LLM inference

MLX enables efficient inference of large-ish transformers on Apple silicon without compromising on ease of use. In this example we will create an inference script for the Llama family of transformer models in which the model is defined in less than 200 lines of python.

Implementing the model

We will use the neural network building blocks defined in the `mlx.nn` module to concisely define the model architecture.

Attention layer

We will start with the Llama attention layer which notably uses the RoPE positional encoding. In addition, our attention layer will optionally use a key/value cache that will be concatenated with the provided keys and values to support efficient inference.

Our implementation uses `mlx.nn.Linear` for all the projections and `mlx.nn.RoPE` for the positional encoding.

```
import mlx.core as mx
import mlx.nn as nn

class LlamaAttention(nn.Module):
    def __init__(self, dims: int, num_heads: int):
        super().__init__()

        self.num_heads = num_heads

        self.rope = nn.RoPE(dims // num_heads, traditional=True)
        self.query_proj = nn.Linear(dims, dims, bias=False)
        self.key_proj = nn.Linear(dims, dims, bias=False)
        self.value_proj = nn.Linear(dims, dims, bias=False)
        self.out_proj = nn.Linear(dims, dims, bias=False)

    def __call__(self, queries, keys, values, mask=None, cache=None):
        queries = self.query_proj(queries)
        keys = self.key_proj(keys)
        values = self.value_proj(values)

        # Extract some shapes
        num_heads = self.num_heads
        B, L, D = queries.shape

        # Prepare the queries, keys and values for the attention computation
        queries = queries.reshape(B, L, num_heads, -1).transpose(0, 2, 1, 3)
        keys = keys.reshape(B, L, num_heads, -1).transpose(0, 2, 1, 3)
        values = values.reshape(B, L, num_heads, -1).transpose(0, 2, 1, 3)

        # Add RoPE to the queries and keys and combine them with the cache
        if cache is not None:
            key_cache, value_cache = cache
            queries = self.rope(queries, offset=key_cache.shape[2])
            keys = self.rope(keys, offset=key_cache.shape[2])
            keys = mx.concatenate([key_cache, keys], axis=2)
            values = mx.concatenate([value_cache, values], axis=2)
        else:
            queries = self.rope(queries)
            keys = self.rope(keys)

        # Finally perform the attention computation
        scale = math.sqrt(1 / queries.shape[-1])
        scores = (queries * scale) @ keys.transpose(0, 1, 3, 2)
        if mask is not None:
            scores = scores + mask
        scores = mx.softmax(scores, axis=-1)
        values_hat = (scores @ values).transpose(0, 2, 1, 3).reshape(B, L, -1)

        # Note that we return the keys and values to possibly be used as a cache
        return self.out_proj(values_hat), (keys, values)
```

Encoder layer

The other component of the Llama model is the encoder layer which uses RMS normalization and SwiGLU. For RMS normalization we will use `mlx.nn.RMSNorm` that is already provided in `mlx.nn`.

```
class LlamaEncoderLayer(nn.Module):
    def __init__(self, dims: int, mlp_dims: int, num_heads: int):
        super().__init__()

        self.attention = LlamaAttention(dims, num_heads)

        self.norm1 = nn.RMSNorm(dims)
        self.norm2 = nn.RMSNorm(dims)

        self.linear1 = nn.Linear(dims, mlp_dims, bias=False)
        self.linear2 = nn.Linear(dims, mlp_dims, bias=False)
        self.linear3 = nn.Linear(mlp_dims, dims, bias=False)

    def __call__(self, x, mask=None, cache=None):
        y = self.norm1(x)
        y, cache = self.attention(y, y, y, mask, cache)
        x = x + y

        y = self.norm2(x)
        a = self.linear1(y)
        b = self.linear2(y)
        y = a * mx.sigmoid(a) * b
        y = self.linear3(y)
        x = x + y

        return x, cache
```

