

# ResNet

class ResNet18(nn.Module):def resnet\_block(self, input\_channels, num\_channels, num\_residuals,  
 first\_block):  
 blk = []  
 for i in range(num\_residuals):  
 if i == 0 and not first\_block:  
 blk.append(ResidualBlock(input\_channels, num\_channels,  
 strides=2))  
 else:  
 blk.append(ResidualBlock(num\_channels, num\_channels))  
 return blk  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 block1 = nn.Sequential(nn.Conv2d(3, 64, kernel\_size=7, stride=2, padding=3), nn.BatchNorm2d(64), nn.ReLU(), nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
 block2 = nn.Sequential(\*self.resnet\_block(64, 64, 2, first\_block=True))  
 block3 = nn.Sequential(\*self.resnet\_block(64, 128, 2, first\_block=False))  
 block4 = nn.Sequential(\*self.resnet\_block(128, 256, 2, first\_block=False))  
 block5 = nn.Sequential(\*self.resnet\_block(256, 512, 2, first\_block=False))  
 self.feature = nn.Sequential(block1, block2, block3, block4, block5, nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())  
 self.fc = nn.Linear(512,10)  
 def forward(self,X):  
 X = self.fc(self.feature(X))  
 return X

# Residual Block

class ResidualBlock(nn.Module):  
 def \_\_init\_\_(self, input\_channels, num\_channels,  
 strides=1):super().\_\_init\_\_()  
 self.conv1 = nn.Conv2d(input\_channels, num\_channels, kernel\_size=3, padding=1, stride=strides)  
 self.conv2 = nn.Conv2d(num\_channels, num\_channels, kernel\_size=3, padding=1)  
 if strides!=1 or input\_channels!=num\_channels:  
 self.conv3 = nn.Conv2d(input\_channels, num\_channels,  
 kernel\_size=1, stride=strides)  
 else:  
 self.conv3 = None  
 self.bn1 = nn.BatchNorm2d(num\_channels)  
 self.bn2 = nn.BatchNorm2d(num\_channels)  
def forward(self, X):Y = F.relu(self.bn1(self.conv1(X)))  
 Y = self.bn2(self.conv2(Y))  
 if self.conv3:  
 X = self.conv3(X)  
 Y += X  
 return F.relu(Y)

# Dilated Convolution

def dilated\_conv2d(inputs,kernels,dilation,padding=0,stride=1):  
 assert inputs.shape[0] == kernels.shape[1], "The numbers of channels of input and kernel do not match."outputs\_c, kernel\_c, kernel\_h, kernel\_w = kernels.shape  
 new\_kernels = np.zeros((outputs\_c, kernel\_c, (kernel\_h-1)\*dilation+1, (kernel\_w-1)\*dilation+1))  
 for c in range(outputs\_c):  
 for kc in range(kernel\_c):  
 for kh in range(kernel\_h):  
 for kw in range(dilation):  
 new\_kernels[c, kc, kh\*dilation, kw\*dilation] = kernels[c, kc, kh, kw]  
 outputs = conv2d(inputs, new\_kernels, padding, stride)return outputs

# Convolution

def conv2d(inputs,kernels,padding=0, stride=1):  
 assert inputs.shape[0] == kernels.shape[1], "The numbers of channels of input and kernel do not match."c, h, w = inputs.shape  
 output\_c, kerel\_c, kernel\_h, kernel\_w = kernels.shape  
 output\_h = (h + 2 \* padding - kernel\_h) // stride + 1  
 output\_w = (w + 2 \* padding - kernel\_w) // stride + 1  
 padded\_inputs = np.zeros((c, h + 2 \* padding, w + 2 \* padding))  
 padded\_inputs[:, padding: (h + padding), padding: (w + padding)] = inputs  
 outputs = np.zeros((output\_c, output\_h, output\_w))  
 for k in range(output\_c):  
 kernel = kernels[k, :, :]  
 for i in range(output\_h):  
 for j in range(output\_w):  
 window = padded\_inputs[:, i\*stride:i\*stride+kernel\_h, j\*stride:j\*stride+kernel\_w]  
 outputs[k, i, j] = np.sum(kernel \* window)return outputs

# Evaluate Model

def eval\_on\_test\_set(model):  
 model.eval()  
 device = torch.cuda.current\_device() if torch.cuda.is\_available() else "cpu"  
 model.to(device)  
 running\_accuracy = 0  
 loss=0  
 for data in test\_loader:inputs,labels = data  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
 *# Forward* outputs = model(inputs)  
 *# Calculate loss* loss += criterion(outputs,labels).item()\*batch\_size

