sheet08

December 11, 2024

Names: Philipp Köhler, Alexander Bespalov

[14]:
```
import numpy as np
import torch
from matplotlib import pyplot as plt
from torch.utils.data import TensorDataset, DataLoader
import h5py
from sklearn.model_selection import train_test_split
```

0.1 Task 1: CNNs for Galaxy Classification

[3]:
```
# create data folder if it does not exist
import os
os.makedirs("data", exist_ok=True)

import urllib.request
_, msg = urllib.request.urlretrieve(
    "http://www.astro.utoronto.ca/~bovy/Galaxy10/Galaxy10.h5",
    "data/Galaxy10.h5"
)
```

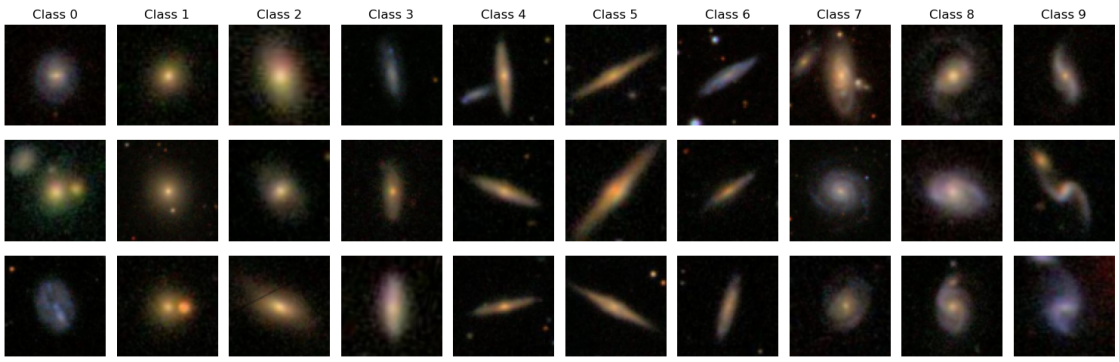
[6]:
```
label_names = [
    'Disk, Face-on, No Spiral',
    'Smooth, Completely round',
    'Smooth, in-between round',
    'Smooth, Cigar shaped',
    'Disk, Edge-on, Rounded Bulge',
    'Disk, Edge-on, Boxy Bulge',
    'Disk, Edge-on, No Bulge',
    'Disk, Face-on, Tight Spiral',
    'Disk, Face-on, Medium Spiral',
    'Disk, Face-on, Loose Spiral'
]
n_classes = len(label_names)

# To get the images and labels from file
with h5py.File('data/Galaxy10.h5', 'r') as F:
    images = np.array(F['images'])
```

```
    labels = np.array(F['ans'])
images = images.astype(np.float32)

# comply to (batch, channel, height, width) convention of pytorch
images = np.moveaxis(images, -1, 1)
# convert to torch
images = torch.from_numpy(images)
labels = torch.from_numpy(labels)

print(f'{images.shape=}, {labels.shape=}')
```

```
images.shape=torch.Size([21785, 3, 69, 69]), labels.shape=torch.Size([21785])
```

0.1.1 (a)

[7]:
```
# TODO: plot three samples of each class
def visualize_data(images, labels):
    fig, axes = plt.subplots(3, 10, figsize=(15, 5))
    unique_classes = torch.unique(labels)
    for i, class_id in enumerate(unique_classes):
        class_images = images[labels == class_id]
        for j in range(3):
            axes[j, i].imshow(class_images[j].permute(1, 2, 0).numpy().
 ↪astype(int))
            axes[j, i].axis("off")
            if j == 0:
                axes[j, i].set_title(f"Class {class_id.item()}")
    plt.tight_layout()
    plt.show()

visualize_data(images, labels)
```



For normalization the mean and the standard deviation across the whole train dataset for each channel is calculated.

