Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Vision Transformer

Procesamiento Avanzado de Imágenes

Alumno: Sebastián Tinoco
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Fecha de entrega: 19 de diciembre de 2022
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Códigos

1. Introducción

La visión computacional ocupa un pilar fundamental tanto a nivel industrial como en las cotidianidad de las personas. Desde aplicaciones como el procesamiento de imágenes médicas (por ejemplo, detección de tumores) hasta el reconocimiento facial como medida de seguridad, el impacto de esta área ha ido creciendo cada vez más a través del tiempo, a tal punto que resulta difícil imaginar un escenario donde no se disponga de esta tecnología.

Con tal de dar continuidad al aprendizaje y entendimiento de esta disciplina, el presente trabajo tiene por objetivo implementar la red neuronal **Vision Transformer** (ViT) con Pytorch, un framework deep learning para Python. De esta forma, se buscará entrenar esta red para resolver la tarea de clasificación de imágenes en el dataset Yoga-82.

Para cumplir esto de forma efectiva, se divide el trabajo en diferentes apartados. La sección 1 introduce de forma general el objetivo del informe y da una idea de lo que se espera en este. La sección 2 contiene la descripción general del modelo, dando especial énfasis en las capas de atención y las diferencias de este modelo con otros modelos de tipo transformer. La sección 3 contiene una breve caracterización del dataset Yoga-82. La sección 4 detalla la implementación del modelo en Pytorch y cada uno de sus componentes. La sección 5 presenta los resultados de entrenar el modelo con el set de datos. Finalmente, la sección 6 finaliza el documento con los principales hallazgos del trabajo realizado, denotando los principales aprendizajes y dificultades.

2. Marco Teórico

2.1. Transformers

Los modelos *transformers* son un tipo de arquitectura de red neuronal la cual tiene la particularidad de implementar el mecanismo de *self-attention*, lo que ha permitido lograr importantes avances en diferentes áreas relevantes a la ciencia de datos. Esta arquitectura y mecanismo fueron introducidos formalmente en la investigación "*Attention is all you need*" [1], marcando un antes y un después en el los modelos basados en aprendizaje supervisado. A continuación, se presenta la arquitectura general de un Transformer:

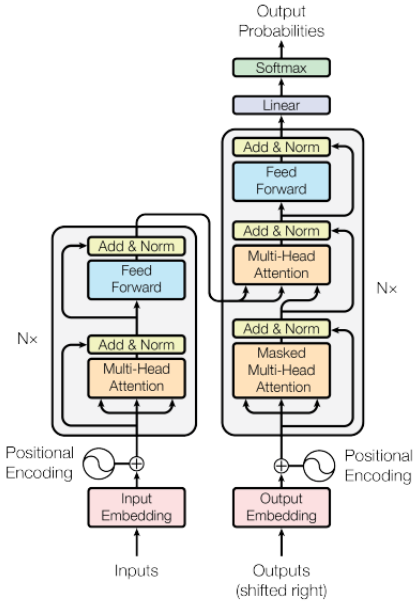

Figura 1: Arquitectura de Transformer

Para explicar esta arquitectura, es necesario distinguir sus dos componentes principales, *Encoder* y *Decoder*:

- *Encoder* (dibujo izquierdo): Su principal función es obtener una mejor representación de los datos de entrada, obteniendo como salida un arreglo de datos de la misma dimensión que la entrada. Para lograr lo anterior, se proyectan los datos de entrada a su representación en *embeddings*, se suma el encoding posicional, para luego aplicar N capas de *Multi-Head Attention* y *Feed Forward*, con una capa de *Layer Normalization* tras cada una de las operaciones anteriores y *skip connection* cada 2 capas.
- *Decoder* (dibujo derecho): De forma similar al *Encoder*, se utilizan embeddings posicionales, capas de *Multi-Head Attention*, *Layer Normalization* y *skip connection*, con la gran diferencia de que recibe los datos "ground truth" para aplicar el mecanismo de atención[1]. Luego, se combina esta información con la salida del *Encoder*, volviendo a aplicar el mecanismo de atención sobre

[1] Donde se hace de vital importancia diferenciar entre los datos disponibles y los datos futuros a través de *Masked Multi-Head Attention*

este conjunto de datos. Finalmente, se proyectan los datos al espacio de salida y se aplica *Softmax* para devolver las probabilidades de cada clase.

Por otro lado, el mecanismo de atención queda descrito por la siguiente ecuación:

$$Attention(Q, K, V) = \frac{QK^T}{\sqrt{d_k}} \tag{1}$$

Donde las matrices Q, K y V corresponden a proyecciones de los datos en un espacio de dimension d_k. De esta forma, *Multi-Head Attention* corresponde a aplicar la ecuación anterior M veces, concatenar estos resultados y luego aplicar volver a la dimensión original de los datos de entrada a través de una proyección lineal. La Figura 2 presenta la ilustración de este mecanismo:

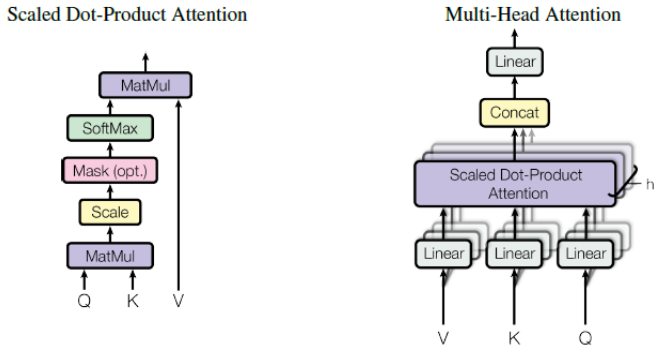

Figura 2: Mecanismo de Atención

De esta forma, se observa que el mecanismo de atención logra capturar el *contexto* de los datos a través del producto punto, lo que induce a resaltar los *tokens* con fuerte relación entre sí.

