Build GPT from Scratch

Na (Michael) Li, Ph.D.

31 March, 2025

Introduction

This notebook uses R to implement the LLM in the book Build a LLM from Scratch. The book uses Python and PyTorch. With torch in R, it is fairly straightforward to translate the codes. Besides familiarity, using R has another advantage in having fewer dependencies to worry out. With Python, one would be install another copy of Python (newer that the one shipped with Macs), myriads of other packages and worry about managing "virtual environment". It is also easier to run R in Emacs with Rmarkdown.

```
install.packages("torch")
torch::install_torch()
```

A "module" (an object of class `nn_module` in Pytorch) in deep learning parlance is simply a mathematical function on a tensor input x. Upon creation (`initalize()`), fixed parameters (hyper-parameters, such as the dimension of the input) are set, and trainable parameters (weights) are initialized with random numbers. The `forward()` method defines the output of the function $f(x)$, and the `backward()` defines the gradient $f'(x)$ used in optimization. (back-propagation).

Note that R uses FORTRAN-style indexing (starting from 1) while Python follows C-style (starting from 0).

Some test data - storing the

```
library(torch)
```

```
## Warning: package 'torch' was built under R version 4.4.1
```

```
inputs <- 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
```

Stacking two 6×3 tensors at the 1st dimension to create a $2 \times 6 \times 3$ tensor.

```
batch <- torch_stack(c(inputs, inputs), dim = 1L)
batch
```

```
## torch_tensor
## (1,.,.) =
##   0.4300  0.1500  0.8900
##   0.5500  0.8700  0.6600
##   0.5700  0.8500  0.6400
##   0.2200  0.5800  0.3300
##   0.7700  0.2500  0.1000
##   0.0500  0.8000  0.5500
##
## (2,.,.) =
##   0.4300  0.1500  0.8900
##   0.5500  0.8700  0.6600
##   0.5700  0.8500  0.6400
##   0.2200  0.5800  0.3300
##   0.7700  0.2500  0.1000
##   0.0500  0.8000  0.5500
## [ CPUFloatType{2,6,3} ]
```

Transformer Block

Multi-Head Attention

```
multi_head_attention <-
    nn_module(
        classname = "multi_head_attention",
        initialize = function(d.in,         # input dimension
                                        # output dimension, must be divisible by num.head
                              d.out,
                              num.heads, #
                              context.len,
                              dropout, # droput rate, between 0 and 1
                              qkv.bias = FALSE) {
            stopifnot(d.out %% num.heads == 0)
            self$d.out <- d.out
            self$n.heads <- num.heads
            self$head.d <- d.out %/% num.heads
            ## three weight matrices Q, K, and V
            self$W.query <- nn_linear(d.in, d.out, bias = qkv.bias)
            self$W.key   <- nn_linear(d.in, d.out, bias = qkv.bias)
            self$W.value <- nn_linear(d.in, d.out, bias = qkv.bias)
            ## a linear layer to combine outputs
            self$out.proj <- torch::nn_linear(d.out, d.out)
            ## dropout
            self$dropout <- torch::nn_dropout(dropout)
            ## causal attention (mask) - convert to bool type
            self$mask <- torch_triu(torch_ones(context.len, context.len),
                                    diag = 1) > 0

           },
           forward = function(x) {
               b <- x$shape[1]
               n.tokens <- x$shape[2]
               d.in <- x$shape[3]

               keys <- self$W.key(x)
               queries <- self$W.query(x)
               values <- self$W.value(x)

               ## "unroll" the last dimension d.out to a matrix
               ## [b, n.tokens, d.out] -> [b, n.tokens, n.heads, head.d]
               keys    <-    keys$view(c(b, n.tokens, self$n.heads, self$head.d))
               queries <- queries$view(c(b, n.tokens, self$n.heads, self$head.d))
               values  <-  values$view(c(b, n.tokens, self$n.heads, self$head.d))

               ## transpose -> [b, n.heads, n.tokens, head.d]
               ## R index starts from 0,
               keys <- keys$transpose(2, 3)
               queries <- queries$transpose(2, 3)
```

```
                values <- values$transpose(2, 3)
                ## multiplication on the last two dimensions [n.tokens, head.d]
                ## -> [b, n.heads, n.tokens, n.tokens]
                attn.scores <- torch_matmul(queries, keys$transpose(3, 4))
                ## apply mask on the last two dimensions
                mask.bool <- self$mask[1:n.tokens, 1:n.tokens]
                attn.scores$masked_fill_(mask.bool, -Inf)
                attn.weights <- nn_softmax(-1)(attn.scores / sqrt(self$head.d))
                attn.weights <- self$dropout(attn.weights)

                ## -> [b, n.heads, n.tokens, head.d]
                ## -> [b, n.tokens, n.heads, head.d]
                context.vec <- torch_matmul(attn.weights, values)$transpose(2, 3)
                ## flatten -> [b, n.tokens, d.out = n.heads * head.d]
                context.vec <- context.vec$contiguous()$view(c(b, n.tokens, self$d.out))
                context.vec <- self$out.proj(context.vec)
                context.vec
            })
```

