MRO - zadanie 6: Konwolucje jednak niepotrzebne...

Piotr Zawiślan

16 Grudzień 2022

1 Wykorzystane technologie

Kod pisałem w języku Python z wykorzystaniem frameworka Pytorch.

2 Przygotowanie danych

Transformację danych wykonałem używając modułu **torchvision.transform**. Pobrałem dataset, a następnie podmieniłem atrybut transform na złożone transformacje, które zajmują się augmentacją danych.

```
cifar_train_ds = datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=Compose([ToTensor(),
                       ConvertImageDtype(torch.float)]),
    target_transform=Lambda(
        lambda y: torch.zeros(10, dtype=torch.float).scatter_(0, torch.tensor(y),
    value=1)
    )
)

cifar_train_ds.transform = Compose([
    ToTensor(),
    RandomHorizontalFlip(p=0.5),
    RandomResizedCrop(size=32, scale=(0.8, 1.0), ratio=(0.9, 1.1)),
    Normalize(
        mean=tuple((cifar_train_ds.data / 255).mean(axis=(0, 1, 2))),
        std=tuple((cifar_train_ds.data / 255).std(axis=(0, 1, 2)))
    ),
    ConvertImageDtype(torch.float)
])
```

Listing 1: Transformacje wejściowe dla zbioru CIFAR10

3 Cięcie obrazu na fragmenty

Aby podzielić obraz na patche, dwa razy użyłem metody **unfold**, a następnie zamieniłem wymiary, tak aby uzyskać kształt nadający się do dalszego przetwarzania.

```
def get_square_patches_from_batch(X: torch.Tensor, size: int) -> torch.Tensor:
    """Returns tensor of patches with shape [B, H // size, W // size, C, size, size]
    """
    assert X.ndim == 4, 'ndim should be 4'
    batch_size, n_channels, n_rows, n_cols = X.shape
    step = size
    patches = X.unfold(2, size, step).unfold(3, size, step).permute(0, 2, 3, 1, 4, 5)
    return patches
```

Listing 2: Funkcja tnąca obraz na kwadratowe patche.

Rysunek 1: Ptak (prawdopodobnie kazuar)

Rysunek 2: Ptak pocięty na patche o rozmiarze 8

4 Definicja modelu

Zaprojektowałem model *Vision Transformera*, jak określone w poleceniu. Niestety, nie udało mi się wyświetlić computational graph sieci, ale raczej działa jak należy :).

```python
class ToPatches(nn.Module):
    """
    Transforms image to flattened patches.

    [B, C, H, W] -> [B, H // size, W // size, C, size, size] ->
    [B, N (n_patches), P (size_of_flattened_patch)]
    """

    def __init__(self, patch_size=4) -> None:
        super().__init__()
        self.patch_size: int = patch_size

    def forward(self, X: torch.Tensor) -> torch.Tensor:
        patches = get_square_patches_from_batch(X, size=self.patch_size)
        flat_patches = patches.flatten(start_dim=3).flatten(start_dim=1, end_dim=2)
        return flat_patches
```

Listing 3: Definicja modułu zwracającego pocięte i spłaszczone patche.

```python
class VisionTransformer(nn.Module):
    def __init__(self, emb_size=256, n_heads=8, dropout_rate=0.2) -> None:
        super().__init__()

        self.emb_size: int = emb_size
        self.n_heads: int = n_heads
        self.dropout_rate: float = dropout_rate

        self.layer_norm_1 = nn.LayerNorm(normalized_shape=self.emb_size)
        self.mh_attention = nn.MultiheadAttention(
            embed_dim=self.emb_size,  num_heads=self.n_heads, batch_first=True
        )

        self.layer_norm_2 = nn.LayerNorm(normalized_shape=self.emb_size)
        self.mlp_1 = nn.Linear(in_features=self.emb_size, out_features=2*self.emb_size
    )
        self.gelu = nn.GELU()
        self.dropout_1 = nn.Dropout(self.dropout_rate)

        self.mlp_2 = nn.Linear(in_features=2*self.emb_size, out_features=self.emb_size
    )
        self.dropout_2 = nn.Dropout(self.dropout_rate)

    def forward(self, X: torch.Tensor) -> torch.Tensor:
        X_norm = self.layer_norm_1(X)

        mh_attention_output, _ = self.mh_attention(
            query=X_norm, key=X_norm, value=X_norm
        )

        residual_add_1 = mh_attention_output + X

        X_mlp_1 = self.layer_norm_2(residual_add_1)
        X_mlp_1 = self.mlp_1(X_mlp_1)
        X_mlp_1 = self.gelu(X_mlp_1)
        result_mlp_1 = self.dropout_1(X_mlp_1)

        X_mlp_2 = self.mlp_2(result_mlp_1)
        result_mlp_2 = self.dropout_2(X_mlp_2)

        residual_add_2 = residual_add_1 + result_mlp_2

        return residual_add_2
```

