FROM LINEAR REGRESSION TO THE TRANSFORMER ARCHITECTURE *

Campbell Rankine
Canberra, ACT
campbellrankine@gmail.com

***K*eywords** Probablistic Transformers · Time Series Analysis · Linear Transformers · Regression Analysis

1 Problem Formulation

1.1 The standard linear model

Early Time Series Forecasting methods attempted to predict some target output Y using input, weight, and bias matrices: Θ, X, β. These inputs form the basis for our standard linear regression equation:

$$Y = \theta^T X + \beta \tag{1}$$

Our weights are initialized uniformly in the most basic example and we optimize our weight matrix using a stochastic gradient descent method. For that we can define our error/loss as:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{N} \sum_{i \epsilon N}^{N} (y - \hat{y})^2 \tag{2}$$

Finally we define our optimization equation:

$$\theta_0 \sim U(0, 1)$$
$$\theta_{t+1} := \theta_t + \eta \nabla \mathcal{L}(y, \hat{y}) \tag{3}$$

2 Encoder-Decoder Architecture

Learning a representation of raw data can often be computationally expensive and often yields worse results as a consequence of the bias-variance tradeoff. To remedy this we introduce auto encoders. Auto encoders learn a lower dimensionality representation of our data by bottlenecking information between neural layers (figure 1). This lower dimensional representation is known as the latent space. We take hyper parameters:

- D_L: The dimension size of the latent space
- $|E|$: Number of encoder layers
- $|D|$: Number of decoder layers

* *Citation*: **Authors. Title. Pages.... DOI:000000/11111.**

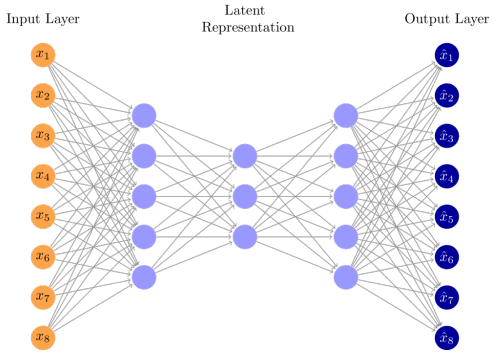

Figure 1: Auto Encoder Architecture

2.1 Vector Quantized Auto Encoders (VQAE)

VQAE's are a type of Probabilistic Auto Encoder comprised of 3 major functions. $P_\phi(\mathbf{z})$: Our encoder layer, $\hat{S}_\theta(\mathbf{z})$: Our decoder layer, and $\mathbf{Z}[X, Y]$: Our vector quantization function. Our input is fed to $P_\phi(\mathbf{z})$, and our vector quantization yields a set of discrete variables $q(z = k \mid x)$ by calculating the nearest neighbour lookup inside the shared embedding space as defined in Eq 4.

$$q(z = k \mid x) = \begin{cases} 1 & k = argmin_j ||z_e(x) - e_j||_2 \\ 0 & else \end{cases} \tag{4}$$

The vector quantization output is fed to the decoder, and serves to minimize reconstruction error, however it is not passed to a model that learns latent representation.

2.2 Reconstruction Error

VQAE's are trained using reconstruction error, a loss function that measures the difference between an arbitrary dimension input and it's corresponding output. The loss function can be found in equation 5.

$$\mathcal{L} = logp(x \mid z_q(x)) + ||sg[z_e(x)] - e||_2^2 + \beta||z_e(x) - sg[e]||_2^2 \tag{5}$$

sg above stands for stopgradient operator, and is a constant defined at training time that is meant to minimize the chance for exploding / vanishing gradients for $\nabla\mathcal{L}$. Terms 2 and 3 above are regularizers that prevent exploding gradients by regularizing against the vector norm of the latent distribution from each end of the encoder.

During training the decoder is optimized according to terms 1 and 3 in $\mathcal{L}$ and the encoder is optimized according to term 2.

2.3 Latent Models

Auto Encoder's provide the means to learn arbitrary dimension representations of data, in a controlled and efficient environment. This is why they are perfect for designing training objectives for computationally intensive data such as images, sentences, or time series representations. We can therefore train models to learn patterns inside the latent space, and then use an encoder-decoder architecture to convert them from and back into their raw format.