[20]:
```
from torchvision.transforms import Normalize

# TODO: Normalize the images
# TODO: Split the data and create tensordatasets and data loaders:
X_train, X_temp, y_train, y_temp = train_test_split(images, labels, test_size=0.
 ↪2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)

means = torch.mean(X_train, axis=(0,2,3))
stds =  torch.std(X_train,axis=(0,2,3))
normalize_transform = Normalize(means,stds)

X_train = normalize_transform(X_train)
X_val = normalize_transform(X_val)
X_test = normalize_transform(X_test)

train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

0.1.2 (b)

[30]:
```
#TODO: implement a small CNN as specified on the sheet
from torch import nn
import torch.nn.functional as F

class GalaxyCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(16 * 14 * 14, 64) # 14 is dimension size after␣
 ↪last pooling
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 10)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
```

```
        x = self.pool(F.relu(self.conv2(x)))
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

0.1.3

0.1.4 (c) + (d) + (e)

Cross Entropy Loss is the loss of choice for multi-class classification.

[36]:
```
import torch.optim as optim
# TODO: Instantiate the model, optimizer and criterion
model = GalaxyCNN()

# optimizer = ?
# criterion = ?
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

train_losses = []
train_accs = []
val_losses = []
val_accs = []

# TODO: Implement the training loop, validating after every epoch, and make the␣
 ↪requested plots.

def validate(model,val_loader):
    losses = []
    correct = 0
    total = 0

    #TODO: Implement the validation loop
    model.eval()
    with torch.no_grad():
        for images, labels in val_loader:
            outputs = model(images)
            loss = criterion(outputs, labels)
            losses.append(loss.item())

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    avg_loss = np.mean(np.array(losses))
```

```
    accuracy = correct / total
    print(f'{accuracy=:.2f}, {avg_loss=:.2e}')
    return avg_loss, accuracy

val_loss, val_acc = validate(model,val_loader)
val_losses.append(val_loss)
val_accs.append(val_acc)
num_epochs = 30
best_val_loss = float('inf')
best_model_wts = None
for epoch in range(num_epochs):
    # TODO: Implement the training loop, validating after every epoch and a␣
 ↪visialization of the loss curves
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    avg_train_loss = running_loss / len(train_loader)
    train_accuracy = correct / total
    train_losses.append(avg_train_loss)
    train_accs.append(train_accuracy)

    val_loss, val_acc = validate(model, val_loader)
    val_losses.append(val_loss)
    val_accs.append(val_acc)
    if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_wts = model.state_dict()
    print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {avg_train_loss:.2e},␣
 ↪Validation Loss: {val_loss:.2e}, Validation Accuracy: {val_acc:.2f}")
```

