🚀 PyTorch Machine Learning Templates

Complete Guide with 8 Real-World Implementations

1 Linear Regression

```
import torch
import torch.nn as nn
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Load data
df = pd.read_csv('data.csv')
X = df.iloc[:, :3].values  # First 3 columns as input
y = df.iloc[:, 3].values.reshape(-1, 1)  # 4th column as output

# Split and scale data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_train = scaler_X.fit_transform(X_train)
X_test = scaler_X.transform(X_test)
y_train = scaler_y.fit_transform(y_train)
y_test = scaler_y.transform(y_test)

# Convert to tensors
X_train = torch.FloatTensor(X_train).to(device)
X_test = torch.FloatTensor(X_test).to(device)
y_train = torch.FloatTensor(y_train).to(device)
y_test = torch.FloatTensor(y_test).to(device)

# Define model
class LinearRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, 1)
```

```
    def forward(self, x):
        return self.linear(x)

model = LinearRegressionModel(3).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Training loop
epochs = 1000
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

# Evaluation
model.eval()
with torch.no_grad():
    predictions = model(X_test)
    test_loss = criterion(predictions, y_test)
    print(f'Test Loss: {test_loss.item():.4f}')
```

📖 Explanation

Linear Regression predicts continuous numerical values by finding a linear relationship between input features and output.

- **Device Setup:** Automatically uses GPU if available for faster computation
- **Data Preprocessing:** StandardScaler normalizes features to have mean=0 and std=1
- **Model Architecture:** Single linear layer that learns weights and bias
- **Loss Function:** MSE (Mean Squared Error) measures prediction accuracy
- **Optimizer:** Adam adjusts weights to minimize loss

🔧 Alternative Models for Regression:

- **Ridge Regression:** Add L2 regularization to prevent overfitting
- **Lasso Regression:** Add L1 regularization for feature selection

- **Polynomial Regression:** Add polynomial features for non-linear relationships
- **ElasticNet:** Combination of L1 and L2 regularization

2 Classification

```
import torch
import torch.nn as nn
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Load data
df = pd.read_csv('data.csv')
X = df.iloc[:, :3].values
y = df.iloc[:, 3].values

# Encode labels if categorical
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
num_classes = len(label_encoder.classes_)

# Split and scale
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to tensors
X_train = torch.FloatTensor(X_train).to(device)
X_test = torch.FloatTensor(X_test).to(device)
y_train = torch.LongTensor(y_train).to(device)
y_test = torch.LongTensor(y_test).to(device)

# Define model
class ClassificationModel(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(ClassificationModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, 64)
        self.fc2 = nn.Linear(64, 32)
```

```
        self.fc3 = nn.Linear(32, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

model = ClassificationModel(3, num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training loop
epochs = 100
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

# Evaluation
model.eval()
with torch.no_grad():
    predictions = model(X_test)
    _, predicted = torch.max(predictions, 1)
    accuracy = (predicted == y_test).sum().item() / len(y_test)
    print(f'Test Accuracy: {accuracy*100:.2f}%')
```

📖 Explanation

Classification predicts discrete categories/classes based on input features.

- **Label Encoding:** Converts categorical outputs to numerical labels
- **Network Architecture:** Multiple hidden layers with ReLU activation
- **Dropout:** Randomly drops neurons during training to prevent overfitting
- **CrossEntropyLoss:** Standard loss for multi-class classification
- **Evaluation:** Uses accuracy metric (correct predictions / total)

🔧 **Alternative Models for Classification:**

- **Logistic Regression:** Simple linear classifier
- **Support Vector Machine (SVM):** Maximum margin classifier
- **Random Forest:** Ensemble of decision trees
- **Gradient Boosting (XGBoost, LightGBM):** Powerful ensemble methods
- **K-Nearest Neighbors (KNN):** Instance-based learning

