Neural Networks: A Deep Introduction

Sanjay Arora
Ulrich Drepper
Machine Learning: A Quick Overview
Why and how does something work?

Inputs \rightarrow f : \text{Inputs} \rightarrow \text{Outputs} \rightarrow \text{Outputs}

System under Study

**Question**: what are the rules governing the system i.e. what is f?
The Scientific Method

Most successful approach till now:

- Observe system with different inputs and measure outputs.
- Guess rules or guess \( f \)
- Make predictions from guessed rules
- Compare predictions to outputs
- If predictions “match” outputs, you \textbf{might} have the correct rules
- If predictions don’t match outputs, you are definitely wrong.
The Scientific Method

• A scientific theory can never be *proven* to be correct. There’s no proof the sun will rise tomorrow – just an overwhelming possibility.

• Comparison of the predictions and outputs is more subtle than it appears. It depends on the degree of accuracy required. Rules might predict outputs within 10% but not within 1% and that might be sufficient to explain the core elements of the system under study.

• Simplicity is key. The best (in terms of predictive power) scientific theories tend to be simple.
How to guess the rules?

Many ways – see Einstein, Dirac, Feynman’s work for some examples

One way:
Collect data and scan for patterns. How Kepler found the laws for planetary motion
What is Machine Learning?

- Collect data and find patterns to discover rules.
- Let computers scan through several guesses/hypotheses and find the best ones.
- Need to invent algorithms that can scan multiple hypotheses efficiently (in time, memory, amount of data required).
Some terminology

- Model: our guess of the rules
- Learning: the process of scanning the data to find the best rules
- Supervised Learning: learning when data has labels (feedback) i.e. we have both the inputs and the outputs
- Unsupervised Learning: learning when data doesn’t have labels (no feedback)
- Reinforcement Learning: learning when feedback is infrequent over time.
Why Probability?
Why Probability?

Probability theory is a central component of machine learning. Why?

Input 1  →  System under Study  →  Outputs
Input 2  →  
Input N  →  

$f : \text{Inputs} \rightarrow \text{Outputs}$
Why Probability?

Don’t know all inputs!!!

N can be large

Some inputs might have small effects on the outputs and get ignored

Inputs might have high degree of correlation but still some independent components
Why Probability

Suppose have system with N independent inputs

\[ x_1, x_2, \ldots, x_N \]

Output (measured) is y

\[ y = f(x_1, x_2, \ldots, x_N) \]

**Goal**: Find this function
Why Probability

Since $N$ is large, we might only measure the dominant inputs

$$y \approx g(x_1, x_2, x_3)$$

Instead of:

$$y = f(x_1, x_2, \ldots, x_N)$$

**Goal:** Find this function
Why Probability

Expect if there are exactly 3 controlled inputs

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>

But the other N-3 inputs are not controlled and might vary

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3 (6/10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 (3/10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 (1/10)</td>
</tr>
<tr>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>7 (3/10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9 (5/10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6 (2/10)</td>
</tr>
</tbody>
</table>

Multiples values! Probabilities!!
Why Probability

**Incomplete knowledge/Ignorance leads to probabilities**

Ignorance can be self-imposed: give up unnecessary detail and simplify system at the cost of introducing probabilities
What is a neural network?
Scanning algorithms

Recall

Need to invent algorithms that can scan multiple hypotheses efficiently (in time, memory, amount of data required)
Scanning algorithms

Many approaches over the last few decades including well-known algorithms

- Decision trees
- Random forests
- Gradient boosted trees
- Linear regression
- Gaussian processes
- Logistic regression
- K-means, spectral etc, clustering
- Neural Networks
- ...
Why focus on neural networks?

Spectacular success in vision and language tasks

Very flexible learners

Interesting open research problems

New progress in learning structure of data (GANs), combinatorial optimization, visualization of data etc.
What is a neural network

Loosely inspired by neurons in the brain
What is a neural network

input > threshold → output = 1
input <= threshold → output = 0
What is a neural network

multiple inputs

input, \( i = w_1 i_1 + w_2 i_2 + \ldots + w_n i_n \)

input > threshold \( \rightarrow \) output = 1
input \( \leq \) threshold \( \rightarrow \) output = 0
What is a neural network

Combine basic neurons into a network

Inputs

$\mathbf{i}_1$

$\mathbf{i}_2$

$\mathbf{i}_3$

$\mathbf{i}_4$

Outputs

$\mathbf{o}_1$

$\mathbf{o}_2$

$\mathbf{o}_3$

$\mathbf{o}_4$

Data Flow Direction
What is a neural network

Feed-forward architecture

Inputs

\(i_1\)
\(i_2\)
\(i_3\)
\(i_4\)

Output Layer

\(o_1\)
\(o_2\)
\(o_3\)
\(o_4\)
Why go through this trouble?

