TRABAJO INTELIGENCIA ARTIFICIAL: MEMORIA TRANSFORMER-6

Repositorio GitHub: https://github.com/jorsilman/TRANSFORMER-6.git

Jorge Sillero Manchón
Alberto Gallego Huerta

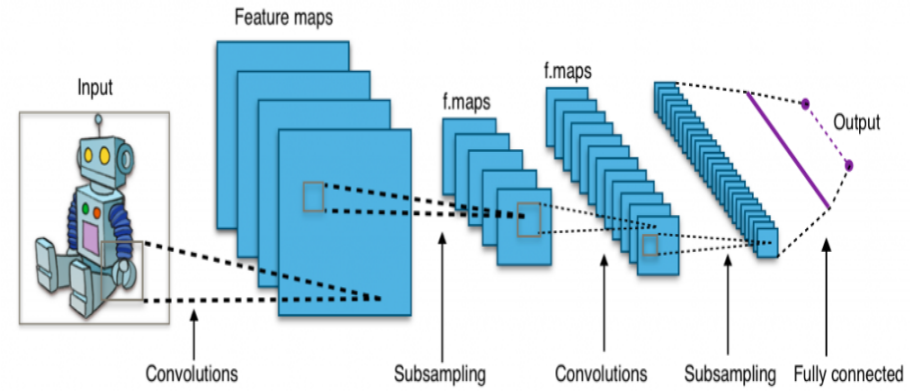
Introducción

En este trabajo hemos realizado la tarea de clasificación de imágenes usando redes neuronales, realizando previamente un entrenamiento de las mismas, para después clasificar las imágenes de test que teníamos asignadas. Para ello hemos usado dos tipos distintos de redes, una red transformer y una red convolucional. Ambas son redes que se usan con este propósito, las convolucionales se llevan usando desde hace bastante tiempo, pero las transformer están causando una revolución, ya que presentan mejores resultados, como veremos a continuación.

Descripción de las técnicas empleadas

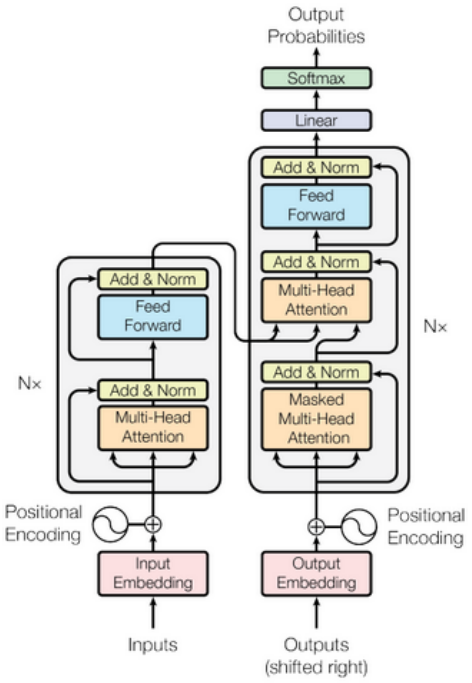
Red Convolucional

Una red convolucional es un tipo de red neuronal artificial que es muy efectiva para la realización de tareas de visión artificial, como en la clasificación y segmentación de imágenes entre otras aplicaciones.
Estas redes constan de varias capas de filtros convolucionales de una o más dimensiones. Después de cada capa, por lo general, se le aplica una función de activación. Al final de la red, se encuentran otras neuronas más sencillas para la clasificación final de las imágenes sobre las características extraídas.
Para que estas redes clasifiquen las imágenes correctamente es necesario un entrenamiento previo de la misma, con una cantidad importante de imágenes.



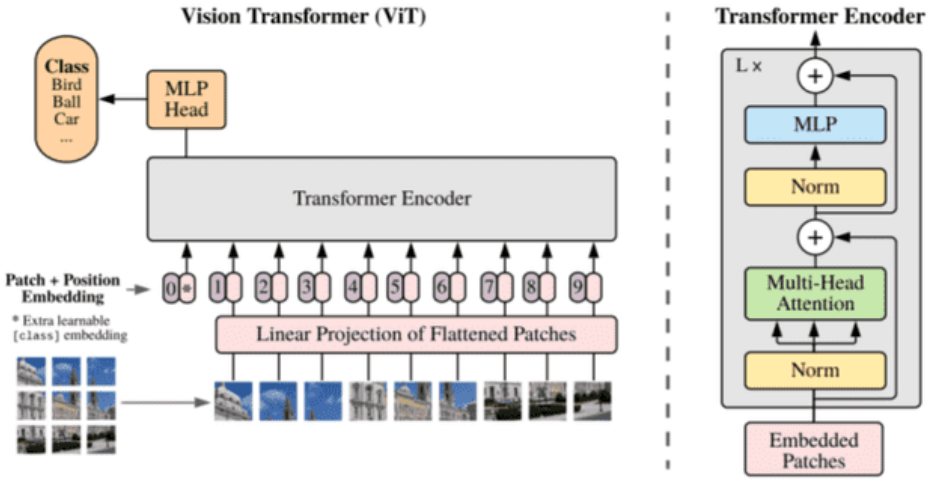
Red Transformer

Una red transformer es una arquitectura de redes neuronales, que resuelve tareas *Seq2Seq* sin los problemas de dependencias largas que presentan las LSTM(Memoria a corto plazo) o RNN(Redes Neuronales Recurrentes).



