A note on more efficient architectures for NLP

Arturs Backurs Mingda Chen Kevin Gimpel

Abstract

We describe simple memory-efficient architectures for NLP tasks.

1 FFT-based model for natural language processing

Consider the sentence "The quick brown fox jumps over the lazy dog". Suppose that our goal is to teach a machine learning model to understand human written language. One way to do it would be to blank out a random word from the sentence and ask the model to predict the missing word. For instance, we might blank out the word "jumps" from the sentence and present the following input to the machine learning model: "The quick brown fox _ over the lazy dog". The model would hopefully predict that the missing word is "jumps" from the rest of the given context. We would feed a lot of fragments of texts to the machine learning model with random subsets of words blanked out and the model would (hopefully) eventually build an understanding of written language. Typically the length (number of words) of a fragment of text fed to a model is fairly large, say, 1024. Let's denote the length of the fragments by n, which is 1024 in our case. Out of the n words, 15% of them get blanked before the fragment is being fed to the model and asked to recover the missing words. Ideally we would like that n is as large as possible so that the model learns to "reason" about texts where some parts of it depends on other parts far away. Unfortunately, for many models, the memory consumption grows quadratically in n, which is impractical as n grows large. Below we describe a simple model with a memory consumption that has a near-linear dependency on n.

For the sake of concreteness, we set n to be large integer, say, $n = 16384$. The model consists of three phases.

Phase 1: embedding We replace every word of the input sentence by a d-dimensional vector. For the sake of concreteness, let $d = 768$. For every word of the English dictionary we maintain a d-dimensional vector. The entries of the vectors are initialized as independent Gaussians but later are learned. Consider the input text "The quick brown fox _ over the lazy dog" (in this example $n = 9$). The text is replaced by a sequence of n vectors

$$v_{\text{The}}, v_{\text{quick}}, v_{\text{brown}}, v_{\text{fox}}, v_{\_}, v_{\text{over}}, v_{\text{the}}, v_{\text{lazy}}, v_{\text{dog}}.$$

Notice that, if the text contains repeating words, then the sequence of vectors contains repeating vectors. Furthermore, notice that there is a special vector corresponding to blanked-out words. An issue with this embedding is that, if D denotes the number of possible distinct words in the English vocabulary, the total number of parameters corresponding to this phase is Dd, which is too much. One way to get around is to split the words into word-pieces - commonly occurring parts of words such that most words can be assembled from a small number of word-pieces. This allows to reduce D to roughly $D = 32000$ and the number of parameters Dd becomes tractable.

Phase 2: contextual embedding This phase consists of l layers. For the sake of concreteness we set $l = 64$. The layers are identical but they don't share the learnable parameters. A single layer proceeds as follows. First, we concatenate all vector into a single nd-dimensional vectors. Next, we apply FFT transform to the resulting vector. PyTorch has an $O(nd \log(nd))$-time implementation of FFT transform and it can be backpropagated through. The backpropagation stores $O(nd \log(nd))$ intermediate values and consumes $O(nd \log(nd))$ memory. This part of the layer doesn't have any learnable parameters. The second part of the layer splits the nd-dimensional vector into n d-dimensional vectors $v_1, \ldots, v_n$ and applies a two layer simple neural network for every vectors. That is, we set

$$v_i \leftarrow B \operatorname{ReLU}(Av_i)$$

for every $i = 1, \ldots, n$, where $A, B \in \mathbb{R}^{d \times d}$ are two learnable linear transformations with entries initialized with independent Gaussians. This finishes the description of one layers. All layers are identical except they don't share the linear transformation matrices A and B.

Phase 3: loss computation After applying the l layers, we obtain n vectors $v_1, \ldots, v_n$. To compute the loss, we learn D vectors $w_1, \ldots, w_D \in \mathbb{R}^d$. D is the number of word-pieces as above and the vectors are initialized as independent Gaussians. Suppose that the i-th word-piece in the original input was a blank and that our goal is to determine what we should replace the blank with. One way to do that is to replace blank with the j word-piece where $j \in \arg\max_{j=1,\ldots,D} w_j^\dagger v_i$. This is the intuition behind the loss computation procedure. First, we compute a probability distribution $p_{i,1}, \ldots, p_{i,D}$ over the word-pieces using soft-max:

$$p_{i,j} = \exp(w_j^\dagger v_i) / \sum_i \exp(w_j^\dagger v_i).$$

Let $j^*(i)$ be the word-piece that was in the i-th position before we blanked it out. We want that $p_{i,j^*(i)}$ is as large as possible (ideally $p_{i,j^*(i)} = 1$). One way to express this objective is as trying to minimize $\log(1/p_{i,j^*(i)})$. The total loss is the sum of terms $\log(1/p_{i,j^*(i)})$ over all i such that the i-th word-piece in the input was blanked-out. We do gradient descent to minimize the total loss.

