



You have 1 free member-only story left this month. [Sign up for Medium and get an extra one](#)

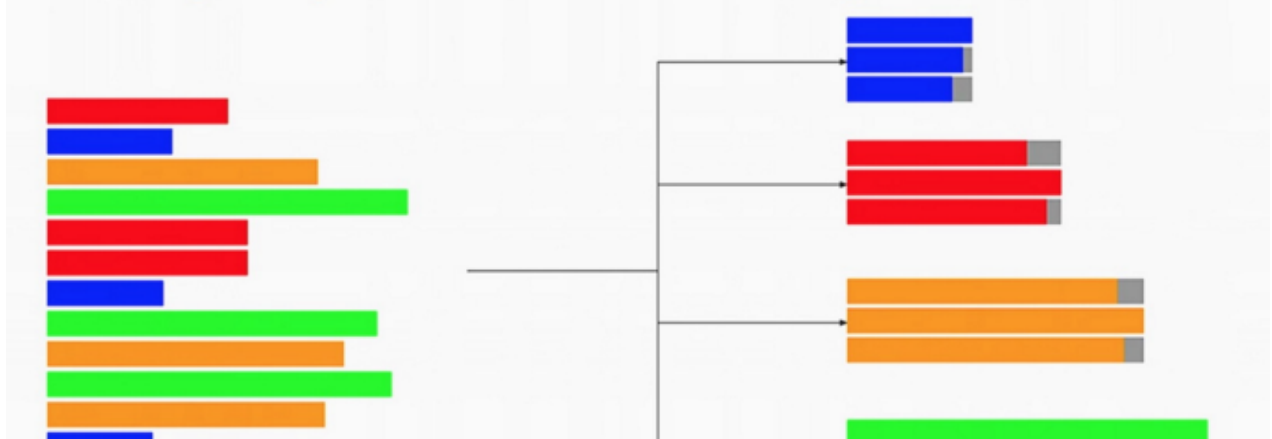
Comprehensive Hands-on Guide to Sequence Model batching strategy: Bucketing technique

An Intro, importance of Bucketing in RNN and math beyond bucketing



Rashmi Margani Apr 30, 2019 · 9 min read ★

Batching Sequence Data: Bucketing



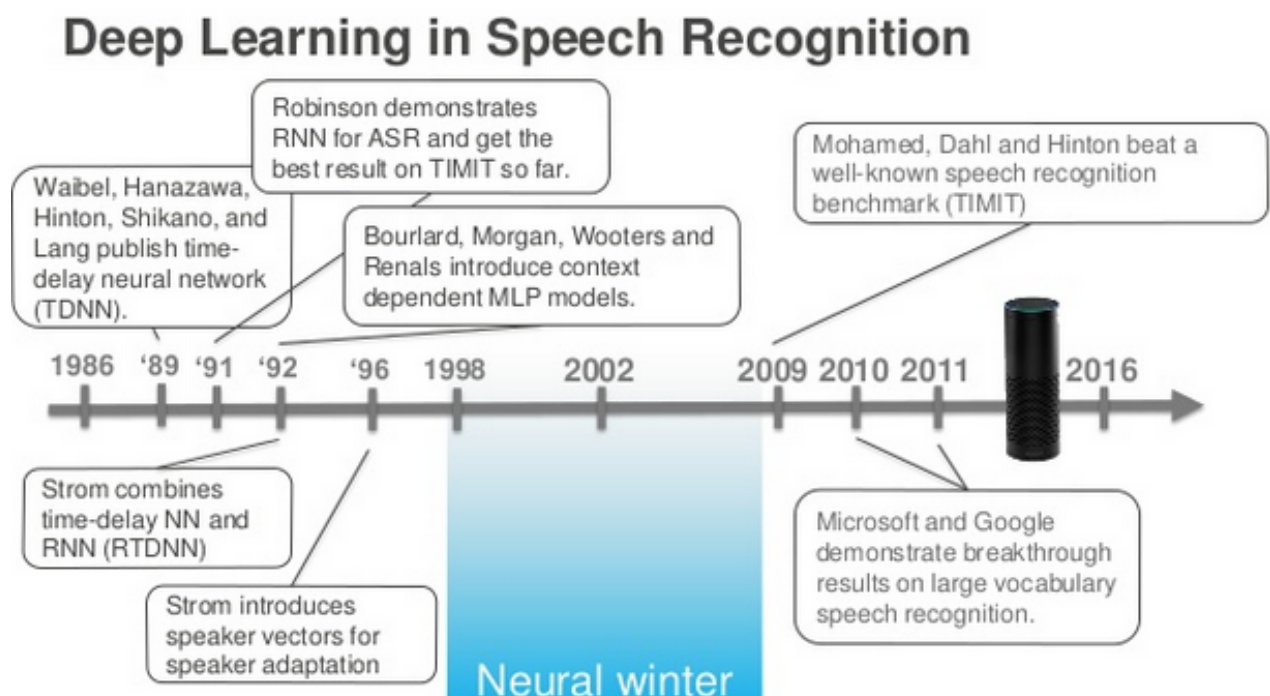
Please go through the RNN(recurrent neural network) model as pre-requisite knowledge to read this story.

