第2章 从"1"开始构建深度学习框架

由于深度学习整个框架从零开始构建过于繁琐，需要重写所有算子，费时费力，这里我们
直接基于pytorch的自动微分机制，开始构建深度学习训练框架的核心模块

1.Parameter类：初始化模型参数

创建初始化的模型参数用到的类是Parameter类，和Tensor仅有细微差别，你就当成是Tensor的别名
这个类定义起来比较麻烦，所以直接把torch.nn里面的parameter.py文件复制拿过来用了
就这个是直接抄过来的，后面的东西都是参考邱老师的nndl写的

In [1]:
```python
import torch
from torch._C import _disabled_torch_function_impl
from collections import OrderedDict

# Metaclass to combine _TensorMeta and the instance check override for Parameter.
class _ParameterMeta(torch._C._TensorMeta):
    # Make `isinstance(t, Parameter)` return True for custom tensor instances that have the _is_param flag.
    def __instancecheck__(self, instance):
        return super().__instancecheck__(instance) or (
            isinstance(instance, torch.Tensor) and getattr(instance, '_is_param', False))


class Parameter(torch.Tensor, metaclass=_ParameterMeta):
    r"""A kind of Tensor that is to be considered a module parameter.

    Parameters are :class:`~torch.Tensor` subclasses, that have a
    very special property when used with :class:`Module` s - when they're
    assigned as Module attributes they are automatically added to the list of
    its parameters, and will appear e.g. in :meth:`~Module.parameters` iterator.
    Assigning a Tensor doesn't have such effect. This is because one might
    want to cache some temporary state, like last hidden state of the RNN, in
    the model. If there was no such class as :class:`Parameter`, these
    temporaries would get registered too.

    Args:
        data (Tensor): parameter tensor.
        requires_grad (bool, optional): if the parameter requires gradient. See
            :ref:`locally-disable-grad-doc` for more details. Default: `True`
    """
    def __new__(cls, data=None, requires_grad=True):
        if data is None:
            data = torch.empty(0)
        if type(data) is torch.Tensor or type(data) is Parameter:
            # For ease of BC maintenance, keep this path for standard Tensor.
            # Eventually (tm), we should change the behavior for standard Tensor to match.
            return torch.Tensor._make_subclass(cls, data, requires_grad)

        # Path for custom tensors: set a flag on the instance to indicate parameter-ness.
        t = data.detach().requires_grad_(requires_grad)
        if type(t) is not type(data):
            raise RuntimeError(f"Creating a Parameter from an instance of type {type(data).__name__} "
                               "requires that detach() returns an instance of the same type, but return "
                               f"type {type(t).__name__} was found instead. To use the type as a "
                               "Parameter, please correct the detach() semantics defined by "
                               "its __torch_dispatch__() implementation.")
        t._is_param = True
        return t

    # Note: the 3 methods below only apply to standard Tensor. Parameters of custom tensor types
    # are still considered that custom tensor type and these methods will not be called for them.
    def __deepcopy__(self, memo):
        if id(self) in memo:
            return memo[id(self)]
        else:
            result = type(self)(self.data.clone(memory_format=torch.preserve_format), self.requires_grad)
            memo[id(self)] = result
            return result

    def __repr__(self):
        return 'Parameter containing:\n' + super().__repr__()

    def __reduce_ex__(self, proto):
        state = torch._utils._get_obj_state(self)

        # See Note [Don't serialize hooks]
        hooks = OrderedDict()
        if not state:
            return (
                torch._utils._rebuild_parameter,
                (self.data, self.requires_grad, hooks)
            )

        return (
            torch._utils._rebuild_parameter_with_state,
            (self.data, self.requires_grad, hooks, state)
        )

    __torch_function__ = _disabled_torch_function_impl


class UninitializedTensorMixin:
    _allowed_methods = [
        torch.Tensor.__hash__,
        torch.Tensor.size,
        torch.Tensor.copy_,
        torch.Tensor.is_floating_point,
        torch.Tensor.half,
        torch.Tensor.float,
        torch.Tensor.double,
        torch.Tensor.char,
        torch.Tensor.short,
        torch.Tensor.int,
        torch.Tensor.long,
        torch.Tensor.cuda,
        torch.Tensor.cpu,
        torch.Tensor.to,
        torch.Tensor.get_device,
        torch._has_compatible_shallow_copy_type,
    ]

    def materialize(self, shape, device=None, dtype=None):
        r"""Create a Parameter or Tensor with the same properties of the uninitialized one.
        Given a shape, it materializes a parameter in the same device
        and with the same `dtype` as the current one or the specified ones in the
