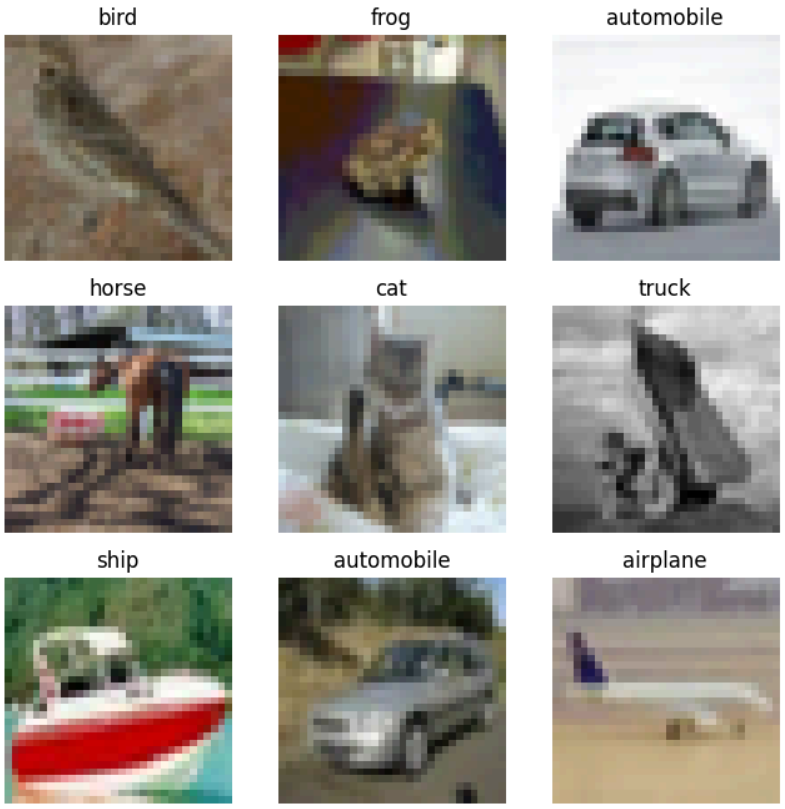
Zadanie 2.

Zadanie wykonałem w frameworku PyTorch. Aby wczytać zbiór CIFAR-10 wykorzystałem moduł `torchvision.datasets` w następujący sposób.

```
from torchvision import datasets

train_set = datasets.CIFAR10(root="./data", train=True, download=True, transform=ToTensor())
test_set = datasets.CIFAR10(root="./data", train=False, download=True, transform=ToTensor())
```

Poniżej zamieściłem kilka obrazów z tego zbioru wraz z poprawną etykietą.

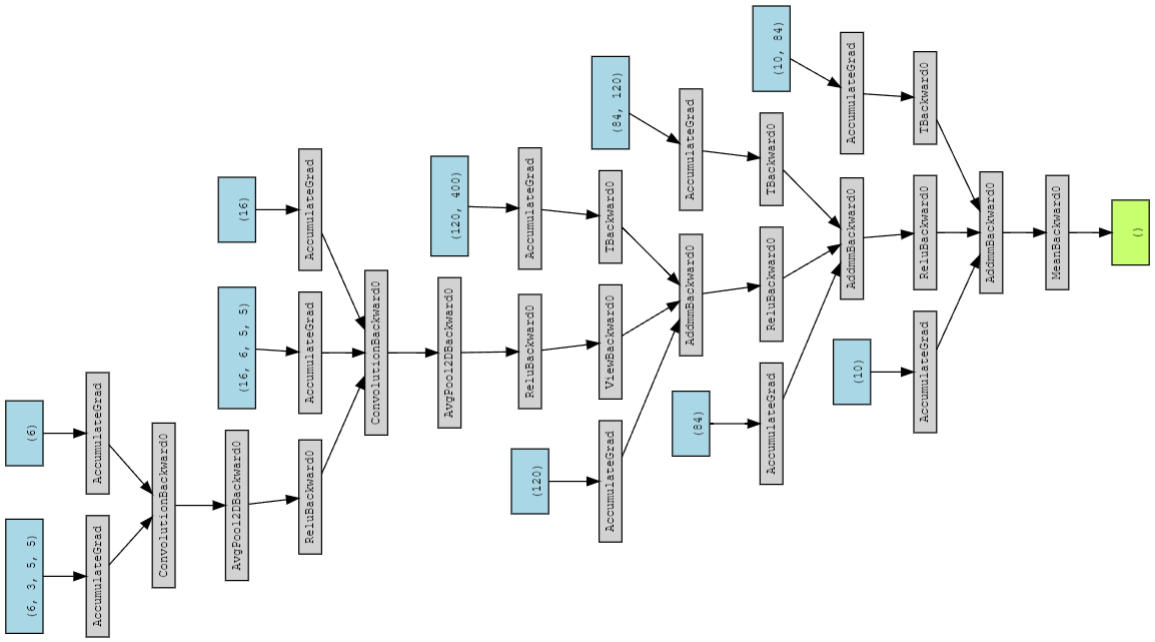
```
labels_map = { 0: "airplane", 1: "automobile", 2: "bird",  3: "cat",  4: "deer",
               5: "dog",      6: "frog",       7: "horse", 8: "ship", 9: "truck"}

# Plot some images with labels
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3

for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(train_set), size=(1,)).item()
    img, label = train_set[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.permute(1, 2, 0).squeeze(), cmap="gray")
plt.show()
```



