View code on Github

```python
import torch
from torch import nn

from labml_helpers.module import Module
from labml_nn.transformers.feed_forward import FeedForward
from labml_nn.transformers.mha import MultiHeadAttention
from labml_nn.utils import clone_module_list
```

Switch Transformer

This is a miniature PyTorch implementation of the paper Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. Our implementation only has a few million parameters and doesn't do model parallel distributed training. It does single GPU training, but we implement the concept of switching as described in the paper.

The Switch Transformer uses different parameters for each token by switching among parameters based on the token. Therefore, only a fraction of parameters are chosen for each token. So you can have more parameters but less computational cost.

The switching happens at the Position-wise Feedforward network (FFN) of each transformer block. Position-wise feedforward network consists of two sequentially fully connected layers. In switch transformer we have multiple FFNs (multiple experts), and we chose which one to use based on a router. The output is a set of probabilities for picking a FFN, and we pick the one with the highest probability and only evaluate that. So essentially the computational cost is the same as having a single FFN. In our implementation this doesn't parallelize well when you have many or large FFNs since it's all happening on a single GPU. In a distributed setup you would have each FFN (each very large) on a different device.

The paper introduces another loss term to balance load among the experts (FFNs) and discusses dropping tokens when routing is not balanced.

Here's the training code and a notebook for training a switch transformer on Tiny Shakespeare dataset.

Open in Colab

Routing among multiple FFNs

```
class SwitchFeedForward(Module):
```

- `capacity_factor` is the capacity of each expert as a factor relative to ideally balanced load
- `drop_tokens` specifies whether to drop tokens if more tokens are routed to an expert than the capacity
- `is_scale_prob` specifies whether to multiply the input to the FFN by the routing probability
- `n_experts` is the number of experts
- `expert` is the expert layer, a FFN module

```
    def __init__(self, *,
                 capacity_factor: float,
                 drop_tokens: bool,
                 is_scale_prob: bool,
                 n_experts: int,
                 expert: FeedForward,
                 d_model: int):
```

- `d_model` is the number of features in a token embedding
- `d_ff` is the number of features in the hidden layer of the FFN
- `dropout` is dropout probability in the FFN

```
        super().__init__()

        self.capacity_factor = capacity_factor
        self.is_scale_prob = is_scale_prob
        self.n_experts = n_experts
        self.drop_tokens = drop_tokens
```

make copies of the FFNs

```
        self.experts = clone_module_list(expert, n_experts)
```

Routing layer and softmax

```
        self.switch = nn.Linear(d_model, n_experts)
        self.softmax = nn.Softmax(dim=-1)
```

```
    def forward(self, x: torch.Tensor):
```

- `x` is the input to the switching module with shape `[seq_len, batch_size, d_model]`

Capture the shape to change shapes later

```
        seq_len, batch_size, d_model = x.shape
```

Flatten the sequence and batch dimensions

```
        x = x.view(-1, d_model)
```

Get routing probabilities for each of the tokens.

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_j^N e^{h(x)_j}}$$

```
        route_prob = self.softmax(self.switch(x))
```

where N is the number of experts `n_experts` and $h(\cdot)$ is the linear transformation of token embeddings.

Get the maximum routing probabilities and the routes. We route to the expert with highest probability

```
        route_prob_max, routes = torch.max(route_prob, dim=-1)
```

Get indexes of tokens going to each expert

```
        indexes_list = [torch.eq(routes, i).nonzero(as_tuple=True)[0] for i in range(self.n_experts)]
```

Initialize an empty tensor to store outputs

```
        final_output = x.new_zeros(x.shape)
```

Capacity of each expert.

$$\text{expert capacity} = \frac{\text{tokens per batch}}{\text{number of experts}} \times \text{cap}$$

```
        capacity = int(self.capacity_factor * len(x) / self.n_experts)
```

Number of tokens routed to each expert.

```
        counts = x.new_tensor([len(indexes_list[i]) for i in range(self.n_experts)])
```

Initialize an empty list of dropped tokens

```
        dropped = []
```

Only drop tokens if `drop_tokens` is `True`.

```
        if self.drop_tokens:
```

Drop tokens in each of the experts

```
            for i in range(self.n_experts):
```

Ignore if the expert is not over capacity

```
                if len(indexes_list[i]) <= capacity:
                    continue
```