In practice often it's better to train the Auto Encoder using the latent objective function as we will learn a representation of the data that is optimal for our training objective. So from here on out we can ignore Eq. 5, however it is useful to see reconstruction error as understanding Auto Encoders is a must for latent learning.

3 Transformer Components

The Transformer architecture is a latent learning model that uses the self attention mechanism to remember/condition outputs on previous values. This process is similar to an LSTM memory cell, although instead of singular datapoints (LSTM) or replay memory (Reinforcement Learning) the self attention mechanism feeds multiple positional embeddings

of a sequence as input to the model. This process is done inside the latent space and provides a higher level representation of past data in memory.

Transformer architectures vary wildly and some well known transformers use Decode Only Architecture like Chat-GPT, or they use a different attention mechanism / encoder-decoder architecture. To simplify explaining the architecture, we'll just use the standard architecture described in the "Attention is all you need" paper.

3.1 Positional Embeddings

For time series analysis / Natural Language Processing we assume the order of datapoints and their position at time $T = t$ heavily influence the target variable. In Encoder Decoder architectures the latent representation of the input removes all positional information. Therefore we must encode position into the input before passing it through the encoder to include positional information in the latent space. To do this we use a positional embedding function, the most common of which is a cosine embedding (Eq. 6 and 7). To embed position we have input vector $v_1 :< x_1, x_2, ..., x_n >$, some scalar value $n = 10000$ (set by the authors of "Attention is all you need"), and the model latent dimension size (D_L) defined in Section 2.

$$P(v_1, 2i) = sin\Big(\frac{v_1}{n^{\frac{2i}{D_L}}}\Big) = p_1 \tag{6}$$

$$P(v_1, 2i+1) = cos\Big(\frac{v_1}{n^{\frac{2i+1}{D_L}}}\Big) = p_1' \tag{7}$$

From the cosine / sin relationship we include relative positional information about v_1 won't be lost once v_1 is encoded. The positional embedding layer will be denoted by a + in figure 2.

3.2 Learned Embeddings

An additional method for embedding positional information into the latent dim is to use a learned positional embedding. This is a fully connected linear layer following the latent dimension shape. The overall structure is a lookup table of embedded vectors and some learnable weight matrix $W_p \sim \mathcal{N}(0, 1)$. We can learn an optimal weight matrix by applying our SGD Optimization method (defined in Eq. 3) so that:

$$P(v_1) = W_p^T v_1$$

yields an optimal encoded vector for our training objective. These embeddings tend to outperform positional embeddings once trained, however they have a fixed input size. This presents a problem for time series data as context windows may not be consistent. Additionally, in real life data sequences aren't of equal length, so we must 0 pad the input vector, taking up additional computational and memory resources.

3.3 Residual / Skip Connections

Given that a layer in our transformer will transform the input, often it's best to pass the input to a later layer of the transformer. This is because we don't want information about our input to be lost as we go through layers of the transformer. To do this we use skip connections that are denoted by arrows in figure 2.

3.4 Add and Norm Layers

To effectively include information about the input in a later layer of the transfomer we add and normalize the input vector v_0 to the output of some layer N inside the transformer. The layer output vector will be denoted v_k. The resultant vector v_i is produced by:

$$v_i = v_0 + v_k$$

$$v_i = \gamma^{(i)}(v_0 + v_k) + \beta_i$$

In summary we perform a learned normalization using weights $\gamma^{(i)}$ to normalize our added vector.

