Laboratorio 7

Manuel Archila 161250

Diego Franco 20240

Juan Diego Avila 20090

Task 1 - Práctica

Ejercicio 1

```
import torch

print(torch.cuda.is_available())
```

```
True
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms

# Definición del modelo
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, padding=2)
        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.tanh(F.avg_pool2d(self.conv1(x), 2))
        x = torch.tanh(F.avg_pool2d(self.conv2(x), 2))
        x = x.view(-1, 16*5*5)
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

```
# Hiperparámetros
BATCH_SIZE = 64
EPOCHS = 15
LR = 0.01

# Transformaciones y carga del dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)

# Instancia del modelo, función de pérdida y optimizador
model = LeNet5().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=LR)

# Entrenamiento
for epoch in range(EPOCHS):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f"Train Epoch: {epoch} [{batch_idx*len(data)}/{len(train_loader.dataset)} "
                  f"({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item():.6f}")

# Evaluación
model.eval()
test_loss = 0
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)
        test_loss += criterion(output, target).item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()


print(f"({100. * correct / len(test_loader.dataset):.0f}%)\n")
```

```
Train Epoch: 0 [0/60000 (0%)]	Loss: 2.290354
Train Epoch: 0 [6400/60000 (11%)]	Loss: 2.315570
Train Epoch: 0 [12800/60000 (21%)]	Loss: 2.299078
Train Epoch: 0 [19200/60000 (32%)]	Loss: 2.298903
Train Epoch: 0 [25600/60000 (43%)]	Loss: 2.300027
Train Epoch: 0 [32000/60000 (53%)]	Loss: 2.312724
Train Epoch: 0 [38400/60000 (64%)]	Loss: 2.307347
Train Epoch: 0 [44800/60000 (75%)]	Loss: 2.314235
Train Epoch: 0 [51200/60000 (85%)]	Loss: 2.299103
Train Epoch: 0 [57600/60000 (96%)]	Loss: 2.297892
Train Epoch: 1 [0/60000 (0%)]	Loss: 2.300854
Train Epoch: 1 [6400/60000 (11%)]	Loss: 2.304880
Train Epoch: 1 [12800/60000 (21%)]	Loss: 2.295555
Train Epoch: 1 [19200/60000 (32%)]	Loss: 2.306973
Train Epoch: 1 [25600/60000 (43%)]	Loss: 2.288721
Train Epoch: 1 [32000/60000 (53%)]	Loss: 2.300674
Train Epoch: 1 [38400/60000 (64%)]	Loss: 2.300735
Train Epoch: 1 [44800/60000 (75%)]	Loss: 2.296987
Train Epoch: 1 [51200/60000 (85%)]	Loss: 2.297271
Train Epoch: 1 [57600/60000 (96%)]	Loss: 2.303181
Train Epoch: 2 [0/60000 (0%)]	Loss: 2.297223
Train Epoch: 2 [6400/60000 (11%)]	Loss: 2.307899
Train Epoch: 2 [12800/60000 (21%)]	Loss: 2.305202
Train Epoch: 2 [19200/60000 (32%)]	Loss: 2.299099
Train Epoch: 2 [25600/60000 (43%)]	Loss: 2.278854
Train Epoch: 2 [32000/60000 (53%)]	Loss: 2.305628
Train Epoch: 2 [38400/60000 (64%)]	Loss: 2.292988
Train Epoch: 2 [44800/60000 (75%)]	Loss: 2.284897
Train Epoch: 2 [51200/60000 (85%)]	Loss: 2.301391
Train Epoch: 2 [57600/60000 (96%)]	Loss: 2.292528
Train Epoch: 3 [0/60000 (0%)]	Loss: 2.300167
Train Epoch: 3 [6400/60000 (11%)]	Loss: 2.318235
Train Epoch: 3 [12800/60000 (21%)]	Loss: 2.282169
Train Epoch: 3 [19200/60000 (32%)]	Loss: 2.308214
Train Epoch: 3 [25600/60000 (43%)]	Loss: 2.293639
Train Epoch: 3 [32000/60000 (53%)]	Loss: 2.287900
Train Epoch: 3 [38400/60000 (64%)]	Loss: 2.289978
Train Epoch: 3 [44800/60000 (75%)]	Loss: 2.289389
Train Epoch: 3 [51200/60000 (85%)]	Loss: 2.282530
Train Epoch: 3 [57600/60000 (96%)]	Loss: 2.280149
Train Epoch: 4 [0/60000 (0%)]	Loss: 2.293365
Train Epoch: 4 [6400/60000 (11%)]	Loss: 2.291999
Train Epoch: 4 [12800/60000 (21%)]	Loss: 2.289127
Train Epoch: 4 [19200/60000 (32%)]	Loss: 2.285772
Train Epoch: 4 [25600/60000 (43%)]	Loss: 2.282268
Train Epoch: 4 [32000/60000 (53%)]	Loss: 2.272134
Train Epoch: 4 [38400/60000 (64%)]	Loss: 2.267790
Train Epoch: 4 [44800/60000 (75%)]	Loss: 2.270984
Train Epoch: 4 [51200/60000 (85%)]	Loss: 2.259552
Train Epoch: 4 [57600/60000 (96%)]	Loss: 2.279252
```

