In [ ]:
```python
import numpy as np
from matplotlib import pyplot as plt
from tqdm.auto import tqdm
import pickle
import os
```

In [ ]:
```python
# Constants
N = 10000   # Number of data points
K = 10      # Number of classes
D = 3072    # Number of dimensions
M = 50      # Number of hidden units
```

In [ ]:
```python
class Data():
    def __init__(self, filename=None):

        self.data = None
        self.hot = None
        self.labels = None

        if filename:
            self.setDataFromFile(filename)
            self.transform()

        # Precompute flipped indices
        indices = np.array(range(D))
        imageidx = indices.reshape(32, 32, 3, order="F")
        flipped_imageidx = np.flip(imageidx, axis=0)
        self.flipped_idx = flipped_imageidx.reshape(D, order="F")

    def loadData(self, filename):
        """ Copied from the dataset website """
        import pickle
        with open('./Datasets/cifar-10-batches-py/'+filename, 'rb') as fo:
            batch = pickle.load(fo, encoding='bytes')

        labels = np.array(batch[b"labels"])

        dict = {
         "labels": labels,
         "data": batch[b"data"].T.astype(float),
         "hot": self.onehotencoding(labels)
         }

        return dict

    def onehotencoding(self, labels):
        """
        One-hot encodes the given labels.

        Args:
            labels (np.array): The labels to be one-hot encoded.

        Returns:
            np.array: The one-hot encoded labels.
        """
        N = len(labels)
        hot = np.zeros((K, N))
        for i in range(N):
            hot[labels[i]][i] = 1
        return hot

    def setDataFromFile(self, fname):
        """
```

```python
        Sets the data to the contents at the file filename.
        """
        batch = self.loadData(fname)
        self.labels = batch["labels"]
        self.data = batch["data"]
        self.hot = batch["hot"]

    def concatData(self, fname):
        """
        Concatenates the data from the given filename to the existing data.

        Args:
            fname (str): The path to the file containing the data.
        """

        batch = self.loadData(fname)

        if self.labels is not None:
            self.labels = np.concatenate((self.labels, batch["labels"]))
        else:
            self.labels = batch["labels"]

        if self.data is not None:
            self.data = np.concatenate((self.data, batch["data"]), axis=1)
        else:
            self.data = batch["data"]

        self.hot = self.onehotencoding(self.labels)

    def transform(self):
        """
        Transforms the given dataset by normalizing the data.
        """
        meanX = np.mean(self.data, axis=0)
        stdX = np.std(self.data, axis=0)
        self.data = (self.data - meanX) / stdX

    def shuffle(self):
        """
        Shuffles the data.
        """
        permutation = np.random.permutation(self.data.shape[1])
        self.data = self.data[:, permutation]
        self.hot = self.hot[:, permutation]
        self.labels = self.labels[permutation]

    def miniBatch(self, batch, range):
        """
        Creates a mini-batch of the given batch.
        """
        self.data = batch.data[:, range[0]:range[1]]
        self.hot = batch.hot[:, range[0]:range[1]]
        self.labels = batch.labels[range[0]:range[1]]

    def flip(self, pflip):
        """
        Flips the data with a given probability.

        Args:
            pflip (float): The probability of flipping the data.
        """
        flip = np.random.rand(self.data.shape[1]) < pflip
        self.data[:,flip] = self.data[:,flip][self.flipped_idx]
```

