Building LLM for Statisticians

Na (Michael) Li, PhD (wuolong@gmail.com)

Updated: 21 March 2025

LLM for Statisticians

- For statisticians like myself, LLMs (Large Language Models) are fairly easy to understand in principle as they are just machine learning models with a few tricks:
  - Nonlinear activation functions allow complex non-linear models with an overall mostly linear (additive) structure
  - Thus easily differentiable which allows straightforward optimization via gradient descent
  - Layered structure so parameters can be updated in manageable blocks (back-propagation)
  - Large matrix (tensor) operations are amenable to parallel execution and perfect for GPUs
- Still it is a bit of a challenge for a statistician like me to understand the *implementation* of LLMs due to esoteric terminology (what the heck is “attention”?), Python codes, and math (either too much or too little, and in unfamiliar notations).
- Build a LLM from Scratch is an excellent book that allows one to take a peek at how LLM is actually implemented. Still it uses Python and very little math. Here my notes use R instead with some math, that’s easier for myself to understand.

From GPT to ChatGPT

Large Language Model (LLM) is a (large) neural network (NN) model for next-word prediction.

1. The input is sequence of tokens (words, punctuation, and possibly other special context token like `end-of-text`) of length *context size* (typically 256).
   - *tokenization*: breaking down input texts into individual tokens
   - *encoding*: converting tokens into integers (token IDs) by looking up inthe dictionary
   - *embedding*: converting one token ID to a vector of real numbers
2. **Pre-training** takes a large corpus of text to create a base or /foundation/ model, such as GPT (Generative Pretrained Transformers). This is called *self-supervised* training as the training data (the target is the next word/token) is created on-the-fly by a *data loader*. This model is capable of text completion (*zero-shot*) and has limited *few-shot* capabilities, meaning learning to perform new tasks based on only a few examples.
3. **Fine-tuning** involves further training the LLM on labeled data. Two popular categories are /instruction/ (instructions and answers) and /classification/ (texts and labels) fine-tuning. ChatGPT is such a fine-tuned model.

LLM in a Nutshell

- Like all models, LLM is a function that maps the input X (a sequence of words or tokens) to an output Y (a vector of multinomial probability for any token in the "vocabulary").

$$Y = f_{\text{LLM}}(X)$$

- In LLM, calculations are done in a step-wise fashion, where each step (or "layer") can be one of the following, and the output of one step is the input of the next.
  - Linear transformation (in matrix form): $f(x) = xA^T + b$
  - Ancillary actions like normalization or dropout to improve computational efficiency and numerical stability.
  - Activation functions to introduce non-linearity.
- At each step, the output is a matrix or a tensor (multidimensional matrices, which can also be viewed as a larger block-sparse matrix) where each element is often fancifully referred to as a "neuron" or a "node" in the neural network.
- Despite the simple structure, this model can approximate any function when sufficiently large. The model parameters (to be optimized/trained) are only involved in the linear operations. Thus model fitting is feasible even for very large models.

What's the Trick?

- **Transformer** refers to the two steps of the process, an *encoder* that prepares the input, and an *decoder* that does the prediction. Both encoder and decoder can be large neural networks that are optimized with training. For LLMs the encoder is relatively simple and GPT focuses on the decoder only.
- LLMs like GPT taken into account contextual information, called *attention mechanism*. It's essentially a weighted correlation matrix of the tokens, where the weights are parameters to be optimized via training.
- GPT-3's vocabulary includes $V = 50257$ tokens, with a maximum context size of 2,049 tokens. It uses an embedding size of $d = 12288$ dimensions and $L = 96$ layers. The total number of parameters is:

$$V \times d + L \times (4d^2 + 8d^2 + 4d) + d \times V \approx 2 \times 50257 \times 12288 + 12 \times 96 \times 12288^2 \approx 175 \times 10^9$$

- As the model scales up (increased number of nodes or parameters) over certain threshold, unexpected capabilities arise, such as chatGPT doing math. This is called emergent behavior.

