GPT2 in Numpy

April 20, 2025

1 Introduction

I thought it might be fun to write chat GPT in numpy. The plan is to slowly build out an autograd engine until we end up with GPT2. The repo for this project is here.

Figure 1: GPT2 architecture

First I implemented it in pytorch so I could see exactly what I needed. That implementation is here. I more or less copied Andrej Kaparhty's implementation from his video on it , except I stuck the multihead attention in a single module using `einsum`.

The way I want to set up the autograd engine is to have a `engine.py` file that contains a class called

7. `nn.CrossEntropyLoss`. Let Y be the target, and our prediction $\hat{Y} = \text{model}(X)$, then CE loss is given by

$$CE(Y, \hat{Y}) = -\frac{1}{N} \sum_{n,c} Y \odot \log\left(\text{softmax}(\hat{Y})\right) \tag{1}$$

   The haddamard product and sum can be handled by one einsum pattern `ij,ij->`, and we already have softmax above, so thats missing is `torch.log` which we can write a function for, and scalar multiplication which we can handle with the `__rmul__` dunder.

8. `nn.LayerNorm`. The equation is

$$ln = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta \tag{2}$$

   where γ, β are the same shape as the dimensions being normalized. This will require a `.mean()` method. We can get variance with

$$\text{var} = \mathrm{E}[x^2] - \mathrm{E}[x]^2 \tag{3}$$

   For all this we need the following dunders: `__pow__`, `__mul__`, `__add__`, `__sub__`. Instead of implemting `__div__` ill use negative exponents and multiplication so itll look like

```
    ln = (x - x.mean()) * ((x**2).mean() - (x.mean())**2 + eps) ** -0.5 ...stuff...
```

That covers it for functions. For class methods theres two things we need that have yet to crop up. Both are in the attention implementation. Here is the forward method, in torch.

```python
# torch #
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, embed_dim, num_heads) -> None:
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = int(embed_dim / num_heads)
        self.in_proj = nn.Linear(embed_dim, 3 * embed_dim, bias=True)
        self.out_proj = nn.Linear(self.head_dim * num_heads, embed_dim, bias=True)

    def forward(self, x):
        B, T, C = x.shape
        qvk = self.in_proj(x)
        q, v, k = tuple(
            rearrange(qvk, "b t (d k h) -> k b h t d", k=3, h=self.num_heads)
        )

        scaled_prod = einsum("bhid,bhjd->bhij", q, k) * (self.head_dim) ** -0.5

        mask = torch.tril(torch.ones_like(scaled_prod))
        scaled_prod = scaled_prod.masked_fill(mask == 0, -float("inf"))

        attention = torch.softmax(scaled_prod, dim=-1)
        out = einsum("bhij,bhjd->bhid", attention, v)
        out = rearrange(out, "b h t d -> b t (h d)")
        return self.out_proj(out)
```

First we need a way to unpack a `Parameter` along an axis, and second we need masking support. All together the class methods we need are:

1. `__add__`

2. `__sub__`
3. `__mul__`
4. `__pow__`
5. `__rmul__`
6. `mean()`
7. `split()`
8. `masked_fill()`

and then some utility stuff

9. `backward()`, something to trigger backpropogation
10. `sum()`, nice to have
11. `broadcast_helper`, broadcasting introduces some complications to backprop. Since addition, subtraction and multiplication all broadcast, we are going to make a seperate method get gradients for broadcasted operations.