Number of parameters Phases 1 and 3 has Dd parameters each. Phase 2 has $2d^2$ parameters per each one of the l layers. So the total number of parameters is $2Dd+2d^2l = O(Dd+d^2l)$. With our choice of D, d and l we get that the number of parameters is $2Dd+2d^2l = 2 \cdot 32000 \cdot 768 + 2 \cdot 768^2 \cdot 64 = 49152000 + 75497472 = 124649472$.

Memory consumption To be able to do the backpropagation, we need to store all the intermediate computation steps. For this model the memory complexity is the same as the time complexity to evaluate the total loss according to the above description. For phase 1 the memory consumption is $O(nd)$. For phase 2 the memory consumption is $O(lnd \log(nd)) + O(lnd^2)$. For the third phase the memory consumption is $O(ndD)$. So the total memory consumption is $O(lnd \log(nd) + lnd^2 + ndD)$.

We note that the memory complexity of the model has a near linear dependency on n. This is in contrast to the typical quadratic dependency on n. There are works that achieve similar near-linear dependency [TDBM20] but the resulting models are significantly more complex. In Appendix A we add a complete implementation of the described model in Python.

Experiments We trained a variant of the model (see Appendix A) on the WikiText-103 data set for 1 billion tokens (around 9 epochs). The model has context length $n = 64$, $l = 12$ layers, dimension $d = 768$ and vocabulary size $D = 32000$. The total number of parameters is $105,871,628$. We used batch size of 64 and Adam optimizer with weight decay (weight decay coefficient 0.01, $\beta_1 = 0.9$, $\beta_2 = 0.999$). We set learning rate to 0.001 with a linear learning rate decay and no warm-up. We finetuned the model on the SST-2 task from the GLUE benchmark and it achieved the accuracy 86.7. We can compare this to the GLUE benchmark leaderboard [glu].

2 Faster models for text generation

For this model we again have an input text sequence of, say, $n = 16384$ word-pieces. Similarly as before, the model processes the input sequence in 3 phases.

Phase 1: embedding We replace every word-piece of the input sentence by a d-dimensional vector. For an example, the input text "The quick brown fox jumps over the lazy dog" is replaced by a sequence of n vectors

$$v_{\text{The}}, v_{\text{quick}}, v_{\text{brown}}, v_{\text{fox}}, v_{\text{jumps}}, v_{\text{over}}, v_{\text{the}}, v_{\text{lazy}}, v_{\text{dog}}.$$

Notice that there are no blanked-out word-pieces in the input sequence unlike for the previously described FFT-based model.

Phase 2: contextual embedding This phase consists of l layers. For the sake of concreteness we set $l = 64$. The layers are identical but they don't share the learnable parameters. A single layer proceeds as follows. First, we replace every vector by the sum of the first i vectors. That is, $v_i \leftarrow \sum_{j=1}^{i} v_j$ for $i = 1, \ldots, n$. This part of the layer doesn't have any learnable parameters. The second part of the layer applies a two layer simple neural network for every vector. That is, for every $i = 1, \ldots, n$, we set

$$v_i \leftarrow B\,\mathrm{ReLU}(Av_i),$$

where $A, B \in \mathbb{R}^{d \times d}$ are two learnable linear transformations. A very important observation about the layers is that the i-th output vector of every layer depends only on the first i input vectors to the layer.

Phase 3: loss computation After applying the l layers, we obtain n vectors $v_1, \ldots, v_n$. We note that for every $i = 1, \ldots, n$, the vector v_i depends only on the first i input vectors to the model (the first i input vectors to the first layer). This follows from the observation from the previous phase.

Our goal is that for every $i = 1, \ldots, n$, the vector v_i (which depends only on the first i word-pieces), predicts, the $(i+1)$-st word-piece. In particular, for $i = n$ we want to predict the $(n+1)$-st word-piece, which is not part of the input sentence. However, we assume that the n input word-pieces come from a larger text and we can read the $(n+1)$-st word-piece from the larger text. Let $p_{i,j}$ be as defined in the description of phase 3 of the FFT-based model. Let $j^*(i)$ be the word-piece that is in the $(i+1)$-st position for $i = 1, \ldots, n$. Then our goal is to minimize $\log(1/p_{i,j^*(i)})$ for all i. We define the loss function $\sum_{i=1}^{n} \log(1/p_{i,j^*(i)})$ and minimize this loss using gradient descent.

Number of parameters and memory consumption An easy computation shows that the number of parameters is $2Dd+2d^2l = O(Dd+d^2l)$ and that the memory (and the time) complexity of the model is $O(lnd^2 + ndD)$.

