Memory updated

Neural Networks - Fundamentals and Multi-Layer Perceptrons (MLPs)

This guide will cover:

1. **Fundamentals of Neural Networks**
    - Neurons, weights, and activation functions
    - Forward propagation
    - Loss functions
    - Backpropagation
    - Gradient descent
2. **Multi-Layer Perceptrons (MLPs)**
    - Structure and layers
    - Fully connected networks
    - Training an MLP
    - Manual implementation (fully commented)

---

1. Fundamentals of Neural Networks

What is a Neural Network?

A neural network is a function that **maps inputs to outputs** by simulating the behavior of biological neurons. Each neuron **takes inputs, applies a weight, adds bias, passes it through an activation function, and sends the result to the next layer**.

Basic Structure:

- **Input Layer** → Takes feature inputs.
- **Hidden Layers** → Perform computations.
- **Output Layer** → Provides final prediction.

Example:
For an image classifier:

- Input: **Pixels of an image**
- Hidden layers: **Learn features like edges, shapes, patterns**
- Output: **Predicted class (dog, cat, etc.)**

2. Neurons and Activation Functions

Neuron Computation (Perceptron)

Each neuron performs:

1. **Weighted sum of inputs:**

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

2. **Apply activation function:**

$$a = f(z)$$

Common Activation Functions

Function	Formula	When to Use
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	Output layer of binary classification
ReLU	$\max(0, z)$	Hidden layers (fast, avoids vanishing gradient)
Tanh	$\frac{e^z - e^{-z}}{e^z + e^{-z}}$	Hidden layers, when values should be centered around zero
Softmax	$\frac{e^{z_i}}{\sum e^{z_j}}$	Output layer for multi-class classification

3. Forward Propagation

Forward propagation moves data **through the network**:

1. Compute **weighted sum** z.
2. Apply **activation function**.
3. Pass output to the next layer.

For an MLP:

$$a^{(l+1)} = f(W^{(l)}a^{(l)} + b^{(l)})$$

4. Loss Functions

The loss function measures how far the predictions are from the actual values.

Loss Function	Formula	Used For
Mean Squared Error (MSE)	$L = \frac{1}{m}\sum(y - \hat{y})^2$	Regression tasks
Cross-Entropy Loss	$L = -\sum y \log \hat{y}$	Classification tasks

---

5. Backpropagation & Gradient Descent

To **train the network**, we use backpropagation and gradient descent:

1. Compute **loss** using forward propagation.
2. Compute **gradients** using chain rule.
3. Update weights:

$$w := w - \alpha \frac{\partial L}{\partial w}$$

Backpropagation **propagates errors backward** to update weights.

1. Universal Approximation Theorem

✅ What It States

- A **feedforward neural network with at least one hidden layer** can approximate **any continuous function** given **enough neurons** and a **non-linear activation function**.

🔥 Why It Matters

- This explains **why deep learning works**—with enough capacity, an NN can model **any complex relationship**.
- However, just because a neural network **can** approximate a function doesn't mean it will **generalize well**—this leads us to **overfitting and regularization**.

2. Overfitting and Generalization in Neural Networks

✅ Why Neural Networks Overfit

- **Too many parameters** → Can memorize training data instead of learning patterns.
- **Too few training samples** → High variance in learned weights.
- **Lack of regularization** → The model relies too much on the dataset and doesn't generalize.

🚀 How to Prevent Overfitting

Method	What It Does
Dropout	Randomly removes neurons during training to prevent reliance on specific pathways.
L1/L2 Regularization	Penalizes large weights to reduce model complexity (L2 = Ridge, L1 = Lasso).
Early Stopping	Stops training once validation loss stops improving.
Data Augmentation	Increases dataset diversity to improve generalization.
Batch Normalization	Helps control activation values, reducing sensitivity to small changes in inputs.

3. Optimization Methods and Training Stability

✅ Key Optimization Algorithms

Optimizer	How It Works	Pros	Cons
Gradient Descent (GD)	Uses all data to compute gradients	Stable, good for convex functions	Slow for large datasets
Stochastic Gradient Descent (SGD)	Uses a **single** random sample per step	Fast, less memory usage	High variance, noisy updates
Mini-Batch Gradient Descent	Uses **small batches** instead of full dataset	Balanced speed and accuracy	Requires tuning batch size
Adam (Adaptive Moment Estimation)	Combines momentum & adaptive learning rates	Fast, works well on noisy data	More hyperparameters, may not generalize well

🚀 Key Concepts You Should Know

- **Momentum**: Helps accelerate gradient descent in the right direction.
- **Learning Rate Scheduling**: Reducing the learning rate over time improves convergence.
- **Exploding and Vanishing Gradients**: Happens when gradients become too large (explode) or too small (vanish), affecting deep networks.

