Lab 8

Aim: Use deep neural networks to design agents that can learn to take actions in a simulated environment.

```
import collections
!pip install datasets
import datasets
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchtext
import tqdm

seed = 1234

np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.backends.cudnn.deterministic = True

train_data, test_data = datasets.load_dataset("imdb", split=["train",
"test"])
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/
_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your
settings tab (https://huggingface.co/settings/tokens), set it as
secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to
access public models or datasets.
  warnings.warn(
```

{"model_id":"0ee7b6c7c2de47a49e1c654cb97311fa","version_major":2,"version_minor":0}

{"model_id":"5af17e79c71d4b3a95851d86223e8423","version_major":2,"version_minor":0}

{"model_id":"0a7eb10c61de4be494a61691de851234","version_major":2,"version_minor":0}

{"model_id":"678395ba8047424daa977a18e91b736f","version_major":2,"version_minor":0}

{"model_id":"44ea1d471fd942d5b4dab79a80df0246","version_major":2,"version_minor":0}

{"model_id":"aeb039f2640b4ca993a353dfea3a6b6f","version_major":2,"version_minor":0}

{"model_id":"f12450f27a5c4eb3b02b2f4d774c380c","version_major":2,"version_minor":0}

```
tokenizer = torchtext.data.utils.get_tokenizer("basic_english")

def tokenize_example(example, tokenizer, max_length):
    tokens = tokenizer(example["text"])[:max_length]
    return {"tokens": tokens}

max_length = 256

train_data = train_data.map(
    tokenize_example, fn_kwargs={"tokenizer": tokenizer, "max_length": max_length}
)
test_data = test_data.map(
    tokenize_example, fn_kwargs={"tokenizer": tokenizer, "max_length": max_length}
)
```

{"model_id":"e37339836ac6409d97b238998f60ca5e","version_major":2,"version_minor":0}

{"model_id":"5611ecd225714ca68465a1d2f587f9b5","version_major":2,"version_minor":0}

```
test_size = 0.25

train_valid_data = train_data.train_test_split(test_size=test_size)
train_data = train_valid_data["train"]
valid_data = train_valid_data["test"]

min_freq = 5
special_tokens = ["<unk>", "<pad>"]

vocab = torchtext.vocab.build_vocab_from_iterator(
    train_data["tokens"],
    min_freq=min_freq,
    specials=special_tokens,
)

unk_index = vocab["<unk>"]
pad_index = vocab["<pad>"]

vocab.set_default_index(unk_index)
```

```python
def numericalize_example(example, vocab):
    ids = vocab.lookup_indices(example["tokens"])
    return {"ids": ids}

train_data = train_data.map(numericalize_example, fn_kwargs={"vocab":
vocab})
valid_data = valid_data.map(numericalize_example, fn_kwargs={"vocab":
vocab})
test_data = test_data.map(numericalize_example, fn_kwargs={"vocab":
vocab})
```

{"model_id":"8f1ff30589634f19a6a2b8fa7c1b1b1d","version_major":2,"vers
ion_minor":0}

{"model_id":"6899b651a87842b1a8b94ea85b5f7b64","version_major":2,"vers
ion_minor":0}