*# Calculate accuracy*

running\_accuracy += (outputs.argmax(dim=-1) == labels).float().sum()

total\_loss=loss/test\_size  
 total\_accuracy = running\_accuracy / test\_size  
 print('Evaluation on test set: loss{:.3f} \t accuracy = {:.2f}%'.format(total\_loss, total\_accuracy \* 100))  
 model.train()  
 return total\_loss, total\_accuracy

# Learned Feature of Different Layers

index\_top = 7  
index\_bottom = 1  
features\_top, features\_bottom, labels = [], [], []  
for batch in test\_loader:  
 imgs, lbls = batch  
 with torch.no\_grad():  
 logits\_top = model.feature[:index\_top](imgs.to(device))  
 logits\_top = logits\_top.view(logits\_top.size()[0], -1)  
 logits\_bottom = model.feature[:index\_bottom](imgs.to(device))  
 logits\_bottom = logits\_bottom.view(logits\_bottom.size()[0], -1)  
 labels.extend(lbls.cpu().numpy())  
 logits\_top = np.squeeze(logits\_top.cpu().numpy())  
 features\_top.extend(logits\_top)  
 logits\_bottom = np.squeeze(logits\_bottom.cpu().numpy())  
 features\_bottom.extend(logits\_bottom)

*# Hyperparameters*epochs = 10  
batch\_size = 256  
learning\_rate = 0.1  
*# Set up optimizer*optimizer = optim.SGD(model.parameters(), lr=learning\_rate)  
*# Define loss function*criterion = torch.nn.CrossEntropyLoss()  
*# Build data loaders*train\_loader = torch.utils.data.DataLoader(train\_set, batch\_size=batch\_size, shuffle=True, num\_workers=0)  
test\_loader = torch.utils.data.DataLoader(test\_set, batch\_size=batch\_size, shuffle=False, num\_workers=0)  
data\_loaders = {"train": train\_loader, "test": test\_loader}  
dataset\_sizes = {"train": train\_size, "test": test\_size}

def forward(self, x):h0 = torch.zeros(self.num\_layers, x.size(0), self.hidden\_size).to(x.device)out, \_ = self.rnn(x, h0) *# out: tensor of shape (batch\_size, seq\_length, hidden\_size)* out = self.fc(out[:, -1, :])  
 return out

# Quantize  
def quantize(data, levels):thresholds = np.linspace(np.min(data), np.max(data), levels + 1)[1:-1]quantized\_data = np.digitize(data, thresholds)  
 return quantized\_datadata = np.array([1.2, 2.3, 3.4, 4.5, 5.6])  
levels = 3  
quantized\_data = quantize(data, levels)

# RNN/LSTM  
class RNNNet(nn.Module):  
 def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, num\_layers):  
 super(RNNNet, self).\_\_init\_\_()  
 self.hidden\_size = hidden\_size  
 self.num\_layers = num\_layersself.rnn = nn.RNN(input\_size, hidden\_size, num\_layers, batch\_first=True)self.fc = nn.Linear(hidden\_size, output\_size)

# Backward of MLP

def relu\_derivative(x):  
 return np.where(x > 0, 1, 0)  
def softmax\_derivative(x):  
 return x \* (1 - x)  
def cross\_entropy\_derivative(y\_pred, y\_true):  
 return y\_pred - y\_true  
def mlp\_backward(X, y, W1, b1, W2, b2, A1, A2, learning\_rate):  
 *# Compute the gradient of the loss with respect to A2* dA2 = cross\_entropy\_derivative(A2, y)  
 *# Compute the gradient of the loss with respect to Z2* dZ2 = dA2 \* softmax\_derivative(A2)  
 *# Compute the gradient of the loss with respect to W2 and b2* dW2 = np.dot(A1.T, dZ2)  
 db2 = np.sum(dZ2, axis=0, keepdims=True)  
 *# Compute the gradient of the loss with respect to A1* dA1 = np.dot(dZ2, W2.T)  
 *# Compute the gradient of the loss with respect to Z1* dZ1 = dA1 \* relu\_derivative(A1)  
 *# Compute the gradient of the loss with respect to W1 and b1* dW1 = np.dot(X.T, dZ1)  
 db1 = np.sum(dZ1, axis=0, keepdims=True)  
 *# Update the weights and biases* W1 -= learning\_rate \* dW1  
 b1 -= learning\_rate \* db1  
 W2 -= learning\_rate \* dW2  
 b2 -= learning\_rate \* db2  
 return W1, b1, W2, b2

