# Konwolucje jednak niepotrzebne... - sprawozdanie

1. Normalizacja, augmentacja, globalne ocieplenie, taka sytuacja. Na szczęście dla niewykształconego w sztucznej inteligencji delikwenta wszystkie potrzebne narzędzia były gotowe w PyTorchu. Ważną rzeczą jest wpierw wczytanie danych celem obliczenia średniej i odchylenia standardowego przed załadowaniem do DataLoadera:

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transforms.ToTensor)
mean = torch.mean(torch.tensor(trainset.data).float(), dim=(0, 1, 2)) / 255.0
std = torch.std(torch.tensor(trainset.data).float(), dim=(0, 1, 2)) / 255.0
transform = transforms.Compose(
    [transforms.RandomHorizontalFlip(),
     transforms.RandomResizedCrop(size=32, scale=(0.8, 1.0), ratio=(0.9, 1.1)),
     transforms.ToTensor(),
     transforms.Normalize(mean, std)])
```

2. Podpięcie transformacji w Torchu zrobiłem poprzez przypisanie pola w trainsecie przed wczytaniem do DataLoadera:

```
trainset.transform = transform
trainloader = torch.utils.data.DataLoader(trainset, batch_size=16, shuffle=True, num_workers=2)
dataiter = iter(trainloader)
images, labels = next(dataiter)
print(images.shape)
```

```
torch.Size([16, 3, 32, 32])
```

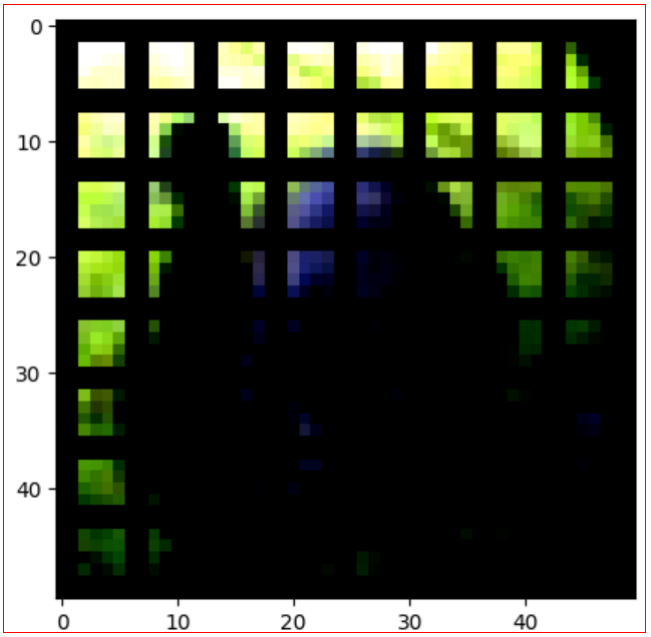
3. Patchowanie ostatnio uprawiałem przy okazji inżynierki, ale nie ma przeciwskazań, by i tutaj ją zastosować:

```
def image_batch_to_patches(image_batch, patch_size):
    batch_size, channels, height, width = image_batch.shape
    num_patches_height = height // patch_size
    num_patches_width = width // patch_size
    patches = torch.zeros((batch_size, num_patches_height * num_patches_width,
                           channels, patch_size, patch_size), dtype=image_batch.dtype, device=cuda0)

    for i in range(batch_size):
        img = image_batch[i]
        for h in range(num_patches_height):
            for w in range(num_patches_width):
                patch = img[:, h * patch_size:(h + 1) * patch_size,
                            w * patch_size:(w + 1) * patch_size]
                patches[i, h * num_patches_width + w] = patch

    return patches
```

Czy efekt jest właściwy względem założenia? Chyba mniej więcej tak:




4. Model sieci transformera jest mocno dyskusyjny: ja myślę, że powinien działać, natomiast ciężko stwierdzić po końcowych wynikach, czy jakiś krok nie jest pominięty/źle zrobiony

