Hannes Bretschneider

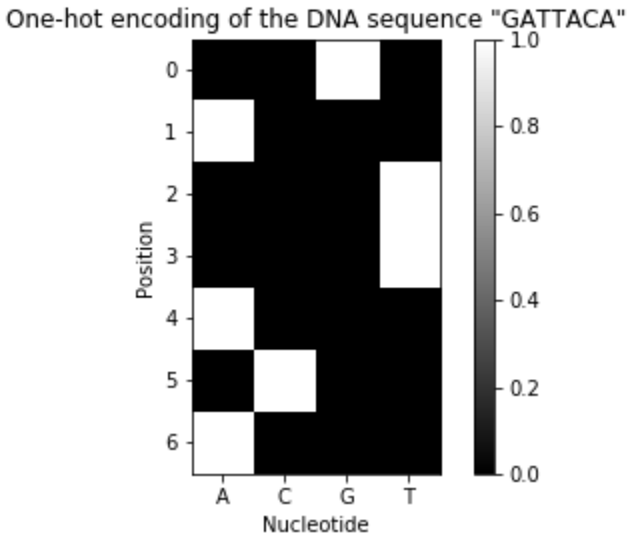
Follow 4 Followers About

Efficient DNA Embedding with TensorFlow

H Hannes Bretschneider Apr 5, 2019 · 8 min read

Introduction

Deep Learning with DNA or other biological sequences like RNA or protein sequence has really taken off in the last few years [1, 2]. Most models that learn from DNA sequence use the *one-hot encoding* scheme which uses four channels for the four possible nucleotides `A`, `C`, `G`, and `T`.



The one-hot encoding of a DNA sequence of length N is a (Nx4) matrix in which the columns correspond to the letters A, C, G, and T and each row has exactly one entry equal to one with the others being zero.

The first step in any such model is therefore to encode the input DNA sequence in the one-hot scheme which can be processed by the following neural network layers whether they are fully connected, convolutional or recurrent.

Encoding the DNA input is usually seen as a trivial step and only warrants a passing mention in most papers. However, after reading code from colleagues and on Github for years, I have seen a surprisingly large number of often colourful and surprising solutions to this superficially easy problem.

Often, one-hot encoding is treated as a pre-processing step that happens before data is fed to the model. For example, a whole training dataset might be pre-encoded and then stored on disk to be used for training. In other cases, the model is wrapped in some Python pre-processing layer that performs the encoding in Python-land, rather than within the TensorFlow graph. For example, the Selene package, which is a framework to work with biological sequences in PyTorch, implements the encoding as a pre-processing step which is implemented in Cython.

However, I have always felt that purely from an interface design standpoint, a model that purports to make a prediction from DNA sequence should accept the sequence as a string and the one-hot encoded representation should be treated as an implementation detail that doesn't need to be visible to the user. This also makes it easier to package the model as a `tf.SavedModel` and share or deploy it, because the graph accepts DNA sequences natively without requiring the user to perform their own encoding.

The encoder only needs to perform a simple job: take a sequence and for each nucleotide, output a vector according to the following mapping.

```
A -> [1, 0, 0, 0]
C -> [0, 1, 0, 0]
G -> [0, 0, 1, 0]
T -> [0, 0, 0, 1]
```

Reading through the TensorFlow API documentation, it is not immediately clear how to perform this operation, nor which one will be the most efficient. Normally, the DNA embedding will be a very cheap operation compared to the cost of running the model itself, but it is also possible that a bad implementation of the embedding could end up becoming a severe bottleneck.

As it turns out, there are a variety of ways to achieve a DNA one-hot encoding using native TensorFlow operations of which I will present three here. However, I'd be interested to hear what other solutions people have come up with.

I'll first present my three implementations and then I'll benchmark them to see whether there are any considerable differences.

Using a lookup table

Learning with DNA sequence is in some ways similar to natural language processing where a lookup table is often used that maps string keys from a vocabulary to integer ids.

TensorFlow has a lookup table class in the `tf.contrib.lookup` module. The `contrib` module will be deprecated with the upcoming TensorFlow 2.0, but there will likely be a very close replacement. TensorFlow also has the `tf.one_hot` function that can convert these integer ids to the one-hot embedding.