Jako model baseline'owy zaimplementowałem opisaną w treści sieć LeNet-5. Sieć zwraca 10 wartości mlogitów.

```
class LeNet(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=(5, 5), stride=1)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=(5, 5), stride=1)

        self.avgpool1 = nn.AvgPool2d(kernel_size=(2, 2), stride=2)
        self.avgpool2 = nn.AvgPool2d(kernel_size=(2, 2), stride=2)

        self.mlp = nn.Sequential(
            nn.Linear(in_features=5 * 5 * 16, 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 = F.relu(self.avgpool1(self.conv1(X)))
        X = F.relu(self.avgpool2(self.conv2(X)))
        X = torch.flatten(X, 1)
        X = self.mlp(X)
        return X
```

Graf obliczeniowy modelu uzyskany za pomocą biblioteki pytorchviz oraz liczbę parametrów zamieściłem poniżej.




```
>>> print(sum(p.numel() for p in LeNet().parameters() if p.requires_grad))

62006
```

Ponieważ jako funkcji aktywacji użyłem funkcji ReLU(), więc odpowiednią inicjalizacją jest inicjalizacja He. Poniżej zamieściłem fragment kodu, który inicjalizuje wagi sieci.

```
def init_weights(m):
    if isinstance(m, (nn.Conv2d, nn.Linear)):
        nn.init.kaiming_normal_(m.weight, nonlinearity="relu")
        nn.init.constant_(m.bias, 0)

model = LeNet()
model.apply(init_weights)
```

Następnie zaimplementowałem funkcję `train_model()` realizującą pętlę treningową. W funkcji tej dla każdej epoki mamy dwie fazy: treningu i ewaluacji na zbiorze walidacyjnym. W obu tych fazach iterujemy po minibatchach zwróconych przez odpowiedni dataloader i w przypadku fazy treningu obliczamy wartość zdefiniowanej funkcji kosztu (w naszym przypadku `nn.CrossEntropyLoss()`), po czym obliczamy gradient po parametrach i aktualizujemy parametry zgodnie ze zdefiniowanym optymalizatorem. W fazie ewaluacji natomiast obliczamy jedynie wartości funkcji kosztu oraz accuracy i aktualizujemy najlepsze znalezione do tej pory parametry.

Poniżej zamieściłem fragment kodu, który trenuje model, po czym ewaluuje go na zbiorze testowym. Funkcja `eval_model()` zwraca wartość accuracy.

```
batch_size = 32
# Define data loaders
dataloaders = {
    "train": DataLoader(train_set, batch_size=batch_size, shuffle=True),
    "valid": DataLoader(valid_set, batch_size=batch_size, shuffle=True),
}

# Define model, loss function and optimizer (with default hyperparams)
model = LeNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters())

# Train model
train_model(model, optimizer, criterion, dataloaders, n_epochs=100)

# Evaluate model performance
test_set = datasets.CIFAR10(root="./data", train=False, download=True, transform=ToTensor())
test_dataloader = DataLoader(test_set, batch_size=batch_size, shuffle=True)
eval_model(model, test_dataloader)
```

Trening modelu bazowego dla domyślnych hiperparametrów optymalizatora AdamW i 100 epok treningu zajął 8 min 46 s na karcie graficznej NVIDIA GeForce GTX 1660. Uzyskany bazowy wynik accuracy na zbiorze testowym wyniósł 56.25%.

