**Пояснительная записка к фреймворку PyTorch**

Основное назначение данного документа – предоставление общей информации о фремйворки PyTorch, разбор основных понятий. Приведены ссылки на инструкции по установке PyTorch. Рассмотрены два примера использования данного фреймворка для тренировки нейронной сети.

PyTorch - основанная на Torch библиотека машинного обучения для Python. Она похожа на NumPy, но с мощной поддержкой графического процессора. Библиотека была разработана AI Research Group из Facebook в 2016 году.

**Преимущества и недостатки PyTorch:**

(+) Процесс моделирования прост и прозрачен благодаря архитектурному стилю фреймворка;

(+) Режим определения по умолчанию больше похож на традиционное программирование, и вы можете использовать общие средства отладки, такие как отладчик pdb, ipdb или PyCharm;

(+) Имеет декларативный параллелизм данных;

(+) Легко писать свои собственные типы слоев и работать на GPU

(+) Имеет много предварительно обученных моделей и модульных частей, которые легко объединяются;

(-) Не хватает модельной порции

(-) Отсутствуют интерфейсы для мониторинга и визуализации, такие как Tensorboard, хотя вы можете подключиться к Tensorboard извне.

**Основы PyTorch**

**Установка**

Ссылки на tutorials по установке PyTorch

<https://github.com/deepmipt/dlschl/wiki/%D0%98%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D0%B8%D1%8F-%D0%BF%D0%BE-%D1%83%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B5-PyTorch>

<https://pytorch.org/get-started/locally/#anaconda-1%23anaconda-1>

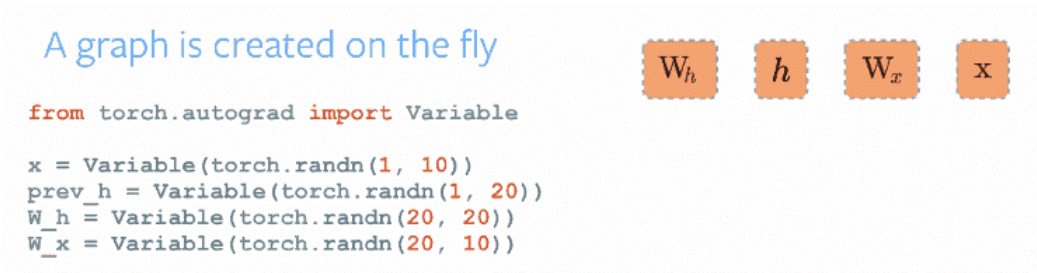
**Вычислительные графы**

Первое, что необходимо понять о любой библиотеке глубокого обучения — идея вычислительных графов. Вычислительный граф — набор вычислений, которые называются узлами (nodes), и которые соединены в прямом порядке вычислений. Другими словами, выбранный узел зависит от узлов на входе, который в свою очередь производит вычисления для других узлов. Преимущества использования вычислительного графа в том, что каждый узел является независимым функционирующим куском кода, если получит все необходимые входные данные. Это позволяет оптимизировать производительность при выполнении расчетов, используя многоканальную обработку, параллельные вычисления.

**Dynamic Computational Graph**

PyTorch предлагает динамический вычислительный граф (DAG). Вычислительные графы - это способ выражения математических выражений в терминах теории графов. Узел выполнит математическую операцию, а ребро - это Тензор, который будет подан в узлы и будет содержать выходные данные узла в Тензор.

DAG - это граф, который имеет произвольную форму и способен выполнять операции между различными входными графами. На каждой итерации создается новый граф. Таким образом, можно иметь такую же структуру графа или создать новый граф с другой операцией. Поэтому его и называют динамическим.



**Тензоры**

Тензоры — подобные матрице структуры данных, которые являются неотъемлемыми компонентами в библиотеках глубокого обучения и используются для эффективных вычислений. Графические процессоры (GPU) эффективны при вычислении операций между тензорами, что стимулировало волну возможностей в глубоком обучении. Пример создания тензора:

import torch

x = torch.Tensor(2, 3)

Этот код создает тензор размера (2,3), заполненный нулями. В данном примере первое число — количество рядов, второе — количество столбцов.