Listing 4: Definicja modułu pojedynczego transformera z multi-head attention.

```python
class CifarViTClassifier(nn.Module):
    def __init__(
        self, n_classes=10, patch_size=4,
        n_attention_heads=8, n_transformers=6,
        emb_size=256, image_size=32, dropout_rate=0.2
    ) -> None:

        super().__init__()

        self.n_classes: int = n_classes
        self.patch_size: int = patch_size
        self.n_attention_heads: int = n_attention_heads
        self.n_transformers: int = n_transformers
        self.emb_size: int = emb_size
        self.image_size: int = image_size
        self.dropout_rate: float = dropout_rate

        self.patch_dim: int = 3 * self.patch_size**2
        self.n_patches: int = (self.image_size // self.patch_size)**2

        # learnable parameters
        self.pos_embedding = nn.Parameter(torch.randn(1, self.n_patches + 1, self.
    emb_size))
        self.cls_token = nn.Parameter(torch.randn(1, 1, self.emb_size))

        # image -> patches -> patches_embedding
        self.to_patch_embedding = nn.Sequential(
            ToPatches(patch_size=self.patch_size),
            nn.Linear(in_features=self.patch_dim, out_features=self.emb_size)
        )

        self.input_dropout = nn.Dropout(p=self.dropout_rate)

        # self.n_transformers x transformers
        self.transformers_seq = nn.Sequential(*[
            VisionTransformer(
                emb_size=self.emb_size,
                n_heads=self.n_attention_heads,
                dropout_rate=self.dropout_rate
            ) for _ in range(self.n_transformers)
        ])

        # linear head
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(self.emb_size),
            nn.Linear(self.emb_size, self.n_classes)
        )


    def forward(self, X: torch.Tensor) -> torch.Tensor:
        batch_size = X.shape[0]
        embedded_patches = self.to_patch_embedding(X)


        cls_tokens = self.cls_token.expand(batch_size, -1, -1)
        embedded_patches_with_cls = torch.cat((cls_tokens, embedded_patches), dim=1)
        embedded_patches_with_cls += self.pos_embedding

        trans_X = self.input_dropout(embedded_patches_with_cls)
        trans_output = self.transformers_seq(trans_X)

        cls_tokens_out = trans_output[:, 0, :]

        result = self.mlp_head(cls_tokens_out)
        return result
```

Listing 5: Definicja modelu ViT.

5 Trening

Wytrenowałem model przez 160 epok, z optimizerem **AdamW** oraz z schedulerem **MultiStepLR**. Trening trwał około 2,6 godziny z użyciem karty GTX 1660 Ti.

```python
def train_step(
    model: nn.Module,
    data_loader: DataLoader,
    loss_fn: nn.CrossEntropyLoss,
    optimizer
) -> None:

    num_batches = len(data_loader)
    epoch_loss = 0.
    epoch_correct_preds = 0
    model.train()
    t_start = perf_counter()

    for batch, (X, y) in enumerate(data_loader):
        X, y = X.to('cuda'), y.to('cuda')
        optimizer.zero_grad()
        prediction = model(X)
        loss = loss_fn(prediction, y)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        correct_preds = (prediction.argmax(dim=1) == y.argmax(dim=1)).count_nonzero().item()
        epoch_correct_preds += correct_preds

    t_end = perf_counter()
    epoch_train_time = t_end - t_start
    epoch_avg_loss = epoch_loss / num_batches
    epoch_avg_acc = epoch_correct_preds / (num_batches * data_loader.batch_size)

    return epoch_avg_loss, epoch_avg_acc, epoch_train_time
```

Listing 6: Funkcja wykonująca jedną epokę treningu.

```python
def val_step(
    model: nn.Module,
    data_loader: DataLoader,
    loss_fn: nn.CrossEntropyLoss
) -> None:

    num_batches = len(data_loader)
    epoch_loss = 0.
    epoch_correct_preds = 0
    model.eval()
    with torch.no_grad():
        for batch, (X, y) in enumerate(data_loader):
            X, y = X.to('cuda'), y.to('cuda')
            prediction = model(X)
            loss = loss_fn(prediction, y)
            epoch_loss += loss.item()
            correct_preds = (prediction.argmax(dim=1) == y.argmax(dim=1)).count_nonzero().item()
            epoch_correct_preds += correct_preds

        epoch_avg_loss = epoch_loss / num_batches
        epoch_avg_acc = epoch_correct_preds / (num_batches * data_loader.batch_size)

        return epoch_avg_loss, epoch_avg_acc
```

