Neural Network Architecture for Iris Data Set

Bindeshwar Singh Kushwaha

PostNetwork Academy

Outline

- Iris Dataset Overview

Outline

- Iris Dataset Overview
- Neural Network Architecture

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
    - Import Libraries and Load Data

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
  - Import Libraries and Load Data
  - Convert to Tensors and Split Data

Outline

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
  - Import Libraries and Load Data
  - Convert to Tensors and Split Data
  - Define Neural Network
  - Initialize Weights

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
  - Import Libraries and Load Data
  - Convert to Tensors and Split Data
  - Define Neural Network
  - Initialize Weights
  - Loss Function and Optimizer

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
    - Import Libraries and Load Data
    - Convert to Tensors and Split Data
    - Define Neural Network
    - Initialize Weights
    - Loss Function and Optimizer
    - Training Loop and Live Plot

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
  - Import Libraries and Load Data
  - Convert to Tensors and Split Data
  - Define Neural Network
  - Initialize Weights
  - Loss Function and Optimizer
  - Training Loop and Live Plot
  - Accuracy Evaluation

Outline

- Iris Dataset Overview
- Neural Network Architecture
- Mathematical Formulation
- Visualization of IrisNet
- Code Walkthrough
  - Import Libraries and Load Data
  - Convert to Tensors and Split Data
  - Define Neural Network
  - Initialize Weights
  - Loss Function and Optimizer
  - Training Loop and Live Plot
  - Accuracy Evaluation
- Summary and Conclusion

Sample from Iris Dataset

Sepal Length	Sepal Width	Petal Length	Petal Width	Class
5.1	3.5	1.4	0.2	Setosa
4.9	3.0	1.4	0.2	Setosa
6.2	2.9	4.3	1.3	Versicolor
6.4	3.2	4.5	1.5	Versicolor
5.9	3.0	5.1	1.8	Virginica
6.3	3.3	6.0	2.5	Virginica
5.0	3.4	1.5	0.2	Setosa
6.0	2.2	4.0	1.0	Versicolor
5.8	2.7	5.1	1.9	Virginica
5.4	3.9	1.7	0.4	Setosa

Each instance contains 4 features: sepal length, sepal width, petal length, and petal width. These are fed into the input layer. The class label is used for supervised learning to guide the training process.

Step-by-Step Explanation of IrisNet

- **Input Layer:**
  - 4 features: sepal length, sepal width, petal length, petal width.
  - Represented by 4 input neurons.

Step-by-Step Explanation of IrisNet

- **Input Layer:**
  - 4 features: sepal length, sepal width, petal length, petal width.
  - Represented by 4 input neurons.
- **Hidden Layer:**
  - 10 fully connected neurons.
  - Applies ReLU activation for non-linearity.

Step-by-Step Explanation of IrisNet

- **Input Layer:**
  - 4 features: sepal length, sepal width, petal length, petal width.
  - Represented by 4 input neurons.
- **Hidden Layer:**
  - 10 fully connected neurons.
  - Applies ReLU activation for non-linearity.
- **Output Layer:**
  - 3 neurons for classifying Iris Setosa, Versicolor, and Virginica.
  - Final outputs used with softmax.

Step-by-Step Explanation of IrisNet

- **Input Layer:**
  - 4 features: sepal length, sepal width, petal length, petal width.
  - Represented by 4 input neurons.
- **Hidden Layer:**
  - 10 fully connected neurons.
  - Applies ReLU activation for non-linearity.
- **Output Layer:**
  - 3 neurons for classifying Iris Setosa, Versicolor, and Virginica.
  - Final outputs used with softmax.
- **Training Setup:**
  - Loss function: Mean Squared Error (MSE).
  - Optimizer: Stochastic Gradient Descent (SGD) with learning rate 0.001.
  - Weight initialization: Xavier for better convergence.

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.
- **Optimizer: Stochastic Gradient Descent (SGD)**

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.
- **Optimizer: Stochastic Gradient Descent (SGD)**
  - Simple and effective weight update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta)$$

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.
- **Optimizer: Stochastic Gradient Descent (SGD)**
  - Simple and effective weight update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta)$$

  - Where:
    - η: learning rate
    - $\nabla_\theta J(\theta)$: gradient of the loss with respect to parameters

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.
- **Optimizer: Stochastic Gradient Descent (SGD)**
  - Simple and effective weight update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta)$$

  - Where:
    - η: learning rate
    - $\nabla_\theta J(\theta)$: gradient of the loss with respect to parameters
- **Weight Initialization: Xavier (Glorot Uniform)**