Full model

To implement any Llama model we simply have to combine `LlamaEncoderLayer` instances with an `mlx.nn.Embedding` to embed the input tokens.

```
class Llama(nn.Module):
    def __init__(
        self, num_layers: int, vocab_size: int, dims: int, mlp_dims: int, num_heads:
int
    ):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, dims)
        self.layers = [
            LlamaEncoderLayer(dims, mlp_dims, num_heads) for _ in range(num_layers)
        ]
        self.norm = nn.RMSNorm(dims)
        self.out_proj = nn.Linear(dims, vocab_size, bias=False)

    def __call__(self, x):
        mask = nn.MultiHeadAttention.create_additive_causal_mask(x.shape[1])
        mask = mask.astype(self.embedding.weight.dtype)

        x = self.embedding(x)
        for l in self.layers:
            x, _ = l(x, mask)
        x = self.norm(x)
        return self.out_proj(x)
```

Note that in the implementation above we use a simple list to hold the encoder layers but using `model.parameters()` will still consider these layers.

Generation

Our `Llama` module can be used for training but not inference as the `__call__` method above processes one input, completely ignores the cache and performs no sampling whatsoever. In the rest of this subsection, we will implement the inference function as a python generator that processes the prompt and then autoregressively yields tokens one at a time.

```
class Llama(nn.Module):
    ...

    def generate(self, x, temp=1.0):
        cache = []

        # Make an additive causal mask. We will need that to process the prompt.
        mask = nn.MultiHeadAttention.create_additive_causal_mask(x.shape[1])
        mask = mask.astype(self.embedding.weight.dtype)

        # First we process the prompt x the same way as in __call__ but
        # save the caches in cache
        x = self.embedding(x)
        for l in self.layers:
            x, c = l(x, mask=mask)
            cache.append(c)  # <--- we store the per layer cache in a
                             #      simple python list
        x = self.norm(x)
        y = self.out_proj(x[:, -1])  # <--- we only care about the last logits
                                     #      that generate the next token
        y = mx.random.categorical(y * (1/temp))

        # y now has size [1]
        # Since MLX is lazily evaluated nothing is computed yet.
        # Calling y.item() would force the computation to happen at
        # this point but we can also choose not to do that and let the
        # user choose when to start the computation.
        yield y

        # Now we parsed the prompt and generated the first token we
        # need to feed it back into the model and loop to generate the
        # rest.
        while True:
            # Unsqueezing the last dimension to add a sequence length
            # dimension of 1
            x = y[:, None]

            x = self.embedding(x)
            for i in range(len(cache)):
                # We are overwriting the arrays in the cache list. When
                # the computation will happen, MLX will be discarding the
                # old cache the moment it is not needed anymore.
                x, cache[i] = self.layers[i](x, mask=None, cache=cache[i])
            x = self.norm(x)
            y = self.out_proj(x[:, -1])
            y = mx.random.categorical(y * (1/temp))

            yield y
```

Putting it all together

We now have everything we need to create a Llama model and sample tokens from it. In the following code, we randomly initialize a small Llama model, process 6 tokens of prompt and generate 10 tokens.

```
model = Llama(num_layers=12, vocab_size=8192, dims=512, mlp_dims=1024, num_heads=8)

# Since MLX is lazily evaluated nothing has actually been materialized yet.
# We could have set the `dims` to 20_000 on a machine with 8GB of RAM and the
# code above would still run. Let's actually materialize the model.
mx.eval(model.parameters())

prompt = mx.array([[1, 10, 8, 32, 44, 7]])  # <-- Note the double brackets because we
                                            #     have a batch dimension even
                                            #     though it is 1 in this case

generated = [t for i, t in zip(range(10), model.generate(prompt, 0.8))]

# Since we haven't evaluated anything, nothing is computed yet. The list
# `generated` contains the arrays that hold the computation graph for the
# full processing of the prompt and the generation of 10 tokens.
#
# We can evaluate them one at a time, or all together. Concatenate them or
# print them. They would all result in very similar runtimes and give exactly
# the same results.
mx.eval(generated)
```