This story uses the Bottom Up approach for discussing the bucketing in sequence model.

1. Introduces to importances of bucketing and different bucketing techniques
2. The math behind the bucketing in sequence model.

Now, will focus on “intro, importances of bucketing and different bucketing techniques”

What is the purpose of Bucketing, Why it is needed?

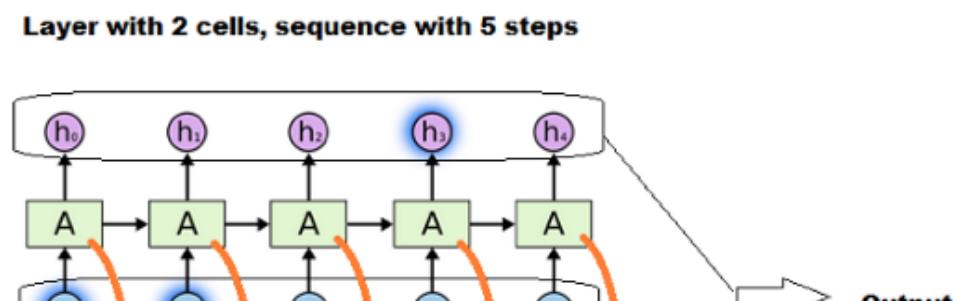


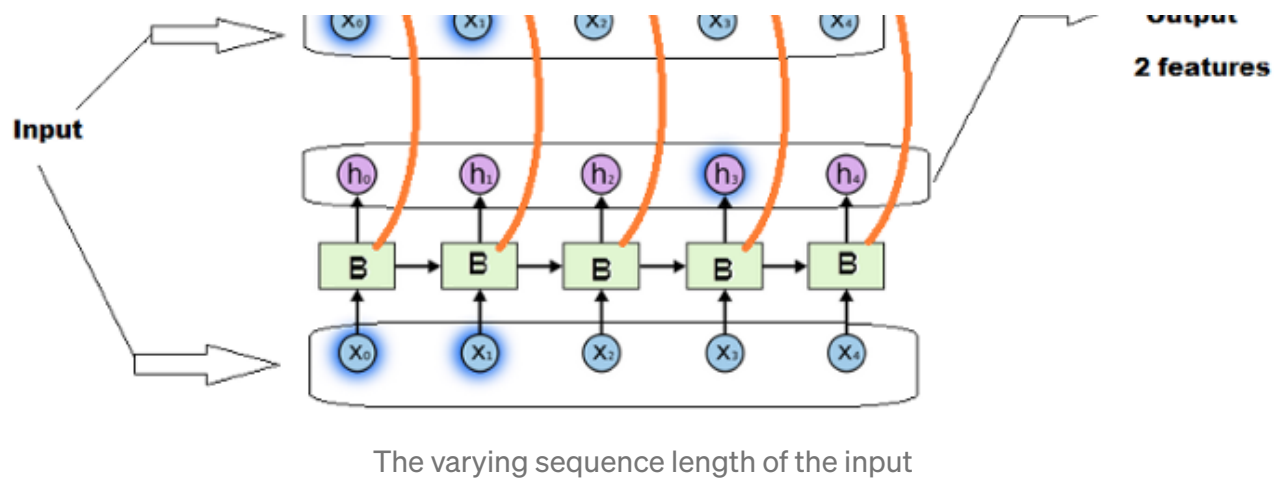
Training of recurrent neural networks for large vocabulary, continuous speech recognition tasks is computationally very expensive and the sequential nature of the recurrent process prohibits to parallelize the training over input frames.