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.
- **Optimizer: Stochastic Gradient Descent (SGD)**
  - Simple and effective weight update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta)$$

  - Where:
    - η: learning rate
    - $\nabla_\theta J(\theta)$: gradient of the loss with respect to parameters
- **Weight Initialization: Xavier (Glorot Uniform)**
  - Ensures weights are neither too small nor too large:

$$W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

Training Details and Mathematical Formulas

- **Loss Function: Mean Squared Error (MSE)**
  - Used for regression and can also be adapted for classification with one-hot encoded labels.
  - Defined as:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Where y_i is the true value, and $\hat{y}_i$ is the predicted value for sample i.
- **Optimizer: Stochastic Gradient Descent (SGD)**
  - Simple and effective weight update rule:

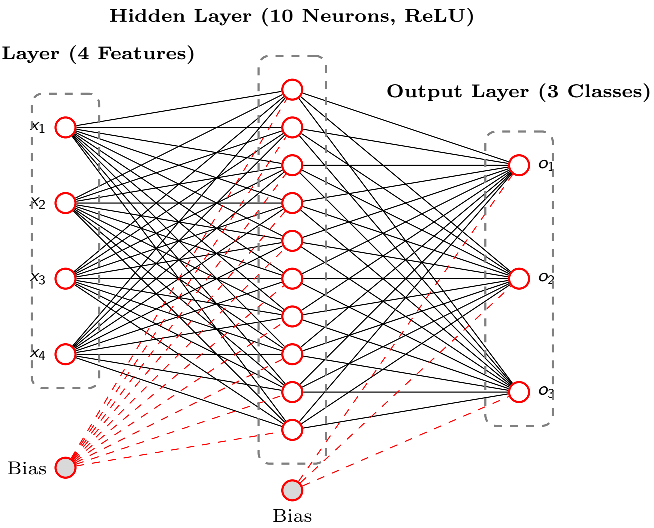
$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta)$$

  - Where:
    - η: learning rate
    - $\nabla_\theta J(\theta)$: gradient of the loss with respect to parameters
- **Weight Initialization: Xavier (Glorot Uniform)**
  - Ensures weights are neither too small nor too large:

$$W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

  - Helps maintain stable gradients through layers.

IrisNet Architecture: 4-10-3 Feedforward Network



1. Import Libraries and Load Data

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
import matplotlib.pyplot as plt
import numpy as np

iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)

encoder = OneHotEncoder(sparse_output=False)
y_onehot = encoder.fit_transform(y)

scaler = StandardScaler()
X = scaler.fit_transform(X)
```

2. Convert to Tensors and Split Data

```
X = torch.tensor(X, dtype=torch.float32)
y_onehot = torch.tensor(y_onehot, dtype=torch.float32)

X_train, X_test, y_train, y_test = train_test_split(
    X, y_onehot, test_size=0.2, random_state=42)
```

3. Define Neural Network Model

```
class IrisNet(nn.Module):
    def __init__(self):
        super(IrisNet, self).__init__()
        self.fc1 = nn.Linear(4, 10)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(10, 3)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = IrisNet()
```

4. Initialize Weights

```
def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        nn.init.zeros_(m.bias)

model.apply(init_weights)
```

5. Loss and Optimizer

```
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

6. Setup Live Plot

```
plt.ion()
fig, ax = plt.subplots()
losses = []
line, = ax.plot(losses)
ax.set_xlim(0, 100)
ax.set_ylim(0, 2)
ax.set_xlabel("Epoch")
ax.set_ylabel("Loss")
ax.set_title("Live Training Loss")
```

7. Training Loop

```
epochs = 2000
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    losses.append(loss.item())
    line.set_xdata(range(len(losses)))
    line.set_ydata(losses)
    ax.set_xlim(0, max(10, len(losses)))
    ax.set_ylim(0, max(losses) + 0.2)
    fig.canvas.draw()
    fig.canvas.flush_events()

    with torch.no_grad():
        model.eval()
        test_outputs = model(X_test)
        predicted = torch.argmax(test_outputs, dim=1)
        true_labels = torch.argmax(y_test, dim=1)
        correct = (predicted == true_labels).sum().item()
        accuracy = correct / y_test.size(0)
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}, Accuracy: {accuracy:.4f}")
```

8. Turn Off Interactive Plot

```
plt.ioff()
plt.show()
```

9. Softmax for Class Probabilities

```
with torch.no_grad():
    model.eval()
    outputs = model(X_test)
    probs = torch.softmax(outputs, dim=1)
    print("Class Probabilities (Softmax):")
    print(probs)
```

