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/TensorFlow implementation)

1.1 Setup

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
import matplotlib.pyplot as plt
```

1.2 1. Multi-Layer Perceptron (MLP)

1.2.1 Basic Architecture

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size, dropout=0.2):
        super().__init__()
        layers = []

        # Input to first hidden
        layers.append(nn.Linear(input_size, hidden_sizes[0]))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(dropout))

        # Hidden layers
        for i in range(len(hidden_sizes) - 1):
            layers.append(nn.Linear(hidden_sizes[i], hidden_sizes[i+1]))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout))

        # Output layer
        layers.append(nn.Linear(hidden_sizes[-1], output_size))

        self.network = nn.Sequential(*layers)
    def forward(self, x):
        return self.network(x)

# Create model
model = MLP(input_size=784, hidden_sizes=[256, 128, 64], output_size=10)
print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")
```

1.2.2 Training Loop

```
def train_model(model, train_loader, val_loader, epochs=20, lr=0.001):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=3)

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

    for epoch in range(epochs):
        # Training
        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()
            optimizer.step()
            train_loss += loss.item()

        # Validation
        model.eval()
        val_loss = 0
        correct = 0
        total = 0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                outputs = model(X_batch)
                val_loss += criterion(outputs, y_batch).item()
                _, predicted = outputs.max(1)
                total += y_batch.size(0)
                correct += predicted.eq(y_batch).sum().item()

        # Record history
        train_loss /= len(train_loader)
        val_loss /= len(val_loader)
        val_acc = correct / total

        history['train_loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)

        scheduler.step(val_loss)

        print(f"Epoch {epoch+1}/{epochs} - "
              f"Train Loss: {train_loss:.4f} - "
              f"Val Loss: {val_loss:.4f} - "
              f"Val Acc: {val_acc:.4f}")

    return history
```

1.3 2. Convolutional Neural Network (CNN)

1.3.1 Image Classification Architecture

```python
class CNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()

        # Convolutional layers
        self.conv_layers = nn.Sequential(
            # Block 1
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            # Block 2
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            # Block 3
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d((4, 4))
        )

        # Fully connected layers
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x

model = CNN(num_classes=10)
```

1.3.2 Transfer Learning (Recommended)

```python
import torchvision.models as models

def create_transfer_model(num_classes, freeze_backbone=True):
    # Load pretrained ResNet
    model = models.resnet18(pretrained=True)

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

    # Replace final layer
    model.fc = nn.Sequential(
        nn.Linear(512, 256),
        nn.ReLU(),
        nn.Dropout(0.3),
        nn.Linear(256, num_classes)
    )

    return model

model = create_transfer_model(num_classes=5)
```

1.4 3. Recurrent Neural Network (LSTM)

1.4.1 Sequence Classification

```
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim,
                 n_layers=2, dropout=0.3):
        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,
                            bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # x: (batch, seq_len)
        embedded = self.dropout(self.embedding(x))

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

        # Concatenate final forward and backward hidden states
        hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)

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

model = LSTMClassifier(vocab_size=10000, embed_dim=128,
                       hidden_dim=256, output_dim=2)
```

1.4.2 Time Series Prediction

```
class TimeSeriesLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers=2):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

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

1.5 4. Data Preparation

1.5.1 Image Augmentation

```
from torchvision import transforms

train_transforms = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

val_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

1.5.2 Text Tokenization

```
from collections import Counter

def build_vocab(texts, max_vocab=10000):
    counter = Counter()
    for text in texts:
        counter.update(text.split())

    vocab = {'<PAD>': 0, '<UNK>': 1}
    for word, _ in counter.most_common(max_vocab - 2):
        vocab[word] = len(vocab)
    return vocab

def encode_text(text, vocab, max_len=256):
    tokens = text.split()[:max_len]
    ids = [vocab.get(t, vocab['<UNK>']) for t in tokens]
    # Pad to max_len
    ids = ids + [vocab['<PAD>']] * (max_len - len(ids))
    return ids
```

1.6 5. Regularization Techniques

1.6.1 Early Stopping

```
class EarlyStopping:
    def __init__(self, patience=5, min_delta=0.001):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0
        return self.early_stop

# Usage
early_stopping = EarlyStopping(patience=5)
for epoch in range(100):
    # ... training code ...
    if early_stopping(val_loss):
        print(f"Early stopping at epoch {epoch}")
        break
```

1.6.2 Weight Decay (L2 Regularization)

```
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)
```

1.6.3 Batch Normalization

```
# Add after each linear/conv layer
nn.BatchNorm1d(num_features)  # For linear layers
nn.BatchNorm2d(num_channels)  # For conv layers
```

1.7 6. Model Evaluation

1.7.1 Classification Metrics

```
from sklearn.metrics import classification_report, confusion_matrix

def evaluate_model(model, test_loader):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            outputs = model(X_batch)
            _, preds = outputs.max(1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(y_batch.cpu().numpy())

    print(classification_report(all_labels, all_preds))
    # Confusion matrix
    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, cmap='Blues')
    plt.colorbar()
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

    return all_preds, all_labels
```

1.8 7. Model Saving and Loading

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

# Save only weights (recommended)
torch.save(model.state_dict(), 'model_weights.pth')

# Save checkpoint
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}, 'checkpoint.pth')

# Load
model = MLP(784, [256, 128], 10)
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```

1.9 Common Hyperparameters

() () ()
* * *
0.3438 0.2188 0.4375

Parameter	Typical Range	Notes
Learning Rate	1e-4 to 1e-2	Start with 1e-3

() () ()
* * *
0.3438 0.2188 0.4375

Parameter	Typical Range	Notes
Batch Size	32 to 256	Larger = faster, smaller = better generalization
Hidden Units	64 to 1024	Depends on problem complexity
Dropout	0.2 to 0.5	Higher for more regularization
Weight Decay	1e-5 to 1e-3	L2 regularization strength

1.10 Practice Projects

1. **MNIST Classifier**: Train MLP to classify handwritten digits
2. **CIFAR-10 CNN**: Image classification with convolutional network
3. **Sentiment Analysis**: LSTM for movie review classification
4. **Transfer Learning**: Fine-tune ResNet on custom dataset

1.11 Next Steps

- Experiment with architecture variations
- Try different optimizers (SGD, AdamW)
- Implement learning rate scheduling
- Read advanced handout for production deployment

The best way to learn is to build. Start simple, iterate, and gradually increase complexity.