Assignement2

April 8, 2020

0.1 Imports

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from functions import *
from sklearn.preprocessing import OneHotEncoder
from tqdm.notebook import tqdm
from IPython.display import clear_output
%matplotlib inline
```

0.2 Load Data

Training

[2]:
```
X_train,Y_train,filenames_train = LoadBatch('data_batch_1')
Y_train_hot = one_hot(10,Y_train)
print(Y_train_hot.shape)

if False:
    w=25
    h=4
    X_visualize = X_train.reshape(10000, 3, 32, 32).transpose(0,2,3,1).
↪astype("uint8")
    visualize_multiple_images(X_visualize,h,w,True)
```

```
(10, 10000)
```

Test

[3]:
```
X_test,Y_test,filenames_test = LoadBatch('test_batch')
Y_test_hot = one_hot(10,Y_test)
Y_test_hot.shape
```

[3]:
```
(10, 10000)
```

Validation

[4]:
```
X_valid,Y_valid,filenames_valid = LoadBatch('data_batch_2')
Y_valid_hot = one_hot(10,Y_valid)
Y_valid_hot.shape
```

[4]: (10, 10000)

0.3 Preprocess

[5]:
```
X_train=preprocess(X_train)
X_test=preprocess(X_test)
X_valid=preprocess(X_valid)
print(X_test.shape)
```

(3072, 10000)

0.4 Layers

[6]:
```
class Layer:

    #A building block. Each layer is capable of performing two things:
    #- Process input to get output:           output = layer.forward(input)
    #- Propagate gradients through itself:    grad_input = layer.
↪backward(input, grad_output)
    #Some layers also have learnable parameters which they update during layer.
↪backward.

    def __init__(self):
        # Here we can initialize layer parameters (if any) and auxiliary stuff.
        # A dummy layer does nothing
        pass

    def forward(self, input):
        # Takes input data of shape [batch, input_units], returns output data␣
↪[batch, output_units]
        # A dummy layer just returns whatever it gets as input.
        return input

    def backward(self, input, grad_output):
        # Performs a backpropagation step through the layer, with respect to␣
↪the given input.
        # To compute loss gradients w.r.t input, we need to apply chain rule␣
↪(backprop):
        # d loss / d x  = (d loss / d layer) * (d layer / d x)
        num_units = input.shape[1]
```

```
        d_layer_d_input = np.eye(num_units)
        return np.dot(grad_output, d_layer_d_input) # chain rule
```

[7]:
```
class ReLU(Layer):
    def __init__(self):
        # ReLU layer simply applies elementwise rectified linear unit to all␣
↪inputs
        self.fc = False # whether it is fully connected layer

    def forward(self, input):
        # Apply elementwise ReLU to [batch, input_units] matrix
        relu_forward = np.maximum(0,input)
        return relu_forward

    def backward(self, input, grad_output):
        # Compute gradient of loss w.r.t. ReLU input
        relu_grad = input > 0
#         return np.dot(grad_output,relu_grad)
        return grad_output*relu_grad
```

[8]:
```
class Dense(Layer):
    """ Standard fully connected layer."""
    def __init__(self, input_units, output_units ):
        self.fc = True # whether it is fully connected layer
        self.ins = input_units
        self.outs = output_units
        self.weights = np.random.normal(loc=0.0,
                                        scale = np.sqrt(1/input_units), #␣
↪Suggested  init
                                        size = (output_units, input_units))
#                        np.random.normal(loc=0.0,
#                                        scale = np.sqrt(2/
↪(input_units+output_units)), # He init
#                                        size = (output_units, input_units))
        self.biases = np.zeros(output_units)



    def forward(self,input):
        # Perform an affine transformation:
        # f(x) = <W*x> + b
        # input shape: [input_units, batch]
        # output shape: [output units, batch]

        if input.ndim == 1:
            input = input.reshape(input.shape[0],1)
```

```
        return np.dot(self.weights, input) + self.biases[:,np.newaxis]

    def backward(self,input,grad_output,eta=0.001, regularization = 0.1):
        # compute d f / d x = d f / d dense * d dense / d x
        # where d dense/ d x = weights transposed
        if input.ndim == 1:
            input = input.reshape(input.shape[0],1)
        grad_input = np.dot(self.weights.T, grad_output)

        # compute gradient w.r.t. weights and biases
        grad_weights = np.dot(grad_output, input.T)/(input.shape[1])
        if regularization:
            grad_weights += 2 * regularization * self.weights

        grad_biases = grad_output.mean(axis=1)

        assert grad_weights.shape == self.weights.shape and grad_biases.shape␣
↪== self.biases.shape

