Modelo en Pytorch

En esta seccion importamos las funciones que se usaran en la creacion del modelo AlexNet desde scratch en Pytorch, asi como verificar si se puede usar la tarjeta grafica integrada de la PC.

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
import torchvision
import torch.optim as optim
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Creacion del modelo

En esta parte ya creamos el modelo AlexNet con los requisitos mencionados en calse, 5 convoluciones, 3 maxpoolings y 3 redes densas con activciones ReLu y dos funciones dropout para evitar el overfitting.

In [2]:
```
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.fc1 = nn.Linear(9216, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 10)
        self.cn1 = nn.Conv2d(3, 64, 11, stride=4)
        self.cn2 = nn.Conv2d(64, 192, 5, stride=1)
        self.cn3 = nn.Conv2d(192, 384, 3, stride=1)
        self.cn4 = nn.Conv2d(384, 256, 3, stride=1)
        self.cn5 = nn.Conv2d(256, 256, 3, stride=1)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = F.relu(self.cn1(x))
        x = F.max_pool2d(x, (3, 3), stride = 2)
        x = F.relu(self.cn2(x))
        x = F.max_pool2d(x, (3, 3), stride = 2)
        x = F.relu(self.cn3(x))
        x = F.relu(self.cn4(x))
        x = F.relu(self.cn5(x))
        x = F.max_pool2d(x, (3, 3), stride = 2)
        x = F.adaptive_avg_pool2d(x, (6, 6))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)

        return x
```

Asignamos el modelo a la GPU

In [3]:
```
model = AlexNet().to(device)
```

Carga de datos

En esta seccion se usa el mismo codigo que se vio en clase para la carga de datos, pero con un batch size distinto, en este caso se uso 128 ya que asi lo permitio la GPU

In [4]:
```
# 1. Load CIFAR-10 dataset
transform = transforms.Compose([
    transforms.Resize(224),  # AlexNet expects 224x224 images
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=128,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Se defininen el criterio de optimizacion y la funcion de optimizacion.

In [5]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005, momentum=0.9)
```

Entrenamineto

En esta celda se entrena por 20 epocas el modelo tardando de 15 a 20 minutos.

In [6]:
```
for epoch in range(20):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        outputs = model(inputs)

        labels = labels.long()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        running_loss += loss.item()
    print(f'[{epoch + 1}] loss: {running_loss / 2000:.3f}')
    running_loss = 0.0

print('Finished Training')
```

```
[1] loss: 0.445
[2] loss: 0.375
[3] loss: 0.311
[4] loss: 0.259
[5] loss: 0.225
[6] loss: 0.195
[7] loss: 0.172
[8] loss: 0.152
[9] loss: 0.133
[10] loss: 0.117
[11] loss: 0.102
[12] loss: 0.086
[13] loss: 0.072
[14] loss: 0.062
[15] loss: 0.052
[16] loss: 0.043
[17] loss: 0.038
[18] loss: 0.032
[19] loss: 0.029
[20] loss: 0.025
Finished Training
```

Prueba del modelo

Se prueba el modelo con los datos de testing, y en general se obtinen una accuracy de 80%

In [7]:
```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct / total:.2f} %')
```

```
Accuracy of the network on the 10000 test images: 75.33 %
```

In [8]:
```
torch.save(model.state_dict(), 'modelo_entrenado.pth')
```

In [9]:
```
torch.save(model, 'modelo_completo.pth')
```

Se evalua el modelo por categoria

In [10]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print(f'Accuracy of {classes[i]:5s} : {100 * class_correct[i] / class_total[i]:.2f} %')
```

```
Accuracy of plane : 79.31 %
Accuracy of car   : 78.57 %
Accuracy of bird  : 72.73 %
Accuracy of cat   : 52.94 %
Accuracy of deer  : 70.37 %
Accuracy of dog   : 72.73 %
Accuracy of frog  : 80.56 %
Accuracy of horse : 72.00 %
Accuracy of ship  : 84.38 %
Accuracy of truck : 79.49 %
```

Modelo pre entrenado en Pytorch de AlexNet

Cargamos el mdoelo preentrenado de AlexNet con solo 10 categorias

In [12]:
```
modelp = torchvision.models.alexnet(pretrained=True)
modelp.eval()
num_features = modelp.classifier[6].in_features
modelp.classifier[6] = nn.Linear(num_features, 10) # 10 output classes
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(modelp.parameters(), lr=0.001, momentum=0.9)
modelp.to(device)
```

```
/home/galo/Diplomado/python/venv/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in th
e future, please use 'weights' instead.
  warnings.warn(
/home/galo/Diplomado/python/venv/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated si
nce 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=AlexNet_Weights.IMAGENET1K_V1`. You can also use `weights=AlexNet_Weights.DEFAULT` to g
et the most up-to-date weights.
  warnings.warn(msg)
```

Out[12]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Entrenamiento del modelo preentrenado

In [13]:
```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        outputs = modelp(inputs)
        # Ensure labels are long type
        labels = labels.long()
        loss = criterion(outputs, labels) # Corrected order: outputs, labels
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f'[{epoch + 1}] loss: {running_loss / 2000:.3f}')
    running_loss = 0.0

print('Finished Training')
```

```
[1] loss: 0.127
[2] loss: 0.075
Finished Training
```

Evaluacion del medoleo preentrenado

In [14]:
```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = modelp(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct / total:.2f} %')
```

