联邦学习+同态加密/差分隐私的实现

使用FATE框架

根目录：https://github.com/FederatedAI/FATE

FATE联邦学习+同态加密（应用示例+库源码解析）

以下采用**自顶向下**的方式说明FATE联邦学习是如何调用同态加密方法的。简单来说，同态加密是FATE做联邦学习的一个可选操作，嵌入在模型聚合的操作中。

顶层联邦应用对于同态的使用：

先给出应用的代码：

```
import torch as t
from fate.arch import Context
from fate.ml.nn.hetero.hetero_nn import HeteroNNTrainerGuest, HeteroNNTrainerHost, TrainingArguments
from fate.ml.nn.model_zoo.hetero_nn_model import HeteroNNModelGuest, HeteroNNModelHost
from fate.ml.nn.model_zoo.hetero_nn_model import SSHEArgument, FedPassArgument, TopModelStrategyArguments

def train(ctx: Context,
          dataset = None,
          model = None,
          optimizer = None,
          loss_func = None,
          args: TrainingArguments = None,
          ):

    if ctx.is_on_guest:
        trainer = HeteroNNTrainerGuest(ctx=ctx,
                                       model=model,
                                       train_set=dataset,
                                       optimizer=optimizer,
                                       loss_fn=loss_func,
                                       training_args=args
                                       )
    else:
        trainer = HeteroNNTrainerHost(ctx=ctx,
                                      model=model,
```

```
                                    train_set=dataset,
                                    optimizer=optimizer,
                                    training_args=args
                                )

    trainer.train()
    return trainer

def predict(trainer, dataset):
    return trainer.predict(dataset)

def get_setting(ctx):

    from fate.ml.nn.dataset.table import TableDataset
    # 准备数据
    if ctx.is_on_guest:
        ds = TableDataset(to_tensor=True)
        ds.load("./breast_hetero_guest.csv")

        bottom_model = t.nn.Sequential(
            t.nn.Linear(10, 8),
            t.nn.ReLU(),
        )
        top_model = t.nn.Sequential(
            t.nn.Linear(8, 1),
            t.nn.Sigmoid()
        )
        model = HeteroNNModelGuest(
            top_model=top_model,
            bottom_model=bottom_model,
            agglayer_arg=SSHEArgument(
                guest_in_features=8,
                host_in_features=8,
                out_features=8,
                layer_lr=0.01
            )
        )

        optimizer = t.optim.Adam(model.parameters(), lr=0.01)
        loss = t.nn.BCELoss()

    else:
        ds = TableDataset(to_tensor=True)
        ds.load("./breast_hetero_host.csv")
        bottom_model = t.nn.Sequential(
            t.nn.Linear(20, 8),
            t.nn.ReLU(),
```

```
        )

        model = HeteroNNModelHost(
            bottom_model=bottom_model,
            agglayer_arg=SSHEArgument(
                guest_in_features=8,
                host_in_features=8,
                out_features=8,
                layer_lr=0.01
            )
        )
        optimizer = t.optim.Adam(model.parameters(), lr=0.01)
        loss = None

    args = TrainingArguments(
        num_train_epochs=3,
        per_device_train_batch_size=256
    )

    return ds, model, optimizer, loss, args

def run(ctx):
    ds, model, optimizer, loss, args = get_setting(ctx)
    trainer = train(ctx, ds, model, optimizer, loss, args)
    pred = predict(trainer, ds)
    if ctx.is_on_guest:
        from sklearn.metrics import roc_auc_score
        print('auc is')
        print(roc_auc_score(pred.label_ids, pred.predictions))


if __name__ == '__main__':
    from fate.arch.launchers.multiprocess_launcher import launch
    launch(run)

```

以上代码用了简单的网络做了一个纵向联邦学习的分类任务，下面阐述**客户端**（Guest）的逻辑。

Step 1. 以上代码在**客户端**模型做同态加密，客户端模型使用**HeteroNNModelGuest**类，其聚合层类型是**SSHEArgument**，其中具体代码是：

```
# 摘取自上一段代码
model = HeteroNNModelGuest(
            top_model=top_model,
            bottom_model=bottom_model,
```

```
            agglayer_arg=SSHEArgument(
                guest_in_features=8,
                host_in_features=8,
                out_features=8,
                layer_lr=0.01
            )
        )
```

