Deep Learning for Computer Vision

Neural Networks: A Review

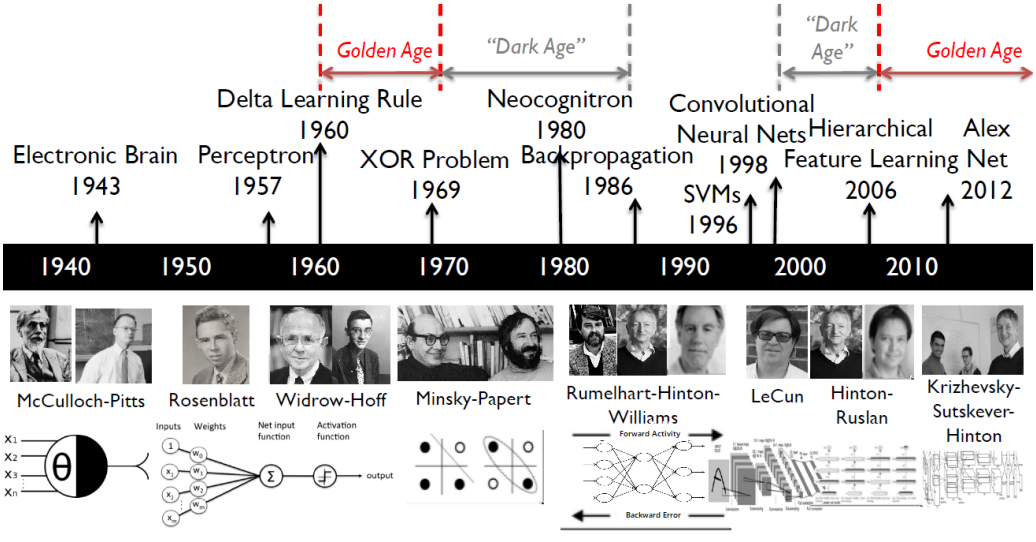
Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Acknowledgements

- Most content of this lecture are based on **Lecture 2** of **CS7015** course taught by Mitesh Khapra at IIT-Madras

History of Neural Networks



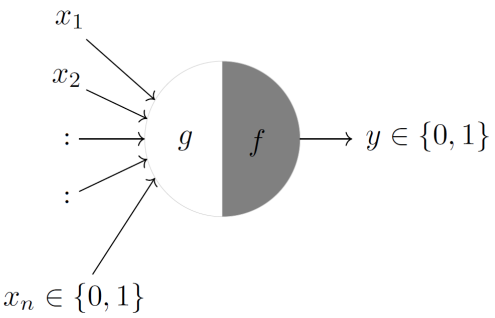
McCulloch-Pitts Neuron

- McCulloch (neuroscientist) and Pitts (logician) proposed a highly simplified computational model of the neuron (1943)
- g aggregates the inputs and the function f takes a decision based on this aggregation
- The inputs can be excitatory or inhibitory
- $y = 0$ if any x_i is inhibitory, else

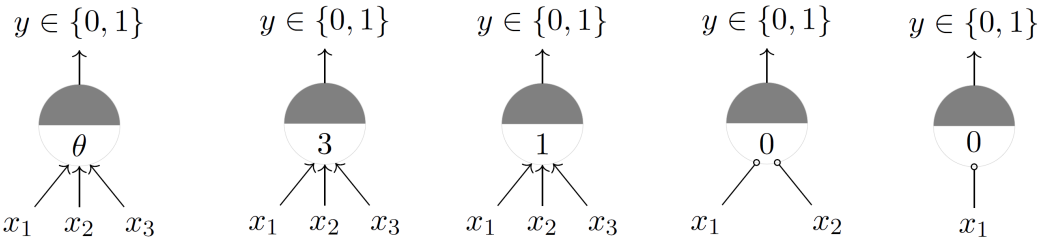
$$g(x_1, x_2, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$y = f(g(\mathbf{x})) = 1 \; if \; g(\mathbf{x}) \geq \theta$$
$$= 0 \; if \; g(\mathbf{x}) < \theta$$

- θ is a thresholding parameter



McCulloch-Pitts Neuron



- Feedforward MP networks can compute any Boolean function $f : \{0,1\}^n \to \{0,1\}$
- Recursive MP networks can simulate any Deterministic Finite Automaton (DFA) (See this paper[1] for more information)

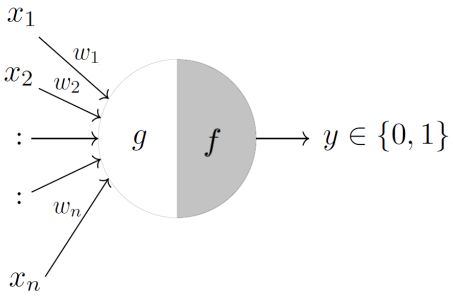
[1] Forcada, Neural Networks: Automata and Formal Models of Computation, 2002

Perceptrons

- Frank Rosenblatt, an American psychologist, proposed the perceptron model (1958)
- Later refined and carefully analyzed by Minsky and Papert (1969)
- A more general computational model than McCulloch-Pitts neurons
- Inputs are no longer limited to boolean values

$$g(x_1, x_2, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} w_i * x_i$$

$$y = f(g(\mathbf{x})) = 1 \; if \; g(\mathbf{x}) \geq \theta$$
$$= 0 \; if \; g(\mathbf{x}) < \theta$$



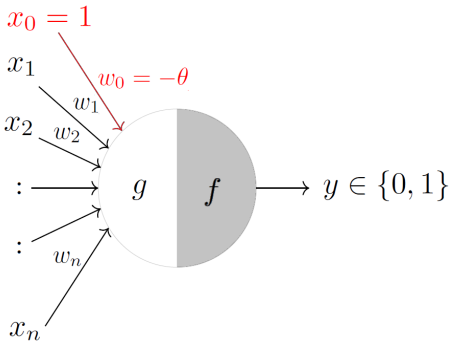
Perceptrons

- Frank Rosenblatt, an American psychologist, proposed the perceptron model (1958)
- Later refined and carefully analyzed by Minsky and Papert (1969)
- A more general computational model than McCulloch-Pitts neurons
- Inputs are no longer limited to boolean values

- A more accepted convention

$$g(x_1, x_2, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=0}^{n} w_i * x_i$$

$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \ if \ g(\mathbf{x}) \geq 0 \\ &= 0 \ if \ g(\mathbf{x}) < 0 \\ where \ x_0 &= 1 \ and \ w_0 = -\theta \end{aligned}$$



Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$

$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$

$P \leftarrow$ input with labels 1;

$N \leftarrow$ input with labels 0;

Initialize w randomly;

while *!convergence* **do**

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$

$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$

$P \leftarrow$ input with labels 1;

$N \leftarrow$ input with labels 0;

Initialize w randomly;

while *!convergence* **do**

    Pick random $\mathbf{x} \in P \cup N$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$

$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$

$P \leftarrow$ input with labels 1;

$N \leftarrow$ input with labels 0;

Initialize w randomly;

while *!convergence* **do**

  Pick random $\mathbf{x} \in P \cup N$

  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$

$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$

$P \leftarrow$ input with labels 1;

$N \leftarrow$ input with labels 0;

Initialize w randomly;

while *!convergence* **do**

  Pick random $\mathbf{x} \in P \cup N$

  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**

    $\mathbf{w} = \mathbf{w} + \mathbf{x}$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$

$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$

$P \leftarrow$ input with labels 1;

$N \leftarrow$ input with labels 0;

Initialize w randomly;

while *!convergence* **do**

  Pick random $\mathbf{x} \in P \cup N$

  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**

    $\mathbf{w} = \mathbf{w} + \mathbf{x}$

  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$
  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- Why would this work?

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$
  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- Why would this work?
- We are interested in finding the line $\sum_{i=0}^{n} w_i * x_i = 0$ or $\mathbf{w}^T\mathbf{x} = 0$ which divides the input space into two halves (binary classifier)

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$
  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- Why would this work?
- We are interested in finding the line $\sum_{i=0}^{n} w_i * x_i = 0$ or $\mathbf{w}^\intercal \mathbf{x} = 0$ which divides the input space into two halves (binary classifier)
- Every point ($\mathbf{x}$) on this line satisfies the equation $\mathbf{w}^\intercal \mathbf{x} = 0$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$
  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- Why would this work?
- We are interested in finding the line $\sum_{i=0}^{n} w_i * x_i = 0$ or $\mathbf{w}^T\mathbf{x} = 0$ which divides the input space into two halves (binary classifier)
- Every point $(\mathbf{x})$ on this line satisfies the equation $\mathbf{w}^T\mathbf{x} = 0$
- What can you tell about the angle (α) between w and any point $(\mathbf{x})$ which lies on this line?

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$
  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- Why would this work?
- We are interested in finding the line $\sum_{i=0}^{n} w_i * x_i = 0$ or $\mathbf{w}^\intercal \mathbf{x} = 0$ which divides the input space into two halves (binary classifier)
- Every point $(\mathbf{x})$ on this line satisfies the equation $\mathbf{w}^\intercal \mathbf{x} = 0$
- What can you tell about the angle (α) between w and any point $(\mathbf{x})$ which lies on this line?
- The angle is 90° ($\because \cos\alpha = \frac{\mathbf{w}^\intercal \mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}$)
- Since the vector w is perpendicular to every point on the line it is actually perpendicular to the line itself

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$?

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal \mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal \mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$? Less than 90°

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal \mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal \mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$? Less than 90°
- What will be the angle between vector $\mathbf{x} \in N$ and $\mathbf{w}$?

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$? Less than 90°
- What will be the angle between vector $\mathbf{x} \in N$ and $\mathbf{w}$? Greater than 90°
- Ponder and convince yourself this is the case

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$? Less than 90°
- What will be the angle between vector $\mathbf{x} \in N$ and $\mathbf{w}$? Greater than 90°
- Ponder and convince yourself this is the case
- For $\mathbf{x} \in P$ if $\mathbf{w}^\intercal\mathbf{x} < 0$ then it means that the angle (α) between this $\mathbf{x}$ and the current $\mathbf{w}$ is greater than 90

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$? Less than 90°
- What will be the angle between vector $\mathbf{x} \in N$ and $\mathbf{w}$? Greater than 90°
- Ponder and convince yourself this is the case
- For $\mathbf{x} \in P$ if $\mathbf{w}^\intercal\mathbf{x} < 0$ then it means that the angle (α) between this $\mathbf{x}$ and the current $\mathbf{w}$ is greater than 90
- But we want it to be less than 90

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What will be the angle between vector $\mathbf{x} \in P$ and $\mathbf{w}$? Less than 90°
- What will be the angle between vector $\mathbf{x} \in N$ and $\mathbf{w}$? Greater than 90°
- Ponder and convince yourself this is the case
- For $\mathbf{x} \in P$ if $\mathbf{w}^\intercal\mathbf{x} < 0$ then it means that the angle (α) between this $\mathbf{x}$ and the current $\mathbf{w}$ is greater than 90
- But we want it to be less than 90
- How is adding $\mathbf{x}$ to $\mathbf{w}$ helping us?

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\cos \alpha_{new} \propto \mathbf{w_{new}}^\intercal \mathbf{x}$$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w} + \mathbf{x})^\intercal\mathbf{x}\end{aligned}$$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w} + \mathbf{x})^\intercal\mathbf{x} \\ &\propto \mathbf{w}^\intercal\mathbf{x} + \mathbf{x}^\intercal\mathbf{x}\end{aligned}$$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w} + \mathbf{x})^\intercal\mathbf{x} \\ &\propto \mathbf{w}^\intercal\mathbf{x} + \mathbf{x}^\intercal\mathbf{x} \\ &\propto \cos\alpha + \mathbf{x}^\intercal\mathbf{x}\end{aligned}$$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w} + \mathbf{x})^\intercal\mathbf{x} \\ &\propto \mathbf{w}^\intercal\mathbf{x} + \mathbf{x}^\intercal\mathbf{x} \\ &\propto \cos\alpha + \mathbf{x}^\intercal\mathbf{x} \\ \cos\alpha_{new} &> \cos\alpha\end{aligned}$$

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w}+\mathbf{x})^\intercal\mathbf{x} \\ &\propto \mathbf{w}^\intercal\mathbf{x} + \mathbf{x}^\intercal\mathbf{x} \\ &\propto \cos\alpha + \mathbf{x}^\intercal\mathbf{x} \\ \cos\alpha_{new} &> \cos\alpha\end{aligned}$$

- Thus α_{new} will be less than α and this is exactly what we want

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w}+\mathbf{x})^\intercal\mathbf{x} \\ &\propto \mathbf{w}^\intercal\mathbf{x} + \mathbf{x}^\intercal\mathbf{x} \\ &\propto \cos\alpha + \mathbf{x}^\intercal\mathbf{x} \\ \cos\alpha_{new} &> \cos\alpha\end{aligned}$$

- Thus α_{new} will be less than α and this is exactly what we want
- We can work out the math for the other case, when $\mathbf{x} \in N$ and $\mathbf{w}^\intercal\mathbf{x} \geq 0$ quite similarly

Perceptron Learning Algorithm

Algorithm 1 Perceptron Learning

$\mathbf{w} = [w_0, w_1, w_2, \ldots, w_n]$
$\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$
$P \leftarrow$ input with labels 1;
$N \leftarrow$ input with labels 0;
Initialize w randomly;
while *!convergence* **do**
  Pick random x $\in P \cup N$
  if $\mathbf{x} \in P$ *and* $\mathbf{w}^\intercal\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ *and* $\mathbf{w}^\intercal\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$
end
/* algorithm converges when all the inputs are classified correctly */

- What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned}\cos\alpha_{new} &\propto \mathbf{w_{new}}^\intercal\mathbf{x} \\ &\propto (\mathbf{w}+\mathbf{x})^\intercal\mathbf{x} \\ &\propto \mathbf{w}^\intercal\mathbf{x} + \mathbf{x}^\intercal\mathbf{x} \\ &\propto \cos\alpha + \mathbf{x}^\intercal\mathbf{x} \\ \cos\alpha_{new} &> \cos\alpha\end{aligned}$$

- Thus α_{new} will be less than α and this is exactly what we want
- We can work out the math for the other case, when $\mathbf{x} \in N$ and $\mathbf{w}^\intercal\mathbf{x} \geq 0$ quite similarly
- For a formal convergence proof, please see this link

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$

- The fourth condition contradicts conditions 2 and 3

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

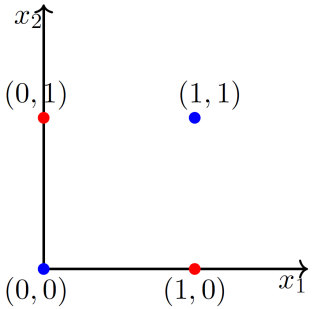
$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$

- The fourth condition contradicts conditions 2 and 3
- No solution possible satisfying this set of inequalities

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$



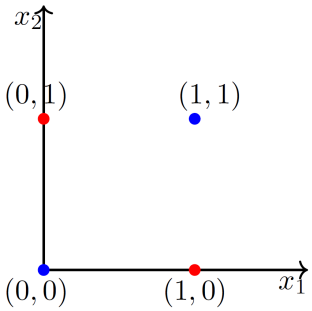
- The fourth condition contradicts conditions 2 and 3
- No solution possible satisfying this set of inequalities

The XOR Conundrum

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

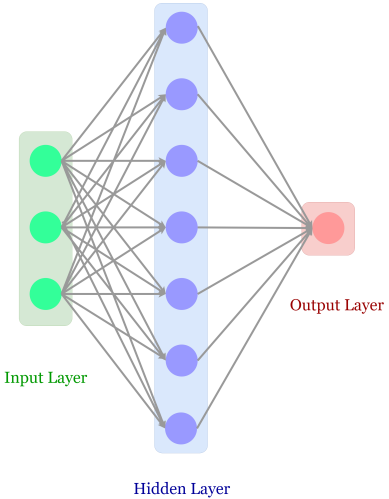
$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$

- The fourth condition contradicts conditions 2 and 3
- No solution possible satisfying this set of inequalities

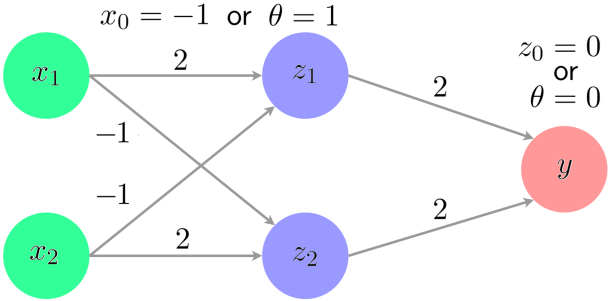


- Indeed you can see that it is impossible to draw a line which separates the red points from the blue points

Multi-Layer Perceptrons



Solving XOR with Multi-Layer Perceptrons



(x_1,x_2)	(z_1,z_2)	y
(0,0)	(0,0)	0
(0,1)	(0,0)	1
(1,0)	(1,0)	1
(1,1)	(0,0)	0

Multi-Layer Perceptrons

Theorem: Any boolean function of n inputs can be represented exactly by a network of perceptrons containing 1 hidden layer with 2^n perceptrons and one output layer containing 1 perceptron

Proof (Informal): How?