Model

In [ ]:
```
class Linear2Layer:
    def __init__(self, k, d, m, seed = None):
        """
        Initializes the weights and biases of the 2 layers.
        """
        if seed:
            np.random.seed(seed)

        self.layer1 = {
            "W": np.random.normal(0, 1/np.sqrt(d), size=(m, d)),
            "b": np.zeros((m, 1)),
            "gradW": None,
            "gradB": None
        }

        self.layer2 = {
            "W": np.random.normal(0, 1/np.sqrt(m), size=(k, m)),
            "b": np.zeros((k, 1)),
            "gradW": None,
            "gradB": None
        }

        self.h = None
        self.p = None

    # Activation functions #
    def softmax(self, x):
        """ Standard definition of the softmax function """
        return np.exp(x) / np.sum(np.exp(x), axis=0)

    def sigmoid(self, x):
        """ Standard definition of the sigmoid function """
        return 1 / (1 + np.exp(-x))

    def relu(self, x):
        """ Standard definition of the ReLU function """
        return np.maximum(0, x)

    ########################################

    def forward(self, batch):
        """
        Evaluate the classifier for a given input.

        Args:
            data (dict): A dictionary containing the data and one-hot encoded labels.
        """
        X = batch.data
        s1 = self.layer1["W"] @ X + self.layer1["b"]
        self.h = self.relu(s1)
        s = self.layer2["W"] @ self.h + self.layer2["b"]
        self.p = self.softmax(s)



    def lcross(self, batch):
        """
        Calculates the cross-entropy loss.

        Args:
            data (dict): A dictionary containing the data and one-hot encoded labels.
```

```python
        Returns:
            float: Cross-entropy loss.
        """
        Y = batch.hot
        return - Y * np.log(self.p)

    def lmultiplebce(self, batch):
        """
        Calculates the K-binary cross-entropy loss.

        Args:
            data (dict): A dictionary containing the data and one-hot encoded labels.

        Returns:
            float: Binary cross-entropy loss.
        """
        Y = batch.hot
        return -Y * np.log(self.p) - (1 - Y) * np.log(1 - self.p)

    def computeCost(self, batch, lmda, loss = "lcross"):
        """
        Compute the cost function for linear regression with regularization.

        Args:
            data (dict): A dictionary containing the data and one-hot encoded labels.
            lmda (float): Regularization parameter.

        Returns:
        float: The computed cost.
        """
        X = batch.data

        reg_term = lmda * np.sum([np.sum(W ** 2) for W in [self.layer1["W"], self.layer2["W"]]])

        if loss == "lcross":
            loss_cross = self.lcross(batch)
            denom = X.shape[1]
        else:
            loss_cross = self.lmultiplebce(batch)
            denom = K

        loss_term = 1 / denom * np.sum(loss_cross)

        return loss_term + reg_term, loss_term

    def computeAcc(self, batch):
        """
        Compute the accuracy of the classifier.

        Args:
            data (dict): A dictionary containing the data and one-hot encoded labels.

        Returns:
            float: Accuracy of the classifier.

        """
        Y = batch.hot
        pred = np.argmax(self.p, axis=0)
        return np.mean(pred == np.argmax(Y, axis=0))

    def backward(self, batch, lmda):
        """
        Compute the gradients of the cost function with respect to the parameters.

        Args:
```

```python
            data (dict): A dictionary containing the data and one-hot encoded labels.
            lmbd (float): Regularization parameter.

        Returns:
            list: A list containing the gradients of the cost function with respect to the weight matrix W and the bias vector b.
        """
        X, Y = batch.data, batch.hot

        G = self.p - Y

        self.layer2["gradW"] = 1 / X.shape[1] * G @ self.h.T + 2 * lmda * self.layer2["W"]
        self.layer2["gradB"] = 1 / X.shape[1] * np.sum(G, axis = 1).reshape((K,1))

        G = self.layer2["W"].T @ G
        G = G * (self.h > 0)

        self.layer1["gradW"] = 1 / X.shape[1] * G @ X.T + 2 * lmda * self.layer1["W"]
        self.layer1["gradB"] = 1 / X.shape[1] * np.sum(G, axis = 1).reshape((M,1))

    def GradCenteredDifference(self, batch, lmda, h = 1e-6):
        """
        Compute the gradients of the cost function with respect to the parameters using centered difference.

        Args:
            data (dict): A dictionary containing the data and one-hot encoded labels.
            lmbd (float): Regularization parameter.
            h (float): The step size for the centered difference.

        Returns:
            list: A list containing the gradients of the cost function with respect to the weight matrix W and the bias vector b.
        """

        gradW1 = np.zeros(self.layer1["W"].shape)
        gradB1 = np.zeros(self.layer1["b"].shape)
        gradW2 = np.zeros(self.layer2["W"].shape)
        gradB2 = np.zeros(self.layer2["b"].shape)

        for i in range(self.layer1["W"].shape[0]):
            for j in range(self.layer1["W"].shape[1]):
                self.layer1["W"][i,j] += h
                self.forward(batch)
                cost1, _ = self.computeCost(batch, lmda)
                self.layer1["W"][i,j] -= 2 * h
                self.forward(batch)
                cost2, _ = self.computeCost(batch, lmda)
                gradW1[i,j] = (cost1 - cost2) / (2 * h)
                self.layer1["W"][i,j] += h

        for i in range(self.layer1["b"].shape[0]):
            self.layer1["b"][i] += h
            self.forward(batch)
            cost1, _ = self.computeCost(batch, lmda)
            self.layer1["b"][i] -= 2 * h
            self.forward(batch)
            cost2, _ = self.computeCost(batch, lmda)
            gradB1[i] = (cost1 - cost2) / (2 * h)
            self.layer1["b"][i] += h

        for i in range(self.layer2["W"].shape[0]):
            for j in range(self.layer2["W"].shape[1]):
                self.layer2["W"][i,j] += h
                self.forward(batch)
                cost1, _ = self.computeCost(batch, lmda)
                self.layer2["W"][i,j] -= 2 * h
                self.forward(batch)
                cost2, _ = self.computeCost(batch, lmda)
```

```python
                gradW2[i,j] = (cost1 - cost2) / (2 * h)
                self.layer2["W"][i,j] += h

        for i in range(self.layer2["b"].shape[0]):
            self.layer2["b"][i] += h
            self.forward(batch)
            cost1, _ = self.computeCost(batch, lmda)
            self.layer2["b"][i] -= 2 * h
            self.forward(batch)
            cost2, _ = self.computeCost(batch, lmda)
            gradB2[i] = (cost1 - cost2) / (2 * h)
            self.layer2["b"][i] += h

        return gradW1, gradB1, gradW2, gradB2

    def update(self, eta):
        """
        Update the parameters of the model.
        """
        for layer in [self.layer1, self.layer2]:
            layer["W"] -= eta * layer["gradW"]
            layer["b"] -= eta * layer["gradB"]
```

In [ ]:

```python
class Visualizer:
    def plotResults(self, title, costs, loss, accs, test_acc = None):
        """
        Plot the results of the training.
        """
        plt.figure(figsize=(16, 6))
        plt.suptitle(title)

        plt.subplot(1, 3, 1)
        plt.plot(costs["train"], label="Training")
        plt.plot(costs["val"], label="Validation")
        plt.xlabel("Epoch")
        plt.ylabel("Cost")
        plt.legend()

        plt.subplot(1, 3, 2)
        plt.plot(loss["train"], label="Training")
        plt.plot(loss["val"], label="Validation")
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.legend()

        plt.subplot(1, 3, 3)
        plt.plot(accs["train"], label="Training")
        plt.plot(accs["val"], label="Validation")
        if test_acc:
            plt.axhline(test_acc, color="red", label="Test Accuracy")
        plt.xlabel("Epoch")
        plt.ylabel("Accuracy")
        plt.legend()

        plt.tight_layout()
        plt.show()

    def genWeightImage(self, slice):
        """
        Generates an image from the given slice of the weight matrix.

        Args:
            slice (numpy.ndarray): The slice of the weight matrix.

        Returns:
```

```python
            numpy.ndarray: The generated image.
        """

        img = slice.reshape(32, 32, 3, order="F")
        img = img - np.min(img)
        img = img / np.max(img)
        return img

    def genMatrixVisualization(self, title, W):
        plt.figure(figsize=(12, 6))

        for i in range(K):
            plt.subplot(2, 5, i+1)
            plt.imshow(self.genWeightImage(W[i, :]))
            plt.title(f"Slice {i}")

        plt.tight_layout()
        plt.suptitle(title)
        plt.show()

    def plotLearningRate(self, title, lrs, lr_max, lr_min, stepsize):
        plt.figure(figsize=(12, 6))
        plt.plot(lrs)
        plt.axhline(lr_max, color="red", label="Max Learning Rate")
        plt.axhline(lr_min, color="green", label="Min Learning Rate")
        plt.xticks(range(0, len(lrs), stepsize))
        plt.xlabel("Epoch")
        plt.ylabel("Learning Rate")
        plt.title(title)
        plt.legend()
        plt.show()

    def plotCyclicResults(self, title, costs, loss, accs, stepsize, test_acc = None):
        """
        Plot the results of the training.
        """
        plt.figure(figsize=(16, 6))
        plt.suptitle(title)

        X = [i*stepsize//10 for i in range(len(costs["train"]))]

        plt.subplot(1, 3, 1)
        plt.plot(X,costs["train"], label="Training")
        plt.plot(X,costs["val"], label="Validation")
        plt.ylim(0, 1.2 * max(costs["train"]))
        plt.xlabel("Step")
        plt.ylabel("Cost")
        plt.legend()

        plt.subplot(1, 3, 2)
        plt.plot(X,loss["train"], label="Training")
        plt.plot(X,loss["val"], label="Validation")
        plt.ylim(0, 1.2 * max(loss["train"]))
        plt.xlabel("Step")
        plt.ylabel("Loss")
        plt.legend()

        plt.subplot(1, 3, 3)
        plt.plot(X,accs["train"], label="Training")
        plt.plot(X,accs["val"], label="Validation")
        if test_acc:
            plt.axhline(test_acc, color="red", label="Test Accuracy")
        plt.ylim(0, 1.2 * max(accs["train"]))
        plt.xlabel("Step")
        plt.ylabel("Accuracy")
        plt.legend()
```