Step 2. 前一步代码实例化了一个**HeteroNNModelGuest**对象，该类的实现在python/fate/ml/nn/model_zoo/hetero_nn_model.py，它包含聚合层（用来做Guest与Host共享信息），但并没有实现聚合层，只是把聚合层的类型**SSHEArgument**传递给下一层，具体代码是：

```
# python/fate/ml/nn/model_zoo/hetero_nn_model.py
if self._agg_layer is None:
    if agglayer_arg is None:
        self._agg_layer = AggLayerGuest()
    elif type(agglayer_arg) == StdAggLayerArgument:
        self._agg_layer = AggLayerGuest(**agglayer_arg.to_dict())
    elif type(agglayer_arg) == FedPassArgument:
        self._agg_layer = FedPassAggLayerGuest(**agglayer_arg.to_dict())
    elif type(agglayer_arg) == SSHEArgument:
        self._agg_layer = SSHEAggLayerGuest(**agglayer_arg.to_dict())
        if self._bottom_model is None:
            raise RuntimeError("A bottom model is needed when running a SSHE model")
### 中间省略若干行 ###
self._agg_layer.set_context(ctx)
```

Step 3. 前一步代码是要实例化一个**SSHEAggLayerGuest**聚合层，它的实现在python/fate/ml/nn/model_zoo/agg_layer/sshe/agg_layer.py，相关代码是：

```
# python/fate/ml/nn/model_zoo/agg_layer/sshe/agg_layer.py
def setup_model(self, ctx: Context):
    generator = torch.Generator()
    self._model = SSHENeuralNetworkAggregatorLayer(
        ### 中间省略若干行 ###
    )
    self._optimizer = SSHENeuralNetworkOptimizerSGD(ctx, self._model.parameters(), lr=self._layer_lr)

class SSHEAggLayerGuest(_AggLayerBase):
    ### 中间省略若干行 ###
    def set_context(self, ctx: Context):
```

```
        if isinstance(ctx, Context):
            super().set_context(ctx)
            setup_model(self, ctx)
     ### 后续省略 ###
```

可以看出这一步仍然不是聚合层模型的真正实现，**SSHEAggLayerGuest**聚合层要实例化一个**SSHENeuralNetworkAggregatorLayer**来做实际的模型。

Step 4. 前一步想要实例化的**SSHENeuralNetworkAggregatorLayer**的实现在python/fate/arch/protocol/mpc/nn/sshe/nn_layer.py，相关代码如下：

```
# python/fate/arch/protocol/mpc/nn/sshe/nn_layer.py
class SSHENeuralNetworkAggregatorLayer(torch.nn.Module):
    def __init__(
        self,
        ctx: Context,
        ### 省略 ###
    ):
        self.aggregator = SSHENeuralNetworkAggregator(
                ctx,
                ### 省略 ###
            )
class SSHENeuralNetworkAggregator:
    def __init__(
        self,
        ctx: Context,
        ### 省略 ###
        cipher_options=None,
    ):
        self.ctx = ctx
        ### 省略 ###
        self.phe_cipher = ctx.cipher.phe.setup(options=cipher_options)
        ### 后续省略 ###
```

以上代码的逻辑是：聚合层的实现调用了**SSHENeuralNetworkAggregator**，核心是最后一行，先调用ctx（FATE框架的上下文）的cipher方法，其返回一个**CipherKit**对象，这个对象相当于一个加密工具，再调用这个对象的phe方法。phe方法返回的又是一个**PHECipherBuilder**对象，最后调用了**PHECipherBuilder**对象的setup方法。其中cipher方法的实现在python/fate/arch/context/_context.py，具体代码是

```
# python/fate/arch/context/_context.py
class Context:
    ### 省略 ###
```

```
    def _set_default_phe(self):
        if "phe" not in self._cipher_mapping:
            self._cipher_mapping["phe"] = {}
        if self._device not in self._cipher_mapping["phe"]:
            if self._device == device_type.CPU:
                self._cipher_mapping["phe"][device_type.CPU] = {
                    "kind": "paillier",
                    "key_length": cfg.safety.phe.paillier.minimum_key_size,
                }
            else:
                logger.warning(f"no impl exists for device {self._device}, fallback to CPU")
                self._cipher_mapping["phe"][device_type.CPU] = self._cipher_mapping["phe"].get(
                    device_type.CPU, {"kind": "paillier", "key_length": cfg.safety.phe.paillier.minimum_key_size}
                )

    @property
    def cipher(self):
        return self._cipher
    ### 后续省略 ###
```

