Unit 2: Artificial Neural Network

What is an Artificial Neural Network?

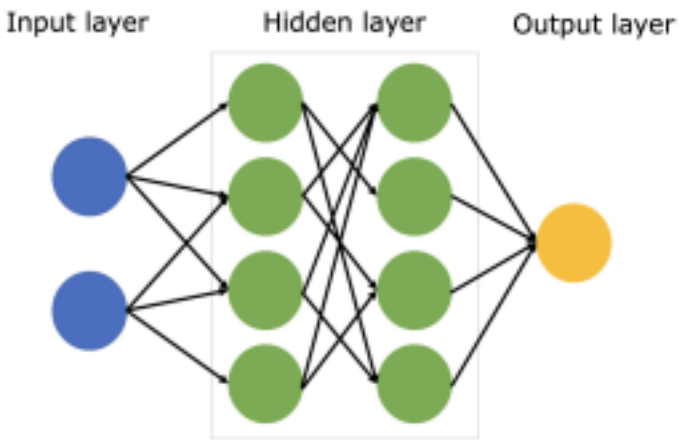
An **Artificial Neural Network (ANN)** is a machine learning model inspired by the structure and functioning of the human brain. It consists of interconnected processing units called **neurons** that work together to learn patterns and relationships in data.

`Artificial Neuron (Perceptron)

- **Biological vs. Artificial Neurons**:
  - **Biological Neurons**: Receive input from dendrites, process signals in the cell body, and transmit output via axons.
  - **Artificial Neurons**: Receive weighted inputs, sum them, add a bias, and pass the result through an activation function.

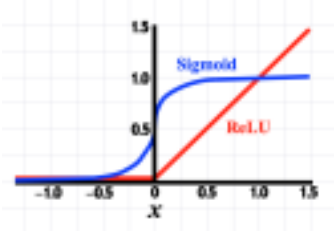
Key Components of ANN:

1. **Neurons (Nodes/Units):** The fundamental processing elements that receive input, apply weights, and generate outputs.
2. **Layers:**
   - **Input Layer:** Receives raw data.
   - **Hidden Layers:** Perform computations and extract features.
   - **Output Layer:** Produces the final result.



Artificial neural networks

3. **Weights & Bias:** Determine the influence of each input on the neuron's output.
4. **Activation Functions:** Introduce non-linearity, allowing the network to learn complex patterns.



Common activation functions:

Logistic sigmoid = $1/(1+e^{-x})$ and ReLU = $max(0,x)$

5. **Training Algorithm:** Adjusts weights using techniques like **gradient descent** and **backpropagation** to minimize errors..

Applications of Neural Networks

Artificial Neural Networks (ANNs) are widely used across various domains due to their ability to learn patterns, recognize features, and make predictions. Below are some key application areas:

1. Image and Video Processing

- **Facial Recognition**: Used in security systems, social media, and smartphone authentication.
- **Object Detection**: Self-driving cars use ANNs to identify pedestrians, vehicles, and road signs.
- **Medical Imaging**: Helps in detecting diseases like cancer from X-rays, MRIs, and CT scans.
- **Image Restoration**: Enhances image quality by reducing noise and improving resolution.

2. Natural Language Processing (NLP)

- **Chatbots and Virtual Assistants**: Power AI-based assistants like Siri, Alexa, and Google Assistant.
- **Machine Translation**: Used in Google Translate and DeepL for multilingual translations. ● **Speech-to-Text Systems**: Converts spoken language into text for applications like voice search.
- **Sentiment Analysis**: Helps businesses analyze customer feedback on social media and reviews.

3. Healthcare and Medicine

- **Disease Diagnosis**: Identifies patterns in patient data to detect conditions like diabetes, cancer, and heart disease.

- **Drug Discovery**: Predicts potential drug interactions and effectiveness in new medicine development.
- **Medical Robotics**: Assists in surgeries and rehabilitation through AI-driven robots.

4. Finance and Banking

- **Fraud Detection**: Identifies unusual transactions to prevent credit card fraud and cybercrime.
- **Stock Market Prediction**: Analyzes financial trends to predict stock movements. ● **Loan and Credit Scoring**: Assesses creditworthiness of borrowers for banks and financial institutions.

