Case Study: Accelerating a CNN on the MNIST dataset

The CNN architecture is used commonly in image-based machine learning (computer vision).

In this blog post, I demonstrate how you can use `moco` to make a `CNN` classification model trained on MNIST with `pytorch_lightning` run faster in inference, or more generally any `nn.Sequential` architecture.

Dataset

Dataset: MNIST

This canonical dataset contains 28x28 images containing handwritten digits and the task is to predict the digit (1-10) for each image. The train set contains 60000 images and the test set contains 10000 images.

Model

Model: CNN Model

Architecture:

```
self.model = nn.Sequential(
    nn.Conv2d(1, 32, 3, 1),  # 28x28 -> 26x26
    nn.ReLU(),
    nn.Conv2d(32, 64, 3, 1), # 26x26 -> 24x24
    nn.ReLU(),
    nn.MaxPool2d(2),         # 24x24 -> 12x12
    nn.Flatten(),
    nn.Linear(64 * 12 * 12, 128),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(128, 10)
)
```

The model gets 97.48% accuracy on this dataset. The model takes 9.485 seconds to evaluate the entire dataset of 60000 samples.

Accelerate it

Methods

We extract intermediate representations from the `CNN` model.

Specifically, we extract representations from after the first ReLU layer, resulting in a vector of shape 60000 x 32 x 26 x 26. We do this because we observe through profiling that Conv2D-2, the second Conv-2D layer is the performance bottleneck.

Then I flattened it to be a 60000 x 21632 sized matrix, and ran it through the analysis system to define rules, that when activated isolated groups of data point sharing the same class designiation.

With this rule defined, the way the model now works is the following:

- it runs the first Conv2D layer, then the ReLU layer.
- now given the output of the attention layer, it has a decision to make: the easy-data-classifier makes a prediction. If the prediction is `1` then by design, the predicted class of the model is the value associated with that rule, so the model early exits and predicts that value. If the prediction is 0, the rest of the model is run as normal.

Analytics Step

```python
from moco import AnalyticsEngine
import numpy as np
import pickle as pkl
from moco import build_embeddings_from_predictions

array, preds_array = build_embeddings_and_predictions(m, data, 2)

np.save('train_layer_2_mnist.npy', array)
np.save('train_preds_mnist.npy', preds_array)
print("Done saving.")

array_flattened = array.reshape((array.shape[0], -1))
preds = preds_array.argmax(axis = 1)
ae = AnalyticsEngine(array_flattened, {'predictions': preds})
for c in range(10):
    rule, out = ae.compute_linear_sufficient_rule("predictions", c,
fit_on_val=True)
    md = rule.metadata
    with open(f'mnist_rule_{c}.pkl', 'wb') as f:
        pkl.dump(md, f)
```

Now, using the rules we defined, we plug these rules into the `BranchedNetwork`, and process the data, evaluating the difference in predictions from baseline to initial and the total latency for each rule constructed.

```python
from moco import BranchedNetwork
import torch

results = {}

for pred in range(10):
    # Load from moco.AnalyticsEngine, or cached from disk.
    rule = load_rule(pred)
    print(rule.class_, rule.decision_function_)
    new_model = BranchedNetwork(m.model, layer = 2, rule = rule)
    total_experimental_time = 0
```