        # Here we perform a stochastic gradient descent step.
        self.weights = self.weights - eta * grad_weights
        self.biases = self.biases - eta * grad_biases

        return grad_input
```

[9]:

```
class Network():
    def __init__(self,layers = []):
        self.layers = layers[:]
        self.startup()

    def add(self,layer):
        self.layers.append(layer)

    def startup(self,n_epochs=40, n_batch=100, eta=0.001, reg=0.01,␣
↪eta_min=None, eta_max=None, n_s=None):
        # Main param
        self.n_batch = n_batch
        self.eta = eta # learning rate
        self.reg = reg # regularization of weights

        ## Cyclic learning rate parameters
        self.eta_min = eta_min
        self.eta_max = eta_max
        self.n_s = n_s
        self.t = 0 # nr of updates

        ## Logging
```

```
        self.train_acc_log = []
        self.val_acc_log = []
        self.train_loss_log = []
        self.val_loss_log = []
        self.train_cost_log = []
        self.val_cost_log = []
        self.eta_log = []
        self.t = 0

    def re_init(self):
        for l in self.layers:
            if l.fc:
                l.__init__(l.ins,l.outs)

    ##############################################
    ######## Cross Entropy loss function ########
    def CrossEntropyLoss(self, softmax_input, P, Y_batch):
        lossgrad = -(Y_batch - P)
        tmp = np.exp(softmax_input)
        return lossgrad

    def SoftMax(self, input):
        tmp = np.exp(input)
        softmax = tmp / np.sum(tmp,0)[np.newaxis,:]
        return softmax

    def loss(self,X_batch, Y_batch):
        activations = self.forward(X_batch)
        softmax_input = activations[-1]
        tmp = np.exp(softmax_input)
        loss_matrix = - softmax_input + np.log(np.sum(tmp,0))[np.newaxis,:]
        return (loss_matrix * Y_batch).sum()/Y_batch.shape[1]
    ##############################################


    ##############################################
    ########          Logging          ########
    def logging(self, X_train, Y_train_hot, X_valid, Y_valid_hot, visualize,␣
↪epoch):
        ll = 0
        for l in self.layers:
            if l.fc:
                ll +=self.reg*np.sum(l.weights[:]**2)
        self.train_acc_log.append(self.accuracy(X_train, Y_train_hot))
        self.val_acc_log.append(self.accuracy(X_valid, Y_valid_hot))
        self.train_loss_log.append(self.loss(X_train,Y_train_hot))
        self.val_loss_log.append(self.loss(X_valid,Y_valid_hot))
```

```
        self.train_cost_log.append(self.train_loss_log[-1] + l1)
        self.val_cost_log.append(self.val_loss_log[-1] + l1)
        if visualize:
            clear_output(wait="True")
        print("Epoch",epoch)
        print("Train accuracy:", self.train_acc_log[-1],"Train loss:","%.4f" % self.train_loss_log[-1] )
        print("Val accuracy:", self.val_acc_log[-1], "Val loss:", "%.4f" % self.val_loss_log[-1],"\n")
        if visualize:
            self.plot_training()

    def plot_training(self):
        fig, axs = plt.subplots(1,3, figsize=(15, 4))
        fig.subplots_adjust(hspace = .5, wspace=0.3)
        axs = axs.ravel()


        axs[0].plot(self.train_cost_log,label='train cost')
        axs[0].plot(self.val_cost_log,label='val cost')
        axs[0].legend(loc='best')
        axs[0].grid()

        axs[1].plot(self.train_loss_log,label='train loss')
        axs[1].plot(self.val_loss_log,label='val loss')
        axs[1].legend(loc='best')
        axs[1].grid()

        axs[2].plot(self.train_acc_log,label='train accuracy')
        axs[2].plot(self.val_acc_log,label='val accuracy')
        axs[2].legend(loc='best')
        axs[2].grid()
        plt.show()
    #############################################

    def forward(self, input):
        tmp = input
        activations = []
        activations.append(tmp)
        for l in self.layers:
            tmp = l.forward(activations[-1])
            activations.append(tmp)
        return activations

    def backward(self, X_batch, Y_batch):
        if Y_batch.ndim == 1:
            Y_batch = Y_batch.reshape(Y_batch.shape[0],1)
```