PyTorch and R

- PyTorch has surpassed TensorFlow as the ML library of choice (in 2025).
- torch is a native-R interface for GPU accelerated array computation, general neural network layers and data loaders.

```
install.packages(c("torch", "libtorch", "bench"))
```

- On Apple Silicon, Metal Performance Shaders (MPS) backend provides GPU acceleration for matrix calculation. CUDA is used with NVIDIA GPUs. Here the tensor (an array) is stored in GPU.

```
torch::backends_mps_is_available()
```

```
## [1] TRUE
```

```
a.mps <- torch::torch_rand(c(1000, 1000), device = "mps")
b.mps <- torch::torch_rand(c(1000, 1000), device = "mps")
a.mps[1:2,1:8]
```

```
## torch_tensor
##  0.2847  0.3973  0.9793  0.3278  0.1432  0.4976  0.3536  0.1886
##  0.3032  0.2595  0.7262  0.2954  0.6430  0.2013  0.5535  0.7115
## [ MPSFloatType{2,8} ]
```

- Note that in R, vector and array index starts from 1 (FORTRAN style), versus 0 (C style) for Python.

GPU Acceleration

- Tensor on CPU

```
a.cpu <- torch::torch_rand(c(1000, 1000), device = "cpu")
(b.cpu <- torch::torch_rand(c(1000, 1000), device = "cpu"))[1:2, 1:8]
```

```
## torch_tensor
##  0.7336  0.2180  0.2343  0.0688  0.1203  0.2866  0.1235  0.9458
##  0.2443  0.5215  0.9779  0.8567  0.3307  0.7691  0.9304  0.1461
## [ CPUFloatType{2,8} ]
```

- Plain matrix

```
a <- matrix(runif(n = 1000*1000), ncol = 1000)
b <- matrix(runif(n = 1000*1000), ncol = 1000)
```

- Multiplication

```
rbind(bench::mark(c.mps <- torch::torch_mm(a.mps, b.mps)),
      bench::mark(c.cpu <- torch::torch_mm(a.cpu, b.cpu)), bench::mark(c <- a %*% b))[, 1:9]
```

```
## # A tibble: 3 x 6
##   expression                             min  median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                           <bch> <bch:t>     <dbl> <bch:byt>    <dbl>
## 1 c.mps <- torch::torch_mm(a.mps, b.~  31us   52.1us   2235.     13.89KB        0
## 2 c.cpu <- torch::torch_mm(a.cpu, b.~ 931us    1.2ms    731.          0B        0
## 3 c <- a %*% b                        322ms   327ms      3.06    7.63MB        0
```

More about Torch Tensor

- Torch tensors are somewhat mysterious R7 (or S7?) class objects, and pointers to Python objects.

```
aa <- torch::torch_tensor(array(c(0.2745, 0.6584, 0.2775, 0.8573, 0.8993, 0.0390, 0.9268, 0.7388,
                                  0.0772, 0.3565, 0.1479, 0.5331, 0.4066, 0.2318, 0.4545, 0.9737),
                                dim = c(1, 2, 2, 4)))
class(aa)
```

```
## [1] "torch_tensor" "R7"
```

- As pointers to Python objects, they have some “methods” available, but some ordinary R (base or from torch library) functions also work.

```
aa$size()
```

```
## [1] 1 2 2 4
```

```
dim(aa)
```

```
## [1] 1 2 2 4
```

- Two ways of combining tensors: “stack” increases the number of dimensions while “cat” does not. The first line concatenate two $1 \times 2 \times 4 \times 2$ tensors to create a tensor of $1 \times 2 \times 4 \times 4$. The second line stacks them to create a tensor of $2 \times 1 \times 2 \times 4 \times 2$.

```
torch::torch_cat(c(aa$transpose(3, 4), torch::torch_transpose(aa, 3, 4)), dim = 4L)
torch::torch_stack(c(aa$transpose(3, 4), torch::torch_transpose(aa, 3, 4)), dim = 1L)
```

