# Exercise 1

## Task 1:

Batch Gradient Descent:

computes the gradient of the loss function using the entire dataset in each update step. It provides a solution with minimal noise and results in stable convergence, but it is computationally very expensive.

Stochastic Gradient Descent:

uses only a single data point per iteration, which makes an iteration faster and less expensive, but it increases the noise. This noise may help to explore new areas but also slow convergence down.

Mini-batch SGD:

is a middle way that computes the gradient on a small subset which is not as expensive as BGD but still introduces some noise.

## Task 2:

a) A fixed learning rate might be too high which leads to divergence or too low which leads to slow progress. To navigate a landscape of a neural network you could use a large learning rate for fast exploration and a low one to fine tune.
b) A learning rate schedule is a technique where we adjust the learning rate over time so that it gives us to highest benefit for training.
**Exponential Decay:**

$$\eta_t = \eta_0 \cdot e^{-\lambda t}$$

- ηt is the learning rate at epoch t.
- η0 is the initial learning rate at the start of training.
- t is the current epoch during training.
- λ is the exponential decay constant.
As t increases the learning rate decreases exponentially. This allows large steps initially and reduces the size of the steps overtime to avoid overshooting and a stable pace to finetune.

Exercise 1

3.

(a) training set is used for machine learning model training, where it is used to adjust the parameters and weights of model and minimize training loss.

Validation set is for checking if model generalize well during training. By using it seperately from training set, it can evaluate the models performance during training.

Test set is for after training, which provides an evaluation on how the model might performs on the expected data.

(b)

By avoiding to use test set during training, it would prevent model from knowing the information ahead and adapt it into the model. which could lead to a biased evaluation because the model will most likely perform well since it has used the set to train itself. It would also likely to increase variance of estimate since it overfits if test set is used during training.

(c)

First define some possible values for each hyperparameters. which is a grid. Then among these assumptions, we use different combination to train and evaluate the model using training and validation set.

compare the results and get the best result grid which then should be the final hyperparameters we choose.

4.

(a) RMSProp is a optimizer which solves the problem of learning rate too large or small. it automatically adjust learning rate for each parameter which helps the model to converge faster and more smoothly

Momentum optimizer is intead of update parameter base on instant gradient result ; it saves the results from previous updates too and take them into account.

(b)

(i) $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ → updated first moment

$= 0.9 \cdot 0.5 + (1-0.9) \cdot 2.0$

$= 0.65$

$V_t = \beta_2 V_{t-1} + (1-\beta_2) g_t^2$ → updated second moment

$= 0.99 \cdot 0.2 + (1-0.99) \cdot (2.0)^2$

$= 0.238$

(ii)

$\Delta w_t = -\alpha \frac{m_t}{\sqrt{V_t} + \epsilon}$

$= -0.01 \cdot \frac{0.65}{\sqrt{0.238} + \epsilon}$

$= -0.01 \cdot 1.33$

$= 0.0133$

(iii)

$v_t'$ would be a lot larger than $v_t$ because $20 \gg 0.2(v_{t-1})$

$|\Delta w_t'|$ would be smaller than $|\Delta w_t|$ because again $v_t'$ is large and then $\sqrt{v_t'}$ on denominator would make the result small

It implies that Adam automatically adjust the learning rate for different parameters.

# MLE25_sheet05

June 8, 2025

# 1 Machine Learning Essentials SS25 - Exercise Sheet 5

## 1.1 Instructions

- `TODO`'s indicate where you need to complete the implementations.
- You may use external resources, but write your own solutions.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

## 1.2 Exercise 1

## 1.3 Exercise 2

[2]:
```python
import matplotlib.pyplot as plt
import numpy as np
# TODO: Import the stuff you need from torch and torchvision
import torch
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn

"""
If you stay in ML-related fields, you will likely be working on a server or␣
↪cluster. If you do so,
always remember to set the number of CPU (or even GPU) threads you're using, as␣
↪Jupyter notebooks or Python scripts
might sometimes use all available threads by default, which will lead to␣
↪unhappy colleagues or classmates that
also want to use some of the threads.
"""
# Example of limiting CPU threads:
# import os
# os.environ["OMP_NUM_THREADS"] = "15"
# os.environ["MKL_NUM_THREADS"] = "15"
# torch.set_num_threads(15) # If you only want to use PyTorch threads
```

[2]:
```
'"\nIf you stay in ML-related fields, you will likely be working on a server or
cluster. If you do so,\nalways remember to set the number of CPU (or even GPU)
threads you\'re using, as Jupyter notebooks or Python scripts \nmight sometimes
```