El esquema básico de la arquitectura transformer se basa en dos elementos principales, un *Encoder* y un *Decoder*. El *encoder* se encarga de analizar el contexto de la secuencia de entrada y el *decoder* es el encargado de generar la secuencia de salida a partir de este contexto.

En nuestro caso, usamos la arquitectura de Visual Transformer, que es la necesaria para la clasificación de imágenes. Esta consiste únicamente de un *encoder*.



La primera capa *Embedded Patches*, se encarga de dividir la imagen en trozos de imágenes más pequeños, para así asignarles un valor numérico y retener la información posicional. La secuencia resultante será la entrada para el *encoder*.

El encoder consiste en capas alternas de auto-atención por multicabezas y bloques MLP. La capa de normalización se aplica antes de cada bloque.

El algoritmo de auto-atención, permite al modelo integrar información de la imagen incluso a las capas más bajas. Es un mecanismo que permite al modelo saber con qué otro elemento de la secuencia está relacionada la información que está procesando. Este mecanismo toma tres valores de entrada:

- Q: se trata de la consulta (query) que representa el vector de entrada.
- K: las keys, que son todos los demás vectores de la secuencia.
- V: el valor vectorial de la información que se procesa en ese momento.

El algoritmo que nos devuelve la importancia de la información que se está procesando es el siguiente:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

La versión del algoritmo que usa transformer es una proyección de Q,K,V en h (cabezas) espacios lineales. Esto permite que cada cabeza se enfoque en diferentes características, para después poder concatenar los resultados. La arquitectura multi-cabeza hace posible aprender dependencias más complejas sin añadir tiempo de entrenamiento, ya que son operaciones que se ejecutan en paralelo.

El bloque MLP contiene dos capas con una función GELU no lineal.

Descripción de los modelos construidos

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

```
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
```

Nuestro modelo está compuesto por dos capas convolucionales y una capa lineal de salida. La primera capa convolucional recibe como canales de entrada 3, ya que la imagen es a color (RGB) y la salida es de 16. El *kernel size* es de 3x3 y el *padding* de 1, esto hace que las dimensiones de la imagen se conserven. La función de activación de esta capa es una *relu*.
Nuestra segunda capa convolucional recibe como valores de entrada 16, que es la salida de la capa anterior, y la salida es de 32 neuronas. El *kernel* y el *padding* son el mismo que en la capa anterior, al igual que la función de activación.
A continuación, aplicamos una capa de *Max Poolling*, que nos va a dividir la imagen de entrada a la mitad. Después de esto aplicamos la función *flatten*, que convierte la imagen de dos dimensiones en un array continuo de una dimensión.
Por último, una capa lineal, que tiene como salida los 400 tipos de clases de pájaros que disponemos.

Red Transformer

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

Esta función nos permite obtener una tupla dado un parámetro. Por ejemplo, le indicamos que el tamaño de la imagen es de 64 y la función nos devuelve 64x64.

```
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)
```

Esta función aplica la normalización de los datos, es la capa de normalización que va antes de los bloques MLP y Multi-Head Attention que hemos visto antes. Consiste en una capa de normalización a la que se le aplica la función de activación que se le indique.

```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)
```

Este es el bloque MLP, como he comentado antes, consta de dos capas, en este caso lineales y una función GELU. Las funciones dropout sirven para evitar el overfitting, añaden ceros de forma aleatoria con la probabilidad que le indiquemos.

```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)
```

Aquí muestro el bloque de Multi-Head Attention donde se puede observar que lo primero que hace es obtener los valores de q, k, v usando la función *to_qkv*, la cual aplica una transformación lineal que toma como tamaño de entrada el parámetro dimensión y como tamaño de salida la dimensión interna por 3. La función *chunk* divide el tensor resultante en las partes indicadas como parámetro.
Posteriormente reordena los componentes de los tensores de las componentes para que sean más “manejables”.
A continuación, multiplica Q por K traspuesta, cambiando la primera y la segunda fila por la última y la penúltima, respectivamente.

Después aplica la función *softmax* y tras esto *dropout*, para evitar el *overfitting*.
Finalmente multiplica el resultado por V y el resultado de esto vuelve a la ordenación inicial de los tensores.
Por último 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.

```
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
```

Este es el *encoder*, como se puede ver, consta del bloque de Multi-Head Attention, y del bloque MLP, a ambos se le aplica la función de normalización que hemos visto anteriormente. Según la profundidad que se le indique habrá más o menos capas.

```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)
```

```python
        #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)
```

Y por último, el modelo en sí, que como podemos observar, primero se realiza en *embedding*, según la dimensión del patch que se le haya indicado. Una vez hecho el *embedding* se llama al *encoder*, se le aplica la función de activación (en este caso es la identidad) y por último como salida de la red, la capa MLP Head, que consta de una capa lineal.

