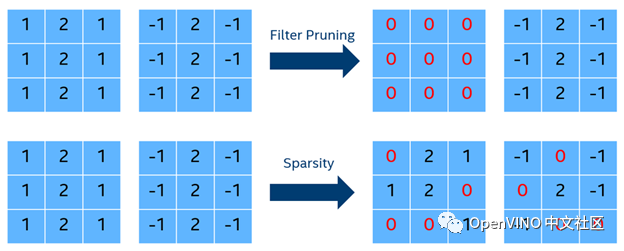
# **网络稀疏化**

### 1.稀疏（Sparsity）

稀疏是另一种神经网络的压缩方法，稀疏的核心思想是**去除一些“参与度不高”的权重值**，神经网络中往往包含了远超单纯解决检测问题所需要的权重值，稀疏压缩就会根据每一个权重对于预测结果准确性的影响，进行权重压缩，保留“贡献大”的权重，删除“贡献小”的权重值。目前主流的稀疏方法有以下两种：

· 结构化稀疏（或者叫剪枝），结构化稀疏以后的模型，会比原始模型的体积更小，因为其中一些layer间的通道或连接将被删除。

· 非结构化稀疏，非结构化稀疏后的模型，体积将和原始模型没有区别。但是权重值将被稀疏化（用更多的0来表示）。相较于结构化稀疏，通过非结构化，我们可以去除更多的敏感度不高权重值。



结构化稀疏是删除整层，非结构化稀疏是将该层部分weight置0（非结构化稀疏要依赖硬件进行稀疏矩阵的加速运算，否则无效）。

Magnitude-based，此方法将**基于阈值**参数来删除对网络结果贡献较小的权重值（设为0），该方案优势在于它的模型压缩过程将比较快速，在稀疏之前，NNCF会使用权重衰减正则化方法，进一步减小权重值，然后再根据阈值，将其置零。

幅度稀疏法采用的是一种天真方法，它基于的假设是，较低权重的贡献率较低，因此可以对其进行修剪。每次训练历时结束后，该方法都会根据当前的稀疏率计算出一个阈值，并将低于该阈值的权重归零。这里有两个选项：

Regularization-based，此方法将**引入额外的loss function**来计算稀疏后的损失，以确保模型预测准确性下降的范围。在实际操作过程中，该方法将**不会直接修改权重值**，相反NNCF为再为每个权重值增加一个**“重要性”的权重参数**，并对其进行训练优化，这样做的好处，在不改变权重值的情况下，模型准确性的下降范围更低，并可以得到有效控制，但是该方法需要更多的训练周期来对模型进行压缩。

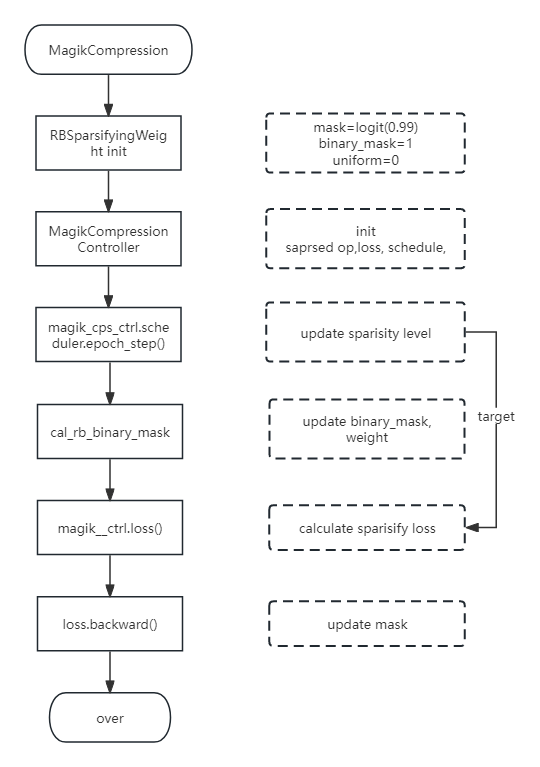
在这两种方法的选择上，如果你的训练样本量比较小，优先推荐Magnitude-based，训练样本量较大，则可以使用Regularization-based获得更好的准确性。同时建议平滑地调整sparsity ratio，以平衡准确性和压缩率。