Multi-Layer Perceptrons

Theorem: Any boolean function of n inputs can be represented exactly by a network of perceptrons containing 1 hidden layer with 2^n perceptrons and one output layer containing 1 perceptron

Proof (Informal): How? Each of the 2^n hidden layer perceptrons can model (or can be fired by) one combination of n inputs.

Note: A network of $2^n + 1$ perceptrons is not necessary but sufficient

But why does this matter?

Multi-Layer Perceptrons

Theorem: Any boolean function of n inputs can be represented exactly by a network of perceptrons containing 1 hidden layer with 2^n perceptrons and one output layer containing 1 perceptron

Proof (Informal): How? Each of the 2^n hidden layer perceptrons can model (or can be fired by) one combination of n inputs.

Note: A network of $2^n + 1$ perceptrons is not necessary but sufficient

But why does this matter? As n increases, the number of perceptrons in the hidden layers increases exponentially. We'd ideally want this to be as few as possible.

Going Beyond Binary Inputs and Outputs

Question

- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbf{R}^n$ (instead of $\{0,1\}^n$) and $y \in \mathbf{R}$ (instead of $\{0,1\}$)?
- Can we use the same perceptron model to represent such functions?

Need for Activation Functions

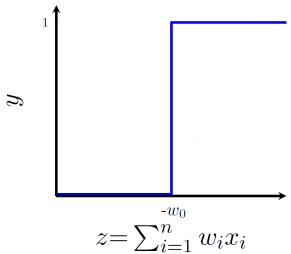
- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$

Need for Activation Functions

- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$
- Thresholding logic could be harsh at times
- E.g., when $-w_0 = 0.5$, though output values 0.49 and 0.51 are very close to each other, the perceptron would assign different labels to them.

Need for Activation Functions

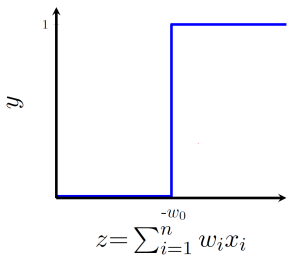
- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$
- Thresholding logic could be harsh at times
- E.g., when $-w_0 = 0.5$, though output values 0.49 and 0.51 are very close to each other, the perceptron would assign different labels to them.
- This behavior is not a characteristic of the problem, the weights or the threshold; it is a characteristic of the perceptron function itself which behaves like a step function



Need for Activation Functions

- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$
- Thresholding logic could be harsh at times
- E.g., when $-w_0 = 0.5$, though output values 0.49 and 0.51 are very close to each other, the perceptron would assign different labels to them.

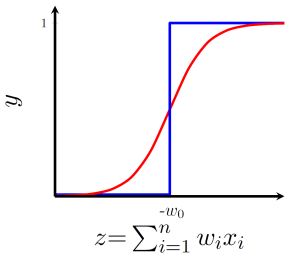
- This behavior is not a characteristic of the problem, the weights or the threshold; it is a characteristic of the perceptron function itself which behaves like a step function
- There will always be a sudden change in decision (from 0 to 1) when $\sum_{i=1}^{n} w_i x_i$ crosses the threshold $(-w_0)$



Need for Activation Functions

- A perceptron only fires if weighted sum of its inputs is greater than threshold $-w_0$
- Thresholding logic could be harsh at times
- E.g., when $-w_0 = 0.5$, though output values 0.49 and 0.51 are very close to each other, the perceptron would assign different labels to them.

- This behavior is not a characteristic of the problem, the weights or the threshold; it is a characteristic of the perceptron function itself which behaves like a step function
- There will always be a sudden change in decision (from 0 to 1) when $\sum_{i=1}^{n} w_i x_i$ crosses the threshold $(-w_0)$
- For most real-world applications, we'd expect a smoother decision function which gradually changes from 0 to 1



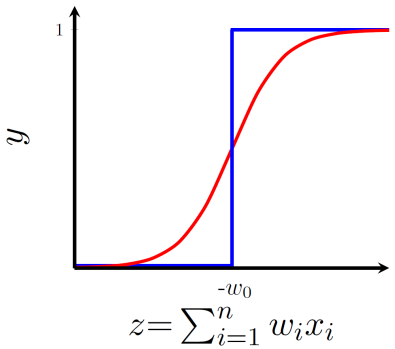
Sigmoid Neurons

- We could use any logistic function to obtain a smoother output function than a step function

Sigmoid Neurons

- We could use any logistic function to obtain a smoother output function than a step function
- One form is the sigmoid function:

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=0}^{n} w_i x_i)}}$$

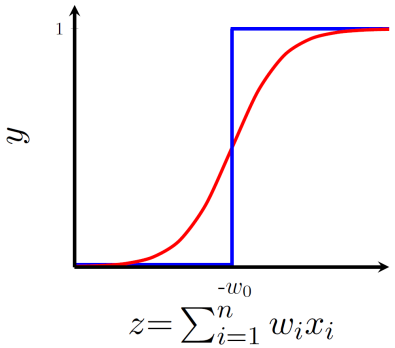


Sigmoid Neurons

- We could use any logistic function to obtain a smoother output function than a step function
- One form is the sigmoid function:

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=0}^{n} w_i x_i)}}$$

- No longer a sharp transition at the threshold $-w_0$

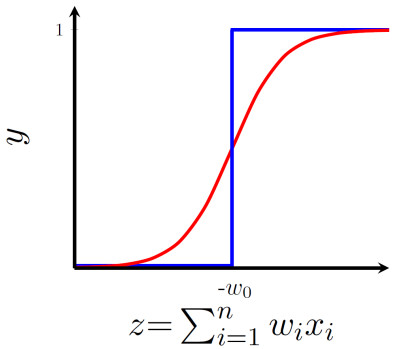


Sigmoid Neurons

- We could use any logistic function to obtain a smoother output function than a step function
- One form is the sigmoid function:

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=0}^{n} w_i x_i)}}$$

- No longer a sharp transition at the threshold $-w_0$
- Also, output is no longer binary but a real value between 0 and 1 which can be interpreted as a probability

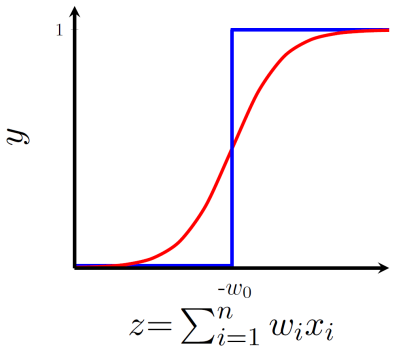


Sigmoid Neurons

- We could use any logistic function to obtain a smoother output function than a step function
- One form is the sigmoid function:

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=0}^{n} w_i x_i)}}$$

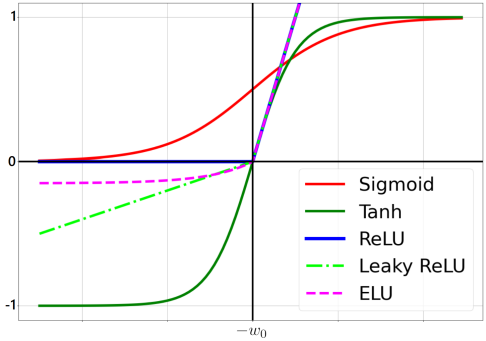
- No longer a sharp transition at the threshold $-w_0$
- Also, output is no longer binary but a real value between 0 and 1 which can be interpreted as a probability
- Unlike the step function, this one is smooth, continuous at $-w_0$ and most importantly **differentiable**



Other Popular Activation Functions

Considering $z = \sum_{i=0}^{n} w_i x_i$

- **Sigmoid:** $y = \frac{1}{1+e^{-z}}$
- **Tanh:** $y = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- **Rectified Linear Unit (ReLU):** $y = \max(0, z)$
- **Leaky ReLU:** $y = \max(\alpha z, z), \alpha \in (0, 1)$
- **Exponential Linear Unit (ELU):** $y = \max(\alpha(e^{z} - 1), z)$, where $\alpha > 0$



Representation Power of MLPs

Theorem: A multilayer network of sigmoid neurons with a single hidden layer can be used to approximate any continuous function to any desired precision. (Also known as **Universal Approximation Theorem**)

Proof: Cybenko, 1989 [2] and Hornik et al, 1989[3]

Note: Also, refer to Chapter 4 of Michael Nielsen's online book for visual explanation of Universal Approximation Theorem

[2] Cybenko, Approximation by superpositions of a sigmoidal function, Mathematics of Control, Signals and Systems, Vol 2, pp 303-314, 1989

[3] Hornik et al, Multilayer feedforward networks are universal approximators, Neural Networks Vol 2:5, pp 359-366, 1989

Homework

Homework

- Solve XOR using an MLP with 4 hidden units.

Readings

- Please refer to Mitesh Khapra's original lecture slides (and videos) for more detailed explanation of some of these topics, available at CS7015: Deep Learning.
- Other good resources:
  - Deep Learning Book: Chapter 6 for Multilayer Perceptrons
  - Stanford CS231n Notes
  - Stanford UFLDL tutorial on Multilayer Neural Networks
  - Neural Networks: A Systematic Introduction by Raúl Rojas

References

George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 303–314.

Tianping Chen, Hong Chen, and Ruey-wen Liu. "Approximation Capability in by Multilayer Feedforward Networks and Related Problems". In: *Neural Networks, IEEE Transactions on* 6 (Feb. 1995), pp. 25 –30.

Mikel L. Forcada. *Neural Networks: Automata and Formal Models of Computation*. https://www.dlsi.ua.es//~mlf/nnafmc/pbook.pdf. 2002.

Michael Collins. *Convergence Proof for the Perceptron Algorithm*. http://www.cs.columbia.edu/~mcollins/courses/6998-2012/notes/perc.converge.pdf. 2012.

Deep Learning for Computer Vision

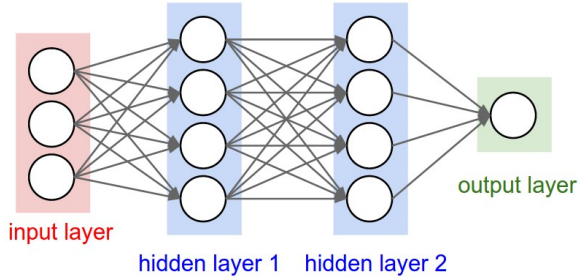
Feedforward Neural Networks and Backpropagation

Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Feedforward Networks

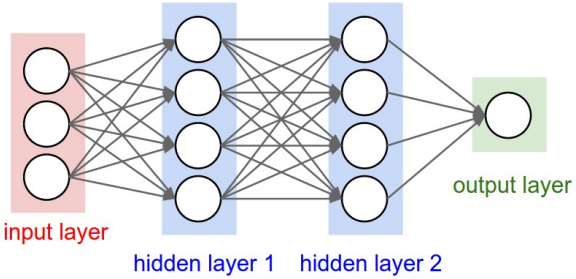
- A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in *layers*.



- It is used to approximate some function f^*. For instance, f^* could be a classifier that maps an input vector $\boldsymbol{x}$ to a category $\boldsymbol{y}$.
- The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input $\boldsymbol{x}$ to output $\boldsymbol{y}$. Hence the term **feedforward**.

Feedforward Networks

- The number of layers in the network (excluding the input layer) is known as **depth**



- Each neuron can be seen as a **vector**-to-**scalar** function which takes a vector of inputs from the previous layer and computes a scalar value.
- Above network can be seen as a composition of functions $y = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, $f^{(1)}$ being the first hidden layer, $f^{(2)}$ being the second and $f^{(3)}$ being the final output layer.

Feedforward Networks

- To approximate some function f^*, we are generally given noisy estimates of $f^*(\boldsymbol{x})$ at different points, in the form of a dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^{M}$.
- Our neural network defines a function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$. Our goal is to learn the parameters (weights and biases) $\boldsymbol{\theta}$ such that f best approximates f^*.
- How to find the values of the parameters i.e., train the network?
- In this lecture, we introduce **Gradient Descent**, the go-to method to train neural networks

Gradient Descent: A 1D Example

- Neural networks are usually trained by minimizing a loss function, such as mean square error:

$$Loss_{MSE} = \frac{1}{M}\sum_{i=1}^{M}(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta}))^2$$

- Let us consider a simple 1D example, where we try to minimize the function $f(x) = x^2$. Specifically, we find out the value x^* gives the smallest value for $f(x)$ i.e., $f(x^*)$.

$$x^* = \arg\min_x f(x)$$

Gradient Descent: A 1D Example

- We can obtain the slope of the function $f(x)$ at x by taking its derivative i.e., $f'(x)$.

Gradient Descent: A 1D Example

- We can obtain the slope of the function $f(x)$ at x by taking its derivative i.e., $f'(x)$.
- This means, if we give a very small *push* to x in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \quad \text{for an infinitesimally small } p$$

Gradient Descent: A 1D Example

- We can obtain the slope of the function $f(x)$ at x by taking its derivative i.e., $f'(x)$.
- This means, if we give a very small *push* to x in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \quad \text{for an infinitesimally small } p$$

- The reverse is also true i.e.,

$$f(x - p \cdot \text{sign}(f'(x))) < f(x) \quad \text{for an infinitesimally small } p$$

- This forms the basis for gradient descent - we start off at a random x, and take small steps in the direction of the **negative** gradient.

