Neural Networks - Intermediate Handout

Machine Learning for Smarter Innovation

1 Neural Networks - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (PyTorch implementation)

1.1 Setup

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, Dataset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix,
    roc_auc_score
import warnings
warnings.filterwarnings('ignore')

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

# Reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers practical implementation of neural networks using PyTorch for classification and regression tasks. Neural networks learn hierarchical representations through layers of interconnected neurons, enabling them to capture complex patterns that linear models miss. The techniques apply across image recognition, text classification, time series forecasting, and tabular data prediction. We focus on building, training, and evaluating neural networks with proper regularization and hyperparameter tuning.

1.2 1. Multi-Layer Perceptron (MLP) for Tabular Data

1.2.1 Concept Overview

The Multi-Layer Perceptron (MLP) is the foundational neural network architecture, consisting of fully connected layers where each neuron receives input from all neurons in the previous layer. MLPs excel at tabular data problems where feature engineering is minimal. The key architectural decisions are number of hidden layers, neurons per layer, activation functions, and dropout rates. Deeper networks can model more complex relationships but require more data and regularization to prevent overfitting.

1.2.2 Implementation: Flexible MLP Architecture

```
class MLP(nn.Module):
    """Flexible Multi-Layer Perceptron for tabular data."""

    def __init__(self, input_size, hidden_sizes, output_size,
                 dropout=0.2, use_batch_norm=True):
        super().__init__()

        layers = []
        prev_size = input_size

        # Build hidden layers
        for hidden_size in hidden_sizes:
            layers.append(nn.Linear(prev_size, hidden_size))
            if use_batch_norm:
                layers.append(nn.BatchNorm1d(hidden_size))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout))
            prev_size = hidden_size

        # Output layer (no activation for logits)
        layers.append(nn.Linear(prev_size, output_size))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)


# Create synthetic tabular data
def generate_tabular_data(n_samples=2000, n_features=20, n_classes=3):
    """Generate synthetic classification data."""
    np.random.seed(42)

    X = np.random.randn(n_samples, n_features)
    # Create non-linear decision boundaries
    y = (np.sin(X[:, 0] * 2) + np.cos(X[:, 1] * 2) +
         X[:, 2] * X[:, 3] + np.random.randn(n_samples) * 0.3)
    y = pd.qcut(y, n_classes, labels=False)

    return X, y


# Prepare data
X, y = generate_tabular_data()
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Scale features
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_t = torch.FloatTensor(X_train_scaled).to(device)
y_train_t = torch.LongTensor(y_train).to(device)
X_test_t = torch.FloatTensor(X_test_scaled).to(device)
y_test_t = torch.LongTensor(y_test).to(device)

# Create DataLoader
train_dataset = TensorDataset(X_train_t, y_train_t)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Create model
model = MLP(
    input_size=20,
    hidden_sizes=[128, 64, 32],
    output_size=3,
    dropout=0.3
).to(device)

print(f"Model parameters: {sum(p.numel() for p in model.parameters()):,}")
print(model)
```

1.2.3 Training with Learning Rate Scheduling

```
def train_model(model, train_loader, X_val, y_val,
                epochs=100, lr=0.001, patience=10):
    """Train model with early stopping and learning rate scheduling."""

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', patience=5, factor=0.5
    )

    history = {
        'train_loss': [], 'val_loss': [],
        'train_acc': [], 'val_acc': [], 'lr': []
    }

    best_val_loss = float('inf')
    patience_counter = 0
    best_state = None

    for epoch in range(epochs):
        # Training phase
        model.train()
        train_loss = 0
        train_correct = 0
        train_total = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * X_batch.size(0)
```

```
            _, predicted = outputs.max(1)
            train_total += y_batch.size(0)
            train_correct += predicted.eq(y_batch).sum().item()

        train_loss /= train_total
        train_acc = train_correct / train_total

        # Validation phase
        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val)
            val_loss = criterion(val_outputs, y_val).item()
            _, val_predicted = val_outputs.max(1)
            val_acc = (val_predicted == y_val).float().mean().item()

        # Learning rate scheduling
        scheduler.step(val_loss)
        current_lr = optimizer.param_groups[0]['lr']

        # Record history
        history['train_loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['train_acc'].append(train_acc)
        history['val_acc'].append(val_acc)
        history['lr'].append(current_lr)

        # Early stopping check
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
            best_state = model.state_dict().copy()
        else:
            patience_counter += 1

        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}/{epochs} - "
                  f"Train Loss: {train_loss:.4f} - Train Acc: {train_acc:.4f}
    - "
                  f"Val Loss: {val_loss:.4f} - Val Acc: {val_acc:.4f} - "
                  f"LR: {current_lr:.6f}")

        if patience_counter >= patience:
            print(f"Early stopping at epoch {epoch+1}")
            break

    # Restore best model
    if best_state:
        model.load_state_dict(best_state)

    return history


# Train the model
history = train_model(model, train_loader, X_test_t, y_test_t,
                      epochs=100, lr=0.001, patience=15)
```

