Metody Rozpoznawania Obrazów
Dominik Szot
Zadanie 02

Tematem loboratorium jest zapoznanie się z technikami klasyfikacji obrazów sieciami konwolucyjnymi oraz poszukiwanie najlepszych ich konfiguracji. Zbiorem używanym do wykonania laboratorium jest CIFAR-10

1. Wszystko zaczyna I kończy się na danych

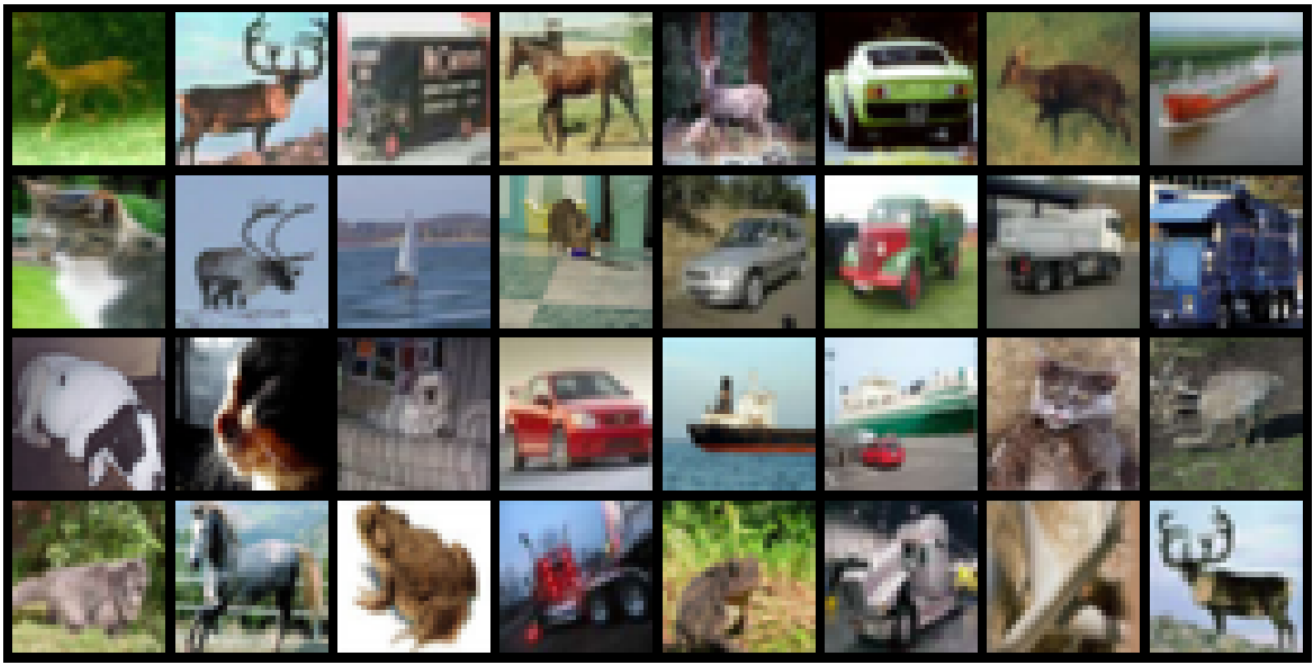
Pierwszym krokiem w celu wykonania laboratorium będzie zaimportowanie zbioru danych, oraz podziale na zbiór testowy oraz treningowy. Poniższy kod ładuje zbiór CIFAR10, który już wcześniej został pobrany.

```
import torch
import torchvision
import torchvision.transforms as transforms

batch_size = 32
transform = transforms.Compose([transforms.ToTensor()])

testset = torchvision.datasets.CIFAR10(
        root="./data", train=False, download=False, transform=transform
)

trainset = torchvision.datasets.CIFAR10(
        root="./data", train=True, download=False, transform=transform
)

test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
```

