifiers, the network parameters also include the feature computations. Thus, NNs can also learn the features!
Many Parameters

Such networks have many parameters!

\[
\begin{align*}
784 \times 128 & \quad 128 \times 64 \quad 64 \times 10 \\
128 & \quad 64 & \quad 10
\end{align*}
\]

= 109,386 parameters
# Training Through Labeled Data

\[ \{(x_i, y_i)\}_{i=1}^{I} \]
We want to have an approach that estimates the parameters by only providing examples of labeled data points!
Exploiting Training Examples

\[ x_i \rightarrow \hat{y}_i = f(x_i) \rightarrow y_i \]

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( \hat{y}_i = f(x_i) )</th>
<th>( y_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“0”</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>“1”</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>“2”</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>“3”</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>“4”</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>“5”</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>“6”</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>“7”</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>“8”</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>“9”</td>
<td>9</td>
</tr>
</tbody>
</table>
Loss Function

- We define a loss (= cost) function over all weights and biases of the network

\[ L(W, b) \rightarrow \mathbb{R} \]

- Computed using training data
- Input to \( L \) are the network parameters \( \theta = W, b \)
- Output is the error of the network with these parameters on the training data
Loss Function \( L_i(\theta) \rightarrow \mathbb{R} \)

\[ \mathcal{X}_i \quad \hat{y}_i = f(\theta, x_i) \quad y_i \quad \]

Compare output layer to the true label

\[ L_i(\theta) = \| \text{output}_i - \text{label}_i \|^2 \]
Loss Over All Examples

We need to evaluate the performance of the network over all examples

\[
L(\theta) = \frac{1}{I} \sum_{i} L_i(\theta) = \frac{1}{I} \sum_{i} \|f(\theta, x_i) - y_i\|^2
\]

{((5, 5), (2, 2), (9, 2), (9, 9), (8, 8), ...)

{(x_0, y_0), (x_1, y_1), (x_2, y_2), ...}
The Parameters We Want

- Parameter $\theta^*$ that minimize the sum of avg. squared losses over all examples

$$\theta^* = \arg \min_{\theta} L(\theta) = \arg \min_{\theta} \sum_{i} \| f(\theta, x_i) - y_i \|^2$$

- The squared loss is only one possible loss, several other options available

- **Goal:** Find the parameter vector $\theta^*$ for the labeled training set $\{(x_i, y_i)\}_{i=1}^{I}$ given the loss $L$
Let’s Start...

- Initialize parameters randomly
- See how well it performs (bad!)
- How to improve the parameters so that the loss decreases?

\[ L(\theta) \hookrightarrow \mathbb{R} \]

high dimensional

(109.386 dim)

That’s complex!
Loss Minimization using Gradient Descent

- Our problem looks like a non-linear least squares problem
- We have a lot of parameters, which makes using GN computationally tricky
- Gradient descent is a better way to perform the minimization
Gradient Descent in 1D

How to move?
Gradient Descent in 1D

How to move?
Exploit the gradient
Gradient Descent in 1D

Strategy:

\[
\frac{\partial L}{\partial \theta} > 0 : \quad \leftarrow \quad \text{(move left)}
\]

\[
\frac{\partial L}{\partial \theta} < 0 : \quad \rightarrow \quad \text{(move right)}
\]
Gradient Descent in 1D
Gradient Descent in 1D

\[ \nabla L = \frac{\partial L}{\partial \theta} \] tells us in which direction to go
Gradient Descent in 1D

Step by step:  \[ \theta^{(j+1)} = \theta^{(j)} - \lambda \nabla L|_{\theta^{(j)}} \]
Gradient Descent in 1D

- First derivative of loss: $\nabla L = \frac{\partial L}{\partial \theta}$
- Learning rate (small value): $\lambda = 0.01$

Gradient descent works by

- $\theta^{(0)} = \text{rand}$
- while (!converged)
  
  $\theta^{(j+1)} = \theta^{(j)} - \lambda \nabla L|_{\theta^{(j)}}$
Gradient Descent in 1D