```
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.layerNorm1 = nn.LayerNorm(256)
        self.mla = nn.MultiheadAttention(256, 8, 0.2)
        self.layerNorm2 = nn.LayerNorm(256)
        self.linear1 = nn.Linear(256, 512)
        self.gelu = nn.GELU()
        self.linear2 = nn.Linear(512, 256)
        self.dropout = nn.Dropout(0.2)

    def first_block(self, x):
        x = self.layerNorm1(x)
        x, _ = self.mla(x, x, x)
        return x

    def second_block(self, x):
        x = self.layerNorm2(x)
        x = self.linear1(x)
        x = self.gelu(x)
        x = self.dropout(x)
        x = self.linear2(x)
        return self.dropout(x)

    def forward(self, x):
        x = x + self.first_block(x)
        x = x + self.second_block(x)
        return x
```

Positional encoding myliło się z positional embedding, ale mam nadzieję, że chodziło o to samo:

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model, device=cuda0)
        position = torch.arange(0, max_seq_length, dtype=torch.float, device=cuda0).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2, device=cuda0).float() * -(math.log(10000.0) /
d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

Finalny boss, czyli cały model złączony w całość:

```
class VisualTransformer(nn.Module):
    def __init__(self):
        super(VisualTransformer, self).__init__()
        self.embedding = nn.Linear(48, 256)
        self.class_token = nn.Parameter(torch.zeros(1, 1, 256))
        self.positional_encoding = PositionalEncoding(256, 65).to(cuda0)
        self.transformer_blocks = nn.ModuleList([Transformer().to(cuda0) for _ in range(6)])
        self.layerNorm = nn.LayerNorm(256)
        self.linear = nn.Linear(256, 10)

    def forward(self, x):
        x = image_batch_to_patches(x, 4)
        x = x.view(x.size(0), x.size(1), -1)
        x = self.embedding(x)
        x = torch.cat((self.class_token.repeat(16, 1, 1), x), dim=1)
        x = self.positional_encoding(x)
        for transformer_block in self.transformer_blocks:
            x = transformer_block(x)
        x = self.layerNorm(x)
        x = x[:, 0]
        x = self.linear(x)
        return x
```

I w trochę bardziej przystępnej formie:

```
VisualTransformer(
  (embedding): Linear(in_features=48, out_features=256, bias=True)
  (positional_encoding): PositionalEncoding()
  (transformer_blocks): ModuleList(
    (0-5): 6 x Transformer(
      (layerNorm1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (mla): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
      )
      (layerNorm2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (linear1): Linear(in_features=256, out_features=512, bias=True)
      (gelu): GELU(approximate='none')
      (linear2): Linear(in_features=512, out_features=256, bias=True)
      (dropout): Dropout(p=0.2, inplace=False)
    )
  )
  (layerNorm): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
  (linear): Linear(in_features=256, out_features=10, bias=True)
)
```

5. Trening może i trwał dość długo, ale za to nie przynosił wymiernych efektów:

```
optimizer = torch.optim.AdamW(visualTransformer.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[100, 150], gamma=0.1)

for epoch in range(epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        real_inputs = inputs.to(cuda0)
        real_labels = labels.to(cuda0)
        optimizer.zero_grad()
        outputs = visualTransformer(real_inputs)
        loss = criterion(outputs, real_labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    scheduler.step()
    print('[%d] loss: %.3f' % (epoch + 1, running_loss / 2000))
    running_loss = 0.0
```

Ze względu na ograniczenia mocy i czasu (jedna epoka ponad 5 minut), wybrałem trening przez 50 epok:

```
[1] loss: 3.635
[2] loss: 3.605
[3] loss: 3.601
[4] loss: 3.600
[5] loss: 3.600
[6] loss: 3.599
[7] loss: 3.599
[8] loss: 3.599
[9] loss: 3.599
[10] loss: 3.599
[11] loss: 3.599
[12] loss: 3.599
[13] loss: 3.599
[14] loss: 3.599
[15] loss: 3.599
[16] loss: 3.599
[17] loss: 3.599
[18] loss: 3.599
[19] loss: 3.599
[20] loss: 3.599
[21] loss: 3.598
[22] loss: 3.598
[23] loss: 3.598
[24] loss: 3.598
[25] loss: 3.598
[26] loss: 3.598
[27] loss: 3.598
[28] loss: 3.598
[29] loss: 3.598
[30] loss: 3.598
[31] loss: 3.598
[32] loss: 3.598
[33] loss: 3.598
[34] loss: 3.598
[35] loss: 3.598
[36] loss: 3.598
[37] loss: 3.598
[38] loss: 3.598
[39] loss: 3.598
[40] loss: 3.598
[41] loss: 3.598
[42] loss: 3.598
[43] loss: 3.598
[44] loss: 3.598
[45] loss: 3.598
[46] loss: 3.598
[47] loss: 3.598
[48] loss: 3.598
[49] loss: 3.598
[50] loss: 3.598
```

6. Rezultaty wytrenowanego modelu były niestety bardzo mierne. Najpierw wczytajmy zbiór danych testowych:

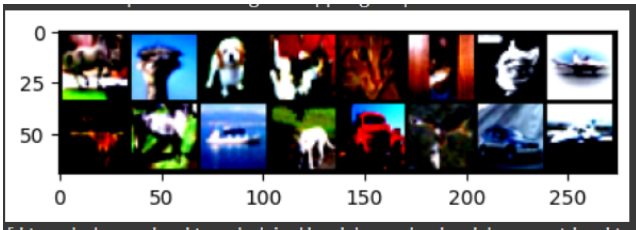
```
testset = torchvision.datasets.CIFAR10(root='./data',  download=True, transform=transforms.ToTensor)
mean = torch.mean(torch.tensor(testset.data).float(), dim=(0, 1, 2)) / 255.0
std = torch.std(torch.tensor(testset.data).float(), dim=(0, 1, 2)) / 255.0
transform = transforms.Compose(
    [transforms.RandomHorizontalFlip(),
     transforms.RandomResizedCrop(size=32, scale=(0.8, 1.0), ratio=(0.9, 1.1)),
     transforms.ToTensor(),
     transforms.Normalize(mean, std)])
testset.transform = transform
testloader = torch.utils.data.DataLoader(testset, batch_size=16, shuffle=True, num_workers=2)
```

A następnie policzmy wynik sieci:

```
all = 0
good = 0
with torch.no_grad():
  for i, data in enumerate(testloader, 0):
      inputs, labels = data
      real_inputs = inputs.to(cuda0)
      real_labels = labels.to(cuda0)
      outputs = visualTransformer(real_inputs)
      for i in range(16):
        if torch.argmax(outputs[i]).item() == real_labels[i].item():
          good += 1
        all += 1
  print(good/all)
```

Skuteczność na zbiorze testowym wyniosła zaledwie 10.19% :(, więc właściwie nie różniła się od losowego wyboru etykiety. Również ręczne sprawdzenie przynosi żenujący efekt (lewa etykieta sieci, prawa etykieta rzeczywista):

```
random_images, random_labels = next(iter(testloader))
imshow(torchvision.utils.make_grid(random_images))
model_labels = visualTransformer(random_images)
print([get_label(torch.argmax(model_labels[i]).item()) + " " + get_label(random_labels[i]) for i in
range(8)])
print([get_label(torch.argmax(model_labels[i]).item()) + " "  + get_label(random_labels[i]) for i in
range(8, 16)])
```




```
['truck horse', 'truck bird', 'deer dog', 'deer cat', 'truck cat', 'dog cat', 'deer cat', 'truck airplane']
['truck deer', 'automobile horse', 'horse ship', 'truck dog', 'deer truck', 'truck frog', 'deer automobile', 'airplane airplane']
```

Niestety wygląda na to, że na którymś etapie projektowania modelu popełniłem błąd. Łudziłem się po wynikach pierwszych epok, że coś się poprawi, co oczywiście nie nastąpiło. Jak to mówił selekcjoner reprezentacji Jerzy Brzęczek: “Próbuj”. Więc i ja spróbowałem.

Michał Wójcik