```
Train Epoch: 5 [0/60000 (0%)]	Loss: 2.265924
Train Epoch: 5 [6400/60000 (11%)]	Loss: 2.263482
Train Epoch: 5 [12800/60000 (21%)]	Loss: 2.271164
Train Epoch: 5 [19200/60000 (32%)]	Loss: 2.255219
Train Epoch: 5 [25600/60000 (43%)]	Loss: 2.227378
Train Epoch: 5 [32000/60000 (53%)]	Loss: 2.241599
Train Epoch: 5 [38400/60000 (64%)]	Loss: 2.201749
Train Epoch: 5 [44800/60000 (75%)]	Loss: 2.225660
Train Epoch: 5 [51200/60000 (85%)]	Loss: 2.214654
Train Epoch: 5 [57600/60000 (96%)]	Loss: 2.207725
Train Epoch: 6 [0/60000 (0%)]	Loss: 2.238866
Train Epoch: 6 [6400/60000 (11%)]	Loss: 2.173890
Train Epoch: 6 [12800/60000 (21%)]	Loss: 2.147619
Train Epoch: 6 [19200/60000 (32%)]	Loss: 2.129619
Train Epoch: 6 [25600/60000 (43%)]	Loss: 2.081799
Train Epoch: 6 [32000/60000 (53%)]	Loss: 2.104464
Train Epoch: 6 [38400/60000 (64%)]	Loss: 2.122292
Train Epoch: 6 [44800/60000 (75%)]	Loss: 2.079769
Train Epoch: 6 [51200/60000 (85%)]	Loss: 2.041947
Train Epoch: 6 [57600/60000 (96%)]	Loss: 1.935064
Train Epoch: 7 [0/60000 (0%)]	Loss: 1.980747
Train Epoch: 7 [6400/60000 (11%)]	Loss: 1.821163
Train Epoch: 7 [12800/60000 (21%)]	Loss: 1.860492
Train Epoch: 7 [19200/60000 (32%)]	Loss: 1.967731
Train Epoch: 7 [25600/60000 (43%)]	Loss: 1.738807
Train Epoch: 7 [32000/60000 (53%)]	Loss: 1.842051
Train Epoch: 7 [38400/60000 (64%)]	Loss: 1.723595
Train Epoch: 7 [44800/60000 (75%)]	Loss: 1.634346
Train Epoch: 7 [51200/60000 (85%)]	Loss: 1.704315
Train Epoch: 7 [57600/60000 (96%)]	Loss: 1.696681
Train Epoch: 8 [0/60000 (0%)]	Loss: 1.585786
Train Epoch: 8 [6400/60000 (11%)]	Loss: 1.460944
Train Epoch: 8 [12800/60000 (21%)]	Loss: 1.504637
Train Epoch: 8 [19200/60000 (32%)]	Loss: 1.538492
Train Epoch: 8 [25600/60000 (43%)]	Loss: 1.371103
Train Epoch: 8 [32000/60000 (53%)]	Loss: 1.488982
Train Epoch: 8 [38400/60000 (64%)]	Loss: 1.280876
Train Epoch: 8 [44800/60000 (75%)]	Loss: 1.339621
Train Epoch: 8 [51200/60000 (85%)]	Loss: 1.340497
Train Epoch: 8 [57600/60000 (96%)]	Loss: 1.295983
Train Epoch: 9 [0/60000 (0%)]	Loss: 1.223771
Train Epoch: 9 [6400/60000 (11%)]	Loss: 1.143815
Train Epoch: 9 [12800/60000 (21%)]	Loss: 1.285403
Train Epoch: 9 [19200/60000 (32%)]	Loss: 1.114302
Train Epoch: 9 [25600/60000 (43%)]	Loss: 1.140909
Train Epoch: 9 [32000/60000 (53%)]	Loss: 1.214866
Train Epoch: 9 [38400/60000 (64%)]	Loss: 1.105295
Train Epoch: 9 [44800/60000 (75%)]	Loss: 1.114371
Train Epoch: 9 [51200/60000 (85%)]	Loss: 0.957275
Train Epoch: 9 [57600/60000 (96%)]	Loss: 1.113677
```

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 1.016166