Tokenization

- Tokenization is the process of breaking down raw text into small pieces (tokens, including words, punctuation, etc) and then convert the character strings to integers through by looking up in the *vocabulary*.
- rtiktoken provides R interface to OpenAI's BPE tokenizer python library tiktoken

```
fname <- "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/refs/heads/main/ch02/01_main-chapter-code/the-verdict.txt"
nchar(raw.text <- readr::read_file(fname))
```

```
## [1] 20479
```

```
enc.text <- rtiktoken::get_tokens(raw.text, model = "gpt2")
length(enc.text)
```

```
## [1] 5145
```

- Encoded values:

```
enc.text[1:10]
```

```
##  [1]    40   367  2885  1464  1807  3619   402   271 10899  2138
```

- Converting back, note that it may not identical to the original text as white spaces etc may have been dropped.

```
rtiktoken::decode_tokens(enc.text[1:10], "gpt2")
```

```
## [1] "I HAD always thought Jack Gisburn rather"
```

Data Sampling with a Sliding Window

- Inputs and targets are generated automatically in training a LLM through a sliding window approach. An input vector of context size 4 (`max.len`) is first selected, the target is then shifted by 1 (for next word prediction).
- The next input is shifted for a stride of size 3 (`stride`). When the stride size equals context size there is no overlap between inputs thus less correlation.

```
n <- length(enc.text)
idx <- seq(1, n - max.len, by = stride)
tmp <- cbind(enc.text[(1:(n-max.len))], enc.text[(2:(n-max.len+1))])
inputs <- targets <- array(0, dim = c(batch.size, max.len))
for (i in seq(batch.size)) {
    this.idx <- idx[i]:(idx[i]+stride)
    inputs[i,] <- tmp[this.idx,1]
    targets[i,] <- tmp[this.idx,2]
}
(inputs.t <- torch::torch_tensor(inputs, dtype = torch::torch_long()))
```

```
## torch_tensor
##    40   367  2885  1464
##  1464  1807  3619   402
## [ CPULongType{2,4} ]
```

```
(targets.t <- torch::torch_tensor(targets, dtype = torch::torch_long()))
```

```
## torch_tensor
##   367  2885  1464  1807
##  1807  3619   402   271
## [ CPULongType{2,4} ]
```

Embeddings

- Embedding creates a continuous vector representation for each token ID, which is necessary for training the deep neural network through backpropagation algorithm.
- These are essentially weights that correspond to the contribution of each token to the bottom layer of the network and are optimized through training.

```
embedding.ex <- torch::nn_embedding(num_embeddings = 6, # size of the vocabulary
                                    embedding_dim = 3)  # dimension or number of neural nodes at this layer
embedding.ex$weight
```

```
## torch_tensor
## -0.7695  1.1331 -1.2419
##  1.1693  2.2465 -0.7827
## -1.6537  1.0635  0.2136
## -1.3453 -0.0734  0.6027
## -1.0212 -0.8909  0.0688
## -0.9263 -1.2714  0.5308
## [ CPUFloatType{6,3} ][ requires_grad = TRUE ]
```

```
input.ex <- torch::torch_tensor(c(1, 2, 4), dtype = torch::torch_long()) # needs Long integers as indices
embedding.ex(input.ex)
```

```
## torch_tensor
## -0.7695  1.1331 -1.2419
##  1.1693  2.2465 -0.7827
## -1.3453 -0.0734  0.6027
## [ CPUFloatType{3,3} ][ grad_fn = <EmbeddingBackward0> ]
```

Embedding with Positional Information

- First the create embeddings for the tokens IDs:

```
output.dim <- 256                                          # more realistic size, GPT-3 uses 12,288
embedding.layer <- torch::nn_embedding(num_embeddings = 50257, # size of the GPT-2 vocabulary
                                       embedding_dim = output.dim)
token.embeddings <- embedding.layer(inputs.t)
token.embeddings$size()
```

```
## [1]   2   4 256
```

