PyTorch Notes for ProgSD Exam

Contents

Chapter 1 PyTorch Notes for Exam Preparation

1.1 Setup and Installation

- PyTorch is an open-source machine learning library commonly used for neural networks, deep learning, and tensor operations.
- Install PyTorch using `pip` or via the official website instructions.
- Verify installation by performing a simple tensor operation.
- Base imports for building a simple fully connected neural network:
  - `torch`: For tensor operations and core PyTorch functionalities.
  - `torch.nn`: For building neural networks.
  - `torch.optim`: For optimization algorithms.

Installation Example:

```
# Install PyTorch and torchvision
pip install torch torchvision
```

Verify Installation:

```python
import torch # Core PyTorch library

# Generate a random tensor
tensor = torch.rand(3, 3) # 3x3 random tensor
print("Tensor created using PyTorch:")
print(tensor)
```

Hypothetical Output:

```
Tensor created using PyTorch:
tensor([[0.1234, 0.5678, 0.9101],
        [0.1122, 0.3344, 0.5566],
        [0.7788, 0.9999, 0.0001]])
```

1.2 Building a Simple Neural Network

- Neural networks in PyTorch are created using the `torch.nn` module.
- Fully connected (dense) layers are represented by `nn.Linear`.
- Define the forward pass using the `forward()` method, specifying how data flows through the layers.
- Activation functions like `ReLU` (Rectified Linear Unit) and `log_softmax` add non-linearity and normalize outputs.

Required Imports

Add the following imports to the top of your Python file:

```
import torch # Core PyTorch library for tensors and
    computations
import torch.nn as nn # Provides modules for creating
    neural networks
import torch.nn.functional as F # Provides functions for
    activation and other operations
```

Code Example: Simple Fully Connected Neural Network

```
# Define a simple neural network by inheriting from
    nn.Module
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()

        # Fully connected layers: input (784), hidden (128,
            64), output (10)
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        # Forward pass with ReLU activations and
            log-softmax for output
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.log_softmax(self.fc3(x), dim=1)
        return x

# Instantiate the model and print its architecture
model = SimpleNN()
print(model)
```

Hypothetical Output:

```
SimpleNN(
  (fc1): Linear(in_features=784, out_features=128,
      bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
)
```

1.3 Training a Neural Network

- Steps to train a neural network:
  - Define a loss function using `torch.nn`.
  - Use an optimizer like `torch.optim.SGD`.
  - Loop through the dataset for multiple epochs:
    - Forward pass: Calculate predictions.
    - Backward pass: Compute gradients.
    - Update parameters using the optimizer.
- Common loss functions:
  - `nn.MSELoss()`: Mean Squared Error, used for regression tasks.
  - `nn.CrossEntropyLoss()`: Cross-Entropy Loss, commonly used for classification tasks.
  - `nn.BCELoss()`: Binary Cross-Entropy Loss, used for binary classification.

Required Imports: Place These at the Top of Your File

```python
import torch.optim as optim # For optimizers like SGD
import torch.nn as nn # For defining loss functions
```

Example: Training a Simple Neural Network with MSELoss

```python
# Define loss function and optimizer
criterion = nn.MSELoss() # Mean Squared Error Loss
optimizer = optim.SGD(model.parameters(), lr=0.01) #
    Stochastic Gradient Descent

# Sample data
inputs = torch.rand(1, 784) # 1 sample, 784 input features
labels = torch.rand(1, 10) # 1 sample, 10 output targets

# Training loop
for epoch in range(5): # 5 epochs
    optimizer.zero_grad() # Reset gradients
    outputs = model(inputs) # Forward pass
    loss = criterion(outputs, labels) # Compute loss
    loss.backward() # Backpropagation
    optimizer.step() # Update weights
    print(f"Epoch [{epoch + 1}/5], Loss: {loss.item():.4f}")
```

Hypothetical Output:

```
Epoch [1/5], Loss: 1.2453
Epoch [2/5], Loss: 0.8542
Epoch [3/5], Loss: 0.5678
Epoch [4/5], Loss: 0.3456
Epoch [5/5], Loss: 0.1234
```