Weighted sum variant of the model For every $i = 1, \ldots, n$, the current model does the $v_i \leftarrow \sum_{j=1}^{i} v_j$. A natural idea is to introduce learnable parameters $w_1, \ldots, w_n$ and make the update rule to be $v_i \leftarrow \sum_{j=1}^{i} v_j w_{i+1-j}$. Given $v_1, \ldots, v_n$, the updated $v_1, \ldots, v_n$ can still be computed efficiently using FFT (by observing that the above operation is convolution and using the convolution theorem). This increase the number of parameters to $2Dd + 2d^2l + ln = O(Dd + d^2l + ln)$ and the memory (and the time) complexity becomes $O(lnd \log(nd) + lnd^2 + ndD)$. The implementation of the model is given in Appendix B

Experiments We trained a variant of the model (see Appendix B) on the WikiText-103 data set with the same hyperparameters as for the FFT-based model. The model has $112,977,932$ parameters and it achieved perplexity 25.8. We can compare this to the the language modelling on WikiText-103 leaderboard [wik]. We also trained our own implementation of the standard attention based model with 12 attention heads and $127,200,536$ parameters and it achieved perplexity 23.2 on the benchmark (with the same hyperparameters).

References

[glu] GLUE Benchmark. `https://gluebenchmark.com/leaderboard`.

[TDBM20] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[wik] WikiText-103 Benchmark. `https://paperswithcode.com/sota/language-modelling-on-wikitext-103`.

A Python implementation of the FFT-based model

The implementation adds a few extra details - including a layer norm and a skip connection (initialized as 0). This slightly increases the number of parameters but it does not change the asymptotic memory bounds.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def fft(x):
    z = torch.view_as_complex(x.contiguous().view(x.shape[0], -1, 2))
    z = torch.view_as_real(torch.fft.fft(z, norm="ortho"))
    return z.contiguous().view_as(x)


class Layer(nn.Module):
    def __init__(self, dimension):
        super().__init__()
        self.norm = nn.LayerNorm(dimension)
        self.linear_1 = nn.Linear(dimension, 4 * dimension)
        self.linear_2 = nn.Linear(4 * dimension, dimension)
        self.scalar = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        x = self.norm(x)
        x = x + self.linear_2(F.gelu(self.linear_1(x))) * self.scalar
        return fft(x)


class Model(nn.Module):
    def __init__(self, vocabulary_size, n_layers, dimension):
        super().__init__()
        self.embedding = nn.Embedding(vocabulary_size, dimension)
        self.layers = nn.ModuleList([Layer(dimension) for _ in range(n_layers)])
        self.linear = nn.Linear(dimension, vocabulary_size)

    def forward(self, x):
        x = self.embedding(x)
        for layer in self.layers:
            x = layer(x)
        return F.log_softmax(self.linear(x), dim=-1)
```

B Python implementation of the weighted sum model

```
class Layer(nn.Module):
    def __init__(self, dimension, length):
        super().__init__()
        self.linear = nn.Linear(dimension, dimension)
        self.weights = nn.Parameter(torch.zeros(length).unsqueeze(0).unsqueeze(0))
        scale = torch.FloatTensor([math.sqrt(1 / (i + 1)) for i in range(length)])
        self.register_buffer('scale', scale.unsqueeze(-1).unsqueeze(0))
        self.norm_1 = nn.LayerNorm(dimension)

        self.linear_1 = nn.Linear(dimension, 4 * dimension)
        self.linear_2 = nn.Linear(4 * dimension, dimension)
        self.scalar = nn.Parameter(torch.zeros(1))
        self.norm_2 = nn.LayerNorm(dimension)

    def forward(self, x):
        length = self.weights.size()[-1]

        #implementing convolution using fft
        x_1 = x.transpose(-2, -1)
        x_1 = torch.fft.fft(x_1, n=2*length) * \
              torch.fft.fft(self.weights, n=2*length)
        x_1 = torch.fft.ifft(x_1).real[..., :length]
        x_1 = x_1.transpose(-2, -1)

        x_1 = self.norm_1(x + x_1 * self.scale)
        x_2 = self.linear_2(F.gelu(self.linear_1(x_1)))
        x_2 = self.norm_2(x_1 + self.scalar * x_2)
        return x_2


class Model(nn.Module):
    def __init__(self, vocabulary_size, n_layers, length, dimension):
        super().__init__()
        self.embedding = nn.Embedding(vocabulary_size, dimension)
        self.layers = \
            nn.ModuleList([Layer(dimension, length) for _ in range(n_layers)])
        self.linear = nn.Linear(dimension, vocabulary_size)

    def forward(self, x):
        x = self.embedding(x)
        for layer in self.layers:
            x = layer(x)
        return F.log_softmax(self.linear(x), dim=-1)
```