Notes: "Image Transformer"

Sun

August 26, 2020

1 Summary:

1. Transformers, but on images.
2. Recent stuff is near-SOTA or SOTA for in-painting and generation. Up there with NVAE and StyleGAN2

1.1 High level:

Common image generation methods have issues. RNNs are autoregressive and poorly parralelizable, leading to slow training times in addition to 'traditional' RNN issues such as vanishing gradients. CNNs have extremely limited receptive field, and increasing this (either via bigger kernels, or more layers) brings a significant amount of parameters.

The Transformer is good for sequence tasks. So lets treat images as a sequence of pixels. We now run into the issue of attention being extremely expensive, so lets localize the attention window to some subset of all the pixels. We can now just apply the transformer, almost without modification, to image generation, and it just works. Praise residual connections.

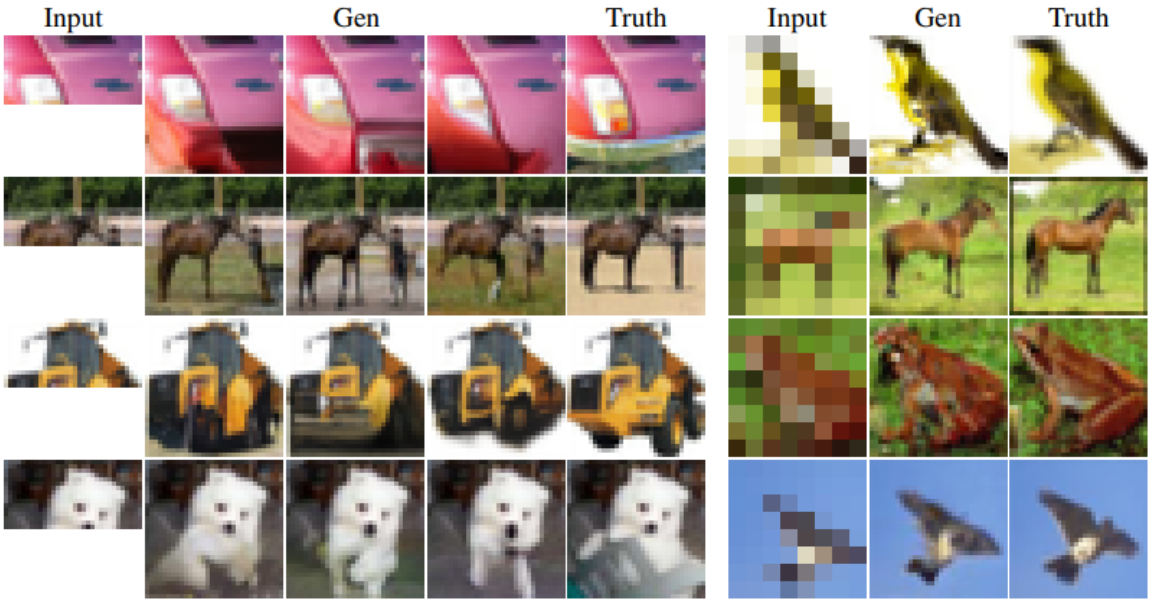

Figure 1: Left: image completion. Right: image super-resolution.

1.2 Medium level:

There are some issues that aren't quite clear.

1.2.1 Attention

Using attention For one, how can we actually use attention? Sure, we do apply inside the multi-head attention, but is that really expressive enough? The paper treats pixels as discrete 8-bit values, then uses an embedding to project this pixel value up to 256 dim, with different embedding layers per channel. I personally don't think this is ideal, and some other transformation that starts contextualized (such as a convolution) would be better, however, this paper probably didn't do that for a reason.

Local attention It's also not entirely obvious how to do local attention. Considering a relatively small image, $32 \cdot 32 \cdot 3$ has 3072 positions, each one of dim 256, we cannot attend to all positions in the sequence. The authors propose multiple methods, but the one with the best scores is '2d local attention', seen in figure 2.