Para complementar la discusión y el entendimiento de *self-attention*, se presentan algunas y desventajas de este macanismo con respecto al operador convolución. Entre ellas, se encuentran:

- Ventajas de *self-attention*:
  - Logra capturar el "contexto" de cada input asignando pesos distintos a los datos dependiendo de la información de entrada
  - Es capaz de identificar la relación entre los datos (en este caso, entre los patches) de forma simultánea
- Desventajas de *self-attention*:
  - Es mas costosa de computar debido a su gran cantidad de parámetros, aumentando el tiempo de entrenamiento de manera significativa.
  - Requiere de una cantidad mucho mayor de datos para converger a buenos resultados

2.2. Vision Transformer

El modelo Vision Transformer es un modelo de red neuronal basado en la arquitectura Transformer y sus mecanismos de *self-attention*. Este modelo presentado formalmente en la investigación *An image*

is worth 16x16 words: Transformers for Image Recognition at scale [2] como una alternativa a las redes neuronales convolucionales para la clasificación de imágenes. A diferencia de los Transformers tradicionales, este modelo usa sólo el *Encoder* de la arquitectura, utilizando una capa *Feed-forward* para hacer la clasificación. La Figura 3 presenta la arquitectura de este modelo:

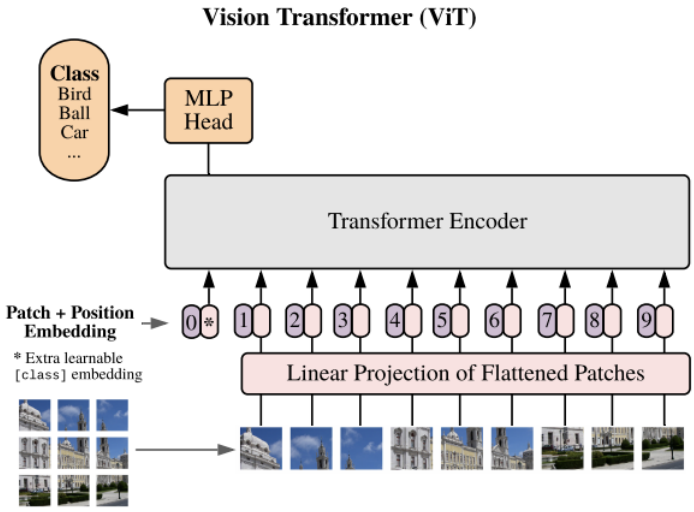

Figura 3: Arquitectura de ViT

Recordando la sección anterior, para hacer uso del mecanismo de atención es necesario contar con secuencias de entrada para así rescatar las relaciones importantes entre sus elementos. Para esto, el modelo ViT requiere transformar la imágen en *patches*, es decir, secciones de una imágen de un tamaño definido (usualmente $16x16$). Estas secciones son después proyectadas en una capa lineal, donde además se le suma el encoding posicional. Es importante añadir que en esta etapa también se añade el *token* de classificación, el cual se utiliza despues para realizar la clasificación. Habiendo realizado todo lo anterior, se transforma la información mediante múltiples capas de *Multi-Head Attention* (similar a lo explicado en el apartado de Transformers), para finalmente utilizar el *token* de clasificación como *input* para la capa lineal de clasificación.

Finalmente, es importante mencionar que el modelo ViT tiene dificultades para manejar imágenes con alta resolución debido a que el costo computacional y de memoria es cuadrático debido al mecanismo de atención, específicamente en la multiplicación QK^T de la ecuación 1. Para resolver esto, los autores redimensionan las imágenes a 224x224, permitiendo que los datos de entrada sean computacionalmente tratables en la gran mayoría de las máquinas virtuales a usar.

3. Datos

El dataset Yoga-82 fue presentado en la investigación "*Yoga-82: A New Dataset for Fine-grained Classification of Human Poses*" [3] con el propósito de contribuir al avance en la inferencia de posiciones humanas mediante procesamiento de imágenes. De esta forma, se publica este set de datos para ser utilizado como benchmark del ajuste de diferentes modelos a la clasificación de las diferentes

posiciones de yoga (en otras palabras, posiciones humanas). El dataset cuenta con 28.450 imágenes etiquetadas en 82 posiciones distintas de yoga, disponiendo desde 64 hasta 1.133 imágenes por cada clase. Una particularidad del dataset es que cuenta con imágenes etiquetadas en 3 niveles (jerarquías): Primer nivel (6 clases), Segundo nivel (20 clases) y Tercer nivel (82 clases). De esta manera, se buscará usar el modelo *Vision Transformer* para clasificar las imágenes en las 3 jerarquías nombradas.

Un primer problema encontrado fue que no todas las imágenes estaban disponibles para ser utilizadas, explicándose tanto en enlaces de descarga caídos o que las imágenes descargadas contenían información corrupta que no podía ser leida. De esta manera, del total de 28.450 imágenes originales postuladas en [3], sólo se pudieron utilizar 20.264. A modo de comparación, la Figura 4 presenta la distribución de las clases en el dataset haciendo la distinción entre el dataset original y las imágenes habilitadas para ser usadas:

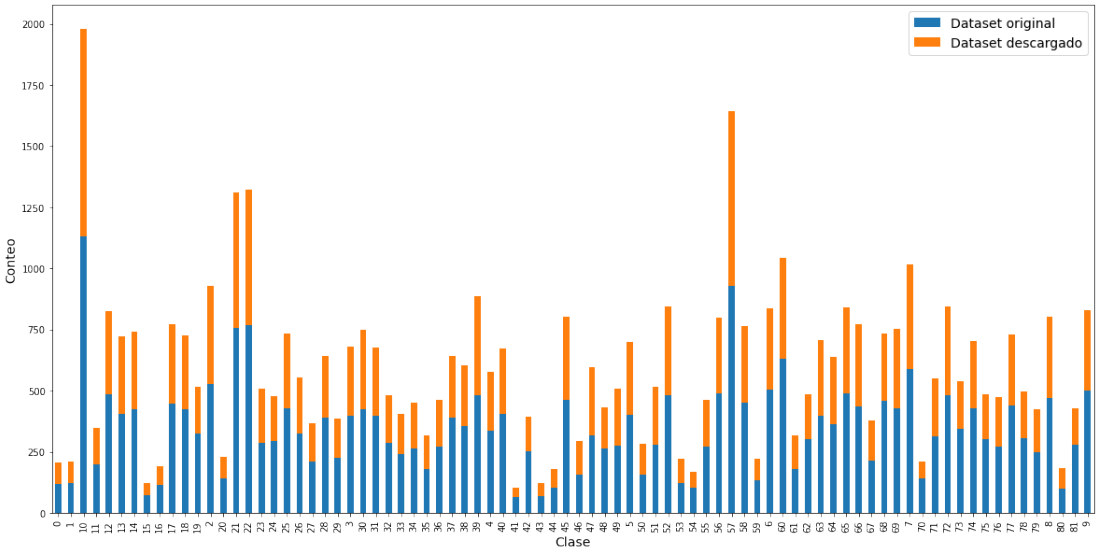

Figura 4: Distribución de clases de Yoga-82

De esta manera, se observa que la distribución de las clases se mantiene relativamente estable, por lo que no sería necesario incurrir en técnicas de rebalanceo de clases.

En cuanto a la distribución de conjuntos de entrenamiento, validación y test, se debió crear el conjunto de validación usando el 20 % de los datos de entrenamiento, manteniendo la distribución de clases vista en 4. De esta forma, el conjunto de entrenamiento quedó conformado por 12.094 observaciones (59.68 %), el conjunto de validación por 3.024 observaciones (14.92 %) y el conjunto de test por 5.146 observaciones (25.39 %).

4. Implementación

Para implementar el modelo ViT se trabajó de forma modular (es decir, dividiendo la implementación en diferentes módulos o clases) para facilitar la comprensión y el diseño de la arquitectura del transformer. En concreto, se diseñaron 5 módulos distintos:

- **gen_patches**: Función que recibe imágenes, devuelve las mismas imágenes segmentadas en

patches cuadrados (ver Anexo A.1).

- **MultiHeadSelfAttention**: Módulo que implementa el "Multi-Head Self Attention" de la arquitectura transformers (ver Anexo A.2).
- **MultiLayerPerceptron**: Módulo que implementa la capa de MLP referente al bloque de Encoder de ViT transformer (ver Anexo A.3).
- **EncoderBlock**: Módulo que implementa un bloque o "capa" del Encoder, referente al modelo ViT transformer. Se apoya de las clases *MultiHeadSelfAttention* y *MultiLayerPerceptron* (ver Anexo A.4)
- **ViT**: Módulo que implementa el modelo ViT transformer. Se apoya de la clase *EncoderBlock* y la función *gen_patches* (ver Anexo A.5).

La totalidad de este código puede ser encontrado en el siguiente repositorio de github.

5. Entrenamiento

Entrenar el modelo *Vision Transformer* con el dataset Yoga-82 no fue una tarea fácil, especialmente considerando que los recursos computacionales[2] disponibles eran limitados. Dado lo anterior, las principales dificultades encontradas en el entrenamiento del modelo fueron el tamaño del dataset, la gran cantidad de parámetros del modelo y la resolución de algunas imágenes, cayendo reiteradas veces en problemas de memoria RAM y/o de agotar el tiempo máximo[3] para utilizar GPU. Para solucionar lo anterior, se abordó el entrenamiento con tres enfoques:

- Se utilizó la arquitectura de *Vision Transformer* en su versión minimalista (**ViT-Ti/16**), la cual cuenta con 12 capas de Encoder, embeddings de dimensión 192, capas lineales MLP de tamaño 768 y 3 cabezas, reduciendo así la cantidad de parámetros. Además, trabaja sobre *patches* de 16x16 pixeles.
- Para resolver el problema del tiempo máximo de ejecución, se incurrió en el guardado automático de los pesos en el disco duro (en este caso, *Google Drive*) cada vez que se obtenía un *accuracy* mayor en el conjunto de validación. De esta manera cada vez que se agotaba el tiempo máximo de ejecución en una cuenta, se cargaba el *checkpoint* en otra cuenta con tiempo disponible y se seguía entrenando.
- Con el fin de obtener mayor eficiencia en el uso del tiempo disponible de GPU, se redimensionaron todas las imágenes del dataset a 256x256 antes del entrenamiento, logrando disminuir el tiempo de procesamiento por iteración a más de la mitad. De esta forma, la única transformación implementada por la librería *torchvision* fue el *crop* a 224x224, volviendo a las dimensiones propuestas por [2].

En cuanto a los hiperparámetros y técnicas de regularización utilizadas, se siguió la metodología expuesta en "*How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers*" [4], donde se utiliza un optimizador *Adam* con *learning rate* de 0.0002, $\beta_1 = 0.9$ y $\beta_2 = 0.999$.

[2] Se usó *Google Colab* para el entrenamiento de los modelos

[3] Google Colab no publica estos límites dado que puede variar en función de la intensidad de las operaciones. Para el caso del trabajo, Google Colab hacía disposición de casi 2 horas de GPU diarias por cuenta.

