MLE Sheet 5

June 2025

Exercise 1: Optimization and Training Tricks

Task 1

Batch Gradient Descent (BGD)

Computes the gradient of the cost function using the entire training dataset.

Update Rule:

$$\theta := \theta - \eta \cdot \nabla J(\theta)$$

where η is the learning rate, and $\nabla J(\theta)$ is the gradient computed on the full dataset.

Advantages:

- Produces stable and accurate gradient estimates.
- Smooth convergence in convex problems.

Disadvantages:

- Computationally expensive for large datasets (each update requires a full pass over the data).
- Slow iteration updates.
- Not suitable for online or streaming data.

Stochastic Gradient Descent (SGD)

Computes the gradient using only one randomly chosen sample at each update step.

Update Rule:

$$\theta := \theta - \eta \cdot \nabla J(\theta; x_i, y_i)$$

Advantages:

- Very fast per update (only one sample needed).
- Can escape local minima due to high variance in updates.

Disadvantages:

- Noisy updates can cause the algorithm to oscillate around the minimum.
- May require more epochs to converge.
- Less stable; harder to choose an optimal learning rate.

Mini-batch SGD

A compromise between BGD and SGD; updates are based on the gradient from a small batch of samples.

Update Rule:

$$\theta := \theta - \eta \cdot \nabla J(\theta; \text{mini-batch})$$

Advantages:

- Faster updates than BGD, more stable than SGD.
- Leverages parallelism (vectorization on GPUs).
- Good balance between noisy updates and computational efficiency.

Disadvantages:

- Still somewhat noisy, depending on batch size.
- Performance depends on mini-batch size (too small: noisy; too big: slow).

Task 2

A

A fixed learning rate may not be ideal throughout the entire training process because:

- In the early stages of training, a high learning rate helps the model make large, rapid improvements.
- In later stages, the same high learning rate can cause the optimization to overshoot the minima or oscillate, preventing convergence.
- Conversely, a learning rate that is too low in the early stages results in slow progress.

Thus, dynamically adjusting the learning rate helps the model transition from rapid exploration to fine-tuning. This can be visualized by a loss landscape where:

- Large steps are useful at the beginning to escape plateaus or poor local minima.
- Small steps are necessary near the optimum to avoid bouncing around.

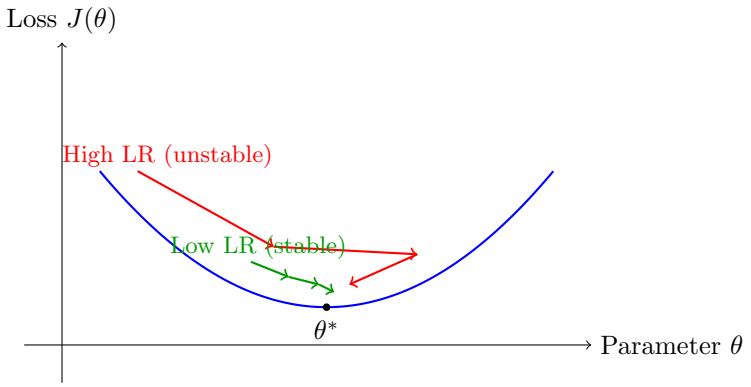


Figure 1: Effect of Learning Rate on Optimization Trajectory

B

A learning rate schedule is a strategy where the learning rate η is adjusted during training based on the epoch number or iteration step. The goal is to improve convergence speed and accuracy by adapting the step size to the training stage.

Example: Exponential Decay

One commonly used schedule is the *exponential decay*, where the learning rate decreases exponentially over time:

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

- η_0: initial learning rate
- λ: decay rate
- t: current epoch or iteration number

Effect on Training:

- Initially, the high learning rate allows for fast convergence.
- As training progresses, the reduced learning rate allows the model to fine-tune its parameters and converge to a minimum with more stability.

Other popular schedules include:

- **Step decay**: reduce the learning rate by a factor every few epochs.
- **Cosine annealing**: slowly reduces the learning rate following a cosine curve, possibly with warm restarts.
- **Polynomial decay** and **1/t decay**: use different mathematical functions to decrease the rate.

