BLM4540 Görüntü İşleme - Ödev 3

Toygar Tanyel
18011094

Yöntem:

Verileri önceden inceleyerek, uygulanacak dönüşümleri hazırlamak gerekmektedir. Overfitting ve benzeri sorunları önlemek/azaltmak için verilerin eğitim öncesinde hazırlanması önemli bir adımdır.

```
stats = ([0.4914, 0.4822, 0.4465],[0.2023, 0.1994, 0.2010])
train_transform = tt.Compose([
    tt.RandomHorizontalFlip(),
    tt.ToTensor(),
    tt.Normalize(*stats)
])

test_transform = tt.Compose([
    tt.ToTensor(),
    tt.Normalize(*stats)
])
```

- “RandomHorizontalFlip”, bir görüntüyü %50 olasılıkla rastgele çevirir. Verilere gürültü eklemek ve modelimizin overfitting olmasını önlemek için bu tür dönüşümleri ekliyoruz. ColorJitter, RandomCrop ve RandomVerticalFlip gibi kullanabileceğiniz başka dönüşümler de bulunmaktadır, ancak bizim amacımız için bunu yeterli buldum.
- “ToTensor”, görüntüyü bir Tensör’e dönüştürür. Renkli bir görüntü olduğu için 3 kanala (R,G,B) sahip olacağı için Tensör 3x32x32 boyutunda olacaktır.
- “Normalize”, tüm veri kümesinin her kanalı için ortalama ve standart sapmayı girdi olarak alır. Normalleştirme, türevlerin kontrolden çıkmamasını sağlamak için verilerimizi benzer bir değer aralığına ölçeklendirir.

Verileri yüklerken Pytorch içerisindeki *"from torchvision.datasets import CIFAR10"* kütüphanesi kullanılabilir. CIFAR-10, MNIST veri seti gibi oldukça bir popüler olduğundan Pytorch ve Tensorflow gibi frameworkler tarafından erişimi kolaylaştırılmıştır.

```
train_data = CIFAR10(download=True, root="./data", transform=train_transform)
test_data = CIFAR10(root="./data", train=False, transform=test_transform)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
170499072/? [00:03<00:00, 53763026.68it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
```

Veri seti:

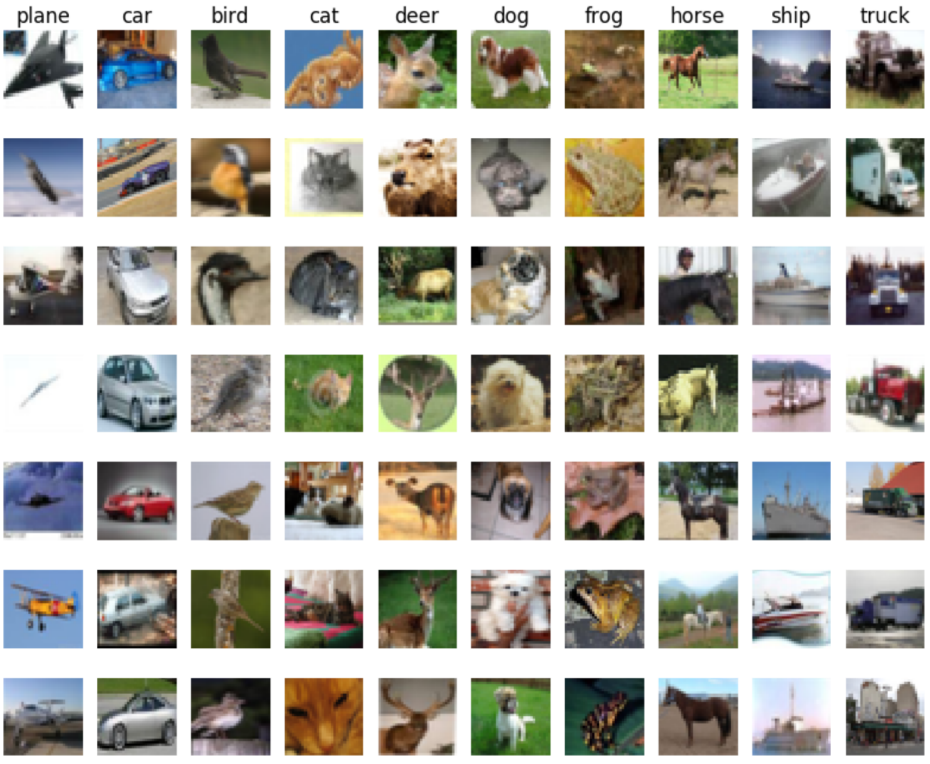
TRAIN

```
{'frog': 5000,
 'truck': 5000,
 'deer': 5000,
 'automobile': 5000,
 'bird': 5000,
 'horse': 5000,
 'ship': 5000,
 'cat': 5000,
 'dog': 5000,
 'airplane': 5000}
```

TEST

```
{'cat': 1000,
 'ship': 1000,
 'airplane': 1000,
 'frog': 1000,
 'automobile': 1000,
 'truck': 1000,
 'dog': 1000,
 'horse': 1000,
 'deer': 1000,
 'bird': 1000}
```

Örnek resimler:



Çoğu görüntü task'ı için belirli batch boyutlarında dataloader oluşturuyoruz. Bunun sebebi bütün resimlerin aynı anda memory'e sığdırılmasının çoğu zaman mümkün olmamasından kaynaklanmaktadır.

```
BATCH_SIZE = 128
train_dl = DataLoader(train_data, BATCH_SIZE, num_workers=2, pin_memory=True, shuffle=True)
test_dl = DataLoader(test_data, BATCH_SIZE, num_workers=2, pin_memory=True, shuffle=False)
```

