GPU Performance Solution

> 机器学习概论lab1
>
> Author:@Rosykunai
>
> Date:2024年10月

GPU Performance Solution

1.预测GPU的运行时间

1.1 数据预处理

(a)

必要的操作:取对数(在Dataset Card的Information 的Note部分有说明)

标准化[Optional]:在`data.ipynb`文件中观察到前十个特征数据分布与后面的特征差异较大,对特征做标准化处理使得不同特征的权重相当

```
def data_preprocessing_regression(data_path: str, saved_to_disk: bool = False) -> Dataset:
    r"""Load and preprocess the training data for the regression task.

    Args:
        data_path (str): The path to the training data.If you are using a dataset saved with save_to_disk(), you can use load_from_disk() to load the dataset.

    Returns:
        dataset (Dataset): The preprocessed dataset.
    """
    # 1.1-a

```

```
    # Load the dataset. Use load_from_disk() if you are using a dataset
saved with save_to_disk()
    if saved_to_disk:
        dataset  = load_from_disk(data_path)
    else:
        dataset = load_dataset(data_path)
    # Preprocess the dataset
    # Use dataset.to_pandas() to convert the dataset to a pandas DataFrame
if you are more comfortable with pandas
    # TODO: You must do something in 'Run_time' column, and you can also
do other preprocessing steps

    dataset = dataset['train'].to_pandas()

    for index in range(10):
        dataset.iloc[:,index] = (dataset.iloc[:,index]-
dataset.iloc[:,index].mean()) / dataset.iloc[:,index].std()

    dataset['Run_time'] = np.log(dataset['Run_time'])

    dataset = Dataset.from_pandas(dataset)
    # dataset = Dataset.from_pandas(dataset) # Convert the pandas
DataFrame back to a dataset
    return dataset
```

涉及到`datasets`库和`pandas`库的基本用法

(b)

划分数据集,首先按照4:1的比例划分训练集和"测试集",再把"测试集"一分为二分别作为验证集和测试集，三种情况分别为:

1. [trainset],[valset]
2. [trainset,valset],[testset]
3. [trainset,valset,testset],[]

注意每次返回的作为训练使用的`Dataloader`需要把参数`train`设置为`True`

```
def data_split_regression(dataset: Dataset, batch_size: int, shuffle:
bool) -> Tuple[DataLoader]:
    r"""Split the dataset and make it ready for training.

    Args:
        dataset (Dataset): The preprocessed dataset.
        batch_size (int): The batch size for training.
        shuffle (bool): Whether to shuffle the data.

    Returns:
```

```
        A tuple of DataLoader: You should determine the number of
DataLoader according to the number of splits.
    """
    # 1.1-b
    # Split the dataset using dataset.train_test_split() or other methods
    # TODO: Split the dataset
    dataset_split = dataset.train_test_split(test_size=0.2)
    trainset = dataset_split["train"] # Train set
    test_set = dataset_split["test"]

    val_test_set = test_set.train_test_split(test_size=0.5)
    val_set = val_test_set["train"] # Validation set
    test_set = val_test_set["test"] # Test set

    trainset = concatenate_datasets([trainset, val_set])

    # Create a DataLoader for each split
    # TODO: Create a DataLoader for each split

    train_loader = DataLoader(trainset, batch_size=batch_size,
shuffle=shuffle, train=True)
    #val_loader = DataLoader(val_set, batch_size=batch_size,
shuffle=shuffle)
    test_loader = DataLoader(test_set, batch_size=batch_size,
shuffle=shuffle)
    return train_loader, test_loader
```

划分数据集的方法在`submission.py`第59行注释,合并数据集的方法在`submission.py`第7行import包含,

`Dataloader`的定义在`utils.py`第170行,其中第196行说明了参数`train`的含义。

1.2 定义模型

线性回归的参数分别为:

1. weight:[in_features, out_features]
2. bias:[out_features]