Learning rate schedules are critical in helping models escape poor local minima early on and then settle efficiently into good minima later in training.

Exercise 1: Tasks 3 and 4

Task 3: Validation and Hyperparameter Tuning

(a) Roles of the datasets

- **Training set:** Used to adjust the model's parameters (weights and biases). The model sees these examples during training and tries to fit them.
- **Validation set:** Used for tuning hyperparameters (e.g., learning rate, batch size) and for model selection. The model doesn't see these examples during training updates but uses them to evaluate performance and adjust hyperparameters iteratively.
- **Test set:** Used to assess the final generalization performance. It should be used only once at the end, not for iterative refinement, to avoid introducing bias in the evaluation.

(b) Why not use the test set for refinement?

If the test set is used to guide decisions (like hyperparameter selection), it effectively becomes part of the training process, leading to an optimistic and biased estimate of the model's true generalization ability. The validation set helps prevent this because it provides an unbiased performance estimate during development. Using the test set repeatedly would underestimate the model's variance (since you're overfitting to that test set), thus leading to misleading performance metrics.

(c) Grid search and the validation set

Concept: Grid search involves defining a set of possible hyperparameter values (a grid) and systematically evaluating all combinations. For example, you might search over different learning rates, batch sizes, or number of layers.

Role of validation set: For each combination of hyperparameters, the model is trained on the training set and evaluated on the validation set. The validation performance guides the choice of the best hyperparameter combination (the one with the highest validation accuracy, for instance).

Task 4: Advanced Optimizers

(a) Core mechanisms

- **i. RMSProp:** RMSProp adapts the learning rate for each parameter by maintaining a moving average of the squared gradients:

  $$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$

  The update step for a parameter:

  $$\Delta w_t = -\alpha \frac{g_t}{\sqrt{v_t + \epsilon}}$$

  This helps prevent the learning rate from being too large for parameters with large gradients, stabilizing updates.

- **ii. Momentum:** Momentum accelerates convergence by smoothing the update direction:

  $$m_t = \beta m_{t-1} + (1-\beta) g_t$$

  The parameter update uses this moving average of past gradients:

  $$\Delta w_t = -\alpha m_t$$

  This allows the optimizer to keep moving in relevant directions even when gradients oscillate.

(b) Adam update step

Given:

$$\beta_1 = 0.9, \quad \beta_2 = 0.99, \quad \alpha = 0.01, \quad \epsilon = 10^{-8}$$
$$m_{t-1} = 0.5, \quad v_{t-1} = 0.2, \quad g_t = 2.0$$

1. **Updated first moment:**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t = 0.9 \cdot 0.5 + 0.1 \cdot 2.0 = 0.45 + 0.2 = 0.65$$

2. **Updated second moment:**

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 = 0.99 \cdot 0.2 + 0.01 \cdot (2.0)^2 = 0.198 + 0.04 = 0.238$$

3. **Update step:**

$$\Delta w_t = -\alpha \frac{m_t}{\sqrt{v_t} + \epsilon} = -0.01 \cdot \frac{0.65}{\sqrt{0.238} + 10^{-8}}$$
$$\sqrt{0.238} \approx 0.4879$$
$$\Delta w_t \approx -0.01 \cdot \frac{0.65}{0.4879} = -0.01 \cdot 1.332 \approx -0.01332$$

(c) Effect of larger second moment history (comparing with w')

- $v'_{t-1} = 20$ is much larger than $v_{t-1} = 0.2$.
- Consequently, v'_t will also be much larger than v_t.
- The denominator $\sqrt{v'_t}$ becomes large, so the update step magnitude $|\Delta w'_t|$ will be **smaller** than $|\Delta w_t|$.

Implication: Adam automatically adapts the learning rate for each parameter. Parameters with large past gradients (large v_t) get smaller updates, helping prevent overshooting in areas with large variance. This stabilizes training and improves convergence.

Machine Learning Essentials SS25 - Exercise Sheet 5

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.