### 2.RB-Sparsity(Regularization-Based Sparsity )

为什么叫RB-Sparsity？这是因为：Loss+=SparseLoss-->update mask-->sparse model，而SparseLoss是L0损失函数，原始Loss加上了一个L0惩罚系数。

**原理：https://github.com/openvinotoolkit/nncf/blob/develop/docs/compression\_algorithms/Sparsity.md**

#### 3.1 基本流程：



#### **3.2 关键代码：**

**ExponentialSparsityScheduler**

def epoch\_step(self, next\_epoch: Optional[int] = None) -&gt; None:  
 super().epoch\_step(next\_epoch)  
 self.\_update\_sparsity\_level()  
  
def \_update\_sparsity\_level(self) -&gt; None:  
 """  
 Calculates the current sparsity level and updates the internal  
 state of the `controller`.  
 """  
 if self.current\_epoch &gt;= self.freeze\_epoch:  
 self.\_controller.freeze() self.\_controller.set\_sparsity\_level(self.\_calculate\_sparsity\_level())  
  
def \_calculate\_sparsity\_level(self) -&gt; float:  
 current\_density = self.schedule(self.current\_epoch)  
 current\_level = 1.0 - current\_density  
 return min(current\_level, self.target\_level)

**ExponentialDecaySchedule**

class ExponentialDecaySchedule:  
 """  
 This schedule applies an exponential decay function to an epoch index,  
 considering a provided `initial\_value` value. It is computed as:  
  
 current\_value = initial\_value \* decay\_rate ^ (epoch / target\_epoch),  
  
 where `decay\_rate` is equal to target\_value / initial\_value.  
 """  
  
 def \_\_init\_\_(self, initial\_value: float, target\_value: float, target\_epoch: int):  
 """  
 Initializes a schedule with an exponential decay function.  
  
 :param initial\_value: The initial value at which the schedule begins.  
 :param target\_value: The final value at which the schedule end.  
 :param target\_epoch: Zero-based index of the epoch from which  
 the function value will be equal to the `target\_value` value.  
 """  
 self.initial\_value = initial\_value  
 self.target\_value = target\_value  
 self.target\_epoch = target\_epoch  
 self.decay\_rate = target\_value / initial\_value  
  
 def \_\_call\_\_(self, epoch: int) -&gt; float:  
 """  
 Calculates the value of the exponential decay function for a given epoch index.  
  
 :param epoch: Zero-based epoch index for which the function value should be got.  
 :return: The value of the exponential decay function for a given epoch index.  
 """  
 if self.target\_epoch == 0:  
 return self.target\_value  
 value = self.initial\_value \* np.power(self.decay\_rate, epoch / self.target\_epoch)  
# 0.99,0.86,0.74,0.64,0.56,0.49(0.01,0.)  
 return max(value, self.target\_value)

**RBSparsifyingWeight.forward()**

def forward(self, weight):  
 if is\_tracing\_state():  
 return weight.mul(self.binary\_mask)  
 tmp\_tensor = self.\_calc\_training\_binary\_mask(weight)  
 return apply\_binary\_mask\_impl(tmp\_tensor, weight)  
def \_calc\_training\_binary\_mask(self, weight):  
 u = self.uniform if self.training and not self.frozen else None  
 if not self.frozen:  
 self.binary\_mask = binary\_mask(self.\_mask)  
 return calc\_rb\_binary\_mask(self.\_mask, u, self.eps)

**functions**

def logit(x):  
 return torch.log(x / (1 - x))  
  
# straight through  
class STThreshold(torch.autograd.Function):  
 @staticmethod  
 def forward(ctx, input\_, threshold: float = 0.5):  
 output = (input\_ &gt; threshold).type(input\_.dtype)  
 return output  
  
 @staticmethod  
 def backward(ctx: Any, \*grad\_outputs: Any) -&gt; Any:  
 return grad\_outputs[0], None  
  
  
def binary\_mask(mask):  
 return STThreshold.apply(torch.sigmoid(mask))  
  
  
def calc\_rb\_binary\_mask(mask, uniform\_buffer, eps):  
 if uniform\_buffer is not None:  
 uniform\_buffer.uniform\_()  
 mask = mask + logit(uniform\_buffer.clamp(eps, 1 - eps))  
 return binary\_mask(mask)