To replicate the results from the book (Chapter 3, page 90), we have to use CPU because the random seed can't be set with MPS device.

```
torch_manual_seed(123)
mha <- multi_head_attention(d.in = batch$shape[3], d.out = 2,
                            num.heads = 2, context.len = batch$shape[2], dropout = 0.0)
context.vecs <- mha(batch)
context.vecs
```

```
## torch_tensor
## (1,.,.) =
##   0.3190  0.4858
##   0.2943  0.3897
##   0.2856  0.3593
##   0.2693  0.3873
##   0.2639  0.3928
##   0.2575  0.4028
##
## (2,.,.) =
##   0.3190  0.4858
##   0.2943  0.3897
##   0.2856  0.3593
##   0.2693  0.3873
##   0.2639  0.3928
##   0.2575  0.4028
## [ CPUFloatType{2,6,2} ][ grad_fn = <ViewBackward0> ]
```

Normalization Layer

Normalize along the last dimension, subtract the mean and divide by the standard deviation. The biased estimate of variance (divided by n instead of $n - 1$) is used for historical reasons.

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

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

Testing (Chapter 3, page 100):

```r
torch_manual_seed(123)
ex.tmp <- torch_randn(c(2, 5))
layer.tmp <- nn_sequential(nn_linear(5, 6), nn_relu())
(out.tmp <- layer.tmp(ex.tmp))
```

```
## torch_tensor
##  0.2260  0.3470  0.0000  0.2216  0.0000  0.0000
##  0.2133  0.2394  0.0000  0.5198  0.3297  0.0000
## [ CPUFloatType{2,6} ][ grad_fn = <ReluBackward0> ]
```

```r
ln <- layer_norm(out.tmp$shape[2])
ln(out.tmp)
```

```
## torch_tensor
##  0.6745  1.5470 -0.9549  0.6431 -0.9549 -0.9549
## -0.0207  0.1228 -1.1913  1.6619  0.6186 -1.1913
## [ CPUFloatType{2,6} ][ grad_fn = <AddBackward0> ]
```

Feed Forward

A *feed forward* module is a small neural network consisting of two Linear layers and a GELU activation function. Its input and output have the same dimensions but the weights are bigger for "exploration of a richer representation space" (whatever that means).

```r
feed_forward <-
    nn_module(
        classname = "feedforward",
        initialize = function(emb.dim) {
            self$layers <- nn_sequential(nn_linear(emb.dim, 4 * emb.dim),
                                         nn_gelu(),
                                         nn_linear(4 * emb.dim, emb.dim))
        },
        forward = function(x) self$layers(x)
    )
```

```r
ffn <- feed_forward(out.tmp$shape[2])
ffn(out.tmp)
```

```
## torch_tensor
##  0.0944 -0.2923  0.0569 -0.0659  0.1458  0.0304
##  0.0294 -0.2923  0.1286 -0.0818  0.1628  0.0571
## [ CPUFloatType{2,6} ][ grad_fn = <AddmmBackward0> ]
```

Transformer Block

Putting everything together, a *transformer* module combines multi-head attention, layer normalization, dropout, feed forward layers, and GELU activation. It is the basic building block of LLMs and is repeated 12 times in the 124-million-parameter GPT-2. This transformer model (*architecture*) is what distinguished

LLMs from earlier deep neural network models such as convolutional (CNN) and recurrent (RNN) neural networks.

Note that in a transformer, the shape of the output is the same as that of the input. The preservation of shape throughout the transformer block architecture is a crucial aspect of its design

First put the hyper-parameters (GPT-2) into a dictionary (a list is used to storage values of different storage types).

```r
GPT2.config <- list(vocab_size = 50257L, # Vocabulary size, used by the BPE tokenizer
                    context_length = 1024L, # Context Length
                    emb_dim = 768L, # Embedding dimension
                    num_heads = 12L,  # Number of attention heads
                    num_layers = 12L, # Number of (Transformer) Layers
                    drop_rate =  0.1, # Dropout rate
                    qkv_bias = FALSE # Quary-Key-Value bias
                    )
```