Exercise 2

In [2]:
```
import torch
print(torch.cuda.is_available())
print(torch.cuda.get_device_name(0))
```

```
True
Tesla T4
```

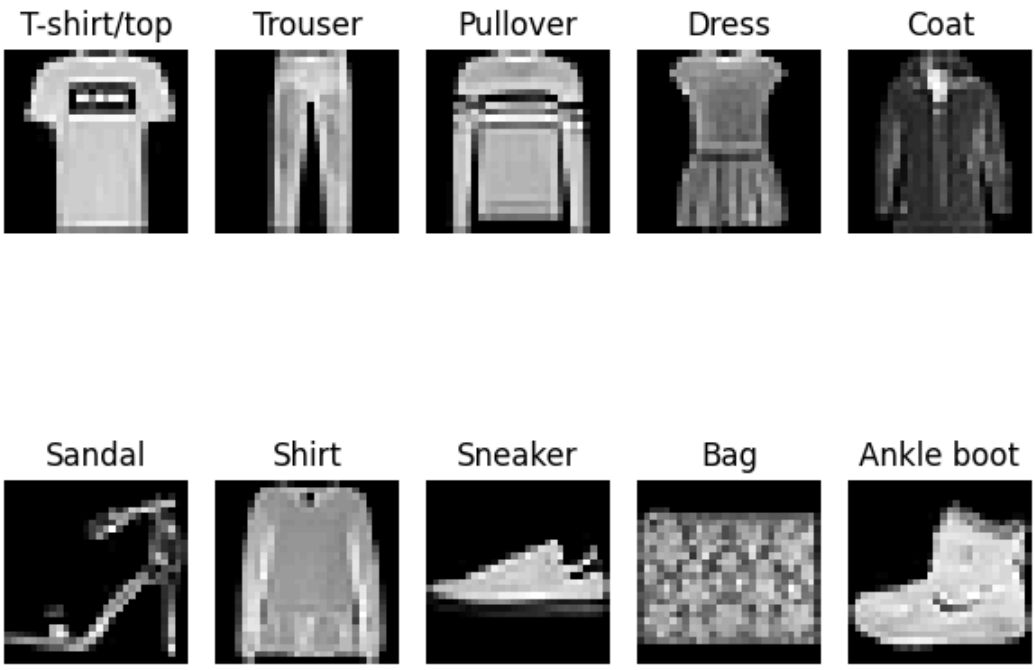
In [3]:
```
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, random_split
from torchvision import transforms, datasets
# TODO: Import the stuff you need from torch and torchvision

# """
# If you stay in ML-related fields, you will likely be working on a server or cluste
r. If you do so,
# always remember to set the number of CPU (or even GPU) threads you're using, as Jup
yter notebooks or Python scripts
# might sometimes use all available threads by default, which will lead to unhappy co
lleagues 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
```

2.1

In [4]:
```
# TODO: Define transformations
# Given statistics of training set:
mu_train = 0.286
std_train = 0.353
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((mu_train,), (std_train,))
])

# TODO: Load FashionMNIST train/testsets
train_dataset_full = datasets.FashionMNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)
test_dataset = datasets.FashionMNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)

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

```
100%|██████████| 26.4M/26.4M [00:02<00:00, 13.2MB/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 208kB/s]
100%|██████████| 4.42M/4.42M [00:01<00:00, 3.92MB/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 9.40MB/s]

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

In [5]:
```
# TODO: Create 5x2 subplot grid w/ example image for each class
class_names = train_dataset_full.classes
fig, axes = plt.subplots(2, 5, figsize=(6, 6))
axes = axes.flatten()

shown = set()
for img, label in train_dataset_full:
    if label not in shown:
        ax = axes[label]
        ax.imshow(img.squeeze(0) * std_train + mu_train, cmap='gray')
        ax.set_title(class_names[label])
        ax.axis('off')
        shown.add(label)
    if len(shown) == 10:
        break

plt.tight_layout()
plt.show()
```



In [6]:
```
# TODO: Create a validation set from the training set
val_size = int(len(train_dataset_full) * 0.1)
train_size = len(train_dataset_full) - val_size
train_dataset, val_dataset = random_split(
    train_dataset_full,
    [train_size, val_size],
    generator=torch.Generator().manual_seed(42)
)
```

