Transformer applied to Facial Landmark detection

Francesco Bertolotti

April 3, 2021

1 Introduction

One of the most popular deep learning (DL) architecture for natural language processing (NLP) is the Transformer architecture. The Transformer was popularized in the famous paper "Attention is all you need" [11]. From the original proposal, many architectural improvements have been proposed, e.g. [1, 7, 13] and many other. Recently, the Transformer is being applied also to task for which wasn't originally designed. One notable example is images processing. In particular, [2] adapt the Transformer architecture to process images. This represents one of the firsts instances of convolution free architecture to achieve comparable results with the state-of-the-art (SOTA). The same idea is further explored in [12] using a pyramidal-like Transformer architecture. This architecture achieves further improvements surpassing results obtained by ResNets architectures [3]. While the Transformer architecture is gaining further traction in image processing tasks the potentialities reamain mostly unexplored. With this project, we aim to apply and adapt the Transformer for the task of "Facial Landmark Detection". While, most of the previous work aim to a convolution-free architectures, we aim to combine the ResNets and the Transformers hoping to gain further improvements.

2 Vision Transformer

In this sections, we are going to expose the inner workings of the Transformer architecture. Firstly, let us start by the input. The Transformer takes as input n embeddings of size d. In our case, embeddings will be patches of an image, as shown by the Fig. 1. Let us call $X = [x_1, \ldots, x_n] \in \mathbb{R}^{n \times d}$. Now that we have our patches, we can feed them to the first Transformer Encoder layer. Let us introduce three parameter matrices $W_Q, W_K, W_V \in \mathbb{R}^{n \times d}$. We proceed to compute the so called quries, keys and value vectors:

$$Q = X^T \cdot W_Q, K = X^T \cdot W_K, V = X^T \cdot W_V$$

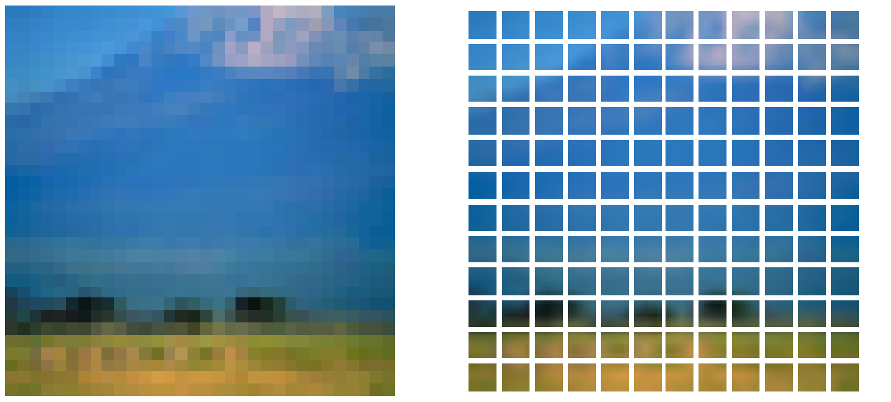
Once we have computed these matrices representing three different linear transformation of the input, we proceed by applying the so called self-attention. But firstly, let us introduce the softmax non-linearity. The softmax function takes as input a vector $v \in \mathcal{R}^{1 \times k}$ and outputs a vector $v' \in \mathcal{R}^{1 \times k}$ such that $\sum_i v'_i = 1$ and $v_i \leq v_j \implies v'_i \leq v'_j \forall i, j$:

$$softmax(v)_i = \frac{e^v_i}{\sum_j e^v_j}, v' = [softmax(v)_1, \ldots, softmax(v)_k]$$

Finally, we can compute the self-attention:

$$Z = softmax(\frac{Q \cdot K^T}{\sqrt{d}}) \cdot V$$

This represent the heart of the Transformer architecture. By $Q \cdot K^T$, we are computing the dot product between each row-vector in Q and each row-vector



(a) full image (b) image divided in $12 \times 12 = n$ patches

Figure 1: both figures are taken from the Keras vision transformer tutorial available at `https://keras.io/examples/vision/image_classification_with_vision_transformer/`