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

In [ ]:
```python
def cyclicLearningRate(eta_min, eta_max, stepsize, t):
    """
    Compute the learning rate for the given stepsize and t.

    Args:
        eta_min (float): The minimum learning rate.
        eta_max (float): The maximum learning rate.
        stepsize (int): The stepsize.
        t (int): The current iteration.

    Returns:
        float: The learning rate.
    """
    l = t // (2 * stepsize)
    if 2*l*stepsize <= t <= (2*l+1)*stepsize:
        return eta_min + (t - 2*l*stepsize) / stepsize * (eta_max - eta_min)
    else:
        return eta_max - (t - (2*l+1)*stepsize) / stepsize * (eta_max - eta_min)
```

In [ ]:
```python
def miniBatchGD(train, lmbd=0.1, n_batch=100, scheduler=["static",[0.001]], n_epochs=20, val=None, pflip=0, seed=65465168, n_cycles=2):
        """
        Perform mini-batch gradient descent.

        Args:
            X (numpy.ndarray): Input data of shape (d, N).
            Y (numpy.ndarray): One-hot encoded true label of shape (K, N).
            W (numpy.ndarray): Weight matrix of shape (K, d).
            b (numpy.ndarray): Bias vector of shape (K, 1).
            lmbd (float, optional): Regularization parameter. Defaults to 0.1.
            n_batch (int, optional): Number of mini-batches. Defaults to 100.
            scheduler (list, optional): Learning rate scheduler. Defaults to static eta = 0.001.
            n_epochs (int, optional): Number of epochs. Defaults to 20.

        Returns:
            tuple: A tuple containing the weight matrix W and the bias vector b.
        """
        np.random.seed(seed)

        costs = {"train" : [], "val" : []}
        loss = {"train" : [], "val" : []}
        accs = {"train" : [], "val" : []}
        lr = []

        model = Linear2Layer(K, D, M)

        if scheduler[0] == "static":
            eta = scheduler[1][0]
        elif scheduler[0] == "cyclic":
            eta_min, eta_max, n_s = scheduler[1]
            eta = eta_min
            t = 0
        else:
            raise ValueError("Invalid scheduler")

        for epoch in range(n_epochs):
            # Shuffle data
            train.shuffle()

            n_mini_batches = int(train.data.shape[1] / n_batch)
            for j in range(n_mini_batches):
                # Assemble mini-batch
```

```
                batch = Data()
                j_start = j * n_batch
                j_end = (j + 1) * n_batch
                batch.miniBatch(train, [j_start, j_end])
                if pflip > 0:
                    batch.flip(pflip)

                if scheduler[0] == "cyclic":
                    eta = cyclicLearningRate(eta_min, eta_max, n_s, t)
                    t += 1

                model.forward(batch)
                model.backward(batch, lmbd)
                model.update(eta)

                lr.append(eta)

                if scheduler[0] == "cyclic" and t % (n_s / 10) == 0:
                    model.forward(train)
                    c, l = model.computeCost(train, lmbd)
                    costs["train"].append(c)
                    loss["train"].append(l)
                    accs["train"].append(model.computeAcc(train))
                    if val:
                        model.forward(val)
                        c, l = model.computeCost(val, lmbd)
                        costs["val"].append(c)
                        loss["val"].append(l)
                        accs["val"].append(model.computeAcc(val))

            if scheduler[0] == "static":
                model.forward(train)
                c, l = model.computeCost(train, lmbd)
                costs["train"].append(c)
                loss["train"].append(l)
                accs["train"].append(model.computeAcc(train))
                if val:
                    model.forward(val)
                    c, l = model.computeCost(val, lmbd)
                    costs["val"].append(c)
                    loss["val"].append(l)
                    accs["val"].append(model.computeAcc(val))


        return model, costs, loss, accs, lr
```

Build Parameters

Restructure for each layer to contain backward pass update depending on activation funcion for modularity. See p. 36 in slides

In [ ]:
```
train, val, test = Data("data_batch_1"), Data("data_batch_2"), Data("test_batch")
```

In [ ]:
```
subset = Data()
subset.miniBatch(train,[0,20])
```

In [ ]:
```
def relerr(ga, gn, eps=1e-6):
        """
        Calculates the relative error between two vectors.

        Args:
            ga (numpy.ndarray): Analytical gradient.
            gn (numpy.ndarray): Numerical gradient.
```

```
        eps (float, optional): A small value to avoid division by zero. Defaults to 1e-6.

    Returns:
        float: The relative error between ga and gn.
    """

    diff = np.linalg.norm(ga - gn)
    norma = np.linalg.norm(ga)
    normn = np.linalg.norm(gn)
    numer = max(eps, norma + normn)
    return diff / numer
```

Sanity check with numerical gradients

In [ ]:
```
model = Linear2Layer(K, D, M)
gradW1, gradB1, gradW2, gradB2 = model.GradCenteredDifference(subset, 0.3, h=1e-5)
model.forward(subset)
model.backward(subset, 0.3)
print("Relative error W1:", relerr(gradW1, model.layer1["gradW"]))
print("Relative error W2:", relerr(gradW2, model.layer2["gradW"]))
print("Relative error B1:", relerr(gradB1, model.layer1["gradB"]))
print("Relative error B2:", relerr(gradB2, model.layer2["gradB"]))
```