3 ANN Regression (Deep Neural Network)

```
import torch
import torch.nn as nn
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from torch.utils.data import DataLoader, TensorDataset

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Load data
df = pd.read_csv('data.csv')
X = df.iloc[:, :3].values
y = df.iloc[:, 3].values.reshape(-1, 1)

# Split and scale
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_train = scaler_X.fit_transform(X_train)
X_test = scaler_X.transform(X_test)
y_train = scaler_y.fit_transform(y_train)
y_test = scaler_y.transform(y_test)

# Create DataLoader
train_dataset = TensorDataset(torch.FloatTensor(X_train), torch.FloatTensor(y_train))
```

```
test_dataset = TensorDataset(torch.FloatTensor(X_test), torch.FloatTensor(y_test))
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)

# Define ANN model
class ANNRegression(nn.Module):
    def __init__(self, input_dim):
        super(ANNRegression, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 1)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)
        self.batch_norm1 = nn.BatchNorm1d(128)
        self.batch_norm2 = nn.BatchNorm1d(64)
        self.batch_norm3 = nn.BatchNorm1d(32)

    def forward(self, x):
        x = self.fc1(x)
        x = self.batch_norm1(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.fc2(x)
        x = self.batch_norm2(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.fc3(x)
        x = self.batch_norm3(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.fc4(x)
        return x

model = ANNRegression(3).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=5)

# Training Loop
epochs = 100
for epoch in range(epochs):
    model.train()
    train_loss = 0
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
```

```
    train_loss /= len(train_loader)
    scheduler.step(train_loss)

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {train_loss:.4f}')

# Evaluation
model.eval()
test_loss = 0
with torch.no_grad():
    for X_batch, y_batch in test_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        predictions = model(X_batch)
        loss = criterion(predictions, y_batch)
        test_loss += loss.item()

test_loss /= len(test_loader)
print(f'Test Loss: {test_loss:.4f}')
```

📖 Explanation

ANN (Artificial Neural Network) Regression uses multiple hidden layers to learn complex non-linear patterns in data.

- **Deep Architecture:** 4 layers (128→64→32→1) capture complex patterns
- **Batch Normalization:** Normalizes activations, speeds up training, reduces overfitting
- **Dropout:** Randomly deactivates 30% of neurons to prevent overfitting
- **DataLoader:** Efficiently loads data in batches for training
- **Learning Rate Scheduler:** Reduces learning rate when loss plateaus

🔧 Alternative Deep Learning Architectures:

- **ResNet (Residual Networks):** Skip connections for very deep networks
- **DenseNet:** Each layer connects to all previous layers
- **Wide & Deep:** Combines memorization and generalization
- **Attention Mechanisms:** Focuses on important features

4 CNN for Image Classification

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

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Data transforms
train_transform = transforms.Compose([
    transforms.Resize((244, 244)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((244, 244)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Load datasets
train_dataset = ImageFolder('image_dataset/train', transform=train_transform)
test_dataset = ImageFolder('image_dataset/test', transform=test_transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)

# Define CNN model
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 30 * 30, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.batch_norm1 = nn.BatchNorm2d(32)
        self.batch_norm2 = nn.BatchNorm2d(64)
        self.batch_norm3 = nn.BatchNorm2d(128)

    def forward(self, x):
        x = self.pool(self.relu(self.batch_norm1(self.conv1(x))))
```

```
        x = self.pool(self.relu(self.batch_norm2(self.conv2(x))))
        x = self.pool(self.relu(self.batch_norm3(self.conv3(x))))
        x = x.view(-1, 128 * 30 * 30)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

model = CNN(num_classes=12).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training Loop
epochs = 50
for epoch in range(epochs):
    model.train()
    train_loss = 0
    correct = 0
    total = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    if (epoch + 1) % 5 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {train_loss/len(train_loader):.4f}, Accu

# Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

📖 Explanation

CNN (Convolutional Neural Network) is specialized for image data, using convolutional layers to detect patterns and features.

