```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
#!pip install matplotlib
import matplotlib.pyplot as plt
import time
print(torch.__version__)
#!pip install torch===1.7.0 torchvision===0.8.1 torchaudio===0.7.0 -f https://downloac
import cv2
print(torch.__version__)

import torchvision.datasets as dset
import torchvision.transforms as T
from keras.datasets import cifar10
import numpy as np, numpy.linalg

USE_GPU = True
dtype = torch.float64 # We will be using float throughout this tutorial.

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss.
print_every = 100
print('using device:', device)
```

```
1.8.1+cu101
1.8.1+cu101
using device: cpu
```

```
#class for forward pass and loss

class custom_multiclass_torch(torch.nn.Module):
    def __init__(self, N, P, C, K, f,a,b,c,beta, x_patches_train, x_patches_val, x_pat
        super(custom_multiclass_torch, self).__init__()


        #initialize several variables
        # parameters, N = number of images, K = number of patches, P = pooling size, (

        self.N = N
        self.P = P
        self.C = C
        self.K = K
        self.f = f
```

```
        self.a = a
        self.b = b
        self.c = c
        self.beta = beta
        self.x_patches_train = x_patches_train
        self.x_patches_val = x_patches_val
        self.x_patches_test = x_patches_test


        #initialize tensor array variables as dictionaries

        self.Z_1_arr_train = nn.ParameterDict({  })
        self.Z_1_arr_prime_train = nn.ParameterDict({})
        self.Z_2_arr_train = nn.ParameterDict({})
        self.Z_2_arr_prime_train = nn.ParameterDict({})
        self.Z_4_arr_train = nn.ParameterDict({})
        self.Z_4_arr_prime_train = nn.ParameterDict({})
        self.Z_arr_train = {}
        self.Z_arr_prime_train = {}

        # fill in dictionaries with Z1 Z2 Z4 tensor parameters
        for i in range(1,self.K//self.P+1):
            for j in range(1,self.C+1):


                self.Z_1_arr_train[str(i)+','+str(j)] =  torch.nn.Parameter(data = tor
                self.Z_1_arr_prime_train[str(i)+','+str(j)] =  torch.nn.Parameter(data
                self.Z_2_arr_train[str(i)+','+str(j)] =  torch.nn.Parameter(data = tor
                self.Z_2_arr_prime_train[str(i)+','+str(j)] =  torch.nn.Parameter(data
                self.Z_4_arr_train[str(i)+','+str(j)] =  torch.nn.Parameter(data = tor
                self.Z_4_arr_prime_train[str(i)+','+str(j)] =  torch.nn.Parameter(data

                nn.init.xavier_uniform_(self.Z_1_arr_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_2_arr_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_4_arr_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_1_arr_prime_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_2_arr_prime_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_4_arr_prime_train[str(i)+','+str(j)])




                self.Z_arr_train[str(i)+','+str(j)] = torch.vstack((torch.hstack((self
                self.Z_arr_prime_train[str(i)+','+str(j)] = torch.vstack((torch.hstack

    def forward(self, i, setting = "train"):

        Z_1_new = {}
        Z_2_new = {}
```

```
        Z_4_new = {}
        Z_1_prime_new = {}
        Z_2_prime_new = {}
        Z_4_prime_new = {}
        Z_new = {}
        Z_new_prime = {}

        if setting == "train":
            xdata = self.x_patches_train
        elif setting == "val":
            xdata = self.x_patches_val
        else:
            xdata = self.x_patches_test

        #transform Z matrices into positve-semidefinite matrices

        # #convertTo = "positive semidefinite"
