AlexNet Challenge

Cecilia Gómez Castañeda
En conjunto: con Marcos Cortes

Este reporte contiene la descripción de los pasos realizados para preprocesar el dataset CIFAR 10 y con este hacer pruebas al modelo AlexNet tanto en Pytorch como en tensorflow, desarrollando el modelo desde scratch, finalmente se incluyen algunas conclusiones sobre el desempeño observado en ambos frameworks.
El desarrollo de los programas y pruebas se hicieron trabajando en línea con Marco Cortes, donde en conjunto adecuamos los modelos, y para tensor flow fue necesario que Marcos corriera las pruebas en su máquina dado que la capacidad de memoria excede la capacidad que ofrece Colab gratuitamente.

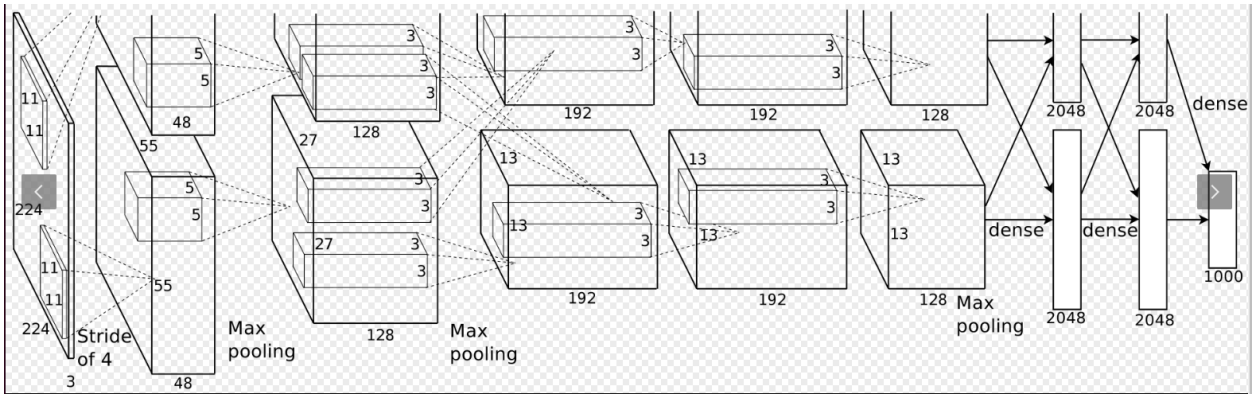
Explorando los datos

CIFAR-10 es un dataset de entrenamiento compuesto por imágenes que pertenecen a las siguientes 10 clases: `'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'`

Para el proceso del entrenamiento se disponen de 5o,ooo imágenes, mientras que para realizar las pruebas se considera un conjunto de 10.000 imágenes
Las imágenes de CIFAR-10 tienen un formato de 32 * 32 * PX, por lo que se requiere un preprocesa mentos para transformar esta imágenes al tamaña 224 * 224 que requiere el modelo AlexNet.

El modelo

El modelo Alex Net revisado en clase consta de 11 capas de procesamiento las cuales se ilustran en la siguiente imagen:



El modelo original está preparado para classificar entre 1000 clases, sin embargo para este ejercicio se modifica la ultima capa para adecuarlo a las 10 clases de CIFAR -10

Implementación en pytorch

Carga de dataset y Preprocesamiento en pytorch, se utilizó el dataset proporcionado por la librería torch visio. dataset, lo que facilitó los paso de tratamiento de datos, y las imágenes fueron procesadas por la función transform para lograr el tamaño de entrada requerido para Alex Net.

```
locBatchSize = 64
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))
]
 ) #normalize the data.

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

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

# order is important !
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

El modelo

El modelo fue obtenido del código disponible del modelo pre entrenado de pytorch. El resumen del modelo se obtuvo con la función summary de torch summary, se ilustra a continuación:

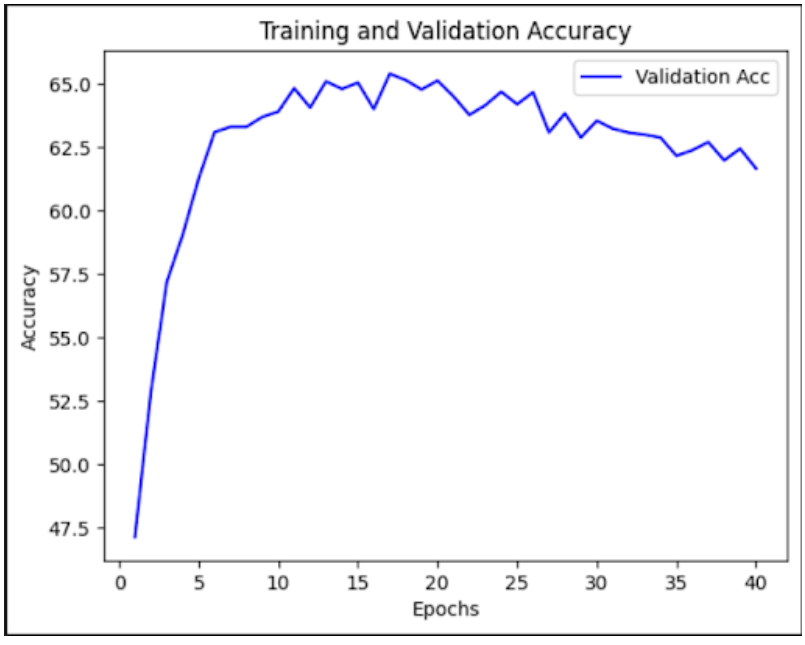
```
Total params: 57,044,810
Trainable params: 57,044,810
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 8.37
Params size (MB): 217.61
Estimated Total Size (MB): 226.55
----------------------------------------------------------------
```

Modelo en pytorch:

```python
class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 10, dropout: float = 0.5) -> None:
        super().__init__()
        #_log_api_usage_once(self)
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Resultados



El resultado evaluando el modelo en pytorch tiene su máxima accurecy aproximadamente en la época 17 llegando al 65% , sin embargo, después de esta época, como se muestra en la siguiente imagen, empieza a decaer

El tiempo de ejecución fue de aproximadamente 1 minuto por época.