```
Accuracy of the network on the 10000 test images: 86.38 %
```

In [15]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = modelp(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print(f'Accuracy of {classes[i]:5s} : {100 * class_correct[i] / class_total[i]:.2f} %')
```

```
Accuracy of plane : 82.76 %
Accuracy of car   : 92.86 %
Accuracy of bird  : 93.94 %
Accuracy of cat   : 82.35 %
Accuracy of deer  : 85.19 %
Accuracy of dog   : 66.67 %
Accuracy of frog  : 88.89 %
Accuracy of horse : 92.00 %
Accuracy of ship  : 93.75 %
Accuracy of truck : 97.44 %
```

Coomparacion de los modelos

Es claro que el modelo preentrenado es mmucho mas rapido y eficaz que el modleo que se creo desde cero, el modleo preentrenado sse tardo aproximadamente dos minutos y medio en completar tan solo dos epocas en cambio el nuevo se tardo veinte minutos para 20 epocas, en cuanto a epocas hablamso se tardan masomenos lo mismo. Un punto que creo haber notaod es que en el nuevo modelo parece haber overfitting ya que la funcion de perdida disminuye muy rapido y ademas es incluso menor a la del modelo preentrenado, eso quiere decir que para los datos con los que se entrena parece funcionar demasiado bien pero al salir a conocer nuevos datos no resulta tan efectivo. Con metodos que traten de evitar el overfitting creo es posible mejorar aun mas el modelo. Otro punto que me parece interesante mencionar es que a ambos modelos les cuesta trabajo reconocer a los gatos y a los perros, y el modleo preentrenado parece tener mas variacion en la precision de las predicciones, en cambio el nuevo es peor pero mas constante.

Modelo en Tensor Flow

Se carga las librerias necesarias para la creacion del modelo

In [7]:
```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
```

Se cargan los datos del modelo y se normalizan

In [2]:
```
# 1. Cargar y preprocesar el dataset CIFAR-10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalizar las imágenes (de 0 a 1)
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Convertir las etiquetas a formato one-hot
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Creacion del modelo

Se tarta de crear un modelo similar al de Pytorch y al visto en clase, con pequenas modificaciones para el correcto funcionamiento y normalizaciones de los pesos. Agregue tambien dropouts para no hacer un overfitting de los datos de entrenamiento

In [5]:
```
#. Creamos el modelo
model = models.Sequential()
model.add(layers.Conv2D(96, 3, strides=1, padding='same', input_shape=(32, 32, 3)))
model.add(layers.Lambda(tf.nn.local_response_normalization))
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D(2, strides=2))
model.add(layers.Conv2D(192, 3, strides=1, padding='same'))
model.add(layers.Lambda(tf.nn.local_response_normalization))
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D(2, strides=2))
model.add(layers.Conv2D(384, 3, strides=1, padding='same'))
model.add(layers.Activation('relu'))
model.add(layers.Conv2D(256, 3, strides=1, padding='same'))
model.add(layers.Activation('relu'))
model.add(layers.Conv2D(256, 3, strides=1, padding='same'))
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D(3, strides=2))
model.add(layers.GloblaAvergaePooling2D())
model.add(layers.Flatten())
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 32, 32, 96)	2,688
lambda (Lambda)	(None, 32, 32, 96)	0
activation_5 (Activation)	(None, 32, 32, 96)	0
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 96)	0
conv2d_6 (Conv2D)	(None, 16, 16, 192)	166,080
lambda_1 (Lambda)	(None, 16, 16, 192)	0
activation_6 (Activation)	(None, 16, 16, 192)	0
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 192)	0
conv2d_7 (Conv2D)	(None, 8, 8, 384)	663,936
activation_7 (Activation)	(None, 8, 8, 384)	0
conv2d_8 (Conv2D)	(None, 8, 8, 256)	884,992
activation_8 (Activation)	(None, 8, 8, 256)	0
conv2d_9 (Conv2D)	(None, 8, 8, 256)	590,080
activation_9 (Activation)	(None, 8, 8, 256)	0
max_pooling2d_5 (MaxPooling2D)	(None, 3, 3, 256)	0
flatten_1 (Flatten)	(None, 2304)	0
dense_3 (Dense)	(None, 4096)	9,441,280
dropout_2 (Dropout)	(None, 4096)	0
dense_4 (Dense)	(None, 4096)	16,781,312
dropout_3 (Dropout)	(None, 4096)	0
dense_5 (Dense)	(None, 10)	40,970

Total params: 28,571,338 (108.99 MB)
Trainable params: 28,571,338 (108.99 MB)
Non-trainable params: 0 (0.00 B)

Entrenamiento

Entrenamos el modelo con 4 epocas y un batch size de 256, tardando alrededor de 20 minutos en completarse el entrenamiento, se escogieron 4 epocas solamente ya que no me fue posible usar la tarejeta grafica para este modelo, con un poco mas de tiempo la hubiera dejado a la par que su contraparte de Pytorch, es decir 20 epocas.

