# Fine-tuning for Classification

## From General LLMs to Specialized Classifiers

Prof. Rongyu Lin

Lecture 7

Quinnipiac University
School of Computing and Engineering

**Reading:** Raschka Ch. 6 (pp. 191-238)
"Transform pretrained models for specific tasks"

# Lecture Agenda

**Today's Journey: 75 minutes**

1. **Fine-tuning Categories & Dataset Prep** (20 min)
2. **Data Loaders & Model Modification** (19 min)
3. **Training Loop & Loss Optimization** (18 min)
4. **Model Evaluation & Applications** (18 min)

**Interactive elements:** 4 hands-on activities

*From general language model to specialized spam classifier*

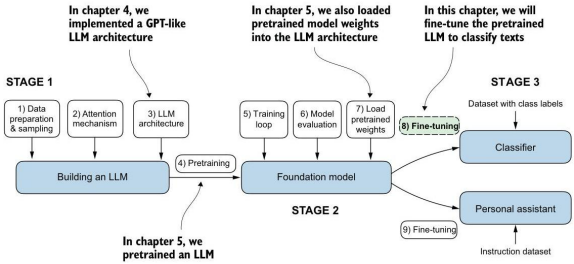
# Learning Objectives

**By the end of this lecture, you will be able to:**

- Distinguish instruction vs classification fine-tuning
- Prepare datasets for text classification tasks
- Modify pretrained architectures with classification heads
- Implement training loops for spam detection
- Evaluate models using accuracy and loss metrics
- Apply fine-tuned models to new data

*Master the art of adapting LLMs for specific tasks*

# Chapter 6 Overview - The Fine-tuning Pipeline




## The Transformation:

- **From:** General pretrained LLM (GPT-2)
- **To:** Specialized spam classifier
- **Key stages:** Dataset prep, model modification, training, evaluation
- **Focus:** Stage 3, Step 8 - Classification fine-tuning

*Three main LLM stages - we focus on classification fine-tuning*

# Two Categories of Fine-tuning

**Instruction Fine-tuning:**

- Trains model to follow natural language instructions
- **Use cases:** Versatile tasks
- **Examples:** Translations, summaries
- **Trade-off:** Flexibility

**Classification Fine-tuning:**

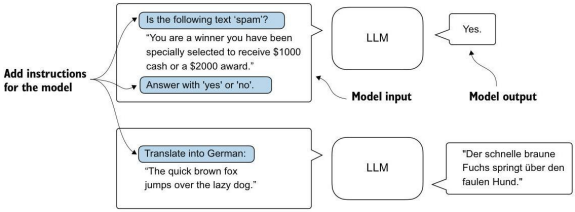
- Trains model to predict specific class labels
- **Use cases:** Specific domains
- **Examples:** Spam, sentiment
- **Trade-off:** Specialization

**Key Decision:** Flexibility vs Specialization

# Instruction Fine-tuning Characteristics

## Key Features:

- **Input:** Natural language instructions + context
- **Output:** Variable responses based on instruction type
- **Example:** "Determine if this text is spam: [text]"
- **Flexibility:** Can handle multiple task types
- **Requirements:** Larger datasets, more compute resources

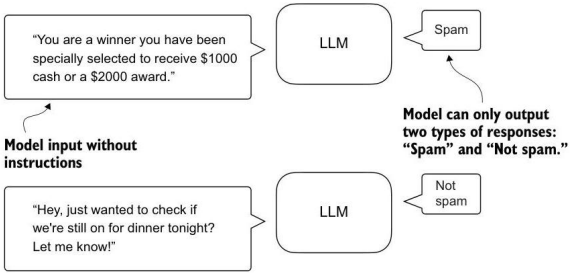



*Instruction fine-tuning scenarios with examples*

# Classification Fine-tuning Characteristics

## Key Features:

- **Input:** Raw text without explicit instructions
- **Output:** Predefined class labels (0, 1, 2, etc.)
- **Example:** Text $\rightarrow$ "spam" or "not spam"
- **Restriction:** Limited to trained classes only
- **Advantages:** Less data needed, faster training, higher accuracy for specific tasks




*Classification scenario for spam detection*

# Choosing the Right Approach

**Classification fine-tuning when:**

- Specific categorization needed
- Limited data available
- Fast inference required

**Instruction fine-tuning when:**

- Multiple tasks needed
- Flexible responses required
- Abundant resources available

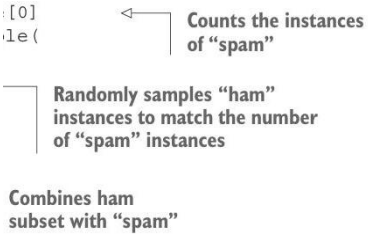
**Our choice:** Classification fine-tuning for spam detection

**Reasoning:** Clear binary task, efficient training, measurable performance

# Dataset Introduction - SMS Spam Collection

## Dataset Overview:

- **Source:** UCI Machine Learning Repository
- **Task:** Binary classification (spam vs ham)
- **Size:** 5,572 text messages
- **Format:** Tab-separated values (Label, Text)
- **Real-world relevance:** Practical application everyone understands




*Dataset preview in pandas DataFrame*

# Dataset Download and Setup

```python
import urllib.request
import zipfile
import pandas as pd

def download_sms_spam_dataset():
    url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
    zip_path = "sms_spam_collection.zip"
    extracted_path = "sms_spam_collection"
    data_file_path = "SMSSpamCollection"

    # Download and extract
    urllib.request.urlretrieve(url, zip_path)
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extracted_path)
```

