Transformers for Time Series Forecasting

Javier Porras-Valenzuela and Alejandro Ribeiro

ESE-2000

1 Introduction

One of the most popular neural network architectures of recent times is called the transformer. Originally, it was very successful in natural language processing tasks, and has since been applied successfully to other domains such as time series forecasting, vision, audio, and multimodal learning. The effectiveness of the transformer can be explained by two main reasons. First, it is able to learn patterns that depend on complex relationships within the context of a sequence, even if the dependencies between elements of the sequence have a long-range. Second, it is highly scalable, which means we can train them on large amounts of data in a parallelized manner, something that earlier sequence processing models, such as RNNs, struggled to accomplish. In this lab, we will explore the use of transformers for time series forecasting.

2 Time series forecasting

Time series forecasting is the discipline of predicting the future as accurately as possible. Traditionally, this is achieved by handcrafting features containing useful information about the series history (e.g. lag features, seasonality and trend components, etc), then fed to a statistical learning model. Since we are interested in time series forecasting using transformers, we will skip most of the basics and look at a more particular (and perhaps complicated) case of time series forecasting: multivariate forecasting. This just means that we want to predict the future value in a time series based on other time series.

Suppose that we want to predict the temperature on a region, given other climatological factors, such as humidity, atmospheric pressure, carbon dioxide measurements, etc. We represent our time series of interest as a sequence of scalar values:

$$y = \{y_1, \dots, y_T\}$$

and we would like to train a model that will learn to predict y_{t+1} based on information known up to time t.

The objective is to learn useful patterns from a sequence of historical data in a window of length N, $\{x_{t-N+1}, \ldots, x_t\}$, where each x_i is a D dimensional vector containing the features that describe the i-th element. In the transformers literature, each sequence element is called a *token*. In this lab, our sequences are time series of weather events with intervals of five minutes, and each token vector has the values for each weather variable at a point in time.

We collect the sequence of tokens into a matrix X of dimension $N \times D$, with the token vectors along the rows[1],

$$X_{t-N+1:t} = \begin{bmatrix} -x_{t-N+1}^T- \\ \vdots \\ -x_t^T- \end{bmatrix}. \tag{1}$$

We might omit the indices whenever it is clear from context, and refer to the feature matrix as X for brevity. Our task is to come up with a model that will predict y_{t+1} from our sequence of tokens X:

$$\hat{y}_{t+1} = \Phi(X) = \Phi(\begin{bmatrix} -x_{t-N+1}^T- \\ \vdots \\ -x_t^T- \end{bmatrix}) \tag{2}$$

We will be using Mean Squared Error (MSE) as our loss function to train this model:

$$\ell(y, \hat{y}) = \frac{1}{T}\sum_{t=1}^{T}(y_t - \hat{y}_t)^2 \tag{3}$$

Task 1 Load the weather.csv dataset available in the course website. Plot the time series for the target variable y "T (degC)". Split the data into training and testing by taking the first 70% of the values as training and the last 30% of the values as testing. Create a PyTorch Dataset object that, given an index, returns a window of size N with the features (TODO ADD FEATURE LIST HERE) at timesteps $t-N+1$ up to t, and the target value y_{t+1} (Hint: override the __get_item__ method).

3 Attention

At the heart of the transformer is one of the most influential ideas in the recent history of AI: the attention operation. Attention is a data-dependent transformation of a sequence into a better representation that might help us to solve a task. To motivate the attention operation, consider the following two sentences:

[1] In many contexts, vectors are arranged as the columns of a matrix, but here we follow the transformer literature, which places token vectors along the rows.

"I spread **jam** on my toast"

"There is a traffic **jam** on the highway"

In these two examples, the word *jam* has two distinct meanings, which can only be inferred from the *context* given by the sentences. It is obvious to us humans that we would not spread traffic congestion onto bread, but this is not the case for a machine learning model. We would therefore like to learn a representation that captures these kinds of nuances.