The following function maps string-formatted DNA inputs to integer ids. I'll leave out the one-hot encoding step for now, so we can benchmark the two steps separately later.

```
def dna_encode_lookup_table(seq, name="dna_encode"):
  """Map DNA string inputs to integer ids using a lookup table."""

  with tf.name_scope(name):
    # Defining the lookup table
    mapping_strings = tf.constant(["A", "C", "G", "T"])
    table = tf.contrib.lookup.index_table_from_tensor(
      mapping=mapping_strings, num_oov_buckets=0, default_value=-1)

    # Splitting the string into single characters
    seq = tf.squeeze(
      tf.sparse.to_dense(
        tf.string_split([seq], delimiter=""),
        default_value=""
      ),
      0
    )

return table.lookup(seq)
```

The function is just slightly complicated by the fact that we need to split the string into individual characters first for which we use the `tf.string_split` function. Since `tf.string_split` returns a sparse tensor however, we need to convert it back to a dense vector (the lookup table only accepts dense vectors).

Finally, `table.lookup(seq)` returns the result as a tensor of integer ids.

Using bit manipulation to compute integer indices

The use of a lookup table that has only four keys seems like a bit of overkill. But what would be a simpler way to map the DNA alphabet to indices for the `tf.one_hot` function? One idea is to use elementary bitwise functions to compute the indices directly. The mapping we need to perform is the following:

```
'A' = 65 = 0b01000001 -> 0 = 0b00000000
'C' = 67 = 0b01000011 -> 1 = 0b00000001
'G' = 71 = 0b01000111 -> 2 = 0b00000010
'T' = 84 = 0b01010100 -> 3 = 0b00000011
```

All we need to do is find a sequence of operations using the elementary bitwise operators `&`, `|`, `^`, `~`, `<<`, and `>>` that transforms the values on the left to the ones on the right.

One such transformation is the following which I first implemented in pure Python for the sake of simplicity:

```
def dna_to_index(seq):
  def nucleotide_to_index(nt):
    nt &= ~(1 << 6 | 1 << 4)  # Clear bits 5 and 7 from the right
    nt >>= 1                  # Discard least significant bit
    nt ^= (nt & 1 << 1) >> 1  # Swap 0b11 and 0b10
    return nt
    return [nucleotide_to_index(ord(s)) for s in seq]
```

For each letter in the sequence, this snippet first clears the 5th and 7th least significant bits (from the right), followed by a shift of one bit to the right which almost get the right bit patterns except for the fact that the values for `G` and `T` are swapped. Therefore, the only step left to do is replace every `3` with a `2` and every `2` with a `3`. The third line uses the expression `(nt & 1 << 1)` as a mask to affect only the values in which the second bit from the right is set and then uses `xor` to flip the rightmost bit. The following table shows the step-by-step transformation of each input.

```
Input           | =& ~(1<<6|1<< 4)  | =>>1     | ^=(nt & 1<<1)>>1
---------------------------------------------------------------
'A' = 01000001 | 00000001          | 00000000 | 00000000 = 0
'C' = 01000011 | 00000011          | 00000001 | 00000001 = 1
'G' = 01000111 | 00000111          | 00000011 | 00000010 = 2
'T' = 01010100 | 00000100          | 00000010 | 00000011 = 3
---------------------------------------------------------------
```

Taken all together, this operation maps the DNA alphabet `ACGT` to the indices 0, 1, 2, 3.

The implementation of this function in TensorFlow is as follows:

```
def dna_encode_bit_manipulation(seq, name='dna_encode'):
  with tf.name_scope(name):
    bytes = tf.decode_raw(seq, tf.uint8)
    bytes = tf.bitwise.bitwise_and(bytes, ~((1 << 6) | (1 << 4))
    bytes = tf.bitwise.right_shift(bytes, 1)

    mask = tf.bitwise.bitwise_and(bytes, 2)
    mask = tf.bitwise.right_shift(mask, 1)
    bytes = tf.bitwise.bitwise_xor(bytes, mask)
  return bytes
```

This function can replace the lookup table above and can be followed by the `tf.one_hot` function to get the final encoding. Since this function uses only elementwise operations it will run very efficiently on the GPU as well.