Example: Training with Cross-Entropy Loss

```
# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss() # Loss for classification
optimizer = optim.SGD(model.parameters(), lr=0.01) #
    Stochastic Gradient Descent

# Sample data (inputs and labels)
inputs = torch.rand(1, 784) # Random input tensor (batch
    size 1, 784 features)
labels = torch.tensor([3]) # Target label (class index)

# Training loop
for epoch in range(5): # Train for 5 epochs
    optimizer.zero_grad() # Clear gradients
    outputs = model(inputs) # Forward pass
    loss = criterion(outputs, labels) # Compute loss
    loss.backward() # Backpropagation
    optimizer.step() # Update weights
    print(f"Epoch [{epoch + 1}/5], Loss: {loss.item():.4f}")
```

Hypothetical Output:

```
Cross-Entropy Loss: 1.0986
```

1.4 Advanced Topics in PyTorch

1.4.1 Working with Pre-trained Models (Transfer Learning)

- PyTorch provides pre-trained models via `torchvision.models`.
- Transfer learning allows leveraging pre-trained models for new tasks by fine-tuning specific layers.

Required Imports: Place These at the Top of Your File

```
from torchvision import models # For pre-trained models
import torch.nn as nn # For modifying model architecture
```

Example: Transfer Learning with ResNet

```
# Load a pre-trained ResNet model
resnet = models.resnet18(pretrained=True)

# Freeze all layers
for param in resnet.parameters():
    param.requires_grad = False

# Replace the last layer for binary classification
num_features = resnet.fc.in_features
resnet.fc = nn.Linear(num_features, 2)

# Example input tensor
input_tensor = torch.rand(1, 3, 224, 224) # 1 image, 3
    channels, 224x224
output = resnet(input_tensor)
print("Output of the pre-trained ResNet model:")
print(output)
```

Hypothetical Output:

```
Output of the pre-trained ResNet model:
tensor([[0.5678, -0.3452]], grad_fn=<AddmmBackward>)
```

1.5 Custom Datasets and DataLoaders

- Custom datasets allow flexible data handling.
- Implement a custom dataset by inheriting from `torch.utils.data.Dataset`.
- Use `torch.utils.data.DataLoader` to create iterable data batches for training and evaluation.

Required Imports: Place These at the Top of Your File

```
from torch.utils.data import Dataset, DataLoader # For
    custom datasets and loaders
```

Example: Creating a Custom Dataset

```
# Custom dataset class
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]

# Example usage
data = torch.rand(100, 3) # 100 samples, 3 features each
labels = torch.randint(0, 2, (100,)) # Binary labels (0 or
    1)

custom_dataset = CustomDataset(data, labels)
dataloader = DataLoader(custom_dataset, batch_size=10,
    shuffle=True)

for batch_data, batch_labels in dataloader:
    print(batch_data.shape, batch_labels.shape) # Process
        batches
```

Hypothetical Output:

```
torch.Size([10, 3]) torch.Size([10])
torch.Size([10, 3]) torch.Size([10])
...
```

1.6 Using GPUs for Training

- PyTorch supports GPU acceleration via CUDA for faster computations.
- Ensure that the correct PyTorch package with GPU support is installed. Visit the official documentation: https://pytorch.org/get-started/locally/ to select the appropriate installation command.
- Example installation command for CUDA 11.8:

```
pip3 install torch torchvision torchaudio --index-url
    https://download.pytorch.org/whl/cu118
```

- After installation, verify that your system supports GPU with the `torch.cuda.is_available()` function.
- Move models and tensors to the GPU using `.to(device)` or `.cuda()`.
- Place all relevant imports at the top of your Python file:

```
import torch # For CUDA operations
```

Example: Training on a GPU

```
# Check for GPU availability
device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")

# Move the model and data to the GPU
model = SimpleNN(input_size=3, hidden_size=5,
    output_size=2).to(device)
inputs = torch.rand(1, 3).to(device)
labels = torch.rand(1, 2).to(device)

# Training loop on GPU
for epoch in range(5): # 5 epochs
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    print(f"Epoch [{epoch+1}/5], Loss: {loss.item():.4f}")
```

Hypothetical Output:

```
Epoch [1/5], Loss: 0.9452
Epoch [2/5], Loss: 0.6843
...
Epoch [5/5], Loss: 0.1234
```

1.7 Practical Exercises

Required Imports

Add the following imports to the top of your Python file:

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

1.7.1 Training a CNN on CIFAR-10

- Load the CIFAR-10 dataset using `torchvision.datasets`.
- Normalize images using `transforms.Normalize`.
- Define and train a CNN model with multiple layers.