5. Autonomous Vehicles and Robotics

- **Self-Driving Cars**: Uses ANNs for lane detection, obstacle avoidance, and decision-making.
- **Industrial Automation**: Enhances efficiency in manufacturing through robotic arms and AI-driven machinery.

6. Gaming and AI Agents

- **Game AI**: Powers intelligent opponents in video games.
- **Deep Reinforcement Learning**: Helps AI agents learn from experience (e.g., AlphaGo defeating human players).

7. Recommendation Systems

- **E-commerce**: Suggests personalized products on platforms like Amazon. ● **Streaming Services**: Recommends movies and music on Netflix, YouTube, and Spotify.

8. Cybersecurity

- **Intrusion Detection**: Identifies unusual network activity to prevent cyberattacks. ● **Malware Detection**: Recognizes harmful software patterns in cybersecurity applications.

9. Weather Forecasting and Disaster Prediction

- **Climate Modeling**: Predicts weather conditions and natural disasters like hurricanes and earthquakes.
- **Agriculture**: Monitors soil conditions and crop health using AI models.

Types of neural network:

1. Perceptron Neural Network
2. Feed Forward NEural Network
3. Backpropagation neural network
4. Deep Neural Network
    a. Convolutional Neural Network
    b. Recurrent Neural Network

Activation Functions in Artificial Neural Networks

Activation functions are a critical component of artificial neural networks. They introduce **non-linearity** into the network, enabling it to learn complex patterns and relationships in data. Without activation functions, a neural network would simply be a linear regression model, regardless of the number of layers.

1. Purpose of Activation Functions

- **Introduce Non-linearity**: Allow the network to model complex, non-linear relationships in data.
- **Determine Neuron Output**: Transform the weighted sum of inputs into an output that is passed to the next layer.
- **Enable Backpropagation**: Provide differentiable outputs for gradient-based optimization.

2. Common Activation Functions

2.1. Binary Step Function

- Binary step function depends on a threshold value that decides whether a neuron should be activated or not.
- The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.
- It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
- **Formula**:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

2.2. Sign Function

- **Formula**:

$$f(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

- **Use Case**: Used in perceptrons for binary classification.

2.3. Sigmoid (Logistic Function)

- It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- **Formula**:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **Output Range**:

$(0,1)$

- **Advantages**:
  - Smooth gradient, differentiable.
  - Useful for binary classification (output can be interpreted as probability).
- **Disadvantages**:
  - Suffers from **vanishing gradients** for extreme inputs.
  - Computationally expensive.

2.4. Tanh (Hyperbolic Tangent)

- **Formula**:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **Output Range**:
- $(-1,1)$
- **Advantages**:
  - Zero-centered, making it easier to model both positive and negative values.

    - Stronger gradients than sigmoid.
- **Disadvantages**:
    - Still suffers from **vanishing gradients** for extreme inputs.

2.5. ReLU (Rectified Linear Unit)

- **Formula**:

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

- **Output Range**:
- $[0, \infty)$
- **Advantages**:
    - Computationally efficient.
    - Avoids vanishing gradient problem for positive inputs.
    - Most commonly used in hidden layers.
- **Disadvantages**:
    - **Dying ReLU Problem**: Neurons can get stuck in the negative side (always output 0).

2.6. Leaky ReLU

- **Formula**:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

where

α is a small constant (e.g., 0.01).

- **Advantages**:
    - Solves the dying ReLU problem by allowing a small gradient for negative inputs.
- **Disadvantages**:
    - Requires tuning of
    - α
    - α.

2.7. SoftMax

- **Formula**:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

- **Output Range**:
  $(0,1)$
- **Use Case**:
  - Used in the output layer for **multi-class classification**.
  - Outputs a probability distribution over classes.

Mathematical Calculations

Let's calculate the output of each activation function for the input values **-10**, **10**, and **100**. We'll go through each activation function step by step and compute the results mathematically.

