Zadanie 6.

Zadanie wykonałem korzystając z frameworka PyTorch. Na początku przygotowałem pipeline do transformacji i augmentacji danych treningowych korzystając z modułu torchvision.transforms.v2. Wcześniej obliczyłem wymagane do normalizacji wartości średniej i odchylenia standardowego dla zbioru treningowego CIFAR-10.

```
CIFAR_MEAN = np.array([0.49139968, 0.48215827, 0.44653124])
CIFAR_STD = np.array([0.24703233, 0.24348505, 0.26158768])

ToTensor = v2.Compose([v2.ToImage(), v2.ToDtype(torch.float32, scale=True)])
transforms = v2.Compose(
    [
        ToTensor,
        v2.RandomHorizontalFlip(p=0.5),
        v2.RandomResizedCrop(size=32, scale=(0.8, 1.0), ratio=(0.9, 1.1)),
        v2.Normalize(CIFAR_MEAN, CIFAR_STD),
    ]
)
```

Tak przygotowany pipeline przekazałem jako argument transform do klasy reprezentującej zbiór CIFAR-10 z modułu torchvision.datasets.

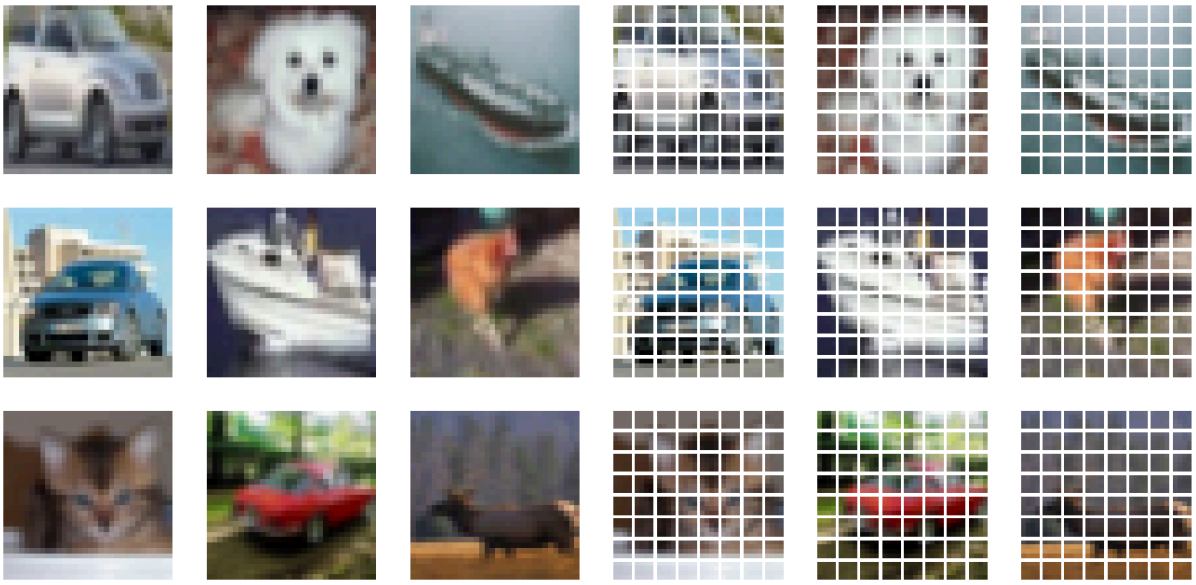
```
TRAIN_SET = datasets.CIFAR10(root="./data", train=True, download=True, transform=transforms)
TRAIN_SET, VALID_SET = torch.utils.data.random_split(TRAIN_SET, [0.9, 0.1])
```