Converting the weights

This section assumes that you have access to the original Llama weights and the SentencePiece model that comes with them. We will write a small script to convert the PyTorch weights to MLX compatible ones and write them in a NPZ file that can be loaded directly by MLX.

```
import argparse
from itertools import starmap

import numpy as np
import torch

def map_torch_to_mlx(key, value):
    if "tok_embedding" in key:
        key = "embedding.weight"

    elif "norm" in key:
        key = key.replace("attention_norm", "norm1").replace("ffn_norm", "norm2")

    elif "wq" in key or "wk" in key or "wv" in key or "wo" in key:
        key = key.replace("wq", "query_proj")
        key = key.replace("wk", "key_proj")
        key = key.replace("wv", "value_proj")
        key = key.replace("wo", "out_proj")

    elif "w1" in key or "w2" in key or "w3" in key:
        # The FFN is a separate submodule in PyTorch
        key = key.replace("feed_forward.w1", "linear1")
        key = key.replace("feed_forward.w3", "linear2")
        key = key.replace("feed_forward.w2", "linear3")

    elif "output" in key:
        key = key.replace("output", "out_proj")

    elif "rope" in key:
        return None, None

    return key, value.numpy()


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Convert Llama weights to MLX")
    parser.add_argument("torch_weights")
    parser.add_argument("output_file")
    args = parser.parse_args()

    state = torch.load(args.torch_weights)
    np.savez(
        args.output_file,
        **{k: v for k, v in starmap(map_torch_to_mlx, state.items()) if k is not None}
    )
```

Weight loading and benchmarking

After converting the weights to be compatible to our implementation, all that is left is to load them from disk and we can finally use the LLM to generate text. We can load numpy format files using the `mlx.core.load()` operation.

To create a parameter dictionary from the key/value representation of NPZ files we will use the `mlx.utils.tree_unflatten()` helper method as follows:

```
from mlx.utils import tree_unflatten

model.update(tree_unflatten(list(mx.load(weight_file).items())))
```

`mlx.utils.tree_unflatten()` will take keys from the NPZ file that look like `layers.2.attention.query_proj.weight` and will transform them to

```
{"layers": [..., ..., {"attention": {"query_proj": {"weight": ...}}}]}
```

which can then be used to update the model. Note that the method above incurs several unnecessary copies from disk to numpy and then from numpy to MLX. It will be replaced in the future with direct loading to MLX.

You can download the full example code in mlx-examples. Assuming, the existence of `weights.pth` and `tokenizer.model` in the current working directory we can play around with our inference script as follows (the timings are representative of an M1 Ultra and the 7B parameter Llama model):

```
$ python convert.py weights.pth llama-7B.mlx.npz
$ python llama.py llama-7B.mlx.npz tokenizer.model 'Call me Ishmael. Some years ago
never mind how long precisely'
[INFO] Loading model from disk: 5.247 s
Press enter to start generation
------
, having little or no money in my purse, and nothing of greater consequence in my mind,
I happened to be walking down Gower Street in the afternoon, in the heavy rain, and I
saw a few steps off, a man in rags, who sat upon his bundle and looked hard into the
wet as if he were going to cry. I watched him attentively for some time, and could not
but observe that, though a numerous crowd was hurrying up and down,
------
[INFO] Prompt processing: 0.437 s
[INFO] Full generation: 4.330 s
```

We observe that 4.3 seconds are required to generate 100 tokens and 0.4 seconds of those are spent processing the prompt. This amounts to a little over **39 ms per token**.

