2_Initial_Model_Runs

April 25, 2025

[1]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt
import numpy as np
import sys
import os
import time
import copy  # To store model state if needed

# Ensure the src directory is in the Python path
# Adjust the path '..' if your notebook is in a different location relative to␣
↪src
module_path = os.path.abspath(os.path.join('..', 'src'))
if module_path not in sys.path:
    sys.path.append(module_path)

# Import modules from src
from utils import load_processed_data
from models import Model_1, Model_2, Model_3
```

[2]:
```python
# --- Configuration ---
BATCH_SIZE = 128 # Reasonable batch size (can try 64, 256)
LEARNING_RATE = 1e-3 # A common default starting LR for Adam
N_MINIBATCHES = 15
EVAL_INTERVAL = 5 # Evaluate on validation set every X mini-batches
SEED = 42 # For reproducibility
```

[3]:
```python
# --- Set Seed ---
torch.manual_seed(SEED)
np.random.seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
# Note: MPS backend reproducibility might have limitations, but setting CPU/
↪CUDA seeds is good practice.
```

```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "mps" if torch.
↪backends.mps.is_available() else "cpu")
print(f"Using device: {DEVICE}")
print(f"Reproducibility seed set to: {SEED}")
```

```
Using device: mps
Reproducibility seed set to: 42
```

[4]:
```
# --- Load Data ---
print("Loading data...")
X_train, y_train, X_val, y_val, _, _ = load_processed_data()
print("Data loaded.")
```

```
2025-04-25 13:23:38,489 - INFO - Loading data from ../data/processed/…
2025-04-25 13:23:38,527 - INFO - Processed data loaded successfully.
2025-04-25 13:23:38,527 - INFO -   Train shapes: X=torch.Size([372336, 90]),
y=torch.Size([372336])
2025-04-25 13:23:38,528 - INFO -   Val shapes: X=torch.Size([71504, 90]),
y=torch.Size([71504])
2025-04-25 13:23:38,528 - INFO -   Test shapes: X=torch.Size([71505, 90]),
y=torch.Size([71505])
```

```
Loading data…
Data loaded.
```

[6]:
```
# Create datasets
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
```

[7]:
```
# Create dataloaders
# Use shuffle=True for training to ensure batches are different each epoch
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,␣
↪generator=torch.Generator().manual_seed(SEED))
# No need to shuffle validation data
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE * 2) # Larger batch␣
↪size for faster validation
```

[8]:
```
# --- Define Training and Evaluation Functions ---

def train_one_step(model, batch, criterion, optimizer, device):
    """Performs a single training step (forward pass, loss calc, backward pass,␣
↪optimizer step)."""
    model.train() # Set model to training mode
    inputs, targets = batch
    inputs, targets = inputs.to(device), targets.to(device)

    # Zero gradients
    optimizer.zero_grad()
```

```
    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # Backward pass and optimize
    loss.backward()
    optimizer.step()

    return loss.item()
```

[9]:
```
def evaluate(model, loader, criterion, device):
    """Evaluates the model on the given data loader."""
    model.eval() # Set model to evaluation mode
    total_loss = 0.0
    correct_predictions = 0
    total_samples = 0

    with torch.no_grad(): # Disable gradient calculations during evaluation
        for batch in loader:
            inputs, targets = batch
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            total_loss += loss.item() * inputs.size(0) # Accumulate loss␣
 ↪weighted by batch size

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total_samples += targets.size(0)
            correct_predictions += (predicted == targets).sum().item()

    avg_loss = total_loss / total_samples
    accuracy = correct_predictions / total_samples
    return avg_loss, accuracy
```

[10]:
```
# --- Experiment Setup ---
model_architectures = {
    "Model_1 (128x128)": Model_1,
    "Model_2 (256x256)": Model_2,
    "Model_3 (256x128x64)": Model_3
}

results = {} # To store detailed results
```

```
criterion = nn.CrossEntropyLoss()
```

[11]:
```
# --- Run Experiments ---

for name, ModelClass in model_architectures.items():
    print(f"\n--- Running Experiment for: {name} ---")
    # Re-seed generator for dataloader for each model if desired (optional, but␣
 ↪good practice)
    # train_loader.generator.manual_seed(SEED) # Reset iterator state implicitly
    train_iter = iter(train_loader) # Create a fresh iterator