Następnie zaimplementowałem procedurę realizującą cięcie obrazów na patche. Zaproponowana implementacja nie jest bardzo szybka, gdyż używa pythonowych list comprehension, ale czas treningu był akceptowalny, więc uznałem ją za wystarczającą i nie próbowałem jej optymalizować.

```
class Patchify(nn.Module):
    def __init__(self, image_size: tuple[int, int], patch_size: tuple[int, int], flat: bool):
        super().__init__()
        self.image_height, self.image_width = image_size
        self.patch_height, self.patch_width = patch_size
        self.flatten = nn.Flatten(start_dim=2) if flat else nn.Identity()

        assert self.image_height % self.patch_height == 0
        assert self.image_width % self.patch_width == 0

    def forward(self, x: Tensor) -> Tensor:
        patches = torch.stack(
            [
                x[..., i : i + self.patch_height, j : j + self.patch_width]
                for i in range(0, self.image_height, self.patch_height)
                for j in range(0, self.image_width, self.patch_width)
            ],
            dim=1,
        )
        return self.flatten(patches)
```

Poniżej zamieściłem efekt działania zaimplementowanej procedury dla przykładowego batcha obrazów.

```
imgs, _ = zip(*choices(TRAIN_SET, k=9))
imgs = torch.stack(imgs)
imgs = v2.Normalize(mean=-CIFAR_MEAN / CIFAR_STD, std=1 / CIFAR_STD)(imgs)

show(imgs, figsize=(10, 10))
show_patches(Patchify(image_size=(32, 32), patch_size=(4, 4), flat=False)(imgs),
             image_size=(32, 32), figsize=(10, 10))
```



Kolejnym krokiem była implementacja modelu Vision Transformer. Przed właściwą implementacją ViT zdefiniowałem dwie pomocnicze warstwy – MultiheadSelfAttention będącą nadbudową nad dostępną w PyTorchu warstwą MultiheadAttention, która przy każdym forward pass zapisuje również macierz atencji przydatną później do metody attention rollout.

```
class MultiheadSelfAttention(nn.Module):
    def __init__(self, dim, n_heads, dim_head, dropout):
        super().__init__()

        embed_dim = dim_head * n_heads
        self.attn_output_weights: Tensor | None = None

        self.q = nn.Linear(dim, embed_dim, bias=False)
        self.k = nn.Linear(dim, embed_dim, bias=False)
        self.v = nn.Linear(dim, embed_dim, bias=False)
        self.attn = nn.MultiheadAttention(embed_dim, n_heads, dropout, batch_first=True)
        self.proj = nn.Sequential(nn.Linear(embed_dim, dim), nn.Dropout(dropout))

    def forward(self, x: Tensor) -> Tensor:
        q, k, v = self.q(x), self.k(x), self.v(x)
        x, self.attn_output_weights = self.attn(q, k, v)
        return self.proj(x)
```

oraz warstwę TransformerLayer – implementującą pojedynczy blok transformerowego enkodera.

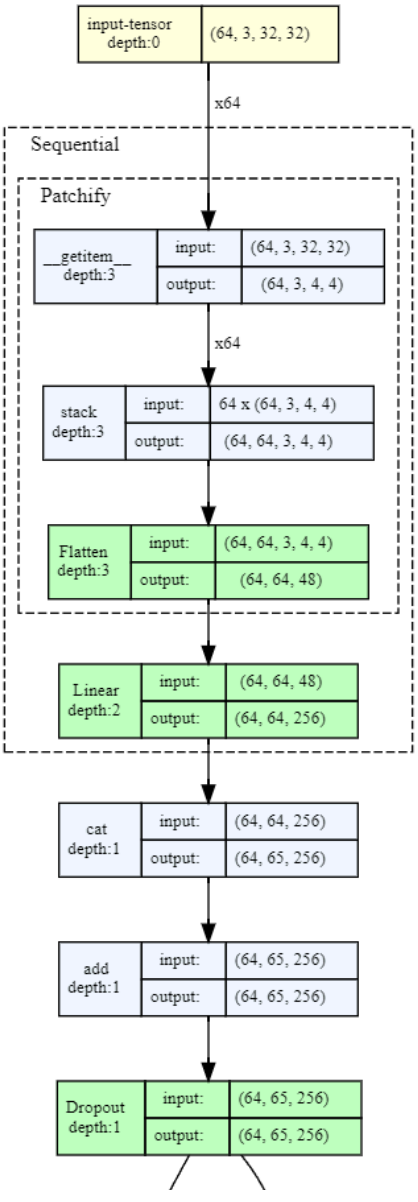
```
class TransformerLayer(nn.Module):
    def __init__(self, dim: int, n_heads: int, dim_head: int, mlp_dim: int, dropout: float):
        super().__init__()

        self.norm1 = nn.LayerNorm(dim)
        self.norm2 = nn.LayerNorm(dim)
        self.atten_blk = MultiheadSelfAttention(dim, n_heads, dim_head, dropout)
        self.dense_blk = nn.Sequential(
            nn.Linear(dim, mlp_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(mlp_dim, dim),
            nn.Dropout(dropout),
        )

    def forward(self, x: Tensor) -> Tensor:
        x = x + self.atten_blk(self.norm1(x))
        x = x + self.dense_blk(self.norm2(x))
        return x
```