El tamaño de cada batch se fijó en 256 pues fue la potencia de 2 más grande que podía sostener la memoria RAM otorgada. En cuanto a técnicas de regularización, se utiliza *RandAugment* con $\alpha = 2$ para realizar *data augmentation*, *gradient clipping* de 5 como norma máxima del gradiente, *scheduler* lineal y dropout de 0.1 entre las capas lineales del *Encoder*. Finalmente, se utilizó la técnica *Early Stopping* con *patience* de 12 épocas para evitar el sobreajuste del modelo a los datos de entrenamiento, lo que tuvo como consecuencia que el modelo se entrenara entre 20 y 40 épocas por cada ejecución.

Con la finalidad de realizar experimentos con *transfer learning* entre modelos, se decide empezar entrenando con las clases del nivel superior debido a su menor número de clases (sólo 6) y así facilitar el ajuste del modelo. A continuación, se presentan las curvas de *loss* y *accuracy* para la primera y última ejecución[4] del modelo:

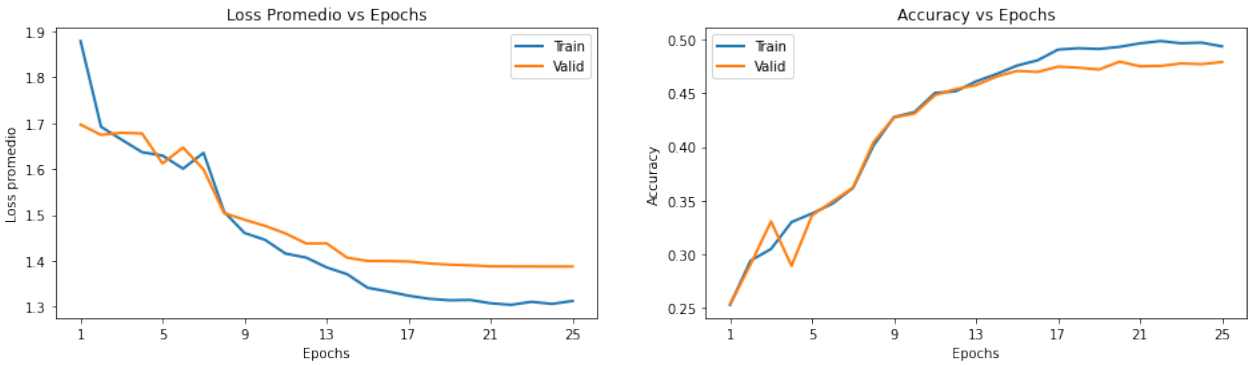

Figura 5: Entrenamiento del modelo en el nivel superior en las primeras 25 épocas

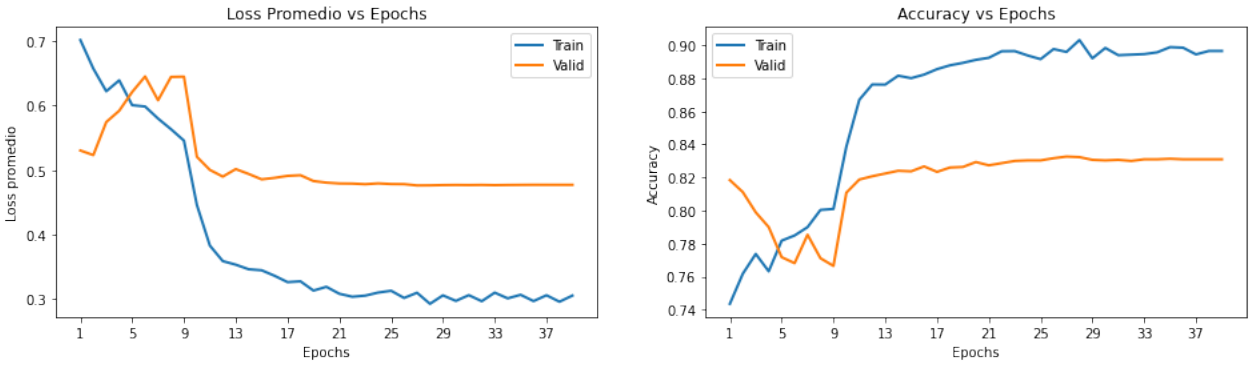

Figura 6: Entrenamiento del modelo en el nivel superior en las últimas 39 épocas

De la Figura 5, se observa que el modelo aprende rápidamente de los datos, posicionándose en el tramo de 40-50 % de *accuracy* en las primeras 10 épocas. Para el resto de las épocas, se observa como el modelo desacelera su aprendizaje, aprendiendo apenas pasar el 45 % de *accuracy*. Esto fenómeno es algo que se observó en todos los experimentos realizados y se puede explicar en el *scheduler* lineal usado, el cual reduce el *learning rate* del optimizador cada cierto número de iteraciones, explicando

[4] Por disponer de recursos computacionales limitados, se debió entrenar el modelo por medio de *checkpoints* y en diferentes ejecuciones

como el modelo casi no obtiene mejoras en su ajuste para las últimas épocas. Otro punto interesante es que debido a las técnicas de regularización aplicadas, el sobreajsute de los datos se mantiene en niveles mínimos, lo que se evidencia en la cercanía entre las curvas de entrenamiento y validación.

Por otro lado, se observa como en la Figura 6 el modelo encuentra dificultades para obtener mejoras en su rendimiento, logrando mejoras apenas notables en el conjunto de validación. Además, se observa como las curvas de *loss* y *accuracy* comienzan desde un punto con mayor ajuste a los datos, lo que es fruto de haber entrenado este modelo bajo el enfoque de *checkpoints*. Por último, se estima[5] que el modelo se entrenó con 120 épocas para alcanzar el ajuste observado en esta Figura.

Con la red entrenada, se procede a evaluar su desempeño en los conjuntos de validación y test, usando el modelo con mejor ajuste en validación conseguido durante la fase de entrenamiento. De esta manera, se consigue un 83.27 % de *accuracy* en el conjunto de validación, en contraste con el 65.66 % de *accuracy* en el conjunto de test.