Zbiór podzielony został na 50 000 przykładów treningowych oraz 10 000 testowych.

```
Train dataset size: 50000
Test dataset size: 10000
```

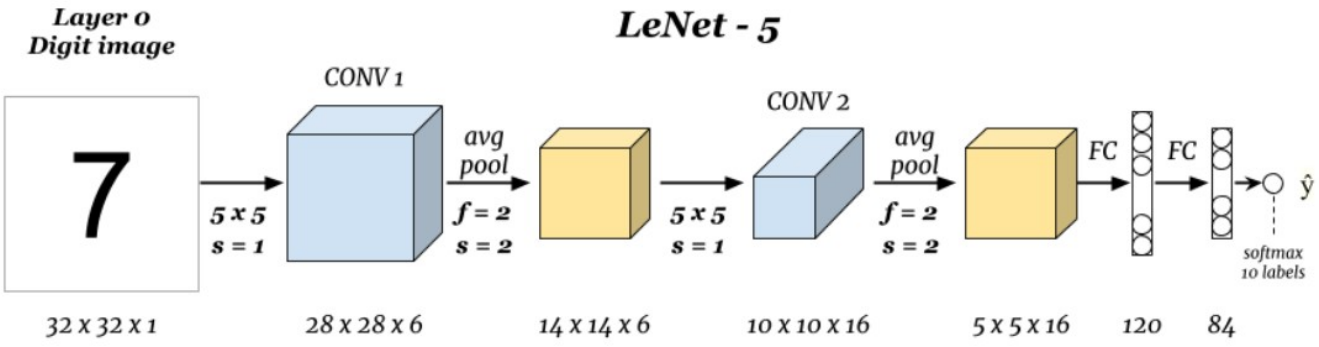
Możemy więc teraz wyświetlić kilka przykładowych obrazów wraz z ich etykietami.



deer	deer	truck	horse	deer	automobile	deer	ship
cat	deer	ship	cat	automobile	truck	truck	truck
dog	cat	dog	automobile	ship	ship	cat	bird
cat	horse	frog	truck	frog	truck	airplane	deer

2. Projektujemy architekturę sieci neuronowej

Modelem pierwszego wyboru będzie klasyczny LeNet – 5, który pojawił się w opisie laboratorium. Jako funkcja aktywacji wybrano ReLU.

