Course 2

Intermediate PyTorch

Comprehensive Course Notes

Module 1: Hyperparameter Optimization with Optuna

Module 2: Computer Vision & Transfer Learning

Module 3: NLP with Transformers

Module 4: PyTorch Lightning

PyTorch Deep Learning Specialization

January 20, 2026

Contents

1 Module 1: Hyperparameter Optimization with Optuna

1.1 Introduction to AutoML

What is AutoML?

AutoML (Automated Machine Learning) automates the machine learning pipeline:

- Feature engineering
- Model selection
- **Hyperparameter optimization** ← Focus of this module
- Architecture search

1.2 Optuna Framework

Optuna Key Concepts

- **Study:** A complete optimization session
- **Trial:** A single training run with specific hyperparameters
- **Objective Function:** Returns the metric to optimize
- **Sampler:** Strategy for selecting hyperparameters (TPE, Random, etc.)
- **Pruner:** Early stops unpromising trials

Listing 1: Basic Optuna Setup

```python
import optuna

def objective(trial):
    # Suggest hyperparameters
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)
    n_layers = trial.suggest_int("n_layers", 1, 5)
    dropout = trial.suggest_float("dropout", 0.1, 0.5)

    # Build model with suggested params
    model = build_model(n_layers, dropout)

    # Train and evaluate
    accuracy = train_and_evaluate(model, lr)

    return accuracy  # Optuna will maximize/minimize this

# Create and run study
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=100)

# Best hyperparameters
print(study.best_params)
print(study.best_value)
```

1.3 Suggest Methods

Method	Usage
`suggest_int(name, low, high)`	Integer values
`suggest_float(name, low, high)`	Float values
`suggest_float(..., log=True)`	Log-scale sampling (for learning rates)
`suggest_categorical(name, choices)`	Choose from list of options

Listing 2: Hyperparameter Suggestions

```
def objective(trial):
    # Learning rate (log scale)
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)

    # Architecture choices
    n_filters = trial.suggest_int("n_filters", 16, 128)
    n_layers = trial.suggest_int("n_layers", 2, 6)

    # Regularization
    dropout = trial.suggest_float("dropout", 0.0, 0.5)

    # Categorical choice
    optimizer_name = trial.suggest_categorical(
        "optimizer", ["Adam", "SGD", "RMSprop"]
    )

    # Build and train model...
```

1.4 FlexibleCNN Architecture

Listing 3: Flexible CNN for Hyperparameter Search

```
class FlexibleCNN(nn.Module):
    def __init__(self, n_layers, n_filters, dropout_rate, num_classes):
        super().__init__()

        layers = []
        in_channels = 3

        for i in range(n_layers):
            out_channels = n_filters * (2 ** i)  # Double channels
            layers.extend([
                nn.Conv2d(in_channels, out_channels, 3, padding=1),
                nn.BatchNorm2d(out_channels),
                nn.ReLU(),
                nn.MaxPool2d(2, 2)
            ])
            in_channels = out_channels

        self.features = nn.Sequential(*layers)
        self.flatten = nn.Flatten()

        # Calculate flattened size
        with torch.no_grad():
            dummy = torch.zeros(1, 3, 32, 32)
            flat_size = self.features(dummy).view(1, -1).size(1)
```

```
        self.classifier = nn.Sequential(
            nn.Linear(flat_size, 256),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(256, num_classes)
        )

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

1.5 Pruning Unpromising Trials

Median Pruner

Stops trials that perform worse than the median of completed trials at the same step:

```
study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.MedianPruner(
        n_startup_trials=5,
        n_warmup_steps=5
    )
)
```

Listing 4: Reporting for Pruning

```
def objective(trial):
    model = build_model(trial)

    for epoch in range(num_epochs):
        train_loss = train_one_epoch(model)
        val_acc = evaluate(model)

        # Report intermediate value for pruning
        trial.report(val_acc, epoch)

        # Check if trial should be pruned
        if trial.should_prune():
            raise optuna.TrialPruned()

    return val_acc
```

1.6 Study Persistence

Listing 5: Saving and Loading Studies

```
# Save to SQLite database
study = optuna.create_study(
    study_name="cnn_optimization",
    storage="sqlite:///optuna_study.db",
    load_if_exists=True  # Resume existing study
)

# Continue optimization
study.optimize(objective, n_trials=50)
```

```python

# Analyze results
print(f"Best trial: {study.best_trial.number}")
print(f"Best params: {study.best_params}")
print(f"Best value: {study.best_value}")