**Download Process:**

- **URL:** UCI ML Repository
- **Process:** Download → Extract → Rename → Load

**Features:**

- Automatic extraction
- Path management
- Data verification

# Class Distribution Analysis

```
# Analyze class distribution
print(df["Label"].value_counts())
# Output:
# ham     4825
# spam     747
```

**The Imbalance Problem:**

- **Ham:** 4,825 messages
- **Spam:** 747 messages
- **Ratio:** 6.5:1 imbalance

**Impact:** Model bias toward majority class

**Our approach:** Undersample ham to match spam count

# Creating Balanced Dataset

```python
def create_balanced_dataset(df):
    # Separate classes
    spam_df = df[df["Label"] == "spam"]
    ham_df = df[df["Label"] == "ham"]

    # Undersample ham to match spam count
    ham_subset = ham_df.sample(
        n=len(spam_df),
        random_state=123
    )

    # Combine
    balanced_df = pd.concat([ham_subset, spam_df])
    return balanced_df.sample(
        frac=1,
        random_state=123
    ).reset_index(drop=True)
```

**Balancing Strategy:**

- **Method:** Random undersampling
- **Target:** 747 examples each class
- **Random seed:** 123 for reproducibility

**Steps:**

- Separate classes
- Sample ham to match spam
- Combine and shuffle

**Result:** 1,494 total examples (747 + 747)

# Label Encoding

```python
# Convert string labels to integers
balanced_df["Label"] = balanced_df["Label"].map({
    "ham": 0,
    "spam": 1
})

# Verify conversion
print(balanced_df["Label"].value_counts())
# Output:
# 0    747  # ham
# 1    747  # spam
```

**Converting Text Labels:**

- **Conversion:** String $\rightarrow$ Integer
- **Mapping:** "ham" $\rightarrow$ 0, "spam" $\rightarrow$ 1
- **Reason:** Neural networks require numerical inputs

**Analogy:** Similar to text tokenization but simpler vocabulary

# Dataset Splitting Strategy

```python
def random_split(df, train_frac, validation_frac):
    # Shuffle dataset
    df = df.sample(
        frac=1,
        random_state=123
    ).reset_index(drop=True)

    # Calculate split points
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(
        len(df) * validation_frac
    )

    # Split
    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df
```

**Three-way Split:**

- **Training:** 70% (1,046 examples)
- **Validation:** 10% (149 examples)
- **Test:** 20% (299 examples)

**Process:**

- Shuffle data randomly
- Calculate split points
- Create separate datasets

**Key:** Stratification maintains class balance

# Hands-on Activity (5 minutes)

## Explore the SMS Spam Dataset

**Task:** Load dataset and examine class distribution
**Goal:** Understand data characteristics and imbalance issues

**Steps:**

1. Download → Load → Analyze → Balance → Split

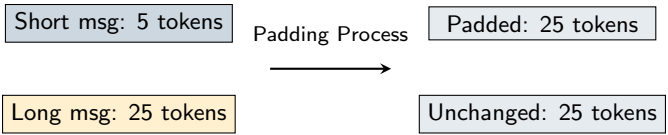
**Questions to explore:**

- What makes a message "spam"?
- How severe is the imbalance?
- What are real-world implications of data imbalance?

# Data Preprocessing Challenges

**The Variable Length Problem:**

- **Variable length:** Messages have different token counts
- **Batching requirement:** Neural networks need fixed-size inputs
- **Solutions:** Truncation vs Padding
- **Our choice:** Padding (preserves information)
- **Padding token:** `<|endoftext|>` (token ID 50256)



# From Dataset to Training

**Completed Steps:**

- ✓ Data download and extraction
- ✓ Class balance correction
- ✓ Dataset splitting
- ✓ Label encoding

**File output:** train.csv, validation.csv, test.csv

**Next step:** Create PyTorch data loaders for batching

*Clean, balanced dataset ready for model training*

# PyTorch Data Loaders Overview

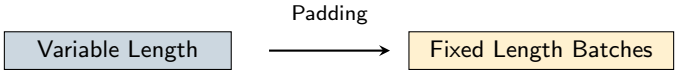
**Purpose:** Efficiently load and batch text data

**The Challenge:** Variable-length text messages

**Approaches:**

- **Previous approach:** Sliding window for uniform chunks
- **Current approach:** Padding to longest sequence length

**Benefits:** Preserve all information, enable batching



# Text Length Handling Options

### Option 1: Truncation

- Truncate all to shortest message length
- **Pro:** Fast processing
- **Con:** Information loss

### Option 2: Padding

- Pad all to longest message length
- **Pro:** Information preservation
- **Con:** More memory usage

**Our choice:** Padding (Option 2)
**Rationale:** Avoid information loss, better performance

# Padding Token Strategy

```python
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
token_id = tokenizer.encode(
    "<|endoftext|>",
    allowed_special={"<|endoftext|>"}
)[0]
print(f"Padding token ID: {token_id}")
# Output: Padding token ID: 50256
```

```python
def random_split(df, train_frac, validation_frac):
    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1)
```

Shuffles the entire DataFrame

Calculates split indices

Splits the DataFrame

Test size is implied to be 0.2 as the remainder.