---

1. Binary Step Function

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Calculations:

- For $x = -10$:

$$f(-10) = 0 \quad (\text{since } -10 < 0)$$

- For $x = 10$:

$$f(10) = 1 \quad (\text{since } 10 \geq 0)$$

- For $x = 100$:

$$f(100) = 1 \quad (\text{since } 100 \geq 0)$$

Results:

$$f(-10) = 0, \quad f(10) = 1, \quad f(100) = 1$$

Summary of Results

Activation Function	$f(-10)$	$f(10)$	$f(100)$
Binary Step	0	1	1
Sigmoid	0.000045	0.999955	1
Tanh	-1	1	1
ReLU	0	10	100
Leaky ReLU	-0.1	10	100
SoftMax	0	0	1

```
import numpy as np
import matplotlib.pyplot as plt

# Define activation functions
def binary_step(x):
 return np.where(x >= 0, 1, 0)

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

def tanh(x):
 return np.tanh(x)

def relu(x):
 return np.maximum(0, x)

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

def softmax(x):
 exp_x = np.exp(x - np.max(x)) # For numerical stability
return exp_x / np.sum(exp_x)

# Create a range of values for x
x = np.linspace(-10, 10, 100)

# Plot Binary Step Function
```

```
plt.figure(figsize=(12, 8))

plt.subplot(2, 3, 1)
plt.plot(x, binary_step(x))
plt.title("Binary Step Function")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()

# Plot Sigmoid Function
plt.subplot(2, 3, 2)
plt.plot(x, sigmoid(x))
plt.title("Sigmoid Function")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()

# Plot Tanh Function
plt.subplot(2, 3, 3)
plt.plot(x, tanh(x))
plt.title("Tanh Function")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()

# Plot ReLU Function
plt.subplot(2, 3, 4)
plt.plot(x, relu(x))
plt.title("ReLU Function")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()

# Plot Leaky ReLU Function
plt.subplot(2, 3, 5)
plt.plot(x, leaky_relu(x))
plt.title("Leaky ReLU Function")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()

# Plot SoftMax Function
plt.subplot(2, 3, 6)
```

```
x_softmax = np.array([1.0, 2.0, 3.0])
y_softmax = softmax(x_softmax)
plt.bar(range(len(x_softmax)), y_softmax)
plt.title("SoftMax Function")
plt.xlabel("Class")
plt.ylabel("Probability")
plt.grid()
# Adjust layout and display plots
plt.tight_layout()
plt.show()
```

1. Single-Layer Neural Network (SLNN)

A **Single-Layer Neural Network** consists of only two layers: the **input layer** and the **output layer**. It is often referred to as a **Perceptron** if it is used for binary classification tasks.

Structure:

- **Input Layer**: The input layer consists of neurons that represent features or attributes of the input data.
- **Output Layer**: The output layer consists of neurons that generate the prediction or

classification.

Key Characteristics:

- **No Hidden Layers**: It has no hidden layers, which means the network directly maps inputs to outputs.
- **Linear Decision Boundary**: Since there is no non-linear transformation, it can only solve linearly separable problems (problems that can be separated by a straight line or hyperplane).

Limitations:

- **Cannot handle complex data**: A single-layer network can only learn linearly separable problems.
- **Limited expressiveness**: It struggles with more complex patterns like non-linear relationships between features.

Example: Perceptron for binary classification:

Perceptron:

The **Perceptron** is a simple binary classifier, often used for problems where the data is linearly separable. It uses a linear decision boundary to classify data points. Here's an example of how a perceptron works for binary classification:

- **Input Vector**: The input vector $X = [x_1, x_2]$ consists of features x_1 and x_2, where each feature represents a specific input value.
- **Weights**: The perceptron has weights $W = [w_1, w_2]$ that determine the importance of each input feature x_1 and x_2. w_1 is associated with x_1, and w_2 is associated with x_2.
- **Bias**: The bias term b allows the perceptron to shift the decision boundary, providing more flexibility in classification.
- **Output**: The perceptron output Y is calculated using the weighted sum of the inputs plus the bias:

$$Y = \text{sign}(w_1 x_1 + w_2 x_2 + b)$$

The **sign function** outputs:

- $+1$ if the result is greater than or equal to 0
- -1 if the result is less than 0

This binary classification model produces outputs based on the threshold determined by the weighted sum of inputs and bias.