Se repite el experimento con las clases del nivel intermedio (20 clases en total). A diferencia del experimento anterior (y considerando el tiempo y recursos limitados para implementar la totalidad del proyecto) el modelo sólo se entrena una ejecución, alcanzando las 37 épocas efectuadas sin hacer uso de *checkpoints*. La Figura 7 presenta las curvas de *loss* y *accuracy* de este experimento:

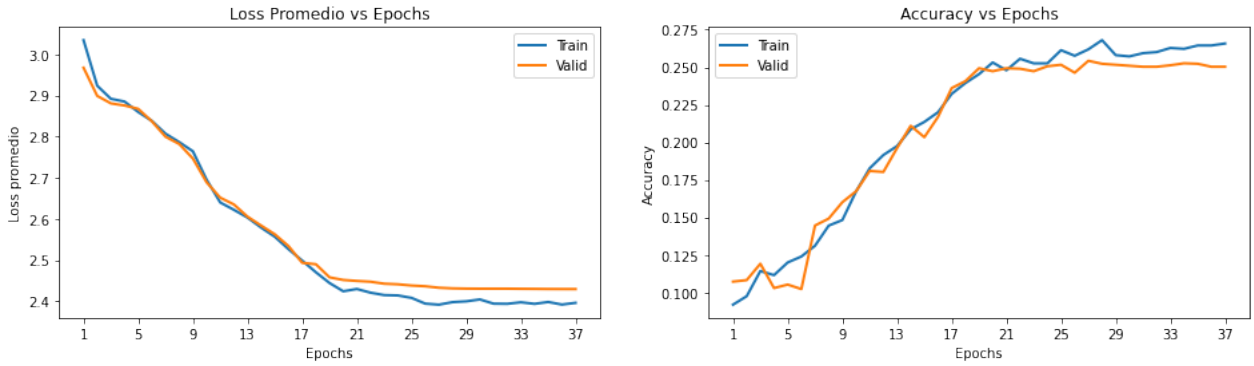

Figura 7: Entrenamiento del modelo en el nivel intermedio

Al igual que en el experimento anterior, se observa como el modelo posee poco sobreajuste, lo que se ilustra en la cercanía de las curvas de entrenamiento y validación. Además, se observa como el mayor número de clases impacta de manera negativa en el *accuracy* del modelo, el cual se estabiliza en el 25-27 % en las últimas épocas del entrenamiento. Esto último se puede explicar en que un mayor número de clases aumenta la complejidad del problema, haciendo que sea mas difícil para el modelo distinguir entre una clase y otra.

Con el motivo de entender mejor el funcionamiento de la red *Vision Transformer* y su ajuste a los datos disponibles, se utiliza *transfer learning* del modelo entrenado con las clases de nivel superior con la intención de lograr un ajuste mayor a los datos de las clases del intermedio. En concreto, se hace uso de la técnica *fine tunning*, es decir, entrenar empezando desde los pesos pre entrenados, con la salvedad de cambiar la dimensión de salida de la red (en este caso, pasar de predecir 6 a 20 clases).

[5] Se perdió el número exacto de épocas efectuadas resultado de aplicar *Early Stopping* y el enfoque de *checkpoints*

La Figura 8 presenta los resultados de haber realizado este experimento:

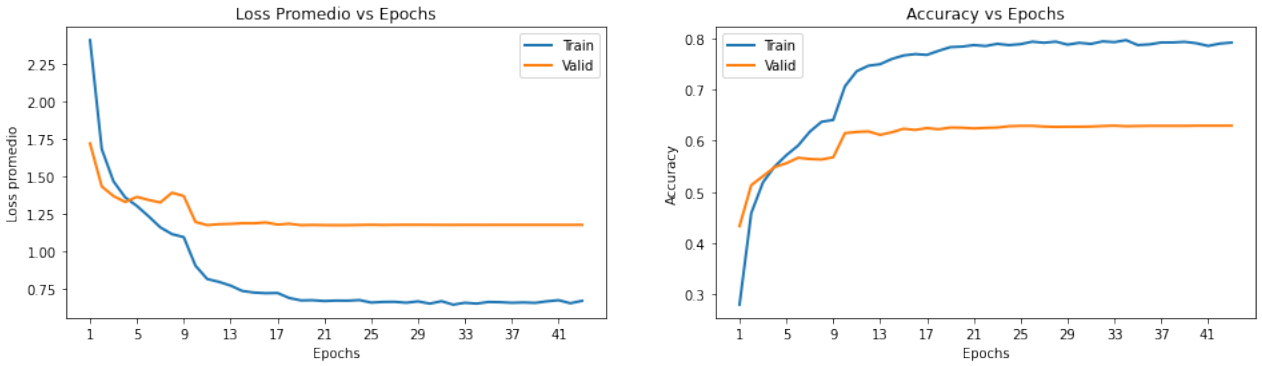

Figura 8: Entrenamiento del modelo en el nivel intermedio con *Fine Tunning*

A partir de los resultados, se observa como la red obtiene un mejor desempeño de forma inmediata, alcanzando el 60 % de *accuracy* en el conjunto de validación en las primeras 10 épocas. Esta diferencia con el entrenamiento "desde cero" es notable, pues es prueba de que el *transfer learning* es una técnica efectiva para alcanzar un buen ajuste en un menor número de épocas. Finalmente, se evalua el ajuste del algoritmo en los datos de validación y test, consiguiendo un *accuracy* de 62.96 % en los datos de validación y 53.46 % en los datos de test.