**Token Details:**

- **Token choice:** <|endoftext|>
- **Token ID:** 50256 (verify with tiktoken)
- **Implementation:** Add token IDs, not text strings

**Process:**

- Import tiktoken
- Get GPT-2 encoding
- Extract token ID
- Verify padding token

*Input preparation with tokenization and padding*

# SpamDataset Class Design

```python
class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer,
                 max_length=None):
        self.data = pd.read_csv(csv_file)
        self.tokenizer = tokenizer

        # Auto-detect max length if not specified
        if max_length is None:
            token_lengths = []
            for text in self.data["Text"]:
                token_lengths.append(
                    len(tokenizer.encode(text))
                )
            self.max_length = max(token_lengths)
        else:
            self.max_length = max_length
```

**PyTorch Dataset Implementation:**

- **Inheritance:** PyTorch Dataset class
- **Key functions:** `__init__`, `__getitem__`, `__len__`
- **Responsibilities:** Load data, tokenize, pad sequences

**Features:**

- Auto-detect max length
- User-specified option
- Efficient data loading

# SpamDataset Implementation Details

```python
def __getitem__(self, index):
    text = self.data.iloc[index]["Text"]
    label = self.data.iloc[index]["Label"]

    # Tokenize text
    token_ids = self.tokenizer.encode(text)

    # Truncate if necessary
    token_ids = token_ids[:self.max_length]

    # Pad with endoftext token (ID: 50256)
    token_ids += [50256] * (
        self.max_length - len(token_ids)
    )

    return (
        torch.tensor(token_ids, dtype=torch.long),
        torch.tensor(label, dtype=torch.long)
    )

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

**Core Methods:**

- **__getitem__:** Process single data point
- **__len__:** Return dataset size

**Process:**

- Tokenize text
- Truncate if needed
- Pad to max length
- Return tensors

**Output:** Token IDs and labels as PyTorch tensors

# Dataset Instantiation

```python
import tiktoken

# Initialize tokenizer
tokenizer = tiktoken.get_encoding("gpt2")

# Create training dataset (auto-detect max length)
train_dataset = SpamDataset(
    "train.csv",
    tokenizer
)
print(f"Max length detected: {train_dataset.max_length}")
# Output: Max length detected: 120

# Create validation and test datasets
val_dataset = SpamDataset(
    "validation.csv",
    tokenizer,
    max_length=train_dataset.max_length
)
test_dataset = SpamDataset(
    "test.csv",
    tokenizer,
    max_length=train_dataset.max_length
)
```

**Creating Dataset Objects:**

- Import tiktoken tokenizer
- Create training dataset first
- Auto-detect max length
- Use same length for val/test

**Consistency:**

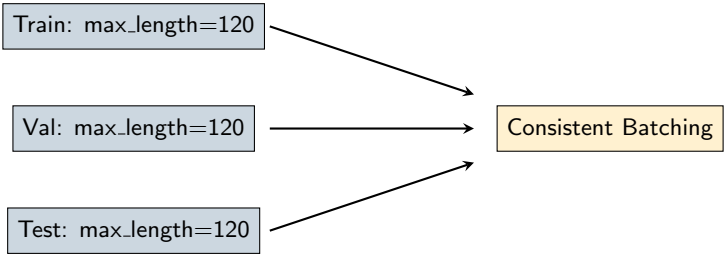
- Same tokenizer for all splits
- Same max length (120 tokens)
- Proper data alignment

**Key insight:** Model supports 1,024 tokens - messages well within limit

# Validation and Test Datasets

**Consistency Requirements:**

- **Length matching:** Use training dataset's max_length
- **Consistency:** All datasets padded to same length (120)
- **Truncation:** Automatic for sequences $>$ max_length
- **Alternative:** Set max_length=None if no long sequences

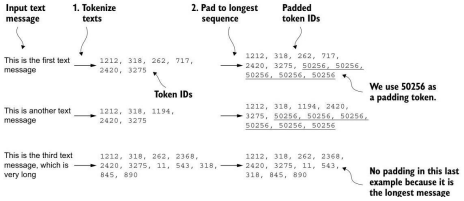


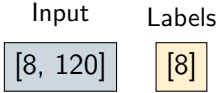

**Result:** All datasets compatible for training and evaluation

# Batch Structure Visualization

## Batch Configuration:

- **Batch size:** 8 training examples
- **Input tensor:** [8, 120] - 8 messages of 120 tokens
- **Label tensor:** [8] - corresponding class labels
- **Padding:** Token ID 50256 fills shorter sequences




*Batch structure with 8 messages and labels*

# Data Loader Configuration

```python
from torch.utils.data import DataLoader

# Training data loader (with shuffling)
train_loader = DataLoader(
    train_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0,
    drop_last=True
)

# Validation data loader (no shuffling)
val_loader = DataLoader(
    val_dataset,
    batch_size=8,
    shuffle=False,
    num_workers=0,
    drop_last=False
)
```

**DataLoader Setup:**

- **Batch size:** 8 examples per batch
- **Training:** Shuffle=True, drop_last=True
- **Validation:** Shuffle=False, drop_last=False

**Key Settings:**

- Shuffling prevents overfitting
- Drop last ensures consistent batch sizes
- No multiprocessing (num_workers=0)

# Data Loader Verification

```python
# Check data loader shapes
print("Checking data loader shapes...")
for i, (inputs, targets) in enumerate(train_loader):
    print(f"Batch {i+1}:")
    print(f"  Input shape: {inputs.shape}")
    # [8, 120]
    print(f"  Target shape: {targets.shape}")
    # [8]
    if i == 2:  # Show first 3 batches
        break