A robust optimization requires to work on large batches of utterances and training time as well as recognition performance can vary strongly depending on the choice of how batches were put together.

The main reason is that combining utterances of different lengths in a mini-batch requires to extend the length of each utterance to that of the longest utterance within the batch, usually by appending zeros. These zero frames are ignored later on when gradients are computed but the forward-propagation of zeros through the RNN is a waste of computing power

How to prevent the wastage of computing power, will minimizing zero padding resolves this issue?, Does it have any effect?





Minimizing zero paddings are to sort the utterances by length and to partition them into batches afterwards. However, there are significant drawbacks to this method. First, the sequence order remains constant in each epoch and therefore the intra-batch variability is very low since the same sequences are usually combined into the same batch. Second, the strategy favours putting similar utterances into the same batch, since short utterances often tend to share other properties.

Now Bucketing comes into the picture.

What is Bucketing, How it prevents waste computation power?

The idea is to perform a bucketing of the training corpus, where each bucket represents a range of utterance lengths and each training sample is assigned to the bucket that corresponds to its length. Afterwards, a batch is constructed by drawing sequences from a randomly chosen bucket. The concept somehow mitigates the issue of zero-paddings if suitable length ranges can be

defined, while still allowing for some level of randomness at least when sequences are selected within a bucket. However, buckets have to be made very large in order to ensure a sufficiently large variability within batches. On the other hand, making buckets too large will increase training time due to irrelevant computations on zero padded frames. Setting these hyper-parameters correctly is therefore of fundamental importance for fast and robust acoustic model training



Now, Will look into the implementation

The function `plot_expected_lengths` samples a number of batches (per default 10000) of a specific batch size from the given lengths, and checks which length this batch would be padded to when using some function `choose_length`. `choose_length` has to return the length which will be padded to given an array of lengths in a batch. In the simplest case, this would be `lambda lengths: lengths.max()`. It could also be `lambda lengths: np.percentile(lengths, q=95)` to pad to the 95th percentile of lengths in a batch.

The lengths each of those 10000 batches are padded to are then plotted as a histogram, and the mean of all lengths with this batch size is plotted as a green line. This length can then be compared to the fixed length (shown in red) which the dataset was previously padded too.

```
def plot_expected_lengths(lengths, batch_sizes,
    choose_length, markers=[], n_batches=10000):
    fig, axarr = plt.subplots(len(batch_sizes), 1, figsize=
    (14, 20), sharex=True)
    expected_lengths = {}

    for i, batch_size in enumerate(batch_sizes):
        maxs = []

        for _ in tqdm(range(n_batches), disable=False):
            val = choose_length(np.random.choice(lengths,
            batch_size))
            maxs.append(math.ceil(val))

        pd.Series(maxs).plot.hist(bins=50, ax=axarr[i],
        density=True, color='black', edgecolor='white', alpha=0.1)
        expected = np.mean(maxs)
        expected_lengths[batch_size] = expected

        max_y = axarr[i].get_ylim()[1]

        axarr[i].vlines([expected], 0, 1e3, 'limegreen',
        lw=4)
        axarr[i].set_ylim([0, max_y])
        axarr[i].set_xlim([0, max(lengths)])
        axarr[i].set_ylabel(f'batch_size={batch_size}',
        rotation=0)
        axarr[i].yaxis.set_label_coords(-0.1, 0.45)
        axarr[i].set_yticks([])

    for marker in markers:
        con = ConnectionPatch(xyA=(marker,
        axarr[0].get_ylim()[1]), xyB=(marker, 0), coordsA='data',
        coordsB='data',
        axesA=axarr[0], axesB=axarr[-1], color='red', lw=4)
        axarr[0].add_artist(con)
```


```
axarr[0].set_zorder(1)
axarr[0].set_title(f'Expected sequence lengths with
various batch sizes (n per batch = {n_batches})')
plt.subplots_adjust(hspace=0)

return expected_lengths

batch_sizes = [128, 256, 512, 1024, 2048, 4096, 8192,
16384, 32768]

expected_lengths = plot_expected_lengths(lengths,
batch_sizes, lambda lengths: lengths.max(), markers=
[MAX_LEN])
plt.xlabel('Maximum sequence length in batch')
print()
```





The expected batch length increases with the batch size. It even surpasses the maximum length of 220 at `batch_size=32768`, and it is significantly smaller than the fixed padding at a reasonable batch size of e. g. 512. When looking at the histogram, you can also see very well that the number of outliers increases when increasing the batch size. Because of padding to the maximum length, the expected batch size is strongly influenced by outliers.