Why is any of this useful? Can’t I just use linear regression please?
Why go through this trouble?: Universal Approximation Theorem

Loose version

A (feed-forward) neural network with one hidden layer can approximate any “reasonable” function, $f$, to arbitrary accuracy.
Why go through this trouble?: Universal Approximation Theorem

**Precise version**

\[
\sigma : \mathbb{R} \to \mathbb{R}
\]

\(\sigma\) non-constant, bounded, continuous

\[
f : [0, 1]^n \to \mathbb{R}
\]

\(f\) continuous on \([0, 1]^n\)
Why go through this trouble?: Universal Approximation Theorem

**Precise version**

\[
\sigma : \mathbb{R} \to \mathbb{R} \quad f : [0, 1]^n \to \mathbb{R}
\]

... can find \( w_i, u_i, b_i, N \)

\[
\left| f(x) - \sum_{i=1}^{N} w_i \sigma(u_i^T x + b_i) \right| < \epsilon
\]

function to learn \quad \text{approximation to } f

Important Technicality: can extend to any compact subset for the domain
Why go through this trouble?: Universal Approximation Theorem

What does this mean for me?
Since we can approximate any reasonable function to an arbitrary degree of accuracy, given enough inputs and corresponding outputs, we can create a highly accurate approximation to the function generating the outputs from inputs.
Why go through this trouble?: Universal Approximation Theorem

Two caveats

Need non-linear activation function, $\phi$ in our neural network (we’ll see what this means shortly)

$N$ refers to the number of neurons (nodes) in the hidden layer. This grows exponentially as $\epsilon$ decreases
Structure of a node (neuron)

\[ i_1 \quad i_2 \quad \cdots \quad i_n \]

\( \sigma \) is a non-linear "activation" function
Linear vs Non-linear

Linear: \( \sigma(x) = C_1 x + C_2 \)

Composition of two linear functions is linear

\[
f(x) = C_1 x + C_2 \quad g(x) = D_1 x + D_2
\]

\[
g(f(x)) = D_1 f(x) + D_2 = D_1 (C_1 x + C_2) + D_2
\]

\[
g(f(x)) = (D_1 C_1) x + (D_1 C_2 + D_2)
\]

\[
\rightarrow \quad E_1 + E_2
\]
Linear vs Non-linear

Non-linear

\[ \log(x) \quad e^x \quad x^2 \quad \sqrt{x} \]

infinitely more

Most complex systems behave non-linearly
Structure of a node (neuron)

\[ o = \sigma(w_1 i_1 + w_2 i_2 + \ldots + w_n i_n + b) \]

Short-hand: \[ o = \sigma(w^T i + b) \]

\[ i = \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad w^T = [w_1 \ w_2 \ \ldots \ w_n] \]

b is called bias or intercept
Activation Functions

What choices can we make?

- Non-linear
- Simple to evaluate

Will need derivatives so hopefully derivatives easy to evaluate
Activation Functions

Binary switch

Derivative not defined at $x = 0$
(not a major issue)

Derivative not informative

$\sigma'(x) = 0$ everywhere except at $x=0$
Activation Functions

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

Continuous switch

\[ \sigma'(x) = \sigma(x)(1 - \sigma(x)) \]

Derivatives in terms of activation itself
Activation Functions
Activation Functions
Activation Functions
Activation Functions
Activation Functions

Many more choices

Choices dictated by:
- Issues during training time (saturation of nodes)
- Type of predictions
Story till now

- Start with simple neuron
- Needed to add non-linear activation to be able to learn/mimic any reasonable function
- Looked at a bunch of activations and added to our neuron
- Time to put neurons together into a network and see how prediction would work.
What is a neural network

Every edge has a “weight”

Input: \((i_1, i_2, i_3, i_4)\)

Hidden:
\[
\begin{align*}
a_1 &= \sigma(w_1^T i + b_1) \\
a_2 &= \sigma(w_2^T i + b_2) \\
a_3 &= \sigma(w_3^T i + b_3)
\end{align*}
\]

Output:
\[
o = v_1 a_1 + v_2 a_2 + v_3 a_3 + c = v^T a + c
\]
What is a neural network

Input: \((i_1, i_2, i_3, i_4)\)

Hidden:
\[
\begin{align*}
a_1 &= \sigma(w_1^T i + b_1) \\
a_2 &= \sigma(w_2^T i + b_2) \\
a_3 &= \sigma(w_3^T i + b_3)
\end{align*}
\]

Output:
\[
ox = \sigma(v_1 a_1 + v_2 a_2 + v_3 a_3 + c) = \sigma(v^T a + c)
\]

Concise matrix notation
\[
ox = V\sigma(Wi + b) + c
\]