Gradient Descent: A 1D Example

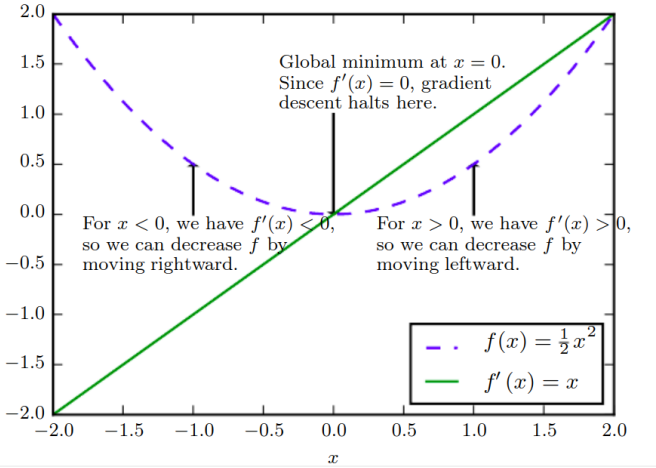


Figure Credit: Goodfellow et al, DL Book Ch 4

Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, $\boldsymbol{\theta}$
- For simplicity, we denote our loss function as $L(\boldsymbol{\theta})$. Our aim is to find the weight vector $\boldsymbol{\theta}$ which minimizes $L(\boldsymbol{\theta})$

Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, $\boldsymbol{\theta}$
- For simplicity, we denote our loss function as $L(\boldsymbol{\theta})$. Our aim is to find the weight vector $\boldsymbol{\theta}$ which minimizes $L(\boldsymbol{\theta})$
- Let $\mathbf{u}$, a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{\mathbf{u}, \mathbf{u}^T\mathbf{u}=1} \mathbf{u}^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights, $\boldsymbol{\theta}$
- For simplicity, we denote our loss function as $L(\boldsymbol{\theta})$. Our aim is to find the weight vector $\boldsymbol{\theta}$ which minimizes $L(\boldsymbol{\theta})$
- Let **u**, a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{\mathbf{u}, \mathbf{u}^T\mathbf{u}=1} \mathbf{u}^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

$$= \min_{\mathbf{u}, \mathbf{u}^T\mathbf{u}=1} \|\mathbf{u}\|_2 \|\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})\|_2 \cos\beta$$

- Since $\|\boldsymbol{u}\|_2 = 1$, we can minimize the above function when $\beta = 180^0$, i.e. when **u** is the direction of **negative** gradient

How to use Gradient Descent

We can use Gradient Descent to train neural networks as follows:

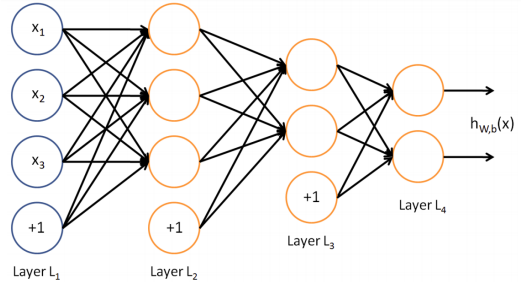
- Start with a random weight vector $\boldsymbol{\theta}$.
- Compute the loss function over the dataset, i.e., $L(\theta)$ with the current network, using a suitable loss function such as mean-squared error
- Compute the gradients of the loss function with respect to each weight value $\frac{\delta L}{\delta \boldsymbol{\theta}_i}$.
- Update the weights as follows, where η is the learning rate i.e., the amount by which the weight is changed in each step:

$$\boldsymbol{\theta}_i^{next} = \boldsymbol{\theta}_i^{curr} - \eta \frac{\delta L}{\delta \boldsymbol{\theta}_i^{curr}}$$

We can repeat the above steps until the gradient is zero.

Gradient Descent

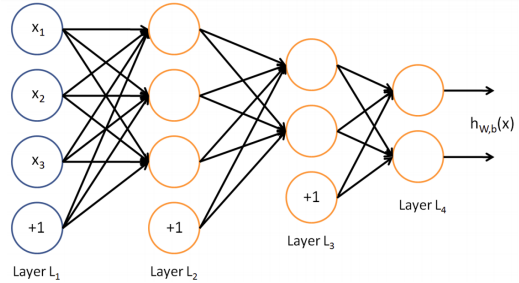
- A feedforward neural network is a composition of multiple functions, organized as layers



- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?

Gradient Descent

- A feedforward neural network is a composition of multiple functions, organized as layers



- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?
- Using the **chain rule in calculus**
  - Computing gradient w.r.to a weight in layer i requires computation of gradients with respect to outputs which involve that weight i.e., all activations from layer $i+1$ to last layer, n_l

Gradient Descent

In the next few slides, we introduce **backpropagation**, a procedure which combines gradient computation using chain rule and parameter updation using Gradient Descent, thus fully describing the neural network training algorithm.

Backpropagation

Consider a simple feed forward neural network (or multilayer perceptron)

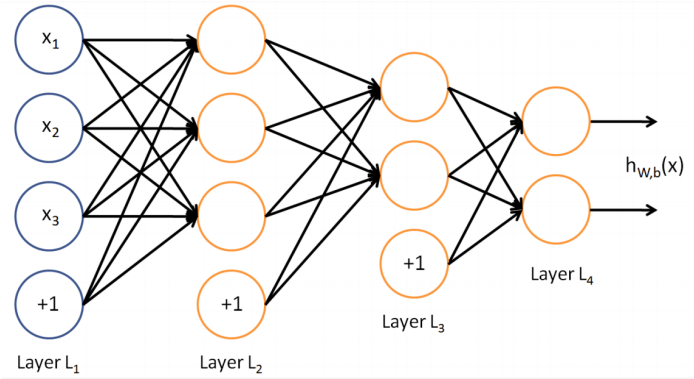


Figure Credit: Stanford UFLDL Tutorial

Backpropagation

- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^{M}$ of M training samples
- Parameters $\theta = \{W, b\}$, weights and biases

Backpropagation

- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^{M}$ of M training samples
- Parameters $\theta = \{W, b\}$, weights and biases
- Mean square cost function for a single example:

$$L(\theta; x, y) = \frac{1}{2}\|h_\theta(x) - y\|^2$$

Backpropagation

- A fixed training set $\{x^{(i)}, y^{(i)}\}_{i=1}^{M}$ of M training samples
- Parameters $\theta = \{W, b\}$, weights and biases
- Mean square cost function for a single example:

$$L(\theta; x, y) = \frac{1}{2}\|h_\theta(x) - y\|^2$$

- Overall cost function is given by:

$$\begin{aligned} L(\theta) &= \frac{1}{M}\sum_{i=1}^{M} L(\theta; x^{(i)}, y^{(i)}) \\ &= \frac{1}{2M}\sum_{i=1}^{M} \|h_\theta(x^{(i)} - y^{(i)}\|^2 \end{aligned}$$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l+1$ is denoted as $W^{(l)}$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l+1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l+1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$

Backpropagation: Notations

- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l+1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

Backpropagation: Notations

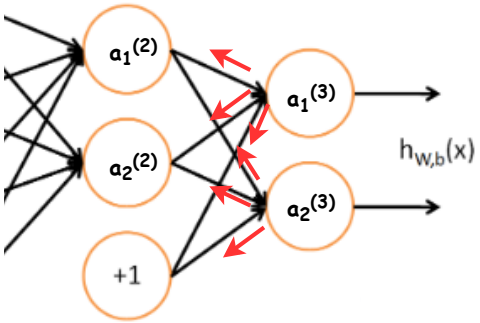
- We have n_l layers in the network, $l = 1, 2, .., n_l$
- We denote activation of node i at layer l as $a_i^{(l)}$
- We denote weight connecting node i in layer l and node j in layer $l+1$ as $W_{ij}^{(l)}$. The weight matrix between layer l and layer $l+1$ is denoted as $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

- Function f can denote any activation function such as sigmoid, ReLU, identity, etc.

Backpropagation: Backward Pass

- During the forward pass, we successively compute each layer's outputs from left to right.
- During backward pass, we aim to compute derivatives of each parameter starting from the right most layer to the left most one i.e., layer $n_l, n_l - 1, ..., 1$.
- Once the derivatives are computed, we use Gradient Descent to update the parameters.



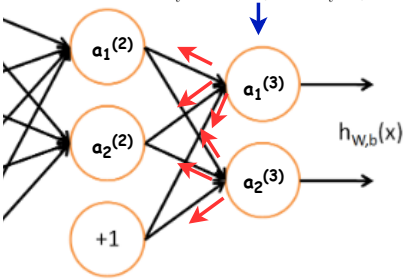
Backpropagation: Backward Pass

- For each node, we define an **error term** $\delta_i^{(l)}$ to denote how much the node was responsible for the loss computed
- If $l = n_l$ i.e., **last layer**, error term computation is straightforward, since we directly take derivative of loss function (MSE, in this case, between output and target values)

$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

Error Term at the output layer

$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$



Error Term at the hidden layer

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \circ f'(z^{(l)})$$

Backpropagation: Backward Pass

- To compute error term for **hidden layers**, $l = n_l - 1, n_l - 2, ..,$ we rely on error terms from subsequent layers

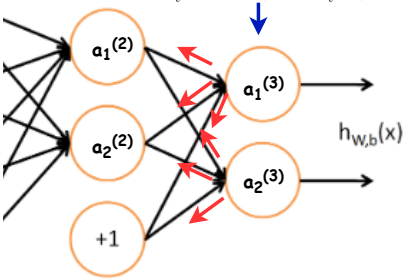
Backpropagation: Backward Pass

- To compute error term for **hidden layers**, $l = n_l - 1, n_l - 2, ..,$ we rely on error terms from subsequent layers
- In particular, we compute error term as sum of error terms in next layer, weighted by weights along connections to next layer:

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ij}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

Error Term at the output layer

$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

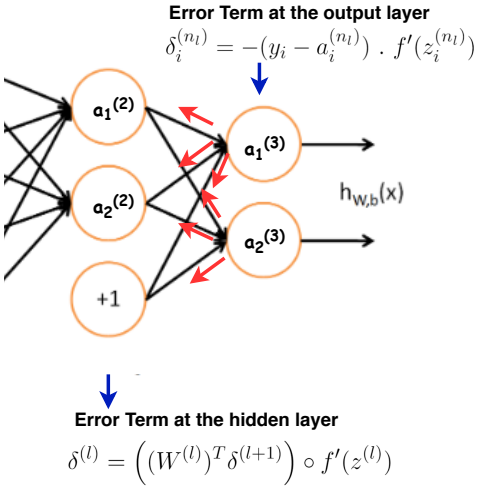


Error Term at the hidden layer

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)} \right) \circ f'(z^{(l)})$$

Backpropagation: Backward Pass

- Note that f' denotes derivative of activation function
- For a linear neuron $f(x) = x$, derivative is 1
- For a sigmoid neuron $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$, derivative turns out to be $\sigma(x)(1 - \sigma(x))$ (How? Homework!)



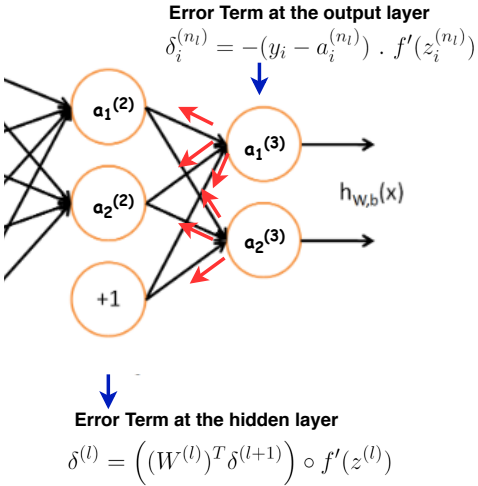
Backpropagation: Algorithm

- Perform a feedforward pass, computing the activations for layers $1, 2, ...n_l$.
- For each output unit i in layer n_l set,

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \circ f'(z^{(n_l)})$$

- For $l = n_l - 1, n_l - 2, n_l - 3, .., 2$
  For each node in layer l set,

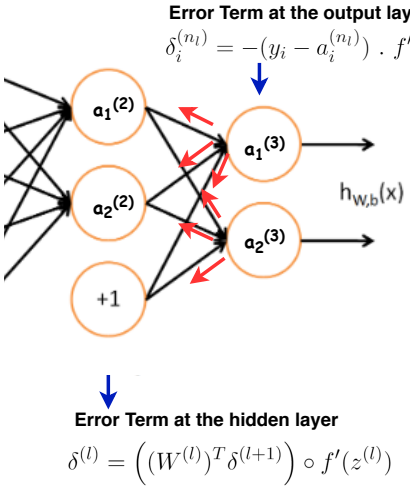
$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \circ f'(z^{(l)})$$



Backpropagation: Algorithm

- Compute the desired partial derivatives, as:

$$\nabla_{W^{(l)}} L(W, b; x, y) = \delta^{l+1}(a^{(l)})^T$$
$$\nabla_{b^{(l)}} L(W, b; x, y) = \delta^{l+1}$$



Gradient Descent using Backpropagation

1. Set $\Delta W^{(l)} := 0, \Delta b^{(l)} = 0$ (matrix/vector of zeros) for all l.
2. For $i = 1$ to M
   1. Use backpropagation to compute $\nabla_{\theta^{(l)}} L(\theta; x, y)$.
   2. Set $\Delta\theta^{(l)} := \Delta\theta^{(l)} + \nabla_{\theta^{(l)}} L(\theta; x, y)$
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \eta \left[\frac{1}{M} \Delta W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \eta \left[\frac{1}{M} \Delta b^{(l)} \right]$$

4. Repeat for all points until convergence.

Homework

Readings

- Chapter 4, Deep Learning Book
- Chapter 6, Deep Learning Book
- Stanford UFLDL tutorial
- Stanford CS231n Notes

Exercises

- Work out the derivative of the `sigmoid` and `tanh` activation functions
- Sigmoid activation function: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Tanh activation function: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

References


Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Deep Learning for Computer Vision

Gradient Descent and Variants

Vineeth N Balasubramanian

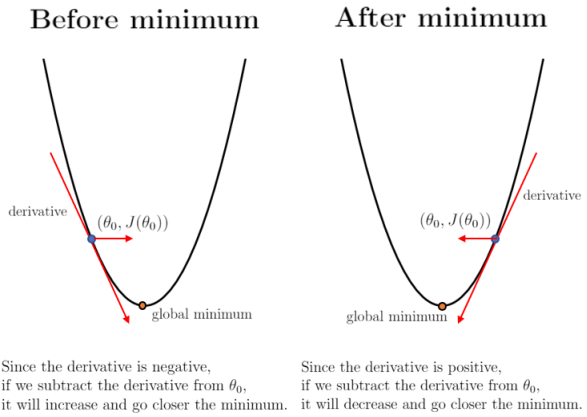
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Review: Gradient Descent (GD)

- Optimization algorithm used to find minima of a given differentiable function
- At each step, parameters (θ) are pushed in negative direction of gradient of a cost function ($J(\theta\ x, y)$, in figure alongside) w.r.t parameters

$$\theta_{new} = \theta_{old} - \alpha \Delta \theta_{old}$$

where α is learning rate



Credit: *Albert Lai*

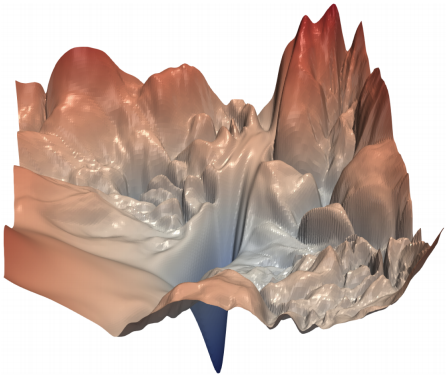
Gradient Descent Algorithm

Require: Learning rate α, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

1: **while** stopping criterion not met **do**
2: Initialize parameter updates $\Delta\theta_t = 0$
3: **for each** $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
4: Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
5: Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
6: **end for**
7: Apply update $\theta_{t+1} = \theta_t - \alpha\frac{1}{|\mathcal{D}_{tr}|}\Delta\theta_t$
8: **end while**

Error Surface of Neural Networks

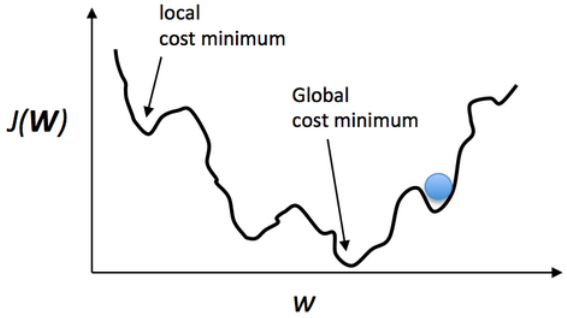
Visualization of error surface of a neural network (ResNet-56)



Credit: Li et al, Visualizing the Loss Landscape of Neural Nets, NeurIPS 2018

Local Minima

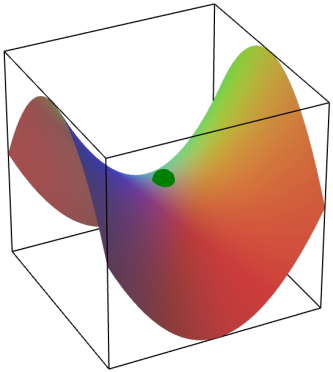
- Unlike convex objective functions that have a global minimum, non-convex functions as in deep neural networks have multiple local minima[1]



[1] Choromanska et al, The Loss Surface of Multilayer Nets, AISTATS 2015

Saddle Points

- Local maximum along one cross-section of cost function and local minimum along another
- Though it isn't a local minimum, gradient is zero (or almost close to zero) $\implies$ gives impression of convergence
- Though local minima are prevalent in lower-dimensional spaces, saddle points become more common in higher-dimensional spaces



Dauphin et al, Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, NeurIPS 2014

Gradient Descent Traversal

Let us see a GD traversal example!

