ECE 1508: Applied Deep Learning

Chapter 1: Preliminaries

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

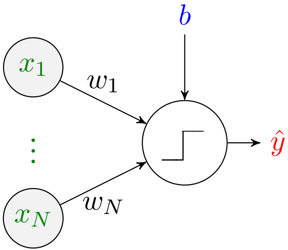
Winter 2025

Training Perceptron Machine: *ML 1*

Let's start our long story with binary classification via perceptron

1. Dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$ with binary labels $y_i \in \{0, 1\}$
2. Perceptron as the model $\boxed{\text{Binary classifier}}$

$$\hat{y} = s(\mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b) = \begin{cases} 1 & \text{if } \mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b \geqslant 0 \\ 0 & \text{if } \mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b < 0 \end{cases}$$



3. Error indicator as the loss function

$$\mathcal{L}(\hat{y}, y) = \mathbb{1}(\hat{y} \neq y) = \begin{cases} 1 & \hat{y} \neq y \\ 0 & \hat{y} = y \end{cases}$$

Training Perceptron Machine: *ML 2*

Let's write down the empirical risk

$$
\begin{aligned}
\hat{R}(\mathbf{w}, b) &= \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}(\hat{y}_i, y_i) \\
&= \frac{1}{I} \sum_{i=1}^{I} \mathbb{1}(\hat{y}_i \neq y_i) \\
&= \frac{1}{I} \sum_{i=1}^{I} \mathbb{1}\left(s(\mathbf{w}^\mathsf{T} \boldsymbol{x}_i + b) \neq y_i\right) \\
&= \frac{\text{\# of times perceptron misclassifies}}{I} \\
&\equiv \text{Error Rate}
\end{aligned}
$$

Training Perceptron Machine: *ML 3*

We should now minimize the empirical risk

$$\mathbf{w}^\star, b^\star = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^N,\ b\in\mathbb{R}} \hat{R}\left(\mathbf{w}, b\right) = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^N,\ b\in\mathbb{R}} \frac{1}{I}\sum_{i=1}^{I} \mathbb{1}\left(s(\mathbf{w}^\mathsf{T}\boldsymbol{x}_i + b) \neq y_i\right)$$

But, how can we solve this optimization? It doesn't look easy!

Let's see what Rosenblatt did: *Rosenblatt's perceptron had no bias, i.e.,* $b = 0$

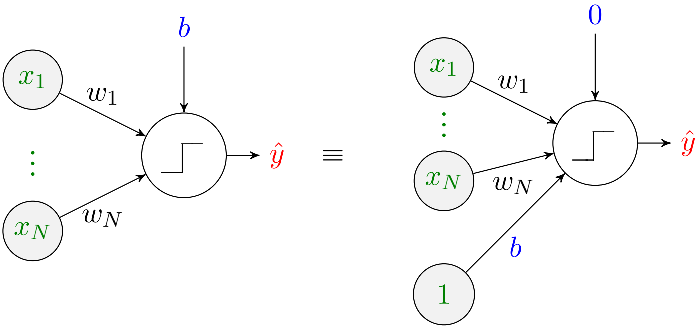
Start with $\mathbf{w} = \mathbf{0}$ or some small random initial value
while $\hat{R}\left(\mathbf{w}\right) \neq 0$ **do**
  for $i = 1 : I$ **do**
    Compute $z_i = \mathbf{w}^\mathsf{T}\boldsymbol{x}_i$ and $\hat{y}_i = s(z_i)$ `# pass through perceptron`
    if $\hat{y}_i \neq y_i$ **then**
      $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}\left(z_i\right)\boldsymbol{x}_i$
    end if
  end for
end while

Training Perceptron Machine: *ML 3*

It's almost zero effort to extend Rosenblatt's algorithm to the case with bias

$$\mathbf{w}^\mathsf{T}\boldsymbol{x}_i + b = [\mathbf{w}^\mathsf{T}, b]\begin{bmatrix}\boldsymbol{x}_i\\ 1\end{bmatrix} = \tilde{\mathbf{w}}^\mathsf{T}\tilde{\boldsymbol{x}}_i$$

