深度学习调参经验分享

先举例，什么是调参?

在深度学习中，调参指的是调整算法模型中的参数，以使模型达到更好的性能。这些参数通常分为两类：一类是模型参数，一类是超参数。良好的调参策略可以极大提高模型的准确率和泛化能力。调参通常是基于经验、试错以及使用一些系统化的方法进行，如网格搜索、随机搜索、贝叶斯优化等。

1. 模型参数：指的是模型内部的可学习参数，这些参数是在学习过程中通过数据自动学习调整的，如神经网络中的权重和偏置。
2. 超参数：指的是模型训练前需要设置的参数，这些参数通常需要手动设定，它们直接影响到模型的训练过程和最终性能。常见的超参数包括：
    - 学习率：决定模型参数更新的速度。
    - 迭代次数：即训练过程中数据将被重复使用的次数。
    - 批量大小（Batch size）：每进行一次参数更新时使用的数据样本数量。
    - 网络结构相关参数：如层数、每层的单元数或过滤器数。
    - 正则化参数：比如L1、L2正则化系数，用来防止过拟合。
    - 激活函数：如ReLU、sigmoid、tanh等。
    - 优化器：比如SGD、Adam、RMSprop等。

代码示例

```
def parse_args():
    import argparse
    parser = argparse.ArgumentParser(description="pytorch unet training")

    parser.add_argument("--data-path", default="./", help="DRIVE root")
    # exclude background
    parser.add_argument("--num-classes", default=1, type=int)
    parser.add_argument("--device", default="cuda", help="training device")
    parser.add_argument("-b", "--batch-size", default=1, type=int)
    parser.add_argument("--epochs", default=100, type=int, metavar="N",
                        help="number of total epochs to train")

    parser.add_argument('--lr', default=0.0001, type=float, help='initial learning rate')
    parser.add_argument('--momentum', default=0.9, type=float, metavar='M',
                        help='momentum')
    parser.add_argument('--wd', '--weight-decay', default=1e-4, type=float,
                        metavar='W', help='weight decay (default: 1e-4)',
                        dest='weight_decay')
    parser.add_argument('--print-freq', default=1, type=int, help='print frequency')
    parser.add_argument('--resume', default='', help='resume from checkpoint')
    parser.add_argument('--start-epoch', default=0, type=int, metavar='N',
                        help='start epoch')
    parser.add_argument('--save-best', default=True, type=bool, help='only save best dice weights')
    # Mixed precision training parameters
    parser.add_argument("--amp", default=False, type=bool,
```

```
                        help="Use torch.cuda.amp for mixed precision training")

    args = parser.parse_args()

    return args
```

提供的代码是一个Python函数 `parse_args()`，它使用 `argparse` 模块创建命令行接口。这个函数定义了训练一个使用PyTorch框架的U-Net模型所需要的输入参数。这个函数在执行时将解析命令行提供的参数，并返回一个命名空间（Namespace），其中包含了所有的参数值。

这些参数包括：

- `--data-path`：数据集的根目录，默认为当前目录 `"./"`。
- `--num-classes`：分类任务中的类别数，默认为1。
- `--device`：训练所使用的计算设备，默认为 `"cuda"`。
- `-b` 或 `--batch-size`：批次大小，默认为1。
- `--epochs`：训练的总周期数，默认为100。
- `--lr`：初始学习率，默认为0.0001。
- `--momentum`：动量参数，默认为0.9。
- `--wd` 或 `--weight-decay`：权重衰减率，默认为1e-4。
- `--print-freq`：打印训练进度的频率，默认为每1个周期。
- `--resume`：从检查点继续训练的路径，默认为空，即不从检查点恢复。
- `--start-epoch`：训练开始的周期，默认为0（从头开始训练）。
- `--save-best`：是否只保存最好的模型，默认为True。
- `--amp`：是否使用混合精度训练，默认为False。