By running with a much bigger prompt we can see that the per token generation time as well as the prompt processing time remains almost constant.

```
$ python llama.py llama-7B.mlx.npz tokenizer.model 'Call me Ishmael. Some years ago never mind how long precisely, having little or no money in my purse, and nothing of greater consequence in my mind, I happened to be walking down Gower Street in the afternoon, in the heavy rain, and I saw a few steps off, a man in rags, who sat upon his bundle and looked hard into the wet as if he were going to cry. I watched him attentively for some time, and could not but observe that, though a numerous crowd was hurrying up and down, nobody took the least notice of him. I stopped at last, at a little distance, as if I had been in doubt, and after looking on a few minutes, walked straight up to him. He slowly raised his eyes, and fixed them upon me for a moment, without speaking, and then resumed his place and posture as before. I stood looking at him for a while, feeling very much pain at heart, and then said to him, “What are you doing there?” Something like a smile passed over his face, as he said slowly, “I am waiting for someone; but it has been three quarters of an hour now, and he has not come.” “What is it you are waiting for?” said I. Still he made no immediate reply, but again put his face down upon his hands, and did not'
[INFO] Loading model from disk: 5.247 s
Press enter to start generation
------
take his eyes from the ground. “What is it you are waiting for?” said I. “I am not accustomed to be thus questioned,” said he. “You look like a reasonable man—tell me, then, what are you waiting for?” “You would not understand,” he replied; “and how could you help me, if I were to tell you?” “I should not only understand, but would do all that I could,” said I. He did not
------
[INFO] Prompt processing: 0.579 s
[INFO] Full generation: 4.690 s
$ python llama.py --num-tokens 500 llama-7B.mlx.npz tokenizer.model 'Call me Ishmael. Some years ago never mind how long precisely, having little or no money in my purse, and nothing of greater consequence in my mind, I happened to be walking down Gower Street in the afternoon, in the heavy rain, and I saw a few steps off, a man in rags, who sat upon his bundle and looked hard into the wet as if he were going to cry. I watched him attentively for some time, and could not but observe that, though a numerous crowd was hurrying up and down, nobody took the least notice of him. I stopped at last, at a little distance, as if I had been in doubt, and after looking on a few minutes, walked straight up to him. He slowly raised his eyes, and fixed them upon me for a moment, without speaking, and then resumed his place and posture as before. I stood looking at him for a while, feeling very much pain at heart, and then said to him, “What are you doing there?” Something like a smile passed over his face, as he said slowly, “I am waiting for someone; but it has been three quarters of an hour now, and he has not come.” “What is it you are waiting for?” said I. Still he made no immediate reply, but again put his face down upon his hands, and did not'
[INFO] Loading model from disk: 5.628 s
Press enter to start generation
------
take his eyes from the ground. “What is it you are waiting for?” said I. “I am not accustomed to be thus questioned,” said he. “You look like a reasonable man—tell me, then, what are you waiting for?” “You would not understand,” he replied; “and how could you help me, if I were to tell you?” “I should not only understand, but would do all that I could,” said I. He did not reply, but still went on looking at the ground, and took hold of his bundle with a nervous trembling. I waited some time, and then resumed. “It is of no use to say you would not understand, if I were to tell you,” said he. “I have not told you why I am waiting for him,” said I. “And I am sure I should not understand,” replied he. “I will tell you then,” said I, “and, perhaps, you would not be surprised.” “No matter,” said he, “I shall be surprised anyhow; so tell me why you are waiting for him.” “He is my friend,” said I. “Yes,” said he, with a slight smile, “I know.” “He has been kind to me,” said I, “and I am waiting for him. I want to see
```

```
him, and could have waited as I am now, for a much longer time.” “He will not soon
come,” said he. “Unless he sees you here, he will not know of your having waited, and
he will be very unlikely to come.” “No matter,” said I, “I shall wait for him.” “This
is a strange thing,” said he, still with the same amused smile. “How did you know,”
said I, “that he was coming? How should you be waiting?” “That is my secret,” said he.
“And you expect him?” “Yes,” said I. “Are you disappointed then, if he does not come?”
“No,” said I, “it is his secret, not mine.” “If he comes,” said he, “do you mean to go
straight away?” “Yes,” said I, “I cannot be happy if I do not go straight away after
him.” “Did you know this place before?” asked he. “Yes,” said I. “Is there any shop to
buy food here?” “
------
[INFO] Prompt processing: 0.633 s
[INFO] Full generation: 21.475 s
```

Scripts

Download the code

The full example code is available in mlx-examples.

[]
Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. and Liu, Y., 2021. Roformer: Enhanced transformer with rotary position embedding. arXiv preprint arXiv:2104.09864.

[]
Zhang, B. and Sennrich, R., 2019. Root mean square layer normalization. Advances in Neural Information Processing Systems, 32.

[]
Shazeer, N., 2020. Glu variants improve transformer. arXiv preprint arXiv:2002.05202.