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.836244
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.917692
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.962960
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.914223
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.935577
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.927864
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.908553
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.985946
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.797681
Train Epoch: 11 [0/60000 (0%)]  Loss: 0.778887
Train Epoch: 11 [6400/60000 (11%)]  Loss: 0.921462
Train Epoch: 11 [12800/60000 (21%)] Loss: 0.739085
Train Epoch: 11 [19200/60000 (32%)] Loss: 0.915735
Train Epoch: 11 [25600/60000 (43%)] Loss: 0.749777
Train Epoch: 11 [32000/60000 (53%)] Loss: 0.927942
Train Epoch: 11 [38400/60000 (64%)] Loss: 0.868798
Train Epoch: 11 [44800/60000 (75%)] Loss: 0.705947
Train Epoch: 11 [51200/60000 (85%)] Loss: 0.867870
Train Epoch: 11 [57600/60000 (96%)] Loss: 0.877646
Train Epoch: 12 [0/60000 (0%)]  Loss: 0.821957
Train Epoch: 12 [6400/60000 (11%)]  Loss: 0.677227
Train Epoch: 12 [12800/60000 (21%)] Loss: 0.888611
Train Epoch: 12 [19200/60000 (32%)] Loss: 0.587170
Train Epoch: 12 [25600/60000 (43%)] Loss: 0.953355
Train Epoch: 12 [32000/60000 (53%)] Loss: 0.589124
Train Epoch: 12 [38400/60000 (64%)] Loss: 0.667878
Train Epoch: 12 [44800/60000 (75%)] Loss: 0.710181
Train Epoch: 12 [51200/60000 (85%)] Loss: 0.603574
Train Epoch: 12 [57600/60000 (96%)] Loss: 0.790573
Train Epoch: 13 [0/60000 (0%)]  Loss: 0.381692
Train Epoch: 13 [6400/60000 (11%)]  Loss: 0.691773
Train Epoch: 13 [12800/60000 (21%)] Loss: 0.644380
Train Epoch: 13 [19200/60000 (32%)] Loss: 0.597041
Train Epoch: 13 [25600/60000 (43%)] Loss: 0.514326
Train Epoch: 13 [32000/60000 (53%)] Loss: 0.674853
Train Epoch: 13 [38400/60000 (64%)] Loss: 0.462263
Train Epoch: 13 [44800/60000 (75%)] Loss: 0.619597
Train Epoch: 13 [51200/60000 (85%)] Loss: 0.621748
Train Epoch: 13 [57600/60000 (96%)] Loss: 0.470695
Train Epoch: 14 [0/60000 (0%)]  Loss: 0.660009
Train Epoch: 14 [6400/60000 (11%)]  Loss: 0.599284
Train Epoch: 14 [12800/60000 (21%)] Loss: 0.528506
Train Epoch: 14 [19200/60000 (32%)] Loss: 0.608190
Train Epoch: 14 [25600/60000 (43%)] Loss: 0.532200
Train Epoch: 14 [32000/60000 (53%)] Loss: 0.468184
Train Epoch: 14 [38400/60000 (64%)] Loss: 0.483214
Train Epoch: 14 [44800/60000 (75%)] Loss: 0.643663
Train Epoch: 14 [51200/60000 (85%)] Loss: 0.454628
Train Epoch: 14 [57600/60000 (96%)] Loss: 0.467120
```