```
        activations = self.forward(X_batch)
        lossgrad = self.CrossEntropyLoss(activations[-1], self.SoftMax(activations[-1]), Y_batch)

        for i, l in reversed(list(enumerate(self.layers))):
            if l.fc:
                lossgrad = l.backward(activations[i], lossgrad, self.eta, self.reg)
            else:
                lossgrad = l.backward(activations[i],lossgrad)

    def predict(self, input):
        p = self.SoftMax(self.forward(input)[-1])
        return p.argmax(0)

    def accuracy(self, X, Y):
        Y_pred = self.predict(X)
        if Y.ndim == 2:
            Y = Y.argmax(0)
        return np.mean(Y_pred == Y)

    def cyclical_learning_rate(self):
        if self.eta_min and self.eta_max and self.n_s:
            self.eta = self.eta_min + np.copysign(np.mod(self.t,self.n_s),self.n_s-self.t)*(self.eta_max- self.eta_min)/self.n_s + (self.n_s<=self.t)*(self.eta_max-self.eta_min)
            self.eta_log.append(self.eta)

    def minibatch_SGD(self, X_train, Y_train_hot):
        n = X_train.shape[1]
        for j in range(0,n, self.n_batch):
                    self.cyclical_learning_rate() # updates the learning rate
                    X_batch = X_train[:, j:j+self.n_batch];
                    Y_batch = Y_train_hot[:, j:j+self.n_batch];
                    loss = self.backward(X_batch, Y_batch)
                    self.t = np.mod(self.t+1, 2*self.n_s) # increase update nr


    def train(self, X_train, Y_train_hot, X_valid, Y_valid_hot,
              shufle=True, n_epochs=40, n_batch=100, eta=0.001, reg=0.01, visualize=False,
              eta_min=None, eta_max=None, n_s=None):

        # Delete old logging
        self.startup(n_epochs, n_batch, eta, reg, eta_min, eta_max, n_s)

        # Reinit weighs
```

```
        self.re_init()

        for epoch in tqdm(range(n_epochs)):
            if shufle:
                X_train, Y_train_hot = shuffle(X_train,Y_train_hot) # shuffle

            ## Logging
            self.logging(X_train, Y_train_hot, X_valid, Y_valid_hot,␣
↪visualize,epoch)

            ## Minibatch SGD
            self.minibatch_SGD(X_train, Y_train_hot)
```

[7]:
```
def shuffle(X,Y):
    indexes = np.random.permutation(X.shape[1])
    X = np.take(X,indexes,axis=1)
    Y = np.take(Y,indexes,axis=1)
    return X,Y

def create_val_set(X,Y,n_val=5000):
    X, Y = shuffle(X,Y)
    X_val = X[:,0:n_val]
    Y_val = Y[:,0:n_val]
    X = X[:,n_val:]
    Y = Y[:,n_val:]
    return X, Y, X_val, Y_val
```

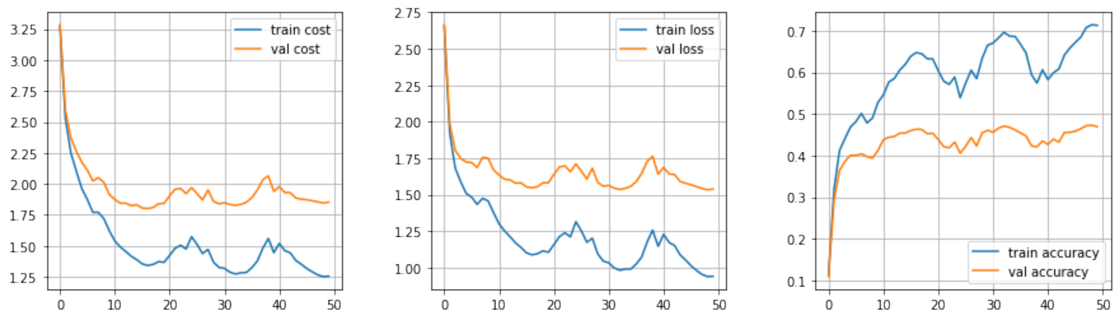
0.4.1 Define the Neural Network

[11]:
```
n_in = X_train.shape[0]
n_out = 10
n_hidden = 50

nn = Network()
nn.add(Dense(n_in, n_hidden))
nn.add(ReLU())
nn.add(Dense(n_hidden, n_out))
```

0.4.2 Training

[26]:
```
nn.train(X_train, Y_train_hot, X_valid, Y_valid_hot,
         shufle = True, n_epochs=50, eta=0.001, reg = 0.01, visualize=True,
         eta_min=1e-5, eta_max=1e-1,n_s=800)
```