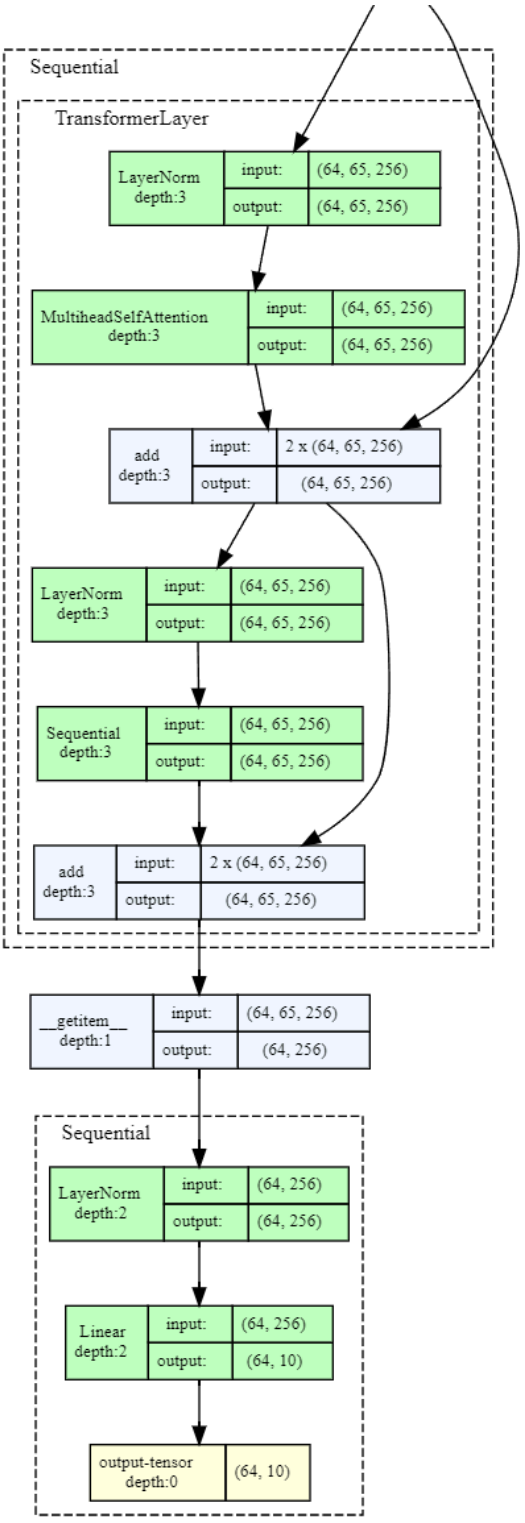
Poniżej zamieściłem implementację właściwego modelu ViT oraz jego computation graph (dla czytelności w grafie jest pokazany tylko jeden blok Transformera) uzyskany z pomocą biblioteki torchview.

```
class ViT(nn.Module):
    def __init__(
        self,
        image_size: tuple[int, int],
        patch_size: tuple[int, int],
        channels: int,
        n_classes: int,
        n_heads: int,
        dim: int,
        dim_mlp: int,
        dim_head: int,
        depth: int,
        dropout: float,
    ):
        super().__init__()

        image_height, image_width = image_size
        patch_height, patch_width = patch_size
        n_patches = (image_height // patch_height) * (image_width // patch_width)

        self.to_patch_embedding = nn.Sequential(
            Patchify(image_size, patch_size, flat=True),
            nn.Linear(channels * patch_height * patch_width, dim),
        )
        self.transformer_layers = [
             TransformerLayer(dim, n_heads, dim_head, dim_mlp, dropout) for _ in range(depth)
        ]
        self.transformer = nn.Sequential(*self.transformer_layers)

        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.pos_embed = nn.Parameter(torch.randn(1, n_patches + 1, dim))
```