        arguments.

        Args:
            shape : (tuple): the shape for the materialized tensor.
            device (:class:`torch.device`): the desired device of the parameters
                and buffers in this module. Optional.
            dtype (:class:`torch.dtype`): the desired floating point type of
                the floating point parameters and buffers in this module. Optional.
        """
        if device is None:
            device = self.data.device
        if dtype is None:
            dtype = self.data.dtype
        self.data = torch.empty(shape, device=device, dtype=dtype)
        self.__class__ = self.cls_to_become

    @property
    def shape(self):
        raise RuntimeError(
            'Can\'t access the shape of an uninitialized parameter or buffer. '
            'This error usually happens in `load_state_dict` when trying to load '
            'an uninitialized parameter into an initialized one. '
            'Call `forward` to initialize the parameters before accessing their attributes.')

    def share_memory_(self):
        raise RuntimeError(
            'Can\'t share memory on an uninitialized parameter or buffer. '
            'Call `forward` to initialize the parameters before calling '
            '`module.share_memory()`.')

    def __repr__(self):
        return f'<{self.__class__.__name__}>'

    def __reduce_ex__(self, proto):
        # See Note [Don't serialize hooks]
        return (
            self.__class__,
            (self.requires_grad,)
        )

    @classmethod
    def __torch_function__(cls, func, types, args=(), kwargs=None):
        # method-wrapper is to detect access to Tensor properties that are
        # wrapped in descriptors
        if func in cls._allowed_methods or func.__class__.__name__ == 'method-wrapper':
            if kwargs is None:
                kwargs = {}
            return super().__torch_function__(func, types, args, kwargs)
        raise ValueError(
            'Attempted to use an uninitialized parameter in {}. '
            'This error happens when you are using a `LazyModule` or '
            'explicitly manipulating `torch.nn.parameter.{}` '
            'objects. When using LazyModules Call `forward` with a dummy batch '
            'to initialize the parameters before calling torch functions'.format(func, cls.__name__))


def is_lazy(param):
    return isinstance(param, UninitializedTensorMixin)


class UninitializedParameter(UninitializedTensorMixin, Parameter):
    r"""A parameter that is not initialized.
    Uninitialized Parameters are a a special case of :class:`torch.nn.Parameter`
    where the shape of the data is still unknown.

    Unlike a :class:`torch.nn.Parameter`, uninitialized parameters
    hold no data and attempting to access some properties, like their shape,
    will throw a runtime error. The only operations that can be performed on a uninitialized
    parameter are changing its datatype, moving it to a different device and
    converting it to a regular :class:`torch.nn.Parameter`.

    The default device or dtype to use when the parameter is materialized can be set
    during construction using e.g. ``device='cuda'``.
    """

    cls_to_become = Parameter

    def __new__(cls, requires_grad=True, device=None, dtype=None) -> None:
        factory_kwargs = {'device': device, 'dtype': dtype}
        data = torch.empty(0, **factory_kwargs)
        return torch.Tensor._make_subclass(cls, data, requires_grad)

    def __deepcopy__(self, memo):
        if id(self) in memo:
            return memo[id(self)]
        else:
            result = type(self)(self.requires_grad, self.data.device, self.data.dtype)
            memo[id(self)] = result
            return result

class UninitializedBuffer(UninitializedTensorMixin, torch.Tensor):
    r"""A buffer that is not initialized.

    Uninitialized Buffer is a a special case of :class:`torch.Tensor`
    where the shape of the data is still unknown.

    Unlike a :class:`torch.Tensor`, uninitialized parameters
    hold no data and attempting to access some properties, like their shape,
    will throw a runtime error. The only operations that can be performed on a uninitialized
    parameter are changing its datatype, moving it to a different device and
    converting it to a regular :class:`torch.Tensor`.

    The default device or dtype to use when the buffer is materialized can be set
    during construction using e.g. ``device='cuda'``.
    """

    cls_to_become = torch.Tensor

    def __new__(cls, requires_grad=False, device=None, dtype=None) -> None:
        factory_kwargs = {'device': device, 'dtype': dtype}
        data = torch.empty(0, **factory_kwargs)
        return torch.Tensor._make_subclass(cls, data, requires_grad)
```