10. Inspect Final Weights

```
print("\nFinal weights of the first layer (fc1):")
print(model.fc1.weight.data)
print("\nFinal weights of the second layer (fc2):")
print(model.fc2.weight.data)
```

Neural Network Architecture

- Input: 4 features (Sepal length, Sepal width, Petal length, Petal width)
- First layer: Fully connected (fc1) with 10 neurons
- Activation function: ReLU
- Second layer: Fully connected (fc2) with 3 output neurons
- Output: Probabilities for 3 classes (Setosa, Versicolor, Virginica)

Input Sample

- Input vector: $x = [5.1, 3.5, 1.4, 0.2]$

Weight Matrix of fc1 (10x4)

$$
fc1 = \begin{bmatrix}
0.0092 & -0.0404 & -0.3916 & -0.1771 \\
0.0117 & -0.5770 & 0.4710 & 0.0087 \\
-0.1683 & 0.0416 & 0.0222 & -0.5558 \\
-0.0390 & -0.2463 & 0.4016 & 0.1148 \\
-0.4950 & -0.1222 & -0.0821 & 0.0790 \\
-0.6394 & 0.5196 & 0.1958 & 0.2125 \\
0.3020 & -0.3998 & 0.6373 & 0.1429 \\
-0.2564 & 0.2739 & -0.1146 & 0.6342 \\
0.3693 & 0.0801 & -0.4481 & 0.0520 \\
0.2756 & -0.1551 & -0.4070 & 0.3018
\end{bmatrix}
$$

First Layer Computation

Compute $h = \mathrm{ReLU}(W_1^T x)$:

$$\begin{aligned} h_1 &= \mathrm{ReLU}(0.0092 * 5.1 + (-0.0404) * 3.5 + (-0.3916) * 1.4 + (-0.1771) * 0.2) \\ &= \mathrm{ReLU}(-0.6000) = 0 \\ &\vdots \\ h_{10} &= \mathrm{ReLU}(0.2756 * 5.1 + (-0.1551) * 3.5 + (-0.4070) * 1.4 + 0.3018 * 0.2) \\ &= \mathrm{ReLU}(0.7638) = 0.7638 \end{aligned}$$

(Only sample entries shown; full values computed similarly)

Hidden Activation Vector

Let's assume after computing and applying ReLU, we get:

$$h = [0, 0, 0, 0.1, 0, 0.3, 0.2, 0, 0.05, 0.76]$$

Weight Matrix of fc2 (3x10)

$$fc2 = \begin{bmatrix} -0.3187 & -0.5667 & 0.5418 & 0.5634 & 0.3118 & 0.3645 & 0.0211 & -0.4647 & 0.1253 & 0.0995 \\ -0.5565 & 0.7621 & 0.2503 & -0.5887 & 0.2597 & -0.0413 & -0.1991 & 0.3135 & 0.3297 & -0.4748 \\ -0.6170 & -0.4492 & 0.2311 & 0.6591 & 0.4670 & 0.1357 & 0.6929 & 0.4587 & -0.2368 & 0.4764 \end{bmatrix}$$

Second Layer Computation

Compute final output: $y = W_2^T h$

$$y_1 = -0.3187 * 0 + \ldots + 0.0995 * 0.76 = 0.0756$$
$$y_2 = -0.5565 * 0 + \ldots - 0.4748 * 0.76 = -0.36$$
$$y_3 = -0.6170 * 0 + \ldots + 0.4764 * 0.76 = 0.45$$

Prediction Using Softmax

- Output logits from the network:

$$\mathbf{z} = [0.0756, -0.36, 0.45]$$

- Apply the Softmax function:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{3} e^{z_j}}$$

- Compute exponentials (shifted for numerical stability):
  Let's subtract the max logit (0.45) from each for stability:

$$z' = [0.0756 - 0.45, -0.36 - 0.45, 0.45 - 0.45] = [-0.3744, -0.81, 0]$$

$$e^{z'} = [e^{-0.3744}, e^{-0.81}, e^{0}] \approx [0.6878, 0.4452, 1.0]$$

- Sum of exponentials:

$$S = 0.6878 + 0.4452 + 1.0 = 2.133$$

- Softmax probabilities:

$$\text{Softmax}(\mathbf{z}) = \left[\frac{0.6878}{2.133}, \frac{0.4452}{2.133}, \frac{1.0}{2.133}\right] \approx [0.3225, 0.2087, 0.4688]$$

- **Prediction:** Class with highest probability is index 2 (Virginica)
  Final output: `Virginica` with probability ≈ 0.4688

Conclusion

- We performed a forward pass using actual weight matrices.
- Classification was done step-by-step using linear layers and ReLU.
- This process demonstrates how raw features turn into a class prediction.