So, bias only adds one dimension to the data-point with value 1



Training Perceptron Machine: *Perceptron Algorithm*

So, we just need to replace $\mathbf{w}$ with $\begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$ in Rosenblatt's algorithm

```
Start with w = 0 and b = 0, or some small random initial value
while R̂(w, b) ≠ 0 do
    for i = 1 : I do
        Compute z_i = wᵀx_i + b and ŷ_i = s(z_i)        # pass through perceptron
        if ŷ_i ≠ y_i then
            w ← w − sign(z_i) x_i and b ← b − sign(z_i)
        end if
    end for
end while
```

+ *Why should perceptron algorithm minimize the empirical risk?*
– Let's inspect the no-bias version step by step

Training Perceptron Machine: *Perceptron Algorithm*

```
Start with w = 0 or some small random initial value
while R̂(w) ≠ 0 do
    for i = 1 : I do
        Compute z_i = w^T x_i and ŷ_i = s(z_i)        # pass through perceptron
        if ŷ_i ≠ y_i then
            w ← w − sign(z_i) x_i
        end if
    end for
end while
```

- Outer loop stops only if $\hat{R}(\mathbf{w}) = 0$ which is minimum empirical risk
- In inner loop, let's say at iteration t error occurs for $\boldsymbol{x}_i$

↳ either $z_i = \mathbf{w}_t^\mathsf{T}\boldsymbol{x}_i > 0$ and $y_i = 0$: *we update as* $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \boldsymbol{x}_i$

$$\mathbf{w}_{t+1}^\mathsf{T}\boldsymbol{x}_i = (\mathbf{w}_t - \boldsymbol{x}_i)^\mathsf{T}\boldsymbol{x}_i = \mathbf{w}_t^\mathsf{T}\boldsymbol{x}_i - \|\boldsymbol{x}_i\|^2 \leqslant \mathbf{w}_t^\mathsf{T}\boldsymbol{x}_i$$

so the algorithm pushes the weights somewhere that z_i could get negative

Training Perceptron Machine: *Perceptron Algorithm*

```
Start with w = 0 or some small random initial value
while R̂(w) ≠ 0 do
    for i = 1 : I do
        Compute z_i = w^T x_i and ŷ_i = s(z_i)        # pass through perceptron
        if ŷ_i ≠ y_i then
            w ← w − sign(z_i) x_i
        end if
    end for
end while
```

- Outer loop stops only if $\hat{R}(\mathbf{w}) = 0$ which is minimum empirical risk
- In inner loop, let's say at iteration t error occurs for $\boldsymbol{x}_i$

↳ or $z_i = \mathbf{w}_t^{\mathsf{T}} \boldsymbol{x}_i < 0$ and $y_i = 1$: *we update as* $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \boldsymbol{x}_i$

$$\mathbf{w}_{t+1}^{\mathsf{T}} \boldsymbol{x}_i = (\mathbf{w}_t + \boldsymbol{x}_i)^{\mathsf{T}} \boldsymbol{x}_i = \mathbf{w}_t^{\mathsf{T}} \boldsymbol{x}_i + \|\boldsymbol{x}_i\|^2 \geqslant \mathbf{w}_t^{\mathsf{T}} \boldsymbol{x}_i$$

so the algorithm pushes the weights somewhere that z_i could get positive

Training Perceptron Machine: *Perceptron Algorithm*

Start with $\mathbf{w} = \mathbf{0}$ or some small random initial value
while $\hat{R}(\mathbf{w}) \neq 0$ **do**
    for $i = 1 : I$ **do**
        Compute $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i$ and $\hat{y}_i = s(z_i)$ `# pass through perceptron`
        if $\hat{y}_i \neq y_i$ **then**
            $\mathbf{w} \leftarrow \mathbf{w} - \mathrm{sign}(z_i)\, \boldsymbol{x}_i$
        end if
    end for
end while

+ *It makes sense that in each iteration $\mathbf{w}$ gets modified towards a right direction! But can we guarantee that this algorithm always converges? In other words, can't it get into an infinity loop?*
- Well! Let's try some examples