Using an embedding table

Another concept from natural language processing is an embedding table which takes integer ids as input (such as the ones that come from a lookup table) and outputs a vector containing the embedding for that id. In natural language processing these embeddings are initialized randomly and trained, but we can make use of the same facilities to map our sequence to a fixed one-hot encoding.

In the two methods above, we mapped the DNA alphabet to the integers `0,…,3` such that we could use the `tf.one_hot` function. But what if we use the integer ASCII codes for `A`, `C`, `G`, and `T` as indices directly? We'll only need to make the table bigger for which we can use the `tf.nn.embedding_lookup` function.

```
def dna_encode_embedding_table(dna_input, name="dna_encode"):
  """Map DNA sequence to one-hot encoding using an
     embedding table."""

  # Define the embedding table
  _embedding_values = np.zeros([84, 4], np.float32)
  _embedding_values[ord('A')] = np.array([1, 0, 0, 0])
  _embedding_values[ord('C')] = np.array([0, 1, 0, 0])
  _embedding_values[ord('G')] = np.array([0, 0, 1, 0])
  _embedding_values[ord('T')] = np.array([0, 0, 0, 1])

  embedding_table = tf.get_variable(
    'dna_lookup_table', _embedding_values.shape,
    initializer=tf.constant_initializer(_embedding_values),
    trainable=False # Ensure that embedding table is not trained
  )

  with tf.name_scope(name):
    dna_input = tf.decode_raw(
      dna_input, tf.uint8) # Interpret string as bytes
    dna_32 = tf.cast(dna_input, tf.int32)
    encoded_dna = tf.nn.embedding_lookup(embedding_table, dna_32)
  return encoded_dna
```

We need to allocate an embedding table with 84 rows because that's the ASCII code of `T`.

This method also has a few other advantages: It can be easily adapted to encode other alphabets like amino acid sequences or we can use this method to account for the IUPAC nucleotide wildcards which for example define an `R` as a purine (`A` or `G`) or `B` as "anything but `A`" (`C`, `G`, or `T`). If this is what we want, we can define the following embedding table:

```
_embedding_values = np.zeros([89, 4], np.float32)
_embedding_values[ord('A')] = np.array([1, 0, 0, 0])
_embedding_values[ord('C')] = np.array([0, 1, 0, 0])
_embedding_values[ord('G')] = np.array([0, 0, 1, 0])
_embedding_values[ord('T')] = np.array([0, 0, 0, 1])
_embedding_values[ord('W')] = np.array([.5, 0, 0, .5])
_embedding_values[ord('S')] = np.array([0, .5, .5, 0])
_embedding_values[ord('M')] = np.array([.5, .5, 0, 0])
_embedding_values[ord('K')] = np.array([0, 0, .5, .5])
_embedding_values[ord('R')] = np.array([.5, 0, .5, 0])
_embedding_values[ord('Y')] = np.array([0, .5, 0, .5])
_embedding_values[ord('B')] = np.array([  0, 1/3, 1/3, 1/3])
_embedding_values[ord('D')] = np.array([1/3,   0, 1/3, 1/3])
_embedding_values[ord('H')] = np.array([1/3, 1/3,   0, 1/3])
_embedding_values[ord('V')] = np.array([1/3, 1/3, 1/3,   0])
_embedding_values[ord('N')] = np.array([.25, .25, .25, .25])
```

Benchmarks

To benchmark these methods, I repeatedly used them to one-hot encode the longest gene in the human genome *DMD* (dystrophine), which is about 2.24 Mbp. I extracted the sequence from the hg38 2bit file using the twobitreader module and computed the reverse complement as *DMD* is on the negative strand:

```
genome = twobitreader.TwoBitFile(args.genome_file)
dmd_sequence_fwd = genome['chrX'][31097676:33339441].upper()

def reverse_complement(seq):
    return "".join("TGCA"["ACGT".index(s)] for s in seq[::-1])

dmd_seq_rev = reverse_complement(dmd_sequence_fwd)
```

The benchmarks are all run on a workstation with an Intel Xeon W-2133 CPU @ 3.60GHz, 16 GB of RAM and an Nvidia Titan Xp. Software used was Python 3.6.8 (Anaconda),TensorFlow 1.12.0 (conda package, GPU enabled and with MKL extensions), and CUDA 9.2.