- GPT uses absolute position embeddings (weights for each position).

```
pos.embed.layer <- torch::nn_embedding(num_embeddings = max.len, # context size
                                       embedding_dim = output.dim)
pos.embeddings <- pos.embed.layer(torch::torch_arange(max.len, dtype = torch::torch_long())) # arange -> sequence 1:4
dim(pos.embeddings)
```

```
## [1]   4 256
```

- Position embeddings and token embeddings add up to the input embeddings.

```
input.embeddings <- token.embeddings + pos.embeddings
input.embeddings[1,1,1:5]
```

```
## torch_tensor
##  0.2646
##  1.3290
## -0.5708
##  0.3094
##  1.7389
## [ CPUFloatType{5} ][ grad_fn = <SliceBackward0> ]
```

Self-Attention

- "Attention" is essentially a weighted correlation between inputs, of context size N and embedding (hidden) dimension d:

$$X_{N\times d} = [x_1, x_2, \ldots, x_N], \text{where } x_i \in \mathcal{R}^d$$

- Query (Q), Key (K) and Values (V) are linear transformations of X using weight matrices of dimensions $d \times d_k$ (typically $d_k = d$ for single-head attention).

$$Q_{N\times d_k} = XW_Q, \qquad K_{N\times d_k} = XW_K, \qquad V_{N\times d_k} = XW_V$$

- In the attention weight, each cell represents how much one token *attend*s to another. It's normalized so the weights sum to 1 across row.

$$\text{Attention}_{N\times N} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

- The output is the attention weights multiplied by the values (weighted inputs):

$$Y_{N\times d_k} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \text{softmax}\left(\frac{X\left(W_Q W_K^T\right)X^T}{\sqrt{d_k}}\right)XW_V$$

Softmax

- The softmax (multinomial logistic) function is a multivariate version of the logistic function.
- It converts a vector z of K real numbers into a probability distribution on K categories.

$$\sigma_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \qquad \text{where } i = 1, 2, \ldots, K$$

- Note that it sums to one.

$$\sum_{i=1}^{K} \sigma_i = 1$$

- In neural networks, softmax function is often used as the last activation function to convert network output into predicated output classes.

Simple Self-Attention (without Weights)

This replicates the codes in Section 3.3.

```
inputs <- torch::torch_tensor(rbind(c(0.43, 0.15, 0.89), # Your
                                    c(0.55, 0.87, 0.66), # journey
                                    c(0.57, 0.85, 0.64), # starts
                                    c(0.22, 0.58, 0.33), # with
                                    c(0.77, 0.25, 0.10), # one
                                    c(0.05, 0.80, 0.55))) # step - X is a 6 x 3 matrix
attn.score <- torch::torch_mm(inputs, inputs$transpose(1, 2)) # A = X %*% X^T
## normalize by row (2nd dimension)
(attn.wghts <- torch::nn_softmax(2)(attn.score))                  # W = softmax(A), each row sums to 1
```

```
## torch_tensor
##  0.2098  0.2006  0.1981  0.1242  0.1220  0.1452
##  0.1385  0.2379  0.2333  0.1240  0.1082  0.1581
##  0.1390  0.2369  0.2326  0.1242  0.1108  0.1565
##  0.1435  0.2074  0.2046  0.1462  0.1263  0.1720
##  0.1526  0.1958  0.1975  0.1367  0.1879  0.1295
##  0.1385  0.2184  0.2128  0.1420  0.0988  0.1896
## [ CPUFloatType{6,6} ]
```

```
(outputs <- torch::torch_mm(attn.wghts, inputs)) # W %*% X
```

```
## torch_tensor
##  0.4421  0.5931  0.5790
##  0.4419  0.6515  0.5683
##  0.4431  0.6496  0.5671
##  0.4304  0.6298  0.5510
##  0.4671  0.5910  0.5266
##  0.4177  0.6503  0.5645
## [ CPUFloatType{6,3} ]
```