这里`out_features`为1,对于输入[B,F],模型的输出会被广播为:

$[B, F] \times [F, 1] + [1] = [B, 1]$

```
class LinearRegression(BaseModel):
    r"""A simple linear regression model.

    This model takes an input shaped as [batch_size, in_features] and
returns
    an output shaped as [batch_size, out_features].
```

```
    For each sample [1, in_features], the model computes the output as:

    .. math::
        y = xW + b

    Args:
        in_features (int): Number of input features.
        out_features (int): Number of output features.

    Example::

        >>> from model import LinearRegression
        >>> # Define the model
        >>> model = LinearRegression(3, 1)
        >>> # Predict
        >>> x = np.random.randn(10, 3)
        >>> y = model(x)
        >>> # Save the model parameters
        >>> state_dict = model.state_dict()
        >>> save(state_dict, 'model.pkl')
    """
    def __init__(self, in_features: int, out_features: int):
        super().__init__()
        # 1.2-a
        # Look up the definition of BaseModel and Parameter in the
utils.py file, and use them to register the parameters
        # TODO: Register the parameters
        self.weight = Parameter(np.random.randn(in_features,
out_features))
        self.bias = Parameter(np.random.randn(out_features))

    def predict(self, x: np.ndarray) -> np.ndarray:
        # 1.2-b
        # Implement the forward pass of the model
        # TODO: Implement the forward pass
        return x @ self.weight + self.bias
```

这里注册参数的方法在`model.py`的第32行,`predict`方法涉及矩阵乘法和`numpy`数组的基本运算方法。

1.3 定义MSELoss

首先把`y_true`的shape从[B,]扩展到[B,1],然后计算`MSE`

在梯度计算部分,`weight`的梯度由

$[B,F]^T \times [B,1] = [F,1](= self.weight.shape)$

消去batch维度

对于bias的梯度记得保持[1,]的shape

```python
class MSELoss(Loss):
    r"""Mean squared error loss.

    This loss computes the mean squared error between the predicted and true values.

    Methods:
        __call__: Compute the loss
        backward: Compute the gradients of the loss with respect to the parameters
    """
    def __call__(self, y_pred: np.ndarray, y_true: np.ndarray) -> float:
        r"""Compute the mean squared error loss.

        Args:
            y_pred: The predicted values
            y_true: The true values

        Returns:
            The mean squared error loss
        """
        # 1.3-a
        # Compute the mean squared error loss. Make sure y_pred and y_true have the same shape
        # TODO: Compute the mean squared error loss
        y_true = y_true.reshape(-1, 1)
        return np.mean((y_pred - y_true) ** 2)

    def backward(self, x: np.ndarray, y_pred: np.ndarray, y_true: np.ndarray) -> dict[str, np.ndarray]:
        r"""Compute the gradients of the loss with respect to the parameters.

        Args:
            x: The input values [batch_size, in_features]
            y_pred: The predicted values [batch_size, out_features]
            y_true: The true values [batch_size, out_features]

        Returns:
            The gradients of the loss with respect to the parameters, Dict[name, grad]
        """
        # 1.3-b
        # Make sure y_pred and y_true have the same shape
```

```
        # TODO: Compute the gradients of the loss with respect to the parameters
        y_true = y_true.reshape(-1, 1) # [batch_size, 1]
        error = y_pred - y_true

        # Gradients with respect to the weights and bias for each sample
        weight_grad = 2 * (x.T @ error) / x.shape[0] # [in_features, out_features]
        bias_grad = 2 * error.mean().reshape(1,)  # [out_features]

        return {"weight": weight_grad, "bias": bias_grad}
```

注意张量求导的计算,尤其注意对齐shape

1.4 Train_loop

首先从`Dataloader`中加载一个`batch`,在issue #2中我们讨论了为什么使用`next`方法而不是`for`循环,

这时`batch`是一个`np.ndarray`,我们按照切片的方式分离`features`和`target`,接下来:

1. `features`通过模型(`model.py`第39行)
2. 计算loss(`Loss`类型的`__call__`方法)并记录
3. 计算梯度(`Loss`类型的`backward`方法)
4. 更新参数(`SGD`类型的`step`方法,`utils.py`第99行)

```
def train(self):
        loss_list = []
        with tqdm(
            initial=self.step,
            total=self.train_num_steps,
        ) as pbar:
            while self.step < self.train_num_steps:
                # 1.4-a
                # load data from train_loader and compute the loss
                # TODO: Load data from train_loader and compute the loss
                batch = next(self.train_loader)
                x = batch[:, :-1]
                y = batch[:, -1]
                y_pred = self.model(x)
                loss = self.criterion(y_pred, y)
                loss_list.append(loss)
                pbar.set_description(f"Loss: {loss:.6f}")



                # Use pbar.set_description() to display current loss in the progress bar


```

```python
                # Compute the gradients of the loss with respect to the parameters
                # Update the parameters with the gradients
                # TODO: Compute gradients and update the parameters
                grads = self.criterion.backward(x, y_pred, y)
                self.opt.step(grads)

                self.step += 1
                pbar.update()
```

按顺序依次调用相关函数即可,需要大家耐心阅读文档和注释。

1.5 Train

只提一点,调参时应该使用命令行传递参数就行示例中的`--reasults_path`一样,许多同学直接修改了`train.py`的默认值,这种做法是不合适的。

1.6 评估模型

这里使用的不是用于训练的`Dataloader`可以使用`for`循环迭代,还是需要注意对齐shape

```python
def eval_LinearRegression(model: LinearRegression, loader: DataLoader) -> Tuple[float, float]:
    r"""Evaluate the model on the given data.

    Args:
        model (LinearRegression): The model to evaluate.
        loader (DataLoader): The data to evaluate on.

    Returns:
        Tuple[float, float]: The average prediction, relative error.
    """
    model.eval()
    pred = np.array([])
    target = np.array([])
    # 1.6-a
    # Iterate over the data loader and compute the predictions
    # TODO: Evaluate the model
    for batch in tqdm(loader):
        x, y = batch[:, :-1], batch[:, -1]
        y_pred = model(x)
        y_pred = y_pred.reshape(-1)
        pred = np.append(pred, y_pred)
        target = np.append(target, y)


    # Compute the mean Run_time as Output
    # You can alse compute MSE and relative error
    # TODO: Compute metrics
```

```
    mse = np.mean((pred - target) ** 2)
    print(f"Mean Squared Error: {mse}")

    from sklearn.metrics import r2_score
    r2 = r2_score(target, pred)
    print(f"R2 Score: {r2}")

    mu = pred.mean()
    mu_target = target.mean()
    print(mu_target)

    relative_error = np.abs(mu - mu_target) / mu_target
    print(f"Relative Error: {relative_error}")

    return mu, relative_error
```

2 对GPU的表现进行分类

2.1 数据预处理

与1.1类似,这次多标准化了几个特征,根据传入的`mean`值进行分类,最后记得删除`Run_time`列

```
def data_preprocessing_classification(data_path: str, mean: float, saved_to_disk: bool = False) -> Dataset:
    r"""Load and preprocess the training data for the classification task.

    Args:
        data_path (str): The path to the training data.If you are using a dataset saved with save_to_disk(), you can use load_from_disk() to load the dataset.
        mean (float): The mean value to classify the data.

    Returns:
        dataset (Dataset): The preprocessed dataset.
    """
    # 2.1-a
    # Load the dataset. Use load_from_disk() if you are using a dataset saved with save_to_disk(
    if saved_to_disk:
        dataset = load_from_disk(data_path)
    else:
        dataset = load_dataset(data_path)
    # Preprocess the dataset
    # Use dataset.to_pandas() to convert the dataset to a pandas DataFrame if you are more comfortable with pandas
    # TODO: You must do something in 'Run_time' column, and you can also do other preprocessing steps
```

```
    dataset = dataset['train'].to_pandas()

    for index in range(14):
        dataset.iloc[:,index] = (dataset.iloc[:,index]-
dataset.iloc[:,index].mean()) / dataset.iloc[:,index].std()

    dataset['Run_time'] = np.log(dataset['Run_time'])

    dataset['label'] = dataset['Run_time'].apply(lambda x: 1 if x > mean
else 0)
    dataset = dataset.drop(columns=['Run_time'])

    dataset = Dataset.from_pandas(dataset)
    # dataset = Dataset.from_pandas(dataset) # Convert the pandas
DataFrame back to a dataset
    return dataset
```