Notice anything interesting?

Credit: Mitesh Khapra, IIT Madras

Gradient Descent Traversal

Let us look at another example with a different initial position
HINT: Pay attention to how the relative position of parameters changes

Credit: Mitesh Khapra, IIT Madras

Gradient Descent Traversal

Top view of previous error surface represented as contours

Notice how parameter updates are smaller at points where gradient of error surface is small and larger at points where gradient is large.

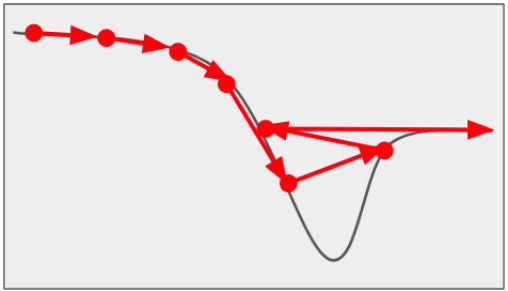
Credit: Mitesh Khapra, IIT Madras

Plateaus and Flat Regions

- They constitute portions of error surface where gradient is highly non-spherical
- Gradient descent spends a long time traversing in these regions as the updates are small
- Can we expedite this process?

Plateaus and Flat Regions

- They constitute portions of error surface where gradient is highly non-spherical
- Gradient descent spends a long time traversing in these regions as the updates are small
- Can we expedite this process?
- How about increasing the learning rate?
- Though traversal becomes faster in plateaus, there is a risk of divergence



Momentum-based GD

- **Intuition:** With increasing confidence, increase step size; and with decreasing confidence, decrease step size
- Weight update given by:

Momentum Term

$$v_t = \gamma v_{t-1} + \alpha \nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$$

$$\theta_{t+1} = \theta_t - v_t$$

Momentum-based GD

- **Intuition:** With increasing confidence, increase step size; and with decreasing confidence, decrease step size
- Weight update given by:

$$v_t = \underbrace{\gamma v_{t-1}}_{\text{Momentum Term}} + \alpha \nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$$

$$\theta_{t+1} = \theta_t - v_t$$

From these equations, can you see how momentum possibly avoids divergence at 5^{th} and 6^{th} steps of previous figure?

Momentum-based GD

- **Intuition:** With increasing confidence, increase step size; and with decreasing confidence, decrease step size
- Weight update given by:

$$v_t = \underbrace{\gamma v_{t-1}}_{\text{Momentum Term}} + \alpha \nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$$

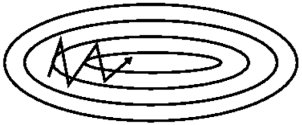
$$\theta_{t+1} = \theta_t - v_t$$

From these equations, can you see how momentum possibly avoids divergence at 5^{th} and 6^{th} steps of previous figure?

After 5^{th} step, momentum and gradient terms are in opposite directions

Momentum-based GD

Without momentum



With momentum

- Damps step sizes along directions of high curvature, yielding a larger effective learning rate along the directions of low curvature.
- Larger the γ, more the previous gradients affect the current step.
- Generally, γ is set to 0.5 until initial learning stabilizes and then increased to 0.9 or higher (practitioners often default it to 0.9/0.95 these days)

Momentum-based GD: Algorithm

Require: Learning rate α, momentum parameter γ, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

Initialize $v_{t-1} = 0$
while stopping criterion not met **do**
    Initialize weight updates $\Delta\theta_t = 0$
    for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
        Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
        Aggregate weight updates: $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
    end for
    Update velocity: $v_t = \gamma v_{t-1} + \alpha\Delta\theta_t$
    Apply update: $\theta_{t+1} = \theta_t - v_t$
end while

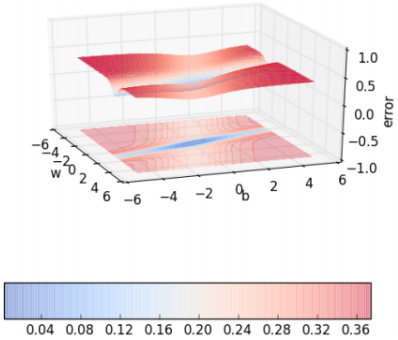
Momentum-based GD: Illustration

Convergence of Momentum vs Gradient Descent

Momentum performance was better than GD. But is the speed always good?

Credit: Mitesh Khapra, IIT Madras

Is Momentum Detrimental?



Any observations?

Credit: Mitesh Khapra, IIT Madras

Issues with Momentum

- Momentum-based GD oscillates around minima before eventually reaching it
- Even then, it converges faster than vanilla GD!
  - After 100 iterations, momentum-based GD has error of 0.00001, whereas vanilla GD is still at error of 0.36
- Nonetheless, it is wasting time in oscillations; how can we reduce oscillations?

Source: Mitesh Khapra, IIT Madras

Nesterov Accelerated Momentum

- Based on Nesterov's Accelerated Gradient Descent published in 1983[2]; re-introduced by Sutskever in ICML'13[3]
- Key idea: **Look before you leap**
- Assess how gradient changes after taking a step of **momentum**, γv_t, and use this to get better estimate of parameters

[2]Nesterov, A method of solving a convex programming problem with convergence rate O($1/k^2$), Soviet Mathematics Doklady, 1983, 27:2, pp 372–376

[3]Sutskever et al, On the importance of initialization and momentum in deep learning, ICML 2013, Vol 28, pp 1139—1147

Nesterov Accelerated Momentum

- Based on Nesterov's Accelerated Gradient Descent published in 1983[2]; re-introduced by Sutskever in ICML'13[3]
- Key idea: **Look before you leap**
- Assess how gradient changes after taking a step of **momentum**, γv_t, and use this to get better estimate of parameters
- Weight update given by:

$$v_t = \gamma v_{t-1} + \alpha \nabla_{\tilde{\theta_t}} \mathcal{L}(\theta_t - \gamma v_{t-1}; x^{(i)}, y^{(i)})$$

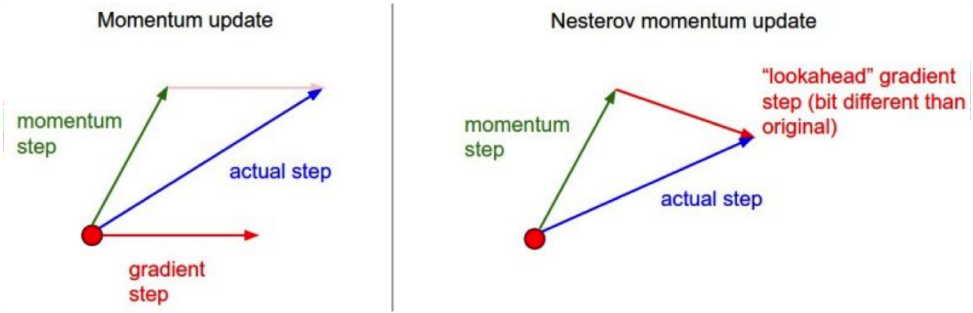
$$\theta_{t+1} = \theta_t - v_t$$

- Empirically found to give good performance

[2]Nesterov, A method of solving a convex programming problem with convergence rate O($1/k^2$), Soviet Mathematics Doklady, 1983, 27:2, pp 372–376

[3]Sutskever et al, On the importance of initialization and momentum in deep learning, ICML 2013, Vol 28, pp 1139—1147

Nesterov Accelerated Momentum: Visualization



Credit: Fei-Fei Li, CS231N course, Stanford Univ

Nesterov Accelerated Momentum: Algorithm

Require: Learning rate α, momentum parameter γ, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

Initialize $v_{t-1} = 0$
while stopping criterion not met **do**
    Initialize gradients $\Delta\theta_t = 0$
    Get look-ahead parameters $\tilde{\theta}_t = \theta_t - \gamma v_{t-1}$
    for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
        Compute gradient using look-ahead parameters $\nabla_{\tilde{\theta}_t} \mathcal{L}(\tilde{\theta}_t; x^{(i)}, y^{(i)})$
        Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\tilde{\theta}_t} \mathcal{L}$
    end for
    Update velocity $v_t = \gamma v_{t-1} + \alpha \Delta\theta_t$
    Apply update $\theta_{t+1} = \theta_t - v_t$
end while

Nesterov Accelerated Momentum: Illustration

Credit: Mitesh Khapra, IIT Madras

GD: Pros and Cons

What do you think?

GD: Pros and Cons

What do you think?

- For every parameter update, GD parses the entire dataset, hence called **Batch GD**
- Advantages of Batch GD
  - Conditions of convergence well-understood
  - Many acceleration techniques (e.g. conjugate gradient) operate in batch GD setting

GD: Pros and Cons

What do you think?

- For every parameter update, GD parses the entire dataset, hence called **Batch GD**
- Advantages of Batch GD
  - Conditions of convergence well-understood
  - Many acceleration techniques (e.g. conjugate gradient) operate in batch GD setting
- Disadvantages of Batch GD
  - Computationally slow
  - E.g. ImageNet (http://www.image-net.org), a commonly used dataset in vision, has ~ 14.2 million samples; an iteration over it is going to be very slow

Stochastic Gradient Descent

Stochastic GD (SGD): Randomly shuffle the training set, and update parameters after gradients are computed for each training example

---

Require: Learning rate α, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

---

while stopping criterion not met **do**
  for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
    Compute gradient using backpropagation $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
    Gradient $\Delta\theta_t = \nabla_{\theta_t} \mathcal{L}$
    Apply update $\theta_{t+1} = \theta_t - \alpha \Delta\theta_t$
  end for
end while

Mini-batch Stochastic Gradient Descent

Mini-Batch Stochastic GD: Update parameters after gradients are computed for a randomly drawn mini-batch of training examples *(default option today, often simply called as SGD)*

Require: Learning rate α, initial parameters θ_t, mini-batch size m, training dataset $\mathcal{D}_{tr}$

1: **while** stopping criterion not met **do**
2: Initialize gradients $\Delta\theta_t = 0$
3: Sample m examples from $\mathcal{D}_{tr}$ (call it $\mathcal{D}_{mini}$)
4: **for each** $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{mini}$ **do**
5: Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
6: Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
7: **end for**
8: Apply update $\theta_{t+1} = \theta_t - \alpha\Delta\theta_t$
9: **end while**

Illustration of SGD

Do the oscillations remind you of something?

Illustration of SGD

Do the oscillations remind you of something? Momentum?

Credit: Mitesh Khapra, IIT Madras

Illustration of Mini-batch SGD

Notice how the traversal is oscillating less for mini-batch GD ($m = 2$) when compared to SGD

Credit: Mitesh Khapra, IIT Madras

SGD: Pros and Cons

SGD: Pros and Cons

Advantages of SGD

- Usually much faster than batch learning; because there is lot of redundancy in batch learning
- Often results in better solutions; SGD's noise can help in escaping local minima (provided neighborhood provides enough gradient information) and saddle points.[a]
- Can be used for tracking changes

[a] http://mitliagkas.github.io/ift6085-2019/ift-6085-bonus-lecture-saddle-points-notes.pdf

SGD: Pros and Cons

Advantages of SGD

- Usually much faster than batch learning; because there is lot of redundancy in batch learning
- Often results in better solutions; SGD's noise can help in escaping local minima (provided neighborhood provides enough gradient information) and saddle points.[a]
- Can be used for tracking changes

[a] http://mitliagkas.github.io/ift6085-2019/ift-6085-bonus-lecture-saddle-points-notes.pdf

Disadvantages of SGD

- Noise in SGD weight updates – can lead to no convergence!
- Can be controlled using learning rate, but identifying proper learning rate is a problem of its own

Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?

Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?
- Possible methods:
  - **Naive linear search** (learning rate remains constant throughout the process)

Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?
- Possible methods:
  - **Naive linear search** (learning rate remains constant throughout the process)
  - Annealing-based methods:
    - **Step decay:** Reduce learning rate after every n iteration/epochs or if certain degenerative conditions are met (e.g. if current error more than previous error)

Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?
- Possible methods:
  - **Naive linear search** (learning rate remains constant throughout the process)
  - Annealing-based methods:
    - **Step decay:** Reduce learning rate after every n iteration/epochs or if certain degenerative conditions are met (e.g. if current error more than previous error)
    - **Exponential decay:** $\alpha = \alpha_0^{-kt}$ where α and k are hyperparameters and t is iteration number

Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?
- Possible methods:
  - **Naive linear search** (learning rate remains constant throughout the process)
  - Annealing-based methods:
    - **Step decay:** Reduce learning rate after every n iteration/epochs or if certain degenerative conditions are met (e.g. if current error more than previous error)
    - **Exponential decay:** $\alpha = \alpha_0^{-kt}$ where α and k are hyperparameters and t is iteration number
    - **1/t Decay:** $\alpha = \frac{\alpha_0}{1+kt}$ where α and k are hyperparameters and t is iteration number

Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?
- Possible methods:
  - **Naive linear search** (learning rate remains constant throughout the process)
  - Annealing-based methods:
    - **Step decay:** Reduce learning rate after every n iteration/epochs or if certain degenerative conditions are met (e.g. if current error more than previous error)
    - **Exponential decay:** $\alpha = \alpha_0^{-kt}$ where α and k are hyperparameters and t is iteration number
    - **1/t Decay:** $\alpha = \frac{\alpha_0}{1+kt}$ where α and k are hyperparameters and t is iteration number
- The above are heuristic however, any automatic way to pick learning rate?