2.Module模块：所有神经元的基类

简单来说就是装底层算子复合运算的一个壳子，相当于复合函数，这里我们做成一个壳子，
并实现一些特殊的功能。
init:初始化模型参数
call:前向计算
forward:前向计算的实现
parameters:获取模型参数
train:使模型切换为训练模式
eval:使模型切换为评估模式

In [2]:
```
import abc
"""
几点说明：
1.这里的Parameter类，就是把torch.nn下的Parameter拷贝过来了，如果自己定义为torch.Tensor的子类，
与Tensor运算后类型会变成Parameter类，也就是子类和父类运算，会变成子类，官方的用到了元类（类的类），反正不清楚怎么弄的运算结果还是te
2.Module就是一个组装器，把构建计算图的过程封装在里面，是个复合函数，这里仅仅实现获取模型参数的功能，暂时够用了
"""
class Module:
    """
    所有神经网络的基类
    """

    def __init__(self):
        pass

    @abc.abstractmethod
    def __call__(self,*args,**kwargs):
        """
        子类实现前向传播
        """

    @abc.abstractmethod
    def forward(self,*args,**kwargs):
        """
        子类实现前向计算
        """

    def parameters(self):
        params = []
        for key, value in self.__dict__.items():
            if type(value) == Parameter:
                params.append(value)
            elif isinstance(value,Module):
                params += value.parameters()
        return params

    def train(self):
        """模型切换为训练模式"""
        pass

    def eval(self):
        """模型切换为评估模式"""
        pass
```

3.Layer

上面是神经元的基类，这里基于Module创建各种Layer，也可以是一些Block

In [3]:
```
class Logistic(Module):

    def __init__(self):
        super(Module, self).__init__()

    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):
        return 1.0 / (1.0 + torch.exp(-X))


class Tanh(Module):

    def __init__(self):
        super(Module, self).__init__()

    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):
        return (torch.exp(X) - torch.exp(-X)) / (torch.exp(X) + torch.exp(-X))


class ReLu(Module):
    """
    实现的是LeakyReLU哦
    """

    def __init__(self):
        super(Module, self).__init__()

    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):
        negative_slope = torch.Tensor([0.1])  # 负半轴的斜率,写死在这里，不需要调整
        return torch.where(X > 0, X, 0) + negative_slope * torch.where(X < 0, X, 0)


class Softmax(Module):
    """
    实现的是LeakyReLU哦
    """

    def __init__(self):
        super(Module, self).__init__()

    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):
        # x_max=torch.max(X,dim=1)[0]
        # t_0=torch.exp((X-x_max))
        t_0 = torch.exp(X)
        t_1 = t_0.sum(dim=1).unsqueeze(1)
        value = t_0 / t_1
        return value

class Linear(Module):
    def __init__(self, input_size, output_size):
        """
        :param input_size: 输入向量的维度
        :param output_size: 神经元个数，即输出个数（输出向量的维度）
        :param activation: 激活函数类型
        """
        Module.__init__(self)
        # 基本属性
        self.input_size = input_size
        self.output_size = output_size

        # 初始化模型参数
        self.weight = Parameter(torch.randn(size=[input_size,output_size]))
        self.bias = Parameter(torch.randn(size=[1]))

    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):
        return torch.matmul(X,self.weight)+self.bias
```

4.Loss Function

这里仅写一个用的最多的交叉熵损失函数的实现
其本质也是包装算子计算的壳，由于不含参数，只要实现计算图构建其实就够了
换句话说，这里不用继承Module实现为一个类，可以简单写成一个函数的形式即可

In [4]:
```
class CrossEntropyLoss(Module):
    """
    预测：[N,C]
    标签：[N]
    输出：[]
    """

    def __init__(self):
        super(Module, self).__init__()
        self.dimension = (2, 1, 0)  # 这个属性的含义是，预测，标签，输出的维度分别是2，1，0，没什么实际用途，容易忘写给自己看的

    def __call__(self, predicts, y):
        assert predicts.shape[0]==y.shape[0]
        return self.forward(predicts, y)

    def forward(self, predicts, y):
        N=predicts.shape[0]
        loss=0
        for i in range(N):
            c=y[i]
            loss-=torch.log(predicts[i][c]+1e-8)
        return loss/N
```

