We are starting at 13:00!

Grab a seat and get ready

Saturdays.AI
Kigali

Saturdays.AI
Kigali

#5 Multilayer Perceptron (MLP)

AI Saturdays Kigali

Introduction to Deep Learning

Saturdays.AI
Kigali

Deep Learning: The Hottest Topic in ML

- Deep Learning is a **subfield of Machine Learning** focused on training **neural networks with many layers (DNNs)**.
- It powers cutting-edge applications in **vision**, **language**, and **autonomous systems**.
- In recent years, deep learning has received massive attention due to its impressive performance across various domains.

What You'll Learn in This Class

- Gaining a **conceptual understanding** of **multilayer NNs**
- **Implementing the fundamental backpropagation algorithm** for NN training from scratch
- **Training a basic multilayer NN** for **image classification**

Saturdays.AI
Kigali

Recap: Single-Layer Neural Networks (Adaline)

Saturdays.AI
Kigali

Recap: Single-Layer NNs (Adaline)

- In lecture 2, we built a **binary classifier** using **Adaline**
- Learned weights via **Gradient Descent (GD)**
- In each **epoch** (full pass through the training set), we updated:
  - Weight vector $\boldsymbol{w}$
  - Bias unit $\boldsymbol{b}$

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b := b + \Delta b$$

- Where:

$$\Delta w_j = -\eta \frac{\partial L}{\partial w_j} \text{ and } \Delta b = -\eta \frac{\partial L}{\partial b}$$

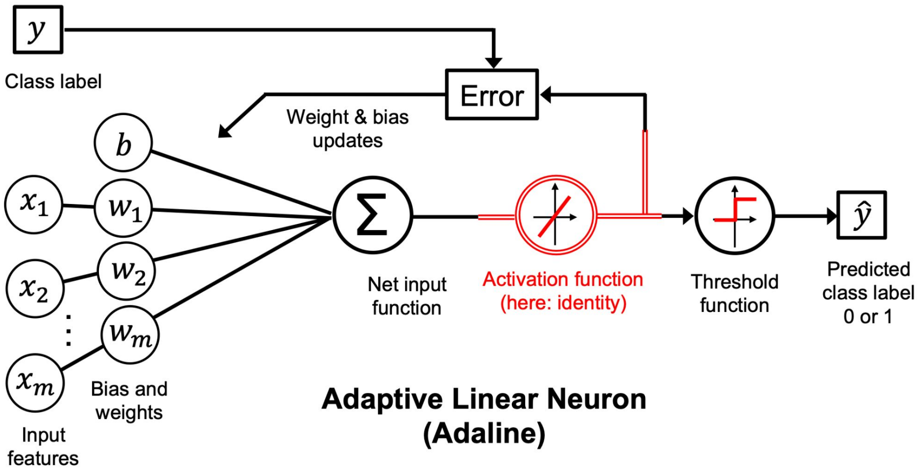


Figure 5.1: The Adaline algorithm

Recap: The Optimization Process

- Minimized the **Mean Squared Error (MSE)** loss function
- Computed the **loss gradient ∇ L(w)** from the entire training set
- Updated weights in the **opposite direction of the gradient**
- Used a **learning rate η** to control step size
  - Too small → slow learning
  - Too large → risk of **overshooting** the optimal solution

Saturdays.AI
Kigali

Recap: Weight Update with Gradient Descent

For each weight w□, computed:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{n}\sum_i \left(y^{(i)} - a^{(i)}\right)^2 = -\frac{2}{n}\sum_i \left(y^{(i)} - a^{(i)}\right)x_j^{(i)}$$

Where:

- $y^{(i)}$ = **target class label** of a particular sample $\boldsymbol{x}^{(i)}$
- $a^{(i)}$ = **activation** of the neuron (linear for Adaline)