```
Epoch 49
Train accuracy: 0.7132 Train loss: 0.9421
Val accuracy: 0.4695 Val loss: 1.5395
```



[27]:

```
print("Train set accuracy:",nn.accuracy(X_train,Y_train_hot))
print("Validation set accuracy:",nn.accuracy(X_valid,Y_valid_hot))
print("Test set accuracy:",nn.accuracy(X_test,Y_test_hot))
```

```
Train set accuracy: 0.6989
Validation set accuracy: 0.4638
Test set accuracy: 0.4619
```

0.4.3 Coarse-to-fine random search to set lambda

The data

[8]:

```
X,Y=load_all_and_preproc()
Y = one_hot(10,Y)
X, Y, X_val, Y_val = create_val_set(X,Y)

print(X.shape,Y.shape, X_val.shape, Y_val.shape)
```

```
(3072, 45000) (10, 45000) (3072, 5000) (10, 5000)
```

The Neural Network

[10]:

```
from NN import *
n_in, n = X.shape
n_out = 10
n_hidden = 50

nn = Network()
nn.add(Dense(n_in, n_hidden))
```

```
nn.add(ReLU())
nn.add(Dense(n_hidden, n_out))
```

[10]:
```
[<NN.Dense at 0x7f27980de1d0>,
 <NN.ReLU at 0x7f27a27fccf8>,
 <NN.Dense at 0x7f27605e2a90>]
```

Searching

[27]:
```
n_s = 2*np.floor(n / 100)
l_min = -5
l_max = -3

data = []
for i in tqdm(range(20)):
    l = l_min + (l_max - l_min)*np.random.rand();
    reg = 10**l;
    nn.train(X, Y, X_val, Y_val,
             shufle=True, n_epochs=4, reg=reg, visualize=False,
             eta_min=1e-5, eta_max=1e-1,n_s=n_s)

    value = (reg,nn.accuracy(X_val,Y_val),nn.accuracy(X,Y))
    data.append(value)

column_name = ['lamba','val_acc','train_acc']
df = pd.DataFrame(data,columns=column_name)
df.to_csv("file"+str(l_min)+str(l_max) +".csv")
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))


Epoch 0
Train accuracy: 0.12502222222222223 Train loss: 2.5188
Val accuracy: 0.1276 Val loss: 2.5132

Epoch 1
Train accuracy: 0.4140888888888889 Train loss: 1.6990
Val accuracy: 0.3926 Val loss: 1.7632

Epoch 2
Train accuracy: 0.4574888888888889 Train loss: 1.5388
Val accuracy: 0.4232 Val loss: 1.6331

Epoch 3
Train accuracy: 0.5113777777777778 Train loss: 1.3836
Val accuracy: 0.4706 Val loss: 1.5146

Epoch 0
```

```
Train accuracy: 0.12653333333333333 Train loss: 2.4641
Val accuracy: 0.1256 Val loss: 2.4664

Epoch 1
Train accuracy: 0.38982222222222224 Train loss: 1.7765
Val accuracy: 0.3736 Val loss: 1.8354

Epoch 2
Train accuracy: 0.4434222222222222 Train loss: 1.5596
Val accuracy: 0.4092 Val loss: 1.6537

Epoch 3
Train accuracy: 0.5199555555555555 Train loss: 1.3651
Val accuracy: 0.4772 Val loss: 1.5061

Epoch 0
Train accuracy: 0.10848888888888888 Train loss: 2.4976
Val accuracy: 0.1104 Val loss: 2.5040

Epoch 1
Train accuracy: 0.42846666666666666 Train loss: 1.6314
Val accuracy: 0.4032 Val loss: 1.6978

Epoch 2
Train accuracy: 0.45411111111111113 Train loss: 1.5375
Val accuracy: 0.425 Val loss: 1.6319

Epoch 3
Train accuracy: 0.5078666666666667 Train loss: 1.3917
Val accuracy: 0.4518 Val loss: 1.5415

Epoch 0
Train accuracy: 0.1019111111111111 Train loss: 2.5473
Val accuracy: 0.097 Val loss: 2.5595

Epoch 1
Train accuracy: 0.4236888888888889 Train loss: 1.6334
Val accuracy: 0.405 Val loss: 1.7078

Epoch 2
Train accuracy: 0.451 Train loss: 1.5993
Val accuracy: 0.4266 Val loss: 1.7028

Epoch 3
Train accuracy: 0.5054888888888889 Train loss: 1.4103
Val accuracy: 0.4574 Val loss: 1.5659