- **Convolutional Layers:** Extract spatial features (edges, textures, shapes)
- **Max Pooling:** Reduces spatial dimensions, retains important features
- **Data Augmentation:** Random flips and rotations increase training diversity
- **Batch Normalization:** Stabilizes training in convolutional layers
- **Image Normalization:** Uses ImageNet mean/std for better convergence

🔧 **Popular CNN Architectures:**

- **LeNet:** Classic architecture for digit recognition
- **AlexNet:** First deep CNN to win ImageNet
- **VGG:** Very deep with small 3x3 filters
- **ResNet:** Skip connections, 50-152 layers
- **Inception:** Multiple filter sizes in parallel
- **EfficientNet:** Optimized accuracy and efficiency

5 Using TorchVision Pre-trained Models

```
import torch
import torch.nn as nn
import torchvision.models as models
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Data transforms
train_transform = transforms.Compose([
    transforms.Resize((244, 244)),
```

```
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((244, 244)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Load datasets
train_dataset = ImageFolder('image_dataset/train', transform=train_transform)
test_dataset = ImageFolder('image_dataset/test', transform=test_transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=32, num_workers=4)

# Load pre-trained ResNet18
model = models.resnet18(pretrained=True)

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

# Replace final layer for 12 classes
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 12)

model = model.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.fc.parameters(), lr=0.001)  # Only train final layer

# Training loop
epochs = 20
for epoch in range(epochs):
    model.train()
    train_loss = 0
    correct = 0
    total = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
```

```
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Epoch [{epoch+1}/{epochs}], Loss: {train_loss/len(train_loader):.4f}, Acc: {ac

# Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

📖 Explanation

TorchVision Models provides pre-trained models on ImageNet dataset, allowing quick deployment with excellent performance.

- **Pre-trained Weights:** Model already learned features from millions of images
- **Feature Extraction:** Freeze early layers, only train final classifier
- **ImageFolder:** Convenient dataset loader for folder-organized images
- **Faster Training:** Much faster than training from scratch
- **Better Performance:** Pre-trained features generalize well

🔧 Available TorchVision Models:

- **ResNet (18, 34, 50, 101, 152):** Deep residual networks
- **VGG (11, 13, 16, 19):** Very deep convolutional networks
- **DenseNet (121, 161, 169, 201):** Dense connections
- **MobileNet V2/V3:** Efficient for mobile devices
- **EfficientNet (B0-B7):** Scaled architectures

- **Vision Transformer (ViT):** Transformer-based vision model

6 Download Different Pre-trained Models

```
import torch
import torch.nn as nn
import torchvision.models as models
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Data preparation
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

train_dataset = ImageFolder('image_dataset/train', transform=transform)
test_dataset = ImageFolder('image_dataset/test', transform=transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)

# Download and use different models
def get_model(model_name, num_classes=12):
    if model_name == 'resnet50':
        model = models.resnet50(pretrained=True)
        num_features = model.fc.in_features
        model.fc = nn.Linear(num_features, num_classes)

    elif model_name == 'vgg16':
        model = models.vgg16(pretrained=True)
        num_features = model.classifier[6].in_features
        model.classifier[6] = nn.Linear(num_features, num_classes)

    elif model_name == 'densenet121':
        model = models.densenet121(pretrained=True)
        num_features = model.classifier.in_features
        model.classifier = nn.Linear(num_features, num_classes)

    elif model_name == 'mobilenet_v2':
        model = models.mobilenet_v2(pretrained=True)
```

```
        num_features = model.classifier[1].in_features
        model.classifier[1] = nn.Linear(num_features, num_classes)

    elif model_name == 'efficientnet_b0':
        model = models.efficientnet_b0(pretrained=True)
        num_features = model.classifier[1].in_features
        model.classifier[1] = nn.Linear(num_features, num_classes)

    else:
        raise ValueError(f'Model {model_name} not supported')

    return model

# Choose model
model_name = 'resnet50'  # Change this to try different models
model = get_model(model_name).to(device)

# Freeze feature extraction layers
for name, param in model.named_parameters():
    if 'fc' not in name and 'classifier' not in name:
        param.requires_grad = False