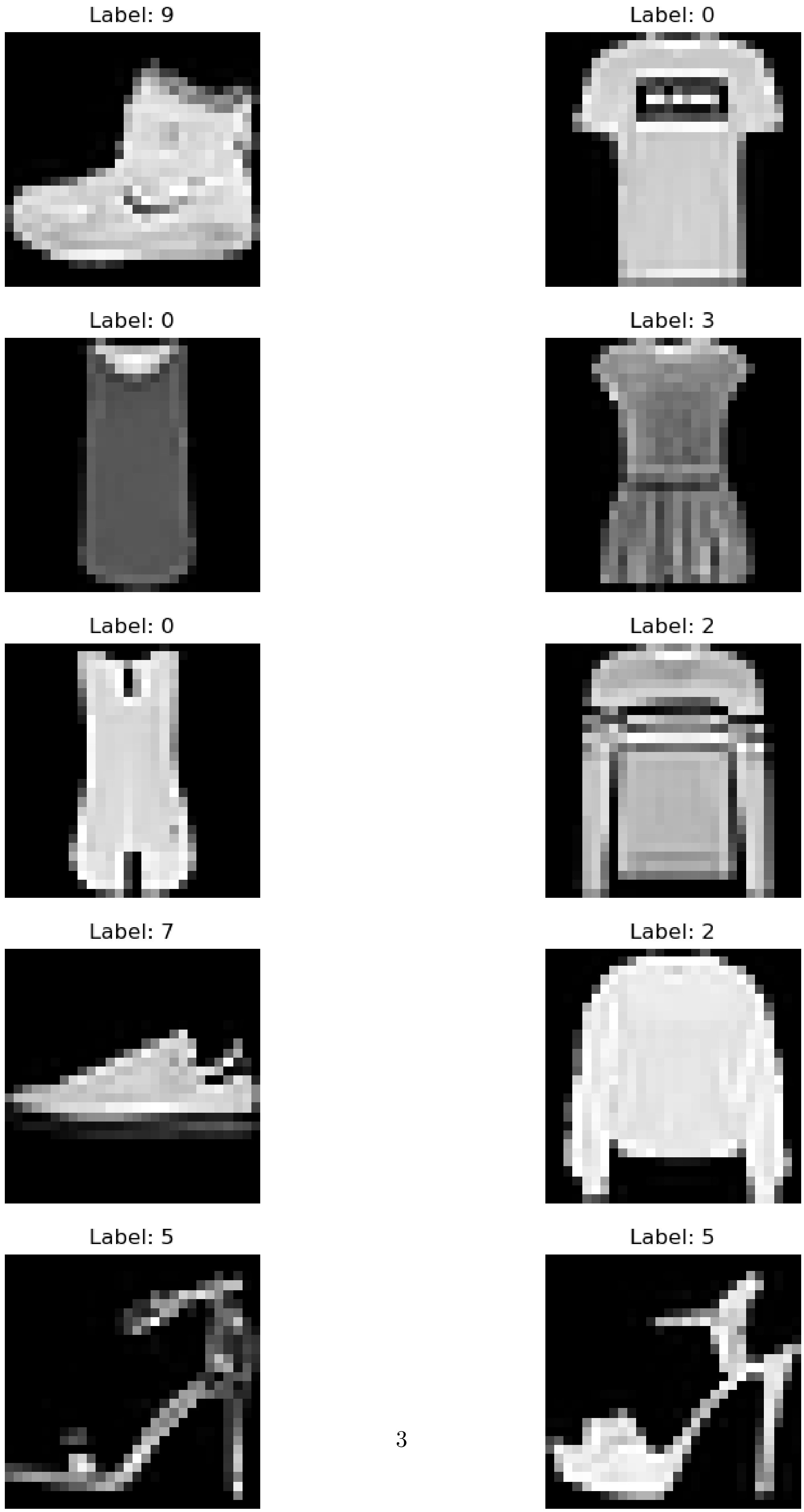
```
use all available threads by default, which will lead to unhappy colleagues or
classmates that\nalso want to use some of the threads.\n'
```

### 1.3.1 2.1

[3]:
```
# TODO: Define transformations
# Given statistics of training set:
mu_train = 0.286
std_train = 0.353
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mu_train, std=std_train),
])

# TODO: Load FashionMNIST train/testsets
train_dataset_full = datasets.FashionMNIST(
    root='./data',
    train=True,
    transform=transform,
    download=True      # Download the dataset at the first run
)
test_dataset = datasets.FashionMNIST(
    root='./data',
    train=False,
    transform=transform,
    download=True
)

print(f"Full training dataset size: {len(train_dataset_full)}")
print(f"Test dataset size: {len(test_dataset)}")
```

```
Full training dataset size: 60000
Test dataset size: 10000
```

