Neural Networks for Data Generation – Transformers and GANs

CS 360 Machine Learning
Week 11, Day 2

April 11, 2024

Contents

1 Transformers

Transformer architectures can be used for almost any task. The classic architecture from the "Attention is all you need" paper is for machine translation. Some parts of the architecture are unnecessary if your goal is generating text, since the input and the output are the same encoding space. We'll discuss some of the key parts of the architecture that are relevant to text generation.

1.1 Positional encodings

Positional encodings are added to the input in order to show the order of words or letters in the sentence. When considering attention mechanisms (discussed last class), this is essentially a weighted average. This is a permutation invariant measure, i.e. you can reorder numbers given as input into a function that takes an average and still get the same average. This can be a useful property for some problems. However for text problems, the word order is important so it needs to be explicitly encoded. You could do something where you explicitly encode the position, e.g., the 15th word, in a sentence. But this doesn't work for arbitrarily long pieces of text and it also doesn't capture the relative ordering, for example where words that are 5 words apart are within a similar relative position to each other even if those positions are 15 to 20 instead of 20 to 25. We could try to do this by throwing data at the problem... but this requires a lot of data. Instead, we can use fixed positional encodings to represent these relative positions. To do this, we use sine and cosine.

Formally, the positional encoding functions we'll use are:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i}/d_{model}}\right) & \text{if } i \mod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)}/d_{model}}\right) & \text{otherwise} \end{cases}$$

where d_{model} is the embedding size and *pos* is the position in the sequence. We'll generate one positional encoding for every embedding dimension i from 0 (inclusive) to d_{model} (exclusive) and for every position *pos* in the sequence. In the case where the position is even, we use the sine function, and if the position is odd then we use the cosine function. This will give us a slightly different position in each dimension for the model because of the shifts used for the sine and cosine function. This creates unique positional encodings for each position in the sequence. Since sine and cosine are bounded between -1 and 1, all the activation functions get to use this bounded domain. This generates a matrix of positions by model dimensions; this creates the input to the transformer. Positional encodings are used essentially whenever the order of the sequence is important. (In biology, the order of specific individuals in a sequence are not ordered, so we would not use a positional encoding in this case.)

1.2 Transformer workflow for text generation

In the slides, you'll find a tutorial with useful visualizations that walks through a workflow that's similar to the one needed for our lab. You can think about the input to the transformer as words or characters and the output is the next most likely word (or character). You'll actually have logits (scores) for more text than just the final word (character), but we only care about that specific prediction. Once we have the input we use a positional encoding, as described above, then go through a series of transformer blocks followed by a more traditional architecture block (e.g., a FNN), finally followed by an activation function.

In order to create the transformer, we need to learn the vocabulary of the language. We'll call the size (dimension) of this vocabulary d_{vocab}. We want to embed this in a lower dimensional space since the vocabulary could be quite large. We also need to learn the context, e.g., the window of the input. These can be small or large depending on how much text you have. The goal of this first part is to transform whatever input we have into the embedding size.

The weights we have to learn for the transformer architecture include weights for the queries, keys, and values for the attention mechanism and then we need to put these all together. We'll also need the weights for the multilayer perceptron (MLP) which is a series of fully connected layers at the end of the transformer block. At the output of this block we'll still have the same dimension as the input into the block (d_{model}). The resulting final block will go from the d_{vocab} size to the activation function and final result. When people talk about large language models (LLMs), they're not really talking about the size of the input, they're talking about the number of weights needed to learn. There can be a lot of such weights!

Now let's more specifically talk through the key pieces of this architecture.

1.2.1 Input steps for the transformer

First, we use a one-hot encoding of the input text. The text is tokenized (e.g., broken into letters or words, for our lab we just use letters) and beginning-of-sentence and end-of-sentence (EOS) markers are added. The EOS marker is our placeholder for the prediction we'll end up doing. Our one-hot encoding generates a matrix that has dimensions d_{vocab} by n_{ctx} or number of tokens in the context, i.e. the number of tokens from the vocabulary that actually appeared in the given input text (often called the *prompt*).

The matrix representing the one-hot encoding of the input is unfortunately too sparse and too big – it would require learning many many 0 weight values. So the next step is to take this one-hot encoding and use an embedding that projects this information down into a lower dimension, d_{model}. Instead of using a learned embedding, we'll use a direct embedding determined based on a weight matrix W_E. This will give us a matrix that is d_{model} by n_{ctx}. Next, we add our positional embedding to the embedded input to get a position embedding version of the input. Finally, we do normalization – we'll skip over this, but in real world transformer use, this step is important. The result is our input matrix.

1.2.2 Transformer block steps

We take the determined input matrix and multiply it separately by weights for the queries, keys, and values, each of which has dimension d_{head}, known as the attention head size, by d_{model}. This is our attention mechanism (discussed last class). This reduces the size of the dimension, but there are many of these which effectively become specialized to help learn different aspects of the problem. The attention head size is chosen independently of the other parameters.

Multiplying the input by this attention gives us matrices that are d_{ctx} by d_{head} for each of queries, keys, and values. Taking the dot product, we get attention maps (discussed last class) showing the interaction between different words in the sentence. This matrix is square (d_{ctx} by d_{ctx}) showing each word in comparison to each other word. Recall from last class that we want to make sure the model doesn't peek ahead of the current existing context in the sentence. We mask the attention scores by adding negative infinity to all attention scores above the diagonal. (The attention maps are one way that transformer maps can be explicitly encoded.)