5.Metric

和loss差不多，丢进去predicts和labels，算出损失或评分
唯一不同的是，使用metric的时候，不要构建计算图，这点在后面可以看出

In [5]:
```
class Metric:
    pass


class Accuracy(Metric):

    def __init__(self):
        # 用于统计正确的样本个数
        self.num_correct = 0
        # 用于统计样本的总数
        self.num_count = 0

    def update(self, predicts, labels):
        """
        输入：
          - predicts: 预测值, shape=[N,class_num]
          - labels: 标签值, shape=[N]
        """
        preds = predicts.argmax(dim=1)
        batch_correct = (preds == labels).sum()
        batch_count = len(labels)

        # 更新num_correct 和 num_count
        self.num_correct += batch_correct
        self.num_count += batch_count

    def accuracy(self):
        # 使用累计的数据，计算总的指标
        return (self.num_correct / self.num_count).item()

    def reset(self):
        # 重置正确的数目和总数
        self.num_correct = 0
        self.num_count = 0


if __name__ == "__main__":
    predicts = torch.tensor([[0.1, 0.2, 0.7], [0.3, 0.5, 0.2]])
    labels = torch.tensor([0, 1])
    metric = Accuracy()
    metric.update(predicts, labels)
    print(metric.accuracy())
```

```
0.5
```

6.Optimizer

实现SGD和Adam

In [6]:
```
class Optimizer:
    """
    优化器的基类
    """

    def __init__(self, params,lr=0.01):
        self.params=params
        self.lr=lr

    def step(self):
        pass

    def zero_grad(self):
        for param in self.params:
            param.grad = None


class SGD(Optimizer):

    def __init__(self,params,lr):
        super(SGD,self).__init__(params,lr)

    def step(self):
        for param in self.params:
            # print("参数地址：",id(param))

            param.data=param.data-self.lr*param.grad  # 注意更新参数时，只更新内部数据，不要更改参数地址



class Adam(Optimizer):
    """
    Adam优化器
    """

    def __init__(self,params,lr, beta1=0.9, beta2=0.99):
        assert 0.0 < beta1 < 1.0 and 0.0 < beta2 < 1.0
        super(Adam, self).__init__(params, lr)
        self.beta1=beta1
        self.beta2=beta2

        # 修正后的方向
        self.M = dict()

        # 方向的缩放系数(历史梯度各分量平方累积)
        self.G = dict()

        # 所有参数初始累积方向和累积缩放系数设置为0
        for param in params:
            self.M[param]=0
            self.G[param]=0

    def step(self):
        for param in self.params:
            # print("参数地址：", id(param))

            # 更新M
            self.M[param]=self.beta1*self.M[param]+(1-self.beta1)*param.grad
            # 更新G
            self.G[param]=self.beta2* self.G[param]+(1-self.beta2)*(param.grad**2)
            # 更新参数
            param.data=param.data-(self.lr/torch.sqrt(self.G[param]+1e-8))*self.M[param]
```

7.Dataset和DataLoader

In [7]:
```
class Dataset:
    """
    需要在init中实现读取数据，并保存在自定义的属性当中
    然后在getitem中返回单个元素
    """
    def __init__(self, *args, **kwargs):
        pass

    @abc.abstractmethod
    def __getitem__(self, idx):
        pass

    @abc.abstractmethod
    def __len__(self):
        pass
```

