Homework #4 A

Spring 2020, CSE 446/546: Machine Learning
Dino Bektesevic

Collaborated: Conor Sayers, Joachim Moeyenes, Jessica Birky, Leah Fulmer

Conceptual Questions

A.1 The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words

a. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function,and therefore gradient descent might not reach the globally-optimal solution.

b. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

c. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

d. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the back-propagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

e. *[2 points]* True or False: Autoencoders, where the encoder and decoder functions are both neural networks with nonlinear activations, can capture more variance of the data in its encoded representation than PCA using the same number of dimensions.

Think before you train

A2. In class, we discussed some of the ways in which many datasets describing crime have various shortcomings in describing the entire landscape of illegal behavior in a city, and that these shortcomings often fall disproportionately on minority communities. Some examples include that crimes are reported at different rates indifferent neighborhoods, that police respond differently to the same crime reported or observed in different neighborhoods, and that police spend more time patrolling in some neighborhoods than others.

a. *[5 points]* Please describe two statistical problems arising from one or more of these issues (or others mentioned in class, or others from some other source that you cite relating to datasets compiled related to crime in the US), and what real-world implications might follow from ignoring these issues. A short paragraph for each statistical problem suffices.

b. *[5 points]* For each of your previously identified statistical concerns above, propose a technical "fix" (e.g.a new sampling strategy, some training method for which few or no distributional assumptions are needed on the training set).

c. *[5 points]* The solutions you described in (b) might only take you so far towards ensuring a model has similarly positive impact on different communities. Describe two reasons why a machine learning model trained to predict $f(x)$ from x might have different accuracy on two different populations, even if the training data is drawn i.i.d. from the true distribution over the event of interest. [These assumptions exclude the possibility that the two populations are sampled from at rates different from their latent frequency, though their latent frequencies may be different.]

Unsupervised Learning with Autoencoders

A3. Last homework we used PCA to project and reconstruct data from the MNIST dataset. In this exercise, we will train two simple autoencoders to perform dimensionality reduction on MNIST. As discussed in lecture, autoencoders are a long-studied neural network architecture comprised of an encoder component to summarize the latent features of input data and a decoder component to try and reconstruct the original data from the latent features.

Weight Initialization and PyTorch

Last assignment, we had you refrain from using **torch.nn** modules. For this assignment, we recommend using **nn.Linear** for your linear layers. You will not need to initialize the weights yourself; the default He/Kaiming uniform initialization in PyTorch will be sufficient for this problem. Hint: we also recommend using the nn.Sequential module to organize your network class and simplify the process of writing the forward pass.

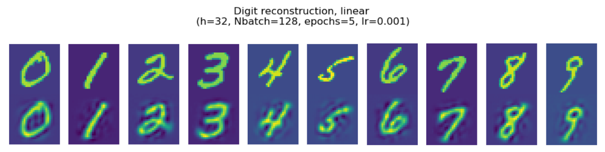
Training

Use optim.Adam for this question. Experiment with different learning rates. Use mean squared error (nn.MSELoss() or F.mseloss()) for the loss function and ReLU for the non-linearity in b

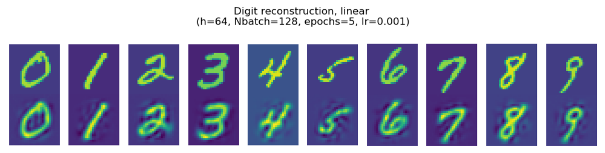
a. *[10 points]* Use a network with a single linear layer. Let $W_e \in \mathbb{R}^{h \times d}$ and $W_d \in \mathbb{R}^{d \times h}$. Given some $x \in \mathbb{R}^d$, the forward pass is formulated as

$$\mathbb{F}_1(x) = W_d W_e x$$

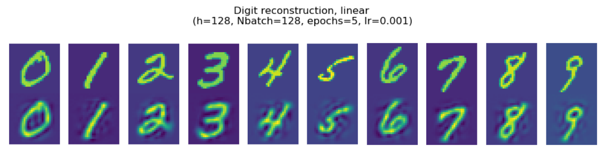
Run experiments for $h \in \{32, 64, 128\}$. For each of the different h values, report your final error and visualize a set of 10 reconstructed digits, side-by-side with the original image. Note: we omit the bias term in the formulation for notational convenience since nn.Linear learns bias parameters alongside weight parameters by default.


Digit reconstruction, linear
(h=32, Nbatch=128, epochs=5, lr=0.001)

(a) After 5 epochs, with 0.001 learning rate, the test loss is 0.00077440


Digit reconstruction, linear
(h=64, Nbatch=128, epochs=5, lr=0.001)

(b) After 5 epochs, with 0.001 learning rate, the test loss is 0.00077228


Digit reconstruction, linear
(h=128, Nbatch=128, epochs=5, lr=0.001)

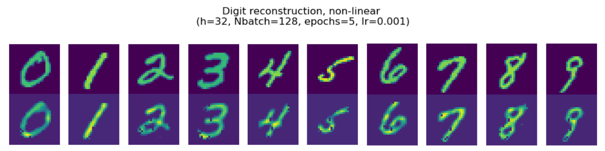
(c) After 5 epochs, with 0.001 learning rate, the test loss is 0.00077527

Figure 1: MNIST reconstructions using various different hyperparameters.

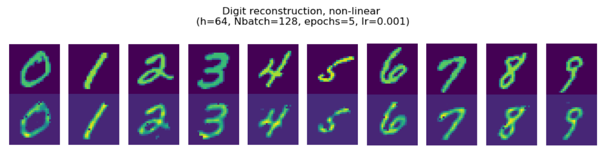
b. *[10 points]* Use a single-layer network with non-linearity. Let $W_e \in \mathbb{R}^{h\times d}$,$W_d \in \mathbb{R}^{d\times h}$, and activation $\sigma : \mathbb{R} \to \mathbb{R}$. Given some $x \in \mathbb{R}^d$, the forward pass is formulated as

$$\mathbb{F}_2(x) = \sigma(W_d \sigma(W_e x))$$

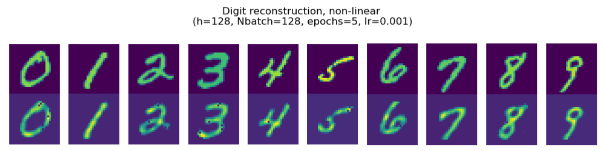
Report the same findings as asked for in part a for $h \in \{32, 64, 128\}$.