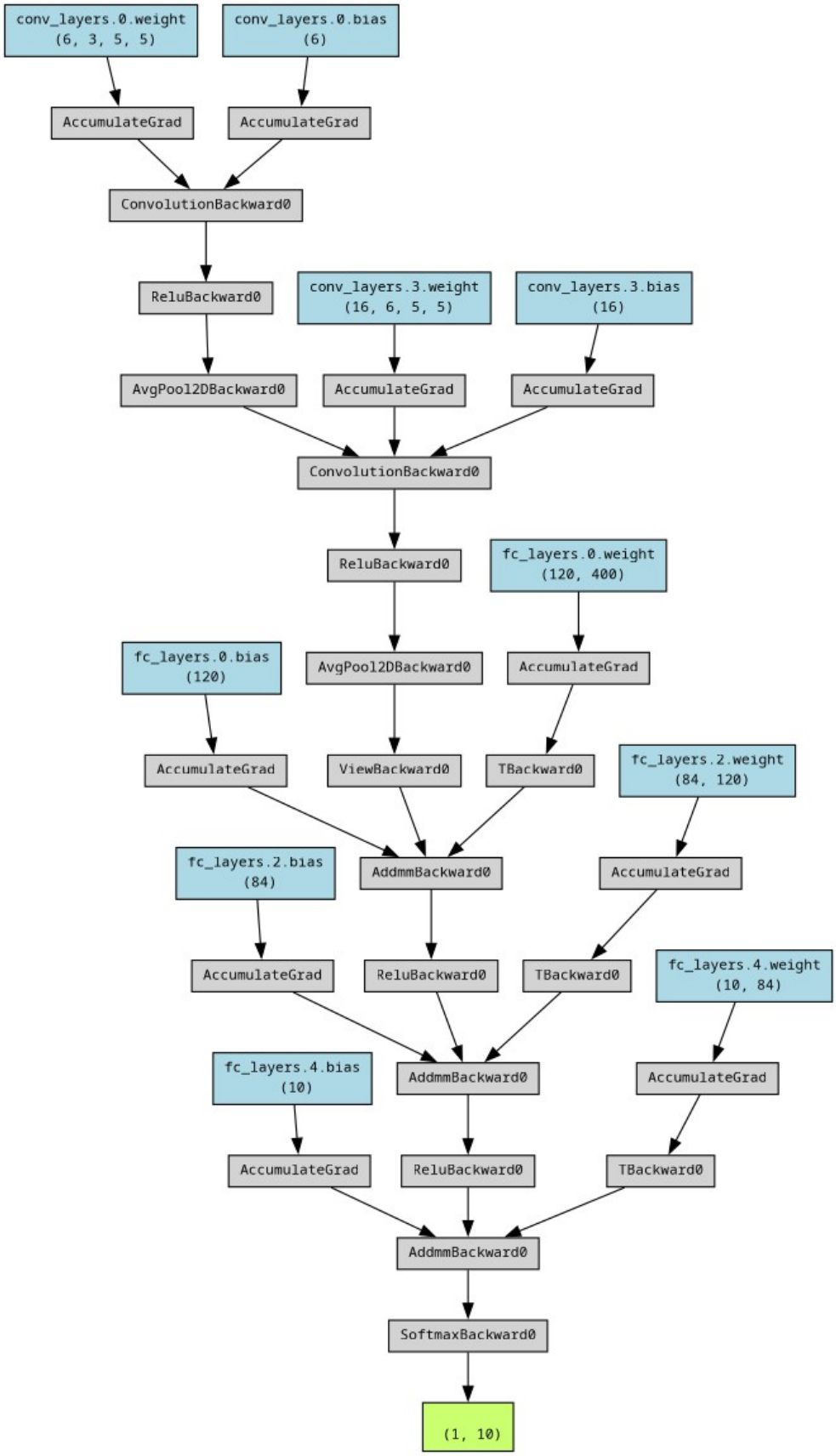


```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LeNet5(nn.Module):
        def __init__(self):
                super().__init__()

                self.conv_layers = nn.Sequential(
                        nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1),
                        nn.ReLU(),
                        nn.AvgPool2d(kernel_size=2, stride=2),
                        nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
                        nn.ReLU(),
                        nn.AvgPool2d(kernel_size=2, stride=2)
                )
                self.fc_layers = nn.Sequential(
                        nn.Linear(in_features=16 * 5 * 5, out_features=120),
                        nn.ReLU(),
                        nn.Linear(in_features=120, out_features=84),
                        nn.ReLU(),
                        nn.Linear(in_features=84, out_features=10)
                )

        def forward(self, x):
                x = self.conv_layers(x)
                x = x.view(-1, 16 * 5 * 5)
                x = self.fc_layers(x)
                x = F.softmax(x, dim=1)
                return x

model = LeNet5()
print(model)
```

Aby zwizualizować model można użyć zwykłego printa...

```
LeNet5(
  (conv_layers): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (4): ReLU()
    (5): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (fc_layers): Sequential(
    (0): Linear(in_features=400, out_features=120, bias=True)
    (1): ReLU()
    (2): Linear(in_features=120, out_features=84, bias=True)
    (3): ReLU()
    (4): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

Lub też użyć narzędzi zewnętrznych, np Torchviz



W PyTorch, podczas inicjalizacji modelu, wagi są domyślnie inicjalizowane przy użyciu metody He dla warstw konwolucyjnych oraz dla warstw liniowych.