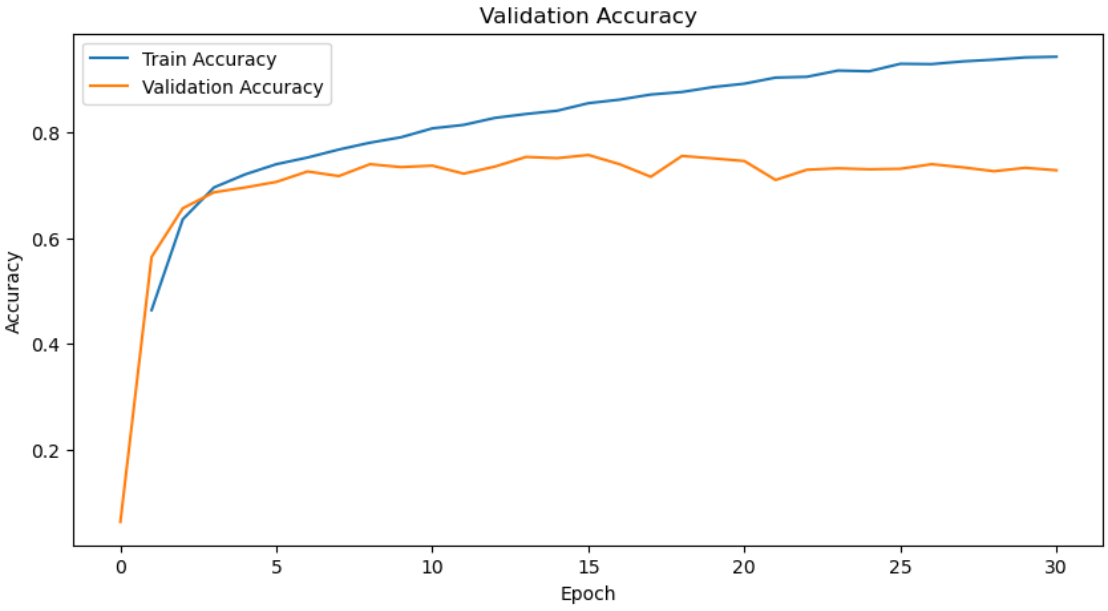
```
accuracy=0.06, avg_loss=2.39e+00
accuracy=0.56, avg_loss=1.13e+00
Epoch 1/30, Training Loss: 1.37e+00, Validation Loss: 1.13e+00, Validation
Accuracy: 0.56
accuracy=0.66, avg_loss=9.95e-01
Epoch 2/30, Training Loss: 9.68e-01, Validation Loss: 9.95e-01, Validation
```

```
Accuracy: 0.66
accuracy=0.69, avg_loss=9.14e-01
Epoch 3/30, Training Loss: 8.28e-01, Validation Loss: 9.14e-01, Validation
Accuracy: 0.69
accuracy=0.70, avg_loss=9.09e-01
Epoch 4/30, Training Loss: 7.58e-01, Validation Loss: 9.09e-01, Validation
Accuracy: 0.70
accuracy=0.71, avg_loss=8.75e-01
Epoch 5/30, Training Loss: 7.04e-01, Validation Loss: 8.75e-01, Validation
Accuracy: 0.71
accuracy=0.73, avg_loss=8.36e-01
Epoch 6/30, Training Loss: 6.72e-01, Validation Loss: 8.36e-01, Validation
Accuracy: 0.73
accuracy=0.72, avg_loss=8.03e-01
Epoch 7/30, Training Loss: 6.24e-01, Validation Loss: 8.03e-01, Validation
Accuracy: 0.72
accuracy=0.74, avg_loss=7.77e-01
Epoch 8/30, Training Loss: 5.95e-01, Validation Loss: 7.77e-01, Validation
Accuracy: 0.74
accuracy=0.73, avg_loss=7.84e-01
Epoch 9/30, Training Loss: 5.67e-01, Validation Loss: 7.84e-01, Validation
Accuracy: 0.73
accuracy=0.74, avg_loss=7.79e-01
Epoch 10/30, Training Loss: 5.22e-01, Validation Loss: 7.79e-01, Validation
Accuracy: 0.74
accuracy=0.72, avg_loss=8.33e-01
Epoch 11/30, Training Loss: 5.03e-01, Validation Loss: 8.33e-01, Validation
Accuracy: 0.72
accuracy=0.74, avg_loss=8.50e-01
Epoch 12/30, Training Loss: 4.70e-01, Validation Loss: 8.50e-01, Validation
Accuracy: 0.74
accuracy=0.75, avg_loss=7.77e-01
Epoch 13/30, Training Loss: 4.47e-01, Validation Loss: 7.77e-01, Validation
Accuracy: 0.75
accuracy=0.75, avg_loss=8.47e-01
Epoch 14/30, Training Loss: 4.24e-01, Validation Loss: 8.47e-01, Validation
Accuracy: 0.75
accuracy=0.76, avg_loss=8.69e-01
Epoch 15/30, Training Loss: 3.90e-01, Validation Loss: 8.69e-01, Validation