Digit reconstruction, non-linear
(h=32, Nbatch=128, epochs=5, lr=0.001)

(a) After 5 epochs, with 0.001 learning rate, the test loss is 0.00221150


Digit reconstruction, non-linear
(h=64, Nbatch=128, epochs=5, lr=0.001)

(b) After 5 epochs, with 0.001 learning rate, the test loss is 0.00205961


Digit reconstruction, non-linear
(h=128, Nbatch=128, epochs=5, lr=0.001)

(c) After 5 epochs, with 0.001 learning rate, the test loss is 0.00215065

Figure 2: MNIST reconstructions using various different hyperparameters.

c. *[5 points]* Now, evaluate $\mathbb{F}_1(x)$ and $\mathbb{F}_2(x)$ (use $h = 128$ here) on the test set. Provide the test reconstruction errors in a table.

	loss
$\mathbb{F}_1$	0.00077527
F_2	0.00215065

d. *[5 points]* In a few sentences, compare the quality of the reconstructions from these two autoencoders, compare with those of PCA from last assignment. You may want to re-run your code for PCA using the different h values as the number of top-k eigenvalues.
Effectively it's a poor-mans PCA in this context - we apply some linear transformation to the dataset and then learn the randomly instantiated weights in the decode step to linearly transform the dataset back. It is not surprising that the reconstructions are nearly identical to the input. Statistically, nonlinear reconstruction performs worse than the linear one. Visually, however, the non-linear reconstruction behaves better. This is because our eyes do not notice the missing bright pixel or two but because of the non-linearity we are able to better ignore the background.
Plots below showcase my analogy with a 2-component PCA. I project the image all the way down to 3 dimensional latent space and then plot them against each other. Points are colored according to labels. This neatly shows how autoencoder seems to dis-entangle various digits by grouping them in its latent space, similarly to PCA.

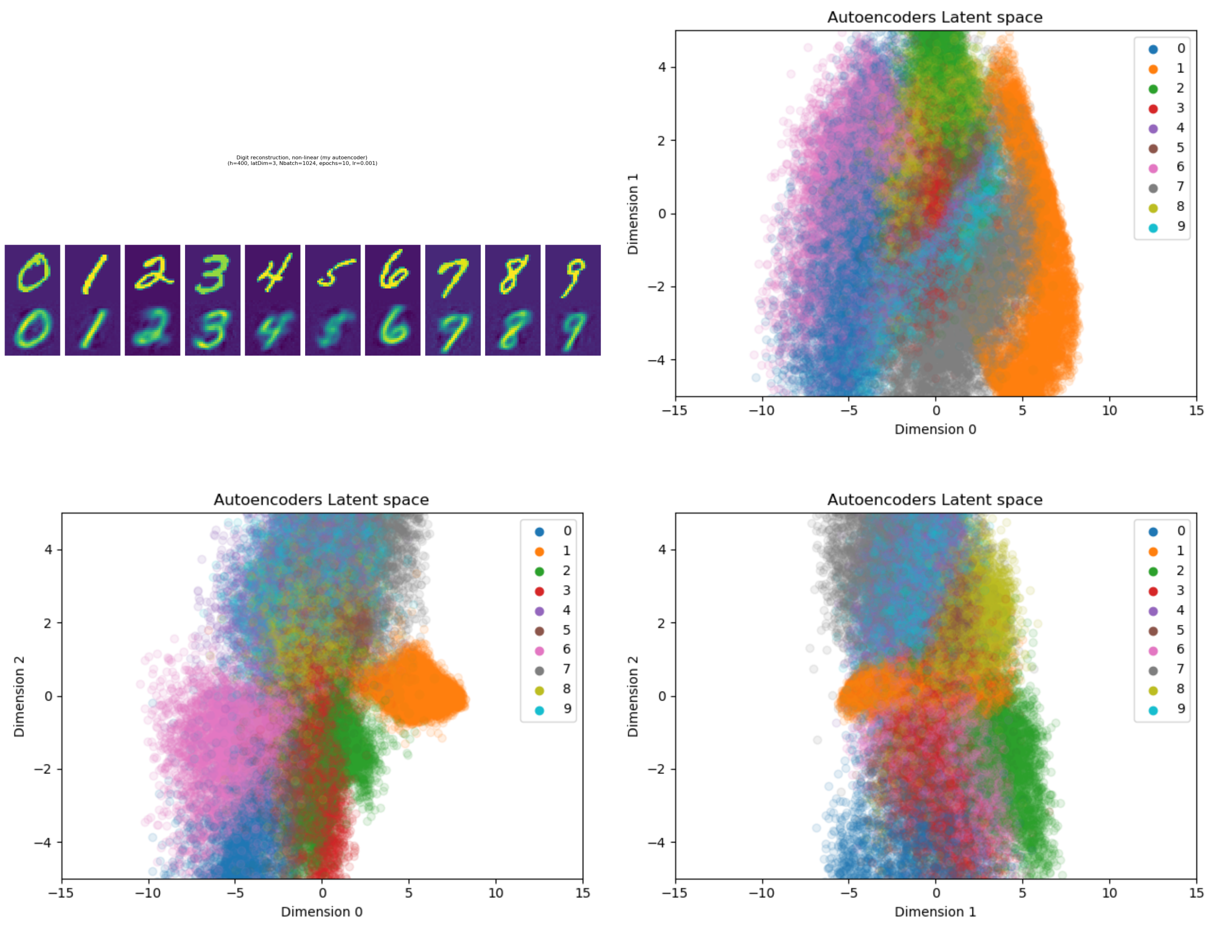


Figure 3: MNIST reconstructions using my autoencoder. They are visually relatively poor, no doubt due to a low-dimensional latent space. Trained for 10 epochs with learning rate at 0.001 the test loss is 0.00039763.