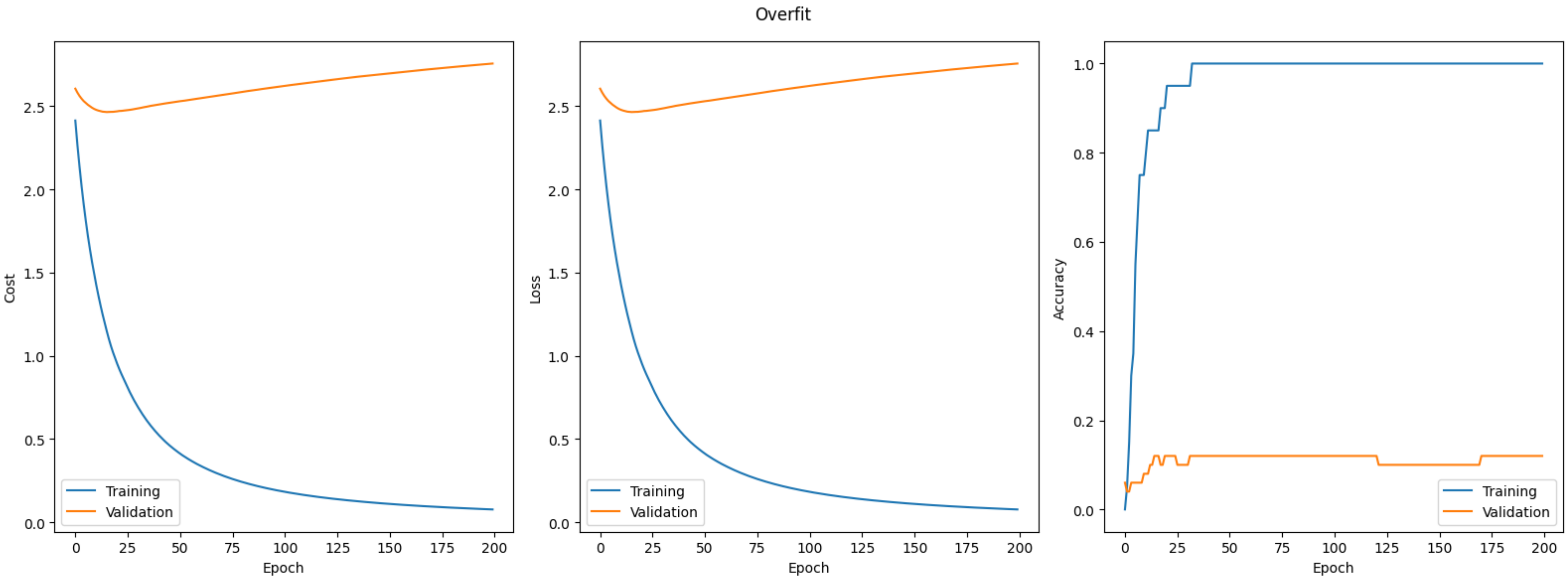
```
Relative error W1: 2.4896663946608827e-09
Relative error W2: 4.746086565087055e-10
Relative error B1: 2.077154023137515e-09
Relative error B2: 3.833461056149047e-10
```

Sanity check with overfitting on small subset

In [ ]:
```
subtrain, subval = Data(), Data()
subtrain.miniBatch(train,[0,100])
subval.miniBatch(train,[100,150])
```

In [ ]:
```
lr_scheduler = ["static",[0.001]]
modelout, costs, loss, accs, lr = miniBatchGD(subset, lmbd=0, n_batch=10, scheduler=lr_scheduler, n_epochs=200, val=subval, pflip=0)
```

In [ ]:
```
vis = Visualizer()
vis.plotResults("Overfit", costs, loss, accs)
```



Cyclical Learning Rate

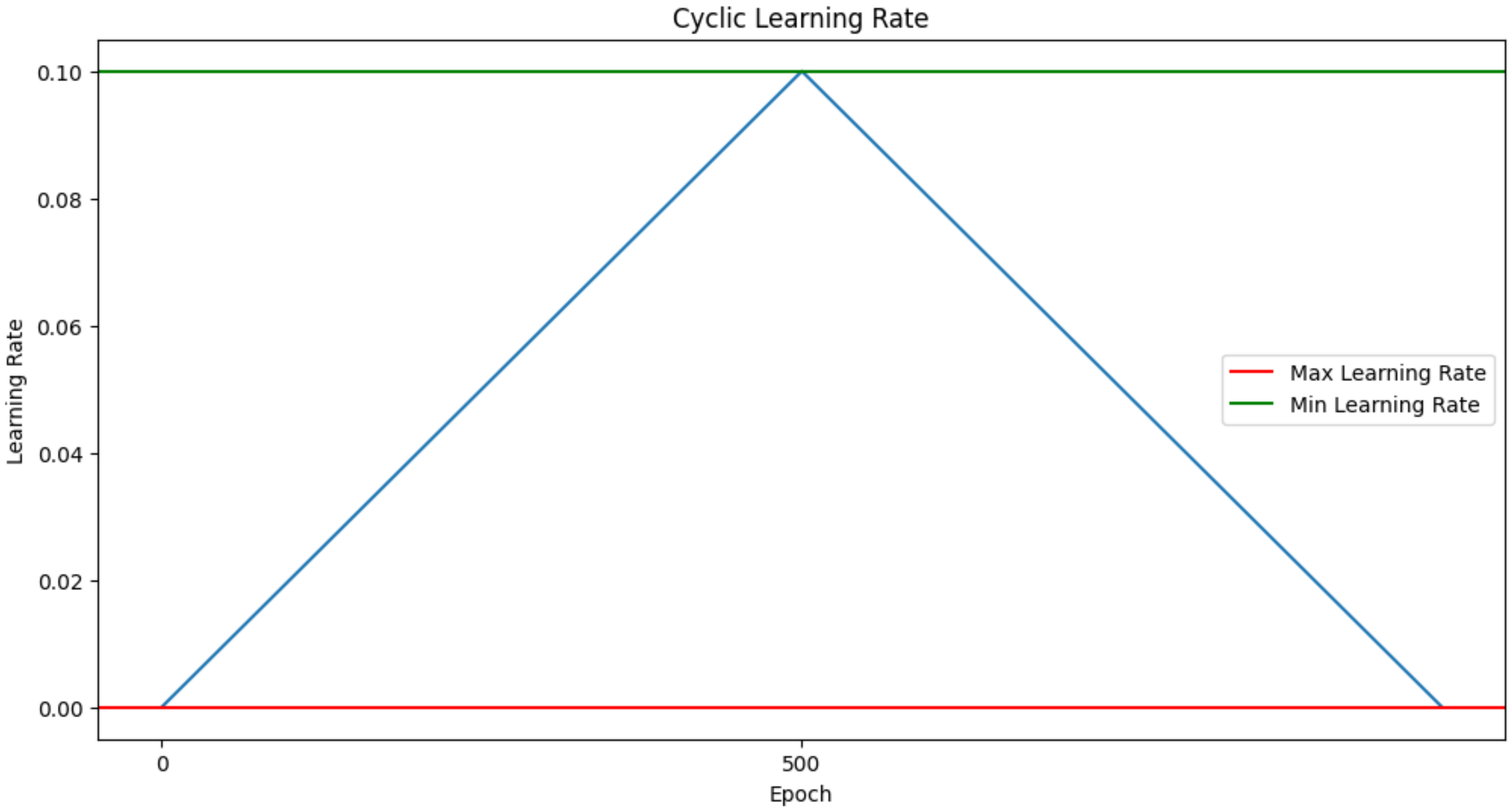
In [ ]:
```
eta_min, eta_max, stepsize = 1e-5, 1e-1, 500

lr_scheduler = ["cyclic",[eta_min, eta_max, stepsize]]

lmbd, n_batch, n_epochs = 0.01, 100, 10
```