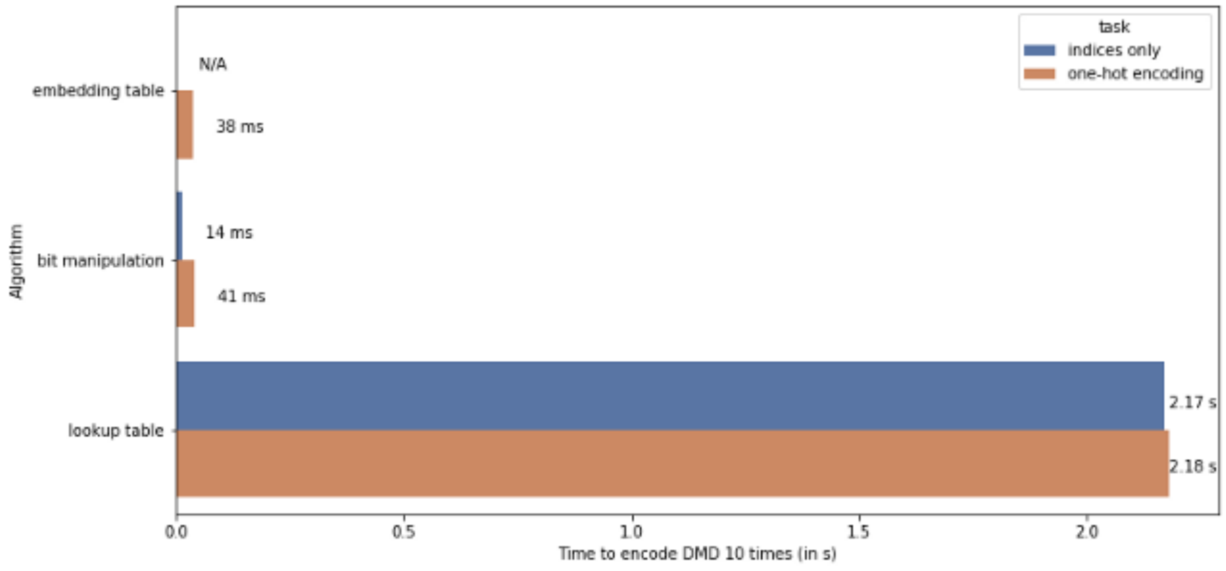
I first ran `dna_encode_lookup_table` and `dna_encode_bit_manipulation` without the `tf.one_hot` step in order to benchmark the two steps individually.

I used `timeit.repeat` with `number=10` and `repeats=10` . The minimum out of 10 repeats to perform the encoding 10 times was 14 ms for the bit manipulation method and 2.17 s for the lookup table method.

The `dna_encode_embedding_table` function took 38 ms to compute the final one-hot encoding directly.

If we benchmark the first two functions followed by the application of `tf.one_hot` , we see that the total time is dominated by computing the one-hot encoding from the indices. The total time of the bit manipulation method and one-hot encoding was 41 ms and the lookup table method followed by one-hot encoding took 2.18 s.

Finally, the following plot shows all the results:



Run time for different implementations of the one-hot encoding for DNA sequence. The task is to embed the complete sequence of the DMD gene (2.24 Mbp) 10 times. Blue bars show only the time to compute integer indices and orange bar show the total time of computing the integer indices followed by one-hot encoding.

The lookup table is obviously a bad choice and very inefficient. Bit manipulation is very fast at computing the integer indices, but if we include the time it takes to compute the one-hot encoding, it is roughly as fast as the embedding method.

Overall, the bit manipulation method has some charm for being so concise, but it cannot be used when embeddings for the IUPAC wildcard values are needed, or when you're working with a different alphabet such as protein sequences. In most cases, the embedding table is likely the most practical as it can be used with any sequence alphabet and runs just as fast as the bit manipulation method.

If you want to play around with these algorithms, the script is available at https://gist.github.com/hannes-brt/54ca5d4094b3d96237fa2e820c0945dd

References

1. Wainberg, M., Merico, D., Delong, A. & Frey, B. J. Deep learning in biomedicine. *Nature Publishing Group* **36,** 829–838 (2018).

2. Ching, T. *et al.* Opportunities and obstacles for deep learning in biology and medicine. *Journal of The Royal Society Interface* **15,** 20170387 (2018).

Machine Learning Genomics Bioinformatics TensorFlow Deep Learning