CNSCC.361 Artificial Intelligence

Lab session 10: Artificial Neural Networks

Aim of the session:

The aim of this lab session is to implement an artificial neural network in PyTorch.

1. Introduction

Artificial Neural Networks (ANNs) are computational models inspired by biological neural networks, designed to recognize patterns and solve complex tasks through layers of interconnected nodes. They excel in tasks like image recognition, natural language processing, and regression by learning hierarchical representations of data. A basic ANN comprises an input layer, hidden layers (for feature extraction), and an output layer, with activation functions introducing non-linearity to enable complex mappings.

PyTorch is an open-source machine learning framework renowned for its flexibility and dynamic computation graph, which allows real-time adjustments during model development. Developed by Meta (formerly Facebook), it provides intuitive APIs for tensor operations, automatic differentiation, and GPU acceleration, making it a preferred choice for researchers and developers. PyTorch's modular design simplifies building and training ANNs, while its integration with libraries like TorchVision streamlines tasks such as dataset loading and preprocessing. Together, PyTorch and ANNs empower rapid prototyping and deployment of deep learning solutions.

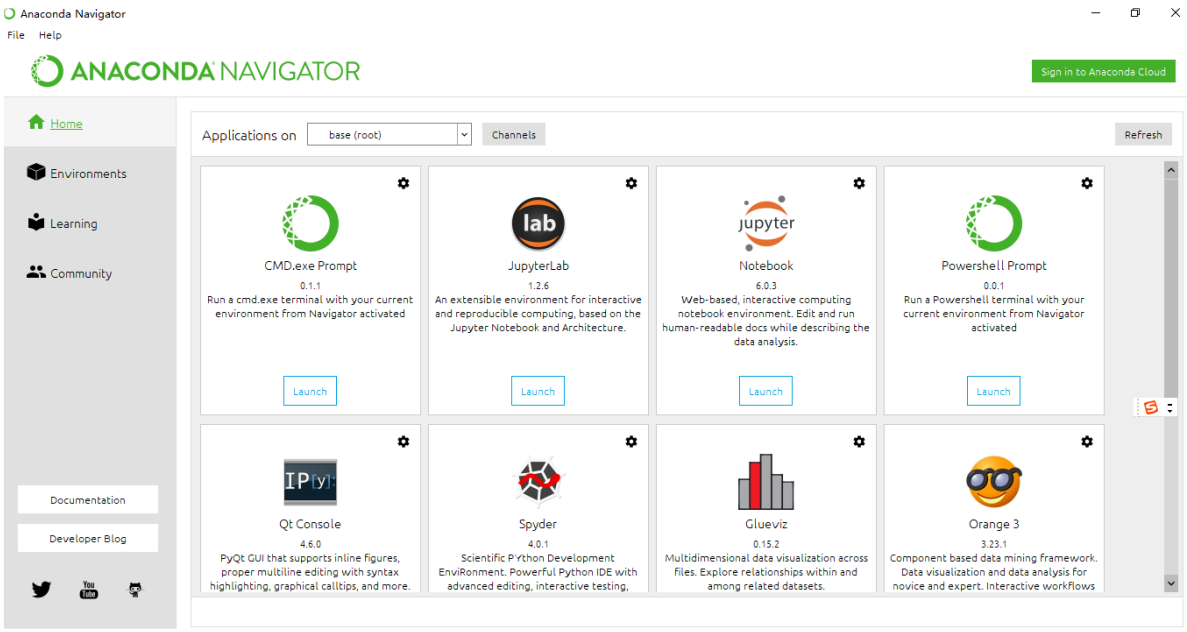
In this session, we will build a basic ANN for image classification on the FashionMNIST dataset in PyTorch.