Perceptron Algorithm: *AND Operator*

Assume that we have the following dataset two-dimensional inputs

$$\mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 1 \right) \right\}$$

We intend to train perceptron with this dataset via perceptron algorithm

---

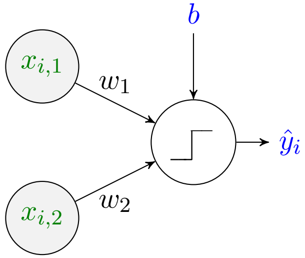
Before we start training, we note that this dataset represents the AND operator

$$\forall \left(\boldsymbol{x}_i, y_i \right) \in \mathbb{D} : y_i = x_{i,1} \wedge x_{i,2}$$

so, we basically want to see, if we could realize this operator via perceptron

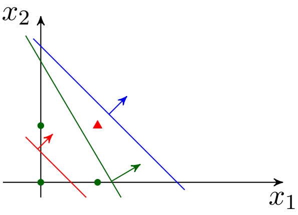
Perceptron Algorithm: *AND Operator*

Start with $\mathbf{w} = [1.1, 1.1]^\mathsf{T}$ and $b = -2.25$: `blue`
while $\hat{R}(\mathbf{w}, b) \neq 0$ **do**
    for $i = 1 : I$ **do**
        Compute $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i + b$ and $\hat{y}_i = s(z_i)$
        if $\hat{y}_i \neq y_i$ **then**
            $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$ and $b \leftarrow b - \text{sign}(z_i)$
        end if
    end for
end while



Let's show data-points with $y_i = 1$ by ▲ and those with $y_i = 0$ by ●

- Updated by $\boldsymbol{x}_i = [1, 1]^\mathsf{T}$
  - ↳ $\mathbf{w} = [2.1, 2.1]^\mathsf{T}$ and $b = -1.25$: `red`
- Updated by $\boldsymbol{x}_i = [0, 1]^\mathsf{T}$
  - ↳ $\mathbf{w} = [2.1, 1.1]^\mathsf{T}$ and $b = -2.25$: `green`



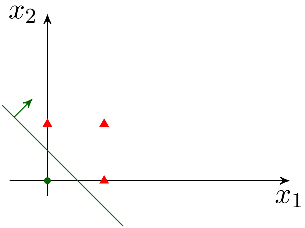
Perceptron Algorithm: *AND Operator*

Great! We trained the perceptron to behave as logical AND

+ *What about logical OR?*
– Easy! Let's write the dataset

$$\mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 1 \right) \right\}$$

Trying the perceptron algorithm, we end up with some linear classifier like

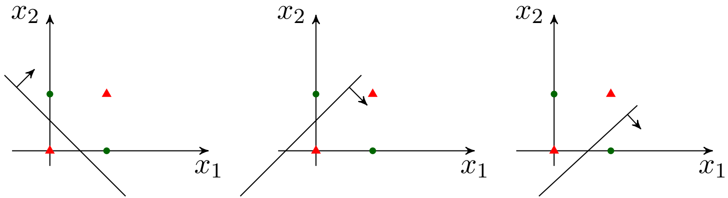


Perceptron Algorithm: *False Conclusion*

A false initial conclusion is that *perceptron can realize any binary function!* But, it is easy to see that *this is not the case!* Let's consider logical XOR:

$$x_1 \oplus x_2 = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases} \rightsquigarrow \mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 0 \right) \right\}$$

It's not hard to see that perceptron cannot learn this function[1]



[1] If you don't see it clearly, no worries! You'll show it as an assignment

Perceptron Algorithm: *Linearly Separable Functions*

+ *What happens if we try the perceptron algorithm on this dataset?*
– It will iterate for ever!
+ *Why does this happen?*
– This is because XOR is not linearly separable

Perceptron can classify only linearly separable functions

At this point, it was concluded that perceptron should be replaced by some other model in order to learn nonlinear function

Finding models that learn nonlinear function led to the birth of

Representation Learning

But, we don't need to study it, since deep learning solves the problem!

Multi-Layering Perceptrons

Let's play a bit with logical XOR: recall that

$$x_1 \oplus x_2 = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases}$$

We can write this function as

$$x_1 \oplus x_2 = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$

where we have used the following notation

$$\bar{x} \equiv \text{complement of } x = 1 - x \qquad \text{and} \qquad \vee \equiv \text{logical OR}$$

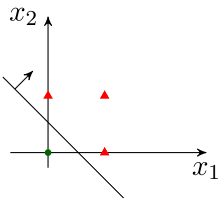
Multi-Layering Perceptrons

Logical XOR expands as

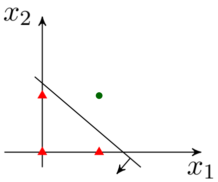
$$y = x_1 \oplus x_2 = \underbrace{(x_1 \vee x_2)}_{h_1} \wedge \underbrace{(\bar{x}_1 \vee \bar{x}_2)}_{h_2} = h_1 \wedge h_2$$