Ağ tasarımı:

```
class BaseModel(nn.Module):
    def training_step(self,batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        return loss

    def validation_step(self,batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        acc = accuracy(out,labels)
        return {"val_loss":loss.detach(),"val_acc":acc}

    def validation_epoch_end(self,outputs):
        batch_losses = [loss["val_loss"] for loss in outputs]
        loss = torch.stack(batch_losses).mean()
        batch_accuracy = [accuracy["val_acc"] for accuracy in outputs]
        acc = torch.stack(batch_accuracy).mean()
        return {"val_loss":loss.item(),"val_acc":acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}"
            epoch, result['lrs'][-1], result['train_loss'], result['val_loss'], result['val_acc'])
```

Genellikle Pytorch'ta model oluşturulurken "inheritance" özelliği kullanılarak kurduğumuz, sonraki sayfadaki "model" buradaki "base model"e bağlanır. Bu sayede kod karmaşıklığı indirgenerek bize training sırasında lazım olacak olan step fonksiyonları eklenebilmektedir.

```
# Creating a CNN class
class CNN(BaseModel):
    #  Determine what layers and their order in CNN object
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv_layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.conv_layer2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3)
        self.max_pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        self.conv_layer3 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3)
        #self.conv_layer4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3)
        self.max_pool2 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        self.fc1 = nn.Linear(1152, 128)
        self.fc2 = nn.Linear(128, num_classes)

        self.relu1 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)


    # Progresses data across layers
    def forward(self, x):
        #CNN Layer 1
        out = self.conv_layer1(x)
        out = self.relu1(out)
        out = self.dropout(out)
        #CNN Layer 2
        out = self.conv_layer2(out)
        out = self.relu1(out)
        out = self.max_pool1(out)
        #CNN Layer 3
        out = self.conv_layer3(out)
        out = self.relu1(out)
```

```
        #out = self.conv_layer4(out)
        out = self.max_pool2(out)
        #Out layers
        out = out.reshape(out.size(0), -1)

        out = self.dropout(out)
        out = self.fc1(out)
        out = self.relu1(out)
        out = self.fc2(out)
        return out
```

Class sayısı parametre olarak verilerek bir model tanımlayabiliriz. Tanımlama esnasında “forward pass” için gerekli olacak tüm katmanlar 1 defa “init” içerisinde önceliklendirilir. Tercih ettiğim parametre ve model tanımlaması aşağıdaki gibidir:

```
epochs = 20
optimizer = torch.optim.Adam
max_lr = 1e-3
grad_clip = 0.1
weight_decay = 1e-5
scheduler = torch.optim.lr_scheduler.OneCycleLR
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
num_classes = len(classes)
```

```
model = CNN(num_classes)
criterion = nn.CrossEntropyLoss()
total_step = len(train_dl)
model = to_device(model, device)
```

"max_lr", öğrenme oranı (learning rate) planlayıcısı (scheduler) için belirlediğim maksimum öğrenme oranıdır. Öğrenme oranı planlayıcısı için, öğrenme oranını düşük bir öğrenme oranına ayarlayan, kademeli olarak maksimum öğrenme oranına yükselten ve ardından tekrar düşük bir öğrenme oranına geri dönen OneCycleLR kullandım.

"grad_clip", geçişlerin çok büyük olmasını engeller.

"weight_decay" esas olarak modeli basitleştirmeye çalışır ve modelin daha iyi genelleştirilmesine yardımcı olur.

Evaluation & Fit:

```
@torch.no_grad()
def evaluate(model,test_dl):
    model.eval()
    outputs = [model.validation_step(batch) for batch in test_dl]
    return model.validation_epoch_end(outputs)
```

```
def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']
```

Pytorch, işlemleri TensorFlow'un aksine biraz daha kullanıcıya bırakmaktadır. Dolayısıyla eğitim için kullanılacak olan .fit fonksiyonunu custom yazdım.

```
def fit (epochs, train_dl, test_dl, model, optimizer, max_lr, weight_decay, scheduler, grad_clip=N
    torch.cuda.empty_cache()

    history = []

    optimizer = optimizer(model.parameters(), max_lr, weight_decay = weight_decay)

    scheduler = scheduler(optimizer, max_lr, epochs=epochs, steps_per_epoch=len(train_dl))

    for epoch in range(epochs):
        model.train()

        train_loss = []
        lrs = []

        for batch in train_dl:
            loss = model.training_step(batch)

            train_loss.append(loss)

            loss.backward()

            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
            optimizer.zero_grad()

            scheduler.step()
            lrs.append(get_lr(optimizer))
        result = evaluate(model, test_dl)
        result["train_loss"] = torch.stack(train_loss).mean().item()
        result["lrs"] = lrs
```

```
        model.epoch_end(epoch,result)
        history.append(result)

    return history
```