[4]:
```
# TODO: Create 5x2 subplot grid w/ example image for each class
fig, axes = plt.subplots(5, 2, figsize=(10, 15))
axes = axes.flatten()
for i in range(10):
    image, label = train_dataset_full[i]
    image_np = image.numpy().reshape(28, 28)
    axes[i].imshow(image_np, cmap='gray')
    axes[i].set_title(f'Label: {label}')
    axes[i].axis('off')
```


Label: 9
Label: 0
Label: 0
Label: 3
Label: 0
Label: 2
Label: 7
Label: 2
Label: 5
Label: 5

[5]:
```
# TODO: Create a validation set from the training set
train_set, val_set = torch.utils.data.random_split(train_dataset_full, [50000,␣
↪10000])
# print(f"Training set size: {len(train_set)}")
# print(f"Validation set size: {len(val_set)}")
```

### 1.3.2 2.2

[6]:
```
# TODO: Define your model architecture: A class called MLP that inherits from␣
↪nn.Module
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        input_size = 28 * 28
        hidden_size = 128
        output_size = 10
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        return x

# TODO: Define appropriate loss
criterion = nn.CrossEntropyLoss()
```

### 1.3.3 2.3

[7]:
```
BATCH_SIZE_DEFAULT = 256 # TODO: Set your default batch size. The␣
↪capitalization is a convention used for global constants in Python

#TODO: Define DataLoaders for training, validation, and test sets
data_loader_train = torch.utils.data.DataLoader(train_set,␣
↪batch_size=BATCH_SIZE_DEFAULT, shuffle=True)
data_loader_val = torch.utils.data.DataLoader(val_set,␣
↪batch_size=BATCH_SIZE_DEFAULT, shuffle=False)
data_loader_test = torch.utils.data.DataLoader(test_dataset,␣
↪batch_size=BATCH_SIZE_DEFAULT, shuffle=False)
```

[8]:
```
def calculate_accuracy(outputs, labels):
    """
```

```python
    Calculate accuracy given model outputs and true labels.
    """
    _, predicted = torch.max(outputs.data, 1) # Prediction = class with highest␣
↪output probability
    total = labels.size(0)
    correct = (predicted == labels).sum().item() #.item() converts a␣
↪single-element tensor to a Python number ("scalar")
    return correct / total

def train_model(model, criterion, optimizer, train_loader, val_loader,␣
↪num_epochs):
    # Device configuration: if available use GPU (needs CUDA installed and GPU␣
↪with PyTorch support), otherwise CPU
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)
    print(f"Training on device: {device}")

    # TODO: Define training loop that for each epoch iterates over all␣
↪mini-batches in the training set
    # Record and return the training&validation loss and accuracy for each epoch
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in range(num_epochs):
        running_loss = 0.
        train_correct = 0
        total = 0

        # train phase
        model.train()
        for batch_idx, (images, labels) in enumerate(train_loader):
            images, labels = images.to(device), labels.to(device)
            output = model.forward(images)
            loss = criterion(output, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            accuracy = calculate_accuracy(output, labels)
            train_correct += accuracy * labels.size(0)
            total += labels.size(0)

        # compute avg loss and accuracy
        train_loss = running_loss / len(train_loader)
        train_accuracy = train_correct / total
```

```
        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)

        # validation phase
        model.eval()

        # (context-manager that disables gradient calculation)
        with torch.no_grad():
            val_loss = 0.
            val_correct = 0
            val_total = 0

            for batch_idx, (images, labels) in enumerate(val_loader):
                images, labels = images.to(device), labels.to(device)
                output = model.forward(images)
                loss = criterion(output, labels)
                val_loss += loss.item()
                accuracy = calculate_accuracy(output, labels)
                val_correct += accuracy * labels.size(0)
                val_total += labels.size(0)

        # ditto
        val_loss /= len(val_loader)
        val_accuracy = val_correct / val_total
        val_losses.append(val_loss)
        val_accuracies.append(val_accuracy)

    return train_losses, val_losses, train_accuracies, val_accuracies
```