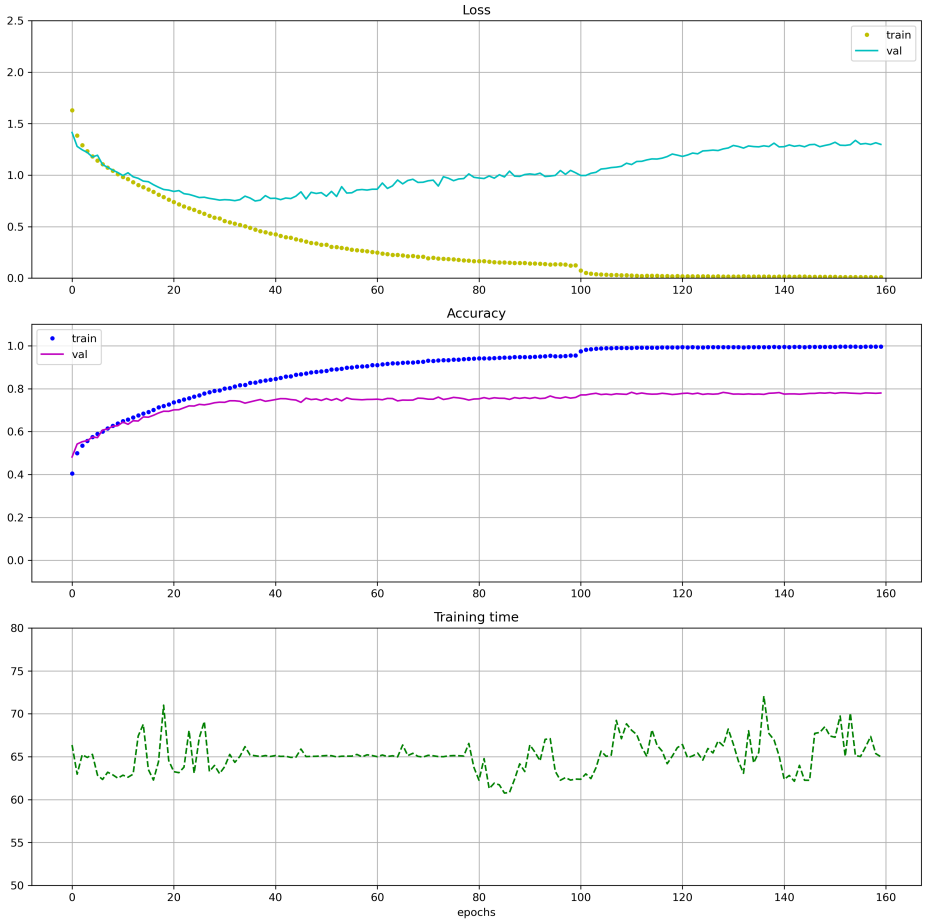
Listing 7: Funkcja wykonująca walidaję po każdej epoce.

W poniższej pętli treningowej wykorzystałem obie powyższe funkcje, oraz zastosowałem metodę *Early stopping*.

```
vit_model = CifarViTClassifier().to('cuda')
vit_model.requires_grad_(True)

loss_function = nn.CrossEntropyLoss().cuda()

optimizer = AdamW(
    vit_model.parameters(),
    lr=5e-4
)

lr_scheduler = MultiStepLR(
    optimizer=optimizer,
    milestones=[100, 150],
    gamma=0.1
)

train_data_loader = DataLoader(cifar_train_ds, batch_size=64, shuffle=True)
test_data_loader = DataLoader(cifar_test_ds, batch_size=64, shuffle=True)

train_loss_acc_time_list = []
val_loss_acc_list = []

max_test_acc = 0.
early_stopped_vit_model = CifarViTClassifier().to('cuda')
early_stopped_vit_model.load_state_dict(vit_model.state_dict())

n_epochs = 160

for epoch in range(n_epochs):
    avg_train_loss, avg_train_acc, epoch_train_time = train_step(
        vit_model,
        train_data_loader,
        loss_function,
        optimizer
    )

    avg_val_loss, avg_val_acc = val_step(
        vit_model,
        test_data_loader,
        loss_function
    )

    lr_scheduler.step()

    train_loss_acc_time_list.append((avg_train_loss, avg_train_acc, epoch_train_time))
    val_loss_acc_list.append((avg_val_loss, avg_val_acc))

    if avg_val_acc > max_test_acc:
        max_test_acc = avg_val_acc
        early_stopped_vit_model.load_state_dict(vit_model.state_dict())

    if epoch in (10, 25, 50, 100, 125):
        torch.save(vit_model.state_dict(), Path.cwd() / f'vit-cifar10-{epoch}.pt')
```