2. PyTorch Installation

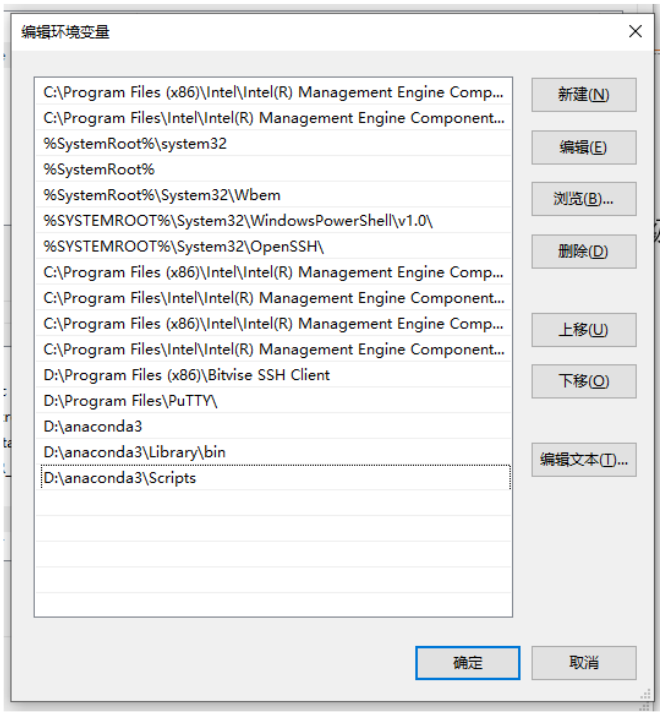
Before implementing the ANN, Anaconda and PyTorch need to be installed first.

Anaconda is a widely used open-source distribution platform for Python and R programming languages, designed to simplify package management and environment configuration for data science, machine learning, and scientific computing. It includes tools like "Conda", a powerful package manager that handles dependencies and isolates project environments, ensuring reproducibility across workflows. Anaconda plays a critical role in PyTorch development by streamlining its installation and dependency resolution.

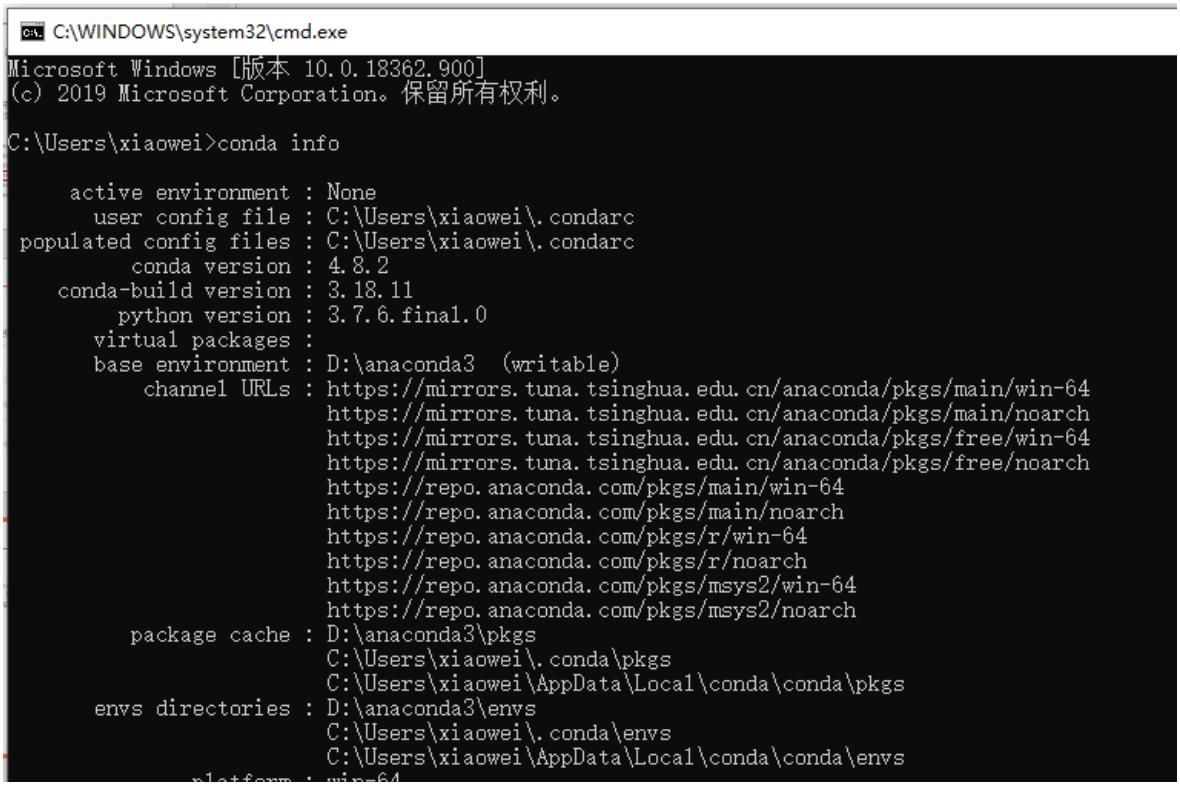
Anaconda can be found from https://www.anaconda.com/ or https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/ (preferred).