(b)

类似1.1-(b)这次不用加载`Dataloader`

```
def data_split_classification(dataset: Dataset) -> Tuple[Dataset]:
    r"""Split the dataset and make it ready for training.

    Args:
        dataset (Dataset): The preprocessed dataset.

    Returns:
        A tuple of Dataset: You should determine the number of Dataset
according to the number of splits.
    """
    # 2.1-b
    # Split the dataset using dataset.train_test_split() or other methods
    # TODO: Split the dataset
    dataset_split = dataset.train_test_split(test_size=0.2)
    train_set = dataset_split["train"] # Train set
    test_set = dataset_split["test"]

    val_test_set = test_set.train_test_split(test_size=0.5)
    val_set = val_test_set["train"] # Validation set
    test_set = val_test_set["test"] # Test set

    train_set = concatenate_datasets([train_set, val_set])

    return train_set, test_set
```

2.2 定义模型

合并注册参数,参数shape为[F+1,1],

在`predict`部分,对输入x:

$\sigma([B, F+1] \times [F+1, 1]) = [B, 1]$

```
class LogisticRegression(BaseModel):
    r"""A simple logistic regression model for binary classification.

    This model takes an input shaped as [batch_size, in_features] and
returns
    an output shaped as [batch_size, 1].

    For each sample [1, in_features], the model computes the output as:

    .. math::
        y = \sigma(xW + b)

    where :math:`\sigma` is the sigmoid function.

    .. Note::
        The model outputs the probability of the input belonging to class
1.
        You should use a threshold to convert the probability to a class
label.

    Args:
        in_features (int): Number of input features.

    Example::

            >>> from model import LogisticRegression
            >>> # Define the model
            >>> model = LogisticRegression(3)
            >>> # Predict
            >>> x = np.random.randn(10, 3)
            >>> y = model(x)
            >>> # Save the model parameters
            >>> state_dict = model.state_dict()
            >>> save(state_dict, 'model.pkl')
    """
    def __init__(self, in_features: int):
        super().__init__()
        # 2.2-a
        # Look up the definition of BaseModel and Parameter in the
utils.py file, and use them to register the parameters
```

```
        # This time, you should combine the weights and bias into a single parameter
        # TODO: Register the parameters
        self.beta = Parameter(np.random.randn(in_features + 1, 1))

    def predict(self, x: np.ndarray) -> np.ndarray:
        r"""Predict the probability of the input belonging to class 1.

        Args:
            x: The input values [batch_size, in_features]

        Returns:
            The probability of the input belonging to class 1 [batch_size, 1]
        """
        # 2.2-b
        # Implement the forward pass of the model
        # TODO: Implement the forward pass
        return 1 / (1 + np.exp(- x @ self.beta))
```