要使用这个函数，你可以在 Python 脚本的开始处加入这段代码，然后在脚本的后续部分通过 `args` 对象来访问用户提供的命令行参数。基于这些参数，你可以设置你的训练配置。

例如，如果你的训练脚本的其余部分需要访问学习率，你可以通过 `args.lr` 获取这个值。如果你需要访问设备类型，可以使用 `args.device`。

其中，--lr 参数表示神经网络训练过程中的学习率。 学习率是神经网络优化中最关键的超参数之一，极大地影响训练随时间的演变。 以下是学习率对训练的影响：

- 收敛速度：较高的学习率可以加速训练过程的收敛，从而有可能使模型学习得更快。 但如果太高，可能会导致模型超出损失函数的最小值甚至发散，导致训练失败。
- 训练稳定性：学习率过高会导致训练不稳定，损失和准确率指标大幅波动，甚至不降反升。 较低的学习率通常会导致收敛速度较慢但更稳定。
- 解决方案的精度：较低的学习率可以让优化算法在搜索损失函数的最小值时采取更小、更精确的步骤。 这可以带来更准确的解决方案，但也可能意味着训练需要更长的时间才能收敛。
- 逃离局部极小值：有时，较高的学习率可以帮助模型摆脱局部极小值，而较低的学习率可能无法实现这一点，从而导致优化器陷入困境。
- 学习率计划的效果：随着时间的推移降低学习率的学习率计划可以带来更好的训练性能，而不是保持学习率恒定。 在训练初期，使用较大的学习率来快速进步，后期则降低学习率来微调解决方案。
- --lr 参数用于设置开始训练过程时的初始学习率。 仔细调整学习率对于神经网络的成功训练至关重要，通常可以通过实验或使用自适应学习率方法来发现，例如学习率计划或 Adam、RMSprop 等优化算法。

关于学习率

选择合适的学习率是优化深度学习模型性能的一个重要方面。学习率决定了在梯度下降或其他优化过程中权重更新的幅度。如果学习率过高，模型可能会在损失函数的最小值点附近震荡，无法收敛；如果学习率过低，训练过程会非常缓慢，并且有可能陷入局部最小值。以下是选择合适学习率的一些策略：

1. **尝试多个学习率**：开始时可以选取一系列不同的学习率进行尝试，观察损失函数随迭代次数的变化，找到使损失函数下降最稳定的那个学习率值。
2. **学习率衰减**：初始时使用较大学习率以便快速前进，随着训练进程逐渐降低学习率。这可以通过设置一个学习率衰减的计划来实现，例如指数衰减、阶梯式衰减等。
3. **使用自适应学习率**：某些优化算法如Adam、Adagrad、RMSprop等，可以自动调整学习率，不需要手动设置固定的学习率。
4. **学习率寻优算法**：例如学习率寻优技术可以通过一定范围内逐渐增加学习率，并记录损失值的变化来找到合适的学习率区间。
5. **交叉验证**：使用交叉验证在验证集上测试不同的学习率，选择使模型性能最优的学习率。
6. **经验法则**：可以从较小的学习率开始，如0.0001, 0.001等，逐渐提高至0.01、0.1等，直到找到性能开始下降的点为止。

最终，选择合适的学习率往往需要多次实验和调整，同时也取决于选择的优化器、模型结构和具体任务。在实践中，往往结合多种策略来获得最佳性能。

三种自适应学习率优化算法

Adam、Adagrad和RMSprop是三种常用的自适应学习率优化算法，它们在更新模型的参数时根据之前的梯度自动调整学习率大小，这样做可以提高模型的收敛速度和性能。下面是它们各自的详细介绍：