After the installation, we need to add related folders to 'Path' environment variable by going to Settings (of Windows) -> Advanced system settings -> Environment variables -> 'Path' (in System variables) -> Edit, and adding three folders (assuming that Anaconda is installed to D:\anaconda3).

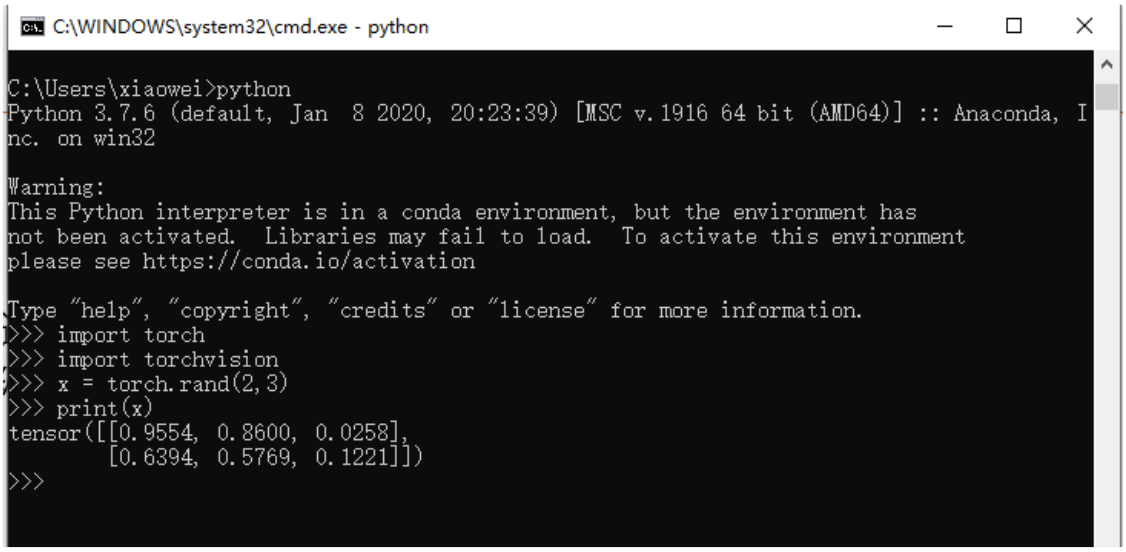


Try **conda info** in Windows Commands (**cmd**) to verify successful installation.



PyTorch can be installed in Windows Commands by typing a line as follows.
conda install pytorch torchvision cpuonly -c pytorch

Try the following code after typing **python** in Windows Commands to verify successful installation.
import torch
import torchvision
x = torch.rand(2,3)
print(x)



Open Jupyter Notebook in Anoconda, and then we can create a new python file for implementing an ANN in PyTorch.