```
layer = nn.Linear(100, 64)
print("Inicjalizacja wag dla warstwy Linear:", layer.weight)
nn.init.kaiming_normal_(layer.weight, mode='fan_in', nonlinearity='relu')
```

3. Zapętlamy się w treningu

Na początku zgodnie z poleceniem określamy funkcję kosztu, w naszym przypadku jest to Entropia Krzyżowa

```
import torch.optim as optim

learning_rate = 0.001
net = LeNet5().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)
```

Kolejnym etapem jest wydzielenie ze zbioru treningowego części walidacyjnej.

```
from torch.utils.data import random_split, DataLoader

train_size = int(0.9 * len(trainset))
val_size = len(trainset) - train_size

train_subset, val_subset = random_split(trainset, [train_size, val_size])

train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
```

Przechodzimy teraz do implementacji pętli treningowej.
Trening modeli w PyTorchu jest dosyć schematyczny i najczęściej rozdziela się go na kilka bloków, dających razem **pętlę uczącą (training loop)**, powtarzaną w każdej epoce:

1. Forward pass - obliczenie predykcji sieci
2. Loss calculation
3. Backpropagation - obliczenie pochodnych oraz zerowanie gradientów
4. Optimalization - aktualizacja wag
5. Other - ewaluacja na zbiorze walidacyjnym, logging etc.

```
for epoch in range(num_epochs): Pętla ucząca powtarzana w każdej epoce
        model.train()
        running_loss = 0.0

        for inputs, labels in train_loader:          Iterujemy po danych treningowych
                outputs = net(inputs)                    Forward pass – obliczanie predykcji sieci
                loss = criterion(outputs, labels)    Loss calculation –  (CrossEntropyLoss).
                loss.backward()                      Backpropagation - obliczenie pochodnych
                optimizer.step()                     Optimalization – aktualizacja wag
                optimizer.zero_grad()                    Zerujemy gradient
                running_loss += loss.item()              Podsumowanie strat dla epoki
                epoch_loss = running_loss / len(train_loader) Obliczenie średniej straty

        print(f"Epoka [{epoch+1}/{num_epochs}], Strata treningowa: {epoch_loss:.4f}")
        model.eval()  Ewaluacja na zbiorze walidacyjnym
        val_loss = 0.0
        with torch.no_grad():
                for inputs, labels in valloader:
                        outputs = net(inputs)
                        loss = criterion(outputs, labels)
                        val_loss += loss.item()
        val_loss /= len(valloader)
```

Po skończonym treningu jesteśmy w stanie sprawdzić dokładność klasyfikacji sieci. Poniższa funkcja dokona predykcji na zbiorze testowym oraz porówna z nimi wartości rzeczywiste.

```
classes_pred = {classname: 0 for classname in classes}
classes_total = {classname: 0 for classname in classes}
total_correct = 0
total_images = 0
net.eval()

with torch.no_grad():
        for data in testloader:
                images, labels = map(lambda x: x.to(device), data)
                outputs = net(images)
                _, predictions = torch.max(outputs, 1)
                total_images += labels.size(0)
                for expected, predicted in zip(labels, predictions):
                        classes_total[classes[expected]] += 1
                        If expected == predicted:
                                classes_pred[classes[predicted]] += 1
                                total_correct += 1

overall_accuracy = total_correct / total_images if total_images > 0 else 0
print(f'Overall Accuracy for the entire test dataset: {overall_accuracy:.2%}')
```

```
Accuracy for [airplane]  : 72.20%
Accuracy for [automobile] : 68.20%
Accuracy for [bird]      : 33.20%
Accuracy for [cat]       : 33.90%
Accuracy for [deer]      : 65.30%
Accuracy for [dog]       : 56.80%
Accuracy for [frog]      : 65.20%
Accuracy for [horse]     : 65.60%
Accuracy for [ship]      : 72.70%
Accuracy for [truck]     : 59.50%
Overall Accuracy for the entire test dataset: 59.26%
```

Nasz podstawowy model LeNet-5 osiągnął dokładność 59.26% trafnych predykcji. Jest to niesamowicie przeciętny wynik – w następnych przykładach postaramy się poprawić możliwie wynik.

4. Hiperparametryzujemy sieć I jej trening.

Zacznamy od zdefiniowania podstawowych parametrów modelu / trenowania