# Get all trials
df = study.trials_dataframe()
```

1.7 Visualization

Listing 6: Optuna Visualization

```python
import optuna.visualization as vis

# Parameter importance
vis.plot_param_importances(study)

# Optimization history
vis.plot_optimization_history(study)

# Parameter relationships
vis.plot_parallel_coordinate(study)

# Hyperparameter slice
vis.plot_slice(study)
```

2 Module 2: Computer Vision & Transfer Learning

2.1 What is Transfer Learning?

Transfer Learning

Using a model pretrained on a large dataset (e.g., ImageNet) and adapting it to a new task:

- Leverages learned features (edges, textures, objects)
- Requires less data for the new task
- Faster training convergence
- Often achieves better results than training from scratch

2.2 TorchVision Models

Listing 7: Loading Pretrained Models

```
import torchvision.models as models

# Load pretrained model
model = models.mobilenet_v3_small(weights='IMAGENET1K_V1')

# Alternative models
resnet18 = models.resnet18(weights='IMAGENET1K_V1')
efficientnet = models.efficientnet_b0(weights='IMAGENET1K_V1')
vgg16 = models.vgg16(weights='IMAGENET1K_V1')
```

Model Selection Considerations

- **MobileNetV3:** Lightweight, fast, mobile-friendly
- **ResNet:** Reliable, well-understood, various depths
- **EfficientNet:** Best accuracy/efficiency tradeoff
- **VGG:** Simple architecture, good for visualization

2.3 Modifying for Custom Tasks

Listing 8: Replacing the Classifier

```
# MobileNetV3
model = models.mobilenet_v3_small(weights='IMAGENET1K_V1')
num_classes = 10

# Get number of input features to classifier
in_features = model.classifier[0].in_features

# Replace classifier
model.classifier = nn.Sequential(
    nn.Linear(in_features, 256),
    nn.Hardswish(),
    nn.Dropout(p=0.2),
    nn.Linear(256, num_classes)
)
```

```
# ResNet (replaces fc layer)
resnet = models.resnet18(weights='IMAGENET1K_V1')
resnet.fc = nn.Linear(resnet.fc.in_features, num_classes)
```

2.4 Freezing Pretrained Weights

Listing 9: Feature Extraction vs Fine-Tuning

```
# Option 1: Feature Extraction (freeze all)
for param in model.parameters():
    param.requires_grad = False

# Unfreeze classifier only
for param in model.classifier.parameters():
    param.requires_grad = True

# Option 2: Gradual Unfreezing (fine-tune later layers)
# Freeze early layers
for name, param in model.named_parameters():
    if 'features.0' in name or 'features.1' in name:
        param.requires_grad = False
```

Freezing Strategy

1. Start with frozen backbone, train classifier only
2. Gradually unfreeze later layers
3. Use lower learning rate for pretrained weights

2.5 ImageFolder Dataset

Listing 10: Using ImageFolder

```
from torchvision.datasets import ImageFolder

# Directory structure:
# data/
#   train/
#     class_a/
#     class_b/
#   val/
#     class_a/
#     class_b/

train_dataset = ImageFolder(
    root='data/train',
    transform=train_transform
)

val_dataset = ImageFolder(
    root='data/val',
    transform=val_transform
)

# Class names
```

```
print(train_dataset.classes)  # ['class_a', 'class_b']
print(train_dataset.class_to_idx)  # {'class_a': 0, 'class_b': 1}
```

2.6 ImageNet Preprocessing

Listing 11: Standard Preprocessing for Pretrained Models

```
# ImageNet statistics
IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224, 0.225]

train_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(IMAGENET_MEAN, IMAGENET_STD)
])

val_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(IMAGENET_MEAN, IMAGENET_STD)
])
```

2.7 Learning Rate Scheduling

Listing 12: Learning Rate Schedulers

```
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau

# Step decay
scheduler = StepLR(optimizer, step_size=10, gamma=0.1)
# Reduce LR by 0.1 every 10 epochs

# Adaptive (reduce on plateau)
scheduler = ReduceLROnPlateau(
    optimizer,
    mode='min',        # Minimize metric
    factor=0.1,        # Multiply LR by this
    patience=5,        # Wait this many epochs
    verbose=True
)