        convertTo = "other"
        for key in self.Z_arr_train:
            if convertTo == "positive semidefinite":
                Z_new[key] = torch.matmul(self.Z_arr_train[key], self.Z_arr_train[key]
            elif convertTo == "symmetric:":
                Z_new[key] = 0.5 * (self.Z_arr_train[key]+self.Z_arr_train[key].T)
            else:
                Z_new[key] = self.Z_arr_train[key]
            Z_1_new[key] = Z_new[key][:3*f**2,:3*f**2]
            Z_2_new[key] = Z_new[key][3*f**2,:3*f**2]
            Z_4_new[key] = Z_new[key][3*f**2,3*f**2]

        for key in self.Z_arr_prime_train:
            if convertTo == "positive semidefinite":
                Z_new_prime[key] = torch.matmul(self.Z_arr_prime_train[key], self.Z_ar
            elif convertTo == "symmetric":
                Z_new_prime[key] = 0.5 * (self.Z_arr_prime_train[key] + self.Z_arr_pri
            else:
                Z_new_prime[key] = self.Z_arr_prime_train[key]
            Z_1_prime_new[key] = Z_new_prime[key][:3*f**2,:3*f**2]
            Z_2_prime_new[key] = Z_new_prime[key][3*f**2,:3*f**2]
            Z_4_prime_new[key] = Z_new_prime[key][3*f**2,3*f**2]


        #print(self.Z_1_arr_train[1,2])
        #print(self.Z_4_arr_train[1,2])


        ypred = torch.zeros((C))


        # performing calculations for ypred scores
        for t in range(1,self.C+1):

            constant_part = 0
```

```
        for k in range(1,self.K//self.P+1):
            Z4diff = self.Z_4_arr_train[str(k)+","+str(t)] - self.Z_4_arr_prime_tr
            constant_part += Z4diff
        constant_part *= self.c
        #print(constant_part)

        linear_part = 0
        for k in range(1,self.K//self.P+1):
            for l in range(1,self.P+1):
                Z2diff = self.Z_2_arr_train[str(k)+","+str(t)] - self.Z_2_arr_prin
                x_data_reshaped = xdata[i][(k-1)*P+ l-1].view(-1, 1)
                linear_part += torch.matmul(x_data_reshaped.T, Z2diff)

                #linear_part += float(torch.matmul(x_data_reshaped.T, Z2diff)[0])
        linear_part *= self.b/self.P

        quadratic_part = 0
        for k in range(1,self.K//self.P+1):
            for l in range(1,self.P+1):
                Z1diff = self.Z_1_arr_train[str(k)+","+str(t)] - self.Z_1_arr_prin
                x_data_reshaped = xdata[i][(k-1)*P+ l-1].view(-1, 1)
                newpart = torch.matmul(x_data_reshaped.T, Z1diff)
                newpart = torch.matmul(newpart, x_data_reshaped)
                quadratic_part += newpart

                #quadratic_part += float(newpart[0,0])
        quadratic_part *= self.a/self.P

        #print(quadratic_part, linear_part, constant_part)

        ypred[t-1] = quadratic_part + linear_part + constant_part

        #randnum = np.random.randint(10)
        #ypred *= randnum
    return ypred



def customloss(self, Yhat, y):
    #convex L2 loss function
    objective1 = 0.5 * torch.norm(Yhat - y)**2


    # sum of Z4 scalars added to loss
    objective2 = 0

    for i in range(1,K//P+1):
        for j in range(1,C+1):
            objective2 += self.Z_4_arr_train[str(i)+","+str(j)] + self.Z_4_arr_pri

    objective = objective1+ self.beta * objective2
```

```
        return objective
```

```
#load cifar data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# shuffler = np.random.permutation(len(y_train))
# X_train = X_train[shuffler]
# y_train = y_train[shuffler]
# shuffler2 = np.random.permutation(len(y_test))
# X_test = X_test[shuffler2]
# y_test = y_test[shuffler2]

#subset data
X_train = X_train[:2000,:,:,:]
y_train = y_train[:2000]