Aby polepszyć wynik bazowy zmieniłem architekturę sieci na ResNet-9 z warstwami typu BatchNorm w każdym bloku konwolucji oraz jedną warstwą typu Dropout między głową klasyfikującą i backbonem konwolucyjnym.

```python
class ConvGroup(nn.Module):
    def __init__(self, channels_in: int, channels_out: int, pool: bool = False):
        super().__init__()
        layers = [
            nn.Conv2d(channels_in, channels_out, kernel_size=3, padding=1),
            nn.BatchNorm2d(channels_out),
            nn.ReLU(),
        ]
        if pool:
            layers.append(nn.MaxPool2d(kernel_size=2))
        self.conv_group = nn.Sequential(*layers)

    def forward(self, X: Tensor):
        return self.conv_group(X)


class ResNet(nn.Module):
    def __init__(self, p: float = 0.5):
        super().__init__()

        self.conv_group1 = ConvGroup(3, 64)
        self.conv_group2 = ConvGroup(64, 128, pool=True)
        self.conv_group3 = ConvGroup(128, 256, pool=True)
        self.conv_group4 = ConvGroup(256, 256, pool=True)

        self.res1 = nn.Sequential(ConvGroup(128, 128), ConvGroup(128, 128))
        self.res2 = nn.Sequential(ConvGroup(256, 256), ConvGroup(256, 256))

        self.dropout = nn.Dropout(p)

        self.clf = nn.Sequential(
            nn.MaxPool2d(kernel_size=4),
            nn.Flatten(),
            nn.Linear(in_features=256, out_features=10),
        )

    def forward(self, X: Tensor):
        y = X
        y = self.conv_group1(y)
        y = self.conv_group2(y)
        y = y + self.res1(y)
        y = self.conv_group3(y)
        y = self.conv_group4(y)
        y = y + self.res2(y)
        y = self.dropout(y)
        y = self.clf(y)
        return y
```

Hiperparametry definiujące sieć oraz trening wyodrębniłem w funkcji `objective()`, która stanowi wejście do optymalizatora z biblioteki Optuna, służącej do poszukiwania parametrów. Funkcja ta sugeruje wartości hiperparametrów – prawdopodobieństwo aktywacji w warstwie Dropout, stałej uczącej, liczby epok, wielkości batcha oraz stałej regularyzacji L2, następnie trenuje model ze zdefiniowanymi hiperparametrami i jako wynik zwraca wartość accuracy na zbiorze testowym.

```python
def objective(trial: optuna.trial.BaseTrial):
    print(f"Trial: {trial.number}")
    print("-" * 10)

    # Suggest values for hyperparameters
    p = trial.suggest_float("p", 0.0, 1.0)
    lr = trial.suggest_float("lr", 1e-5, 1e-3)
    n_epochs = trial.suggest_int("n_epochs", 10, 60)
    batch_size = trial.suggest_categorical("batch_size", [32, 64, 128, 256, 512])
    weight_decay = trial.suggest_float("weight_decay", 1e-4, 1e-1)

    # ...

study = optuna.create_study(study_name="Hyperparameter search", direction="maximize")
study.optimize(objective, n_trials=12)
```

Ze względu na ograniczone zasoby obliczeniowe wykonałem 12 iteracji przeszukiwania przestrzeni hiperparametrów. Poniżej zamieściłem najlepszą konfigurację.

```
{"p": 0.5224514140060675,
 "lr": 0.000775797223876935,
 "n_epochs": 41,
 "batch_size": 64,
 "weight_decay": 0.04453415571734414}
```

Cały proces trwał około 3 godzin, natomiast ostateczny wynik accuracy na zbiorze testowym wyniósł **86.41%**. Wynik jest znacznie lepszy od baseline, jednak jest on tylko niewiele lepszy (około 1 p.p) od wyniku uzyskanego dla domyślnych hiperparametrów optymalizatora, 50 epok i stałej uczącej równej 3e-4, zatem z tej perspektywy dodatkowy koszt obliczeniowy związany z przeszukiwaniem przestrzeni hiperparametrów nie zwrócił się. Poniżej zamieściłem kilka obrazów ze zbioru testowego wraz z ich prawdziwą etykietą oraz etykietą zwróconą przez najlepszy model.