Credit: Mitesh Khapra, IIT Madras

Adaptive Gradients (Adagrad)

Intuition

- Sparse but important features may often have small gradients when compared to others; learning is slow in their direction
- We can impose different learning rates to each feature such that sparse features have a higher learning rate

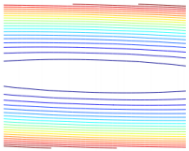
Weight update given by:

$$r_t = r_{t-1} + (\Delta\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_t}}\Delta\theta_t$$

Adagrad

Why adapt to geometry?



y_t	$\phi_{t,1}$	$\phi_{t,2}$	$\phi_{t,3}$
1	1	0	0
-1	.5	0	1
1	-.5	1	0
-1	0	0	0
1	.5	0	0
-1	1	0	0
1	-1	1	0
-1	-.5	0	1

1. Frequent, irrelevant
2. Infrequent, predictive
3. Infrequent, predictive

Figure Credit: http://seed.ucsd.edu/mediawiki/images/6/6a/Adagrad.pdf

Adagrad Algorithm

Require: Learning rate α, initial parameters θ_t, small constant δ (usually 10^{-7} for numeric stability), training dataset $\mathcal{D}_{tr}$

Initialize gradient accumulation $r_{t-1} = 0$
while stopping criterion not met **do**
    Initialize gradients $\Delta\theta_t = 0$
    for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
        Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
        Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
    end for
    Update gradient accumulation $r_t = r_{t-1} + (\Delta\theta_t)^2$
    Apply update $\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_t}}\Delta\theta_t$
end while

RMSProp

Intuition

- Adagrad increases denominator term very quickly for frequent features as their gradients
- Ultimately, the effective learning rate becomes too low to take a step
- Why not apply a weighted decay to the denominator to curb its aggressive increase?

Weight update given by:

$$r_t = (\rho) r_{t-1} + (1-\rho)(\Delta\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_t}} \Delta\theta_t$$

RMSProp Algorithm

Require: Learning rate α, decay rate ρ, initial parameters θ_t, small constant δ(usually 10^{-6} for numeric stability), training dataset $\mathcal{D}_{tr}$

Initialize gradient accumulation $r_{t-1} = 0$
while stopping criterion not met **do**
    Initialize gradients $\Delta\theta_t = 0$
    for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
        Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
        Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
    end for
    Update gradient accumulation $r_t = (\rho)r_{t-1} + (1-\rho)(\Delta\theta_t)^2$
    Apply update $\theta_{t+1} = \theta_t - \frac{\alpha}{\delta+\sqrt{r_t}}\Delta\theta_t$
end while

Adaptive Moments (ADAM)

Intuition

Combine Momentum and RMSProp alorithms

Weight update given by:

$$s_t = (\rho_1) r_{t-1} + (1 - \rho_1)(\Delta\theta_t)$$

$$r_t = (\rho_2) r_{t-1} + (1 - \rho_2)(\Delta\theta_t)^2$$

$$\text{Bias Correction: } \tilde{s_t} = \frac{s_t}{1 - \rho_1^t}, \tilde{r_t} = \frac{r_t}{1 - \rho_2^t}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\tilde{s_t}}{\delta + \sqrt{\tilde{r_t}}}$$

Since we are using a running average over both moments, for the initial few steps, both moments are biased towards initial moments s_0 and r_0.

ADAM algorithm

Require: Learning rate α, decay rate for moment estimates ρ_1 **and** ρ_2, initial parameters θ_t, small constant δ(usually 10^{-8} for numeric stability), training dataset $\mathcal{D}_{tr}$

Initialize first and second moment estimates $r_{t-1} = 0, s_{t-1} = 0$
while stopping criterion not met **do**
  Initialize gradients $\Delta\theta_t = 0$
  for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
    Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
    Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
  end for
  Update first moment estimate $s_t = (\rho_1)r_{t-1} + (1-\rho_1)(\Delta\theta_t)$
  Update second moment estimate $r_t = (\rho_2)r_{t-1} + (1-\rho_2)(\Delta\theta_t)^2$
  Correct for biases $\tilde{s_t} = \frac{s_t}{1-\rho_1^t}, \tilde{r_t} = \frac{r_t}{1-\rho_2^t}$
  Apply update $\theta_{t+1} = \theta_t - \alpha\frac{\tilde{s_t}}{\delta+\sqrt{\tilde{r_t}}}$
end while

Training NNs with SGD: Challenges

Issues that we might encounter when traversing error surfaces using GD

- Plateaus and flat regions
- Local minima and saddle points
- Vanishing and exploding gradients/cliffs
- Other challenges[a]
  - Ill-conditioning[b]
  - Inexact gradients
  - Poor correspondence between local and global structure
  - Choosing learning rate and other hyperparameters

[a] http://www.deeplearningbook.org/contents/optimization.html
[b] ftp://ftp.sas.com/pub/neural/illcond/illcond.html

Training NNs with SGD: Challenges

Given the issues:

- Cost surface is often non-quadratic, non-convex, high-dimensional
- Potentially, many minima and flat regions
- No strong guarantees that
  - Network will converge to a good solution
  - Convergence is swift
  - Convergence occurs at all

But it works!

Homework

Readings

- DL Book Chapter 8
- https://cs231n.github.io/neural-networks-3/sgd

Questions

- How to know if you are in a local minima (or any other critical point on the loss surface)?
- Why is training deep neural networks using GD and Mean-Squared Error (MSE) as cost function, a non-convex optimization problem?
- Let us assume a linear deep neural network (only linear activation functions, $f(x) = x$). Would using GD and MSE still be a non-convex problem?

References

 Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Deep Learning for Computer Vision

Regularization in Neural Networks

Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Review

Questions

- How to know if you are in a local minima (or any other critical point on the loss surface)?

Review

Questions

- How to know if you are in a local minima (or any other critical point on the loss surface)? We will see this in this lecture
- Why is training deep neural networks using GD and Mean-Squared Error (MSE) as cost function, a non-convex optimization problem?

Review

Questions

- How to know if you are in a local minima (or any other critical point on the loss surface)? We will see this in this lecture
- Why is training deep neural networks using GD and Mean-Squared Error (MSE) as cost function, a non-convex optimization problem? Non-linear activation functions can make the problem non-convex
- Let us assume a linear deep neural network (only linear activation functions, $f(x) = x$). Would using GD and MSE still be a non-convex problem?

Review

Questions

- How to know if you are in a local minima (or any other critical point on the loss surface)? We will see this in this lecture
- Why is training deep neural networks using GD and Mean-Squared Error (MSE) as cost function, a non-convex optimization problem? Non-linear activation functions can make the problem non-convex
- Let us assume a linear deep neural network (only linear activation functions, $f(x) = x$). Would using GD and MSE still be a non-convex problem? Yes! Weight symmetry is a reason

What is Regularization? An Intuitive View

What's my rule?

- 1 2 3 $\implies$ satisfies rule
- 4 5 6 $\implies$ satisfies rule
- 7 8 9 $\implies$ satisfies rule
- 9 2 31 $\implies$ does not satisfy rule

What is Regularization? An Intuitive View

What's my rule?

- 1 2 3 $\implies$ satisfies rule
- 4 5 6 $\implies$ satisfies rule
- 7 8 9 $\implies$ satisfies rule
- 9 2 31 $\implies$ does not satisfy rule

Plausible rules

- 3 consecutive single digits
- 3 consecutive integers
- 3 numbers in ascending order
- 3 numbers whose sum is less than 25
- 3 numbers < 10
- 1, 4, or 7 in first column
- "yes" to first 3 sequences, "no" to all others

What is Regularization? An Intuitive View

What's my rule?

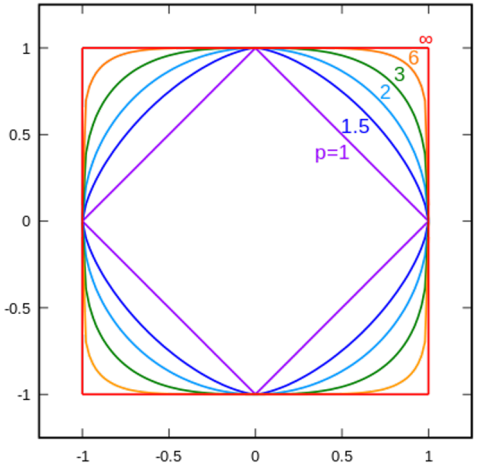
- 1 2 3 $\Longrightarrow$ satisfies rule
- 4 5 6 $\Longrightarrow$ satisfies rule
- 7 8 9 $\Longrightarrow$ satisfies rule
- 9 2 31 $\Longrightarrow$ does not satisfy rule

Plausible rules

- 3 consecutive single digits
- 3 consecutive integers
- 3 numbers in ascending order
- 3 numbers whose sum is less than 25
- 3 numbers < 10
- 1, 4, or 7 in first column
- "yes" to first 3 sequences, "no" to all others

Regularization corresponds to methods used in machine learning to improve **generalization performance** (avoid overfitting to training data)

Review: L_p Norms



- All p-norms penalize larger weights
- $p < 2$ tends to create sparse (i.e. lots of 0 weights)
- $p > 2$ tends to like similar weights

Credit: David Kauchak, Pomona Univ

L2 Regularization

- Penalty on L2-norm of parameters commonly known as L_2 **weight decay**, or simply **weight decay**
- Loss function defined by:

$$\widetilde{\mathcal{L}}(w) = \mathcal{L}(w) + \frac{\alpha}{2} \|w\|^2$$

Credit: Ali Ghodsi, University of Waterloo; Mitesh Khapra, IIT Madras

L2 Regularization

- Penalty on L2-norm of parameters commonly known as L_2 **weight decay**, or simply **weight decay**
- Loss function defined by:

$$\widetilde{\mathcal{L}}(w) = \mathcal{L}(w) + \frac{\alpha}{2} \|w\|^2$$

- Gradient of total objective function:

$$\nabla \widetilde{\mathcal{L}}(w) = \nabla \mathcal{L}(w) + \alpha w$$

Credit: Ali Ghodsi, University of Waterloo; Mitesh Khapra, IIT Madras

L2 Regularization

- Penalty on L2-norm of parameters commonly known as L_2 **weight decay**, or simply **weight decay**
- Loss function defined by:

$$\widetilde{\mathcal{L}}(w) = \mathcal{L}(w) + \frac{\alpha}{2} \|w\|^2$$

- Gradient of total objective function:

$$\nabla \widetilde{\mathcal{L}}(w) = \nabla \mathcal{L}(w) + \alpha w$$

- Update rule:

$$w_{t+1} = w_t - \eta \nabla \mathcal{L}(w_t) - \eta \alpha w_t$$

Credit: Ali Ghodsi, University of Waterloo; Mitesh Khapra, IIT Madras

L2 Regularization: Analysis

- Let w^* be optimal solution in absence of regularization [i.e. $\nabla \mathcal{L}(w^*) = 0$]

L2 Regularization: Analysis

- Let w^* be optimal solution in absence of regularization [i.e. $\nabla \mathcal{L}(w^*) = 0$]
- Consider $u = w - w^*$. Using Taylor series approximation upto 2^{nd} order:

$$\mathcal{L}(w^* + u) = \mathcal{L}(w^*) + u^T \nabla \mathcal{L}(w^*) + \frac{1}{2} u^T H u$$

L2 Regularization: Analysis

- Let w^* be optimal solution in absence of regularization [i.e. $\nabla\mathcal{L}(w^*) = 0$]
- Consider $u = w - w^*$. Using Taylor series approximation upto 2^{nd} order:

$$\mathcal{L}(w^* + u) = \mathcal{L}(w^*) + u^T\nabla\mathcal{L}(w^*) + \frac{1}{2}u^T H u$$

$$\begin{aligned}\mathcal{L}(w) &= \mathcal{L}(w^*) + (w - w^*)^T\nabla\mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \\ &= \mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \qquad (\because \nabla\mathcal{L}(w^*) = 0)\end{aligned} \tag{1}$$

L2 Regularization: Analysis

- Let w^* be optimal solution in absence of regularization [i.e. $\nabla\mathcal{L}(w^*) = 0$]
- Consider $u = w - w^*$. Using Taylor series approximation upto 2^{nd} order:

$$\mathcal{L}(w^* + u) = \mathcal{L}(w^*) + u^T\nabla\mathcal{L}(w^*) + \frac{1}{2}u^T H u$$

$$\begin{aligned}\mathcal{L}(w) &= \mathcal{L}(w^*) + (w - w^*)^T\nabla\mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \\ &= \mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \qquad (\because \nabla\mathcal{L}(w^*) = 0)\end{aligned} \tag{1}$$

- Taking derivative:

$$\nabla\mathcal{L}(w) = \nabla\mathcal{L}(w^*) + H(w - w^*) = H(w - w^*)$$

L2 Regularization: Analysis

- Let w^* be optimal solution in absence of regularization [i.e. $\nabla\mathcal{L}(w^*) = 0$]
- Consider $u = w - w^*$. Using Taylor series approximation upto 2^{nd} order:

$$\mathcal{L}(w^* + u) = \mathcal{L}(w^*) + u^T\nabla\mathcal{L}(w^*) + \frac{1}{2}u^T H u$$

$$\begin{aligned}\mathcal{L}(w) &= \mathcal{L}(w^*) + (w - w^*)^T\nabla\mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \\ &= \mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \qquad (\because \nabla\mathcal{L}(w^*) = 0)\end{aligned} \tag{1}$$

- Taking derivative:

$$\nabla\mathcal{L}(w) = \nabla\mathcal{L}(w^*) + H(w - w^*) = H(w - w^*)$$

- Gradient of total objective function (in presence of L2 regularization):

$$\nabla\widetilde{\mathcal{L}}(w) = \nabla\mathcal{L}(w) + \alpha w = H(w - w^*) + \alpha w$$

Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

$$\therefore (H + \alpha\mathbb{I})\widetilde{w} = Hw^*$$

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

$$\therefore (H + \alpha\mathbb{I})\widetilde{w} = Hw^*$$

$$\implies \widetilde{w} = (H + \alpha\mathbb{I})^{-1}Hw^*$$

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

$$\therefore (H + \alpha\mathbb{I})\widetilde{w} = Hw^*$$

$$\implies \widetilde{w} = (H + \alpha\mathbb{I})^{-1}Hw^*$$

- If $\alpha \to 0$, then $\widetilde{w} \to w^*$; and therefore no regularization

L2 Regularization: Analysis

- What happens when $\alpha \neq 0$?