# In training loop
for epoch in range(num_epochs):
    train_loss = train_epoch(...)
    val_loss = validate(...)

    # For StepLR
    scheduler.step()

    # For ReduceLROnPlateau
    scheduler.step(val_loss)
```

2.8 Differential Learning Rates

Listing 13: Different LR for Different Layers

```
# Lower LR for pretrained features, higher for new classifier
optimizer = optim.Adam([
    {'params': model.features.parameters(), 'lr': 1e-5},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
])
```

3 Module 3: NLP with Transformers

3.1 Text Classification Pipeline

NLP Pipeline Steps

1. **Tokenization:** Convert text to tokens (words/subwords)
2. **Encoding:** Convert tokens to numerical IDs
3. **Embedding:** Map IDs to dense vectors
4. **Processing:** Apply transformer/RNN layers
5. **Classification:** Final prediction layer

3.2 Tokenization

Listing 14: Basic Tokenization

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('distilbert-base-uncased')

# Tokenize text
text = "I love PyTorch!"
tokens = tokenizer(
    text,
    padding='max_length',
    truncation=True,
    max_length=128,
    return_tensors='pt'
)

print(tokens['input_ids'].shape)      # [1, 128]
print(tokens['attention_mask'].shape) # [1, 128]
```

Tokenizer Output

- **input_ids:** Token indices for the vocabulary
- **attention_mask:** 1 for real tokens, 0 for padding
- **Special tokens:** [CLS] at start, [SEP] at end

3.3 DistilBERT for Classification

Listing 15: Loading DistilBERT

```
from transformers import DistilBertModel, DistilBertConfig

# Load pretrained DistilBERT
bert = DistilBertModel.from_pretrained('distilbert-base-uncased')

# Get hidden size
hidden_size = bert.config.hidden_size  # 768
```

3.4 Custom Text Classifier

Listing 16: Text Classification Model

```
class TextClassifier(nn.Module):
    def __init__(self, num_classes, dropout=0.3):
        super().__init__()

        self.bert = DistilBertModel.from_pretrained(
            'distilbert-base-uncased'
        )
        self.dropout = nn.Dropout(dropout)
        self.classifier = nn.Linear(
            self.bert.config.hidden_size,  # 768
            num_classes
        )

    def forward(self, input_ids, attention_mask):
        # Get BERT output
        outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask
        )

        # Use [CLS] token representation
        cls_output = outputs.last_hidden_state[:, 0, :]

        # Classify
        x = self.dropout(cls_output)
        logits = self.classifier(x)
        return logits
```

3.5 Custom Dataset for Text

Listing 17: Text Dataset Class

```
class TextDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        encoding = self.tokenizer(
            self.texts[idx],
            padding='max_length',
            truncation=True,
            max_length=self.max_length,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].squeeze(0),
            'attention_mask': encoding['attention_mask'].squeeze(0),
            'label': torch.tensor(self.labels[idx], dtype=torch.long)
```

```
        }
```

3.6 Handling Imbalanced Classes

Listing 18: Weighted Cross-Entropy

```
# Calculate class weights (inverse frequency)
class_counts = [1000, 200, 50]  # Example counts
weights = [1.0 / c for c in class_counts]
weights = torch.tensor(weights) / sum(weights) * len(weights)

# Weighted loss
criterion = nn.CrossEntropyLoss(weight=weights.to(device))
```

3.7 Fine-Tuning Strategies

Transformer Fine-Tuning Tips

- Use small learning rate (2e-5 to 5e-5)
- Train for few epochs (2-4 typically sufficient)
- Watch for overfitting on small datasets
- Consider freezing early layers

Listing 19: Fine-Tuning Setup

```
# Freeze BERT, train classifier only
for param in model.bert.parameters():
    param.requires_grad = False

# Or use different learning rates
optimizer = optim.AdamW([
    {'params': model.bert.parameters(), 'lr': 2e-5},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
])
```

3.8 Word Embeddings Concepts

Embedding Layer

Maps discrete tokens to continuous vectors:

$$\text{embedding}(x) = W_e[x]$$

Where W_e is the embedding matrix of shape (vocab_size, embedding_dim)

```
embedding = nn.Embedding(
    num_embeddings=30000,  # Vocabulary size
    embedding_dim=768      # Vector dimension
)
```

4 Module 4: PyTorch Lightning

4.1 Why PyTorch Lightning?

Lightning Benefits

- Removes boilerplate training code
- Built-in best practices (gradient clipping, logging, checkpointing)
- Easy multi-GPU training
- Clean, organized code structure
- Focus on research, not engineering