Example:

Input:

- $X = [1, -1]$
- $W = [0.5, 0.5]$
- $b = 0$

Step 1: Compute the Weighted Sum

$$z = 0.5 \cdot 1 + 0.5 \cdot (-1) + 0 = 0.5 - 0.5 + 0 = 0$$

Step 2: Apply the Activation Function

$$Y = \text{sign}(0) = 1$$

So, the **predicted label** is $Y = 1$.

2. Multi-Layer Neural Network (MLNN)

A **Multi-Layer Neural Network** consists of three or more layers: an **input layer**, one or more **hidden layers**, and an **output layer**. This network is capable of learning more complex, non-linear patterns.

Structure:

- **Input Layer**: Receives the input data.
- **Hidden Layers**: One or more layers that process the data by applying weights and biases, followed by an activation function.
- **Output Layer**: Produces the final prediction or classification.

Key Components:

1. **Input Layer**: Contains neurons that represent features of the input data and pass it to the hidden layers for processing.
2. **Hidden Layers**: These layers, located between the input and output, consist of neurons that process the data by performing weighted sums and applying activation functions (e.g., sigmoid, ReLU).
3. **Weights and Biases**: Each connection between neurons has a weight, and each neuron has a bias term. These parameters are adjusted during training to minimize errors in predictions.
4. **Activation Functions**: Introduce nonlinearity into the network, allowing it to model complex data relationships. Common functions include sigmoid, tanh, and ReLU. 5. **Output Layer**: Produces the final output (e.g., class label for classification or value for regression). The number of neurons depends on the task, such as two for binary classification or one for regression.

Training Process:

1. **Forward Propagation**: The input data passes through the network, and the final output is generated.
2. **Loss Calculation**: The error between predicted and actual outputs is calculated using a loss function (e.g., cross-entropy for classification, MSE for regression).
3. **Backpropagation**: The error is propagated back to adjust the weights and biases using **gradient descent**, which helps minimize the loss iteratively.

The **"deep"** in deep neural networks refers to networks with many hidden layers, enabling them to learn complex, hierarchical representations from raw data. This ability has made deep learning successful in a variety of tasks like **computer vision**, **speech recognition**, and **natural language processing**.

BackPropagation

⁇ What is Backpropagation?

Backpropagation (Backward Propagation of Errors) is the **core algorithm** for training neural networks. It efficiently updates the weights of the network by computing the gradient of the loss

function with respect to each weight, using the chain rule of differentiation.

✅ Steps of Backpropagation

1. **Forward Propagation**
   - The input passes through the network layer by layer, producing an output (prediction).
   - The loss function calculates the difference between the predicted output and the actual label.
2. **Compute Loss**
   - Loss measures how far the prediction is from the actual output.
   - Common loss functions:
     - **MSE (Mean Squared Error)** for regression
     - **Cross-Entropy Loss** for classification
3. **Backward Propagation (Gradient Calculation)**
   - Using the **chain rule of differentiation**, gradients of the loss function with respect to weights and biases are computed, layer by layer, from the output to the input.
   - This tells us how much to adjust each parameter to reduce the loss.
4. **Weight Update (Gradient Descent Optimization)**
   - Parameters (weights and biases) are updated using **Gradient Descent** (or its variants like Adam, RMSprop, etc.).
   -

     Formula:

     $$W = W - \alpha \cdot \frac{\partial L}{\partial W}$$

     where α is the learning rate and $\frac{\partial L}{\partial W}$ is the gradient of the loss with respect to the weight.
5. **Repeat Until Convergence**
   - Steps 1-4 are repeated for multiple iterations (epochs) until the model minimizes the loss.

Example:

https://theneuralblog.com/forward-pass-backpropagation-example/

Gradient Descent:

Gradient Descent is an optimization algorithm used to minimize the loss function in neural networks by adjusting the model's weights and biases. It is a crucial component of **Backpropagation**, which is the process of training neural networks.

Types of Gradient Descent

There are **three main types** of gradient descent:

Problems using Gradient Descent

1. Vanishing Gradient Problem
2. Exploding Gradient Descent

1. Vanishing Gradient Problem:

The **vanishing gradient problem** is a common issue in training deep neural networks, where the gradients become extremely small during backpropagation, causing the earlier layers of the network to learn very slowly or stop learning altogether. This problem is especially prevalent in deep networks using **sigmoid** or **tanh** activation functions.