Accuracy: 0.76
accuracy=0.74, avg_loss=8.82e-01
Epoch 16/30, Training Loss: 3.74e-01, Validation Loss: 8.82e-01, Validation
Accuracy: 0.74
accuracy=0.72, avg_loss=9.51e-01
Epoch 17/30, Training Loss: 3.51e-01, Validation Loss: 9.51e-01, Validation
Accuracy: 0.72
accuracy=0.76, avg_loss=9.30e-01
Epoch 18/30, Training Loss: 3.30e-01, Validation Loss: 9.30e-01, Validation
```

```
Accuracy: 0.76
accuracy=0.75, avg_loss=9.99e-01
Epoch 19/30, Training Loss: 3.02e-01, Validation Loss: 9.99e-01, Validation
Accuracy: 0.75
accuracy=0.75, avg_loss=9.73e-01
Epoch 20/30, Training Loss: 2.92e-01, Validation Loss: 9.73e-01, Validation
Accuracy: 0.75
accuracy=0.71, avg_loss=1.10e+00
Epoch 21/30, Training Loss: 2.64e-01, Validation Loss: 1.10e+00, Validation
Accuracy: 0.71
accuracy=0.73, avg_loss=1.03e+00
Epoch 22/30, Training Loss: 2.58e-01, Validation Loss: 1.03e+00, Validation
Accuracy: 0.73
accuracy=0.73, avg_loss=1.16e+00
Epoch 23/30, Training Loss: 2.28e-01, Validation Loss: 1.16e+00, Validation
Accuracy: 0.73
accuracy=0.73, avg_loss=1.18e+00
Epoch 24/30, Training Loss: 2.24e-01, Validation Loss: 1.18e+00, Validation
Accuracy: 0.73
accuracy=0.73, avg_loss=1.31e+00
Epoch 25/30, Training Loss: 1.93e-01, Validation Loss: 1.31e+00, Validation
Accuracy: 0.73
accuracy=0.74, avg_loss=1.22e+00
Epoch 26/30, Training Loss: 1.94e-01, Validation Loss: 1.22e+00, Validation
Accuracy: 0.74
accuracy=0.73, avg_loss=1.34e+00
Epoch 27/30, Training Loss: 1.81e-01, Validation Loss: 1.34e+00, Validation
Accuracy: 0.73
accuracy=0.73, avg_loss=1.44e+00
Epoch 28/30, Training Loss: 1.66e-01, Validation Loss: 1.44e+00, Validation
Accuracy: 0.73
accuracy=0.73, avg_loss=1.55e+00
Epoch 29/30, Training Loss: 1.59e-01, Validation Loss: 1.55e+00, Validation
Accuracy: 0.73
accuracy=0.73, avg_loss=1.51e+00
Epoch 30/30, Training Loss: 1.56e-01, Validation Loss: 1.51e+00, Validation
Accuracy: 0.73
```