La última ola de experimentos fue intentar clasificar las posiciones de yoga de las clases del nivel inferior (82 clases en total). Similar al experimento anterior de 20 clases, el modelo se entrena sólo una ejecución, alcanzando las 34 épocas efectuadas sin hacer uso de *checkpoints*. La Figura 9 presenta las curvas de *loss* y *accuracy* de este experimento:

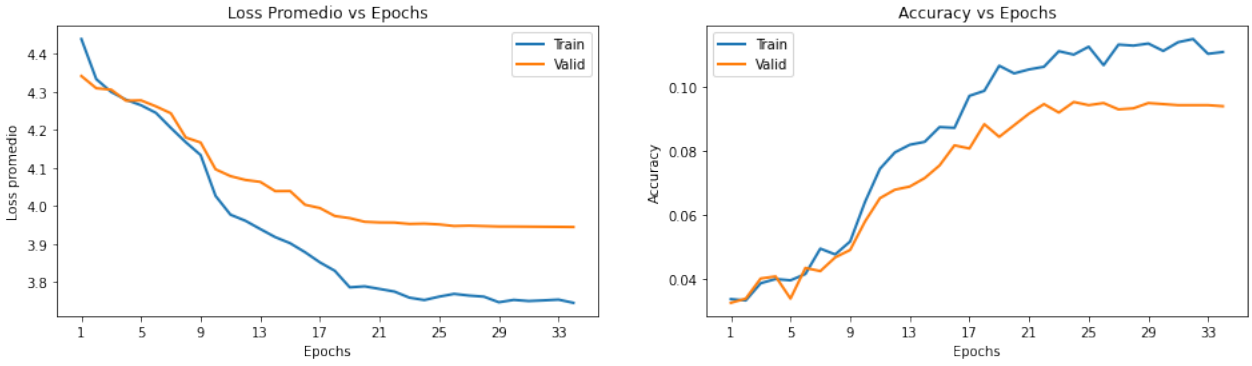

Figura 9: Entrenamiento del modelo en el nivel inferior

Similar a los dos experimentos anteriores donde el modelo se entrena "desde cero", se observa como los niveles de sobreajuste se mantienen en el mínimo. Además, se observa nuevamente como un mayor número de clases impacta de forma negativa sobre el *accuracy* del modelo, el cual se estabiliza entre 9-10 % en el conjunto de validación en las últimas épocas de entrenamiento.

Por otro lado, se repite el experimento de usar los pesos del modelo entrenado con las clases del nivel superior para hacer *transfer learning* (en específico, *fine tunning*) y asi obtener un mayor rendimiento en la predicción de las 82 clases. La Figura 10 presenta los resultados de este experimento:

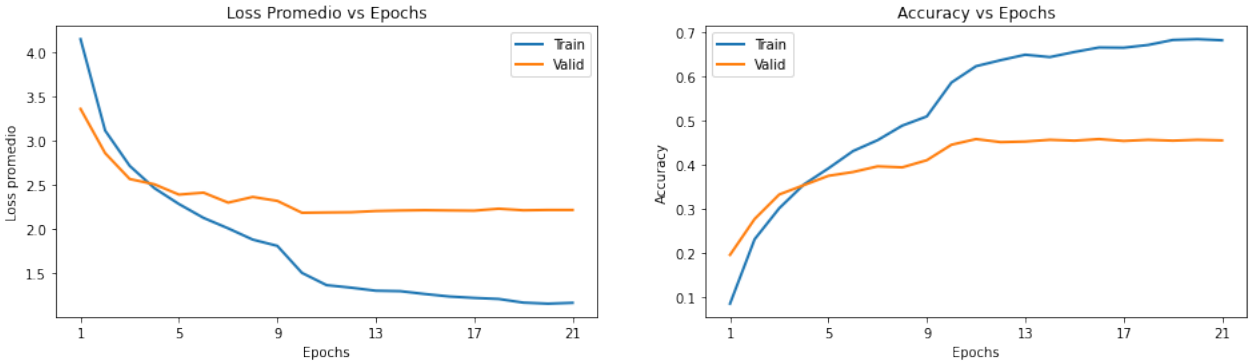

Figura 10: Entrenamiento del modelo en el nivel inferior con *Fine Tunning*

Al igual que en el caso anterior, se observa como la red obtiene un mejor desempeño de forma inmediata, posicionando su *accuracy* sobre el 40 % en las primeras 13 épocas. Es interesante notar también que el aprendizaje de la red se estanca rápidamente, fenómeno que puede ser explicado en la red habiendo extraído todo el conocimiento "útil" de las clases superiores en las primeras épocas, para luego verse en la obligación de aprender de los datos para seguir mejorando su ajuste.

La Tabla 1 resume los experimentos realizados a través de esta sección:

Tabla 1: Accuracy en Entrenamiento y Validación por jerarquía de clases

Clases	**Transfer Learning**	**Validación**	**Test**
6	No	83.27 %	65.66 %
20	Sí	62.96 %	53.46 %
82	Sí	45.77 %	42.54 %
20	No	25.43 %	25.77 %
82	No	9.52 %	9.09 %

6. Conclusiones

La visión computacional tiene un rol fundamental en el funcionamiento de la sociedad, tanto en la cotidianidad de las personas como a nivel industrial. En el trabajo expuesto fue posible dilucidar el diseño y entrenamiento de la red neuronal *Vision Transformer* para resolver el problema de clasificación de imágenes del dataset Yoga-82 por medio de Pytorch. Además, se estudió el impacto de realizar *transfer learning* sobre el rendimiento en datos dentro y fuera de la muestra.

En la sección 2 se da una explicación del funcionamiento del modelo *Vision Transformer*, haciendo un contraste del mecanismo de atención con la operación de convolución y explicando también el problema con la dimensión de los *patches* de entrada. En la sección 3 se da una explicación breve de la composición del dataset Yoga-82. En la sección 4 se explican los principales módulos programados para implementar el modelo *Vision Transformer* en *Pytorch*. En la sección 5 se muestran los resultados de haber entrenado el modelo con el dataset Yoga-82.

