به نام خدا

گزارش کار پروژه هوش مصنوعی

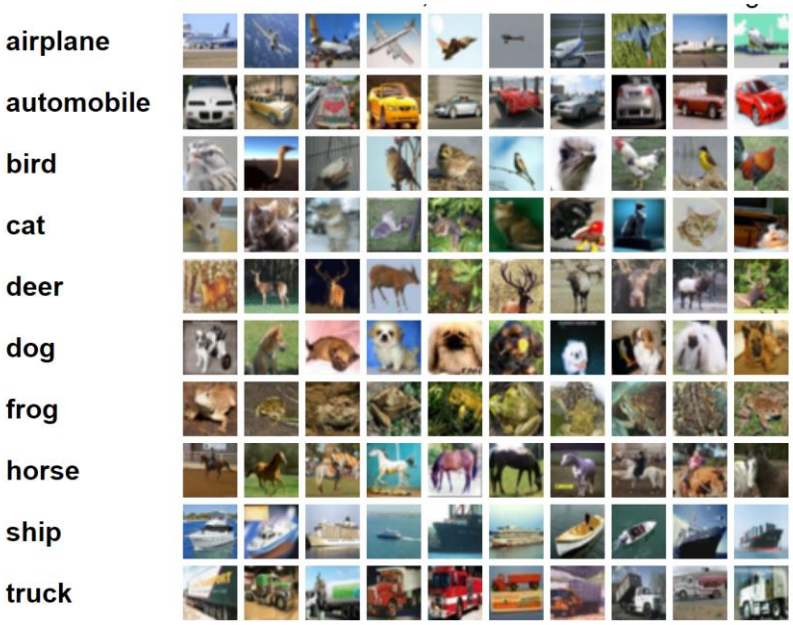
اعضای گروه:

کیارش قاسم زاده، علیرضا مداح و ماناحاتم زاده

-مقدمه :

Cifar-10 :

مجموعه داده CIFAR-10 (Canadian Institute for Advanced Research, 10 classes) یک زیرمجموعه از مجموعه داده Tiny Images است و شامل 60000 تصویر رنگی 32*32 می‌باشد. تصاویر با یکی از 10 دسته‌بندی متقابل برچسب‌گذاری شده‌اند: هواپیما، اتومبیل، پرنده، گربه، گوزن، سگ، قورباغه، اسب، کشتی و کامیون. در هر دسته 6000 تصویر وجود دارد که شامل 5000 تصویر برای آموزش و 1000 تصویر برای آزمون می‌باشد.



حال در ان پروژه به دنبال پیاده سازی و مقایسه عملکرد مدل های مختلف یادگیری ماشین (machine learning) و شبکه های عصبی (neural networks) هستیم تا به کمک آنها بتوانیم این دیتاست را به روش حل مسئله کلاس بندی (classification) تشخیص دهیم.

-پیش پردازش:

ابتدا پیش از انجام پیش پردازش، کتابخانه های مورد نیاز برای انجام این پروژه را اضافه می کنیم :

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score
import seaborn as sns
```

در ادامه به بخش پیش پردازش اشاره می کنیم که از آنجا که در این پروژه از پایتورچ کمک گرفته ایم، نیاز است تا فرمت عکس های دیتاست را به تنسور تبدیل کنیم. پس آن دیتاست را دانلود و آپلود می کنیم و نرمال سازی را بر روی آن انجام می دهیم (میانگین و انحراف معیار را بر روی 0.5 تنظیم می کنیم.). سپس تقسیم بندی دیتاست به trainset و testset آنها را به صورت رندوم به دسته های 100 تایی توسط testloader و trainloader تقسیم می کنیم. کلاس بندی مربوط به دیتاست را نیز در ده دسته مشخص از قبل تعریف شده انجام می دهیم.

- شبکه عصبی (neural network) :

در ادامه برای پردازش دیتاست از دو مدل شبکه عصبی (MLP , Encoder/decoder) استفاده می کنیم :

- MLP :

در این مدل که شبکه عصبی دارای 5 لایه است به صورت کاهشی عمل می کند که به شرح زیر است : (از تابع فعال ساز Relu استفاده شده است)

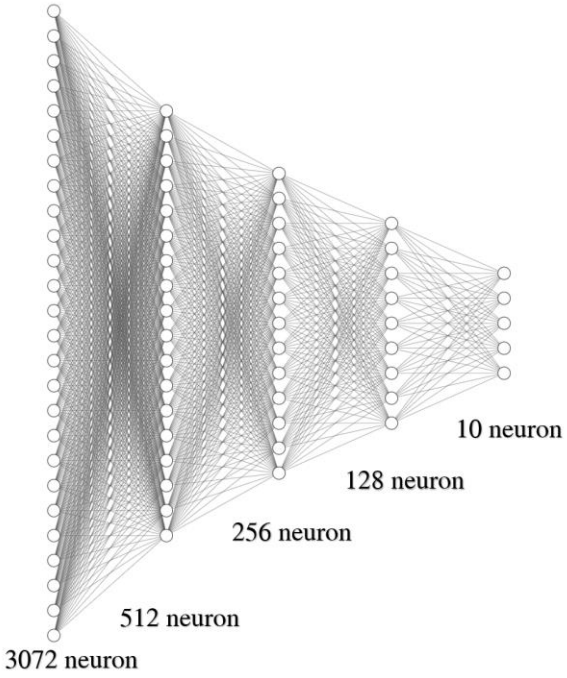
۱- ورودی شبکه با 3072 نورون

۲- کاهش نورون ها به 512

۳- کاهش نورون ها به 256

۴- کاهش نورون ها به 128

۵- کاهش نهایی نورون ها به 10




