Neural Network Exercise

Objective

Build and train a neural network with one hidden layer using PyTorch to classify a dataset with multiple classes. Implement the network without using high-level abstractions like `torch.nn` or `torch.optim`. Visualize the cost reduction over iterations to ensure that gradient descent is working effectively.

Dataset

- Generate a dataset using the `make_blobs` function from `sklearn.datasets`.
- The dataset should have 500 samples, 4 classes, and 2 features.
- Use a random state of 42 for reproducibility.
- Separate out 100 samples for testing.

Neural Network Specifications

- The network should have one hidden layer.
- The input layer should have 2 neurons (corresponding to the 2 features of the dataset).
- The hidden layer should have 5 neurons.
- The output layer should have 4 neurons (corresponding to the 4 classes).
- Use the sigmoid activation function for the hidden layer.
- Use the softmax activation function for the output layer.
- Initialize the weights randomly from a normal distribution.
- Initialize the biases to zeros.

Training Specifications

- Use the negative log likelihood (logarithmic loss) as the cost function.
- Implement gradient descent to update the weights and biases.
- Do not use `torch.optim` or any other optimization library.
- Use a learning rate of 0.01.
- Train the network for 1000 epochs.
- Print the cost every 100 epochs.

Visualization

- Plot the cost over the epochs to visualize the training progress.

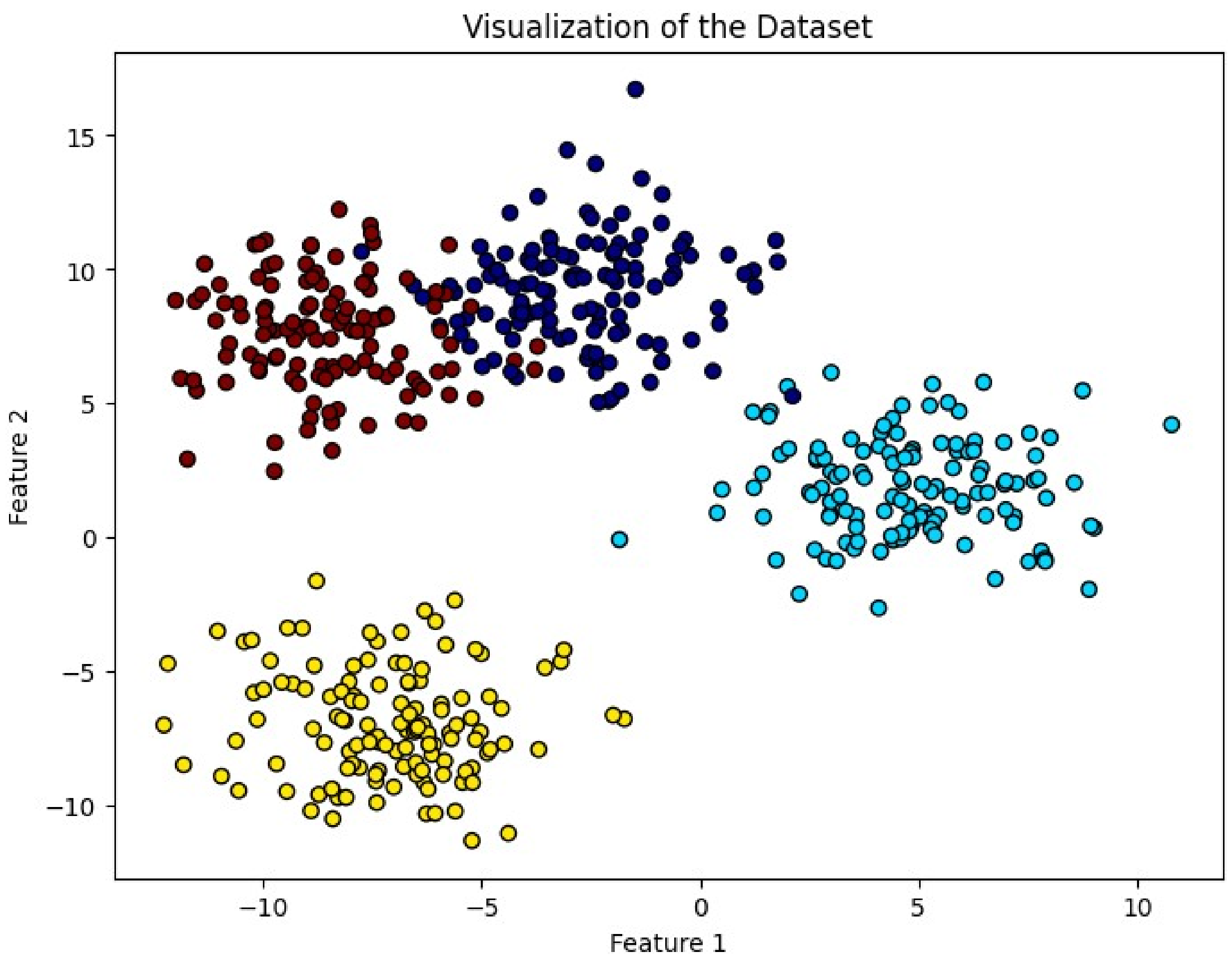
Evaluation

- After training, compute and print the accuracy of the model on the training and testing datasets.

