Yapay Sinir Ağları

Final Projesi

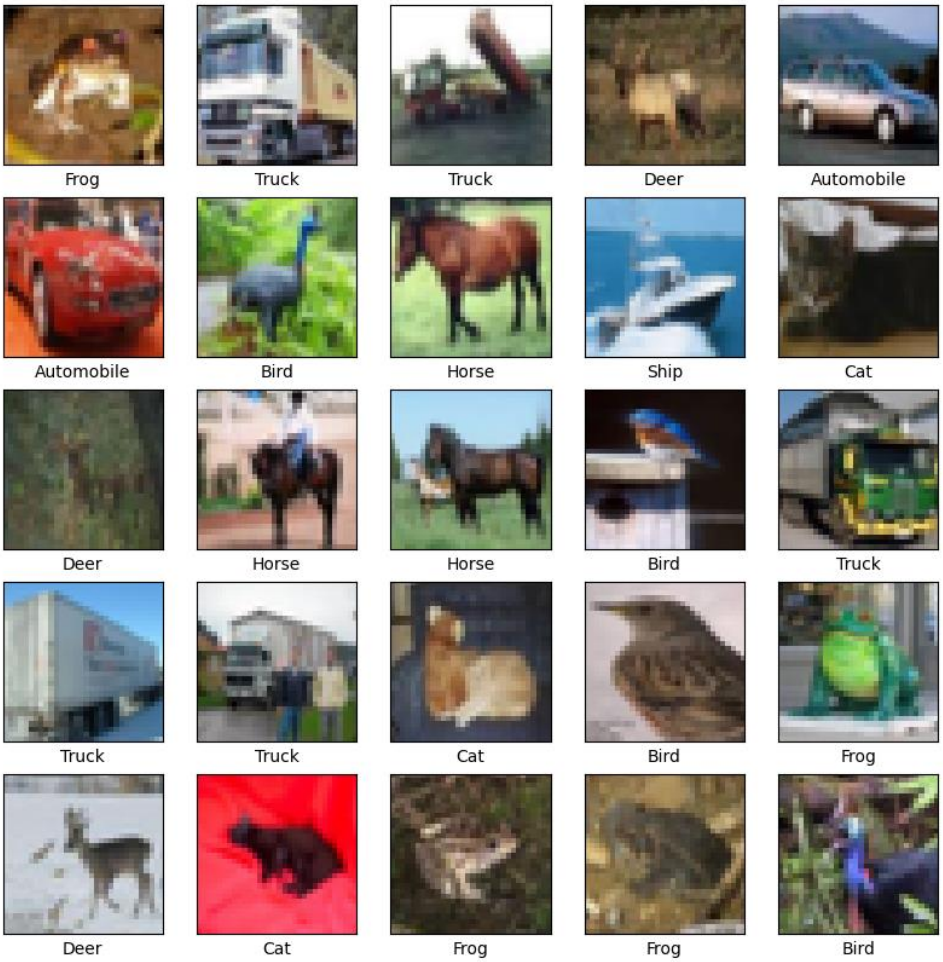
10.02.2021

Mehmet Şerbetçioğlu -- 040160056

Ahmet Hulusi Tarhan -- 040170738

CIFAR10 Veri Seti:

CIFAR10 veri seti 60000 farklı 32x32 çözünürlüklü resimlerin oluşturduğu bir veri setidir. Bu resimler renkli olup RGB değerlere sahiptir. Ayrıca her resim 10 farklı, birbiriyle çakışmayan sınıfa dahildir. Bu sınıflar; uçak, araba, kuş, kedi, geyik, köpek, kurbağa, at, gemi ve kamyondan oluşur. Sınıflar birbiriyle çakışmadığı için "araba" sınıfına dahil olan resimlerde kamyon resmi yoktur. "Kamyon" sınıfında ise yalnızca büyük kamyonlar olup iki sınıf arasında bir ortaklık gözlenmez. Veri setinden örnek resimler Figür 1'de görülebilir.



Figür 1: Veri setinden örnek resimler.

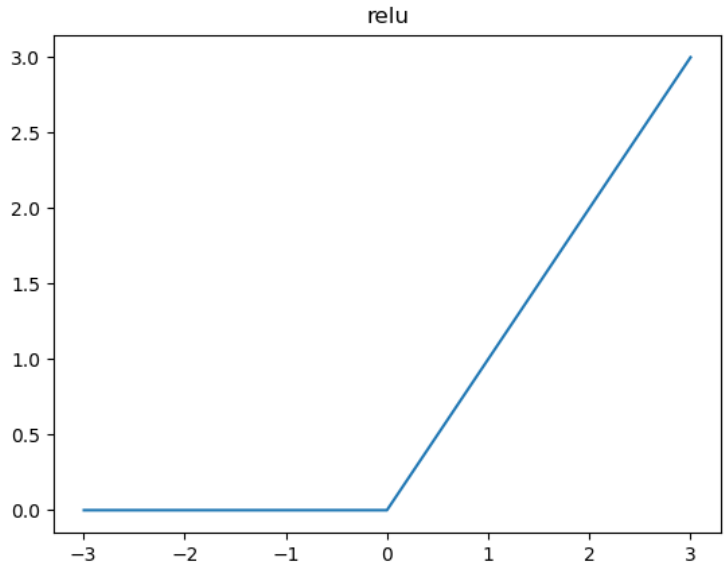
Veri seti daha önce birçok kez sınıflandırma problemlerinde kullanılmıştır. Çoğunlukla "Convolutional Neural Network" (CNN) yapılarının kullanıldığı ve başarılı olduğu görüntü işleme problemine bu projede çok katmanlı algılayıcı ile yaklaşılmıştır.

"Scratch" Ağ Yapısı:

Problemin çözümünde çok katmanlı algılayıcı kullanılmıştır. Çok katmanlı algılayıcı Rosenblatt'ın "Perceptron" nöron modelinin katmanlarla oluşturulmuş bir şekilde kurulduğu, nonlineer problemleri çözebilen bir ağ yapısıdır. Bu yapıları kuran ve eğiten birçok modül olmasına rağmen bu ağ yalnızca matematik işlemlerinde ve veri yapısı için kullanılan "Numpy" modülü kullanılarak oluşturulmuştur.

Ağda verilerin bulunduğu giriş katmanı, nöronların bulunduğu iki gizli katman ve çıkış katmanı bulunmaktadır. Giriş katmanında kullanılan verilerin boyutları, veriler 32x32 çözünürlüklü renkli resimler oldukları için 3072'dir. Çıkış katmanındaki nöron sayısı da sınıf sayını temsil ettiği için 10'dur. Gizli katmanlar bu iki değerin arasında seçilir ve ilk gizli katmanda 128, ikinci gizli katmanda 32 nöron bulunmaktadır.

Gizli katmanlarda kullanılan aktivasyon fonksiyonu, Figür 2'de görüldüğü gibi, "Rectified Linear Unit" (ReLU) seçilmiştir. Bu fonksiyon, modern yapay sinir ağlarında sıklıkla kullanılmakta olup öğrenme hızını oldukça geliştirdiği görülmüştür.



Figür 2: ReLU aktivasyon fonksiyonu.

Çıkış katmanında ise, çıkış değerlerinin toplamının 1 olmasını sağlayan bir aktivasyon fonksiyonu olan "Softmax" kullanılmıştır.

$$\phi(v_i) = \frac{e^{v_i}}{\sum_j e^{v_j}}$$

Ağın "overfitting" yaşamaması, yani gürültülere değil de veriye uyum göstermesi için ileri yolda düşürme (dropout) eklenmiştir. Düşürme, bir katmanda seçilen bir orandaki nöronların inaktif duruma getirilmesi, ileri yol ve eğitim sırasında hesaba katılmamasıyla gerçekleştirilir. Bu işlem ağda tüm nöronların daha sağlıklı eğitilebilmesini sağlar.

İleri yol ile elde edilen ağ çıkışı geleneksel olarak ortalama karesel hata (mean squared error) ile değerlendirilerek ağdaki ağırlıklar güncellenir. Bu ağda ise "Cross entropy" (CE) fonksiyonu kullanılmıştır. CE, olasılıksal bir yaklaşımdır. Kısaca, ağ çıkışlarındaki olasılıkların "Likelihood" fonksiyonları ile elde edilen bir maliyet fonksiyonudur. Softmax ile ağ çıkışları (0,1) aralığında olduğu için bu olasılıksal yaklaşım iyi bir performans gösterir.

Ağın eğitilmesinde "backpropagation" metodu kullanılmıştır. Bu metodla ağ çıkışında elde edilen hatanın ağırlıklara göre hangi yönde değiştiği hesaplanıp, ağırlıkların hatayı azaltmaya yönelik hareket etmesi sağlanır.

Ek olarak, eğitimi hızlandırması ve ağın lokal minimumlarda takılmaması için bir momentum terimi eklenmiştir. Momentum, bir önceki ağırlıkla güncel ağırlığın farkına göre ağırlığın daha fazla veya daha az değişmesini sağlar. Bu şekilde lokal minimumu çevreleyen lokal maksimumlar atlatılabilir ve düşük öğrenme hızlarında sürekli azalan hatalarda daha hızlı bir öğrenim gerçekleşebilir.

Momentum bir dezavantajı, ağı elde edilen lokal minimumlardan daha yüksekte bulunan lokal minimumlara çıkarabilmesidir. Bu sebeple kurulan ağda elde edilen en düşük hata noktası ve o noktayı elde eden ağ saklanır. Belli bir süre elde edilen en düşük hata noktasının altında bir nokta elde edilmezse, ağ en iyi durumdaki haline geri döner ve eğitime devam eder. Bu işlemin gerçekleştiği adım sayısı 8 alınmıştır.

Momentum oranı, öğrenme hızı ve düşürme oranı optimal değerlerinin bulunması için farklı değerler verilerek test edilmiştir. Test edilen değişken dışındaki değerler sabit tutulup, standart değerleri;

- Öğrenme hızı = 0.01,
- Momentum oranı = 0.8,
- Düşürme oranı = 0.8

alınmıştır. Burada düşürme oranı, isminden biraz farklı olarak aktif tutulan nöronları ifade etmektedir. Bu oran 1 değerini aldığında düşürmenin ağa bir etkisi yoktur.

Ağın Python'da İmplementasyonu:

Ağ Python'da yalnızca matematik ve veri saklama işlemlerini gerçekleştiren "Numpy" modülü kullanılarak oluşturulmuştur. Verileri çekmekte, yazdırmakta, çizdirmekte ve farklı verileri toplamakta yardımcı olan "MatPlotLib", "Xlwt", "Pickle" gibi modüller de kullanılmıştır. Modüllerin tam listesi Figür 3'te de görülebilir.

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from matplotlib import cm
import os
import pickle
import xlwt
from xlwt import Workbook
from xlrd import open_workbook
import time
```

Figür 3: Kullanılan modüller.

Verilerin ve Ağın Hazırlanması:

Öncelikle verilerin bulunduğu "Data" klasörü adresi, verilerin giriş ve çıkış boyutları belirtilir (Figür 4).

```
# Kodun bulunduğu adres, girişteki veri boyutu ve çıkış boyutu atanır.
__location__ = os.path.realpath(os.path.join(os.getcwd(), os.path.dirname(__file__)))
dataFile = os.path.join(__location__, 'Data')
dim = 3072
outdim = 10
```

Figür 4: Veri adresinin ve boyutlarının belirlenmesi.

Veriler, veri setinde belirtilen "unpickle" fonksiyonunun modifiye edilmiş haliyle çekilir (Figür 5).

```
# Eğitim ve test kümeleri liste halinde çekilir. Bu işlem sırasında 32x32'lik renkli resimler
# tek vektör olarak ve [0,1] aralığında ölçeklenerek listelere çekilir. Ek olarak veri setinde
# resim sınıfları [0-9] aralığındaki sayılardan oluşmaktadır. Bu sayılar, (10,1)'lik bir vektörde
# sınıf indisi 1 olup geri kalanlarının 0 olduğu bir vektöre dönüştürülür ve verinin sonuna eklenir.
trainList1 = normalize_List(loadList(os.path.join(dataFile, 'data_batch_1')), dim)
testList = normalize_List(loadList(os.path.join(dataFile, 'test_batch')), dim)

# batch_size: her sınıftan kaç resmin eğitim ve test veri setlerinde bulundurulacağı bilgisi.
# Eğitim kümesinde 10000 resim olduğu için çok uzun süre eğitim gerçekleşir. Ayrıca eğitim sırasında
# ağın bir sınıfa uzun süre rastlamaması ihtimali de bulunmaktadır. Çok büyük bir eğitim kümesi kullanıldığında
# Eğitim yavaşlar ve verimsizleşir. Bu nedenle her sınıftan batch_size = 200 resim bulundurmak üzere toplam
# 2000 resimden oluşan bir eğitim kümesi oluşturulur.
batch_size = 200
trainList = batchset(trainList1[:], batch_size, dim, outdim)
```

Figür 5: Veri kümelerinin oluşumu.

"unpickle" fonksiyonunun orjinalinde veriler kütüphane halinde bulunmaktadır ve sınıf bilgileri 0-9 aralığındaki rakamlarla ifade edilmektedir. Modifiye edilen halinde ise bu kütüphaneden göre 3072 boyutluk resim verisi çekilir. Ardından bu vektörün sonuna, 10 boyutlu olan ve sınıf bilgisindeki indisin 1 olup geri kalanının 0 olduğu bir vektör eklenir. Bu şekilde hem sınıf hem de resim bilgileri tek bir vektörde toplanır. Bu vektörler daha sonra bir listede toplanıp fonksiyon dışına aktarılır (Figür 6).

```
def loadList(file):
    # Listeden veriler çekilir. Bu fonksiyon verisetinin belirttiği "unpickle"
    # fonksiyonunun modifiye edilmiş halidir. Başlıca değişiklikler; sınıf bilgisinin
    # [0-9] şeklinde değil de 10 boyutlu, sınıf indisinde 1 geri kalanında 0 olan bir
    # vektöre çevrilmesi, verilerin bir numpy dizisi yerine liste halinde çekilmesidir.
    with open(file, 'rb') as fo:
        batch = pickle.load(fo, encoding='bytes')
        # print(batch.keys())
    features = batch[b'data']
    labels = np.array(batch[b'labels'])
    labels = labels.reshape(labels.size,1)
    # print(labels.shape)
    # print(features.shape)
    init = 0
    for label in labels:
        if init == 0:
            labelArr = np.zeros((1,outdim))
            labelArr[0,label] = 1
            init = 1
        else:
            tempArr = np.zeros((1,outdim))
            tempArr[0,label] = 1
            labelArr = np.concatenate((labelArr, tempArr),axis=0)
    dataArr = np.concatenate((features, labelArr), axis=1)

    print(batch[b'batch_label'], " loaded")
    return dataArr.tolist()
```

Figür 6: Verilerin çekilmesi ve veri vektörlerinin oluşumu.

Resim verileri 0-255 aralığında RGB değerlerden oluşmaktadır. Ağın işleyebilmesi için bu değerler [0,1] aralığında ölçeklenir (Figür 7). Bu, veri boyutunun büyüklüğünden dolayı ağda kullanılan işlemlerin sorunsuz çalışması için de önemlidir. Örnek olarak "numpy.exp()" fonksiyonu ağda sıklıkla kullanılmaktadır ve çok yüksek ve çok düşür değerleri kaldıramamaktadır.

```python
def normalize_List(dataList, dim):
    # Veriler RGB değerlere sahip 32x32'lik resimlerin pixel bilgilerinden oluşur.
    # RGB değerleri 0-255 aralığında olup işlem yapılırken overflow yaşanmaması için
    # 0-1 aralığında ölçeklenir.
    for data in dataList:
        data[:dim] = [x / 255 for x in data[:dim]]
    return dataList
```

Figür 7: Giriş verilerinin ölçeklenmesi.

Veri setinde 60000 resim bulunup bunlar 10000'er resim bulunduran 6 farklı veri setine ayrılmıştır. Bu veri setlerinden 1'i test, 5'i ise eğitim seti olarak adlandırılmıştır. Eğitim setlerinden herhangi birinin kullanılması her iterasyonda 10000 veri için eğitim yapılması demektir ve bu ağı fazlasıyla yavaşlatacaktır. Bu sebeple eğitim seti, her sınıftan eşit sayıda resmin bulunduğu daha küçük parçalara ayrılıp kullanılmıştır (Figür 8). Her sınıftan bulunan resim sayısı "batch_size" ile seçilmektedir ve bu projede 200 seçilmiştir. Bu şekilde ağ 2000 resimlik bir veri seti ile eğitim görür. Kullanılan veri setinde sınıflar kendi içinde de ayrık desenler göstermektedirler. Örnek olarak "kamyon" sınıfına dahil resimler farklı renklerde, farklı tiplerde kamyonları belirttiği gibi oyuncak kamyonları bile içerebilmektedir. Ayrıca bu nesneler farklı rotasyonlarda ve resmin farklı konumlarında bulunabilmektedirler. Aslında bu sebeplerden dolayı, veri seti daha çok CNN ile sınıflandırmaya uygundur. Bu durum eğitilen ağın birçok farklı veriye ihtiyaç duymasını da sağlar. Eğitim kümesinin küçülmesi, eğitimi hızlandırırken test sırasında kötü performans alınmasını sağlayabilir.

```python
def batchset(dataList, batch_size, dim, class_count):
    # Veri kümesini daha küçük kümelere ayıran fonksiyon. "batch_size" değeri
    # tek bir sınıftan kaç verinin alınacağını belirler. Girilen bir veri listesinden
    # her sınıftan "batch_size" sayısında veri yeni bir listeye eklenip döndürülür.
    newList = []
    batchArr = np.zeros(class_count)
    random.shuffle(dataList)
    for data in dataList:
        clsdata = np.array(data[dim:],dtype=float).reshape(class_count,1)
        for i in range(class_count):
            if clsdata[i] == float(1) and batchArr[i] < batch_size:
                newList.append(data)
                batchArr[i] += 1
    return newList
```

Figür 8: Eğitim kümesinin küçültülmesi.

Eğitim ve test kümeleri hazırlandıktan sonra ağ ve eğitimde kullanılacak değerler seçilir. Eğitimin gerçekleşeceği iterasyon sayısı “epoch”, hata için durdurma koşulu “eth” seçilir. Ek olarak sonuçların kaydedilmesi için klasör ve boş bir liste oluşturulur (Figür 9). Testler sırasında ise ağın oluşumunda kullanılan gizli katman nöron sayıları “h1n” ve “h2n”, öğrenme hızı “lrate”, düşürme oranı “drate”, momentum oranı “mrate”, aktivasyon fonksiyonu “act_func” seçilir (Figür 10).

```python
# Maksimum iterasyon sayısı ve eğitimin durdurulacağı hata limiti seçilir.
epoch = 50
eth = 1e-2

# Sonuçların toplanacağı klasör atanır.
resultfolder = os.path.join('Scratch', 'Results')

# Öğrenme hızının farklı değerleri için eğitim ve test gerçekleştirilir. Her test 5 kere gerçekleştirilip
# yorumlanmak üzere sonuçların ortalaması alınacaktır. Her testin sonucu kaydedilir.
# Seçilen öğrenme hızları ve {0.0001, 0.001, 0.01, 0.1} şeklindedir. 0.1 öğrenme hızından sonrasında ağın eğitilmediği
# gözlemlenmiştir. 0.0001 durumunda da ağın fazla yavaş eğitilip yüksek lokal minimum noktalarından kurtulamadığı görülür.
results = []
lrateT = (0.0001, 0.001, 0.01, 0.1)
```

Figür 9: Eğitim parametrelerinin, sonuç listesinin oluşturulması.

```python
for i in range(4):
    # Ağ özellikleri atanır. Gizli katman nöron sayıları, öğrenme hızı, dropout oranı, momentum oranı, gizli katmanların
    # kullanacağı aktivasyon fonksiyonu seçilir.
    # Çıkış katmanı Softmax fonksiyonu kullanıp gizli katmanlar ReLU kullanacaktır.
    # Öğrenme hızı, dropout oranı ve momentum oranının standart değerleri sırasıyla 0.01, 0.8 ve 0.8'dir. Oranların değiştiği
    # testlerde diğer oranlar bu sayılarda sabit tutulmuştur.
    # Dropout rate katmanda eğitim sırasında aktif tutulan nöron oranının bilgisini verir. drate = 0.8, nöronların yaklaşık
    # %20'sinin inaktif olduğunu söyler.
    h1n = 128
    h2n = 32
    lrate = lrateT[i]
    drate = 0.8
    mrate = 0.8
    act_func = 'relu'
```

Figür 10: Ağ parametrelerinin belirtilmesi.

Ağ sonuçlarının incelenmesi için kullanılacak "confusion matrix" için boş bir dizi oluşturulur, sonuçların kaydedileceği alt klasör oluşturulur (Figür 11).

```
# Ağın performansının incelenmesi için "Confusion matrix" matrisi atanır.
conf_mat = np.zeros((outdim, outdim), dtype=float)

# Test sonuçlarının kaydedileceği klasör oluşturulur.
savefolder = os.path.join(resultfolder, 'Learning_Rate_' + str(lrate))
if not os.path.exists(savefolder):
    os.makedirs(savefolder)
```

Figür 11: "Confusion matrix" ve sonuç klasörünün oluşturulması.

Daha sonra her testte eğitimden önce, daha önce belirlenmiş parametreleri kullanan yeni bir ağ oluşturulur (Figür 12).

```
# Her test 3 kez tekrarlanıp sonuçları kaydedilir.
for j in range(3):
    # Daha önce belirlenmiş değerlerle iki gizli katmanlı ağ oluşturulur.
    network = create_2h_network(dim, h1n, h2n, outdim, lrate, drate, mrate, True, act_func)
```

Figür 12: Ağın oluşturulması.

Ağın oluşumunda katmanlardaki nöronların bulunacağı üç liste oluşturulur. Daha sonra seçilen nöron sayısı kadar bu listelere nöronlar atanır. Bu nöronlar daha önce seçilen parametreler dahilinde oluşturulmaktadır. Buna istisna olan çıkış katmanında, düşürme oranı her zaman 1 olup, aktivasyon fonksiyonu olarak "softmax" kullanılmaktadır (Figür 13).

```
def create_2h_network(data_size, l1_size, l2_size, lo_size, learning_rate, dropout_rate, momentum_rate, bias, function):
    # İki gizli katmana sahip bir çok katmanlı algılayıcı ağı üreten fonksiyon.
    l1_List = []
    l2_List = []
    lo_List = []
    # Öğrenme hızı, biasın eklenip eklenmemesi, momentum oranı, ağdaki nöron sayısı gibi argümanlar kullanılır.
    # Dropout oranı ve aktivasyon fonksiyonu gizli katmanlarda kullanılacaktır ve çıkış katmanında sabittir.
    for i in range(l1_size):
        l1_List.append(Neuron(data_size, learning_rate, dropout_rate, momentum_rate, bias, function))
    for i in range(l2_size):
        l2_List.append(Neuron(l1_size, learning_rate, dropout_rate, momentum_rate, bias, function))
    for i in range(lo_size):
        lo_List.append(Neuron(l2_size, learning_rate, 1, momentum_rate, bias))
    # Oluşturulan katmanlar bir liste halinde döndürülür.
    Layers = np.array([l1_List, l2_List, lo_List])
    return Layers
```

Figür 13: Ağ listesinin, katmanlarının oluşumu.

Nöron Sınıfı:

Ağ oluşumunda katmanlara dağıtılan nöronlar, nöron sınıfına ait objelerden oluşur. Nöron sınıfı, içinde ileri yol, güncelleme gibi birçok fonksiyona sahip bir sınıf olup ağdaki tek bir nöronu temsil etmektedir (Figür 14).

```
class Neuron():
    # Nöron yapısı.
    def __init__(self, dims, learning_rate = 1e-2, dropout_rate = 1, momentum_rate = 0.9, bias = True, activation_type = "sigmoid"):

    def reset(self): ···

    def forward(self, data): ···

    def out_preforward(self, data): ···

    def out_postforward(self, exp_sum): ···

    def activation_function(self, data): ···

    def update(self, grad): ···
```

Figür 14: Nöron sınıfının genel yapısı.

Bir nöron objesi ilk kez oluşturulduğunda; giriş boyutu, öğrenme hızı, düşürme oranı, momentum oranı, bias değerinin eklenip eklenmeyeceği ve aktivasyon fonksiyonu parametrelerini kullanır. Bu değerler atandıktan sonra başlangıçtaki durumunun oluşturulacağı "reset" fonksiyonu çağrılır (Figür 15).

```
def __init__(self, dims, learning_rate = 1e-2, dropout_rate = 1, momentum_rate = 0.9, bias = True, activation_type = "sigmoid"):
    # Nöronun parametreleri atanır. Bu parametrelerde giriş boyutu, öğrenme hızı,
    # dropout oranı, momentum oranı, bias eklenip eklenmeyeceği ve aktivasyon fonksyionu bulunur.
    self.dims = dims
    self.learning_rate = learning_rate
    self.dropout_rate = dropout_rate
    self.momentum_rate = momentum_rate
    self.bias = bias
    self.activation_type = activation_type

    # Nöronun başlangıç değerleri atanır.
    self.reset()
```

Figür 15: Nöronun başlangıç değerleri atanması.

"reset" fonksiyonunda öncelikle nöronun ağırlıkları atanır. Ağırlıklar [-0.05, 0.05] aralığında rastgele seçilen değerlere sahiptir. Bu değerin küçük seçilmesi kod çalışırken "overflow" yaşanmaması için önemlidir. Ek olarak önceki ağırlıklar, giriş verisi, lineer kombinasyon, nöron çıkışı için boş diziler ve değerler atanıp nöron "dropped" değeri yanlış alınarak aktif durumda başlatılır (Figür 16).

```python
def reset(self):
    # Nöronda ağırlık vektörü oluşturulur. Ayrıca momentum için kullanılacak,
    # önceki ağırlıkların saklanacağı bir vektör oluşturulur.
    if self.bias is True:
        self.biasval = 1
        self.weights = np.zeros((1, self.dims + 1), dtype=float)
        self.weights[0,:] = np.random.rand(self.dims + 1)*0.1 - 0.05
        self.prevweights = np.zeros((1, self.dims + 1), dtype=float)
    else:
        self.weights = np.zeros((1, self.dims), dtype=float)
        self.weights[0,:] = np.random.rand(self.dims)*0.1 - 0.05
        self.prevweights = np.zeros((1, self.dims), dtype=float)
    # Giriş verisi, lineer kombinasyon ve aktivasyon için boş değişkenler atanır.
    self.data = np.zeros(self.dims)
    self.lin_comb = 0
    self.activation = 0
    # İnaktiflik durumu da "False" olup nöron aktif olarak başlatılır.
    self.dropped = False
```

Figür 16: Ağırlıkların başlangıç değerleri atanması ve nöron değerlerinin oluşturulması.

Nöron sınıfında ara katmanların kullandığı ileri yol fonksiyonuna ek olarak çıkış katmanının kullandığı iki farklı ileri yol fonksiyonu bulunmaktadır. Ara katmanların kullandığı ileri yol fonksyionunda girişlerle ağırlıklar çarpılıp aktivasyon fonksiyonundan geçirilir ve nöron çıkışı elde edilir (Figür 17).

```python
def forward(self, data):
    # Gizli katmanlar için ileri yol işlemi. Giriş verisi düzenlenir ve biasla birlikte vektör haline getirilir.
    if self.bias is True:
        self.data = np.concatenate((data, np.array([self.biasval],dtype=float).reshape(1,1)), axis = 0)
    else:
        self.data = np.array(data)
    w = self.weights
    # Ardından ağırlıklarla çarpılarak lineer kombinasyon elde edilir.
    self.lin_comb = np.dot(w, self.data)
    # if np.isinf(self.lin_comb):
    #     print("lincomb inf")
    # if np.isnan(self.lin_comb):
    #     print("lincomb nan")
    # Aktivasyon fonksiyonu ile de nöron çıkışı elde edilir.
    self.activation = self.activation_function(self.lin_comb)
    return self.activation
```

Figür 17: Gizli katman ileri yol fonksiyonu.

Çıkış katmanının farklı fonksiyonlar kullanmasının sebebi “softmax” fonksiyonu kullanımıdır. Bu fonksiyonda nöron çıkışının elde edilmesi için öncelikle tüm katmandaki lineer kombinasyonların eksponansiyellerinin toplamı elde edilmelidir. Bu sebeple lineer kombinasyonların elde edildiği ve nöron çıkışının elde edildiği iki fonksiyon kullanılır (Figür 18-19).

```python
def out_preforward(self, data):
    # Çıkış katmanı için ileri yolun başlangıç işlemi.
    # Veri düzenlenir ve bias değeri eklenir.
    if self.bias is True:
        self.data = np.concatenate((data, np.array([self.biasval],dtype=float).reshape(1,1)), axis = 0)
    else:
        self.data = np.array(data)
    w = self.weights
    # Lineer kombinasyon elde edilir.
    self.lin_comb = np.dot(w, self.data)
    # if np.isinf(self.lin_comb):
    #     print("lincomb inf")
    # if np.isnan(self.lin_comb):
    #     print("lincomb nan")
    # Katman boyunca toplanmak üzere, lineer kombinasyonun eksponansiyeli döndürülür.
    return np.exp(self.lin_comb)
```

Figür 18: Çıkış katmanı lineer kombinasyon fonksiyonu.