✅ Why Does It Happen?

During backpropagation, gradients are propagated backward from the output layer to the input layer. If the activation functions have very small derivatives (as in sigmoid and tanh), the gradients get smaller at each layer. In very deep networks, these small gradients **shrink exponentially**, making it hard for earlier layers to update their weights effectively.

✅ Consequences

- Early layers learn very slowly or not at all.
- Training deep networks becomes inefficient.
- The model may fail to converge or take an excessively long time.

[Vanishing Gradient Problem || Quickly Explained]

2. Exploding Gradient Problem

The **exploding gradient problem** occurs when gradients become excessively large during backpropagation, leading to unstable updates in deep neural networks. As a result, the model may fail to converge, experience extremely large weight updates, or even produce **NaN (Not-a-Number) errors** during training. This issue is especially common in **very deep networks** and **recurrent neural networks (RNNs)**.

✅ Why Does It Happen?

During **backpropagation**, gradients are passed from the output layer back to earlier layers. In

deep networks, **small weight updates can accumulate**, but if the weights and their gradients are **too large**, their magnitudes grow exponentially. This results in unstable learning and drastic changes in weight values.

Key causes of exploding gradients:

- **Poor weight initialization** → If initial weights are too large, the gradients grow exponentially.
- **Deep networks** → More layers mean more backpropagation steps, increasing the risk of large gradients.
- **High learning rates** → Large weight updates amplify instability.

Impact of Learning Rate in Machine Learning

The learning rate (α\alphaα) is a crucial hyperparameter in machine learning, especially in optimization algorithms like **Gradient Descent**. It controls how much the model updates its weights in response to the computed error during training. Choosing an appropriate learning rate is critical for achieving optimal model performance.

1. Effects of Different Learning Rates

(a) Too High Learning Rate (α alphaα is Large)

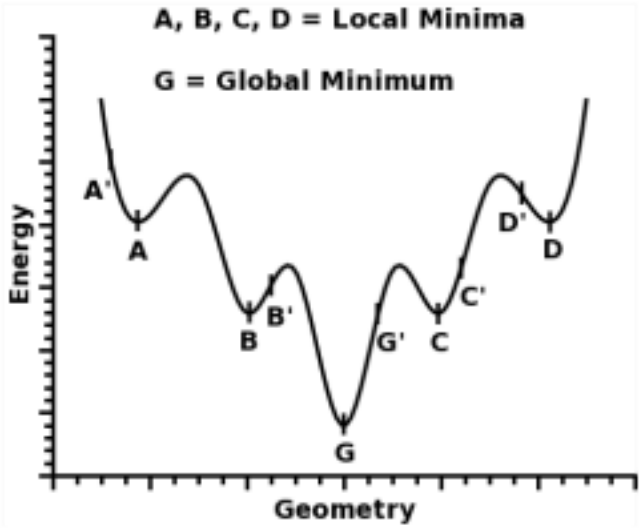
- The model makes large updates to weights.
- It can cause the loss function to **oscillate** and **diverge**, failing to converge to the optimal solution.
- Can skip over the minima and never settle.

Example:

- Learning rate = **1.0**
- Loss function fluctuates wildly and doesn't decrease steadily.

(b) Too Low Learning Rate (α alphaα is Small)

- The model updates weights very slowly.
- Convergence is **slow**, leading to longer training times.
- May get stuck in **local minima**, unable to reach the global minimum in reasonable time.



Example:

- Learning rate = **0.0001**
- Loss function decreases very slowly, requiring more iterations.

(c) Optimal Learning Rate

- Achieves a **balance** between speed and stability.
- Ensures that the model converges to a good minimum efficiently. ● Avoids excessive oscillations while maintaining reasonable speed.

Example:

- Learning rate = **0.01**
- Loss function steadily decreases and reaches the optimal point.

2. Choosing the Right Learning Rate

- **Learning Rate Scheduling:** Gradually decrease the learning rate over time. ● **Adaptive Learning Rates:** Methods like **Adam, RMSprop, and Adagrad** adjust the learning rate dynamically.
- **Grid Search or Hyperparameter Tuning:** Experimenting with different values to find the best learning rate.

