CLASIFICACIÓN IMÁGENES CON TRANSFORMERS

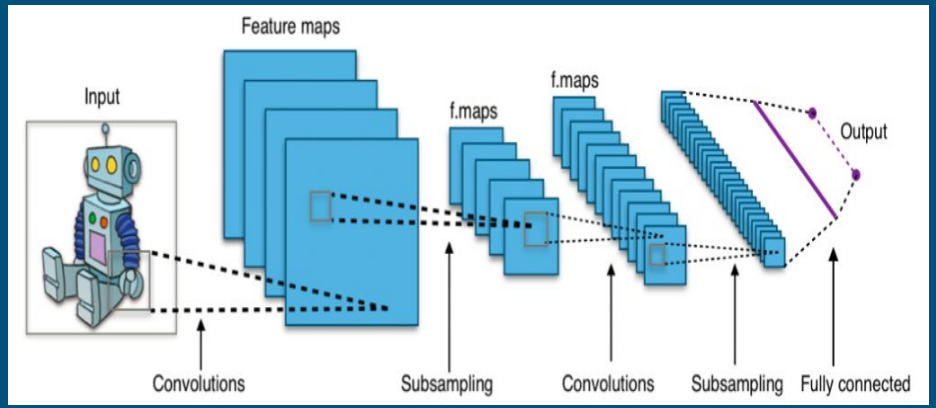
Grupo Transformer-6

ALBERTO GALLEGO HUERTA
JORGE SILLERO MANCHÓN

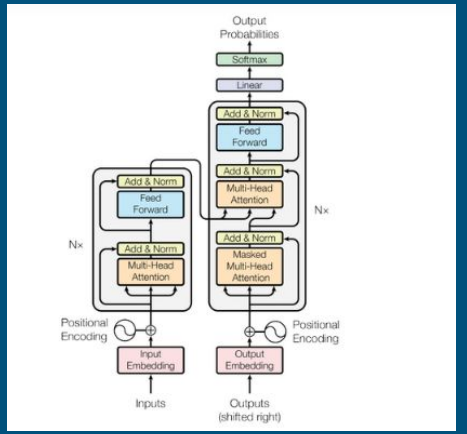
INTRODUCCIÓN

Para realizar la tarea hemos usado dos tipos de redes neuronales:

CONVOLUCIONALES



TRANSFORMER



CARGA DE DATOS

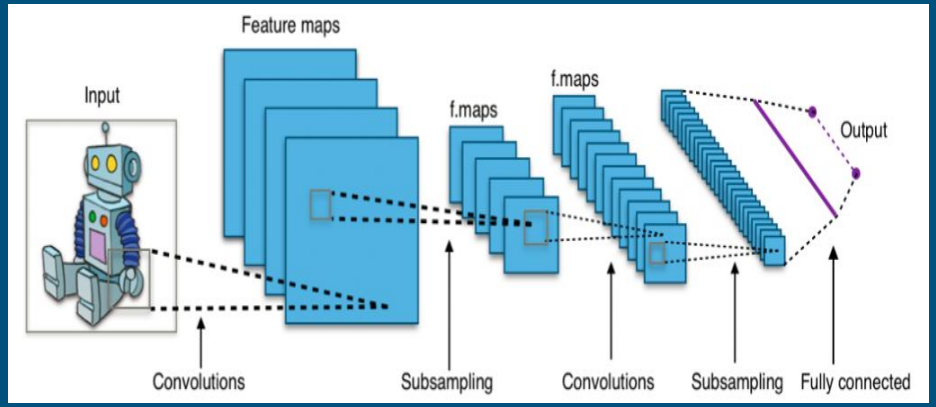
Hemos utilizado los *datasets* de *Pytorch*, concretamente la función *ImageFolder*.
También a cada una de las imágenes le hemos aplicado una transformación para redimensionarla a 64x64 y pasarla a Tensor.
Finalmente hemos dividido el conjunto de imágenes en conjunto de entreno y conjunto de test (usa un 10% de las imágenes). Mencionar también que hemos usado un tamaño de batch de 200.

RED CONVOLUCIONAL

Esta red es muy útil para la clasificación de imágenes.

Constan de varias capas las cuales van filtrando poco a poco la imagen(cogiendo trozos más pequeños) los cuales finalmente con una función de activación se clasifican y finalmente se unen.

Tienen el problema de que se necesita mucha memoria si el conjunto es muy grande y necesita un conjunto de entrenamiento amplio.



RED CONVOLUCIONAL

CAPA 1 Nuestra red está compuesta por dos capas convolucionales y una capa lineal de salida.