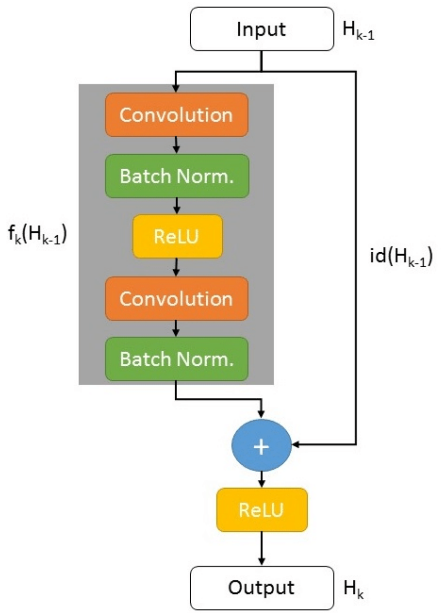
in K. This returns a similarity matrix scoring each query against each key. Afterwards, we turn these scores (by applying the softmax function row-wise) into probability distributions. By now, we have a probability distribution for each patch in our image. Each probability distribution tell us how much similar is a patch wrt. the others. The last matrix multiplication is going to use these distributions to sum all the patches weighted by the probability score. Ultimately, self-attention is way to reroute information between vectors based on their similarity. The last component of the encoder layer applies a standard feed forward network and a skip connection with the original input. Now, multiple encoder can be stacked togheted to achieves, usually, higher results.

You may have noticed that the Transformer Encoder has no way of knowing if a patch comes from the center of the image or comes from the left corner of the image. To mitigate this issue, we introduce the positional embeddings. These are simply paramenter vectors $[e_1, \ldots, e_n] \in \mathcal{R}^{d \times n}$. These vectors are meant to represent positions and are summed altoghter with our input. We can redefine the Transformer Encoder input as follows:

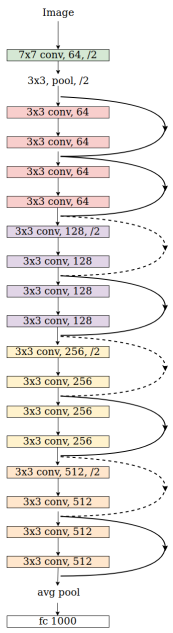
$$X = [x_1 + e_1, \ldots, x_n + e_n]$$

Another important component is the Transformer Decoder. However, in the final architecture of this project will not be used. Thus, we are not disscuss this component. Nonetheless, additional material about the transformer can be foung `http://jalammar.github.io/illustrated-transformer/`.

With these considerations, our overview of the Transformer architecture is complete.



(a) Depiction of a convolutional block from [5]



(b) Depiction of a ResNet from [3]

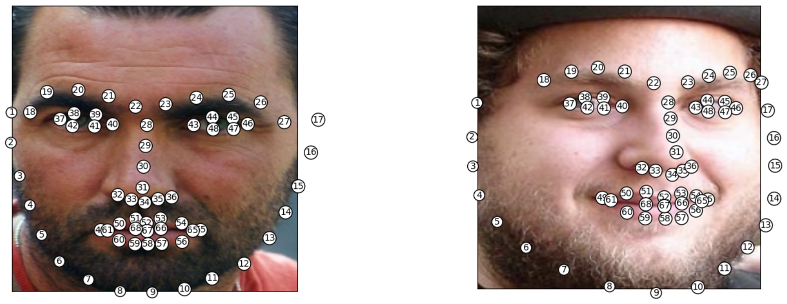
3 ResNet

As said before, we are going to combine both the architectures of ResNets and Transformers. While, we have covered most of the concepts of the Transformers, we are still lacking the necessary concepts about ResNets. Let us start with input of this architectures: an image $X \in \mathbb{R}^{C \times H \times W}$, where C represents the number of channels, H represents the image height and W represents the image width.

The input image will go through a series of convolutional blocks. Each convolutional block is shown in Fig. 2a. As you can see, the input (a tensor of dimension $C \times H \times W$) goes through convolutional layers an activation layer and normalization layers. After this, the tensor is summed with the input and another activation is applied. One of the most important thing to note is the skip connection named `id`. This type of connections, discovered to be effective by [3], are resposible for improving performance, reduce the vanishing/exploding gradient problem and smoothing the loss landscape [6]. Following to this simple structure, usally, a max-pooling layer is applied reducing the number of features per channel.

A full depiction of a resnet architectures is shown in Fig. 2b. As you can see, many convolutional block are applied one after another increasing the number of channels from the original number (usually 3) to 512. The final depicted block is responsible for outputting the classification.