Shuffle indexes before dropping

```python
                indexes_list[i] = indexes_list[i][torch.randperm(len(indexes_list[i]))]
```

Collect the tokens over capacity as dropped tokens

```python
                dropped.append(indexes_list[i][capacity:])
```

Keep only the tokens upto the capacity of the expert

```python
                indexes_list[i] = indexes_list[i][:capacity]
```

Get outputs of the expert FFNs

```python
        expert_output = [self.experts[i](x[indexes_list[i], :]) for i in range(self.n_experts)]
```

Assign to final output

```python
        for i in range(self.n_experts):
            final_output[indexes_list[i], :] = expert_output[i]
```

Pass through the dropped tokens

```python
        if dropped:
            dropped = torch.cat(dropped)
            final_output[dropped, :] = x[dropped, :]

        if self.is_scale_prob:
```

Multiply by the expert outputs by the probabilities $y = p_i(x) E_i(x)$

```python
            final_output = final_output * route_prob_max.view(-1, 1)
        else:
```

Don't scale the values but multiply by $\frac{p}{\hat{p}} = 1$ so that the gradients flow (this is something we experimented with).

```python
            final_output = final_output * (route_prob_max / route_prob_max.detach()).view(-1, 1)
```

Change the shape of the final output back to `[seq_len, batch_size, d_model]`

```python
        final_output = final_output.view(seq_len, batch_size, d_model)
```

Return

```python
        return final_output, counts, route_prob.sum(0), len(dropped), route_prob_max
```

- the final output
- number of tokens routed to each expert
- sum of probabilities for each expert
- number of tokens dropped.
- routing probabilities of the selected experts

These are used for the load balancing loss and logging

```
class SwitchTransformerLayer(Module):
```

Switch Transformer Block

This is the same as normal transformer block with handling extra outputs of switch feedforward module.

- `d_model` is the token embedding size
- `attn` is the attention module
- `feed_forward` is the feed forward module (which is the switching module in this case)
- `dropout_prob` is the probability of dropping out after self attention and FFN

```
    def __init__(self, *,
                 d_model: int,
                 attn: MultiHeadAttention,
                 feed_forward: SwitchFeedForward,
                 dropout_prob: float):
```

```
        super().__init__()
        self.size = d_model
        self.attn = attn
        self.feed_forward = feed_forward
```

```python
        self.dropout = nn.Dropout(dropout_prob)
        self.norm_self_attn = nn.LayerNorm([d_model])
        self.norm_ff = nn.LayerNorm([d_model])
```

```python
    def forward(self, *,
                x: torch.Tensor,
                mask: torch.Tensor):
```

Normalize the vectors before doing self attention

```python
        z = self.norm_self_attn(x)
```

Run through self attention, i.e. keys and values are from self

```python
        self_attn = self.attn(query=z, key=z, value=z, mask=mask)
```

Add the self attention results

```python
        x = x + self.dropout(self_attn)
```

Normalize for feed-forward

```python
        z = self.norm_ff(x)
```

Pass through the switching feed-forward network

```python
        ff, counts, route_prob, n_dropped, route_prob_max = self.feed_forward(z)
```

Add the feed-forward results back

```python
        x = x + self.dropout(ff)

        return x, counts, route_prob, n_dropped, route_prob_max
```

Switch Transformer

```python
class SwitchTransformer(Module):
```

```python
    def __init__(self, layer: SwitchTransformerLayer, n_layers: int):
        super().__init__()
```

Make copies of the transformer layer

```python
        self.layers = clone_module_list(layer, n_layers)
```

Final normalization layer

```python
        self.norm = nn.LayerNorm([layer.size])
```

```python
    def forward(self, x: torch.Tensor, mask: torch.Tensor):
```

Run through each transformer layer

```python
        counts, route_prob, n_dropped, route_prob_max = [], [], [], []
        for layer in self.layers:
            x, f, p, n_d, p_max = layer(x=x, mask=mask)
            counts.append(f)
            route_prob.append(p)
            n_dropped.append(n_d)
            route_prob_max.append(p_max)
```

Finally, normalize the vectors

```python
        x = self.norm(x)
```

```python
        return x, torch.stack(counts), torch.stack(route_prob), n_dropped, torch.stack(route_prob_max)
```