En vista y función de los resultados obtenidos a lo largo de este informe, se observa como el modelo *Vision Transformer* obtiene mejores predicciones al utilizar *transfer learning* entre modelos y jerarquías. Este hallazgo va acorde con lo esperado y hace sentido con la teoría, pues utilizar redes pre entrenadas es una forma ampliamente usada para aumentar el rendimiento de los modelos, en especial si los pesos provienen de una red entrenada con la misma data. Además, se observa como la red disminuye su ajuste a medida que aumenta el número de clases a predecir. Este resultado también va acorde a lo esperado, pues un mayor número de clases aumenta la complejidad del problema, siendo mas difícil para el modelo diferenciar entre las clases y hacer predicciones precisas.

En consideración del ajuste de la red *Vision Transformer* a los datos en donde se logra predecir de forma parcial las distintas jerarquías de clases, se concluye la obtención de resultados aceptables en el entrenamiento del modelo.

A pesar de haber implementado con éxito el modelo, el trabajo expuesto no estuvo exento de dificultades. La principal dificultad estuvo en lidiar con los recursos computacionales limitados para entrenar el modelo, donde se tuvo que recurrir al entrenamiento fragmentado por medio de *checkpoints*. Otra complejidad encontrada fue entender la compleja arquitectura y mecanismos subyacentes del modelo para su implementación.

En retrospectiva del trabajo realizado, se consolida el aprendizaje de el diseño y entrenamiento de la red *Vision Transformer* mediante código Python y la librería Pytorch.

Finalmente, queda como trabajo propuesto entrenar el modelo con recursos suficientes para aumentar la cantidad de parámetros del modelo y prescindir del enfoque de *checkpoints*. Se espera que con la inclusión de estos cambios se disponga de un mayor tiempo de entrenamiento y así conseguir un mejor ajuste a los datos.

Referencias

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, $\mathcal{L}$., y Polosukhin, I., "Attention is all you need," Advances in neural information processing systems, vol. 30, 2017.

[2] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," arXiv preprint arXiv:2010.11929, 2020.

[3] Verma, M., Kumawat, S., Nakashima, Y., y Raman, S., "Yoga-82: a new dataset for fine-grained classification of human poses," en Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops, pp. 1038–1039, 2020.

[4] Steiner, A., Kolesnikov, A., Zhai, X., Wightman, R., Uszkoreit, J., y Beyer, L., "How to train your vit? data, augmentation, and regularization in vision transformers," arXiv preprint arXiv:2106.10270, 2021.

Anexo A. Código implementado

A.1. Pregunta 1

Código A.1: Generar patches

```python
import torch

def gen_patches(images, patch_size: int = 16):

  '''
  Función que recibe imágenes, devuelve las mismas imágenes segmentadas en patches cuadrados.
  images: iterable con imágenes
  patch_size: tamaño de cada patch (int)
  '''

  batch_size, C, H, W = images.shape # dimensiones de la imagen

  assert H == W # assert imagenes cuadradas
  assert H % patch_size == 0 # assert imagenes divisibles por patch_size

  n_patches = H // patch_size # cantidad de patches en una arista

  output = [] # lista de imagenes en formato batches
  for img in images:
    patches = [] # batches de una imagen
    for row in range(n_patches):
      for col in range(n_patches):
        patches.append(img[:, patch_size * row : patch_size * (row + 1), patch_size * (col) : patch_size * (col + 1)].flatten())
      ↪ # tensor (1, patch_size)

    output.append(torch.stack(patches))

  output = torch.stack(output)
  return output
```

Código A.2: Multi-Head Self Attention

```python
import numpy as np
import torch
import torch.nn as nn

class MultiHeadSelfAttention(nn.Module):
  def __init__(self, linear_dim: int, heads: int):

    '''
    Clase que implementa el "Multi-Head Self Attention" de la arquitectura transformers.
    linear_dim: dimensión lineal de los datos de entrada (int)
    heads: número de "heads" por los que hacer "self-attention" en paralelo (int)
    '''

    super(MultiHeadSelfAttention, self).__init__()

    self.linear_dim = linear_dim
    self.heads = heads

    self.W_q = nn.ModuleList([nn.Linear(linear_dim, linear_dim) for _ in range(heads)]) # matriz de pesos Q
    self.W_k = nn.ModuleList([nn.Linear(linear_dim, linear_dim) for _ in range(heads)]) # matriz de pesos K
```

```
    self.W_v = nn.ModuleList([nn.Linear(linear_dim, linear_dim) for _ in range(heads)]) # matriz de pesos V

    self.W_o = nn.Linear(linear_dim * heads, linear_dim) # h x d_v

  def forward(self, x):

    '''
    Función que recibe un tensor de entrada, computa "Multi-Head Self Attention" sobre este.
    Devuelve un tensor con las mismas dimensiones.
    x: tensor a transformar (torch.tensor)
    '''

    output = [] # lista de resultados por head
    for head in range(self.heads):

      Q = self.W_q[head](x) # Proyección de x: Q
      K = self.W_k[head](x) # Proyección de x: K
      V = self.W_v[head](x) # Proyección de x: V

      Z = (Q @ K.transpose(1, 2)) / np.sqrt(self.linear_dim)
      Z = torch.softmax(Z, dim = 1) # softmax
      Z = Z @ V # seq x d_v
      output.append(Z) # append de resultados

    output = torch.cat(output, dim = 2) # concatenamos resultados
    output = self.W_o(output) # capa lineal para volver a dimension original

    return output
```