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0

# Training function
def train_model(model, train_loader, criterion, optimizer, epochs=10):
    for epoch in range(epochs):
        model.train()
        train_loss = 0
        correct = 0
        total = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {train_loss/len(train_loader):.4f}, Acc:

# Evaluation function
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
```

```
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

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

# Train and evaluate
train_model(model, train_loader, criterion, optimizer, epochs=15)
evaluate_model(model, test_loader)
```

📖 Explanation

Model Hub approach allows easy experimentation with different architectures to find the best one for your task.

- **Model Selection:** Easy switching between architectures
- **Automatic Download:** PyTorch downloads pre-trained weights automatically
- **Unified Interface:** Same training code works for all models
- **Architecture Comparison:** Test multiple models quickly
- **Best Practices:** Freeze backbone, train classifier

🔧 Model Selection Guide:

- **ResNet50/101:** Good balance of accuracy and speed
- **VGG16:** Simple architecture, high memory usage
- **DenseNet:** Efficient, lower parameters, good accuracy
- **MobileNet:** Fast inference, mobile-friendly
- **EfficientNet:** Best accuracy vs efficiency trade-off

7 Transfer Learning (Fine-tuning)

```
import torch
import torch.nn as nn
import torchvision.models as models
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Data transforms with augmentation
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(20),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

train_dataset = ImageFolder('image_dataset/train', transform=train_transform)
test_dataset = ImageFolder('image_dataset/test', transform=test_transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=32, num_workers=4)

# Load pre-trained model
model = models.resnet50(pretrained=True)

# STAGE 1: Feature Extraction (freeze all but final layer)
print("Stage 1: Training final layer only")
for param in model.parameters():
    param.requires_grad = False

num_features = model.fc.in_features
model.fc = nn.Sequential(
    nn.Dropout(0.5),
    nn.Linear(num_features, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, 12)
)
```

```
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.fc.parameters(), lr=0.001)

# Train stage 1
epochs_stage1 = 10
for epoch in range(epochs_stage1):
    model.train()
    train_loss = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    print(f'Stage 1 - Epoch [{epoch+1}/{epochs_stage1}], Loss: {train_loss/len(train_loade

# STAGE 2: Fine-tuning (unfreeze last few layers)
print("\nStage 2: Fine-tuning entire network")

# Unfreeze all layers
for param in model.parameters():
    param.requires_grad = True

# Different learning rates for different layers
optimizer = torch.optim.Adam([
    {'params': model.layer4.parameters(), 'lr': 1e-4},
    {'params': model.fc.parameters(), 'lr': 1e-3}
], lr=1e-5)

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=3)

# Train stage 2
epochs_stage2 = 20
best_accuracy = 0
for epoch in range(epochs_stage2):
    model.train()
    train_loss = 0
    correct = 0
    total = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
```

```
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_accuracy = 100 * correct / total
        avg_loss = train_loss / len(train_loader)
        scheduler.step(avg_loss)

        # Validation
        model.eval()
        val_correct = 0
        val_total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()

        val_accuracy = 100 * val_correct / val_total

        print(f'Stage 2 - Epoch [{epoch+1}/{epochs_stage2}]')
        print(f'Train Loss: {avg_loss:.4f}, Train Acc: {train_accuracy:.2f}%, Val Acc: {val_ac

        # Save best model
        if val_accuracy > best_accuracy:
            best_accuracy = val_accuracy
            torch.save(model.state_dict(), 'best_model.pth')
            print(f'Best model saved with accuracy: {best_accuracy:.2f}%')