### 1.3.4 2.4

[14]:
```
# TODO: Define hyperparameter grid for tuning
hyperparameter_grid = {
    'learning_rate': [0.001],
    'batch_size': [64, 128, 256],
    'num_epochs': [5, 10]
}

# TODO: For each hyperparameter setting, instantiate model&optimizer,
# train the model, and store the results for evaluation later
adam_results = []
for lr in hyperparameter_grid['learning_rate']:
    for batch_size in hyperparameter_grid['batch_size']:
        for num_epochs in hyperparameter_grid['num_epochs']:
            print(f"Training with learning rate={lr}, batch_size={batch_size},␣
 ↪num_epochs={num_epochs}")
            model = MLP()
```

```
            optimizer = torch.optim.Adam(model.parameters(), lr=lr)
            train_loader = torch.utils.data.DataLoader(train_set,␣
↪batch_size=batch_size, shuffle=True)
            val_loader = torch.utils.data.DataLoader(val_set,␣
↪batch_size=batch_size, shuffle=False)

            train_losses, val_losses, train_accuracies, val_accuracies =␣
↪train_model(
                model, criterion, optimizer, train_loader, val_loader,␣
↪num_epochs
            )

            adam_results.append({
                'learning_rate': lr,
                'batch_size': batch_size,
                'num_epochs': num_epochs,
                'train_losses': train_losses,
                'val_losses': val_losses,
                'train_accuracies': train_accuracies,
                'val_accuracies': val_accuracies
            })
```

```
Training with learning rate=0.001, batch_size=64, num_epochs=5
Training on device: cpu
Training with learning rate=0.001, batch_size=64, num_epochs=10
Training on device: cpu
Training with learning rate=0.001, batch_size=128, num_epochs=5
Training on device: cpu
Training with learning rate=0.001, batch_size=128, num_epochs=10
Training on device: cpu
Training with learning rate=0.001, batch_size=256, num_epochs=5
Training on device: cpu
Training with learning rate=0.001, batch_size=256, num_epochs=10
Training on device: cpu
```

[ ]:
```