```
LEARNING_RATE = 0.001
BATCH_SIZE = 32
NUM_EPOCHS = 10
EARLY_STOPPING_PATIENCE = 10
MAX_EPOCHS = 300

NUM_FEATURES = 28*28
NUM_CLASSES = 10

DEVICE = "cpu"
GRAYSCALE = False
```

Na zmodyfikowany LeNet – 5 składać się będą następujące modyfikacje:

- regularyzacja Dropout
- regularyzacja L2
- wiele warstw konwolucyjnych ze zmodyfikowanym rozmiarem
- zarządanie rozmiarem wejściowym

Do poprawienia naszego modelu dokonamy optymalizacji następujących hiperparametrów::

- regularyzacja Dropout
- regularyzacja L2
- liczba warstw konwolucyjnych
- współczynnik uczenia
- liczba warstw konwolucyjnych

Dodatkowo w kodzie można zauważyć stałą regulującą regularyzacje early stopping. Która teoretycznie powinna przyśpieszyć uczenie się modelu poprzez przerywanie nierokujących treningów modelu.

```
class LeNet(nn.Module):
        def __init__(self, num_classes, num_conv_layers, dropout_p):
                super(LeNet, self).__init__()

                self.num_conv_layers = num_conv_layers
                self.dropout_p = dropout_p
                self.conv_layers = nn.ModuleList()
                in_channels = 3

                self.output_size = 32
                for i in range(num_conv_layers):
                        out_channels = 2 ** (i + 3)

                        self.conv_layers.append(nn.Conv2d(
                                in_channels, out_channels,  kernel_size=3,  stride=1,  padding=1
                        ))

                        self.conv_layers.append(nn.ReLU())
                        self.conv_layers.append(nn.AvgPool2d(kernel_size=2, stride=2))
                        in_channels = out_channels

                        self.output_size = (self.output_size // 2)

                self.fc_layers = nn.Sequential(
                        nn.Linear(out_channels * self.output_size * self.output_size, 120),
                        nn.ReLU(),
                        nn.Dropout (dropout_p),
                        nn.Linear(120, 84),
                        nn.ReLU(),
                        nn.Dropout(dropout_p),
                        nn.Linear(84, num_classes)
                )

        def forward(self, x):
                for layer in self.conv_layers:
                        x = layer(x)

                x = x.view(x.size(0), -1)
                x = self.fc_layers(x)

                return x
```

Kolejnym krokiem będzie zdefiniowanie funkcji zwracającej wartość którą będziemy optymalizować, w tym przypadku będzie to accuracy.

```
import torch.nn.functional as F
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

def objective(trial):
        learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2)
        batch_size = trial.suggest_categorical('batch_size', [32, 64, 128])
        l2_strength = trial.suggest_float('l2_strength', 1e-4, 1e-2)
        dropout_prob = trial.suggest_float('dropout_prob', 0.0, 0.5)
        num_conv_layers = trial.suggest_int('num_conv_layers', 2, 5)

        model = LeNet(
                num_classes=NUM_CLASSES, num_conv_layers = num_conv_layers,
                dropout_p=dropout_prob)
        .to(device)
        loss_fn = nn.CrossEntropyLoss()
        optimizer = optim.AdamW( model.parameters(),
                lr=learning_rate, weight_decay=l2_strength
        )

        test_dataloader = torch.utils.data.DataLoader(
                testset, batch_size=batch_size, shuffle=False)

        train_dataloader = torch.utils.data.DataLoader(
                trainset, batch_size=batch_size, shuffle=True)