{"model_id":"6e7eaf0f52ac497094a7d9041e317d6f","version_major":2,"vers
ion_minor":0}

```python
train_data = train_data.with_format(type="torch", columns=["ids",
"label"])
valid_data = valid_data.with_format(type="torch", columns=["ids",
"label"])
test_data = test_data.with_format(type="torch", columns=["ids",
"label"])

def get_collate_fn(pad_index):
    def collate_fn(batch):
        batch_ids = [i["ids"] for i in batch]
        batch_ids = nn.utils.rnn.pad_sequence(
            batch_ids, padding_value=pad_index, batch_first=True
        )
        batch_label = [i["label"] for i in batch]
        batch_label = torch.stack(batch_label)
        batch = {"ids": batch_ids, "label": batch_label}
        return batch

    return collate_fn

def get_data_loader(dataset, batch_size, pad_index, shuffle=False):
    collate_fn = get_collate_fn(pad_index)
    data_loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=batch_size,
        collate_fn=collate_fn,
        shuffle=shuffle,
    )
    return data_loader
```

```python
batch_size = 512

train_data_loader = get_data_loader(train_data, batch_size, pad_index,
shuffle=True)
valid_data_loader = get_data_loader(valid_data, batch_size, pad_index)
test_data_loader = get_data_loader(test_data, batch_size, pad_index)

class CNN(nn.Module):
    def __init__(
        self,
        vocab_size,
        embedding_dim,
        n_filters,
        filter_sizes,
        output_dim,
        dropout_rate,
        pad_index,
    ):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim,
padding_idx=pad_index)
        self.convs = nn.ModuleList(
            [
                nn.Conv1d(embedding_dim, n_filters, filter_size)
                for filter_size in filter_sizes
            ]
        )
        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, ids):
        # ids = [batch size, seq len]
        embedded = self.dropout(self.embedding(ids))
        # embedded = [batch size, seq len, embedding dim]
        embedded = embedded.permute(0, 2, 1)
        # embedded = [batch size, embedding dim, seq len]
        conved = [torch.relu(conv(embedded)) for conv in self.convs]
        # conved_n = [batch size, n filters, seq len - filter_sizes[n]
+ 1]
        pooled = [conv.max(dim=-1).values for conv in conved]
        # pooled_n = [batch size, n filters]
        cat = self.dropout(torch.cat(pooled, dim=-1))
        # cat = [batch size, n filters * len(filter_sizes)]
        prediction = self.fc(cat)
        # prediction = [batch size, output dim]
        return prediction

vocab_size = len(vocab)
embedding_dim = 300
n_filters = 100
```

```
filter_sizes = [3, 5, 7]
output_dim = len(train_data.unique("label"))
dropout_rate = 0.25

model = CNN(
    vocab_size,
    embedding_dim,
    n_filters,
    filter_sizes,
    output_dim,
    dropout_rate,
    pad_index,
)

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if
p.requires_grad)


print(f"The model has {count_parameters(model):,} trainable
parameters")
```

```
The model has 6,941,402 trainable parameters
```

```
def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.Conv1d):
        nn.init.kaiming_normal_(m.weight, nonlinearity="relu")
        nn.init.zeros_(m.bias)

model.apply(initialize_weights)
```

```
CNN(
  (embedding): Embedding(21635, 300, padding_idx=1)
  (convs): ModuleList(
    (0): Conv1d(300, 100, kernel_size=(3,), stride=(1,))
    (1): Conv1d(300, 100, kernel_size=(5,), stride=(1,))
    (2): Conv1d(300, 100, kernel_size=(7,), stride=(1,))
  )
  (fc): Linear(in_features=300, out_features=2, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
)
```

```
vectors = torchtext.vocab.GloVe()
```

```
.vector_cache/glove.840B.300d.zip: 2.18GB [06:50, 5.30MB/s]

100%|██████████| 2196016/2196017 [05:59<00:00, 6116.33it/s]
```

```python
pretrained_embedding = vectors.get_vecs_by_tokens(vocab.get_itos())

model.embedding.weight.data = pretrained_embedding

optimizer = optim.Adam(model.parameters())

criterion = nn.CrossEntropyLoss()

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

device
```

```
device(type='cpu')
```