4 Attention Functions

4.1 Scaled Dot Product Attention

Now we introduce the foundation behind the transformer architecture, the Scaled Dot Product Attention Mechanism. For this we introduce 3 learnable weight matrices: W^q, W^v, and W^k. The purpose is to learn different representations of the data depending on position and layer output. Now during training we have our Query, Key, Value matrices defined below:

$$Query = W^q x_i$$
$$Key = W^k x_i$$
$$Value = W^v x_i$$

Additionally to calculate $Attention(Q, K, V)$ we will use a softmax activation function to provide normalized outputs in relation to our query. The softmax function ($\sigma(x)$) is defined in equation 8:

$$\sigma(x) = \frac{Exp(x_i)}{\sum_{i \epsilon N}^{N} Exp(x_j)} \tag{8}$$

Now we follow intuition from vector similarity functions like cosine similarity and calculate a similarity value for the Query and Key vectors. Finally we calculate the dot product between the resultant vector and the Value vector to produce the attention value defined in Eq. 9. Additionally we normalize the query key dot product using some constant defined at training time: d_k.

$$Attention(Q, K, V) = \sigma\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V \tag{9}$$

4.2 Multi Headed Attention

Calculating the scaled dot product attention would lead to high instability and high computation time for vectors with a high dimensionality. Instead we can take linear projections of the K, Q, and V matrices, defined by a fully connected linear layer, and pass these projections to H scaled dot product attention modules. We introduce another learned dimensionality reduction for the Q, K, V matrices, and we calculate and concatenate attention of each of those reduced dimensionality inputs. This is more efficient than scaled dot product attention for our full dimensionality Q, K, V matrices as we can run these reduced dimensionality heads in parallel, and learn more representations of the data. Multiheaded attention can be calculated using Eq 10. W^0 is our final learnable matrix for attention. It projects the attention value back to the dimensionality of the model, using the concatenated reduced dimensionality attention vector of size h

$$MultiHead(Q, K, V) = Concat([head_1, head_2, ..., head_h])W^0 \tag{10}$$

Where $head_i$ can be calculated using equation 11. Each head value is just the attention of the reduced dimensionality projection of Q, K, and V.

$$head_i = Attention(QW_i^q, KW_i^k, VW_i^v) \tag{11}$$

A diagram of both multi headed and scaled dot product attention can be found in figure 2.

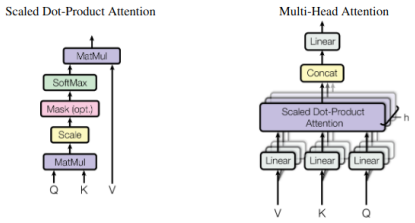

Figure 2: Auto Encoder Architecture

5 Transformer Architecture

Finally we can put these pieces together to yield our final training architecture shown in figure 3. Feed forward layers simply describe fully connected linear layers with a softmax activation function. We then train with a variant of the objective function defined for linear regression in Equation 2.

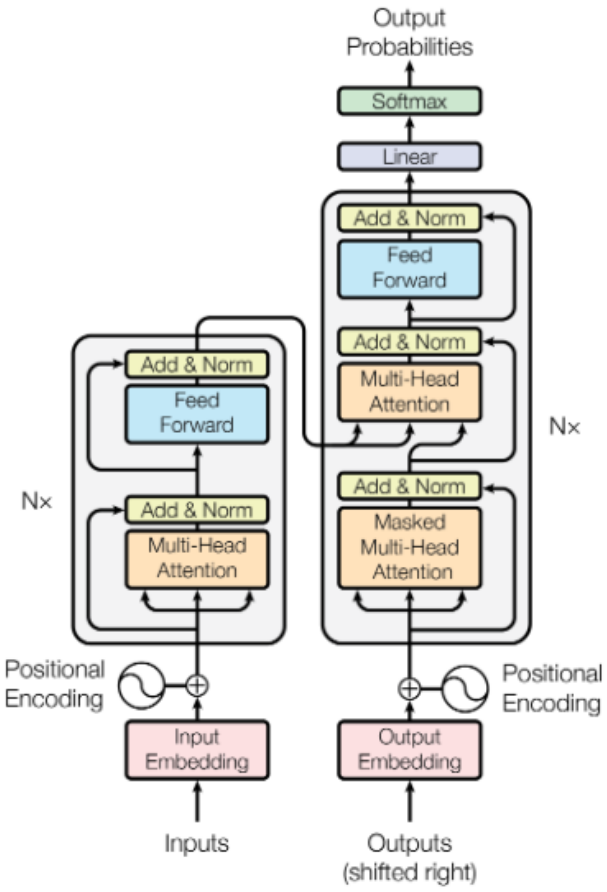

Figure 3: Auto Encoder Architecture