```
#####################################################################
#                                   #
# CS435 Generative AI: Security, Ethics and Governance      #
#                                   #
# Instructor: Dr. Adnan Masood                  #
# Contact:   adnanmasood@gmail.com                  #
#                                   #
# Notebook is MIT Licensed                  #
#####################################################################
```

LLM Instruction Tuning vs. Fine-Tuning

Welcome to our in-depth Jupyter Notebook on the difference between **Instruction Tuning** and **Fine-Tuning** in Large Language Models (LLMs). We will explore these concepts at multiple levels of detail, from a very simple explanation for beginners to a more advanced, PhD-level discussion.

We'll use **PyTorch** for our examples and code demonstrations.

Table of Contents

Building an Intuitive Understanding

Goal: Explain the concept of instruction tuning vs. fine-tuning in the simplest possible way (like for a middle schooler).

- **Fine-Tuning**: Think of it like teaching a dog new tricks by showing it examples over and over so it learns exactly how to do them.
- **Instruction Tuning**: This is like giving the dog simple instructions ("Sit!" "Roll Over!") and the dog learns to follow many different instructions without being shown examples for each trick hundreds of times. You just guide it on how to follow instructions.

That's the basic idea: **Fine-tuning** = training with many examples of a specific task. **Instruction tuning** = teaching the model to understand and follow instructions in general.

Goal: Provide a more detailed explanation (like for high school students).

- **Fine-Tuning**: You have a big language model that already understands English text. You give it a bunch of examples for a particular task (like summarizing paragraphs or classifying emails) and let the model adjust its internal parameters (weights) so it gets better and better at that specific task.
- **Instruction Tuning**: Instead of focusing on just one task, you teach the model to follow instructions in a general sense. You show it examples of many different instructions and how they should be answered. This way, it becomes more flexible; it can handle all sorts of different tasks simply by reading your instructions.

Goal: Dive deeper into the difference and talk about training strategies.

- **Fine-Tuning**:
  - You have a pre-trained LLM, such as GPT-like architecture.
  - You collect a large dataset **specific** to the task you care about.
  - You continue training ("fine-tuning") on this dataset.
  - This modifies the model's weights to make it better for that single or narrow set of tasks.
  - End result: The model is specialized for your particular problem.
- **Instruction Tuning**:

    - You gather many examples of *instructions and correct answers* (spanning multiple tasks).
    - The model sees how instructions are structured and how to respond.
    - The focus is on teaching the model a general strategy to interpret any new instruction.
    - End result: The model is more robust and can handle a variety of tasks when prompted.

Why does it matter?

- Fine-tuning can lead to high performance on a single task but might not generalize well to tasks it hasn't seen.
- Instruction tuning aims for more generalizable capabilities, letting the model follow your command on unseen tasks.

Goal: Discuss more technical details, referencing architecture, data, and optimization.

During **fine-tuning**, you typically:

1. Take a pre-trained transformer (e.g., GPT-2, GPT-Neo, or other LLMs) that's been trained on a massive corpus of text.
2. Use a labeled dataset, possibly with input-output pairs for a specific task (e.g., sentiment analysis, question answering, etc.).
3. Update the entire network's weights or sometimes just a subset (like a LoRA approach with low-rank adapters) to overfit to the domain/task of interest.

For **instruction tuning**:

1. You collect or create a dataset of *(instruction, response)* pairs from multiple tasks.
2. You might use techniques like **Reinforcement Learning from Human Feedback (RLHF)** to refine the model's responses to align with what humans expect.
3. The objective is to minimize a loss function that measures how well the model's response matches the "ideal" response or the high-quality labeled answer.
4. The model emerges as a versatile system that can handle tasks it hasn't explicitly been fine-tuned on, as long as instructions are well-formed.

In short:

- **Fine-Tuning** = narrower scope, modifies knowledge for a single domain or task.
- **Instruction Tuning** = broader, allows the model to interpret and handle diverse tasks.

Goal: Provide deep insights into the conceptual and research-based difference.

From a research standpoint, **fine-tuning** is often about adapting a large pre-trained model's weights to a low-data or domain-specific scenario. It was historically used in computer vision (e.g., fine-tuning ResNet on a custom image classification problem) and is widely adopted in NLP.

Instruction tuning has emerged in the wake of self-supervised language models that are so large and powerful, they can handle extensive tasks if guided properly. Projects like *FLAN, T5, GPT*, and others have demonstrated that providing a mixture of tasks during training with explicit instructions yields a single model capable of zero-shot or few-shot performance on new tasks.