We can learn h_1 and h_2 by two different perceptrons

$$h_1 = x_1 \vee x_2$$

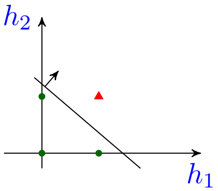


$$h_2 = \bar{x}_1 \vee \bar{x}_2 \equiv \overline{x_1 \wedge x_2}$$

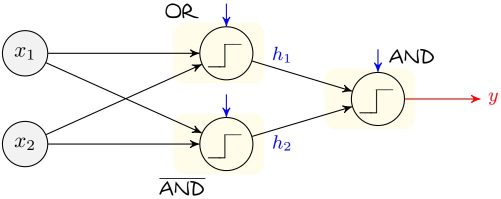


Multi-Layering Perceptrons

We can further learn $y = h_1 \wedge h_2$ by another perceptron

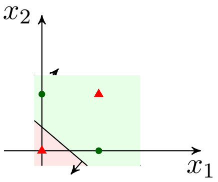


Well! It's true that we cannot learn XOR by a single perceptron, but we can learn it with a network of three perceptrons!



Multi-Layering Perceptrons: *Geometrical Interpretation*

Let's see geometrically what this network of perceptrons does



- First perceptron classifies
- Second perceptron classifies
- Third perceptron intersects the two regions

Multi-Layering Perceptrons: *Correct Conclusion*

We can learn any binary function using a network of perceptrons!

This has given birth to the idea of neural network

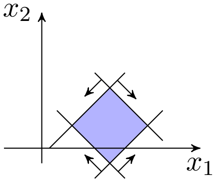
Neural Networks: *Intuitive Introduction*

looking at perceptron as an artificial neuron, we can learn any complicated function when we have a network of these neurons $\equiv$ *a Neural Network*

+ *But, why should we care only about binary functions? Don't we learn other type of functions as well?*
– Well! We can extend the idea to other problems as well!

Multi-Layering Perceptrons: *More Complicated Functions*

Assume we have binary classification with real-valued inputs: we want to train a classifier that distinguishes between the two-dimensional points inside the blue area and outside of it



- We classify *Region 1* with three perceptrons
- We classify *Region 2* with three perceptrons
- We intersect the two regions with a perceptron

A network of 7 perceptrons is more than enough!

Multi-Layering Perceptrons: *More Complicated Functions*

+ *What if the classification problem is not binary?*
– Well! We already have seen that we can reduce a multi-class classification to a series of binary classifications[2]

We want to classify an image as dog, cat or car

- *Binary Classification 1:* Is it *class 0: dog* or *class 1: {cat, car}*?
  - ↳ *If class 0* $\rightsquigarrow$ *classification ended*
  - ↳ *If class 1* $\rightsquigarrow$ *Binary Classification 2:* Is it *class 0: cat* or *class 1: car*?

Moral of Story

We can learn any classifier with high accuracy using a network of perceptrons

[2]There are better multi-class classification techniques that we learn later

Multi-Layering Perceptrons: *More Complicated Functions*

+ *Fair enough! We are happy with classification! But, what about the case that we have real-valued labels? Can we do it by perceptrons?*

– This is the regression problem!

– And, Yes! We can do this as well using perceptrons

Regression is a supervised learning problem in which

labels belong to a continuous set, e.g., $y_i \in \mathbb{R}$

the label y_i in this case is the function sample at $\boldsymbol{x}_i$

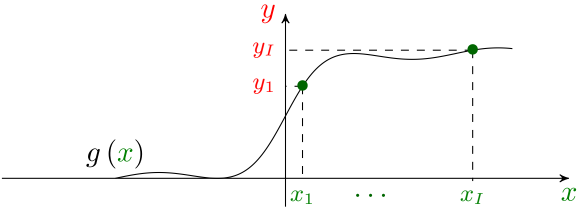
Multi-Layering Perceptrons: *Regression*