La primera capa recibe 3 canales de entrada (RGB)y su salida es 16

El tamaño del kernel es de 3x3 y el padding de 1, lo que hace que las dimensiones de la imagen se conservan

La función de activación de esta capa es una ReLU

CAPA 2

Recibe como entrada la salida de la anterior y su salida es de 32 neuronas. Mismo padding,kernel y función activación

Se aplica un Max Poolling, que nos divide la imagen a la mitad, tras esto aplanamos la imagen a una dimensión

CAPA SALIDA

Aplicamos la capa lineal que tiene como salida los 400 tipos de aves posibles

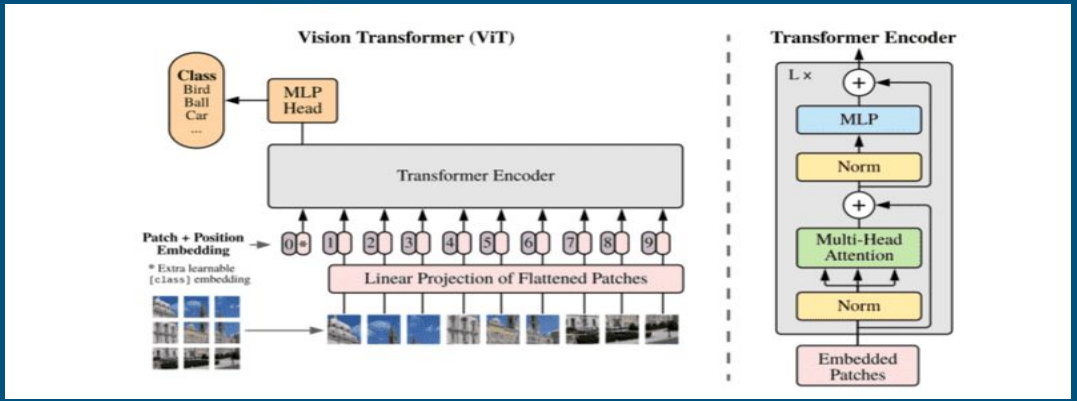
```
channel1 = 16
channel2 = 32

class NET(nn.Module):
    def __init__(self):
        super(NET, self).__init__()
        self.cnn = nn.Sequential(nn.Conv2d(in_channels=3, out_channels=channel1,kernel_size=3, padding=1),
                                 nn.ReLU(),
                                 nn.Conv2d(in_channels=channel1, out_channels=channel2,kernel_size=3, padding=1),
                                 nn.ReLU(),
                                 nn.MaxPool2d(2, 2),
                                 nn.Flatten(),
                                 nn.Linear(in_features=32*32*channel2, out_features=400))

    def forward(self, x):
        x = self.cnn(x)
        return x
```

RED TRANSFORMER

Es una arquitectura de redes neuronales que resuelve tareas de Seq2Seq.

La red se conforme de un **Encoder**, analiza el contexto de entrada, y un **Decoder**, genera la salida a partir de la salida del Encoder.

En nuestro caso usaremos la arquitectura Visual Transformer para clasificar imágenes (un solo Encoder).



1. Se divide la imagen en trozos más pequeños.
2. A cada trozo se le asocia un valor y una posición.
3. El encoder procesa los trozos
4. Se concatenan los resultados de mecanismo de auto-atención y el encoder
5. Se devuelve la clase

ESTRUCTURA CÓDIGO

```python
def pair(t):
    return t if isinstance(t, tuple) else (t, t)
```

Función auxiliar que crea un Pair

```python
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)
```

Normalizamos los datos antes de pasarlos a los bloques MLP y MHA. Consiste en una capa de normalización a la que se le aplica la función de activación que se pase como parámetro.

```python
class MLP(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.net(x)
```

Bloque MLP que consta de dos capas, lineales y una función GELU.
Dropout se usa para evitar el overfitting, y añade ceros de forma aleatoria con la probabilidad indicada como parámetro.

ESTRUCTURA CÓDIGO

```python
class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head *  heads
        project_out = not (heads == 1 and dim_head == dim)

        self.heads = heads
        self.scale = dim_head ** -0.5

        self.attend = nn.Softmax(dim = -1)
        self.dropout = nn.Dropout(dropout)

        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)

        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
        ) if project_out else nn.Identity()

    def forward(self, x):
        qkv = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h = self.heads), qkv)

        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale

        attn = self.attend(dots)
        attn = self.dropout(attn)

        out = torch.matmul(attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)
```

1. El bloque de Multi-Head Attention obtiene primero los valores de Q,K,V.
2. Posteriormente reordena los componentes de los tensores de cada uno de los vectores para que sean más “manejable”.
3. Una vez reordenados multiplica Q por K traspuesta, cambiando la primera y segunda fila por la última y penúltima, respectivamente.
4. Posteriormente al resultado de esto se le aplica softmax, al resultado de esto se le hace un dropout para evitar overfitting.
5. A continuación se multiplica el resultado obtenido tras el dropout por V y el resultado se vuelve a reordenar, para dejarlo como al principio.
6. Finalmente si project_out no es “false” aplica una transformación lineal con tamaño de entrada de la dimensión interna y con tamaño de salida la dimensión. Si esto no se cumple se usa la función identidad.