2.2

In [7]:
```
# TODO: Define your model architecture: A class called MLP that inherits from nn.Modu
le
class MLP(nn.Module):
    def __init__(self, input_size=28*28, hidden_sizes=[256, 128], num_classes=10, dro
pout=0.2):
        super().__init__()
        layers, in_features = [], input_size
        for h in hidden_sizes:
            layers += [
                nn.Linear(in_features, h),
                nn.ReLU(),
                nn.Dropout(dropout)
            ]
            in_features = h
        layers.append(nn.Linear(in_features, num_classes))
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.network(x)
# TODO: Define appropriate loss
criterion = nn.CrossEntropyLoss()
```

2.3

In [8]:
```
BATCH_SIZE_DEFAULT = 128  # 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
train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE_DEFAULT,
    shuffle=True,
    num_workers=2,
    pin_memory=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE_DEFAULT,
    shuffle=False,
    num_workers=2,
    pin_memory=True
)
test_loader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE_DEFAULT,
    shuffle=False,
    num_workers=2,
    pin_memory=True
)
```

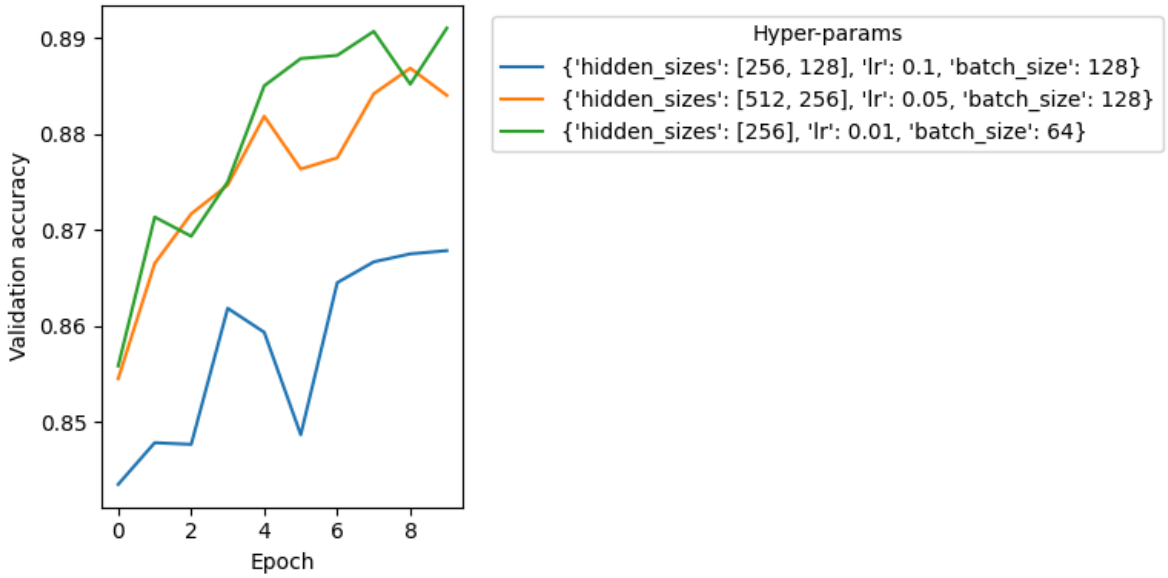
In [9]:
```
def calculate_accuracy(outputs, labels):
    """
    Calculate accuracy given model outputs and true labels.
    """
    _, predicted = torch.max(outputs.data, 1) # Prediction = class with highest outpu
t probability
    total = labels.size(0)
    correct = (predicted == labels).sum().item() #.item() converts a single-element t
ensor 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 P
yTorch 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 i
n 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):
        #Training
        model.train()
        run_loss, run_correct, run_total = 0.0, 0, 0
        for X, y in train_loader:
            X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            outputs = model(X)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer.step()

            run_loss += loss.item() * y.size(0)
            _, preds = torch.max(outputs, 1)
            run_correct += (preds == y).sum().item()
            run_total += y.size(0)

        train_losses.append(run_loss / run_total)
        train_accuracies.append(run_correct / run_total)

        # Validation
        model.eval()
        v_loss, v_correct, v_total = 0.0, 0, 0
        with torch.no_grad():
            for X, y in val_loader:
                X, y = X.to(device), y.to(device)
                outputs = model(X)
                loss = criterion(outputs, y)

                v_loss += loss.item() * y.size(0)
                _, preds = torch.max(outputs, 1)
                v_correct += (preds == y).sum().item()
                v_total += y.size(0)

        val_losses.append(v_loss / v_total)
        val_accuracies.append(v_correct / v_total)

        print(f"Epoch {epoch+1}/{num_epochs} |"
              f" train acc: {train_accuracies[-1]:.4f} |"
              f" val acc: {val_accuracies[-1]:.4f}")

    return train_losses, val_losses, train_accuracies, val_accuracies
```

