---

1. Artificial Neurons

Structure and Mathematical Representation

An artificial neuron is inspired by biological neurons and consists of the following components:

- **Inputs (x1,x2,...,xnx_1, x_2, ..., x_n)**: Features or data points fed into the neuron.
- **Weights (w1,w2,...,wnw_1, w_2, ..., w_n)**: Each input has an associated weight that determines its importance.
- **Bias (bb)**: A constant added to shift the activation.
- **Summation Function**: Computes the weighted sum of inputs: z=∑(wixi)+bz = \sum (w_i x_i) + b
- **Activation Function (f(z)f(z))**: Introduces non-linearity to decide whether the neuron should be "activated" or not.

Mathematical Representation

y=f(∑i=1nwixi+b)y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)

Example

Let's implement a simple artificial neuron in Python:

```
import numpy as np


# Define inputs, weights, and bias

inputs = np.array([0.5, 0.3, 0.2])

weights = np.array([0.4, 0.7, 0.2])

bias = 0.1


# Compute the weighted sum

z = np.dot(inputs, weights) + bias

print(f"Summation (z): {z}")


# Apply activation function (Sigmoid)

activation = 1 / (1 + np.exp(-z))

print(f"Activated Output: {activation}")
```

---

2. Activation Functions

Activation functions introduce non-linearity into neural networks.

Types of Activation Functions

1. Linear Activation

f(z)=zf(z) = z

- Used in regression tasks.
- No non-linearity, so it's rarely used in deep networks.

2. Sigmoid Activation

f(z)=11+e−zf(z) = \frac{1}{1 + e^{-z}}

- Outputs values in the range (0,1).
- Suitable for binary classification.
- Problem: **Vanishing gradient** in deep networks.

```
import matplotlib.pyplot as plt


def sigmoid(z):
    return 1 / (1 + np.exp(-z))


z = np.linspace(-10, 10, 100)
plt.plot(z, sigmoid(z))
plt.title("Sigmoid Activation Function")
plt.grid()
plt.show()
```

3. Tanh Activation

f(z)=ez−e−zez+e−zf(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}

- Output range (-1,1), centered around 0.
- Better than Sigmoid but still suffers from vanishing gradients.

```
def tanh(z):
    return np.tanh(z)
```

```
plt.plot(z, tanh(z))

plt.title("Tanh Activation Function")

plt.grid()

plt.show()
```

4. ReLU (Rectified Linear Unit)

$f(z) = \max(0, z)$

- Solves vanishing gradient for positive values.
- Computationally efficient.
- **Problem:** Neurons can "die" when $z < 0$ (Dying ReLU).

```
def relu(z):
   return np.maximum(0, z)


plt.plot(z, relu(z))

plt.title("ReLU Activation Function")

plt.grid()

plt.show()
```

5. Leaky ReLU

$f(z) = \max(0.01z, z)$

- Helps with the **Dying ReLU problem** by allowing small gradients when $z < 0$.

```
def leaky_relu(z, alpha=0.01):
   return np.where(z > 0, z, alpha * z)


plt.plot(z, leaky_relu(z))

plt.title("Leaky ReLU Activation Function")

plt.grid()

plt.show()
```

6. Softmax Activation

$f(z_i) = \frac{e^{z_i}}{\sum_{j} e^{z_j}}$

- Used in multi-class classification.

```
def softmax(z):
    exp_z = np.exp(z - np.max(z))
    return exp_z / exp_z.sum()

z = np.array([1.0, 2.0, 3.0])
print("Softmax output:", softmax(z))
```

---

3. ANN Architecture

Layers of an ANN

- **Input Layer**: Takes input features.
- **Hidden Layers**: Perform computations using activation functions.
- **Output Layer**: Produces final predictions.

Weight Initialization

- **Random Initialization** (Standard approach).
- **Xavier Initialization** (for Sigmoid/Tanh).
- **He Initialization** (for ReLU).

```
import tensorflow as tf

# Xavier Initialization
initializer = tf.keras.initializers.GlorotUniform()
weights = initializer(shape=(3, 3))
print(weights.numpy())
```

---

4. Forward and Backward Propagation

Forward Propagation

1. Compute weighted sum: z=W·X+bz = W \cdot X + b
2. Apply activation function.

Backward Propagation (Learning Process)

- Uses **Gradient Descent** and **Chain Rule** to update weights.

Chain Rule for Gradients

If y=f(g(x))y = f(g(x)), then

dydx=f′(g(x))⋅g′(x)\frac{dy}{dx} = f'(g(x)) \cdot g'(x)

Example

```
import torch

# Define loss function
loss_fn = torch.nn.MSELoss()

# Dummy data
y_pred = torch.tensor([0.4, 0.7, 0.2], requires_grad=True)
y_true = torch.tensor([0.5, 0.6, 0.3])

# Compute loss
loss = loss_fn(y_pred, y_true)
loss.backward()  # Backpropagation

# Print gradients
print(y_pred.grad)
```

---

5. Training Neural Networks (Hands-on Implementation)

Let's implement a simple neural network using **TensorFlow (Keras)**.

Dataset: MNIST Digit Classification

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Load dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
# Normalize
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define model
model = models.Sequential([
   layers.Flatten(input_shape=(28, 28)),
   layers.Dense(128, activation='relu'),
   layers.Dense(10, activation='softmax')
])

# Compile model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train model
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```

Using PyTorch

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms

# Load data
train_loader = torch.utils.data.DataLoader(
   datasets.MNIST('.', train=True, transform=transforms.ToTensor()),
   batch_size=32, shuffle=True)
```

```
# Define Model
class NeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x.view(-1, 28*28)))
        return torch.softmax(self.fc2(x), dim=1)

# Training
model = NeuralNet()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for epoch in range(5):
    for images, labels in train_loader:
        optimizer.zero_grad()
        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
```