**Автоматическое дифференцирование в PyTorch**

В библиотеках глубокого обучения есть механизмы вычисления градиента ошибки и обратного распространения ошибки через вычислительный граф. Этот механизм, называемый автоградиентом в PyTorch, легко доступен и интуитивно понятен. Переменный класс — главный компонент автоградиентной системы в PyTorch. Переменный класс обертывает тензор и позволяет автоматически вычислять градиент на тензоре при вызове функции backward(). Объект содержит данные из тензора, градиент тензора (единожды посчитанный по отношению к некоторому другому значению, потеря) и содержит также ссылку на любую функцию, созданную переменной (если это функция, созданная пользователем, ссылка будет пустой).

Создадим переменную из простого тензора:

x = Variable(torch.ones(2, 2) \* 2, requires\_grad=True)

z = 2 \* (x \* x) + 5 \* x

z.backward(torch.ones(2, 2))

print(x.grad)

Мы ознакомились с базовыми понятиями.

C примерами построения сеток на Pytorch можно ознакомиться по следующим ссылкам:

<https://neurohive.io/ru/tutorial/glubokoe-obuchenie-s-pytorch/>

<https://www.guru99.com/pytorch-tutorial.html>

Еще несколько полезных ссылок:

<https://github.com/jwyang/faster-rcnn.pytorch> - A Faster Pytorch Implementation of Faster R-CNN

<https://github.com/longcw/faster_rcnn_pytorch> - Faster RCNN with PyTorch

<https://github.com/chenyuntc/simple-faster-rcnn-pytorch> - A Simple and Fast Implementation of Faster R-CNN

<https://github.com/ruotianluo/pytorch-faster-rcnn> - A pytorch implementation of faster RCNN detection framework

Так как в дальнейшей работе мы будем использовать PyTorch для тренировки сетки, то детальнее разберем как обучать и тренировать сетки на PyTorch.

Весь процесс делится на следующие этапы:

1. Загрузка данных

2. Определение CNN

3. Тренировка модели

4. Оценка производительности обученной модели на наборе данных

**Пример 1**

Далее рассмотрен пример тренировки сетки модели ResNet 50, сетка включена в подпакет PyTorch уже обученных моделей.

**Организация своего учебный набора данных:** PyTorch ожидает, что данные будут организованы по папкам, для каждого класса определена своя папка. Есть несколько вариантов - часто предполагается, что есть «верхние» папки – для обучения, тестирования и валидации, а внутри них расположены «нижние» папки – папки, соответствующие классам.

В примере используется другой способ разбить набор данных на обучающие и тестовые наборы - на лету, т.е. часть данных случайным образом определяется как тренировочные, а другие – как проверочные (описание метода - <https://medium.com/@contactsunny/how-to-split-your-dataset-to-train-and-test-datasets-using-scikit-learn-e7cf6eb5e0d>). (в данном примере отсутствует разделение на наборы для тестирования и валидации, то есть набора всего два)

Сначала импортируем модули:

%matplotlib inline  
%config InlineBackend.figure\_format = 'retina'  
import matplotlib.pyplot as plt

import numpy as np  
import torch  
from torch import nn  
from torch import optim  
import torch.nn.functional as F  
from torchvision import datasets, transforms, models

Далее мы определим загрузчик набора данных тренировки / проверки, используя SubsetRandomSampler для разделения:

data\_dir = '/data/train'

def load\_split\_train\_test(datadir, valid\_size = .2):  
 train\_transforms = transforms.Compose([transforms.Resize(224),  
 transforms.ToTensor(),  
 ])

test\_transforms = transforms.Compose([transforms.Resize(224),  
 transforms.ToTensor(),  
 ])

train\_data = datasets.ImageFolder(datadir,   
 transform=train\_transforms)  
 test\_data = datasets.ImageFolder(datadir,  
 transform=test\_transforms)

num\_train = len(train\_data)  
 indices = list(range(num\_train))  
 split = int(np.floor(valid\_size \* num\_train))  
 np.random.shuffle(indices)  
 from torch.utils.data.sampler import SubsetRandomSampler  
 train\_idx, test\_idx = indices[split:], indices[:split]  
 train\_sampler = SubsetRandomSampler(train\_idx)  
 test\_sampler = SubsetRandomSampler(test\_idx)  
 trainloader = torch.utils.data.DataLoader(train\_data,  
 sampler=train\_sampler, batch\_size=64)  
 testloader = torch.utils.data.DataLoader(test\_data,  
 sampler=test\_sampler, batch\_size=64)  
 return trainloader, testloader

trainloader, testloader = load\_split\_train\_test(data\_dir, .2)  
print(trainloader.dataset.classes)

Далее определим, есть ли графический процессор. Предполагается, что работа ведется на машине с графическим процессором, в противном случае код будет как минимум в 10 раз медленнее. Но это хорошая идея, чтобы обобщить и проверить наличие графического процессора.