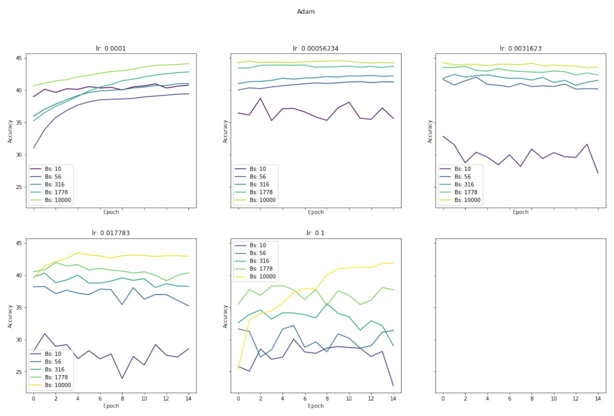
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import cm

import torch
from torch import nn, optim
from torch.nn import functional
import torch.utils.data as datutils

from torchvision import datasets, transforms


torch.manual_seed(0)
np.random.seed(0)
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


def load_mnist_dataset(path="data/mnist_data/", digit=None, batchSize=1):
    """Loads MNIST data located at path.

    MNIST data are 28x28 pixel large images of numbers.

    Parameters
    ----------
    path : `str`
        Path to the data directory
    digit : `int` or `None`, optional
        Load only data for a specific digit.
    batchSize: `int`, optional
        Batch size.

    Returns
    -------
    train : `torch.DataLoader`
        A generator that will shuffle the data at every iteration, yields train
        datasets
    trainlabels : `torch.DataLoader`
        A generator that returns the test dataset.

    Notes
    -----
    Data are normalized upon loading to the mean of the dataset.
    Data are downloaded if not present at path.
    """
    trans = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
    train = datasets.MNIST(path, train=True, download=True, transform=trans)
    test = datasets.MNIST(path, train=False, transform=trans)

    trainMask, testMask = np.arange(len(train)), np.arange(len(test))
    if digit is not None:
        trainMask = np.where(train.targets == digit)[0]
        testMask = np.where(test.targets == digit)[0]

    maskTrain = datutils.Subset(train, trainMask)
    maskTest = datutils.Subset(test, testMask)
    trainLoader = datutils.DataLoader(maskTrain, batch_size=batchSize, shuffle=True)
    testLoader = datutils.DataLoader(maskTest, batch_size=batchSize, shuffle=False)

    return trainLoader, testLoader


class MNISTAutoencoder(nn.Module):
    """MNIST Autoencoder Module.

    Parameters
    ----------
    encode: `torch.nn.Sequential`
        Encoding steps
    decode: `torch.nn.Sequential`
        Decoding steps
    imgSize: `int`, optional
        The image dimensions multiplied. Defaults to 28*28
    """
    def __init__(self, encode, decode, *args, **kwargs):
        super().__init__()
        self.encode = encode
        self.decode = decode
        self.imgSize = kwargs.pop("imgSize", 28*28)

    def forward(self, x):
        """Encode the given data and then decode it.

        Parameters
        -----------
        x : `torch.tensor`
            Data to encode

        Returns
        -------
        decoded : `torch.tensor`
            The decoded encoded data.
        """
        z = self.encode(x.view(-1, self.imgSize))
        return self.decode(z)

    def loss(self, x):
        """Given the data, alculate MSE loss of the reconstruction. Ensure the
        autoencoder has been trained.

        Parameters
        -----------
        x : `torch.tensor`
            Data

        Returns
        -------
        loss : `nn.MSELoss`
            MSE Loss
        """
        reconstructed = self.forward(x)
        loss = functional.mse_loss(reconstructed, x.view(-1, self.imgSize))
        return loss


class A3aAutoencoder(MNISTAutoencoder):
    """Simple single linear encode and decode layer autoencoder.

    Parameters
    ----------
    imgSize : `int`, optional
        Size of the images. Default 28*28
    hiddenSize : `int`, optional
        Size of the encoding layer. Default 64
    """
    def __init__(self, imgSize=28*28, hiddenSize=64, **kwargs):
        super().__init__(
            encode = nn.Sequential(nn.Linear(imgSize, hiddenSize)),
            decode = nn.Sequential(nn.Linear(hiddenSize, imgSize)),
            imgSize = imgSize,
            **kwargs
        )
        self.hiddenSize = hiddenSize


class A3bAutoencoder(MNISTAutoencoder):
    """Simple single non-linear (ReLU) encode and decode layer autoencoder.

    Parameters
    ----------
    imgSize : `int`, optional
        Size of the images. Default 28*28
    hiddenSize : `int`, optional
        Size of the encoding layer. Default 64
    """

    def __init__(self, imgSize=28*28, hiddenSize=64, **kwargs):
        super().__init__(
            encode = nn.Sequential( nn.Linear(imgSize, hiddenSize), nn.ReLU() ),
            decode = nn.Sequential( nn.Linear(hiddenSize, imgSize), nn.ReLU() ),
            imgSize = imgSize,
            **kwargs
        )
        self.hiddenSize = hiddenSize


def train(dataLoader, model, optimizer, epoch, verbosity=20):
    """Trains the model using the given batched data. Calculates loss for each
    batch as well as the total average loss across the epoch and prints them.

    Parameters
    ----------
    dataLoader: `torch.DataLoader`
        Generator that returnes batched train data.
    model : `obj`
        One of the Autoencoders or some other nn.Module with a loss method.
    optimizer : `obj`
        One of pytorches optimizers (f.e. torch.optim.Adam)
    epoch : `int`
        What epoch is this training step performing, used to calculate loss
    verbosity: `int`
        How often are batch losses printed, larger number means less prints.
        Epoch average loss is always printed.
    """
    # DataLoader length is number of batches that fit in the dataset.
    # Length of the dataset is the actual number of data points
    # used (f.e. the total number of CIFAR images)
    nAll, nBatches = len(dataLoader.dataset), len(dataLoader)
    verbosity = 1 if nBatches/verbosity < 1 else np.ceil(nBatches/verbosity)
    trainLoss, avgTrainLoss = 0, 0
    print(f"Epoch: {epoch}:")
    for i, (data, labels) in enumerate(dataLoader):
        data = data.to(DEVICE)

        optimizer.zero_grad()
        loss = model.loss(data)
        loss.backward()
        optimizer.step()

        lss = loss.item()
        trainLoss += lss
        avgTrainLoss += lss
        if i % verbosity == 0 and i!=0:
            # The length of data, loaded by loader, is at most the batch size,
            # not neccessarily equal for all batches (i.e. last one might be shorter).
            nBatch = len(data)
            print(
                f"    [{i*nBatch:>6}/{nAll:>6} ({100.0*i/nBatches:<5.4}%)]"
                f"    Loss: {trainLoss/verbosity:>10.8f}"
            )
            trainLoss = 0.0

    print(f"    Avg train loss: {avgTrainLoss/nBatches:>15.4f}")


def test(dataLoader, model, optimizer):
    """Calculate and print test losses.