    # Instantiate model and move to device
    # Note: Parameter initialization depends on the global torch seed set␣
 ↪earlier
    model = ModelClass().to(DEVICE)
    print(model) # Print architecture details

    # Use Adam optimizer with a default learning rate
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    # Store performance metrics
    minibatch_losses = []
    eval_batches = [] # Batch numbers where evaluation was performed
    val_losses = []
    val_accuracies = []

    start_time = time.time()

    # Training loop for N_MINIBATCHES
    batch_count = 0

    while batch_count < N_MINIBATCHES:
        try:
            # Fetch the next batch
            batch = next(train_iter)
        except StopIteration:
            # Reset iterator if it runs out (shouldn't happen in 15 batches␣
 ↪normally)
            print("Resetting train_loader iterator...")
            train_iter = iter(train_loader)
            batch = next(train_iter)

        # Perform one training step
        loss = train_one_step(model, batch, criterion, optimizer, DEVICE)
        minibatch_losses.append(loss)
        batch_count += 1
```

```
        # Optional: Print progress
        # print(f"  Batch {batch_count}/{N_MINIBATCHES}, Loss: {loss:.4f}") #␣
↪Can be verbose

        # Intermediate Evaluation
        if batch_count % EVAL_INTERVAL == 0 or batch_count == N_MINIBATCHES:
            eval_start_time = time.time()
            val_loss, val_accuracy = evaluate(model, val_loader, criterion,␣
↪DEVICE)
            eval_end_time = time.time()
            eval_batches.append(batch_count)
            val_losses.append(val_loss)
            val_accuracies.append(val_accuracy)
            print(f"  Batch {batch_count}/{N_MINIBATCHES} -> Train Loss (last␣
↪batch): {loss:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_accuracy:.4f}␣
↪(Eval took {eval_end_time - eval_start_time:.2f}s)")


    end_time = time.time()
    total_training_time = end_time - start_time

    print(f"Finished {name}.")
    print(f"  Total Training Time for {N_MINIBATCHES} batches:␣
↪{total_training_time:.2f} seconds")

    # Store results
    results[name] = {
        'minibatch_losses': minibatch_losses,
        'eval_batches': eval_batches, # e.g., [5, 10, 15]
        'val_losses': val_losses,      # List of val losses at eval points
        'val_accuracies': val_accuracies, # List of val accuracies at eval␣
↪points
        'final_val_loss': val_losses[-1], # Get the last recorded val loss
        'final_val_accuracy': val_accuracies[-1], # Get the last recorded val␣
↪accuracy
        'training_time': total_training_time
    }
```

```
--- Running Experiment for: Model_1 (128x128) ---
Model Architecture: Model_1
  (layer_1): Linear(in_features=90, out_features=128, bias=True)
  (relu_1): ReLU()
  (layer_2): Linear(in_features=128, out_features=128, bias=True)
  (relu_2): ReLU()
  (output_layer): Linear(in_features=128, out_features=10, bias=True)
```

```
Total Trainable Parameters: 29,450

  Batch 5/15 -> Train Loss (last batch): 2.1247, Val Loss: 2.0497, Val Acc:
0.5651 (Eval took 0.45s)
  Batch 10/15 -> Train Loss (last batch): 1.7969, Val Loss: 1.7646, Val Acc:
0.5802 (Eval took 0.42s)
  Batch 15/15 -> Train Loss (last batch): 1.5498, Val Loss: 1.4841, Val Acc:
0.5802 (Eval took 0.41s)
Finished Model_1 (128x128).
  Total Training Time for 15 batches: 1.53 seconds

--- Running Experiment for: Model_2 (256x256) ---
Model Architecture: Model_2
  (layer_1): Linear(in_features=90, out_features=256, bias=True)
  (relu_1): ReLU()
  (layer_2): Linear(in_features=256, out_features=256, bias=True)
  (relu_2): ReLU()
  (output_layer): Linear(in_features=256, out_features=10, bias=True)
Total Trainable Parameters: 91,658