Также загрузим предварительно обученную модель. Для этого случая выбрана ResNet 50.

device = torch.device("cuda" if torch.cuda.is\_available()   
 else "cpu")  
model = models.resnet50(pretrained=True)  
print(model)

Далее «замораживаются» предварительно обученные слои, чтобы не пропустить их обратно в обучение. Затем переопределяется последний полностью связанный слой, который мы будем тренировать с нашими изображениями. Также создаем критерий (функция потерь) и подбираем оптимизатор (в данном случае Адама) и скорость обучения.

for param in model.parameters():  
 param.requires\_grad = False  
   
model.fc = nn.Sequential(nn.Linear(2048, 512),  
 nn.ReLU(),  
 nn.Dropout(0.2),  
 nn.Linear(512, 10),  
 nn.LogSoftmax(dim=1))  
criterion = nn.NLLLoss()  
optimizer = optim.Adam(model.fc.parameters(), lr=0.003)  
model.to(device)

В этом примере есть только одна эпоха, но в большинстве случаев понадобится больше. Базовый процесс довольно интуитивно понятен из кода: загружаются пакеты изображений и выполняется цикл прямой связи. Затем рассчитывается функция потерь и используется оптимизатор для применения градиентного спуска при обратном распространении – в PyTorch это просто. Большая часть приведенного ниже кода имеет дело с отображением потерь и вычислением точности каждых 10 пакетов, поэтому происходит обновление во время обучения. Во время проверки не забудьте установить модель в режим eval (), а затем вернитесь к train (), как только вы закончите.

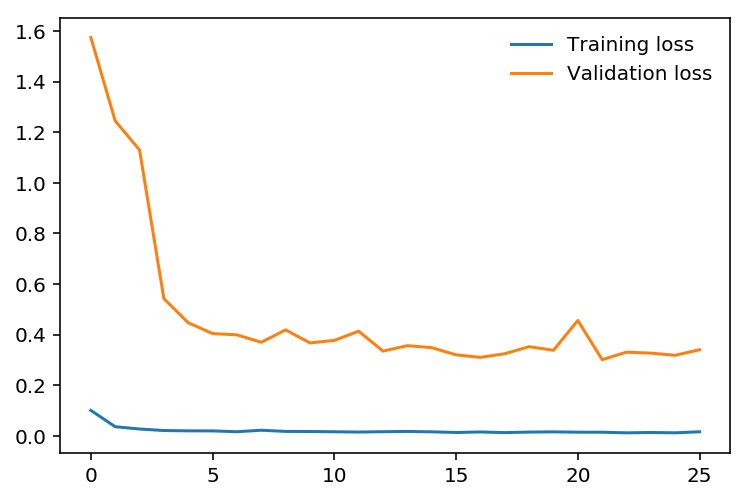
epochs = 1  
steps = 0  
running\_loss = 0  
print\_every = 10  
train\_losses, test\_losses = [], []

for epoch in range(epochs):  
 for inputs, labels in trainloader:  
 steps += 1  
 inputs, labels = inputs.to(device), labels.to(device)  
 optimizer.zero\_grad()  
 logps = model.forward(inputs)  
 loss = criterion(logps, labels)  
 loss.backward()  
 optimizer.step()  
 running\_loss += loss.item()  
   
 if steps % print\_every == 0:  
 test\_loss = 0  
 accuracy = 0  
 model.eval()  
 with torch.no\_grad():  
 for inputs, labels in testloader:  
 inputs, labels = inputs.to(device),  
 labels.to(device)  
 logps = model.forward(inputs)  
 batch\_loss = criterion(logps, labels)  
 test\_loss += batch\_loss.item()  
   