Step by step: 
\[
\theta^{(j+1)} = \theta^{(j)} - \lambda \nabla L_{|\theta^{(j)}}
\]
Gradient Descent in 2D

- We can do the same in 2D
- Gradients are direction vectors
Gradient Descent in 2D

- We can do the same in 2D
- Gradients are direction vectors
Gradient Descent in 2D

- We can do the same in 2D
- Gradients are direction vectors
Gradient Descent in 2D

- We can do the same in 2D
- Gradients are direction vectors

\[ \nabla L = \left[ \frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2} \right]^T \]
Meaning of the Gradient Vector

- Some dimensions are more important than others to reduce the loss
- Gradient indicates which change leads to the fastest reduction of the loss

\[
\begin{align*}
\frac{\partial L}{\partial \theta_2} &\quad \frac{\partial L}{\partial \theta_1} \\
\theta_2 &\quad \theta_1
\end{align*}
\]

changes in $\theta_2$ are more relevant than changes in $\theta_1$ to reduce loss

\[
\begin{align*}
\theta_2 &\quad \theta_1
\end{align*}
\]

changes in $\theta_2$ have the same relevance to reduce loss
Gradient Descent in Higher Dimensional Spaces

- Same situation as before, but the gradient vector has more dimensions
- Update rule

\[ \theta^{(j+1)} = \theta^{(j)} - \lambda \nabla L \bigg|_{\theta^{(j)}} \]

In our classification example, all these vectors have 109.386 dimensions!
Keep in Mind...

Parameters are weights and biases.

\[ \theta \]

Difference between what we get and what we want:

\[ L_i(\theta) = \| f(\theta, x_i) - y_i \|^2 \]

Total loss is the averaged loss of all examples:

\[ L(\theta) = \frac{1}{I} \sum_i L_i(\theta) \]

We need to adjust the parameters to minimize the total loss!
Gradient Over All Examples

- Loss function sums over all examples

\[ L(\theta) = \frac{1}{I} \sum_i L_i(\theta) \]

- This means for the gradient

\[ \nabla L = \frac{1}{I} \sum_i \nabla L_i \]

We need to sum over all training examples and compute all gradients whenever we perform a single GD step!
Two Challenges

1. How to optimize the process if we have a lot of training examples?
2. How to compute the gradients for complex and nested functions?

We need to perform these operations often, so we need to be able to execute them efficiently!
1: Handling Large Training Sets

1st trick: Compute a gradient only on a small, sampled subset of examples
1: Handling Large Training Sets

1st trick: Compute a gradient only on a small, sampled subset of examples

□ = mini-batch to be used
1: Stochastic Gradient Descent

- 1\textsuperscript{st} trick: Compute a gradient only on a small, sampled subset of examples
- We \textbf{sample a mini-batch} in each step of gradient descent
- Use only mini-batch $B$ to compute

$$\nabla L = \frac{1}{|B|} \sum_{i \in B} \nabla L_i$$

- This \textbf{approximates} the real gradient
1: Stochastic Gradient Descent

Approximate down-hill steps but much faster to compute
2: Computing the Gradient

- $2^{nd}$ trick: Compute $\nabla L_i$ step by step
- Neuron activations are chains of activation functions and matrix-vector multiplications
- Many connections between the neurons
- Computing this 109.386 dimensional gradient can be tricky...

backpropagation algorithm
2: Backpropagation

- The idea is to break down the gradient computation into smaller steps

- Key ingredients of backpropagation:
  \[
  \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}
  \]
  chain rule
  local variables along paths through the NN
Backpropagation
Computational Graph