# Count total batches
print(f"Training batches: {len(train_loader)}")
# 130 batches
print(f"Validation batches: {len(val_loader)}")
# 19 batches
print(f"Test batches: {len(test_loader)}")
# 38 batches
```

**Testing Our Setup:**

- Check input/target shapes
- Verify batch dimensions
- Count total batches

**Expected Results:**

- Input shape: [8, 120]
- Target shape: [8]
- Training: 130 batches
- Validation: 19 batches
- Test: 38 batches

**Verification:** Shapes correct, counts match expectations

## Create and Test Data Loaders

**Task:** Implement SpamDataset and create DataLoaders
**Goal:** Understand batching and padding mechanics

**Steps:**

1. Create dataset → Set up loaders → Check batch shapes

**Exploration points:**

- Examine padded sequences
- Verify token IDs
- Understand: Why padding vs truncation?

# Model Architecture Preparation

**Starting Point:**

- **Base model:** Pretrained GPT-2 model (124M parameters)
- **Configuration:** Same as pretraining setup
- **Model loading:** Import weights from OpenAI
- **Verification:** Generate coherent text to confirm loading
- **Next step:** Modify for classification


*Transform general language model into specialized classifier*

# GPT-2 Model Configuration

```python
BASE_CONFIG = {
    "vocab_size": 50257,
    "context_length": 1024,
    "drop_rate": 0.0,
    "qkv_bias": True
}

model_configs = {
    "gpt2-small (124M)": {
        "emb_dim": 768,
        "n_layers": 12,
        "n_heads": 12
    },
    # Other configurations...
}
```

**Model Specifications:**

- **Size:** gpt2-small (124M parameters)
- **Vocabulary:** 50,257 tokens
- **Context length:** 1,024 tokens
- **Architecture:** 12 layers, 768 embedding dimensions

**Configuration:**

- Base config with GPT-2 specs
- Model-specific dimensions
- Exact match required

# Loading Pretrained Weights

```python
def download_and_load_gpt2(model_size, models_dir):
    # Download weights from OpenAI
    model = GPTModel(BASE_CONFIG)

    # Load pretrained weights
    load_weights_into_gpt(
        model,
        model_size,
        models_dir
    )
    model.eval()

    # Test generation to verify loading
    text = generate(
        model=model,
        tokenizer=tokenizer,
        start_context="Every effort moves you",
        max_new_tokens=25
    )
    print("Generated text:", text)
    return model
```

**Weight Loading Process:**

- Create model with base config
- Download OpenAI weights
- Load into model architecture
- Set to evaluation mode

**Verification:**

- Test text generation
- Check coherent output
- Confirm successful loading

# Pre-fine-tuning Instruction Test

```python
# Test if model can follow classification instructions
test_prompt = """Is the following text 'spam'?
Answer with 'yes' or 'no':

'Congratulations! You have won a $1000 cash prize.
Call now to claim your reward!'

Answer:"""

response = generate(
    model=model,
    tokenizer=tokenizer,
    start_context=test_prompt,
    max_new_tokens=10
)
print("Model response:", response)
# Output: Incoherent, doesn't follow instructions
```

**Testing Instruction Following:**

- Create classification prompt
- Test with clear spam example
- Generate model response
- Analyze instruction following

**Expected Result:**

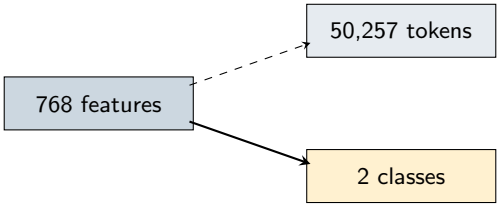
- Incoherent output
- Doesn't follow instructions
- Normal for base GPT-2

**Conclusion:** Pretrained model needs fine-tuning

# Classification Head Concept

**Architecture Modification:**

- **Original:** 768 $\rightarrow$ 50,257 (vocabulary size)
- **New:** 768 $\rightarrow$ 2 (binary classification)
- **Rationale:** Match number of output classes
- **Generalization:** n classes $\rightarrow$ n output nodes




Replace with classification head

# Layer Freezing Strategy

```python
# Freeze most layers
for param in model.parameters():
    param.requires_grad = False

# Unfreeze last transformer block
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True

# Unfreeze final layer norm
for param in model.final_norm.parameters():
    param.requires_grad = True
```

**Selective Training Approach:**

- **Frozen:** Embedding + first 11 blocks
- **Trainable:** Final block + LayerNorm + classification head

**Benefits:**

- Faster training
- Prevent overfitting
- Preserve pretrained knowledge
- Fewer parameters to train

# Classification Head Implementation

```python
# Replace output head with classification layer
model.out_head = torch.nn.Linear(
    in_features=768,  # GPT embedding dimension
    out_features=2    # Binary classification
)

# Classification head is trainable by default
print("New layer requires_grad:",
      model.out_head.weight.requires_grad)
# Output: True

# Verify architecture changes
print("Model output shape test:")
test_input = torch.randint(0, 50257, (1, 4))
output = model(test_input)
print(f"Output shape: {output.shape}")
# [1, 4, 2]
```

**Adding Classification Layer:**

- Replace output head
- 768 features $\rightarrow$ 2 classes
- Trainable by default
- Test architecture changes

**Shape Changes:**

- Original: [batch, seq, 50257]
- New: [batch, seq, 2]
- Binary classification ready

# Modified Model Behavior

**Processing Pipeline:**

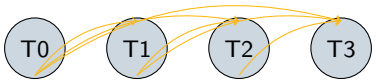
- **Input processing:** Same as before (tokenization)
- **Output shape:** Changed from [1, 4, 50257] to [1, 4, 2]
- **Key insight:** Focus on last output token only
- **Reasoning:** Causal attention mask makes last token most informed

**Example:** "Do you have time" $\rightarrow$ final token represents full sequence

# Last Token Focus Strategy

## Attention Flow Logic:

- **Attention mechanism:** Each token attends to previous tokens
- **Information flow:** Final token has access to entire sequence
- **Classification decision:** Based on last token's representation
- **Code:** `outputs[:, -1, :]` extracts last token




Final token sees everything

*Visual representation of attention flow to final token*

# From Logits to Predictions