```python
def out_postforward(self, exp_sum):
    # Çıkış katmanı için ileri yolun bitiş işlemi.
    # Toplam eksponansiyel değer ile ağ çıkışı elde edilir.
    self.activation = np.exp(self.lin_comb)/exp_sum
    return self.activation
```

Figür 19: Çıkış katmanı nöron çıkışı fonksiyonu.

Nöron sınıfında farklı aktivasyon fonksiyonlarının ve türevlerinin kullanıldığı bir fonksiyon da bulunmaktadır. Tanımlı fonksiyonlar tanjant hiperbolik (tanh), lojistik (sigmoid), "rectified linear unit" (relu) ve "leaky rectified linear unit" (lrelu) fonksiyonlarıdır (Figür 20).

```
def activation_function(self, data):
    # Farklı aktivasyon fonksiyonları ve lokal gradyan hesaplanmasında
    # kullanılmak üzere fonksiyonların türevlerinin elde edilir.
    if self.activation_type == "tanh":
        y = tanh(data)
        self.activation_derivative = y[1]
        return y[0]
    elif self.activation_type == "sigmoid":
        y = sigmoid(data)
        self.activation_derivative = y[1]
        return y[0]
    elif self.activation_type == "relu":
        y = relu(data)
        self.activation_derivative = y[1]
        return y[0]
    elif self.activation_type == "lrelu":
        y = lrelu(data)
        self.activation_derivative = y[1]
        return y[0]
    else:
        return data
```

Figür 20: Aktivasyon fonksiyonlarının bulunduğu fonksiyon.

Ağırlıkların güncellendiği "update" fonksiyonuna, ileri yol sonrası elde edilen lokal gradyan ve momentum kullanılır (Figür 21).

```
def update(self, grad):
    # Nöronun ağırlıklarının güncellendiği fonksiyon.
    momentumterm = self.momentum_rate*(self.weights - self.prevweights)
    self.prevweights = self.weights
    # print(self.weights.shape)
    self.weights += ((self.learning_rate*grad[0])*self.data).reshape(1, self.weights.size) + momentumterm
    # print(self.weights.shape)
```

Figür 21: Ağırlıkların güncellendiği fonksiyon.

Eğitim Süreci:

Oluşturulan ağ ve belirlenen eğitim parametreleri kullanılarak eğitim başlatılır (Figür 22).

```
# Eğitim başlatılır ve eğitim süresi, hata oranı gibi sonuçlar kaydedilir ve yazdırılır.
start_time = time.time()
result = educationprocess(trainList, network, epoch, eth, dim, savefolder, j)
ed_time = time.time() - start_time
print("Education time: %f" % ed_time)
print("Training is over in %d steps with %.4f error with the best performing network." % (result[0], result[1]))
```

Figür 22: Eğitim sürecinin başlatılması.

Eğitim işlemi sırasında, ağ eğitim kümesindeki tüm veriler için güncellendiğinde bir iterasyon biter. Eğitimde ulaşılan en iyi hata performansına sahip ağ durumu seçilir. Eğitimin bitmesiyle eğitim boyunca elde edilen hatanın durumu çizdirilip kaydedilir (Figür 23).

```
def educationprocess(dataList, network, epoch, errorthreshold, dim, savefolder, testiter):
    # Eğitim işleminin gerçekleştiği fonksiyon.
    # Öncelikle sonuç elde etmek ve çizim yapılabilmesi için boş listeler ve diziler atanır.
    y = []
    result = np.zeros(2)
    result[0] = epoch
    # Eğitimde en iyi performansı gösteren ağın çıkması sağlanacaktır. Bunun için başlangıçtaki
    # en iyi hatayı gösterecek "bestE"; yüksek bir değer, bu durum için 999, atanır.
    # "bestE_streak" ise ağın bulunan minimuma yakın şekilde eğitilmesi için kullanılır.
    # 8 iterasyonda ağ gelişmemişse en düşük hatayı elde ettiği duruma geri döner ve eğitime
    # devam eder.
    bestE_streak = 0
    bestE = 999
    best_network = network[:]
    for i in range(epoch):⋯
    # Eğitim bittikten sonra test edilmek üzere en iyi durumdaki ağ yapısı seçilir.
    network = best_network[:]

    # Eğitim sırasındaki ortalama hata değişimi çizdirilir.
    plt.figure()
    yshape = int(round(result[0]))
    x = np.arange(1, yshape + 1)
    plt.plot(x, np.array(y).reshape(yshape), color = 'blue')
    plt.xlabel("Iteration")
    plt.ylabel("Error")
    plt.title("Error during education")
    plt.savefig(os.path.join(savefolder, 'Test' + str(testiter) + '.png'))
    plt.close()

    # Elde edilen en iyi hata döndürülür.
    result[1] = bestE
    return result
```

Figür 23: Genel eğitim süreci.

Her iterasyonun başında eğitim kümesi karıştırılmaktadır. Ardından veriler, resim verisi ve sınıf verisi olarak ayrılır ve eğitim için "educate" fonksiyonuna girilir. Tüm veriler için eğitim gerçekleştikten sonra hatanın ortalaması alınır. Eğer hata 8 iterasyon boyunca iyileşmemişse, ağ en iyi performansı gösterdiği duruma geri döner ve eğitime devam eder. Hatanın belirtilen sınırın altına düşmesiyle veya "epoch" iterasyon sayısının dolmasıyla eğitime son verilir (Figür 24).

```
for i in range(epoch):
    # Her iterasyonun başında eğitim kümesi karıştırılır.
    random.shuffle(dataList)
    Eort = 0
    # Eğitim kümesindeki her veri için ileri yol ve geri yol gerçekleşir.
    for data in dataList:
        # Giriş verisi "imdata" ve sınıf verisi "clsdata" ayrılır ve numpy dizisi haline getirilir.
        imdata = np.array(data[:dim],dtype=float).reshape(dim,1)
        clsdata = np.array(data[dim:],dtype=float).reshape(len(data) - dim,1)
        # Veri için ağ eğitilir.
        E = educate(imdata, network, clsdata)
        # Tüm hatalar toplanır. Tüm veriler için eğitim gerçekleştiğinde hataların ortalaması alınır.
        Eort += E
    Eort = Eort/len(dataList)
    # Eğer ortalama hata ağın en iyi durumundaki hatadan daha iyiyse yeni en iyi ağ yapısı olarak kaydedilir.
    if Eort < bestE:
        if bestE == 999:
            print("%d iteration, %.4f error" % (i + 1, Eort))
        else:
            print("%d iteration, %.4f error, improved from %.4f error" % (i + 1, Eort, bestE))
        best_network = network[:]
        bestE_streak = 0
        bestE = Eort
    # Aksi takdirde ise, 8 kere ağ gelişmemişse, en iyi olduğu duruma geri döner ve eğitime devam eder.
    else:
        if bestE_streak < 8:
            print("%d iteration, %.4f error" % (i + 1, Eort))
            bestE_streak += 1
        else:
            print("%d iteration, %.4f error, reverted to best performing network with %.4f error" % (i + 1, Eort, bestE))
            network = best_network[:]
            bestE_streak = 0
    # Eğitim sırasındaki hata değişimi çizdirilmek üzere kaydedilir.
    y.append(Eort)
    # Ortalama hata, istenen hata limitinin altına düşmüşse eğitim durdurulur.
    if Eort < errorthreshold:
        result[0] = i + 1
        break
```

Figür 24: Eğitimin tek bir iterasyonu.

Her veri, “educate” fonksiyonunda ağda ileri yol için kullanılır, “cross entropy” hatasını elde edilir ve elde edilen hataya göre ağırlıkların güncellenir. Öncelikle gizli katmanların çıkışları elde edilir. Bir önceki katmanın çıkışı, güncel katmanın girişi olarak kullanılır. Bu sırada [0,1) aralığında rastgele seçilen bir sayının düşürme oranından yüksek olması durumunda nöron inaktif hale getirilir ve çıkışı 0 alınır (Figür 25).

```
def educate(data, network, clsdata):
    # Tek bir veri için eğitimin gerçekleştiği fonksiyon.
    # Eğitim sırasında kullanılacak iterasyon değişkenleri
    # ve katman çıkışları dizileri oluşturulur.
    y1 = np.zeros((len(network[0]), 1), dtype=float)
    y2 = np.zeros((len(network[1]), 1), dtype=float)
    yo = np.zeros((len(network[2]), 1), dtype=float)
    i1 = 0
    i2 = 0
    io = 0
    # Gizli katmanlar için ileri yol izlenir ve katman çıkışı elde edilir.
    # Dropout oranına göre nöronların inaktif olma ihtimali bulunmaktadır.
    # Dropout oranı kesin bir oran vermemekte, [0,1) aralığında seçilen bir
    # sayının bu orandan büyük olup olmaması gözlenecektir.
    # İnaktif bir nöronun çıkışı 0 alınır ve inaktif olduğu kaydedilir.
    # Çıkış katmanında dropout oranı her zaman 1'dir, tüm nöronlar hep aktiftir.
    for Neuron in network[0]:
        if np.random.rand() > Neuron.dropout_rate:
            y1[i1] = 0
            Neuron.dropped = True
            Neuron.activation_derivative = 0
            i1 += 1
            continue
        y1[i1] = Neuron.forward(data)
        i1 += 1
    for Neuron in network[1]:
        if np.random.rand() > Neuron.dropout_rate:
            y2[i2] = 0
            Neuron.dropped = True
            Neuron.activation_derivative = 0
            i2 += 1
            continue
        y2[i2] = Neuron.forward(y1)
        i2 += 1
```

Figür 25: Tek veri için gizli katmanlarda ileri yol.

Çıkış katmanı için ise düşürme oranı olmayıp, ileri yolda iki farklı fonksiyon kullanılır. Birinci fonksiyon ile lineer kombinasyonların eksponansiyeli toplamı elde edilir ve bu ikinci fonksiyonda nöron çıkışının eldesinde kullanılır (Figür 26).

```
# Çıkış katmanında softmax fonksiyonu kullanılacağı için öncelikle tüm nöronların
# w*x + b = v değerleri, yani lineer kombinasyonları hesaplanmalıdır. Bu değerlerin ekponansiyelleri
# toplanacak ve nöron çıkışı için kullanılacaktır. Bu nedenle ileri yol iki parçaya ayrılmıştır.
# İlk parçada lineer kombinasyonlar bulunup ekponansiyelleri toplamı elde edilir.
exp_sum = 0
for Neuron in network[2]:
    exp_sum += Neuron.out_preforward(y2)
# İkinci parçada ise bu toplam kullanılarak nöron çıkışları hesaplanır ve kaydedilir.
for Neuron in network[2]:
    yo[io] = Neuron.out_postforward(exp_sum)
    io += 1
```

Figür 26: Ağ çıkışında ileri yol.

Ardından "cross entropy" hatası hesaplanır. Güncellenmede kullanılacak lokal gradyanlar hesaplanır ancak burada da çıkış katmanı ve gizli katmanlar için farklı fonksiyonlar kullanılmaktadır (Figür 27).

```
# Hata hesaplanması için cross entropy kullanılır. Ağın mükemmel bir şekilde sınıflandırması
# durumunda doğru sınıfların çıkışlarında 1 görüleceği için hata E = ln(1) = 0 olacaktır.
# Örnek olarak başlangıç durumunda ise yaklaşık 0.1 çıkışları olacağı için E = ln(0.1) = 2.3 olacaktır.
E = 0
for i in range(yo.size):
    E -= clsdata[i]*np.log(yo[i,0])

# İleri yolun ardından gradyanlar için boş diziler açılır.
grad1 = np.zeros((len(network[0]), 1), dtype=float)
grad2 = np.zeros((len(network[1]), 1), dtype=float)
grado = np.zeros((len(network[2]), 1), dtype=float)

# Lokal gradyanlar hesaplanır. Gizli katmanlarda gradyan hesabı ortalama karesel hata kullanıldığı
# geriye yönelim durumuyla aynıdır. Çıkış katmanında ise işlem kolaylığı sağlayan cross entropy ve softmax
# fonksiyonlarının birlikte kullanıldığında elde edilen bir gradyan hesaplaması kullanılır.
grado = local_grad_out(yo, clsdata)
grad2 = local_grad(network[1], network[2], nextgrad = grado)
grad1 = local_grad(network[0], network[1], nextgrad = grad2)
```

Figür 27: Lokal gradyanların elde edilmesi.

Çıkış katmanı için kullanılan lokal gradyan fonksiyonu, “softmax” ve “cross entropy” birlikte kullanıldığında hatanın lineer kombinasyona türeviyle elde edilir (Figür 28).

```
def local_grad_out(y, clsdata):
    # Softmax ve cross entropy birlikte kullanıldığında elde edilen gradyan değeri.
    return (clsdata - y)
```

Figür 28: Ağ çıkışı lokal gradyanlarının elde edilmesi.

Gizli katmanların kullandığı lokal gradyan fonksiyonu, “backpropagation” metodunda kullanılan standart hesaplama olup bir sonraki katmanın lokal gradyanından, bias ağırlıklarının çıkarıldığı bir ağırlık matrisinden ve aktivasyon fonksiyonu türevinden faydalanmaktadır (Figür 29).

```
def local_grad(list1, list2, nextgrad):
    # Lokal gradyanın hesaplandığı fonksiyon.
    ncount = len(list1)
    act_derivative = np.zeros((ncount, 1), dtype=float)
    grad = np.zeros((ncount, 1), dtype=float)
    # Ağırlık matrisinin transpozu alınır ve bias değerlerinin ağırlıkları çıkarılır.
    wrow = (list2[0].weights.size) - 1
    outcount = len(list2)
    wmat = np.zeros((wrow, outcount), dtype=float)
    for i in range(outcount):
        wmat[:, i] = list2[i].weights[0, 0:wrow]
    # Aktivasyonların türevleri bir vektör halinde çekilir.
    for i in range(ncount):
        act_derivative[i] = list1[i].activation_derivative
    # Lokal gradyan hesaplanırken sonraki katmanın lokal gradyanı da kullanılır.
    # grad(k) = [wT.grad(k+1)] x f'(v(k)) şeklinde ifade edilebilir.
    grad = np.multiply(np.dot(wmat, nextgrad), act_derivative)
    return grad
```

Figür 29: Gizli katman lokal gradyanlarının elde edilmesi.

Lokal gradyanlar da hesaplandıktan sonra ağ boyunca ağırlıklar güncellenir. Gizli katmanlarda düşürülen, yani inaktif hale getirilen, nöronlar güncellenmeyip bir sonraki iterasyon için tekrar aktif hale getirilirler. Bu şekilde bir veri için eğitim gerçekleşir ve elde edilen "cross entropy" hatası döndürülür (Figür 30).

```
# Hesaplanan lokal gradyanlara göre nöronların ağırlıkları güncellenir.
# İnaktif nöronlar güncelleme aşamasında atlanır ve bir sonraki iterasyon için tekrar aktif olurlar.
i1 = 0
i2 = 0
io = 0
for Neuron in network[0]:
    if Neuron.dropped == True:
        Neuron.dropped = False
        i1 += 1
        continue
    Neuron.update(grad1[i1])
    i1 += 1
for Neuron in network[1]:
    if Neuron.dropped == True:
        Neuron.dropped = False
        i2 += 1
        continue
    Neuron.update(grad2[i2])
    i2 += 1
for Neuron in network[2]:
    Neuron.update(grado[io])
    io += 1

# Elde edilen hata döndürülür.
return E.item(0)
```

Figür 30: Ağırlıkların güncellenmesi ve tek veri için eğitim sonu.

Test Süreci:

Ağın eğitiminin bitmesiyle test süreci başlatılır. Bu süreçte daha önce elde edilen 10000 resimden oluşan test kümesi kullanılacaktır. Test sonucu "confusion matrix" şeklinde ve daha detaylı bir şekilde excel dosyasına kaydedilecektir (Figür 31).

```
# Eğitilen ağ ile test kümesindeki resimlerin tahmin edilmesi sağlanır. Doğru ve tahmin edilen sınıflar,
# yanlış tahmin oranı değerleri kaydedilir.
testresult = testprocess(testList, network, dim, conf_mat)
print("Test is over with %.4f error" % testresult[2])
print("Made %d wrong predictions out of %d patterns" % (testresult[0], testresult[1]))
testpredictionerror = testresult[0]/testresult[1]
```

Figür 31: Test aşamasının başlatılması.

Test süreci eğitim sürecine benzerlik göstermektedir. Her veri resim bilgisi ve sınıf bilgisi olarak ayrılır. Ardından ağda resim bilgisi kullanılarak ağ çıkışı elde edilir ve sınıf bilgisiyle karşılaştırılır. Ağ çıkışı ve sınıf bilgisinde maksimum değerin bulunduğu indisler karşılaştırılır. Farklılarsa hatalı seçim sayısı artırılır. Ek olarak "confusion matrix"te belirtilen indisli "hücre" de artırılır. Bu matris test işlemi sonunda, test kümesindeki veri sayısına bölünerek [0,1] aralığında değerler elde edilecek ve bu şekilde çizdirilecektir. Test sonucunda hatalı tahmin sayısı, test kümesi büyüklüğü, ortalama "cross entropy" hatası ve "confusion matrix" matrisi elde edilir (Figür 32).

```
def testprocess(dataList, network, dim, conf_mat):
    # Test aşamasında ileri yol izlenir ve tahmin edilen sınıfla gerçek sınıf karşılaştırılır.
    # Tahmin edilen ve gerçek değerlerin bulunduğu bir matris oluşturulur. Sonuç olarak hatalı tahmin sayısı,
    # test kümesinin büyüklüğü, hata oranı ve oluşturulan "Confusion Matrix" matrisi döndürülür.
    Errorcount = 0
    totalcount = 0
    random.shuffle(dataList)
    Loss = 0
    for data in dataList:
        imdata = np.array(data[:dim],dtype=float).reshape(dim,1)
        clsdata = np.array(data[dim:],dtype=float).reshape(len(data) - dim,1)
        # İleri yol gerçekleştirilip hata ve ağ çıkışı elde edilir.
        [pred, E] = testdata(imdata, clsdata, network)
        # Ağ çıkışındaki maksimum değer 1 olacak şekilde ölçeklenir.
        pred = pred/np.amax(pred, axis = 0)
        # Ağ çıkışında ve sınıf verisinde 1 değerinin bulunduğu indis kaydedilir.
        for i in range(pred.size):
            if pred[i] == float(1):
                pred_ind = i
            if clsdata[i] == float(1):
                cls_ind = i
        # "Confusion Matrix" içinde bu iki indis kullanılarak eşleşen hücre artırılır.
        conf_mat[cls_ind, pred_ind] += 1
        Loss += E
        # Eğer tahmin edilen ve gerçek sınıfın indisleri uyuşmuyorsa hata sayısına bir eklenir.
        if pred_ind != cls_ind:
            Errorcount += 1
        totalcount += 1
    Loss = Loss/len(dataList)
    result = np.array([Errorcount, totalcount, Loss, conf_mat])
    return result
```

Figür 32: Genel olarak test aşaması.

Sınıfların tahmin edilmesi, eğitim aşamasındaki ileri yol ve hata hesaplamasıyla birebir aynıdır (Figür 33).

```
def testdata(data, clsdata, network):
    # Eğitimde gerçekleştirilen ileri yol ve hata hesaplaması burada da yapılır.
    # Tüm ağ değerlendirildiği için tüm nöronlar hep aktiftir.
    y1 = np.zeros((len(network[0]), 1), dtype=float)
    y2 = np.zeros((len(network[1]), 1), dtype=float)
    yo = np.zeros((len(network[2]), 1), dtype=float)
    i1 = 0
    i2 = 0
    io = 0
    for Neuron in network[0]:
        y1[i1] = Neuron.forward(data)
        i1 += 1
    for Neuron in network[1]:
        y2[i2] = Neuron.forward(y1)
        i2 += 1

    exp_sum = 0
    for Neuron in network[2]:
        exp_sum += Neuron.out_preforward(y2)
    for Neuron in network[2]:
        yo[io] = Neuron.out_postforward(exp_sum)
        io += 1
    E = 0
    for i in range(yo.size):
        E -= clsdata[i]*np.log(yo[i,0])

    return yo, E.item(0)
```

Figür 33: Test aşamasında ileri yol ile ağ çıkışı elde edilmesi.

Test sonuçları listeye eklenir, “confusion matrix” tüm testlerdeki eldelerin ortalaması şeklinde çizdirilir. Test sonuçları excel dosyasına yazdırılır (Figür 34).

```
        # Eğitim ve test sonuçları sonuç listesine eklenir.
        results.append([j+1, h1n, h2n, lrate, mrate, drate, act_func, result[0], ed_time, ed_time/result[0], result[1], testpredictionerror, testresult[2]])

    # "Confusion Matrix" içindeki değerlerin 0-1 aralığına alınması sağlanır. Bu değerler
    # aynı zamanda birçok testin ortalamasıdır. Daha sonra bu grafik çizdirilirve kaydedilir.
    # Kod çalışırken çizimin görülmesi için plot_conf_mat() fonksiyonuna show = True argümanı eklenmelidir.
    conf_mat = conf_mat/(len(testList)*5)
    plot_conf_mat(conf_mat, savefolder)
# Test sonuçları kaydedilir.
saveResults(results, resultfolder, 'Results_LRate.xls')
```

Figür 34: Test sonrasında sonuçların kaydedilmesi.

Sonuçların Elde Edilmesi:

"Confusion matrix" çizdirilirken "label" değerleri için sınıf isimleri atanır. Daha sonra 12.8x9.6 inçlik figür oluşturulur. "Tick" parametreleri atanır ve matrisin çizilmesi için "matshow" fonksiyonu ve "Blues" renk haritası seçilir. Etiketler ve başlıklar eklenir, "hücre"lere karşılık gelen değerler yazdırılır ve çizim kaydedilir (Figür 35).

```
def plot_conf_mat(mat, savefolder, show = False):
    # "Confusion Matrix" çiziminin yapıldığı fonksyion.
    # X ve Y'deki indislerin belirttiği sınıfların isimleri bir listede toplanır.
    label = [" ", "Airplane", "Automobile", "Bird", "Cat", "Deer", "Dog", "Frog", "Horse", "Ship", "Truck"]
    # 12.8x9.6 inçlik bir çizim açılır.
    fig = plt.figure(figsize=(12.8, 9.6))
    ax = fig.add_subplot(111)
    # Çizimde "tick"lerdeki yazıların boyutları okunabilmesi için 10 punto seçilir.
    ax.tick_params(axis='both', labelsize=10)
    # "Blues" renk haritası kullanılarak matris çizdirilir.
    cax = ax.matshow(mat, interpolation='nearest', cmap=plt.cm.Blues)
    # "tick"lerdeki etiketler atanır.
    ax.set_xticklabels(label)
    ax.set_yticklabels(label)
    # Renk barı çizdirilir.
    plt.colorbar(cax)
    # Eksenlerin etiketleri yazdırılır.
    plt.title("Confusion Matrix")
    ax.set_ylabel('Truth')
    ax.set_xlabel('Predicted')
    ax.xaxis.set_label_position('top')
    # Matris içindeki renklerin değerleri, hücrelere yazdırılır.
    for (i, j), z in np.ndenumerate(mat):
        ax.text(j, i, '{:0.3f}'.format(z), ha='center', va='center', fontsize=8)
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
    # show = True seçilmesi durumunda eğitim sonrasında matris figür olarak çizdirilir.
    # Aksi takdirde sadece kaydedilir.
    if show == True:
        plt.show()
    # Figür kaydedilir.
    plt.savefig(os.path.join(savefolder, 'Confusion_Matrix.png'))
    plt.close()
```

Figür 35: "Confusion matrix" matrisinin çizdirilmesi.

Excel'e yazdırılırken ise elde edilen sonuçlar için bir excel dosyası oluşturulup sırayla ilgili konumdaki hücrelere yazdırılır ve kaydedilir (Figür 36).

```python
def saveResults(results, savefolder, filename):
    # Elde edilen sonuçlar belirtilen excel dosyasında yazdırılır.
    if not os.path.exists(savefolder):
        os.makedirs(savefolder)
    filedir = os.path.join(savefolder,filename)
    wb = Workbook()
    sheet1 = wb.add_sheet('Sheet1')
    col = 0
    row = 0
    sheet1.write(row, col, 'Test Number')
    sheet1.write(row, col+1, 'Neuron Count (Hidden Layer 1)')
    sheet1.write(row, col+2, 'Neuron Count (Hidden Layer 2)')
    sheet1.write(row, col+3, 'Learning Rate')
    sheet1.write(row, col+4, 'Momentum Rate')
    sheet1.write(row, col+5, 'Dropout Rate')
    sheet1.write(row, col+6, 'Activation Function')
    sheet1.write(row, col+7, 'Education Iteration')
    sheet1.write(row, col+8, 'Education Time')
    sheet1.write(row, col+9, 'Education Time-per-iteration')
    sheet1.write(row, col+10, 'Education Error')
    sheet1.write(row, col+11, 'Test Prediction Error')
    sheet1.write(row, col+12, 'Test Cross Entropy Error')
    row += 1
    for result in results:
        for i in range(len(result)):
            sheet1.write(row, col+i, result[i])
        row +=1