 ps = torch.exp(logps)  
 top\_p, top\_class = ps.topk(1, dim=1)  
 equals =   
 top\_class == labels.view(\*top\_class.shape)  
 accuracy +=  
 torch.mean(equals.type(torch.FloatTensor)).item()  
 train\_losses.append(running\_loss/len(trainloader))  
 test\_losses.append(test\_loss/len(testloader))   
 print(f"Epoch {epoch+1}/{epochs}.. "  
 f"Train loss: {running\_loss/print\_every:.3f}.. "  
 f"Test loss: {test\_loss/len(testloader):.3f}.. "  
 f"Test accuracy: {accuracy/len(testloader):.3f}")  
 running\_loss = 0  
 model.train()  
torch.save(model, 'aerialmodel.pth')

После этого модель будет сохранена для последующих прогнозов.

Есть еще одна вещь, которую можно сделать - это отразить потери обучения и проверки:

plt.plot(train\_losses, label='Training loss')  
plt.plot(test\_losses, label='Validation loss')  
plt.legend(frameon=False)  
plt.show()



Итак, вы обучили свою модель, сохранили ее, и вам необходимо использовать ее в приложении. Для этого вам понадобится сделать простой вывод на изображение.

Объявим папку с изображениями снова, чтобы использовать некоторые примеры оттуда:

data\_dir = '/datadrive/FastAI/data/aerial\_photos/train'

test\_transforms = transforms.Compose([transforms.Resize(224),  
 transforms.ToTensor(),  
 ])

Затем снова проверяем доступность графического процессора, загружаем модель и переводим ее в режим evalution (чтобы параметры не изменялись):

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
model=torch.load('aerialmodel.pth')  
model.eval()

Функция, которая предсказывает класс конкретного изображения, очень проста. Обратите внимание, что для этого требуется Pillow image (Pillow - библиотека языка Python, предназначенная для работы с растровой графикой), а не путь к файлу.

def predict\_image(image):  
 image\_tensor = test\_transforms(image).float()  
 image\_tensor = image\_tensor.unsqueeze\_(0)  
 input = Variable(image\_tensor)  
 input = input.to(device)  
 output = model(input)  
 index = output.data.cpu().numpy().argmax()  
 return index

Теперь для упрощения тестирования также можно создать функцию, которая будет выбирать несколько случайных изображений из папок набора данных:

def get\_random\_images(num):  
 data = datasets.ImageFolder(data\_dir, transform=test\_transforms)  
 classes = data.classes  
 indices = list(range(len(data)))  
 np.random.shuffle(indices)  
 idx = indices[:num]  
 from torch.utils.data.sampler import SubsetRandomSampler  
 sampler = SubsetRandomSampler(idx)  
 loader = torch.utils.data.DataLoader(data,   
 sampler=sampler, batch\_size=num)  
 dataiter = iter(loader)  
 images, labels = dataiter.next()  
 return images, labels

Наконец, для демонстрации получаем случайную выборку изображений, обрабатываем их и показываем результаты:

to\_pil = transforms.ToPILImage()  
images, labels = get\_random\_images(5)  
fig=plt.figure(figsize=(10,10))  
for ii in range(len(images)):  
 image = to\_pil(images[ii])  
 index = predict\_image(image)  
 sub = fig.add\_subplot(1, len(images), ii+1)  
 res = int(labels[ii]) == index  
 sub.set\_title(str(classes[index]) + ":" + str(res))  
 plt.axis('off')  
 plt.imshow(image)  
plt.show()

Git-репозиторий этого примера - <https://github.com/cfotache/pytorch_imageclassifier>

**Пример 2**

В данном примере набор данных был создан вручную, используя изображения из набора данных лиц LFW для «позитивов». Также добавлены несколько случайных изображений для «негативов», которые включают изображения транспортных средств, животных, мебель и т. д.

Данные должны быть разделены на тренировочный, тестовый и валидационный. Тренировочный будет использоваться для обучения модели, валидационный набор будет использоваться для проверки модели после каждой эпохи, а тестовый наборы данных будет использоваться для оценки модели после ее обучения.