Note that the difference between the green line and the red line for each batch size does not directly relate to the speedup; there is some small overhead to dynamically padding the sequences.

So it is questionable whether it makes sense to use sequence bucketing when padding to some percentile of the lengths for this dataset. We could just as well statically pad to the length at that percentile.

But it definitely does make sense if we want to get the last bit of performance out of our model and don't want to lose any information by truncating sequences. And it is still faster than the

static padding of 220 with reasonably small batch sizes in that case.

Method 1: TextDataset with collate_fn

It uses a custom dataset that stores the text as a regular list with variable lengths and implements sequence bucketing in a `collate_fn` in the data loader.

```
class TextDataset(data.Dataset):
    def __init__(self, text, lens, y=None):
        self.text = text
        self.lens = lens
        self.y = y

    def __len__(self):
        return len(self.lens)

    def __getitem__(self, idx):
        if self.y is None:
            return self.text[idx], self.lens[idx]
        return self.text[idx], self.lens[idx], self.y[idx]

class Collator(object):
    def __init__(self, test=False, percentile=100):
        self.test = test
        self.percentile = percentile

    def __call__(self, batch):
        global MAX_LEN

        if self.test:
            texts, lens = zip(*batch)
        else:
            texts, lens, target = zip(*batch)

        lens = np.array(lens)
        max_len = min(int(np.percentile(lens,
self.percentile)), MAX_LEN)
        texts = torch.tensor(sequence.pad_sequences(texts,
```

```

maxlen=max_len), dtype=torch.long)

        if self.test:
            return texts

        return texts, torch.tensor(target,
dtype=torch.float32)

train_collate = Collator(percentile=100)
train_dataset = TextDataset(x_train, lengths,
y_train_torch.numpy())
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, collate_fn=train_collate)

n_repeats = 10

start_time = time.time()
for _ in range(n_repeats):
    for batch in tqdm(train_loader):
        pass
method1_time = (time.time() - start_time) / n_repeats

```

Method 2: SequenceDataset with default loader

Uses a custom dataset to do all the work (sequence bucketing, splitting into batches, shuffling)

```

class SequenceDataset(torch.utils.data.Dataset):
    """
        Dataset using sequence bucketing to pad each batch
        individually.

        Arguments:
            sequences (list): A list of variable length tokens
            (e. g. from keras tokenizer.texts_to_sequences)
            choose_length (function): A function which receives
            a numpy array of sequence lengths of one batch as input
            and returns the length
            this batch should be padded to.
    """

```

other_features (list, optional): A list of tensors with other features that should be fed to the NN alongside the sequences.

labels (Tensor, optional): A tensor with labels for the samples.

indices (np.array, optional): A numpy array consisting of indices to iterate over.

shuffle (bool): Whether to shuffle the dataset or not. Default false.

batch_size (int): Batch size of the samples. Default 512.

```
"""
    def __init__(self, sequences, choose_length,
other_features=None, labels=None,
                indices=None, shuffle=False,
batch_size=512):
        super(SequenceDataset, self).__init__()

        self.sequences = np.array(sequences)
        self.lengths = np.array([len(x) for x in
sequences])
        self.n_samples = len(sequences)
        self.choose_length = choose_length
        self.other_features = other_features
        self.labels = labels

        if indices is not None:
            self.indices = indices
        else:
            self.indices = np.arange(len(sequences))

        self.batch_size = batch_size
        self.shuffle = shuffle

        if self.shuffle:
            self._shuffle()

    def __len__(self):
        return math.ceil(len(self.indices) /
self.batch_size)

    def _shuffle(self):
        self.indices = np.random.permutation(self.indices)

    def __getitem__(self, i):
        idx = self.indices[(self.batch_size * i):
(self.batch_size * (i + 1))]

        if self.shuffle and i == len(self) - 1:
            self._shuffle()
```

```

        pad_length
        =math.ceil(self.choose_length(self.lengths[idx]))

        padded_sequences=sequence.pad_sequences(self.sequences[idx]
        , maxlen=pad_length)

        x_batch = [torch.tensor(padded_sequences,
        dtype=torch.long)]

        if self.other_features is not None:
            x_batch += [x[idx] for x in
            self.other_features]

        if self.labels is not None:
            out = x_batch, self.labels[idx]
        else:
            out = x_batch

        return out

train_dataset = SequenceDataset(x_train, lambda lengths:
lengths.max(), other_features=[lengths], shuffle=False,
batch_size=batch_size)
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=1, shuffle=False)

n_repeats = 10

start_time = time.time()
for _ in range(n_repeats):
    for batch in tqdm(train_loader):
        pass
method2_time = (time.time() - start_time) / n_repeats