In [8]:
```
class DataLoader:
    def __init__(self, dataset=None, batch_size=1, shuffle=False, collate_fn=None, drop_last=False):
        """
        :param dataset:数据集，要能使用len()获取长度，并且能够使用切片获取元素
        :param batch_size:一个批次的大小
        :param shuffle:是否打乱样本
        :param collate_fn:数据集后处理函数
        :param drop_last:是否舍弃最后一个不够batch_size的数据
        """
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.collate_fn = collate_fn
        self.drop_last = drop_last

    def random_sample(self):
        n = len(self.dataset)
        return torch.randperm(n).tolist()

    def sequential_sample(self):
        n = len(self.dataset)
        return list(range(n))

    def __iter__(self):
        if self.shuffle:
            index = self.random_sample()
        else:
            index = self.sequential_sample()
        n = len(self.dataset)
        stride = self.batch_size
        if n % stride == 0 or self.drop_last:
            N = int(n / stride)
        else:
            N = int(n / stride) + 1
        for i in range(N):
            batch_index = index[i * stride:min(i * stride + stride, n)]
            batch_data = self.dataset[batch_index]
            if self.collate_fn:
                batch_data = self.collate_fn(batch_data)
            yield batch_data

    def __len__(self):
        n = len(self.dataset)
        stride = self.batch_size
        if n % stride == 0 or self.drop_last:
            N = int(n / stride)
        else:
            N = int(n / stride) + 1
        return N


if __name__ == "__main__":
    def collate_fn(batch_data):
        batch_data[0]=88888
        return batch_data
    dataloader = DataLoader(dataset=torch.tensor([3, 5, 4, 56, 3, 7]), batch_size=2, drop_last=True,collate_fn=collate_fn)

    for data in dataloader:
        print(data)
```

```
tensor([88888,     5])
tensor([88888,    56])
tensor([88888,     7])
```

8.Runner

In [9]:
```
import matplotlib.pyplot as plt

class RunnerV1():
    def __init__(self, model=None, loss_fn=None, optimizer=None, metric=None):

        self.model = model
        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.metric = metric

        self.train_epoch_loss = []  # 每个epoch的训练损失
        self.train_step_loss = []  # 每步的训练损失
        self.dev_stride_score = []  # 每stride步的测试分数
        self.dev_stride_loss = []  # 每stride步的测试损失
        self.best_score = 0  # 记录评估时最好的得分

    def train(self, train_loader, dev_loader=None, num_epochs=1, log_stride=1):
        # 当前训练步数（全局步数）
        step = 0
        total_steps = num_epochs * len(train_loader)

        for epoch in range(num_epochs):
```

```
        # 记录当前epoch上所有样本的平均损失
        total_loss = 0
        for x, y in train_loader:
            # 切换为训练模式
            self.model.train()
            # 前向传播
            predicts = self.model(x)
            loss = self.loss_fn(predicts, y)
            # 记录损失并打印
            total_loss += loss.item()
            self.train_step_loss.append((step,loss.item()))
            if step % log_stride == 0 or step == total_steps - 1:
                print("[Train]  epoch:{}/{} step:{}/{} loss:{:.4f}".format(epoch, num_epochs, step, total_steps,
                                                                           loss.item()))
            # 反向传播，更新参数
            loss.backward()
            self.optimizer.step()
            self.optimizer.zero_grad()
            # 评估并记录打印
            if step % log_stride == 0 or step == total_steps - 1:
                dev_score, dev_loss = self.evaluate(dev_loader)
                self.dev_stride_score.append((step,dev_score))
                self.dev_stride_loss.append((step,dev_loss))
                print("[Evaluate]  score:{:.4f} loss:{:.4f}".format(dev_score, dev_loss))

                # # 记录最优分数，并保存模型
                # if dev_score > self.best_score:
                #     self.best_score = dev_score
                #     saves_path = os.getcwd()
                #     saves_name = "step{}.params".format(step)
                #     self.save(path=os.path.join(saves_path, saves_name))
                #     print("Best model has been updated and saved!")

            # 步数+1
            step += 1
        # 记录当前epoch损失
        self.train_epoch_loss.append(total_loss / len(train_loader))
    self.plot()

@torch.no_grad()
def evaluate(self, dev_loader):
    # 切换为评估模式
    self.model.eval()
    self.metric.reset()
    total_loss = 0
    for x, y in dev_loader:
        predicts = self.model(x)
        loss = self.loss_fn(predicts, y)
        total_loss += loss.item()
        self.metric.update(predicts, y)
    dev_loss = total_loss / len(dev_loader)
    dev_score = self.metric.accuracy()
    return dev_score, dev_loss

@torch.no_grad()
def predict(self, x):
    self.model.eval()
    predicts = self.model(x)
    preds = predicts.argmax()
    return preds

def plot(self):
    """
    在notebook中交互效果很好，pycharm中窗口一闪而过
    """
    plt.figure(figsize=(8,8))
    # 损失函数曲线
    sample_step=20
    plt.subplot(2,1,1)
    plt.xlabel("step")
    plt.ylabel("loss")
    train_items=self.train_step_loss[::sample_step]
    train_x=[x[0] for x in train_items]
    train_loss=[x[1] for x in train_items]
    plt.plot(train_x,train_loss,label="Train loss")
    plt.plot(*zip(*self.dev_stride_loss),label="Dev loss")
    plt.legend()

    # 得分曲线
    plt.subplot(2,1,2)
    plt.xlabel("step")
    plt.ylabel("score")
    plt.plot(*zip(*self.dev_stride_score),label="Dev score")
    plt.legend()
    # 显示图标与图像

    plt.tight_layout()
    plt.draw()

# def save(self, path):
#     torch.save(self.model.state_dict(), path)
#
```

```
    # def load(self, path):
    #     state_dict = torch.load(path)
    #     self.model.set_state_dict(state_dict)
```