    wb.save(filedir)
```

Figür 36: Sonuçların excel dosyasında kaydedilmesi.

Eğitim ve Test Sonuçları:

Eğitim ve test aşamasında elde edilen sonuçlar, ağın bu problemde gösterdiği performansın incelenmesi için karşılaştırılır. Yapılan testler;

- Öğrenme hızı = {0.0001, 0.001, 0.01, 0.1}
- Momentum oranı = {0, 0.2, 0.4, 0.6, 0.8, 1}
- Düşürme oranı = {0.5, 0.6, 0.7, 0.8, 0.9, 1}

değerleriyle oluşturulan ağlar ile eğitim ve test aşamasının gerçekleşmesini kapsar. Her seferinde bir değer değişken olup değişken olmayanlar öğrenme hızı = 0.01, momentum oranı 0.8, düşürme oranı = 0.8 değerlerine sahiptir.

Öğrenme Hızı (η) Testleri:

Momentum ve düşürme oranı sabit tutulurken öğrenme hızının farklı değerleri için testler gerçekleştirilir.

Öğrenme Hızı	Eğitim Süresi	İterasyon Başına Eğitim Süresi	Eğitim Hatası	Test-Tahmin Hatası	Test Hatası
0,0001	715,1657	14,30331	2,10411	0,7567	2,064203
0,001	713,4909	14,26982	1,497898	0,651767	2,021343
0,01	698,6792	13,97358	1,488809	0,677133	2,434884
0,1	701,2036	14,02407	2,320094	0,899967	2,326881

Eğitim süreleri, ağ yapısı her testte aynı olduğundan anlamlı bir değişim görülmemektedir.

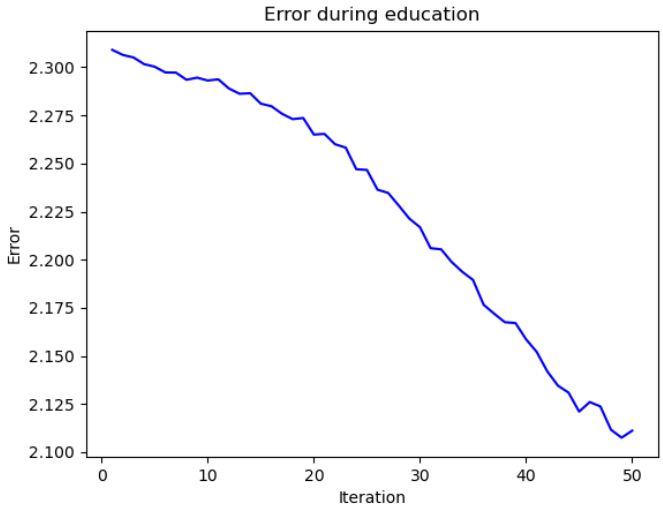
Eğitimdeki hataya bakıldığında, 0.0001 değerinin çok yavaş kaldığı ve 0.1 değerinin efektif olmadığı görülür. Ara değerler ise benzer sonuçlar sergilemişlerdir.

Bu durum test aşamasındaki tahminlerde de görülmektedir. Hatta 0.1 değerinin, resimleri rastgele tahmin etmeye yakın bir sonuç elde ettiği görülür. En iyi tahmin eden 0.001 değeri resimlerin %65'ini yanlış tahmin etmektedir. Bu, problemin çok katmanlı algılayıcı için pek uygun olmadığına işaret eder. Daha gelişmiş metodlar kullanılarak ve ağ optimize edilerek bu değer düşürülebilecek olsa bile anlamlı bir düşüş görülmesi beklenmez.

Test hatası "cross entropy" kullanarak hesaplanmış olup daha önce iyi ve kötü performans sergileyen durumlar arasında anlamlı bir değişme görülmemektedir. Bunun sebebinin aynı sınıftaki resimler arasında çok büyük farklılıklar olmasından kaynaklandığı düşünülmektedir.

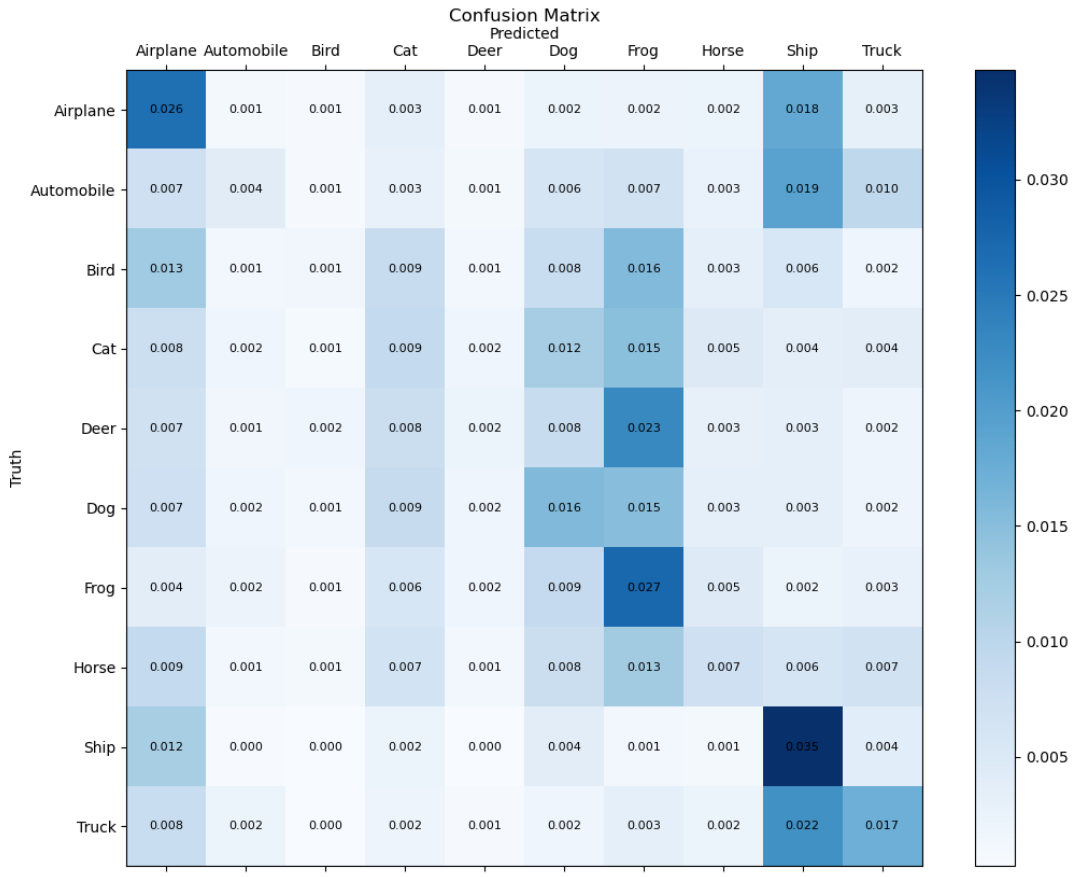
η = 0.0001:

Öğrenme hızı çok düşük olup seçilen sürede yeterince başarılı bir öğrenme gözlenmediği görülür. Eğitim bittiğinde hata düşmeye devam etse de çok yavaş bir düşüş görülmüştür (Figür 37).



Figür 37: Testlerden birinde elde edilen hata grafiği.

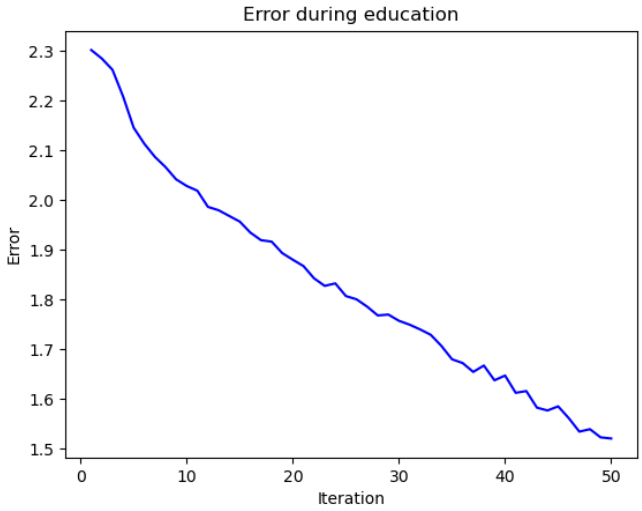
"Confusion matrix" matrisine bakıldığında, otomobil istisnası ile taşıtların diğerlerine göre daha doğru tahmin edildikleri görülür. Kurbağa resimleri de doğru tahmin edilmektedirler. Bu resimlerin tahminlerinin daha kolay olduğu, özellikle hayvan resimlerinin tahmininde zorlanıldığı diğer test sonuçlarında da görülmektedir (Figür 38).



Figür 38: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

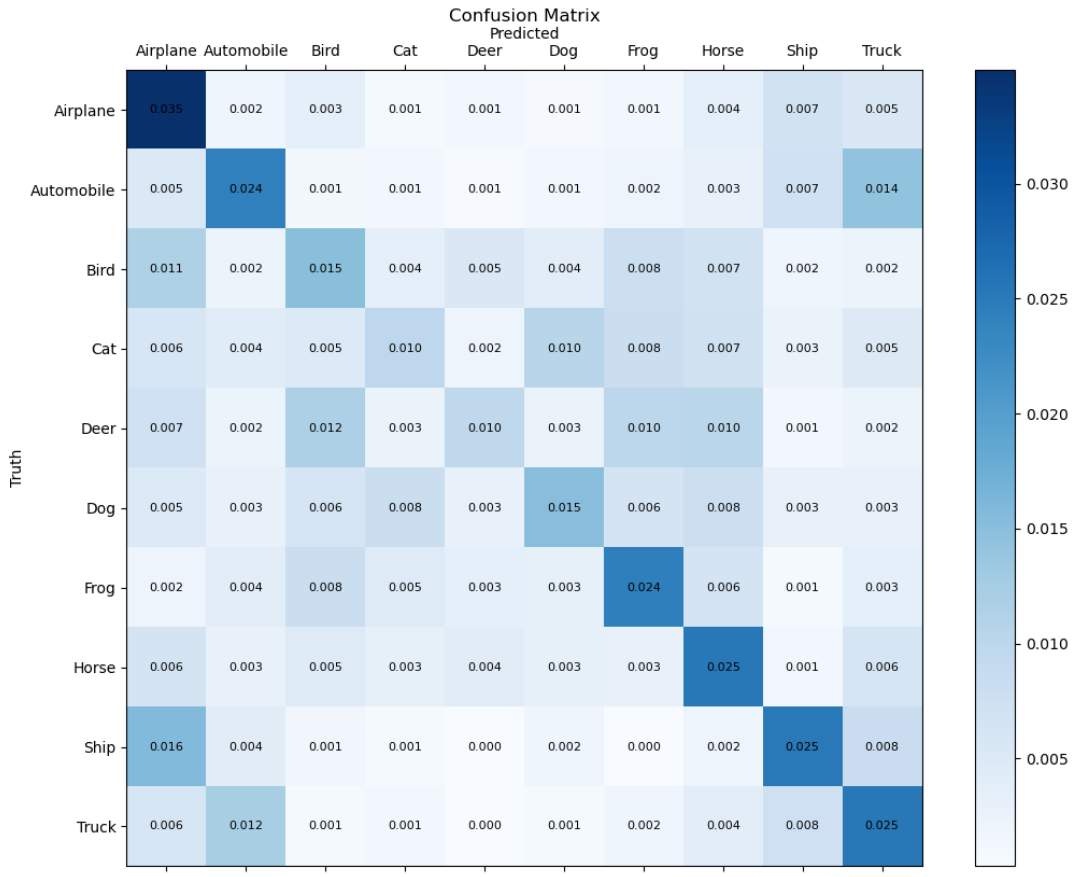
η = 0.001:

Bu öğrenme hızı, hatada stabil ve yeterince hızlı bir azalma görülmesinde başarılı olmuştur. Elde edilen hatanın 1.5 değerine yakın bir noktada sonlandığı görülür (Fig 39).



Figür 39: Testlerden birinde elde edilen hata grafiği.

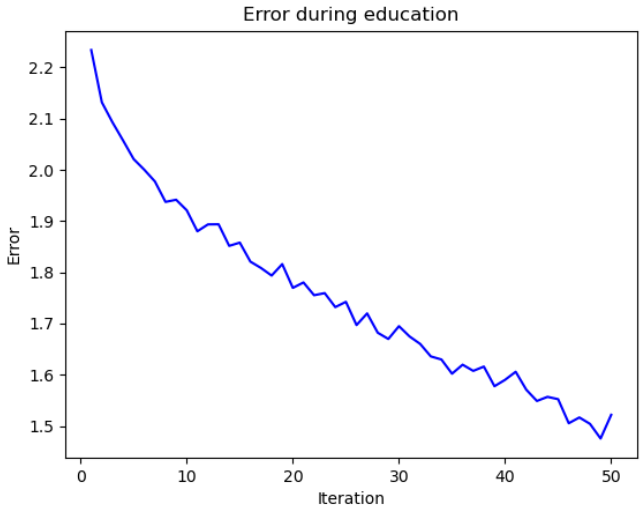
"Confusion matrix" matrisine bakıldığında öncekine göre daha iyi tahminlerin gerçekleştiği görülmektedir. Taşıtlar başarılı bir şekilde tahmin edilirken, at ve kurbağa haricinde hayvanların tahminlerinde zorlanıldığı görülür. Hayvan resimleri, kürk renkleri ve fiziksel özellikleri açısından birbirine fazlasıyla benzemektedirler. Başarılı tahmin edilebilmesi için resimlerde öne çıkan bir özellik bulunması gerekmektedir (Figür 40).



Figür 40: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

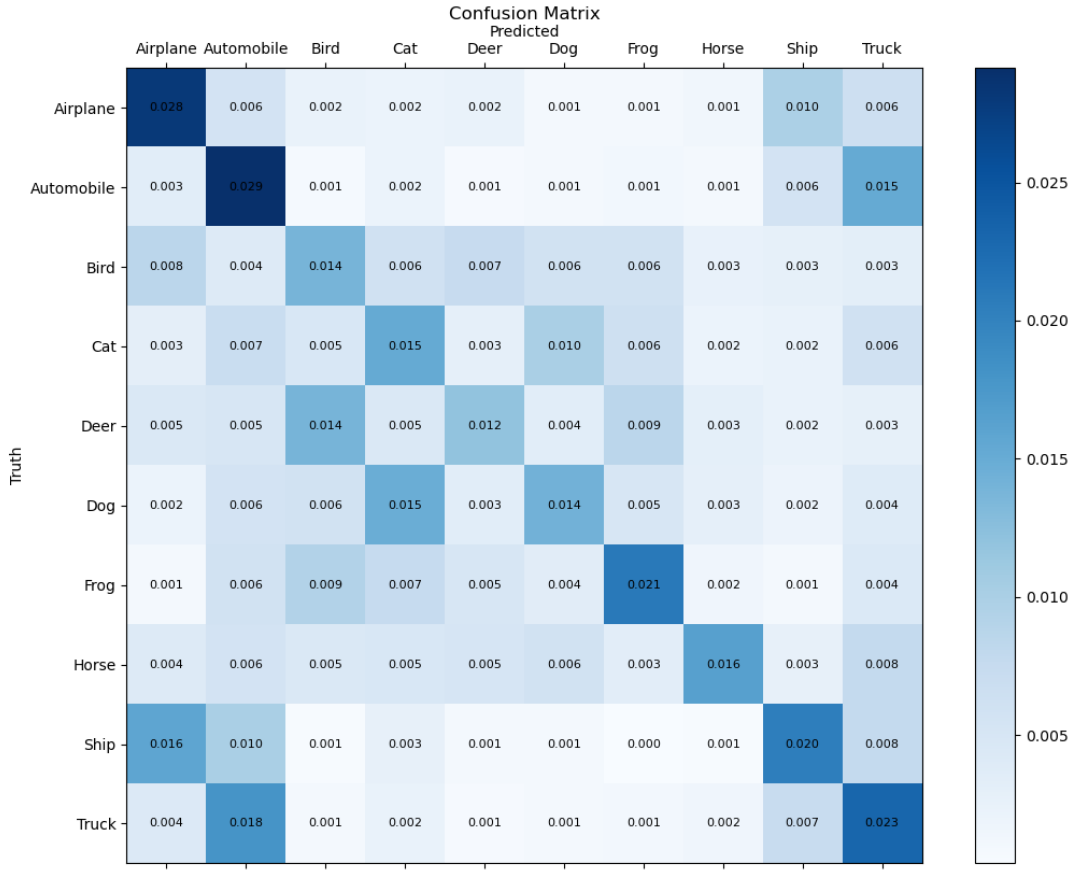
η = 0.01:

Önceki testtekine benzer sonuçlar elde edilmiştir. Hatanın, eğitimin sonuna doğru öncekine kıyasla daha yavaş azaldığı görülmektedir (Figür 41).



Figür 41: Testlerden birinde elde edilen hata grafiği.

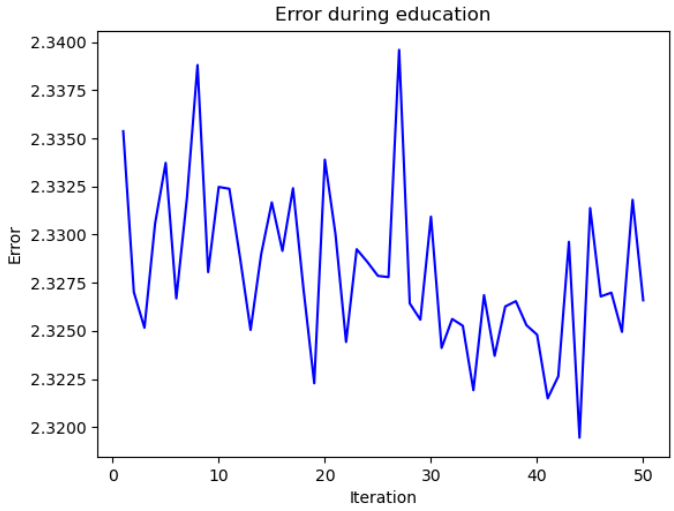
Bu testte hayvanların öncekine kıyasla daha iyi tahmin edildiği fakat taşıtlarda özellikle gemi-uçak ve otomobil-kamyon tahminlerinde karışılıklık yaşandığı görülür (Figür 42).



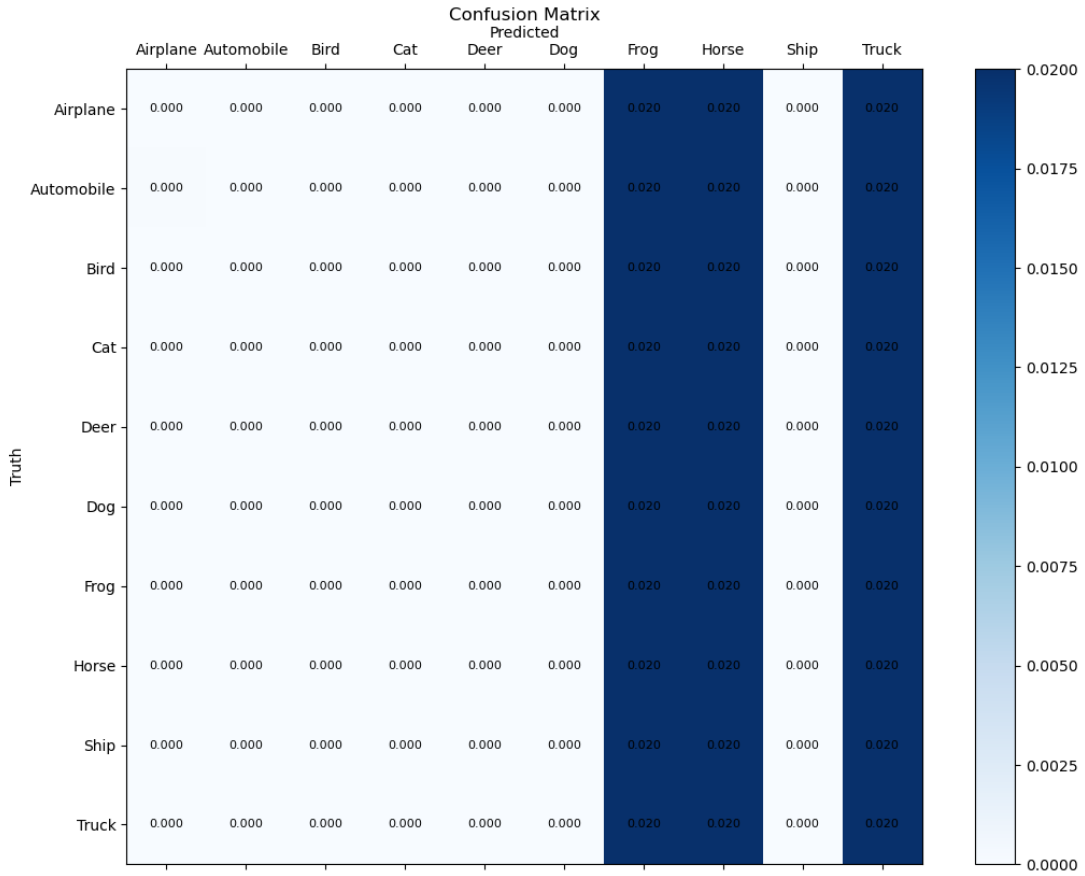
Figür 42: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

η = 0.1:

Bu öğrenme hızının kesinlikle başarısız olduğu görülür. Ağ sınıfları rastgele bir şekilde tahmin etmek zorunda kalmış ve eğitimde bir gelişme göstermemiştir (Figür 43,44).



Figür 43: Testlerden birinde elde edilen hata grafiği.



Figür 44: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

Momentum Oranı (m) Testleri:

Öğrenme hızı ve düşürme oranı sabit tutulurken öğrenme hızının farklı değerleri için testler gerçekleştirilir.

Momentum Oranı	Eğitim Süresi	İterasyon Başına Eğitim Süresi	Eğitim Hatası	Test-Tahmin Hatası	Test Hatası
0	709,0683	14,18137	1,512735	0,673067	2,146603
0,2	700,5862	14,01172	1,499167	0,665333	2,281132
0,4	708,9214	14,17843	1,496306	0,666533	2,188167
0,6	699,3968	13,98794	1,481246	0,661233	2,220591
0,8	712,5114	14,25023	1,466547	0,6554	2,271511
1	711,4078	14,22816	1,488752	0,666533	2,263387

Eğitim süreleri, ağ yapısı değişmediği için yine benzer değerleri gösterir.

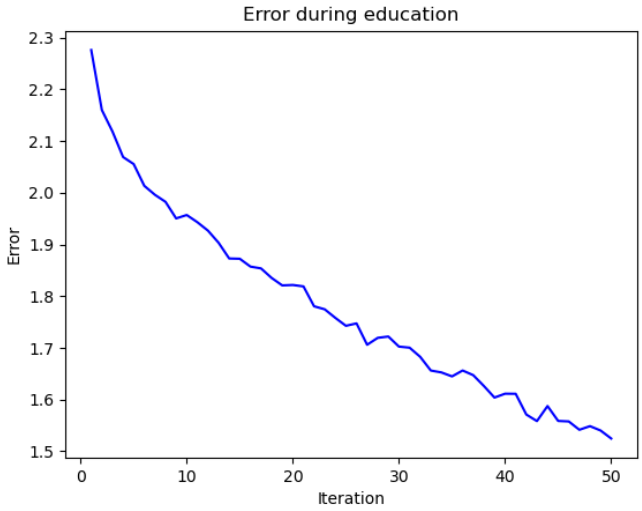
Eğitim hatasında momentumun dahil edilmesinin küçük bir miktar da olsa ağı geliştirdiği görülür. Momentumun hiç olmadığı m = 0 durumunda hata 1.51'ken bu değer m = 0.8'deki 1.47'ye kadar düşmüştür. Bu noktadan sonra bu değer tekrar artmış olup, test edilen değerler arasında eğitim hatasında en başarılısının m = 0.8 değeri olduğu görülür.