2.4

In [10]:
```
# TODO: Define hyperparameter grid for tuning
hyperparameter_grid = [
    {"hidden_sizes": [256, 128], "lr": 0.1, "batch_size": 128},
    {"hidden_sizes": [512, 256], "lr": 0.05, "batch_size": 128},
    {"hidden_sizes": [256], "lr": 0.01, "batch_size": 64},
]
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
results = []
# TODO: For each hyperparameter setting, instantiate model&optimizer,
# train the model, and store the results for evaluation later

for cfg in hyperparameter_grid:
    tl = DataLoader(train_dataset, batch_size=cfg["batch_size"],
                    shuffle=True, num_workers=2, pin_memory=True)
    vl = DataLoader(val_dataset, batch_size=cfg["batch_size"],
                    shuffle=False, num_workers=2, pin_memory=True)

    model = MLP(hidden_sizes=cfg["hidden_sizes"]).to(device)
    optim_cfg = optim.SGD(model.parameters(), lr=cfg["lr"], momentum=0.9)

    _, _, _, val_accs = train_model(model, criterion, optim_cfg, tl, vl, num_epochs=1
0)

    results.append({"cfg": cfg, "val_accs": val_accs})
```

```
Training on device: cuda
Epoch 1/10 | train acc: 0.7862 | val acc: 0.8435
Epoch 2/10 | train acc: 0.8285 | val acc: 0.8478
Epoch 3/10 | train acc: 0.8429 | val acc: 0.8477
Epoch 4/10 | train acc: 0.8514 | val acc: 0.8618
Epoch 5/10 | train acc: 0.8571 | val acc: 0.8593
Epoch 6/10 | train acc: 0.8621 | val acc: 0.8487
Epoch 7/10 | train acc: 0.8606 | val acc: 0.8645
Epoch 8/10 | train acc: 0.8663 | val acc: 0.8667
Epoch 9/10 | train acc: 0.8678 | val acc: 0.8675
Epoch 10/10 | train acc: 0.8726 | val acc: 0.8678
Training on device: cuda
Epoch 1/10 | train acc: 0.8020 | val acc: 0.8545
Epoch 2/10 | train acc: 0.8553 | val acc: 0.8665
Epoch 3/10 | train acc: 0.8691 | val acc: 0.8717
Epoch 4/10 | train acc: 0.8781 | val acc: 0.8747
Epoch 5/10 | train acc: 0.8824 | val acc: 0.8818
Epoch 6/10 | train acc: 0.8895 | val acc: 0.8763
Epoch 7/10 | train acc: 0.8928 | val acc: 0.8775
Epoch 8/10 | train acc: 0.8971 | val acc: 0.8842
Epoch 9/10 | train acc: 0.9001 | val acc: 0.8868
Epoch 10/10 | train acc: 0.9023 | val acc: 0.8840
Training on device: cuda
Epoch 1/10 | train acc: 0.8141 | val acc: 0.8558
Epoch 2/10 | train acc: 0.8612 | val acc: 0.8713
Epoch 3/10 | train acc: 0.8717 | val acc: 0.8693
Epoch 4/10 | train acc: 0.8804 | val acc: 0.8750
Epoch 5/10 | train acc: 0.8856 | val acc: 0.8850
Epoch 6/10 | train acc: 0.8925 | val acc: 0.8878
Epoch 7/10 | train acc: 0.8965 | val acc: 0.8882
Epoch 8/10 | train acc: 0.9016 | val acc: 0.8907
Epoch 9/10 | train acc: 0.9036 | val acc: 0.8852
Epoch 10/10 | train acc: 0.9064 | val acc: 0.8910
```

