Deep Learning book, by Ian Goodfellow, Yoshua Bengio and Aaron Courville

Chapter 6: Deep Feedforward Networks

Benoit Massé    Dionyssos Kounades-Bastian
Linear regression (and classification)

Input vector $x$

Output vector $y$

Parameters Weight $W$ and bias $b$

Prediction: $y = W^T x + b$
Linear regression (and classification)

Input vector $x$

Output vector $y$

Parameters Weight $W$ and bias $b$

Prediction: $y = W^T x + b$
Input vector $x$

Output vector $y$

Parameters Weight $W$ and bias $b$

Prediction: $y = W^T x + b$
Linear regression (and classification)

Input vector $\mathbf{x}$

Output vector $\mathbf{y}$

Parameters Weight $\mathbf{W}$ and bias $\mathbf{b}$

Prediction: $\mathbf{y} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$
Linear regression (and classification)

**Advantages**
- Easy to use
- Easy to train, low risk of overfitting

**Drawbacks**
- Some problems are inherently non-linear
Solving XOR

There is no value for $W$ and $b$ such that $\forall (x_1, x_2) \in \{0, 1\}^2$

$W^\top (x_1 x_2) + b = \text{or} (x_1, x_2)$
Solving XOR

There is no value for $W$ and $b$ such that $\forall (x_1, x_2) \in \{0, 1\}^2$

\[ W^\top \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + b = \text{xor}(x_1, x_2) \]
Solving XOR

What about...?

\[
\begin{align*}
x_1 & \rightarrow u_1 & W, b \\
x_2 & \rightarrow u_2 & V, c \\
\end{align*}
\]

\[
\begin{align*}
u_1 & \rightarrow y \\
u_2 & \rightarrow y \\
\end{align*}
\]

Strictly equivalent:
The composition of two linear operations is still a linear operation.

Benoit Massé, Dionyssos Kounades-Bastian

Deep Feedforward Networks
What about... ?

\[
\begin{align*}
W, b & \quad \rightarrow \quad u_1 \\
V, c & \quad \rightarrow \quad y
\end{align*}
\]

The composition of two linear operations is still a linear operation

Strictly equivalent:
Solving XOR

And about... ?

In which $\phi(x) = \max\{0, x\}$
Solving XOR

And about...?

In which $\phi(x) = \max\{0, x\}$

It is possible!

With $W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $b = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$, $V = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$ and $c = 0$,

$V \phi(Wx + b) = \text{xor}(x_1, x_2)$
Neural network with one hidden layer

Compact representation

$\mathbf{x} \xrightarrow{\mathbf{W}, \mathbf{b}, \phi} \mathbf{h} \xrightarrow{\mathbf{V}, \mathbf{c}} \mathbf{y}$

Neural network

Hidden layer with non-linearity

→ can represent broader class of function

Benoit Massé, Dionyssos Kounades-Bastian
Universal approximation theorem

Theorem
A neural network with one hidden layer can approximate any continuous function

More formally, given a continuous function $f : C_n \mapsto \mathbb{R}^m$ where $C_n$ is a compact subset of $\mathbb{R}^n$,

$$\forall \varepsilon, \exists f_{\text{NN}}^\varepsilon : x \mapsto \sum_{i=1}^{K} v_i \phi(w_i^\top x + b_i) + c$$

such that

$$\forall x \in C_n, \|f(x) - f_{\text{NN}}^\varepsilon(x)\| < \varepsilon$$
Problems

**Obtaining the network**

The universal theorem gives no information about HOW to obtain such a network

- Size of the hidden layer $h$
- Values of $W$ and $b$
Problems

Obtaining the network

The universal theorem gives no information about HOW to obtain such a network

- Size of the hidden layer $h$
- Values of $W$ and $b$

Using the network

Even if we find a way to obtain the network, the size of the hidden layer may be prohibitively large.
Deep neural network

Why Deep?
Let’s stack $l$ hidden layers one after the other; $l$ is called the length of the network.