In [ ]:
```
epoch_updates = train.data.shape[1]/n_batch
cycle_updates = 2 * stepsize
n_epochs_cycle = cycle_updates/epoch_updates

print("Updates per epoch:", epoch_updates)
print("Updates per cycle:", cycle_updates)
print("Number of epochs per cycle:", n_epochs_cycle)
```

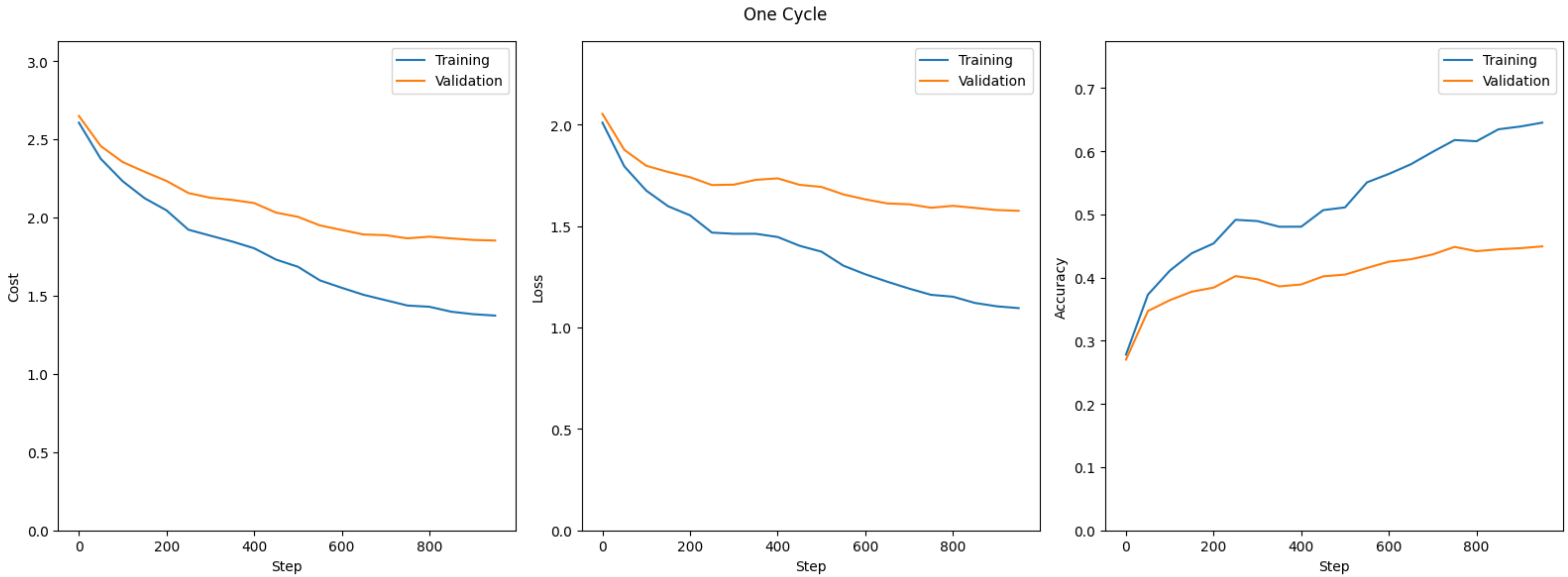
```
Updates per epoch: 100.0
Updates per cycle: 1000
Number of epochs per cycle: 10.0
```

In [ ]:
```
modelout, costs, loss, accs, lr = miniBatchGD(train, lmbd=lmbd, n_batch=n_batch, scheduler=lr_scheduler, n_epochs=n_epochs, val=val, pflip=0)
```

In [ ]:
```
vis.plotLearningRate("Cyclic Learning Rate", lr, eta_min, eta_max, stepsize)
```



