面向对象的程序设计(C++)

CPPBP: 高扩展性的神经网络框架

姓　　名	熊恪峥　杨夏婕　徐杰 黄彬茹
学　　号	22920202204622 2292020220786 22920202204623 22920202202780
日　　期	2022年6月1日
学　　院	信息学院
课程名称	算法设计与分析

CPPBP: 高扩展性的神经网络框架

目录

CPPBP: 高扩展性的神经网络框架

熊恪峥 杨夏婕 徐杰 黄彬茹
22920202204622 2292020220786 22920202204623 22920202202780

摘要 反向传播算法在深度学习领域有着重要的地位。它是训练一个神经网络的理论基础。由全连接层组成的MLP是一种基本的神经网络结构。同时，近两年来 [1]、 [2] 证实了纯MLP组成的网络结构，在计算机视觉领域并不是完全低效的，可以在分类任务上达到卷积神经网络级别的性能。并且更少的归纳偏置带来了更强大的拟合能力和泛化能力。本项目实现了一个静态建图的、高可扩展性的基于反向传播算法的神经网络框架*cppbp*，提供了统一的接口以保证代码的简洁可读，以及常用的优化器、损失函数、正则化方法的实现。代码已经在`https://github.com/SmartPolarBear/cppbp`上开源。

1 架构设计

架构设计是提高可扩展性的重要保障。我们首先对该框架的需求进行了合理划分，如图 1。我们将整体工作流划分为模型构建、数据处理、训练与测试、可持久化与部署四个部分。

图 1: 模块划分

这四个部分分别完成模型的搭建与反向传播的处理；数据的读入、打乱、增强；训练流程搭建和训练效果测试；模型参数的保存读取功能。

1.1 高可扩展性与整体设计

从可扩展性的视角考虑，我们首先就上述任务的中最普遍、最一般的需求设计了一组接口。C++没有语言级别的接口定义，像Java和C#等语言具备的那样。因此我们严格限制被称为接口的类的内容，使其仅仅包含纯虚函数以供子类重写。

然后，我们从构建神经网络的任务中最常见的需求出发，实现具体的类来完成模块的整体功能。包括优化器、全连接层、DropOut层和LayerNorm层、数据加载器等内容，具体内容如图 2。

通过这种实现方法，我们既保证了内置组件的接口风格统一，也保证了任何自定义组件只要实现对应的接口，就可以和其他内置组件无缝协同工作，良好地实现了高可扩展性愿景。

1.2 高可用性与API设计

在API设计上，为了保证良好的代码可读性，我们坚持让代码的形式结构和执行逻辑相统一的设计理念。即*代码的文本本身应当对运行时发生的事有一种恰当的隐喻*。在此基础上，我们积极保证接口设计和现代C++的STL库设计相适应的原则。

这里以管道运算符的设计为例。我们实现了管道运算符(Pipe Operators)以表达层之间的连接。该运算符的使用如代码 1中所示。

代码 1: 管道运算符

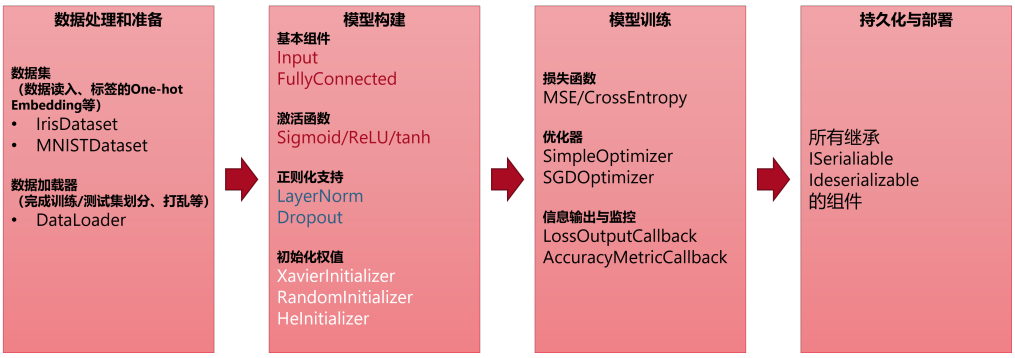
```
Input in{4};
FullyConnected fc1{5, relu};
FullyConnected fc2{8, sigmoid};
DropOut drop1{0.05};
FullyConnected fc3{12, sigmoid};
FullyConnected out{3, softmax};
CrossEntropyLoss loss{};
Model model{in | fc1 | fc2 | drop1 | fc3 | out,
    loss};
```

其中最后一行使用管道运算符连接各层，在此过程中计算图的构建和各层参数的调整会自动完成。这样使得构建模型的过程看起来更加简单直接。这种设计来自于C++20 Ranges[1] 对范围适配器进行复合时使用的设计。