phe方法和setup方法(在不同的类中)的实现在python/fate/arch/context/_cipher.py，具体代码是：

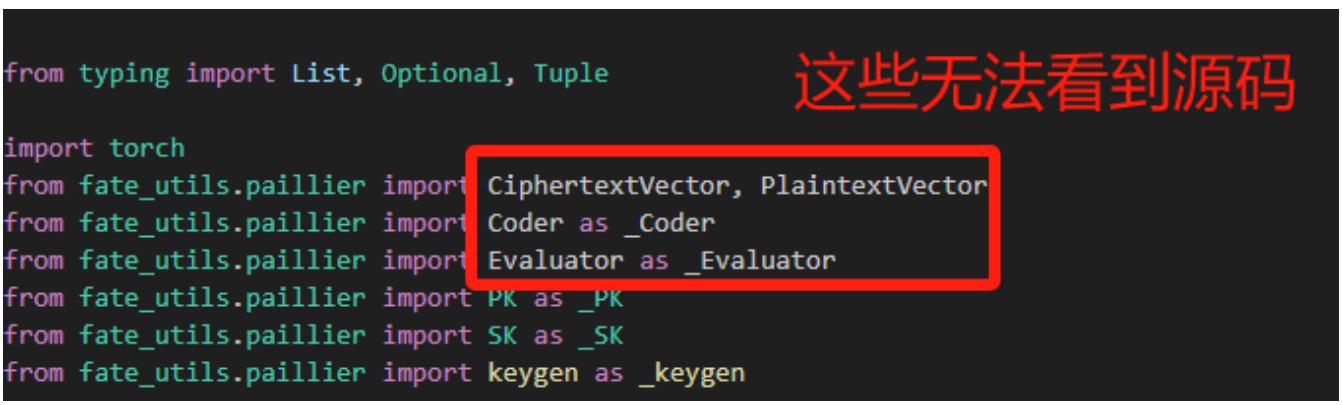
```
# python/fate/arch/context/_cipher.py
class CipherKit:
    ### 省略 ###
    @property
    def phe(self):
        if self.ctx is None:
            raise ValueError("context not set")
        self._set_default_phe()
        if self._device not in self._cipher_mapping["phe"]:
            raise ValueError(f"no impl exists for device {self._device}")
        return PHECipherBuilder(self.ctx, **self._cipher_mapping["phe"][self._device])
class PHECipherBuilder:
    ### 省略 ###
    def setup(self, options: typing.Optional[dict] = None):
        if options is None:
            kind = self.kind
            key_size = self.key_length
        else:
            kind = options.get("kind", self.kind)
```

```
            key_size = options.get("key_length", self.key_length)

        if kind == "paillier":
            if not cfg.safety.phe.paillier.allow:
                raise ValueError("paillier is not allowed in config")
            if key_size < cfg.safety.phe.paillier.minimum_key_size:
                raise ValueError(
                    f"key size {key_size} is too small, minimum is {cfg.safety.phe.paillier.minimum_key_size}"
                )
            from fate.arch.protocol.phe.paillier import evaluator, keygen
            from fate.arch.tensor.phe import PHETensorCipher

            sk, pk, coder = keygen(key_size)
            tensor_cipher = PHETensorCipher.from_raw_cipher(pk, coder, sk, evaluator)

            return PHECipher(kind, key_size, pk, sk, evaluator, coder, tensor_cipher, True, True, True)
            ### 后续省略 ###
```

由于python/fate/arch/context/_context.py的可以看到setup方法开始构建paillier加密算法了，这里生成的一个加密器，它是一个**PHETensorCipher**对象，还有一个做加密后的同态运算的evaluator，那么同态所需组件就齐全了。