Epoch 0
```

```
Train accuracy: 0.09444444444444444 Train loss: 2.5210
Val accuracy: 0.0958 Val loss: 2.5419

Epoch 1
Train accuracy: 0.4166222222222222 Train loss: 1.6555
Val accuracy: 0.3992 Val loss: 1.7213

Epoch 2
Train accuracy: 0.4679111111111111 Train loss: 1.5210
Val accuracy: 0.4288 Val loss: 1.6208

Epoch 3
Train accuracy: 0.5207555555555555 Train loss: 1.3788
Val accuracy: 0.4666 Val loss: 1.5199

Epoch 0
Train accuracy: 0.07748888888888888 Train loss: 2.4885
Val accuracy: 0.085 Val loss: 2.4798

Epoch 1
Train accuracy: 0.43504444444444446 Train loss: 1.6114
Val accuracy: 0.4104 Val loss: 1.6747

Epoch 2
Train accuracy: 0.4646666666666667 Train loss: 1.5145
Val accuracy: 0.4286 Val loss: 1.6177

Epoch 3
Train accuracy: 0.5104 Train loss: 1.3916
Val accuracy: 0.4544 Val loss: 1.5392

Epoch 0
Train accuracy: 0.08217777777777778 Train loss: 2.5532
Val accuracy: 0.0772 Val loss: 2.5614

Epoch 1
Train accuracy: 0.4162888888888889 Train loss: 1.6664
Val accuracy: 0.3928 Val loss: 1.7082

Epoch 2
Train accuracy: 0.45782222222222224 Train loss: 1.5407
Val accuracy: 0.433 Val loss: 1.6290

Epoch 3
Train accuracy: 0.5025555555555555 Train loss: 1.4050
Val accuracy: 0.4566 Val loss: 1.5545

Epoch 0
```

```
Train accuracy: 0.10355555555555555 Train loss: 2.6017
Val accuracy: 0.106 Val loss: 2.5948

Epoch 1
Train accuracy: 0.4159777777777778 Train loss: 1.6672
Val accuracy: 0.3976 Val loss: 1.7238

Epoch 2
Train accuracy: 0.4502888888888889 Train loss: 1.5434
Val accuracy: 0.408 Val loss: 1.6575

Epoch 3
Train accuracy: 0.5145555555555555 Train loss: 1.3802
Val accuracy: 0.4532 Val loss: 1.5276

Epoch 0
Train accuracy: 0.09 Train loss: 2.5000
Val accuracy: 0.0962 Val loss: 2.4852

Epoch 1
Train accuracy: 0.4394 Train loss: 1.6043
Val accuracy: 0.4138 Val loss: 1.6828

Epoch 2
Train accuracy: 0.44606666666666667 Train loss: 1.5456
Val accuracy: 0.4246 Val loss: 1.6453

Epoch 3
Train accuracy: 0.5122666666666666 Train loss: 1.3838
Val accuracy: 0.4626 Val loss: 1.5228

Epoch 0
Train accuracy: 0.09264444444444445 Train loss: 2.5194
Val accuracy: 0.0888 Val loss: 2.5346

Epoch 1
Train accuracy: 0.427 Train loss: 1.6499
Val accuracy: 0.3952 Val loss: 1.7312

Epoch 2
Train accuracy: 0.4644888888888889 Train loss: 1.5214
Val accuracy: 0.4284 Val loss: 1.6338

Epoch 3
Train accuracy: 0.5129777777777778 Train loss: 1.3836
Val accuracy: 0.4582 Val loss: 1.5401

Epoch 0
```

```
Train accuracy: 0.12751111111111111 Train loss: 2.4551
Val accuracy: 0.1262 Val loss: 2.4443

Epoch 1
Train accuracy: 0.4328 Train loss: 1.6236
Val accuracy: 0.4022 Val loss: 1.6971

Epoch 2
Train accuracy: 0.46177777777777778 Train loss: 1.5165
Val accuracy: 0.4284 Val loss: 1.6140

Epoch 3
Train accuracy: 0.50208888888888889 Train loss: 1.4058
Val accuracy: 0.46 Val loss: 1.5540

Epoch 0
Train accuracy: 0.09084444444444445 Train loss: 2.4581
Val accuracy: 0.095 Val loss: 2.4587

Epoch 1
Train accuracy: 0.42388888888888887 Train loss: 1.6469
Val accuracy: 0.3966 Val loss: 1.7105

Epoch 2
Train accuracy: 0.45468888888888886 Train loss: 1.5487
Val accuracy: 0.4196 Val loss: 1.6353