X_test = X_test[:400,:,:,:]
y_test = y_test[:400]

#set up val/test split
X_val = X_test[:X_test.shape[0]//2,:,:,:]
X_test = X_test[X_test.shape[0]//2:,:,:,:]

y_val = y_test[:y_test.shape[0]//2]
y_test = y_test[:y_test.shape[0]//2]


#parameters: f is filter size, P is pooling size, C is class count, a,b,c are the poly

f = 4



train_images = X_train.astype(np.float64)
train_labels = y_train.astype(np.float64)
test_images = X_test.astype(np.float64)
test_labels = y_test.astype(np.float64)
val_images = X_val.astype(np.float64)
val_labels = y_val.astype(np.float64)
```

```
# #meaning out the images
# mean_image = np.mean(train_images, axis = 0)
# train_images -= mean_image
# test_images -= mean_image
# val_images -= mean_image

# # RGB 0 to 255
# #-127.5 to +127.5
# #-1 to 1

# # scale to [-1, 1]

# train_images /= 127.5
# test_images /= 127.5
# val_images /= 127.5


train_images /= 255
train_images *= 2
train_images -= 1

test_images /= 255
test_images *= 2
test_images -= 1

val_images /= 255
val_images *= 2
val_images -= 1

train_images_v2 = np.swapaxes(train_images.reshape(train_images.shape[0], 3, 32, 32),
test_images_v2 = np.swapaxes(test_images.reshape(test_images.shape[0], 3, 32, 32), 2,
val_images_v2 = np.swapaxes(val_images.reshape(val_images.shape[0],3, 32, 32), 2, 3)

train_images_v3 = torch.tensor(train_images_v2, device = device, dtype = dtype)
test_images_v3 = torch.tensor(test_images_v2, device = device, dtype = dtype)
val_images_v3 = torch.tensor(val_images_v2, device = device, dtype = dtype)

#set s_val = torch.nn.functional.unfold(val_images_v3, kernel_size=(f,f), stride=f, pa
#setting up patches
patches_train = torch.nn.functional.unfold(train_images_v3, kernel_size=(f,f), stride=
patches_test = torch.nn.functional.unfold(test_images_v3, kernel_size=(f, f), stride=f
patches_val = torch.nn.functional.unfold(val_images_v3, kernel_size=(f,f), stride=f, p



patches_train = patches_train.permute(0,2,1)
patches_test = patches_test.permute(0,2,1)
patches_val = patches_val.permute(0,2,1)

N = X_train.shape[0]
```

```
K = patches_train.shape[1]

Yhat_train =  None
Yhat_test = None

P = K
C = 10
a=0.09
b=0.5
c=0.47
beta = 1e-7

print("K: ", K)
print("N: ", N)
print("patches_train size: ", patches_train.size())
print("patches_val size: ", patches_val.size())
print("patches_test size: ", patches_test.size())
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 4s 0us/step
170508288/170498071 [==============================] - 4s 0us/step
K:  64
N:  2000
patches_train size:  torch.Size([2000, 64, 48])
patches_val size:  torch.Size([200, 64, 48])
patches_test size:  torch.Size([200, 64, 48])
```

```
def train(model, loss_fn, optimizer, epochs=1):
    model = model.to(device=device) # move the model parameters to CPU/GPU

    loss_train = []
    accuracies_train = []
    accuracies_val = []
    start = time.time()
    times = []
    times.append(time.time)
    zzz = 0
    #accuracies_train.append(check_accuracy(X_train, y_train, model, segment = "train'

    for e in range(epochs):
        #idx = torch.randperm(train_images.shape[0])
        #sample_images = train_images[idx].view(train_images.size())
        #sample_images = sample_images[]
        for t, (x, y) in enumerate(zip(train_images, train_labels)):
            model.train() # put model to training mode

            scores = model(t, setting = "train")
```

```
            y_hot = torch.zeros(scores.size(), dtype = dtype)

            y_hot[y] = 1


            #print("Scores before: ", scores)
            loss = loss_fn(scores, y_hot)
            loss_train.append(loss)