    Parameters
    ----------
    dataLoader: `torch.DataLoader`
        Generator that returnes batched train data.
    model : `obj`
        One of the Autoencoders or some other nn.Module with a loss method.
    optimizer : `obj`
        One of pytorches optimizers (f.e. torch.optim.Adam)
    """
    testLoss = 0
    with torch.no_grad():
        for i, (data, labels) in enumerate(dataLoader):
            data = data.to(DEVICE)
            testLoss += model.loss(data).item()
    testLoss /= len(dataLoader.dataset)
    print(f"Test set loss: {testLoss:.8f}\n")


def learn(trainDataLoader, testDataLoader, model, epochs, learningRate=1e-3,
          verbosity=20):
    """Trains a model for n epochs using Adam optimizer, and then estimates
    test dataset losses.

    Parameters
    ---------
    trainDataLoader: `torch.DataLoader`
        Generator that returnes batched train data.
    testDataLoader: `torch.DataLoader`
        Generator that returnes batched test data.
    model : `obj`
        One of the Autoencoders or some other nn.Module with a loss method.
    epochs : `int`
        Number of epochs to train for.
    learningRate : `float`, optional
        Learning rate of the Adam optimizer. Default: 0.001
    verbosity: `int`
        How often are batch losses printed, larger number means less prints.
        Epoch average loss is always printed. Default: 20
    """
    optimizer = optim.Adam(model.parameters(), lr=learningRate)
    for epoch in range(1, epochs + 1):
        train(trainDataLoader, model, optimizer, epoch, verbosity=verbosity)
        test(testDataLoader, model, optimizer)


def showimages(images, interpolation="nearest"):
    """Given a list of images (np.array or torch.tensor) creates a figure with
    sufficient number of columns and plots each image in a separate one.

    Parameters
    ----------
    images : `list` or `tuple`
        List of images to plot

    Returns
    -------
    fig : `matplotlib.pyplot.Figure`
        Figure
    axes : `matplotlib.pyplot.Axes`
        Axes
    """
    fig, axes = plt.subplots(1, len(images), figsize=(10, 3), sharex=True, sharey=True)
    axes = np.array([axes]) if len(images) == 1 else axes
    for img, ax in zip(images, axes.ravel()):
        ax.imshow(img.cpu(), interpolation=interpolation)
        ax.set_axis_off()
    fig.set_tight_layout(True)
    return fig, ax


def plot_reconstructions(dataLoader, model):
    """Given a trained model reconstructs a single example of each of the unique
    labels present in the data and plots them side by side.

    Parameters
    ----------
    dataLoader : `torch.DataLoader`
        Generator that returns batched data
    model : `obj`
        One of the Autoencoders, or some other nn.Module.
    """
    # get unique digits in the dataset via this monstrosity. First dataset is
    # the subset, second dataset is the whole dataset, the targets are the
    # labels of that dataset. We can only plot for the digits we trained for,
    # so sample unique digits from the subset. Convert to numpy to get non-
    # tensor values and finally list gets us .pop()
    allData = dataLoader.dataset.dataset
    subsetIdxs = dataLoader.dataset.indices
    digits = list(allData.targets[subsetIdxs].unique().numpy())
    stacks = []
    for data, label in allData:
        data = data.to(DEVICE)
        if label == digits[0]:
            with torch.no_grad():
                reconstruction = model(data)
            stack = torch.cat([data[0], reconstruction.view(28, 28)])
            stacks.append(stack)
            digits.pop(0)
        if len(digits) == 0:
            break
    return showimages(stacks)


def A3a(epochs=5, learningRate=1e-3, batchSize=128, verbosity=5):
    """Loads MNIST train and test datasets, batches them, trains a single layer
    linear Autoencoder, reports on the train and test losses and plots the
    reconstructons.

    Parameters
    ----------
    epochs : `int`, optional
       Number of training epochs. Default:
    learningRate : `float`, optional
       Optimizers learning rate Default: 1e-3.
    batchSize : `int`, optional
        Size of the batched dataset. Default: 128
    """
    trainLoader, testLoader = load_mnist_dataset(batchSize=batchSize)
    for latentSize in (32, 64, 128):
        AC = A3aAutoencoder(latentSize=latentSize).to(DEVICE)
        learn(trainLoader, testLoader, AC, epochs, learningRate, verbosity)
        fig, axes = plot_reconstructions(trainLoader, AC)
        fig.suptitle("Digit reconstruction, linear \n"
                     f"(h={latentSize}, Nbatch={batchSize}, epochs={epochs}, "
                     f"lr={learningRate})")
        fig.savefig(f"../HW4_plots/A3a_h{latentSize}.png")
    plt.show()


def A3b(epochs=5, learningRate=1e-3, batchSize=128, verbosity=5):
    """Loads MNIST train and test datasets, batches them, trains a single layer
    non-linear Autoencoder, reports on the train and test losses and plots the
    reconstructons.

    Parameters
    ----------
    epochs : `int`, optional
       Number of training epochs. Default:
    learningRate : `float`, optional
       Optimizers learning rate Default: 1e-3.
    batchSize : `int`, optional
        Size of the batched dataset. Default: 128
    """
    trainLoader, testLoader = load_mnist_dataset(batchSize=batchSize)

    for latentSize in (32, 64, 128):
        AC = A3bAutoencoder(latentSize=latentSize).to(DEVICE)
        learn(trainLoader, testLoader, AC, epochs, learningRate, verbosity)
        fig, axes = plot_reconstructions(trainLoader, AC)
        fig.suptitle("Digit reconstruction, non-linear \n"
                     f"(h={latentSize}, Nbatch={batchSize}, epochs={epochs}, "
                     f"lr={learningRate})")
        fig.savefig(f"../HW4_plots/A3b_h{latentSize}.png")
    plt.show()


def A3():
    """Runs part a and b of problem A3 in HW4"""
    A3a()
    A3b()