Part 1: Dataset and DataLoader Setup

```python
# Data transformations: Convert to tensor and normalize
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 training dataset
train_dataset = datasets.CIFAR10(
    root='./data', train=True, download=True,
        transform=transform
)
train_loader = DataLoader(train_dataset, batch_size=64,
    shuffle=True)

# Load CIFAR-10 test dataset
test_dataset = datasets.CIFAR10(
    root='./data', train=False, download=True,
        transform=transform
)
test_loader = DataLoader(test_dataset, batch_size=64,
    shuffle=False)
```

Part 2: Define the CNN Model

```
# Define a simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

        # Convolutional Layer 1
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3,
            stride=1, padding=1)

        # Convolutional Layer 2
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3,
            stride=1, padding=1)

        # Pooling layer to reduce dimensions
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully Connected Layer 1 (adjusted for pooling)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10) # Output Layer (10
            classes)

    def forward(self, x):
        x = torch.relu(self.conv1(x)) # Apply ReLU to conv1
        x = self.pool(x) # Apply pooling after conv1
        x = torch.relu(self.conv2(x)) # Apply ReLU to conv2
        x = self.pool(x) # Apply pooling after conv2
        # Flatten the tensor for the fully connected layers
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x)) # Apply ReLU to fc1
        x = self.fc2(x) # Output layer
        return x
```

Part 3: Training Loop

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss() # Use Cross-Entropy Loss
optimizer = optim.SGD(cnn_model.parameters(), lr=0.01,
    momentum=0.9)

# Training loop
epochs = 5 # Adjust to 1 for faster training if needed
cnn_model.train()
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device),
            labels.to(device)

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

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch [{epoch+1}/{epochs}], Loss: {running_loss
        / len(train_loader):.4f}")
```

Part 4: Evaluate the Model

```
# Evaluate the trained model
cnn_model.eval()
correct = 0
total = 0

with torch.no_grad(): # Disable gradient computation for
    evaluation
    for images, labels in test_loader:
        # Move tensors to device and ensure labels are
            int64
        images, labels = images.to(device),
            labels.to(device).long()

        outputs = cnn_model(images) # Forward pass

        # Get predicted class
        _, predicted = torch.max(outputs, 1)

        total += labels.size(0) # Total samples

        # Count correct predictions
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total # Calculate accuracy
print(f"Test Accuracy: {accuracy:.2f}%")
```

Hypothetical Output:

```
Epoch [1/5], Loss: 1.7234
Epoch [2/5], Loss: 1.4321
Epoch [3/5], Loss: 1.2543
Epoch [4/5], Loss: 1.0890
Epoch [5/5], Loss: 0.9654
Test Accuracy: 82.45%
```

1.8 Custom Datasets and DataLoaders

- PyTorch provides flexible tools to create and work with custom datasets.
- Implement a custom dataset by inheriting from `torch.utils.data.Dataset`.
- Use `torch.utils.data.DataLoader` to handle batching and shuffling of data.
- Place the following import at the top of your Python file:

```
from torch.utils.data import Dataset, DataLoader # For
    custom datasets and loaders
```

Example: Creating a Custom Dataset

```
# Custom dataset class
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data # Input data
        self.labels = labels # Corresponding labels

    def __len__(self):
        return len(self.data) # Length of the dataset

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx] #
            Data-label pair at index idx

# Example data
data = torch.rand(100, 3) # 100 samples, each with 3
    features
labels = torch.randint(0, 2, (100,)) # Binary labels (0 or
    1)

# Create dataset and dataloader
custom_dataset = CustomDataset(data, labels)
dataloader = DataLoader(custom_dataset, batch_size=10,
    shuffle=True)

# Process batches
for batch_data, batch_labels in dataloader:
    print(batch_data.shape, batch_labels.shape)
```

Hypothetical Output:

```
torch.Size([10, 3]) torch.Size([10])
torch.Size([10, 3]) torch.Size([10])
...
```

1.9 Saving and Loading Models

- PyTorch allows saving and loading model state dictionaries.
- Always save and load models with the same architecture.
- Place the following import at the top of your Python file:

```
import torch # For saving and loading models
```

1.9.1 Saving a Model

- Save the model's state dictionary using `torch.save()`.
- Specify a file path to store the model's state.

Example: Saving a Model