4. Activation Functions – Why They Matter

✅ Key Activation Functions

Function	Formula	When to Use	Pros	Cons
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	Binary classification (output layer)	Squashes values between (0,1)	Vanishing gradients
Tanh	$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	Hidden layers	Zero-centered	Still has vanishing gradients
ReLU	$f(z) = \max(0, z)$	Most hidden layers	Computationally efficient	Dead neurons (outputs stuck at 0)
Leaky ReLU	$f(z) = \max(0.01z, z)$	When ReLU suffers from dead neurons	Fixes dead neurons	Still unbounded
Softmax	$\frac{e^{z_i}}{\sum e^{z_j}}$	Multi-class classification (output layer)	Probabilities sum to 1	Computationally expensive

🔥 Why Does Activation Function Choice Matter?

- ReLU is **preferred** for hidden layers because it avoids vanishing gradients.
- Sigmoid and Tanh **work well for shallow networks** but are poor for deep ones.
- Softmax is the best choice for **multi-class classification**.

5. Neural Network Weight Initialization

✅ Why Initialization Matters

- Poor initialization can cause **slow training** or **dead neurons**.

🚀 Weight Initialization Strategies

Method	Formula	When to Use
Random Initialization	Small random values	Small networks
Xavier Initialization	$W \sim \mathcal{N}(0, \frac{1}{n})$	Sigmoid/Tanh activations
He Initialization	$W \sim \mathcal{N}(0, \frac{2}{n})$	ReLU-based networks

---

6. Batch Normalization (BN)

✅ What It Does

- **Normalizes activations** in each layer to stabilize training.
- Reduces **internal covariate shift** (shifting distributions during training).
- Improves **gradient flow**, making deep networks train faster.

🚀 When to Use It

- If training **deep networks** (many layers).
- If using **high learning rates** (BN allows higher rates without instability).
- If activation values become **too large or too small**.

7. Training Best Practices & Hyperparameter Tuning

✅ Hyperparameters to Tune

Hyperparameter	Effect
Learning Rate	Too high → won't converge, Too low → slow training
Batch Size	Small → noisy, Large → smoother but needs more memory
Number of Layers	More layers → better representation, but harder to train
Number of Neurons per Layer	Too few → underfitting, Too many → overfitting
Dropout Rate	Higher → more regularization, Lower → risk of overfitting

🚀 Tuning Strategies

1. **Grid Search** – Try all possible hyperparameter combinations (computationally expensive).
2. **Random Search** – Randomly pick hyperparameters, faster than Grid Search.
3. **Bayesian Optimization** – Smartly searches best hyperparameters based on past results.
4. **Gradient-Based Optimization** – AutoML techniques like **Hyperband, Optuna**.

8. Transfer Learning & Pretraining

✅ Why Transfer Learning Works

- Instead of training from scratch, start with a **pretrained model**.
- Fine-tune **only the last layers** for a new task.
- Saves **time and computation**.

🚀 When to Use It

- When you **don't have much data** (e.g., medical images, specialized datasets).
- When **pretrained models already exist** (e.g., ResNet, BERT for NLP).

---

9. Neural Network Interpretability

✅ Why It's Hard

- NNs are **black boxes**, hard to understand compared to linear models.
- We need ways to explain **why a model made a certain prediction**.

🚀 How to Interpret Neural Networks

Method	How It Helps
SHAP Values	Measures each feature's impact on predictions.
Feature Visualization	See which parts of an image activate neurons.
Saliency Maps	Highlights important pixels for a CNN.

6. Multi-Layer Perceptrons (MLPs)

An **MLP** is a type of neural network with:

- **Multiple hidden layers**
- **Fully connected neurons**
- **Non-linear activation functions**

It's the foundation for **deep learning models**.