In [11]:
```
# TODO: Plot evolution of validation accuracy for each hyperparameter setting
plt.figure(figsize=(8, 4))
for res in results:
    plt.plot(res["val_accs"], label=str(res["cfg"]))
plt.xlabel("Epoch"); plt.ylabel("Validation accuracy")
plt.legend(title="Hyper-params", bbox_to_anchor=(1.05, 1))
plt.tight_layout(); plt.show()
```



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

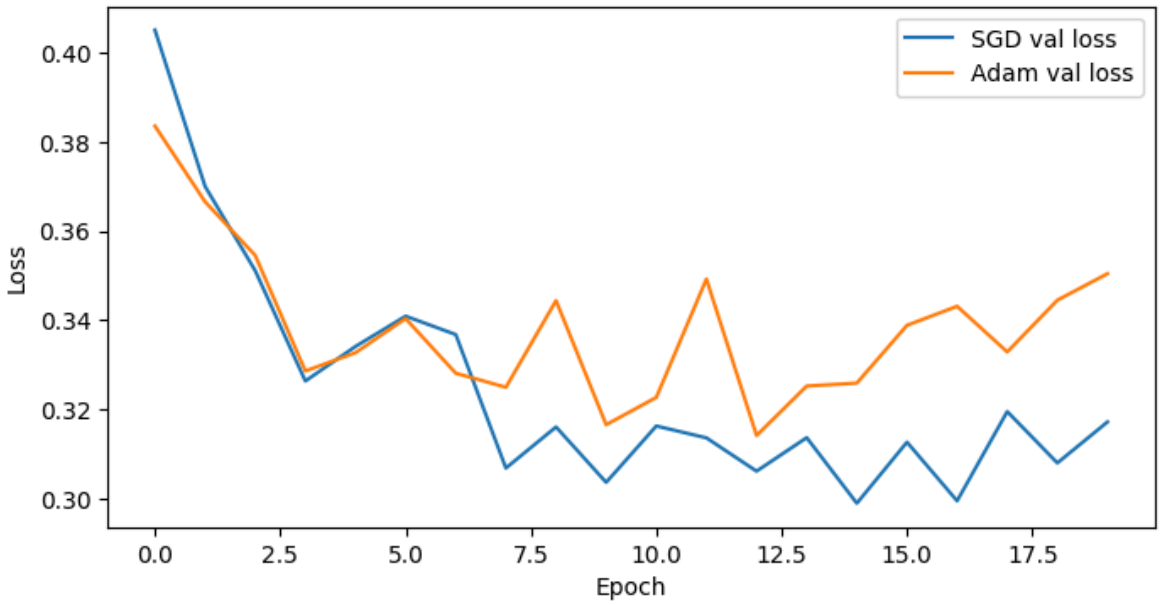
Answer: I'd choose the batch_size = 64 with lr = 0.01 and hidden size = 256. It gave the highest validation accuracy accross the three. It climbs steadily and keeps on improving, even though the batch size is small. Also the lr is small which I believe pairs with noisier grads and hence it avoids overshooting which we can see in the blue run.