Recap: Activation and Prediction

- Furthermore, we defined the **activation function** $\sigma(\cdot)$ as follows:

$$\sigma(\cdot) = z = a$$

- Here, the **net input**, **z**, is a linear combination of the weights that are connecting the input layer to the output layer:

$$z = \sum_j w_j x_j + b = \boldsymbol{w}^T \boldsymbol{x} + b$$

- We implemented a **threshold function** to squash the continuous-valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0; \\ 0 & \text{otherwise} \end{cases}$$

Saturdays.AI
Kigali

Recap: Stochastic Gradient Descent

Unlike GD, **SGD** updates weights after each sample (or mini-batch)

Advantages of SGD:

- **Faster learning** due to frequent updates (mini-batch learning)
- Adds **random noise**, helpful for:
  - **Escaping local minima** in non-convex loss functions
  - Multilayer, nonlinear networks

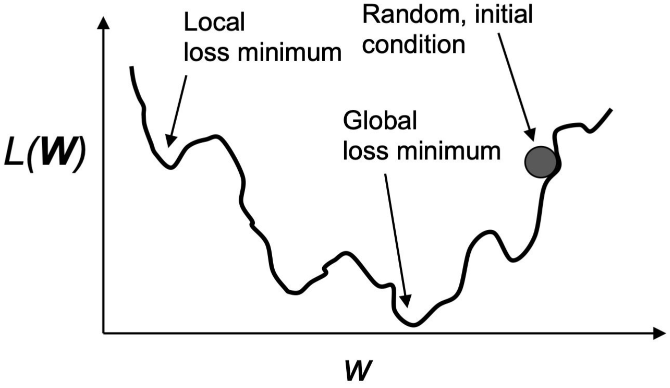


Figure 5.2: Optimization algorithms can become trapped in local minima

Saturdays.AI
Kigali

Multilayer Neural Network Architecture

Saturdays.AI
Kigali

From Single Neurons to Multilayer Networks

- We now connect multiple single neurons to form a **Multilayer Feedforward Neural Network**
- This structure is called a **Multilayer Perceptron (MLP)**

MLP Architecture: A Closer Look

- An MLP typically includes:
  - **Input layer**: receives the features
  - **Hidden layer(s)**: fully connected to the input
  - **Output layer**: fully connected to the hidden layer

What Makes It “Deep”?

- If the network has **more than one hidden layer**, it's referred to as a **Deep Neural Network (DNN)**

Saturdays.AI
Kigali

MLP Architecture: A Closer Look

- **Two-layer MLP**: **one hidden layer** and **one output layer**.
- **Fully connected**: Hidden layer units connect to inputs; output layer connects to hidden layer