\(W, V\) are matrices of weights
\(b, c\) are vectors of biases
The game in a nutshell

Start with labeled data

elephant
dog
snow leopard

penguin
octopus
The game in a nutshell

Pick network architecture (more details later)
The game in a nutshell

Make predictions (forward propagation)

\[
o = V \sigma (W i + b) + c
\]

a number weight activation weight input bias bias
The game in a nutshell

Make predictions (forward propagation)

\[ o = V \sigma (W i + b) + c \]

a number weight activation weight input bias bias

Compare output, o to actual value, v
The game in a nutshell

Make predictions (forward propagation)

\[ o = V \sigma (W i + b) + c \]

a number weight activation weight input bias bias

Compare output, \( o \) to actual value, \( v \)

Loss: \[ L = \frac{1}{2} (o - v)^2 \]

= 0 if \( o = v \) (prediction matches value)

larger the worse predictions get
The game in a nutshell

Compare output, $o$ to actual value, $v$

$$L = \frac{1}{2} (o - v)^2$$

$= 0$ if $o = v$ (prediction matches value)

larger the worse predictions get

Tweak $W$, $V$, $b$ and $c$ to adjust $o$
and minimize $L$

Minimize: $L = L(V, W, b, c)$
The game in a nutshell

Predict (Forward propagation)

\[ o = V \sigma(Wi + b) + c \]

\[ o = \text{polar bear} \]

\[ v = \text{bunny} \]

\[ L(o, v) \]

Adjust \( V, W, b, c \)

Minimize \( L \)
Loss Functions
Loss functions

Measuring deviation between predictions and targets/labels

Each example is treated independently:

\[
\text{Loss, } \mathcal{L} = \sum_{i=1}^{n} C[\hat{y}_i, y_i]
\]

Cost for example i
Loss functions

\[ \mathcal{L} = \sum_{i=1}^{n} C[\hat{y}_i, y_i] \]

Cost for example i

Choice of \( C \) dictated by problem type

\( C \) bounded by below i.e. \( C \geq 0 \)
Loss functions: Regression

Mean Squared Error

\[ C[\hat{y}, y] = (\hat{y} - y)^2 \]

Any deviation from \( y \) is punished
Loss functions: Binary Classification

Target: 0 or 1

Prediction of belonging to class 1: $p \in [0, 1]$

How to measure deviation?
Loss functions: Binary Classification

Pick a threshold, say 0.5

Convert $p$ to label:

$p \leq 0.5 \rightarrow \text{prediction} = 0$

$p > 0.5 \rightarrow \text{prediction} = 1$

Accuracy: % of predictions correct???
Loss functions: Binary Classification

Accuracy: % of predictions correct???

Problems:

Dependent on threshold

Not a smooth function of predictions → indirect control of weights
Loss functions: Binary Classification

\[ C = \left( y - p \right)^2 \]

0 or 1  \( \in [0,1] \)

Unnatural to compare class label to probability

Penalty bounded above by 1 \( \rightarrow \) too mild
Loss functions: Binary Classification

Given $p, y$: what’s the likelihood of getting $y$?

$$p^y(1 - p)^{1-y}$$

$y = 0: p^0(1 - p)^{1-0} = 1 - p$

$y = 1: p^1(1 - p)^{1-1} = p$
Loss functions: Binary Classification

All samples are independent

Total likelihood: $$\prod_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1-y_i}$$

Predict $$p_i$$ that maximize this likelihood, $$\mathcal{L}$$
Loss functions: Binary Classification

\[ \mathcal{L} = \prod_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1-y_i} \]

Maximizing \( \mathcal{L} \) equivalent to maximizing \( \log \mathcal{L} \)
(log is a monotonically increasing function)

\[ \log \mathcal{L} = \sum_{i=1}^{n} y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \]
Loss functions: Binary Classification

Usually have algorithms to minimize functions

Maximizing $f$ equivalent to minimizing $-f$

Minimize:

$$\mathcal{L}^\prime = - \log \mathcal{L} = - \sum_i^n y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$
Loss functions: Binary Classification

\[ \mathcal{L} = -\sum_{i}^{n} y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \]

For one example, if \( y = 1 \): \( \mathcal{L} = -\log(p_i) \)