3. Dataset

We will use the FashionMNIST dataset, which is a popular benchmark dataset in machine learning, designed as a more challenging alternative to the classic MNIST handwritten digit dataset. It consists of 60,000 training images and 10,000 test images, each representing grayscale clothing items from 10 distinct categories, such as T-shirts, trousers, dresses, and sneakers. All images are standardized to a resolution of 28x28 pixels, mirroring the format of MNIST but focusing on fashion products.

4. ANN Implementation

In this section, we explain how the ANN can be implemented in PyTorch.

1. Import Required Libraries

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

Explanations:

- torch: Core PyTorch library.
- torch.nn: Neural network modules (e.g., layers, loss functions).
- torch.optim: Optimization algorithms (e.g., Adam).
- torchvision.datasets: Prebuilt datasets (e.g., FashionMNIST).
- torch.utils.data.DataLoader: Batch data loading utility.

2. Set Hyperparameters

```
# Hyperparameters
input_size = 784    # 28x28 flattened pixels
hidden_size = 256   # Neurons in hidden layer
num_classes = 10    # 10 clothing categories
num_epochs = 10     # Training cycles
batch_size = 64     # Samples per batch
learning_rate = 0.001
```

Key Parameters to Adjust Later:

- hidden_size: Controls model capacity.
- learning_rate: Affects convergence speed.
- num_epochs: Determines training duration.

3. Prepare the Dataset

```
# Data preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),                    # Convert images to tensors
    transforms.Normalize((0.5,), (0.5,))      # Normalize to [-1, 1]
])

# Load datasets
train_dataset = datasets.FashionMNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)
test_dataset = datasets.FashionMNIST(
    root='./data',
    train=False,
    transform=transform
)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Notes:

- transforms.Normalize standardizes pixel values for faster convergence.
- shuffle=True prevents order bias during training.
- Dataset will auto-download to ./data on first run.

4. Define the Neural Network

```python
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)  # Input layer
        self.relu = nn.ReLU()                          # Activation function
        self.fc2 = nn.Linear(hidden_size, num_classes) # Output layer

    def forward(self, x):
        x = x.view(-1, 28*28)       # Flatten 28x28 image to 1D vector
        out = self.fc1(x)           # First fully connected layer
        out = self.relu(out)        # Non-linear activation
        out = self.fc2(out)         # Output layer (no softmax needed)
        return out
```

Architecture Details:

- **Input Layer**: 784 neurons (flattened image).
- **Hidden Layer**: 256 neurons with ReLU activation.
- **Output Layer**: 10 neurons (logits for class probabilities).

5. Initialize Model, Loss Function, and Optimizer

```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
criterion = nn.CrossEntropyLoss()  # Combines softmax and negative log likelihood
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

Note:

- `CrossEntropyLoss` automatically applies softmax, so no explicit activation is needed in the output layer.
- `Adam` optimizer adapts learning rates for each parameter.

6. Training Loop

```python
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Move data to GPU (if available)
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        optimizer.zero_grad()  # Clear previous gradients
        loss.backward()        # Compute gradients
        optimizer.step()       # Update weights

        # Print training progress
        if (i+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{total_step}], Loss: {loss.item():.4f}')
```

Key Observations:

- Loss should decrease over epochs.
- GPU utilization (if available) significantly speeds up training.

```
Epoch [8/10], Step [400/938], Loss: 0.2066
Epoch [8/10], Step [500/938], Loss: 0.2055
Epoch [8/10], Step [600/938], Loss: 0.3337
Epoch [8/10], Step [700/938], Loss: 0.3444
Epoch [8/10], Step [800/938], Loss: 0.2833
Epoch [8/10], Step [900/938], Loss: 0.3274
Epoch [9/10], Step [100/938], Loss: 0.1987
Epoch [9/10], Step [200/938], Loss: 0.2272
Epoch [9/10], Step [300/938], Loss: 0.1161
Epoch [9/10], Step [400/938], Loss: 0.3064
Epoch [9/10], Step [500/938], Loss: 0.1591
Epoch [9/10], Step [600/938], Loss: 0.1947
Epoch [9/10], Step [700/938], Loss: 0.2639
Epoch [9/10], Step [800/938], Loss: 0.2687
Epoch [9/10], Step [900/938], Loss: 0.3208
Epoch [10/10], Step [100/938], Loss: 0.2783
Epoch [10/10], Step [200/938], Loss: 0.2707
Epoch [10/10], Step [300/938], Loss: 0.3541
Epoch [10/10], Step [400/938], Loss: 0.3969
Epoch [10/10], Step [500/938], Loss: 0.1731
Epoch [10/10], Step [600/938], Loss: 0.2231
Epoch [10/10], Step [700/938], Loss: 0.3760
Epoch [10/10], Step [800/938], Loss: 0.2787
Epoch [10/10], Step [900/938], Loss: 0.2184
```