Transformer module (Chapter 4, page 115):

```r
transformer_block <- nn_module(
    classname = "transformer_block",
    initialize = function(cfg) {
        self$attn = multi_head_attention(d.in = cfg[["emb_dim"]],
                                         d.out = cfg[["emb_dim"]], # same dimension output
                                         num.heads = cfg[["num_heads"]],
                                         context.len = cfg[["context_length"]],
                                         dropout = cfg[["drop_rate"]],
                                         qkv.bias = cfg[["qkv_bias"]])
        self$ff = feed_forward(emb.dim = cfg[["emb_dim"]])
        self$norm1 = layer_norm(emb.dim = cfg[["emb_dim"]])
        self$norm2 = layer_norm(emb.dim = cfg[["emb_dim"]])
        self$drop = nn_dropout(cfg[["drop_rate"]])

    },
    forward = function(x) {
        shortcut <- x # shortcut connection
        x <- self$norm1(x)
        x <- self$attn(x)
        x <- self$drop(x)
        x <- x + shortcut

        shortcut <- x
        x <- self$norm2(x)
        x <- self$ff(x)
        x <- self$drop(x)
        x <- x + shortcut
        x
    })
```

Testing

```r
x <- torch_randn(2, 4, GPT2.config[["emb_dim"]])
block <- transformer_block(GPT2.config)
output <- block(x)
rbind("input shape" = x$shape, "output shape" = output$shape)
```

```
##              [,1] [,2] [,3]
## input shape     2    4  768
## output shape    2    4  768
```

The GPT Model

Finally the GPT model is simply made up initial embedding layers, multiple layers of transformers, and final output layer.

```r
gpt_model <- nn_module(
    classname = "gpt_model",
    initialize = function(cfg) {
        emb.d <- cfg[["emb_dim"]]
        self$tok.emb <- nn_embedding(cfg[["vocab_size"]], emb.d)
        self$pos.emb <- nn_embedding(cfg[["context_length"]], emb.d)
        self$drop.emb <- nn_dropout(cfg[["drop_rate"]])
        ## a trick to allow variable number of layers
        self$tfr.blocks <- do.call("nn_sequential",
                                   lapply(seq(cfg[["num_layers"]]),
                                          function(x) transformer_block(cfg)))
        self$final.norm <- layer_norm(emb.d)
        self$out.head <- nn_linear(emb.d,  cfg[["vocab_size"]], bias = FALSE)

    },
    forward = function(idx) { # a matrix where each row is a vector of tokens (integer)
        batch.size <- idx$shape[1]
        seq.len <- idx$shape[2]
        tok.embeds <- self$tok.emb(idx)
        pos.embeds <- self$pos.emb(torch_arange(seq.len))
        x <- tok.embeds + pos.embeds # embedding
        x <- self$drop.emb(x)        # dropout
        x <- self$tfr.blocks(x)      # transformer blocks
        x <- self$final.norm(x)      # final normalization
        self$out.head(x)            # output
    })
```