Step 5. 前一步提到的加密器和evaluator，这两者的python实现都在python/fate/arch/protocol/phe/paillier.py，其中前者作为**PHETensorCipher**对象先调用python/fate/arch/tensor/phe/_keypair.py，形成公钥、私钥、同态编码器的逻辑关系之后，再调用的paillier.py来做这三者各自的实现（**paillier.py是paillier同态加密的python源码，但不是最底层**，它调用的是rust源码中的一些开放的python接口，此处省略SK, PK, Coder的py源码）。

如果使用VSCode打开paillier.py文件，会发现有一些类是白色的，也无法用Ctrl+左键进入源码：



这是因为它们根本不是用python实现的，而是用rust。

Step 6. 真正的paillier同态加密的加密、同态运算的底层实现，在一个rust文件rust/fate_utils/crates/fate_utils/src/paillier/paillier.rs，它用**pyclass**和**pymethods**写了一些接口，

然后就能把这个.rs文件编译成python能够import的类或者方法：

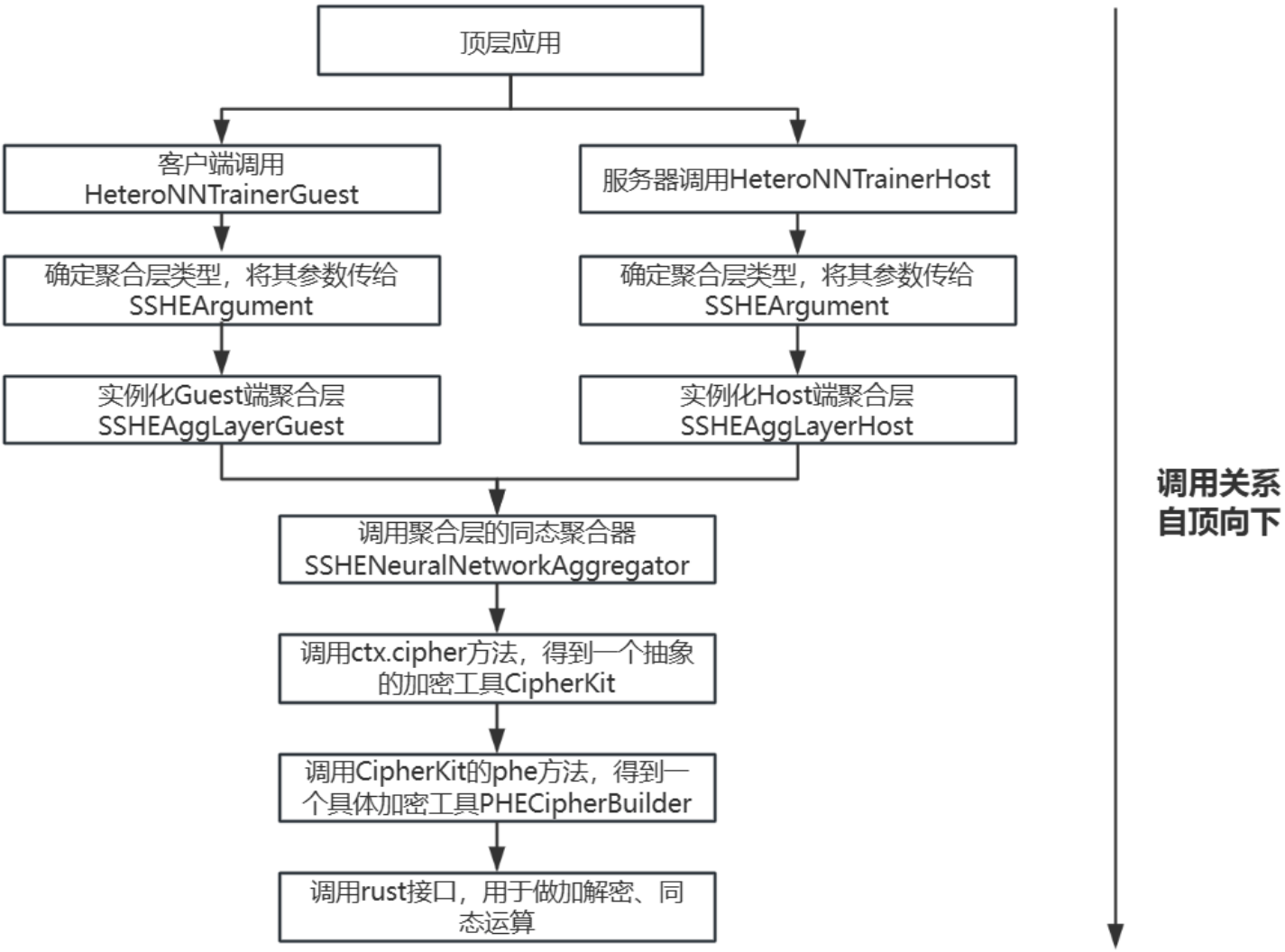
```rust
#[pyclass(module = "fate_utils.paillier")]
#[derive(Default)]
pub struct PK(fixedpoint_paillier::PK);

#[pyclass(module = "fate_utils.paillier")]
#[derive(Default)]
pub struct SK(fixedpoint_paillier::SK);

#[pyclass(module = "fate_utils.paillier")]
#[derive(Default)]
pub struct Coder(fixedpoint_paillier::Coder);

#[pyclass(module = "fate_utils.paillier")]
#[derive(Default)]
pub struct Ciphertext(fixedpoint_paillier::Ciphertext);

#[pyclass(module = "fate_utils.paillier")]
#[derive(Default, Debug)]
pub struct CiphertextVector(fixedpoint_paillier::CiphertextVector);

#[pyclass(module = "fate_utils.paillier")]
#[derive(Default, Debug)]
pub struct PlaintextVector(fixedpoint_paillier::PlaintextVector);

#[pyclass(module = "fate_utils.paillier")]
#[derive(Default)]
pub struct Plaintext(fixedpoint_paillier::Plaintext);

#[pyclass]
pub struct Evaluator {}

#[pymethods]
impl PK {
    //省略
}

#[pymethods]
impl SK {
    //省略
}

#[pymethods]
impl Coder {
    // 省略
```

```
45 }
```