正如Bjarne Stroustrup的论文*Thriving in a Crowded and Changing World: C++ 2006-2020*中的观点 [3]， Ranges是对C++ STL的重要改进。我们借鉴其设计，能够面向未来地使我们的库与未来的C++标准库更为接近，使得初学者容易学习、容易上手、容易理解。

[1]Ranges library `https://en.cppreference.com/w/cpp/ranges`

图 2: 主要功能



2 实现与功能

2.1 反向传播公式的改进实现

为了更好地实现高可扩展性的架构设计，我们需要改进常见的的反向传播公式。

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}} &= L^{'}(\mathbf{z^{(L)}}) \cdot f^{'}(\mathbf{z^{(L)}}) \cdot \mathbf{a^{(L-1)}} \\ \sigma^{(L)} &= L^{'}(\mathbf{z^{L}}) \cdot f^{'}(\mathbf{z^{L}})\end{aligned} \tag{1}$$

反向传播公式计算损失函数对最后一层参数的梯度的公式的的常见形式如(1)。这种写法有较强的局限性。例如不是所有的激活函数f都是一元函数。一个常用的多元激活函数是Softmax函数，它的定义是$f(x_j;\mathbf{x}) = \frac{e^{x_j}}{\sum_{x_i \in \mathbf{x}} e^{x_i}}$，对于这种多元函数，该公式是无法处理的。因此我们从最根本的链式法则出发使得反向传播公式更加符合我们的高可扩展性愿景。设最后一层激活值为$\mathbf{z}$，有(2)

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \tag{2}$$

则各个分量有(3)

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial z_j} \cdot \frac{\partial z_j}{\partial \mathbf{x_i}} \tag{3}$$

设最后一层各神经元有激活函数$f_1, f_2, \ldots, f_n$，令$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$，$d^{\mathsf{T}} = \begin{bmatrix} \frac{\partial L}{\partial x_1} & \cdots & \frac{\partial L}{\partial x_n} \end{bmatrix}$
则(2)可以写成(4)

$$\frac{\partial L}{\partial \mathbf{x}} = d^{\mathsf{T}} J \tag{4}$$

这样做的好处不仅仅是能够处理Softmax这样的多元激活函数，更重要的是矩阵-向量的乘法可以良好地利用SIMD(Single Instruction Multiple Data)指令进行并行化处理，能够充分利用现代硬件的性能。

2.2 激活函数

激活函数是重要的组件。非线性的激活函数是多层神经网络能够拟合任意复杂的函数的强大能力的重要基础。我们主要实现了四种激活函数：ReLU、Softmax、Tanh、Sigmoid。

每个激活函数都要实现$IActivationFunction$中的$eval$和$derive$，分别进行求值和求导，求导的对象包括标量和向量。如代码 2

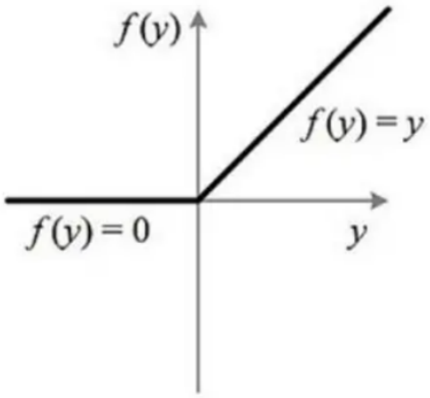
代码 2: 接口$IActivationFunction$

```
class IActivationFunction
        : public base::ITypeId
{
public:
    virtual double operator()(double x) = 0;

    virtual double eval(double x) = 0;

    virtual double derive(double y) = 0;

    virtual Eigen::VectorXd eval(Eigen::VectorXd
        x) = 0;

    virtual Eigen::MatrixXd derive(Eigen::
        VectorXd y) = 0;

    virtual std::shared_ptr<IWeightInitializer>
        default_initializer() = 0;
};
```