if __name__ == "__main__":
    A3()
    pass
```

```
class MyAutoencoder(MNISTAutoencoder):
    def __init__(self, imgSize=28*28, hiddenSize=128, latSize=2, **kwargs):
        super().__init__(
            encode = nn.Sequential(
                nn.Linear(imgSize, hiddenSize),
                nn.ReLU(),
                nn.Linear(hiddenSize, latSize)
            ),
            decode = nn.Sequential(
                nn.Linear(latSize, hiddenSize),
                nn.ReLU(),
                nn.Linear(hiddenSize, imgSize)
            ),
            imgSize = imgSize,
            hiddenSize = hiddenSize,
            latentSize = latSize,
            **kwargs
        )

    def forward_mu(self, x):
        """Get the latent space layer back."""
        return self.encode(x.view(-1, self.imgSize))



def plot_latent(dataLoader, model, vectors=(0,1)):
    """For 2D latent spaces
    """

    digits = range(10)
    colors = [cm.tab10(digit/10.0) for digit in digits]
    coldict = {d:c for d, c in zip(digits, colors)}

    fig, axes = plt.subplots()

    # fake legend
    for d, c in zip(digits, colors):
        plt.scatter(-100, -100, c=[c], label=d)
    plt.legend()

    # get the latent space values and plot them in colors of digits
    for i, (data, labels) in enumerate(dataLoader):
        with torch.no_grad():
            data = data.to(DEVICE)
            mu = model.forward_mu(data.view(-1, 784))
            for digit, color in zip(digits, colors):
                digitMask = labels == digit
                x, y = vectors
                plt.scatter(mu[:, x][digitMask].cpu(), mu[:, y][digitMask].cpu(),
                            c=[color], alpha=0.12)

    plt.title("Autoencoders Latent space")
    plt.xlabel(f"Dimension {vectors[0]}")
    plt.ylabel(f"Dimension {vectors[1]}")
    plt.xlim(-15, 15)
    plt.ylim(-5, 5)
    plt.tight_layout()
    return fig, axes



def extra_autoencoder(epochs=10, learningRate=1e-3, batchSize=1024):
    trainLoader, testLoader = load_mnist_dataset(batchSize=batchSize)

    hiddenDim, latDim = 400, 3
    AC = MyAutoencoder(hiddenSize=hiddenDim, latSize=latDim).to(DEVICE)
    learn(trainLoader, testLoader, AC, epochs, learningRate)
    fig, axes = plot_reconstructions(trainLoader, AC)
    fig.suptitle("Digit reconstruction, non-linear (my autoencoder) \n"
                 f"(h={hiddenDim}, latDim={latDim}, Nbatch={batchSize}, epochs={epochs}, "
                 f"lr={learningRate})")
    plt.show()

    fig, axes = plot_latent(trainLoader, AC)
    fig.savefig("../HW4_plots/MyAutoencoder_LatentSpace1.png")
    plt.show()
    fig, axes = plot_latent(trainLoader, AC, (0, 2))
    fig.savefig("../HW4_plots/MyAutoencoder_LatentSpace2.png")
    plt.show()
    fig, axes = plot_latent(trainLoader, AC, (1, 2))
    fig.savefig("../HW4_plots/MyAutoencoder_LatentSpace3.png")
    plt.show()