Во-первых, нужно поместить набор данных в среду, что можно сделать с помощью: («face» - это имя каталога, который содержит положительный и отрицательный пример лиц)

train\_data = datasets.ImageFolder('face',transform=transform)

Также необходимо определить объект преобразования для выполнения шагов предварительной обработки. Мы можем указать в объекте, какие виды обработки нам нужны. В следующем коде определен объект преобразования, который выполняет горизонтальное переворачивание, случайное вращение, конвертирует массив изображений в PyTorch (поскольку библиотека работает только с Tensors, который является аналогом массива numpy) и, наконец, нормализует изображение.

transform = transforms.Compose([  
 transforms.RandomHorizontalFlip(),  
 transforms.RandomRotation(20),  
 transforms.ToTensor(),  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
 ])

Как только мы закончим с загрузкой набора данных и определением объекта преобразования, мы можем разделить набор данных на наборы обучения, тестирования и проверки, как обсуждалось ранее. Для проведения разделения:

#For test  
num\_data = len(train\_data)  
indices\_data = list(range(num\_data))  
np.random.shuffle(indices\_data)  
split\_tt = int(np.floor(test\_size \* num\_data))  
train\_idx, test\_idx = indices\_data[split\_tt:], indices\_data[:split\_tt]

#For Valid  
num\_train = len(train\_idx)  
indices\_train = list(range(num\_train))  
np.random.shuffle(indices\_train)  
split\_tv = int(np.floor(valid\_size \* num\_train))  
train\_idx, valid\_idx = indices\_train[split\_tv:],indices\_train[:split\_tv]

# define samplers for obtaining training and validation batches  
train\_sampler = SubsetRandomSampler(train\_idx)  
test\_sampler = SubsetRandomSampler(test\_idx)  
valid\_sampler = SubsetRandomSampler(valid\_idx)

#Loaders contains the data in tuple format   
# (Image in form of tensor, label)  
train\_loader = torch.utils.data.DataLoader(train\_data, batch\_size=batch\_size, sampler=train\_sampler, num\_workers=1)

valid\_loader = torch.utils.data.DataLoader(train\_data, batch\_size=batch\_size, sampler=valid\_sampler, num\_workers=1)

test\_loader = torch.utils.data.DataLoader(train\_data, sampler = test\_sampler, batch\_size=batch\_size,num\_workers=1)

# variable representing classes of the images  
classes = [0,1]

(в данном примере существует всего два класса, 0 соответствует отсутствию лица на изображении, 1 – наличию лица на изображении)

Train\_loader, test\_loader и valid\_loader будут использоваться для передачи входных данных в модель.

Далее рассмотрен пример инициализации CNN:

class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 # convolutional layer  
 self.conv1 = nn.Conv2d(3, 16, 5)  
 # max pooling layer  
 self.pool = nn.MaxPool2d(2, 2)  
 self.conv2 = nn.Conv2d(16, 32, 5)  
 self.dropout = nn.Dropout(0.2)  
 self.fc1 = nn.Linear(32\*53\*53, 256)  
 self.fc2 = nn.Linear(256, 84)  
 self.fc3 = nn.Linear(84, 2)  
 self.softmax = nn.LogSoftmax(dim=1)  
   
 def forward(self, x):  
 # add sequence of convolutional and max pooling layers  
 x = self.pool(F.relu(self.conv1(x)))  
 x = self.pool(F.relu(self.conv2(x)))  
 x = self.dropout(x)  
 x = x.view(-1, 32 \* 53 \* 53)  
 x = F.relu(self.fc1(x))  
 x = self.dropout(F.relu(self.fc2(x)))  
 x = self.softmax(self.fc3(x))  
 return x

# create a complete CNN  
model = Net()

Также нужно инициализировать функцию потерь и оптимизатор. Функция потерь поможет рассчитать потери, сравнив прогноз и исходную метку. Оптимизатор минимизирует потери, обновляя параметры модели после каждой эпохи. Они могут быть инициализированы:

# Loss function  
criterion = torch.nn.CrossEntropyLoss()

# Optimizer  
optimizer = torch.optim.SGD(model.parameters(), lr = 0.003, momentum= 0.9)

Для обучения необходимо выполнить следующие шаги:

1. Очистить градиенты всех оптимизированных переменных:

Могут быть градиенты от предыдущих пакетов (batch), поэтому необходимо очищать градиент после каждой эпохи

1. Прямой проход:

Этот шаг вычисляет прогнозируемые результаты путем передачи входных данных в сверточную модель нейронной сети.