Örnek Eğitim (3 conv, 3 kernel_size, 32 filter, 128 batch_size):

```
%%time
history = fit(epochs=epochs, train_dl=train_dl, test_dl=test_dl, model=model,
              optimizer=optimizer, max_lr=max_lr, grad_clip=grad_clip,
              weight_decay=weight_decay, scheduler=scheduler)
```

```
Epoch [0], last_lr: 0.00010, train_loss: 2.1217, val_loss: 1.8519, val_acc: 0.3517
Epoch [1], last_lr: 0.00028, train_loss: 1.7292, val_loss: 1.5366, val_acc: 0.4548
Epoch [2], last_lr: 0.00052, train_loss: 1.5168, val_loss: 1.3711, val_acc: 0.5176
Epoch [3], last_lr: 0.00076, train_loss: 1.3726, val_loss: 1.2049, val_acc: 0.5749
Epoch [4], last_lr: 0.00094, train_loss: 1.2559, val_loss: 1.1236, val_acc: 0.6013
Epoch [5], last_lr: 0.00100, train_loss: 1.1453, val_loss: 1.0195, val_acc: 0.6408
Epoch [6], last_lr: 0.00099, train_loss: 1.0762, val_loss: 0.9352, val_acc: 0.6743
Epoch [7], last_lr: 0.00095, train_loss: 1.0077, val_loss: 0.8889, val_acc: 0.6893
Epoch [8], last_lr: 0.00089, train_loss: 0.9528, val_loss: 0.8345, val_acc: 0.7107
Epoch [9], last_lr: 0.00081, train_loss: 0.9084, val_loss: 0.8159, val_acc: 0.7177
Epoch [10], last_lr: 0.00072, train_loss: 0.8763, val_loss: 0.7729, val_acc: 0.7330
Epoch [11], last_lr: 0.00061, train_loss: 0.8496, val_loss: 0.7648, val_acc: 0.7396
Epoch [12], last_lr: 0.00050, train_loss: 0.8273, val_loss: 0.7434, val_acc: 0.7446
Epoch [13], last_lr: 0.00039, train_loss: 0.7986, val_loss: 0.7242, val_acc: 0.7501
Epoch [14], last_lr: 0.00028, train_loss: 0.7807, val_loss: 0.7048, val_acc: 0.7592
Epoch [15], last_lr: 0.00019, train_loss: 0.7596, val_loss: 0.7093, val_acc: 0.7557
Epoch [16], last_lr: 0.00011, train_loss: 0.7479, val_loss: 0.6948, val_acc: 0.7634
Epoch [17], last_lr: 0.00005, train_loss: 0.7400, val_loss: 0.6894, val_acc: 0.7630
Epoch [18], last_lr: 0.00001, train_loss: 0.7285, val_loss: 0.6841, val_acc: 0.7653
Epoch [19], last_lr: 0.00000, train_loss: 0.7267, val_loss: 0.6846, val_acc: 0.7644
CPU times: user 59.8 s, sys: 11.8 s, total: 1min 11s
Wall time: 5min 38s
```

Çok basit bir model tasarlamış olmamıza rağmen %76'ya kadar çıkabiliyor olmamız yeterli bir başarım olarak değerlendirilebilir.

Her sınıfın kendi içindeki doğruluk oranı:

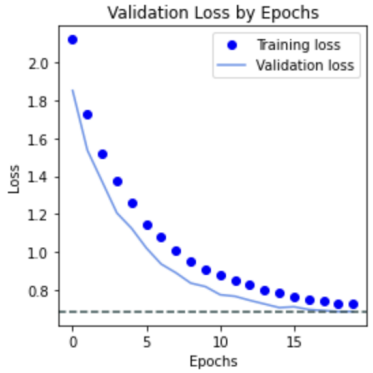
```
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

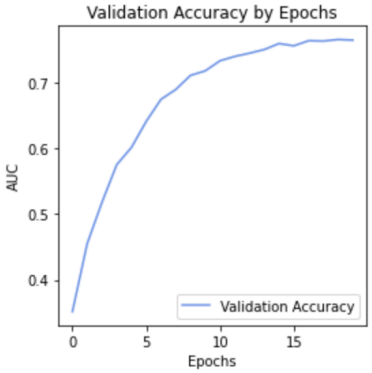
```
Accuracy for class: plane is 81.5 %
Accuracy for class: car   is 90.4 %
Accuracy for class: bird  is 62.5 %
Accuracy for class: cat   is 56.0 %
Accuracy for class: deer  is 72.3 %
Accuracy for class: dog   is 69.3 %
Accuracy for class: frog  is 82.0 %
Accuracy for class: horse is 83.1 %
Accuracy for class: ship  is 84.4 %
Accuracy for class: truck is 83.8 %
```

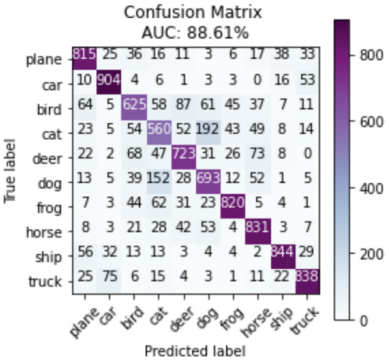
Tasarladığım “plot_training_metrics” fonksiyonu ile gerekli olan tüm bilgileri ekrana birlikte gösteriyorum:

```
Model AUC 88.61%, Accuracy 76.53% on Test Data
```

	precision	recall	f1-score	support
plane	0.78	0.81	0.80	1000
car	0.85	0.90	0.88	1000
bird	0.69	0.62	0.65	1000
cat	0.59	0.56	0.57	1000
deer	0.74	0.72	0.73	1000
dog	0.65	0.69	0.67	1000
frog	0.85	0.82	0.84	1000
horse	0.77	0.83	0.80	1000
ship	0.89	0.84	0.87	1000
truck	0.85	0.84	0.84	1000
accuracy			0.77	10000
macro avg	0.76	0.77	0.76	10000
weighted avg	0.76	0.77	0.76	10000





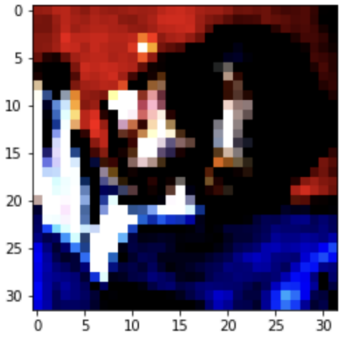


Metrik olarak CIFAR-10 gibi veri setlerinde accuracy’i rahatlıkla tercih edebiliyoruz. Bunun ana sebebi hem train hem test setinde tüm sınıflardan eşit sayıda olmasıdır.