Test aşamasındaki tahminlerde de bu durum kendini göstermektedir. Eğitim hatası ve tahmin hatası arasında bir ilişki olduğu görülmektedir. Buna rağmen momentum sonucu fazla etkilememiş olup maksimum %1.77'lik bir değişim görülmüştür.

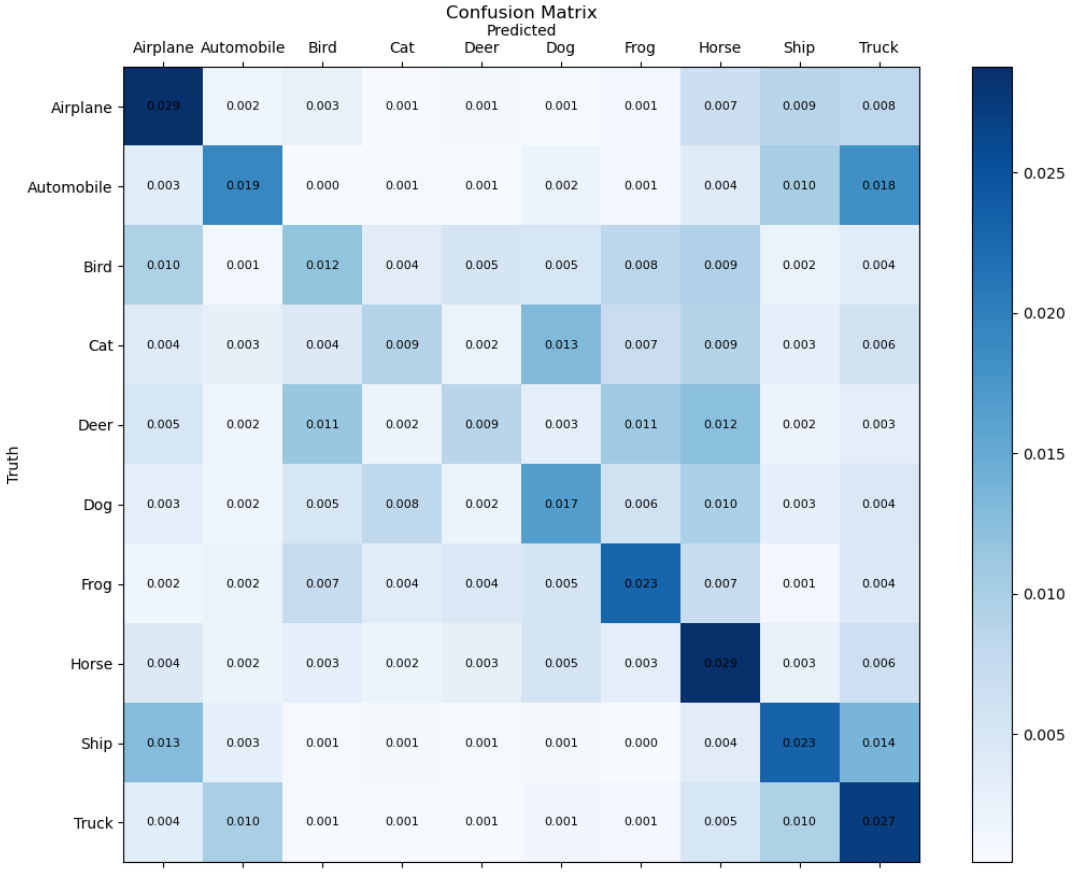
Test aşamasında elde edilen "cross entropy" hatasının yine anlamlı bir değişim yaşamadığı gözlenir.

m = 0:

Eğitim sırasındaki hata ve elde edilen "confusion matrix" matrisleri yorumlanacak derecede anlamlı bir farklılık göstermemektedir. Daha önce de olduğu gibi taşıtlar daha isabetli tahmin edilmekte olup hayvanların tahmininde ağ zorlanmıştır.

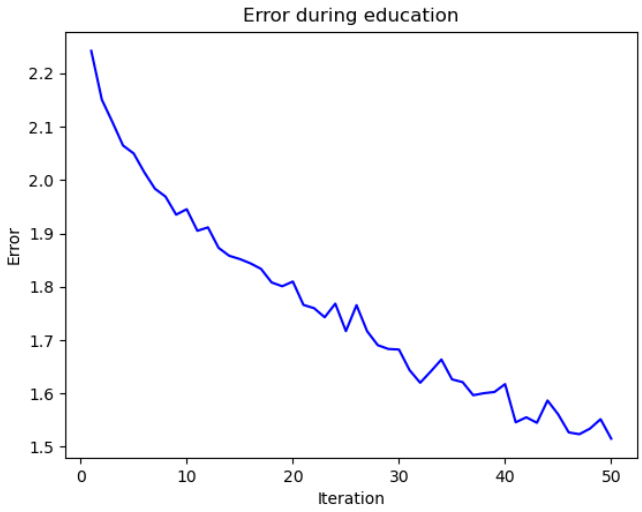


Figür 45: Testlerden birinde elde edilen hata grafiği.

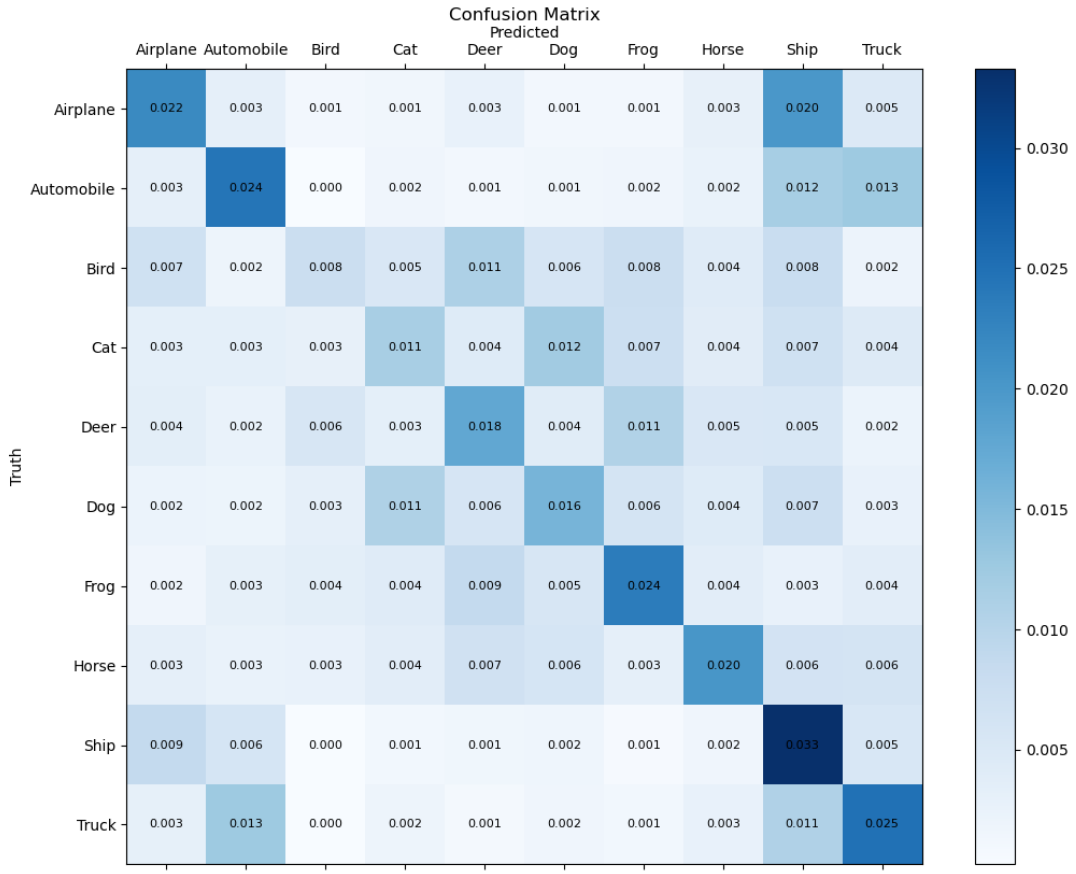


Figür 46: Test sonrası elde edilen ortalama “confusion matrix” matrisi.

m = 0.2:

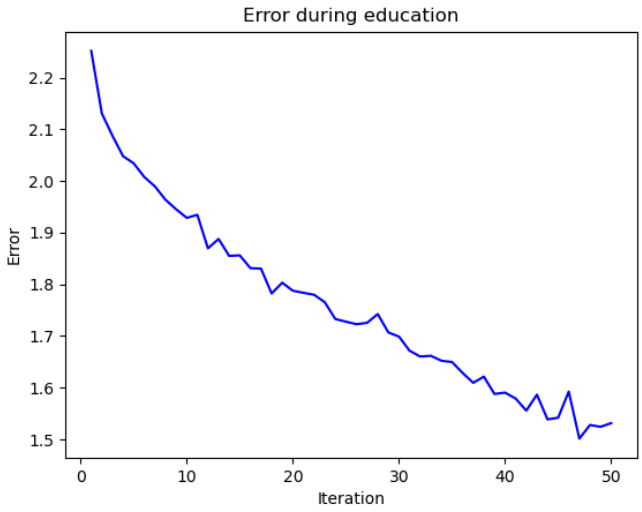


Figür 47: Testlerden birinde elde edilen hata grafiği.

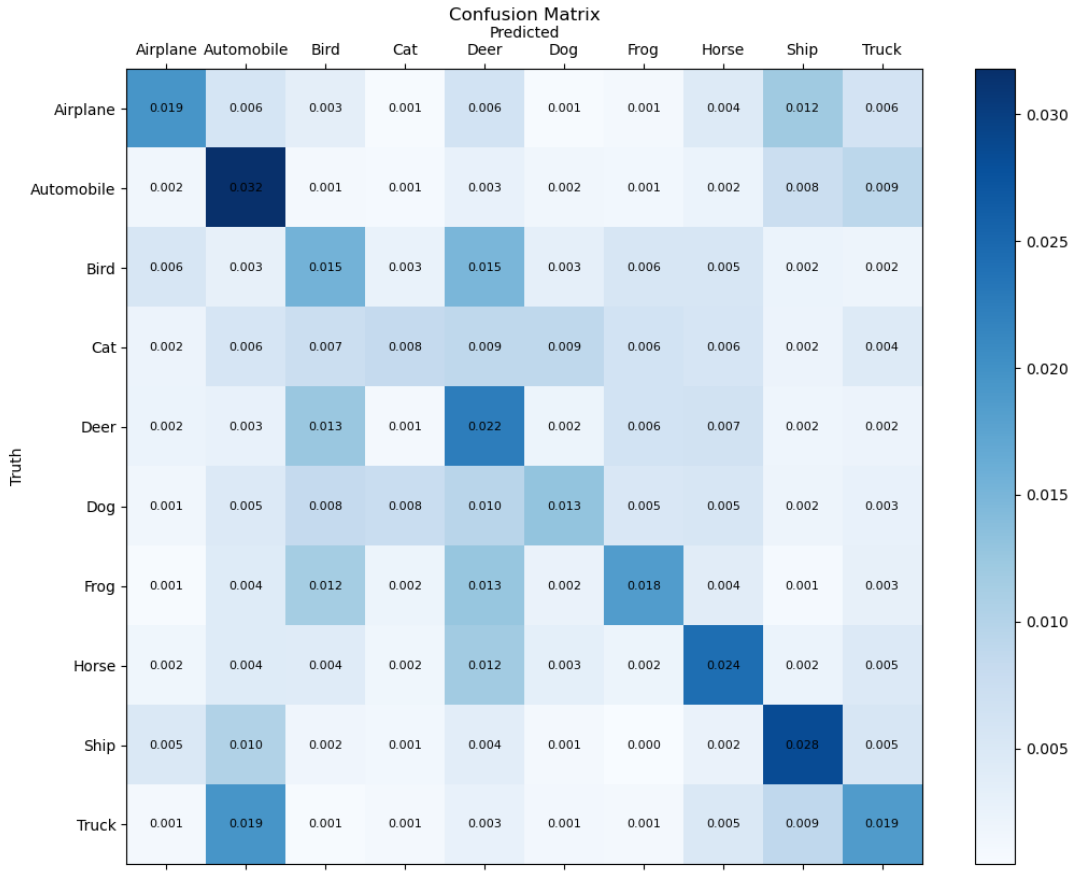


Figür 48: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

m = 0.4:

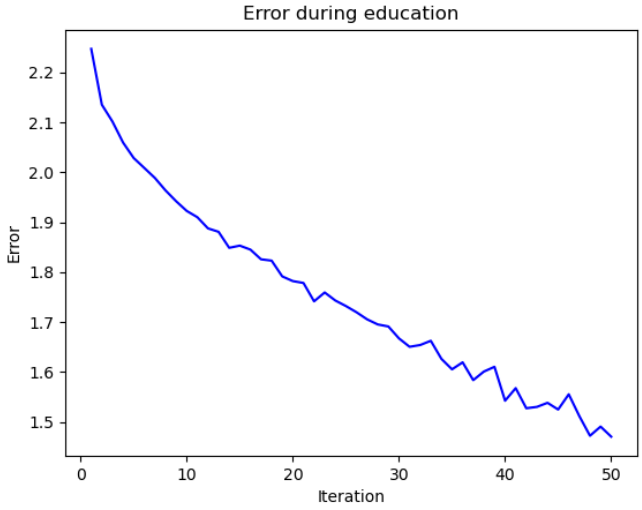


Figür 49: Testlerden birinde elde edilen hata grafiği.

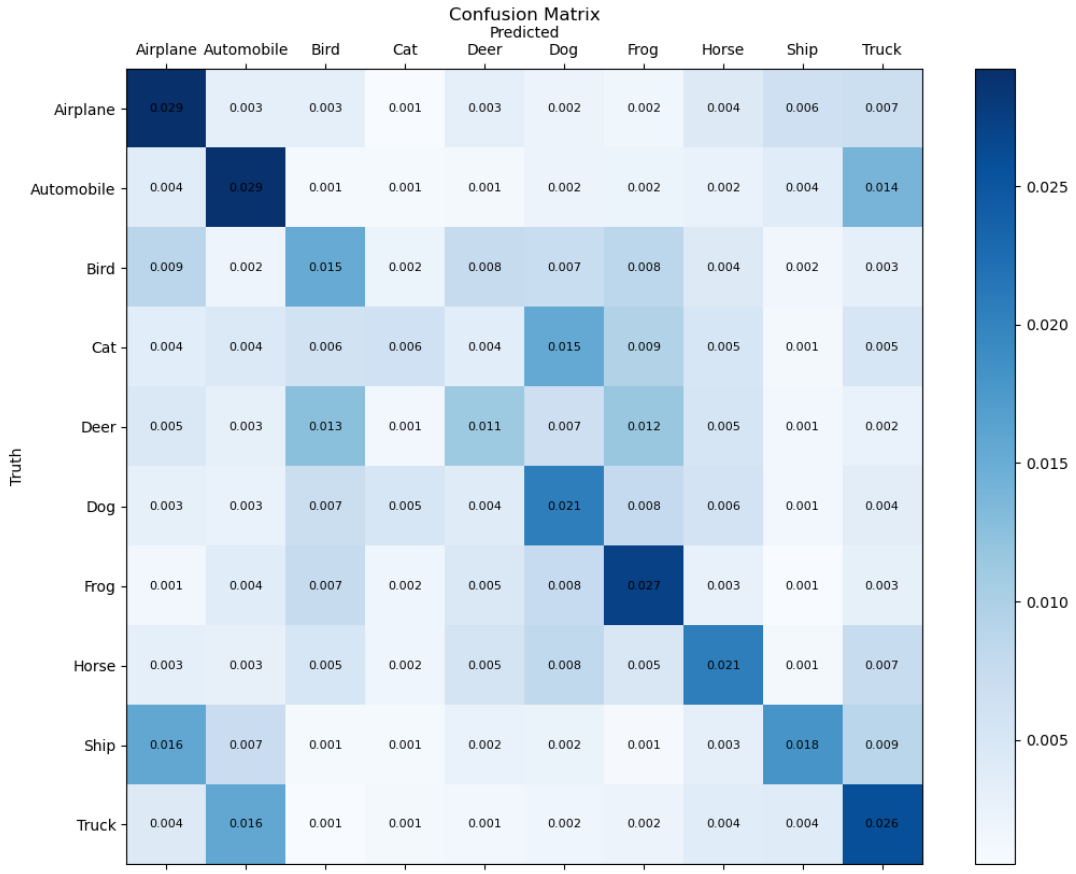


Figür 50: Test sonrası elde edilen ortalama “confusion matrix” matrisi.

m = 0.6:

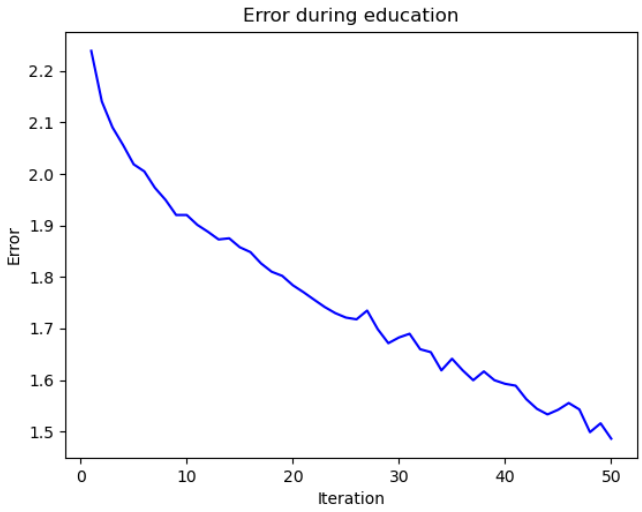


Figür 51: Testlerden birinde elde edilen hata grafiği.

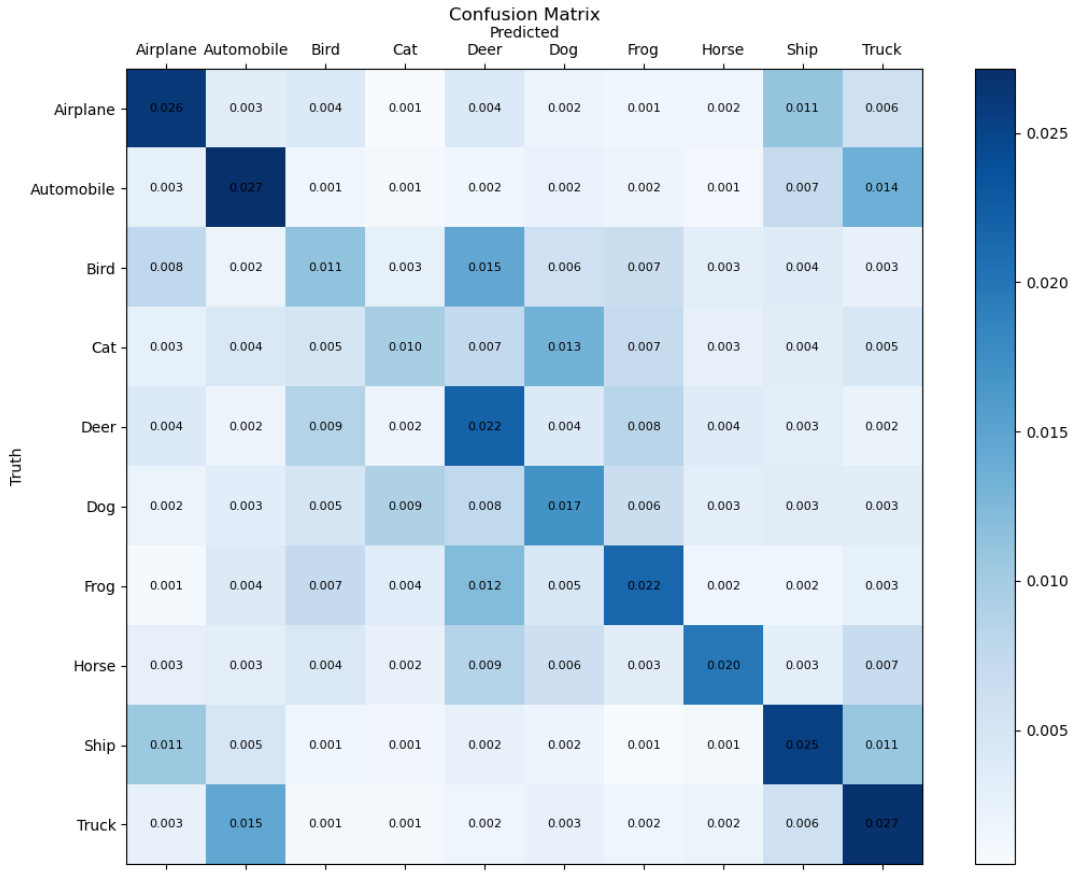


Figür 52: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

m = 0.8:

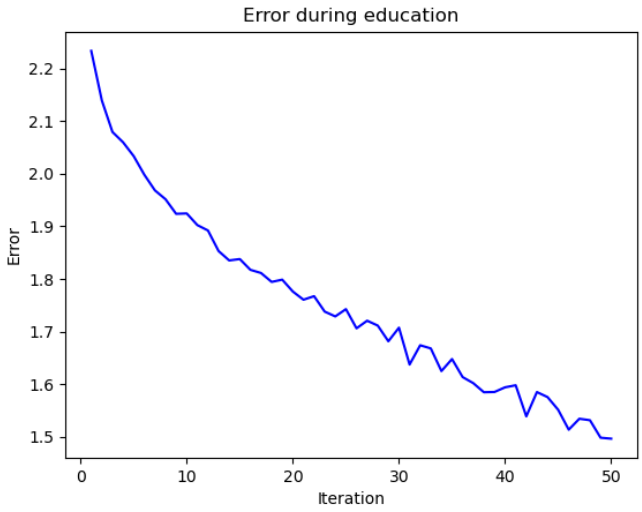


Figür 53: Testlerden birinde elde edilen hata grafiği.

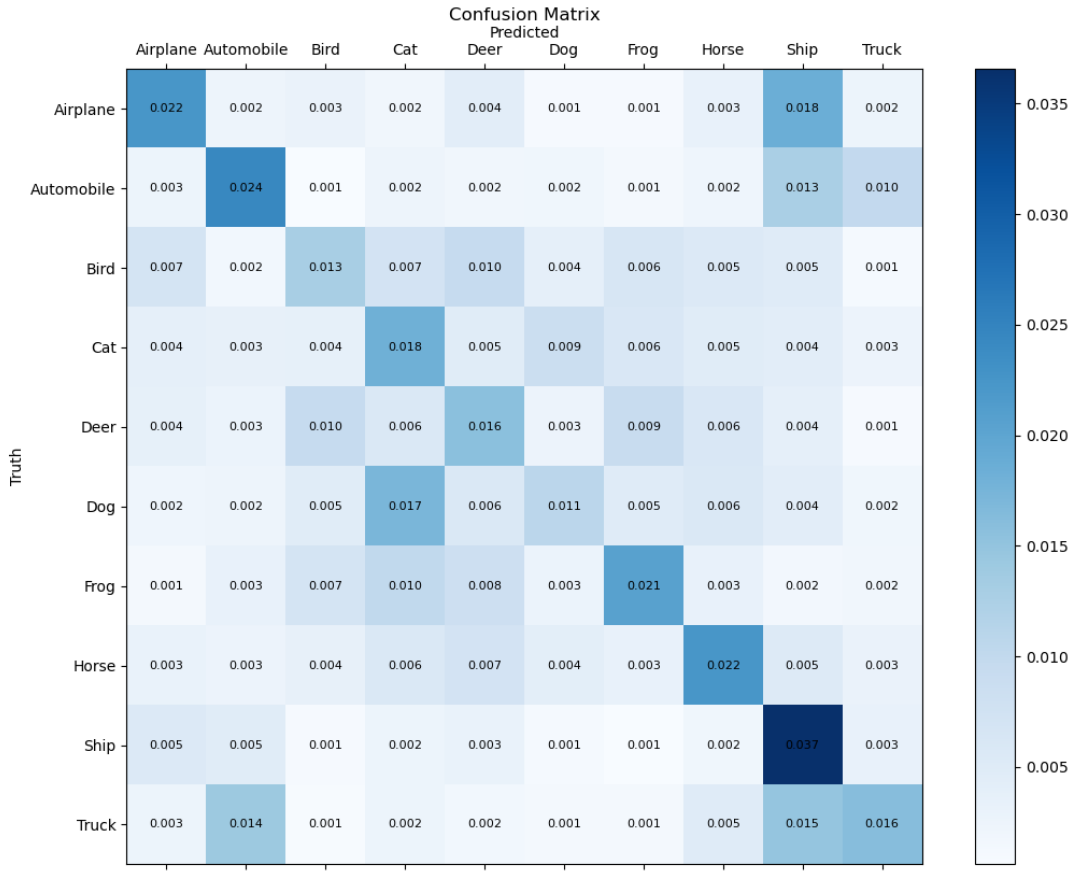


Figür 54: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

m = 1:



Figür 55: Testlerden birinde elde edilen hata grafiği.



Figür 56: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

Düşürme Oranı (d) Testleri:

Momentum ve öğrenme hızı sabit tutulurken öğrenme hızının farklı değerleri için testler gerçekleştirilir.

Düşürme Oranı	Eğitim Süresi	İterasyon Başına Eğitim Süresi	Eğitim Hatası	Test-Tahmin Hatası	Test Hatası
0,5	482,1294	9,642588	1,872237	0,700533	2,50886
0,6	552,6806	11,05361	1,768226	0,6983	2,324807
0,7	635,6835	12,71367	1,62317	0,669267	2,323259
0,8	706,8825	14,13765	1,47654	0,664433	2,358301
0,9	774,0548	15,4811	1,279327	0,680667	2,386735
1	849,0992	16,98198	1,034075	0,675733	2,571362

Eğitim süresinde, düşürme oranına orantılı bir değişim görülür. Düşürülen nöronların ileri yol hesapları ve güncellenmeleri es geçildiği için işlem yükü azaltılmaktadır. d = 0.5 durumunda yarısı düşürülmüş bir ağdaki eğitim süresinin, d = 1 yani tüm nöronların aktif olduğu ağa kıyasla yaklaşık %43.2 daha hızlı çalıştığı görülmektedir. Düşürme, eğitim süresinin artırılmasında fazlasıyla etkili bir yöntemdir.

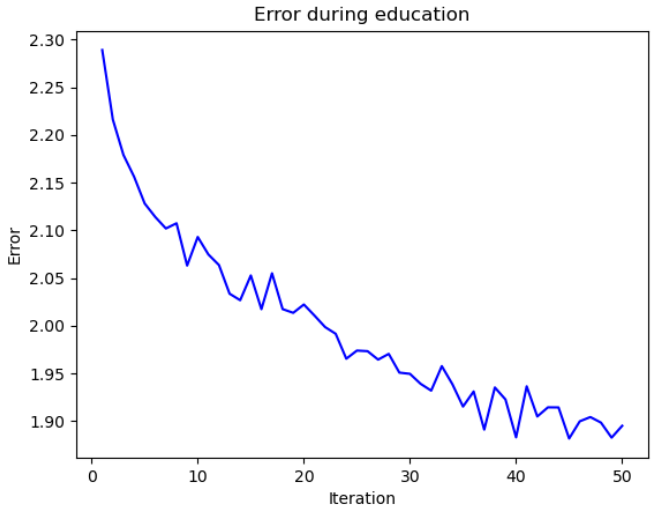
Eğitim hatasında ise düşürmenin sık yaşandığı ağın daha kötü bir performans sergilediği görülür. Düşürme, ağda kritik nöronların bulunmasını engeller ve tüm katmanın çalışmasını sağlar. Bu hatayı artırabilen bir davranıştır.

Test aşamasındaki tahmin hatasına bakıldığında da eğitim hatasıyla ilişkili bir sonuç görülür. Yine de bu fark d = 0.5 ve d = 1 olan ağlar arasında yalnızca %2.53 olup eğitim süresinin iyileştirilmesinden dolayı tercih edilebilebilir.

Testteki “cross entropy” hatası, önceki testlerde olduğu gibi anlamlı bir bilgi sağlamamaktadır.

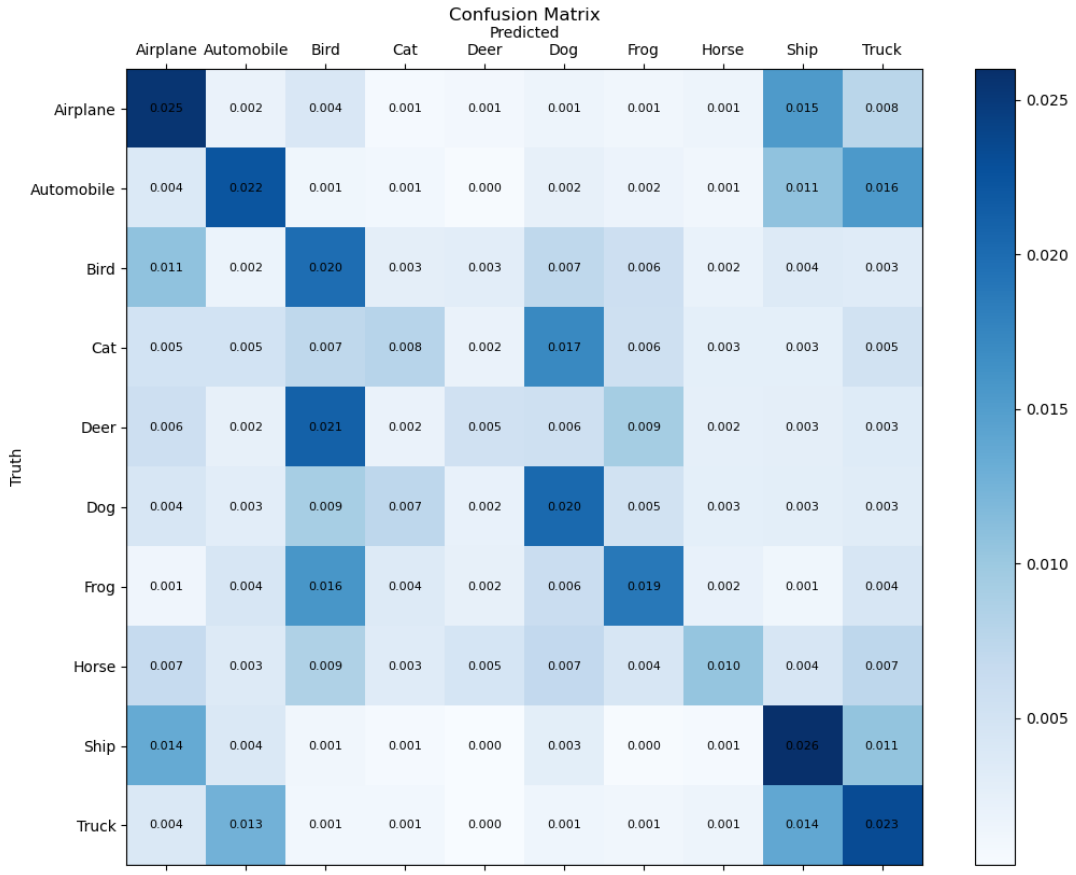
d = 0.5:

Eğitim hatasının düşüşü, düşürmenin sık yaşandığı durumlarda daha dalgalı olup daha yavaş bir azalma görülmektedir. Bu daha önce bahsedilen kritik nöronların bulunmamasından kaynaklanmaktadır.



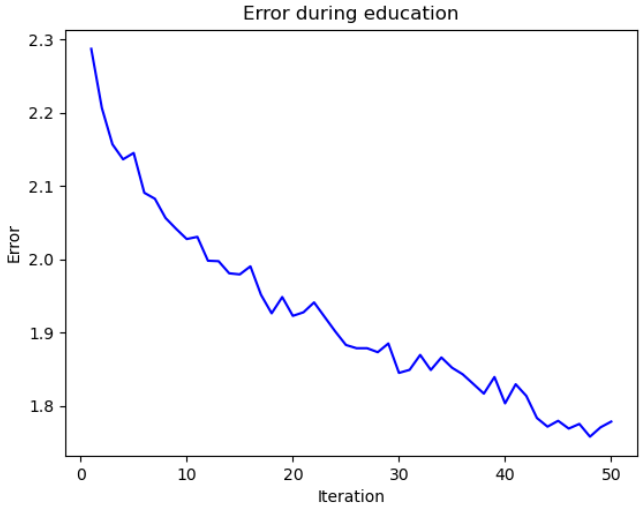
Figür 57: Testlerden birinde elde edilen hata grafiği.

"Confusion matrix" incelendiğinde ise, düşürmenin sık yaşandığı durumlarda genel anlamda daha kötü bir tahmin performansı görülse de birbirine benzer resimlerin tahmininde daha başarılı olunduğu görülür. Daha uzun süreli bir eğitimle düşürmenin uygulandığı ağların yüksek potansiyele sahip olduğu görülür.

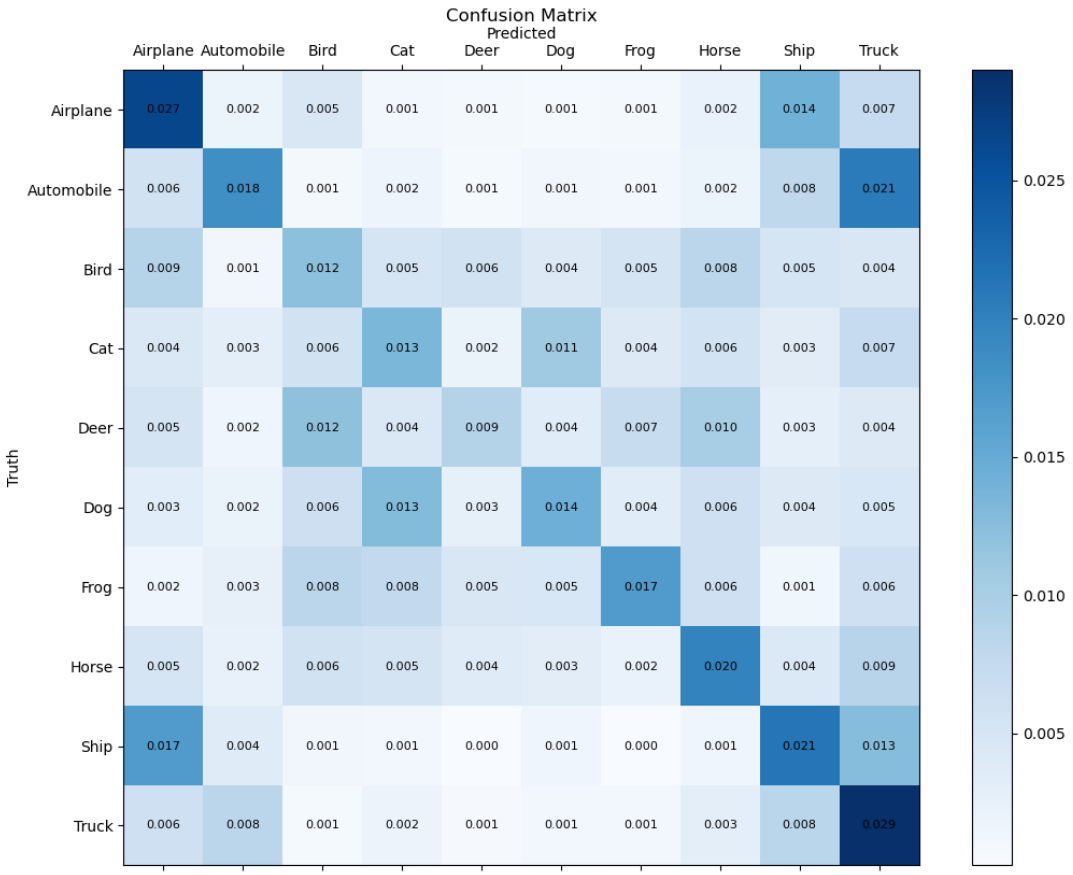


Figür 58: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

d = 0.6:

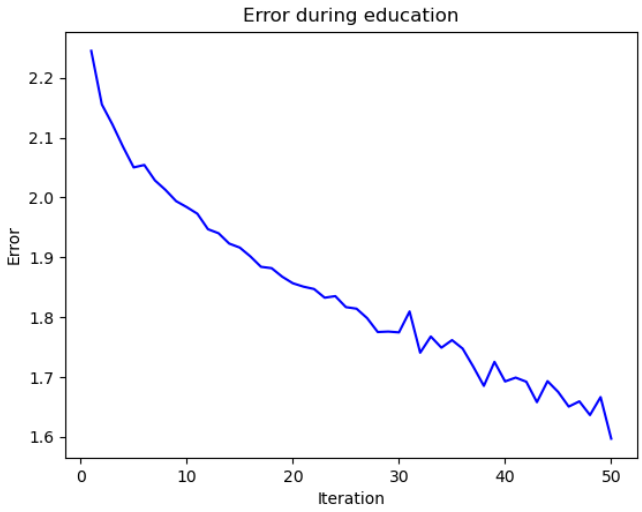


Figür 55: Testlerden birinde elde edilen hata grafiği.

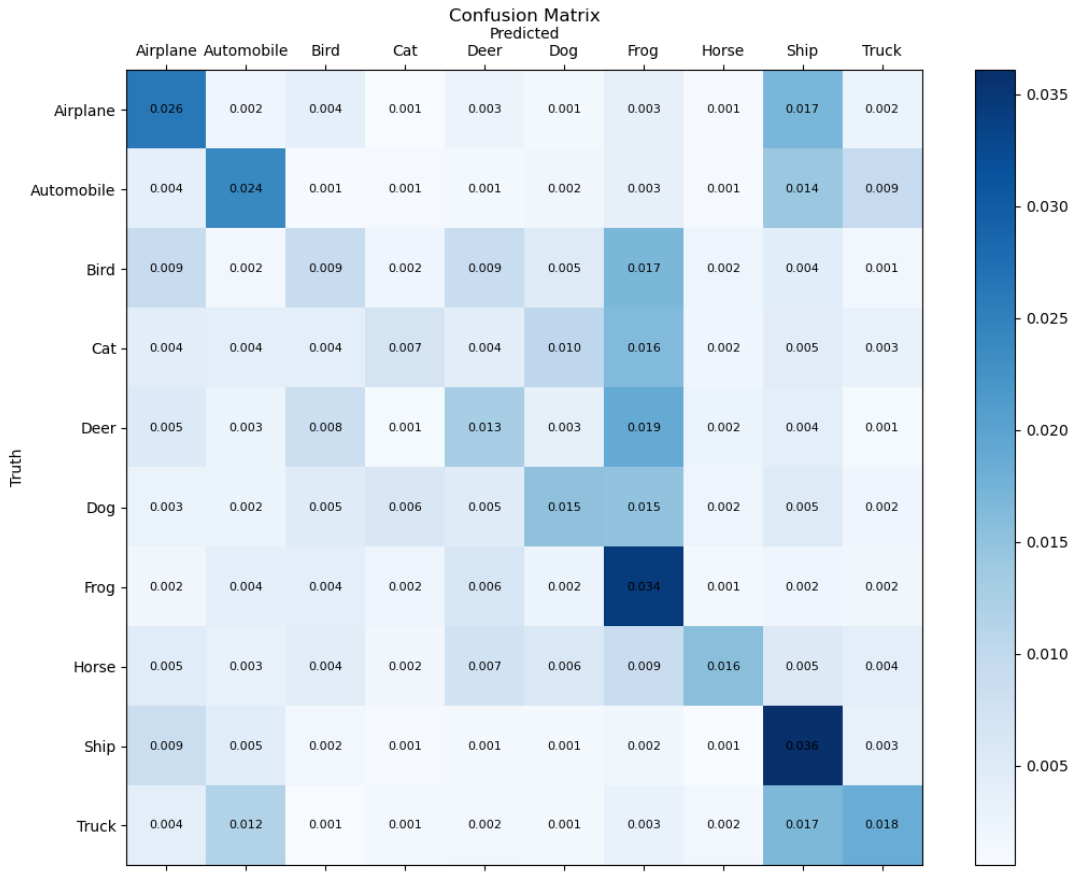


Figür 56: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

d = 0.7:

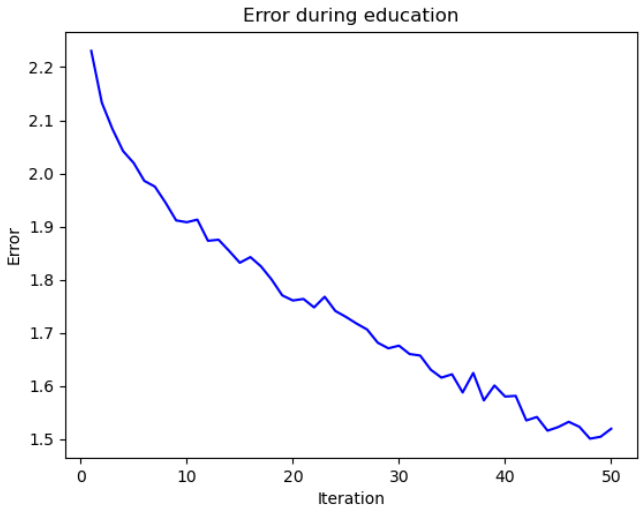


Figür 55: Testlerden birinde elde edilen hata grafiği.

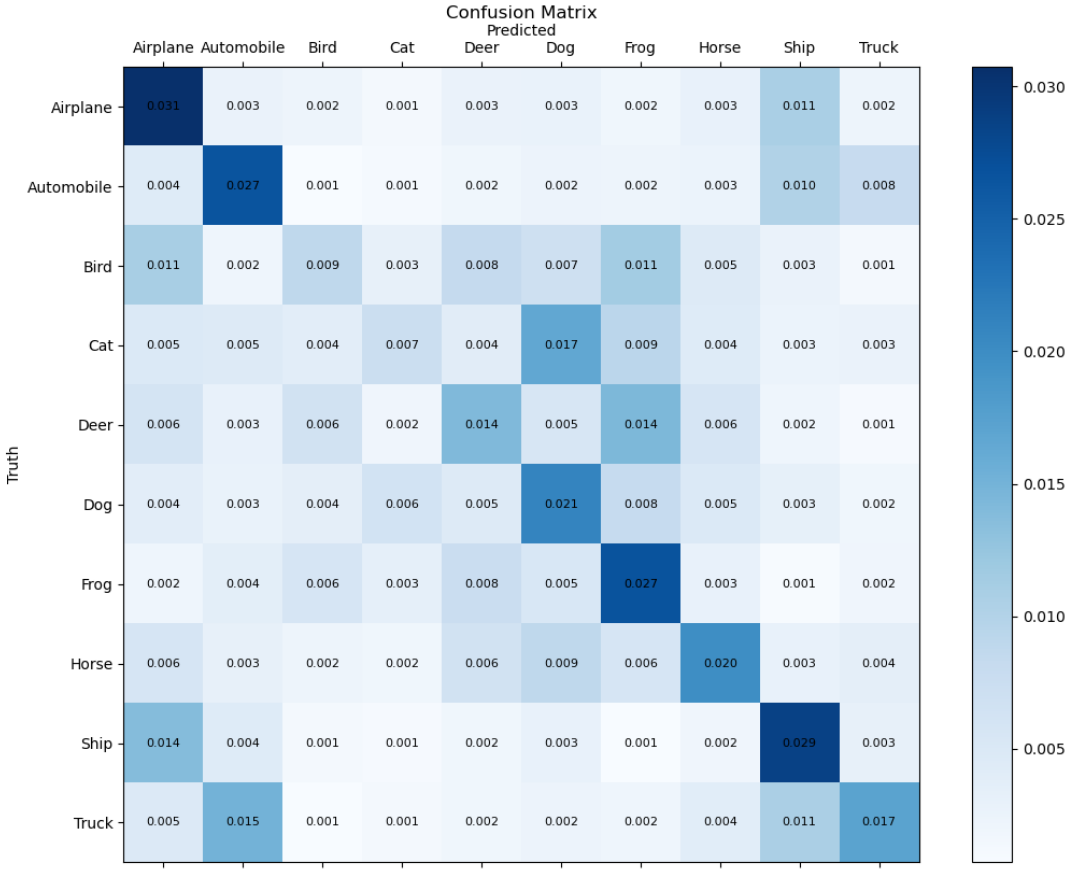


Figür 56: Test sonrası elde edilen ortalama “confusion matrix” matrisi.

d = 0.8:

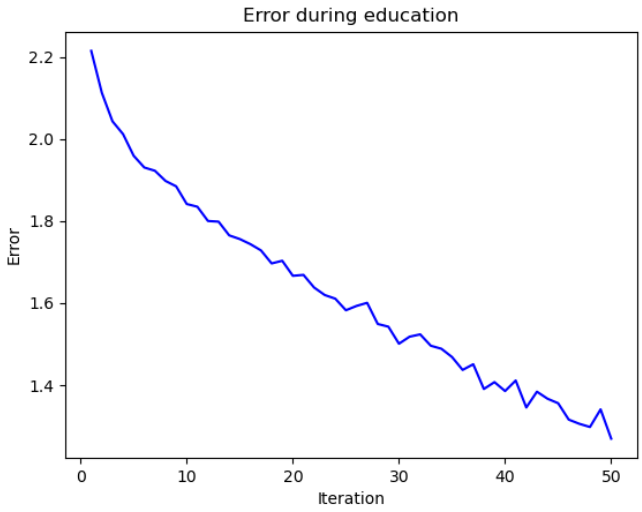


Figür 55: Testlerden birinde elde edilen hata grafiği.

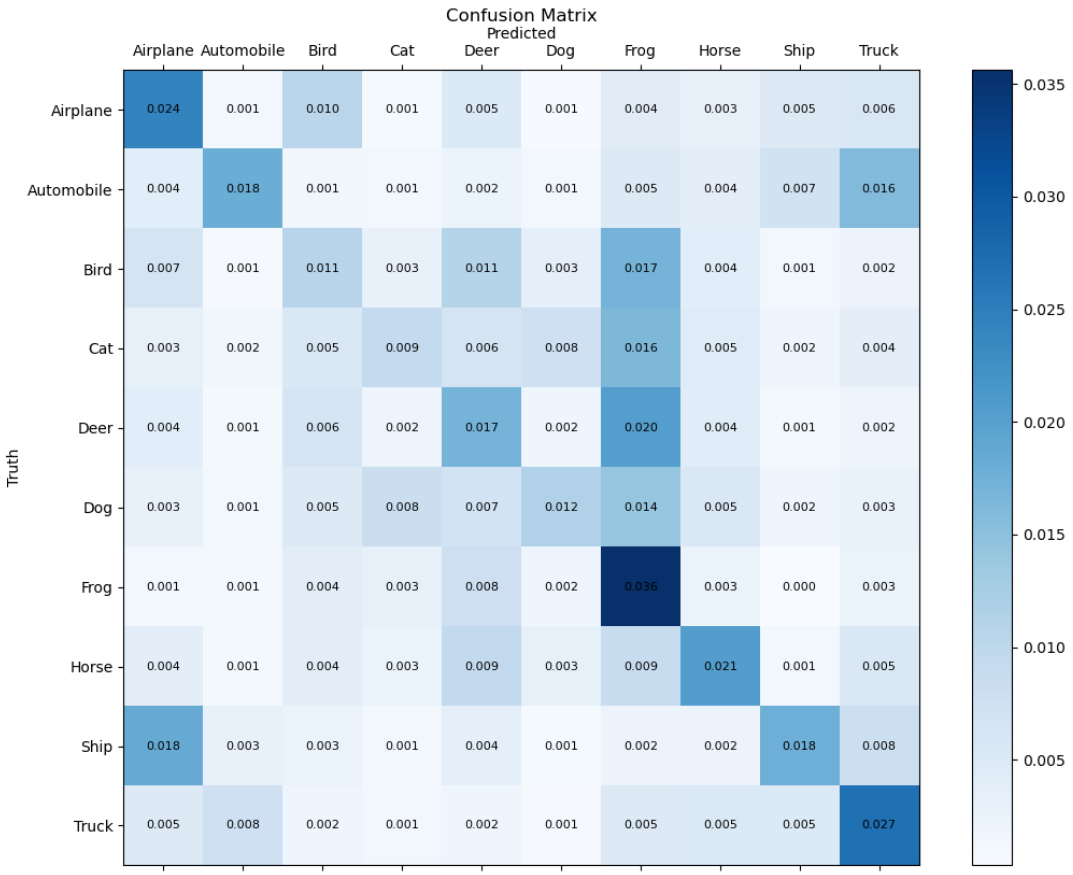


Figür 56: Test sonrası elde edilen ortalama “confusion matrix” matrisi.

d = 0.9:

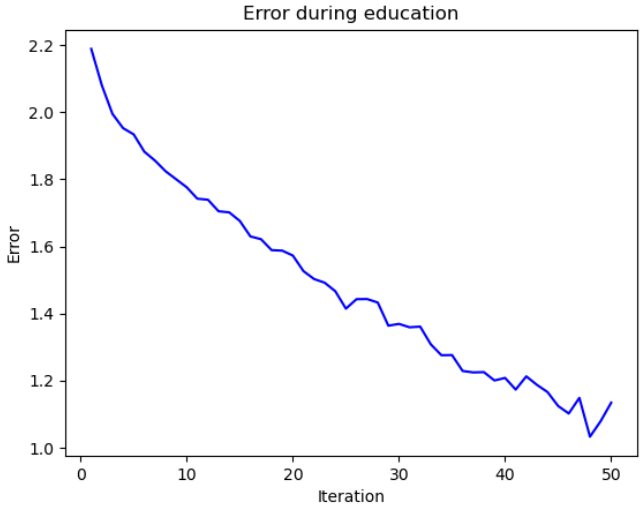


Figür 55: Testlerden birinde elde edilen hata grafiği.

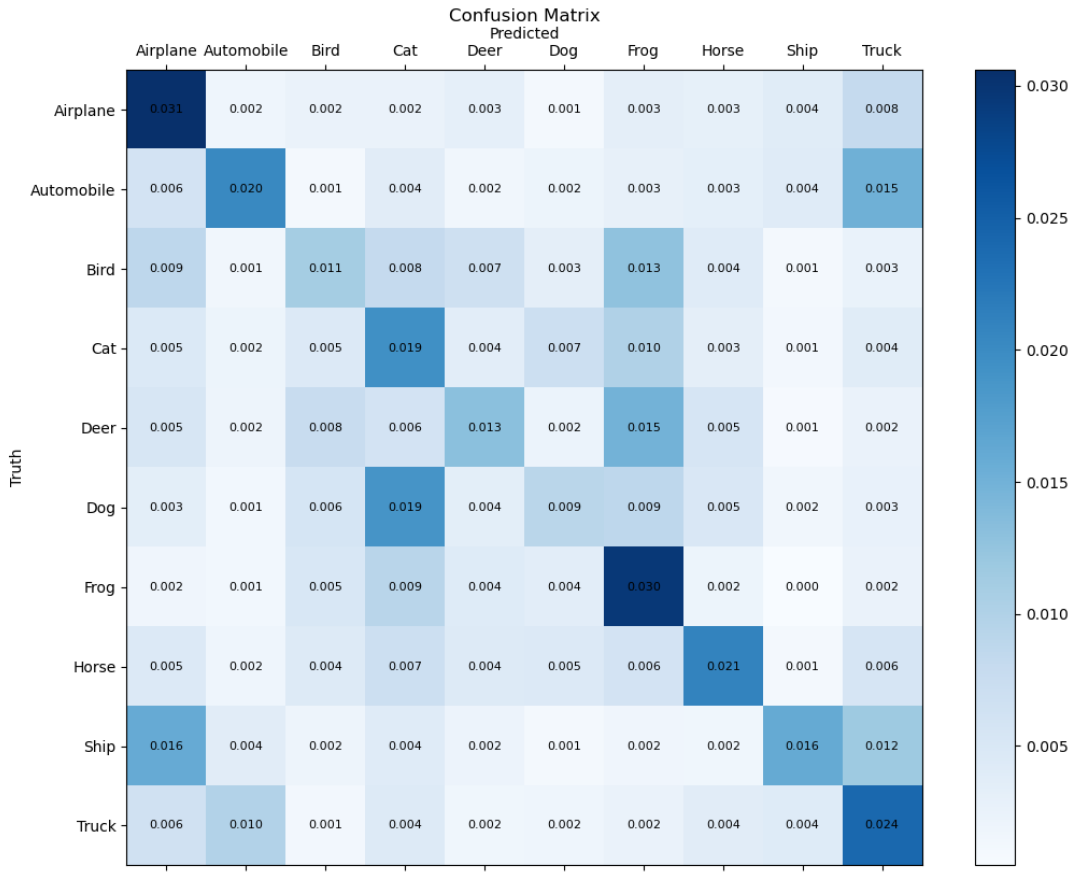


Figür 56: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

d = 1:



Figür 55: Testlerden birinde elde edilen hata grafiği.



Figür 56: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

Ağın İçin Tensorflow İmplementasyonu:

Önceki kısımın üstüne Tensorflow modülü eklenmiştir. Tensorflow, makine öğrenimi ve derin öğrenme görevlerine yönelik yoğun hesaplama çalışmaları için Google Brain Trust tarafından oluşturulan açık kaynaklı bir kitapıktır. Verileri çekmekte, yazdırmakta, çizdirmekte ve farklı verileri toplamakta yardımcı olan "MatPlotLib", "Xlwt", "Pickle" gibi modüller de kullanılmıştır. Modüllerin tam listesi Figür 57'de de görülebilir.