2.3 定义BCELoss

计算`BCELoss`时涉及`log`运算,为了避免溢出,我们需要把`y_pred`裁剪到一个合理区间内,其余操作和1.3类似

```
class BCELoss(Loss):
    r"""Binary cross entropy loss.

    This loss computes the binary cross entropy loss between the predicted and true values.

    Methods:
        __call__: Compute the loss
        backward: Compute the gradients of the loss with respect to the parameters
    """
    def __call__(self, y_pred: np.ndarray, y_true: np.ndarray) -> float:
        r"""Compute the binary cross entropy loss.

        Args:
            y_pred: The predicted values
            y_true: The true values

        Returns:
            The binary cross entropy loss
        """
        # 2.3-a
```

```
        # Compute the binary cross entropy loss. Make sure y_pred and
y_true have the same shape
        # TODO: Compute the binary cross entropy loss
        y_true = y_true.reshape(-1, 1)
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1
- y_pred))

    def backward(self, x: np.ndarray, y_pred: np.ndarray, y_true:
np.ndarray) -> dict[str, np.ndarray]:
        r"""Compute the gradients of the loss with respect to the
parameters.

        Args:
            x: The input values [batch_size, in_features]
            y_pred: The predicted values [batch_size, out_features]
            y_true: The true values [batch_size, out_features]

        Returns:
            The gradients of the loss with respect to the parameters
[Dict[name, grad]]
        """
        # 2.3-b
        # Make sure y_pred and y_true have the same shape
        # TODO: Compute the gradients of the loss with respect to the
parameters
        y_true = y_true.reshape(-1, 1) # [batch_size, 1]
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        error = y_pred - y_true

        # Gradients with respect to the weights and bias for each sample
        grad = (x.T @ error) / x.shape[0]

        return {"beta": grad}
```

2.4 Train_loop

与1.4不同的是,这次需要对`features`手动添加一列"1"来适配我们的模型。除此之外,这次我们使用梯度下降,比随机梯度下降收敛速度快很多,可以在`Loss`变化不大时即时停止。

```
def train(self):
        loss_list = []
        with tqdm(
            initial=self.step,
            total=self.train_num_steps,
        ) as pbar:
            while self.step < self.train_num_steps:
```

```
                # 2.4-a
                # load data from train_loader and compute the loss
                # TODO: Load data from train_loader and compute the loss
                x = self.dataset[:, :-1]
                x = np.hstack([x, np.ones([x.shape[0], 1])])
                y = self.dataset[:, -1]
                y_pred = self.model(x)
                loss = self.criterion(y_pred, y)
                loss_list.append(loss)
                if (self.step > 10) and (abs(loss_list[-1] -
loss_list[-2]) < 1e-6)  :
                    break
                # Use pbar.set_description() to display current loss in
the progress bar
                pbar.set_description(f"Loss: {loss:.6f}")
                # Compute the gradients of the loss with respect to the
parameters
                # Update the parameters with the gradients
                # TODO: Compute gradients and update the parameters
                grads = self.criterion.backward(x, y_pred, y)
                self.opt.step(grads)
                self.step += 1
                pbar.update()
```

2.6 评估模型

和1.6类似,注意给`features`手动添加一列"1"来适配我们的模型

```
def eval_LogisticRegression(model: LogisticRegression, dataset:
np.ndarray) -> float:
    r"""Evaluate the model on the given data.

    Args:
        model (LogisticRegression): The model to evaluate.
        dataset (np.ndarray): Test data

    Returns:
        float: The accuracy.
    """
    model.eval()
    correct = 0
    # 2.6-a
    # Iterate over the data and compute the accuracy
    # This time, we use the whole dataset instead of a DataLoader.Don't
forget to add a bias term to the input
    # TODO: Evaluate the model
    for i in tqdm(range(dataset.shape[0])):
```

```python
        x, y = dataset[i, :-1], dataset[i, -1]
        x = np.hstack((x, np.array([1])))
        x = x.reshape(1, -1)
        y_pred = model(x)
        y_pred = np.where(y_pred > 0.5, 1, 0)
        y_pred = y_pred.reshape(-1)
        if y_pred == y:
            correct += 1

    accuracy = correct / dataset.shape[0]

    return accuracy
```