2.2.1 $ReLU$函数

ReLU(Rectified Linear Unit)是一个分段线性函数，如果输入为正，直接输出；否则，输出为零。函数图像如下：

图 3: ReLU



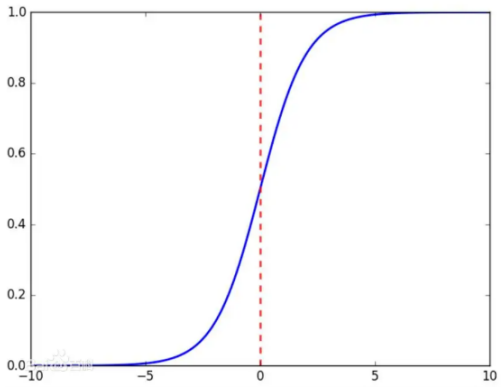
公式是 $f(x) = \max(0, x)$

ReLU函数的求导结果也较为简单，在$x > 0$处为1，否则为0。ReLU的导数恒为1，不容易导致梯度消失的问题，在深度神经网络模型中非常常用。

2.2.2 *Sigmoid*函数

Sigmoid函数也是常被用作神经网络的激活函数，它将输入映射到$[0, 1]$之间。因此有防止激活值过大的功能函数图像如图 4。

图 4: Sigmoid函数



公式是$f(x) = \frac{1}{1+e^{-x}}$。该函数导数最大处为0.25，可能导致梯度消失。

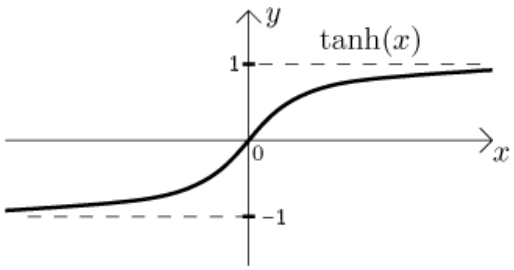
Sigmoid函数的导数值可以由函数值来表示成$f'(x) = f(x) \cdot (1 - f(x))$

2.2.3 *Tanh*函数

Tanh函数是双曲函数中的双曲正切，能将输入映射到$[-1, 1]$之间，图像如图 5。

公式是$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$。该激活函数常用于RNN中。

图 5: Tanh函数



类似Sigmoid函数，tanh函数的导数值同样可以由函数值表示为$f'(x) = 1 - f^2(x)$。

2.3 损失函数

神经网络的训练过程就是通过梯度下降法最小化损失函数的过程。我们实现了MSE和交叉熵损失函数。

损失函数需要实现*ILossFunction*抽象类中声明的*eval*和 *derive*，它们分别执行求值和求导。如代码 3。

代码 3: 接口*ILossFunction*

```
class ILossFunction
        : public base::ITypeId
{
 public:
        virtual double operator()(Eigen::VectorXd
             value, Eigen::VectorXd label) = 0;
        virtual double eval(Eigen::VectorXd value
            , Eigen::VectorXd label) = 0;
        virtual Eigen::VectorXd derive(Eigen::
            VectorXd value, Eigen::VectorXd
            label) = 0;
};
```

2.3.1 MSE损失函数

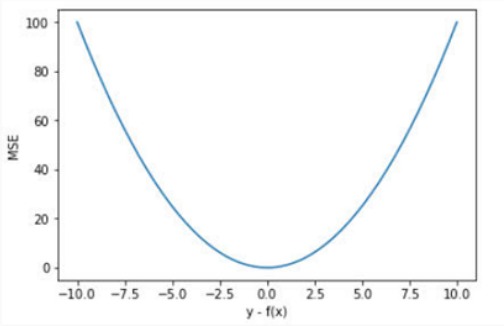
MSE是回归损失函数中常用的误差，它是预测值$f(x)$与目标值y之间差值平方和的均值。如(5)

$$MSE(f(x), y) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 \qquad (5)$$

函数图像如图 6。

平方损失函数是光滑函数，能够用梯度下降方法进行优化，随着误差的减小，梯度也在减小，这有利于收敛，然而预测值和真实值差距越大，平方损失的惩罚力度越大，即它对于异常点比较敏感，受异常数

图 6: MSE损失函数



值影响较大。如果样本中存在离群点，MSE会给离群点更高的权重，这就会牺牲其他正常点数据的预测效果，最终降低整体的模型性能。

MSE损失函数求最后一层梯度时，梯度与最后一层激活函数的导数正相关，如果激活函数的导数数值较小，那么梯度更新幅度较小。收敛时间较长。反之则收敛时间短，效果较好。因此有较为不稳定的特点。

2.3.2 交叉熵损失函数

交叉熵损失函数的标准形式如(6)

$$CE(f(x),y)=\frac{1}{n}\sum_{i=1}^{n}-y_i\log(f(x_i))-(1-y_i)\log(1-f(x_i)) \tag{6}$$

其中x_i表示样本，y_i表示实际的标签，n表示样本总数量。它本质上也是一种对数似然函数，可用于二分类和多分类任务中。当使用Sigmoid函数作为激活函数的时候，常用交叉熵损失函数而不用均方误差损失函数，因为它可以完美解决平方损失函数权重更新过慢的问题，具有“误差大的时候，权重更新快；误差小的时候，权重更新慢”的良好性质。此时损失函数对于最后　层权重的梯度不再跟激活函数的导数相关，只跟输出值和真实值的差值有关，此时收敛较快。
因为反向传播是连乘的，因此整个权重矩阵的更新都会加快。另外，多分类交叉熵损失求导更简单，损失仅与正确类别的概率有关。

