Module 46 : Batching for sequence models in pytorch

Batching : process where multiple instances are passed forward or backward.

*) till eval fn, everything written in the previous practical model for sequence labeling is used.

top k metric

```
def eval (net, n_points, k, x_, y_, device = 'cpu'):
    net = net.eval() .to (device)
    data_ = dataloader (n_points, x_, y_)
    correct = 0

    for name, language, name_one, lang_rep in data_:
        output = infer (net, name)
        val, indices = output.top (k)
        indices = indices.to ('cpu')

        if lang_repr in indices:
            correct += 1

    accuracy = correct / n_points
    return accuracy
```

*) Batching in sequence models : two aspects :

i) vectorisation : large performance improvement through vectorisation. for a given task, I can either perform it individually or I can create larger vectors of different data items, and do the work parallely. for example point wise addition of vectors. Also called **data parallelism**, and is exploited heavily in GPUs, but even on CPU (due to the presence of memory cache, which can be fully saturated with larger vectors). So TLDR, vectorisation improves performance due to the **single instruction multiple data** (SIMD) parallelism

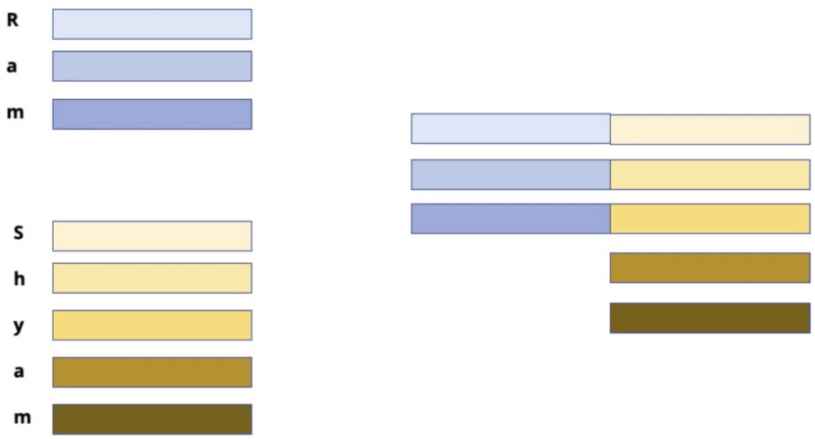
ii) aggregated gradient computation : reduces variability/noise in parameter updates. updating using one sample will introduce high variance. Parameter updates based on multiple sample based computation would have less variance. Here, batching is employed. we aggregate the gradients for individual samples and then perform an update of the network parameters. Leads to much more stable learning.

(already done)

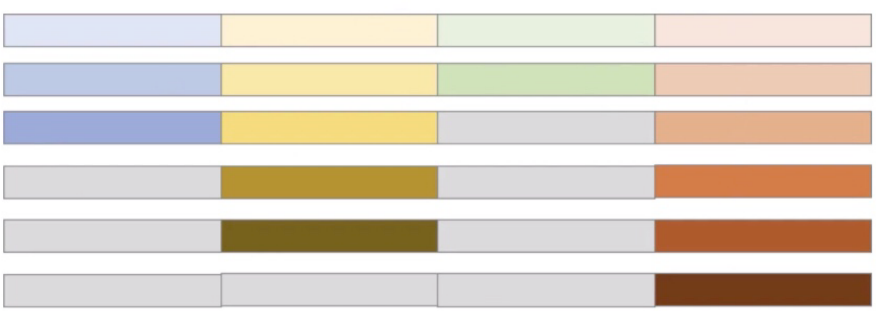
aggregated gradient comp. can be performed w/o vectorisation.

*) vectorisation cannot be done for characters of the same word due to the underlying sequential nature.

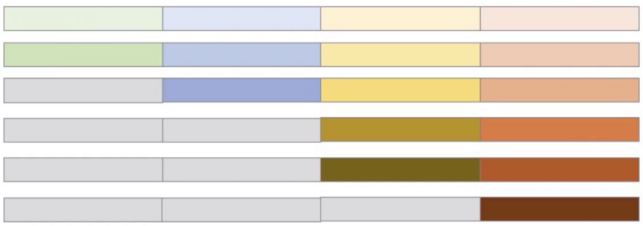
*) Another option is to concatenate the one representation of different characters in different names. Here the issue of length comes in, and that is one hurdle to get over.