Test the GPT model (Chapter 4, page 120)

```r
txt <- c("Every effort moves you", "Every day holds a")
(batch <- torch_tensor(do.call("rbind", rtiktoken::get_tokens(txt, "gpt2")), dtype = "long"))
```

```
## torch_tensor
##  6109  3626  6100   345
##  6109  1110  6622   257
## [ CPULongType{2,4} ]
```

```r
torch_manual_seed(123)
model <- gpt_model(GPT2.config)
out <- model(batch)
out
```

```
## torch_tensor
## (1,.,.) =
##  Columns 1 to 6  1.4593e+00  3.1971e-01  6.0224e-01  3.3625e-01  3.7032e-01 -6.6830e-01
##  -2.0472e-01  1.6462e-01 -4.0896e-01  9.9220e-02 -6.5007e-02  8.0615e-02
##   8.1893e-01  7.8078e-01  3.0777e-01  3.7483e-01  4.5006e-03  4.1724e-01
##  -8.5622e-01  8.6768e-03  1.0867e-01 -9.7716e-02 -2.1040e-01  8.1199e-02
##
## Columns 7 to 12 -2.2592e-01  6.4881e-01  5.3089e-01  2.5143e-01  4.5661e-02 -1.5140e-01
##  -3.2609e-01 -4.2830e-01  1.3508e+00  1.1869e+00 -9.2019e-02  1.7794e-01
##  -1.0332e-01  6.9154e-01  8.4409e-04  1.2893e+00 -5.0486e-01  9.4459e-01
##   9.6265e-02  3.1245e-01  1.1621e-01  7.0639e-01  5.9318e-01  3.4117e-01
##
## Columns 13 to 18  4.9941e-01  5.1920e-01  6.4758e-01  1.0130e+00 -2.4650e-01  1.6234e+00
##  -2.6004e-02 -1.1146e+00  4.6488e-01  6.4222e-01 -8.5051e-01  4.7130e-01
##   1.0719e-01 -1.8617e-01 -3.9632e-01 -3.4621e-02 -1.0718e-02 -8.7991e-01
##  -3.2673e-01  2.0255e-01  1.5022e-02 -2.6259e-01  2.5709e-01  4.0163e-01
##
## Columns 19 to 24  1.4839e+00  8.3255e-03  5.8839e-01 -4.4515e-01  9.7905e-02 -8.5936e-01
##   1.0204e-01  3.4152e-02  1.3470e-01 -1.0938e+00 -4.6612e-01  2.2549e-01
##   7.9313e-01  6.3839e-01  1.9227e-01 -5.3041e-02  2.3833e-01 -1.6479e-01
##  -4.5306e-01  1.0328e+00 -9.0633e-01 -9.6356e-01 -2.8546e-01 -1.3975e-01
##
## Columns 25 to 30 -5.6907e-01  4.0515e-01 -3.9161e-01 -6.1416e-01  1.6509e-01  9.4325e-01
##  -7.4596e-01 -4.4881e-01  9.1341e-01 -7.4767e-01  3.9105e-01  5.7092e-01
##  -1.7102e-01  2.9709e-01  4.3593e-01  2.4054e-01  2.5153e-01  1.0420e-01
##  -1.8727e+00  1.7983e-01 -4.3142e-01 -1.4966e-02  5.9813e-01  4.2204e-01
##
## Columns 31 to 36 -1.1197e-01  6.7174e-01 -8.2168e-01 -8.1264e-01  3.2899e-01  3.8802e-02
##   1.9537e-01  5.2022e-01 -4.8574e-01 -3.8242e-01 -7.2619e-03  9.1579e-01
##  -8.3569e-02  4.7754e-01  1.2863e+00  2.8164e-01 -9.3445e-01  2.7093e-01
##  -6.8711e-01  5.6206e-01 -4.1818e-01 -5.7161e-01 -7.1047e-01  8.8437e-01
## ... [the output was truncated (use n=-1 to disable)]
## [ CPUFloatType{2,4,50257} ][ grad_fn = <UnsafeViewBackward0> ]
```

The total number of parameters in the model. In GPT-2 the output layer reuses the same weights from the token embedding layer hence has fewer parameters.

```r
total.params <- sum(sapply(model$parameters, function(x) x$numel()))
cat("Total number of parameters is: ", scales::label_comma()(total.params), "\n")
```

```
## Total number of parameters is:  163,009,536
```

```r
gpt2.params <- total.params - model$parameters$out.head.weight$numel()
cat("Total number of parameters in GPT-2 is: ", scales::label_comma()(gpt2.params), "\n")
```

```
## Total number of parameters in GPT-2 is:  124,412,160
```

Generating Text

To generate text, the output vector is converted to multinomial probabilities by a softmax function, and this simple function returns the token with the highest probability.

```r
simple.text.generator <- function(model,
                                  idx, # (batch, n_tokens)
                                  max.new.tokens, context.size) {
    for (i in seq(max.new.tokens)) {
        n <- ncol(idx)
        idx.cond <- idx[, min(1, (n-context.size)):n] # lacking Pythong's shortcut [:, -context.size:]
        with_no_grad(logits <- model(idx.cond))  # (batch, n_tokens, vocab_size)
        logits <- logits[,logits$shape[2],] # (batch, vocab_size)
        probs <- nn_softmax(-1)(logits)     # (batch, vocab_size)
        idx.next <- torch_argmax(probs, dim = -1, keepdim = TRUE)
        ## adding the predicted token to the list of tokens
        idx <- torch_cat(c(idx, idx.next), dim = 2)
    }
    idx
}
```

Testing generating text from the GPT model (Chapter 4, page 126)

```r
start.context <- "Hello, I am"
(encoded <- rtiktoken::get_tokens(start.context, model = "gpt2"))
```

```
## [1] 15496    11   314   716
```

```
(encoded.tensor = torch_tensor(encoded, dtype = "long")$unsqueeze(1)) # -> 1 x 4 tensor
```

```
## torch_tensor
##  15496     11    314    716
## [ CPULongType{1,4} ]
```

```
model$eval()                                                    # no dropout in 'eval' mode
(out <- simple.text.generator(model, encoded.tensor, 6,
                              context.size = GPT2.config[["context_length"]]))
```

```
## torch_tensor
##  15496     11    314    716  13008  49330  41978   4272   9914  19960
## [ CPULongType{1,10} ]
```

The decoded text is gibberish since the model has not been trained yet.

```
(decoded.text = rtiktoken::decode_tokens(out, model = "gpt2"))
```

```
## [1] "Hello, I am wallet resided brochalingCar tended"
```