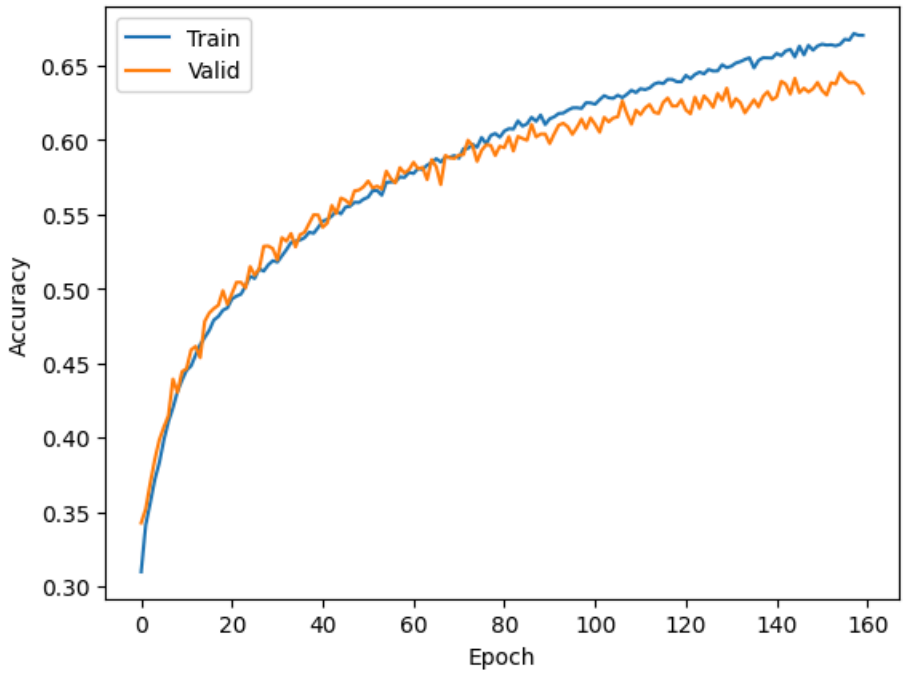
```
        self.clf_head = nn.Sequential(nn.LayerNorm(dim), nn.Linear(dim, n_classes))
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: Tensor) -> Tensor:
        # (B, C, H, W)
        b = x.shape[0]

        # (B, T, N_emb)
        x = self.to_patch_embedding(x)

        # (B, T+1, N_emb)
        x = torch.cat((self.cls_token.repeat(b, 1, 1), x), dim=1)
        x = self.dropout(x + self.pos_embed)
        x = self.transformer(x)

        # (B, N_emb)
        x = x[:, 0, :]

        # (B, N_cls)
        x = self.clf_head(x)

        return x
```

```
model_graph = draw_graph(vit, input_size=(64, 3, 32, 32), depth=3, expand_nested=True)
model_graph.visual_graph
```