```python
# Example logits for a sample
logits = torch.tensor([[-3.5983, 3.9902]])
# Shape: [1, 2]

# Get class prediction
predicted_class = torch.argmax(logits, dim=-1)
print(f"Predicted class: {predicted_class.item()}")
# Output: 1 (spam)

# Get probability scores
probabilities = torch.softmax(logits, dim=-1)
print(f"Probabilities: {probabilities}")
# Output: [0.0003, 0.9997]
# 99.97% spam confidence
```

**Processing Output Logits:**

- **Raw output:** Two logits per classification
- **Interpretation:** Higher value = predicted class
- **Conversion:** `torch.argmax` for prediction

**Example:**

- Logits: [-3.5983, 3.9902]
- Predicted: class 1 (spam)
- Confidence: 99.97%

# Classification Accuracy Function

```python
def calc_accuracy_loader(data_loader, model, device,
      num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))

    for i, (input_batch, target_batch) in
            enumerate(data_loader):
        if i < num_batches:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[:, -1, :]
                predicted_labels = torch.argmax(logits, dim=-1)
                num_examples += predicted_labels.shape[0]
                correct_predictions += (predicted_labels ==
  target_batch).sum().item()
        else:
            break
    return correct_predictions / num_examples
```

**Measuring Performance:**

- Set model to evaluation mode
- Process batches with no_grad
- Extract last token logits
- Compare predictions to targets

**Formula:**

- correct_predictions / total_examples
- Returns decimal (0.0 to 1.0)
- Multiply by 100 for percentage

# Initial Model Performance

```python
# Test initial performance
train_accuracy = calc_accuracy_loader(
    train_loader, model, device
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
# 46.25%
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
# 45.00%
print(f"Test accuracy: {test_accuracy*100:.2f}%")
# 48.75%
```

**Before Fine-tuning Results:**

- **Training:** 46.25%
- **Validation:** 45.00%
- **Test:** 48.75%

**Analysis:**

- Near random performance
- Classification head has random weights
- Model needs training

**Random baseline:** 50% for binary classification

# Classification Loss Function

```python
def calc_loss_batch(input_batch, target_batch,
                    model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)

    # Forward pass
    logits = model(input_batch)[:, -1, :]
    # Focus on last token

    # Calculate cross-entropy loss
    loss = torch.nn.functional.cross_entropy(
        logits, target_batch
    )
    return loss

# Example usage
sample_batch = next(iter(train_loader))
inputs, targets = sample_batch
loss = calc_loss_batch(inputs, targets,
                       model, device)
print(f"Sample batch loss: {loss.item():.4f}")
```

**Cross-entropy Loss:**

- Move inputs to device
- Forward pass through model
- Extract last token logits
- Calculate cross-entropy

**Key Points:**

- Differentiable proxy for accuracy
- Focus on last token only
- Standard for classification

# Loss Calculation Implementation

```python
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))

    for i, (input_batch, target_batch) in
            enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch,
                model, device)
            total_loss += loss.item()
        else:
            break

    return total_loss / num_batches

# Initial losses (before training)
train_loss = calc_loss_loader(train_loader, model, device, num_batches
      =5)
val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
test_loss = calc_loss_loader(test_loader, model, device, num_batches
      =5)

print(f"Training loss: {train_loss:.3f}")  # 2.453
print(f"Validation loss: {val_loss:.3f}")  # 2.583
print(f"Test loss: {test_loss:.3f}")       # 2.322
```

**Dataset-wide Loss:**

- Iterate through data loader
- Calculate loss per batch
- Accumulate total loss
- Return average loss

**Initial Results:**

- Training: 2.453
- Validation: 2.583
- Test: 2.322

**High values:** Expected for untrained classification head

## Implement Classification Metrics

**Task:** Code accuracy and loss calculation functions
**Goal:** Understand evaluation metrics for classification

**Steps:**

1. Implement functions $\rightarrow$ Test on sample batch $\rightarrow$ Compare results

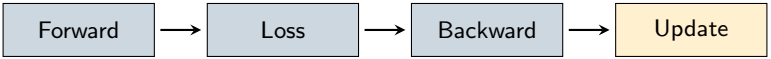
**Analysis questions:**

- Why are initial accuracies near random?
- Loss vs accuracy as optimization targets?

# Training Loop Overview

**Training Structure:**

- **Similar to:** Pretraining but with classification focus
- **Key differences:** Track examples (not tokens), calculate accuracy
- **Process:** Forward pass → Loss calculation → Backpropagation → Update
- **Monitoring:** Loss and accuracy after each epoch




*PyTorch training loop for classification*

# Fine-tuning Function Implementation

```python
def train_classifier_simple(model, train_loader, val_loader, optimizer
    , device, num_epochs, eval_freq, eval_iter):
    # Tracking lists
    train_losses, val_losses = [], []
    train_accs, val_accs = [], []
    examples_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch,
                model, device)
            loss.backward()
            optimizer.step()

            examples_seen += input_batch.shape[0]
            global_step += 1

            # Periodic evaluation
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(model,
  train_loader,
                    val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
```

**Training Function Structure:**

- Initialize tracking lists
- Loop through epochs
- Process each batch
- Calculate loss and gradients

**Key Steps:**

- Zero gradients
- Forward pass
- Backward pass
- Update parameters
- Periodic evaluation

# Optimizer and Hyperparameters