1. **Adagrad（Adaptive Gradient Algorithm）**：
   - Adagrad是一种参数优化方法，它会为每个参数维护一个参数专属的学习率，以此适应该参数的重要性。
   - 它能够通过累积过去所有梯度的平方和来调整每个参数的学习率，这意味着经常更新的参数的学习率会变得更小，罕见更新的参数的学习率会相对较大。
   - 这对稀疏数据特别有用，但在长期训练中可能会遇到学习率降低过早、过快的问题，从而使得训练过程提前结束。
2. **RMSprop（Root Mean Square Propagation）**：
   - RMSprop算法是为了解决Adagrad学习率单调递减的问题而提出的。
   - 它利用了梯度的均方根（root mean square）来调整各个参数的学习率，不是简单地累加之前所有梯度的平方，而是引入了一个衰减系数，这样历史较远的梯度权重会慢慢减小，防止学习率持续降低。
   - RMSprop通常认为是一种很好的实践选择，特别是对循环神经网络等不稳定或非平稳目标函数的训练。
3. **Adam（Adaptive Moment Estimation）**：
   - Adam是目前最受欢迎的深度学习优化算法之一，它融合了Momentum和RMSprop两种优化方法的思想。
   - Adam不仅保存了过去梯度的均方值（类似于RMSprop），还保存了过去梯度的指数衰减平均值，这相当于给梯度加了动量（Momentum）。
   - Adam添加了偏差修正机制，用来在训练初期对梯度的估计进行校准，这有助于防止初始估计值过低。
   - Adam被认为既实用又高效，适用于许多不同的深度学习应用。

这些算法都有自己独特的特点和适应场景，而在具体应用中，往往需要根据实际情况和任务的需求来选择最合适的优化算法。

关于过拟合

过拟合是指模型在训练数据上学习得太好，以至于它开始捕捉数据中的噪声和误差，而不仅仅是底层的数据分布或趋势。这意味着它对训练数据的特定特点过分敏感，包括那些并不具有普遍性的特点。结果，模型可能在训练集上表现得非常好，但是一旦遇到新的、未见过的数据，它的预测能力却下降了，因为它没有学会从数据中抽象出一般的模式。

换句话说，过拟合是机器学习模型的一种现象，模型变得对有限的训练数据过度专化，并且无法有效地泛化到新的数据集上。这通常发生在模型过于复杂时，它有着足够的参数去"记住"训练数据集中的每个样本，而不是去"学习"能够概括所有数据的规律。过拟合是机器学习模型优化过程中的一个主要挑战，防止过拟合是模型开发者在训练模型时经常需要考虑的问题。

如何判断模型是否过拟合?

判断模型是否过拟合通常涉及以下几个步骤和指标：

1. **训练集与验证集的性能差异**：
   - 如果模型在训练数据上的表现很好（比如，准确率很高、误差很低），但在未见过的验证数据上表现不佳，这是过拟合的一个显著迹象。
2. **学习曲线观察**：
   - 绘制训练和验证误差随训练时间（或训练周期）变化的曲线。在过拟合的情况下，随着训练的深入，训练误差会持续下降，而验证误差在达到某一点后会开始上升。
3. **检查模型的复杂度**：
   - 如果你使用一个高度复杂的模型（例如，参数数量很多）去拟合一个相对简单或小规模的数据集，模型可能会学习到数据中的噪声，而不是其背后的真实分布。
4. **交叉验证**：
   - 通过交叉验证的方式，如果模型在不同的验证数据子集上都呈现出类似的性能下降，这同样表示可能出现了过拟合。
5. **统计测试**：
   - 使用如假设检验等统计学方法来评估模型在训练集和验证集上的性能，从而确定模型的性能是否显著不同。
6. **性能指标**：
   - 查看模型在训练集和验证集上的具体性能指标，例如精确度、召回率、F1分数等，发现差异可能表明过拟合。
7. **可视化技术**：
   - 对于一些模型，如决策树或深度神经网络，可以直观地查看模型结构或特征权重等。如果发现模型对训练数据的特定样本或特征过度敏感，则可能是过拟合。
8. **外部测试集**：
   - 如果可能，提供一个外部测试集，这可以帮助进一步验证模型的泛化能力。

当你怀疑存在过拟合时，可以通过简化模型，添加正则化，增加数据量，或者使用数据增强等技术来尝试解决这个问题。避免过拟合的关键在于保持模型的泛化能力，使其能够良好地推广到新的、未见过的数据上。

遇到过拟合问题怎么办?