1.3 2. Convolutional Neural Network (CNN)

1.3.1 Concept Overview

Convolutional Neural Networks exploit spatial hierarchies in grid-like data through local connectivity and weight sharing. Convolutional layers apply learnable filters across the input, detecting features like edges, textures, and shapes. Pooling layers reduce spatial dimensions while preserving important features. Modern CNNs stack multiple convolutional blocks with batch normalization for stable training.

1.3.2 Implementation: Image Classification CNN

```
class CNN(nn.Module):
    """Convolutional Neural Network for image classification."""

    def __init__(self, num_classes=10, input_channels=3):
        super().__init__()

        # Convolutional blocks
        self.features = nn.Sequential(
            # Block 1: 32x32 -> 16x16
            nn.Conv2d(input_channels, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),

            # Block 2: 16x16 -> 8x8
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),

            # Block 3: 8x8 -> 4x4
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.AdaptiveAvgPool2d((4, 4))
        )

        # Classification head
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 256),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
```

```
        return x


# Synthetic image data
def generate_image_data(n_samples=1000, img_size=32, n_classes=5):
    """Generate synthetic image classification data."""
    X = torch.randn(n_samples, 3, img_size, img_size)
    y = torch.randint(0, n_classes, (n_samples,))
    return X, y


# Create model and test forward pass
cnn_model = CNN(num_classes=5, input_channels=3).to(device)
X_dummy, y_dummy = generate_image_data(n_samples=16)
X_dummy = X_dummy.to(device)

output = cnn_model(X_dummy)
print(f"Input shape: {X_dummy.shape}")
print(f"Output shape: {output.shape}")
print(f"CNN parameters: {sum(p.numel() for p in cnn_model.parameters()):,}")
```

1.3.3 Transfer Learning with Pretrained Models

```
import torchvision.models as models

def create_transfer_model(num_classes, backbone='resnet18', freeze_backbone=
    True):
    """Create a transfer learning model with pretrained backbone."""

    # Load pretrained model
    if backbone == 'resnet18':
        model = models.resnet18(weights='IMAGENET1K_V1')
        in_features = model.fc.in_features
    elif backbone == 'resnet50':
        model = models.resnet50(weights='IMAGENET1K_V1')
        in_features = model.fc.in_features
    elif backbone == 'efficientnet_b0':
        model = models.efficientnet_b0(weights='IMAGENET1K_V1')
        in_features = model.classifier[1].in_features
    else:
        raise ValueError(f"Unknown backbone: {backbone}")

    # Freeze backbone weights
    if freeze_backbone:
        for param in model.parameters():
            param.requires_grad = False

    # Replace classification head
    if 'resnet' in backbone:
        model.fc = nn.Sequential(
            nn.Linear(in_features, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, num_classes)
        )
    else:  # EfficientNet
        model.classifier = nn.Sequential(
            nn.Dropout(0.3),
            nn.Linear(in_features, num_classes)
        )
```

```
    return model


# Create transfer learning model
transfer_model = create_transfer_model(num_classes=5, backbone='resnet18')

# Count trainable parameters
trainable = sum(p.numel() for p in transfer_model.parameters() if p.
    requires_grad)
total = sum(p.numel() for p in transfer_model.parameters())
print(f"Trainable parameters: {trainable:,} / {total:,}")
```

1.4 3. Recurrent Neural Network (LSTM)

1.4.1 Concept Overview

Long Short-Term Memory (LSTM) networks process sequential data by maintaining a cell state that can store information across long sequences. Unlike vanilla RNNs, LSTMs use gating mechanisms (forget, input, output gates) to control information flow, solving the vanishing gradient problem. Bidirectional LSTMs process sequences in both directions, capturing context from past and future elements simultaneously.