We would therefore like to map each $\{x_1, \ldots, x_n\}$ into a new sequence of representations $\{h_1, \ldots, h_n\}$ that captures meaningful interactions between the tokens. One way of doing this is by taking a weighted sum of the features of the other words in the sequence:

$$h_n = \sum_{m=1}^{N} a_{nm} x_m. \tag{4}$$

In Equation 4, the scalars $0 \leq a_{nm} \leq 1$, called *attention coefficients*, quantify how important the token m is to the representation of the token n. In other words, it's how much token n should *atttend to* token m.

Now the question is: how do we know how much to pay attention to each token? If we assume tokens that are close to each other are more relevant, one option is to use the inner product between each pair of tokens:

$$a'_{nm} := x_m^T x_n. \tag{5}$$

And in order to enforce that each attention coefficient is between 0 and 1, we can use the softmax function:

$$a_{nm} := \text{Softmax}(a'_{nm}) = \frac{\exp(a'_{nm})}{\sum_{i=1}^{N} \exp(a'_{ni})} \tag{6}$$

Notice that in 6, we are normalizing over the attention coefficients to token n. Together, we can write 5 and 6 into 4 in matrix form as:

$$H = \text{Softmax}[XX^T]X \tag{7}$$

.

3.1 Softmax self-attention

So far we have illustrated attention using a static matrix X. In order to learn for our task, we need to introduce a parametrization:

$$Q = XW_Q \tag{8}$$
$$K = XW_K \tag{9}$$
$$V = XW_V \tag{10}$$
$$\tag{11}$$

where Q,K, and V are called the Query, Key and Value matrices, respectively. W_Q,W_K,W_V are learnable parameters, of dimensions $D \times d_k$, $D \times d_k$ and $D \times d_v$. For now, $d_k = d_v = D$, but this will change later when we introduce multi-headed attention.

Let's fix one more detail: we will normalize the inputs to softmax by a factor of $1/d_k$, to avoid vanishing gradients when there are extreme values. Thus we have our parameterized attention layer:

$$\text{Attention}(\text{Q}, \text{K}, \text{V}) = \text{Softmax}[\frac{QK^T}{\sqrt{d_k}}]V \tag{12}$$
$$= \text{Softmax}[\frac{XW_QW_K^TX^T}{\sqrt{d_k}}]XW_V. \tag{13}$$

Equation 13 is called *softmax self-attention*, because it is computing the attention of X onto *itself*. In other cases, the Q,K and V matrices are constructed from different sources, and the operation is referred to as *cross-attention*.

If you look closely at equation 13, you will see that there is nothing preventing the attention weights from making x_t pay attention to tokens x_{t+1} and beyond. At some point, we will want our model to be able to forecast based on past observations, rather than cheating by paying attention to the future. We can prevent this by *masking* the output of the QK^T unweighted coefficients. This simply means we set to $-\infty$ to the attention coefficients that our model is not allowed to look at:

$$\text{Mask}[QK^T]_{[n,m]} = \begin{cases} QK^T_{nm} & n > m \\ -\infty & \text{otherwise} \end{cases} \tag{14}$$

The consequence of Equation 14 is that, when we pass this to the softmax function, all coefficients with value $-\infty$ will evaluate to zero.

Together, the *masked softmax self-attention* operation is:

$$\text{MaskedAttention}(Q, K, V) = \text{Softmax}[\text{Mask}[\frac{QK^T}{d_k}]]V \tag{15}$$

Task 2: Implement a PyTorch Module for softmax self-attention (Equation 15). Make sure to parameterize d_k and d_v separately, as well as allowing for masked and unmasked attention, we will need them later.

Task 3 Train an unmasked softmax self-attention model to predict the temperature. Report its train, and test MSE and plot its predictions.

We see that the performance of this first model is not great. We have a few more steps before arriving at a complete transformer.