Código A.3: Multi Layer Perceptron

```

import torch.nn as nn
import torch.nn.functional as F

class MultiLayerPerceptron(nn.Module):
  def __init__(self, embedding_dim: int, mlp_dim: int, dropout: float, dropout_2: float = 0.0):

    '''
    Clase que implementa la capa de MLP referente al bloque de Encoder de ViT transformer.
    embedding_dim: Dimensión del embedding de los datos de entrada (int)
    mlp_dim: Dimensión de la capa lineal MLP para transformar los datos (int)
    dropout: Dropout a aplicar entre ambas capas lineales (float)
    dropout_2: Dropout a aplicar después de la última capa lineal (float)
    '''

    super(MultiLayerPerceptron, self).__init__()

    self.fc1 = nn.Linear(embedding_dim, mlp_dim)
    self.fc2 = nn.Linear(mlp_dim, embedding_dim)
    self.dropout = nn.Dropout(dropout)
    self.dropout_2 = nn.Dropout(dropout_2)

  def forward(self, x):

    '''
    Función que recibe un tensor de entrada, la transforma a través de capas lineales "Feed-forward".
    x: tensor a transformar (torch.tensor)
    '''

```

```
    x = self.fc1(x) # Capa 1
    x = F.gelu(x) # GELU
    x = self.dropout(x) # Dropout
    x = self.fc2(x) # Capa 2
    x = self.dropout_2(x) # Dropout

    return x
```

Código A.4: Bloque de Encoder

```

from model.MultiHeadSelfAttention import MultiHeadSelfAttention
from model.MultiLayerPerceptron import MultiLayerPerceptron
import torch.nn as nn

class EncoderBlock(nn.Module):
  def __init__(self, embedding_dim: int, mlp_dim: int, heads: int, dropout: float, dropout_2: float):

    '''
    Clase que implementa un bloque o "capa" del Encoder, referente al modelo ViT transformer.
    Se apoya de las clases MultiHeadSelfAttention y MultiLayerPerceptron.
    embedding_dim: dimensión del embedding de los datos de entrada (int)
    mlp_dim: Dimensión de la capa lineal MLP para transformar los datos (int)
    heads: número de "heads" por los que hacer "self-attention" en paralelo (int)
    dropout: Dropout a aplicar entre ambas capas lineales (float)
    dropout_2: Dropout a aplicar después de la última capa lineal (float)
    '''

    super(EncoderBlock, self).__init__()

    self.attention = MultiHeadSelfAttention(linear_dim = embedding_dim, heads = heads)
    self.mlp = MultiLayerPerceptron(embedding_dim = embedding_dim, mlp_dim = mlp_dim, dropout = dropout,
      ↪ dropout_2 = dropout_2)
    self.ln1 = nn.LayerNorm(embedding_dim)
    self.ln2 = nn.LayerNorm(embedding_dim)

  def forward(self, x):

    '''
    Función que recibe un tensor de entrada, computa "Multi-Head Self Attention" y lo transforma con capas lineales.
    x: tensor a transformar (torch.tensor)
    '''

    res = x # skip connection
    output = self.ln1(x) # LN
    output = self.attention(output) + res # MHSA + skip connection

    res = output # skip connection
    output = self.ln2(output) # LN
    output = self.mlp(output) + res # MLP + skip connection

    return output
```

Código A.5: Modelo ViT

```

from model.EncoderBlock import EncoderBlock
from model.gen_patches import gen_patches
import torch
import torch.nn as nn
```

```python

class ViT(nn.Module):
  def __init__(self, img_size: int, patch_size: int, embedding_dim: int, mlp_dim: int, n_blocks: int, heads: int,
               n_classes: int, n_channels: int = 3, dropout: float = 0.1, dropout_2: float = 0.1):

    '''
    Clase que implementa el modelo ViT transformer. Se apoya de la clase EncoderBlock y la función gen_patches.
    img_size: tamaño de las imágenes de entrada (int)
    patch_size: tamaño de los patches (int)
    embedding_dim: tamaño de los embeddings a ocupar (int)
    mlp_dim: tamaño de la capa lineal del MLP a ocupar en el Encoder (int)
    n_blocks: cantidad de bloques o "capas" en el Encoder
    heads: número de "heads" por los que hacer "self-attention" en paralelo (int)
    n_classes: cantidad de clases a predecir (int)
    n_channels: número de canales de las imágenes de entrada (int)
    dropout: Dropout a aplicar entre ambas capas lineales (float)
    dropout_2: Dropout a aplicar después de la última capa lineal (float)
    '''

    super(ViT, self).__init__()

    self.img_size = img_size # dimension de H y W
    self.patch_size = patch_size # dimension de patches
    self.linear = nn.Linear(n_channels * self.patch_size ** 2, embedding_dim) # capa lineal de proyección de patches

    self.cls = nn.Parameter(torch.rand(1, embedding_dim)) # token cls
    self.pos = nn.Parameter(torch.rand((img_size // patch_size) ** 2 + 1, embedding_dim)) # embedding posicional

    self.blocks = nn.ModuleList(
        [EncoderBlock(embedding_dim = embedding_dim,
                      mlp_dim = mlp_dim,
                      heads = heads,
                      dropout = dropout,
                      dropout_2 = dropout_2) for _ in range(n_blocks)]) # bloques encoder

    self.header = nn.Linear(embedding_dim, n_classes) # header de clasificación

  def forward(self, x):

    '''
    Función que recibe imágenes de entrada en formato de tensor, devuelve una predicción para cada una.
    x: tensor a transformar (torch.tensor)
    '''

    batch_size, C, H, W = x.shape # dimensiones de entrada

    x = gen_patches(x, patch_size = self.patch_size) # generamos patches

    x = self.linear(x) # proyección lineal de patches

    token = self.cls.repeat(batch_size, 1, 1) # token cls
    pos = self.pos.repeat(batch_size, 1, 1) # embeddings posicionales

    x = pos + torch.cat((x, token), dim = 1) # sumamos embeddings posicionales + token cls

    # capas de atención
    for block in self.blocks:
      x = block(x)

    x = x[:, 0] # recuperamos cls token

```

```
67      x = self.header(x) # header de clasificación
68
69      return x
```