```
import tensorflow as tf
import random
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from matplotlib import cm
import seaborn as sns
import os
import pickle
import xlwt
from xlwt import Workbook
from xlrd import open_workbook
import time
```

Figür 57: Kullanılan modüller.

Veri hazırlama fonksiyonu:

Veri setini kullanım için yüklediğimiz ve kodu ağ için hazırladığımız kısım. Veri setimizde 10 tane sınıf ve 32x32 boyutunda 50000 eğitim ve 10000 test kümesi bulunmaktadır. Verimizi yüklemek için Figür 58'deki kodu kullanıyoruz.

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

Figür 58: Yükleme kodu.

Verimiz 0-255 arasında RGB kanalında olduğu için 255'e bölerek verimizi normalize ediyoruz böylece verimizi 0-1 arasında ölçekleyebiliyoruz (Figür 59).

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Figür 59: Normalizsyon Kodu.

Daha sonra eğitim kümemizi gruplar halinde küçültüyoruz. ‘def batchset_Arr’ Tek bir sınıftan kaç verinin alınacağını belirler, girilen bir veri listesinden her sınıftan "batch_size" sayısında veri yeni bir listeye eklenip döndürülür (Figür 60). Her sınıftan 200 veri kullanıldı.

```
def batchset_Arr(dataArr, labelArr, batch_size, class_count):
    # Veri kümesini daha küçük kümelere ayıran fonksiyon. "batch_size" değeri
    # tek bir sınıftan kaç verinin alınacağını belirler. Girilen bir veri listesinden
    # her sınıftan "batch_size" sayısında veri yeni bir listeye eklenip döndürülür.
    newdataArr = np.zeros((int(batch_size*class_count), 32, 32, 3), dtype=np.float64)
    newlabelArr = np.zeros((int(batch_size*class_count), 1), dtype=np.uint8)
    batchArr = np.zeros(class_count)
    i = 0
    for data in dataArr:
        j = labelArr[i]
        if batchArr[j] < batch_size:
            newdataArr[i, :, :, :] = data
            newlabelArr[i, 0] = labelArr[i, 0]
            batchArr[j] += 1
            i += 1
    return newdataArr, newlabelArr
```

Figür 60: Küme fonksiyonu.

Yapay sinir ağı oluşturma fonksiyonu:

Ağ oluşturmak ‘Softmax’ ve ‘ReLu’ fonksiyonları kullanılmıştır (Figür 61). Softmax, genellikle bir ağın son katmanında kullanılan lojistik regresyonun bir genellemesidir. Belirli bir çıktının birçok farklı şeyin bir listesi olabileceği çoklu sınıflandırma modellerine yardımcı olur. Ek olarak belirli bir sınıfa ait çıktının olasılığını veren 0 ile 1 arasında değerler sağlar. Düzeltilmiş lineer aktivasyon fonksiyonu ‘ReLU’, pozitif ise girişi doğrudan çıkaran parçalı bir lineer fonksiyondur, aksi takdirde sıfır çıktı verir. Birçok sinir ağı türü için varsayılan etkinleştirme işlevi haline gelmiştir çünkü onu kullanan bir modelin eğitilmesi daha kolaydır ve genellikle daha iyi performans elde eder.

```
def Create_Train_Test_Network(lrate, mrate, drate, epoch, h1n, h2n, train_images, train_labels, test_images, test_labels, test_iter, savefolder):
    if not os.path.exists(savefolder):
        os.makedirs(savefolder)
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(32, 32, 3)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(1 - drate),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dropout(1 - drate),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
```

Figür 61: Ağ fonksiyonu.

Bu ağdaki ilk katman olan ‘tf.keras.layers.Flatten’ , görüntülerin formatını iki boyutlu bir diziden (32 x 32 piksel) tek boyutlu bir diziye dönüştürür. Bu katmanda öğrenilecek parametre yoktur,

Yalnızca verileri yeniden biçimlenririr. Pikseller düzleştirildikten sonra, ağ 'tf.keras.layers.Dense' katman 'tf.keras.layers. Dense' oluşur. Bunlar yoğun şekilde bağlı veya tamamen bağlantılı sinir katmanlarıdır.

Modelin eğitime hazır olması için bir kaç fonksiyon daha kullanmamız gerekmektedir kullanılan fonksiyonları Figür 62 'de görebilirsiniz.

-Kayıp işlevi — Bu, modelin eğitim sırasında eğitimin ne kadar doğru olduğunu ölçer. Modeli doğru yöne "yönlendirmek" için bu işlevi en aza indirmek istenilir.

-Uygunlaştırıcı (Optimizer) — Model, gördüğü verilere ve kayıp işlevine göre nasıl güncellenir.

-Metrikler — Eğitim ve test adımlarını izlemek için kullanılır. Aşağıdaki örnek, doğru şekilde sınıflandırılan görüntülerin oranı olan doğruluğu kullanır.

```
opt = tf.keras.optimizers.SGD(learning_rate=lrate, momentum=mrate)

start_time = time.time()
model.compile(optimizer=opt,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
ed_time = time.time() - start_time

history = model.fit(train_images, train_labels, epochs=epoch,
                    validation_data=(test_images, test_labels))
```

Figür 62: Kayıp, Uygunlaştırma ve Metrik Fonksiyonları.

'model.fit()' fonksiyonuna ilk olarak eğitim verisindeki girdi değerlerimizi, ikinci olarak ise çıktı değerlerimizi veriyoruz. Verdiğimiz "epoch" parametresi ise modelin kaç kere ileri ve geri yayılım yapacağını belirliyoruz. Ardından model öğrenme aşamasına geçiyor.

Ağ sonuçlarının incelenmesi için kullanılacak "confusion matrix" fonksiyonunu Figür 63'de gösterilmektedir.

```
y_pred = np.argmax(model.predict(test_images), axis=1)
y_true = test_labels

confusion_mtx = tf.math.confusion_matrix(y_true, y_pred)

plt.figure(figsize=(12.8, 9.6))
sns.heatmap(confusion_mtx, xticklabels=class_names, yticklabels=class_names,
            annot=True, fmt='g')
plt.xlabel('Prediction')
plt.ylabel('Truth')
plt.savefig(os.path.join(savefolder, str(test_iter) + '_Conf_Mtx.png'))
plt.close()
```

Figür 63: Confusion matrix fonksiyonu.

Test Fonksiyonları:

Ağın değişkenlerinin nasıl etkilediğini anlamak için (Öğrenme Hızı (Figür 64), Momentum oranı (Figür 65), Düşürme oranı (Figür 66)) değişkenleri belirleyip sırasıyla değerlerini değiştirerek sonuçlarını inceledik.

```
resultfolder = os.path.join('TF', 'Results')
results = []
lrateT = (0.0001, 0.001, 0.01, 0.1)
for i in range(4):
    lrate = lrateT[i]
    mrate = 0.8
    drate = 0.8
    savefolder = os.path.join(resultfolder, 'Learning_Rate_' + str(lrate))
    for j in range(3):
        result = Create_Train_Test_Network(lrate, mrate, drate, epoch, h1n, h2n, train_images, train_labels, test_images, test_labels, j, savefolder)
        results.append(result)
saveResults(results, resultfolder, 'Results_LRate.xls')
```

Figür 64: Öğrenme Hızı.

```
results = []
mrateT = (0, 0.2, 0.4, 0.6, 0.8, 1)
for i in range(6):
    lrate = 0.01
    mrate = mrateT[i]
    drate = 0.8
    savefolder = os.path.join(resultfolder, 'Momentum_Rate_' + str(mrate))
    for j in range(3):
        result = Create_Train_Test_Network(lrate, mrate, drate, epoch, h1n, h2n, train_images, train_labels, test_images, test_labels, j, savefolder)
        results.append(result)
saveResults(results, resultfolder, 'Results_MRate.xls')
```

Figür 65: Momentum oranı.