Listing 8: Główna pętla treningowa

Historię treningu przedstawiłem na poniższym wykresie. Wyraźnie widać poprawę dokładności na zbiorze treningowym (a na walidacyjnym może i też) po setnej epoce (wtedy, kiedy **lr_scheduler** pomnożył *learning rate* przez 0.1.



Rysunek 3: Statystyki treningu

Przetestowałem model na kilku obrazach ze zbioru testowego i zazwyczaj, jeżeli ja byłem w stanie rozpoznać obraz, to sieć też sobie radziła dobrze (przykładowo: niektóre żaby są strasznie rozmazane i ciężko je rozpoznać :)).

6 Attention rollout

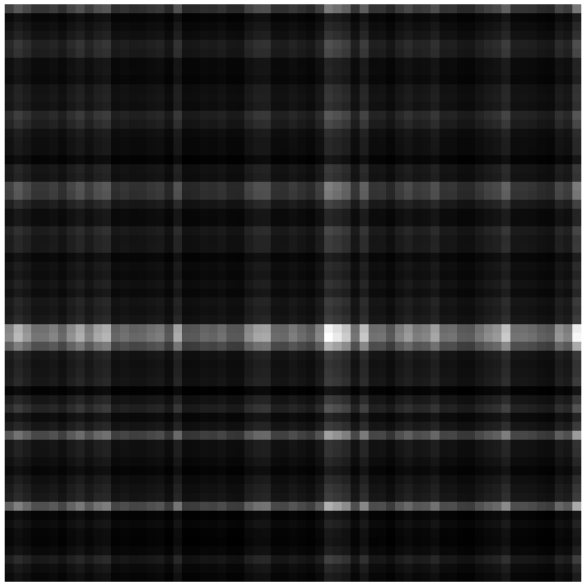
Aby uzyskać mapę atencji metodą *Attention rollout*, wykorzystałem funkcję, modyfikującą *forward hook* modułów **MultiHeadAttention**, tak aby zapisywane zostały macierze atencji. Następnie, przemnożyłem rekurencyjnie te macierze i uzyskałem *attention rollout matrix*.

```python
def get_mha_matrix_from_nth_transformer(
    vit_m: CifarViTClassifier,
    n: int,
    image: torch.Tensor
) -> torch.Tensor:
    """Get multi-head attention matrix from nth transformer using a forward hook."""

    result_activation = None

    def output_hook(model, _input, _output_tuple):
        nonlocal result_activation
        result_activation = _output_tuple[1].detach()

    handle = vit_m.transformers_seq[n].mh_attention.register_forward_hook(output_hook)

    _ = vit_m(image.unsqueeze(0))

    handle.remove()

    return result_activation.squeeze()
```

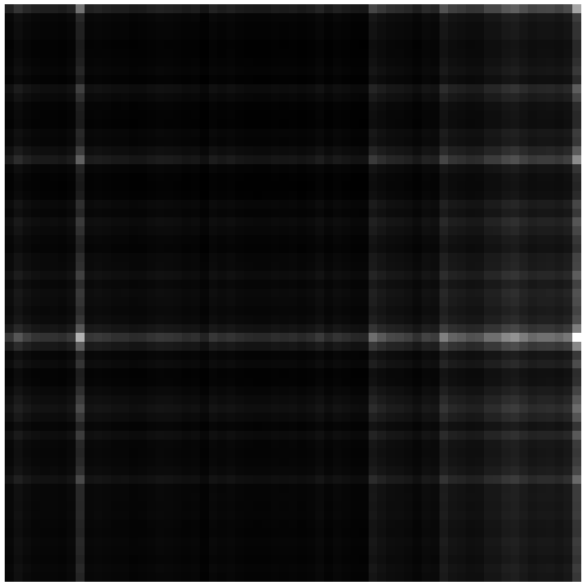
Listing 9: Funkcja zwracająca macierz atencji z n-tego transformera.

```python
def get_patches_attention_matrix_rollout(
    vit_m: CifarViTClassifier,
    image: torch.Tensor
) -> torch.Tensor:

    att_mat = get_mha_matrix_from_nth_transformer(vit_m, 0, image)
    for i in range(1, vit_m.n_transformers):
        ith_att_mat = get_mha_matrix_from_nth_transformer(vit_m, i, image)
        att_mat = att_mat.T @ ith_att_mat

    return att_mat
```