```python
def predict_image(img, model):
    xb = to_device(img.unsqueeze(0), device)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return test_data.classes[preds[0].item()]
```

```python
img, label = test_data[0]
plt.imshow(img.permute(1, 2, 0))
print('Label:', test_data.classes[label], ', Predicted:', predict_image(img, model))
```

```
Label: cat , Predicted: cat
```



Herhangi bir resim için tahminleme yapabiliriz.

Uygulama:

İ. Elinizdeki her bir sınıf için (toplamda 10 adet sınıf var) 3'er tane *rasgele* resim seçiniz. Sorgu olarak kullandığınız resim ile en yüksek benzerliğe sahip 5 adet sınıfı bir tablo biçiminde paylaşınız.

-> Tablo gibi yapınca sığdırması zordu, daha okunaklı olduğu için sonuçları tablosuz sunuyorum.

Airplane:

```
Selected Query Image -> label_idx: 0, label: airplane
TOP 5 LABELS:
label_idx: 0, label: airplane
label_idx: 2, label: bird
label_idx: 3, label: cat
label_idx: 5, label: dog
label_idx: 4, label: deer
```

```
Selected Query Image -> label_idx: 0, label: airplane
TOP 5 LABELS:
label_idx: 0, label: airplane
label_idx: 8, label: ship
label_idx: 3, label: cat
label_idx: 2, label: bird
label_idx: 1, label: automobile
```

```
Selected Query Image -> label_idx: 0, label: airplane
TOP 5 LABELS:
label_idx: 0, label: airplane
label_idx: 8, label: ship
label_idx: 1, label: automobile
label_idx: 9, label: truck
label_idx: 4, label: deer
```

Automobile:

```
Selected Query Image -> label_idx: 1, label: automobile
TOP 5 LABELS:
label_idx: 1, label: automobile
label_idx: 9, label: truck
label_idx: 3, label: cat
label_idx: 8, label: ship
label_idx: 6, label: frog
```

```
Selected Query Image -> label_idx: 1, label: automobile
TOP 5 LABELS:
label_idx: 1, label: automobile
label_idx: 9, label: truck
label_idx: 8, label: ship
label_idx: 0, label: airplane
label_idx: 3, label: cat
```

```
Selected Query Image -> label_idx: 1, label: automobile
TOP 5 LABELS:
label_idx: 1, label: automobile
label_idx: 9, label: truck
label_idx: 8, label: ship
label_idx: 3, label: cat
label_idx: 6, label: frog
```

Bird:

```
Selected Query Image -> label_idx: 2, label: bird
TOP 5 LABELS:
label_idx: 2, label: bird
label_idx: 7, label: horse
label_idx: 3, label: cat
label_idx: 0, label: airplane
label_idx: 5, label: dog
```

```
Selected Query Image -> label_idx: 2, label: bird
TOP 5 LABELS:
label_idx: 2, label: bird
label_idx: 4, label: deer
label_idx: 7, label: horse
label_idx: 6, label: frog
label_idx: 3, label: cat
```

```
Selected Query Image -> label_idx: 2, label: bird
TOP 5 LABELS:
label_idx: 2, label: bird
label_idx: 6, label: frog
label_idx: 3, label: cat
label_idx: 4, label: deer
label_idx: 0, label: airplane
```

Cat:

```
Selected Query Image -> label_idx: 3, label: cat
TOP 5 LABELS:
label_idx: 3, label: cat
label_idx: 7, label: horse
label_idx: 4, label: deer
label_idx: 5, label: dog
label_idx: 2, label: bird
```

```
Selected Query Image -> label_idx: 3, label: cat
TOP 5 LABELS:
label_idx: 3, label: cat
label_idx: 5, label: dog
label_idx: 4, label: deer
label_idx: 2, label: bird
label_idx: 0, label: airplane
```

```
Selected Query Image -> label_idx: 3, label: cat
TOP 5 LABELS:
label_idx: 3, label: cat
label_idx: 5, label: dog
label_idx: 4, label: deer
label_idx: 2, label: bird
label_idx: 6, label: frog
```

Deer:

```
Selected Query Image -> label_idx: 4, label: deer
TOP 5 LABELS:
label_idx: 4, label: deer
label_idx: 2, label: bird
label_idx: 3, label: cat
label_idx: 6, label: frog
label_idx: 5, label: dog
```

```
Selected Query Image -> label_idx: 4, label: deer
TOP 5 LABELS:
label_idx: 4, label: deer
label_idx: 5, label: dog
label_idx: 3, label: cat
label_idx: 2, label: bird
label_idx: 7, label: horse
```

```
Selected Query Image -> label_idx: 4, label: deer
TOP 5 LABELS:
label_idx: 4, label: deer
label_idx: 2, label: bird
label_idx: 0, label: airplane
label_idx: 5, label: dog
label_idx: 3, label: cat
```

Dog:

```
Selected Query Image -> label_idx: 5, label: dog
TOP 5 LABELS:
label_idx: 5, label: dog
label_idx: 7, label: horse
label_idx: 3, label: cat
label_idx: 4, label: deer
label_idx: 2, label: bird
```

```
Selected Query Image -> label_idx: 5, label: dog
TOP 5 LABELS:
label_idx: 5, label: dog
label_idx: 3, label: cat
label_idx: 4, label: deer
label_idx: 7, label: horse
label_idx: 6, label: frog
```