7. Model Evaluation

```
model.eval()  # Set model to evaluation mode
with torch.no_grad():  # Disable gradient computation
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)  # Get class predictions
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

```
Test Accuracy: 88.21%
```

Expected Result:

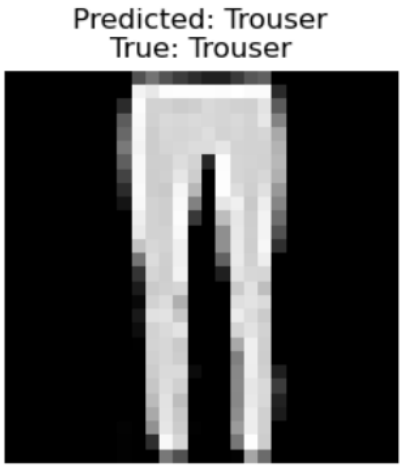
- Baseline accuracy: ~85% on test set.

8. Random Prediction

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

# Define class names for FashionMNIST
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Switch to evaluation mode
model.eval()

# Randomly select a test sample
random_index = np.random.randint(0, len(test_dataset))
sample_image, sample_label = test_dataset[random_index]

# Add batch dimension and move to device
image_tensor = sample_image.unsqueeze(0).to(device)

# Make prediction
with torch.no_grad():
    output = model(image_tensor)
    _, predicted = torch.max(output.data, 1)
    predicted_class = predicted.cpu().item()
```

```python
# Convert tensor to numpy array for visualization
image_np = sample_image.numpy().squeeze()

# Inverse normalization for visualization
image_np = (image_np * 0.5) + 0.5  # Scale back to [0,1]

# Create plot
plt.figure(figsize=(6, 3))
plt.imshow(image_np, cmap='gray')
plt.title(f"Predicted: {class_names[predicted_class]}\nTrue: {class_names[sample_label]}")
plt.axis('off')

# Add prediction confidence
probabilities = torch.nn.functional.softmax(output, dim=1).cpu().numpy()[0]
confidence = np.max(probabilities) * 100

plt.show()

print(f"\nSample Details:")
print(f"- Index:      {random_index}")
print(f"- True Label: {sample_label} ({class_names[sample_label]})")
print(f"- Prediction: {predicted_class} ({class_names[predicted_class]})")
print(f"- Confidence: {confidence:.2f}%")
```

Expected Result:




```
Sample Details:
- Index:      5175
- True Label: 1 (Trouser)
- Prediction: 1 (Trouser)
- Confidence: 100.00%
```


```
Sample Details:
- Index:      9034
- True Label: 9 (Ankle boot)
- Prediction: 9 (Ankle boot)
- Confidence: 100.00%
```

Exercise 1:

Adjust `hidden_size` to 128 or 512. How does accuracy change?

Exercise 2:

Replace `ReLU` with `Sigmoid` or `LeakyReLU`. Observe performance differences.

Exercise 3:

Increase `num_epochs` to 20. Does the model overfit?

Exercise 4:

Replace `Adam` with `SGD` and compare convergence speed.

Exercise 5:

Find a way to modify this architecture to achieve higher accuracy.