print(f'\nBest Validation Accuracy: {best_accuracy:.2f}%')
```

📖 Explanation

Transfer Learning leverages knowledge from pre-trained models, fine-tuning them for specific tasks with less data and training time.

- **Two-Stage Training:** First train classifier, then fine-tune entire network
- **Differential Learning Rates:** Lower LR for pre-trained layers, higher for new layers
- **Gradual Unfreezing:** Prevents catastrophic forgetting of learned features
- **Model Checkpointing:** Save best model based on validation accuracy
- **Learning Rate Scheduling:** Adaptive learning rate based on performance

🔧 Transfer Learning Strategies:

- **Feature Extraction:** Freeze all layers, train only classifier
- **Fine-tuning Last Layers:** Unfreeze last few conv blocks
- **Full Fine-tuning:** Train entire network with small LR
- **Progressive Unfreezing:** Gradually unfreeze layers during training
- **Discriminative Learning Rates:** Different LR for different layers

💡 **Fine-tuning Tips:**

- Use smaller learning rates (1e-4 to 1e-5) for pre-trained layers
- Apply more aggressive data augmentation
- Use larger dropout for regularization
- Monitor validation loss to prevent overfitting

8 Time Series Analysis (LSTM)

```
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import TimeSeriesSplit
from torch.utils.data import DataLoader, TensorDataset

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Load time series data
df = pd.read_csv('timeseries_data.csv')
df['date'] = pd.to_datetime(df['date'])  # First column: date
df = df.sort_values('date')

# Extract features and targets
dates = df['date'].values
targets = df.iloc[:, 1:4].values  # 3 output columns

# Scale data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(targets)
```

```python
# Create sequences
def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)

seq_length = 30  # Use 30 time steps to predict next
X, y = create_sequences(scaled_data, seq_length)

# Time Series Cross-Validation Split
tscv = TimeSeriesSplit(n_splits=5)
for train_idx, test_idx in tscv.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]
    break  # Use first split for demo

# Convert to tensors
X_train = torch.FloatTensor(X_train).to(device)
X_test = torch.FloatTensor(X_test).to(device)
y_train = torch.FloatTensor(y_train).to(device)
y_test = torch.FloatTensor(y_test).to(device)

# Create DataLoaders
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64)

# Define LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim, dropout=0.2):
        super(LSTMModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers,
                            batch_first=True, dropout=dropout)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state and cell state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)

        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))

        # Get output from last time step
        out = self.fc(out[:, -1, :])
        return out

# Model parameters
```

```python
input_dim = 3  # 3 output features
hidden_dim = 128
num_layers = 2
output_dim = 3

model = LSTMModel(input_dim, hidden_dim, num_layers, output_dim).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=10)

# Training loop
epochs = 100
best_loss = float('inf')

for epoch in range(epochs):
    model.train()
    train_loss = 0

    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(train_loader)

    # Validation
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            val_loss += loss.item()

    val_loss /= len(test_loader)
    scheduler.step(val_loss)

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Train Loss: {train_loss:.4f}, Val Loss: {val_l

    # Save best model
    if val_loss < best_loss:
        best_loss = val_loss
        torch.save(model.state_dict(), 'best_lstm_model.pth')

# Load best model and evaluate
model.load_state_dict(torch.load('best_lstm_model.pth'))
model.eval()

# Make predictions
with torch.no_grad():
```

```
    predictions = model(X_test).cpu().numpy()
    actual = y_test.cpu().numpy()

    # Inverse transform to original scale
    predictions = scaler.inverse_transform(predictions)
    actual = scaler.inverse_transform(actual)