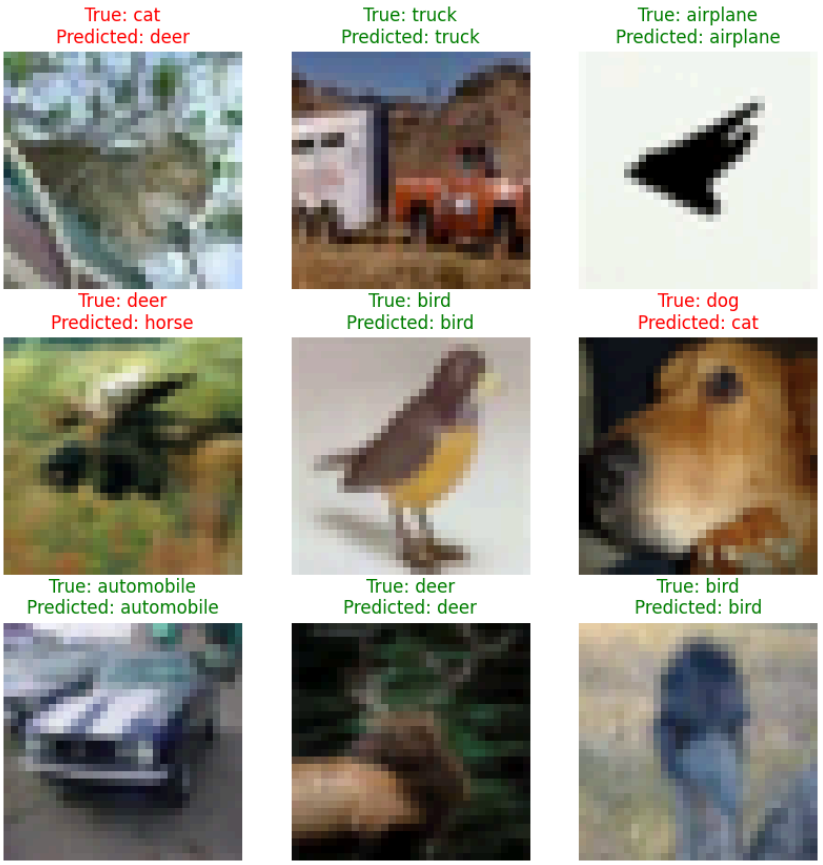




Następnie zaimplementowałem pętlę treningową i wytrenowałem model. Jako scheduler dla stałej uczącej wykorzystałem MultiStepLR z parametrami milestones=[100,150] i gamma=0.1. Cały trening zajął około 3h. Poniżej zamieściłem wykres accuracy w kolejnych epokach na zbiorze treningowym i walidacyjnym. Na wykresie widać, iż model zaczyna się przeuczać, a sam wynik accuracy jest mało zadowalający.