Entrenamiento de la red

Para el entrenamiento de la red hemos usado como función de pérdida o coste *CrossEntropyLoss*, y como optimizador Adam, con un *learning rate* de 1e-3.

Con respecto al test o validación, iteramos sobre el *dataloader* de test que habíamos creado, calculamos la predicción sobre cada imagen y la comparábamos con su label correspondiente para ver si coincidían o no.

Para la red convolucional hemos usado cinco épocas(epochs), ya que añadir más era innecesario, debido a que el porcentaje de precisión no aumentaba. Sin embargo, para la red transformer, hemos usado quince épocas, debido a que seguía aumentando la precisión en cada época.
No hemos querido añadir demasiadas épocas debido a que queríamos evitar el *overfitting*, es decir, que se especialice en el conjunto de entrenamiento y baje la precisión a la hora de hacer el test.

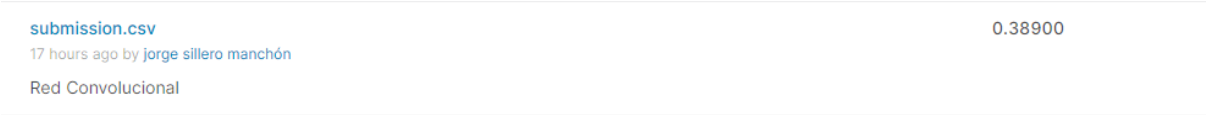
Creación del CSV

Para la creación del CSV, primero, guardamos el estado del modelo, a continuación, volvíamos a cargar el modelo para asegurarnos que el test se hacía con los datos de entrenamiento guardados.
Para el dataset del test, tuvimos que crear uno personalizado, ya que necesitábamos obtener el id de la imagen. Esto no nos dió muchos problemas, ya que una vez entendido cómo funcionaba fue sencillo. Cabe añadir que el dataloader usa un tamaño de batch de 1.
En cuanto a la creación del CSV en sí, primero añadimos la cabecera (Id, Category) y finalmente iteramos sobre las imágenes, calculando su predicción y añadiéndola al CSV junto con su id.

Resultados

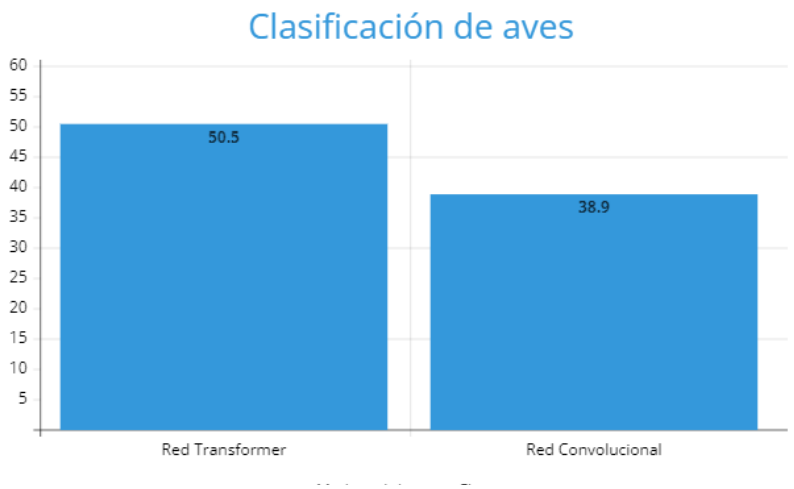
Los resultados que hemos obtenidos tras varias pruebas con cada una de las redes son los siguientes:

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



- Mientras que 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

En este trabajo hemos aprendido a hacer uso del framework de *pytorch*, creando redes neuronales, entrenandolas y clasificando imágenes.
Con respecto a las dos redes que hemos implementado, hemos llegado a la conclusión de que las redes convolucionales han sido una buena herramienta para la tarea de clasificación de imágenes durante muchos años, pero basándonos en los resultados que hemos obtenido, así como la información que hemos leído, podemos decir que los transformers mejoran bastantes las prestaciones de estas, por lo que el uso de las redes convolucionales se está viendo reducido.

Bibliografía

- https://github.com/JACantoral/DL_fundamentals
- https://pytorch.org/tutorials/beginner/basics/intro.html
- https://github.com/lucidrains/vit-pytorch/tree/main/vit_pytorch
- https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://arxiv.org/pdf/2010.11929.pdf
- 👀🔥🗣️ Modelos de secuencia. Aprende TODO (RNN, LSTM, GRU BERT) (themachinelearners.com)
- 🤖👨‍🎨 Transformer: domina el mundo (NLP): explicación SENCILLA (themachinelearners.com)
- https://es.wikipedia.org/wiki/Red_neuronal_convolucional