# TODO: Define hyperparameter grid for tuning
hyperparameter_grid = {
    'learning_rate': [0.001],
    'batch_size': [64, 128, 256],
    'num_epochs': [5, 10]
}

# TODO: For each hyperparameter setting, instantiate model&optimizer,
# train the model, and store the results for evaluation later
sgd_results = []
for lr in hyperparameter_grid['learning_rate']:
    for batch_size in hyperparameter_grid['batch_size']:
```

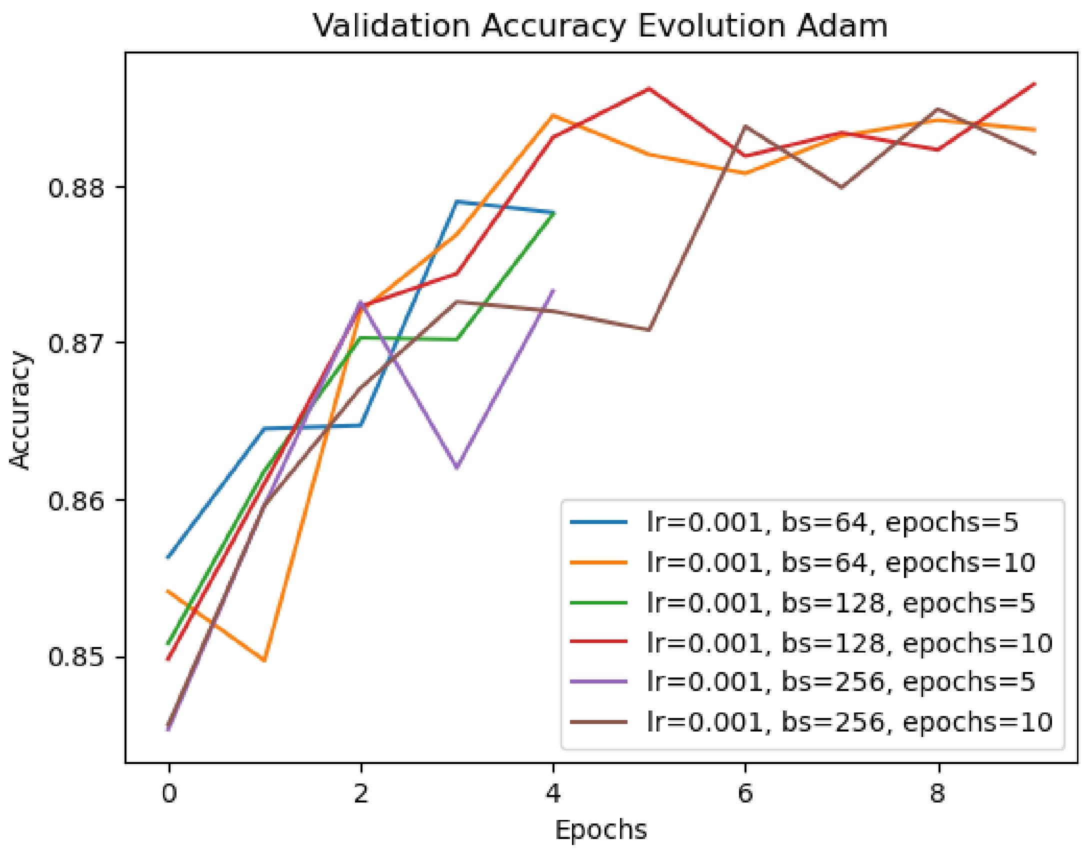
```
        for num_epochs in hyperparameter_grid['num_epochs']:
            print(f"Training with learning rate={lr}, batch_size={batch_size},␣
↪num_epochs={num_epochs}")
            model = MLP()
            optimizer = torch.optim.SGD(model.parameters(), lr=lr)
            train_loader = torch.utils.data.DataLoader(train_set,␣
↪batch_size=batch_size, shuffle=True)
            val_loader = torch.utils.data.DataLoader(val_set,␣
↪batch_size=batch_size, shuffle=False)

            train_losses, val_losses, train_accuracies, val_accuracies =␣
↪train_model(
                model, criterion, optimizer, train_loader, val_loader,␣
↪num_epochs
            )

            sgd_results.append({
                'learning_rate': lr,
                'batch_size': batch_size,
                'num_epochs': num_epochs,
                'train_losses': train_losses,
                'val_losses': val_losses,
                'train_accuracies': train_accuracies,
                'val_accuracies': val_accuracies
            })
```

```
Training with learning rate=0.001, batch_size=64, num_epochs=5
Training on device: cpu
Training with learning rate=0.001, batch_size=64, num_epochs=10
Training on device: cpu
Training with learning rate=0.001, batch_size=128, num_epochs=5
Training on device: cpu
Training with learning rate=0.001, batch_size=128, num_epochs=10
Training on device: cpu
Training with learning rate=0.001, batch_size=256, num_epochs=5
Training on device: cpu
Training with learning rate=0.001, batch_size=256, num_epochs=10
Training on device: cpu
```