```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(3 * 32 * 32, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 3 * 32 * 32)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

توجه داریم که در هر مرحله کاهش نورون ها، تابع فعال ساز نیز به شبکه اعمال می شود.

- Encoder/decoder :

در این مدل تا حدودی مانند مدل قبل عمل می کنیم، با این تفاوت که پس از یک بار کاهش نورون ها در لایه دوم، لایه سوم دچار افزایش تعداد نورون ها می شود. علت این امر این است که با این روش می توان دقت و تشخیص ویژگی ها (features) را نسبت به حالت تماماً کاهشی افزایش داد. حال ساختار این مدل را می توان به صورت زیر تشریح کرد : (تابع فعال ساز Relu استفاده شده است)

۱- ورودی شبکه با تعداد 3072 نورون

۲- کاهش تعداد نورون ها به 512

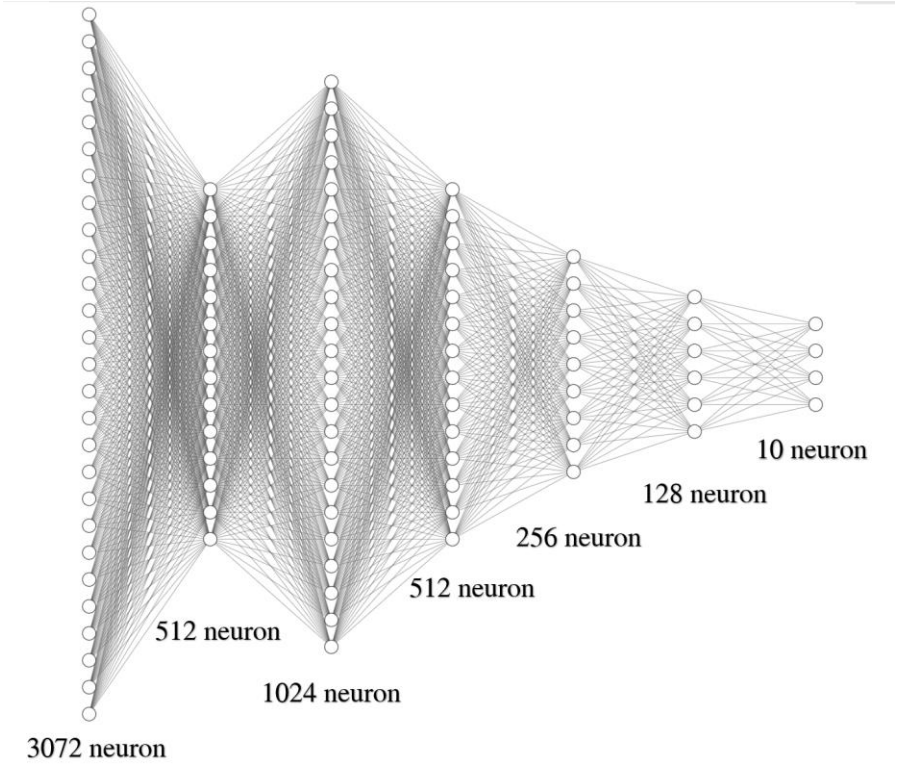
۳- افزایش تعداد نورون ها به 1024

۴- کاهش تعداد نورون ها به 512

۵- کاهش تعداد نورون ها به 256

۶- کاهش تعداد نورون ها به 128

۷- کاهش تعداد نورون ها به 10



```python
class EncoderDecoder(nn.Module):
    def __init__(self):
        super(EncoderDecoder, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(3 * 32 * 32, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU()
        )
        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )
```

```python
  def forward(self, x):
      x = x.view(-1, 3 * 32 * 32)
      x = self.encoder(x)
      x = self.decoder(x)
      return x
```

توجه داریم در این مدل نیز پس از هر بار کاهش یا افزایش، تابع فعال سازی Relu به شبکه اعمال شده است.