```
Selected Query Image -> label_idx: 5, label: dog
TOP 5 LABELS:
label_idx: 5, label: dog
label_idx: 7, label: horse
label_idx: 3, label: cat
label_idx: 0, label: airplane
label_idx: 2, label: bird
```

Frog:

```
Selected Query Image -> label_idx: 6, label: frog
TOP 5 LABELS:
label_idx: 6, label: frog
label_idx: 3, label: cat
label_idx: 2, label: bird
label_idx: 1, label: automobile
label_idx: 0, label: airplane
```

```
Selected Query Image -> label_idx: 6, label: frog
TOP 5 LABELS:
label_idx: 6, label: frog
label_idx: 3, label: cat
label_idx: 2, label: bird
label_idx: 5, label: dog
label_idx: 1, label: automobile
```

```
Selected Query Image -> label_idx: 6, label: frog
TOP 5 LABELS:
label_idx: 6, label: frog
label_idx: 5, label: dog
label_idx: 2, label: bird
label_idx: 3, label: cat
label_idx: 1, label: automobile
```

Horse:

```
Selected Query Image -> label_idx: 7, label: horse
TOP 5 LABELS:
label_idx: 7, label: horse
label_idx: 5, label: dog
label_idx: 4, label: deer
label_idx: 3, label: cat
label_idx: 9, label: truck
```

```
Selected Query Image -> label_idx: 7, label: horse
TOP 5 LABELS:
label_idx: 7, label: horse
label_idx: 5, label: dog
label_idx: 3, label: cat
label_idx: 2, label: bird
label_idx: 4, label: deer
```

```
Selected Query Image -> label_idx: 7, label: horse
TOP 5 LABELS:
label_idx: 5, label: dog
label_idx: 8, label: ship
label_idx: 2, label: bird
label_idx: 7, label: horse
label_idx: 3, label: cat
```

Ship:

```
Selected Query Image -> label_idx: 8, label: ship
TOP 5 LABELS:
label_idx: 8, label: ship
label_idx: 2, label: bird
label_idx: 0, label: airplane
label_idx: 6, label: frog
label_idx: 4, label: deer
```

```
Selected Query Image -> label_idx: 8, label: ship
TOP 5 LABELS:
label_idx: 8, label: ship
label_idx: 0, label: airplane
label_idx: 2, label: bird
label_idx: 6, label: frog
label_idx: 1, label: automobile
```

```
Selected Query Image -> label_idx: 8, label: ship
TOP 5 LABELS:
label_idx: 8, label: ship
label_idx: 0, label: airplane
label_idx: 9, label: truck
label_idx: 2, label: bird
label_idx: 3, label: cat
```

Truck:

```
Selected Query Image -> label_idx: 9, label: truck
TOP 5 LABELS:
label_idx: 1, label: automobile
label_idx: 9, label: truck
label_idx: 8, label: ship
label_idx: 0, label: airplane
label_idx: 3, label: cat
```

```
Selected Query Image -> label_idx: 9, label: truck
TOP 5 LABELS:
label_idx: 9, label: truck
label_idx: 8, label: ship
label_idx: 1, label: automobile
label_idx: 0, label: airplane
label_idx: 2, label: bird
```

```
Selected Query Image -> label_idx: 9, label: truck
TOP 5 LABELS:
label_idx: 9, label: truck
label_idx: 1, label: automobile
label_idx: 2, label: bird
label_idx: 8, label: ship
label_idx: 3, label: cat
```

ii. Denediğiniz bütün test resimleri için karışıklık matrisini veriniz.

-> Tüm test sonuçları: Sonuçlar bölümünde verilmiştir.

Sonuçlar:

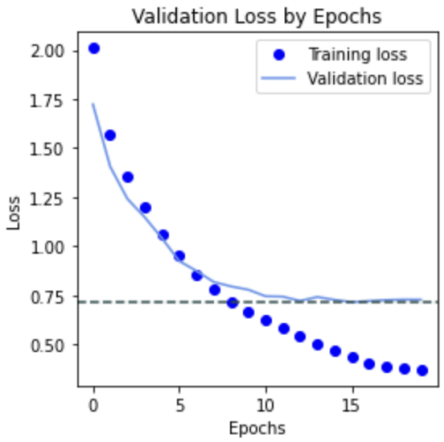
Modeller 15-20 epochs arasında doygunluğa ulaştığı (ortalama 15-20 epoch'tan itibaren validation skoru sabit ya da düşmeye başlıyor) için karşılaştırmayı sabit 20 epochs çalıştırma üzerinden yaptım.

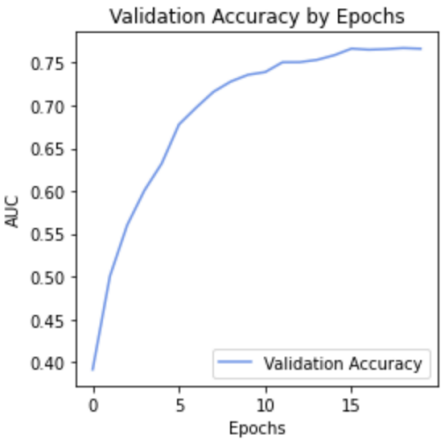
Sabit hiperparametreler:

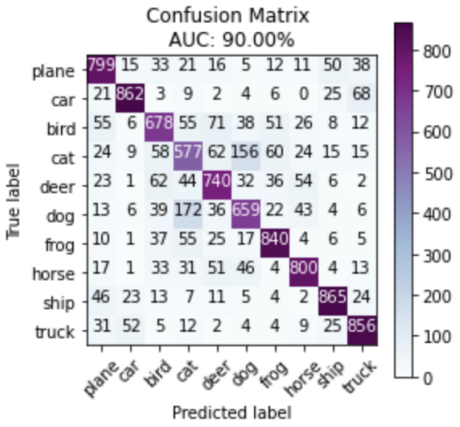
```
epochs = 20
optimizer = torch.optim.Adam
max_lr = 1e-3
grad_clip = 0.1
weight_decay = 1e-5
scheduler = torch.optim.lr_scheduler.OneCycleLR
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
num_classes = len(classes)
```

1) (3 conv, 3 kernel_size, 32 filter, 128 batch_size, dropout yok)