- Directed graph
- Nodes contain mathematical operations
- Edges encode input/output values
- Example:

```
+-------
|       |
|   x   |
|       |
+-------
  `--->
     |
     y

x + y
```
Function and Corresponding Computational Graph

\[ f(x, y, z) = (x + y) z^2 \]
Evaluating the Function

\[ f(x, y, z) = (x + y) z^2 \]

\[ f(2, -3, 2) = -4 \]
Add Local Variables

\[ f(x, y, z) = (x + y) \, z^2 \]

\[ a = x + y \]
\[ b = z^2 \]
\[ f = a \, b \]

forward pass
Computing the Gradient

- With the forward pass, we evaluate the function at a given point
- Next: compute the gradient of the function at the given point
- We can do this by traversing the graph backwards
- At each note, we compute the local derivative of the function w.r.t. the local inputs
Gradient At Point $[2, -3, 2]$

$\nabla f = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right]^\top$

$\nabla f\big|_{[2,-3,2]^\top} = ?$

backward pass
For a Single Node (Chain Rule)

\[ f = a \cdot b \]

\[ \frac{\partial L}{\partial a} \]

\[ \frac{\partial L}{\partial b} \]

\[ \frac{\partial L}{\partial f} \]

Assume to be given

to be computed

to be computed
For a Single Node (Chain Rule)

\[
\frac{\partial L}{\partial a} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial a} = db
\]

\[
\frac{\partial f}{\partial a} = b
\]

\[
f = ab
\]

\[
\frac{\partial L}{\partial f} = d
\]

\[
\frac{\partial L}{\partial b} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial b} = da
\]
For a Single Node (Chain Rule)

\[
\frac{\partial L}{\partial a} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial a} = db
\]

\[
f = ab
\]

\[
\frac{\partial f}{\partial a} = b
\]

\[
\frac{\partial f}{\partial b} = a
\]

\[
\frac{\partial L}{\partial f} = d
\]
For a Single Node (Chain Rule)

\[
\frac{\partial L}{\partial a} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial a}
= d \cdot b
= 1 \cdot 4 = 4
\]

\[
\frac{\partial f}{\partial a} = b
f = a \cdot b
\]

\[
\frac{\partial f}{\partial b} = a
\]

\[
\frac{\partial L}{\partial f} = b
\]

\[
\frac{\partial L}{\partial b} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial b}
= d \cdot a = 1 \cdot (-1) = -1
\]
For a Single Node (Chain Rule)

\[ \frac{\partial L}{\partial a} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial a} \]

\[ = d \cdot b \]

\[ = 1 \cdot 4 = 4 \]

\[ \frac{\partial f}{\partial a} = b \]

\[ \frac{\partial f}{\partial b} = a \]

\[ f = a \cdot b \]

\[ \frac{\partial L}{\partial f} \]

\[ \frac{\partial L}{\partial b} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial b} \]

\[ = d \cdot a = 1 \cdot (-1) = -1 \]
Backward Pass

\[
\frac{\partial f}{\partial f} = 1
\]

known variable to derive for
Backward Pass

\[ f = a \cdot b \]

\[ \frac{\partial f}{\partial a} = b \]

\[ \frac{\partial f}{\partial b} = a \]

\[ \frac{\partial f}{\partial f} = 1 \]
Backward Pass

\[
\frac{\partial f}{\partial a} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial a} = 1 \frac{\partial f}{\partial a} = b = 4
\]

\[
\frac{\partial f}{\partial a} = b
\]

\[
f = ab
\]

\[
\frac{\partial f}{\partial f} = 1
\]
Backward Pass

Function:

\[ f = ab \]

\[ \frac{\partial f}{\partial b} = a \]

\[ \frac{\partial f}{\partial f} = 1 \]

\[ \frac{\partial f}{\partial b} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial b} = 1 \frac{\partial f}{\partial b} = a = -1 \]
Backward Pass