Key papers and references:

- **Fine-Tuning**: Howard & Ruder, "Universal Language Model Fine-tuning for Text Classification (ULMFiT)", etc.
- **Instruction Tuning**: Wei et al. "Finetuned Language Models Are Zero-Shot Learners" (a.k.a. FLAN), Ouyang et al. "Training language models to follow instructions with human feedback" (RLHF for InstructGPT), etc.

Conclusion at the research level

- Fine-tuning has a narrower but deeper effect on a model's parameters.
- Instruction tuning is more about multi-task, universal coverage and generalizing from instructions.

Intuition Behind the Technology

Large Language Models (LLMs) learn patterns from huge amounts of text. Think of them as giant "prediction machines" that guess the next word in a sentence. However, once trained to do that, we can show them examples (fine-tuning) or instructions (instruction tuning) that guide them toward a certain type of behavior.

If you simply want to adapt the model to do one task **really well**, you fine-tune it on that task. But if you want the model to understand your instructions ("Translate this text" or "Summarize this article"), you perform instruction tuning with many tasks so it learns the general skill of following instructions.

Brief History, Invention, and Underlying Tech

1. **Origins**: Transformer models were first introduced in "Attention is All You Need" (Vaswani et al.).

2. **Pre-training + Fine-tuning**: BERT (Devlin et al.) showed that massive unsupervised pre-training can be adapted (fine-tuned) for new tasks.
3. **Instruction Tuning**: With GPT-like models, researchers found that instructing the model with different tasks yields a single model capable of many tasks.
4. **Reinforcement Learning from Human Feedback (RLHF)**: Takes instruction tuning further by letting humans rank or score outputs, refining the model's responses.

Math Behind It (and Beyond)

Basic Neural Network Setup

A language model is basically a neural network with parameters (weights) W and biases b. We pass an input sequence x, and the model predicts an output y. If we represent the neural network's function as $f_W(x)$, then the training process tries to find W that minimizes a *loss function* $L(y, f_W(x))$.

Fine-Tuning

- We start with a pre-trained model's parameters $W_{\text{pretrained}}$.
- We keep training on a task-specific dataset $D = \{(x_i, y_i)\}$ for that specialized task.
- We update W using gradient descent:

$$W \leftarrow W - \eta \frac{\partial}{\partial W} \left(\sum_{(x_i, y_i) \in D} L(y_i, f_W(x_i)) \right)$$

- Result: $W_{\text{fine-tuned}}$ is specialized.

Instruction Tuning

- We have multiple tasks, each with instructions. The model is trained to map instructions to the correct output, effectively learning a function $g_W(\text{instruction}) \rightarrow \text{output}$.
- We again perform gradient descent, but now across multiple tasks/instructions:

$$W \leftarrow W - \eta \frac{\partial}{\partial W} \left(\sum_{j=1}^{k} \sum_{(I_{j,i}, O_{j,i}) \in T_j} L(O_{j,i}, f_W(I_{j,i})) \right)$$

- Result: $W_{\text{instr-tuned}}$ can handle diverse tasks by simply reading an instruction.

Illustrative Example with Code

Below is a **toy** example to show how we might fine-tune or instruction-tune a small model in PyTorch on a very small, made-up dataset.