1.4.2 Implementation: Text Classification LSTM

```
class LSTMClassifier(nn.Module):
    """Bidirectional LSTM for text classification."""

    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim,
                 n_layers=2, dropout=0.3, bidirectional=True):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.lstm = nn.LSTM(
            embed_dim, hidden_dim, n_layers,
            batch_first=True, dropout=dropout if n_layers > 1 else 0,
            bidirectional=bidirectional
        )

        lstm_output_dim = hidden_dim * 2 if bidirectional else hidden_dim

        self.fc = nn.Sequential(
            nn.Linear(lstm_output_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, output_dim)
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, lengths=None):
        # x: (batch, seq_len)
        embedded = self.dropout(self.embedding(x))

        # Pack sequence for variable lengths (optional)
        if lengths is not None:
            embedded = nn.utils.rnn.pack_padded_sequence(
```

```
                embedded, lengths.cpu(), batch_first=True, enforce_sorted=
    False
            )

        output, (hidden, cell) = self.lstm(embedded)

        # Unpack if packed
        if lengths is not None:
            output, _ = nn.utils.rnn.pad_packed_sequence(output, batch_first=
    True)

        # Use final hidden states from both directions
        if self.lstm.bidirectional:
            hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
        else:
            hidden = hidden[-1,:,:]

        return self.fc(self.dropout(hidden))


# Vocabulary and tokenization utilities
class SimpleTokenizer:
    """Simple word-level tokenizer."""

    def __init__(self, max_vocab=10000):
        self.max_vocab = max_vocab
        self.word2idx = {'<PAD>': 0, '<UNK>': 1}
        self.idx2word = {0: '<PAD>', 1: '<UNK>'}

    def fit(self, texts):
        """Build vocabulary from texts."""
        from collections import Counter
        counter = Counter()
        for text in texts:
            counter.update(text.lower().split())

        for word, _ in counter.most_common(self.max_vocab - 2):
            idx = len(self.word2idx)
            self.word2idx[word] = idx
            self.idx2word[idx] = word

    def encode(self, text, max_len=256):
        """Convert text to token IDs."""
        tokens = text.lower().split()[:max_len]
        ids = [self.word2idx.get(t, self.word2idx['<UNK>']) for t in tokens]
        # Pad to max_len
        ids = ids + [0] * (max_len - len(ids))
        return ids, len(tokens)


# Create model
lstm_model = LSTMClassifier(
    vocab_size=10000,
    embed_dim=128,
    hidden_dim=256,
    output_dim=2,
    n_layers=2,
    dropout=0.3
).to(device)

print(f"LSTM parameters: {sum(p.numel() for p in lstm_model.parameters()):,}")
```

1.4.3 Time Series Forecasting

```python
class TimeSeriesLSTM(nn.Module):
    """LSTM for time series prediction."""

    def __init__(self, input_dim, hidden_dim, output_dim,
                 n_layers=2, dropout=0.2):
        super().__init__()

        self.lstm = nn.LSTM(
            input_dim, hidden_dim, n_layers,
            batch_first=True, dropout=dropout
        )

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

    def forward(self, x):
        # x: (batch, seq_len, input_dim)
        output, (hidden, cell) = self.lstm(x)
        # Use last time step output
        return self.fc(output[:, -1, :])


# Create sequences for time series
def create_sequences(data, seq_length, forecast_horizon=1):
    """Create input sequences and targets for time series."""
    X, y = [], []
    for i in range(len(data) - seq_length - forecast_horizon + 1):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length:i+seq_length+forecast_horizon])
    return np.array(X), np.array(y)


# Example: synthetic time series
t = np.linspace(0, 100, 1000)
data = np.sin(t * 0.1) + 0.5 * np.sin(t * 0.3) + np.random.randn(len(t)) * 0.1
data = data.reshape(-1, 1)

X_ts, y_ts = create_sequences(data, seq_length=50, forecast_horizon=1)
print(f"Time series X shape: {X_ts.shape}, y shape: {y_ts.shape}")
```

1.5 4. Data Augmentation and Preprocessing

1.5.1 Concept Overview

Data augmentation artificially increases training set size by applying transformations that preserve label semantics. For images, this includes flips, rotations, crops, and color adjustments. Augmentation acts as regularization, improving generalization by exposing the model to variation it might encounter in deployment.