3. Visualization of Learning Rate Impact

A graph plotting **loss vs. epochs** for different learning rates shows:

- High learning rate: Fluctuations and divergence.
- Low learning rate: Very slow decrease in loss.
- Optimal learning rate: Smooth and fast convergence.

Conclusion

- The learning rate plays a **critical role** in training efficiency and model accuracy. ● A well-chosen learning rate **improves convergence and performance**. ● Techniques like **learning rate decay** and **adaptive optimizers** help in selecting the best rate dynamically.

Would you like a **Python visualization** of different learning rates' impact on gradient descent?

Regularization in Neural Networks

Regularization is a set of techniques used to prevent overfitting in neural networks by reducing model complexity and improving generalization to unseen data. Overfitting occurs when a neural network learns the noise and specific details of the training data instead of the underlying pattern, leading to poor performance on new data.

Common Regularization Techniques in Neural Networks

1. L1 and L2 Regularization (Weight Decay)

- **L1 Regularization (Lasso)**: Adds the sum of the absolute values of weights to the loss function. This encourages sparsity (some weights become exactly zero), which can be useful for feature selection.
  Loss function with L1:
  $$L = \text{Loss} + \lambda \sum |w|$$

- **L2 Regularization (Ridge)**: Adds the sum of the squared values of weights to the loss function. This discourages large weight values and smooths the model.
  Loss function with L2:
  $$L = \text{Loss} + \lambda \sum w^2$$
  L2 regularization is commonly used in deep learning as **weight decay**.

2. Dropout

Dropout is a regularization technique used in training neural networks to prevent overfitting. During training, it randomly "drops out" or deactivates a fraction of the neurons in a network. This means that in each training step, certain neurons are ignored (i.e., their outputs are set to zero), and they don’t contribute to the forward pass or the backpropagation process. This prevents the network from becoming too reliant on any single neuron and forces the model to learn more robust, distributed features.

Here’s how it works:

1. **During Training:** Dropout is applied at each training step. A fixed percentage of neurons (often between 20-50%) are randomly "dropped" or ignored. This results in a different sub-network being trained each time.
2. **During Testing:** Dropout is not applied. The full network is used, but the weights of the neurons are scaled down by the dropout rate (to account for the fact that during training, fewer neurons were used).

The main benefits of dropout are:

- **Prevents overfitting:** By not relying on any particular neuron, dropout helps the model generalize better on unseen data.
- **Improves robustness:** It forces the network to learn redundant representations, making it less sensitive to specific input features.

A typical dropout rate is around 50%, but it can vary depending on the task and architecture.
Before Dropout
After dropout

Code

Using a **neural network** built with **Keras (TensorFlow)** to classify the **Iris dataset**, which is a famous dataset used for classification tasks in machine learning.

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset
iris = load_iris()

# Convert to pandas DataFrame for better readability
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['species'] = iris.target
iris_df['species'] = iris_df['species'].map({0: 'setosa', 1: 'versicolor',
```

```
2: 'virginica'})

# Split the dataset into features and target
X = iris.data
y = iris.target

# Convert the labels to categorical (one-hot encoding)
y = to_categorical(y, num_classes=3)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Initialize a Sequential model
model = Sequential()

# Add layers to the model
model.add(Dense(10, input_dim=4, activation='relu')) # Input layer with 4
features, first hidden layer with 10 neurons
model.add(Dense(10, activation='relu')) # Second hidden layer with 10
neurons
model.add(Dense(3, activation='softmax')) # Output layer with 3 neurons
 (for 3 classes) and softmax activation

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

# Train the model and capture the training history
history = model.fit(X_train, y_train, epochs=50, batch_size=10,
validation_data=(X_test, y_test), verbose=1)

# Plot accuracy curve
plt.figure(figsize=(12, 6))

# Plot training accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
# Plot loss curve
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# Evaluate the model on the test set
y_pred = model.predict(X_test)

# Convert predictions from one-hot to label encoding
y_pred_labels = y_pred.argmax(axis=1)
y_true_labels = y_test.argmax(axis=1)

# Evaluate accuracy and print classification report
print("Accuracy:", accuracy_score(y_true_labels, y_pred_labels))
print("Classification Report:\n", classification_report(y_true_labels,
y_pred_labels))
```

Key Point:

- **Backpropagation** and **weight updates** are carried out automatically by Keras whenever you call `fit()`. You don't need to manually implement the backpropagation algorithm as it is abstracted away by TensorFlow/Keras.
- The `adam` optimizer used in this code combines both gradient descent and backpropagation to adjust the weights in an efficient way, and this process is continuously done during each epoch of training.