2.4 优化器

我们实现了简单的梯度下降优化器和有动量优化和Nestrov优化的SGD优化器。

2.4.1 简单优化器

简单优化器实现了简单的定步长梯度下降，即公式(7)。

$$w_i^{(t+1)}=w_i^{(t)}-\alpha\frac{dJ}{dw_i} \tag{7}$$

这是一种基础的优化器。其中α是学习率。它的选择对效果有很大的影响。

2.4.2 SGD优化器

我们实现了和pytorch中的SGD功能同样强大的SGD优化器，这包括动量优化和Nestrov优化。

SGD中动量优化的概念使得前几轮的梯度也会加入到当前的计算中（会有一定衰减），通过对前面一部分梯度的指数加权平均使得梯度下降过程更加平滑，减少动荡，收敛也比普通的SGD快。当前梯度方向与累计梯度方向一致时，梯度会被加强，从而这一步下降幅度增大，若方向不一致，则会减弱当前下降的梯度幅度。更新方式如算法 1

算法 1 动量优化的梯度下降

```
if momentum > 1 then
    b_t ← momentum * b_{t-1}
else
    b_t ← grads
```

动量主要解决SGD的两个问题：

1. 随机梯度的方法（引入的噪声）；
2. Hessian矩阵病态问题（可以理解为SGD在收敛过程中和正确梯度相比来回摆动比较大的问题）。可以形象化的理解为，当前权值的改变会受到上一次权值改变的影响时，类似于小球向下滚动的时候带上了惯性。这样可以加快小球向下滚动的速度。

SGD的优势是：

- 相较于非随机算法，SGD能更好的利用、排除冗余信息。
- SGD在前期的迭代效果卓越。
- 在样本数量较大时，SGD的计算复杂度具有优势梯度下降在强凸的情况下收敛的速度是线性收敛的，最差情况下，至少需要迭代$O(log(\frac{1}{\epsilon}))$次，才能达到$\|\sum_{i=1}^{n}f_i(x_t)-f^\star\|\leqslant\epsilon$的精度，加上每次梯度下降计算n个梯度，所以总的计算复杂度是$O(nlog(\frac{1}{\epsilon}))$。而SGD，为了达到$\|\sum_{i=1}^{n}f_i(x_t)-f^\star\|\leqslant\epsilon$，在最差情况下，我们需要迭代的次数是$O(\frac{1}{\epsilon})$,但是每次就计算一个梯度，所以计算复杂度为$O(\frac{1}{\epsilon})$。

Nesterov是对动量方法的一种改进，由Ilya Sutskever提出，它的主要思想是先按照原来的更新方向更新一步，然后在该位置计算梯度值，然后再用这个梯度值修正最终的更新方向。如(8)。

$$\delta x_t = \rho \Delta x_{t-1} - \mu \Delta f(x_t + \rho \Delta x_{t-1}) \tag{8}$$

我们整体的SGD实现参照pytorch完成，如算法 2

算法 2 SGD优化器更新算法

Input: γ(学习率), f(目标函数), λ(权重衰减), μ(动量)
Input: τ(抑制参数), $nestrov$(是否使用nesterov优化)
Input: $maxmize$(是最大化还是最小化)

for $t = 1$ *to* ... **do**
  $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda \theta_{t-1}$
  if $\mu \neq 0$ **then**
    if $t > 1$ **then**
      $b_t \leftarrow \mu b_t + (1 - \tau) g_t$
    else
      $b_t \leftarrow g_t$
    if $nesterov$ **then**
      $g_t \leftarrow g_{t-1} + \mu b_t$
    else
      $g_t \leftarrow b_t$
  if $maxmize$ **then**
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

2.5 层与正则化

层是模型构建中的主体，它要实际地处理反向传播的过程。我们在实现了最为关键也最为基础的全连接层的基础上，也实现了常见的用于进行正则化的层，如LayerNorm和DropOut。

2.5.1 全连接层

全连接层是一种基础的层，它用于建模最基础的含有n个神经元的层。前向传播公式为(??)，

$$z^l = f(\boldsymbol{W}^l \boldsymbol{x} + \boldsymbol{b}^l) \tag{9}$$

反向传播公式如(10)。

$$\boldsymbol{\delta}^l = ((\boldsymbol{\omega}^{l+1})^\intercal \boldsymbol{\delta}^{l+1}) \odot f'(\boldsymbol{z}^l) \tag{10}$$

2.5.2 LayerNorm层

Layer Normalization [4]的作用是减少曾与层之间的协变偏差(Covariate Shift)，加快收敛，在Transformer结构的网络中特别常用。它的前向传播公式如(11)。

$$\boldsymbol{z}^l = \boldsymbol{g}^{\boldsymbol{l}} \odot \frac{1}{\sqrt{\sigma^2 + \epsilon}} (\boldsymbol{x} - \mu_l) + \boldsymbol{b}^l \tag{11}$$