至此，FATE框架在**Guest端**做Paillier同态加密的流程，自顶向下解释完毕。

从**Step 1** - **Step 3**，在**Host端**的动作是基本对称的，不同之处在于**Step 3**这一步，在python/fate/ml/nn/model_zoo/agg_layer/sshe/agg_layer.py文件中，对两个端做了行为的区分：Guest端做的从本地数据中提取特征，并将这些特征传递给聚合层再做加密；Host端做的是处理从Guest端接收到的加密后的聚合特征，可以执行特征变换、非线性映射等计算，最终将结果传递给聚合层，用于同态运算（evaluation）。

从**Step 4**开始，Host端和Guest端的动作就一致了，它们调用的都是**SSHENeuralNetworkAggregatorLayer**，这个类用于实现聚合层的加密器。虽然是调用同一个类，但是仍然是2个不同的对象（是2个分居两端的聚合层）。因为Guest端做加解密，Host端做同态evaluation。



FATE联邦学习+差分隐私（理论，暂无实现方案）

在FATE的源码中，有一个文件做了一种待完善的差分隐私的分布式训练方法，在python/fate/arch/protocol/mpc/nn/privacy/dp_split.py，它提出了联邦+同态+差分的训练步骤：

1. 客户端在明文上做前向传播（forward pass），得到模型的原始输出分数（通常记为Z或logits）；
2. 客户端根据logits，使用sigmoid、softmax等函数得到预测结果；
3. 客户端计算损失L，对L和Z做（同态）加密；
4. 客户端在密文上计算梯度dL/dZ，把梯度加上噪声，再发送给服务器；
5. 服务器对于所有客户端各自处理后的梯度做聚合，聚合结果返回给客户端；
6. 客户端做解密，得到聚合后梯度的明文；
7. 客户端更新模型梯度。