遇到过拟合问题时，可以考虑以下几种策略来调整参数：

1. **增加数据量**: 更多的训练数据可以提高模型的泛化能力。如果无法获取新数据，可以使用数据增强技术，如旋转、缩放、翻转或裁剪图像。
2. **降低模型复杂度**: 减少网络层或每层的神经元数量可以降低模型的学习能力，从而减少过拟合。

3. **正则化**: 应用L1或L2正则化能够惩罚大的权重值，使得权重分布更加平滑，这有助于提升模型的泛化能力。
    - 在示例代码中，可以调整 `--weight-decay` 参数来增加L2正则化的强度。
4. **Dropout**: 在网络中引入Dropout层，通过随机断开网络连接，可以减少神经元之间复杂的共适应关系。
5. **Early Stopping**: 当验证集上的性能不再提升时停止训练，以避免模型在训练集上过度学习。
6. **调整学习率**: 如果学习率过高，模型可能会过拟合。尝试降低 `--lr` 参数的值以减慢学习进度。
7. **学习率衰减**: 随着训练的进行逐渐减少学习率，可以在保持快速进度的同时减少过拟合风险。
8. **使用复杂的学习率调度**: 如使用学习率预热（warm-up）和循环学习率等策略。
9. **集成学习**：使用模型集成技术，通过组合多个模型减少过拟合和提高泛化能力。这包括bagging和boosting等技术。
10. **交叉验证**：使用交叉验证来更好地监测模型的泛化性能，并基于此进行参数调整。

对于深度学习模型，通常需要尝试几种不同的组合并根据验证集上的性能来选择最佳的参数配置。

这些策略并不是相互独立的，而是需要根据具体情况结合使用。记住要持续监控训练集和验证集的性能差异，这有助于你判断是否采取了正确的措施来减轻过拟合。

有哪些常用的正则化方法可以应用于减少过拟合问题?

在机器学习和深度学习中，减少过拟合的常用正则化方法包括：

1. **权重衰减（L2正则化）**：通过在损失函数中添加一个与权重大小成比例的惩罚项，减少模型复杂度。这通常导致权重向量较小，从而有助于防止模型在训练数据中捕捉噪声。
2. **L1 正则化**: 类似于L2正则化，但使用的是权重的绝对值。它倾向于产生稀疏的权重矩阵，从而能够进行特征选择，有些权重会变为零。
3. **Dropout**: 在训练过程中随机使一些神经元的输出为零，这防止了模型依赖于任何一个特征，并迫使网络学习更加鲁棒的特征表示。
4. **Early Stopping**: 监控验证集的性能，当看到性能开始下降时停止训练，以防止模型过度拟合训练数据。
5. **数据增强**: 对训练数据进行修改，如旋转、平移、缩放、添加噪声等，可以有效增加样本的多样性，促使模型学习到更加泛化的特征。
6. **批量归一化（Batch Normalization）**：通过对每个小批量数据的输入进行归一化处理，可以加快训练速度并减少过拟合。
7. **噪声注入****: 在输入数据或者权重上添加噪声，可以提高模型的稳健性，减少过拟合。**
8. **Label Smoothing**: 对训练标签进行平滑处理，减少模型对于硬标签的过度自信。 当实施正则化方法时，需要小心选择参数（如dropout比例、L1/L2正则化强度等），因为这将影响模型的最终性能。通常，这些参数需要通过交叉验证等技术进行调整以找到最佳值。

如何选择合适的正则化方法和参数来应对过拟合问题?