ESTRUCTURA CÓDIGO

```
class Encoder(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout = 0.):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Attention(dim, heads = heads, dim_head = dim_head, dropout = dropout)),
                PreNorm(dim, MLP(dim, mlp_dim, dropout = dropout))
            ]))
    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
        return x
```

El Encoder consta del bloque de Multi-Head Attention, y del bloque MLP, a ambos se le aplica la función de normalización(PreNorm), para poder trabajar con los datos.
La salida de MHA se suma con la entrada inicial(imagen sin normalizar)
El resultado anterior se suma a la salida del bloque MLP, esto es el resultado final del Encoder

ESTRUCTURA CÓDIGO

```python
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads, mlp_dim, pool = 'cls', channels = 3, dim_head = 64, dropout = 0., emb_dropout = 0.):
        super().__init__()
        #Obtiene la altura y el ancho de la imagen
        image_height, image_width = pair(image_size)
        #Obtiene la altura y el ancho del patch
        patch_height, patch_width = pair(patch_size)

        #Comprueba que la altura de la imagen sea divisible por la altura del patch y lo mismo con el ancho
        assert image_height % patch_height == 0 and image_width % patch_width == 0, 'Image dimensions must be divisible by the patch size.'

        #Obtiene el numero de veces en el que se ha dividido la imagen
        num_patches = (image_height // patch_height) * (image_width // patch_width)
        #Las dimensiones de cad subtrozo de imagen
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls token) or mean (mean pooling)'

        #Capa EMBEDDED PATCHES
        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height, p2 = patch_width),
            nn.Linear(patch_dim, dim),
        )

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)
        #ENCODER
        self.encoder = Encoder(dim, depth, heads, dim_head, mlp_dim, dropout)

        self.pool = pool
        self.to_latent = nn.Identity()

        #MLP HEAD
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        #EMBEDDED PATCHES
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape
        cls_tokens = repeat(self.cls_token, '1 n d -> b n d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)

        #Se aplica el ENCODER
        x = self.encoder(x)

        #Si se ha declarado que se use la media se usa, si no, se toma la primera columna
        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

        #Función de activación la identidad
        x = self.to_latent(x)
        return self.mlp_head(x)
```

Aquí tenemos el modelo, el cual primero realiza una división de la imagen en trozos según patch_size.

Una vez dividido se le pasa al Encoder, y se le aplica como función de activación(identidad en nuestro caso).

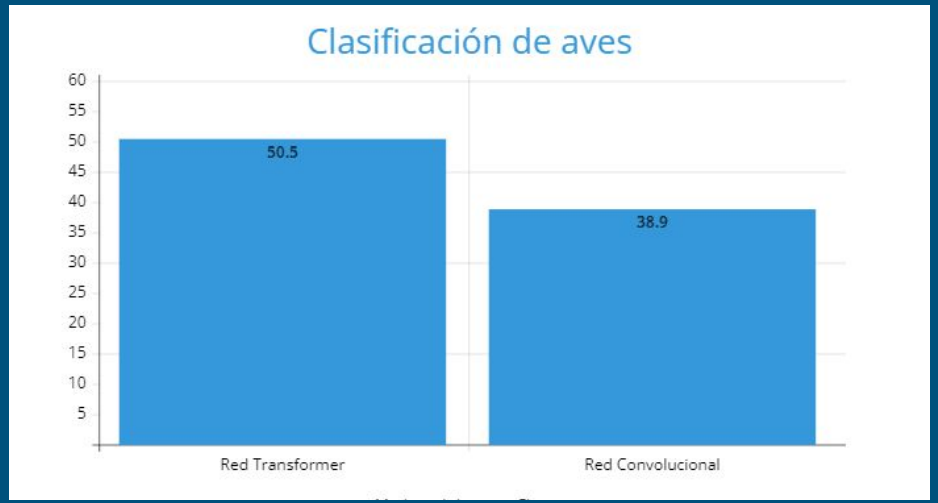
Por último como salida de la red tenemos la capa MLP Head, que consta normaliza la entrada que le llega y luego le aplica una capa lineal.

RESULTADOS

Los resultados que hemos obtenidos tras varias pruebas con cada una de las redes son los siguientes:

- Para la red convolucional, hemos obtenido una precisión del 38,9%.
- Para la red transformer, hemos obtenido una precisión del 50,5%.

Esto nos hace ver que la red transformer es la más indicada para la tarea de clasificación de imágenes, aunque ambas sirven y dan buenos resultados.



CONCLUSIONES

- Hemos comprendido el framework Pytorch
- Hemos aprendido a crear redes neuronales, entrenarlas y sacar resultados.
- Hemos comprendido cómo funcionan las redes convolucionales
- Hemos comprendido cómo funcionan las redes transformer.
- Hemos comparado la eficiencia de ambas redes y concluimos que las redes transformer son mejores para tareas de clasificación de imágenes.