            # Zero out all of the gradients for the variables which the optimizer
            # will update.
            optimizer.zero_grad()

            # backwards pass
            loss.backward()

            # update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()
            #print("Scores after: ", scores)
            #if t % 100 == 0:
            if t==0:
              times.append(time.time)
              print('Iteration %d, loss = %.4f' % (t, loss.item()))
              accuracies_train.append(check_accuracy(X_train, y_train, model, segment
              accuracies_val.append(check_accuracy(X_val, y_val, model, segment = "val


       # times.append(time.time)
        #print('Iteration %d, loss = %.4f' % (t, loss.item()))
        #accuracies_train.append(check_accuracy(X_train, y_train, model, segment = "tr

    for i in range(len(times)):
        times[i] -= start


    return loss_train, accuracies_train, accuracies_val, times








def check_accuracy(X, Y, model, segment = "train"):
    if segment=='train':
        print('Checking accuracy on train set')
    elif segment == "val":
        print('Checking accuracy on val set')
    else:
        print('Checking accuracy on test set')
```

```
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for t, (x, y) in enumerate(zip(X, Y)):
            scores = model(t, setting = segment)
            max_score_idx = torch.argmax(scores)
            y = torch.tensor(y)
            #if t % 1000 == 0:
              #print(t, scores, y)
              #for name, param in model.named_parameters():
               #   if param.requires_grad:
                #      if name == "Z_2_arr_train.1,1" or name == "Z_2_arr_prime_train.
                 #       print(name, param.data.T)
                        #break


            addvalue = 1 if (max_score_idx == y) else 0
            num_correct += addvalue
            num_samples += 1

        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)'% (num_correct, num_samples, 100 * acc))
        return acc


model = custom_multiclass_torch(N, P, C, K, f,a,b,c,beta, patches_train, patches_val,
```

```
count = 0
for param in model.parameters():
    #print(param.shape)
    count += 1
print(count)
```

```
60
```

```
loss_fn = model.customloss

optimizer = optim.Adam(model.parameters(), lr=1e-3, amsgrad=True)

#optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

loss_train, accuracies_train, accuracies_val, times = train(model, loss_fn, optimizer,
```

```
Iteration 0, loss = 2.1351
Checking accuracy on train set
Got 220 / 2000 correct (11.00)
Checking accuracy on val set
```

```
Got 22 / 200 correct (11.00)
Iteration 0, loss = 0.5060
Checking accuracy on train set
Got 486 / 2000 correct (24.30)
Checking accuracy on val set
Got 44 / 200 correct (22.00)
```

```
plt.figure(figsize=(10,5))
plt.title('Training Accuracy')
#print(accuracies_train)
plt.plot(accuracies_train)
plt.xlabel('Epochs')
plt.ylabel('Val Accuracy')
#plt.legend()
plt.show()


plt.figure(figsize=(10,5))
plt.title('Validation Accuracy')
#print(accuracies_val)
plt.plot(accuracies_val)
plt.xlabel('Epochs')
plt.ylabel('Val Accuracy')
#plt.legend()
plt.show()



plt.figure(figsize=(10,5))
plt.title('Training Loss')
#print(loss_train)
plt.plot(loss_train)
plt.xlabel('Epochs')
plt.ylabel('Train Loss')
#plt.legend()
plt.show()


plt.figure(figsize=(10,5))
plt.title('Training Loss')
#print(loss_train)
plt.plot(loss_train)
plt.xlabel('Epochs')
plt.ylabel('Train Loss')
#plt.legend()
plt.show()

plt.figure(figsize=(10,5))
plt.title('Training Time')
#print(loss_train)
```

```
#print(loss_train)
plt.plot(times)
plt.xlabel('Epochs')
plt.ylabel('Time (seconds)')
#plt.legend()
plt.show()


check_accuracy(X_test, y_test, model, segment = "test")
```