**SparseLoss.calculate()**

def calculate(self) -&gt; torch.Tensor:  
 if self.disabled:  
 return 0  
  
 params = 0  
 loss = 0  
 sparse\_prob\_sum = 0  
 for sparse\_layer in self.\_sparse\_layers:  
 if not self.disabled and sparse\_layer.frozen:  
 raise AssertionError(  
 "Invalid state of SparseLoss and SparsifiedWeight: mask is frozen for enabled loss"  
 )  
 if not sparse\_layer.frozen:  
 sw\_loss = sparse\_layer.loss() ## binary mask  
 params = params + sw\_loss.view(-1).size(0) ## num of binary mask  
 loss = loss + sw\_loss.sum() ## binary\_mask sum  
 sparse\_prob\_sum += torch.sigmoid(sparse\_layer.mask).sum() ## mask sum  
 self.mean\_sparse\_prob = (sparse\_prob\_sum / params).item()  
 # loss / params  
 self.current\_sparsity = 1 - loss / params  
 return ((loss / params - self.target) / self.p).pow(2)

mask\_befor: [[4.59512, 4.59512, 4.59512, 4.59512, 4.59512, 4.59512],

[4.59512, 4.59512, 4.59512, 4.59512, 4.59512, 4.59512],

[4.59512, 4.59512, 4.59512, 4.59512, 4.59512, 4.59512],

[4.59512, 4.59512, 4.59512, 4.59512, 4.59512, 4.59512],

[4.59512, 4.59512, 4.59512, 4.59512, 4.59512, 4.59512],

[4.59512, 4.59512, 4.59512, 4.59512, 4.59512, 4.59512]]]], device='cuda:0', requires\_grad=True) None

mask\_after:([[[[4.59026, 4.59232, 4.59016, 4.59184, 4.59238, 4.59350],

[4.59020, 4.59101, 4.59118, 4.59225, 4.60000, 4.60000],

[4.59047, 4.59489, 4.59107, 4.60002, 4.59306, 4.59740],

[4.60006, 4.60001, 4.59125, 4.60004, 4.59052, 4.59038],

[4.59962, 4.59971, 4.59944, 4.60003, 4.59436, 4.59938],

[4.60004, 4.59993, 4.59203, 4.59802, 4.59013, 4.59016]],

grad:

[[ 6.63506e-08, -4.49599e-07, 2.46099e-07, 5.95490e-07, 2.05250e-07, -8.34185e-07],

[-1.65160e-08, 4.65233e-09, 6.20627e-08, 3.56366e-06, 2.34646e-07, -5.58701e-07],

[ 1.02088e-07, 1.08393e-07, -2.74475e-08, 8.18941e-10, -2.06285e-08, 4.74269e-07],

[ 6.60358e-06, 1.91318e-07, -2.78296e-07, 1.57566e-06, 1.63964e-07, 4.42404e-07],

[-1.17970e-06, 7.85006e-08, -3.82966e-07, -6.84191e-06, 4.51463e-08, 8.04159e-09],

[ 2.53475e-06, 4.94008e-08, -2.40743e-08, 4.05796e-07, 1.14776e-09, 3.45321e-07]]]]

#### **3.3 statistics**

def \_calculate\_sparsity\_level\_for\_model(weight\_descriptions: List[WeightDescription]) -&gt; float:  
 """  
 Calculates the sparsity level for the whole model.  
  
 :param weight\_descriptions: Descriptions for weights of the model.  
 :return: Sparsity level for the whole model.  
 """  
 total\_params = sum(w.num\_params for w in weight\_descriptions)  
 total\_num\_zero = sum(w.num\_zero for w in weight\_descriptions)  
 sparsity\_level = total\_num\_zero / total\_params  
  
 return sparsity\_level  
  
class BaseSparseModelStatisticsCollector(StatisticsCollector):  
 """  
 Base class for the sparse model statistics collector.  
 """  
  
 @abstractmethod  
 def \_collect\_weights\_descriptions(self) -&gt; List[WeightDescription]:  
 """  
 Collects descriptions of the weights of the model.  
  