Example Scenario

- We want the model to respond to instructions.
- We have a mini-dataset of instructions and correct outputs.
- We'll build a mini neural network (not a full transformer, for brevity!).

```
import torch
import torch.nn as nn
import torch.optim as optim

# Sample instructions and responses (tiny dataset)
# Suppose we have instructions for adding two numbers, reversing a string, etc.
instructions = [
    "Add 1 and 2",
    "Add 3 and 4",
    "Reverse cat",
    "Reverse dog",
    "Add 10 and 20",
    "Reverse hello"
]

responses = [
    "3",  # result of 1+2
    "7",  # result of 3+4
    "tac",  # reverse of cat
    "god",  # reverse of dog
    "30",
    "olleh"
]

# For simplicity, let's pretend each instruction/response is a vector.
# We'll do a naive approach: convert each character to an integer encoding.
```

```
char_to_idx = {ch: i+1 for i, ch in enumerate("abcdefghijklmnopqrstuvwxyz0123456789 ")}  # +1 so we can have 0 be 'pad'
idx_to_char = {v: k for k, v in char_to_idx.items()}

def encode(text, max_len=10):
    # very naive encoding, pad or truncate to max_len
    vec = [char_to_idx.get(ch.lower(), 0) for ch in text]
    vec = vec[:max_len]
    vec += [0]*(max_len - len(vec))
    return vec

def decode(vec):
    return ''.join(idx_to_char.get(x, '?') for x in vec if x != 0)

max_len = 10
X = torch.tensor([encode(instr, max_len) for instr in instructions])
y = torch.tensor([encode(resp, max_len) for resp in responses])

# We'll create a very small model to map from instruction -> response.
# In a real scenario, you would have a large transformer. This is just a demonstration.

class TinyModel(nn.Module):
    def __init__(self, vocab_size, embed_dim=16, hidden_dim=32, max_len=10):
        super(TinyModel, self).__init__()
        self.embed = nn.Embedding(vocab_size+1, embed_dim)  # +1 for pad
        self.rnn = nn.GRU(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size+1)
        self.max_len = max_len

    def forward(self, x):
        # x shape: (batch_size, seq_len)
        embedded = self.embed(x)   # (batch_size, seq_len, embed_dim)
        out, hidden = self.rnn(embedded)
        # we want to produce an output of shape (batch_size, seq_len, vocab_size+1)
        logits = self.fc(out)
        return logits  # raw logits

# Initialize model
vocab_size = len(char_to_idx)
model = TinyModel(vocab_size)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

epochs = 300
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(X)
    # outputs shape: (batch_size, seq_len, vocab_size+1)
    # we need to reshape so we can apply cross entropy properly
    outputs_reshaped = outputs.view(-1, vocab_size+1)
    # target shape: (batch_size, seq_len)
    targets_reshaped = y.view(-1)
    loss = criterion(outputs_reshaped, targets_reshaped)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 50 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")

# Let's see how the model does
model.eval()
with torch.no_grad():
    test_outputs = model(X)
    predictions = torch.argmax(test_outputs, dim=2)

    for i, instr in enumerate(instructions):
        pred = decode(predictions[i].tolist())
        print(f"Instruction: {instr}, Predicted Response: {pred}")
```

```
Epoch 50/300, Loss: 0.3463
Epoch 100/300, Loss: 0.2989
Epoch 150/300, Loss: 0.2953
Epoch 200/300, Loss: 0.2941
Epoch 250/300, Loss: 0.2935
Epoch 300/300, Loss: 0.2931
Instruction: Add 1 and 2, Predicted Response: 3
Instruction: Add 3 and 4, Predicted Response: 3
Instruction: Reverse cat, Predicted Response: tod
Instruction: Reverse dog, Predicted Response: tod
Instruction: Add 10 and 20, Predicted Response: 3
Instruction: Reverse hello, Predicted Response: tod
```

Discussion

1. **Fine-tuning**: If this model was a pre-trained model on language tasks, we would be updating its weights on our new tasks of "Add two numbers" or "Reverse a string." This is effectively *fine-tuning* for these tasks.
2. **Instruction Tuning**: If we had many, many tasks with instruction-answer pairs, the model could learn to interpret instructions ("Add X and Y," "Reverse word"), effectively bridging multiple tasks in a single model.

Here, our example is extremely simplified (toy RNN, tiny dataset). Real instruction tuning uses huge models (millions/billions of parameters) and thousands to millions of instructions from many domains.

Example Calculations

- **Weights (W)**: In a neural network layer, these are the numbers multiplied by the inputs. For an embedding layer of dimension E, you have a weight matrix of size $V \times E$ where V is vocabulary size.
- **Bias (b)**: An extra parameter added to the weighted inputs, allowing the model to shift the activation function.
- **Loss Function**: Measures how far off the model's predictions are from the desired outputs. We used cross-entropy in the code.
- **Gradient Descent**: We compute the gradient of the loss function wrt. parameters ($\partial L / \partial W$) and adjust the parameters in the opposite direction to reduce the loss.

Example of Weight Update

- Suppose the weight is initially 1.0.
- The gradient says the weight should be decreased by 0.1 to get a better prediction.
- Then the new weight is 1.0 - 0.1 = 0.9.

That's the simplest mock calculation for how neural nets learn!