Vemos que con tan solo 4 epocas tiene un accuracy del 55%, un poco bastante malo a decir verdad, pero hay que tomar en cuenta la baja cantidad de epocas comparada con otros modelos.

In [ ]:
```
# 3. Compilar el modelo
model.compile(optimizer=optimizers.SGD( learning_rate=0.001,momentum=0.9),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# 4. Entrenar el modelo
history = model.fit(x_train, y_train, batch_size=256, epochs=4)

# 5. Evaluar el modelo en el conjunto de prueba
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```

```
Epoch 1/4
2025-03-07 12:30:41.543370: W external/local_xla/xla/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 614400000 exceeds 10% of free system memory.
196/196 ━━━━━━━━━━━━━━━━━━━━ 328s 2s/step - accuracy: 0.1284 - loss: 2.2778
Epoch 2/4
196/196 ━━━━━━━━━━━━━━━━━━━━ 310s 2s/step - accuracy: 0.2681 - loss: 1.9423
Epoch 3/4
196/196 ━━━━━━━━━━━━━━━━━━━━ 309s 2s/step - accuracy: 0.3948 - loss: 1.6424
Epoch 4/4
196/196 ━━━━━━━━━━━━━━━━━━━━ 302s 2s/step - accuracy: 0.4735 - loss: 1.4386
313/313 ━━━━━━━━━━━━━━━━━━━━ 17s 54ms/step - accuracy: 0.5542 - loss: 1.2465
Test accuracy: 0.5507000088691711
```

Modelo preentrenado

Puesto que no hay un mocelo preentrenado de AlexNet en Tensor Flow, use el modleo VGG16 que se puede encontrar ya preentrenad en keras.

In [43]:
```
from keras.applications import vgg16 as vgg
from keras.layers import Dropout, Dense, GlobalAveragePooling2D,BatchNormalization
from keras import Model
```

Cargamos el modelo con la forma deseada de CIFAR10 y sin la ultima capa

In [39]:
```
base_model = vgg.VGG16(weights='imagenet',
                       include_top=False,
                       input_shape=(32, 32, 3))
```

Enntrenamiento

Se modifica la ultima capa para entrenar los pesos de esta capa. Se entrena con un batch size de 256 y 6 epocas, se siguen realizando los calculos en el CPU pero ya son muchos menos que con el mdoelo desde cero, es por esto que solo se tardo 5 minutos en procesar.

In [41]:
```
last = base_model.get_layer('block3_pool').output
# Add classification layers on top of it
x = GlobalAveragePooling2D()(last)
x= BatchNormalization()(x)
x = Dense(256, activation='relu')(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.6)(x)
pred = Dense(10, activation='softmax')(x)
model = Model(base_model.input, pred)
for layer in base_model.layers:
    layer.trainable = False
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.Adam(learning_rate=0.01),
              metrics=['accuracy'])
history = model.fit(x_train, y_train, batch_size=256, epochs=6)
```

```
Epoch 1/6
2025-03-07 13:35:25.321220: W external/local_xla/xla/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 614400000 exceeds 10% of free system memory.
196/196 ━━━━━━━━━━━━━━━━━━━━ 54s 264ms/step - accuracy: 0.4883 - loss: 0.2567
Epoch 2/6
196/196 ━━━━━━━━━━━━━━━━━━━━ 53s 271ms/step - accuracy: 0.6629 - loss: 0.1660
Epoch 3/6
196/196 ━━━━━━━━━━━━━━━━━━━━ 53s 268ms/step - accuracy: 0.6968 - loss: 0.1519
Epoch 4/6
196/196 ━━━━━━━━━━━━━━━━━━━━ 54s 277ms/step - accuracy: 0.7133 - loss: 0.1433
Epoch 5/6
196/196 ━━━━━━━━━━━━━━━━━━━━ 54s 278ms/step - accuracy: 0.7301 - loss: 0.1364
Epoch 6/6
196/196 ━━━━━━━━━━━━━━━━━━━━ 51s 260ms/step - accuracy: 0.7357 - loss: 0.1335
```

Como era de esperar el modelo preentreando tardo mucho mennos y ademas tuvo una precision mucho mayor a la creada desde cero, un accuracy de alrdedor de 71%

In [42]:
```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 11s 36ms/step - accuracy: 0.7140 - loss: 0.1460
Test accuracy: 0.7129999995231628
```

Comparacion de los modelos en Tensor Flow

Sabemos que debe de ser mejor el modelo preentrenado pero realmente nuestro modleo no se quedo taan lejos de porder ser igual, yo creo que con dos epocas mas de entrenamiento de nuestro modelo se podria haber llegado a el accuracy de 71%. Tambien si logro poder usar la tarjeta rafica con este programa se podria mejorar exponensialmente el modelo y llegar hasta las 20 epocas como se hizo con Pytorch a pesar de no tener una GPU con grandes prestaciones.