# SmoothGrad

from torch.autograd import Variable  
def compute\_smoothgrad(X, y, model, num\_samples=50, stdev\_spread=0.15): *Inputs:  
 - X: Input images; Tensor of shape (N, 3, H, W)  
 - y: Labels for X; Tensor of shape (N,)* model.eval()X.requires\_grad\_()  
 for i in range(num\_samples):  
 noise = Variable(X.data.new(X.size()).normal\_(0, stdev\_spread\*\*2))  
 X\_noise = X + noise  
 X\_noise.requires\_grad\_()  
 scores = model(X\_noise)  
 scores = (scores.gather(1, y.view(-1, 1)).squeeze())  
 scores.backward(torch.FloatTensor([1.0]\*scores.shape[0]).to(X.device))if i == 0:  
 smoothgrad = X.grad.data.abs()  
 else:  
 smoothgrad += X.grad.data.abs()smoothgrad /= num\_samplessmoothgrad = (smoothgrad - smoothgrad.min()) / (smoothgrad.max() - smoothgrad.min())  
 return smoothgrad

# Forward of MLP  
def relu(x):  
 return np.maximum(0, x)  
def softmax(x):  
 e\_x = np.exp(x - np.max(x))  
 return e\_x / e\_x.sum(axis=0)  
def mlp\_forward(X, W1, b1, W2, b2):  
 *# Compute the input to the hidden layer* Z1 = np.dot(X, W1) + b1  
 *# Apply the activation function (e.g., ReLU)* A1 = relu(Z1)  
 *# Compute the input to the output layer* Z2 = np.dot(A1, W2) + b2  
 *# Apply the activation function (e.g., softmax)* A2 = softmax(Z2)  
 return A2

# Fooling Image

def make\_fooling\_image(X, target\_y, model): *- X: Input image; Tensor of shape (1, 3, H, W)  
 - target\_y: An integer in the range [0, 10)  
 - model: A pretrained CNN* X\_fooling = X.clone()  
 X\_fooling = X\_fooling.requires\_grad\_()  
 learning\_rate = 1while True:  
 scores = model(X\_fooling)  
 \_, idx = torch.max(scores, 1)  
 if (idx != target\_y):  
 scores[:,target\_y].backward()  
 dX = learning\_rate\*X\_fooling.grad.data/torch.norm(X\_fooling.grad.data)  
 X\_fooling.data += dX.data  
 X\_fooling.grad.data.zero\_()  
 else:  
 breakreturn X\_fooling

# Saliency Maps

def compute\_saliency\_maps(X, y, model): *- X: Input images; Tensor of shape (N, 3, H, W)  
 - y: Labels for X; Tensor of shape (N,)  
 - model: A pretrained CNN that will be used to compute the saliency map.  
 Returns:  
 - saliency: A Tensor of shape (N, H, W) giving the saliency maps for the input images.* model.eval() *# Make input tensor require gradient* X.requires\_grad\_()  
 saliency = None  
 *#forward pass* scores = model(X)  
 *#choose the score corresponding to the ground truth class for each image* scores = (scores.gather(1, y.view(-1, 1)).squeeze())  
 *#backward pass* scores.backward(torch.FloatTensor([1.0]\*scores.shape[0]).to(X.device))  
 *#saliency* saliency, \_ = torch.max(X.grad.data.abs(), dim=1)return saliency

# Train Model

def train\_for\_one\_epoch(model):  
 model.train()device = torch.cuda.current\_device() if torch.cuda.is\_available() else "cpu"  
 print(f"Using device {device} to train the model.")  
 model.to(device)running\_loss = 0  
 running\_accuracy = 0  
 for data in train\_loader:inputs, labels = data  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
 *# zero the parameter gradients* optimizer.zero\_grad()  
 *# forward + backward + optimize* outputs = model(inputs)  
 loss = criterion(outputs, labels)  
 loss.backward()  
 optimizer.step()  
 *# Compute batch loss* running\_loss += loss.item()\*batch\_size  
 *# Compute batch accuracy* running\_accuracy += (outputs.argmax(dim=-1) == labels).float().sum() *# Compute stats for the full training set* total\_loss = running\_loss / train\_size  
 total\_accuracy = running\_accuracy / train\_size  
 return total\_loss, total\_accuracy