Cool, we have our grocery list of methods and functions so lets get down to business

2 The class

The `backward()` method is going to dictate a lot of what we do here so lets settle that first. I liked the way Andrej Kaparthy laid it out in his micrograd video. Every `Parameter` will have a lambda function `_backward` that dictates how its gradient interacts with its children during backpropogation, and calling the class method `backward()` will call the `_backward` function of everything in the computational graph. For an example of how this should work

```
a, b = Parameter(2), Parameter(3)
c = a + b
d = c ** 2 - c

def d_backward():
  c.grad += 2 * c - 1

def c_backward():
  a.grad += c.grad
  b.grad += c.grad

d._backward = d_backward
c._backward = c_backward

d.backward()

print(a.grad, b.grad) # 9, 9, theoretically...
```

with that were ready to start

2.1 init and backward

```
import numpy as np
from typing import List, Tuple
```

```
class Parameter:
    def __init__(self, data, _children=()) -> None:
        self.data = = data
        self.grad = np.zeros_like(self.data)
        self._children = _children
        self.shape = self.data.shape
        self.dim = len(self.shape) if self.shape else 0
        self._backward = lambda: None

    def backward(self):
        assert self.grad.shape == ()
        self.grad = 1.0
        visited, stack = set(), []

        def dfs(node):
            visited.add(node)
            for child in node._children:
                if child not in visited:
                    dfs(child)
            stack.append(node)

        dfs(self)

        for node in stack[::-1]:
            node._backward()
```

For backward we first assert that its a scalar, and then call the `_backward` method of everything in the reverse order of the computational graph.

2.2 split

We want to be able to split a `Parameter`, do stuff to the children, and have the gradient backpropogate correctly.

```
    def split(self, dim=0) -> List["Parameter"]:
        data = np.moveaxis(self.data, dim, 0)
        kids = []
        for idx, slice in enumerate(data):
            kid = Parameter(slice, _children=(self,))

            def _undo_split(idx=idx, kid=kid):
                np.moveaxis(self.grad, dim, 0)[idx] += kid.grad

            kid._backward = _undo_split
            kids.append(kid)
        return kids
```

Using `np.moveaxis` to bring the split axis to the zero dimension, we split the `Parameter` into children. To backpropogate we use `np.moveaxis` with the reverse operands to return a view, and then just add the child grad.

2.3 masked fill

```
    def masked_fill(self, mask: np.ndarray, value: float) -> "Parameter":
        out_data = np.copy(self.data)
        out_data[mask] = value
        out = Parameter(out_data, _children=(self,))

        def _backward():
            masked_grad = np.copy(out.grad)
            masked_grad[mask] = 0
            self.grad += masked_grad

        out._backward = _backward
        return out
```

For this, any value that gets masked has a zero gradient. For mask you would pass a boolean array like `x == 0`

2.4 sum

```
    def sum(self, dim=None, keepdim=False) -> "Parameter":
        out = Parameter(self.data.sum(axis=dim, keepdims=keepdim), _children=(self,))

        def _backward():
            self.grad += (
                np.expand_dims(out.grad, dim)
                if (dim is not None and not keepdim)
                else out.grad
            )

        out._backward = _backward
        return out
```

When `keepdim == True` dims are summed to be 1, so there are no broadcasting issues. If its false, suppose `a.shape = [2, 3, 4]`, and `c = a.sum(-1, 1)`, then backward pass will have `a.grad += c.grad`, which bricks since adding shapes `[2, 3, 4] += [3]` isnt valid. The solution is expand collapsed dims to 1.

2.5 mean

```
    def mean(self, dim: Tuple[int], keepdim=True) -> "Parameter":
        m = np.mean(self.data, dim, keepdims=keepdim)
        out = Parameter(m, _children=(self,))

        def _backward():
            original_shape = [int(_) for _ in self.data.shape]
            new_shape = [original_shape[d] for d in dim]
            out_grad = out.grad if keepdim else np.expand_dims(out.grad, dim)
            self.grad += out_grad / np.prod(new_shape)

        out._backward = _backward
        return out
```

The grad for mean is just

$$\frac{\partial}{\partial X}[\frac{1}{N}\sum X] = \frac{1}{N} \tag{4}$$

and we use the same reshaping we used for `sum`

2.6 dunder

Lets first solve the broadcasting issue. Lets say somewhere in our network we have `a, b` of shapes `[2, 3, 4], [2, 3]`, and `c = a + b`. The backward pass will require

```
 a.grad += c.grad
 b.grad += c.grad
```

but line 2 fails since `[2, 3, 4]` cant be broadcast into `[2, 3]`. So what is the grad supposed to be? When `a, b` get added, the broadcasting adds `b` to each array along the 0'th dimension of `a`. So the gradient w.r.t `a` is `c.grad` summed along the 0'th dimension. In general, to get the grad w.r.t. the broadcasted operand you just sum the grad from the left until it has the same shape as the operand.