```
drateT = (0.5, 0.6, 0.7, 0.8, 0.9, 1)
for i in range(6):
    lrate = 0.01
    mrate = 0.8
    drate = drateT[i]
    savefolder = os.path.join(resultfolder, 'Dropout_Rate_' + str(drate))
    for j in range(3):
        result = Create_Train_Test_Network(lrate, mrate, drate, epoch, h1n, h2n, train_images, train_labels, test_images, test_labels, j, savefolder)
        results.append(result)
saveResults(results, resultfolder, 'Results_DRate.xls')
```

Figür 66: Düşürme oranı

Eğitim ve Test Sonuçları:

Eğitim ve test aşamasında elde edilen sonuçlar, ağın bu problemde gösterdiği performansın incelenmesi için karşılaştırılır. Yapılan testler;

- Öğrenme hızı = {0.0001, 0.001, 0.01, 0.1}
- Momentum oranı = {0, 0.2, 0.4, 0.6, 0.8, 1}
- Düşürme oranı = {0.5, 0.6, 0.7, 0.8, 0.9, 1}

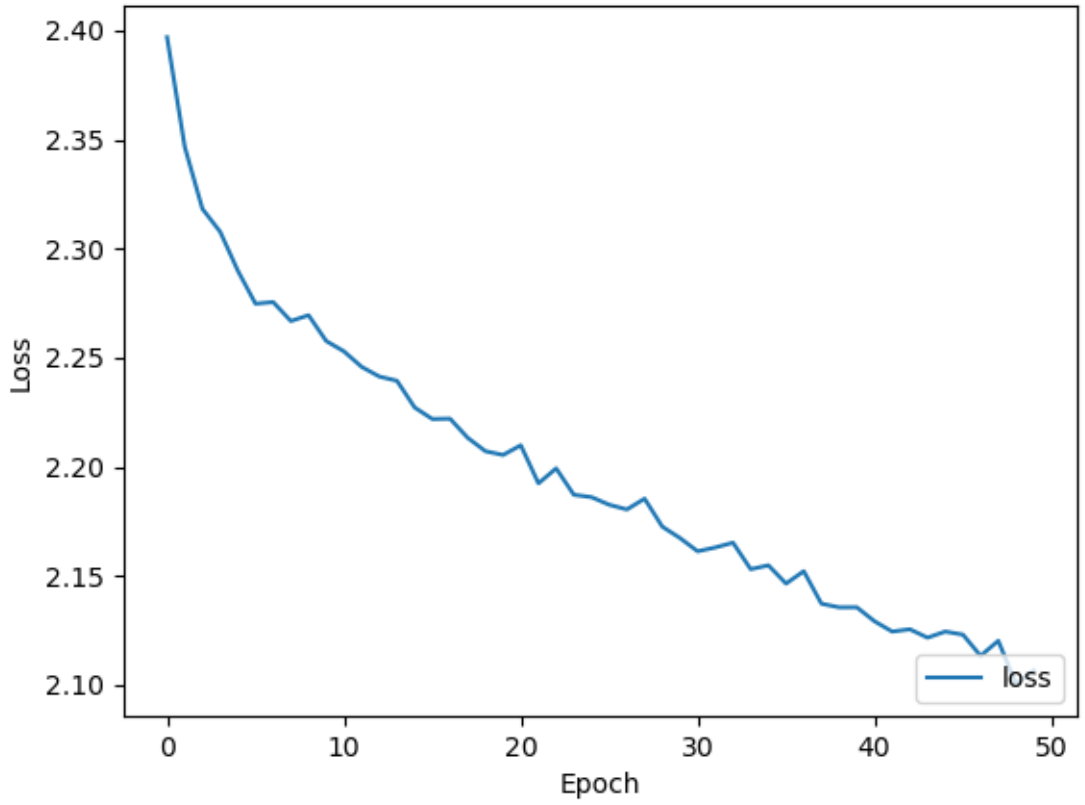
değerleriyle oluşturulan ağlar ile eğitim ve test aşamasının gerçkeleşmesini kapsar. Her seferinde bir değer değişken olup değişken olmayanlar öğrenme hızı = 0.01, momentum oranı 0.8, düşürme oranı = 0.8 değerlerine sahiptir.

Öğrenme Hızı (η) Testleri:

Momentum ve düşürme sabit tutulurken öğrenme hızının farklı değerleri için testler gerçekleştirilir.

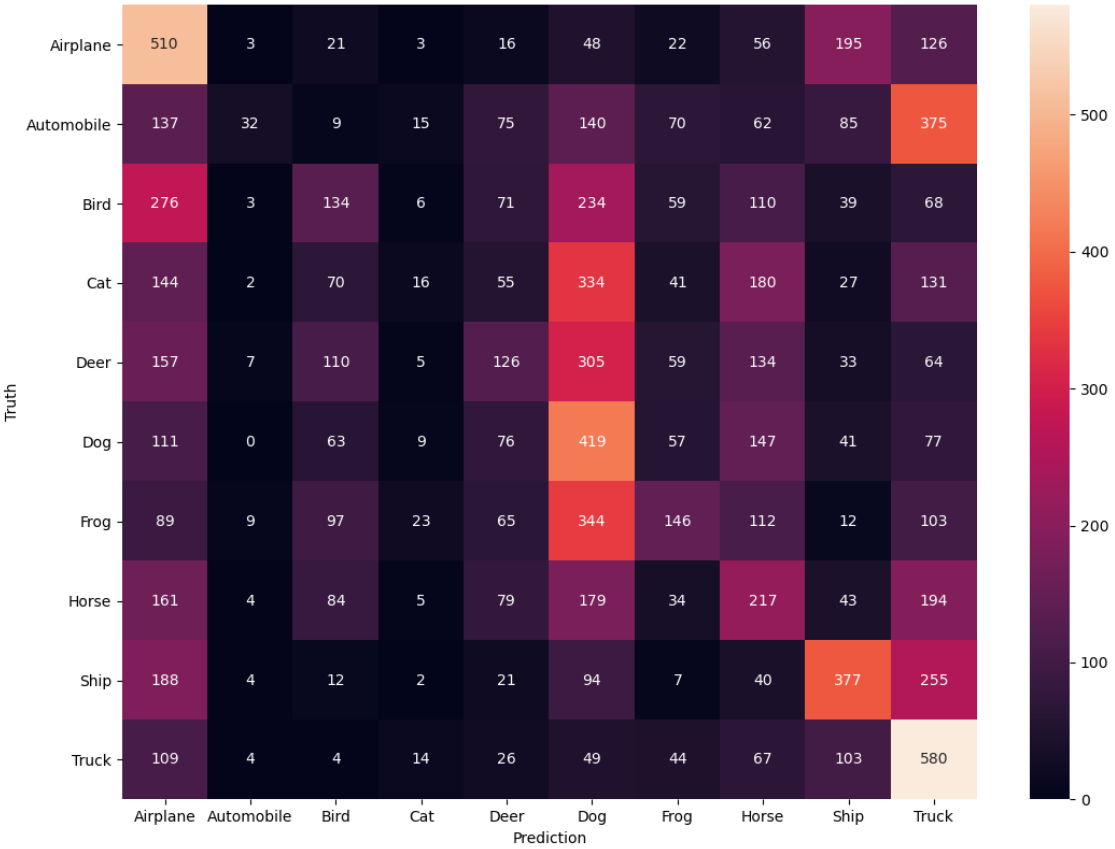
η = 0.0001:

Öğrenme hızı çok düşük olup seçilen sürede yeterince başarılı bir öğrenme gözlenmediği görülür. Eğitim bittiğinde hata düşmeye devam etse de çok yavaş bir düşüş görülmüştür (Figür 67).



Figür 67: Testlerden birinde elde edilen hata grafiği.

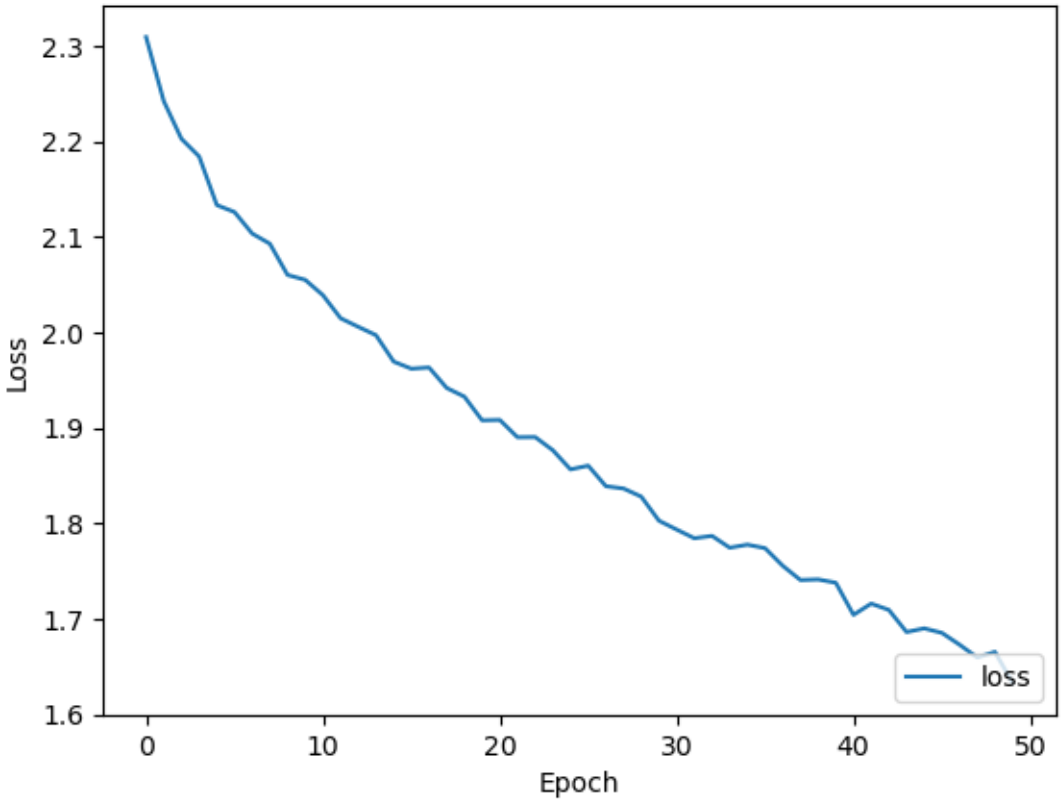
"Confusion matrix" matrisine bakıldığında uçak, köpek, gemi ve kamyon resimlerinin daha doğruluğu yüksek olduğu görülmektedir. Birbirine benzer yapısından dolayı otomobil ve tır resimlerinin birbiriyle karıştığı görülmüştür. (Figür 68).



Figür 68: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

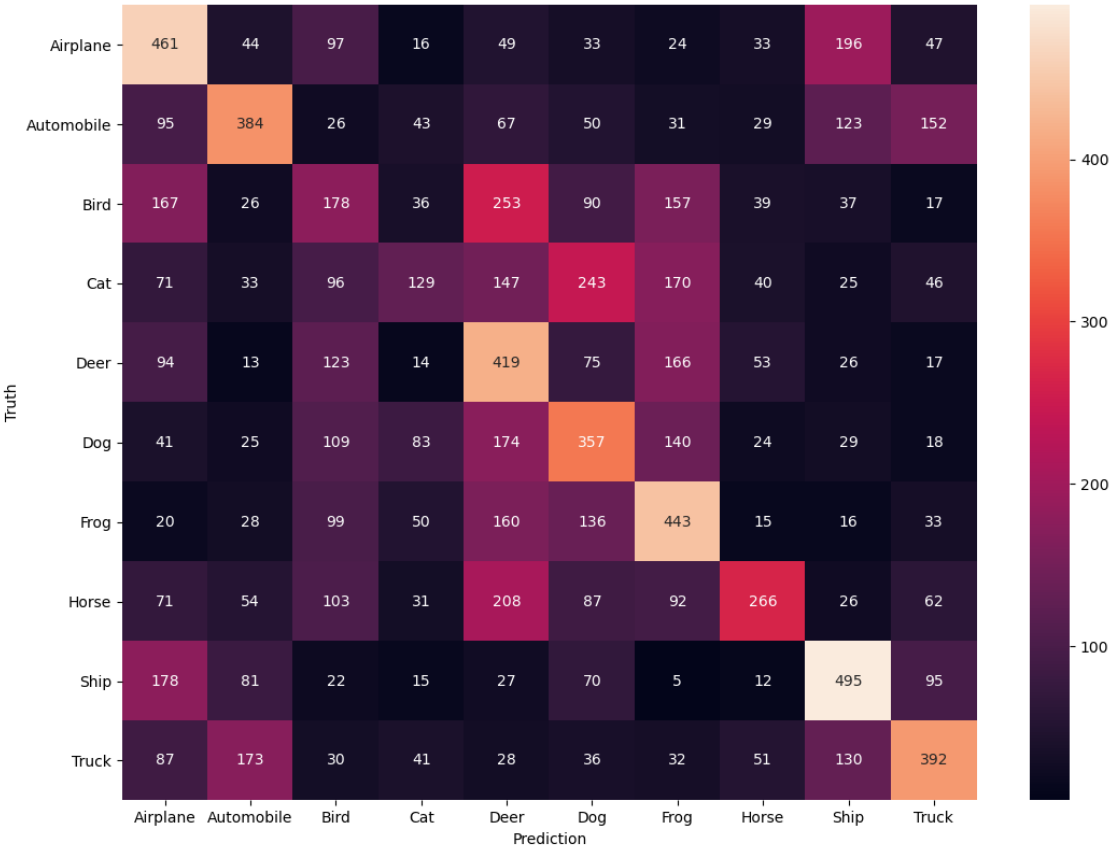
η = 0.001:

Diğer hızlarla karşılaştırıldığında en doğruluğu yüksek olan hız gözükmektedir (Figür 69).



Figür 69: Testlerden birinde elde edilen hata grafiği.

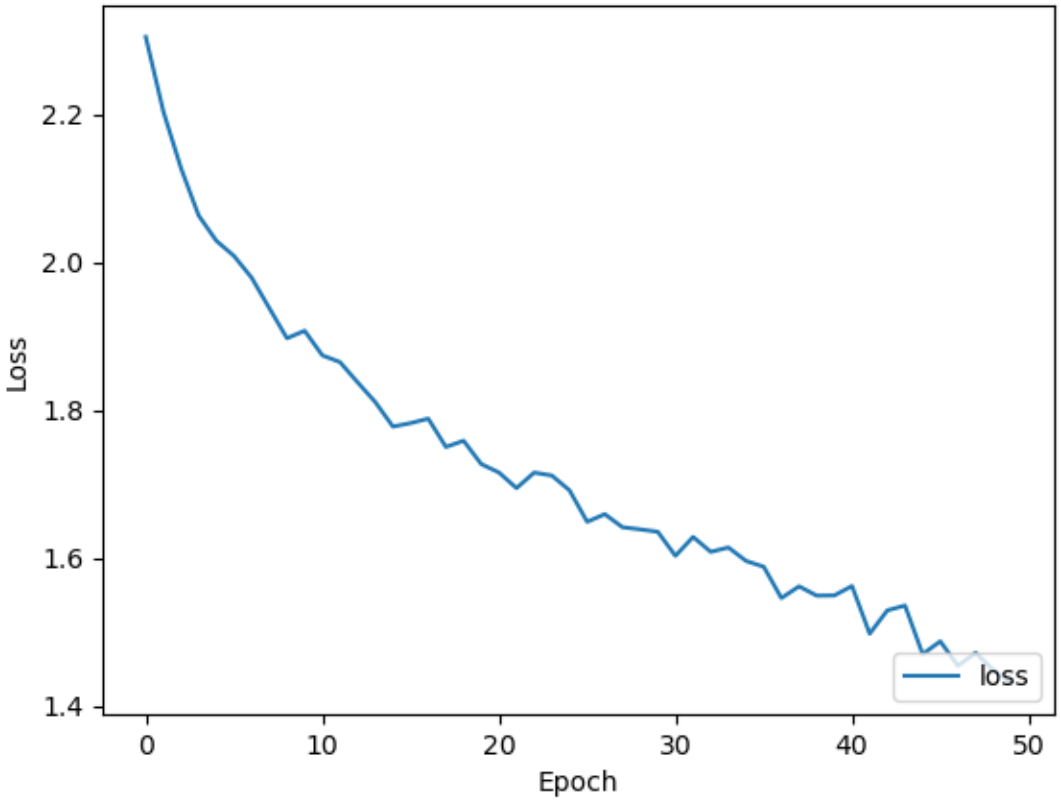
"Confusion matrix" matrisine bakıldığında daha çok resmin birbirine benzetildiği incelenmiştir (Figür 70).



Figür 70: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

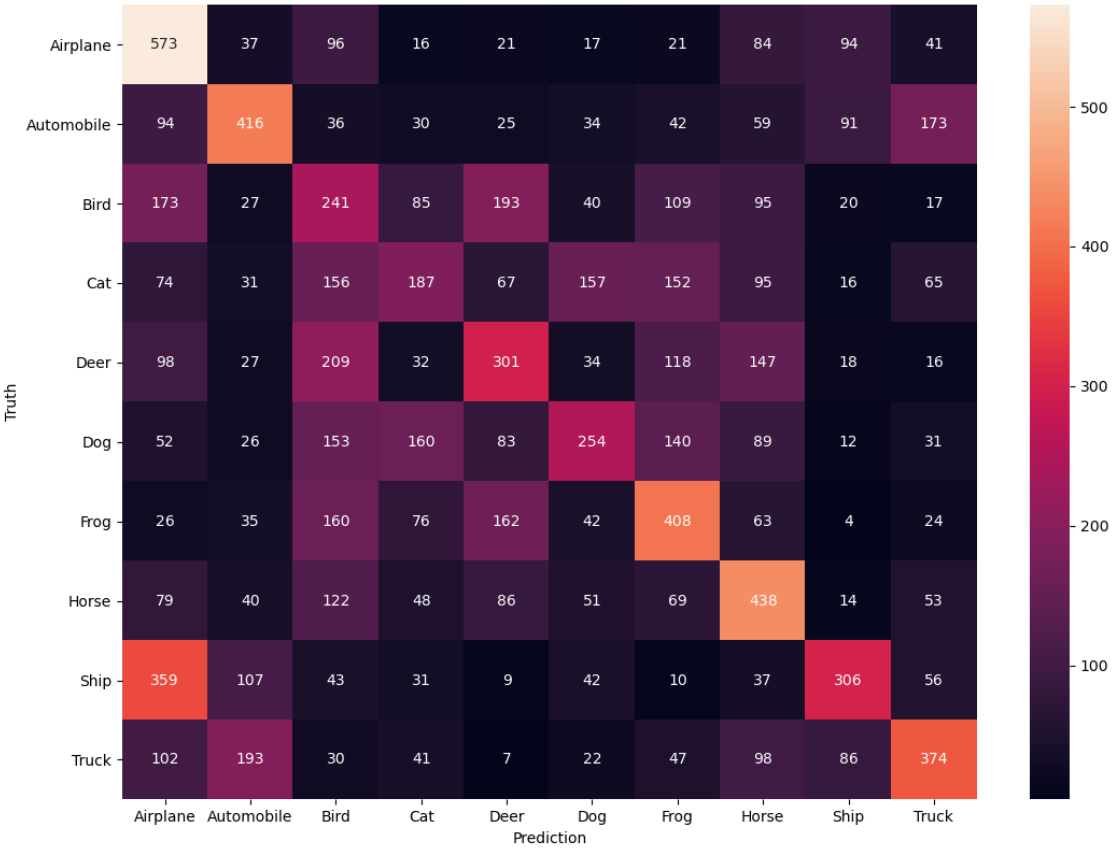
η = 0.01:

Önceki hıza göre benzer bir şekil çıkmıştır (Figür 71).



Figür 71: Testlerden birinde elde edilen hata grafiği.

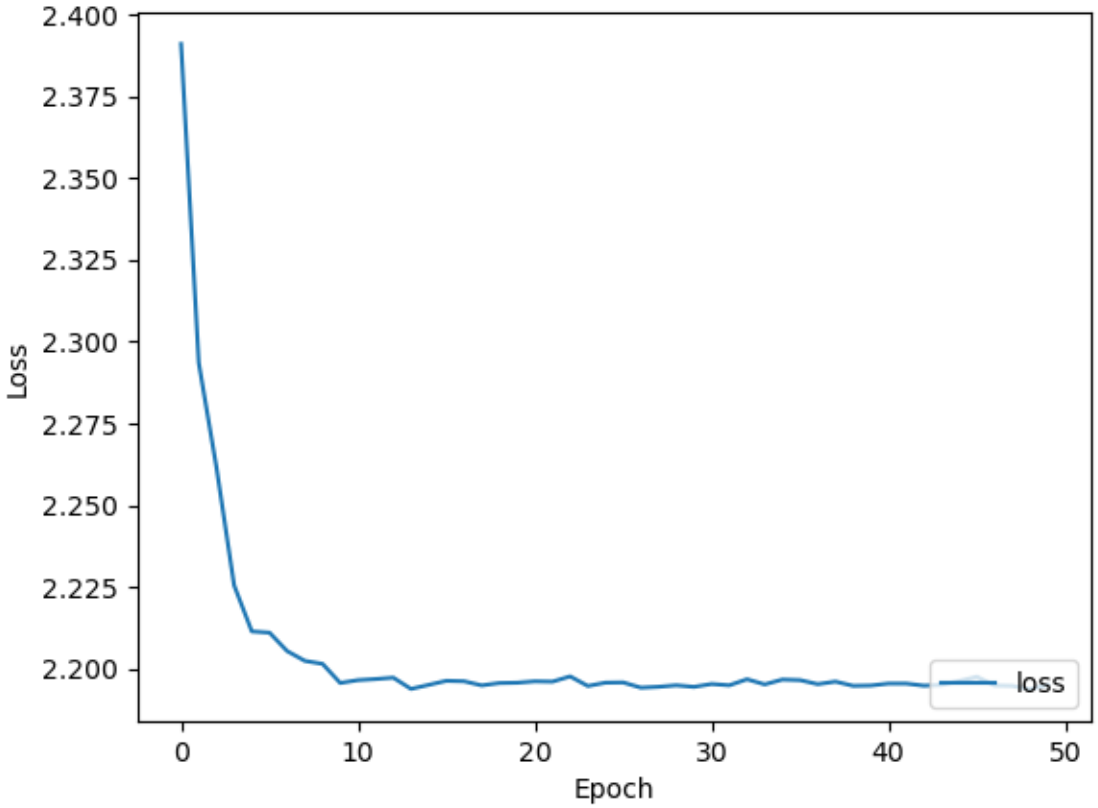
"Confusion matrix" matrisine bakıldığında. Bir önceki hızına göre yine benzer bir matrix ortaya çıktığı görülmüştür biraz daha doğruluk azalmıştır (Figür 72).



Figür 72: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

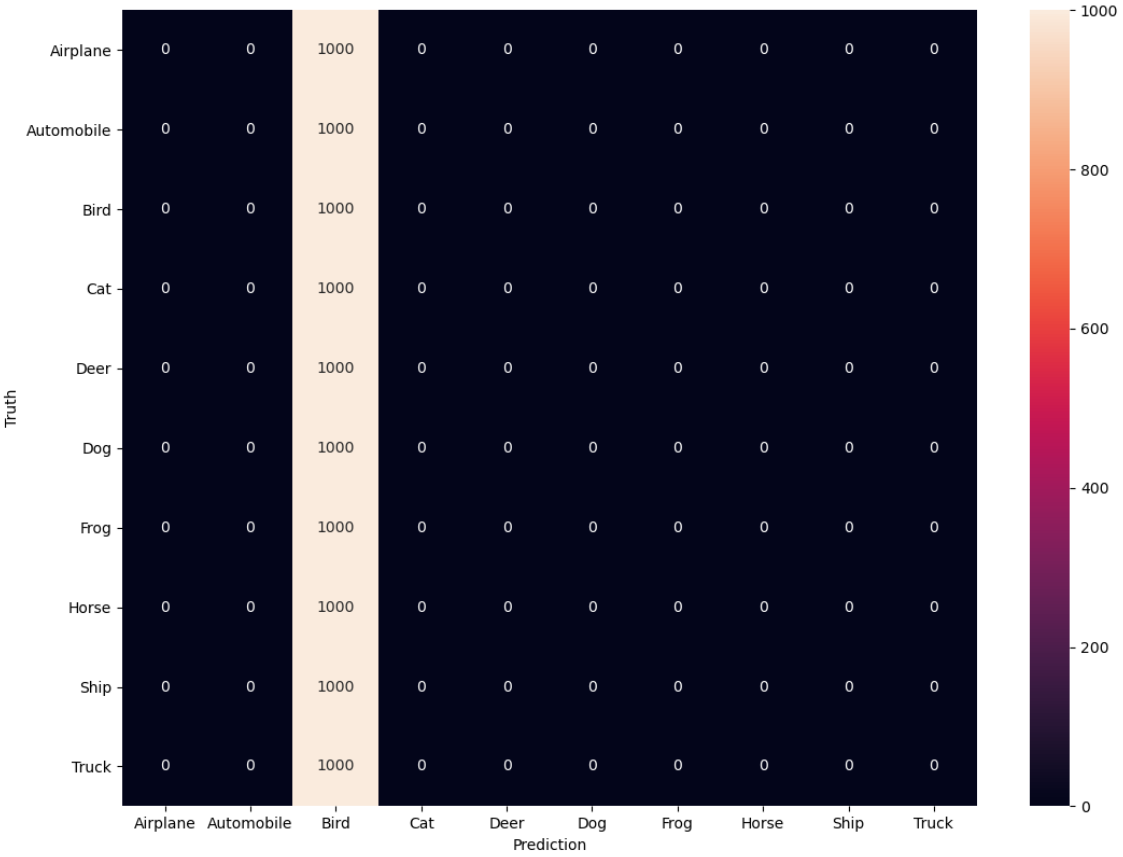
η = 0.1:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 73).



Figür 73: Testlerden birinde elde edilen hata grafiği.

"Confusion matrix" matrisine bakıldığında, Kuş resminin hepsini eşleştirdiği ve tam doğru olduğu yorumlanmıştır hızı artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 74).