Recall that our goal is to predict the next character at every step. Next, we apply the softmax function for each row. This gives a probability distribution over every word, showing how much attention is applied per word to each of the previous words in the sentence. Each of these rows sums to 1. This gives a weight for each of these previous words. We next multiply this by the values – this is the self-attention step. The result of this multiplication is the result of applying the attention to the values giving the weights of the values going into the next step. We can think of this as a weighted average. Finally, we transform this into the dimension of the model, d_{model}, for input into the next self-attention layer. This, along with some other details including fully connected layers, completes the transformer block. These transformer blocks can be repeated multiple times.

The output of the transformer block is a matrix with size context times the embedding size. A weight matrix can then transform this back from the embedding to the full vocabulary size (d_{vocab}) to make the prediction. For each step in the sentence we try to predict the next word (or character).

Finally, in order to predict text, you can look at the softmax probabilities for the EOS token to predict the next token. Optionally, you could use the notion of temperature here to vary the next token prediction. Higher or lower temperatures may also be more or less appropriate depending on application; for example, for mathematical or scientific applications, we might want a lower temperature while for creative endeavors like writing a model we may want a higher temperature.

Using the predicted token, you can then create the new input – tacking on this predicted token to the previous context minus the original first token – and repeat the process from the beginning. Note that this doesn't require retraining of the model, just an encoding of the new input.

2 Generative Adversarial Networks (GANs)

GANs have been very useful for a lot of different fields and are very interesting. GANs have improved a lot from their introduction in 2014. They can be used to generate, for example, fake images of people. At this point, they can generate faces that are essentially indistinguishable from real people. You can also generate art work or other images, depending on the training data. These have also been used for image generation based on text. These can be useful if you have a small amount of data, or even random noise, and you want to generate a high dimensional output, such as an image with many pixels. Transformers have largely replaced GANs for text, but GANs are still very useful for images and biology applications.

Suppose that you have an art forger, someone who is just starting out in the business. Can you tell which art piece is real and which is fake? Perhaps over time the art forger gets better and it becomes harder to tell the difference between real and fake pieces. The key idea of GANs is this idea of improvement. This works by having two architectures that are both trying to get better at a task. The *generator* architecture is trying to get better at generating fake images or other data; the *discriminator* is trying to get better at telling the difference between real and fake creations. We can think of the generator as the forger and the discriminator as the art critic trying to decide which is the forgery and which is real.

Both of these architectures are often CNNs, these can be used for both generators and discriminators. Unlike many machine learning tasks, the goal of the overall GAN is to reach 50% accuracy of the discriminator, i.e. we want the discriminator to have trouble telling the difference between real or fake. That's why we call GAN's "adversarial" – because these architectures are competing against each other and getting better over time.

2.1 GANs for Biology

How could GANs be useful for biology; why would we want fake data? GANs can be useful because we often want realistic simulated data for different real world scenarios. Because we can't ever run evolution twice to get data again, we use simulated data a lot for experimental purposes. To create simulated data, traditionally people guess various parameters that have real-world biological meaning and are used as part of hand-created evolutionary models. But these weren't successfully matching the real-world observed data. Sara and her students worked on creating a GAN to generate synthetic data for any population. This method takes real data and generated data fed into the discriminator, which guesses whether it's real or fake and uses various evolutionary models to generate the (fake) data. The resulting synthetic data does a much better job of matching real-world population data than previous methods.

2.2 Training GANs

What do the loss functions for GANs look like? Usually, the real data is denoted X and the simulated data is denoted Z. M is the number of examples of data for both real and fake data. The generator loss is:

$$\mathcal{L}_G(\Theta) = -\frac{1}{M}\sum_{m=1}^{M} \log D\left(z^{(m)}\right)$$

where D is the probability of something being real, and $1-D$ is the probability that the data is fake. The discriminator loss is:

$$\mathcal{L}_D(\Theta, X) = -\frac{1}{M}\sum_{m=1}^{M}\left[\log D\left(x^{(m)}\right) + \log\left(1 - D\left(z^{(m)}\right)\right)\right]$$

The training of these architectures alternates. Note that the generator loss doesn't include information about the real data – it's just trying to get better at creating fake data, i.e. at making convincing forgeries.

One way of thinking about the discriminator is to consider what happens if the data is real. We have an implicit label in this case where $y_i = 1$ If we think about the binary cross entropy loss:

$$H(y, \hat{y}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)$$

then when $y_i = 1$ this gives us $\log D(\vec{x}_i) + 0$ and if the data is fake, i.e. if $y_i = 0$ then this gives us $\log(1 - D(z_i))$. Assuming that the true value is one-hot encoded, then only one term will be non-zero in the cross entropy loss. For the generator, we could similarly consider this as the cross entropy loss if we pretend that the label $y_i = 1$, giving $\log D(z_i)$. So the intuition of both of these loss functions is essentially cross entropy loss.

When training GANs, the discriminator and generators need to be balanced in how fast they're learning. At some point, the discriminator will classify everything as real... when this happens, the generator cannot learn and reduce its loss. When training, we want to make sure that training happens gradually so that both the discriminator and generator can actually learn.