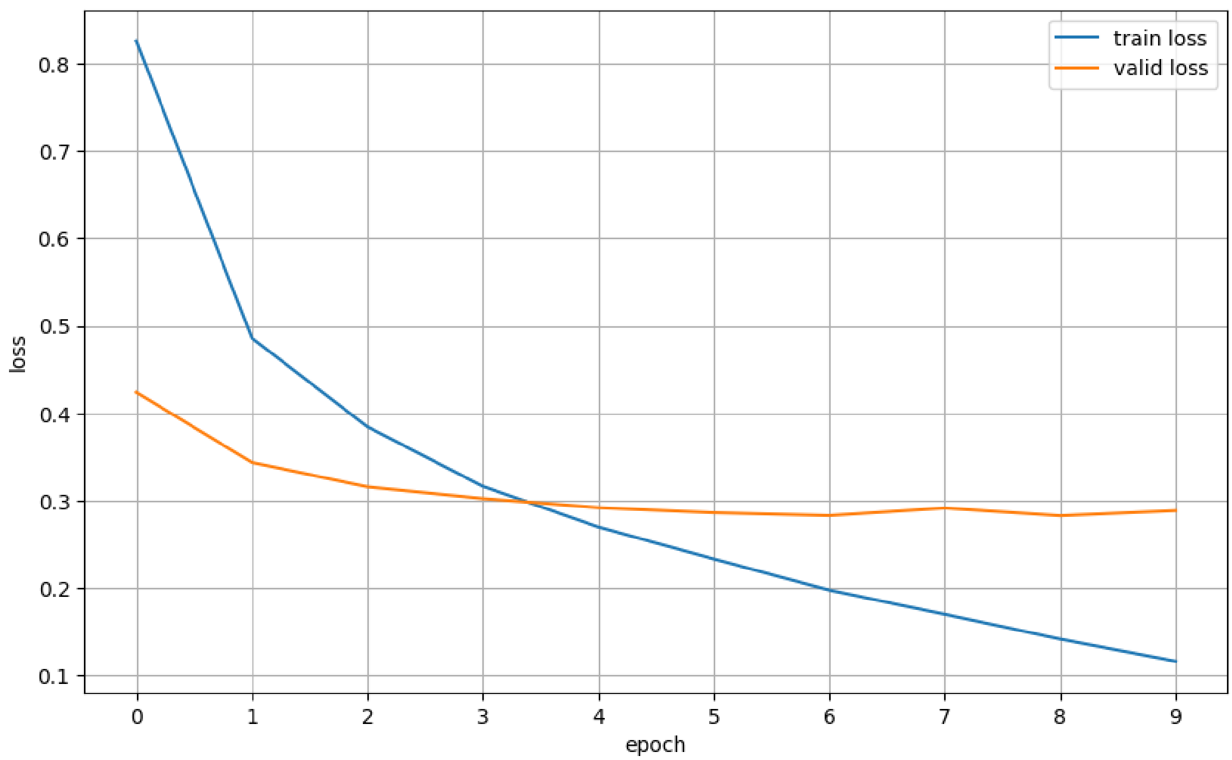
```python
model = model.to(device)
criterion = criterion.to(device)

def train(data_loader, model, criterion, optimizer, device):
    model.train()
    epoch_losses = []
    epoch_accs = []
    for batch in tqdm.tqdm(data_loader, desc="training..."):
        ids = batch["ids"].to(device)
        label = batch["label"].to(device)
        prediction = model(ids)
        loss = criterion(prediction, label)
        accuracy = get_accuracy(prediction, label)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_losses.append(loss.item())
        epoch_accs.append(accuracy.item())
    return np.mean(epoch_losses), np.mean(epoch_accs)

def evaluate(data_loader, model, criterion, device):
    model.eval()
    epoch_losses = []
    epoch_accs = []
    with torch.no_grad():
        for batch in tqdm.tqdm(data_loader, desc="evaluating..."):
            ids = batch["ids"].to(device)
            label = batch["label"].to(device)
            prediction = model(ids)
            loss = criterion(prediction, label)
            accuracy = get_accuracy(prediction, label)
            epoch_losses.append(loss.item())
            epoch_accs.append(accuracy.item())
    return np.mean(epoch_losses), np.mean(epoch_accs)

def get_accuracy(prediction, label):
    batch_size, _ = prediction.shape
```

```
    predicted_classes = prediction.argmax(dim=-1)
    correct_predictions = predicted_classes.eq(label).sum()
    accuracy = correct_predictions / batch_size
    return accuracy

n_epochs = 10
best_valid_loss = float("inf")

metrics = collections.defaultdict(list)

for epoch in range(n_epochs):
    train_loss, train_acc = train(
        train_data_loader, model, criterion, optimizer, device
    )
    valid_loss, valid_acc = evaluate(valid_data_loader, model,
criterion, device)
    metrics["train_losses"].append(train_loss)
    metrics["train_accs"].append(train_acc)
    metrics["valid_losses"].append(valid_loss)
    metrics["valid_accs"].append(valid_acc)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), "cnn.pt")
    print(f"epoch: {epoch}")
    print(f"train_loss: {train_loss:.3f}, train_acc: {train_acc:.3f}")
    print(f"valid_loss: {valid_loss:.3f}, valid_acc: {valid_acc:.3f}")

training...: 100%|██████████| 37/37 [06:42<00:00, 10.89s/it]
evaluating...: 100%|██████████| 13/13 [00:38<00:00,  2.93s/it]

epoch: 0
train_loss: 0.826, train_acc: 0.608
valid_loss: 0.424, valid_acc: 0.813

training...: 100%|██████████| 37/37 [06:44<00:00, 10.93s/it]
evaluating...: 100%|██████████| 13/13 [00:40<00:00,  3.13s/it]

epoch: 1
train_loss: 0.486, train_acc: 0.776
valid_loss: 0.344, valid_acc: 0.853

training...: 100%|██████████| 37/37 [07:13<00:00, 11.71s/it]
evaluating...: 100%|██████████| 13/13 [00:40<00:00,  3.13s/it]

epoch: 2
train_loss: 0.384, train_acc: 0.834
valid_loss: 0.316, valid_acc: 0.867

training...: 100%|██████████| 37/37 [06:56<00:00, 11.27s/it]
evaluating...: 100%|██████████| 13/13 [00:38<00:00,  2.96s/it]
```