    # Calculate metrics
    mse = np.mean((predictions - actual) ** 2)
    mae = np.mean(np.abs(predictions - actual))
    print(f'\nTest MSE: {mse:.4f}')
    print(f'Test MAE: {mae:.4f}')
```

📖 Explanation

LSTM (Long Short-Term Memory) networks are specialized RNNs designed for time series data, capable of learning long-term dependencies.

- **Sequence Creation:** Creates sliding windows of historical data
- **TimeSeriesSplit:** Proper cross-validation that respects temporal order
- **LSTM Architecture:** Memory cells maintain information across time steps
- **Gradient Clipping:** Prevents exploding gradients in RNNs
- **Multi-output Prediction:** Predicts 3 values simultaneously

🔧 Time Series Models:

- **LSTM:** Handles long-term dependencies, best for complex patterns
- **GRU (Gated Recurrent Unit):** Simpler than LSTM, faster training
- **Bi-directional LSTM:** Processes sequences forward and backward
- **Transformer (Temporal Fusion Transformer):** State-of-the-art for time series
- **TCN (Temporal Convolutional Network):** CNN-based, parallelizable
- **Prophet:** Facebook's forecasting tool for business time series
- **ARIMA/SARIMA:** Classical statistical methods

💡 Time Series Cross-Validation (Group CV):

- **TimeSeriesSplit:** Respects temporal order, no data leakage
- **Rolling Window:** Fixed-size training window moves forward

- **Expanding Window:** Training set grows, always uses all past data
- **Blocked CV:** Creates non-overlapping time blocks

🔍 **Important Considerations:**

- Never shuffle time series data during training
- Handle seasonality and trends appropriately
- Check for stationarity (constant mean/variance)
- Be careful of data leakage in feature engineering
- Use appropriate evaluation metrics (MAE, RMSE, MAPE)

9 Bonus: GRU for Time Series

```
import torch
import torch.nn as nn

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define GRU model
class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim, dropout=0.2):
        super(GRUModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.gru = nn.GRU(input_dim, hidden_dim, num_layers,
                          batch_first=True, dropout=dropout)

        self.fc = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(hidden_dim // 2, output_dim)
        )

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

```
# Initialize model (use same data preparation as LSTM example)
model = GRUModel(input_dim=3, hidden_dim=128, num_layers=2, output_dim=3).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

print("GRU model initialized. Use same training loop as LSTM example.")
```

📖 Explanation

GRU (Gated Recurrent Unit) is a simpler alternative to LSTM with fewer parameters and faster training.

- **Simpler Architecture:** Only 2 gates (reset and update) vs LSTM's 3
- **Faster Training:** Fewer parameters mean faster computation
- **Similar Performance:** Often performs comparably to LSTM
- **Better for Shorter Sequences:** LSTM better for very long sequences

10 Bonus: LSTM with Attention

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Attention mechanism
class Attention(nn.Module):
    def __init__(self, hidden_dim):
        super(Attention, self).__init__()
        self.attention = nn.Linear(hidden_dim, 1)

    def forward(self, lstm_output):
        # lstm_output shape: (batch, seq_len, hidden_dim)
        attention_weights = F.softmax(self.attention(lstm_output), dim=1)
        context_vector = torch.sum(attention_weights * lstm_output, dim=1)
        return context_vector, attention_weights

# LSTM with Attention
```

```python
class LSTMAttention(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim, dropout=0.2):
        super(LSTMAttention, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers,
                            batch_first=True, dropout=dropout)
        self.attention = Attention(hidden_dim)
        self.fc = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim // 2, output_dim)
        )

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)

        # LSTM output for all time steps
        lstm_out, _ = self.lstm(x, (h0, c0))

        # Apply attention
        context_vector, attention_weights = self.attention(lstm_out)

        # Final prediction
        out = self.fc(context_vector)
        return out

# Initialize model
model = LSTMAttention(input_dim=3, hidden_dim=128, num_layers=2, output_dim=3).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

print("LSTM with Attention initialized. Use same training loop as previous examples.")
```

📖 Explanation

Attention Mechanism allows the model to focus on the most relevant time steps when making predictions.

- **Weighted Combination:** Each time step gets an importance weight
- **Better Long Sequences:** Helps model focus on relevant information
- **Interpretability:** Attention weights show which time steps matter
- **Improved Accuracy:** Often outperforms vanilla LSTM

🔧 **Advanced Time Series Techniques:**

- **Multi-head Attention:** Multiple attention mechanisms in parallel
- **Seq2Seq with Attention:** For multi-step forecasting
- **Autoencoder LSTM:** Unsupervised feature learning
- **Ensemble Methods:** Combine multiple models

🎯 **Summary:** This guide covers 10 complete PyTorch implementations from basic regression to advanced time series forecasting.

💻 **All code is production-ready and uses GPU when available.**