\( y = 0 \): \( \mathcal{L} = -\log(1 - p_i) \)
Loss functions: Binary Classification

\[ y = 1 : \mathcal{L} = - \log(p_i) \]

\[ p_i = 1 \text{(correct)} \mathcal{L} = - \log(1) = 0 \]

\[ p_i = 0 \text{(incorrect)} \mathcal{L} = - \log(0) \rightarrow \infty \]
Loss functions: Binary Classification

\[ y = 0 : \mathcal{L} = - \log(1 - p_i) \]

\[ p_i = 0 \text{(correct)} \rightarrow \mathcal{L} = - \log(1 - 0) = 0 \]

\[ p_i = 1 \text{(incorrect)} \rightarrow \mathcal{L} = - \log(1 - 1) \rightarrow \infty \]
Loss functions: Multi-class Classification

\[ \mathcal{L} = - \sum_{i=1}^{n_{\text{examples}}} y_{i1} \log(p_{i1}) + y_{i2} \log(p_{i2}) + \ldots + y_{in} \log(p_{in}) \]

\[ \mathcal{L} = - \sum_{i=1}^{n_{\text{examples}}} \sum_{j=1}^{n} y_{ij} \log(p_{ij}) \]

Note: \( p_{i1} + \ldots + p_{in} = 1, y_{i1} + \ldots + y_{in} = 1, y_{ij} \in 0, 1 \)
Loss functions: Custom design

You can design one based on your problem

Well-designed loss greatly aids learning/training

Example: Kullback-Leibler divergence for comparing two distributions
Why do neural networks work? - Intuition
Simple Problem

Binary Classification: Given \((x, y)\), predict class 0 or 1
Simple Problem

Logistic Regression

Compute: $\sigma(x, y) = \frac{1}{1 + e^{-(ax + by + c)}}$

Probability of belonging to class 1
Simple Problem

\(a, b, c\) found by minimizing cross-entropy loss

\[
\mathcal{L} = -\sum_{i=1}^{n} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))
\]

Label \(y_i = 0\) or \(y_i = 1\)

Prediction: \(p_i = p_i(a, b, c) \in [0, 1]\)
Simple Problem

Positive $x \rightarrow 1$ (on)

Activation Function: sigmoid

$\sigma(ax + by + c)$

Negative $x \rightarrow 0$ (off)
Simple Problem

Class 1 (100%)

Probability

Class 0 (0%)

Activation Function: sigmoid

50% → Decision boundary
Simple Problem

Class 1 (100%)

Probability

Class 0 (0%)

\[ ax + by + c = 0 \]

50% → Decision boundary
Simple Problem

\[ \sigma(ax + by + c) = 50\% \implies ax + by + c = 0 \]

Straight Line

\[ ax + by + c = 0 \implies y = -\frac{c}{b} - \frac{a}{b}x \]
Simple Problem

No straight line can separate class 0 from class 1

Not \textit{linearly separable}
Simple Problem

No straight line can separate class 0 from class 1

One solution:
Convert to polar coordinates

\[ r^2 = x^2 + y^2 \]

What about high dimensional data?
Simple Problem

Minimize: Cross-entropy

Activation = Sigmoid
Simple Problem

\[a = \sigma(w_1 x + w_2 y + b)\]

Decision boundary:
\[w_1 x + w_2 y + b = 0\]
Simple Problem

\[ a = \sigma(w_3x + w_4y + c) \]

Decision boundary:
\[ w_3x + w_4y + c = 0 \]

Input Layer  Hidden Layer  Output Layer
Simple Problem

Decision boundaries for each node (1) in hidden layer

Hidden Nodes = 1
Simple Problem

Hidden Nodes = 2
Simple Problem

Hidden Nodes = 3
Simple Problem

Hidden Nodes = 5
Simple Problem

Hidden Nodes = 10
Simple Problem: Alternate View

Hidden Nodes = 1
Hidden Nodes = 2

Node = 0.5

Node = 1

Node = 0

C = 0

C = 1

Decision boundaries for each node (2) in hidden layer
Decision boundaries for each node (2) in hidden layer

Hidden Nodes = 2

Node = 0.5

Node = 0.5

D = 0

Node = 0

Node = 0.5

Node = 0.5

D = 1
Hidden Nodes = 2

Decision boundaries for each node (2) in hidden layer

C=1, D=0
C=0, D=0
C=0, D=1
Split input space into 3 regions

$C = 0, D = 0 \quad \rightarrow \quad \text{Mixed - both class 0 and class 1}$

$C = 1, D = 0 \quad \rightarrow \quad \text{Pure class 1}$

$C = 0, D = 1 \quad \rightarrow \quad \text{Pure class 1}$
N hidden binary nodes can split input into $2^N$ regions

Keep increasing N (without overfitting) till regions ”pure”
Map 2-dim input to N-dim space

\[(x, y) \rightarrow (a_1, a_2, \ldots, a_N) = (1, 0, \ldots, 1)\]

corners of N-dim hypercube

in the hope of linearly separating classes
Map 2-dim input to 2-dim space

C=1, D=0

C=0, D=0

C=0, D=0

C=1, D=0

C=0, D=0

C=0, D=1
Map 2-dim input to 2-dim space

C=1, D=0

C=0, D=1

C=0, D=0

C=1, D=0
Map 2-dim input to 3-dim space

Isolate Red in corner
A second example: Regression

Given $x$, predict $y = \sin(x)$
Minimize:

Mean Squared Error

Activation = Sigmoid
Output = Linear
\[ a = \sigma(w_1 x + b) \]

Decision boundary:

\[ w_1 x + b = 0 \implies x = -\frac{b}{w_1} \]
Decision boundary:

\[ w_2 x + c = 0 \quad \Rightarrow \quad x = -\frac{c}{w_2} \]
1 decision boundary per hidden node

\[ \delta_1 = -\frac{b_1}{w_1} \]

\[ \delta_2 = -\frac{b_2}{w_2} \]

\[ \vdots \]

\[ \delta_n = -\frac{b_n}{w_n} \]
Walk from left to right

Each time you cross a boundary, a hidden node either:
• Turns on i.e. goes from 1 → 0
• Turns off i.e. goes from 0 → 1
\[
\begin{align*}
\begin{bmatrix}
1 \\
0 \\
0 \\
1
\end{bmatrix}
\rightarrow 1 \cdot u_1 + 0 \cdot u_2 + 0 \cdot u_3 + 1 \cdot u_4 + d &= \boxed{u_1 + u_4 + d}
\end{align*}
\]
$u_1 + u_4 + d$

\[
\begin{bmatrix}
1 \\
0 \\
0 \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
1 \\
0 \\
1 \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
0 \\
1 \\
1 \\
0
\end{bmatrix}
\rightarrow
\begin{bmatrix}
0 \\
1 \\
1 \\
0
\end{bmatrix}
\]

$\delta_3 \rightarrow \delta_1 \rightarrow \delta_4 \rightarrow \delta_2 \rightarrow x$
\[ u_1 + u_3 + u_4 + d \]

\[ u_1 + u_4 + d \]

\[
\begin{bmatrix}
1 \\
0 \\
0 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
1 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
1 \\
0
\end{bmatrix}
\begin{bmatrix}
0 \\
1 \\
1 \\
0
\end{bmatrix}
\]

\[ \delta_3 \quad \delta_1 \quad \delta_4 \quad \delta_2 \quad x \]
\[ u_1 + u_3 + u_4 + d \]

\[ u_1 + u_4 + d \]

\[ u_3 + u_4 + d \]

\[
\begin{bmatrix}
1 \\
0 \\
0 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
1 \\
0 \\
1 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
1 \\
1 \\
1
\end{bmatrix}
\]
\[ u_1 + u_3 + u_4 + d \]

\[ u_1 + u_4 + d \]

\[ u_3 + u_4 + d \]

\[ u_3 + d \]
\[ u_1 + u_3 + u_4 + d \]

\[ u_2 + u_3 + d \]

\[ u_1 + u_4 + d \]

\[ u_3 + d \]

\[
\begin{bmatrix}
1 \\
0 \\
0 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
1 \\
0 \\
1 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
0 \\
1 \\
1
\end{bmatrix}
\]
Five distinct regions where NN predicts constant values with sigmoid turn on and turn off
Hidden Nodes = 2
Hidden Nodes = 3
Hidden Nodes = 4
Hidden Nodes = 10
Hidden Nodes = 2
Hidden Nodes = 3
Hidden Nodes = 10
Hidden Nodes = 100
Mathematical Optimization
Optimization: what is it?

Mathematical and numerical techniques to find the maximum or minimum of a function
Optimization: some terminology

Global maximum: the maximum value the function takes across its domain

Local maximum: the maximum value of the function in a small neighborhood around an x (input value)

Similar definitions for minima
Optimization: some terminology

Will only talk about minima from now on
Optimization: why do we care?

**Training**: Finding weights that minimize the loss function on the test set (“out of sample”) for a neural network is an optimization problem.

**Hyperparameter tuning**: Finding the number of layers, the activation function, the number of nodes in each layer, and other parameters is also an optimization problem.
Optimization: obstacles

- $f$ might be computationally expensive to evaluate.
- $f$ might be discrete so no way to evaluate derivatives which can guide search for minima.
- Even if $f$ is continuous and well-behaved, evaluating higher derivatives (second, third etc.) is very expensive.
- $f$ might have very complex structure with multiple (possibly infinite) local minima.
- $f$ might be very high-dimensional i.e. it has a large number of inputs and hence we are searching for the minima in a high-dimensional space with many more directions to explore.
Gradient Descent

$f(x)$

$slope < 0$
move right

$slope > 0$
move left

$x_L$

$x_*$

global minimum

$x_R$
Gradient Descent

- Iterative method
- Start at some reasonable point and keep updating

\[ x^{(t+1)} = x^{(t)} + \text{update} \]
\[ x^{(t+1)} = x^{(t)} - \eta \frac{df}{dx}(x^{(t)}) \]
\[ \eta = \text{learning rate} \]
Gradient descent: Intuition and Examples
Gradient descent: Intuition and Examples
Gradient descent: Intuition and Examples