4.2 LightningModule Structure

Listing 20: Basic LightningModule

```python
import pytorch_lightning as pl

class LitModel(pl.LightningModule):
    def __init__(self, num_classes, learning_rate=1e-3):
        super().__init__()
        self.save_hyperparameters()  # Log all __init__ args

        # Define model
        self.model = nn.Sequential(
            nn.Linear(784, 256),
            nn.ReLU(),
            nn.Linear(256, num_classes)
        )
        self.criterion = nn.CrossEntropyLoss()

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

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = self.criterion(logits, y)

        # Logging (automatic tensorboard)
        self.log('train_loss', loss, prog_bar=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = self.criterion(logits, y)

        # Calculate accuracy
        preds = logits.argmax(dim=1)
        acc = (preds == y).float().mean()

        self.log('val_loss', loss, prog_bar=True)
        self.log('val_acc', acc, prog_bar=True)
```

```
    def configure_optimizers(self):
        optimizer = torch.optim.Adam(
            self.parameters(),
            lr=self.hparams.learning_rate
        )
        return optimizer
```

4.3 LightningDataModule

Listing 21: Data Module Structure

```
class MNISTDataModule(pl.LightningDataModule):
    def __init__(self, batch_size=64, data_dir='./data'):
        super().__init__()
        self.batch_size = batch_size
        self.data_dir = data_dir
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])

    def prepare_data(self):
        # Download only (called once on main process)
        MNIST(self.data_dir, train=True, download=True)
        MNIST(self.data_dir, train=False, download=True)

    def setup(self, stage=None):
        # Called on every process
        if stage == 'fit' or stage is None:
            mnist_full = MNIST(
                self.data_dir, train=True, transform=self.transform
            )
            self.train_data, self.val_data = random_split(
                mnist_full, [55000, 5000]
            )

        if stage == 'test' or stage is None:
            self.test_data = MNIST(
                self.data_dir, train=False, transform=self.transform
            )

    def train_dataloader(self):
        return DataLoader(self.train_data, batch_size=self.batch_size,
                          shuffle=True, num_workers=4)

    def val_dataloader(self):
        return DataLoader(self.val_data, batch_size=self.batch_size,
                          num_workers=4)

    def test_dataloader(self):
        return DataLoader(self.test_data, batch_size=self.batch_size,
                          num_workers=4)
```

4.4 Trainer Configuration

Listing 22: Training with Lightning

```
from pytorch_lightning import Trainer

model = LitModel(num_classes=10)
data = MNISTDataModule(batch_size=64)

trainer = Trainer(
    max_epochs=10,
    accelerator='auto',        # Auto-detect GPU/CPU
    devices='auto',            # Use all available devices
    precision=16,              # Mixed precision (faster)
    gradient_clip_val=1.0,     # Gradient clipping
    log_every_n_steps=50
)

# Train
trainer.fit(model, data)

# Test
trainer.test(model, data)
```

4.5 Callbacks

Listing 23: Built-in Callbacks

```
from pytorch_lightning.callbacks import (
    ModelCheckpoint, EarlyStopping, LearningRateMonitor
)

# Save best model
checkpoint_callback = ModelCheckpoint(
    monitor='val_loss',
    mode='min',
    save_top_k=3,
    filename='model-{epoch:02d}-{val_loss:.2f}'
)

# Stop if no improvement
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    mode='min'
)

# Log learning rate
lr_monitor = LearningRateMonitor(logging_interval='step')

trainer = Trainer(
    callbacks=[checkpoint_callback, early_stop, lr_monitor],
    max_epochs=100
)
```

4.6 Learning Rate Scheduling

Listing 24: Adding Scheduler to LightningModule

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
```

```
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        mode='min',
        factor=0.1,
        patience=3
    )

    return {
        'optimizer': optimizer,
        'lr_scheduler': {
            'scheduler': scheduler,
            'monitor': 'val_loss',  # Metric to monitor
            'interval': 'epoch',
            'frequency': 1
        }
    }
```

4.7 Logging Metrics

Listing 25: Advanced Logging

```
def training_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    loss = self.criterion(logits, y)

    # Multiple metrics
    preds = logits.argmax(dim=1)
    acc = (preds == y).float().mean()

    # Log with different settings
    self.log('train_loss', loss, on_step=True, on_epoch=True)
    self.log('train_acc', acc, on_step=False, on_epoch=True)