Self-Attention with Weights

This replicates the code in Section 3.4. The `nn_linear` function returns a *function* with optimized initial weights and performs the linear transformation (multiplication) efficiently.

```
d.in <- d.out <- ncol(inputs)
W.query <- torch::nn_linear(d.in, d.out, bias = FALSE)
W.key   <- torch::nn_linear(d.in, d.out, bias = FALSE)
W.value <- torch::nn_linear(d.in, d.out, bias = FALSE)
self.attn.score <- torch::torch_mm(W.query(inputs), W.key(inputs)$transpose(1, 2)) # As = X W_Q %*% (X W_K)^T
(self.attn.wghts <- torch::nn_softmax(2)(self.attn.score / sqrt(d.out))) # Aw = softmax(A)
```

```
## torch_tensor
##  0.1637  0.1623  0.1620  0.1737  0.1626  0.1757
##  0.1594  0.1630  0.1629  0.1743  0.1651  0.1752
##  0.1595  0.1633  0.1632  0.1739  0.1656  0.1745
##  0.1624  0.1644  0.1643  0.1715  0.1651  0.1724
##  0.1625  0.1686  0.1690  0.1646  0.1742  0.1611
##  0.1615  0.1619  0.1615  0.1757  0.1607  0.1789
## [ CPUFloatType{6,6} ][ grad_fn = <SoftmaxBackward0> ]
```

```
(outputs <- torch::torch_mm(self.attn.wghts, W.value(inputs))) # Aw %*% W_V X
```

```
## torch_tensor
## -0.3640  0.0766  0.2043
## -0.3647  0.0764  0.2053
## -0.3648  0.0761  0.2054
## -0.3648  0.0751  0.2052
## -0.3657  0.0689  0.2069
## -0.3643  0.0786  0.2044
## [ CPUFloatType{6,3} ][ grad_fn = <MmBackward0> ]
```

Causal Attention

- Casual (masked) attention restricts a model to only consider correlation with *previous* tokens, by multiplying the attention weights with a mask and re-normalize.

```
mask <- torch::torch_tril(torch::torch_ones(nrow(inputs), nrow(inputs)))
```

- Alternatively, apply the mask first on the attention scores and re-normalize (`-Inf` is treated as 0 by softmax function).

```
(casual.attn.score <- self.attn.score$masked_fill(mask < 0.5, -Inf))
```

```
## torch_tensor
## -0.1961    -inf    -inf    -inf    -inf    -inf
## -0.2356 -0.1969    -inf    -inf    -inf    -inf
## -0.2259 -0.1853 -0.1868    -inf    -inf    -inf
## -0.1446 -0.1233 -0.1248 -0.0503    -inf    -inf
##  0.0126  0.0762  0.0804  0.0341  0.1322    -inf
## -0.2466 -0.2426 -0.2469 -0.1008 -0.2552 -0.0694
## [ CPUFloatType{6,6} ][ grad_fn = <MaskedFillBackward0> ]
```

```
(casual.attn.wghts <- torch::nn_softmax(2)(casual.attn.score / sqrt(d.out)))
```

```
## torch_tensor
##  1.0000  0.0000  0.0000  0.0000  0.0000  0.0000
##  0.4944  0.5056  0.0000  0.0000  0.0000  0.0000
##  0.3282  0.3360  0.3357  0.0000  0.0000  0.0000
##  0.2451  0.2481  0.2479  0.2588  0.0000  0.0000
##  0.1938  0.2010  0.2015  0.1962  0.2076  0.0000
##  0.1615  0.1619  0.1615  0.1757  0.1607  0.1789
## [ CPUFloatType{6,6} ][ grad_fn = <SoftmaxBackward0> ]
```