In [ ]:
```
vis.plotCyclicResults("One Cycle", costs, loss, accs, stepsize)
```

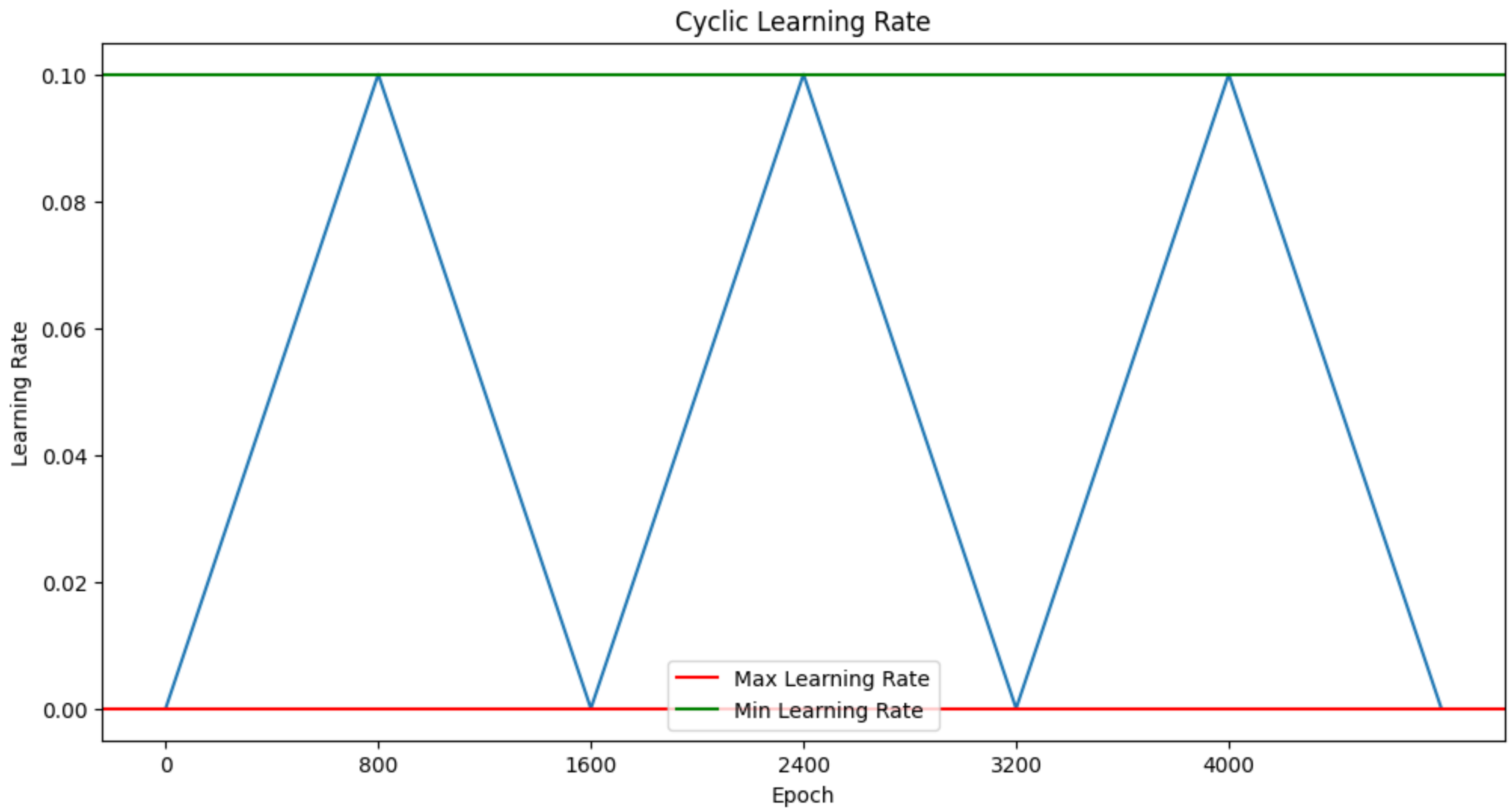


Real Training

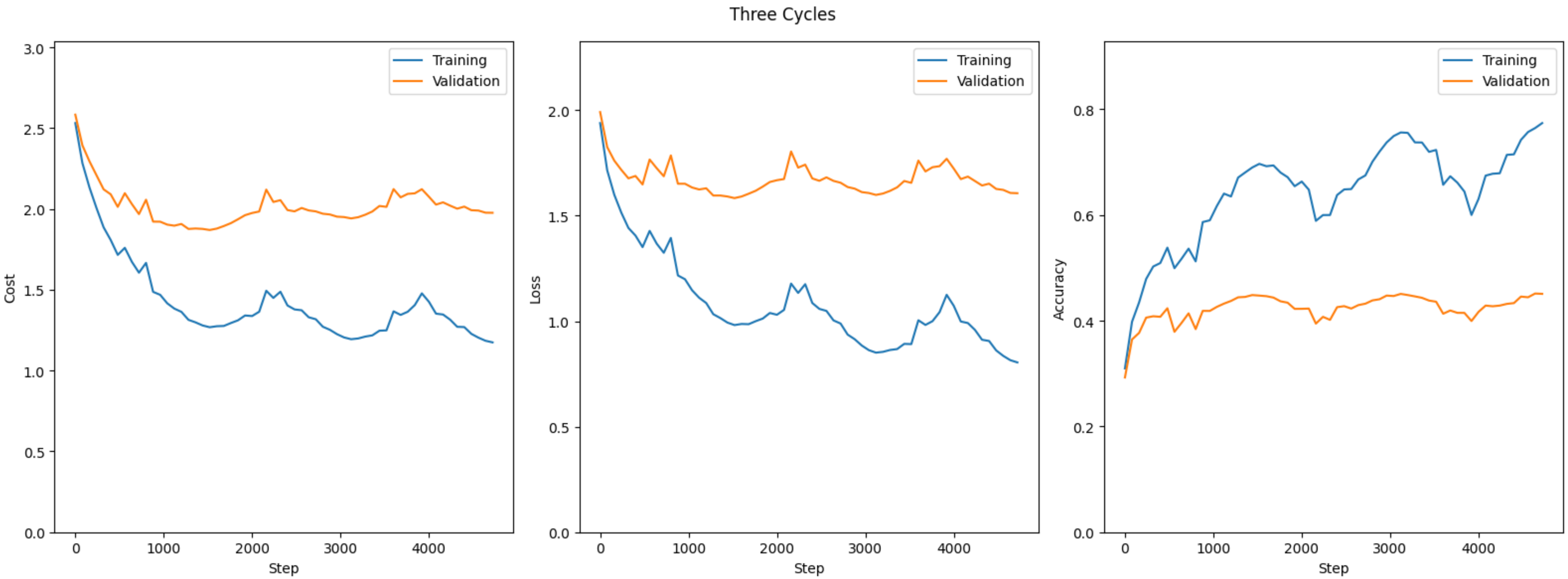
```
In [ ]: eta_min, eta_max, stepsize = 1e-5, 1e-1, 800

        lr_scheduler = ["cyclic",[eta_min, eta_max, stepsize]]

        epoch_updates = train.data.shape[1]/n_batch
        cycle_updates = 2 * stepsize
        n_epochs_cycle = cycle_updates/epoch_updates

        # 3 cycles worth of epochs
        n_epochs = int(n_epochs_cycle * 3)
```

```
In [ ]: modelout, costs, loss, accs, lr = miniBatchGD(train, lmbd=lmbd, n_batch=n_batch, scheduler=lr_scheduler, n_epochs=n_epochs, val=val, pflip=0)
```

```
In [ ]: vis.plotLearningRate("Cyclic Learning Rate", lr, eta_min, eta_max, stepsize)
```



In [ ]:
```
vis.plotCyclicResults("Three Cycles", costs, loss, accs, stepsize)
```




```
In [ ]: modelout.forward(test)
        print("Test accuracy: ", modelout.computeAcc(test))
```

```
Test accuracy:  0.4503
```

Hyperparameter Tuning

Coarse Search

```
In [ ]: # Load All Data

        # Combine all training batches
        alltrain = Data()
        for _ in range(5):
            alltrain.concatData(f"data_batch_{_+1}")
        alltrain.transform()

        val = Data()
        val.miniBatch(alltrain, [0, 5000])

        train = Data()
        train.miniBatch(alltrain, [5000,alltrain.data.shape[1]])

        del alltrain

        # Partition test batch into validation and test sets
        test = Data("test_batch")
```