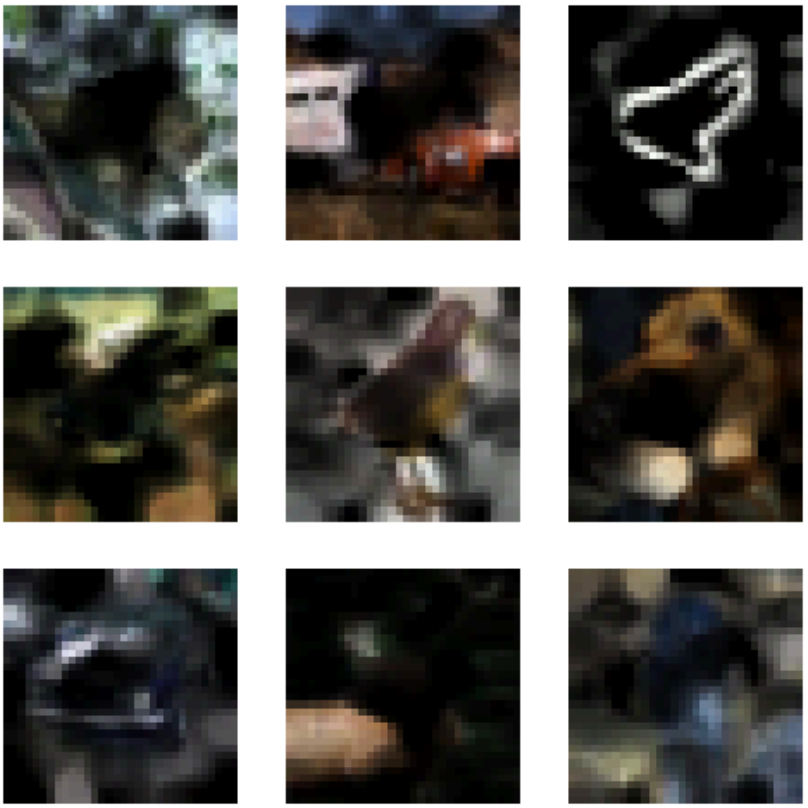


Najlepszy wytrenowany model ViT uzyskał na zbiorze testowym wynik accuracy 64.55%, co jest wynikiem bardzo słabym, zwłaszcza biorąc pod uwagę spory koszt obliczeniowy wymagany do jego wytrenowania. Poniżej zamieściłem kilka przykładowych predykcji modelu na niewielkim batchu obrazów ze zbioru testowego.



Ostatnim etapem było zaimplementowanie metody attention rollout i wizualizacja uwagi sieci. Metoda attention rollout wykorzystuje tzw. attention weights (macierze uwagi?) pojedynczego bloku Transformera, czyli macierze wymiaru (liczba tokenów) x (liczba tokenów), których element (i,j) określa ilościowo uwagę jaką przykłada i-ty token do j-tego. W przypadku wielu głów możemy brać np. minimum albo średnią. Warstwa MultiheadAttention w PyTorchu na szczęście zwraca attention weights jako jedno z jej wyjść i domyślnie używa uśredniania po wielu głowach, więc nie musimy sami wyciągać attention weights. Algorytm attention rollout podaje przepis jak obliczyć taką wynikową macierz uwagi dla sekwencji bloków Transformera i jest dany jako $R(i) = R(i-1) \mathbin{@} (A(i) + I)$, gdzie i to indeks bloku, $A(i)$ to macierz uwagi w i-tym bloku Transformera, a I to macierz jednostkowa, która pozwala uwzględnić połączenia rezydualne. Ostatecznie otrzymujemy zatem macierz R wymiaru (liczba tokenów) x (liczba tokenów), która zawiera zagregowaną ilościową informację jaką uwagę przykłada każdy token do każdego innego. Aby otrzymać teraz coś co możemy nałożyć na obraz bierzemy pierwszy wiersz macierzy R, który zawiera wartości uwagi jaką przykłada class token do każdego z patchy obrazka (ignorujemy tutaj self-attention class tokena), robimy reshape tak, aby z wektora wymiaru (1, n_patches**2) otrzymać macierz wymiaru (n_patches, n_patches), po czym wykonujemy upsampling, aby dostać macierz jednakowego rozmiaru jak każdy z kanałów wejściowego obrazka. Możemy teraz wykorzystać uzyskaną macierz np. do przyciemnienia tych obszarów obrazka, które nie są ważne. Poniżej zamieściłem pełną implementację opisanej metody oraz wyniki dla przykładowych obrazków zamieszczonych wyżej.

```
def attention_rollout(As: list[Tensor], i: int = 0) -> Tensor:
    assert 0 <= i < len(As)
    R = As[i]
    for A in As[i:]:
        R = R @ (A + torch.eye(A.shape[1]))
    return R

As = [layer.atten_blk.attn_output_weights.detach() for layer in vit.transformer_layers]
R = attention_rollout(As)
R = R[:, 0, 1:].reshape(-1, 1, 8, 8)
R = F.interpolate(R, (32, 32), mode="bicubic")

m, _ = R.min(0)
M, _ = R.max(0)
R = (R - m) / (M - m)
R = R.repeat((1, 3, 1, 1), R)

imgs_with_attn = R * test_imgs
show(imgs_with_attn, figsize=(10, 10))
```



Jak widać metoda ta działa sensownie, tj. dla poprawnie zaklasyfikowanych obrazków najważniejsze są te rzeczywiście istotne elementy obrazka. Widzimy, że samolot, czy ptak na środkowym obrazku zostały elegancko wycięte z nieistotnego tła, a w przypadku samochodu istotnym elementem są koła. Natomiast dla obrazków niepoprawnie zaklasyfikowanych class token w ogólne nie bierze pod uwagę tych istotnych elementów.