选择合适的正则化方法和参数以应对过拟合问题是一个需要细心调试的过程，这里是一些步骤和方法，可以帮助你找到最佳配置：

1. **诊断过拟合程度**：首先，要确切地了解模型是否真的过拟合，以及过拟合的严重程度。这通常通过比较训练和验证数据集上的性能来实现。
2. **了解数据**：观察和分析你的数据，因为正则化的选择很大程度上依赖于数据的特性。例如，如果你的数据很嘈杂或者特征维度很高，稀疏性可能是一个有益的属性，这时可以选择L1正则化。
3. **简化模型**：在增加正则化之前，确保模型结构不是过于复杂。如果网络太深或者参数太多，试着简化模型结构。

4. **尝试标准正则化方法**：通常首先考虑加入L2权重衰减，它是最常用且经典的正则化方法，并且它对模型的准确度影响相对较小。
5. **实验不同的正则化策略**:
   - 实施Dropout，并调整不同层的dropout比率。
   - 试验不同的数据增强技术，比如变形、裁剪、颜色变换等。
   - 应用Batch Normalization，观察模型的学习速率和泛化效果是否有所改进。
6. **交叉验证**：使用交叉验证来评估不同正则化策略的效果。这可以帮你比较不同设置下模型的性能，并确定哪些策略对减少过拟合最有效。
7. **调整参数**：开始时，选择一个中等大小的正则化参数，然后通过验证集上的性能来调整大小。如果过拟合仍然存在，逐步增加正则化强度。如果模型表现出欠拟合，减少正则化强度。
8. **学习曲线**：观察训练和验证误差随时间的变化，找到正则化参数导致的性能改变的最佳点。
9. **实施网格搜索或随机搜索**：自动化搜索最优的正则化参数，这可以是一个值得投资的时间长期回报的过程。
10. **追踪模型复杂度**：在添加正则化的同时，要注意模型的复杂度是否保持在合理的范围内，防止模型由于过度简化而无法捕捉到数据的关键模式，即欠拟合问题。

最后，记得要记录每次实验的设置和结果，这样可以帮助你了解哪些方法有效，哪些不有效，并在未来的项目中迅速定位问题。正则化参数的选择通常需要根据经验不断调整和尝试，结合模型性能的实际观察来做出决定。

Dropout层的作用是什么？如何选择合适的dropout率？

Dropout层的主要作用是防止神经网络发生过拟合。它通过在训练过程中的每次迭代中随机“丢弃”（即临时移除）网络中的一些神经元（以及它们的连接），来减少网络复杂度和神经元之间的共适应性。这意味着网络的每个激活项（neuron）都有一定的概率不会对下一层的激活产生影响。因此，这迫使网络学习更加健壮的特征，这些特征不依赖于具体的一些神经元，增强了模型对新数据的泛化能力。

选择合适的Dropout率（即丢弃概率）通常需要下面的考虑:

1. **问题复杂度**:
   - 对于比较简单的问题，可能不需要很高的Dropout率，因为模型本身不太可能过拟合数据。
2. **神经网络的大小**:
   - 如果网络规模很大，包含许多参数，那么更高的Dropout率可能有助于防止过拟合。
3. **数据量**:
   - 如果训练数据量不足，模型更容易过拟合，这时使用Dropout可以减少这种风险。大数据量情况下，可能需要较低的Dropout率，甚至不需要Dropout。
4. **实验和经验**:
   - 不同的任务和数据集可能需要不同的Dropout率，通常需要通过实验进行参数调优来找到这个值。一个常见的起始点是0.5，但这个值可能根据上述因素进行调整。
5. **模型训练阶段**:
   - 在训练初期，可能会使用较高的Dropout率来保证模型不会过度拟合。而在微调或后期训练时，可能会减少Dropout率来允许模型更细致地学习数据。
6. **模型的层**:
   - 在不同的层上可能使用不同的Dropout率。例如，在输入层可能使用较低的Dropout率，在靠近输出层的地方使用较高的Dropout率，因为深层网络更可能捕捉到复杂的模式从而导致过拟合。