```
Accuracy: 0.9777777777777777
Classification Report:
 precision recall f1-score support

  0  1.00  1.00  1.00  19   1  1.00  0.92  0.96  13   2  0.93
1.00  0.96  13

  accuracy  0.98  45   macro  avg  0.98  0.97  0.97  45
weighted avg 0.98 0.98 0.98 45
```

1. Dropout Regularization

Dropout is a regularization technique that randomly sets a fraction of input units to zero at each update during training to prevent overfitting. It helps the model generalize better by preventing

neurons from co-adapting too much.

To add **Dropout** to your model, you can use the `Dropout` layer from Keras:

python
CopyEdit

```
from tensorflow.keras.layers import Dropout

# Add Dropout to the model
model.add(Dense(10, input_dim=4, activation='relu')) # Input layer
with 4 features, first hidden layer with 10 neurons
model.add(Dropout(0.2)) # Dropout with a rate of 20%
model.add(Dense(10, activation='relu')) # Second hidden layer with 10
neurons
model.add(Dropout(0.2)) # Dropout with a rate of 20%
model.add(Dense(3, activation='softmax')) # Output layer with 3
neurons (for 3 classes) and softmax activation
```

- In this example, `Dropout(0.2)` means that 20% of the neurons in the layer will be randomly set to zero during training.
- Dropout is usually applied after activation functions in the hidden layers (e.g., after `ReLU`).

2. L1/L2 Regularization

L1 and L2 regularization are techniques to add penalty terms to the loss function to reduce the complexity of the model, thus preventing overfitting. L1 regularization adds the absolute value of the weights to the loss, while L2 regularization adds the squared value of the weights.

- **L1 Regularization**: Can lead to sparse models, where some weights are zero.
- **L2 Regularization**: Penalties large weights but doesn't force them to zero.

You can add **L1/L2 regularization** using the `regularizers` module in

Keras. Here's how to add L2 regularization to your `Dense` layers:

```
from tensorflow.keras import regularizers
# Add L2 Regularization to the model
model.add(Dense(10, input_dim=4, activation='relu',
                kernel_regularizer=regularizers.l2(0.01))) # L2
regularization with lambda=0.01
model.add(Dense(10, activation='relu',
                kernel_regularizer=regularizers.l2(0.01))) # L2
regularization with lambda=0.01
```

```
model.add(Dense(3, activation='softmax'))
```

- `regularizers.l2(0.01)` adds L2 regularization with a **lambda** value of 0.01. You can adjust this value depending on how much regularization you want to apply.

If you wanted to use **L1 regularization**, you would replace `l2` with `l1`:

```
model.add(Dense(10, input_dim=4, activation='relu',
                kernel_regularizer=regularizers.l1(0.01))) # L1
regularization with lambda=0.01
```

Adding Both Dropout and Regularization

You can combine **Dropout** and **L2 regularization** together for a more regularized model:

```
from tensorflow.keras import regularizers
from tensorflow.keras.layers import Dropout

model = Sequential()

# First hidden layer with L2 regularization and Dropout
model.add(Dense(10, input_dim=4, activation='relu',
                kernel_regularizer=regularizers.l2(0.01)))
model.add(Dropout(0.2))

# Second hidden layer with L2 regularization and Dropout
model.add(Dense(10, activation='relu',
                kernel_regularizer=regularizers.l2(0.01)))
model.add(Dropout(0.2))
# Output layer
model.add(Dense(3, activation='softmax'))
```

Summary:

- **Dropout**: Prevents overfitting by randomly "dropping" a percentage of neurons during training.
- **L1/L2 Regularization**: Adds penalties to the weights during training to reduce the complexity of the model and prevent overfitting.
- In your code, **dropout** and **regularization** (L1/L2) are not applied, but you can easily add them by modifying the layers as shown above.