Dropout

- Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively “dropping” them out.
- This helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units.
- Dropout is only used during training and is disabled afterward.

```
dropout <- torch::nn_dropout(p = 0.5) # drop 50%
dropout(casual.attn.wghts)
```

```
## torch_tensor
##  2.0000  0.0000  0.0000  0.0000  0.0000  0.0000
##  0.9888  1.0112  0.0000  0.0000  0.0000  0.0000
##  0.6565  0.0000  0.6715  0.0000  0.0000  0.0000
##  0.0000  0.0000  0.4959  0.0000  0.0000  0.0000
##  0.0000  0.4020  0.0000  0.3923  0.0000  0.0000
##  0.3230  0.0000  0.3229  0.0000  0.3214  0.3577
## [ CPUFloatType{6,6} ][ grad_fn = <MulBackward0> ]
```

- Note that the remaining weights are scaled by a factor of $1/p$.

Multi-Head Attention

- The term “multi-head” refers to dividing the attention mechanism into multiple “heads,” each with its own set of (smaller) weight matrices (parameters), operating independently.
  - Computational efficiency: smaller weight matrices to deal with and can be optimized in batches
  - Specialization: each head is allowed to “learn” a different aspect of the sequence — syntax, semantics, coreference, etc.
- In practical terms, this involves creating multiple instances of the self-attention mechanism (input of length d, output of length d_k, where commonly $d_k = d/H$) each with its own weight matrices (Q, K, V), and then combining the outputs into a vector of length $H \times d_k$.
- Or more efficiently, we create the query, key and value tensors of dimension $d \times (H \cdot d_k)$ through one matrix multiplication and then split each of them into a tensor of $H \times d \times d_k$ for use in subsequent calculations.

Example of Tensor Multiplication

Here is an example of tensor multiplication which is carried out on the last two dimensions. Multiplying a tensor of size $1 \times 2 \times 4 \times 2$ with another of size $1 \times 2 \times 2 \times 4$ yields a tensor of size $1 \times 2 \times 2 \times 2$.

```
aa
```

```
## torch_tensor
## (1,1,.,.) =
##   0.2745  0.8993  0.0772  0.4066
##   0.2775  0.9268  0.1479  0.4545
##
## (1,2,.,.) =
##   0.6584  0.0390  0.3565  0.2318
##   0.8573  0.7388  0.5331  0.9737
## [ CPUFloatType{1,2,2,4} ]
```

```
torch::torch_matmul(aa, aa$transpose(3, 4))
```

```
## torch_tensor
## (1,1,.,.) =
##   1.0554  1.1059
##   1.1059  1.1644
##
## (1,2,.,.) =
##   0.6158  1.0090
##   1.0090  2.5131
## [ CPUFloatType{1,2,2,2} ]
```

Layer Normalization

- Layer normalization is used to improve numerical stability by adjusting the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1.
- In GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module, and before the final output layer.

```
batch.ex <- torch::torch_randn(2, 5)                                    # tensor of 2 x 5
layer.ex <- torch::nn_sequential(torch::nn_linear(5, 6), torch::nn_relu()) # one linear layer (5 x 6) + RELU activation function
(out.ex <- layer.ex(batch.ex))                                          # 2 x 6 tensor
```

```
## torch_tensor
##  0.1025  0.0176  1.6049  0.0503  0.0000  0.8693
##  0.0000  0.0073  1.0538  0.0000  0.0000  0.1109
## [ CPUFloatType{2,6} ][ grad_fn = <ReluBackward0> ]
```

```
(out.ex - out.ex$mean(-1, # operating on the last dimension
                      keepdim = TRUE # output a 2 x 1 tensor, instead of dropping to a vector of length 2
                      ))/(sqrt(out.ex$var(-1, keepdim = TRUE,
                                          unbiased = FALSE # use n as denominator instead of unbiased n-1, this was used in GPT-2
                                            # and is TensorFlow's default behavior, doesn't matter much when n is large
                                          ) + 1e-5)) # a small bias to avoid division by zero
```

```
## torch_tensor
## -0.5613 -0.7022  1.9317 -0.6479 -0.7314  0.7111
## -0.5060 -0.4872  2.2240 -0.5060 -0.5060 -0.2188
## [ CPUFloatType{2,6} ][ grad_fn = <DivBackward0> ]
```