        best_model, val_accuracy = train_loop(
                model = model,
                optimizer = optimizer,
                loss_fn = loss_fn,
                train_dataloader = test_dataloader,
                test_dataloader = train_dataloader,
                classes = classes,
                device = device,
                max_epochs = MAX_EPOCHS,
                early_stopping_patience = EARLY_STOPPING_PATIENCE
        )

        logger.info(f'Trial {trial.number}: learning_rate={learning_rate}, batch_size={batch_size},
        'f'l2_strength={l2_strength}, dropout_prob={dropout_prob} => Accuracy={val_accuracy}')
        return val_accuracy
```

Musimy jeszcze zdefiniować funkcję trenującą, która dla danych hiperparametrów wytrenuje model oraz zwróci najlepszą wartość accuracy.

```
def train_loop(model, optimizer, loss_fn, train_dataloader, test_dataloader, classes, device, max_epochs=10,
early_stopping_patience=3):
        steps_without_improvement = 0
        best_overall_accuracy = 0
        best_model_state = None
        num_classes = len(classes)
        classes_pred = torch.zeros(num_classes, dtype=torch.int32, device=device)
        classes_total = torch.zeros(num_classes, dtype=torch.int32, device=device)

        for epoch in range(max_epochs):
                model.train()        # Trenujemy model z danymi parametrami
                for X_batch, y_batch in train_dataloader:
                        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
                        outputs = model(X_batch)
                        loss = loss_fn(outputs, y_batch)
                        loss.backward()
                        optimizer.step()
                        optimizer.zero_grad()

        [...] ewaluowanie modelu,
        overall_accuracy = total_correct / total_images
        print(f'Overall Accuracy for the entire test dataset: {overall_accuracy:.2%}')

        If overall_accuracy > best_overall_accuracy:
                best_overall_accuracy = overall_accuracy
                steps_without_improvement = 0
                best_model_state = deepcopy(model.state_dict())
        else:
                steps_without_improvement += 1
                if steps_without_improvement == early_stopping_patience:
                        model.load_state_dict(best_model_state)
                        return model, best_overall_accuracy

        return model, best_overall_accuracy
```

Posiadamy więc wszystkie potrzebne elementy aby wykonać teraz optymalizacje za pomocą Optuny.

```
study_tpe = optuna.create_study(direction='maximize',
sampler=optuna.samplers.TPESampler())
study_tpe.optimize(objective, n_trials=3, n_jobs=9)

print("Najlepsze hiperparametry (TPE):", study_tpe.best_params)
print("Najlepsza wartość metryki (TPE):", study_tpe.best_value)
```

Optuna zacznie automatycznie dobierać optymalne wartości hiperparametrów dla naszego modelu. Przeprowadzone zostanie wiele prób oceniających różne konfiguracje. W celu przeszukiwania przestrzeni hiperparametrów optuna może stosować takie techniki jak Optymalizacja Bayessowska (model probabilistyczny przestrzeni hiperparametrów z maksymalizacją funkcji celu z wykorzystaniem wcześniejszych wyników).

5. Obraz nędzy i rozpaczy

Niestety z powodu braku czasu, braku sprzętu (trenowanie na CPU) optymalizacja hiperparametrów zakończyła się niesamowitą klęską:

```
Najlepsze hiperparametry (TPE): {'learning_rate': 0.000984234326011197,
'batch_size': 128, 'l2_strength': 0.009965883914012338, 'dropout_prob':
0.3557615247281309}
Najlepsza wartość metryki (TPE): 0.51068
```

Zoptymalizowane hiperparametry nie pozwalają nowemu modelowi osiągnąć większej dokładności od naszego klasycznego LeNet-5, wytrenowanego w punkcie 3.

Z możliwych powodów takiego stanu rzeczy wyliczyć można:

- **Niewłaściwy zakres hiperparametrów**
- **Potencjalny Overfitting**
- **Błędy w kodzie**
- **Błędna architektura modelu**

W celu klasyfikacji obrazów ze zbioru testowego używać będziemy więc starego modelu z ptk. 3.