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
import numpy as np

# Generate a 2D dataset with 4 centers
X, y = make_blobs(n_samples=500, centers=4, n_features=2,
cluster_std=2.0, random_state=42)
# separate out 20% of the data for testing
test_size = 0.2
test_size = int(test_size * X.shape[0])
X_train, X_test = X[:-test_size].copy(), X[-test_size:].copy()
y_train, y_test = y[:-test_size].copy(), y[-test_size:].copy()

# Visualize the dataset
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.jet, edgecolors='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Visualization of the Dataset')
plt.show()
```



```python
import torch

X = torch.tensor(X_train, dtype=torch.float32)
y = torch.tensor(y_train, dtype=torch.long)

# Initialize parameters
input_size = 2 # two features (x,y)
hidden_size = 5
output_size = 4 # four outputs (which group)
learning_rate = 0.01
epochs = 1000

# W1, b1 are for going from the input layer to the hidden layer
W1 = torch.randn(input_size, hidden_size, requires_grad=True)
b1 = torch.zeros(hidden_size, requires_grad=True)

# W2, b2 are for going from the hidden layer to the output layer
W2 = torch.randn(hidden_size, output_size, requires_grad=True)
b2 = torch.zeros(output_size, requires_grad=True)

# Convert labels to one-hot encoding
Y = torch.zeros(y.size(0), output_size)
Y[torch.arange(y.size(0)), y] = 1

# Training the model
costs = []
for epoch in range(epochs):
    # Forward pass
    Z1 = X.mm(W1) + b1 # hidden layer
    Z2 = Z1.mm(W2) + b2 # output layer
    A = torch.softmax(Z2, dim=1)

    # Compute cost (negative log likelihood loss)
    log_likelihood = -torch.sum(Y * torch.log(A)) / y.size(0)
    cost = log_likelihood
    costs.append(cost.item())

    # Backward pass
    cost.backward()

    # Update parameters
    with torch.no_grad():
        W1 -= learning_rate * W1.grad
        b1 -= learning_rate * b1.grad
        W2 -= learning_rate * W2.grad
        b2 -= learning_rate * b2.grad

        W1.grad.zero_()
        b1.grad.zero_()
        W2.grad.zero_()
```

```
        b2.grad.zero_()

    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Cost: {cost.item()}')

# Plotting the cost, uncomment the following lines
# plt.plot(costs)
# plt.xlabel('Epochs')
# plt.ylabel('Cost')
# plt.title('Cost Reduction Over Iterations')
# plt.show()


# Evaluate accuracy on the training set
with torch.no_grad():
    # Forward pass for test data
    Z1_train = X.mm(W1) + b1
    Z2_train = Z1.mm(W2) + b2
    A_train = torch.softmax(Z2_train, dim=1)

    # Get predictions
    predictions_train = torch.argmax(A_train, dim=1)

    # Calculate accuracy
    accuracy_train = torch.mean((predictions_train == y).float()) *
100
    print(f'Accuracy on the training set: {accuracy_train.item()}%')

X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.long)

# Evaluate on the test set
with torch.no_grad():
    Z1_test = X_test.mm(W1) + b1
    Z2_test = Z1_test.mm(W2) + b2
    A_test = torch.softmax(Z2_test, dim=1)

    predictions_test = torch.argmax(A_test, dim=1)

    accuracy_test = torch.mean((predictions_test == y_test).float()) *
100
    print(f'Accuracy on the test set: {accuracy_test.item()}%')
```

```
Epoch 0, Cost: nan
Epoch 100, Cost: nan
Epoch 200, Cost: nan
Epoch 300, Cost: nan
Epoch 400, Cost: nan
Epoch 500, Cost: nan
Epoch 600, Cost: nan
```

```
Epoch 700, Cost: nan
Epoch 800, Cost: nan
Epoch 900, Cost: nan
Accuracy on the training set: 24.75%
Accuracy on the test set: 26.0%

/tmp/ipykernel_4508/1360135762.py:78: UserWarning: To copy construct
from a tensor, it is recommended to use sourceTensor.clone().detach()
or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  X_test = torch.tensor(X_test, dtype=torch.float32)
/tmp/ipykernel_4508/1360135762.py:79: UserWarning: To copy construct
from a tensor, it is recommended to use sourceTensor.clone().detach()
or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  y_test = torch.tensor(y_test, dtype=torch.long)
```