 :return: Descriptions of the weights of the model.  
 """  
  
 def collect(self) -&gt; SparsifiedModelStatistics:  
 """  
 Collects statistics for the sparse model.  
  
 :return: An instance of the `SparsifiedModelStatistics` class.  
 """  
 weights\_descriptions = self.\_collect\_weights\_descriptions()  
 sparsity\_level\_for\_model = \_calculate\_sparsity\_level\_for\_model(weights\_descriptions)  
  
 total\_params = sum(w.num\_params for w in weights\_descriptions if w.is\_sparse)  
 total\_num\_zero = sum(w.num\_zero for w in weights\_descriptions if w.is\_sparse)  
 sparsity\_level\_for\_sparse\_layers = total\_num\_zero / total\_params  
  
 sparse\_layers\_summary = []  
 for w in weights\_descriptions:  
 if not w.is\_sparse:  
 continue  
  
 weight\_percentage = 100 \* (w.num\_params / total\_params)  
 sparse\_layers\_summary.append(SparsifiedLayerSummary(w.name, w.shape, w.sparsity\_level, weight\_percentage))  
  
 sparse\_model\_stats = SparsifiedModelStatistics(  
 sparsity\_level\_for\_model, sparsity\_level\_for\_sparse\_layers, sparse\_layers\_summary  
 )  
  
 return sparse\_model\_stats  
 def \_collect\_weights\_descriptions(self) -&gt; List[WeightDescription]:  
 weights\_descriptions = []  
 #processed\_modules = []  
  
 for minfo in self.\_sparse\_modules\_info:  
 sparse\_weight = minfo.operand.apply\_binary\_mask(minfo.module.model\_weight)  
  
 weights\_descriptions.append(  
 WeightDescription(  
 minfo.module\_node\_name,  
 list(sparse\_weight.shape),  
 sparse\_weight.count\_nonzero().item(),  
 is\_sparse=minfo.is\_sparse,  
 )  
 )  
  
 if minfo.module.model\_bias is not None:  
 bias = minfo.module.model\_bias  
 name = f"{minfo.module\_node\_name}/bias"  
 if self.\_supports\_sparse\_bias:  
 pass  
 # sparse\_bias = minfo.operand.apply\_binary\_mask(bias, is\_bias=True) # TODO(yujie): breaking changes  
 # weights\_descriptions.append(  
 # WeightDescription(  
 # name, list(sparse\_bias.shape), sparse\_bias.count\_nonzero().item(), is\_sparse=True  
 # )  
 # )  
 else:  
 weights\_descriptions.append(  
 WeightDescription(name, list(bias.shape), bias.count\_nonzero().item(), is\_sparse=False)  
 )  
  
 #processed\_modules.append(minfo.module)  
 return weights\_descriptions

==> Statistics of the sparsified model:

+-----------------------------------------+-------+

| Statistic's name | Value |

+=============================+

| Sparsity level of the whole model | 0.488 | ## 模型整体稀疏度

+-----------------------------------------+-------+

| Sparsity level of all sparsified layers | 0.488 | ## 需要稀疏的层的稀疏度

+-----------------------------------------+-------+

Statistics by sparsified layers:

+--------------------+-------------------+----------------+--------------------+

| Layer's name | Weight's shape | Sparsity level | Weight's percentage |

+====================+=============+============+

| model.0.conv | [48, 3, 6, 6] | 0.059 | 0.025 |

+--------------------+-------------------+----------------+--------------------+

| model.1.conv | [96, 48, 3, 3] | 0.084 | 0.199 |

+--------------------+-------------------+----------------+--------------------+

| model.2.cv1.conv | [48, 96, 1, 1] | 0.021 | 0.022 |

+--------------------+-------------------+----------------+--------------------+

Sparsity level：当前层的稀疏度

Weight's percentage ：当前层权重占模型百分比

Statistics of the RB-sparsity algorithm:

+----------------------------------------------------------------------+-------+

| Statistic's name | Value |===============================+===============+

| A target level of the sparsity for the algorithm for the current epoch | 0.510 |