[39]:
```python
plt.figure(figsize=(10, 5))
plt.plot(range(1, 31), train_losses, label='Train Loss')
plt.plot(range(0, 31), val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
```

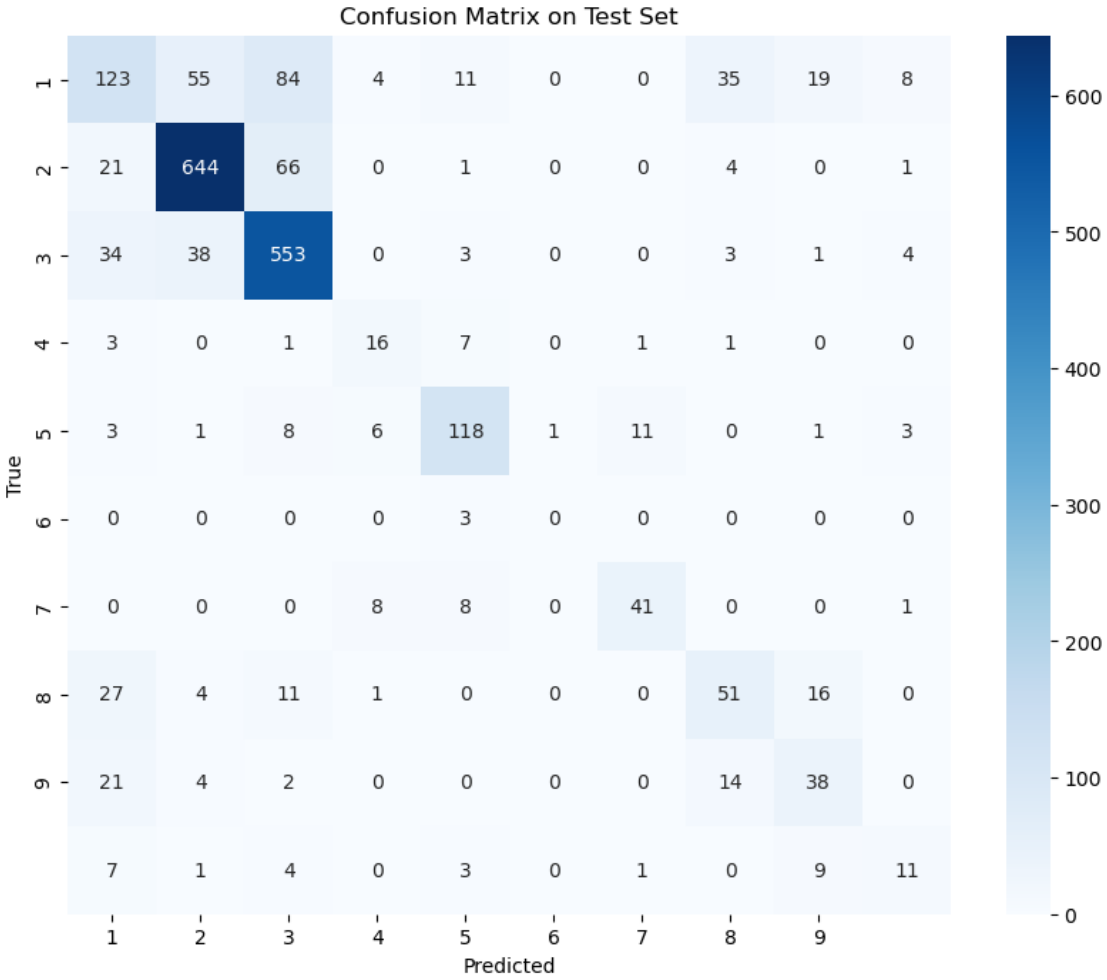
```
plt.figure(figsize=(10, 5))
plt.plot(range(1, 31), train_accs, label='Train Accuracy')
plt.plot(range(0, 31), val_accs, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Validation Accuracy')
plt.show()
```



The model tends to overfit. Taking an earlier checkpoint reduces the problem. Better results can be achieved with regularization or larger datasets.

0.1.5 (e)