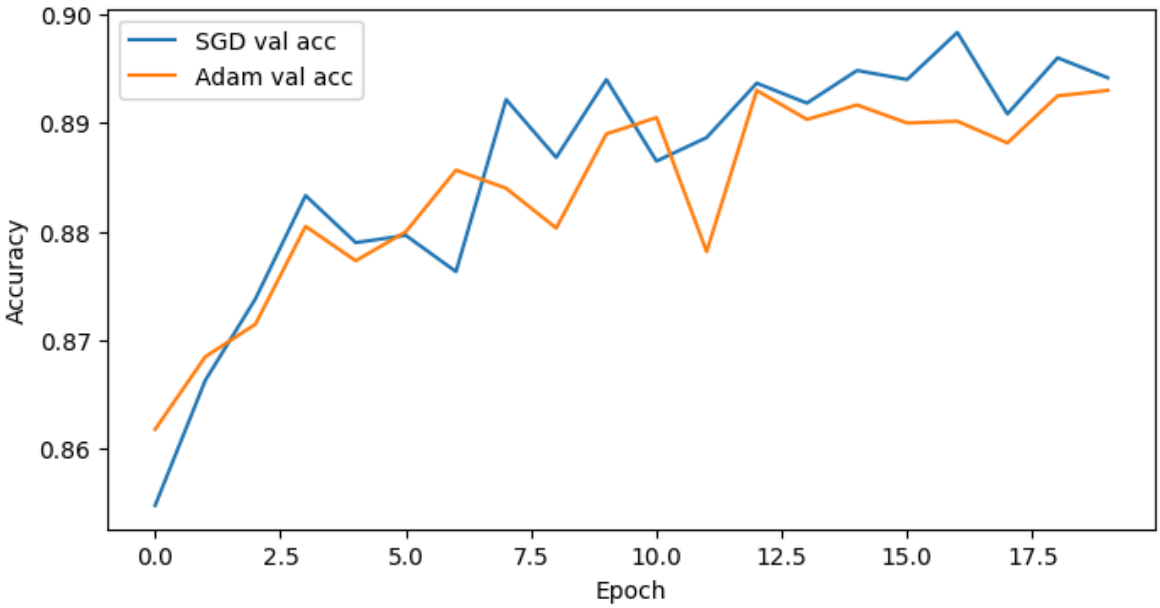
2.4

In [12]:
```
# TODO: Train model w/ best hyperparameters for SGD and compare to default Adam optim
izer

best_cfg = max(results, key=lambda r: r["val_accs"][-1])["cfg"]

tl_best = DataLoader(train_dataset, batch_size=best_cfg["batch_size"],
                     shuffle=True, num_workers=2, pin_memory=True)
vl_best = DataLoader(val_dataset, batch_size=best_cfg["batch_size"],
                     shuffle=False, num_workers=2, pin_memory=True)

best_sgd = MLP(hidden_sizes=best_cfg["hidden_sizes"]).to(device)
opt_sgd  = optim.SGD(best_sgd.parameters(), lr=best_cfg["lr"], momentum=0.9)
sgd_tr_loss, sgd_val_loss, sgd_tr_acc, sgd_val_acc = train_model(
    best_sgd, criterion, opt_sgd, tl_best, vl_best, num_epochs=20)

adam = MLP(hidden_sizes=best_cfg["hidden_sizes"]).to(device)
opt_adam = optim.Adam(adam.parameters(), lr=1e-3)
adam_tr_loss, adam_val_loss, adam_tr_acc, adam_val_acc = train_model(
    adam, criterion, opt_adam, tl_best, vl_best, num_epochs=20)
```

```
Training on device: cuda
Epoch 1/20 | train acc: 0.8136 | val acc: 0.8548
Epoch 2/20 | train acc: 0.8600 | val acc: 0.8663
Epoch 3/20 | train acc: 0.8729 | val acc: 0.8738
Epoch 4/20 | train acc: 0.8815 | val acc: 0.8833
Epoch 5/20 | train acc: 0.8870 | val acc: 0.8790
Epoch 6/20 | train acc: 0.8904 | val acc: 0.8797
Epoch 7/20 | train acc: 0.8952 | val acc: 0.8763
Epoch 8/20 | train acc: 0.9001 | val acc: 0.8922
Epoch 9/20 | train acc: 0.9039 | val acc: 0.8868
Epoch 10/20 | train acc: 0.9064 | val acc: 0.8940
Epoch 11/20 | train acc: 0.9099 | val acc: 0.8865
Epoch 12/20 | train acc: 0.9129 | val acc: 0.8887
Epoch 13/20 | train acc: 0.9144 | val acc: 0.8937
Epoch 14/20 | train acc: 0.9173 | val acc: 0.8918
Epoch 15/20 | train acc: 0.9196 | val acc: 0.8948
Epoch 16/20 | train acc: 0.9218 | val acc: 0.8940
Epoch 17/20 | train acc: 0.9248 | val acc: 0.8983
Epoch 18/20 | train acc: 0.9245 | val acc: 0.8908
Epoch 19/20 | train acc: 0.9275 | val acc: 0.8960
Epoch 20/20 | train acc: 0.9290 | val acc: 0.8942
Training on device: cuda
Epoch 1/20 | train acc: 0.8256 | val acc: 0.8618
Epoch 2/20 | train acc: 0.8628 | val acc: 0.8685
Epoch 3/20 | train acc: 0.8720 | val acc: 0.8715
Epoch 4/20 | train acc: 0.8820 | val acc: 0.8805
Epoch 5/20 | train acc: 0.8860 | val acc: 0.8773
Epoch 6/20 | train acc: 0.8934 | val acc: 0.8800
Epoch 7/20 | train acc: 0.8963 | val acc: 0.8857
Epoch 8/20 | train acc: 0.9002 | val acc: 0.8840
Epoch 9/20 | train acc: 0.9031 | val acc: 0.8803
Epoch 10/20 | train acc: 0.9057 | val acc: 0.8890
Epoch 11/20 | train acc: 0.9072 | val acc: 0.8905
Epoch 12/20 | train acc: 0.9113 | val acc: 0.8782
Epoch 13/20 | train acc: 0.9130 | val acc: 0.8930
Epoch 14/20 | train acc: 0.9159 | val acc: 0.8903
Epoch 15/20 | train acc: 0.9189 | val acc: 0.8917
Epoch 16/20 | train acc: 0.9202 | val acc: 0.8900
Epoch 17/20 | train acc: 0.9234 | val acc: 0.8902
Epoch 18/20 | train acc: 0.9227 | val acc: 0.8882
Epoch 19/20 | train acc: 0.9269 | val acc: 0.8925
Epoch 20/20 | train acc: 0.9266 | val acc: 0.8930
```