Epoch 3
Train accuracy: 0.5158 Train loss: 1.3778
Val accuracy: 0.4682 Val loss: 1.5114

Epoch 0
Train accuracy: 0.10344444444444445 Train loss: 2.5194
Val accuracy: 0.0996 Val loss: 2.5344

Epoch 1
Train accuracy: 0.427 Train loss: 1.6296
Val accuracy: 0.4054 Val loss: 1.6841

Epoch 2
Train accuracy: 0.45995555555555556 Train loss: 1.5285
Val accuracy: 0.4314 Val loss: 1.6052

Epoch 3
Train accuracy: 0.49722222222222223 Train loss: 1.4206
Val accuracy: 0.447 Val loss: 1.5675

Epoch 0
```

```
Train accuracy: 0.09135555555555555 Train loss: 2.5937
Val accuracy: 0.0938 Val loss: 2.6185

Epoch 1
Train accuracy: 0.4208 Train loss: 1.6618
Val accuracy: 0.3984 Val loss: 1.7232

Epoch 2
Train accuracy: 0.45693333333333336 Train loss: 1.5359
Val accuracy: 0.4184 Val loss: 1.6460

Epoch 3
Train accuracy: 0.5195777777777778 Train loss: 1.3690
Val accuracy: 0.4662 Val loss: 1.5106

Epoch 0
Train accuracy: 0.09044444444444444 Train loss: 2.5139
Val accuracy: 0.0856 Val loss: 2.5283

Epoch 1
Train accuracy: 0.42493333333333333 Train loss: 1.6342
Val accuracy: 0.4116 Val loss: 1.6896

Epoch 2
Train accuracy: 0.46024444444444446 Train loss: 1.5244
Val accuracy: 0.4276 Val loss: 1.6207

Epoch 3
Train accuracy: 0.5247333333333334 Train loss: 1.3581
Val accuracy: 0.4682 Val loss: 1.4963

Epoch 0
Train accuracy: 0.12248888888888888 Train loss: 2.5588
Val accuracy: 0.1228 Val loss: 2.5427

Epoch 1
Train accuracy: 0.41604444444444444 Train loss: 1.6995
Val accuracy: 0.3916 Val loss: 1.7825

Epoch 2
Train accuracy: 0.45355555555555555 Train loss: 1.5472
Val accuracy: 0.4172 Val loss: 1.6586

Epoch 3
Train accuracy: 0.5187333333333334 Train loss: 1.3640
Val accuracy: 0.4708 Val loss: 1.5016

Epoch 0
```

```
Train accuracy: 0.0872 Train loss: 2.5805
Val accuracy: 0.0792 Val loss: 2.5991

Epoch 1
Train accuracy: 0.43164444444444444 Train loss: 1.6241
Val accuracy: 0.4078 Val loss: 1.7035

Epoch 2
Train accuracy: 0.44108888888888886 Train loss: 1.6164
Val accuracy: 0.4128 Val loss: 1.7413

Epoch 3
Train accuracy: 0.5168222222222222 Train loss: 1.3819
Val accuracy: 0.4608 Val loss: 1.5153

Epoch 0
Train accuracy: 0.09091111111111111 Train loss: 2.4833
Val accuracy: 0.0946 Val loss: 2.4770

Epoch 1
Train accuracy: 0.4253777777777778 Train loss: 1.6458
Val accuracy: 0.4078 Val loss: 1.7012

Epoch 2
Train accuracy: 0.46542222222222224 Train loss: 1.5099
Val accuracy: 0.4356 Val loss: 1.5950

Epoch 3
Train accuracy: 0.5131111111111111 Train loss: 1.3904
Val accuracy: 0.4708 Val loss: 1.5107

Epoch 0
Train accuracy: 0.08051111111111112 Train loss: 2.5253
Val accuracy: 0.0756 Val loss: 2.5240

Epoch 1
Train accuracy: 0.42835555555555554 Train loss: 1.6349
Val accuracy: 0.4006 Val loss: 1.7055

Epoch 2
Train accuracy: 0.4547111111111111 Train loss: 1.5367
Val accuracy: 0.4244 Val loss: 1.6469

Epoch 3
Train accuracy: 0.5144888888888889 Train loss: 1.3785
Val accuracy: 0.4552 Val loss: 1.5436