While the full intuition behind this architecture is extremely important it



(a) Example 1 (b) Example 2

Figure 3: Two example with respective annotations from the 300W dataset

also out of the scope of this report. However, it is worth to spend few more words about convolution. It is known that, initial conlutional layer captures low-level features of the image e.g. lines, corners and so on. Instead, higher conlutional layer uses the low level features to capture more and more abstract features like eyes, mouths and so on.

4 Dataset, Applications and Metrics

Dataset In this section we are going to discuss the dataset and the task that is proposed i.e. Facial Landmark Detection. Our dataset of choice is the 300W dataset [9, 10, 8]. Many other datasets are available, however this one is one of the most popular. 300W comes already split into three parts: Test, Validation and Train each composed of $\sim 600, 130, 3700$ respectively. Each sample in the dataset is composed of an image and 68 landmark points on the image. The objective is to train a model to match the given points to the image. Fig. 3 shows two examples.

Applications Facial landmark detection can be applied to a variety of important task. For example, *face animation* and *face reenactment* can both exploit the automatic generation of facial landmark. Other important tasks are *driver status tracking* and *emotion classification* and *face recognition* [4].

Metrics The goal of this task is predict accurate landmark on the human face. Therefore, a metric to measure the goodness of a model must take into account this fact. The most popular metric is the *mean error*:

$$ME = \frac{1}{68}\sum_{i=1}^{68} ||y_i - \hat{y_i}||$$

Where $y_i \in \mathbb{R}^2$ represent the prediction of i-th landmark and $\hat{y_i} \in \mathbb{R}^2$ represent the true i-th landmark position. Despite seeming perfectly reasonable as metric

ME lack an important normalization factor. In fact, An error of 10 pixel in $4K$ images can be negligeble while the same error on images of size 224×224 can be considered a poor classification. To overcome this issue a normalization factor is introduced d. While the importance of d is out of discussion there is not a true consensum among which normalization to adopt. In some cases d is the bounding box dimension, in other cases is the distance between the pupils but, most commonly, it is the distance between the 37-th landmark and 46-th landmark. In this report, we will consider only the latter.

$$NME = \frac{1}{68} \sum_{i=1}^{68} \frac{||y_i - \hat{y_i}||}{d}$$

5 Combining ResNets with Transformers

In this section we are going to discuss the chosen architecture. Firstly, we consider a pre-trained ResNet on ImageNet. In particular, ResNet101. We proceed by gaining access to different layers of the ResNet:

```python
self.resnet  = torchvision.models.resnet101(pretrained=True, progress=True)
self.resnet3 = torch.nn.Sequential(*list(self.resnet.children())[6:8])
self.resnet2 = torch.nn.Sequential(*list(self.resnet.children())[5:6])
self.resnet1 = torch.nn.Sequential(*list(self.resnet.children())[3:5])
self.resnet0 = torch.nn.Sequential(*list(self.resnet.children())[0:3])
...
res0 = self.resnet0(imgs)
res1 = self.resnet1(res0)
res2 = self.resnet2(res1)
res3 = self.resnet3(res2)
```

In the above listing, we split ResNet101 into four steps. These steps give us access to low-level and high-level features (`res0`,`res1`,`res2`,`res3`). These features will be fed to our Transformer Encoder. In particular, the sizes of these features are: $\text{imgs} \in \mathbb{R}^{b\times3\times224\times224}$, $\text{res}_0 \in \mathbb{R}^{b\times64\times112\times112}$, $\text{res}_1 \in \mathbb{R}^{b\times256\times56\times56}$, $\text{res}_2 \in \mathbb{R}^{b\times512\times28\times28}$, $\text{res}_3 \in \mathbb{R}^{b\times2048\times7\times7}$. Where b represents the batch size. Next, we interpret these features as images. Thus, we can subdivide these images into patches.

```python
from einops import rearrange as r
res0 = r(res0,"b c (h1 h2) (w1 w2) -> b (c h1 w1) (h2 w2)",h1=2,w1=2)
res1 = r(res1,"b c (h1 h2) (w1 w2) -> b (c h1 w1) (h2 w2)",h1=2,w1=2)
res2 = r(res2,"b c (h1 h2) (w1 w2) -> b (c h1 w1) (h2 w2)",h1=2,w1=2)
res3 = r(res3,"b c h w -> b (h w) c")
```