Figür 74: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

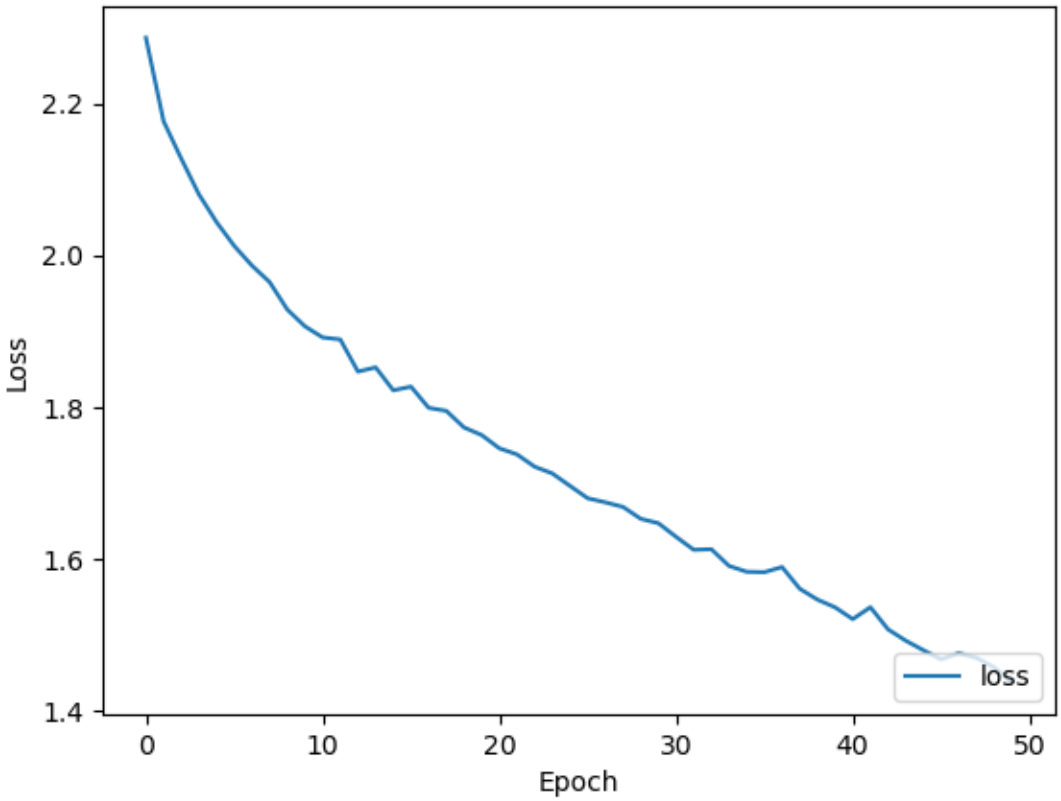
Momentum Oranı (m) Testleri:

Öğrenme hızı ve düşürme oranı sabit tutulurken öğrenme hızının farklı değerleri için testler gerçekleştirilir.

Momentum hızlarının değişimi doğruluğu çok etkilememiştir ama bir seviyeden sonra ağ yapısını bozduğu görülmüştür. Eğitim hatası ve tahmin hatası arasında bir ilişki olduğu görülmektedir.

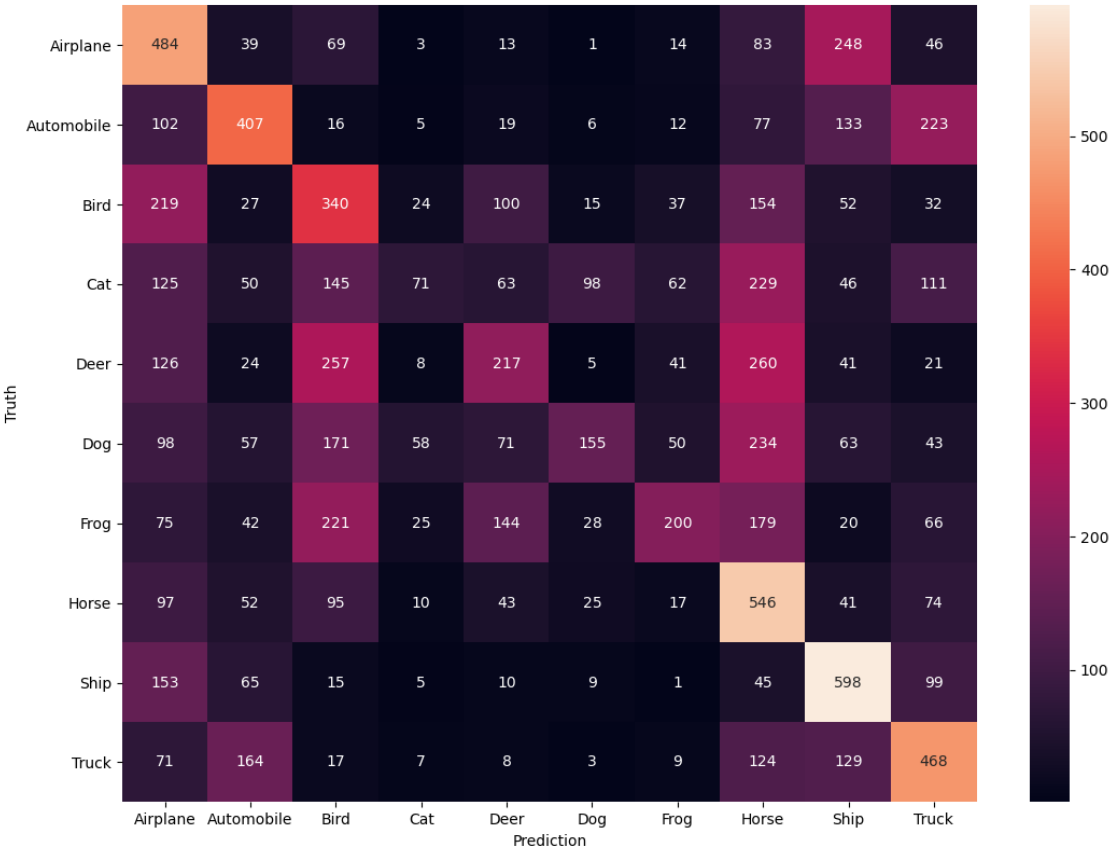
m = 0:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 75).



Figür 75: Testlerden birinde elde edilen hata grafiği.

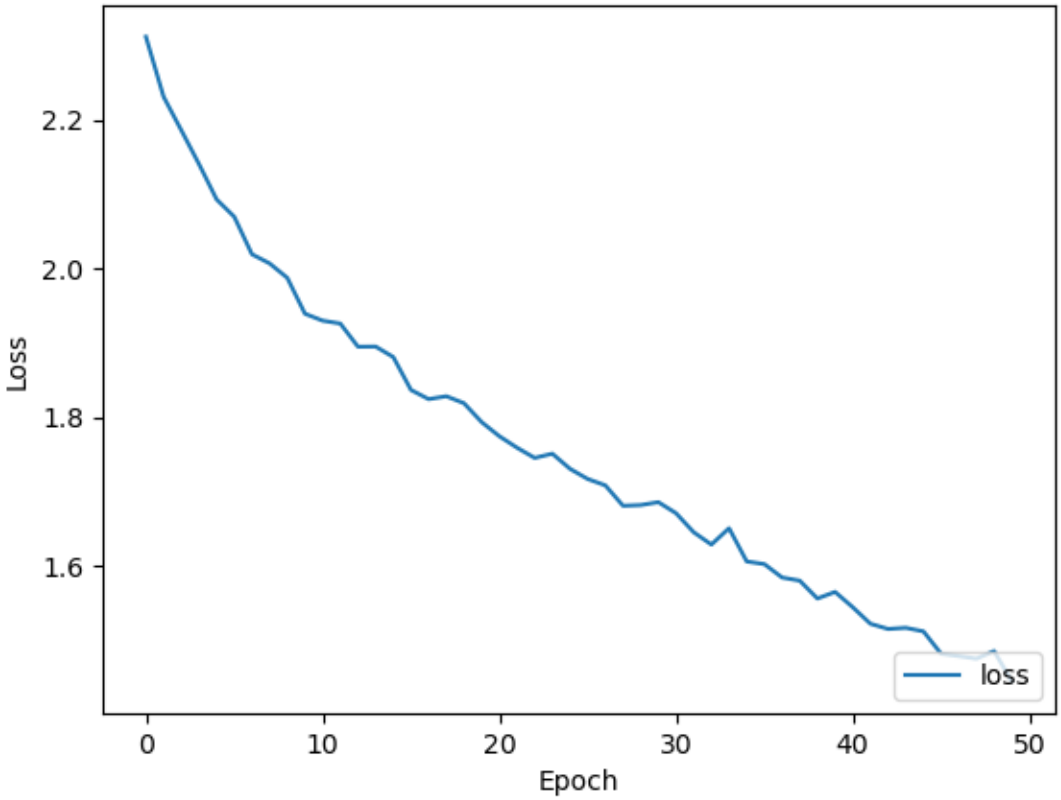
"Confusion matrix" matrisine bakıldığında, Kuş resminin hepsini eşleştirdiği ve tam doğru olduğu yorumlanmıştır hızı artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 76).



Figür 76: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

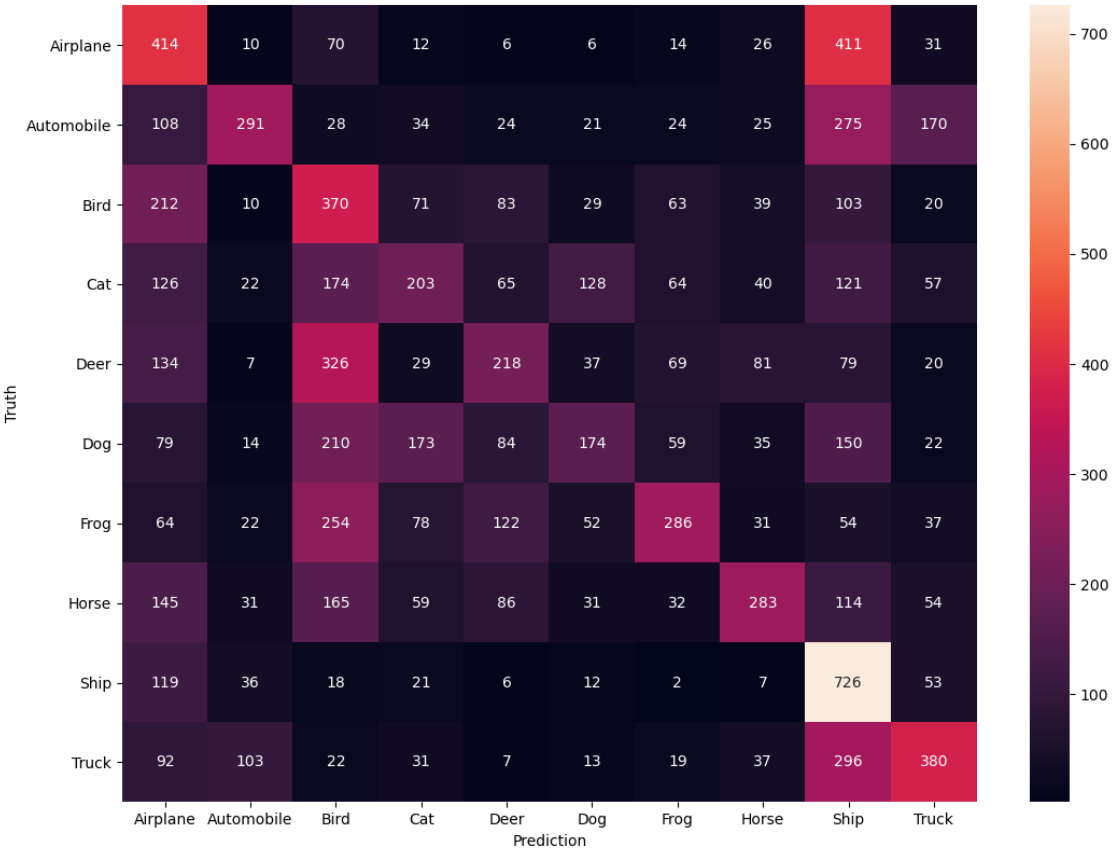
m = 0.2:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 77).



Figür 77: Testlerden birinde elde edilen hata grafiği.

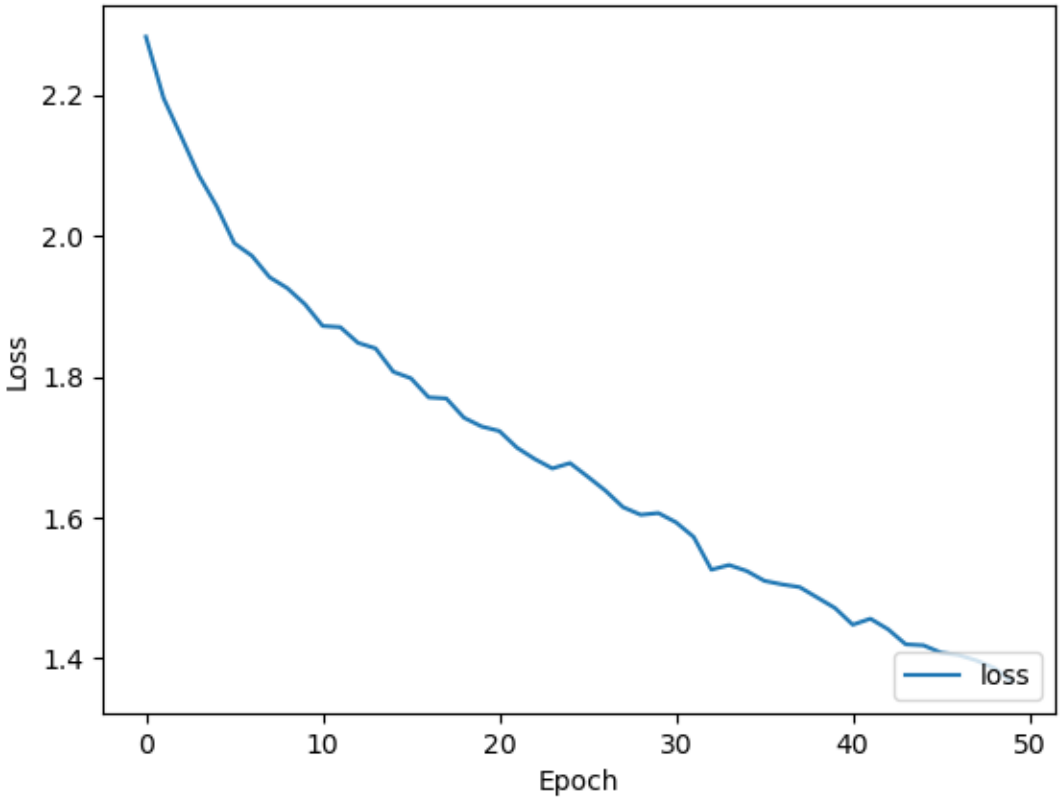
"Confusion matrix" matrisine bakıldığında, Kuş resminin hepsini eşleştirdiği ve tam doğru olduğu yorumlanmıştır hızı artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 78).



Figür 78: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

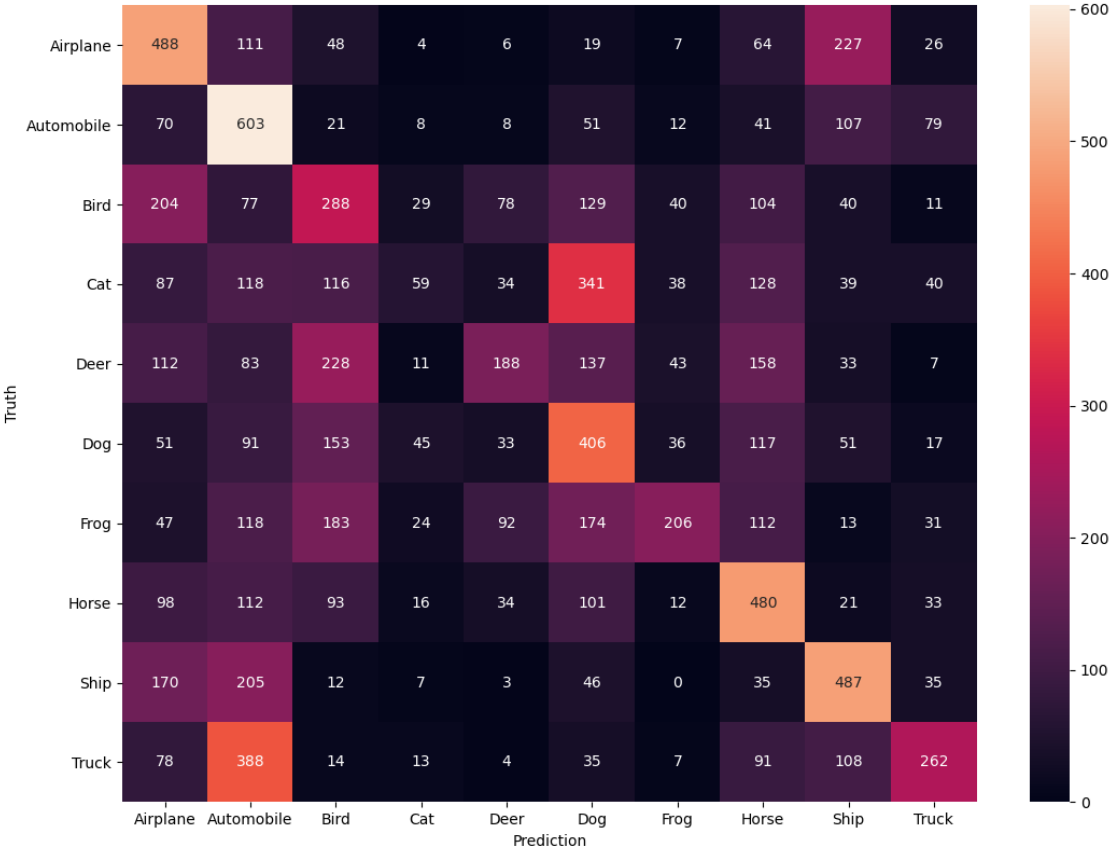
m = 0.4:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 79).



Figür 79: Testlerden birinde elde edilen hata grafiği.

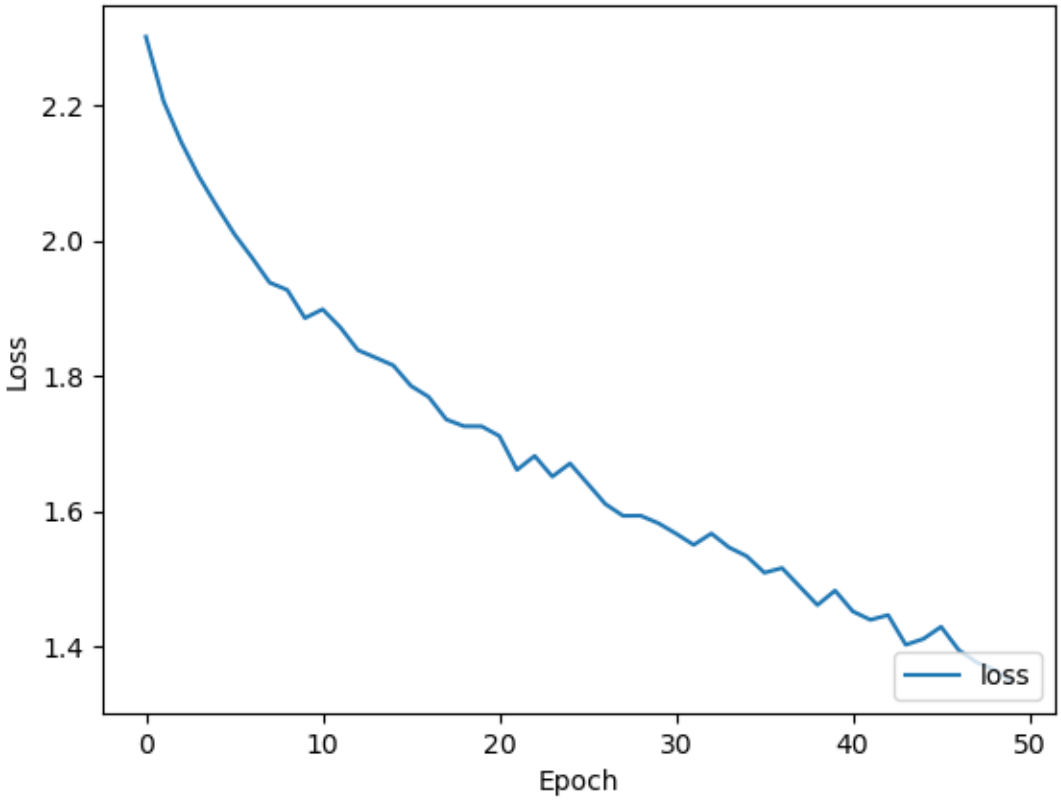
"Confusion matrix" matrisine bakıldığında, Kuş resminin hepsini eşleştirdiği ve tam doğru olduğu yorumlanmıştır hızı artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 38).



Figür 79: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

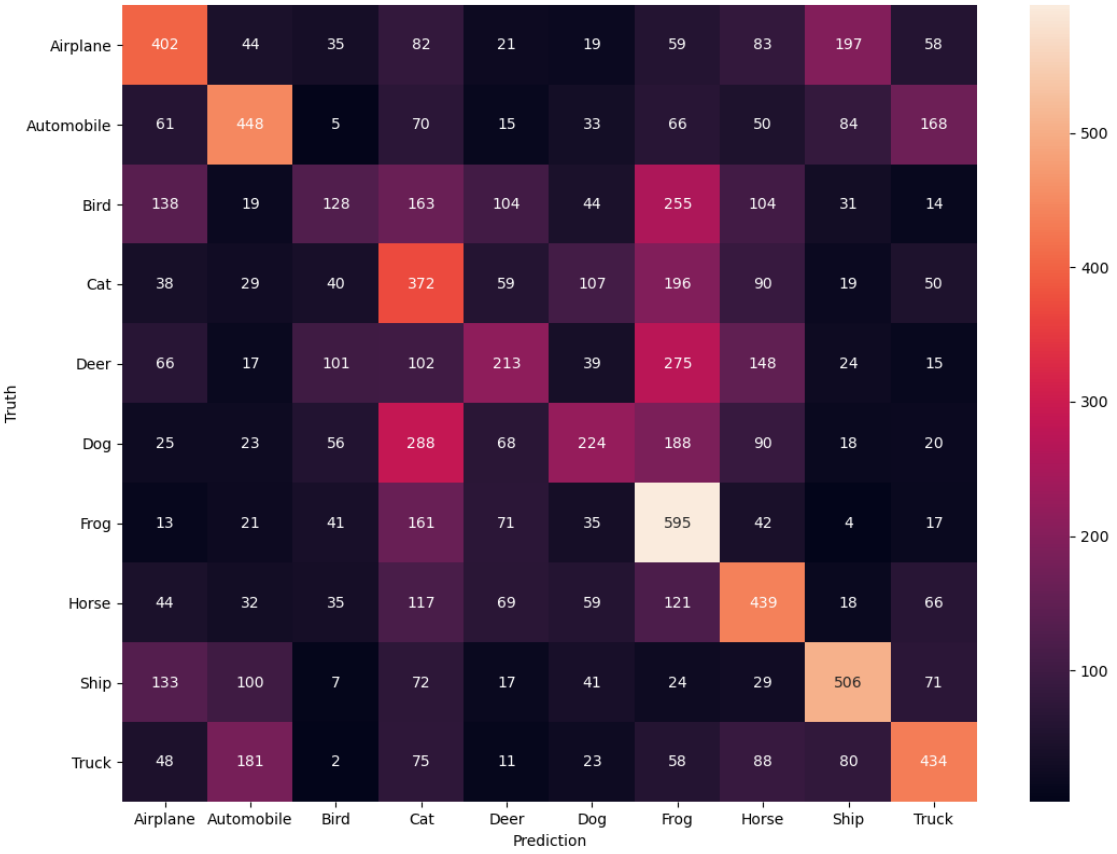
m = 0.6:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 80).



Figür 80: Testlerden birinde elde edilen hata grafiği.

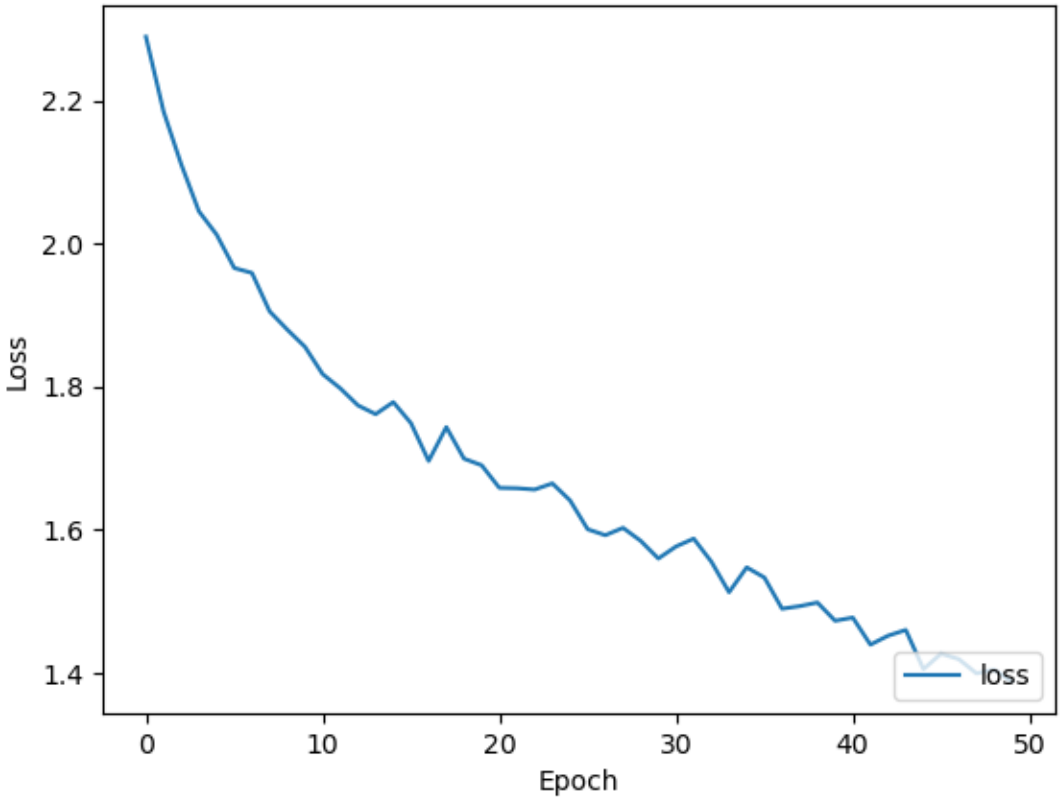
"Confusion matrix" matrisine bakıldığında, Kuş resminin hepsini eşleştirdiği ve tam doğru olduğu yorumlanmıştır hızı artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 81).



Figür 81: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

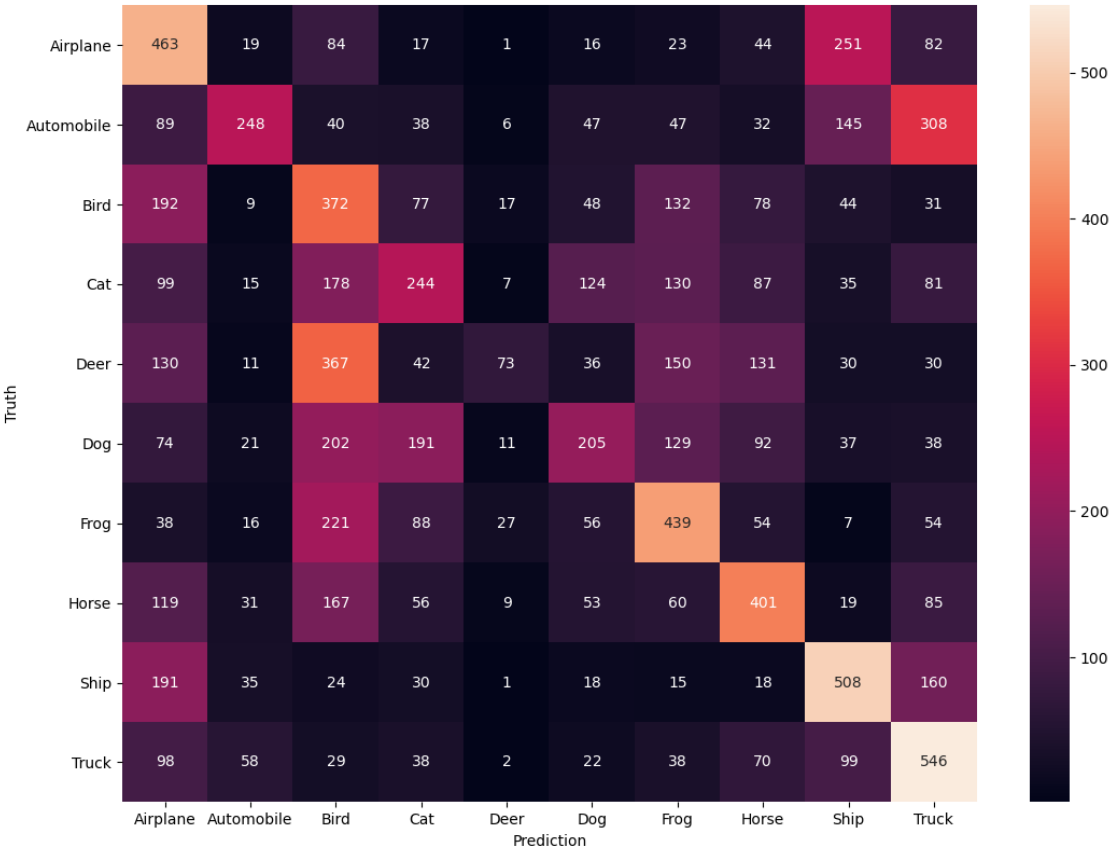
m = 0.8:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 82).



Figür 82: Testlerden birinde elde edilen hata grafiği.

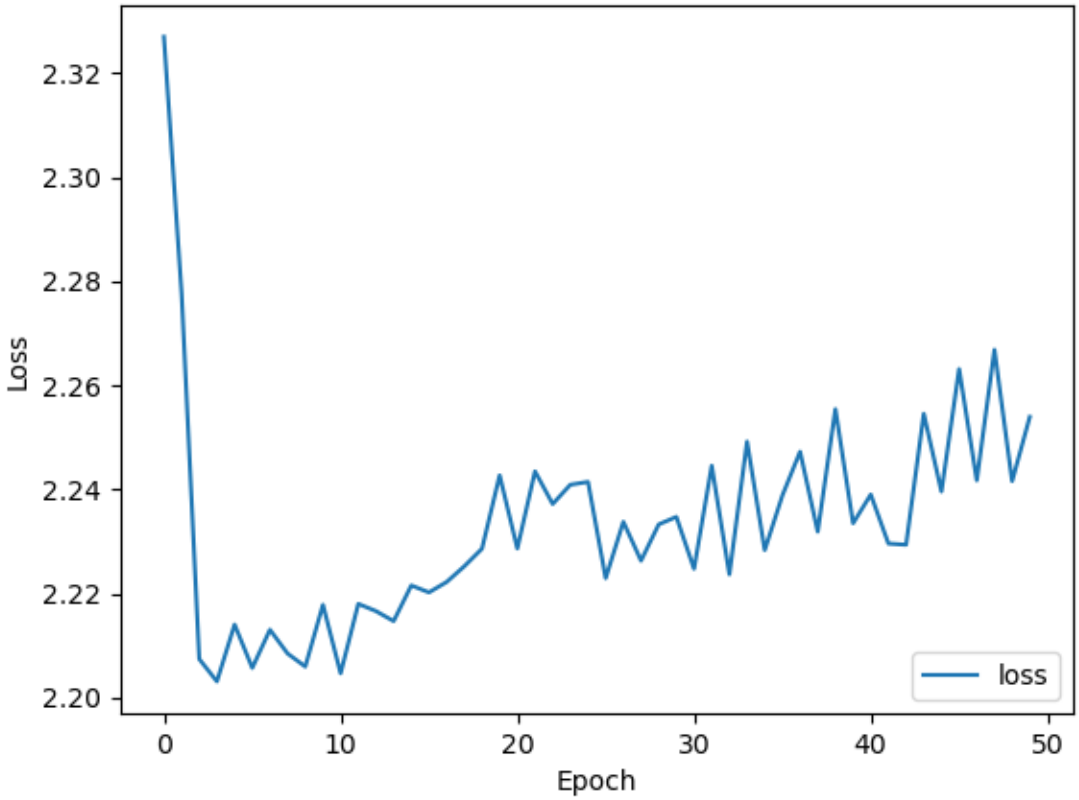
"Confusion matrix" matrisine bakıldığında, Kuş resminin hepsini eşleştirdiği ve tam doğru olduğu yorumlanmıştır hızı artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 83).



Figür 83: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

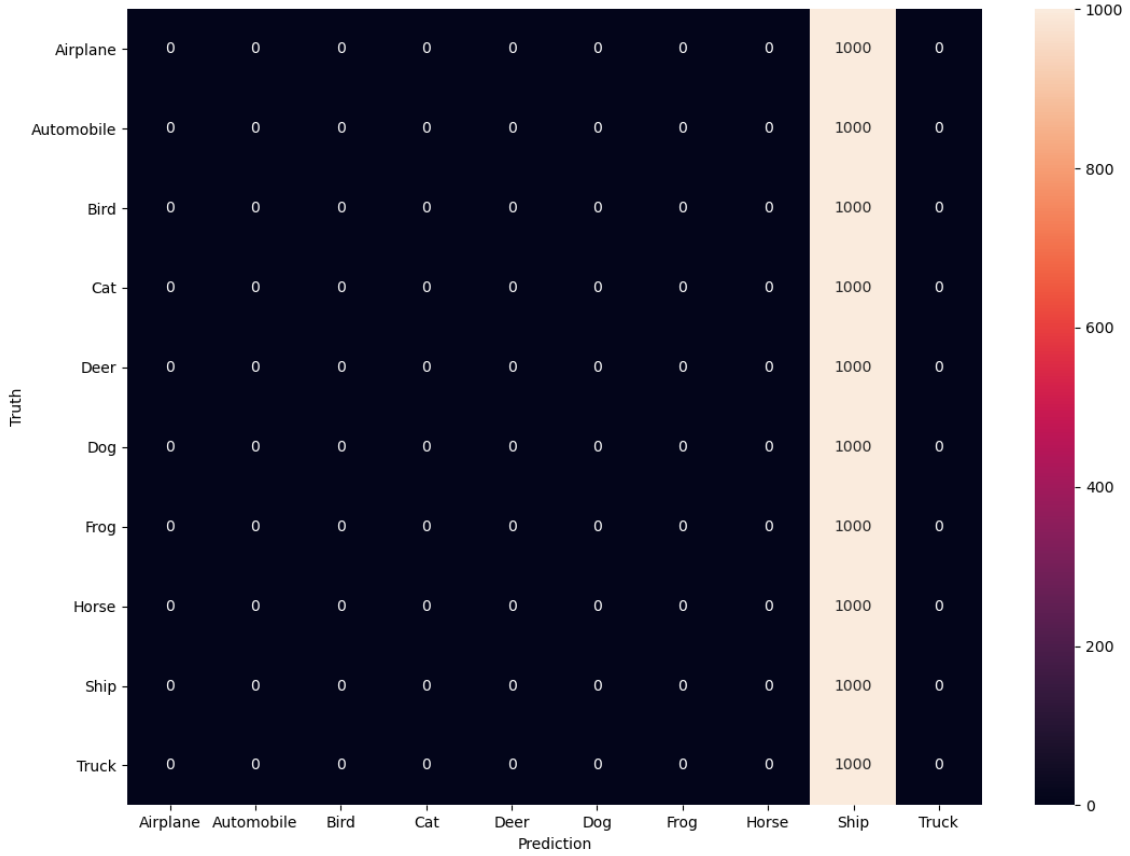
m = 1:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 84).



Figür 84: Testlerden birinde elde edilen hata grafiği.

"Confusion matrix" matrisine bakıldığında, Gemi resminin hepsini eşleştirdiği ve tam doğru olduğu çıktısı elde edilmiştir momentumu çok artırmak çıkışın hatalı olmasına sebep olmuştur. (Figür 85).