- شبکه های کانولوشنی (CNN) :

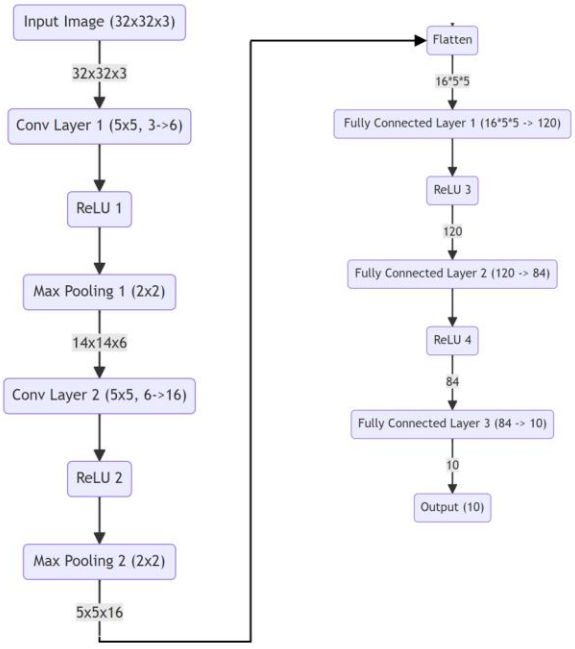
در ادامه به تعریف دو مدل دیگر برای پردازش دیتاست می پردازیم :

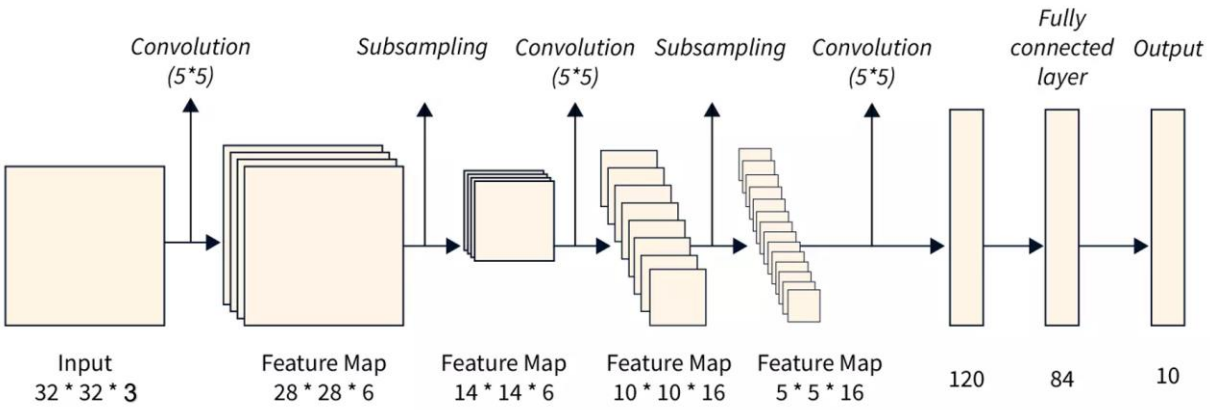
۱- Lenet

۲- Resnet

-Lenet :

به طور کلی، LeNet به LeNet-5 اشاره دارد و یک شبکه عصبی کانولوشنی ساده است. شبکه‌های عصبی کانولوشنی نوعی از شبکه‌های عصبی پیش‌خور هستند که نورون‌های مصنوعی آن‌ها می‌توانند به بخشی از سلول‌های اطراف در محدوده پوشش واکنش نشان دهند و در پردازش تصویر در مقیاس بزرگ عملکرد خوبی دارند.





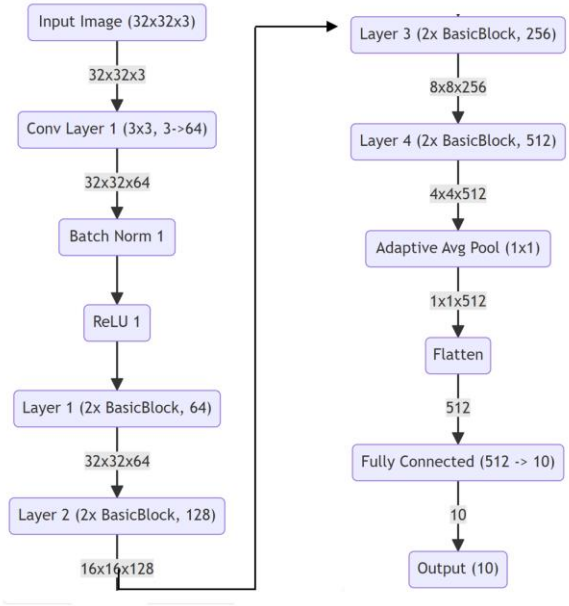
با استفاده از ساختار های بالا، کد مربوط به آن را پیاده سازی می کنیم :

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(-1, 16 * 5 * 5)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

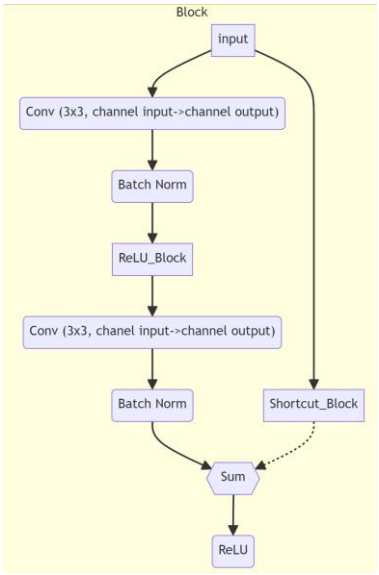
در هر مرحله پیاده سازی همانند ساختار نشان داده شده در بالا تابع فعال ساز Relu به شبکه اعمال می شود.