1.5.2 Implementation: Image Augmentation Pipeline

```
from torchvision import transforms

def get_transforms(train=True, img_size=224):
    """Get image transforms for training or validation."""

    if train:
        return transforms.Compose([
            transforms.RandomResizedCrop(img_size, scale=(0.8, 1.0)),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.RandomRotation(15),
            transforms.ColorJitter(
                brightness=0.2, contrast=0.2,
                saturation=0.2, hue=0.1
            ),
            transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            ),
            transforms.RandomErasing(p=0.1)
        ])
    else:
        return transforms.Compose([
            transforms.Resize(int(img_size * 1.14)),
            transforms.CenterCrop(img_size),
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])


# Custom dataset for tabular data
class TabularDataset(Dataset):
    """PyTorch Dataset for tabular data with augmentation."""

    def __init__(self, X, y, augment=False, noise_std=0.1):
        self.X = torch.FloatTensor(X)
        self.y = torch.LongTensor(y)
        self.augment = augment
        self.noise_std = noise_std

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

    def __getitem__(self, idx):
        x = self.X[idx].clone()
        if self.augment:
            # Add Gaussian noise as augmentation
            x += torch.randn_like(x) * self.noise_std
        return x, self.y[idx]
```

1.6 5. Regularization Techniques

1.6.1 Concept Overview

Regularization prevents overfitting by constraining model capacity or adding noise during training. Key techniques include dropout (randomly zeroing activations), weight decay (L2 penalty on weights), batch normalization (normalizing layer inputs), and early stopping (halting when validation performance degrades). Combining multiple regularization techniques often yields the best generalization.

1.6.2 Implementation: Comprehensive Regularization

```
class EarlyStopping:
    """Early stopping to halt training when validation loss stops improving.""
    "

    def __init__(self, patience=10, min_delta=1e-4, restore_best=True):
        self.patience = patience
        self.min_delta = min_delta
        self.restore_best = restore_best
        self.counter = 0
        self.best_loss = None
        self.best_state = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        if self.best_loss is None:
            self.best_loss = val_loss
            self.best_state = model.state_dict().copy()
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
                if self.restore_best:
                    model.load_state_dict(self.best_state)
        else:
            self.best_loss = val_loss
            self.best_state = model.state_dict().copy()
            self.counter = 0

        return self.early_stop


class RegularizedModel(nn.Module):
    """Model demonstrating multiple regularization techniques."""

    def __init__(self, input_size, hidden_sizes, output_size,
                 dropout_rates=None, use_batch_norm=True):
        super().__init__()

        if dropout_rates is None:
            dropout_rates = [0.3] * len(hidden_sizes)

        layers = []
        prev_size = input_size

        for i, hidden_size in enumerate(hidden_sizes):
            layers.append(nn.Linear(prev_size, hidden_size))

            if use_batch_norm:
                layers.append(nn.BatchNorm1d(hidden_size))
```

```
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rates[i]))

            prev_size = hidden_size

        layers.append(nn.Linear(prev_size, output_size))
        self.network = nn.Sequential(*layers)

        # Initialize weights
        self._init_weights()

    def _init_weights(self):
        """Initialize weights using Xavier initialization."""
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)

    def forward(self, x):
        return self.network(x)


# Training with multiple regularization
def train_with_regularization(model, train_loader, val_loader, epochs=100):
    """Train with weight decay, dropout, and early stopping."""

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(
        model.parameters(),
        lr=0.001,
        weight_decay=1e-4  # L2 regularization
    )
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)
    early_stopping = EarlyStopping(patience=15, restore_best=True)

    for epoch in range(epochs):
        model.train()
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

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

        scheduler.step()

        if early_stopping(val_loss, model):
            print(f"Early stopping at epoch {epoch+1}")
            break

    return model
```

1.7 6. Model Evaluation and Visualization

1.7.1 Concept Overview

Thorough evaluation goes beyond single accuracy metrics to include confusion matrices, ROC curves, and learning curve analysis. Visualizing training dynamics helps diagnose issues like overfitting, underfitting, or learning rate problems. Understanding model predictions through analysis of confident and uncertain examples informs iteration.