```
# Save the model's state dictionary
torch.save(model.state_dict(), "simple_nn.pth")
```

1.9.2 Loading a Saved Model

- To load the model, initialize the same architecture and use `torch.load()`.
- Set the model to evaluation mode using `model.eval()` to disable dropout and batch normalization during inference.

Example: Loading a Saved Model

```
# Initialize the same model architecture
loaded_model = SimpleNN(input_size=3, hidden_size=5,
    output_size=2)

# Load the saved state dictionary
loaded_model.load_state_dict(torch.load("simple_nn.pth"))
loaded_model.eval() # Set the model to evaluation mode
```

Hypothetical Output:

```
Model state loaded successfully.
```

1.10 Advanced PyTorch: Transfer Learning with Pre-trained Models

- Transfer learning allows leveraging pre-trained models for new tasks.
- Commonly used pre-trained models include ResNet, VGG, and AlexNet.
- Modify the final layer of the model to suit the new dataset.
- Ensure only the last layer is trainable by freezing the earlier layers.

Important:

- Install the required packages for GPU compatibility:

```
pip3 install torch torchvision torchaudio --index-url
    https://download.pytorch.org/whl/cu118
```

- Redirect to the official PyTorch site for specific GPU driver requirements: `https://pytorch.org`.

Imports: Add at the Top of the File:

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision.models import resnet18, ResNet18_Weights
from torchvision import models
```

Example: Transfer Learning with ResNet on CIFAR-10

Part 1: Data Preparation and Loading

```
# Data transformations and loaders
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        # Normalize data
])

# Load CIFAR-10 training and test datasets
trainset =
    datasets.CIFAR10(root='./data/cifar-10-batches-py',
    train=True, download=True, transform=transform)
testset =
    datasets.CIFAR10(root='./data/cifar-10-batches-py',
    train=False, download=True, transform=transform)

# Create DataLoaders for batching
trainloader = torch.utils.data.DataLoader(trainset,
    batch_size=512, shuffle=True)
testloader = torch.utils.data.DataLoader(testset,
    batch_size=512, shuffle=False)
```

Part 2: Model Setup and Customization

```
# Load the pre-trained ResNet model
weights = ResNet18_Weights.DEFAULT
model = resnet18(weights=weights)

# Modify the final fully connected layer for CIFAR-10 (10
    classes)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 10) # Adjust for 10 output
    classes

# Freeze earlier layers to retain pre-trained features
for param in model.parameters():
    param.requires_grad = False
for param in model.fc.parameters():
    param.requires_grad = True

# Move model to the device (CPU or GPU)
device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")
model = model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.fc.parameters(), lr=0.1,
    momentum=0.9) # Higher learning rate for speed
```

Part 3: Training and Evaluation

```
# Training loop (3 epochs)
model.train()
epochs = 3 # Reduce to 1 for faster training if needed
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        images, labels = images.to(device),
            labels.to(device)

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

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {epoch + 1}/{epochs}, Loss: {running_loss
        / len(trainloader):.4f}")

# Evaluation loop
model.eval()
correct = 0
total = 0
with torch.no_grad(): # No gradient calculation for
    evaluation
    for images, labels in testloader:
        images, labels = images.to(device),
            labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1) # Predicted
            class
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

Hypothetical Output:

```
Epoch 1/3, Loss: 1.9456
Epoch 2/3, Loss: 1.5842
Epoch 3/3, Loss: 1.3721
Test Accuracy: 72.34%
```

1.11 Putting It Together

- This section walks through building, training, and evaluating a CNN on the CIFAR-10 dataset using a pre-trained ResNet-18 model.
- The example uses:
  - **Cross-Entropy Loss** for classification.
  - **SGD Optimizer** for parameter updates.
  - **ResNet-18 Pre-trained Model**.
  - GPU for training, if available.
- The steps include:
  - Importing necessary libraries.
  - Preparing the CIFAR-10 dataset.
  - Modifying the ResNet-18 model for CIFAR-10.
  - Training the model for 3 epochs (can be reduced to 1 for speed).
  - Evaluating the trained model.

Step 1: Import Required Libraries

```python
# Core PyTorch Libraries
import torch
import torch.nn as nn
import torch.optim as optim

# Libraries for Data Loading and Preprocessing
from torchvision import datasets, transforms
from torchvision.models import resnet18, ResNet18_Weights
```

Step 2: Check for GPU and Set Device