Activation Functions

- Activation functions introduce non-linearity into a neural network and are used after each layer. A different activation function (e.g., softmax) may be used after the final output layer.
- Common activation functions:

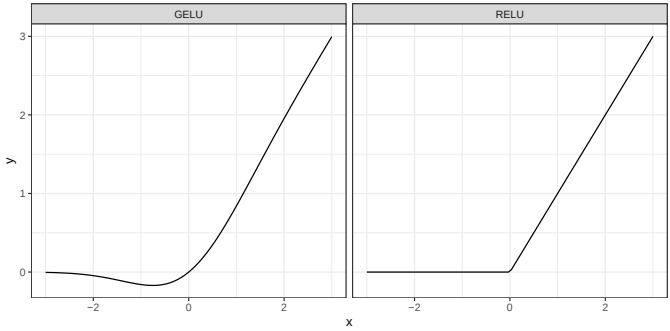
$$\begin{aligned}
\text{RELU (Rectified Linear Unit)} \quad & f(x) = \max(0, x) \\
\text{GELU (Gaussian error linear unit)} \quad & f(x) = x\Phi(x) \\
& \approx \frac{1}{2}x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.0044715 \cdot x^3\right)\right]\right)
\end{aligned}$$

  $\Phi()$ is the CDF of normal distribution, and an approximation is used in the original GPT-2.
- The activation function is always applied element-wise to the input tensor. Note each element is often referred as a “neuron”.

GELU vs RELU

- GELU is similar to RELU but is a smooth function.

```
x <- seq(from = -3, to = 3, length = 100)
exdata <- rbind(data.frame(f = "GELU", x, y = as.numeric(torch::nn_gelu()(x))),
                data.frame(f = "RELU", x, y = as.numeric(torch::nn_relu()(x))))
```



Shortcut Connection

- Shortcut connection adds the input of a layer (function) to its output (i.e., $y = f(x) + x$), which effectively adds 1 to the gradient (derivative) of the layer.
- It is designed to overcome the vanishing gradient problem, where the gradients become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers.

Transform Block

- A *feed forward* layer is a small network that can be easily stacked to form more layers, because its input and output have the same dimension while at the same time allows more complexity in between.
  1. a linear layer that expands the embedding dimensions
  2. a nonlinear GELU activation function
  3. another linear layer that contracts the embedding dimensions to match the original input.
- A *transformer* block combines multi-head attention, layer normalization, dropout, feed forward layers, and GELU activations. It is the basic building block of a LLM and is repeated 12 times in the 124-million-parameter GPT-2.

Implementing a Transformer Block

```
x <- input
shortcut <- x
x <- layer_norm(x)
x <- multi_head_attention(x)
x <- dropout(x)
x <- x + shortcut
shortcut <- x
x <- layer_norm(x)
x <- nn_sequential(nn_linear(d.in = emb.d, d.out = 4 * emb.d),
                   GELU(),
                   nn_linear(d.in = 4 * emb.d, d.out = emb.d))(x) # feed forward
x <- dropout(x)
output <- x + shortcut
```

```
layer_norm <- torch::nn_module(
    classname = "layer_norm",
    initialize = function(emb.dim) {
        self$eps <- 1e-5
        self$scale <- torch::nn_parameter(torch::torch_ones(emb.dim))
        self$shift <- torch::nn_parameter(torch::torch_zeros(emb.dim))

    },
    forward = function(x) {
        mean <- x$mean(dim = -1, keepdim = TRUE)
        var <- x$var(dim = -1, keepdim = TRUE, unbiased = FALSE)
        norm.x <- (x - mean) / torch::torch_sqrt(var + self$eps)
        self$scale * norm.x + self$shift
    }
)
ln <- layer_norm(5)
```