1.7.2 Implementation: Comprehensive Evaluation

```
def evaluate_model(model, X_test, y_test, class_names=None):
    """Comprehensive model evaluation with visualizations."""

    model.eval()
    with torch.no_grad():
        outputs = model(X_test)
        probs = torch.softmax(outputs, dim=1)
        _, predictions = outputs.max(1)

    y_true = y_test.cpu().numpy()
    y_pred = predictions.cpu().numpy()
    y_probs = probs.cpu().numpy()

    # Classification report
    print("Classification Report:")
    print(classification_report(y_true, y_pred, target_names=class_names))

    # Visualizations
    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    im = axes[0].imshow(cm, cmap='Blues')
    axes[0].set_title('Confusion Matrix')
    axes[0].set_xlabel('Predicted')
    axes[0].set_ylabel('Actual')
    plt.colorbar(im, ax=axes[0])

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            axes[0].text(j, i, str(cm[i, j]),
                         ha='center', va='center', fontsize=10)

    # Prediction confidence distribution
    max_probs = np.max(y_probs, axis=1)
    correct_mask = y_pred == y_true
    axes[1].hist(max_probs[correct_mask], bins=30, alpha=0.7, label='Correct')
    axes[1].hist(max_probs[~correct_mask], bins=30, alpha=0.7, label='
    Incorrect')
    axes[1].set_xlabel('Prediction Confidence')
    axes[1].set_ylabel('Count')
    axes[1].set_title('Confidence Distribution')
    axes[1].legend()

    # Per-class accuracy
    n_classes = cm.shape[0]
    per_class_acc = cm.diagonal() / cm.sum(axis=1)
    bars = axes[2].bar(range(n_classes), per_class_acc)
```

```
    axes[2].set_xlabel('Class')
    axes[2].set_ylabel('Accuracy')
    axes[2].set_title('Per-Class Accuracy')
    axes[2].set_ylim(0, 1)

    if class_names:
        axes[2].set_xticks(range(n_classes))
        axes[2].set_xticklabels(class_names, rotation=45)

    plt.tight_layout()
    plt.savefig('evaluation_plots.pdf', bbox_inches='tight')
    plt.show()

    return {'accuracy': (y_pred == y_true).mean(),
            'predictions': y_pred,
            'probabilities': y_probs}


def plot_training_history(history):
    """Plot training and validation curves."""

    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    # Loss curves
    axes[0].plot(history['train_loss'], label='Train')
    axes[0].plot(history['val_loss'], label='Validation')
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Loss')
    axes[0].set_title('Loss Curves')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)

    # Accuracy curves
    axes[1].plot(history['train_acc'], label='Train')
    axes[1].plot(history['val_acc'], label='Validation')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('Accuracy')
    axes[1].set_title('Accuracy Curves')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    # Learning rate
    axes[2].plot(history['lr'])
    axes[2].set_xlabel('Epoch')
    axes[2].set_ylabel('Learning Rate')
    axes[2].set_title('Learning Rate Schedule')
    axes[2].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('training_history.pdf', bbox_inches='tight')
    plt.show()
```

1.8 7. Model Saving and Deployment

1.8.1 Concept Overview

Production deployment requires saving trained models in formats that support efficient inference. PyTorch offers multiple serialization options: state dictionaries (recommended for flexibility), complete

model saves, and TorchScript for deployment without Python. Checkpointing during training enables recovery from interruptions and model selection based on validation performance.