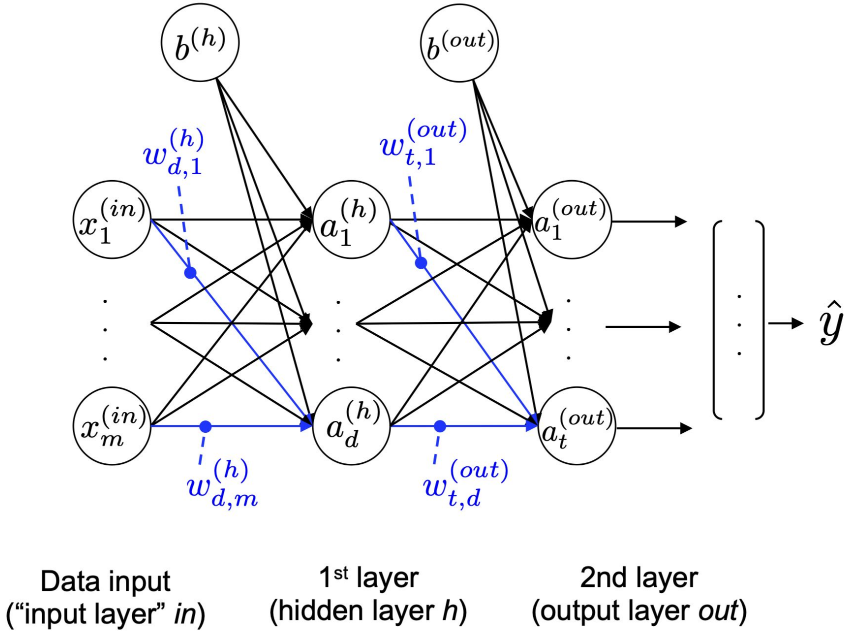


Figure 5.3: A two-layer MLP

Saturdays.AI Kigali

MLP Learning Procedure

To compute the output of a Multilayer Perceptron (MLP), we follow a simple 3-step learning process:

1. **Forward Propagation**
   Feed input data through the network to generate predictions.
2. **Compute Loss**
   Compare the predictions with true labels using a loss function.
3. **Backpropagation & Update**
   Calculate gradients and adjust weights and biases to reduce loss.

Once trained over multiple epochs, we:

- Use forward propagation to make predictions
- Apply a threshold to convert outputs to **one-hot encoded** class labels

Saturdays.AI
Kigali

Forward Propagation in MLP

To generate predictions, we just forward-propagate the input features through the network:

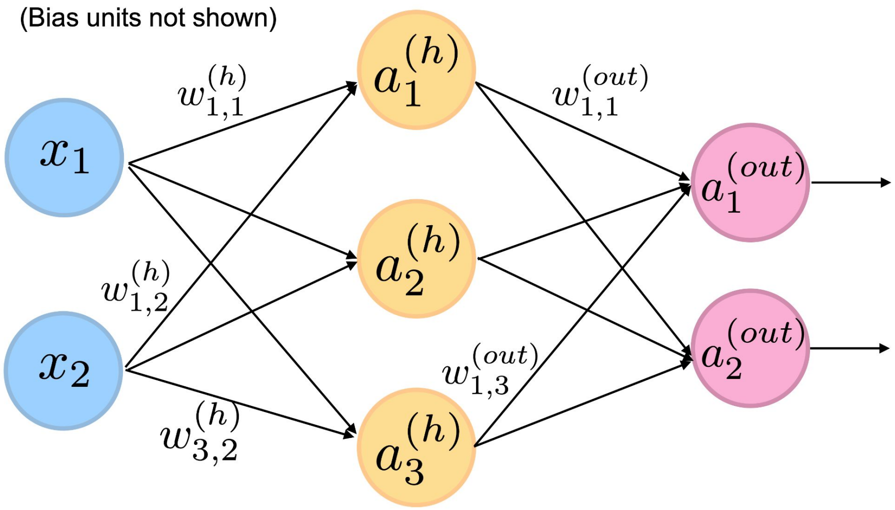


Figure 5.4: Forward-propagating the input features of an NN

Saturdays.AI
Kigali

Forward Propagation in MLP

Each hidden unit is connected to all input features. For hidden unit $a_1^{(h)}$

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)}$$

$$a_1^{(h)} = \sigma\left(z_1^{(h)}\right)$$

To be able to solve complex problems such as image classification, we need **nonlinear activation functions** in our MLP model, for example, the **sigmoid** (logistic) activation function

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

Vectorized form:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{X}^{(in)} \boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)} \quad \text{(net input of the hidden layer)}$$

$$\boldsymbol{A}^{(h)} = \sigma\left(\boldsymbol{Z}^{(h)}\right) \quad \text{(activation of the hidden layer)}$$

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)} \boldsymbol{W}^{(out)T} + \boldsymbol{b}^{(out)} \quad \text{(net input of the output layer)}$$

$$\boldsymbol{A}^{(out)} = \sigma\left(\boldsymbol{Z}^{(out)}\right) \quad \text{(activation of the output layer)}$$

Saturdays.AI
Kigali

Computing the Loss

After forward propagation, we measure how close our predictions are to the true labels using a loss function.