其中σ和μ是输入值标准差和均值，g_i和b_i是可学习的参数。反向传播公式如(12)。

$$\begin{aligned}
&D = (\sigma^2 + \epsilon)^{-\frac{1}{2}} \\
&\frac{\partial J}{\partial x_i} = D \left[\frac{\partial J}{\partial y_i} g_i - \frac{1}{H} \left(\sum_{j=1}^{H} \frac{\partial J}{\partial y_i} + \hat{x_i} \sum_{j=1}^{H} \frac{\partial J}{\partial y_j} g_j \cdot \hat{x_j} \right) \right] \\
&\frac{\partial J}{\partial g_i} = \frac{\partial J}{\partial y_i} \cdot \hat{x_i} \\
&\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y_i} \cdot 1 \\
&\delta_i = \frac{\partial J}{\partial x_i}
\end{aligned} \tag{12}$$

实验证明，该层对较深的ANN相当有用。

2.5.3 DropOut层

DropOut [5]在神经网络训练过程中随机选中一些神经元并将其隐藏，然后进行训练和优化。是一种常见的正则化手段。它的前向传播公式如(13)。

$$\begin{aligned}
r_j^{(l)} &\sim Bernoulli(p) \\
\widetilde{\boldsymbol{y}}^{(l)} &= \boldsymbol{r}^{(l)} \odot \boldsymbol{y}^{(\boldsymbol{l})}
\end{aligned} \tag{13}$$

反向传播公式如(14)。

$$\frac{\partial J}{\partial \boldsymbol{x}} = \frac{\partial J}{\partial \boldsymbol{y}} \odot \boldsymbol{r}^{(l)} \tag{14}$$

2.6 模型

模型是我们框架的核心。它要处理训练的具体过程和参数的加载与保存。这是神经网络训练和使用过程中的核心任务

2.6.1 训练

对模型的训练和评估任务，我们提供了$train$和$evaluate$方法。他们的主体部分相似，

区别在于*train*计算完损失函数以后，会进一步开始反向传播过程和参数优化过程，而*evaluate*不会。

在训练过程中，损失函数的数值、各测度，例如Top-k accuracy，对神经网络的训练者而言是重要的参考。为了能够在输出信息的过程中实现高可扩展性的愿景，我们定义了接口*IModelCallback*，它包括各个环节中可能会用到的回调函数定义。*train*和*evaluate*方法会在特定的阶段调用它们，并且提供合适的参数以供使用者计算各种参量。如代码4。

代码 4: 接口*IModelCallback*

```
class IModelCallback
{
 public:
        template<typename T, typename... Args>
        static inline std::shared_ptr<
            IModelCallback> make(Args&& ... args
            )
        {
                return std::make_shared<T>(std::
                    forward<Args>(args)...);
        }

        virtual std::string before_world() = 0;
        virtual std::string after_world() = 0;

        virtual std::string before_train(size_t
            step) = 0;
        virtual std::string train_step(size_t
            step,
                const dataloader::DataPair& dp,
                Eigen::VectorXd predicts,
                double loss) = 0;
        virtual std::string after_train(size_t
            epoch) = 0;

        virtual std::string before_eval(size_t
            epoch) = 0;
        virtual std::string eval_step(size_t step
            ,
                const dataloader::DataPair& dp,
                Eigen::VectorXd predicts,
                double loss) = 0;
        virtual std::string after_eval(size_t
            epoch) = 0;
};
```

为了提供基本的信息输出功能，我们实现了*LossOutputCallback*和 *AccuracyMetricCallback*。它们能够输出损失函数值和Top-k accuracy测度。

2.6.2 加载和保存

为了实现模型的加载与保存，我们要求层的实现提供*Load*和*Save*方法。在需要保存和加载时，模型会依次对各层调用相应的方法，来将参数写入流中，或从流中读取。C++的输入输出流是恰当地管理数据输入输出的统一接口，它是十分强大而有效的。

2.7 数据和数据加载

数据是神经网路训练中不可缺少的部分。我们将数据的加载和使用分为*IDataset*和 *Dataloader*两个类来完成。*IDataset*提供了一组实现读取数据集所必需的接口。为适配任何一种数据集，只要继承该类就能使这个数据集和我们的框架配合使用。