In [13]:
```
# TODO: Plot "learning curves" of the best SGD model and the Adam model
plt.figure(figsize=(8, 4))
plt.plot(sgd_val_loss,  label="SGD val loss")
plt.plot(adam_val_loss, label="Adam val loss")
plt.xlabel("Epoch"); plt.ylabel("Loss"); plt.legend(); plt.show()

plt.figure(figsize=(8, 4))
plt.plot(sgd_val_acc,  label="SGD val acc")
plt.plot(adam_val_acc, label="Adam val acc")
plt.xlabel("Epoch"); plt.ylabel("Accuracy"); plt.legend(); plt.show()
```





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

In [14]:
```
#TODO: Evaluate the best model on the test set, print the test/train/validation accur
acy
final_model = best_sgd if sgd_val_acc[-1] >= adam_val_acc[-1] else adam

def eval_acc(model, loader):
    model.eval()
    correct = total = 0
    with torch.no_grad():
        for X, y in loader:
            X, y = X.to(device), y.to(device)
            _, preds = torch.max(model(X), 1)
            total += y.size(0)
            correct += (preds == y).sum().item()
    return correct / total

print("Train accuracy:      ", eval_acc(final_model, tl_best))
print("Validation accuracy: ", eval_acc(final_model, vl_best))
print("Test accuracy:       ", eval_acc(final_model, test_loader))
```

```
Train accuracy:       0.9418703703703704
Validation accuracy:  0.8941666666666667
Test accuracy:        0.8857
```

Q: Briefly discuss your results.

Answer

- With exactly the same network and data-handling, SGD (blue) edges out ADAM(orange) throughout most of training. Its val-loss keeps trending downward and its val-accuracy peaks just under 0.90, whereas Adam levels off ≈0.002-0.003 higher in loss and ≈0.5 pp lower in accuracy.
- SGD's accuracy curve is smoother and keeps improving, while Adam oscillates after epoch 10. That indicates the Adam LR (1×10^{-3}) is on the high side once the model reaches better parts of the landscape.
- For this architecture and hyper-parameter budget, SGD with momentum is the better choice—it generalises slightly better, is more stable in the late epochs, and costs no extra compute. Adam remains a competitive fallback if we later switch to deeper nets or noisier tasks, but here SGD should be the default.