Resnet :

شبکه عصبی Resnet یک مدل مهم یادگیری عمیق است که در آن لایه‌های وزنی توابع باقیمانده را نسبت به ورودی‌های لایه یاد می‌گیرند. تفاوت Resnet از سایر شبکه های عصبی این است که یک لایه فیدبک در ورودی دارد. به این صورت که این امر موجه عدم محو گرادیان می شود و نسبت به شبکه Lenet دارای دقت و عملکرد بهتری می باشد. به صورت کلی می توان ساختار اصلی این شبکه عصبی را به شرح زیر توصیف کرد :



که در درون هر بلوک (basic-block) از آن ساختار زیر حاکم است :



حال با استفاده از این ساختار ها، ابتدا بلوک اولیه Resnet و در نهایت اصل مدل را تعریف می کنیم :

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
```

```
    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=1)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=1)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=1)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.avg_pool2 = nn.AvgPool2d((2,2))
        self.fc = nn.Linear(512, num_classes)
```

```
    def _make_layer(self, block, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.avg_pool2(x)
        x = self.layer3(x)
        x = self.avg_pool2(x)
        x = self.layer4(x)
        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

def ResNet18():
    return ResNet(BasicBlock, [2, 2, 2, 2])
```

- بخش train :

در ارتباط با این بخش ابتدا ی توان گفت که هر مدل را 25 بار train شده است که به صورت زیر کد آن را پیاده سازی کردیم :

```
from tqdm import tqdm
def train_model(model, trainloader, criterion, optimizer, epochs=25):
    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(tqdm(trainloader, 0)):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = model(inputs.to("cuda"))
            loss = criterion(outputs, labels.to("cuda"))
            loss.backward()
            optimizer.step()

    print('Finished Training')
```

قابل توجه است که در اینجا از طریق یک حلقه for از روی trainloader، دیتا ها را به همراه شاخص (index) آن می خوانیم و بر اساس آن خروجی دریافت و تابع هزینه (loss function) نیز محاسبه می شود. همچنین با دستور loss.backward() تمام گرادیان ها و هزینه های مورد نیاز حساب می شود که با انجام دستور بعدی خود (optimizer.step()) آنها به روز می شوند.

قابل ذکر است که در این پروژه از تابع هزینه cross entropy (loss function) و همچنین در بخش بهینه ساز (optimizer) از بهینه ساز آدام (adam) و بهینه ساز SGD استفاده شده است.

- cross entropy :

به طور کلی تابع هزینه cross entropy به صورت زیر است :

$$L(\hat{y}, y) = -\sum_{k}^{K} y^{(k)} \log \hat{y}^{(k)}$$

- بهینه ساز آدام (adam) :

به طور کلی بهینه ساز adam یکی از رایج ترین الگوریتم بهینه سازی در یادگیری عمیق (deep learning) است. علت استفاده از آن به این دلیل می باشد که نسبت به بقیه الگوریتم های بهینه سازی دارای سرعت و دقت بیشتری است. همچنین در حل مسائل نقاط زینی و نقاط اکسترمم های محلی (Local extremes) دارای عملکرد بهتری می باشد. مقدار الفا در این بهینه ساز با سرچ به عدد معمول ۰.۰۰۱ رسیدیم

- بهینه ساز SGD :

این بهینه ساز مرسوم است که در آن الفا را با تحقیق عدد ۰.۰۱ قرار دادیم.

- بخش سنجش:

در این بخش به ارزیابی چهار مدل پیاده سازی شده در این پروژه می پردازیم :

```
def evaluate_model(model, testloader):
    y_pred = []
    y_true = []
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            inputs, labels = data
            outputs = model(inputs.to("cuda"))
            _, predicted = torch.max(outputs, 1)
            y_pred.extend(predicted.cpu().numpy())
            y_true.extend(labels.cpu().numpy())
            total += labels.size(0)
            correct += (predicted == labels.to("cuda")).sum().item()
    accuracy = 100 * correct / total
    f1 = f1_score(y_true, y_pred, average='weighted')
    return accuracy, f1, y_true, y_pred

def plot_confusion_matrix(y_true, y_pred, classes):
    cf_matrix = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cf_matrix, annot=True, fmt='d', xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()
```