(87%)

Ejercicio 2

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F

class AlexNet(nn.Module):
    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()

        # Capa convolucional 1
        self.conv1 = nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=2)
        self.relu1 = nn.ReLU(inplace=True)
        self.lrn1 = nn.LocalResponseNorm(5, alpha=1e-4, beta=0.75, k=2)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2)

        # Capa convolucional 2
        self.conv2 = nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2)
        self.relu2 = nn.ReLU(inplace=True)
        self.lrn2 = nn.LocalResponseNorm(5, alpha=1e-4, beta=0.75, k=2)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=2)

        # Capa convolucional 3
        self.conv3 = nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU(inplace=True)

        # Capa convolucional 4
        self.conv4 = nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1)
        self.relu4 = nn.ReLU(inplace=True)

        # Capa convolucional 5
        self.conv5 = nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1)
        self.relu5 = nn.ReLU(inplace=True)
        self.pool5 = nn.MaxPool2d(kernel_size=3, stride=2)

        # Capas completamente conectadas
        self.fc6 = nn.Linear(256 * 6 * 6, 4096)
        self.relu6 = nn.ReLU(inplace=True)
        self.dropout6 = nn.Dropout(0.5)

        self.fc7 = nn.Linear(4096, 4096)
        self.relu7 = nn.ReLU(inplace=True)
        self.dropout7 = nn.Dropout(0.5)

        self.fc8 = nn.Linear(4096, num_classes)
```

```
    def forward(self, x):
        x = self.pool1(self.lrn1(self.relu1(self.conv1(x))))
        x = self.pool2(self.lrn2(self.relu2(self.conv2(x))))
        x = self.relu3(self.conv3(x))
        x = self.relu4(self.conv4(x))
        x = self.pool5(self.relu5(self.conv5(x)))

        x = x.view(x.size(0), 256 * 6 * 6)

        x = self.dropout6(self.relu6(self.fc6(x)))
        x = self.dropout7(self.relu7(self.fc7(x)))

        x = self.fc8(x)

        return F.softmax(x, dim=1)

# Función para entrenar el modelo
def train_model(model, train_loader, criterion, optimizer, num_epochs=30):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader)}")

# Función para evaluar el modelo
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f"Accuracy on test set: {accuracy}%")

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

```
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

model = AlexNet(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

train_model(model, train_loader, criterion, optimizer, num_epochs=30)

evaluate_model(model, test_loader)
```

```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1/30, Loss: 2.302589490895381
Epoch 2/30, Loss: 2.3025704125309234
Epoch 3/30, Loss: 2.3025254858729176
Epoch 4/30, Loss: 2.302226912944823
Epoch 5/30, Loss: 2.25119497983352
Epoch 6/30, Loss: 2.213179388619445
Epoch 7/30, Loss: 2.1572559105465783
Epoch 8/30, Loss: 2.111590341533846
Epoch 9/30, Loss: 2.071355868635885
Epoch 10/30, Loss: 2.041391796163281
Epoch 11/30, Loss: 1.9850449238896675
Epoch 12/30, Loss: 1.9442216006996076
Epoch 13/30, Loss: 1.9191369117068513
Epoch 14/30, Loss: 1.886217120815726
Epoch 15/30, Loss: 1.8580890584479817
Epoch 16/30, Loss: 1.833253474948961
Epoch 17/30, Loss: 1.8073494653872517
Epoch 18/30, Loss: 1.7937313595696178
Epoch 19/30, Loss: 1.7717730893808252
Epoch 20/30, Loss: 1.7616859060114303
Epoch 21/30, Loss: 1.758428267048448
Epoch 22/30, Loss: 1.7453365409770585
Epoch 23/30, Loss: 1.732283755031693
Epoch 24/30, Loss: 1.710803150216027
Epoch 25/30, Loss: 1.705536770546223
Epoch 26/30, Loss: 1.6985302739740942
Epoch 27/30, Loss: 1.695783346967624
Epoch 28/30, Loss: 1.6854584367988665
Epoch 29/30, Loss: 1.678006345658656
Epoch 30/30, Loss: 1.6738292587077832
Accuracy on test set: 75.48%
```

a. ¿Cuál es la diferencia principal entre ambas arquitecturas?