+----------------------------------------------------------------------+-------+

| The probability that one weight will be zeroed | 0.480 |

+----------------------------------------------------------------------+-------+

target\_level:稀疏参数设置的目标稀疏度

probability: loss.mean\_sparse\_prob

**为什么probability和Sparsity level of the whole model不一致？**

计算模型稀疏度时params=sum(weight+bias)，而probability计算为mask sum，相当于params=sum(wights)；且分子也不一样，前者为统计非0个数，后者为mask和。

### **3.对比实验**

模型：yolov5m-ccb

稀疏参数：

"sparsity\_init": 0.01,

"sparsity\_target": 0.51,

"sparsity\_target\_epoch":5,

"sparsity\_freeze\_epoch":10

**3.1 alpha\*compression\_loss(Adam,lr=0.005,batch=64)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 8bit | 0.6\*c\_loss | c\_loss | 2\*c\_loss | 3\*c\_loss |
| sparsity\_level | 0 | 0.488 | 0.503 | 0.510 | 0 |
| mAP@.5 | 0.414 | 0.383 | 0.381 | 0.38 | 0 |
| mAP@.5:.95 | 0.241 | 0.214 | 0.212 | 0.212 | 0 |

alpha对精度无明显影响，对稀疏程度有一定影响，较大的alpha有利于向目标稀疏度靠拢，但当alpha过大的时候会出现Loss出现NAN值（**alpha多大会导致这种情况？？？模型变了alpha的最大值也会发生变化吗？？？**），从而导致无法进行稀疏

#### **3.2 ema/scaler(Adam,lr=0.001,batch=10)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 8bit | - | ema | scaler | ema+scaler |
| sparsity\_level | 0 | 0.51 | 0.51 | 0.51 | 0.509 |
| map@.5 | 0.414 | 0.403 | 0.403 | 0.385 | 0.386 |
| map@.5:.95 | 0.241 | 0.231 | 0.231 | 0.216 | 0.214 |

ema,scaler对稀疏程度无明显影响，ema加快模型收敛速度，不影响精度，scaler会降低精度。

#### **3.3 batch，lr**

batch与lr呈正相关

|  |  |  |  |
| --- | --- | --- | --- |
|  | batch=10  lr=0.001 | batch=64  lr=0.001 | batch=64  lr=0.005 |
| sparsity\_level | 0.51 | 0 | 0.503 |
| map@.5 | 0.403 | - | 0.381 |
| map@.5:.95 | 0.231 | - | 0.212 |

此次对比实验主要探究稀疏化训练训练超参数（batch,lr，alpha）及AMP，EMA对稀疏程度和精度的影响，batch和lr呈正相关变化，对模型稀疏度和精度有影响,对精度影响很大；alpha对模型稀疏度呈正相关，对精度无影响；AMP，EMA对模型稀疏度无影响，AMP会降低模型精度，EMA会提升模型收敛速度。