$x \xrightarrow{W^1, b^1, \phi} h_1 \xrightarrow{W^2, b^2, \phi} \ldots \xrightarrow{W^l, b^l, \phi} h_l \xrightarrow{V, c} y$

Properties of DNN
- The universal approximation theorem also apply
- Some functions can be approximated by a DNN with $N$ hidden unit, and would require $O(e^N)$ hidden units to be represented by a shallow network.
## Comparison

<table>
<thead>
<tr>
<th>Linear classifier</th>
<th>Shallow Neural network (Exactly one hidden layer)</th>
<th>Deep Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited representational power</td>
<td>Unlimited representational power</td>
<td>Unlimited representational power</td>
</tr>
<tr>
<td>Simple</td>
<td>Sometimes prohibitively wide</td>
<td>Relatively small number of hidden units needed</td>
</tr>
</tbody>
</table>
## Summary

### Comparison

<table>
<thead>
<tr>
<th>Linear classifier</th>
<th>Shallow Neural network (Exactly one hidden layer)</th>
<th>Deep Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited representational power</td>
<td>Unlimited representational power</td>
<td>Unlimited representational power</td>
</tr>
<tr>
<td>Simple</td>
<td>Sometimes prohibitively wide</td>
<td>Relatively small number of hidden units needed</td>
</tr>
</tbody>
</table>

### Remaining problem

How to get this DNN?
Hyperparameters

First, we need to define the architecture of the DNN

- The depth $l$
- The size of the hidden layers $n_1, \ldots, n_l$
- The activation function $\phi$
- The output unit
On the path of getting my own DNN

Hyperparameters
First, we need to define the architecture of the DNN
- The depth $l$
- The size of the hidden layers $n_1, \ldots, n_l$
- The activation function $\phi$
- The output unit

Parameters
When the architecture is defined, we need to train the DNN
- $W^1, b^1, \ldots, W^l, b^l$
Hyperparameters

- The depth $l$
- The size of the hidden layers $n_1, \ldots, n_l$
  - Strongly depend on the problem to solve

The activation function $\phi$:
- ReLU: $x \mapsto \max\{0, x\}$
- Sigmoid: $x \mapsto \frac{1 + e^{-x}}{1 - e^{-x}} - 1$
- Many others: tanh, RBF, softplus...

The output unit:
- Linear output: $E[y] = V^\top h_l + c$
  - For regression with Gaussian distribution $y \sim \mathcal{N}(E[y], I)$
- Sigmoid output: $\hat{y} = \sigma(w^\top h_l + b)$
  - For classification with Bernouilli distribution $P(y = 1 | x) = \hat{y}$
Hyperparameters

- The depth \( l \)
- The size of the hidden layers \( n_1, \ldots, n_l \)
  - Strongly depend on the problem to solve
- The activation function \( \phi \)
  - ReLU \( g : x \mapsto \max\{0, x\} \)
  - Sigmoid \( \sigma : x \mapsto \left(1 + e^{-x}\right)^{-1} \)
  - Many others: tanh, RBF, softplus...
Hyperparameters

- The depth \( l \)
- The size of the hidden layers \( n_1, \ldots, n_l \)
  - Strongly depend on the problem to solve
- The activation function \( \phi \)
  - ReLU \( g : x \mapsto \max\{0, x\} \)
  - Sigmoid \( \sigma : x \mapsto (1 + e^{-x})^{-1} \)
  - Many others: tanh, RBF, softplus...
- The output unit
  - Linear output \( \mathbb{E}[y] = V^\top h_l + c \)
    - For regression with Gaussian distribution \( y \sim \mathcal{N}(\mathbb{E}[y], I) \)
  - Sigmoid output \( \hat{y} = \sigma(w^\top h_l + b) \)
    - For classification with Bernouilli distribution \( P(y = 1|x) = \hat{y} \)
Objective

Let’s define $\theta = (W^1, b^1, \ldots, W^l, b^l)$. We suppose we have a set of inputs $X = (x^1, \ldots, x^N)$ and a set of expected outputs $Y = (y^1, \ldots, y^N)$. The goal is to find a neural network $f_{NN}$ such that

$$\forall i, \ f_{NN}(x^i, \theta) \approx y^i.$$
Cost function

To evaluate the error that our current network makes, let’s define a cost function $\mathcal{L}(X, Y, \theta)$. The goal becomes to find

$$\argmin_{\theta} \mathcal{L}(X, Y, \theta)$$

Loss function

Should represent a combination of the distances between every $y_i$ and the corresponding $f_{\text{NN}}(x_i, \theta)$

- Mean square error (rare)
- Cross-entropy
The basic idea consists in computing $\hat{\theta}$ such that

$$\nabla_\theta \mathcal{L}(X, Y, \hat{\theta}) = 0.$$ 

This is difficult to solve analytically e.g. when $\theta$ have millions of degrees of freedom.
Gradient descent

Let's use a numerical way to optimize $\theta$, called the gradient descent (section 4.3). The idea is that

$$f(\theta - \varepsilon u) \simeq f(\theta) - \varepsilon u^\top \nabla f(\theta)$$

So if we take $u = \nabla f(\theta)$, we have $u^\top u > 0$ and then

$$f(\theta - \varepsilon u) \simeq f(\theta) - \varepsilon u^\top u < f(\theta).$$