9.使用我们上面构建的框架完成鸢尾话分类任务

In [10]:
```
import numpy as np
import pandas as pd
 # 创建浮点数时默认设为float64, 保证数据x和模型参数为同一类型(labels必须为int64)
torch.set_default_dtype(torch.float64)

 # 在notebook中运行如果提示"The kernel appears to have died. It will restart automatically."
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

# 数据打乱和模型参数初始化时用到了随机, 设置随机种子保证模型结果可复现
torch.manual_seed(102)
```

Out[10]:
```
<torch._C.Generator at 0x1fa9d95e030>
```

第一步: Dataset和DataLoader定义

In [11]:
```
data=pd.read_csv("../data/dataset/Iris.csv",index_col="Id")
data.head()
```

Out[11]:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
Id					
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa

In [12]:
```
class IRIS_Dataset(Dataset):
    def __init__(self):
        data=pd.read_csv("../data/dataset/Iris.csv",index_col="Id")
        self.x=torch.tensor(data.loc[:,"SepalLengthCm":"PetalWidthCm"].values)
        vocab={'Iris-setosa':0,'Iris-versicolor':1,'Iris-virginica':2}
        self.y=torch.tensor(data.Species.map(vocab).values)
    def __getitem__(self,idx):
        return self.x[idx],self.y[idx]
    def __len__(self):
        return self.x.shape[0]
```

In [13]:
```
train_set=IRIS_Dataset()
train_loader=DataLoader(train_set,shuffle=True,batch_size=4)
```