Revised Pseudo Code with Gradient Descent Types

plaintext Copy Edit

```
# Initialize Neural Network
1. Set hyperparameters (learning rate, epochs, hidden layer size, batch size).
2. Initialize weights (random small values) and biases (zeros).

# Training Process
FOR each epoch:
    FOR each batch (if using Mini-Batch GD or SGD):
        1. Forward Propagation:
            a. Compute weighted sum for hidden layer: Z1 = W1 * X + b1
            b. Apply activation function: A1 = activation(Z1)
            c. Compute weighted sum for output layer: Z2 = W2 * A1 + b2
            d. Apply activation function: A2 = activation(Z2) (final prediction)

        2. Compute Loss:
            a. Use Cross-Entropy Loss for classification
            b. Use Mean Squared Error (MSE) for regression

        3. Backpropagation:
            a. Compute gradient of loss w.r.t. A2 (output layer).
            b. Compute gradient of loss w.r.t. W2 and b2.
            c. Compute error signal backpropagated to hidden layer.
            d. Compute gradient of loss w.r.t. W1 and b1.

        4. Update Weights (Gradient Descent Step):
            a. **Batch Gradient Descent (BGD):** Update weights using all training samples.
                W1 := W1 - learning_rate * dW1
                b1 := b1 - learning_rate * db1
                W2 := W2 - learning_rate * dW2
                b2 := b2 - learning_rate * db2

            b. **Stochastic Gradient Descent (SGD):** Update weights using one random sample at a time.
                Select a random (X_i, y_i)
                Compute gradients using (X_i, y_i)
                W1 := W1 - learning_rate * dW1 (for sample i)
                b1 := b1 - learning_rate * db1
                W2 := W2 - learning_rate * dW2
                b2 := b2 - learning_rate * db2
```

```
            c. **Mini-Batch Gradient Descent:** Update weights using a small batch of samples.
                Select a batch of `m` random samples.
                Compute average gradients over batch.
                W1 := W1 - learning_rate * dW1 (batch average)
                b1 := b1 - learning_rate * db1
                W2 := W2 - learning_rate * dW2
                b2 := b2 - learning_rate * db2

        5. Print loss every 100 epochs for monitoring.

# Making Predictions
1. Perform forward propagation on new data.
2. If using classification, convert probability to class label.
```

3. [25 points] A Simple Neural Network

Let $X = \{x^{(1)}, \cdots, x^{(n)}\}$ be a dataset of n samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 1:

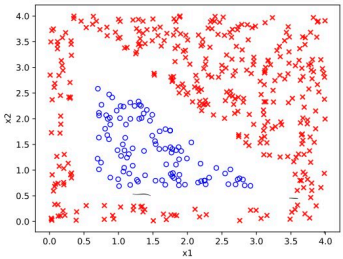

Figure 1: Plot of dataset X.

The examples in class 1 are marked as as "×" and examples in class 0 are marked as "○". We want to perform binary classification using a simple neural network with the architecture shown in Figure 2:

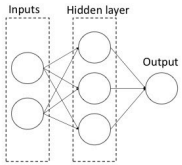

Figure 2: Architecture for our simple neural network.

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{n} \sum_{i=1}^{n} \left(o^{(i)} - y^{(i)} \right)^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

(a) [5 points] Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w_{1,2}^{[1]}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

For hidden layer:

$$z_j^{(i)} = w_{0j} + \sum_{d=1}^{3} w_{jd}^{[1]} x^{(i)}$$

$$\Rightarrow h_j^{(i)} = \sigma(z_j^{(i)}) = \frac{1}{1+e^{-z_j^{(i)}}}$$

$\Rightarrow$ Output layer:

$$z^{*(i)} = w_0^{[2]} + \sum_{j=1}^{3} w_j^{[2]} h_j^{(i)}$$

$$\Rightarrow o = \sigma(z^{*(i)}) = \frac{1}{1+e^{-z^{*(i)}}}$$

$$l = \frac{1}{n}\sum_{i=1}^{n} \left(o^{(i)} - y^{(i)}\right)^2 \Rightarrow l^{(i)} = \left(o^{(i)} - y^{(i)}\right)^2$$

$\Rightarrow$ To get $\frac{\partial l^{(i)}}{\partial w_{ij}^{[1]}}$, backprop:

$$\Rightarrow \frac{\partial l^{(i)}}{\partial w_{ij}^{[1]}} = \frac{\partial l^{(i)}}{\partial o^{(i)}} \frac{\partial o^{(i)}}{\partial z^{*(i)}} \frac{\partial z^{*(i)}}{\partial h_j^{(i)}} \frac{\partial h_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial z_j^{(i)}}{\partial w_{ij}^{[1]}}$$

$$\frac{\partial l^{(i)}}{\partial o^{(i)}} = 2\left(o^{(i)} - y^{(i)}\right)$$

$$\frac{\partial o^{(i)}}{\partial z^{*(i)}} = \left\{\text{quotient rule}\right\} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{(1+e^{-z})}\left(\frac{e^{-z}}{1+e^{-z}}\right) = \sigma^{(i)}\left(1-\sigma^{(i)}\right)$$

$$\frac{\partial z^{*(i)}}{\partial h_j^{(i)}} = w_j^{[2]}$$

$$\frac{\partial h_j^{(i)}}{\partial z_j^{(i)}} = \left\{\text{similar to } \frac{\partial o^{(i)}}{\partial z^{*(i)}}\right\} = h_j^{(i)}\left(1-h_j^{(i)}\right)$$

$$\frac{\partial z_j^{(i)}}{\partial w_{ij}^{[1]}(i)} = x_j^{(i)}$$

$$\Rightarrow \frac{\partial l^{(i)}}{\partial w_{ij}^{[1]}} = 2\left(\sigma^{(i)} - y^{(i)}\right)\sigma^{(i)}\left(1-\sigma^{(i)}\right) w_j^{[2]} h_j^{(i)}\left(1-h_j^{(i)}\right) x_j^{(i)}$$

$$\Rightarrow w_{ij}^{[1](t+1)} = w_{ij}^{[1](t)} - \alpha \frac{\partial l^{(i)}}{\partial w_{ij}^{[1]}}$$