```
lenet_optimizer = optim.SGD(lenet_model.parameters(), lr=0.01)
resnet_optimizer = optim.Adam(resnet_model.parameters(), lr=0.01)
```

به این صورت که ابتدا هر مدل بر اساس ورودی دریافتی و خروجی که پیش بینی می کند، سنجش می شود و در هر مرحله خروجی با مقدار درست و اصلی دیتا مقایسه می شود. طبق این موارد می توان پارامتر های ارزیابی مدل را استخراج کرد. در این پروژه به بررسی و استخراج پارامتر های دقت (accuracy)، recall، f_1-score ، precision و confusion matrix می پردازیم و بر اساس فرمول های حاکم بر آنها مقدار عددی آنها بدست می آوریم.

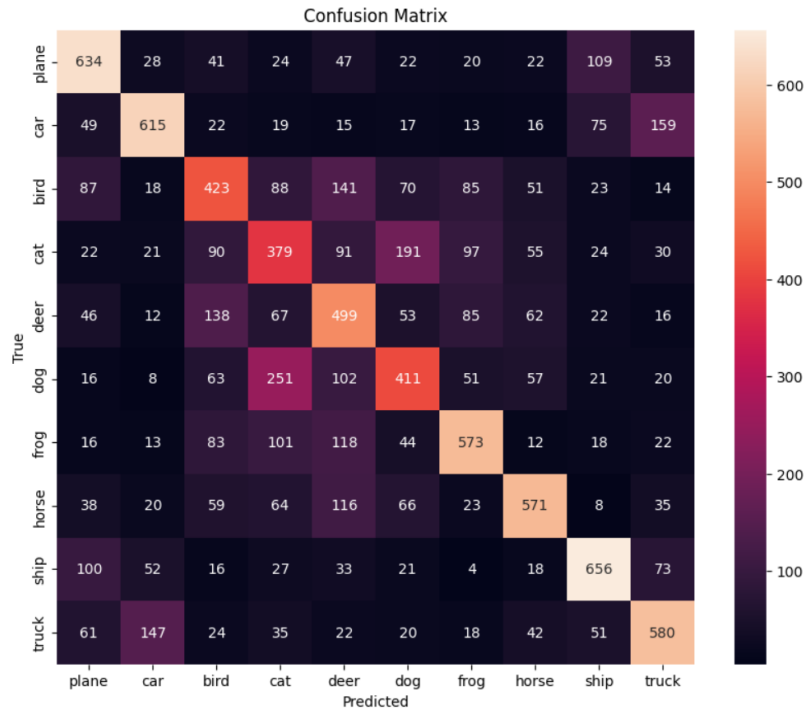
- ارزیابی مدل MLP و encoder/decoder :

```
# Initialize models
mlp_model = MLP()
encoder_decoder_model = EncoderDecoder()

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
mlp_optimizer = optim.Adam(mlp_model.parameters(), lr=0.001)
encoder_decoder_optimizer = optim.Adam(encoder_decoder_model.parameters(), lr=0.001)

# Train models
print("Training MLP Model")
train_model(mlp_model.to("cuda"), trainloader, criterion, mlp_optimizer)

print("Training Encoder-Decoder Model")
train_model(encoder_decoder_model.to("cuda"), trainloader, criterion, encoder_decoder_optimizer)

# Evaluate models
print("Evaluating MLP Model")
mlp_accuracy, mlp_f1, mlp_y_true, mlp_y_pred = evaluate_model(mlp_model, testloader)
print(f'MLP Test Accuracy: {mlp_accuracy:.2f}%')
print(f'MLP F1 Score: {mlp_f1:.2f}')
plot_confusion_matrix(mlp_y_true, mlp_y_pred, classes)
```

```
print("Evaluating Encoder-Decoder Model")
encoder_decoder_accuracy, encoder_decoder_f1, encoder_decoder_y_true, encoder_decoder_y_pred = evaluate_model(encoder_decoder_model, testloader)
print(f'Encoder-Decoder Test Accuracy: {encoder_decoder_accuracy:.2f}%')
print(f'Encoder-Decoder F1 Score: {encoder_decoder_f1:.2f}')
plot_confusion_matrix(encoder_decoder_y_true, encoder_decoder_y_pred, classes)
```

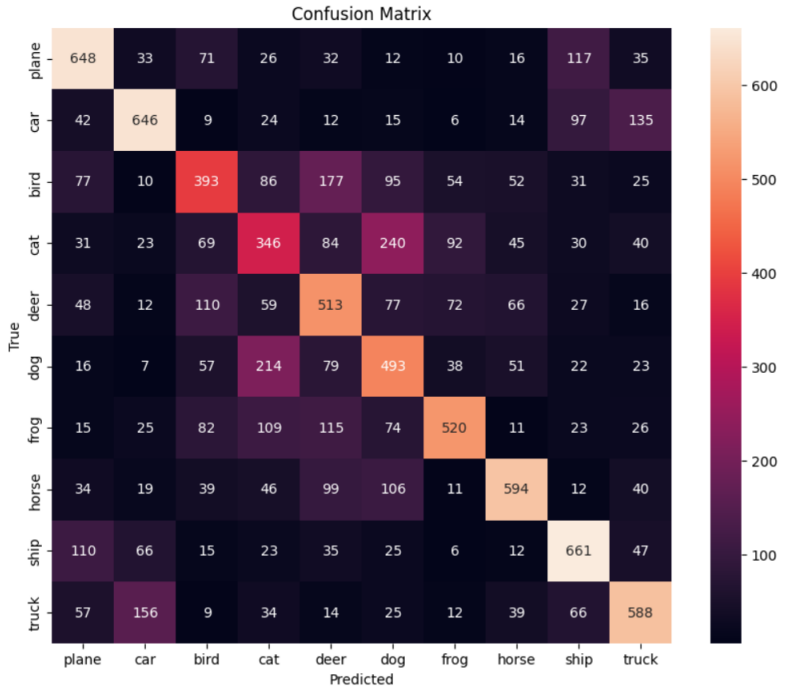
در این بخش به ارزیابی دو MLP و Encoder/decoder پرداختیم و پارامتر های ارزیابی که در بالا ذکر شد را بر هر کدام از این مدل ها بدست می آوریم :