第二步: 依次定义model,loss_fn,optimizer,metric

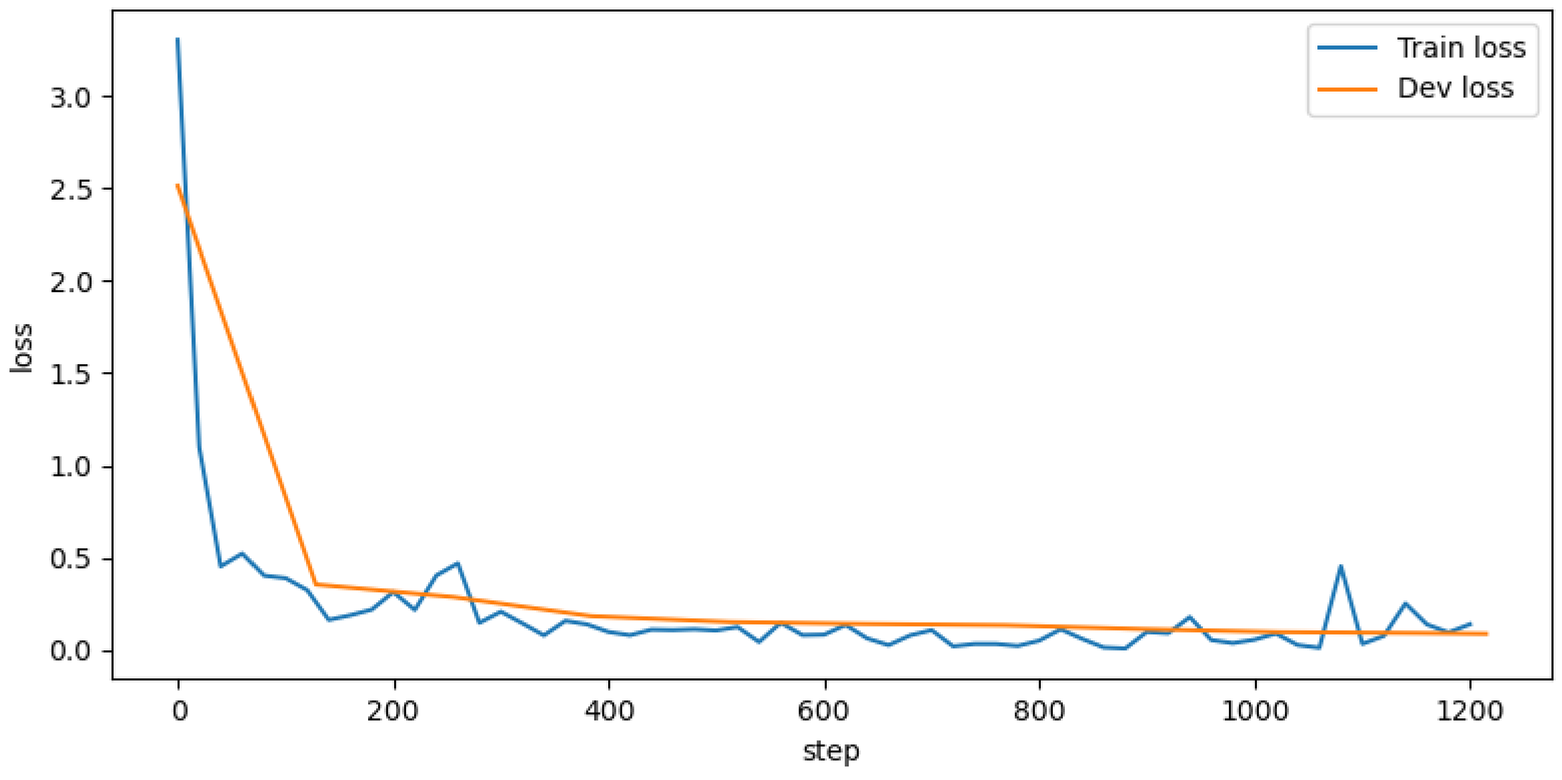
In [14]:
```
class MLP_L2(Module):
    def __init__(self):
        self.linear1=Linear(input_size=4,output_size=8)
        self.logistic=Logistic()
        self.linear2=Linear(input_size=8,output_size=4)
        self.softmax=Softmax()
    def __call__(self,x):
        return self.forward(x)
    def forward(self,x):
        x=self.linear1(x)
        x=self.logistic(x)
        x=self.linear2(x)
        x=self.softmax(x)
        return x
```