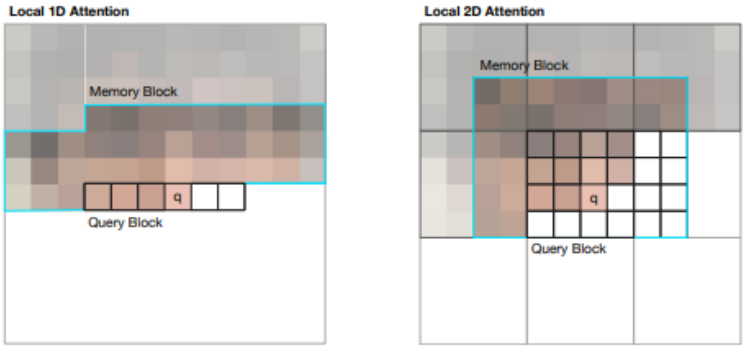

Figure 2: The proposed local attentions.

So in the end, we do go back to a limited receptive field, but it is arguably larger and more expressive than most convolutional networks.

This method is actually very smart, and is a good example where 'traditional' CS can be applied to engineering problems.

We first partition a given image into query blocks, and associate each query block with a large "memory" block that also contains this query block. For all queries in a given query block, attention attends to the same memory block, with proper masking. This allows for parralelization of attention, something effectively required due to its huge computational cost.

1.2.2 Loss:

Using a 'traditional' categorical distribution here is inefficient, considering you'd have $32 \cdot 32 \cdot 3 \cdot 256 = 786,432$ different outputs. There are also some inherit issues with treating pixel values as discrete, which I mentioned earlier. Instead of going with naive CE, they used the discretized mixture of logistics (DMoL).

DMoL has a lot of benefits for this case, mostly:

1. It reduces the number of outputs from 780k to 102k, reducing memory overhead and allowing more meaningful gradients.

2. It changes from discrete outputs to continuous, better capturing the continuous nature of pixel intensities,

3. It allows for more explicit dependence between channels, reducing the need for the transformer decoder to internalize this property.

1.3 Low level:

1.3.1 DMoL:

I didn't mention *what* DMoL was above for a reason; it's worth its own explanation.

We want the model to output two things:

1. What it thinks the pixel intensity is,
2. How confident it is in this choice.

Naive CE fits this; we get both of the above criteria. However, it considers dimensions independent, e.g.: if we confidently predict the pixel intensity to be x, and it is actually $x+1$, the model will be harshly penalized the same as if it predicted x when it is actually $x + 200$.

We can't output a single scalar, as it doesn't include confidence.

What we do instead, is we output a μ and a $\log(s)$ for each dimension, where μ is the mean, and s is the scale, of a logistic distribution. This allows the model to output both a value that it thinks it is (μ) and how confident it is (s, the scale). s is roughly anagolous to σ for a Guassian, so $s \to 0$, confidence $\to \infty$.

We could theoretically just use MSE on μ, but this would not allow us to optimize s.

Instead, let us look at some helpful graphics.[1]

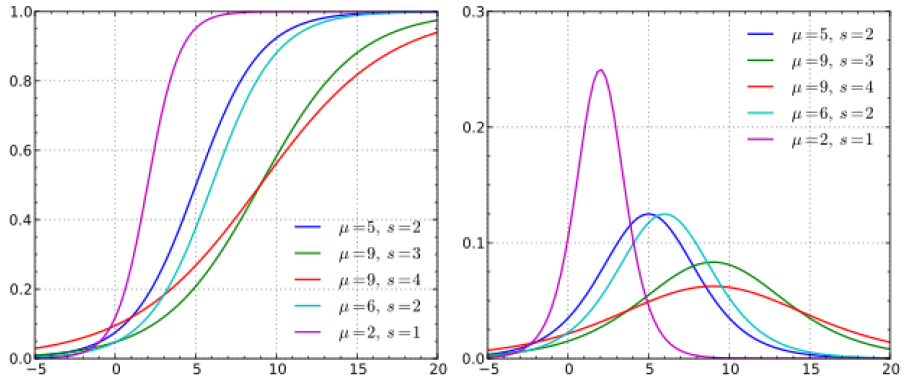

Figure 3: Some logistic distributions.