```
MLP Test Accuracy: 53.41%
MLP F1 Score: 0.54
```



نتایج بدست آمده از MLP

```
Evaluating Encoder-Decoder Model
Encoder-Decoder Test Accuracy: 54.02%
Encoder-Decoder F1 Score: 0.54
```



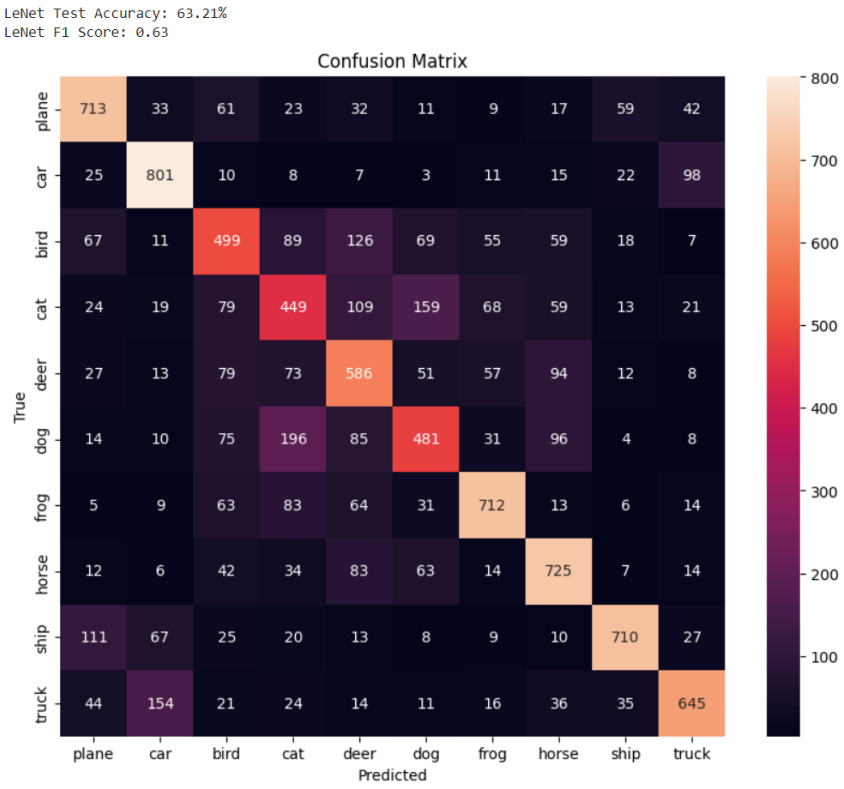
نتایج بدست آمده از Ecoder/decoder

- ارزیابی مدل Lenet و Resnet :

```python
# Initialize models
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

mlp_model = MLP().to(device)
encoder_decoder_model = EncoderDecoder().to(device)
lenet_model = LeNet().to(device)
resnet_model = ResNet18().to(device)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()

lenet_optimizer = optim.Adam(lenet_model.parameters(), lr=0.001)
resnet_optimizer = optim.Adam(resnet_model.parameters(), lr=0.001)
print("Training ResNet Model")
train_model(resnet_model, trainloader, criterion, resnet_optimizer)