*Dataloader*提供了将数据给到模型进行训练的过程中不可缺少的重要功能，例如按一定大小取Batch、随机打乱测试数据等，为模型类提供了一个统一的接口。

3 代码编写

在代码编写中，我们秉承可靠、安全、现代的理念，积极使用语言新特性、严格按照编码规范，使得项目有较高的代码质量、更加适应现代C++的演进方向，富有前瞻性。

3.1 Concept的使用

在C++20标准诞生之前，为了更有效地利用模板，人们使用SFINAE(Substitution Failure Is Not An Error) 来对模板的类型参数加以限制。这种方法很不直观、实现复杂、会导致编译器输出长度较高、内容混杂、甚至无法被理解的错误信息。是一种具有时代局限性的做法。

然而不对模板参数加以限制，就会导致错误在编译期模板实例化时报错，错误信息也难以被理解，并且不方便IDE和静态分析工具对程序进行分析。

因此，我们使用了C++20引入的Concept特性对模板参数加以限制。类模板、函数模板和类模板的成员都可以和一个约束相关联，它指定了对模板参数的要求，可以被用于选择最正确的函数重载和模板的特化。而有名称的一个要求的集合称之为*Concept*。每一个Concept都是一个谓词，它在编译期被计算。

例如我们定义了如代码 5中的Concept。

代码 5: Concept

```
template<typename T>
concept IterableData=
requires(T t)
{
    { t.begin() };
    { t.end() };
};

template<typename T>
concept SizedData=
requires(T t, int i)
{
    { t.size() }->std::convertible_to<size_t>;
    { t.get(i) };
};
```

这段代码被用于模型类的实现中。它们分别约束要求提供STL式的迭代器接口和Python-like的容器大小/获取项目接口。由于Concept现在会参与重载决议，因此有如代码 6中的重载函数就可以按照数据集对象提供的具体接口而选择正确的重载版本来完成预测工作。这大大地提高了接口的泛用性，既使用了统一的接口，又为数据集实现提供了更大的自由。

代码 6: Concept的使用

```
template<IterableData T>
std::vector<base::VectorType> predict(const T &
    dataset)
{
        std::vector<base::VectorType> ret{};
        for (const auto &d: dataset)
        {
                const auto &[data, label] = d;
                ret.push_back((*this)(data));
        }
        return ret;
}
template<SizedData T>
std::vector<base::VectorType> predict(const T &
    dataset)
{
        std::vector<base::VectorType> ret{};
        for (int i = 0; i < dataset.size(); i++)
        {
                auto [data, label] = dataset.get(
                    i);
                ret.push_back((*this)(data));
        }
        return ret;
}
```

3.2 编码规范

为了提高代码质量，保证可维护性，我们严格执行了编码规范。在代码风格上，我们按照谷歌开源项目风格指南 [6]的规范进行编写。有效地保证了代码风格的统一，进而方便错误排查，增强了代码的安全性，同时，也与面向开源社区的开发方式的要求相适应。同时，该指南对类是否可复制、可移动的要求有效地保证了性能。

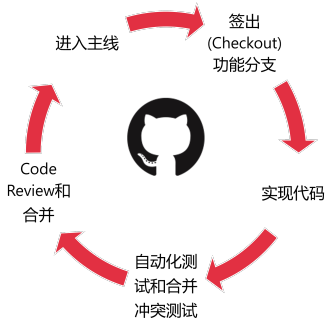
在语言功能的取舍上，我们按照*C++ Core Guidelines* [7]的指南，这包括

- 使用Concept而不是SFINAE来约束模板
- 使用智能指针管理对象的生命周期
- 使用GSL
- 其他参考该指南的内容

这些规则使得编码过程中使用到的功能更符合C++演进的方向，使得代码实现更加安全，有更少的潜在错误。

3.3 团队协作

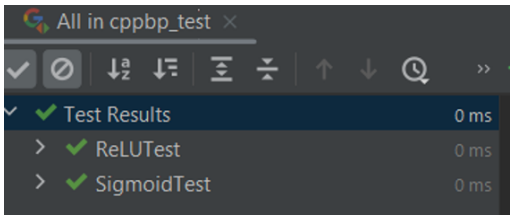
图 7: 基于Git的团队合作



在团队协作上，我们使用了如图 7所示的基于Git的工作方式。这种方式在现代软件生产中有很广泛的应用。利用源代码版本控制工具，这种方式实现了修改可溯源可回滚、合并过程清晰，能够有效结合Github Action进行自动化测试。这既能使开发过程更为顺利，也能避免人为错误对开发进度和软件质量造成负面影响。

为了进一步确保软件质量，我们基于Google Test为各模块编写了单元测试。单元测试是一种常用的黑盒测试，是现代软件生产中保证软件质量的重要方式。

图 8: 单元测试



3.4 数据统计

使用有名的开源代码量统计工具cloc，我们对项目的代码量进行统计，如表1。

表格 1: 代码量统计

语言	文件	空行	注释	代码
C++	25	394	83	1482
C++ 头文件	38	342	125	868
Markdown	1	32	0	79
CMake	9	34	0	67
Yaml	1	0	1	65
总计	74	802	209	2561