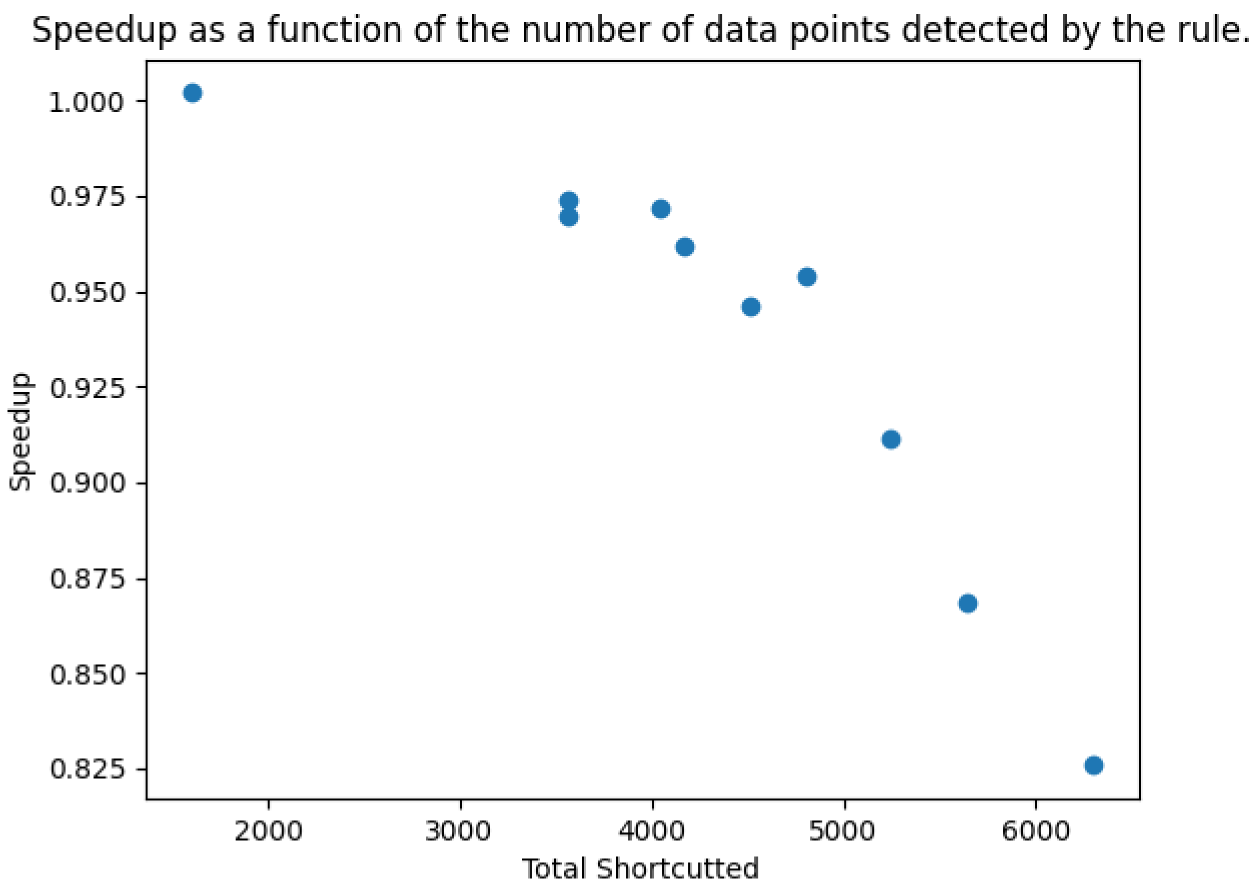
```
        bl_time = 0
        prediction_n = []
        prediction_o = []
        total_shortcutted = 0
        for i in tqdm(range(0, len(data), 1000)):
            x = data[i: i + 1000].type(torch.FloatTensor)
            start = time.time()
            p_new, mask = new_model(x)
            total_shortcutted += mask.sum()
            prediction_n.append(p_new)
            end = time.time()
            total_experimental_time += end - start

            start = time.time()
            p_old = m.model(x)
            end = time.time()
            bl_time += end - start

            prediction_o.append(p_old)
        end = time.time()
        preds_old = torch.vstack(prediction_o)
        preds_new = torch.vstack(prediction_n)

        matchy = (preds_old.argmax(axis = 1)) == (preds_new.argmax(axis = 1))

        results[pred] = {
            'baseline_time': bl_time,
            "experimental_time": total_experimental_time,
            'n_match' : matchy.float().mean(),
            'total_shortcutted': total_shortcutted
        }

    df = pd.DataFrame.from_dict(results, orient = 'index')
    df.to_markdown('mnist_results.md')
```

Results

For each class, 0-9, we run the algorithm to create a simple rule to predict that class. We see, as expected, larger speed gains when the simple rules successfully picks up a large number of data points from within that class. We see speed slow downs in the case of class = 5, where the rule only picked up on 1598 5's.

	baseline_time	experimental_time	n_match	total_shortcutted
0	9.77313	8.48082	0.999983	5637
1	9.68577	8.03922	1	6304
2	8.69256	8.55458	1	4050
3	8.93277	8.72282	0.999983	4168
4	9.06341	8.73758	1	4510

	baseline_time	experimental_time	n_match	total_shortcutted
5	8.74312	8.93458	1	1598
6	9.21802	8.37652	0.999983	5240
7	8.90966	8.47124	0.999983	4804
8	8.78675	8.64843	0.999967	3569
9	9.0489	8.81263	0.999983	3561



The rule associated with the 1-class is the best, resulting in a -17% latency improvement.

Conclusion

- We see clearly that `moco` can immediately accelerate **CNN** for edge and at-scale applications, *without* risk of accuracy loss.
- We are improving the computational efficiency of ML models, which has additional implications for energy savings, implicating running CNN on an edge device and saving battery.