extra_autoencoder()
```

Unsupervised Learning with Autoencoders

A5. In this problem we will explore different deep learning architectures for image classification on the CIFAR-10 dataset.
For all of the following, apply a hyperparameter tuning method (grid search, random search, etc.) using the validation set, report the hyperparameter configurations you evaluated and the best set of hyperparameters from this set, and plot the training and validation classification accuracy as a function of iteration. Produce a separate line or plot for each hyperparameter configuration evaluated. Finally, evaluate your best set of hyperparameters on the test data and report the accuracy. On the larger networks, you should expect to tune hyperparameters and train to at least 70% accuracy. Here are the network architectures you will construct and compare.

a. *[15 points]* Fully-connected output, 0 hidden layers (logistic regression): this network has no hidden layers and linearly maps the input layer to the output layer.



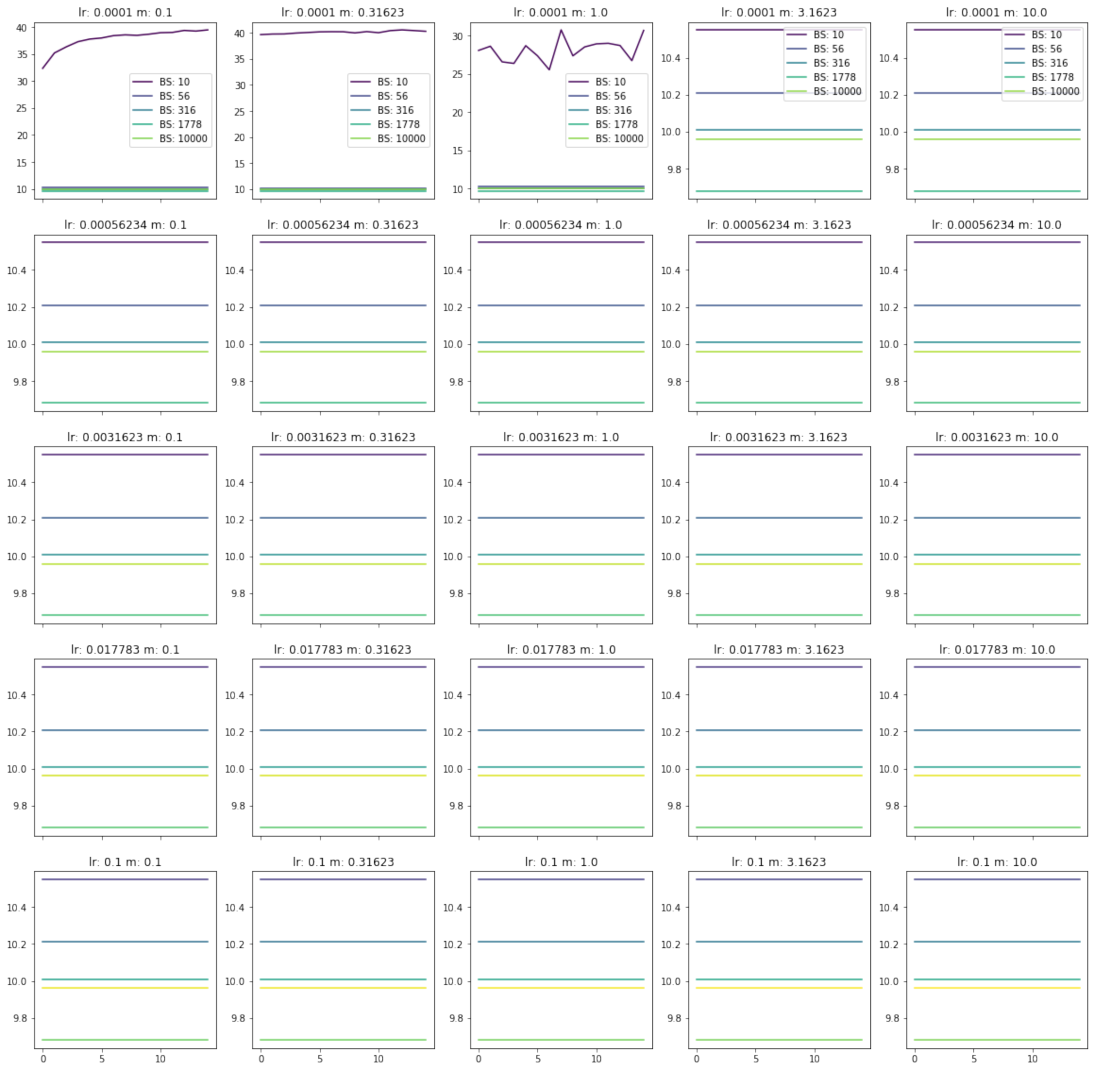


Figure 4: SGD optimizer

b. *[15 points]* Fully-connected output, 1 fully-connected hidden layer: this network has one hidden layer denoted. M will be a hyperparameter you choose (M could be in the hundreds). The non linearity applied to the hidden layer will be the relu.

c. *[15 points]* Convolutional layer with max-pool and fully-connected output.

d. Tuning: Return to the original network you were left with at the end of the tutorial Training a classifier. Tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully-connected layers, step size, etc.) and train for a sufficient number of iterations to achieve a test accuracy of at least 70%. Provide the hyperparameter configuration used to achieve this performance.

This code produced the above plots and contains the code for the remaining solutions. I ran into a problem with the grid_search implementation because I opted to use the GPUs to speed up the process. If you look at the SGD plots they are not correct after the first iteration. Running the sampling (commented out code at the end) for the same parameters I get different results than the ones that actually get saved. I don't know hwere the problem is. It obviously should not be constant accuracy across the epoch (and as I said, when I run that sampling manually it isn't) but when I run the A5a sampler then suddenly it is. I've struggled for two days trying to fix that problem but haven't been able to figure out where the problem is. I'm out of time so I'm submitting what I have. The are different when run out of the loop, but in it they all end up being the same. Running on CPU took 558 minutes (9.3 hours) for A5a which means it's just not possible for me to finish this on time unless I run on GPU (about 1.5hours on a Tesla). Hopefully it'll count for some points at least. Things to note from manual runs is that SGD seems to perform better on smaller batches (because averaging over many points tends to get it stuck in some local minima whereas jumps are more erratic on smaller batch sizes) and Adam seems to perform as expected. It's not too difficult to reach 70% accuracy in those cases for enough epochs.

```python
import timeit

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import cm

import torch
from torch import nn, optim
from torch.nn import functional
import torch.utils.data as datutils

from torchvision import datasets, transforms
from torchvision.utils import make_grid


torch.manual_seed(0)
np.random.seed(0)
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
PATH = 'data/cifarTrained.pth'


def load_cifar_dataset(path="data/cifar_data/", pickClass=None, batchSize=1,
                       validationFrac=0.2, nWorkers=2):
    """Loads CIFAR data located at path.

    CIFAR data are 28x28 pixel large images of numbers.

    Parameters
    ----------
    path : `str`
        Path to the data directory
    pickClass : `int` or `str`, optional
        Pick which class to load. If None, all classes are loaded.
    batchSize: `int`, optional
        Batch size. Default: 1
    validationFrac : `float`, optional
        Fraction of train data that will be separated into validation
        dataset.
    nWorkers : `int`
        Number of threads that will dedicately load the data.

    Returns
    -------
    trainLoader : `torch.DataLoader`
        A generator that will shuffle and batch the data at every iteration,
        yields train datasets
    validationLoader : `torch.DataLoader`
        Generator that returns the whole validation dataset.
    trainLoader : `torch.DataLoader`
        Generator that returns the whole test dataset.

    Notes
    -----
    Data are normalized upon loading to the mean of the dataset.
    Data are downloaded if not present at path.
    """
    trans = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    train = datasets.CIFAR10(path, train=True, download=True, transform=trans)
    test = datasets.CIFAR10(path, train=False, transform=trans)

    trainMask, testMask = np.arange(len(train)), np.arange(len(test))
    if pickClass is not None:
        if type(pickClass) == str:
            pickClass = train.classes.index(pickClass)
        trainMask = np.where(train.targets == pickClass)[0]
        testMask = np.where(test.targets == pickClass)[0]

    maskedTrain = datutils.Subset(train, trainMask)
    maskedTest = datutils.Subset(test, testMask)

    trainLen = int((1-validationFrac)*len(maskedTrain))
    validLen = int(validationFrac*len(maskedTrain))
    maskedTrain, maskedValidation = datutils.random_split(maskedTrain,
                                                          [trainLen, validLen])

    trainLoader = datutils.DataLoader(maskedTrain, batch_size=batchSize,
                                      shuffle=True, num_workers=nWorkers)
    validationLoader = datutils.DataLoader(maskedValidation, batch_size=batchSize,
                                           shuffle=False, num_workers=nWorkers)
    testLoader = datutils.DataLoader(maskedTest, batch_size=batchSize,
                                     shuffle=False, num_workers=nWorkers)

    return trainLoader, validationLoader, testLoader


def train(dataLoader, model, optimizer, epoch, validationLoader=None,
          verbosity=5, toSTD=False):
    """Trains an epoch of the model using the given batched data. Calculates
    loss for each batch as well as the total average loss across the epoch
    and prints them.