[40]:
```
# TODO: Evaluate the best validation model on the test set and create a␣
↪confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns


model.load_state_dict(best_model_wts)
def evaluate_on_test_set(model, test_loader):
    model.eval()
    y_true = []
    y_pred = []

    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            y_true.extend(labels.numpy())
            y_pred.extend(predicted.numpy())

    cm = confusion_matrix(y_true, y_pred)
    return cm

cm = evaluate_on_test_set(model, test_loader)

plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=np.arange(1,␣
↪10), yticklabels=np.arange(1, 10))
plt.title('Confusion Matrix on Test Set')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```



0.1.6 (f)

Invariance under rotations can be achieved by augmentation of the training set with rotation. Another approach is to use group invariant CNNs, where the convolutions are eqivariant to groups as rotations.

0.2 2 Contrastive learning as an example of self-supervised learning

0.2.1 (a)

$$\text{softmax}\left(\frac{\text{sim}(z_{i,t}, z_{j,t'})}{\tau}\right) = \frac{\exp\left(\frac{\frac{z_{i,t}\cdot z_{j,t'}}{\|z_{i,t}\|_2\|z_{j,t'}\|_2}}{\tau}\right)}{\sum_{t,t'\in\mathcal{T}}\sum_{i=1}^{N}\sum_{j=1}^{N}\exp\left(\frac{\frac{z_{i,t}\cdot z_{j,t'}}{\|z_{i,t}\|_2\|z_{j,t'}\|_2}}{\tau}\right)}$$

0.2.2 (b)

$$\mathcal{L}_{\text{InfoNCE}} = -\frac{1}{|S|} \sum_{(i,j)} y_{itjt'} \log \frac{\exp\left(\text{sim}(z_{i,t}, z_{j,t'})/\tau\right)}{\sum_{t,t' \in \mathcal{T}} \sum_{i=1}^{N} \sum_{j=1}^{N} \exp\left(\frac{\text{sim}(z_{i,t}, z_{j,t'})}{\tau}\right)}$$

$|S|$ is the number of pairs.

0.2.3 (c)

Use $y_{itjt'} = \delta_{ij}$.

$$\Rightarrow \mathcal{L}_{\text{InfoNCE}} = -\frac{1}{|P|} \sum_{(i,j) \in P} \log \frac{\exp\left(\text{sim}(z_{i,t}, z_{j,t'})/\tau\right)}{\sum_{t,t' \in \mathcal{T}} \sum_{i=1}^{N} \sum_{j=1}^{N} \exp\left(\frac{\text{sim}(z_{i,t}, z_{j,t'})}{\tau}\right)}$$

Where $ P $ is the set of all positive pairs (i.e., pairs $ (i, j) $ where $ i = j $).

$$\Leftrightarrow \mathcal{L}_{\text{InfoNCE}} = -\frac{1}{|P|} \sum_{i} \log \frac{\exp\left(\text{sim}(z_{i,t}, z_{i,t'})/\tau\right)}{\sum_{t,t' \in \mathcal{T}} \sum_{i=1}^{N} \sum_{j=1}^{N} \exp\left(\frac{\text{sim}(z_{i,t}, z_{j,t'})}{\tau}\right)}$$

0.2.4 (d)