综上所述，选择Dropout率没有硬性规则，通常是基于上述因素进行交叉验证和实验来确定。而调整Dropout率是一个迭代过程，需要根据模型在验证集上的性能来不断调整和优化。

一个具体的简单的例子

让我们以一个简单的神经网络模型为例，假设我们用PyTorch框架来对MNIST手写数字数据集进行分类。我们会关注如何调整网络的层数和大小来防止过拟合。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score


# 定义变换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# 下载并加载训练数据
trainset = datasets.MNIST('', download=True, train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)


# 构建神经网络
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # flatten image input
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # 添加 Dropout 层来防止过拟合
        x = F.dropout(x, p=0.5)
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.fc5(x)  # 不需要激活，因为这是输出层
        return F.log_softmax(x, dim=1)


# 实例化网络
model = Net()

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.SGD(model.parameters(), lr=0.03, momentum=0.9)

# 训练模型
for epoch in range(10):  # 这里只循环10个epoch来节约时间
    running_loss = 0.0
    all_predictions = []
    all_labels = []

    for images, labels in trainloader:
        optimizer.zero_grad()

        # 前向传播
        outputs = model(images)
        loss = criterion(outputs, labels)

        # 反向传播和优化
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # 记录预测结果和标签
        _, predictions = torch.max(outputs, 1)
        all_predictions.extend(predictions.tolist())
        all_labels.extend(labels.tolist())

    # 计算评估指标
    accuracy = accuracy_score(all_labels, all_predictions)
    precision = precision_score(all_labels, all_predictions, average='macro')
    recall = recall_score(all_labels, all_predictions, average='macro')
    f1 = f1_score(all_labels, all_predictions, average='macro')

    print(
        f'Epoch {epoch + 1}, Loss: {running_loss / len(trainloader)}, Accuracy:
{accuracy}, Precision: {precision}, Recall: {recall}, F1 Score: {f1}')