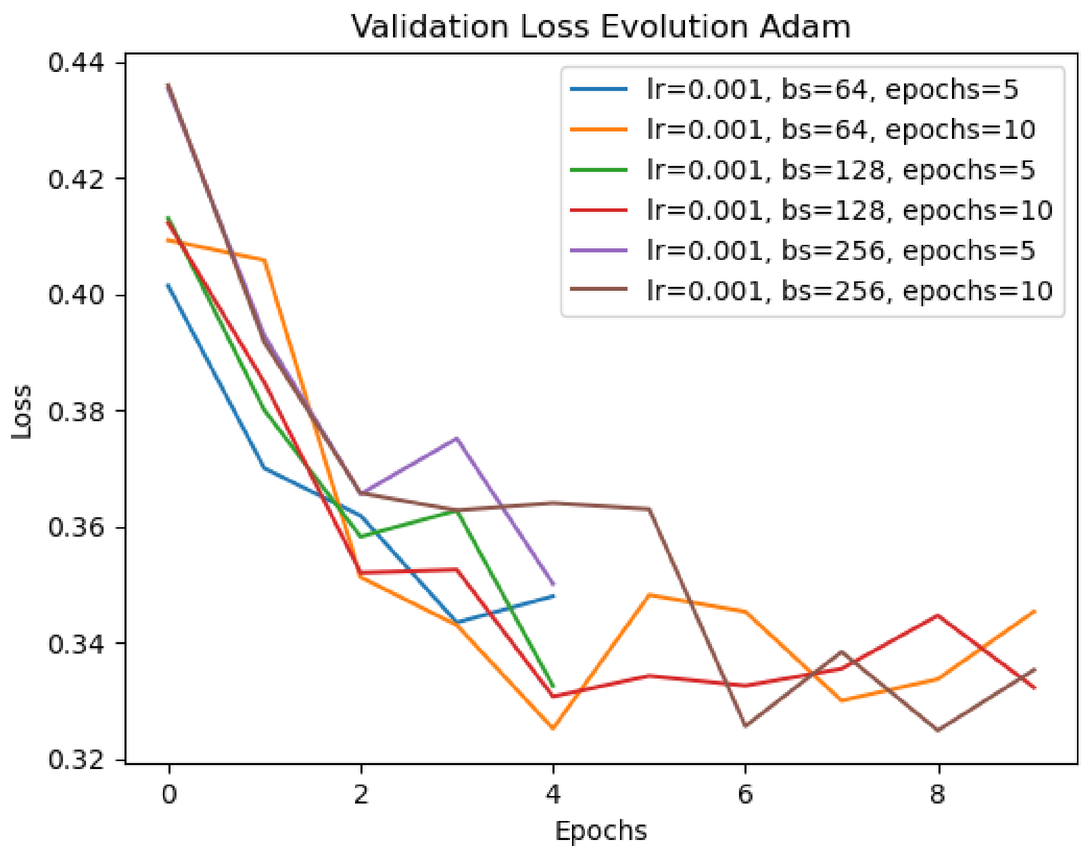
[ ]:

```
# TODO: Plot evolution of validation accuracy for each hyperparameter setting
for result in adam_results:
    plt.plot(result['val_accuracies'], label=f"lr={result['learning_rate']},␣
↪bs={result['batch_size']}, epochs={result['num_epochs']}")
plt.title('Validation Accuracy Evolution Adam')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

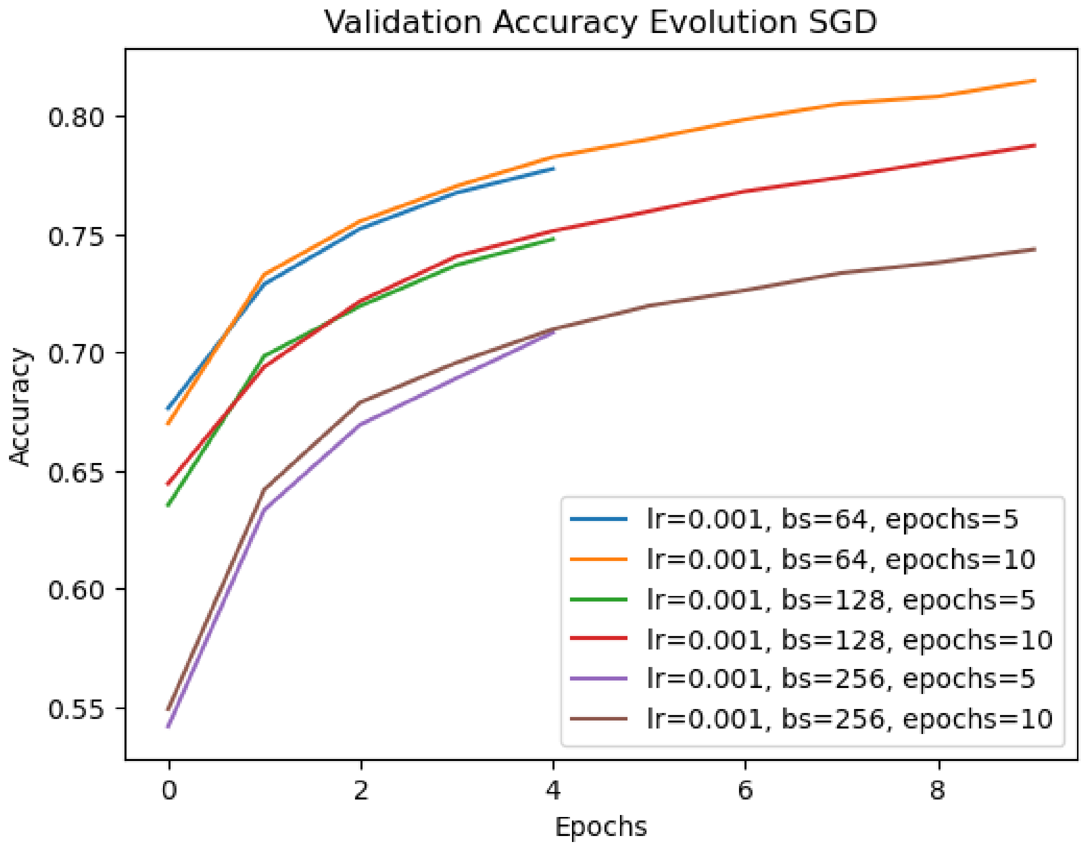
