大作业

一个简单的深度学习系统

姓名：刘浩男

学号：24011211245

引入

为了更好的理解什么为什么深度学习可以正常的工作，所以制作了一个简单的可以进行深度学习的框架，当然，其十分的简陋，速度很慢，但是可以很好的了解深度学习的每个环节，以及他们深度学习每个环节在干什么。当然，可以更好的理解why,即深度学习为什么要这么做。自己搭建之后，深度学习的环节是自然的。

自动求导的计算方法

在深度学习的网络训练的过程当中，我们总是需要一步一步的进行正向传播和反向传播，并且在这个过程之中不可避免的涉及到一次又一次的求导的过程，所以弄清楚自动求导的过程是非常有必要的。涉及到求导的过程，感觉可以分成如下的几种常见的求导方法。

1. 手动根据数学知识写出求导的公式，然后将其转化为计算机实现的函数实现。
2. 利用导数的原始定义，通过有限差分近似方法完成求导，直接求解微分值。
3. 基于数学规则和程序表达式变换完成求导。利用求导规则对表达式进行自动计算，其计算结果是导函数的表达式而非具体的数值。即，先求解析解，然后转换为程序，再通过程序计算出函数的梯度.
4. 结合上溯几个方法的融合方法。

首先第一种方法是手动纸上计算的，不适合计算机的计算。第二种方法是数值计算，其核心的公式是：

$$df(x) = lim_{h->0}\frac{f(x+h)-f(x)}{h}$$

显然，我们可以通过如下的一个简单的Python程序来查看其效果

```
def num_df(x : float , h : float , f: Callable[[float],float]):
    return (f(x+h) - f(x)) /h
def test_num_df(h : float)->list[float]:
    squre_def = lambda x : x**3
    arrlen = 20
    x_arr = [i for i in range(arrlen)]
    fx_arr = [squre_def(x) for x in x_arr]
    x_h_arr = [x+h for x in x_arr ]
    fx_h_arr = [squre_def(x_h) for x_h in x_h_arr]
    dx = []
    for i in range(arrlen):
        dx.append((fx_h_arr[i] - fx_arr[i]) / h)
    return dx
```

通过运行这个程序可以得到其结果和理论计算的结果有偏差，根据理论上的分析可以知道其结果为

$$dx = 3x^2 + 3hx + h^2$$

和标准的 $3x^2$ 有误差，并且误差和h有关。增大h的精度可以求出更好的结果，但是增加精度会扩大h的数据位宽，增大占用的内存，减缓执行的速度。这是想到了方法3,对于简单可以很好求导的函数，可以利用先验知识简单的求出其导数，无需数值求解。常见的导数如下:

$$d(x^n) = nx^{n-1}dx$$

$$d(ln(x)) = \frac{1}{x}dx$$

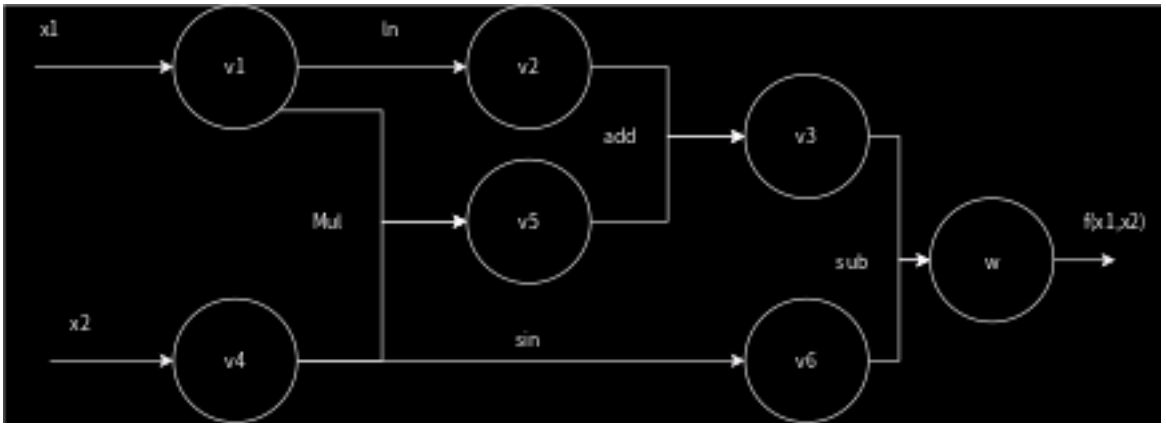
这样以来两种结合使用便可以很好的解决求导的误差问题。

自动求导的计算模式

对于神经网络的计算而言，可以分成简单的前向模式或者前向累积梯度和反向模式或者说反向累计梯度。下面以

$$f(x_1, x_2) = lnx_1 + x_1x_2 - sin(x_2)$$

这个函数为例子，首先画出其计算图。



根据简单的求导法则可以得到

$$\frac{dw}{dx_1} = \frac{dv_1}{dx_1}(\frac{dv_2}{dv_1}\frac{dv_3}{dv_2} + \frac{dv_5}{dv_1}\frac{dv_3}{dv_5})\frac{dw}{dv_3}$$

首先从前向的梯度计算开始，即计算图从左向右计算梯度，下面举一个简单的例子，以简单的输入 $(x_1, x_2) = (2, 5)$ 为例子，首先是简单的节点的数值计算。

1. 首先是赋值 $v_1 = x_1 = 2$,$v_4 = x_2 = 5$
2. 其次计算 $v_2 = ln(v_1) = ln2$
3. 其次计算 $v_5 = v_1 v_4 = 10$
4. 其次计算 $v_3 = v_2 + v_5 = 10 + ln2$
5. 其次计算 $v_6 = sin(v_4) = sin5$
6. 最后计算 $w = v_3 - v_6 = 10 + ln2 - sin5$

然后是对应的前向梯度的计算，其过程如下:

1. 由于 $v_1 = x_1$ 所以可以得到 $\frac{dv_1}{dx_1} = 1$
2. 由于 $v_4 = x_2$ 所以可以得到 $\frac{dv_4}{dx_1} = 0$
3. 计算 $v_2 = ln(v_1) = ln(x_1)$ 所以可以得到 $\frac{dv_2}{dx_1} = \frac{1}{2}$
4. 计算 $dv_5 = \frac{dv_1}{dx_1}v_4 + v_1\frac{dv_4}{dx_1} = 5dx_1$
5. 计算 $dv_6 = 0dx_1$
6. 最后计算 $\frac{dw}{dx_1} = 5.5$

通过上面的这个例子可以看出前向梯度计算有如下的优缺点:

- 实现起来很简单；
- 也不需要很多额外的内存空间。
- 每次前向计算只能计算对一个自变量的偏导数，对于一元函数求导是高效的，但是机器学习模型的自参数（入参）数量级大。
- 如果有一个函数，其输入有 n 个，输出有 m 个，对于每个输入来说，前向模式都需要遍历计算过程以得到当前输入的导数，求解整个函数梯度需要 n 遍如上计算过程。

由于前向梯度计算有上述的优缺点，所以可见下面的反向求导数的方法：反向模式根据从后向前计算，依次得到对每个中间变量节点的偏导数，直到到达自变量节点处，这样就得到了每个输入的偏导数。在每个节点处，根据该节点的后续节点（前向传播中的后续节点）计算其导数值。就上面的计算图来说，从 w 到 x_1 可以明显的分成两条路径:

- $w->v_3->v_2->v_1->x_1$
- $w->v_3->v_5->v_1->x_1$

通过两条路径求导可以得到第一条路径得到的导数数值为0.5,第二条得到的导数数值为5。综上，可以得到的数值为5.5 同理对于 $\frac{dw}{dx_2}$ 的导数也可以得到，显然和上面的前向求导相比，反向传播有如下的优势:

- 通过一次反向传输，就计算出所有偏导数，中间的偏导数计算只需计算一次。
- 减少了重复计算的工作量，在多参数的时候后向自动微分的时间复杂度更低。
- 需要额外的数据结构记录正向过程的计算操作，用于反向使用；
- 带来了大量内存占用，为了减少内存操作，需要 AI 框架进行各种优化，也带来了额外限制和副作用。