In regression, we learn a real-valued function

Let's look at a simple case with one-dimensional inputs, where we can visualize

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \dots, I\}$$

A visualization of this dataset is



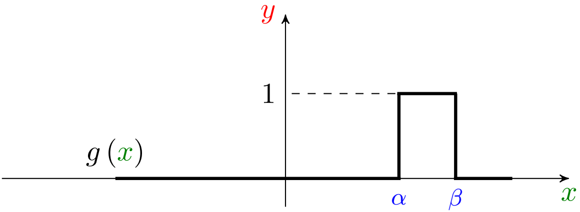
and we want to learn $g(\cdot)$

Multi-Layering Perceptrons: *Regression*

The main question is

> *can we realize an arbitrary $g(\cdot)$ via a network of perceptrons?*

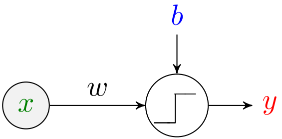
Let's start with a simple function: *the unit pulse*



We can realize this function using three perceptrons

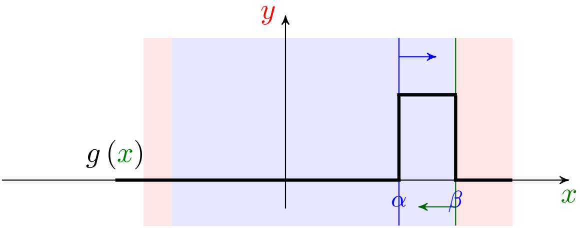
Multi-Layering Perceptrons: *Regression*

First, let's recall how perceptron looks with one-dimensional input

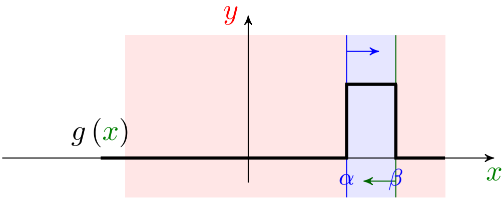


- ↳ with $w = 1$ and $b = -\alpha$: `blue`
- ↳ with $w = -1$ and $b = \beta$: `green`

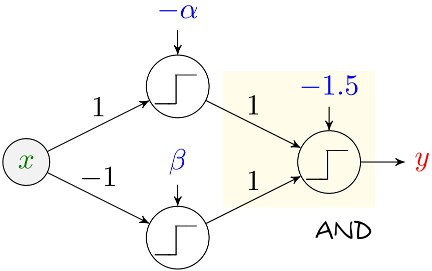
We realize the pulse by applying AND on the outputs of these perceptrons



Multi-Layering Perceptrons: *Regression*

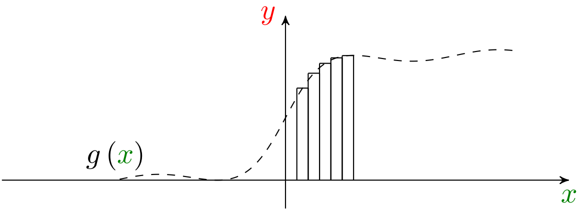


So, we can realize the unit pulse via the following network of perceptrons



Multi-Layering Perceptrons: *Regression*

We can approximate a general function via a weighted sum of unit pulses



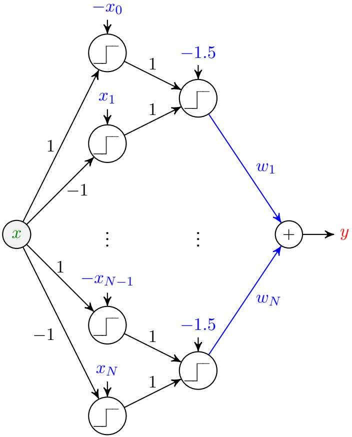
This means that we can approximate any function by

- realizing each pulse by a network of perceptrons
- applying linear transform on the outputs of these networks

Multi-Layering Perceptrons: *Regression*

Real-valued functions are *well-approximated* by

a network of perceptrons + a linear transform



Multi-Layering Perceptrons: *Summary*

A network of perceptrons seems to be a very sophisticated model

- It can learn any classifier
- It can approximate any function with arbitrary accuracy

Universal Approximation Theorem *(informal)*

Given function $g(\cdot)$ *and* $\varepsilon > 0$, *there exists a neural network* $f_{\mathbf{h}}(\cdot|\mathbf{w})$ *that*

$$\sup_{\boldsymbol{x}} \| g(\boldsymbol{x}) - f_{\mathbf{h}}(\boldsymbol{x}|\mathbf{w}) \| \leqslant \varepsilon$$

So, it seems natural to train them for our learning tasks

A network of perceptrons is a special artificial neural network

↳ *we now formally introduce artificial neural network*