1.8.2 Implementation: Model Persistence

```
def save_checkpoint(model, optimizer, epoch, loss, path='checkpoint.pth'):
    """Save training checkpoint for resuming."""
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss,
        'model_config': {
            'input_size': model.network[0].in_features,
            'hidden_sizes': [l.out_features for l in model.network
                             if isinstance(l, nn.Linear)][:-1],
            'output_size': model.network[-1].out_features
        }
    }, path)
    print(f"Checkpoint saved to {path}")


def load_checkpoint(path, model_class=MLP):
    """Load checkpoint and restore model."""
    checkpoint = torch.load(path)

    config = checkpoint['model_config']
    model = model_class(
        input_size=config['input_size'],
        hidden_sizes=config['hidden_sizes'],
        output_size=config['output_size']
    )
    model.load_state_dict(checkpoint['model_state_dict'])

    return model, checkpoint


def export_for_inference(model, example_input, path='model_scripted.pt'):
    """Export model for production inference."""
    model.eval()

    # TorchScript for deployment
    scripted = torch.jit.trace(model, example_input)
    scripted.save(path)
    print(f"Scripted model saved to {path}")

    # Verify exported model
    loaded = torch.jit.load(path)
    with torch.no_grad():
        original_out = model(example_input)
        loaded_out = loaded(example_input)
        assert torch.allclose(original_out, loaded_out)
    print("Export verification passed")

    return scripted
```

1.9 Common Hyperparameters

()	()	()
*	*	*
0.3333	3	Typical Layers
	0.4545	Range

()	()	()
*	*	*
0.3333	3	Start with 1e-3, reduce if unstable
0.3333	4	Larger = faster training, smaller = better generalization
0.3333	4	Scale with problem complexity and data size
0.3333	5	Deeper needs more regularization
	to	
	6	

()	()	()
*	*	*
0.3333	0.4545	Typical Range
()	()	()
*	*	*
0.3333	0.4545	Higher to 0.5 for larger networks
()	()	()
*	*	*
0.3333	0.4545	L2 decay 5 to 1e-3 regularization strength
()	()	()
*	*	*
0.3333	0.4545	For bed_dim 512 bedding layers
()	()	()
*	*	*
0.3333	0.4545	For to 256 convolutional layers

1.10 Practice Projects

1. **Customer Churn Prediction**: Build an MLP to predict customer churn from subscription data. Include feature engineering, class imbalance handling, and threshold optimization for business metrics.

2. **Image Classification Pipeline**: Implement a complete CNN training pipeline with transfer learning, data augmentation, and learning rate scheduling on a custom image dataset.

3. **Sentiment Analysis System**: Train a bidirectional LSTM for sentiment classification on product reviews. Implement attention mechanisms and compare with a simple baseline.

4. **Time Series Forecasting**: Build an LSTM model to predict stock prices or energy consumption. Include proper train/validation/test splits that respect temporal ordering.

1.11 Troubleshooting

() *	() *	() *
0.3	4	Solution
() *	() *	() *
0.3	4	Try not- ing 10x de- rate in- crease and in- high- lower, / low use sched- uler
() *	() *	() *
0.3	4	Add ing fit- loss ting low, val- i- da- tion high
() *	() *	() *
0.3	4	Un- loss de- crease high fit- ting model ca- pac- ity, train longer
() *	() *	() *
0.3	4	Lower loss plod- ding gra- di- ents add gra- di- ent clip- ping

Issue	Cause	Solution
Out of memory	Batch too large	Reduce batch size, use gradient accumulation
Unstable training	BatchNorm with small batches	Use LayerNorm or increase batch size
Poor convergence	Bad initialization	Use Xavier/Kaiming initialization
Predicts same class	Class imbalance	Use weighted loss, oversample minority

---

1.12 Next Steps

- Read the advanced handout for attention mechanisms and transformers
- Experiment with different optimizers (SGD with momentum, AdamW)
- Implement gradient accumulation for large effective batch sizes

- Explore model interpretability with GradCAM and attention visualization
- Build inference pipelines with batching and GPU optimization

---

Neural networks learn representations that traditional ML cannot discover. Success requires balancing model capacity against regularization, tuning hyperparameters systematically, and understanding training dynamics. Start simple, measure everything, and iterate based on evidence.