4 Multi-headed attention

The self-attention model that we implemented can only learn a single way of weighting tokens. It is reasonable to think that we want to learn different attention patterns from the same data. We can achieve this by creating multiple *attention heads*. Redefine the learning parametrizations with K heads as

$$Q_k = XW_Q^{(k)} \tag{16}$$
$$K_k = XW_K^{(k)} \tag{17}$$
$$V_k = XW_V^{(k)}, \tag{18}$$
$$H_k = \text{Attention}(Q_k, K_k, V_k) \tag{19}$$

where $k = 1, \ldots, K$ and the dimensions of the parameter matrices are the same as before. The Attention function can either be masked or unmasked. We then concatenate them together and project them into a final output parametrization:

$$H = \text{MultiHeadAttention}(X) := \text{Concat}[H_1, \ldots, H_K]W_O, \tag{20}$$

with W_O of dimensions $hd_k \times D$. This is called multi-headed attention (MHA) Typically the learning matrices here are low-rank projections with dimensions $d_k = d_v = D/K$. This has two advantages: one, it improves learning efficiency, as we are able to learn multiple attention patterns using the same number of parameters as a single head. Two, it is convenient because the output representation H has the same dimensions as our input X, which means we can stack many MHA layers without making our feature space smaller.

Task 4: Implement a PyTorch module for multi-headed attention. Reuse your module for single-headed attention.

Task 5: Train an unmasked, multi-headed self-attention model to predict the temperature, making sure that the number of parameters is the same as the single-headed attention of the previous section. Report its train, and test MSE and plot its predictions. How does it compare to the previous model?

5 Encoder and decoder blocks

After computing the updated representation H, usually the next step would be to pass it through a Multilayer Perceptron to learn nonlinear combinations of the

attention weighted representations. Since we are working with time series, we will replace it with a one-dimensional CNN layer, like the ones used in previous labs, to be able to capture temporal patterns.

$$\text{Encoder}(X)_\ell := \sigma(\text{Conv}(\text{UnmaskedMHA}(Q_{\ell-1}, K_{\ell-1}, V_{\ell-1}))) \tag{21}$$

The *encoder* layers of the transformer use unmasked MHA, so they are allowed to learn from the whole context window.

$$\text{Decoder}(\text{X})_\ell = \sigma(\text{Conv}(\text{MaskedMHA}(\text{Q}_{\ell-1}, \text{K}_{\ell-1}, \text{V}_{\ell-1}))), \quad 1 \leq \ell \leq L \tag{22}$$

where σ is some pointwise nonlinearity, typically ReLU. The values Q_0, K_0, and V_0 are the same as Equations 16, 17, 18, and the output of layer ℓ becomes the input of layer $\ell + 1$. In the literature, this is called a *decoder* layer (encoder layers use unmasked attention, which are used in other tasks).

We are ready to implement a full Transformer for Time Series forecasting. Our full architecture is a series of decoder layers, followed by one final linear map to generate the scalar forecast $\hat{y}_{t+1}$:

$$\hat{y}_{t+1} = \Phi(X) = \text{Flatten}[\text{Decoder}(X)_L] \cdot W_F \tag{23}$$

where Flatten$[\cdot]$ denotes flattening the $N \times D$ representation into a single vector of dimensions ND, and $W_F \in \mathbb{R}^{ND \times 1}$ is set of learnable weights.

Task 6: Implement an Encoder module. Train a three-layer encoder-only Transformer. Try varying the number of heads and model parameters and report on which parameters worked best.

Task 7: Implement a Decoder module. Train a three-layer decoder-only Transformer", varying the number heads and model parameters and report on which parameters worked best.

5.1 Full Transformer Architecture

We are finally ready to build a full transformer for time series forecasting. Figure **??** shows the full architecture. Notice that in the second attention block of the decoder, the input keys and values come from the output of the encoder, while the queries come from the previous layer of the decoder. This is called *cross-attention*.

Task 8: Implement a full Transformer module. Train the transformer, and report on the results. Compare with decoder-only and encoder-only blocks.

6 References

1. Bishop (Deep Learning)
2. Vashwani (Attention is all you need)