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

$$\therefore (H + \alpha\mathbb{I})\widetilde{w} = Hw^*$$

$$\implies \widetilde{w} = (H + \alpha\mathbb{I})^{-1}Hw^*$$

- If $\alpha \to 0$, then $\widetilde{w} \to w^*$; and therefore no regularization

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

$$\therefore (H + \alpha\mathbb{I})\widetilde{w} = Hw^*$$

$$\implies \widetilde{w} = (H + \alpha\mathbb{I})^{-1}Hw^*$$

- If $\alpha \to 0$, then $\widetilde{w} \to w^*$; and therefore no regularization

- What happens when $\alpha \neq 0$? If H is symmetric positive semidefinite, $H = Q\Lambda Q^T$, where Q is orthogonal i.e. $QQ^T = Q^TQ = \mathbb{I}$

L2 Regularization: Analysis

- Let $\widetilde{w}$ be optimal solution in presence of regularization, i.e. $\nabla\widetilde{\mathcal{L}}(\widetilde{w}) = 0$

$$H(\widetilde{w} - w^*) + \alpha\widetilde{w} = 0$$

$$\therefore (H + \alpha\mathbb{I})\widetilde{w} = Hw^*$$

$$\implies \widetilde{w} = (H + \alpha\mathbb{I})^{-1}Hw^*$$

- If $\alpha \to 0$, then $\widetilde{w} \to w^*$; and therefore no regularization

- What happens when $\alpha \neq 0$? If H is symmetric positive semidefinite, $H = Q\Lambda Q^T$, where Q is orthogonal i.e. $QQ^T = Q^TQ = \mathbb{I}$

$$\begin{aligned}
\widetilde{w} &= (H + \alpha\mathbb{I})^{-1}Hw^* \\
&= (Q\Lambda Q^T + \alpha\mathbb{I})^{-1}Q\Lambda Q^Tw^* \\
&= (Q\Lambda Q^T + \alpha Q\mathbb{I}Q^T)^{-1}Q\Lambda Q^Tw^* \\
&= [Q(\Lambda + \alpha\mathbb{I})Q^T]^{-1}Q\Lambda Q^Tw^* \\
&= Q^{T^{-1}}(\Lambda + \alpha\mathbb{I})^{-1}Q^{-1}Q\Lambda Q^Tw^* \\
&= Q(\Lambda + \alpha\mathbb{I})^{-1}\Lambda Q^Tw^* (\because Q^{T^{-1}} = Q)
\end{aligned}$$

Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

L2 Regularization: Analysis

- Each element i of $Q^T w^*$ gets scaled by $\frac{\lambda_i}{\lambda_i + \alpha}$ before it is rotated back by Q

L2 Regularization: Analysis

- Each element i of $Q^T w^*$ gets scaled by $\frac{\lambda_i}{\lambda_i+\alpha}$ before it is rotated back by Q
- If $\lambda_i >> \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 1$

L2 Regularization: Analysis

- Each element i of $Q^T w^*$ gets scaled by $\frac{\lambda_i}{\lambda_i+\alpha}$ before it is rotated back by Q
- If $\lambda_i >> \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 1$
- If $\lambda_i << \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 0$

L2 Regularization: Analysis

- Each element i of $Q^T w^*$ gets scaled by $\frac{\lambda_i}{\lambda_i+\alpha}$ before it is rotated back by Q
- If $\lambda_i >> \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 1$
- If $\lambda_i << \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 0$
- Thus, only significant directions (larger eigenvalues) will be retained.

$$\text{Effective parameters} = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda_i + \alpha} < n$$

L2 Regularization: Analysis

- Each element i of $Q^T w^*$ gets scaled by $\frac{\lambda_i}{\lambda_i+\alpha}$ before it is rotated back by Q
- If $\lambda_i >> \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 1$
- If $\lambda_i << \alpha$ then $\frac{\lambda_i}{\lambda_i+\alpha} = 0$
- Thus, only significant directions (larger eigenvalues) will be retained.

$$\text{Effective parameters} = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda_i + \alpha} < n$$

- Summary: The weight vector (w^*) is getting rotated to $\widetilde{w}$ on using L2 regularization. All of its elements are shrinking but some are shrinking more than the others. This ensures that only important features are given high weights.

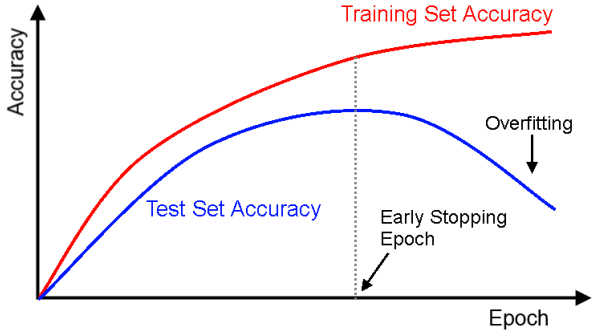
Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

Early Stopping

- **Simple idea:** keep monitoring cost function, and not let it become too low consistently; stop at an earlier iteration

Early Stopping

- **Simple idea:** keep monitoring cost function, and not let it become too low consistently; stop at an earlier iteration



Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs $\rightarrow$ Bad idea: can't assume one-size-fits-all approach

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs → Bad idea: can't assume one-size-fits-all approach
- **Error-change criterion:**
  - Stop when error isn't dropping over a window of, say, 10 epochs

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs → Bad idea: can't assume one-size-fits-all approach
- **Error-change criterion:**
  - Stop when error isn't dropping over a window of, say, 10 epochs
  - Train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs → Bad idea: can't assume one-size-fits-all approach
- **Error-change criterion:**
  - Stop when error isn't dropping over a window of, say, 10 epochs
  - Train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)
- **Weight-change criterion:**
  - Compare weights at epochs $t-10$ and t and test: $\max_i \|w_i^t - w_i^{t-10}\| < \rho$

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs → Bad idea: can't assume one-size-fits-all approach
- **Error-change criterion:**
  - Stop when error isn't dropping over a window of, say, 10 epochs
  - Train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)
- **Weight-change criterion:**
  - Compare weights at epochs $t - 10$ and t and test: $\max_i \|w_i^t - w_i^{t-10}\| < \rho$
  - Don't base on length of overall weight change vector

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs → Bad idea: can't assume one-size-fits-all approach
- **Error-change criterion:**
  - Stop when error isn't dropping over a window of, say, 10 epochs
  - Train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)
- **Weight-change criterion:**
  - Compare weights at epochs $t-10$ and t and test: $\max_i \|w_i^t - w_i^{t-10}\| < \rho$
  - Don't base on length of overall weight change vector
  - Possibly express as a percentage of the weight

Dataset Augmentation

- Creation of data using some knowledge of task

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)
  - Rotations

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)
  - Rotations
  - Color changes

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)
  - Rotations
  - Color changes
  - Noise injection

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)
  - Rotations
  - Color changes
  - Noise injection
  - Mirroring

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)
  - Rotations
  - Color changes
  - Noise injection
  - Mirroring
- Helps increase data; is useful when training data provided is less (DNNs need large amounts of training data to work!)

Dataset Augmentation

- Creation of data using some knowledge of task
- Exploit the fact that certain transformations to image do not change label of image
- Methods:
  - Data jittering (E.g. Distortion and blurring of images)
  - Rotations
  - Color changes
  - Noise injection
  - Mirroring
- Helps increase data; is useful when training data provided is less (DNNs need large amounts of training data to work!)
- Also acts as regularizer (by avoiding overfitting to provided data)

Dataset Augmentation: Example[1]

[1] Wu, Ren, et al. "Deep image: Scaling up image recognition." arXiv 2015

Dataset Augmentation: Example[1]

[1]Wu, Ren, et al. "Deep image: Scaling up image recognition." arXiv 2015

Data Augmentation: Newer Methods

Mixup

- Create virtual training examples as below ($\lambda \in [0, 1]$):

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

where x_i,x_j are input vectors, and y_i,y_j are one-hot label encodings

- *Variants:* Manifold Mixup, AugMix

CutOut

- Randomly mask out square regions of input during training

- *Variants:* CutMix

Noise Injection

Injecting noise is another form of regularization; has shown impressive results in certain applications:

- **Data Noise**

[2] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995

Noise Injection

Injecting noise is another form of regularization; has shown impressive results in certain applications:

- **Data Noise**
  - Addd noise to data while training

[2] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995

Noise Injection

Injecting noise is another form of regularization; has shown impressive results in certain applications:

- **Data Noise**
  - Addd noise to data while training
  - Adding Gaussian noise to input is equivalent to weight decay (L_2 regularisation) when loss function is sum-of-squared error[2]. Homework!

[2] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995

Noise Injection

Injecting noise is another form of regularization; has shown impressive results in certain applications:

- **Data Noise**
  - Addd noise to data while training
  - Adding Gaussian noise to input is equivalent to weight decay (L_2 regularisation) when loss function is sum-of-squared error[2]. Homework!
  - Can be interpreted as form of **dataset augmentation.**

[2] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995

Noise Injection

Injecting noise is another form of regularization; has shown impressive results in certain applications:

- **Data Noise**
  - Addd noise to data while training
  - Adding Gaussian noise to input is equivalent to weight decay (L_2 regularisation) when loss function is sum-of-squared error[2]. Homework!
  - Can be interpreted as form of **dataset augmentation.**
- **Label Noise**

[2] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995

Noise Injection

Injecting noise is another form of regularization; has shown impressive results in certain applications:

- **Data Noise**
  - Addd noise to data while training
  - Adding Gaussian noise to input is equivalent to weight decay (L_2 regularisation) when loss function is sum-of-squared error[2]. Homework!
  - Can be interpreted as form of **dataset augmentation.**
- **Label Noise**
- **Gradient Noise**

[2] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995

Regularization through Label Noise[3]

- Disturb each training sample with probability α.
- For each disturbed sample, label randomly drawn from uniform distribution over $\{1, 2, \cdots, C\}$, regardless of true label

Algorithm 1 DisturbLabel

Input: $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$, noise rate α.
Initialization: a network model $\mathbb{M}$: $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}_0) \in \mathbb{R}^C$;
for each mini-batch $\mathcal{D}_t = \{(\mathbf{x}_m, \mathbf{y}_m)\}_{m=1}^{M}$ **do**
    for each sample $(\mathbf{x}_m, \mathbf{y}_m)$ **do**
        Generate a disturbed label $\widetilde{\mathbf{y}}_m$ with Eqn (2);
    end for
    Update the parameters $\boldsymbol{\theta}_t$ with Eqn (1);
end for
Output: the trained model $\mathbb{M}'$: $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}_T) \in \mathbb{R}^C$.

$$\begin{cases} \widetilde{c} & \sim & \mathcal{P}(\alpha), \\ \widetilde{y}_{\widetilde{c}} & = & 1, \\ \widetilde{y}_i & = & 0, \qquad \forall i \neq \widetilde{c}. \end{cases} \tag{2}$$

[3] Xie, Lingxi, et al. "DisturbLabel: Regularizing CNN on the Loss Layer." CVPR 2016.

Regularization through Gradient Noise[4]

- **Idea:** Add noise to gradient

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

[4] Neelakantan, Arvind, et al. "Adding gradient noise improves learning for very deep networks." arXiv preprint arXiv:1511.06807 (2015)

Regularization through Gradient Noise[4]

- **Idea:** Add noise to gradient

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

- Annealed Gaussian noise by decaying variance

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

[4] Neelakantan, Arvind, et al. "Adding gradient noise improves learning for very deep networks." arXiv preprint arXiv:1511.06807 (2015)

Regularization through Gradient Noise[4]

- **Idea:** Add noise to gradient

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

- Annealed Gaussian noise by decaying variance

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

- Showed significant improvement in performance in certain applications

[4] Neelakantan, Arvind, et al. "Adding gradient noise improves learning for very deep networks." arXiv preprint arXiv:1511.06807 (2015)

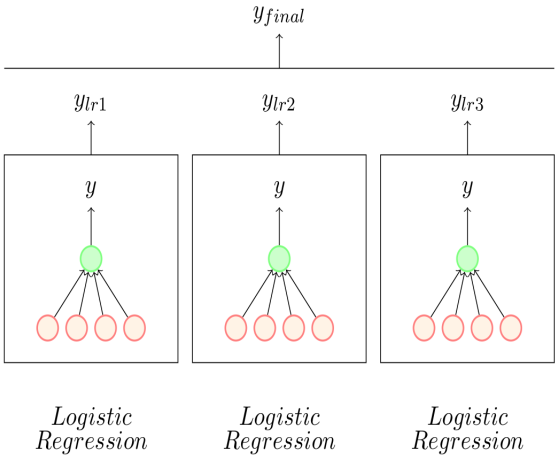
Ensemble Methods

- Train several different models separately, then have all models vote on the output for test examples
- An example of a general strategy in machine learning called **model averaging**
- Models can correspond to different classifiers, or could be different instances of same classifier trained with:
  - different hyperparameters
  - different features
  - different samples of the training data

Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

Ensemble Methods

- **Bagging** (short for bootstrap aggregating): reduces generalization error by forming an ensemble using different instances of same classifier
- For e.g., consider a set of k logistic regression models
- Given a dataset, construct multiple training sets by sampling with replacement $(T_1, T_2, ..., T_k)$
- Train i^{th} instance of classifier using training set T_i



Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

Ensemble Methods

- Suppose that each model makes an error ε_i on a test example

Ensemble Methods

- Suppose that each model makes an error ε_i on a test example
- Let ε_i be drawn from a zero-mean multivariate normal distribution with:

$$\text{Variance} = \mathbb{E}[\varepsilon_i^2] = V$$
$$\text{Covariance} = \mathbb{E}[\varepsilon_i \varepsilon_j] = C$$

Ensemble Methods

- Suppose that each model makes an error ε_i on a test example
- Let ε_i be drawn from a zero-mean multivariate normal distribution with:

$$\text{Variance} = \mathbb{E}[\varepsilon_i^2] = V$$
$$\text{Covariance} = \mathbb{E}[\varepsilon_i \varepsilon_j] = C$$

- Error made by average prediction of all models is $\frac{1}{k} \sum_i \varepsilon_i$

Ensemble Methods

- Suppose that each model makes an error ε_i on a test example
- Let ε_i be drawn from a zero-mean multivariate normal distribution with:

$$\text{Variance} = \mathbb{E}[\varepsilon_i^2] = V$$
$$\text{Covariance} = \mathbb{E}[\varepsilon_i \varepsilon_j] = C$$

- Error made by average prediction of all models is $\frac{1}{k}\sum_i \varepsilon_i$

Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

- Expected squared error of ensemble predictor is:

$$\begin{aligned} MSE &= \mathbb{E}\left[\left(\frac{1}{k}\sum_i \varepsilon_i\right)^2\right] \\ &= \frac{1}{k^2}\mathbb{E}\left[\sum_i \sum_{j=i} \varepsilon_i \varepsilon_j + \sum_i \sum_{j\neq i} \varepsilon_i \varepsilon_j\right] \\ &= \frac{1}{k^2}\mathbb{E}\left[\sum_i \varepsilon_i^2 + \sum_i \sum_{j\neq i} \varepsilon_i \varepsilon_j\right] \\ &= \frac{1}{k^2}(kV + k(k-1)C) \\ &= \frac{1}{k}V + \frac{k-1}{k}C \end{aligned}$$

Ensemble Methods

- Expected squared error of ensemble predictor:

$$MSE = \frac{1}{k}V + \frac{k-1}{k}C$$

What does this tell you?

Ensemble Methods

- Expected squared error of ensemble predictor:

$$MSE = \frac{1}{k}V + \frac{k-1}{k}C$$

  What does this tell you?

- If errors of model are perfectly correlated, then $V = C$ and $MSE = V$, i.e. bagging does not help: the MSE of ensemble is as bad as individual models

Ensemble Methods

- Expected squared error of ensemble predictor:

$$MSE = \frac{1}{k}V + \frac{k-1}{k}C$$

What does this tell you?