In [ ]:
```
n_batch = 100
n = train.data.shape[1]
eta_min, eta_max, stepsize = 1e-5, 1e-1, 2*np.floor(n/n_batch)

lr_scheduler = ["cyclic",[eta_min, eta_max, stepsize]]

epoch_updates = n/n_batch
cycle_updates = 2 * stepsize
n_epochs_cycle = cycle_updates/epoch_updates

# 2 cycles worth of epochs
n_epochs = int(n_epochs_cycle * 2)
```

In [ ]:
```
n_search = 8
lmin, lmax = -5, -1
l = np.linspace(0, 1, n_search) * (lmax - lmin) + lmin
lambdas = np.power(10, l)
```

In [ ]:
```
val_accs = []

for i, lmbd in enumerate(lambdas):
    # If result exists, load it, otherwise train
    if os.path.isfile(f"results/lmbd_{i}.pkl"):
        with open(f"results/lmbd_{i}.pkl", "rb") as f:
            res = pickle.load(f)
    else:
        modelout, costs, loss, accs, lr = miniBatchGD(train, lmbd=lmbd, n_batch=n_batch, scheduler=lr_scheduler, n_epochs=n_epochs, val=val, pflip=0)
        res = {
            "model": modelout,
            "costs": costs,
            "loss": loss,
            "accs": accs,
            "lr": lr
        }
        with open(f"results/lmbd_{i}.pkl", "wb") as f:
            pickle.dump(res, f)

    val_accs.append(res["accs"]["val"][-1])
    print(f"Lambda {i} : {lmbd}, Validation Accuracy: {res['accs']['val'][-1]}")

best_lambda = lambdas[np.argmax(val_accs)]
print(f"Best lambda: {best_lambda}")
```

```
Lambda 0 : 9.999999999999999e-06, Validation Accuracy: 0.5108
Lambda 1 : 3.727593720314938e-05, Validation Accuracy: 0.5044
Lambda 2 : 0.00013894954943731373, Validation Accuracy: 0.5068
Lambda 3 : 0.0005179474679231212, Validation Accuracy: 0.5164
Lambda 4 : 0.0019306977288832496, Validation Accuracy: 0.5166
Lambda 5 : 0.007196856730011514, Validation Accuracy: 0.518
Lambda 6 : 0.026826957952797246, Validation Accuracy: 0.4728
Lambda 7 : 0.09999999999999999, Validation Accuracy: 0.3856
Best lambda: 0.007196856730011514
```

Finer search

In [ ]:
```
# 3 cycles worth of epochs
n_epochs = int(n_epochs_cycle * 3)

val_accs2 = []

lambdas2 = np.linspace(lambdas[np.argmax(val_accs)-1], lambdas[np.argmax(val_accs)+1], 10)

for i, lmbd in enumerate(lambdas2):
    # If result exists, load it, otherwise train
```

```
    if os.path.isfile(f"results/lmbd2_{i}.pkl"):
        with open(f"results/lmbd2_{i}.pkl", "rb") as f:
            res = pickle.load(f)
    else:
        modelout, costs, loss, accs, lr = miniBatchGD(train, lmbd=lmbd, n_batch=n_batch, scheduler=lr_scheduler, n_epochs=n_epochs, val=val, pflip=0)
        res = {
            "model": modelout,
            "costs": costs,
            "loss": loss,
            "accs": accs,
            "lr": lr
        }
        with open(f"results/lmbd2_{i}.pkl", "wb") as f:
            pickle.dump(res, f)

    val_accs2.append(res["accs"]["val"][-1])
    print(f"Lambda {i} : {lmbd}, Validation Accuracy: {res['accs']['val'][-1]}")

best_lambda = lambdas2[np.argmax(val_accs2)]
print(f"Best lambda: {best_lambda}")
```

```
Lambda 0 : 0.0019306977288832496, Validation Accuracy: 0.5094
Lambda 1 : 0.004696948864873693, Validation Accuracy: 0.5222
Lambda 2 : 0.007463200000864137, Validation Accuracy: 0.517
Lambda 3 : 0.01022945113685458, Validation Accuracy: 0.5124
Lambda 4 : 0.012995702272845024, Validation Accuracy: 0.5038
Lambda 5 : 0.015761953408835468, Validation Accuracy: 0.5032
Lambda 6 : 0.018528204544825913, Validation Accuracy: 0.4896
Lambda 7 : 0.021294455680816355, Validation Accuracy: 0.4856
Lambda 8 : 0.0240607068168068, Validation Accuracy: 0.4816
Lambda 9 : 0.026826957952797246, Validation Accuracy: 0.4734
Best lambda: 0.004696948864873693
```

Final

In [ ]:

```
# Load All Data

# Combine all training batches
alltrain = Data()
for _ in range(5):
    alltrain.concatData(f"data_batch_{_+1}")
alltrain.transform()

val = Data()
val.miniBatch(alltrain, [0, 1000])

train = Data()
train.miniBatch(alltrain, [1000,alltrain.data.shape[1]])

del alltrain

# Partition test batch into validation and test sets
test = Data("test_batch")
```