```
Model AUC 90.00%, Accuracy 76.76% on Test Data

              precision    recall  f1-score   support

       plane       0.77      0.80      0.78      1000
         car       0.88      0.86      0.87      1000
        bird       0.71      0.68      0.69      1000
         cat       0.59      0.58      0.58      1000
        deer       0.73      0.74      0.73      1000
         dog       0.68      0.66      0.67      1000
        frog       0.81      0.84      0.82      1000
       horse       0.82      0.80      0.81      1000
        ship       0.86      0.86      0.86      1000
       truck       0.82      0.86      0.84      1000

    accuracy                           0.77     10000
   macro avg       0.77      0.77      0.77     10000
weighted avg       0.77      0.77      0.77     10000
```



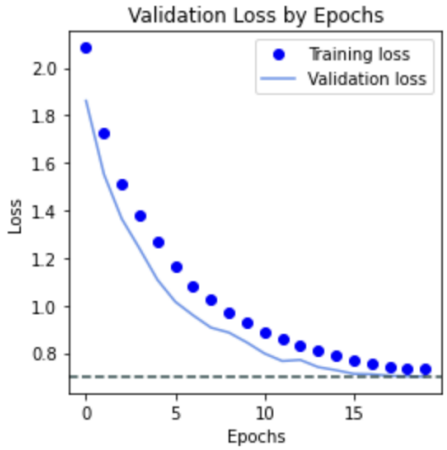


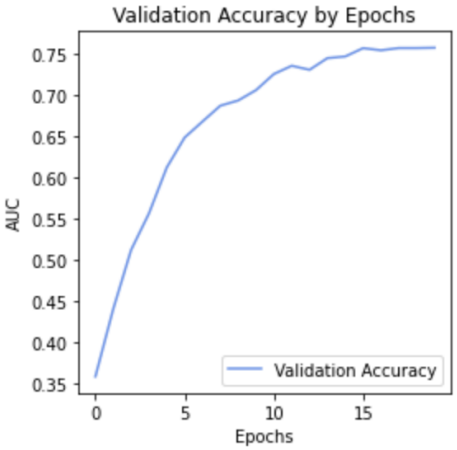


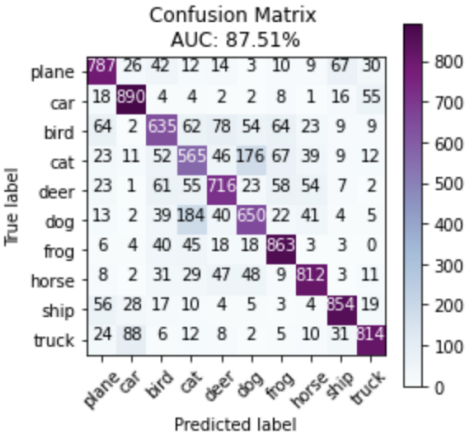
Model AUC 87.51%, Accuracy 75.86% on Test Data

	precision	recall	f1-score	support
plane	0.77	0.79	0.78	1000
car	0.84	0.89	0.87	1000
bird	0.69	0.64	0.66	1000
cat	0.58	0.56	0.57	1000
deer	0.74	0.72	0.73	1000
dog	0.66	0.65	0.66	1000
frog	0.78	0.86	0.82	1000
horse	0.82	0.81	0.81	1000
ship	0.85	0.85	0.85	1000
truck	0.85	0.81	0.83	1000
accuracy			0.76	10000
macro avg	0.76	0.76	0.76	10000
weighted avg	0.76	0.76	0.76	10000

3 conv,
3 kernel_size,
32 filter,
128 batch_size,
dropout var



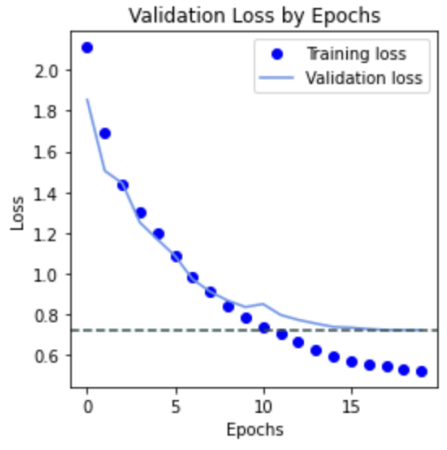


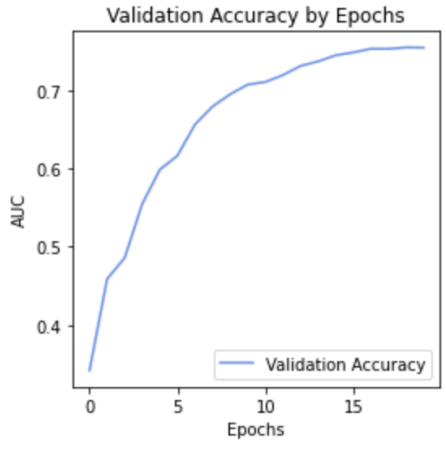


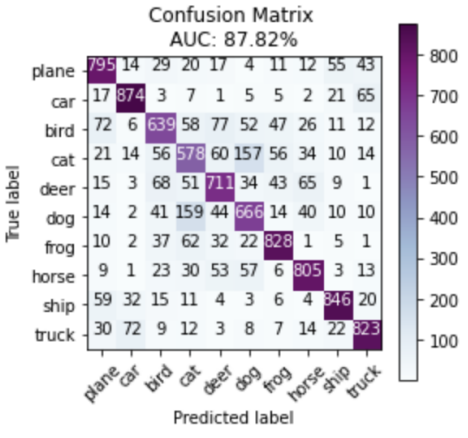
Model AUC 87.82%, Accuracy 75.65% on Test Data

	precision	recall	f1-score	support
plane	0.76	0.80	0.78	1000
car	0.86	0.87	0.87	1000