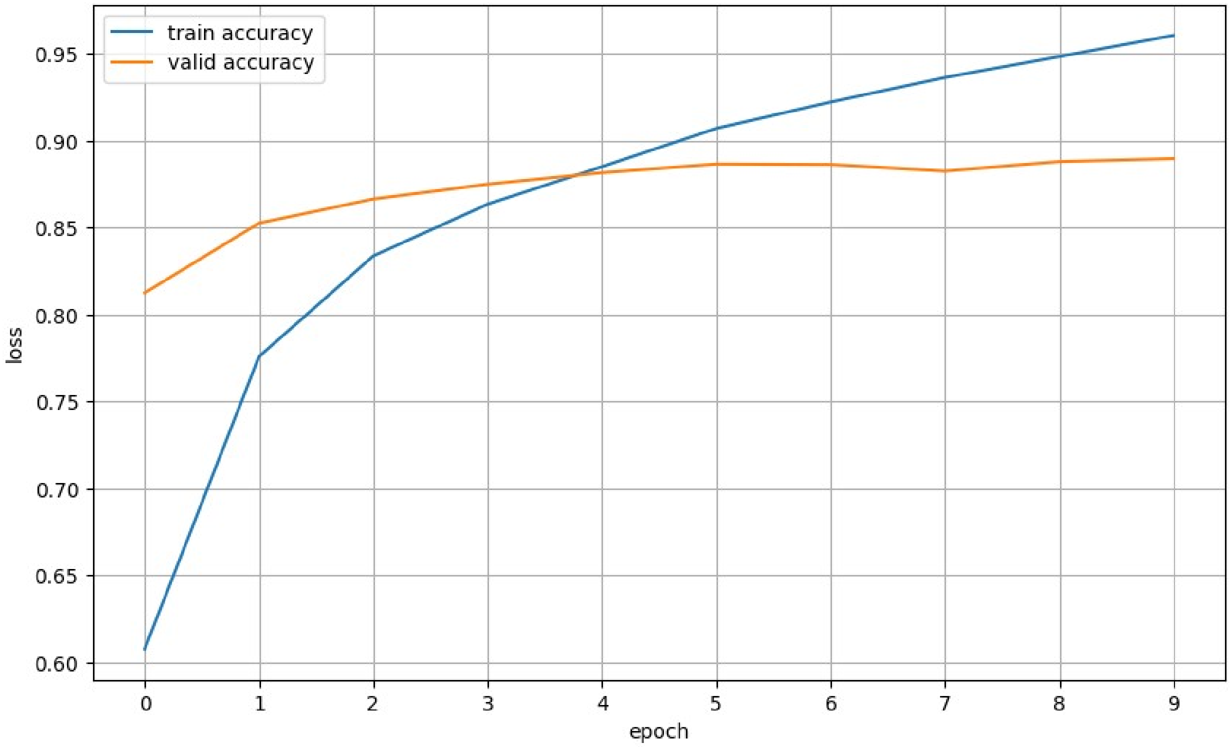
```
epoch: 3
train_loss: 0.316, train_acc: 0.864
valid_loss: 0.302, valid_acc: 0.875

training...: 100%|██████████| 37/37 [06:39<00:00, 10.81s/it]
evaluating...: 100%|██████████| 13/13 [00:39<00:00,  3.04s/it]

epoch: 4
train_loss: 0.270, train_acc: 0.885
valid_loss: 0.292, valid_acc: 0.882

training...: 100%|██████████| 37/37 [06:50<00:00, 11.09s/it]
evaluating...: 100%|██████████| 13/13 [00:39<00:00,  3.02s/it]

epoch: 5
train_loss: 0.233, train_acc: 0.907
valid_loss: 0.286, valid_acc: 0.887

training...: 100%|██████████| 37/37 [06:50<00:00, 11.11s/it]
evaluating...: 100%|██████████| 13/13 [00:38<00:00,  2.96s/it]

epoch: 6
train_loss: 0.197, train_acc: 0.922
valid_loss: 0.283, valid_acc: 0.886

training...: 100%|██████████| 37/37 [06:46<00:00, 10.99s/it]
evaluating...: 100%|██████████| 13/13 [00:42<00:00,  3.29s/it]

epoch: 7
train_loss: 0.170, train_acc: 0.936
valid_loss: 0.291, valid_acc: 0.883

training...: 100%|██████████| 37/37 [06:47<00:00, 11.00s/it]
evaluating...: 100%|██████████| 13/13 [00:38<00:00,  2.96s/it]

epoch: 8
train_loss: 0.141, train_acc: 0.949
valid_loss: 0.283, valid_acc: 0.888

training...: 100%|██████████| 37/37 [06:44<00:00, 10.94s/it]
evaluating...: 100%|██████████| 13/13 [00:38<00:00,  2.97s/it]

epoch: 9
train_loss: 0.116, train_acc: 0.961
valid_loss: 0.289, valid_acc: 0.890
```

```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(metrics["train_losses"], label="train loss")
ax.plot(metrics["valid_losses"], label="valid loss")
```

```
ax.set_xlabel("epoch")
ax.set_ylabel("loss")
ax.set_xticks(range(n_epochs))
ax.legend()
ax.grid()
```




```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(metrics["train_accs"], label="train accuracy")