4 实验

我们在MNIST数据集和Iris数据集上进行了实验，分别进行鸢尾花分类任务和MNIST图像分类任务。

4.1 Iris数据集

Iris数据集是一个较小的常用数据集。它包含了三种鸢尾花的四类不同参数。数据分布见附录：鸢尾花数据集分布图 中的图 13。可见该数据集包含了难以线性分类的特征，是测试简单神经网络的理想数据集。

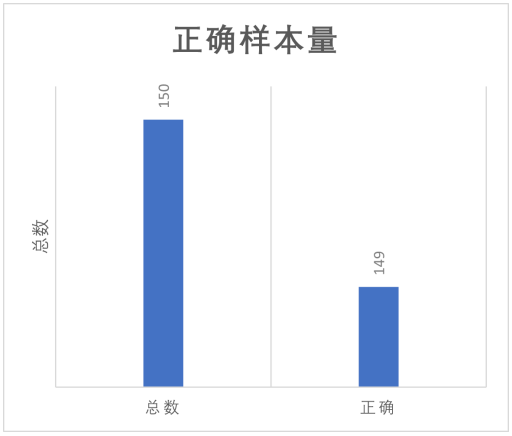
我们采用了如图 9a的网络结构来完成分类。它使用层规范化层(Layer Normalization)和Drop DropOut层和LayerNorm层来进行正则化。

在分类任务中，我们使用8 : 2的比例划分训练集和测试集并进行随机打乱。训练方法如表 2。 进行训练之后多次测量，达到了最高99.3% Top-1 Accuracy和100% Top-3 Accuracy的准确率。如图 10所示。

表格 2: 训练参数

参数	值
Batch Size	16
学习率	0.05
Epoch	1000
损失函数	Cross Entropy
初始化方法	Xavier

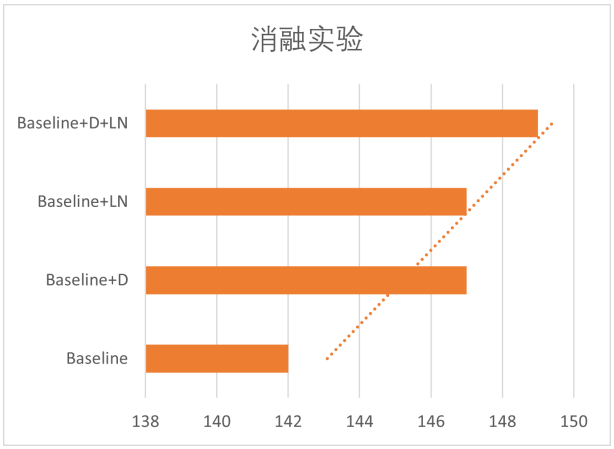
图 10: Iris数据集实验结果



4.1.1 消融实验

为了证明正则化手段的有效性，我们进行了消融实验(Ablation Study)，如图 11（D=Drop Out, LN=Layer Normalization）

图 11: 消融实验结果



可见Layer Normalization和Drop Out都为提高准确率给出了正贡献。它们的互相作用没有相互拮抗的现象。

4.2 MNIST数据集

Iris数据集是经典的图像分类任务数据集，它由近60000张包含一个手写数字的灰度图组成。我们使用如图 9b的网络结构来进行实验。

由于MNIST数据集相对较大。我们没有实现GPU计算，因此训练过程相当缓慢，我们几乎无法

(a) Iris分类网络结构 (b) MNIST识别网络结构

图 9: 实验网络结构

开展进一步的实验。但是在较少轮数的训练之后该简单的全连接神经网络亦达到了80% Top-1 Accuracy的准确率。如图 12。

图 12: MNIST数据集训练结果

```
Step:29 Loss:0.254842286410899 34
After eval, average loss is 0.6087038401523841 Top 1 accuracy: 0.8;
```

5 结论

在此次大作业中，我们使用C++完成了一神经网络框架。在这个过程中，我们既体验了现代软件开发的流程，也使用面向对象的程序设计思想亲自完成了框架的设计和编写。体会到了面向对象的程序设计思想能够更好地分解和处理问题、更好地组织代码，从而提高程序的可读性和可维护性。我们实现的神经网络框架在推理速度上达到了与现有成熟框架同一数量级的结果。

参考文献

[1] Ilya O. Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision. *CoRR*, abs/2105.01601, 2021.

[2] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training. *CoRR*, abs/2105.03404, 2021.

[3] Bjarne Stroustrup. Thriving in a crowded and changing world: C++ 2006–2020. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–168, 2020.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[6] Benjy Weinberger, Craig Silverstein, Gregory Eitzmann, Mark Mentovai, and Tashana Landray. Google c++ style guide. *Section: Line Length. url: http://google-styleguide. googlecode. com/svn/trunk/cppguide. xml# Line_Length*, 2013.