In our MLP, we use:

- **Mean Squared Error (MSE)** for simplicity in gradient derivation.
- Later, we'll explore **Cross-Entropy Loss** (common in classification).

Saturdays.AI
Kigali

What We Compare

We compare:

- **Predicted output vector** from the network
- **Target label** in **one-hot encoding**

Example (Predicting class 2 out of 4):

Predicted output:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Saturdays.AI
Kigali

Loss Formula for Multiclass MSE

To calculate MSE loss across:

- **t output neurons**
- **n training samples**

We sum over all neurons and average over all samples:

$$L(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{n} \sum_{1}^{n} \frac{1}{t} \sum_{j=1}^{t} \left(y_j^{[i]} - a_j^{(out)[i]} \right)^2$$

Saturdays.AI
Kigali

Why MSE (For Now)?

✔ Simpler gradients for backprop
✘ Not ideal for classification (we'll fix this later)
✔ Helps us understand the mechanics before switching to cross-entropy

Saturdays.AI
Kigali

Backpropagation & the Chain Rule

- A **computationally efficient** method for computing **partial derivatives**
- Helps optimize complex, **non-convex loss functions** in multilayer neural networks

Chain Rule Refresher

- The chain rule in calculus is used to compute the derivative of nested functions: $f(g(x))$

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

- This principle generalizes to deeper function compositions: $F(x) = f(g(h(u(v(x)))))$

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

Saturdays.AI
Kigali

Backward Propagation in MLPs

- Applies the **chain rule** in reverse—from **output layer to input**
- Gradually computes the **gradient of the loss** with respect to each **weight (and bias)**
- Enables **efficient training** of deep neural networks via **gradient descent**

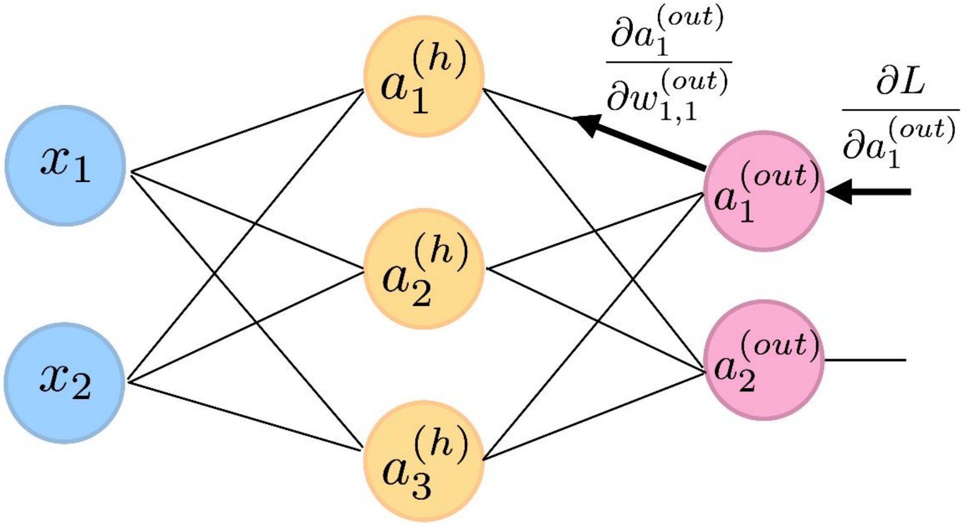


Figure 5.5: Backpropagating the error of an NN

Gradient for output layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

Saturdays.AI Kigali

Anatomy of a Neural Network

Saturdays.AI
Kigali

Anatomy of a Neural Network

When implementing a neural network from scratch (e.g., using NumPy), the architecture is often organized into a class.