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial x} = 4 \frac{\partial a}{\partial x} = 4
\]

function

\[
a = x + y
\]

\[
\frac{\partial a}{\partial x} = 1
\]

\[
\frac{\partial a}{\partial y} = 1
\]
Backward Pass

\[
\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial y} = 4 \frac{\partial a}{\partial y} \quad 4 = 4
\]

\[
\frac{\partial f}{\partial a} = 4
\]

\[
\frac{\partial a}{\partial x} = 1
\]

\[
\frac{\partial a}{\partial y} = 1
\]
Backward Pass

\[ f = b \]

\[ b = z^2 \]

\[ \frac{\partial b}{\partial z} = 2z \]

\[ \frac{\partial f}{\partial z} = \frac{\partial f}{\partial b} \frac{\partial b}{\partial z} = (-1) \cdot \frac{\partial b}{\partial z} = -2z = -4 \]
Backward Pass

\[ \nabla f \bigg| _{[2, -3, 2]^\top} = \begin{bmatrix} 4 \\ 4 \\ -4 \end{bmatrix} \]
Backpropagation (Backprop)

- Approach is called **backpropagation** and computes the gradient of any function expressed as such a graph.
- Combines **chain rule** and local variables for the graph nodes.
- Recursively traverses the graph.
- Can be used for computing both, numerical and analytical gradient computation.
Example for One Neuron

\[ a = \sigma \left( \sum w_n a_n + b \right) \]

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

sigmoid activation
Example for One Neuron with Two Incoming Activations

\[ a(x) = \frac{1}{1 + \exp(- (w_0 a_0 + w_1 a_1 + b))} \]
Forward Pass

\[
a(x) = \frac{1}{1 + \exp(-(w_0 a_0 + w_1 a_1 + b))}
\]
Forward Pass

\[ a(x) = \frac{1}{1 + \exp(-(w_0 a_0 + w_1 a_1 + b))} \]
Backward Pass

\[ \frac{\partial a}{\partial a} = 1 \]
Backward Pass

\[
\begin{align*}
& w_0 \
& a_0 \
& w_1 \
& a_1 \
& b \\
& 2 \quad -1 \quad -2 \
& -3 \quad 6 \quad 4 \
& -3 \quad -2 \
& -3 \\
& + \
& 1 \
& -1 \\ + \
& \exp \\
& +1 \
& \frac{1}{x} \\
& 1 \\
& 0.37 
\end{align*}
\]

Local function:

\[
f(x) = \frac{1}{x}
\]

\[
\frac{\partial f}{\partial x} = \frac{-1}{x^2}
\]

\[
\frac{-1}{x^2} = \frac{-1}{1.37^2} = -0.53
\]
Backward Pass

local function

\[ f(x) = x + 1 \]

\[ \frac{\partial f}{\partial x} = 1 \]
Backward Pass

\[
\begin{align*}
\frac{\partial f}{\partial x} &= \exp(x) \\
\end{align*}
\]

local function

\[
\begin{align*}
\frac{\partial f}{\partial x} &= \exp(x) \\
\end{align*}
\]

\[
\begin{align*}
\exp(x) &= -0.53 \cdot \exp(-1) = -0.2
\end{align*}
\]
Backward Pass

\[ f(x) = -x \]
\[ \frac{\partial f}{\partial x} = -1 \]
Backward Pass

local function
\[ f(x, y) = x + y \]
\[ \frac{\partial f}{\partial x} = 1 \]
\[ \frac{\partial f}{\partial y} = 1 \]

\[
\begin{align*}
0.2 & \times 1 = 0.2 \\
0.2 & \times 1 = 0.2
\end{align*}
\]
Backward Pass

Local function

$$f(x, y) = x + y$$

$$\frac{\partial f}{\partial x} = 1$$

$$\frac{\partial f}{\partial y} = 1$$
**Backward Pass**

Local function:
\[ f(w, a) = w \cdot a \]
\[ \frac{\partial f}{\partial w} = a \]
\[ \frac{\partial f}{\partial a} = w \]

**Example Calculation**

\[ 0.2 \cdot a = 0.2 \cdot (-1) = -0.2 \]
\[ 0.2 \cdot w = 0.2 \cdot 2 = 0.4 \]
Backward Pass

\[ f(w, a) = w \cdot a \]
\[ \frac{\partial f}{\partial w} = a \]
\[ \frac{\partial f}{\partial a} = w \]

\[
0.2 \cdot a = 0.2 \cdot (-2) = -0.4 \\
0.2 \cdot w = 0.2 \cdot (-3) = -0.6
\]
Backward Pass

\[ \nabla \sigma \bigg|_{[2, -1, -3, -2, -3]^\top} = \begin{bmatrix} -0, 2, 0.4, -0.4, -0.6, 0.2 \end{bmatrix}^\top \]
The example illustrated that we can compute the gradient for a neuron.