print("Training LeNet Model")
train_model(lenet_model, trainloader, criterion, lenet_optimizer)
```

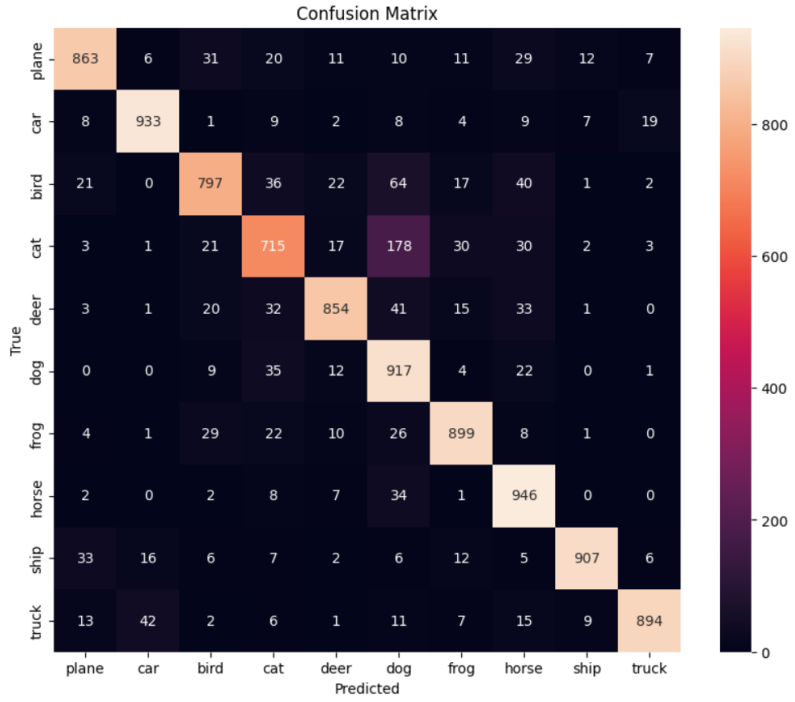
```
print("Evaluating LeNet Model")
lenet_accuracy, lenet_f1, lenet_y_true, lenet_y_pred = evaluate_model(lenet_model, testloader)
print(f'LeNet Test Accuracy: {lenet_accuracy:.2f}%')
print(f'LeNet F1 Score: {lenet_f1:.2f}')
plot_confusion_matrix(lenet_y_true, lenet_y_pred, classes)

print("Evaluating ResNet Model")
resnet_accuracy, resnet_f1, resnet_y_true, resnet_y_pred = evaluate_model(resnet_model, testloader)
print(f'ResNet Test Accuracy: {resnet_accuracy:.2f}%')
print(f'ResNet F1 Score: {resnet_f1:.2f}')
plot_confusion_matrix(resnet_y_true, resnet_y_pred, classes)
```

در این بخش به ارزیابی دو Lenet و Resnet پرداختیم و پارامتر های ارزیابی که در بالا ذکر شد را بر هر کدام از این مدل ها بدست می آوریم.



نتایج بدست آمده از Lenet

```
Evaluating ResNet Model
ResNet Test Accuracy: 87.25%
ResNet F1 Score: 0.87
```



نتایج بدست آمده از Resnet

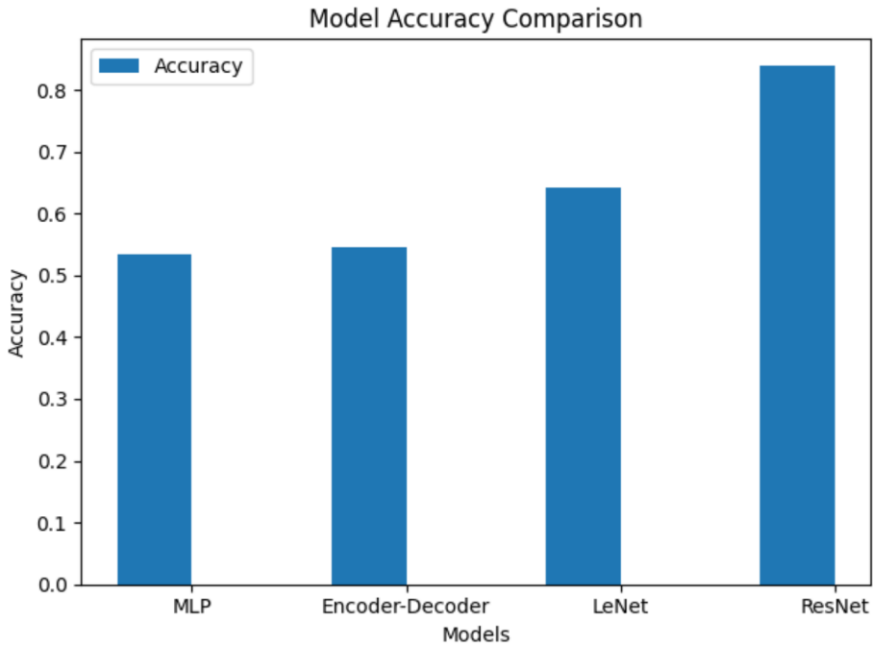
نتیجه گیری :

در نهایت برای مقایسه راحت و دقیق تر دقت و عملکرد چهار مدل پیاده سازی شده تمام نتایج را کنار هم گذاشته و داریم :