A typical **`NeuralNetwork`** class includes the following responsibilities:

1. Initialize parameter
2. Perform forward propagation
3. Compute loss
4. Perform backward propagation
5. Update weights

Initialize Network: `__init__`

- Set up the structure (number of layers, neurons)
- Randomly initialize weights and biases
- Store activation functions

```
class NeuralNetwork:
    def __init__(self):
        self.linear = Linear(...) # Set up the network structure.
        self.act = Sigmoid() # Set up the activation function.
        self.weights = [...] # Randomly initialize weights.
        self.bias = [...] # Randomly initialize biases.
```

Saturdays.AI
Kigali

Forward Propagation: `forward()`

- Takes input X
- Propagates it through the layers
- Applies activation functions
- Stores intermediate outputs (needed for backprop)

```
class NeuralNetwork:
    def __init__(self):
        ...

    def forward(self, x):
        z = self.layer(x) # Propagates x through the layers.
        out = self.act(z) # Applies the activation function.
        return out
```

Backward Propagation: `backward()`

- Implements backpropagation
- Calculates gradients for weights and biases
- Uses chain rule to compute partial derivatives

```
class NeuralNetwork:
    def __init__(self):
        ...

    def forward(self, x):
        ...

    def backward(self, d_out):
        d_z = self.act.backward(d_out) # Backpropagates dl/dout through the sigmoid.
        _ = self.linear.backward(d_z) # Backpropagates dout/dz through the layer.
        return out
```

Anatomy of a NN Training loop

At its core, the training loop performs:

1. Reset all gradient
2. Forward pass
3. Loss computation
4. Backward pass (backpropagation)
5. Parameter updates

This is repeated for a number of epochs:

```
for epoch in range(n_epochs):
    zero_grad() # Reset all gradients to zero.
    prediction = forward() # Forward pass.
    loss = compute_loss() # Compute the loss.
    backward() # Backward pass to compute the gradients.
    update_weights() # Update the weights using gradient.
```

Choosing activation functions for multilayer NN

Choosing Activation Functions in MLPs

Recap:

- Up to now, we've mostly used the **sigmoid** activation function.
- In our MLP example, sigmoid was used in both the **hidden layer** and the **output layer**.

Theory: General rule?

- In theory, **any differentiable function** can serve as an activation function in a multilayer network.
- **Even linear activation functions** (like in Adaline) are valid choices.

Practice: Why linear functions are rarely used in all (hidden) layers:

- No added nonlinearity → model remains linear (the sum of linear functions is still linear).
- Can't solve complex, non-linear problems.

Key takeaway: Nonlinear activation functions are essential for real-world, complex tasks.

Saturdays.AI
Kigali

Estimating Multiclass Probabilities: Softmax

Purpose:

- Convert raw model scores (*logits*) into **probabilities** for each class.
- Works for **multiclass classification** (more than 2 classes).
- Used in the **last (output) layer**

Key Properties:

- All probabilities are **positive**.
- All probabilities **sum to 1**.
- Higher score → higher probability.

Choosing Activation Functions in MLPs

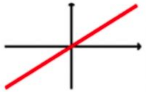
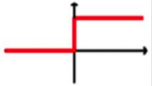
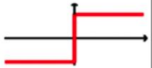
Activation function	Equation	Example	1D graph
Linear	$\sigma(z) = z$	Adaline, linear regression	
Unit step (Heaviside function)	$\sigma(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\sigma(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise linear	$\sigma(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\sigma(z) = \dfrac{1}{1 + e^{-z}}$	Logistic regression, multilayer NN	
Hyperbolic tangent (tanh)	$\sigma(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\sigma(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

Figure 5.6: The activation functions

Practice

Classification with Multilayer Perceptron (MLP)

Classifying handwritten digits.

[Hello world of deep learning]

Saturdays.AI
Kigali

Break

Practice

We give you code. What happens?

Run. Learn. Repeat.

Challenges & Next steps!

Saturdays.AI
Kigali

Best part of the day! 🥳

Kahoot!

kahoot.it

Saturdays.AI
Kigali

Any questions?

Saturdays.AI
Kigali

THANKS

kigali@saturdays.ai
coming soon
coming soon