Theres one more case to handle. If we had `a, b` of shapes `[2, 3, 4], [2, 1, 4]`, we'll throw a broadcasting error in the backward pass since `[2, 3, 4]` can be broadcast into `[2, 1, 4]` The solution is sum `c.grad` to 1 along the 1'th dimension. `[2, 3, 4] -> [2, 1, 4]`. In general, the grad has to be summed to 1 in whichever dims the broadcasted operand has dimension length 1.

combinging both these cases

```
    @staticmethod
    def broadcast_helper(grad: np.ndarray, a: np.ndarray) -> np.ndarray:
        if grad.shape == a.shape:
            return grad
        else:
            sum_dims = tuple(range(len(grad.shape) - len(a.shape)))
            sum_to_one = tuple(_ for _, __ in enumerate(a.shape) if __ == 1)
            return grad.sum(sum_dims).sum(sum_to_one, keepdims=True)
```

First we sum from the left until `grad` and `a` have the same number of dimensions, then whichever dims have length 1 in `a` get summed to 1 in `grad`. With this out of the way we can write our dunders. These are all straight forward so ill show addition and multiplication. `__pow__` doesnt have any broadcasting and the implementation is exactly what you expect.

```
    def __add__(self, other) -> "Parameter":
        other = other if isinstance(other, Parameter) else Parameter(other)
        out = Parameter(self.data + other.data, _children=(self, other))

        def _backward():
            self.grad += self.broadcast_helper(out.grad, self.grad)
            other.grad += self.broadcast_helper(out.grad, other.grad)

        out._backward = _backward
        return out

    def __mul__(self, other: "Parameter") -> "Parameter":
        out = Parameter(self.data * other.data, _children=(self, other))

        def _backward():
            self.grad += self.broadcast_helper(out.grad * other.data, self.grad)
            other.grad += self.broadcast_helper(out.grad * self.data, other.grad)

        out._backward = _backward
        return out
```

3 Functions

I said this was going to be in numpy but I guess I lied.

```
import numpy as np
from einops import repeat
```

```python
from einops import rearrange as erearrange
import string
from .engine import Parameter
from scipy.stats import norm
```

SciPy is for `GELU`, and the einops stuff is because I'm going to implement a rearrange function for `Parameter` that just calls the einops version lol. We've built out far enough that a lot of the function implementations are straightforward.

3.1 einsum

I have a more detailed post about how this is calculated here, but heres the final code

```python
def einsum(ptrn: str, *args: Parameter) -> Parameter:
    out = Parameter(np.einsum(ptrn, *[_.data for _ in args]), _children=tuple(args))

    def _backward():
        in_ptrn, out_ptrn = ptrn.split("->")
        in_ptrns = in_ptrn.split(",")
        if not out_ptrn:
            out_ptrn = "".join(list(set(string.ascii_lowercase) - set(in_ptrn))[0])
            temp_out_grad = np.expand_dims(out.grad, 0)
        else:
            temp_out_grad = out.grad

        def calc_grad(idx):
            op_ptrn, op = in_ptrns[idx], args[idx]
            other_op_ptrns = in_ptrns[:idx] + in_ptrns[idx + 1 :]
            known_dims = "".join(
                [c for c in op_ptrn if c in "".join(other_op_ptrns) + out_ptrn]
            )
            grad_string = f"{out_ptrn},{','.join(other_op_ptrns)}->{known_dims}"
            if not other_op_ptrns:
                grad_string = grad_string.replace(",", "")
            grad = np.einsum(
                grad_string, temp_out_grad, *[_.data for _ in args if _ != op]
            )
            if known_dims != op_ptrn:
                expand_dims = tuple(
                    _ for _, __ in enumerate(op_ptrn) if __ not in known_dims
                )
                grad = np.expand_dims(grad, expand_dims)
            return grad

        for idx, arg in enumerate(args):
            arg.grad += calc_grad(idx)

    out._backward = _backward
    return out
```

For an einsum pattern like `ij,jk->ik`, the grad w.r.t. operand 0 is `ik,jk->ij`. Problems arise if you sum to a scalar. E.g. `ij,jk->` would reverse to `,jk-ij`.