    Parameters
    ----------
    dataLoader: `torch.DataLoader`
        Generator that returns batched train data.
    model : `obj`
        One of the Autoencoders or some other nn.Module with a loss method.
    optimizer : `obj`
        One of pytorches optimizers (f.e. torch.optim.Adam)
    epoch : `int`
        What epoch is this training step performing, used to calculate loss
    validationLoader : `torch.DataLoader`, optional
        Generator that returns batched validation data. If None, accuracy is
        not evaluated for the validation set.
    verbosity: `int`, optional
        How often are batch losses printed, larger number means more outputs.
    toSTD: `bool`, optional
        By default False, i.e. no output is printed. When True it prints each
        individual mini-batch, average and validation (if present) accuracy and
        loss.

    Returns
    -------
    validationAccuracy : `float` or `None`
        If validation dataset is provided validation accuracy is returned,
        otherwise nothing is returned.
    """
    if toSTD:
        verbosity = 1 if nBatches/verbosity < 1 else np.ceil(nBatches/verbosity)
        trainLoss, avgTrainLoss, trainAcc, avgTrainAcc = 0, 0, 0, 0
        print(f"Epoch: {epoch}:")

    # DataLoader length is number of batches that fit in the dataset.
    # Length of the dataset is the actual number of data points
    # used (f.e. the total number of CIFAR images)
    nAll, nBatches = len(dataLoader.dataset), len(dataLoader)
    for i, (data, labels) in enumerate(dataLoader):
        data = data.to(DEVICE)
        labels = labels.to(DEVICE)

        optimizer.zero_grad()
        # allow for multi-GPU via DataParallel
        try:
            loss = model.loss(data, labels)
        except AttributeError:
            loss = model.module.loss(data, labels)
        loss.backward()
        optimizer.step()

        if toSTD:
            acc = model.accuracy(data, labels)
            lss = loss.item()
            trainAcc += acc
            avgTrainAcc += acc
            trainLoss += lss
            avgTrainLoss += lss
            if i % verbosity == 0 and i!=0:
                # The length of data, loaded by loader, is at most the batch size,
                # not neccessarily equal for all batches (i.e. last one might be shorter).
                nBatch = len(data)
                print(
                    f"    [{i*nBatch:>6}/{nAll:>6} ({100.0*i/nBatches:<5.4}%)]"
                    f"    Loss: {trainLoss/verbosity:>10.8f}    Accuracy: {trainAcc/verbosity:>10.8f}"
                )
                trainLoss = 0.0
                trainAcc = 0.0

            msg = (f"    Avg train loss: {avgTrainLoss/nBatches:>15.4f} \n"
                   f"    Avg train accuracy: {avgTrainAcc/nBatches:>11.4f}\n")

    if validationLoader is not None:
        validationAccuracy = model.accuracy(validationLoader)
        if toSTD:
            msg += f"    Validation accuracy: {validationAccuracy[-1]:>10.4f}"

    if toSTD:
        print(msg+"\n")
    if validationLoader is not None:
        return validationAccuracy


def test(dataLoader, model):
    """Calculate and print test losses.

    Parameters
    ----------
    dataLoader: `torch.DataLoader`
        Generator that returnes batched train data.
    model : `obj`
        One of the Autoencoders or some other nn.Module with a loss method.
    optimizer : `obj`
        One of pytorches optimizers (f.e. torch.optim.Adam)
    """
    classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
               'ship', 'truck')
    totCorrect = 0
    total = 0
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))
    with torch.no_grad():
        for data, labels in dataLoader:
            data = data.to(DEVICE)
            labels = labels.to(DEVICE)
            outputs = net(data)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct = (predicted == labels)
            totCorrect += correct.sum().item()
            c = correct.squeeze()
            for i in range(4):
                label = labels[i]
                class_correct[label] += c[i].item()
                class_total[label] += 1

    for i in range(10):
        print('Accuracy of %5s : %2d %%' % (
            classes[i], 100 * class_correct[i] / class_total[i]))

    print('Test accuracy: %d %%' % (
        100 * totCorrect / total))


def imshow(img):
    """Unnormalizes and displays CIFAR images."""
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


class ConvolutionalNeuralNet(nn.Module):
    """Base class for convolutional neural networks. Defines functionality
    common to all networks without defining a network itself.
    Children need to overload the forward method.
    """
    def __init__(self, *args, **kwargs):
        """Create an instance of object.

        Parameters
        ----------
        lossFunction : `obj`
            One of the `torch.nn` loss function modules. By default
            `toch.nn.CrossEntropyLoss()`
        """
        super().__init__()
        self.lossFunction = kwargs.pop("lossFunction", nn.CrossEntropyLoss())

    def forward(self, x):
        """Projects the data to the latent space. Needs to be implemented by
        children.
        """
        raise NotImplementedError("Forward must be implemented by child class!")

    def loss(self, data, labels=None):
        """Given the data, calculate MSE loss of the reconstruction. Ensure the
        autoencoder has been trained before calling loss.

        Parameters
        -----------
        x : `torch.tensor`
            Data

        Returns
        -------
        loss : `nn.CrossEntropy`
            Cross entropy Loss
        """
        reconstructed = self.forward(data)
        labels = data.view(-1, self.imgSize) if labels is None else labels
        loss = self.lossFunction(reconstructed, labels)
        return loss

    def nFeatures(self, x):
        """Given data (plural) resolves the dimensions of datum and returns it.
        Useful for flattening incoming data into a 1D array of features.

        Parameters
        ----------
        x : `torch.Tensor`
           Data

        Returns
        -------
        numFeatures : `int`
            Dimensionality of single datum.
        """
        size = x.size()[1:]
        numFeatures = 1
        for s in size:
            numFeatures *= s
        return numFeatures

    def _tensorAccuracy(self, data, labels):
        """Given data and labels calculates the total number of labels and
        number correctly predicted labels.

        Parameters
        ----------
        data : `torch.Tensor`
            Data
        labels : `torch.Tensor`
           Labels

        Returns
        -------
        correct : `int`
            Number of correctly predicted labels
        total : `int`
            Total number of labels given.
        """
        with torch.no_grad():
            outputs = self.forward(data)
        _, predicted = torch.max(outputs.data, 1)
        total = labels.size(0)
        correct = (predicted == labels).sum().item()
        return correct, total

    def accuracy(self, data, labels=None):
        """Given data calculates the prediction accuracy.