自动求导的实现

下面是自动求导的实现:

```
import numpy as np


class Value:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op
    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+')

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

        return out

    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other), '*')

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

        return out

    def __pow__(self, other):
        assert isinstance(other, (int, float)), "不是int,float类型"
        out = Value(self.data**other, (self,), f'**{other}')

        def _backward():
            self.grad += (other * self.data**(other-1)) * out.grad
        out._backward = _backward

        return out

    def relu(self):
        out = Value(0 if self.data < 0 else self.data, (self,), 'Re

        def _backward():
            self.grad += (out.data > 0) * out.grad
        out._backward = _backward

        return out

    def backward(self):
```

```
        topo = []
        visited = set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topo(child)
                topo.append(v)
        build_topo(self)

        self.grad = 1
        for v in reversed(topo):
            v._backward()

    def __neg__(self):
        return self * -1

    def __radd__(self, other):
        return self + other

    def __sub__(self, other):
        return self + (-other)

    def __rsub__(self, other):
        return other + (-self)

    def __rmul__(self, other):
        return self * other

    def __truediv__(self, other):
        return self * other**-1

    def __rtruediv__(self, other):
        return other * self**-1

    def __repr__(self):
        return f"Value(data={self.data}, grad={self.grad})"
```

通过测试发现其结果是正确的测试的代码是test_engine.py并且回顾之前的过程可以发现我们输出了一组初值，得到了对应的梯度数值。一个更进一步的思想，如果得到的梯度为0,会怎么样?下面对 $x^2 + y^2$ 展开运算

```
def grad_show():
    fx_grad_arr = []
    for i in range(10):
        for j in range(10):
            x = -1 + i/5
            y = -1 + j/5
            x = Value(x)
            y = Value(y)
            v1 = x**2
            v2 = y**2
            out = v1 + v2
            out.backward()
            fx_grad = (x.data,y.data,x.grad,y.grad,out.data)
            fx_grad_arr.append(fx_grad)
    minindex ,_ = min(enumerate(fx_grad_arr) , key=lambda x : x[1][
    for i in fx_grad_arr:
        print(i)
    print(fx_grad_arr[minindex])
```

可以发现其最小数的点落在(0,0)但是寻找这个点消耗了大量的资源，上面采用的这个最小数值点进行了100次运算，为了减少计算的量，采用了SGD,随机梯度下降。及每次采用如下的方法进行迭代

$$x_{i+1} = x_i - \mu \frac{df}{dx_i}$$

下面是实例程序

```
def SGD_test(lr : float = 0.3):
    x1 = Value(1)
    x2 = Value(1)
    for _ in range(10):
        v1 = x1 ** 2
        v2 = x2 ** 2
        out = v1 + v2
        out.backward()
        x1.data = x1.data - lr * x1.grad
        x2.data = x2.data - lr * x2.grad
        ## 重要 防止梯度累计
        x1.grad = 0
        x2.grad = 0
        print(x1.data)
        print(x2.data)

if __name__ == "__main__":
    SGD_test()
```

通过运行程序可以发现只经过10次迭代预测数值和真实数值之间的误差便达到了千分之一，效果很好。但是在大多数的情况下，我们是不知道f的，f才是需要通过神经网络计算的量。多数情况是我们知道 $(x_1, y_1)(x_2, y_2), ...(x_n, y_n)$ 需要找到对应的 $\hat{y} = f(x)$ 让f(x)最接近y,这个

时候需要有一个评判标准，所以引入了损失函数。同理，对于不同的任务来说，损失函数的设计有好有坏，为了设计一个简单的损失函数，首先想到的是在最小二乘法中引入的表示预测的数值和真实数值之间的衡量标准，其形式如下