```
plt.show()
```



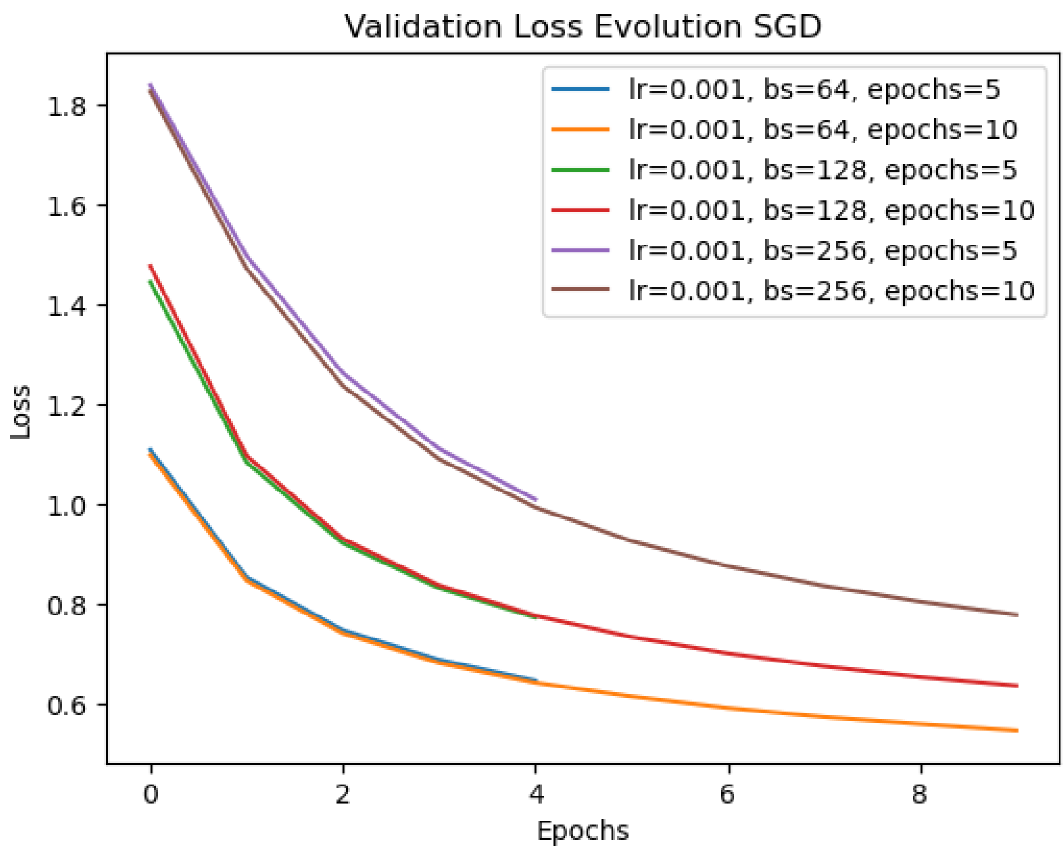

[ ]:
```
# TODO: Plot evolution of validation accuracy for each hyperparameter setting
for result in adam_results:
    plt.plot(result['val_losses'], label=f"lr={result['learning_rate']},␣
 ↪bs={result['batch_size']}, epochs={result['num_epochs']}")
plt.title('Validation Loss Evolution Adam')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