This bricks becasue operand zero needs a string, and the `i` in the output string is unknown. The first problem we solve by making grad 1d if its scalar, and assigning it a unique letter. We solve the second issue by making the out string of our reversed einsum only the known dimensions. i.e. the dimensions of the operand contained in the other operands or output. So the grad string of ij,jk-> would be q,jk->j, where q is an arbitrary letter. This is of course the wrong shape but we solve by expanding to 1 along the missing dimensions, and then the gradient broadcasts with no issues.

This works for any number of arguments, im pretty sure.

At this point we are basically done, really. We have core operations in our Parameter class methods and we just added support for arbitrary einsum operations. From here on out we dont even need to think. Softmax is just

```python
def exp(x: Parameter) -> Parameter:
    out = Parameter(np.exp(x.data), _children=(x,))

    def _backward():
        x.grad += out.data * out.grad

    out._backward = _backward
    return out


def softmax(x: Parameter, dim=-1) -> Parameter:
    e = exp(x)
    out = e * (e.sum(dim, keepdim=True) ** -1)
    return out
```

and cross entropy is just

```python
def log(x: Parameter) -> Parameter:
    out = Parameter(np.log(x.data), _children=(x,))

    def _backward():
        x.grad += out.grad * (1 / x.data)

    out._backward = _backward
    return out


def cross_entropy_loss(x: Parameter, y: Parameter, dim=-1) -> Parameter:
    if any([_.data.dtype != np.float64 for _ in (x, y)]):
        raise TypeError("cross entropy takes float64")
    log_soft = log(softmax(x, dim=dim))
    ptrn = string.ascii_lowercase[: len(x.data.shape)]
    return (float(-x.data.shape[0]) ** -1) * einsum(f"{ptrn},{ptrn}->", log_soft, y)
```

I'm going to leave out the rest of the functions, since the implementation is exactly what you expect.

4 GPT

At this point I wrote a bunch of code to make class wrappers for our functions so they feel more like pytorch. For example heres the embedding

```python
class Embedding(Module):
    def __init__(self, num_embeddings, embedding_dim) -> None:
        super().__init__()
        self.weight = Parameter(shape=(num_embeddings, embedding_dim))

    @classmethod
    def _from_torch(cls, x: torch.nn.Module) -> "Module":
        self = cls.__new__(cls)
        attrs = [("weight", self._weight_to_param)]
        return self.set_attrs(x, self, attrs)

    def forward(self, x: Parameter):
        return embed(x.data, self.weight)
```

The `_from_torch` method allows me to initialize an `Embedding` object from a `nn.Embedding` by copying+detaching its weight, converting it to a `Parameter`, and assigning it to the object. The `_weight_to_param` does what the name implies, and theres another method `_do_nothing` used for assigning non-learnable attributes like softmax dim, layer norm shape, ...etc. Finally in theres a global function `convert_nn_module` that converts `nn.Module`'s to their `Parameter` based equivalent by checking a dictionary.

```python
import torch.nn as nn
CONVERSION_DICT = {
    nn.Linear: Linear,
    nn.Softmax: Softmax,
    nn.CrossEntropyLoss: CrossEntropyLoss,
    nn.ReLU: ReLU,
    nn.GELU: GELU,
    nn.Embedding: Embedding,
    nn.Dropout: Dropout,
    nn.LayerNorm: LayerNorm,
    nn.Sequential: Sequential,
    r.MultiHeadSelfAttention: MultiheadAttention,
    r.Block: Block,
}


def convert_nn_module(x: nn.Module):
    return CONVERSION_DICT[type(x)]._from_torch(x)
```