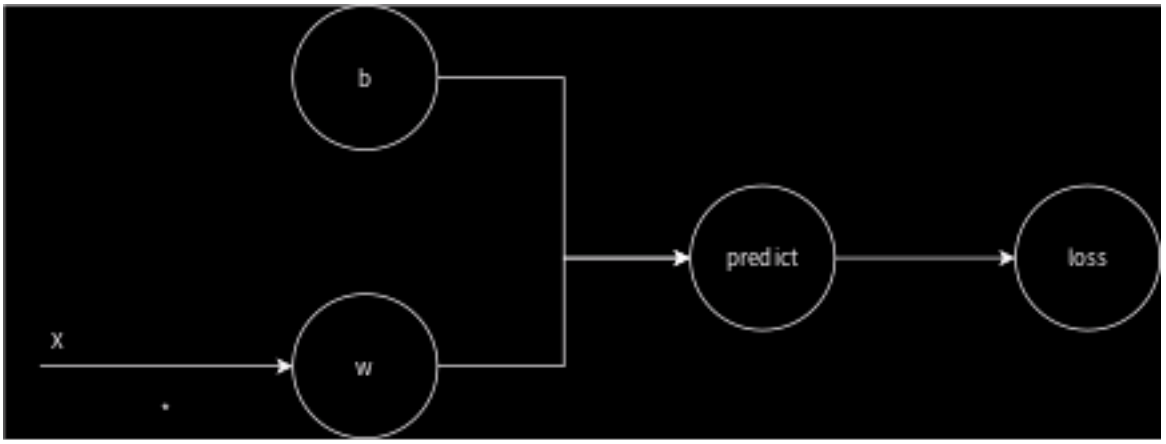
$$Loss = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2$$

为了验证其有效性，首先采取其来验证 $y = wx + b$ 这样的线性模型，采用随机梯度下降的方法，观察其是否会逐步的收敛到y, 下面便使用一个简短的测试用例

```
def generate_data(num_samples=100):
    rng = jax.random.PRNGKey(0)
    X = jax.random.normal(rng, (num_samples, 1))  # 输入数据
    true_w = jnp.array([[2.0]])
    true_b = jnp.array([5.0])
    noise = 0.1 * jax.random.normal(rng, (num_samples, 1))
    y = jnp.dot(X, true_w) + true_b + noise
    return X, y

# 初始化权重和偏置
def init_params(key, input_size,output_size):
    key1,key2 = jax.random.split(key, 2)
    w1 = normal(key1, (input_size, output_size)) * 0.1
    b1 = jnp.zeros((output_size,))
    return [w1, b1]

# 定义MLP网络
def mlp(params, X):
    w1, b1,= params
    output = jnp.dot(X, w1) + b1  # 输出层
    return output

# 损失函数（均方误差）
def loss_fn(params, X, y):
    preds = mlp(params, X)
    return jnp.mean((preds - y) ** 2)

# 梯度下降优化
def update_params(params, grads, lr=0.01):
    return [p - lr * g for p, g in zip(params, grads)]

# 主函数
def train_linear_regression():
    # 超参数
    num_samples = 100
    input_size = 1
    output_size = 1
    num_epochs = 100
    lr = 0.01

    # 数据生成
    X, y = generate_data(num_samples)

    # 初始化参数
    key = PRNGKey(42)
    params = init_params(key, input_size,output_size)

    # 编译加速
    loss_fn_jit = jit(loss_fn)
    grad_fn = jit(grad(loss_fn))

    # 训练循环
    for epoch in range(num_epochs):
```

```
            # 计算梯度
            grads = grad_fn(params, X, y)

            # 更新参数
            params = update_params(params, grads, lr)

            # 打印损失
            current_loss = loss_fn_jit(params, X, y)
            print(f"Epoch {epoch + 1}, Loss: {current_loss:.4f}")
            print(f"w :{params[0]} , b:{params[1]}")

        return params

    if __name__ == "__main__":
        trained_params = train_linear_regression()
```

通过100次迭代之后，可以发现w=1.92 b=4.32 和真实的w=2 ,b=5比较接近了。可以预见这种方法是有效的，并且更进一步的，和之前梯度下降迭代对应的输入x不同，这里采用梯度更新的是参数w,需要优化的是loss函数，希望找到loss函数的极小值点，这样预测的f(x)就趋近与真实数值y。同时观察loss的数值可以发现在线性回归任务中loss是严格减少的(其他的任务不一定，严格减少是很罕见的情况)，可以表明符合预期。