```python
import torch.optim as optim

# Optimizer setup
optimizer = optim.AdamW(
    model.parameters(),
    lr=5e-5,          # Learning rate
    weight_decay=0.1 # Regularization
)

# Training hyperparameters
num_epochs = 5    # Sufficient for fine-tuning
batch_size = 8    # Memory efficient
eval_freq = 50    # Evaluate every 50 steps
eval_iter = 1     # Use 1 batch for evaluation
```

**Training Configuration:**

- **Optimizer:** AdamW
- **Learning Rate:** 5e-5
- **Weight Decay:** 0.1
- **Epochs:** 5

**Key Choices:**

- AdamW: Better weight decay
- Small LR: Fine-tuning
- Regularization: Prevent overfitting
- Few epochs: Sufficient

# Training Execution and Monitoring

```python
import time
start_time = time.time()

train_losses, val_losses, train_accs, val_accs,
examples_seen = train_classifier_simple(
    model, train_loader, val_loader,
    optimizer, device,
    num_epochs=5, eval_freq=50, eval_iter=1
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in "
      f"{execution_time_minutes:.2f} minutes")

# Example output during training:
# Ep 1 (Step 000050): Train loss 1.793,
#                     Val loss 1.623
# Ep 1 (Step 000100): Train loss 0.969,
#                     Val loss 0.765
# Ep 1 (Step 000130): Train loss 0.441,
#                     Val loss 0.327
```

**Running the Training:**

- Track execution time
- Call training function
- Monitor progress output
- Observe loss decreases

**Performance:**

- 6 minutes on laptop
- $<$30 seconds on GPU
- Rapid improvement

**Early signs:** Rapid improvement in first epoch

# Training Results Analysis

## Epoch-by-Epoch Progress:

| Epoch | Train Accuracy | Validation Accuracy |
|---|---|---|
| 1 | 70.0% | 72.5% |
| 2 | 82.5% | 85.0% |
| 3 | 90.0% | 90.0% |
| 4 | 95.0% | 95.0% |
| 5 | 100.0% | 97.5% |

## Pattern Analysis:

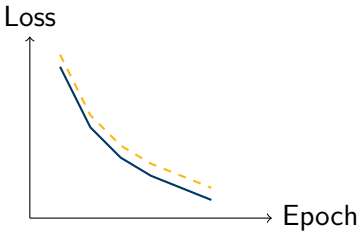
- **Rapid learning:** Dramatic improvement in epoch 1
- **Good generalization:** Small gap between train/validation
- **Convergence:** Gradual stabilization by epoch 5

**Success:** Excellent performance achieved quickly

# Loss Curve Visualization

**Loss Progression:**

- **Training loss:** Rapid decline from 2.15 to 0.08
- **Validation loss:** Parallel decline, slightly higher
- **Pattern:** Sharp drop in epoch 1, gradual stabilization
- **Interpretation:** Good learning without overfitting




Training and validation loss curves

# Overfitting Analysis

**Signs to Watch:**

- **Warning sign:** Training accuracy >> Validation accuracy
- **Current status:** Small gap (100% vs 97.5%)
- **Validation loss:** Continues decreasing (good sign)
- **Conclusion:** Model generalizes well to unseen data

**Prevention Strategies:**

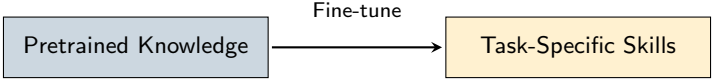
- Early stopping based on validation performance
- Regularization (weight decay, dropout)
- Reduce model complexity
- Increase training data

*Good learning pattern - minimal overfitting detected*

# Training Convergence Insights

## Why Fine-tuning Works So Well:

- **Rapid learning:** Benefits of pretrained features
- **Transfer learning:** Model adapts existing knowledge
- **Efficiency:** Much faster than training from scratch
- **Quality:** High performance with minimal training
- **Foundation:** Pretrained representations accelerate learning




Transfer linguistic understanding to classification

*Standing on the shoulders of giants*

# Training to Evaluation Transition

**Training Achievements:**

- ✓ High-performing spam classifier created
- ✓ Training complete: 100% training, 97.5% validation accuracy
- ✓ Good convergence with minimal overfitting
- ✓ Understanding of fine-tuning process

**Next step:** Comprehensive evaluation on test set

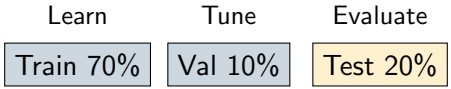
**Applications:** Deploy model for real spam detection

*From general language model to specialized classifier*

# Final Model Evaluation

**Test Set Assessment:**

- **Purpose:** Unbiased performance estimate
- **Process:** Load best model, evaluate on test set
- **Metrics:** Accuracy, loss, confusion matrix
- **Expected:** Similar to validation performance ( 97%)
- **Importance:** Real-world performance indicator




*Test set provides final, unbiased performance estimate*

# Test Set Results

```python
# Final evaluation on test set
test_accuracy = calc_accuracy_loader(
    test_loader, model, device
)
test_loss = calc_loss_loader(
    test_loader, model, device
)

print(f"Test accuracy: {test_accuracy*100:.2f}%")
# ~97%
print(f"Test loss: {test_loss:.3f}")
# Low, consistent