Gradient descent

function

\( \ell = 0.8, \text{start} = -5 \)
Gradient descent: Intuition and Examples
Gradient descent: Intuition and Examples

Gradient descent

\[ f(x) \]

function

lr=0.05, start=-0.5

\[ x \]

-4 -2 0 2 4
Gradient descent: Intuition and Examples
Gradient descent: Intuition and Examples
Gradient descent: Intuition and Examples

- Convergence very sensitive to learning rate
  - Learning rate too small:
    - Will converge to local minimum
    - Might take a long time
  - Learning rate too large:
    - Will bounce around or even escape valley one started in
    - If function decreases gradually, will take a long time to converge.
- Obvious question:
  - Can learning rate be adaptive i.e. change in response to location?
Gradient descent: Convex Convergence

Every smooth function can be locally approximated as a quadratic (convex) function

\[ f(x) = f(x_{\text{min}}) + f'(x_{\text{min}})(x - x_{\text{min}}) + \frac{1}{2} f''(x_{\text{min}})(x - x_{\text{min}})^2 + O((x - x_{\text{min}})^3) \]

Taylor expansion around local minimum

Function of one variable
Gradient descent: Convex Convergence

Every smooth function can be locally approximated as a quadratic (convex) function

\[ f(x, y) = f(x_{\text{min}}, y_{\text{min}}) + \frac{\partial f}{\partial x}(x - x_{\text{min}}) + \frac{\partial f}{\partial y}(y - y_{\text{min}}) + \frac{1}{2} \frac{\partial^2 f}{\partial x^2}(x - x_{\text{min}})^2 + \frac{1}{2} \frac{\partial^2 f}{\partial y^2}(y - y_{\text{min}})^2 + \frac{\partial^2 f}{\partial x \partial y}(x - x_{\text{min}})(y - y_{\text{min}}) + O(\text{cubic}) \]

\[ f(x, y) = a + b(x - x_{\text{min}})^2 + d(y - y_{\text{min}})^2 + g(x - x_{\text{min}})(y - y_{\text{min}}) \]

Taylor expansion around local minimum

All derivatives evaluated at \((x_{\text{min}}, y_{\text{min}})\)
Gradient descent: Convex Convergence

Every smooth function can be locally approximated as a quadratic (convex) function

\[ f(x, y) = a + b(x - x_{min})^2 + d(y - y_{min})^2 + g(x - x_{min})(y - y_{min}) \]

Change of coordinates: \( x' = x - x_{min}, y' = y - y_{min} \)

\[ f(x, y) = a + \begin{bmatrix} x' & y' \end{bmatrix} \begin{bmatrix} b & g\epsilon \\ g(1 - \epsilon) & d \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} \]

\[ \epsilon = \frac{1}{2} \rightarrow \text{diagonalize symmetric matrix} \rightarrow f(x'', y'') = a + \alpha x''^2 + \beta y''^2 \]
Gradient descent: Convex Convergence

\[ f(x) = a + \frac{b}{2}(x - c)^2 \]

Start point: \( x^{(0)} \)

Update: \( x^{(t)} = x^{(t-1)} - \eta \frac{df}{dx}(x^{(t-1)}) \)

\[
\boxed{x^{(t)} = x^{(t-1)} - \eta b(x^{(t-1)} - c)}
\]
Gradient descent: Convex Convergence

\[ x^{(t)} = x^{(t-1)} - \eta b (x^{(t-1)} - c) \]

Minimum at: \( x = c \)

Consider Error: \( \epsilon_t \equiv |x^{(t)} - c| \)

\[ |x^{(t)} - c| = |x^{(t-1)} - \eta b (x^{(t-1)} - c) - c| = |(x^{(t-1)} - c)| |1 - \eta b| \]

\[ \epsilon_t = \epsilon_{t-1} |1 - \eta b| \]
Gradient descent: Convex Convergence

$$
\epsilon_t = \epsilon_{t-1} |1 - \eta b|
$$

$$
\epsilon_t = \epsilon_0 |1 - \eta b|^t
$$

Distance from minimum at time $t$ (ideally close to 0)  
Distance from minimum at time 0
Gradient descent: Convex Convergence

$$\epsilon_t = \epsilon_0 |1 - \eta b|^t \quad 0 < 1 - \eta b < 1$$

Question: how many steps till we get “close” to minimum at c?