但是，通过观察上面的这个神经网络可以发现其还有明显的缺陷。

1. 只能拟合线性的函数，非线性的函数拟合不了。
2. 收敛的速度很慢，经过100次才收敛到这个数值，loss下降慢(当然可以通过调超参数解决收敛很慢的问题，但是需要不停调参的网络可以认为是不好的)

于是隐藏层出现，(数学上来说解释这个为什么work很困难，但是姑且认为他work就可以了)，先得到一些中间的输出，在由这些中间的输出得到最终的输出，这就是MLP。

```
import jax
import jax.numpy as jnp
from jax import grad, jit, vmap
from jax.random import PRNGKey, normal

# 生成数据
def generate_data(num_samples=100):
    rng = jax.random.PRNGKey(0)
    X = jax.random.normal(rng, (num_samples, 1))  # 输入数据
    true_w = jnp.array([[2.0]])
    true_b = jnp.array([5.0])
    noise = 0.1 * jax.random.normal(rng, (num_samples, 1))
    y = jnp.dot(X, true_w) + true_b + noise
    return X, y

# 初始化权重和偏置
def init_params(key, input_size, hidden_size, output_size):
    key1, key2, key3 = jax.random.split(key, 3)
    w1 = normal(key1, (input_size, hidden_size)) * 0.1
    b1 = jnp.zeros((hidden_size,))
    w2 = normal(key2, (hidden_size, output_size)) * 0.1
    b2 = jnp.zeros((output_size,))
    return [w1, b1, w2, b2]

# 定义MLP网络
def mlp(params, X):
    w1, b1, w2, b2 = params
    hidden = jax.nn.relu(jnp.dot(X, w1) + b1)  # 隐藏层
    output = jnp.dot(hidden, w2) + b2  # 输出层
    return output

# 损失函数（均方误差）
def loss_fn(params, X, y):
    preds = mlp(params, X)
    return jnp.mean((preds - y) ** 2)

# 梯度下降优化
def update_params(params, grads, lr=0.01):
    return [p - lr * g for p, g in zip(params, grads)]

# 主函数
def train_linear_regression():
    # 超参数
    num_samples = 100
    input_size = 1
    hidden_size = 10
    output_size = 1
    num_epochs = 1000
    lr = 0.01

    # 数据生成
    X, y = generate_data(num_samples)
```

```
    # 初始化参数
    key = PRNGKey(42)
    params = init_params(key, input_size, hidden_size, output_size)

    # 编译加速
    loss_fn_jit = jit(loss_fn)
    grad_fn = jit(grad(loss_fn))

    # 训练循环
    for epoch in range(num_epochs):
        # 计算梯度
        grads = grad_fn(params, X, y)

        # 更新参数
        params = update_params(params, grads, lr)

        # 打印损失
        if epoch % 100 == 0 or epoch == num_epochs - 1:
            current_loss = loss_fn_jit(params, X, y)
            print(f"Epoch {epoch + 1}, Loss: {current_loss:.4f}")

    return params

if __name__ == "__main__":
    trained_params = train_linear_regression()
```

通过这个简单的流程可以附带的引出许多有意思的问题，正如同费曼名言：凡不是我创作的，我无法理解。通过一个简单的深度学习流程的搭建可以对于深度学习有一个更加深刻的理解。譬如在上面的过程中所有的操作都是很低效的。并且就这个简单的系统而言可以发现在深度学习中的大多数运算都和矩阵乘法相关，并且需要建立计算图。在这个基础上现代的深度学习的框架也衍生出了动态图和静态图的概念，并且通过一步一步的扩大参数量可以发现在线性回归上的收敛速度愈来愈快。最主要的是了解了反向传播的具体实现，虽然这个实现不是很高效。但是至少建立了反向传播是如何传播的。就这个例子中的反向传播而言，姑且可以认为是构建了一个树，而loss节点就是树的根节点，利用链式求导法则不断的去构建一个反向传播的求导树，不断的更新导数。并且和数学上的求导做了区分，数学上的求导(数值求导)可以说是会带来一定的误差的，自动求导是基于算子的求导，加法的算子求导，乘法的算子求导，是数学上预先计算好的。

综上，便是本次大作业