$\Rightarrow$ Case where $i=1, j=2$:

$$w_{1,2}^{[1](t+1)} = w_{ij}^{[1](t+1)} - \alpha\, 2\left(\sigma^{(i)} - y^{(i)}\right)\sigma^{(i)}\left(1-\sigma^{(i)}\right) w_2^{[2]} h_2^{(i)}\left(1-h_2^{(i)}\right) x_1^{(i)}$$

```python
import numpy as np

class MLP:
    """
    A simple Multi-Layer Perceptron (MLP) with one hidden layer.
    Uses sigmoid activation in output layer (binary classification).
    """
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        """
        Initialize network parameters.
        :param input_size: Number of input features.
        :param hidden_size: Number of neurons in hidden layer.
        :param output_size: Number of output neurons.
        :param learning_rate: Step size for weight updates.
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases with small random values
        self.W1 = np.random.randn(hidden_size, input_size) * 0.01  # Weights: Input → Hidden
        self.b1 = np.zeros((hidden_size, 1))  # Bias: Hidden Layer
        self.W2 = np.random.randn(output_size, hidden_size) * 0.01  # Weights: Hidden → Output
        self.b2 = np.zeros((output_size, 1))  # Bias: Output Layer

    def sigmoid(self, z):
        """Compute sigmoid activation function."""
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, a):
        """Compute derivative of sigmoid function."""
        return a * (1 - a)

```

```
    def forward(self, X):
        """
        Perform forward propagation.
        :param X: Input feature matrix.
        :return: Output probabilities.
        """
        self.Z1 = np.dot(self.W1, X) + self.b1  # Compute hidden layer input
        self.A1 = self.sigmoid(self.Z1)  # Apply activation function

        self.Z2 = np.dot(self.W2, self.A1) + self.b2  # Compute output layer input
        self.A2 = self.sigmoid(self.Z2)  # Output probabilities

        return self.A2

    def backward(self, X, y):
        """
        Perform backpropagation to compute gradients and update weights.
        :param X: Input features.
        :param y: True labels.
        """
        m = X.shape[1]  # Number of samples

        # Compute loss gradient at output layer
        dZ2 = self.A2 - y  # Derivative of binary cross-entropy loss
        dW2 = (1 / m) * np.dot(dZ2, self.A1.T)  # Gradient w.r.t. W2
        db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)  # Gradient w.r.t. b2

        # Backpropagate to hidden layer
        dA1 = np.dot(self.W2.T, dZ2)  # Gradient w.r.t. hidden layer activations
        dZ1 = dA1 * self.sigmoid_derivative(self.A1)  # Gradient of activation function
        dW1 = (1 / m) * np.dot(dZ1, X.T)  # Gradient w.r.t. W1
        db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)  # Gradient w.r.t. b1

        # Update weights using gradient descent
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2
```

```python
    def train(self, X, y, epochs=1000):
        """
        Train the MLP using gradient descent.
        :param X: Input features.
        :param y: Target labels.
        :param epochs: Number of iterations.
        """
        for epoch in range(epochs):
            self.forward(X)  # Forward propagation
            self.backward(X, y)  # Backpropagation

            # Print loss every 100 iterations
            if epoch % 100 == 0:
                loss = -np.mean(y * np.log(self.A2) + (1 - y) * np.log(1 - self.A2))
                print(f"Epoch {epoch}: Loss = {loss:.4f}")

    def predict(self, X):
        """
        Make predictions.
        :param X: Feature matrix.
        :return: Predicted labels.
        """
        A2 = self.forward(X)
        return (A2 > 0.5).astype(int)  # Convert probabilities to binary predictions
```

2. Scikit-Learn Implementation of Multi-Layer Perceptron (MLP)

python Copy Edit

```python
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Generate synthetic classification dataset
X, y = make_classification(n_samples=1000, n_features=10, random_state=42)

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Standardize features (important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize MLP Classifier
mlp = MLPClassifier(hidden_layer_sizes=(10,), activation='relu', solver='adam',
                    learning_rate_init=0.01, max_iter=1000, random_state=42)

# Train the model
mlp.fit(X_train, y_train)

# Evaluate the model
accuracy = mlp.score(X_test, y_test)
print(f"MLP Accuracy: {accuracy:.4f}")
```

Final Takeaways

✅ Neural Networks can approximate any function, but generalization requires regularization.
✅ Optimization and activation functions determine training efficiency.
✅ Proper initialization, batch normalization, and tuning improve model stability.
✅ Interpretability remains a challenge, but tools like SHAP and saliency maps help.