Suppose we want $$\frac{\epsilon_t}{\epsilon_0} = \delta$$, a small number

$$t = \frac{\log \delta}{\log |1 - \eta b|} \approx -\frac{\log \delta}{\eta b} \quad \eta b \text{ small}$$
Backpropagation: Derivatives for Gradient Descent
Backpropagation: Calculating derivatives

- Gradient descent and its variants need the first derivatives:

  Minimize: \( f(x_1, x_2, \ldots, x_n) \)

  Initial guess: \( (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}) \)

  Update: \( x_i^{(t+1)} = x_i^{(t)} - \eta \frac{\partial f}{\partial x_i}(x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)}) \)
Backpropagation: Calculating derivatives

- Neural network has cost or loss function that depends on weights:
  \[ C[\vec{w}] = C[w_1, w_2, \ldots, w_n] \]

- Need to compute:
  \[ \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n} \]
Backpropagation: Toy example

Input

$\mathbf{w}_{01} \quad \mathbf{w}_{12} \quad \mathbf{w}_{23}$

$\mathbf{x}_0 \rightarrow \mathbf{x}_1 \rightarrow \mathbf{x}_2 \rightarrow \mathbf{x}_3$

Output

Forward propagation:

$x_1 = \mathbf{w}_{01} x_0$

$x_2 = \mathbf{w}_{12} x_1 \implies x_3 = \mathbf{w}_{23} \mathbf{w}_{12} \mathbf{w}_{01} x_0$

$x_3 = \mathbf{w}_{23} x_2$
Backpropagation: Toy example

\[ x_3 = w_{23}w_{12}w_{01}x_0 \]

Ignore for now: we’ll generalize this in a moment
Backpropagation: Toy example

\[
\begin{align*}
\text{Cost} &= \frac{1}{2} (x_3 - y)^2 \\
x_3 &= \text{prediction, } y = \text{label/actual value/target}
\end{align*}
\]
Backpropagation: Toy example

\[ \text{Cost} = \frac{1}{2} (x_3 - y)^2 \]

Question: what are \( \frac{\partial C}{\partial w_{01}} \), \( \frac{\partial C}{\partial w_{12}} \), \( \frac{\partial C}{\partial w_{23}} \)?
Backpropagation: Toy example

\[ C = \frac{1}{2} \left( w_{23} w_{12} w_{01} x_0 - y \right)^2 \]

\[
\frac{\partial C}{\partial w_{01}} = (x_3 - y) w_{23} w_{12} x_0 = (x_3 - y) w_{23} w_{12} w_{01} x_0
\]

\[
\frac{\partial C}{\partial w_{12}} = (x_3 - y) w_{23} w_{01} x_0 = (x_3 - y) w_{23} w_{12} w_{01} x_0
\]

\[
\frac{\partial C}{\partial w_{23}} = (x_3 - y) w_{12} w_{01} x_0 = (x_3 - y) w_{23} w_{12} w_{01} x_0
\]
Backpropagation: Toy example

Define forward chains:

\[ x_0 \]
\[ w_{01} x_0 = x_1 \]
\[ w_{12} w_{01} x_0 = x_2 = w_{12} x_1 \]
\[ w_{23} w_{12} w_{01} x_0 = x_3 = w_{23} x_2 \]

\[ (x_3 - y) = \Delta_0 \]
\[ (x_3 - y) w_{23} = \Delta_1 = w_{23} \Delta_0 \]
\[ (x_3 - y) w_{23} w_{12} = \Delta_2 = w_{12} \Delta_1 \]
\[ (x_3 - y) w_{23} w_{12} w_{01} = \Delta_3 = w_{01} \Delta_2 \]
Backpropagation: Toy example

\[ \frac{\partial C}{\partial w_{01}} = (x_3 - y)w_{23}w_{12} w_{01}x_0 = \Delta_2 x_0 \]

\[ \frac{\partial C}{\partial w_{12}} = (x_3 - y)w_{23} w_{12} w_{01}x_0 = \Delta_1 x_1 \]

\[ \frac{\partial C}{\partial w_{12}} = (x_3 - y)w_{23} w_{12}w_{01}x_0 = \Delta_0 x_2 \]
Backpropagation: what’s so great about this?

Forward propagation: calculate $x_i$

Backward propagation: calculate $\Delta_i$

Calculations iterative: just multiply by weights
Backpropagation: Non-linear toy example

Input

$p_0 \xrightarrow{\sigma_0} q_0 \xrightarrow{w_{01}} p_1 \xrightarrow{\sigma_1} q_1 \xrightarrow{w_{12}} p_2 \xrightarrow{\sigma_2} q_2 \xrightarrow{w_{23}} p_3 \xrightarrow{\sigma_3} q_3$

Output

Layer 0: $p_0 \quad q_0 = \sigma_0(p_0)$
Layer 1: $p_1 = w_{01}q_0 \quad q_1 = \sigma_1(p_1)$
Layer 2: $p_2 = w_{12}q_1 \quad q_2 = \sigma_2(p_2)$
Layer 3: $p_3 = w_{23}q_2 \quad q_3 = \sigma_3(p_3)$