Figür 85: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

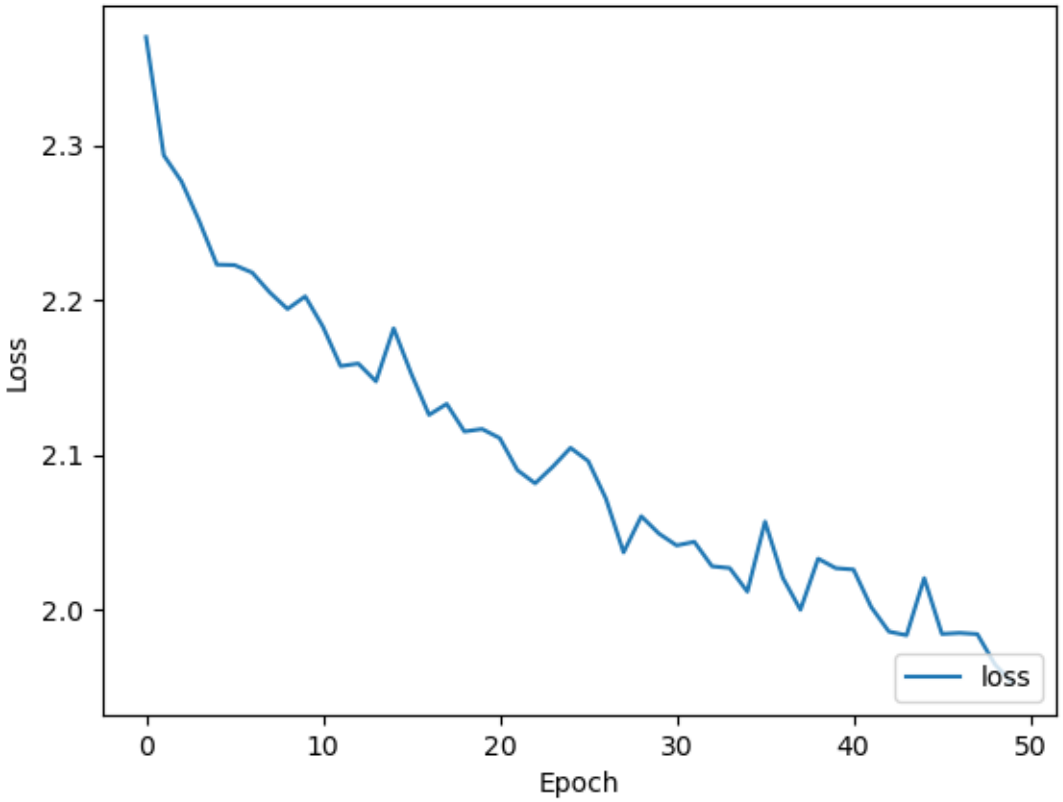
Düşürme Oranı (d) Testleri:

Momentum ve öğrenme hızı sabit tutulurken öğrenme hızının farklı değerleri için testler gerçekleştirilir.

Düşürme oranını artırdıkça daha başarılı sonuçların aldığı görülmüştür. Düşük oranlarda bir resmi diğer resimlerle eşleştirerek hatalı sonuçlar çıkarıldığı görülmüştür.

d = 0.5:

Düşürme oranının genel durumda kaybı azalttığı incelenmiştir. Başarılı bir öğrenme gerçeklememiştir (Fig 86).



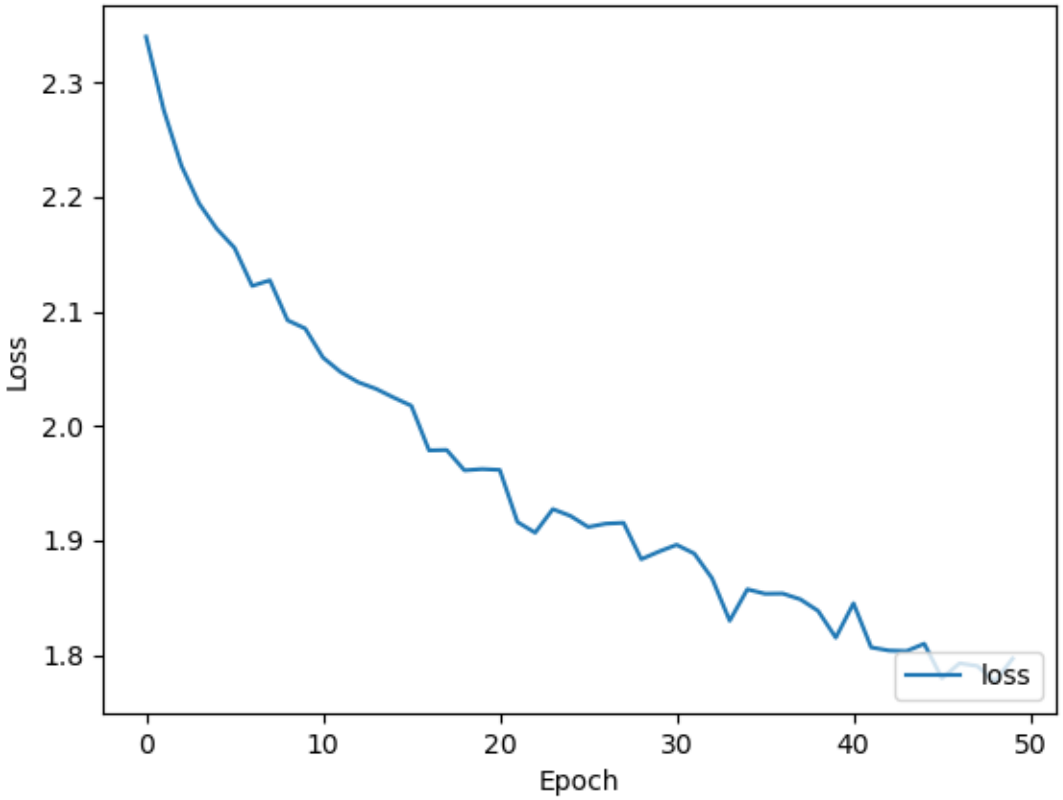
Figür 86: Testlerden birinde elde edilen hata grafiği.

"Confusion matrix" matrisine bakıldığında, Uçak resmini diğer bütün resimlerle eşleştirmiştir (Figür 87).

Figür 87: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

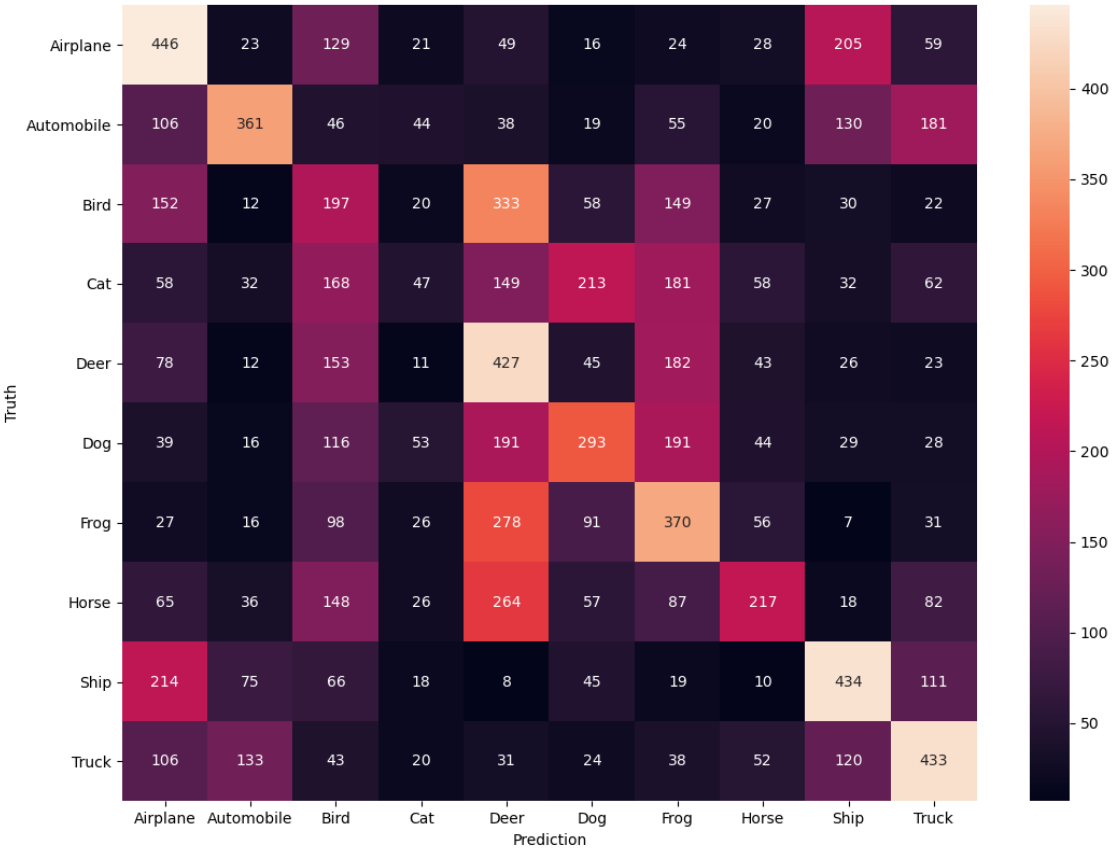
d = 0.6:

Düşme oranını artırdığımızda daha başarılı bir sonuç elde edilse de süreç başarılı olmamıştır. (Fig 88).



Figür 88: Testlerden birinde elde edilen hata grafiği.

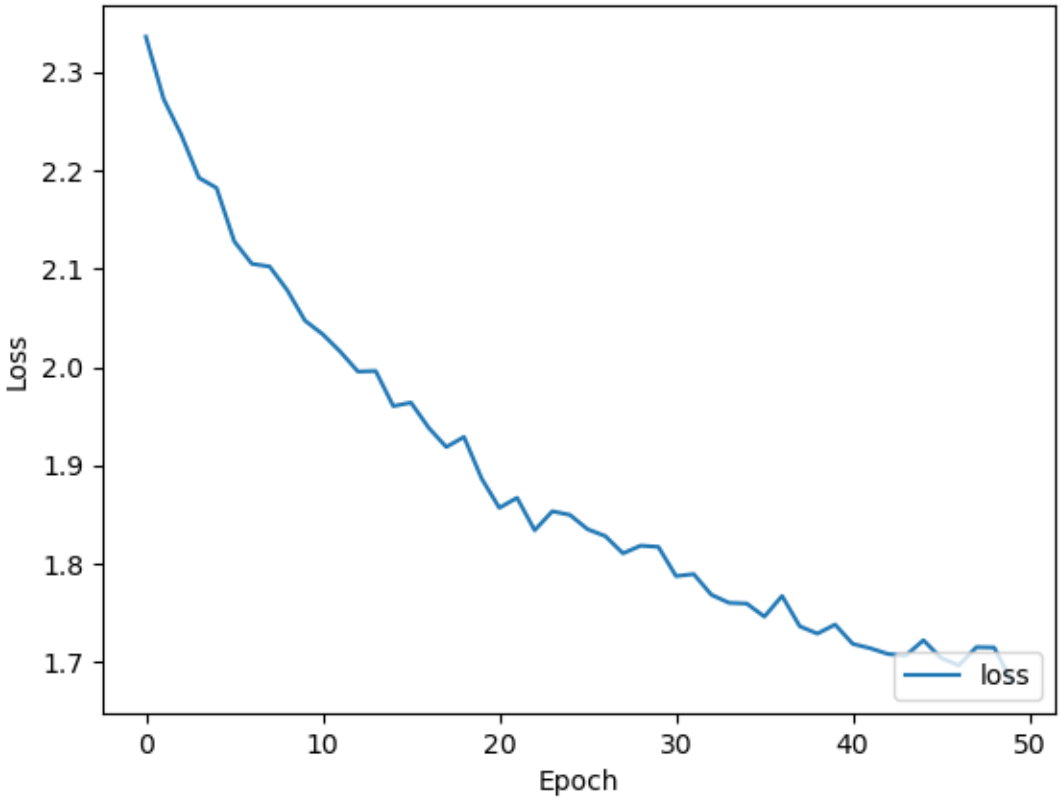
"Confusion matrix" matrisine bakıldığında, geyik, kuş ve kurbağa resimlerini diğerleriyle eşleştirildiği görülmüştür. (Figür 89).



Figür 89: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

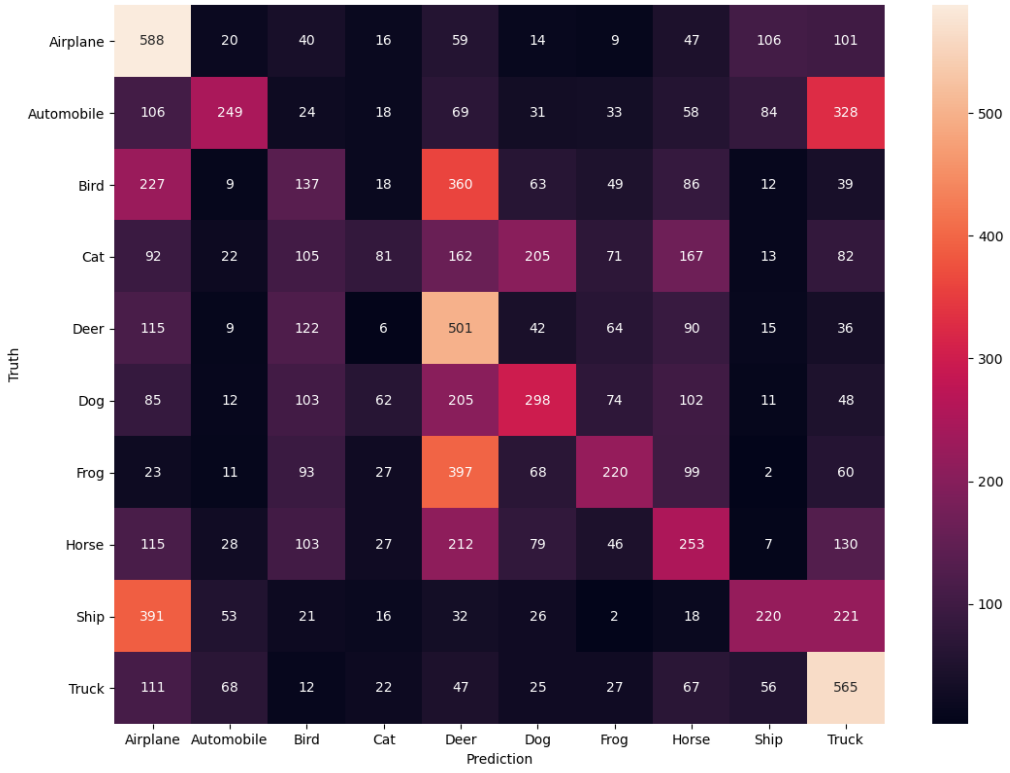
d = 0.7:

Önceki çıktılara göre daha anlamlı hale gelmiştir ama hala başarılı bir sonuç elde edilmemiştir. (Fig 90).



Figür 90: Testlerden birinde elde edilen hata grafiği.

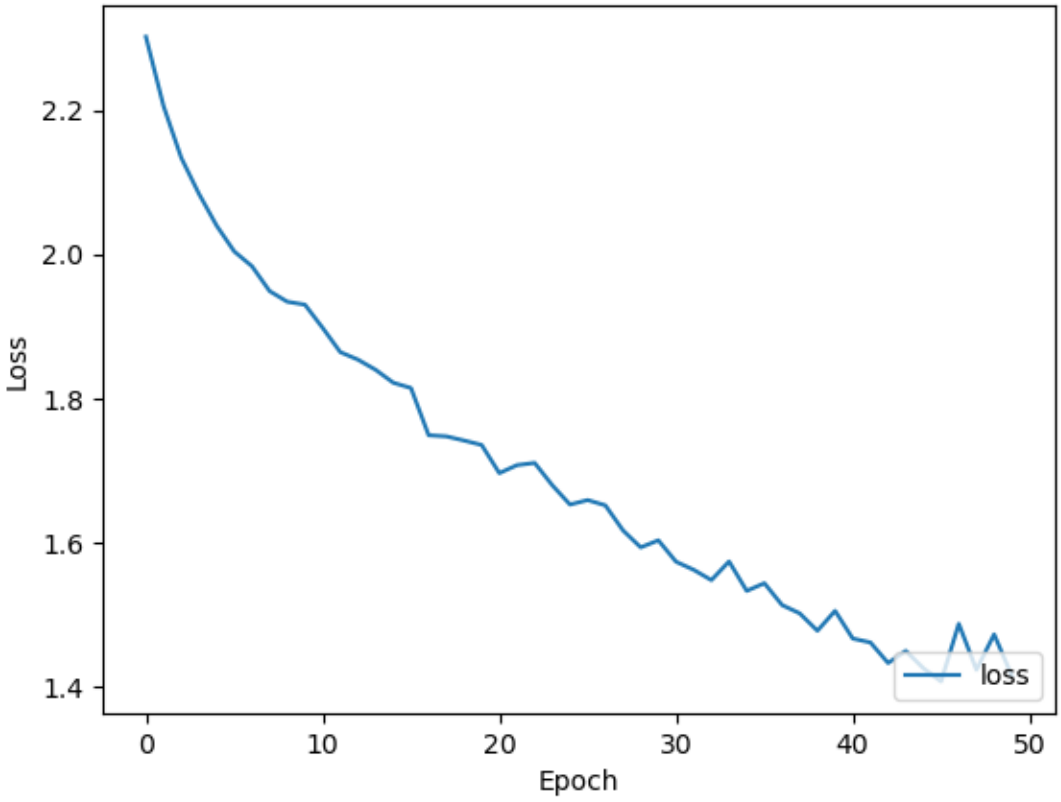
"Confusion matrix" matrisine bakıldığında, Kedi ve kuş haricinde diğer resimleri eşleştirmeyi doğru yapıyor. (Figür 91).



Figür 91: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

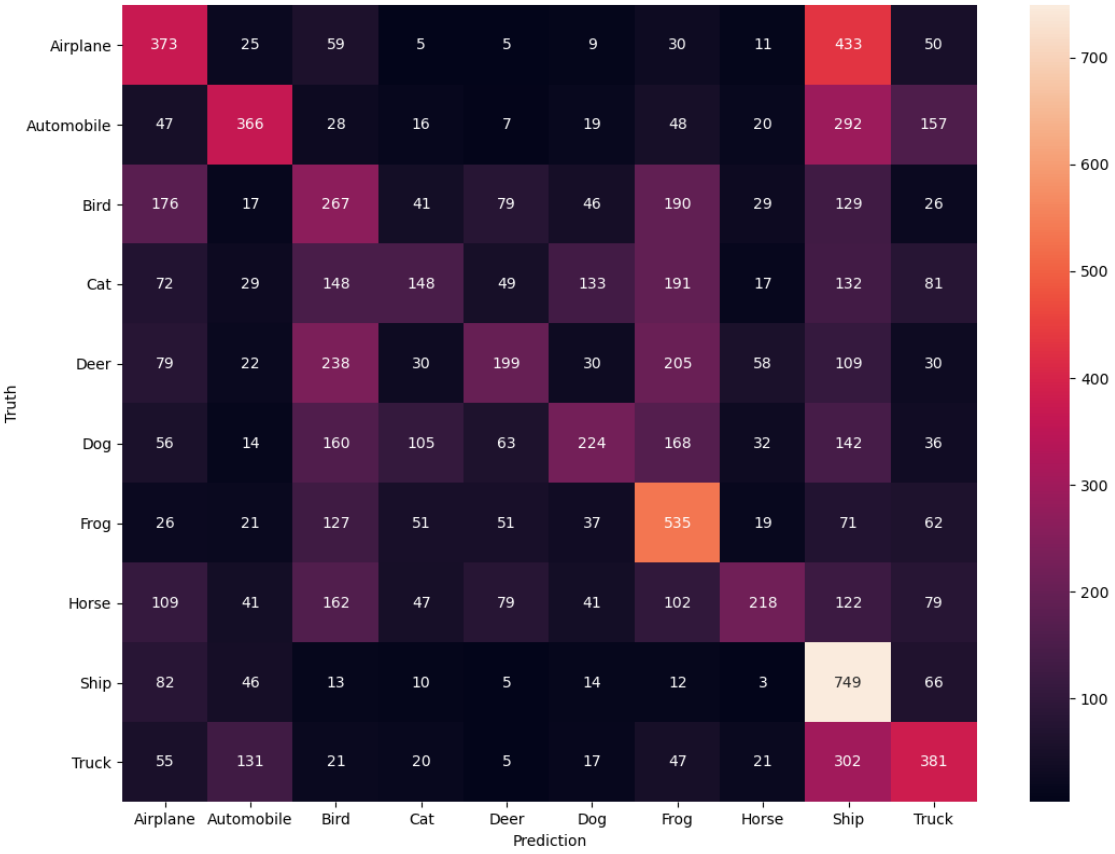
d = 0.8:

Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 92).



Figür 92: Testlerden birinde elde edilen hata grafiği.

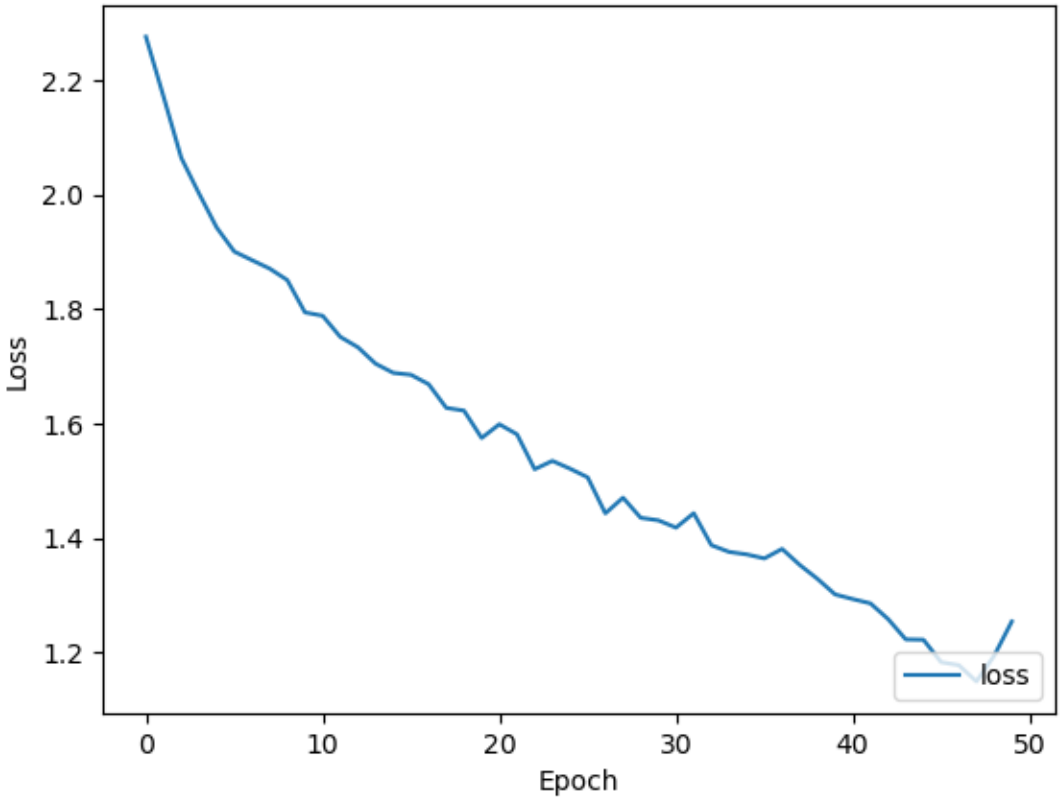
"Confusion matrix" matrisine bakıldığında, gemi resimlerinin diğerlerine göre daha çok doğru eşleştirdiği görülüyor (Figür 93).



Figür 93: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

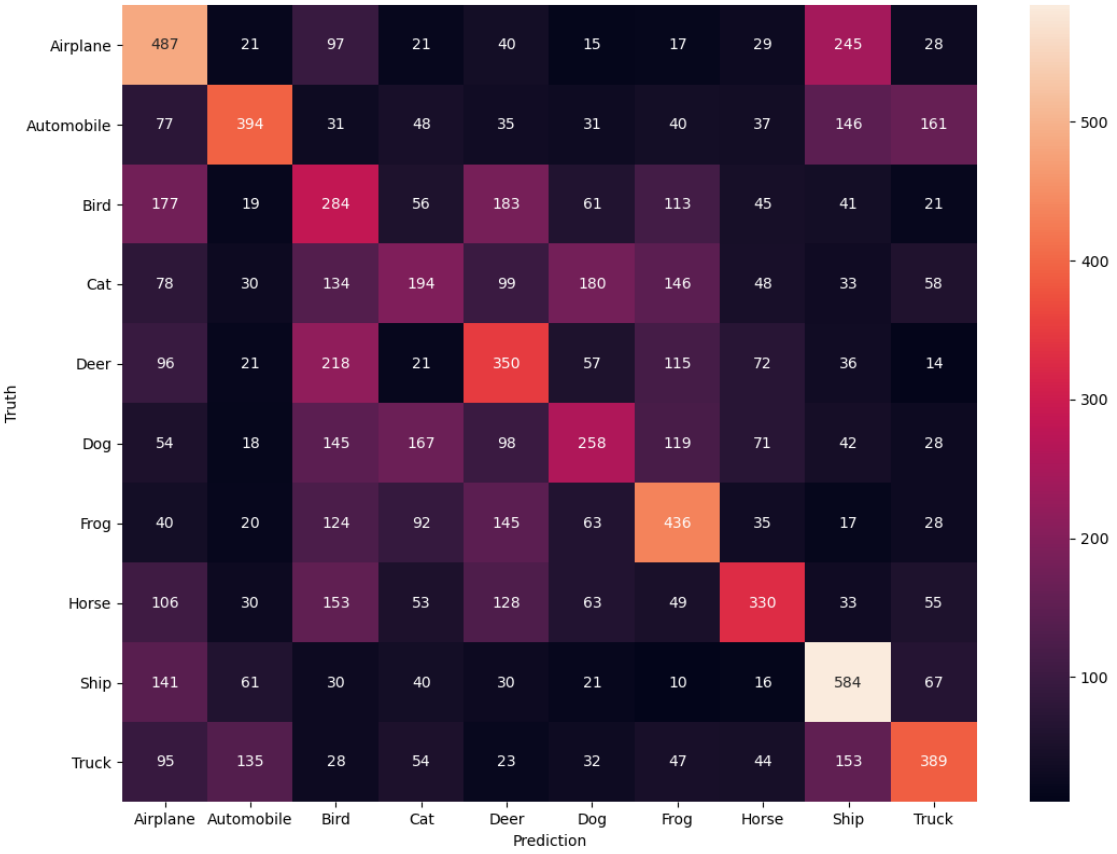
d = 0.9:

Düşme hızı bu seviyede başarılı sonuçlar vermeye başlamıştır ağ kullanılabilecek hale gelmeye başlamıştır. (Fig 94).



Figür 94: Testlerden birinde elde edilen hata grafiği.

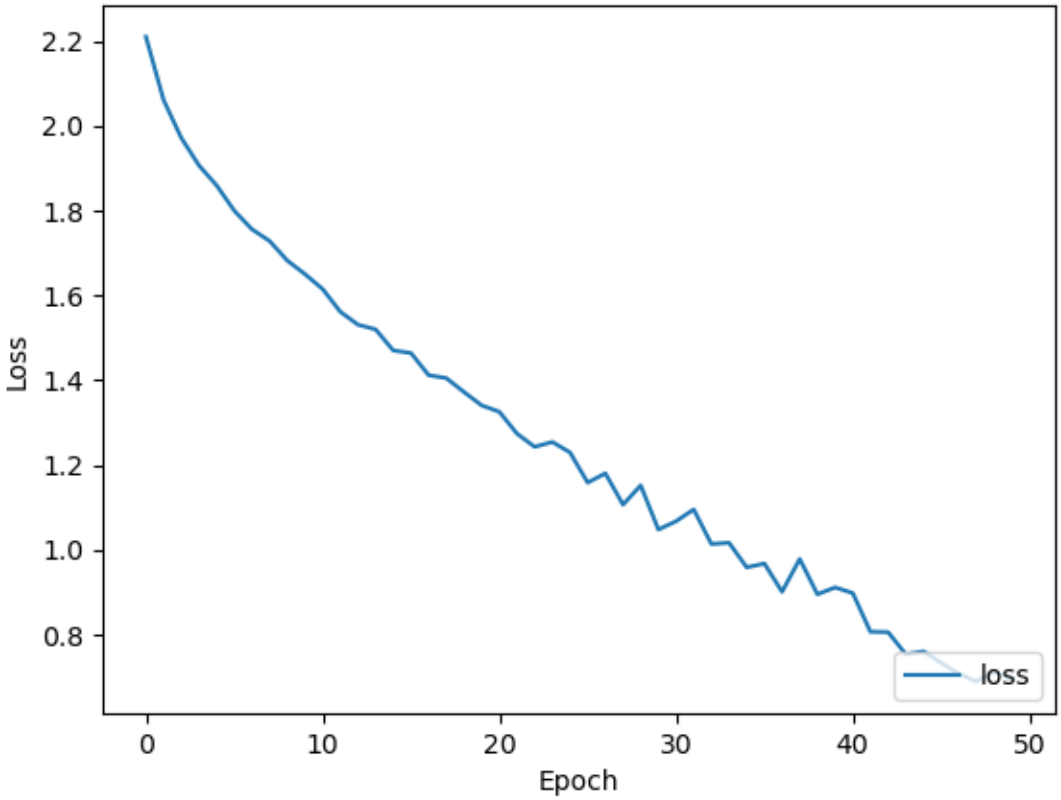
"Confusion matrix" matrisine bakıldığında, yine gemi resimlerinin diğerlerine göre daha çok doğru eşleştirdiği görülüyor (Figür 95).



Figür 95: Test sonrası elde edilen ortalama "confusion matrix" matrisi.

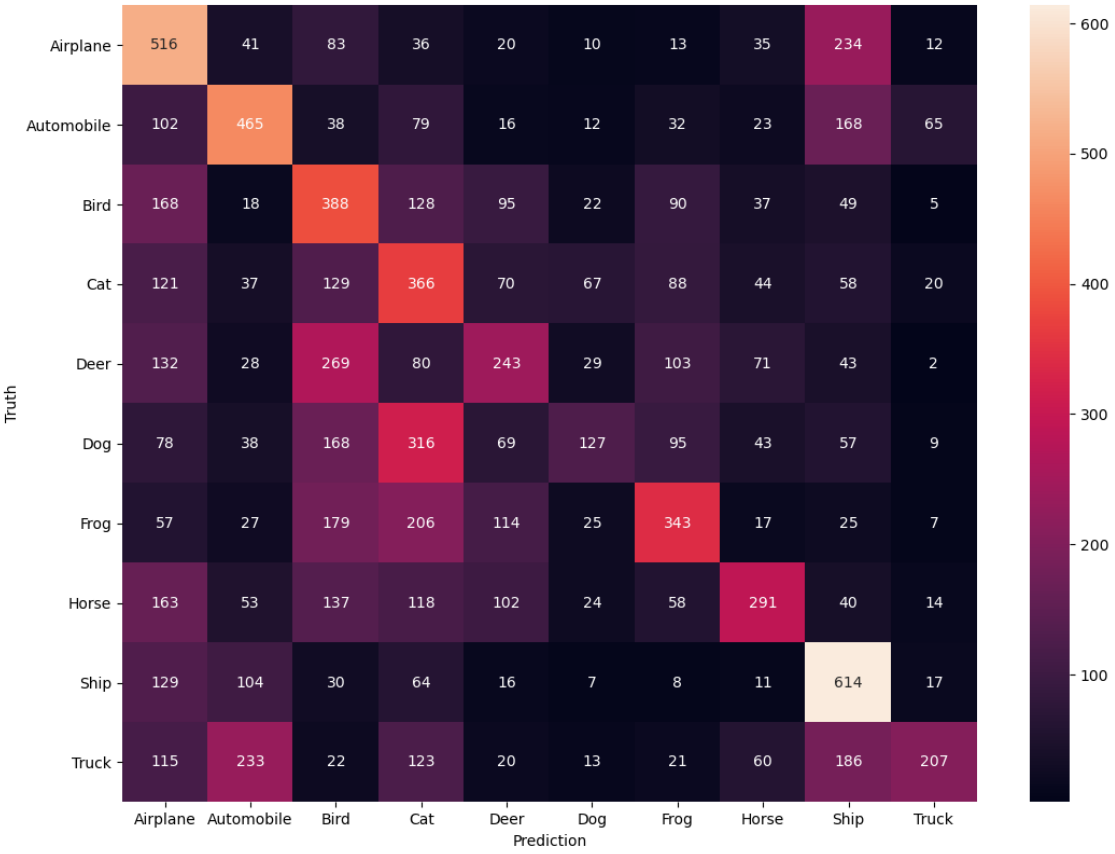
d = 1:

En başarılı sonuç bu oranda tespit edildi Kaybın direk bir düşüş yaptığı incelenmiştir ama doğruluk yoktur. (Fig 96).



Figür 96: Testlerden birinde elde edilen hata grafiği.

"Confusion matrix" matrisine bakıldığında, daha dengeli dağılım ve doğruluk görülmüştür (Figür 97).



Figür 97: Test sonrası elde edilen ortalama "confusion matrix" matrisi.