If $f$ is a function to minimize, we have an update rule that improves our estimate.
Gradient descent algorithm

1. Have an estimate $\hat{\theta}$ of the parameters
2. Compute $\nabla_\theta \mathcal{L}(X, Y, \hat{\theta})$
3. Update $\hat{\theta} \leftarrow \hat{\theta} - \varepsilon \nabla_\theta \mathcal{L}$
4. Repeat step 2-3 until $\nabla_\theta \mathcal{L} < \text{threshold}$
Gradient descent algorithm

1. Have an estimate $\hat{\theta}$ of the parameters
2. Compute $\nabla_{\theta} \mathcal{L}(X, Y, \hat{\theta})$
3. Update $\hat{\theta} \leftarrow \hat{\theta} - \varepsilon \nabla_{\theta} \mathcal{L}$
4. Repeat step 2-3 until $\nabla_{\theta} \mathcal{L} < \text{threshold}$

Problem

How to estimate efficiently $\nabla_{\theta} \mathcal{L}(X, Y, \hat{\theta})$?

- Back-propagation algorithm
Consider the architecture:

\[
y = \phi(w_2 \phi(w_1 x)),
\]

with function:

some training pairs \( T = \{\hat{x}_n, \hat{y}_n\}_{n=1}^N \), and
an activation-function \( \phi() \).
Learn \( w_1, w_2 \) so that: Feeding \( \hat{x}_n \) results \( \hat{y}_n \).
For learning to be possible $\phi()$ has to be differentiable.

Let $\phi'(x) = \frac{\partial \phi(x)}{\partial x}$ denote the derivative of $\phi(x)$.

For example when $\phi(x) = \text{ReLU}(x)$ we have:
Gradient-based Learning

- Minimize the loss function $\mathcal{L}(w_1, w_2, T)$.
- We will learn the weights by iterating:

\[
\begin{bmatrix}
    w_1 \\
    w_2
\end{bmatrix}_{\text{updated}} = \begin{bmatrix}
    w_1 \\
    w_2
\end{bmatrix} - \gamma \begin{bmatrix}
    \frac{\partial \mathcal{L}}{\partial w_1} \\
    \frac{\partial \mathcal{L}}{\partial w_2}
\end{bmatrix}, \tag{1}
\]

- $\mathcal{L}$ is the loss function (must be differentiable): In detail is $\mathcal{L}(w_1, w_2, T)$ and we want to compute the gradient(s) at $w_1, w_2$.
- $\gamma$ is the learning rate (a scalar typically known).
Calculate intermediate values on all units:

1. \[ a = w_1 \hat{x}_n. \]
2. \[ b = \phi(w_1 \hat{x}_n). \]
3. \[ c = w_2 \phi(w_1 \hat{x}_n). \]
4. \[ d = \phi(w_2 \phi(w_1 \hat{x}_n)). \]
5. \[ \mathcal{L}(d) = \mathcal{L}(\phi(w_2 \phi(w_1 \hat{x}_n))). \]

The partial derivatives are:

6. \[ \frac{\partial \mathcal{L}(d)}{\partial d} = \mathcal{L}'(d). \]
7. \[ \frac{\partial d}{\partial c} = \phi'(w_2 \phi(w_1 \hat{x}_n)). \]
8. \[ \frac{\partial c}{\partial b} = w_2. \]
9. \[ \frac{\partial b}{\partial a} = \phi'(w_1 \hat{x}_n). \]
Calculating the Gradients I

- Apply chain rule:

\[
\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial w_1},
\]

\[
\frac{\partial L(d)}{\partial w_2} = \frac{\partial L(d)}{\partial d} \frac{\partial d}{\partial c} \frac{\partial c}{\partial w_2}.
\]
Apply chain rule:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} w_1,$$

$$\frac{\partial L(d)}{\partial w_2} = \frac{\partial L(d)}{\partial d} \frac{\partial d}{\partial c} w_2.$$

Start the calculation from left-to-right.

We propagate the gradients (partial products) from the last layer towards the input.
Calculating the Gradients

And because we have $N$ training pairs:

$$\frac{\partial L}{\partial w_1} = \sum_{n=1}^{N} \frac{\partial L(d_n)}{\partial d_n} \frac{\partial d_n}{\partial c_n} \frac{\partial c_n}{\partial b_n} \frac{\partial b_n}{\partial a_n} \frac{\partial a_n}{w_1},$$

$$\frac{\partial L}{\partial w_2} = \sum_{n=1}^{N} \frac{\partial L(d_n)}{\partial d_n} \frac{\partial d_n}{\partial c_n} \frac{\partial c_n}{w_2}.$$
Thank you!