```python
# Evaluate models
print("Evaluating MLP Model")
mlp_accuracy, mlp_f1, mlp_y_true, mlp_y_pred = evaluate_model(mlp_model, testloader)
print(f'MLP Test Accuracy: {mlp_accuracy:.2f}%')
print(f'MLP F1 Score: {mlp_f1:.2f}')
plot_confusion_matrix(mlp_y_true, mlp_y_pred, classes)

print("Evaluating Encoder-Decoder Model")
encoder_decoder_accuracy, encoder_decoder_f1, encoder_decoder_y_true, encoder_decoder_y_pred = evaluate_model(encoder_decoder_model, testloader)
print(f'Encoder-Decoder Test Accuracy: {encoder_decoder_accuracy:.2f}%')
print(f'Encoder-Decoder F1 Score: {encoder_decoder_f1:.2f}')
plot_confusion_matrix(encoder_decoder_y_true, encoder_decoder_y_pred, classes)



print("Evaluating LeNet Model")
lenet_accuracy, lenet_f1, lenet_y_true, lenet_y_pred = evaluate_model(lenet_model, testloader)
print(f'LeNet Test Accuracy: {lenet_accuracy:.2f}%')
print(f'LeNet F1 Score: {lenet_f1:.2f}')
plot_confusion_matrix(lenet_y_true, lenet_y_pred, classes)

print("Evaluating ResNet Model")
resnet_accuracy, resnet_f1, resnet_y_true, resnet_y_pred = evaluate_model(resnet_model, testloader)
print(f'ResNet Test Accuracy: {resnet_accuracy:.2f}%')
print(f'ResNet F1 Score: {resnet_f1:.2f}')
plot_confusion_matrix(resnet_y_true, resnet_y_pred, classes)
```

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import f1_score, accuracy_score, recall_score, precision_score

def calculate_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    precision = precision_score(y_true, y_pred, average='weighted')
    return accuracy, f1, recall, precision

def plot_metrics_comparison(models_metrics):
    models = list(models_metrics.keys())
    accuracies = [metrics['accuracy'] for metrics in models_metrics.values()]

    x = np.arange(len(models))
    width = 0.35

    fig, ax = plt.subplots()
    rects1 = ax.bar(x - width/2, accuracies, width, label='Accuracy')

    ax.set_xlabel('Models')
    ax.set_ylabel('Accuracy')
    ax.set_title('Model Accuracy Comparison')
    ax.set_xticks(x)
    ax.set_xticklabels(models)
    ax.legend()
```

```python
# Calculate metrics for each model
mlp_metrics = calculate_metrics(mlp_y_true, mlp_y_pred)
encoder_decoder_metrics = calculate_metrics(encoder_decoder_y_true, encoder_decoder_y_pred)
lenet_metrics = calculate_metrics(lenet_y_true, lenet_y_pred)
resnet_metrics = calculate_metrics(resnet_y_true, resnet_y_pred)

# Store metrics in a dictionary for comparison
models_metrics = {
    'MLP': {'accuracy': mlp_metrics[0], 'f1': mlp_metrics[1], 'recall': mlp_metrics[2], 'precision': mlp_metrics[3]},
    'Encoder-Decoder': {'accuracy': encoder_decoder_metrics[0], 'f1': encoder_decoder_metrics[1], 'recall': encoder_decoder_metrics[2], 'precision': encoder_decoder_metrics[3]},
    'LeNet': {'accuracy': lenet_metrics[0], 'f1': lenet_metrics[1], 'recall': lenet_metrics[2], 'precision': lenet_metrics[3]},
    'ResNet': {'accuracy': resnet_metrics[0], 'f1': resnet_metrics[1], 'recall': resnet_metrics[2], 'precision': resnet_metrics[3]}
}

# Plot the comparison
plot_metrics_comparison(models_metrics)

# Print the metrics for each model
for model, metrics in models_metrics.items():
    print(f"{model} Metrics:")
    print(f"  Accuracy: {metrics['accuracy']:.5f}")
    print(f"  F1 Score: {metrics['f1']:.5f}")
    print(f"  Recall: {metrics['recall']:.5f}")
    print(f"  Precision: {metrics['precision']:.5f}")
    print()
```



	Accuracy	F_1-score	Recall	precision
MLP	0.5346	0.5353	0.5346	0.5411
Encoder/decoder	0.5447	0.5438	0.5447	0.5441
Lenet	0.6425	0.6419	0.6425	0.6451
Resnet	0.8701	0.8699	0.8701	0.8719

در انتها نتیجه می گیریم که هر چه مدل دارای پارامتر های اصلی و ساختار پیچیده باشد می تواند بهتر یادگیری انجام دهد و نیز دقت و عملکرد بهتری از خود نسبت به بقیه مدل ها نشان بدهد. همچنین به صورت کلی می توان گفت که شبکه های عصبی کانولوشنی نسبت به شبکه های عصبی ساده دارای کارایی و عملکرد بهتری هستند و در پردازش نیز تعداد نورون های کمتری نیاز دارند. این در حالی است که در شبکه های عصبی ساده اگر دیتا بزرگ شوند لایه ها و تعداد نورون های شبکه عصبی به شدت افزایش می یابد و در اصل کار پردازش را نا ممکن می سازد. همچنین تفاوت چندانی در استفاده از بهینه ساز های مختلف مشاهده نشد.