[ ]:
```
# TODO: Plot evolution of validation accuracy for each hyperparameter setting
for result in sgd_results:
    plt.plot(result['val_accuracies'], label=f"lr={result['learning_rate']},␣
 ↪bs={result['batch_size']}, epochs={result['num_epochs']}")
plt.title('Validation Accuracy Evolution SGD')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```




[ ]:
```
# TODO: Plot evolution of validation accuracy for each hyperparameter setting
for result in sgd_results:
    plt.plot(result['val_losses'], label=f"lr={result['learning_rate']}, bs={result['batch_size']}, epochs={result['num_epochs']}")
plt.title('Validation Loss Evolution SGD')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




**TODO**: Justify which batch size and learning rate combination you will go with.

#### 1.3.5 2.4

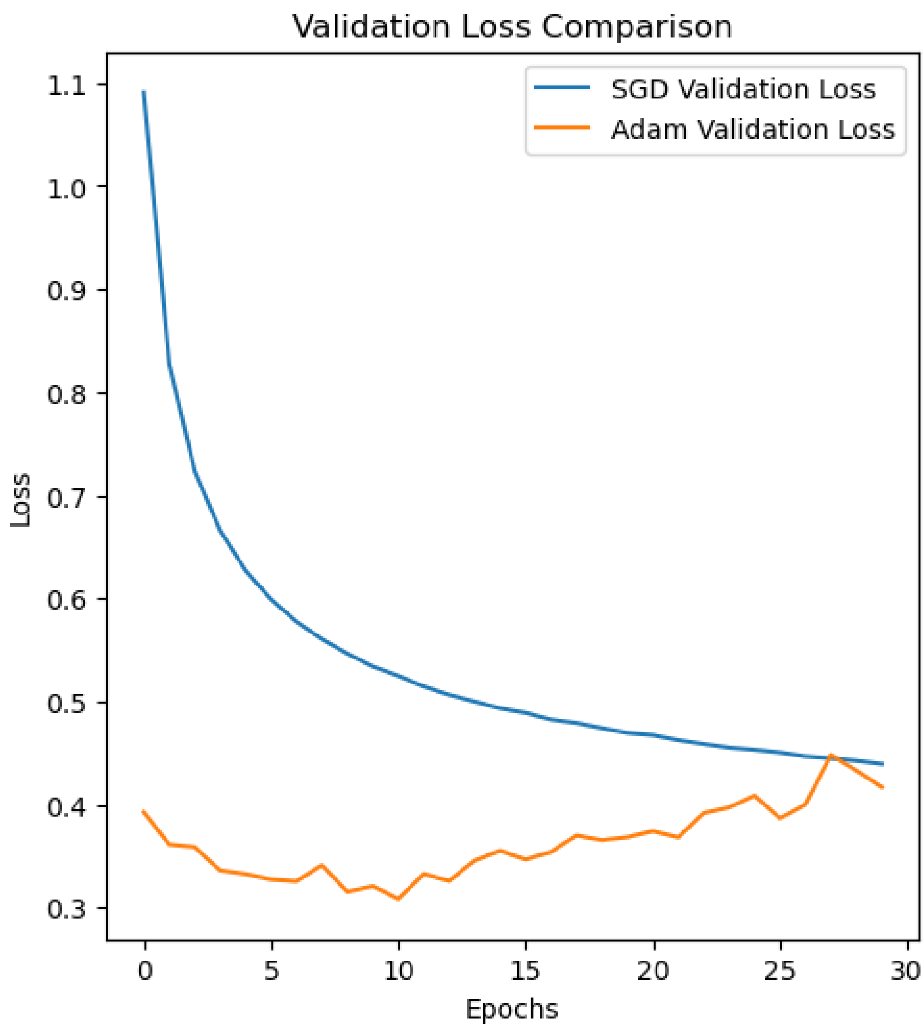
[25]:
```
# TODO: Train model w/ best hyperparameters for SGD and compare to default Adam␣
↪optimizer
model_adam = MLP()
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64,␣
↪shuffle=True)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=64, shuffle=False)
optimizer = torch.optim.Adam(model_adam.parameters(), lr=lr)

adam_results_best_hyper_param = []
sgd_results_best_hyper_param = []

train_losses, val_losses, train_accuracies, val_accuracies = train_model(
    model_adam, criterion, optimizer, train_loader, val_loader, 30
)

adam_results_best_hyper_param.append({
```

```
    'learning_rate': lr,
    'batch_size': batch_size,
    'num_epochs': num_epochs,
    'train_losses': train_losses,
    'val_losses': val_losses,
    'train_accuracies': train_accuracies,
    'val_accuracies': val_accuracies
})

model_sgd = MLP()
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64,␣
 ↪shuffle=True)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=64, shuffle=False)
optimizer = torch.optim.SGD(model_sgd.parameters(), lr=lr)

train_losses, val_losses, train_accuracies, val_accuracies = train_model(
    model_sgd, criterion, optimizer, train_loader, val_loader, 30
)

sgd_results_best_hyper_param.append({
    'learning_rate': lr,
    'batch_size': batch_size,
    'num_epochs': num_epochs,
    'train_losses': train_losses,
    'val_losses': val_losses,
    'train_accuracies': train_accuracies,
    'val_accuracies': val_accuracies
})
```

```
Training on device: cpu
Training on device: cpu
```

[26]:
```
# TODO: Plot "learning curves" of the best SGD model and the Adam model
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sgd_results_best_hyper_param[-1]['val_losses'], label='SGD Validation␣
 ↪Loss')
plt.plot(adam_results_best_hyper_param[-1]['val_losses'], label='Adam␣
 ↪Validation Loss')
plt.title('Validation Loss Comparison')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

[26]:
```
<matplotlib.legend.Legend at 0x323305110>
```




**TODO**: Based on plots, compare (mini-batch) SGD and Adam, select overall best Model

[27]:
```
#TODO: Evaluate the best model on the test set, print the test/train/validation␣
↪accuracy
print("Evaluating on test set...")
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
model.eval()
test_correct = 0
test_total = 0
with torch.no_grad():
    for images, labels in data_loader_test:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        test_correct += (outputs.argmax(dim=1) == labels).sum().item()
```

```
        test_total += labels.size(0)
test_accuracy = test_correct / test_total
print(f"Test Accuracy: {test_accuracy:.4f}")
```

```
Evaluating on test set…
Test Accuracy: 0.8900
```

**TODO**: Briefly discuss your results.

We set the batch size as 64 and run 30 epoches for SGD and Adam respectively. The loss curve of SGD keeps going down whilst the curve of Adam oscillates. The lowest loss (ca. 0.3) for Adam is observed at epoch=10. Thereafter, the loss has an increasing tendency, probably because of overfitting. Besides, the loss of SGD is almost everywhere bigger than that of Adam when epoch lies in the interval between 0 and 30$. However, we can expect it to outperform Adam, if this tendency goes on.