Epoch 0
```

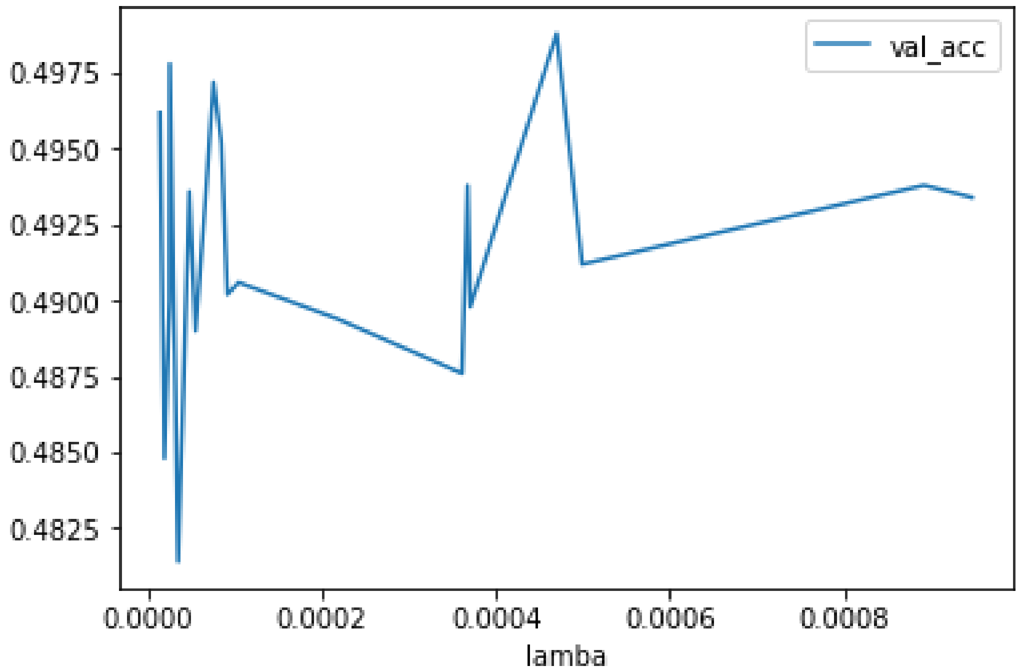
```
Train accuracy: 0.08964444444444444 Train loss: 2.4963
Val accuracy: 0.093 Val loss: 2.4939

Epoch 1
Train accuracy: 0.4333111111111111 Train loss: 1.6190
Val accuracy: 0.4092 Val loss: 1.6751

Epoch 2
Train accuracy: 0.4275333333333333 Train loss: 1.6138
Val accuracy: 0.3942 Val loss: 1.7058

Epoch 3
Train accuracy: 0.5156 Train loss: 1.3779
Val accuracy: 0.4684 Val loss: 1.5196
```

Visualization

[15]:
```
df =pd.read_csv('file-5-3.csv')
df2=df.sort_values('lamba')
print(df2.sort_values('val_acc'))

df2.plot(x='lamba', y='val_acc')
plt.plot()
```

```
    Unnamed: 0     lamba  val_acc  train_acc
7            7  0.000035   0.4814   0.555467
9            9  0.000019   0.4848   0.553244
13          13  0.000361   0.4876   0.557622
18          18  0.000055   0.4890   0.556022
8            8  0.000218   0.4894   0.556089
11          11  0.000371   0.4898   0.553400
2            2  0.000025   0.4900   0.558067
1            1  0.000091   0.4902   0.552978
6            6  0.000104   0.4906   0.553978
14          14  0.000499   0.4912   0.552556
0            0  0.000066   0.4932   0.555044
3            3  0.000948   0.4934   0.556644
12          12  0.000048   0.4936   0.553689
16          16  0.000893   0.4938   0.552267
5            5  0.000368   0.4938   0.555489
15          15  0.000085   0.4952   0.556644
10          10  0.000014   0.4962   0.556644
19          19  0.000075   0.4972   0.558511
4            4  0.000025   0.4978   0.556067
17          17  0.000470   0.4988   0.554933
```

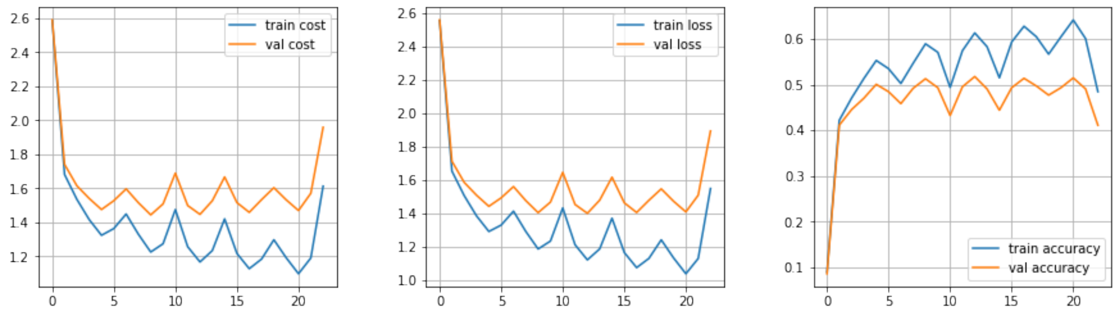
[15]: []



0.4.4 Proper Training

[ ]:
```
X,Y=load_all_and_preproc()
Y = one_hot(10,Y)
X, Y, X_val, Y_val = create_val_set(X,Y)