    # Log dictionary
    self.log_dict({
        'train/loss': loss,
        'train/acc': acc
    }, prog_bar=True)

    return loss
```

4.8 Loading Checkpoints

Listing 26: Checkpoint Management

```
# Load from checkpoint
model = LitModel.load_from_checkpoint(
    'checkpoints/best-model.ckpt',
    num_classes=10
)

# Resume training
trainer = Trainer(max_epochs=20)
trainer.fit(model, data, ckpt_path='checkpoints/last.ckpt')
```

```
# Inference
model.eval()
with torch.no_grad():
    predictions = model(test_input)
```

5 Quiz Review: Key Concepts

5.1 Quiz 5: Hyperparameter Optimization

1. **AutoML scope:** Hyperparameters, architecture, feature selection
2. **suggest_float(..., log=True)** for learning rates
3. Trial pruning stops **unpromising configurations early**
4. `trial.report()` used to report intermediate results
5. Flexible CNN layer structure enables architecture search
6. **n_startup_trials** sets trials before pruner activates
7. Study results accessed via `study.best_params`
8. SQLite storage enables study persistence/resumption
9. `suggest_categorical()` for discrete choices like optimizer
10. Optuna is not neural-network specific (works with any ML)

5.2 Quiz 6: Computer Vision

1. **Transfer learning:** Uses pretrained models as starting point
2. ImageNet models trained on **1000 classes**
3. **Freezing:** Setting requires_grad=False
4. **ImageFolder** expects class subfolders
5. ImageNet input size: **224×224**
6. Standard normalization: mean=[0.485, 0.456, 0.406]
7. **Feature extraction:** Freeze backbone, train new head
8. MobileNetV3 optimized for **mobile/edge devices**
9. Learning rate schedulers adjust LR during training
10. Model surgery: freeze weights + replace classifier

5.3 Quiz 7: NLP Fundamentals

1. **Tokenization:** First step in NLP pipeline
2. Attention mask: 1=real token, 0=padding

CLS token used for **classification**

3. **Embeddings:** Dense vector representations
4. Weighted loss handles class imbalance
5. DistilBERT is a **distilled** (compressed) BERT
6. Fine-tuning adapts pretrained models to new tasks
7. Subword tokenization handles unknown words
8. Truncation handles sequences longer than max_length
9. **last_hidden_state** contains contextualized embeddings

5.4 Quiz 8: PyTorch Lightning

1. Lightning reduces boilerplate, provides best practices
2. **LightningModule** encapsulates model + training logic
3. `configure_optimizers()` returns optimizer (and scheduler)
4. **LightningDataModule** handles all data loading
5. `prepare_data()` for downloads, `setup()` for splits
6. EarlyStopping prevents overfitting
7. `self.log()` for automatic metric tracking
8. Trainer handles training loop, devices, precision
9. Callbacks extend functionality (checkpoints, early stopping)
10. `save_hyperparameters()` logs all __init__ args

6 Quick Reference

6.1 Optuna Template

```
import optuna

def objective(trial):
    # Suggest hyperparameters
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)
    model = build_model(trial)

    for epoch in range(num_epochs):
        val_acc = train_and_validate(model, lr)
        trial.report(val_acc, epoch)
        if trial.should_prune():
            raise optuna.TrialPruned()

    return val_acc

study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.MedianPruner()
)
study.optimize(objective, n_trials=100)
```

6.2 Transfer Learning Template

```
import torchvision.models as models

model = models.mobilenet_v3_small(weights='IMAGENET1K_V1')

# Freeze backbone
for param in model.features.parameters():
    param.requires_grad = False

# Replace classifier
model.classifier = nn.Sequential(
    nn.Linear(576, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, num_classes)
)
```

6.3 Lightning Template

```
class LitModel(pl.LightningModule):
    def __init__(self, num_classes, lr=1e-3):
        super().__init__()
        self.save_hyperparameters()
        self.model = ...
        self.criterion = nn.CrossEntropyLoss()

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

    def training_step(self, batch, batch_idx):
```

```
        x, y = batch
        loss = self.criterion(self(x), y)
        self.log('train_loss', loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(),
                                lr=self.hparams.lr)

trainer = pl.Trainer(max_epochs=10, accelerator='auto')
trainer.fit(model, datamodule)
```