The Cross Entropy Loss has a classficiation nature itself. Looking at the argument of the log it becomes apparent that the loss concentrates to maximize the values of the valid pairs as those are in the numerator and minimize the values of all other pairs as they only occur in the denominator. The resulting embedding is non-trivial as the cosine similarity and the loss align similar pairs (i.e. makes them collinear) and makes unsimilar pairs perpenticular. This results in the spread of unsimilar data in the hyperspace. Further, the applied transformations ensure desirable invariances of the encoding.
The projection of g may lose information due to the alignment and the optimization towards the InfoNCE Loss. f does not encouter this problem. It preserves relevant information while being invariant to the applied transformations.

0.3 3 Positional Encoding

$$E \in \mathbb{R}^{p \times n}$$

$$E_{(2k),i} = \sin\left(i \cdot \exp\left(-\frac{2k \cdot \log(10000)}{p}\right)\right)$$

$$E_{(2k+1),i} = \cos\left(i \cdot \exp\left(-\frac{2k \cdot \log(10000)}{p}\right)\right)$$

0.3.1 (a)

(i)

$$K^T Q = (X + E)^T (X + E) = X^T X + E^T X + X^T E + E^T E$$

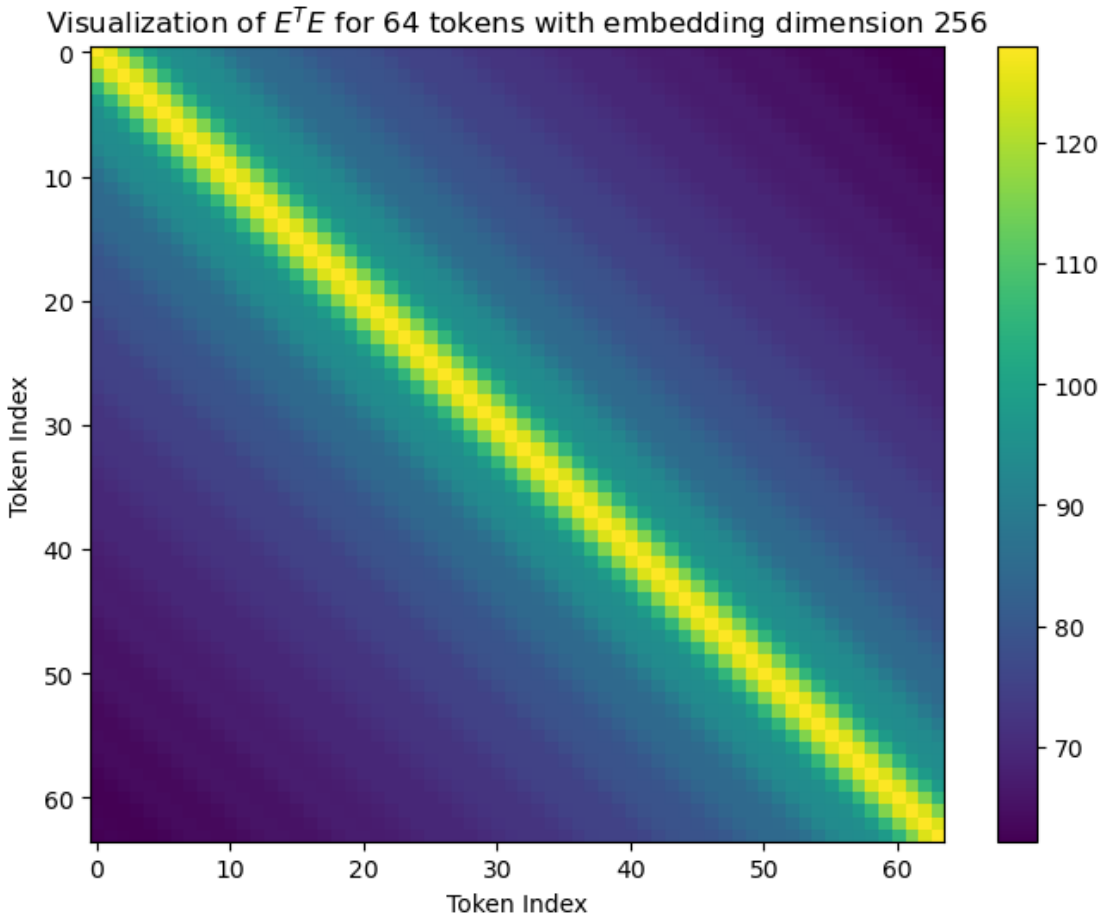
(ii)

$$K^T Q = cat(X, E)^T cat(X, E) = X^T X + E^T X + X^T E + E^T E$$

Both representations contain the same information. The advantage of (i) is the simplicity and less computational effort during self attention. In contast to that (ii) is more expressive and allows weightning of the importance of the position, but needs in general more parameters in the transformer as the dimension becomes $2p \times n$

0.3.2 (b)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

def positional_encoding(n_tokens, embedding_dim):
    E = np.zeros((embedding_dim, n_tokens))

    for i in range(n_tokens):
        for k in range(embedding_dim // 2):
            E[2 * k, i] = np.sin(i * np.exp(-2 * k * np.log(10000) /␣
 ↪embedding_dim))
            E[2 * k + 1, i] = np.cos(i * np.exp(-2 * k * np.log(10000) /␣
 ↪embedding_dim))

    return E

n_tokens = 64
embedding_dim = 256

E = positional_encoding(n_tokens, embedding_dim)
E_transpose_E = np.dot(E.T, E)

plt.figure(figsize=(8, 6))
plt.imshow(E_transpose_E, cmap='viridis', aspect='auto')
plt.colorbar()
plt.title("Visualization of $E^T E$ for 64 tokens with embedding dimension 256")
plt.xlabel('Token Index')
plt.ylabel('Token Index')
plt.show()
```