$q_3 = \sigma_3 \left[ w_{23} \sigma_2 \left[ w_{12} \sigma_1 \left[ w_{01} \sigma_0(p_0) \right] \right] \right]$
Backpropagation: Toy example

\[
\text{Cost} = \frac{1}{2} (q_3 - y)^2
\]

\(q_3 = \text{prediction}, y = \text{label/actual value/target}\)
Backpropagation: Toy example

\[
\text{Cost} = \frac{1}{2} (q_3 - y)^2
\]

Question: what are \( \frac{\partial C}{\partial w_{01}} \), \( \frac{\partial C}{\partial w_{12}} \), \( \frac{\partial C}{\partial w_{23}} \)?
Backpropagation: Toy example

\[ C = \frac{1}{2} \left( \sigma_3 \left[ w_{23} \sigma_2 \left[ w_{12} \sigma_1 \left[ w_{01} \sigma_0 \left[ p_0 \right] \right] \right] \right] - y \right)^2 \]

\[ \frac{\partial C}{\partial w_{23}} = \begin{cases} \frac{q_3}{q_2} \end{cases} \]

\[ \frac{\partial C}{\partial w_{12}} = \begin{cases} \frac{q_3 - y}{q_2} \sigma'_3 (p_3) \sigma'_2 (p_2) \end{cases} \]

\[ \frac{\partial C}{\partial w_{01}} = \begin{cases} \frac{q_3 - y}{q_2} \sigma'_3 (p_3) \sigma'_2 (p_2) \sigma'_1 (p_1) \end{cases} \]

\[ q_0 \]
Backpropagation: Toy example

Define forward chains:

\[ q_0 \]
\[ q_1 = \sigma_1(p_1) = \sigma_1(w_{01}q_0) \]
\[ q_2 = \sigma_2(p_2) = \sigma_2(w_{12}q_1) \]
\[ q_3 = \sigma_3(p_3) = \sigma_3(w_{23}q_2) \]

Define backward chains:

\[ (q_3 - y)\sigma'_3(p_3) = \Delta_0 \]
\[ (q_3 - y)\sigma'_3(p_3)w_{23}\sigma'_2(p_2) = \Delta_1 = w_{23}\sigma'_2(p_2)\Delta_0 \]
\[ (q_3 - y)\sigma'_3(p_3)w_{23}\sigma'_2(p_2)w_{12}\sigma'_1(p_1) = \Delta_2 = w_{12}\sigma'_1(p_1)\Delta_1 \]
\[ (q_3 - y)\sigma'_3(p_3)w_{23}\sigma'_2(p_2)w_{12}\sigma'_1(p_1)w_{01}\sigma'_0(p_0) = \Delta_3 = \sigma'_0(p_0)w_{01}\Delta_2 \]
Backpropagation: Toy example

\[ \frac{\partial C}{\partial w_{01}} = (x_3 - y)w_{23}w_{12} w_{01} x_0 = \Delta_2 x_0 \]

\[ \frac{\partial C}{\partial w_{12}} = (x_3 - y)w_{23} w_{12} w_{01} x_0 = \Delta_1 x_1 \]

\[ \frac{\partial C}{\partial w_{12}} = (x_3 - y)w_{23} w_{12} w_{01} x_0 = \Delta_0 x_2 \]
Backpropagation: what’s so great about this?

Forward propagation: calculate $x_i$
Backward propagation: calculate $\Delta_i$
Calculations iterative: just multiply by weights
Conclusion: Next Steps
Learn through applying

**Computer Vision**: image classification, object detection, segmentation, pose estimation, captioning and many more

**Language Processing and Understanding**: Generating language, translation, time-series prediction

**Unsupervised learning**: Auto-encoders, Restricted Boltzmann Machines, Generative Adversarial Networks

**Traditional regression, classification, recommendation engine problems**
Learn through open-source tools

Recommend **PyTorch** or **TensorFlow** both for use and understanding

Highly recommend implementing your own architectures (at least once) in plain python + numpy or C++ or CUDA
Learn through experiments

Pick simple toy examples like the ones here and explore what the neural network is doing, how the cost varies, what the weight space looks like etc.

This kind of empirical approach leads to many useful ideas for further investigation
Learn through mathematics

If you are mathematically inclined, look at work by Max Tegmark, Sanjeev Arora, John Hopfield (and many others)

Look at papers/tutorials published at ICML, ICLR, NIPS

But also, understand everything from first principles. Ask how you would re-discover something after you read it in a paper.
Contribute to Research

Very active area with low barriers to entry.

Jump in, keep an open mind and ask fundamental questions.