但是代码中有很多必要的函数没有实现，因此此处暂时无法给出这一套技术组合的实现方案。

并且，这种方案下存在一个难以解决的问题：所有客户端都加噪声的情况下，如果客户端增多，噪声会变得特别大，从而会很大程度掩盖原始数据的特征，导致训练结果较差。

使用Pysyft框架

github仓库：https://github.com/OpenMined/PySyft

官方教程：https://docs.openmined.org/en/latest/getting-started/introduction.html

新版本Pysyft（≥0.6.0）

目前（0.6.0版本及以后）没有在Pysyft框架下的联邦+同态/差分的相关指引，可能是因为引入了**Datasite**的概念（从website借鉴而来，Datasite帮助数据科学家从服务器中的数据下载问题的答案，而无需下载数据本身）以后，Pysyft能让数据所有者可以通过定义访问控制策略，确保数据不被未授权方访问，那么也就基本消除了对数据做同态/差分变换的需求。

例如数源方可以自己随意定义虚假的X_mock和y_mock，让其他方知道数据形状、字段格式的同时，隐藏其他所有信息（实际上形状、字段格式也可以是虚假的，但是这样做无意义）。在定义好X和y的情况下，代码如下：

```
import numpy as np
# fix seed for reproducibility
SEED = 12345
np.random.seed(SEED)
X_mock = X.apply(lambda s: s + np.mean(s) + np.random.uniform(size=len(s)))
y_mock = y.sample(frac=1, random_state=SEED).reset_index(drop=True)
```

可以看到这里的X_mock和y_mock跟原来的数据可以没有任何关系。此后的目标是：在本地或者中心服务器上建立一个Datasite，其他方获取数据时就只能从Datasite上获取到X_mock和y_mock，而不能获得X和y。代码如下：

```
features_asset = sy.Asset(
    name="Breast Cancer Data: Features",
    data = X,      # real data
    mock = X_mock  # mock data
)

targets_asset = sy.Asset(
    name="Breast Cancer Data: Targets",
    data = y,      # real data
    mock = y_mock  # mock data
)

# Metadata
description = f'{metadata["abstract"]}\n{metadata["additional_info"]["summary"]}'
paper = metadata["intro_paper"]
citation = f'{paper["authors"]} - {paper["title"]}, {paper["year"]}'
summary = "The Breast Cancer Wisconsin dataset can be used to predict whether the cancer is benign or malignant."

# Dataset creation
breast_cancer_dataset = sy.Dataset(
    name="Breast Cancer Biomarker",
    description=description,
    summary=summary,
    citation=citation,
    url=metadata["dataset_doi"],
)

breast_cancer_dataset.add_asset(features_asset)
breast_cancer_dataset.add_asset(targets_asset)
```

这样，其他方就能通过访问这个Datasite来获取想要的虚拟数据。并且，其他方也能够根据虚拟数据来做模型训练，因为Datasite能通过指针的方式把其他方的建模动作映射到数源上来做实际的建模和计算，然后把结果返回给其他方。

旧版本Pysyft（<0.6.0）

如果使用更早版本的Pysyft框架，可参考Pysyft的官方课程（是间接开源，我把它放在 https://github.com/FCLV/syft_courses），课程官网在 https://courses.openmined.org/courses/foundations-of-private-computation，是基于syft 0.5.0实现的。但是在Datasite被提出来之后，可能不再需要使用。

以下给出旧版本Pysyft做联邦学习+X的实现逻辑：

代码可参考https://github.com/FCLV/syft_courses/tree/master/federated-learning/duet_fl目录下的3个文件。

步骤如下：

1. 每个数源启动一个duet会话；
2. 中心服务器加入所有数源的duet会话；
3. 数源方在本地训练模型，得到梯度；
4. 数源方对梯度做同态/差分处理，通过duet发送给中心服务器（这里可使用带有Python接口的同态加密库或者差分隐私库，例如代码整体在PyTorch框架下，则使用同态加密库TenSEAL，差分隐私库Opacus，代码示例中暂未给出加密或者差分处理操作）；
5. 中心服务器通过所有duet收集到的梯度，完成聚合；
6. 中心服务器将聚合之后的梯度通过各个duet发送给各个数源；
7. 数源方做解密或者直接得到更新后的梯度。