The final GPT implementation looks alot like the original pytorch implementation.

```python
class MultiheadAttention(Module):
    def __init__(self, embed_dim, num_heads) -> None:
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = int(embed_dim / num_heads)
        self.in_proj = Linear(embed_dim, 3 * embed_dim, bias=True)
        self.out_proj = Linear(embed_dim, embed_dim, bias=True)

    @classmethod
    def _from_torch(cls, x: nn.Module):
        self = cls.__new__(cls)
        attrs = [
            ("in_proj", convert_nn_module),
            ("out_proj", convert_nn_module),
            ("embed_dim", self._do_nothing),
            ("num_heads", self._do_nothing),
            ("head_dim", self._do_nothing),
        ]
        return self.set_attrs(x, self, attrs)

    def forward(self, x: Parameter):
        qvk = self.in_proj(x)
        qvk = rearrange(qvk, "b t (d k h) -> k b h t d", k=3, h=self.num_heads)
        q, v, k = qvk.split(0)
        scaled_product = (self.head_dim**-0.5) * einsum("bhid,bhjd->bhij", q, k)

        mask = np.tril(np.ones_like(scaled_product.data))
        scaled_product = scaled_product.masked_fill(mask == 0, -np.inf)

        attention = softmax(scaled_product, dim=-1)
        out = einsum("bhij,bhjd->bhid", attention, v)
        out = rearrange(out, "b h t d -> b t (h d)", h=self.num_heads, d=self.head_dim)
        return self.out_proj(out)

    def parameters(self):
        return self.in_proj.parameters() + self.out_proj.parameters()


class Block(Module):
    def __init__(self, embed_dim, num_heads, p=0.0):
        super().__init__()
        self.ln1, self.ln2 = [
            LayerNorm(normalized_shape=(embed_dim,)) for _ in range(2)
        ]
        self.attn = MultiheadAttention(embed_dim, num_heads)

        self.mlp = Sequential(
            Linear(embed_dim, embed_dim * 4),
            GELU(),
            Linear(embed_dim * 4, embed_dim),
            Dropout(p),
        )

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x

    @classmethod
    def _from_torch(cls, x: nn.Module):
        self = cls.__new__(cls)
        attrs = [
            ("ln1", convert_nn_module),
            ("ln2", convert_nn_module),
            ("attn", convert_nn_module),
            ("mlp", convert_nn_module),
        ]
        return self.set_attrs(x, self, attrs)

    def parameters(self):
        return (
            self.ln1.parameters()
            + self.ln2.parameters()
            + self.attn.parameters()
            + self.mlp.parameters()
        )


class GPT(Module):
    def __init__(self, vocab_size, embed_dim, num_heads, seq_length, n_blocks) -> None:
        super().__init__()
        self.token_embedding = Embedding(vocab_size, embed_dim)
        self.position_embedding = Embedding(seq_length, embed_dim)
        self.blocks = Sequential(
            *[Block(embed_dim, num_heads) for _ in range(n_blocks)]
        )
        self.ln_f = LayerNorm((embed_dim,))
        self.lm_head = Linear(embed_dim, vocab_size)

    def forward(self, idx: Parameter):
        B, T = idx.shape
        tok_emb = self.token_embedding(idx)
        pos_emb = self.position_embedding(Parameter(np.arange(T)))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)
        return logits

    @classmethod
    def _from_torch(cls, x: nn.Module):
        self = cls.__new__(cls)
        attrs = [
            ("token_embedding", convert_nn_module),
            ("position_embedding", convert_nn_module),
            ("blocks", convert_nn_module),
            ("ln_f", convert_nn_module),
            ("lm_head", convert_nn_module),
        ]
        return self.set_attrs(x, self, attrs)
```

And thats it. We got GPT2 running in numpy. I tested this with no dropout, since I dont know how to synchronize random states between numpy and torch, and the otuputs/gradients are all identical. Pretty dope.