1. Рассчитать потери:

Когда модель тренируется, функция потерь вычисляет потери после каждой эпохи, а затем она используется оптимизатором.

1. Обратный проход:

Этот шаг вычисляет градиент потерь относительно параметров модели

1. Оптимизация

Выполняется один шаг оптимизации / обновление параметров для модели.

1. Обновление средней потери обучения

Ниже приведен код для обучения модели (для одной эпохи)

# number of epochs to train the model

n\_epochs = 5

valid\_loss\_min = np.Inf # track change in validation loss

for epoch in range(1, n\_epochs+1):

# keep track of training and validation loss

train\_loss = 0.0

valid\_loss = 0.0

# train the model #

model.train()

for data, target in train\_loader:

# move tensors to GPU if CUDA is available

if train\_on\_gpu:

data, target = data.cuda(), target.cuda()

# clear the gradients of all optimized variables

optimizer.zero\_grad()

# forward pass: compute predicted outputs by passing inputs to the model

output = model(data)

# calculate the batch loss

loss = criterion(output, target)

# backward pass: compute gradient of the loss with respect to model parameters

loss.backward()

# perform a single optimization step (parameter update)

optimizer.step()

# update training loss

train\_loss += loss.item()\*data.size(0)

# Validating the model

model.eval()

for data, target in valid\_loader:

if train\_on\_gpu:

data, target = data.cuda(), target.cuda()

# forward pass: compute predicted outputs by passing inputs to the model

output = model(data)

# calculate the batch loss

loss = criterion(output, target)

# update average validation loss

valid\_loss += loss.item()\*data.size(0)

# calculate average losses

train\_loss = train\_loss/len(train\_loader.dataset)

valid\_loss = valid\_loss/len(valid\_loader.dataset)

# print training/validation statistics

print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(

epoch, train\_loss, valid\_loss))

# save model if validation loss has decreased

if valid\_loss <= valid\_loss\_min:

print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model ...'.format(

valid\_loss\_min,

valid\_loss))

torch.save(model.state\_dict(), 'model\_cifar.pt')

valid\_loss\_min = valid\_loss

Чтобы оценить модель, она должна быть изменена с model.train () на model.eval ()

model.eval()  
# iterate over test data  
len(test\_loader)  
for data, target in test\_loader:  
 # move tensors to GPU if CUDA is available  
 if train\_on\_gpu:  
 data, target = data.cuda(), target.cuda()  
 # forward pass  
 output = model(data)  
 # calculate the batch loss  
 loss = criterion(output, target)  
 # update test loss   
 test\_loss += loss.item()\*data.size(0)  
 # convert output probabilities to predicted class  
 \_, pred = torch.max(output, 1)   
 # compare predictions to true label  
 correct\_tensor = pred.eq(target.data.view\_as(pred))  
 correct = np.squeeze(correct\_tensor.numpy()) if not train\_on\_gpu else np.squeeze(correct\_tensor.cpu().numpy())  
 # calculate test accuracy for each object class  
 for i in range(batch\_size):   
 label = target.data[i]  
 class\_correct[label] += correct[i].item()  
 class\_total[label] += 1

# average test loss  
test\_loss = test\_loss/len(test\_loader.dataset)  
print('Test Loss: {:.6f}\n'.format(test\_loss))

for i in range(2):  
 if class\_total[i] > 0:  
 print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (  
 classes[i], 100 \* class\_correct[i] / class\_total[i],  
 np.sum(class\_correct[i]), np.sum(class\_total[i])))  
 else:  
 print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (  
 100. \* np.sum(class\_correct) / np.sum(class\_total),  
 np.sum(class\_correct), np.sum(class\_total)))

Получаем следующие результаты:

Test Loss: 0.006558

Test Accuracy of 0: 99% (805/807)   
Test Accuracy of 1: 98% (910/921)

Test Accuracy (Overall): 99% (1715/1728)

Поскольку эта модель научилась извлекать черты лица, ее можно использовать и для распознавания лиц – для этого нужно обучить классификатор лиц на ваших собственных изображениях и создать систему распознавания лиц с помощью transfer learning.

Git - репозиторий примера :<https://github.com/jayrodge/Binary-Image-Classifier-PyTorch>