- If errors of model are perfectly correlated, then $V = C$ and $MSE = V$, i.e. bagging does not help: the MSE of ensemble is as bad as individual models
- If errors of model are independent or uncorrelated, then $C = 0$ and MSE of ensemble reduces to $\frac{1}{k}V$
- On average, **ensemble will perform at least as well as its individual members**

Credit: Ali Ghodsi, Univ of Waterloo; Mitesh Khapra, IIT Madras

Dropout

- One major issue in learning large neural networks is **co-adaptation**
  - As network is trained iteratively, powerful connections are learned more while weaker ones are ignored
  - After many iterations, only a fraction of node connections participate
  - Therefore, expanding neural network size may not help

Dropout

- One major issue in learning large neural networks is **co-adaptation**
  - As network is trained iteratively, powerful connections are learned more while weaker ones are ignored
  - After many iterations, only a fraction of node connections participate
  - Therefore, expanding neural network size may not help
- **Dropout**: Regularization method to address this issue

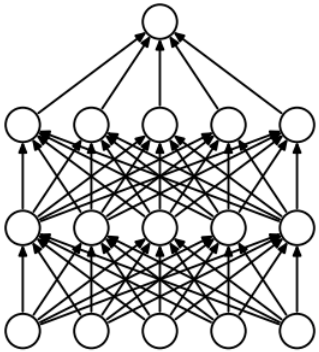
Dropout

- One major issue in learning large neural networks is **co-adaptation**
  - As network is trained iteratively, powerful connections are learned more while weaker ones are ignored
  - After many iterations, only a fraction of node connections participate
  - Therefore, expanding neural network size may not help
- **Dropout**: Regularization method to address this issue
- **Training Phase:** For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, p, of nodes (and corresponding activations)

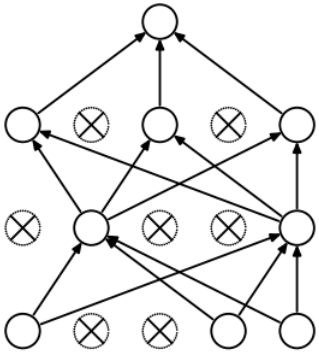
Dropout

- One major issue in learning large neural networks is **co-adaptation**
  - As network is trained iteratively, powerful connections are learned more while weaker ones are ignored
  - After many iterations, only a fraction of node connections participate
  - Therefore, expanding neural network size may not help
- **Dropout**: Regularization method to address this issue
- **Training Phase:** For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, p, of nodes (and corresponding activations)
- **Test Phase:** Use all activations, but reduce them by factor p (to account for missing activations during training)

Dropout



(a) Standard Neural Net



(b) After applying dropout.

[5]

[5] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

Dropout

- Recall ensemble methods: with H hidden units, each of which can be dropped, we can have 2^H possible models

Dropout

- Recall ensemble methods: with H hidden units, each of which can be dropped, we can have 2^H possible models
- This is prohibitively large and we cannot possibly train so many networks

Dropout

- Recall ensemble methods: with H hidden units, each of which can be dropped, we can have 2^H possible models
- This is prohibitively large and we cannot possibly train so many networks
- Trick: Each of the 2^{H-1} models that includes hidden unit h must share the same weight for the unit
  - serves as a form of regularization
  - makes the models cooperate

Dropout

- Recall ensemble methods: with H hidden units, each of which can be dropped, we can have 2^H possible models
- This is prohibitively large and we cannot possibly train so many networks
- Trick: Each of the 2^{H-1} models that includes hidden unit h must share the same weight for the unit
  - serves as a form of regularization
  - makes the models cooperate
- Including all hidden units at test time with scaling of $\frac{1}{2}$ is equivalent to computing geometric mean of all 2^H models.

Dropout

- Recall ensemble methods: with H hidden units, each of which can be dropped, we can have 2^H possible models
- This is prohibitively large and we cannot possibly train so many networks
- Trick: Each of the 2^{H-1} models that includes hidden unit h must share the same weight for the unit
  - serves as a form of regularization
  - makes the models cooperate
- Including all hidden units at test time with scaling of $\frac{1}{2}$ is equivalent to computing geometric mean of all 2^H models. (How?)

Dropout for Single Non-Linear Unit

- Consider a logistic sigmoidal function on input x:

$$o = \sigma(x) = \frac{1}{1 + ce^{-\lambda x}} \qquad c >= 0$$

Dropout for Single Non-Linear Unit

- Consider a logistic sigmoidal function on input x:

$$o = \sigma(x) = \frac{1}{1 + ce^{-\lambda x}} \qquad c >= 0$$

- 2^n possible sub-networks indexed by k

Dropout for Single Non-Linear Unit

- Consider a logistic sigmoidal function on input x:

$$o = \sigma(x) = \frac{1}{1 + ce^{-\lambda x}} \qquad c >= 0$$

- 2^n possible sub-networks indexed by k
- Geometric mean of outputs from all sub-networks: $G = \prod_k o_k^{\frac{1}{2^n}}$

Dropout for Single Non-Linear Unit

- Consider a logistic sigmoidal function on input x:

$$o = \sigma(x) = \frac{1}{1 + ce^{-\lambda x}} \qquad c >= 0$$

- 2^n possible sub-networks indexed by k
- Geometric mean of outputs from all sub-networks: $G = \prod_k o_k^{\frac{1}{2^n}}$
- Geometric mean of complementary outputs: $G' = \prod_k (1 - o_k)^{\frac{1}{2^n}}$

Dropout for Single Non-Linear Unit

- Consider a logistic sigmoidal function on input x:

$$o = \sigma(x) = \frac{1}{1 + ce^{-\lambda x}} \qquad c >= 0$$

- 2^n possible sub-networks indexed by k
- Geometric mean of outputs from all sub-networks: $G = \prod_k o_k^{\frac{1}{2^n}}$
- Geometric mean of complementary outputs: $G' = \prod_k (1 - o_k)^{\frac{1}{2^n}}$

- Normalized geometric mean $NGM = \frac{G}{G+G'}$. Hence:

$$\begin{aligned} NGM &= \frac{1}{1 + \left(\prod_k \frac{1-\sigma(x_k)}{\sigma(x_k)}\right)^{\frac{1}{2^n}}} \\ &= \frac{1}{1 + \left(\prod_k ce^{-\lambda x_k}\right)^{\frac{1}{2^n}}} \left(\because \frac{1-\sigma(x)}{\sigma(x)} = ce^{-\lambda x} \right) \\ &= \frac{1}{1 + c[e^{-\lambda \sum_k x_k/2^n}]} = \sigma(\mathbb{E}[x]) \end{aligned}$$

where $\mathbb{E}[x] = \sum_k x_k/2^n$

Dropout for Single Non-Linear Unit

- Consider a logistic sigmoidal function on input x:

$$o = \sigma(x) = \frac{1}{1 + ce^{-\lambda x}} \qquad c >= 0$$

- 2^n possible sub-networks indexed by k
- Geometric mean of outputs from all sub-networks: $G = \prod_k o_k^{\frac{1}{2^n}}$
- Geometric mean of complementary outputs: $G' = \prod_k (1 - o_k)^{\frac{1}{2^n}}$

- Normalized geometric mean $NGM = \frac{G}{G+G'}$. Hence:

$$\begin{aligned} NGM &= \frac{1}{1 + \left(\prod_k \frac{1-\sigma(x_k)}{\sigma(x_k)}\right)^{\frac{1}{2^n}}} \\ &= \frac{1}{1 + \left(\prod_k ce^{-\lambda x_k}\right)^{\frac{1}{2^n}}} \left(\because \frac{1-\sigma(x)}{\sigma(x)} = ce^{-\lambda x} \right) \\ &= \frac{1}{1 + c[e^{-\lambda \sum_k x_k/2^n}]} = \sigma(\mathbb{E}[x]) \end{aligned}$$

where $\mathbb{E}[x] = \sum_k x_k/2^n$

NGM is equivalent to output of overall network with weights divided by two

Homework

Readings

- Deep Learning book, Chapter 7, Sections 7.1-7.5, 7.8, 7.12
- Tutorial on Dropout for more information (Optional)

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE.

References I

Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580.

Pierre Baldi and Peter Sadowski. "The dropout learning algorithm". In: *Artificial intelligence* 210 (2014), pp. 78–122.

Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958.

David Warde-Farley et al. "An empirical analysis of dropout in piecewise linear networks". In: *CoRR* abs/1312.6197 (2014).

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Terrance DeVries and Graham W Taylor. "Improved Regularization of Convolutional Neural Networks with Cutout". In: *arXiv preprint arXiv:1708.04552* (2017).

References II

Hongyi Zhang et al. "mixup: Beyond Empirical Risk Minimization". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018.

Ghodsi, Ali, STAT 946 - Deep Learning (Fall 2015). URL: https://uwaterloo.ca/data-analytics/sites/ca.data-analytics/files/uploads/files/f15_stat946_deep_learning_outline_1.pdf (visited on 06/05/2020).

Khapra, Mitesh M., CS 7015 - Deep Learning (Spring 2019). URL: https://www.cse.iitm.ac.in/~miteshk/CS7015.html (visited on 06/03/2020).

Deep Learning for Computer Vision

Improvements in Training of Neural Networks

Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Review

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE

- When we add Gaussian noise to inputs, variance of noise is amplified by squared weight before going to next layer

Review

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE

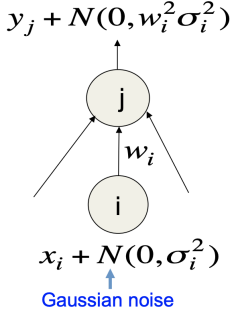
- When we add Gaussian noise to inputs, variance of noise is amplified by squared weight before going to next layer
- In a simple net with a linear output unit directly connected to inputs, the amplified noise gets added to output

Review

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE

- When we add Gaussian noise to inputs, variance of noise is amplified by squared weight before going to next layer
- In a simple net with a linear output unit directly connected to inputs, the amplified noise gets added to output
- This makes an additive contribution to the squared error, i.e. minimizing squared error tends to minimize squared weights when inputs are noisy!



Credit: Tijmen Tieleman, Univ of Toronto

Review

Mathematically, consider one input with added noise. We would have:
$y_{\text{noisy}} = \sum_i w_i x_i + \sum_i w_i \epsilon_i$, where ϵ_i is sampled from $N(0, \sigma_i^2)$. Given expected output t, we have:

$$\begin{aligned}
\mathbb{E}\left[\left(y_{\text{noisy}} - t\right)^2\right] &= \mathbb{E}\left[\left(y + \sum_i w_i\epsilon_i - t\right)^2\right] = \mathbb{E}\left[\left((y-t) + \sum_i w_i\epsilon_i\right)^2\right] \\
&= (y-t)^2 + \mathbb{E}\left[2(y-t)\sum_i w_i\epsilon_i\right] + \mathbb{E}\left[\left(\sum_i w_i\epsilon_i\right)^2\right] \\
&= (y-t)^2 + \mathbb{E}\left[\left(\sum_i w_i\epsilon_i\right)^2\right] \\
&\left(\because \epsilon_i \text{ is independent of } \epsilon_j, \text{ and } \epsilon_i \text{ is independent of } (y-t)\right) \\
&= (y-t)^2 + \sum_i w_i^2\sigma_i^2 \implies \text{Penalty on L2-norm of weights!}
\end{aligned}$$

Activation Functions

- Non-linear function applied to the output of a neuron.
- What characteristics must activation functions possess?

Activation Functions

- Non-linear function applied to the output of a neuron.
- What characteristics must activation functions possess?
- Must be continuous, differentiable (need to perform backpropagation), non-decreasing and easy to compute.

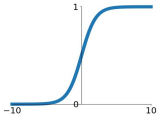
Activation Functions

- Non-linear function applied to the output of a neuron.
- What characteristics must activation functions possess?
- Must be continuous, differentiable (need to perform backpropagation), non-decreasing and easy to compute.
- Common activation functions available- Sigmoid, Tanh, ReLU, etc.
  - Which one to choose?
- What effect does a chosen activation function have on training?

Activation Functions

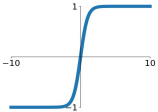
Sigmoid

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



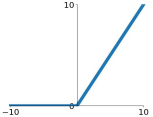
tanh

$$\tanh(x)$$



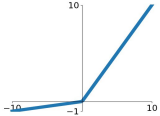
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

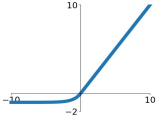


Maxout

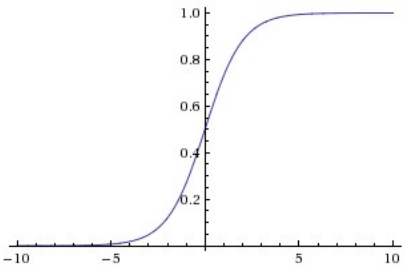
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



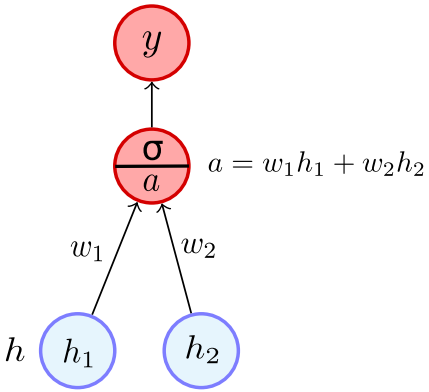
Activation Functions: Sigmoid



Sigmoid

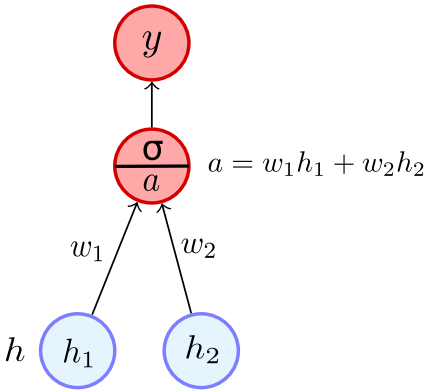
- $\sigma(x) = \frac{1}{1+e^{-x}}$
- Compresses all inputs to range $[0, 1]$
- Gradient of sigmoid function is: $\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$
- Sigmoid activations are rarely used in vanilla NNs today, and more often in specific architectures. Why?

Activation Functions: Sigmoid



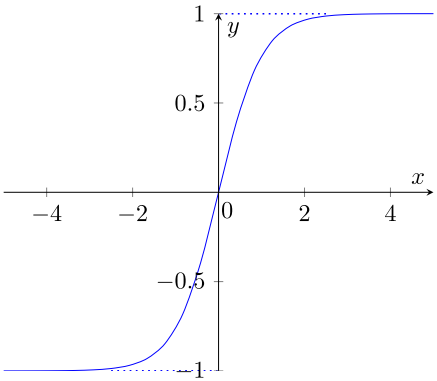
- Gradient of sigmoid neuron at saturation is very small $\implies$ training is slow
- Sigmoid is not zero-centered
- $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_1$
  $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_2$
  If h_1 and h_2 are outputs of sigmoid neurons, they are positive. All gradients at a layer are either positive or negative.
- Computationally expensive, need to compute exponential function

Activation Functions: Sigmoid



- Gradient of sigmoid neuron at saturation is very small $\implies$ training is slow
- Sigmoid is not zero-centered
- $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_1$
  $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_2$
  If h_1 and h_2 are outputs of sigmoid neurons, they are positive. All gradients at a layer are either positive or negative.
- Computationally expensive, need to compute exponential function

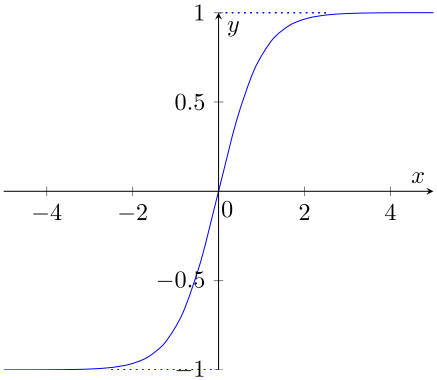
Activation Functions: Tanh



f(x) = tanh(x)

- $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Compresses all inputs to range $[-1, 1]$
- Gradient of tanh function is: $\frac{\partial \tanh(x)}{\partial x} = (1 - \tanh^2(x))$
- What advantage does it have over sigmoid?