[7] Bjarne Stroustrup and Herb Sutter. C++ core guidelines. *Web. Last accessed February*, 2018.

附录：鸢尾花数据集分布图

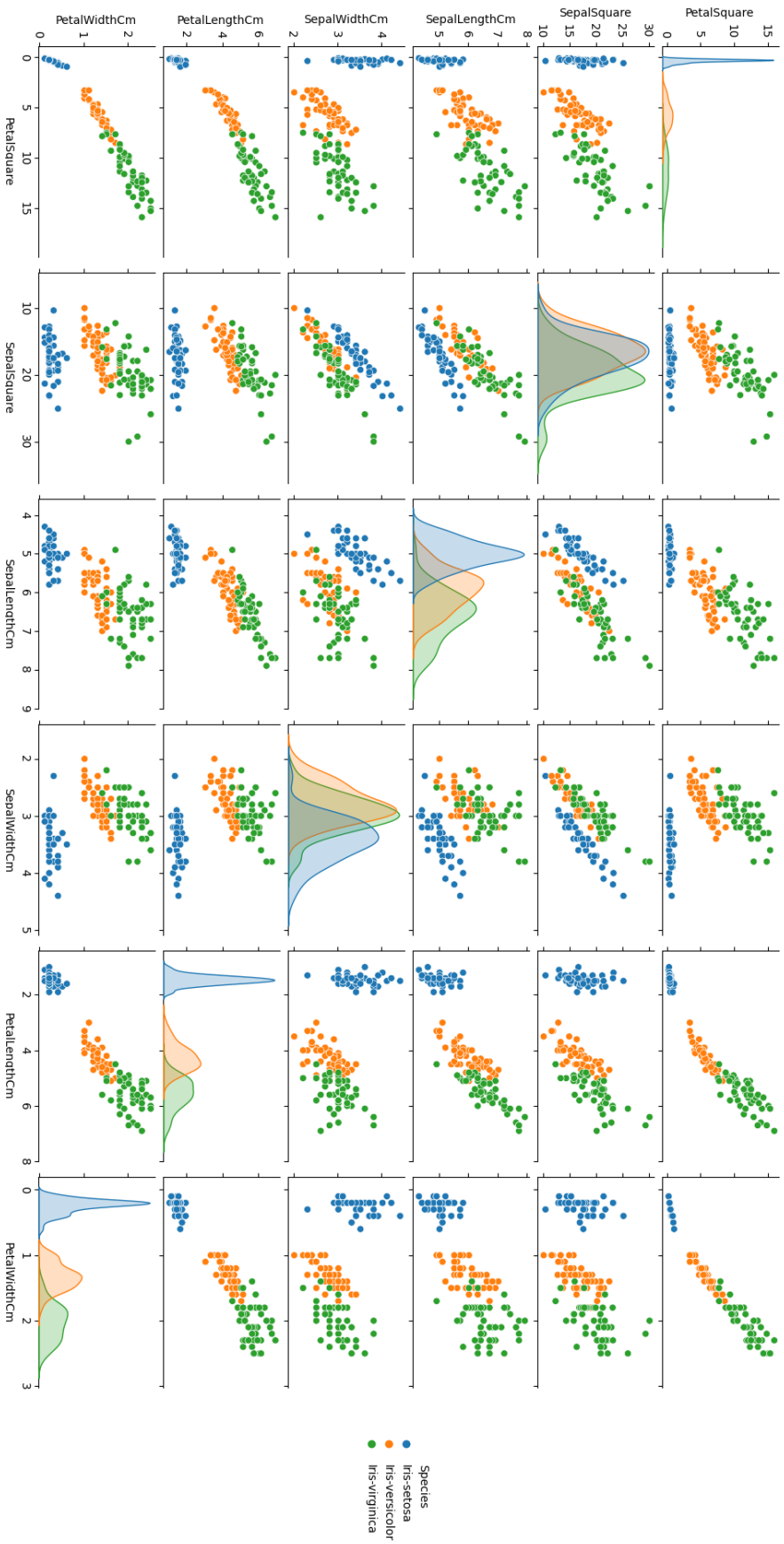


图 13: Iris数据集数据分布

附录：分工

表格 3: 分工

成员	分工
熊恪峥	框架整体设计、反向传播的公式推导、变形和实现、框架主体代码编写； 报告撰写和整理、PPT制作、上台汇报。
徐杰	反向传播的公式推导、变形；交叉熵损失函数的实现；Softmax激活函数实现；部分报告撰写
杨夏婕	SGD优化器实现；部分报告撰写
黄彬茹	ReLU和Tanh激活函数实现；部分报告撰写