Listing 10: Funkcja zwracająca attention rollout matrix (liczba patchy + 1 x liczba patchy + 1)



Rysunek 4: Macierz atencji dla zdjęcia ptaka ze zbioru treningowego.

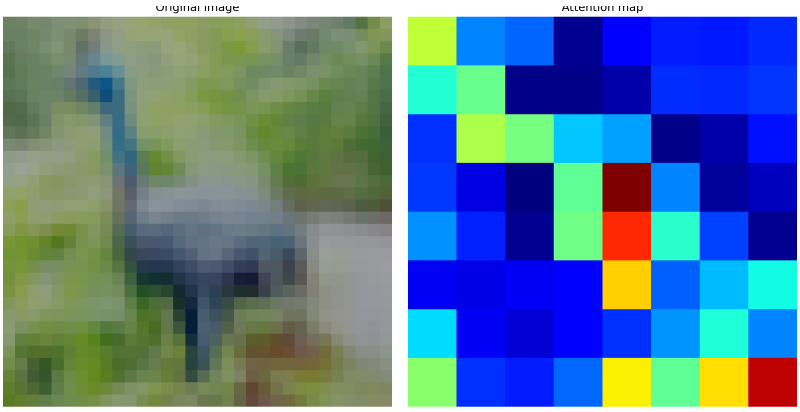


Rysunek 5: Macierz atencji dla zdjęcia żaglówki ze zbioru testowego.

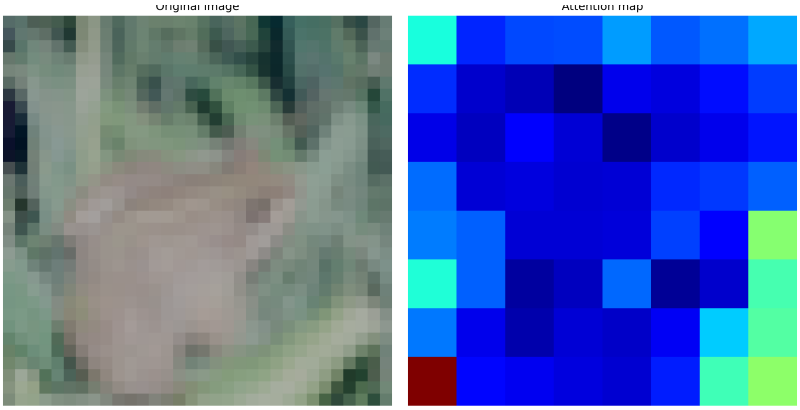
Mapy atencji uzyskałem z macierzy atencji korzystając z poniższego skrawku kodu.
Poniższe mapy przedstawiają (a przynajmniej powinny w teorii) atencję odnośnie *cls_token*.

```
#  plot attention with regard to cls token
enlarged_attention_map = att_matrix_rollout[0, 1:].reshape(
    int(np.sqrt(n_patches)), int(np.sqrt(n_patches))
)
enlarged_attention_map = resize(
    enlarged_attention_map.unsqueeze(0), size=(im_size, im_size),
    interpolation=torchvision.transforms.InterpolationMode.NEAREST
).squeeze()
plt.imshow(np.moveaxis(enlarged_attention_map.detach().numpy(), 0, -1), cmap='jet')
```

Listing 11: Skrawek prezentujący otrzymanie mapy atencji z attention matrix rollout i wyplotowanie jej.



Rysunek 6: Mapa atencji dla zdjęcia ptaka ze zbioru treningowego.



Rysunek 7: Mapa atencji dla zdjęcia żaby ze zbioru testowego.

Chętnie bym się dowiedział, czy zaimplementowałem metodę *attention rollout* poprawnie, bo mam podejrzenia, że gdzieś mógł wkraść się błąd. Druga możliwość jest taka, że sieć jest nadmiernie dopasowana do zbioru treningowego i należałoby poprawić generalizację modelu (np. stosując bardziej zaawansowane metody augmentacji).