Implementación en tensor flow

Carga de dataset y Preprocesamiento en pytorch, el dataset de COFAR-10 que se utilizó fue el que ofrece keras en su librerias de dataset, al igual que en pytroch fue necesario hacer una función de procesamiento de las imágenes, así como usar la función: to_categorical para poder tener los labels en un formato que se puede usar en el entrenamiento con la función de pérdida `loss='categorical_crossentropy'`

```
(train_images, train_labels), (test_images, test_labels) = keras.datasets.cifar10.load_data()
CLASS_NAMES= ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
train_labels = tf.keras.utils.to_categorical(train_labels, num_classes=10)
test_labels = tf.keras.utils.to_categorical(test_labels, num_classes=10)

def process_image(image,label):
    image=tf.image.per_image_standardization(image)
    image=tf.image.resize(image,(224,224))
    return image,label

train_ds=(train_ds
          .map(process_image)
          .shuffle(buffer_size=50000)
          .batch(batch_size=64,drop_remainder=True)
         )
test_ds=(test_ds
          .map(process_image)
          .shuffle(buffer_size=10000)
          .batch(batch_size=64,drop_remainder=True)
```

Finalmente, se usa la función from_tensor_slices para formar datasets para unir las imágenes preprocesadas con sus labels correspondientes y usarlos en el entrenamiento.

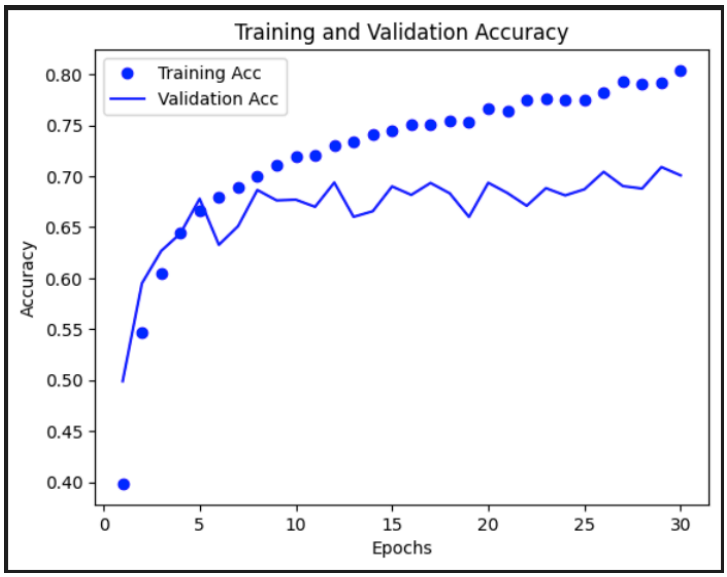
```
train_ds=tf.data.Dataset.from_tensor_slices((train_images,train_labels))
test_ds=tf.data.Dataset.from_tensor_slices((test_images,test_labels))
```

El modelo en tensorflow

A continuación se muestra el modelo implementado en tensor flow

```
# Definir el modelo AlexNet
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=11, strides=4, padding='same', activation='relu', input_shape=(224, 224, 3)),
    tf.keras.layers.MaxPooling2D(pool_size=3, strides=2),
    tf.keras.layers.Conv2D(192, kernel_size=5, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=3, strides=2),
    tf.keras.layers.Conv2D(384, kernel_size=3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(256, kernel_size=3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(256, kernel_size=3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=3, strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.summary()
```

El resultado de entrenamiento



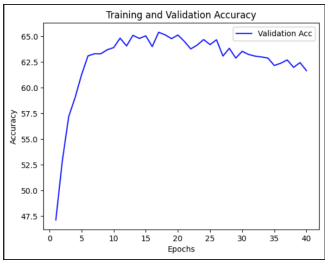
El resultado evaluando el modelo en tensor flow tiene su máxima accuracy en la época 30 llegando al 80% , sin embargo, lo cual muestra un mejor comportamiento respecto a pytorch.
El tiempo de ejecución fue similar, aproximadamente 1 minuto por época.

CONCLUSIONES

En este ejercicio utilizar el framework de python ofrece ventajas frente a tensor flow, primero porque se observó que el proceso de entrenamiento consumió significativamente más memoria en tensor flow, de hecho no fue posible realizar el entrenamiento en colab para tensor flow mientra que en pytorch aunque fue tardada la ejecución de entrenamiento fue posible finalizarlo. Otra ventaja en tensorflow fue que resultó más fácil realizar el preprocesamiento.

Por otro lado, tensorflow también mostró ventajas sobre pytorch , debido a que una vez compilado el modelo, realizar el entrenamiento en tensoflow resulta mucho más sencillo de implementar que pytorch. Finalmente, como se muestra en las imágenes siguientes, las pruebas mostraron mejor precisión en el modelo implementado en tensort flow que en pytorch por lo que pensamos es necesario hacer ajustes al modelo en este último para llegar a mejores resultados.

Pytorch



Tensor Flow