  Batch 5/15 -> Train Loss (last batch): 1.8090, Val Loss: 1.6533, Val Acc:
0.5802 (Eval took 0.48s)
  Batch 10/15 -> Train Loss (last batch): 1.3203, Val Loss: 1.3351, Val Acc:
0.5802 (Eval took 0.40s)
  Batch 15/15 -> Train Loss (last batch): 1.3752, Val Loss: 1.2711, Val Acc:
0.5802 (Eval took 0.39s)
Finished Model_2 (256x256).
  Total Training Time for 15 batches: 1.40 seconds

--- Running Experiment for: Model_3 (256x128x64) ---
Model Architecture: Model_3
  (layer_1): Linear(in_features=90, out_features=256, bias=True)
  (relu_1): ReLU()
  (layer_2): Linear(in_features=256, out_features=128, bias=True)
  (relu_2): ReLU()
  (layer_3): Linear(in_features=128, out_features=64, bias=True)
  (relu_3): ReLU()
  (output_layer): Linear(in_features=64, out_features=10, bias=True)
Total Trainable Parameters: 65,098

  Batch 5/15 -> Train Loss (last batch): 2.0922, Val Loss: 2.0090, Val Acc:
0.5788 (Eval took 0.47s)
  Batch 10/15 -> Train Loss (last batch): 1.6210, Val Loss: 1.5653, Val Acc:
0.5802 (Eval took 0.42s)
  Batch 15/15 -> Train Loss (last batch): 1.3323, Val Loss: 1.3519, Val Acc:
0.5802 (Eval took 0.47s)
Finished Model_3 (256x128x64).
  Total Training Time for 15 batches: 1.47 seconds
```

[17]:
```python
# Plot Training Loss (per mini-batch)
plt.figure(figsize=(14, 8))
plt.subplot(2, 1, 1) # Create subplot 1
for name, data in results.items():
    plt.plot(range(1, N_MINIBATCHES + 1), data['minibatch_losses'],␣
↪label=f"{name}", alpha=0.8)

plt.xlabel("Mini-batch Number")
plt.ylabel("Training Loss")
plt.title("Training Loss per Mini-batch")
plt.legend()
plt.grid(True)
plt.xticks(range(1, N_MINIBATCHES + 1))
```

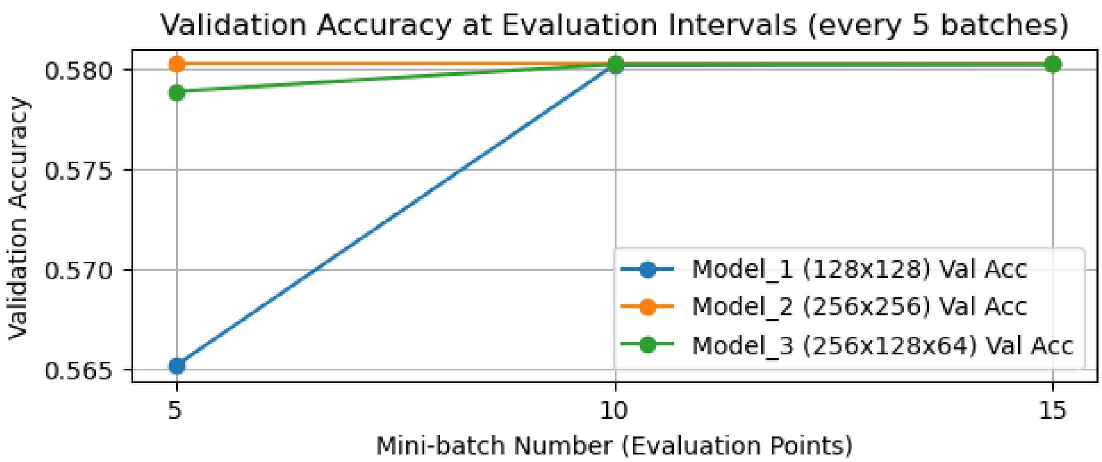
[17]:
```
([<matplotlib.axis.XTick at 0x146a91b20>,
  <matplotlib.axis.XTick at 0x146a91fd0>,
  <matplotlib.axis.XTick at 0x1469f4ce0>,
  <matplotlib.axis.XTick at 0x146aaf140>,
  <matplotlib.axis.XTick at 0x146aade20>,
  <matplotlib.axis.XTick at 0x146aafef0>,
  <matplotlib.axis.XTick at 0x146ad8830>,
  <matplotlib.axis.XTick at 0x146ad91c0>,
  <matplotlib.axis.XTick at 0x146ad9af0>,
  <matplotlib.axis.XTick at 0x146aafa40>,
  <matplotlib.axis.XTick at 0x146ada120>,
  <matplotlib.axis.XTick at 0x146ada9c0>,
  <matplotlib.axis.XTick at 0x146adb2f0>,
  <matplotlib.axis.XTick at 0x146adbce0>,
  <matplotlib.axis.XTick at 0x146af8650>],
 [Text(1, 0, '1'),
  Text(2, 0, '2'),
  Text(3, 0, '3'),
  Text(4, 0, '4'),
  Text(5, 0, '5'),
  Text(6, 0, '6'),
  Text(7, 0, '7'),
  Text(8, 0, '8'),
  Text(9, 0, '9'),
  Text(10, 0, '10'),
  Text(11, 0, '11'),
  Text(12, 0, '12'),
  Text(13, 0, '13'),
  Text(14, 0, '14'),
  Text(15, 0, '15')])
```

[16]:

```
# Plot Validation Performance (at evaluation intervals)
plt.subplot(2, 1, 2) # Create subplot 2
for name, data in results.items():
    # Plot validation accuracy
    plt.plot(data['eval_batches'], data['val_accuracies'], label=f"{name} Val␣
 ↪Acc", marker='o', linestyle='-')
    # Optionally plot validation loss on a secondary y-axis if scales differ␣
 ↪too much
    # (Let's keep it simple for now and focus on accuracy)

plt.xlabel("Mini-batch Number (Evaluation Points)")
plt.ylabel("Validation Accuracy")
plt.title(f"Validation Accuracy at Evaluation Intervals (every {EVAL_INTERVAL}␣
 ↪batches)")
plt.legend()
plt.grid(True)
plt.xticks(results[list(results.keys())[0]]['eval_batches']) # Use eval batches␣
 ↪from first result as ticks

plt.tight_layout() # Adjust layout to prevent overlap
plt.show()
```



[18]:
```
# Print summary table (using final validation metrics after 15 batches)
print("\n--- Summary of Initial Runs (Performance after 15 Mini-batches) ---")
print(f"{'Architecture':<25} | {'Final Val Loss':<15} | {'Final Val Accuracy':
↪<18} | {'Training Time (s)':<15}")
print("-" * 80)
for name, data in results.items():
    print(f"{name:<25} | {data['final_val_loss']:.4f}{' ':<10} |␣
↪{data['final_val_accuracy']:.4f}{' ':<13} | {data['training_time']:.2f}")
```

```
--- Summary of Initial Runs (Performance after 15 Mini-batches) ---
Architecture              | Final Val Loss  | Final Val Accuracy | Training Time
(s)
--------------------------------------------------------------------------------
Model_1 (128x128)         | 1.4841          | 0.5802             | 1.53
Model_2 (256x256)         | 1.2711          | 0.5802             | 1.40
Model_3 (256x128x64)      | 1.3519          | 0.5802             | 1.47
```

[19]:
```
# --- Select Best Performing Architecture ---
# Based on final validation accuracy primarily
best_model_name = ""
best_val_accuracy = -1.0

for name, data in results.items():
    if data['final_val_accuracy'] > best_val_accuracy:
        best_val_accuracy = data['final_val_accuracy']
        best_model_name = name
    # Could add tie-breaking logic using final_val_loss if needed

print(f"\nBased on final validation accuracy after {N_MINIBATCHES} batches, the␣
↪best performing architecture appears to be: {best_model_name}")
```

```
print(f"(Achieved {best_val_accuracy:.4f} accuracy)")
print("(Note: This is based on very limited training. The validation trajectory␣
↪plot provides more context.)")
```

```
Based on final validation accuracy after 15 batches, the best performing
architecture appears to be: Model_1 (128x128)
(Achieved 0.5802 accuracy)
(Note: This is based on very limited training. The validation trajectory plot
provides more context.)
```