Now, `res0`,`res1`,`res2` and `res3` contains more channels with linearized patches. Once again, it is worth looking at the shapes of these tensors. $\text{res}_0 \in \mathbb{R}^{b\times1024\times784}$, $\text{res}_1 \in \mathbb{R}^{b\times1024\times784}$, $\text{res}_2 \in \mathbb{R}^{b\times2048\times196}$, $\text{res}_3 \in \mathbb{R}^{b\times2048\times49}$. Now, it should not be too hard to interpret these linearized patches as tokens (much like in NLP) and feed them to a Transformer. However, there are two considerations to be made. Firstly, all patches sizes should be the same. To achieve this, we can employ a feed forward network. Secondly, we have $1024 + 1024 + 2048 + 2048 = 6144$ tokens which require too much memory. Thus, to reduce the memory footprint, we average several tokens togheter.

```python
from einops import reduce as rd
res0 = rd(ffB0(dropout(gelu(ffA0(res0)))),"b e (c1 c2)->b c1 e","max",c2=16)
res1 = rd(ffB1(dropout(gelu(ffA1(res1)))),"b e (c1 c2)->b c1 e","max",c2=8)
res2 = rd(ffB2(dropout(gelu(ffA2(res2)))),"b e (c1 c2)->b c1 e","max",c2=4)
res3 = rd(ffB3(dropout(gelu(ffA3(res2)))),"b e (c1 c2)->b c1 e","max",c2=1)
```

With the previous snippet, we solved the discussed issues. Now, we have a reasonable ammount of tokens (441) each one of them having the same size (512). We can concatenate these tokens togheter and proceed to add the positional embeddings.

```python
self.positions = torch.nn.Embedding(441,512).weight.unsqueeze(0)
...
srcs = torch.cat([res0,res1,res2,res3],1)
srcs += self.positions.repeat(batch_size,1,1)
```

At this point, we have prepared the input for the transformer. We feed the prepared patches to the Transformer. Next, we can downscale the first 68 tokens to only to *two* features representing the landmark position on the face.

```python
from torch.nn import TransformerEncoder
from torch.nn import TransformerEncoderLayer
self.encoder = TransformerEncoder(
                    TransformerEncoderLayer(512,
                        nhead=8,
                        dim_feedforward=2048,
                        activation="gelu",
                        dropout=.1),
                    num_layers=4)
self.downscale = torch.nn.Linear(512,2)
...
mems = self.encoder(srcs.transpose(0,1)).transpose(0,1)
ldmk = self.downscale(mems[:,:68])
```

With this last piece, we have practically defined the entire architecture. The only thing remained to discuss is the training procedure. We used a *l1 loss* with *Adam* optimizer, *learning rate* set to $1e^{-4}$ and *batch size* of 32.

Since the proposed dataset (300W) is quite small, we introduce three augmentation techniques.

1. random rotations.
2. random crop.
3. random jitter.

6 results

In this section, we are going to show the results achieved by the proposed architecture in terms of normalized mean error (NME).

References

[1] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[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] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[4] Khabarlak, K., and Koriashkina, L. Fast facial landmark detection and applications: A survey. *arXiv preprint arXiv:2101.10808* (2021).

[5] Kumra, S., and Kanan, C. Robotic grasp detection using deep convolutional neural networks.

[6] Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913* (2017).

[7] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[8] Sagonas, C., Antonakos, E., Tzimiropoulos, G., Zafeiriou, S., and Pantic, M. 300 faces in-the-wild challenge: Database and results. *Image and vision computing 47* (2016), 3–18.

[9] Sagonas, C., Tzimiropoulos, G., Zafeiriou, S., and Pantic, M. 300 faces in-the-wild challenge: The first facial landmark localization challenge. In *Proceedings of the IEEE International Conference on Computer Vision Workshops* (2013), pp. 397–403.

[10] Sagonas, C., Tzimiropoulos, G., Zafeiriou, S., and Pantic, M. A semi-automatic methodology for facial landmark annotation. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops* (2013), pp. 896–903.

[11] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[12] Wang, W., Xie, E., Li, X., Fan, D.-P., Song, K., Liang, D., Lu, T., Luo, P., and Shao, L. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. *arXiv preprint arXiv:2102.12122* (2021).

[13] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237* (2019).