bird	0.69	0.64	0.67	1000
cat	0.59	0.58	0.58	1000
deer	0.71	0.71	0.71	1000
dog	0.66	0.67	0.66	1000
frog	0.81	0.83	0.82	1000
horse	0.80	0.81	0.80	1000
ship	0.85	0.85	0.85	1000
truck	0.82	0.82	0.82	1000
accuracy			0.76	10000
macro avg	0.76	0.76	0.76	10000
weighted avg	0.76	0.76	0.76	10000

3 conv,
3 kernel_size,
32 filter,
256 batch_size,
dropout yok



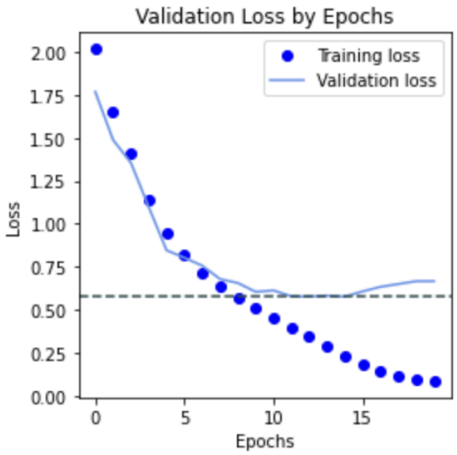


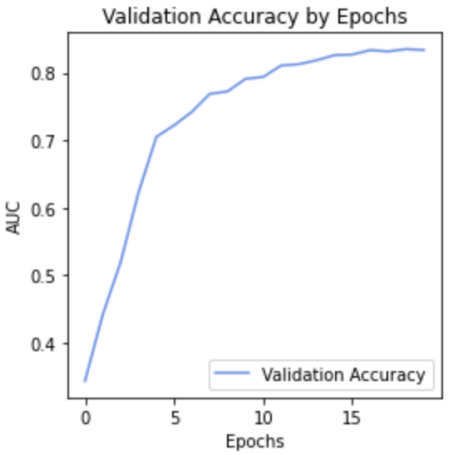


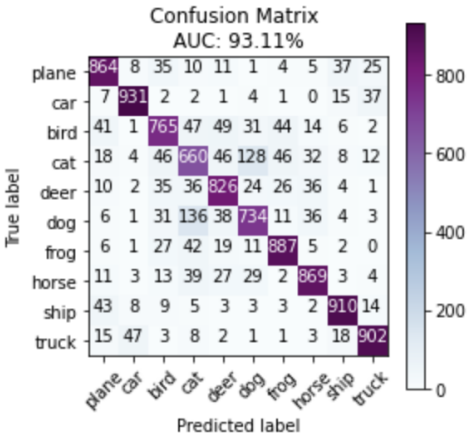
Model AUC 93.11%, Accuracy 83.48% on Test Data

	precision	recall	f1-score	support
plane	0.85	0.86	0.86	1000
car	0.93	0.93	0.93	1000
bird	0.79	0.77	0.78	1000
cat	0.67	0.66	0.66	1000
deer	0.81	0.83	0.82	1000
dog	0.76	0.73	0.75	1000
frog	0.87	0.89	0.88	1000
horse	0.87	0.87	0.87	1000
ship	0.90	0.91	0.91	1000
truck	0.90	0.90	0.90	1000
accuracy			0.83	10000
macro avg	0.83	0.83	0.83	10000
weighted avg	0.83	0.83	0.83	10000

5 conv,
5 kernel_size,
64 filter,
128 batch_size,
dropout yok



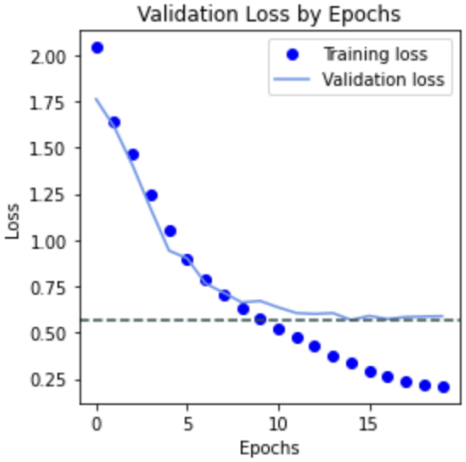


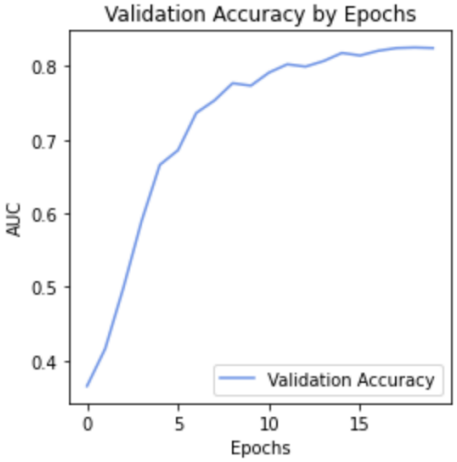


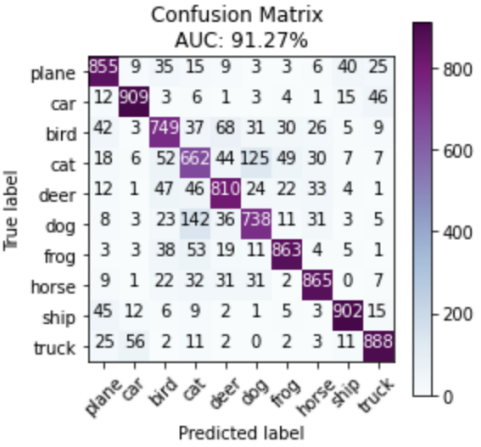
Model AUC 91.27%, Accuracy 82.41% on Test Data

	precision	recall	f1-score	support
plane	0.83	0.85	0.84	1000
car	0.91	0.91	0.91	1000
bird	0.77	0.75	0.76	1000
cat	0.65	0.66	0.66	1000
deer	0.79	0.81	0.80	1000
dog	0.76	0.74	0.75	1000
frog	0.87	0.86	0.87	1000
horse	0.86	0.86	0.86	1000
ship	0.91	0.90	0.91	1000
truck	0.88	0.89	0.89	1000
accuracy			0.82	10000
macro avg	0.82	0.82	0.82	10000
weighted avg	0.82	0.82	0.82	10000

5 conv,
3 kernel_size,
64 filter,
128 batch_size,
dropout yok

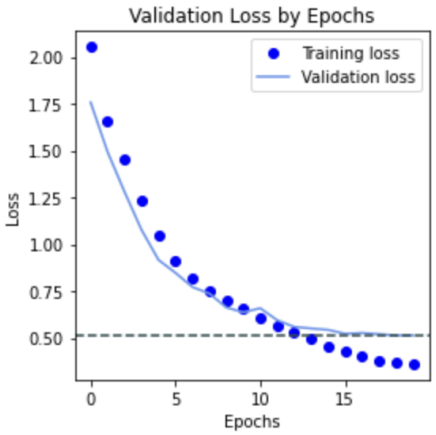


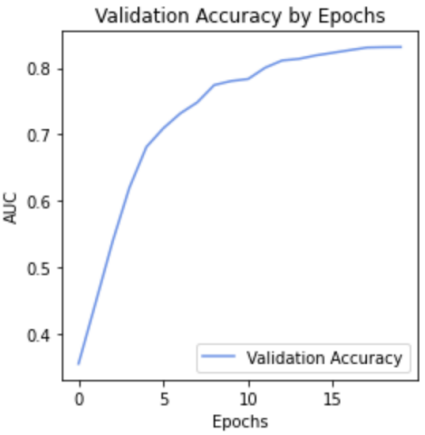


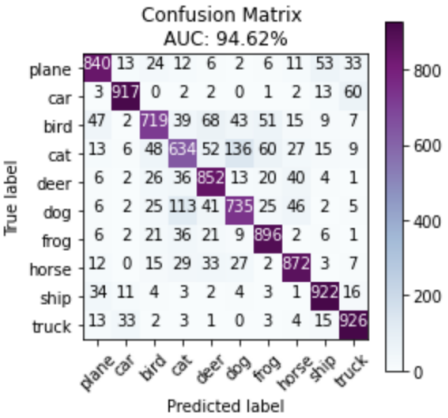