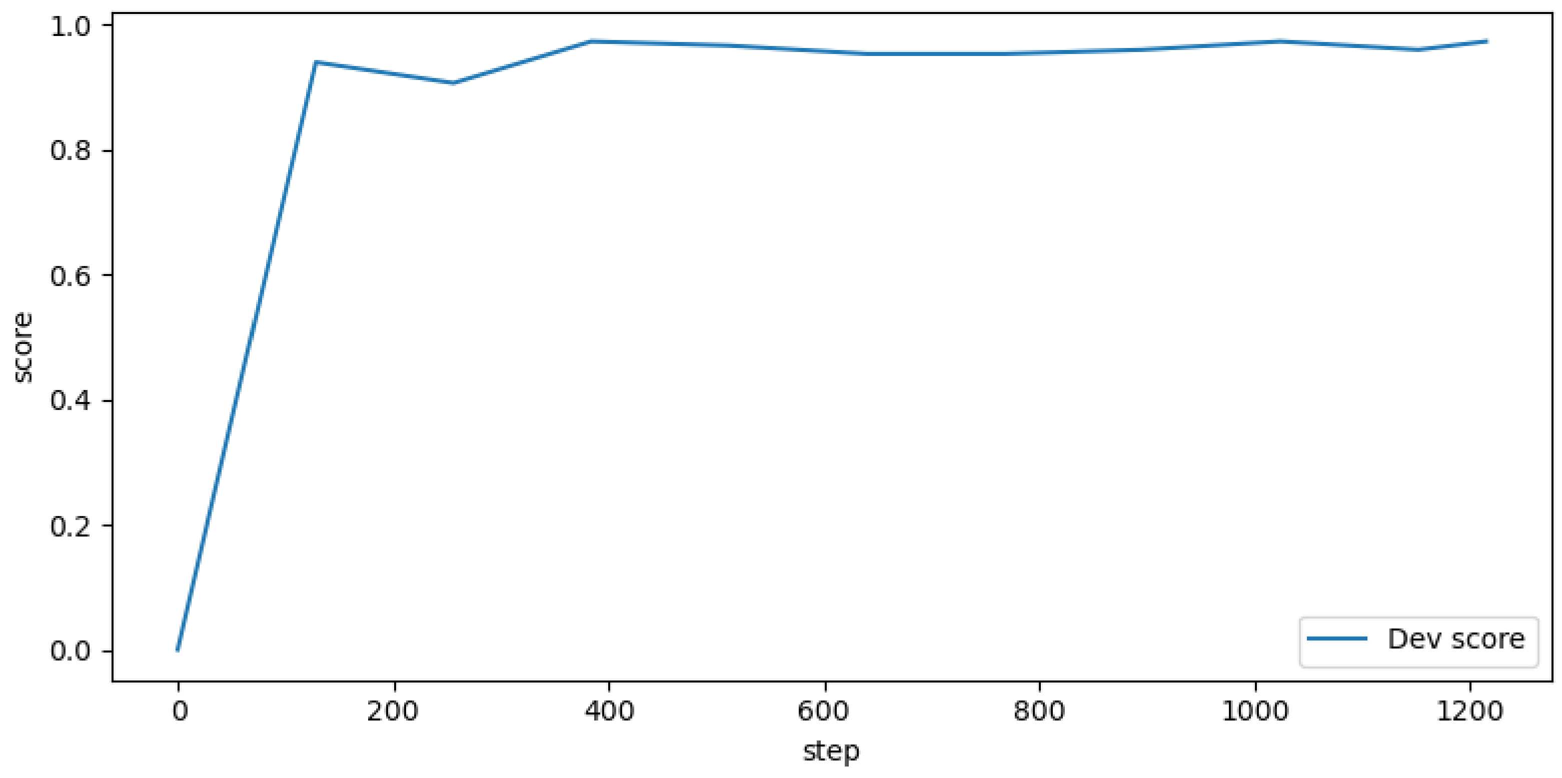
In [15]:
```
model=MLP_L2()
loss_fn=CrossEntropyLoss()
metric=Accuracy()
opt=Adam(model.parameters(),lr=0.01)
```

第三步: 实例化Runner, 并训练

In [16]:
```
runner=RunnerV1(model=model,loss_fn=loss_fn,optimizer=opt,metric=metric)
runner.train(train_loader=train_loader,dev_loader=train_loader,num_epochs=32,log_stride=128)
```

```
[Train]  epoch:0/32 step:0/1216 loss:3.3022
[Evaluate]  score:0.0000 loss:2.5119
[Train]  epoch:3/32 step:128/1216 loss:0.4986
[Evaluate]  score:0.9400 loss:0.3553
[Train]  epoch:6/32 step:256/1216 loss:0.0605
[Evaluate]  score:0.9067 loss:0.2872
[Train]  epoch:10/32 step:384/1216 loss:0.2235
[Evaluate]  score:0.9733 loss:0.1839
[Train]  epoch:13/32 step:512/1216 loss:0.0627
[Evaluate]  score:0.9667 loss:0.1522
[Train]  epoch:16/32 step:640/1216 loss:0.0636
[Evaluate]  score:0.9533 loss:0.1417
[Train]  epoch:20/32 step:768/1216 loss:0.0807
[Evaluate]  score:0.9533 loss:0.1348
[Train]  epoch:23/32 step:896/1216 loss:0.0173
[Evaluate]  score:0.9600 loss:0.1137
[Train]  epoch:26/32 step:1024/1216 loss:0.0197
[Evaluate]  score:0.9733 loss:0.0975
[Train]  epoch:30/32 step:1152/1216 loss:0.0533
[Evaluate]  score:0.9600 loss:0.0923
[Train]  epoch:31/32 step:1215/1216 loss:0.2865
[Evaluate]  score:0.9733 loss:0.0884
```


Train loss
Dev loss
loss
step


Dev score
score
step