print('Finished Training')
```

这段代码是一个简单的用PyTorch构建的神经网络，用于对MNIST数据集进行手写数字分类。我将逐行解释代码的功能：

1. `import`语句导入了PyTorch库中的必要模块，包括神经网络模块、优化器、数据集和数据变换。还导入了评估指标的计算函数`accuracy_score`、`precision_score`、`recall_score`和`f1_score`，这些函数来自于`sklearn.metrics`模块。
2. `transform`定义了数据变换的组合。这里使用了两个变换：`ToTensor()`将图像转换为张量，并将像素值归一化到[-1, 1]的范围内。
3. `trainset`和`trainloader`分别下载并加载了MNIST数据集的训练集。`trainloader`将数据集分成大小为64的小批量，并打乱数据，以便随机梯度下降。
4. `Net`类定义了神经网络模型。该模型包含5个全连接层，每个层之间使用ReLU激活函数。在第三个全连接层和第四个全连接层之间使用了Dropout层，以减少过拟合。
5. `forward`方法定义了前向传播过程，即定义了模型的结构。输入图像首先被展平，然后通过一系列的全连接层和激活函数进行处理，最后通过log_softmax函数进行分类输出。
6. `model`实例化了定义好的神经网络模型。
7. `criterion`定义了损失函数，这里使用的是交叉熵损失函数。

8. `optimizer`定义了优化器，这里使用的是随机梯度下降（SGD）优化器，学习率为0.03，动量为0.9。
9. 训练过程使用了嵌套的循环结构，外层循环是epoch的数量，内层循环是每个epoch中对数据集的迭代。在每个epoch中，通过模型的前向传播计算损失，然后通过反向传播和优化器来更新模型的参数。
10. 在每个epoch的结束处打印出平均损失。然后，在每个epoch结束时，我们将所有预测值和标签收集到列表中，并使用这些列表计算评估指标。最后，将评估指标的值打印出来以监视模型的性能。
11. 最后输出"Finished Training"表示训练过程结束。

这段代码实现了一个简单的MNIST手写数字分类器的训练过程。

在上面的代码中，我们构建了一个相对复杂的神经网络，其中包括5个全连接层。这种网络在MNIST数据集上容易出现过拟合。为了降低这种风险，我们在第三层和第四层之间增加了一个Dropout层（`p=0.5`表示网络有一半的神经元将被随机丢弃），以随机地减少神经元之间的共适应性，从而增强模型的泛化能力。

然而，如果模型依旧过拟合，你可以尝试以下方法：

- 减少层数或神经元数量来降低模型复杂度。
- 增加Dropout率，例如将`p`从`0.5`增加到`0.6`或更高。
- 增加数据增强方法。
- 采用更复杂的正则化方法，如L1或L2权重衰减。
- 调整学习率和其他优化器参数。

每次调整参数后，观察在验证集上的性能来评估改变是否有效。这个过程可能需要多次迭代，因为每个环节的调整都可能影响模型的最终表现。

以下是对代码进行调整以应用上述提到的防止过拟合的方法的示例：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# 定义变换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# 下载并加载训练数据和验证数据
trainset = datasets.MNIST('', download=True, train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
validset = datasets.MNIST('', download=True, train=False, transform=transform)
validloader = torch.utils.data.DataLoader(validset, batch_size=64,
shuffle=False)

# 构建神经网络
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 512)
```

```
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(p=0.5)  # 添加Dropout层

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # flatten image input
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.dropout(x)  # 使用Dropout层
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.fc5(x)  # 不需要激活，因为这是输出层
        return F.log_softmax(x, dim=1)

# 实例化网络
model = Net()

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.03, momentum=0.9,
weight_decay=1e-5)  # 添加L2正则化项

# 训练模型
for epoch in range(10):
    running_loss = 0.0
    all_predictions = []
    all_labels = []

    # 训练阶段
    model.train()
    for images, labels in trainloader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predictions = torch.max(outputs, 1)
        all_predictions.extend(predictions.tolist())
        all_labels.extend(labels.tolist())

    # 计算训练集上的评估指标
    train_accuracy = accuracy_score(all_labels, all_predictions)
    train_precision = precision_score(all_labels, all_predictions,
average='macro')
    train_recall = recall_score(all_labels, all_predictions, average='macro')
    train_f1 = f1_score(all_labels, all_predictions, average='macro')

    # 在验证集上进行评估
    model.eval()
    all_predictions_valid = []
    all_labels_valid = []
    with torch.no_grad():
```

```
        for images, labels in validloader:
            outputs = model(images)
            _, predictions = torch.max(outputs, 1)
            all_predictions_valid.extend(predictions.tolist())
            all_labels_valid.extend(labels.tolist())

    # 计算验证集上的评估指标
    valid_accuracy = accuracy_score(all_labels_valid, all_predictions_valid)
    valid_precision = precision_score(all_labels_valid, all_predictions_valid,
average='macro')
    valid_recall = recall_score(all_labels_valid, all_predictions_valid,
average='macro')
    valid_f1 = f1_score(all_labels_valid, all_predictions_valid,
average='macro')

    print(f'Epoch {epoch + 1}, Loss: {running_loss/len(trainloader)}, Train
Accuracy: {train_accuracy}, Train Precision: {train_precision}, Train Recall:
{train_recall}, Train F1 Score: {train_f1}, Valid Accuracy: {valid_accuracy},
Valid Precision: {valid_precision}, Valid Recall: {valid_recall}, Valid F1
Score: {valid_f1}')

print('Finished Training')
```

在这个示例中，我们对代码进行了以下调整：

1. 添加了L2正则化项（weight_decay=1e-5），将它传递给优化器，以减少模型的复杂性。
2. 在神经网络模型中添加了一个Dropout层，以减少过拟合。
3. 每个epoch结束后，在训练集和验证集上计算了准确率、精确率、召回率和F1分数，以监视模型在训练和验证集上的性能。