```
Model AUC 94.62%, Accuracy 83.13% on Test Data
              precision    recall  f1-score   support

       plane       0.86      0.84      0.85      1000
         car       0.93      0.92      0.92      1000
        bird       0.81      0.72      0.76      1000
         cat       0.70      0.63      0.66      1000
        deer       0.79      0.85      0.82      1000
         dog       0.76      0.73      0.75      1000
        frog       0.84      0.90      0.87      1000
       horse       0.85      0.87      0.86      1000
        ship       0.88      0.92      0.90      1000
       truck       0.87      0.93      0.90      1000

    accuracy                           0.83     10000
   macro avg       0.83      0.83      0.83     10000
weighted avg       0.83      0.83      0.83     10000
```

5 conv,
3 kernel_size,
64 filter,
128 batch_size,
dropout var







Yukarıdaki sonuçları özetlemek gerekirse, CNN ağlarında katman sayısı arttıkça model performansı da artar. Bizim örneğimizde filtre boyutu 5 için daha iyi sonuçlar vermiştir. Aynı zamanda dropout eklemek hem 3 conv2d hem de 5 conv2d katmanı içeren modellerde genel performansı düşürmüştür. En iyi sonucu "5 conv, 5 kernel_size, 64 filter, 128 batch_size, dropout yok" sağlamıştır. (93.11% AUC ve 83.48% Accuracy) Ayrıca batch_size artışının olumsuz etkisi 1. ve 3. testlere bakılarak yorumlanabilir (128 vs 256).

Genel yapıya baktığımızda en zor ayrımı yapılan sınıflar kedi, kuş ve köpek olmakla birlikte en kolay ayrım araba, gemi ve kamyon için yapılır. Yukarıdaki sonuçlarda "recall" metriğine bakarak ilgili sınıfın ne kadar doğru ayrılabildiğini yorumlayabiliriz. Aynı şekilde bu veri seti için "tüm sınıf miktarları aynı olduğundan: accuracy ≅ F1 skor" metrikleri, bu veri seti için bakılması uygun olacak olan metriklerdendir.

Ayrıca modelin büyüklüğü uzun eğitimleri taşıyamamaktadır. 15 epochs'tan itibaren eğitim büyük oranda azalmaktadır. Dolayısıyla daha yüksek skor almak için daha kompleks modeller kurulmalıdır.