```

Method 3: Default TensorDataset with custom collate_fn

A new method using the DataLoader collate_fn for sequence bucketing that works with a regular TensorDataset .

```

class SequenceBucketCollator():
    def __init__(self, choose_length, sequence_index,
length_index, label_index=None):
        self.choose_length = choose_length
        self.sequence_index = sequence_index
        self.length_index = length_index
        self.label_index = label_index

    def __call__(self, batch):
        batch = [torch.stack(x) for x in list(zip(*batch))]

        sequences = batch[self.sequence_index]
        lengths = batch[self.length_index]

        length = self.choose_length(lengths)
        mask = torch.arange(start=maxlen, end=0, step=-1) <
length
        padded_sequences = sequences[:, mask]

        batch[self.sequence_index] = padded_sequences

        if self.label_index is not None:
            return [x for i, x in enumerate(batch) if i !=
self.label_index], batch[self.label_index]

        return batch

```

In [20]:

```

dataset = data.TensorDataset(x_train_padded, lengths,
y_train_torch)
train_loader = data.DataLoader(dataset,
batch_size=batch_size,
collate_fn=SequenceBucketCollator(lambda x: x.max(), 0, 1,
2))

```

In [21]:

```
n_repeats = 10
```


```
start_time = time.time()
for _ in range(n_repeats):
    for batch in tqdm(train_loader):
        pass
method3_time = (time.time() - start_time) / n_repeats
```

In [22]:

```
plt.figure(figsize=(20, 10))
sns.set(font_scale=1.2)
barplot = sns.barplot(x=[
    'TextDataset with collate_fn',
    'SequenceDataset with default loader',
    'Default TensorDataset with custom collate_fn'], y=[
    method1_time,
    method2_time,
    method3_time
])


plt.title('Speed comparison of sequence bucketing methods')
plt.ylabel('Time in seconds to iterate over whole train dataset')
print()
```





Method 1 is quite a bit slower than the rest, but method 2 and 3 are pretty close to each other, keep in mind that the majority of the time it takes to train the NN is spent in the actual computation, not while loading.

Now, will discuss “**The math behind the bucketing in sequence model**”



As we know the bucketing approach increases the training speed for tasks where a length of the input sequence may vary significantly. Now, we will discuss on the optimal batch bucketing by input sequence length and data parallelization on multiple

graphical processing units with Math and workflow of bucketing in RNN.

And the evaluation is performed in terms of the wall clock time, the number of epochs, and validation loss value.

The workflow of the training algorithm in RNN.



The RNN model training algorithm that runs in parallel on multiple Graphical Processing Units (GPUs). The developed solution uses a map-reduce approach for parallel computing of individual models by sub-partitioning training data. The training data is shuffled before every epoch and is equally redistributed between different GPU processes. Each training process applies batch bucketing optimization scheme by clustering training sequences considering input lengths. Final model parameters are obtained by reducing the results of each training process.

Sequence Model bucketing algorithm

The bucketing can be described as an optimization problem. Let $S = \{S_1, S_2, \dots, S_n\}$ be the set of sequences and $l_i = |s_i|$ is the length of sequence i . Each GPU processes sequences in a mini-batch in a synchronized parallel manner, so processing time of a mini-batch $B = \{S_1, S_2, \dots, S_k\}$ is proportional to



taking a maximum of each set of sequence

And the processing time of the whole set is expressed as:



Sequence model bucketing processing time

The minimum and maximum sequence length in a mini-batch might be very different if sequences were shuffled randomly before splitting. Experiment with different shuffling, find interesting insight with various dataset in your sequence model using the above discussed bucketing technique.

Hope you enjoyed this story, thanks for reading.

Thanks to [Ludovic Benistant](#) for editing this article.

[Machine Learning](#)

[Deep Learning](#)

[Artificial Intelligence](#)

[Data Science](#)

[Towards Data Science](#)

[About](#)

[Help](#)

[Legal](#)