```python
# Check if GPU is available; fallback to CPU if not
device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")
print(f"Using device: {device}")
```

Step 3: Prepare the CIFAR-10 Dataset

```
# Define transformations for preprocessing images
transform = transforms.Compose([
    transforms.ToTensor(), # Convert PIL images to PyTorch
        tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        # Normalize with mean and std
])

# Load CIFAR-10 training and test datasets
trainset = datasets.CIFAR10(root='./data', train=True,
    download=True, transform=transform)
testset = datasets.CIFAR10(root='./data', train=False,
    download=True, transform=transform)

# Define DataLoaders for batching
trainloader = torch.utils.data.DataLoader(trainset,
    batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(testset,
    batch_size=64, shuffle=False)
```

Step 4: Load and Modify the Pre-trained ResNet-18 Model

```
# Load the pre-trained ResNet-18 model
weights = ResNet18_Weights.DEFAULT # Load default
    pre-trained weights
model = resnet18(weights=weights)

# Modify the final fully connected layer for CIFAR-10 (10
    classes)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 10) # Replace the output
    layer

# Freeze earlier layers (optional, but speeds up training)
for param in model.parameters():
    param.requires_grad = False
for param in model.fc.parameters():
    param.requires_grad = True

# Move model to the specified device (GPU or CPU)
model = model.to(device)
```

Step 5: Define Loss Function and Optimizer

```python
# Define loss function
criterion = nn.CrossEntropyLoss()

# Define optimizer for the new fully connected layer
optimizer = optim.SGD(model.fc.parameters(), lr=0.01,
    momentum=0.9)
```

Step 6: Train the Model

```python
# Set the number of epochs
epochs = 3 # Reduce to 1 for faster training if needed

# Set model to training mode
model.train()

# Training loop
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        # Move data to the same device as the model
        images, labels = images.to(device),
            labels.to(device)

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

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

        running_loss += loss.item()

    # Print epoch loss
    print(f"Epoch [{epoch + 1}/{epochs}], Loss:
        {running_loss / len(trainloader):.4f}")
```

Step 7: Evaluate the Model

```
# Set model to evaluation mode
model.eval()

# Disable gradient computation during evaluation
correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        # Move data to the same device as the model
        images, labels = images.to(device),
            labels.to(device)

        # Forward pass
        outputs = model(images)

        # Get predictions
        _, predicted = torch.max(outputs, 1)

        # Update correct predictions count
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# Calculate accuracy
accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%")
```

Expected Output:

```
Using device: cuda
Epoch [1/3], Loss: 1.5432
Epoch [2/3], Loss: 1.3421
Epoch [3/3], Loss: 1.1234
Test Accuracy: 75.34%
```

Chapter A Appendix

A.1 Commonly Used PyTorch Commands

- Create tensors:

```
tensor = torch.tensor([1, 2, 3])
random_tensor = torch.rand(3, 3)
zeros_tensor = torch.zeros(3, 3)
```

- Basic tensor operations:

```
tensor1 = torch.tensor([1, 2, 3])
tensor2 = torch.tensor([4, 5, 6])
result_add = tensor1 + tensor2 # Element-wise addition
result_mul = tensor1 * tensor2 # Element-wise
    multiplication
```

- Move tensors to GPU:

```
device = torch.device("cuda" if
    torch.cuda.is_available() else "cpu")
tensor = tensor.to(device)
```

- Save and load models:

```
# Save model
torch.save(model.state_dict(), 'model.pth')

# Load model
model.load_state_dict(torch.load('model.pth'))
```

- Set manual seed for reproducibility:

```
torch.manual_seed(42)
```

A.2 Common Errors and Troubleshooting

- **CUDA Out of Memory Error:**
  - Reduce batch size or use `torch.cuda.empty_cache()` to clear unused memory.
- **Gradient Issues:**
  - Ensure `optimizer.zero_grad()` is called before backpropagation to avoid accumulating gradients.
- **Shape Mismatch:**
  - Use `.view()` or `.reshape()` to adjust tensor shapes.
- **Device Mismatch:**
  - Ensure tensors and models are on the same device (`CPU` or `GPU`).

A.3 Further Resources

- Official PyTorch Documentation: `https://pytorch.org/docs/`
- PyTorch Tutorials: `https://pytorch.org/tutorials/`