- LaNet-5: es una arquitectura más simple, contando con 2 capas convolucionales y 3 capas completamente conectadas. Se utiliza la función de ctiación de tanh y existe un menor número de parámetros y menor profundidad que logra AlexNet.
- AlexNet: es una arquitectura más compleja, que posee 5 capas convolucionales y 3 capas completamente conectadas. Utiliza la función de activación ReLU e implementa funciones como dropout para evitar el sobreajuste. Permite mayor profundiad y tiene un mayor número de parámetros que LaNet-5.

b. Podría usarse LeNet-5 para un problema como el que resolvió usando AlexNet? ¿Y viceversa?

- Tecnicamente LeNet-5 si se podría usar para problemas que resolvio Alexnet. Sin embargo LeNet-5 tiene un menor capaciddad y profundidad haciendo que no funcione tan bien como Alexnet. Por otro lado, Alexnet si se podría usar para problemas que resolvio LeNet-5, pero al tener una mayor capacidad y profundidad. Esta mayor capacidad puede hace que se use un modelo más complejo para problrmas más sencillos, lo cual puede llevar a un uso innecesario de recursos.

c. Indique claramente qué le pareció más interesante de cada arquitectura

- Lo que más nos llamó la antención de LeNet-5 es la simplicidad del modelo. A pesar de ser considerablemente simple, logró obtener resultados bastante buenos. Por otro lado, lo que más nos llamó la atención de Alexnet es la complejidad del modelo. A pesar de ser un modelo complejo, no logró resultados tan buenos como LeNet-5. Esto nos hace pensar que la complejidad de un modelo no necesariamente se traduce en mejores resultados.

Investigue e indique en qué casos son útiles las siguientes arquitecturas, agregue imagenes si esto le ayuda a una mejor comprensión

a. GoogleNet (Inception)

- GoogleNet, también conocida como Inception, es una arquitectura de CNN desarrollada por Google. Se destacó por su profundidad y eficiencia en la utilización de los recursos.
- Es útil en casos donde se requieren redes profundas pero se desea mantener un uso eficiente de los recursos computacionales. GoogleNet utiliza una estructura llamada "módulos Inception" que combina múltiples tamaños de filtros de convolución en paralelo, permitiendo la extracción de características a diferentes escalas.
- Útil para tareas de clasificación de imágenes, detección de objetos y segmentación semántica.

b. DenseNet (Densely Connected Convolutional Networks)

- DenseNet es una arquitectura de CNN que se caracteriza por su densa conectividad entre capas. Cada capa está conectada directamente con todas las capas subsiguientes.
- Es útil en casos donde se desea un mejor flujo de información y gradientes más fuertes a lo largo de la red, lo que facilita el entrenamiento de redes profundas.
- Útil para tareas de clasificación de imágenes, detección de objetos y segmentación semántica.

c. MobileNet

- MobileNet es una arquitectura de CNN diseñada para aplicaciones en dispositivos móviles y embebidos con recursos computacionales limitados.
- Es útil en casos donde se necesita una red ligera y rápida, como en aplicaciones de visión por computadora en dispositivos móviles.
- Útil para tareas de clasificación de imágenes, detección de objetos en tiempo real y otras aplicaciones de visión en dispositivos móviles.

d. EfficientNet

- EfficientNet es una familia de arquitecturas de CNN que buscan optimizar el equilibrio entre el rendimiento y la eficiencia computacional mediante el uso de escalado compuesto.
- Es útil en casos donde se desean modelos con un buen rendimiento pero que sean escalables en términos de tamaño y requisitos computacionales.
- Útil para una variedad de tareas de visión por computadora, desde clasificación de imágenes hasta detección de objetos y segmentación semántica.

¿Cómo la arquitectura de transformers puede ser usada para image recognition?

La arquitectura de Transformers se puede usar en el reconocimiento de imágenes al tratar las imágenes como secuencias de parches y aplicar mecanismos de atención y transformación para capturar información espacial y contextual. Esto se puede hacer al obtener las características, el uso de atención multi-cabeza y agregar información. Los modelos de visión Transformer se entrenan en conjuntos de datos etiquetados, se ajustan finamente en tareas específicas y permiten la transferencia de aprendizaje.