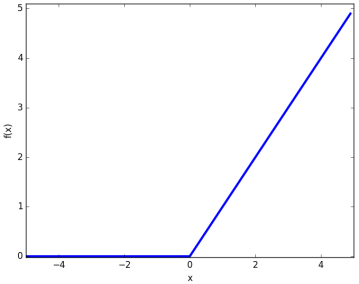
Activation Functions: Tanh



f(x) = tanh(x)

- $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Compresses all inputs to range $[-1, 1]$
- Gradient of tanh function is: $\frac{\partial \tanh(x)}{\partial x} = (1 - \tanh^2(x))$
- What advantage does it have over sigmoid? It is zero-centered
- However, gradient of tanh at saturation also vanishes
- Also computationally expensive due to the exponential

Activation Functions: Rectified Linear Unit (ReLU)



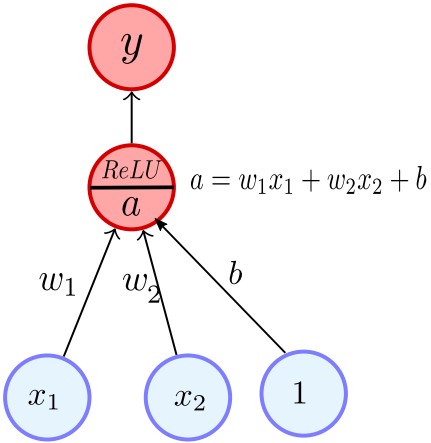
$$f(x) = max(0, x)$$

- Default activation used today
- ReLU computes $f(x) = \max(0, x)$
- Gradient of ReLU is:

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

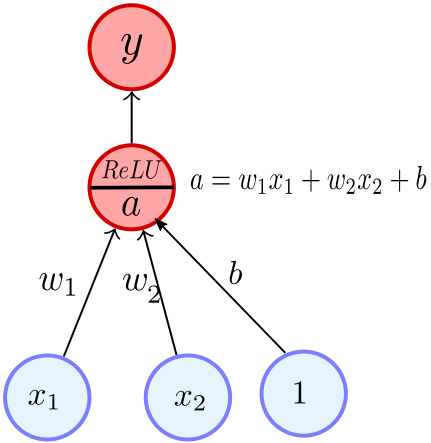
- Non-saturating for positive inputs
- Found to accelerate convergence of SGD compared to sigmoid/tanh functions (by a factor of 6 in AlexNet)
- Computationally inexpensive

The Dying ReLU Problem



- If input to ReLU neuron is negative, output is zero ("**dead neuron**"). How do the gradients flow?
  $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial ReLU} \frac{\partial ReLU}{\partial a} x_1$
- if $a < 0$, gradient term $\frac{\partial ReLU}{\partial a}$ is zero, which makes whole gradient zero. Thus, weights w_1 and w_2 do not get updated

The Dying ReLU Problem



- If input to ReLU neuron is negative, output is zero ("**dead neuron**"). How do the gradients flow?
  $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial ReLU} \frac{\partial ReLU}{\partial a} x_1$
- if $a < 0$, gradient term $\frac{\partial ReLU}{\partial a}$ is zero, which makes whole gradient zero. Thus, weights w_1 and w_2 do not get updated
- Initializing bias to small positive value can help
- Using variants of ReLU (Leaky ReLU, Exponential Linear Unit) can help

Activation Functions: Which one to choose?

- Use ReLU non-linearity, be careful with learning rates and monitor the fraction of 'dead' units in a network.
- Try Leaky ReLU, ELU, Maxout
- Try tanh, but expect worse performance
- Sigmoid not used much, unless a gating is required
- In general, a good idea for an activation function to be in its linear region for most part of training

Credit: Fei-Fei Li, CS231N, Stanford Univ

Weight Initialization

- Why is weight initialization important?

Weight Initialization

- Why is weight initialization important? Recall the neural network error surface. Where you start is critical to which local minima you will reach
- What happens if we initialize network weights to zero?

Weight Initialization

- Why is weight initialization important? Recall the neural network error surface. Where you start is critical to which local minima you will reach
- What happens if we initialize network weights to zero?
  - In fact, any constant initialization scheme will perform very poorly
  - Consider a neural network with two hidden units (with, say, ReLU activation), and assume we initialize all biases to 0 and the weights with some constant α

Weight Initialization

- Why is weight initialization important? Recall the neural network error surface. Where you start is critical to which local minima you will reach
- What happens if we initialize network weights to zero?
  - In fact, any constant initialization scheme will perform very poorly
  - Consider a neural network with two hidden units (with, say, ReLU activation), and assume we initialize all biases to 0 and the weights with some constant α
  - If we forward propagate an input (x_1, x_2) in the network, output of both hidden units will be relu$(\alpha x_1 + \alpha x_2)$!

Weight Initialization

- Why is weight initialization important? Recall the neural network error surface. Where you start is critical to which local minima you will reach
- What happens if we initialize network weights to zero?
  - In fact, any constant initialization scheme will perform very poorly
  - Consider a neural network with two hidden units (with, say, ReLU activation), and assume we initialize all biases to 0 and the weights with some constant α
  - If we forward propagate an input (x_1, x_2) in the network, output of both hidden units will be relu$(\alpha x_1 + \alpha x_2)$!
  - Both hidden units will have identical influence on cost $\implies$ identical gradients
  - Thus, both neurons evolve symmetrically throughout training, preventing different neurons from learning different things

Credit: https://www.deeplearning.ai/ai-notes

Weight Initialization

How are weights initialized then?

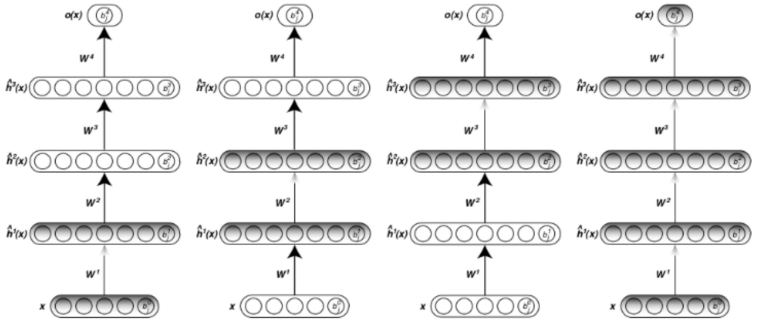
Weight Initialization

How are weights initialized then?

- Typically chosen randomly, e.g. sample weights from a Gaussian distribution
- Both very large and small weights can cause certain activation functions (sigmoid/tanh) to saturate, leading to very small gradients

Weight Initialization through Unsupervised Pretraining

- In 2006, Salakhutdinov and Hinton[1] introduced **greedy layerwise unsupervised training** → largely considered to be a significant catalyst in initial success of DNNs at that time (now replaced by better weight initialization methods)



(a) First hidden layer pre-training (b) Second hidden layer pre-training (c) Third hidden layer pre-training (d) Fine-tuning of whole network

[1] Salakhutdinov and Hinton, "Reducing the Dimensionality of Data with Neural Networks", Science, 2006

Weight Initialization through Unsupervised Pretraining

Optimizes each piece independently rather than jointly

Greedy +

Independent pieces are layers of network; training proceeds one layer at a time

Layerwise +

Without labels, only data

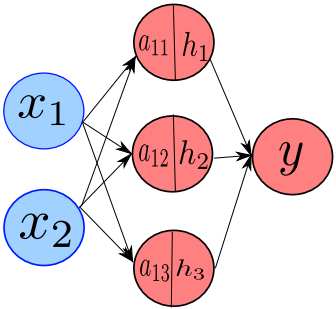
Unsupervised +

only a first step before training the entire model with labels

Pretraining

Achieved using networks called autoencoders or Restricted Boltzmann Machines (or equivalent), which we will see later

Newer Weight Initialization Methods

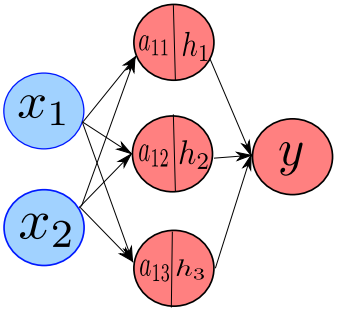


$$Var(a_i) = nVar(w)Var(x)$$

- Consider inputs with mean 0 and variance 1, weights with mean 0 and variance. Note that $a_1 = \sum_{i=1}^{n} w_{i1}x_i$. Then:

$$\text{Var}(a_1) = \sum_{i=1}^{n} \text{Var}(w_{i1}x_i) = n\text{Var}(w)\text{Var}(x)$$

Newer Weight Initialization Methods



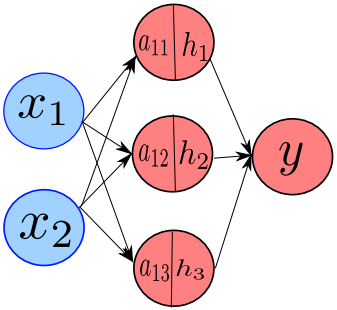
$$Var(a_i) = nVar(w)Var(x)$$

- Consider inputs with mean 0 and variance 1, weights with mean 0 and variance. Note that $a_1 = \sum_{i=1}^{n} w_{i1}x_i$. Then:

$$\text{Var}(a_1) = \sum_{i=1}^{n} \text{Var}(w_{i1}x_i) = n\text{Var}(w)\text{Var}(x)$$

- What happens to variance when we go deeper?

Newer Weight Initialization Methods



$$Var(a_i) = nVar(w)Var(x)$$

- Consider inputs with mean 0 and variance 1, weights with mean 0 and variance. Note that $a_1 = \sum_{i=1}^{n} w_{i1}x_i$. Then:

$$\text{Var}(a_1) = \sum_{i=1}^{n} \text{Var}(w_{i1}x_i) = n\text{Var}(w)\text{Var}(x)$$

- What happens to variance when we go deeper?

$$\text{Var}(a_{ik}) = (n\text{Var}(W))^k\text{Var}(x)$$

- To prevent variance of any layer from blowing up or becoming zero, we need $n\text{Var}(w) = 1$
- For a good initialization, weights can be drawn from standard normal distribution and then scaled by $\frac{1}{\sqrt{n}}$, where n is node's fan-in

Weight Initialization

Most recommended today (removed the need for unsupervised pre-training):

- **Xavier's (or Glorot's) initialization**[2]: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
- **He's initialization**[3]: $\texttt{uniform}(-\frac{4}{fan_{in}+fan_{out}}, \frac{4}{fan_{in}+fan_{out}})$

[2] Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTATS 2010

[3] He et al, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", CVPR 2015

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$
  - Considering the backward pass during backprop, it may be wise to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{out}}}$ too; hence, go with the average $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$
  - Considering the backward pass during backprop, it may be wise to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{out}}}$ too; hence, go with the average $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$
  - If $w \sim \texttt{Uniform}[-a, a]$, $\text{Var}(w) = \frac{(2a)^2}{12} = \frac{(a)^2}{3}$ ($\because$ variance of uniform distribution in interval $[m, n]$ is $\frac{(n-m)^2}{12}$)

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$
  - Considering the backward pass during backprop, it may be wise to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{out}}}$ too; hence, go with the average $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$
  - If $w \sim \texttt{Uniform}[-a, a]$, $\text{Var}(w) = \frac{(2a)^2}{12} = \frac{(a)^2}{3}$ ($\because$ variance of uniform distribution in interval $[m, n]$ is $\frac{(n-m)^2}{12}$)
  - Since we want $fan_{in}\text{Var}(w) = 1 \implies fan_{in}\frac{(a)^2}{3} = 1 \implies a = \frac{\sqrt{3}}{\sqrt{fan_{in}}}$

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$
  - Considering the backward pass during backprop, it may be wise to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{out}}}$ too; hence, go with the average $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$
  - If $w \sim \texttt{Uniform}[-a, a]$, $\text{Var}(w) = \frac{(2a)^2}{12} = \frac{(a)^2}{3}$ ($\because$ variance of uniform distribution in interval $[m, n]$ is $\frac{(n-m)^2}{12}$)
  - Since we want $fan_{in}\text{Var}(w) = 1 \implies fan_{in}\frac{(a)^2}{3} = 1 \implies a = \frac{\sqrt{3}}{\sqrt{fan_{in}}}$
  - Considering the backward pass, we get the proposed initialization

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$
  - Considering the backward pass during backprop, it may be wise to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{out}}}$ too; hence, go with the average $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$
  - If $w \sim \texttt{Uniform}[-a, a]$, $\text{Var}(w) = \frac{(2a)^2}{12} = \frac{(a)^2}{3}$ ($\because$ variance of uniform distribution in interval $[m, n]$ is $\frac{(n-m)^2}{12}$)
  - Since we want $fan_{in}\text{Var}(w) = 1 \implies fan_{in}\frac{(a)^2}{3} = 1 \implies a = \frac{\sqrt{3}}{\sqrt{fan_{in}}}$
  - Considering the backward pass, we get the proposed initialization
- **He's initialization**: $\texttt{uniform}(-\frac{4}{fan_{in}+fan_{out}}, \frac{4}{fan_{in}+fan_{out}})$. Homework!

Batch Normalization

- **Covariate Shift** refers to change in input distribution between training and test scenario. This is a problem because the network has to adapt to the new distribution
- During training, the input distribution to a layer keeps changing over iterations. This is known as **internal covariate shift**. How to handle?

[4] Lecun et al, Efficient Backpropagation, 1998

Batch Normalization

- **Covariate Shift** refers to change in input distribution between training and test scenario. This is a problem because the network has to adapt to the new distribution
- During training, the input distribution to a layer keeps changing over iterations. This is known as **internal covariate shift**. How to handle?
- It is known that network training converges faster if its inputs are whitened[4] i.e linearly transformed to have zero means and unit variances.
- Explicitly ensure each layer's inputs are unit Gaussians (across each dimension), i.e.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathsf{Var}(x^{(k)})}}$$

How do we ensure this?

[4] Lecun et al, Efficient Backpropagation, 1998

Batch Normalization

- The mini-batch! $\mathbb{E}[x^{(k)}]$ and $\text{Var}(x^{(k)})$ are computed empirically from a mini-batch, i.e. ensure that distribution of inputs does not change across batches
- Let's go one step further - introduce $\gamma^{(k)}$ and $\beta^{(k)}$, additional parameters that network learns. This allows network to learn a suitable distribution than use a Gaussian

$$\gamma^{(k)} = \sqrt{\text{Var}(x^{(k)})}$$
$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- BN layer is usually inserted before non-linearity (activation)
- Allows higher learning rates. Why?

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- BN layer is usually inserted before non-linearity (activation)
- Allows higher learning rates. Why? No fear of exploding weights/gradients
- Reduces strong dependence on initialization
- Acts as a form of regularization
- BatchNorm layer functions differently at test time: Mean/std are not computed based on test batch; instead, a running average of training mini-batch mean and variance (computed during training) is used.

Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Homework

Readings

- Deep Learning book, Chapter 8, Section 8.7
- Efficient Backprop by Yann LeCun, 1998 (Optional, old article but has interesting insights, worth a read!)

Exercise

- Visit https://www.deeplearning.ai/ai-notes/initialization, and try out the animations for weight initialization!
- BN layer is fully differentiable. Consider a neural network with a single hidden layer. Can you show how you would use backpropagation to compute gradients for the batchnorm parameters, γ and β?
- Derive Kaiming He's initialization.

References

Yann A. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Springer Berlin Heidelberg, 2012, pp. 9–48.

K. He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1026–1034.

Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *ICML*. 2015, 448–456.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.