ax.plot(metrics["valid_accs"], label="valid accuracy")
ax.set_xlabel("epoch")
ax.set_ylabel("loss")
ax.set_xticks(range(n_epochs))
ax.legend()
ax.grid()
```




```
model.load_state_dict(torch.load("cnn.pt"))

test_loss, test_acc = evaluate(test_data_loader, model, criterion,
device)
```

```
evaluating...: 100%|██████████| 49/49 [02:36<00:00,  3.20s/it]
```

```
print(f"test_loss: {test_loss:.3f}, test_acc: {test_acc:.3f}")
```

```
test_loss: 0.301, test_acc: 0.875
```

```
def predict_sentiment(text, model, tokenizer, vocab, device,
min_length, pad_index):
    tokens = tokenizer(text)
    ids = vocab.lookup_indices(tokens)
    if len(ids) < min_length:
        ids += [pad_index] * (min_length - len(ids))
    tensor = torch.LongTensor(ids).unsqueeze(dim=0).to(device)
    prediction = model(tensor).squeeze(dim=0)
    probability = torch.softmax(prediction, dim=-1)
    predicted_class = prediction.argmax(dim=-1).item()
    predicted_probability = probability[predicted_class].item()
    return predicted_class, predicted_probability

text = "This film is terrible!"
min_length = max(filter_sizes)
```

```python
predict_sentiment(text, model, tokenizer, vocab, device, min_length, pad_index)
```

```
(0, 0.9275755286216736)
```

```python
text = "This film is great!"

predict_sentiment(text, model, tokenizer, vocab, device, min_length, pad_index)
```

```
(1, 0.9775592684745789)
```

```python
text = "This film is not terrible, it's great!"

predict_sentiment(text, model, tokenizer, vocab, device, min_length, pad_index)
```

```
(0, 0.8567021489143372)
```

```python
text = "This film is not great, it's terrible!"

predict_sentiment(text, model, tokenizer, vocab, device, min_length, pad_index)
```

```
(1, 0.7653632760047913)
```