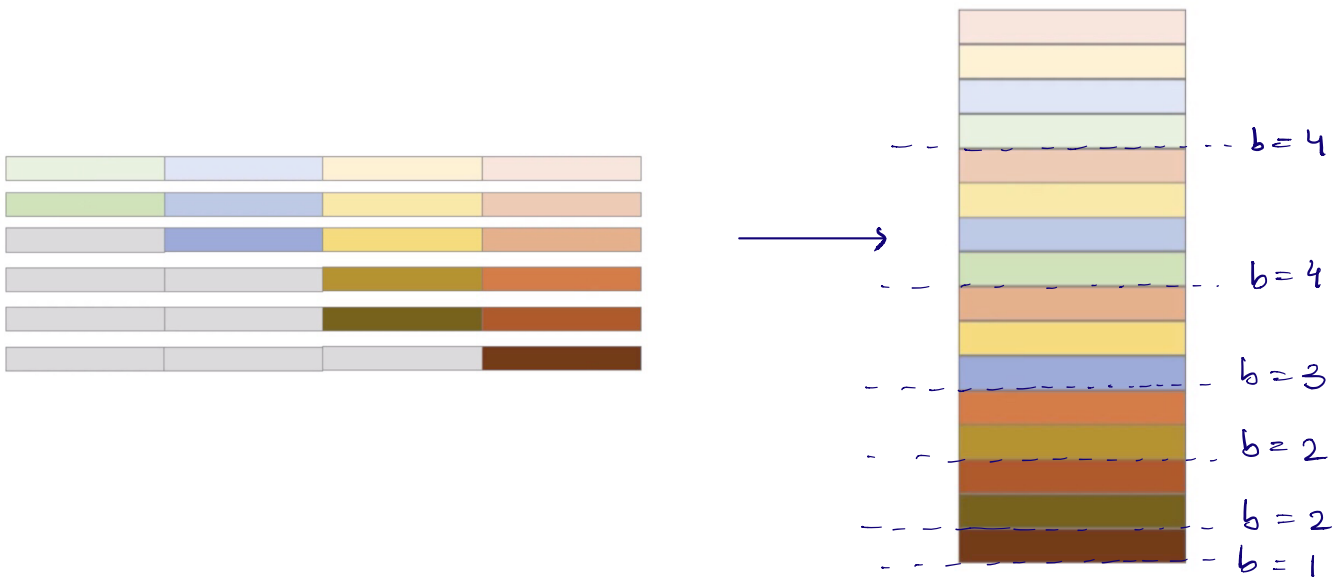
*) To solve this issue, we can use padding. But this can become quite inefficient. (especially on storage, which is expensive on GPU).



*) Doing better : Step 1 : Sort



Step 2 : Pack → Packing the representation into a more dense representation.

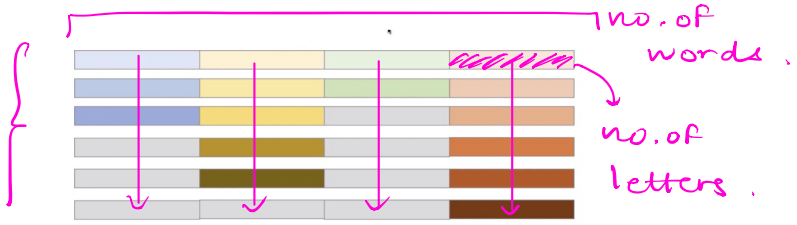


Step 3 : Batch sizes : vectorization for a dynamic batch size. for the single instruction.

*) We generate a packed data structure with the compact data structure, and the batch sizes

*) Batching

max word size




```
def batched_name_rep(names, max_word_size):
```

slowest growing index ↑ ; batch_size →

```
    rep = torch.zeros(max_word_size, len(names),
```

↙ corresponding to levels

```
                      n_letters)
```

↳ encoding

```
    for name_index, name in enumerate(names):
        for letter_index, letter in enumerate(name):
            pos = all_letters.find(letter)
            rep[letter_index][name_index][pos] = 1
    return rep
```

↓ all zeros to employ implicit padding via the definition

```
def print_char(name_reps):
```

(to visualize the repr achieved)

```
    name_reps = name_reps.view((-1, name_reps.size()
                                [-1]))
    for t in name_reps:
        if (torch.sum(t) == 0)
            print('<pad>')
        else:
            index = t.argmax()
            print(all_letters[index])
```

24 * 57

```
out_ = batched_name_rep (['Shyam', 'Ram'], 5)
print (out_)
print (out_.shape)
print_char (out_)
```

```
def batched_lang_rep (langs):
    rep = torch.zeros ([len (langs)], dtype = torch.long)
    for index, lang in enumerate (langs):
        rep [index] = languages.index (lang)
    return rep
```

to return the packed repr of the data

```
def batched_dataloader (npoints, x_, y_, verbose = False,
                        device = 'cpu'):
```

for debugging purposes → involves print stmts.

```
    names = []
    langs = []
    x_lengths = []

    for i in range (n_points):
        index_ = np.random.randint (len (X_))
        name, lang = x_ [index], Y_ [index]
        x_lengths.append (len (name))
```

```
        names.append(name)
        langs.append(lang)
    max_length = max(x_lengths)
```

padded repr.

```
    names_rep = batched_name_rep(names, max_length).
                                        to(device)
    langs_rep = batched_lang_rep(langs).to(device)
```

to packed repr

```
    packed_names_rep = torch.nn.utils.rnn.pack_
        padded_sequence(names_rep, x_lengths,
        enforce_sorted = False)
```

packed_names_rep: not a std. tensor

x_lengths: required to figure batch sizes

```
    if verbose:
        print(names_rep.shape, padded_names_rep.
                                    data.shape)
        print('--')

    if verbose:
        print(names)
        print-char(names_rep)
        print('--')
```

for packed_names_repr, we can use the 'data' & 'batch_sizes' fields to access the req. info.

```
    if verbose:
        print_char(packed_names_rep.data)
        print('Lang Rep', langs_rep.data)
        print('Batch sizes', padded_names_rep.
                                batch_sizes)

    return packed_names_rep.to(device),
                                langs_rep
```

```
p,l = batched_dataloader(2, X_train, y_train, True)
```

```
def train_batch(net, opt, criterion, n_points, device='cpu'):

    net.train().to(device)
    opt.zero_grad()

    batch_input, batch_groundtruth =
        batched_dataloader(n_points, X_train, y_train,
                                False, device)

    output, hidden = net(batch_input)
```

cleaner than the non-batched version

```
loss = criterion ( output , batch_groundtruth)
loss.backward()
opt.step()
return loss
```

→ full_training - setup remains same from unbatched example.

*) Both model and data have to be moved to GPU.