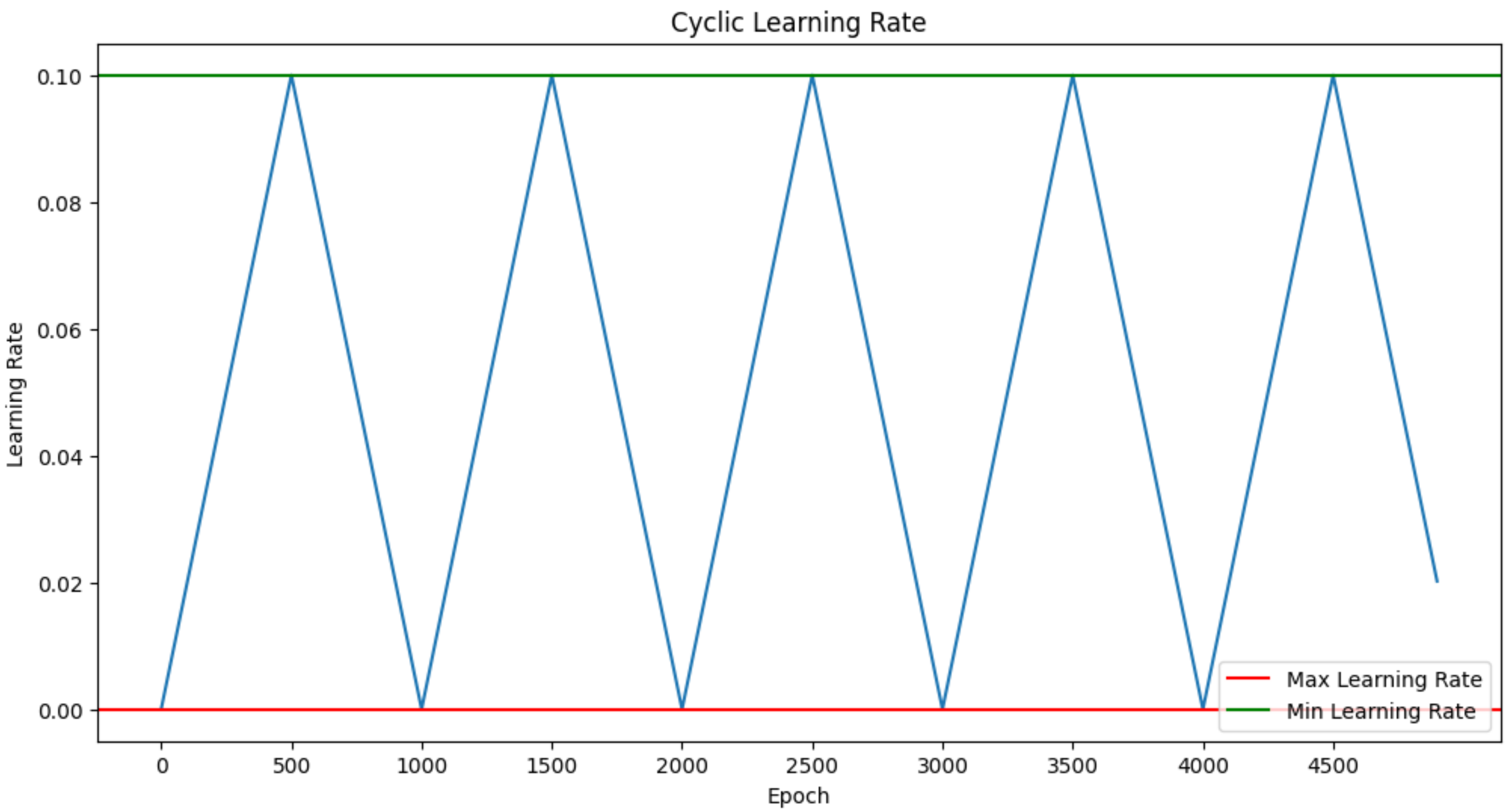
In [ ]:

```
n = train.data.shape[1]
stepsize = 500

lr_scheduler = ["cyclic",[eta_min, eta_max, stepsize]]

epoch_updates = n/n_batch
cycle_updates = 2 * stepsize
n_epochs_cycle = cycle_updates/epoch_updates

# 5 cycles worth of epochs
```

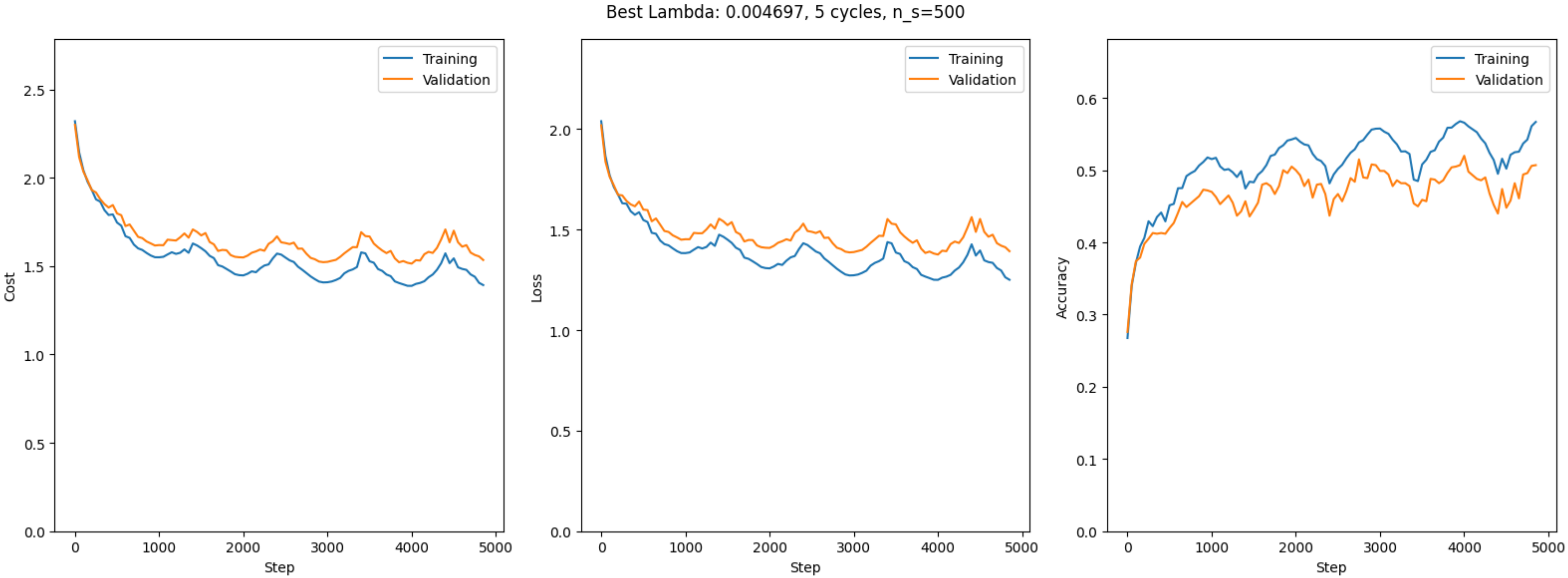
```python
n_epochs = int(n_epochs_cycle * 5)
```

In [ ]:
```python
lmbd = best_lambda

modelout, costs, loss, accs, lr = miniBatchGD(train, lmbd=lmbd, n_batch=n_batch, scheduler=lr_scheduler, n_epochs=n_epochs, val=val, pflip=0)
```

In [ ]:
```python
vis.plotLearningRate("Cyclic Learning Rate", lr, eta_min, eta_max, int(stepsize))
```



In [ ]:
```python
vis.plotCyclicResults(f"Best Lambda: {lmbd:4f}, 5 cycles, n_s={stepsize}", costs, loss, accs, stepsize)
```



In [ ]:
```
accs["val"][-1]
```

Out[ ]: 0.507

In [ ]:
```
modelout.forward(test)
print("Test accuracy: ", modelout.computeAcc(test))
```

```
Test accuracy:  0.5045
```