### 补充：

1. 正则化

* L0正则化的值是模型参数中**非零参数**的个数。
* L1正则化表示各个参数**绝对值**之和。
* L2正则化标识各个参数的**平方和的开方值**。

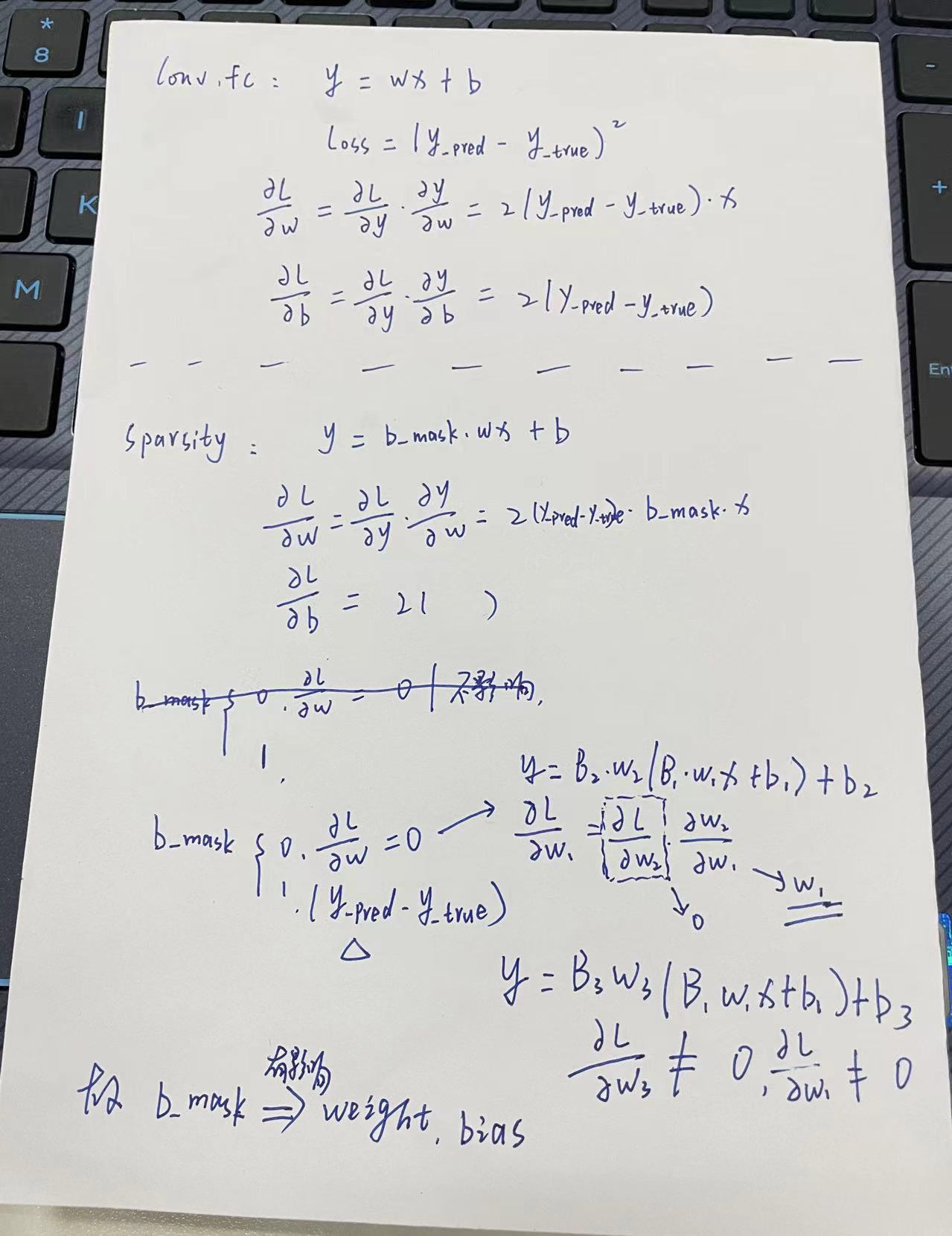
1. 大数定律

这是因为根据大数定律，当我们从概率分布 Ps 中抽取足够多的样本时，这些样本的平均值会逐渐收敛到随机变量s的数学期望。换句话说，随着样本数量的增加，样本的平均值将越来越接近真实的数学期望。

这种现象是由大数定律保证的，它表明在一定条件下，随机变量序列的算术平均值会收敛到其数学期望。因此，通过从概率分布中采样一组值，并计算这些值的平均值，我们可以得到对数学期望的估计。

需要注意的是，这种估计方法依赖于样本数量的大小以及样本的独立性和同分布性。通常情况下，当样本数量足够大时，这种估计方法会提供一个较为准确的数学期望估计。

1. binary\_mask对weight,bias的梯度的影响



1. 附件

[请至钉钉文档查看附件《results-ema.csv》](https://alidocs.dingtalk.com/i/nodes/kDnRL6jAJMDb1Pw3Cy1elx3xVyMoPYe1?cid=1474006293%3A1474006293&corpId=ding9b8f6aba810b351035c2f4657eb6378f&doc_type=wiki_doc&iframeQuery=anchorId%253DX02ltgmr83vhuxjr8h2gx&utm_medium=im_card&utm_scene=person_space&utm_source=im)

[请至钉钉文档查看附件《results.csv》](https://alidocs.dingtalk.com/i/nodes/kDnRL6jAJMDb1Pw3Cy1elx3xVyMoPYe1?cid=1474006293%3A1474006293&corpId=ding9b8f6aba810b351035c2f4657eb6378f&doc_type=wiki_doc&iframeQuery=anchorId%253DX02ltgmreiimy6b6pvuxm&utm_medium=im_card&utm_scene=person_space&utm_source=im)