On the left, the CDF. On the right, the PDF. If you look closely at the CDFs, you may notice that we can optimize both terms we want at once.

The point of inflection ($f''(x) = 0$) on the CDF corresponds to μ, and the more confident the prediction (the lower the s), the higher the slope.

From here, given a label y, we can pick some small region ϵ around it to look at. We can then say that we want the difference between $y-\epsilon$ and $y+\epsilon$ to be as large as possible; the difference between these two points is very high when the model confidently predicts the correct value. It is lower when it unconfidently predicts it correctly, and very low when incorrect, going lower the more confidently it predicts incorrectly. By optimizing this value to be high, we encourage the model to confidently predict correctly.

[1] From the wikipedia article on the Logistic Distribution

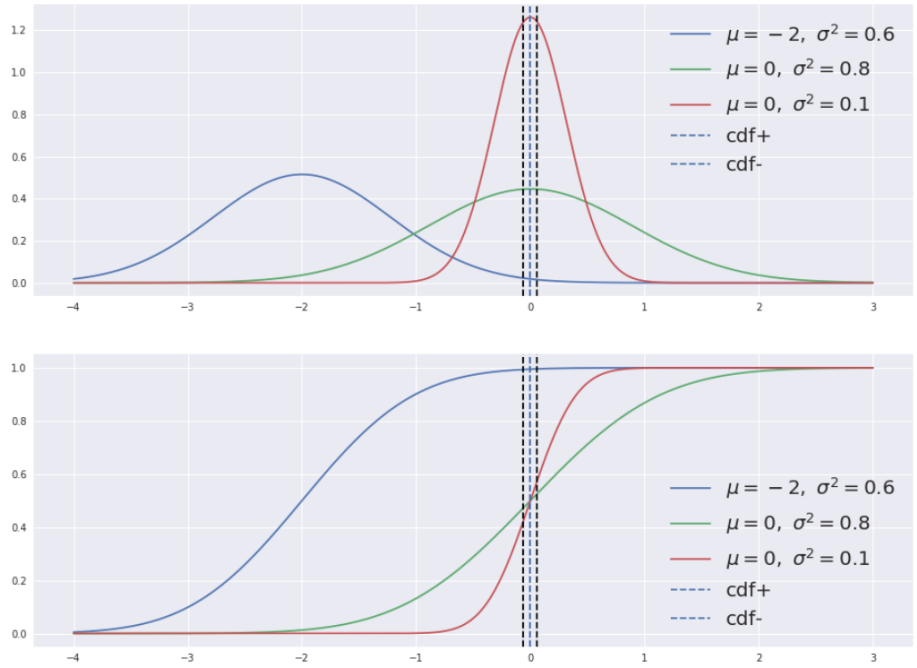

Figure 4: Top: logistic PDF. Bottom: logistic CDF.

We can look at this visually, too.[2]

In the below above, each black line represents $y \pm \epsilon$. Looking at different logistic curves, we can clearly see this working. For the red curve, the difference is high, as the model predicted correctly. For the blue curve, the difference is low considering its unconfident incorrect prediction. And for the green curve, it is lower than the red one due to the lack of confidence, despite being correct.

Epsilon itself is a small number, equal for each 'class', such that $[y-\epsilon, y+\epsilon]$ represents a single 'class', very similar to quantization buckets.

The method above is simply a way to implement DMoL. DMoL itself is MLE, however, I couldn't easily tie the $y \pm \epsilon$ stuff to the distribution, so you'll have to take my word for it or read the citation yourself.

All of the above examples have been with single distributions. In the paper, the model uses 10 mixtures, leading to 100 outputs per pixel. Per mixture, it predicts:

1. One weighting amount for each mixture,
2. One μ for each channel,
3. One s for each channel,
4. One coefficient which captures linear dependence between channels.

[2] I used graphics from Rayhane Mamah, https://github.com/Rayhane-mamah/Tacotron-2/issues/155