Step-by-Step Example of Creating This Technology from Scratch

We'll outline a simplified process of building an instruction-tuned model:

1. **Collect Data**: Gather a variety of tasks and their instructions. E.g., summarization, sentiment analysis, translation, etc., with actual sample instructions.
2. **Pre-processing**: Tokenize the instructions and outputs (split into pieces that the model can handle).
3. **Model Architecture**: Choose a large Transformer-based architecture.
4. **Initialize Weights**: Typically from a pre-trained checkpoint like GPT-2 or GPT-Neo.
5. **Loss Function**: Use cross-entropy that compares predicted token distributions to the gold standard.
6. **Optimization**: Use an optimizer like AdamW or similar.
7. **Training Loop**:
    - For each batch, feed the instructions and desired outputs into the model.
    - Compute the loss.
    - Backpropagate.
    - Update weights.
8. **Evaluation**: Check if the model can follow unseen instructions properly.

At a large scale, you would do this with millions of instructions and a huge compute cluster.

Illustrative Problem It Solves

It solves the problem of making a language model capable of answering different instructions. Instead of training separate models for each task—like one model for translation, one for Q&A, one for summarization—an instruction-tuned model can handle them all, if you simply provide the right instruction.

Real-World Problem & How to Solve Using This Tech

Real-World Example: Customer Support Chatbot

- Suppose you want a chatbot that can help customers with a variety of issues:
    1. Provide shipping information.
    2. Provide refund policies.

3. Troubleshoot connectivity issues.
4. Make suggestions about products.
5. Summarize user complaints.

Solution: Use an instruction-tuned LLM. You can create instructions like "Customer wants to track an order," "Customer wants to know the refund policy," etc. The model, after instruction tuning, can handle these diverse requests just by reading the prompt.

This saves huge amounts of time because you don't need to build separate specialized models for each new type of query.

Points to Ponder (Questions)

1. What is the main difference between **fine-tuning** and **instruction tuning**?
2. Why do we typically start with a **pre-trained** model?
3. What does it mean to **align** a model with human preferences?
4. How is **instruction tuning** related to **multi-task learning**?
5. What are **weights** and **biases**, and why do they matter?
6. How does **gradient descent** update the parameters?
7. Why do we need a **loss function**?
8. How does **tokenization** help in text tasks?
9. Can **instruction-tuned** models handle tasks they have never seen before?
10. Why do we say **"Attention is All You Need"** in transformers?

Answers to the Questions with Code Examples

1. **Difference between fine-tuning and instruction tuning**:
    - Fine-tuning: Train model on one specific task with labeled data.
    - Instruction tuning: Train model to respond to general instructions spanning many tasks.
2. **Why start with a pre-trained model?**
    - Because it already knows a lot about language structure; we only refine it for our purpose.
3. **Align a model with human preferences**:
    - Use human feedback (e.g., RLHF) to shape the model's responses so they are more correct or more helpful.
4. **Instruction tuning as multi-task learning**:
    - We effectively train the model on multiple tasks at once, capturing a universal instruction-following ability.
5. **Weights and biases**:
    - Numerical parameters that determine how inputs get transformed into outputs.
6. **Gradient descent updates parameters**:
    - By computing how changing each parameter slightly changes the loss, then nudging parameters in the direction that reduces loss.
7. **Why we need a loss function**:
    - We need a quantitative measure of "how bad" the model's predictions are, to know how to improve.
8. **Tokenization**:
    - Splits text into smaller units (tokens). The model processes each token. This helps the model handle variable-length text.
9. **Can instruction-tuned models handle unseen tasks?**
    - Often yes, if they are related or can be described with instructions. The model learned how to interpret instructions in general.
10. **"Attention is All You Need"** in transformers:
    - It's the name of the seminal paper that introduced the Transformer architecture, which uses attention mechanisms instead of older RNN-based methods to process sequences more efficiently.