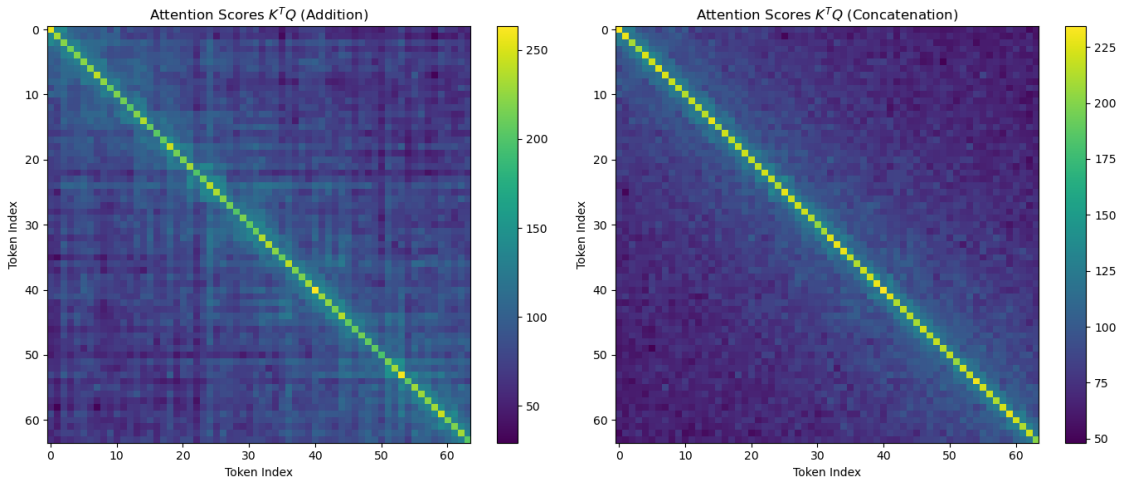


The matrix has diagonal elements and tokens near by have higher outputs than tokens farther apart. Additionally, periocicity can be observed due to the nature of the encoding with sin and cos.

0.3.3 (c)

[5]:
```
sig_E = np.std(E)
X = np.random.randn(embedding_dim, n_tokens) *sig_E

# (i) K = Q = X + E
K_add = X + E
Q_add = X + E
attention_add = np.dot(K_add.T, Q_add)

# (ii) K = Q = cat(X, E)
K_cat = np.concatenate((X, E), axis=0)
Q_cat = np.concatenate((X, E), axis=0)
attention_cat = np.dot(K_cat.T, Q_cat)
```

```
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.imshow(attention_add, cmap='viridis', aspect='auto')
plt.colorbar()
plt.title("Attention Scores $K^T Q$ (Addition)")
plt.xlabel('Token Index')
plt.ylabel('Token Index')


plt.subplot(1, 2, 2)
plt.imshow(attention_cat, cmap='viridis', aspect='auto')
plt.colorbar()
plt.title("Attention Scores $K^T Q$ (Concatenation)")
plt.xlabel('Token Index')
plt.ylabel('Token Index')

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



The concatenation delivers more concentration along the diagonal. This is not expected as both methods should be mathematically identical according to (a). An idea for an explanation is that due to the extra dimensions with the concatenation the positions and features are not as mixed but this should also be reflected in the caluclations in (a). Why is that?

0.3.4 (d)

When both lie in a low-dimensional subspace of the high-dimensional euclidean space the vectors become sparse. Due to their sparsity the addition of the vectors has minor influence on the informations of each of the added vectors as most likely a 0 is added. This means the information is

mostly encoded in a subspace. In the concatenation the information in also encoded in the same subspace even if the original space is twice as large. This results in the same attention scores as the same information is now mapped in the same dimension.