print(X.shape,Y.shape, X_val.shape, Y_val.shape)

n_in, n = X.shape
n_out = 10
n_hidden = 50

nn = Network()
nn.add(Dense(n_in, n_hidden))
nn.add(ReLU())
nn.add(Dense(n_hidden, n_out))


nn.train(X, Y, X_val, Y_val,
         shufle = True, n_epochs=32, eta=0.001, reg = 0.000470, visualize=True,
         eta_min=1e-5, eta_max=1e-1,n_s=900)
```

```
Epoch 22
Train accuracy: 0.4844 Train loss: 1.5478
Val accuracy: 0.411 Val loss: 1.8934
```



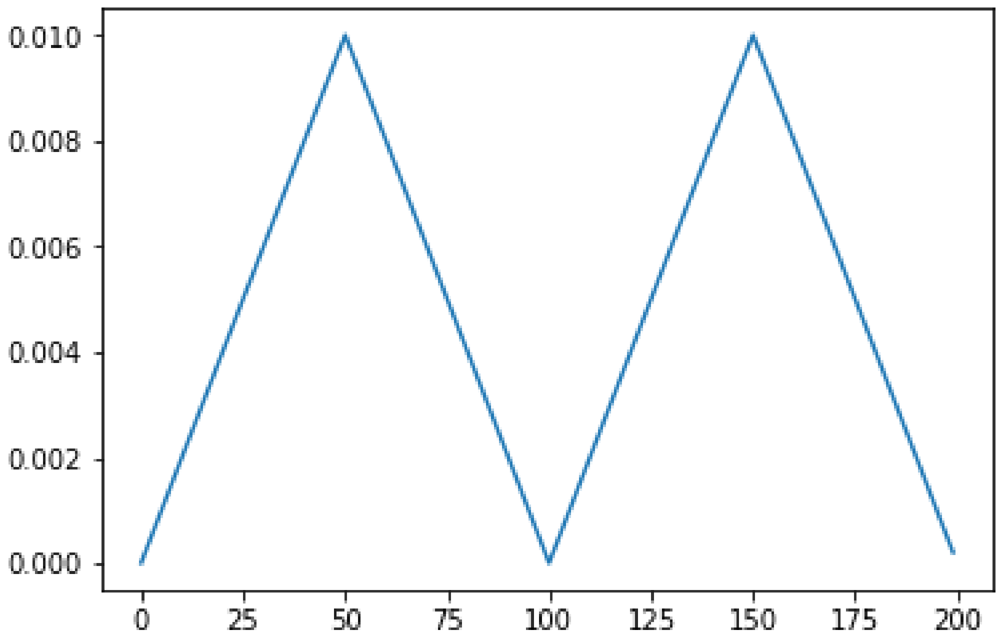
[ ]:
```python
print("Train set accuracy:",nn.accuracy(X,Y))
print("Validation set accuracy:",nn.accuracy(X_val,Y_val))
print("Test set accuracy:",nn.accuracy(X_test,Y_test_hot))
```

[ ]:
```python

```

Visualize the cyclical learning rate

[369]:
```python
eta_min=1e-5
eta_max=1e-2
lamda=0.01
n_s=50
def funcc(t):
    t = np.mod(t,2*n_s)
    eta = eta_min + np.copysign(np.mod(t,n_s),n_s-t)*(eta_max- eta_min)/n_s + (n_s<=t)*(eta_max-eta_min)
    return eta

aa = np.arange(200)
xx = []
for t in aa:
    xx.append(funcc(t))
#xx = funcc(aa)
plt.plot(aa,xx)
print(xx[0])
```

```
1e-05
```


0.010
0.008
0.006
0.004
0.002
0.000
0
25
50
75
100
125
150
175
200