        Parameters
        ----------
        data : `torch.Tensor` or `torch.DataLoader`
            Data on which to evaluate accuracy. When DataLoader the total
            accuracy is predicted.
        labels : `torch.Tensor`, optional
            Labels. Can be None when data is an instance of DataLoader.

        Returns
        -------
        accuracy : `float`
            Accuracy
        """
        total, correct = 0, 0
        if isinstance(data, datutils.DataLoader):
            for data, labels in data:
                data, labels = data.to(DEVICE), labels.to(DEVICE)
                good, tot = self._tensorAccuracy(data, labels)
                total += tot
                correct += good
        elif labels is not None:
            correct, total = self._tensorAccuracy(data, labels)
        return 100 * correct/total


class TutorialNet(ConvolutionalNeuralNet):
    """Modified tutorial convolutional neural network. Problem A5d.
    """
    def __init__(self, M1=6, M2=16, k1=120, k2=84, *args, **kwargs):
        super().__init__(**kwargs)
        self.forward1 = nn.Sequential(
            nn.Conv2d(3, M1, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(M1, M2, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
        )
        self.forward2 = nn.Sequential(
            nn.Linear(M2*5*5, k1),
            nn.ReLU(),
            nn.Linear(k1, k2),
            nn.ReLU(),
            nn.Linear(k2, 10)
        )
        self.M1 = M1
        self.M2 = M2
        self.k1 = k1
        self.k2 = k2

    def forward(self, x):
        y = self.forward1(x)
        z = y.view(-1, self.nFeatures(y))
        return self.forward2(z)


class NoLayerNet(ConvolutionalNeuralNet):
    """Convolutional network as defined by problem A5a."""
    def __init__(self, *args, **kwargs):
        super().__init__(**kwargs)
        self.linear = nn.Linear(3072, 10)

    def forward(self, x):
        x = x.view(-1, self.nFeatures(x))
        return self.linear(x)


class SingleLayerNet(ConvolutionalNeuralNet):
    """Convolutional network as defined by problem A5b."""
    def __init__(self, N, M, k, *args, **kwargs):
        super().__init__(**kwargs)
        finSize = int( ((33 - k) / N)**2 * M)
        self.forward1 = nn.Sequential(
            nn.Linear(3072, M),
            nn.ReLU(),
            nn.Linear(M, 10)
        )
    def forward(self, x):
        y = x.view(-1, self.nFeatures(x))
        return self.forward1(x)


class ConvLayerNet(ConvolutionalNeuralNet):
    """Convolutional network as defined by problem A5c."""
    def __init__(self, N, M, k, *args, **kwargs):
        super().__init__(**kwargs)
        finSize = int( ((33 - k) / N)**2 * M)
        self.forward1 = nn.Sequential(
            nn.Conv2d(3, M, k),
            nn.ReLU(),
            nn.MaxPool2d(N, N)
        )
        self.forward2 = nn.Conv2d(finSize, 10)
        self.N = N
        self.M = M
        self.k = k
        self.finSize = finSize

    def forward(self, x):
        y = self.forward1(x)
        z = y.view(-1, self.nFeatures(y))
        return self.forward2(x)


def learn(trainDataLoader, validationDataLoader, model, optimizer=None,
          epochs=10, learningRate=1e-3, momentum=0.9, verbosity=5):
    """Trains a model for n epochs using given optimizer, and then records
    validation accuracies.

    Parameters
    ---------
    trainDataLoader: `torch.DataLoader`
        Generator that returnes batched train data.
    validationDataLoader: `torch.DataLoader`
        Generator that returnes batched validation data.
    model : `obj`
        One of the Autoencoders or some other nn.Module with a forward and
        accuracy method.
    optimizer : `obj`, optional
        An instantiated `torch.optim` optimizer. If None `toch.SGD` is used.
    epochs : `int`, optional
        Number of epochs to train for. Default: 10
    learningRate : `float`, optional
        Learning rate of the SGD optimizer. Default 0.001.
    momentum : `float`, optional
        Momentum of the SGD optimizer, default: 0.9
    verbosity: `int`, optional
        How often are batch losses printed, larger number means less prints.
        Epoch average loss is always printed. Default: 5

    Returns
    -------
    accuracy : `list`
        Accuracy per training epoch.
    """
    if optimizer is None:
        optimizer = optim.SGD(model.parameters(), lr=learningRate, momentum=momentum)

    acc = []
    for epoch in range(1, epochs + 1):
        vac = train(trainDataLoader, model, optimizer, epoch, verbosity=verbosity,
                    validationLoader=validationDataLoader)
        acc.append(vac)

    return acc


def A5a():
    """Runs grid search over different learning rates, batch sizes, momenta for
    both Adam and SGD optimizers and saves the recorded data.
    """
    netSGD = NoLayerNet()
    netSGD = netSGD.to(DEVICE)
    netAdam = NoLayerNet()
    netAdam = netAdam.to(DEVICE)

    epochs = 15
    batchSizes = np.logspace(1, 4, 5, dtype=int)
    momenta = np.logspace(-1, 1, 5)
    learningRates = np.logspace(-4, -1, 5)

    counter = 0
    startt = timeit.default_timer()
    validationAccuracySGD = []
    validationAccuracyAdam = []
    for batchSize in batchSizes:
        trainData, validationData, testData = load_cifar_dataset(batchSize=int(batchSize))
        for learningRate in learningRates:
            print(f"Batch: {np.where(batchSizes == batchSize)[0][0]}/{len(batchSizes)} "
                  f"    LR: {np.where(learningRates == learningRate)[0][0]}/{len(learningRates)} \n"
                  f"    Time elapsed: {(timeit.default_timer() - startt)/60.0} minutes")
            for momentum in momenta:
                    SGD = optim.SGD(netSGD.parameters(), lr=learningRate, momentum=momentum)
                    accs = learn(trainData, validationData, netSGD, SGD, epochs=epochs)
                    validationAccuracySGD.append(accs)
            Adam = optim.Adam(netAdam.parameters(), lr=learningRate)
            accs = learn(trainData, validationData, netAdam, Adam, epochs=epochs)
            validationAccuracyAdam.append(accs)

    np.savez("A5a.npz", batches=batchSizes, epochs=epochs, momenta=momenta,
             learningRates=learningRates, sgdacc=validationAccuracySGD,
             adamacc=validationAccuracyAdam)

A5a()
```