MX Code Integration

Figure 1 below illustrates an example of using the MX library. The left pane shows source code for a Transformer MLP layer, including **LayerNorm**, two **fully connected layers**, **GELU activation**, and **residual add**. The right panel shows the same MLP layer modified for training and inference with MX and Bfloat quantization. The code changes are highlighted in green. You can also find this example under *examples* in the repo.

1. **Import** the necessary layers and functions from the MX library
2. **Add *mx_specs*** as an argument to the ResidualMLP layer. This is a dict used to configure the MX format
3. **Replace Pytorch modules** (Linear and LayerNorm) with replacements from the MX library. The replacements take mostly the same arguments but also require *mx_specs*
4. **Replace Pytorch functions** (gelu, matmul, softmax) with replacements from the MX library. The replacements take mostly the same arguments but also require *mx_specs*
5. **For residual layers**, we need to explicitly split the input into two paths using **simd_split**, then later add the two paths together using **simd_add**

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



class ResidualMLP(torch.nn.Module):
    def __init__(self, hidden_size):

        super(ResidualMLP, self).__init__()



        self.layernorm = torch.nn.LayerNorm(
            hidden_size
        )


        self.dense_4h = torch.nn.Linear(
            hidden_size,
            4 * hidden_size
        )


        self.dense_h = torch.nn.Linear(
            4 * hidden_size,
            hidden_size
        )


    def forward(self, inputs):




        norm_outputs = self.layernorm(inputs)

        # MLP
        proj_outputs = self.dense_4h(norm_outputs)
        proj_outputs = F.gelu(proj_outputs)

        mlp_outputs = self.dense_h(proj_outputs)

        # Residual Connection
        outputs = inputs + mlp_outputs


        return outputs
```

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

from mx import Linear, LayerNorm
from mx import gelu, imd_split, simd_add

class ResidualMLP(torch.nn.Module):
    def __init__(self, hidden_size,
                 mx_specs):
        super(ResidualMLP, self).__init__()

        self.mx_specs = mx_specs

        self.layernorm = LayerNorm(
            hidden_size,
            mx_specs=mx_specs
        )

        self.dense_4h = Linear(
            hidden_size,
            4 * hidden_size,
            mx_specs= mx_specs
        )

        self.dense_h = Linear(
            4 * hidden_size,
            hidden_size,
            mx_specs= mx_specs
        )

    def forward(self, inputs):
        # Explicitly split the input
        inputs, residual = simd_split(inputs,
            mx_specs=self.mx_specs)

        norm_outputs = self.layernorm(inputs)

        # MLP
        proj_outputs = self.dense_4h(norm_outputs)
        proj_outputs = gelu(proj_outputs,
            mx_specs=self.mx_specs)
        mlp_outputs = self.dense_h(proj_outputs)

        # Residual Add
        outputs = simd_add(residual, mlp_outputs
            mx_specs=self.mx_specs)

        return outputs
```

You can use **add_mx_args** and **get_mx_specs** to parse MX arguments and create the *mx_specs* object.

```python
import argparse
from mx import add_mx_args, get_mx_specs

parser = argparse.ArgumentParser()
parser = add_mx_args(parser)         # This adds all MX config related arguments

args = parser.parse_args()

mx_specs = get_mx_specs(args)        # This creates the mx_specs dict
```

Taking Care of isinstance

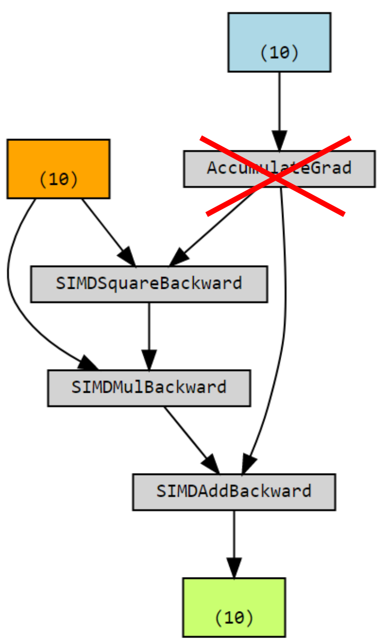
Weight initialization code may iterate though all the instances of a module (e.g., nn.Linear) and do something. Make sure to include our MX Linear module in that list:

```python
if isinstance(module, nn.Linear):
    module.weight.data.normal_(mean=0.0, std=1.0)
```

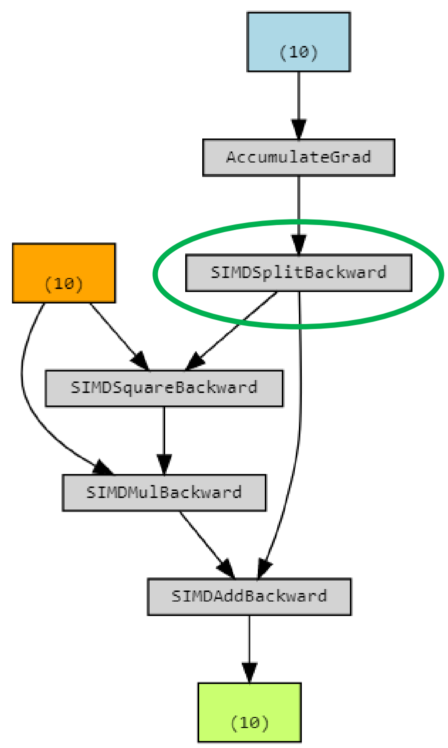
```python
from mx import Linear

if isinstance(module, (nn.Linear, Linear)):
    module.weight.data.normal_(mean=0.0, std=1.0)
```

Taking Care of Residual Adds

Always use **simd_split** and **simd_add** to quantize the residual path.



Incorrect residual. There's no op to quantize the gradient summation from the two paths!



Correct residual. The SIMDSplitBackward op will quantize the gradient summation.