A Sample Exercise

Below is a **starter code** to illustrate a mini version of a text classification fine-tuning task. Please complete the TODO items and run.

```
# TODO: Import necessary PyTorch libraries
import torch
import torch.nn as nn
import torch.optim as optim
```

```
# We have some sample text classification data
# Suppose we want to classify short texts as either 'greeting' or 'farewell'.

texts = [
    "Hello there",
    "Hi, how are you?",
    "Goodbye",
    "Bye bye",
    "See you later",
    "Good evening",
    "Farewell",
    "Hey"
]

# Labels: greeting=0, farewell=1
labels = [0, 0, 1, 1, 1, 0, 1, 0]

# TODO: Create a vocabulary and a method to tokenize/encode these texts.
vocab = {}
index = 1  # start indexing at 1

def build_vocab(text_list):
    global vocab, index
    for t in text_list:
        for word in t.lower().split():
            if word not in vocab:
                vocab[word] = index
                index += 1

# Build the vocabulary
build_vocab(texts)

print("Vocabulary:", vocab)

def encode_sentence(sentence, max_len=5):
    encoded = []
    for word in sentence.lower().split():
        encoded.append(vocab.get(word, 0))  # 0 if not found
    encoded = encoded[:max_len]
    # pad if needed
    while len(encoded) < max_len:
        encoded.append(0)
    return encoded

# Encode all sentences
X_data = [encode_sentence(s) for s in texts]
y_data = labels

X_tensor = torch.tensor(X_data)
y_tensor = torch.tensor(y_data)

# Simple classification model
class SimpleClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim=8, hidden_dim=16, num_classes=2):
        super(SimpleClassifier, self).__init__()
        self.embed = nn.Embedding(vocab_size+1, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        embedded = self.embed(x)
        out, hidden = self.rnn(embedded)
        # Use the last hidden state for classification
        last_hidden = hidden[-1]
        logits = self.fc(last_hidden)
        return logits

# Instantiate model
model = SimpleClassifier(vocab_size=len(vocab))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# TODO: Write a training loop to train this classifier
epochs = 50
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    logits = model(X_tensor)
    loss = criterion(logits, y_tensor)
    loss.backward()
    optimizer.step()
```

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

# Evaluate
model.eval()
with torch.no_grad():
    preds = model(X_tensor)
    predicted_labels = torch.argmax(preds, dim=1)
    correct = (predicted_labels == y_tensor).sum().item()
    print(f"Accuracy: {correct / len(y_tensor)*100:.2f}%")

# TODO: Try your own sentences:
new_text = "Hello friend"
encoded_new = torch.tensor([encode_sentence(new_text)])
output = model(encoded_new)
pred_label = torch.argmax(output, dim=1).item()
print(f"Sentence: '{new_text}' -> Predicted label: {pred_label} (0=greeting, 1=farewell)")
```

```
Vocabulary: {'hello': 1, 'there': 2, 'hi,': 3, 'how': 4, 'are': 5, 'you?': 6, 'goodbye': 7, 'bye': 8, 'see': 9, 'you': 10, 'later': 11, 'good': 12, 'evening': 13, 'farewell': 14, 'hey': 15}
Epoch 10/50, Loss: 0.4503
Epoch 20/50, Loss: 0.0283
Epoch 30/50, Loss: 0.0017
Epoch 40/50, Loss: 0.0006
Epoch 50/50, Loss: 0.0003
Accuracy: 100.00%
Sentence: 'Hello friend' -> Predicted label: 0 (0=greeting, 1=farewell)
```

Explanation:

1. We build a small vocabulary from the data.
2. We tokenize each sentence into integers.
3. We feed this into a small GRU-based classification model.
4. We train with CrossEntropyLoss for classification.
5. After training, we check how well the model does, then we test with a new sentence.

You can expand this example, add more tasks, and eventually build an instruction-based dataset to see how instruction tuning might work in principle.

Glossary

LLM (Large Language Model): A neural network model that's been trained on massive text corpora to predict or generate text.

Transformer: A neural architecture based on the attention mechanism, introduced in the seminal paper "Attention is All You Need." Forms the backbone of most modern LLMs.

Fine-Tuning: Taking a pre-trained model and further training it on a more specific task to specialize it.

Instruction Tuning: Training a model to follow a variety of instructions by exposing it to instruction–response pairs across multiple tasks.

Weights and Biases: Trainable parameters in a neural network that are adjusted via optimization.

Loss Function: A function that measures how far the model's predictions deviate from the desired targets. We aim to minimize this.

Gradient Descent: An optimization algorithm used to minimize the loss function by iteratively moving in the negative direction of the gradient.

Cross Entropy Loss: A common loss function for classification problems, measuring the distance between two probability distributions.

Epoch: One full pass of the training data through the model.

Tokenization: Splitting text into small, discrete units (tokens) for the model to process.

RNN/GRU: Recurrent Neural Networks / Gated Recurrent Units – older but still relevant architecture for sequential data.

Transformer-based Models: State-of-the-art models that rely on attention mechanisms, e.g., BERT, GPT, T5, etc.

RLHF (Reinforcement Learning from Human Feedback): A method of refining an LLM's behavior by learning from how humans rate or prefer certain outputs.

End of notebook.

```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
```

```
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-29 01:32:21.780175
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```

Start coding or generate with AI.