We can also model the sigmoid using a single node ("sigmoid gate").

\[ \sigma(x) = \frac{1}{1 + \exp(-x)} \]
\[ \frac{\partial \sigma(x)}{\partial x} = (1 - \sigma(x)) \cdot \sigma(x) \]
Backpropagation from Neuron to Neuron

- Recursive application from neuron to neuron through the layers
- For multiple outgoing edges, we need to compute the sum of gradients

- Backpropagation also generalizes directly to multivariate function
Neural Networks As Computation Graphs

- We can use BP for computing the gradient of the loss function $L(\theta)$
- We add a loss layer
Quadratic Loss for Backpropagation

training data

$y_i$

$x_i$

$\theta$

parameters

$NN$

$\hat{y}_i$

$\| \cdot \|^2$

$L_i$
Backpropagation (Backprop)

- BP allows us to compute gradients of nested and complex functions
- Combines chain rule and local variables
- Recursive traversal of the network
- Forward pass computes the linearization point for the gradient
- Backward pass computes the gradient
Learning a Neural Network

Repeat until convergence

1. Sample mini-batch from training data
2. Run backprop to compute gradient for SGD using mini-batch
   \[ \nabla L|_{\theta^{(j)}} \]
3. Execute SGD step to find better parameters reducing the loss
   \[ \theta^{(j+1)} = \theta^{(j)} - \lambda \nabla L|_{\theta^{(j)}} \]

Return parameter vector
Convolutional Neural Networks

In image-related learning tasks, CNNs play an important role.
Convolutional Neural Networks

In image-related learning tasks, CNNs play an important role.
Convolutional Neural Networks

In image-related learning tasks, CNNs play an important role.

[Image courtesy: van Veen]
Convolutional Neural Networks

In image-related learning tasks, CNNs play an important role.
What Is “Deep Learning”? 

“Learning neural networks with many hidden layers”

Definition: #hidden layers > 2

Very deep networks today have up to 150 hidden layers (still growing...)

Summary – Part 2

- Leaning multi-layer perceptrons
- Parameters are the weights and biases
- Learning = estimate weights & biases
- Minimization of a loss (cost) function
- Gradient descent for parameter optimization
- Backpropagation to compute gradients
- End-to-end: no manual features
- CNN for image processing
Literature & Resources

- Online Book by Michael Nielsen, Chapter 1:  
  http://neuralnetworksanddeeplearning.com/chap1.html
- Nielsen, Chapter 1, Python3 code:  
  https://github.com/MichalDanielDobrzanski/DeepLearningPython
- MNIST database:  
  http://yann.lecun.com/exdb/mnist/
- Grant Sanderson, Neural Networks  
  https://www.3blue1brown.com/
- Online book by Deisenroth, Faisal, Ong:  
  Mathematics for Machine Learning  
  https://mml-book.github.io/
- Alpaydin, Introduction to Machine Learning
- Standford AI Lectures by Li et al.
Slide Information

- The slides have been created by Cyrill Stachniss as part of the photogrammetry and robotics courses.
- **I tried to acknowledge all people from whom I used images or videos. In case I made a mistake or missed someone, please let me know.**
- Huge thank you to Grant Sanderson (3blue1brown) for his great educational videos that influenced this lecture.
- Thanks to Michael Nielsen for his free online book & code as well as to Fei-Fei Li et al. for the Stanford AI lectures.
- If you are a university lecturer, feel free to use the course material. If you adapt the course material, please make sure that you keep the acknowledgements to others and please acknowledge me as well. To satisfy my own curiosity, please send me email notice if you use my slides.

Cyrill Stachniss, cyrill.stachniss@igg.uni-bonn.de