# Compare with initial performance
print("\nPerformance comparison:")
print(f"Initial test accuracy: 48.75%")
print(f"Final test accuracy: {test_accuracy*100:.2f}%")
print(f"Improvement: "
      f"{test_accuracy*100 - 48.75:.2f} "
      f"percentage points")
```

**Final Performance Metrics:**

- Evaluate on test set
- Calculate accuracy and loss
- Compare to initial performance

**Results:**

- **Final:** 97% accuracy
- **Initial:** 48.75% accuracy
- **Improvement:** 48 points

**Conclusion:** Model generalizes well to unseen data

# Model Inference on New Data

```python
def classify_review(text, model, tokenizer, device, max_length=None):
    model.eval()

    # Tokenize and prepare input
    token_ids = tokenizer.encode(text)
    if max_length:
        token_ids = token_ids[:max_length]
        token_ids += [50256] * (max_length - len(token_ids))  # Pad

    # Convert to tensor and add batch dimension
    input_tensor = torch.tensor(token_ids, dtype=torch.long).unsqueeze
      (0).to(device)

    # Get model prediction
    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
        probabilities = torch.softmax(logits, dim=-1)
        predicted_label = torch.argmax(logits, dim=-1).item()

    return predicted_label, probabilities.cpu().numpy()[0]

# Example usage
text = "Free money! Click now to claim your prize!"
label, probs = classify_review(
    text, model, tokenizer, device, max_length=120)
print(f"Prediction: {'Spam' if label == 1 else 'Ham'}")
print(f"Confidence: Ham={probs[0]:.3f}, "
      f"Spam={probs[1]:.3f}")
```

**Inference Pipeline:**

- Set model to eval mode
- Tokenize input text
- Apply padding if needed
- Convert to tensor
- Get predictions

**Output:**

- Predicted label (0 or 1)
- Class probabilities
- Human-readable result

# Spam Detection Examples

```python
# Test examples
examples = [
    "Free money! Click now!",          # Clear spam
    "Meeting at 3pm today",            # Clear ham
    "Win a free prize today!",         # Borderline
    "Your package will arrive tomorrow", # Ham
    "URGENT: Claim your reward now!!!"   # Clear spam
]

for text in examples:
    label, probs = classify_review(
        text, model, tokenizer,
        device, max_length=120
    )
    result = "Spam" if label == 1 else "Ham"
    confidence = probs[label]
    print(f"'{text}' -> {result} "
          f"({confidence:.3f} confidence)")

# Sample output:
# 'Free money! Click now!' -> Spam
#   (0.987 confidence)
# 'Meeting at 3pm today' -> Ham
#   (0.923 confidence)
# 'Win a free prize today!' -> Spam
#   (0.756 confidence)
```

**Testing Real Examples:**

- Clear spam examples
- Clear ham examples
- Borderline cases
- Confidence scores

**Analysis:**

- Model learns spam indicators
- Recognizes urgency words
- Handles free/win/prize patterns

**Robustness:** Test edge cases and borderline examples

## Test Your Own Messages

**Task:** Input custom text messages for classification
**Goal:** Understand model decision-making process

**Examples to try:**

- Create obvious spam messages
- Create obvious ham messages
- Test borderline cases

**Analysis questions:**

- What patterns does the model recognize?
- How would you deploy this in real-world scenarios?

# Model Limitations and Considerations

**Known Limitations:**

- **Training data:** Limited to SMS-style messages
- **Domain adaptation:** May not work well on emails/other text
- **Adversarial examples:** Deliberately crafted to fool model
- **Bias:** Reflects patterns in training data
- **Updates:** Need retraining as spam tactics evolve

**Mitigation Strategies:**

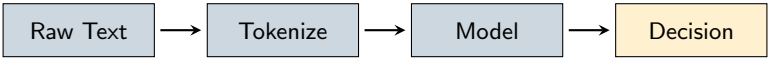
- Regular model updates with new data
- Domain adaptation techniques
- Adversarial training
- Bias detection and correction
- Human-in-the-loop validation

*Understanding limitations is key to responsible deployment*

# Deployment Architecture

**Production Pipeline:**

- **Input processing:** Text → tokens → padding → model
- **Model serving:** Load trained weights, inference mode
- **Output processing:** Logits → probabilities → decisions
- **Scalability:** Batch processing for efficiency
- **Integration:** API endpoints, real-time vs batch processing




End-to-end inference pipeline

# Performance Optimization

## Optimization Strategies:

- **Model size:** Trade-offs between accuracy and speed
- **Quantization:** Reduce model precision for faster inference
- **Caching:** Store common preprocessing results
- **GPU acceleration:** Batch processing on GPU
- **Edge deployment:** Optimize for mobile/embedded devices

## Performance Metrics:

- Inference latency (milliseconds per prediction)
- Throughput (predictions per second)
- Memory usage (GPU/CPU requirements)
- Model size (storage requirements)
- Energy consumption (for edge devices)

*Balance accuracy, speed, and resource requirements*

# Alternative Fine-tuning Strategies

**Beyond Basic Freezing:**

- **Full model fine-tuning:** Train all layers (more compute, potentially better)
- **Layer selection:** Experiment with different layer combinations
- **Learning rates:** Different rates for different layers
- **Gradual unfreezing:** Progressive layer training
- **LoRA/Adapters:** Parameter-efficient fine-tuning methods

**Parameter-Efficient Methods:**

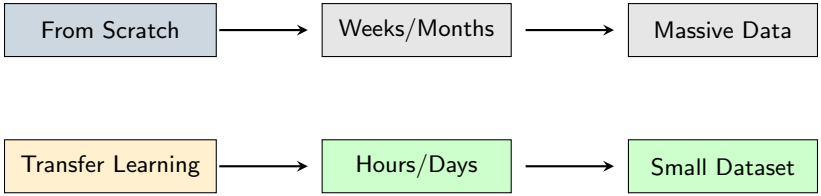
- Low-Rank Adaptation (LoRA)
- Adapter layers
- Prefix tuning
- Prompt tuning

*Choose strategy based on data, compute, and performance requirements*

# Transfer Learning Benefits

## Why Transfer Learning Works:

- **Speed:** Much faster than training from scratch
- **Data efficiency:** Good performance with less data
- **Performance:** Often better than random initialization
- **Knowledge reuse:** Leverage pretrained linguistic understanding
- **Cost:** Reduced computational requirements




*Transfer learning democratizes access to powerful models*

# Beyond Binary Classification

**Extension Possibilities:**

- **Multi-class:** Extend to 3+ categories (news topics, sentiment)
- **Hierarchical:** Category and subcategory prediction
- **Multi-label:** Multiple categories per text
- **Regression:** Predict continuous values (ratings, scores)
- **Sequence labeling:** Token-level classification (NER, POS)

**Implementation Changes:**

- Change output layer size (2 $\rightarrow$ n classes)
- Modify loss function (multi-class, multi-label)
- Adjust evaluation metrics
- Update data preprocessing

*Same principles apply to many classification tasks*

# Comparison with Other Approaches

## Alternative Methods:

| Approach | Accuracy | Training Time | Data Needed |
|---|---|---|---|
| Traditional ML (SVM, NB) | Medium | Fast | Medium |
| CNN/LSTM from scratch | Medium | Slow | Large |
| Word2Vec + Classifier | Good | Medium | Medium |
| BERT Fine-tuning | Excellent | Medium | Small |
| GPT-2 + Classification | Excellent | Fast | Small |

## Our approach advantages:

- Excellent performance with minimal data
- Fast training due to frozen layers
- Leverages powerful pretrained representations
- Simple implementation

**Trade-offs:** Model size vs traditional ML approaches

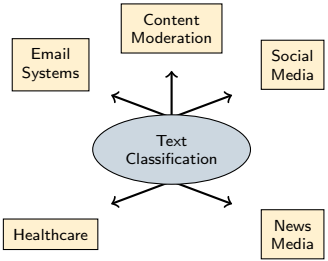
# Real-world Applications

## Industry Use Cases:

- **Email filtering:** Spam detection in email systems
- **Content moderation:** Identify inappropriate content
- **Sentiment analysis:** Product reviews, social media
- **News categorization:** Automatic article classification
- **Medical text:** Clinical note classification

## Success Stories:

- Gmail spam filtering (99.9% accuracy)
- Facebook content moderation
- Amazon review sentiment analysis
- Reuters news categorization
- Healthcare documentation processing



# Evaluation Metrics Deep Dive

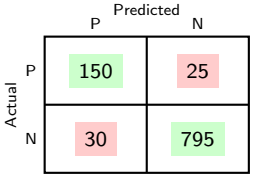
## Beyond Accuracy:

- **Accuracy:** Overall correctness percentage
- **Precision:** True positives / (True positives + False positives)
- **Recall:** True positives / (True positives + False negatives)
- **F1-score:** Harmonic mean of precision and recall
- **Confusion matrix:** Detailed breakdown of predictions

## When to Use Each:

- **Accuracy:** Balanced datasets, general performance
- **Precision:** When false positives are costly
- **Recall:** When false negatives are costly
- **F1-score:** Imbalanced datasets, overall balance

**Confusion Matrix**




Precision = 83.3%

Recall = 85.7%

Accuracy = 94.5%

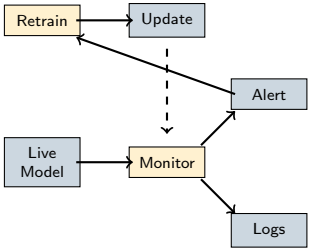
# Model Monitoring and Maintenance

## Production Monitoring:

- **Performance tracking:** Monitor accuracy over time
- **Data drift:** Detect changes in input distribution
- **Model updates:** Regular retraining schedules
- **A/B testing:** Compare model versions
- **Feedback loops:** Incorporate user corrections

## Key Metrics to Track:

- Prediction accuracy trends
- Response time/latency
- Input data distribution changes
- User feedback and corrections
- Model confidence scores




Continuous Improvement Cycle

# Chapter 6 Summary & Key Takeaways

**What We Accomplished:**

- **Fine-tuning categories:** Classification vs instruction fine-tuning
- **Dataset preparation:** Balance, split, tokenize, pad
- **Architecture modification:** Replace output layer, freeze layers
- **Training process:** Cross-entropy loss, accuracy monitoring
- **Evaluation:** Test set performance, real-world applications
- **Success:** 97% accuracy on spam detection task

**Key Insights:**

- Transfer learning dramatically reduces training requirements
- Frozen layers preserve pretrained knowledge
- Classification fine-tuning is efficient and effective
- Real-world deployment requires careful consideration

*From general language model to specialized classifier in 75 minutes*

# Looking Ahead - Next Steps

**Upcoming Topics:**

- **Chapter 7:** Instruction fine-tuning with human feedback
- **Advanced topics:** Parameter-efficient fine-tuning (LoRA, Adapters)
- **Scaling:** Larger models, more complex tasks
- **Production:** Deployment, monitoring, maintenance
- **Your journey:** From pretraining through specialized applications

**Skills Gained Today:**

- Dataset preparation for classification
- Model architecture modification
- Training loop implementation
- Performance evaluation and analysis

## Questions & Discussion