HW0(F24版)

2024年9月3日 15:50

由于找不到S24版数据集，所以HW0用的F24版的。

0 作业概述

该作业的主要目的是：

- **熟悉PyTorch的基本使用**：通过阅读教程和实践任务，学生能够掌握PyTorch的基本操作，包括模型构建、数据处理、训练和评估等。
- **学习Weights & Biases工具的使用**：通过在代码中集成wandb，学生可以学会如何追踪和可视化模型的训练过程和性能。
- **探索不同的模型和优化器**：通过修改初始模型结构和使用不同的优化器，学生可以理解不同设计选择对模型性能的影响。

步骤如下：

1. 阅读PyTorch教程。（中文版教程：https://pytorch.ac.cn/tutorials/beginner/basics/intro.html）
2. 阅读Weights and Biases（wandb）教程。
3. 查看HW0的初始代码。你会发现它与PyTorch教程中描述的代码非常相似。
4. 修改初始代码，使其包含Weights & Biases日志记录功能。
5. 运行指定的实验，并通过wandb界面以表格或图表的形式报告你的结果。
6. 进一步修改代码，以支持不同的模型（你可以自行选择模型！）。
7. 让你的代码选择不同的优化器（你可以自行选择优化器！）。
8. 运行其他实验，以更好地理解PyTorch。

评分标准：

Question	Points
Background Reading	3
Image Classification	43
Text Classification	24
Code Upload	0
Collaboration Questions	2
Total:	72

1 阅读（3分）

阅读pytorch教程：

- 阅读章节：Learn the Basics ‖ Quickstart ‖ Tensors ‖ Datasets & DataLoaders ‖ Transforms ‖ Build Model ‖ Autograd ‖ Optimization ‖ Save & Load Model（学习基础知识 || 快速入门 || 张量 || 数据集和数据加载器 || 变换 || 构建模型 || Autograd || 优化 || 保存和加载模型）
- 中文版pytorch教程网址：https://pytorch.ac.cn/tutorials/beginner/basics/quickstart_tutorial.html

阅读 Weights & Biases 教程（Quickstart部分）： https://docs.wandb.ai/quickstart

2 图像分类（43分）

最简单的图像3分类（鹦鹉、独角鲸、墨西哥钝口螈），分类的图像全部源自文生图，样貌如下：

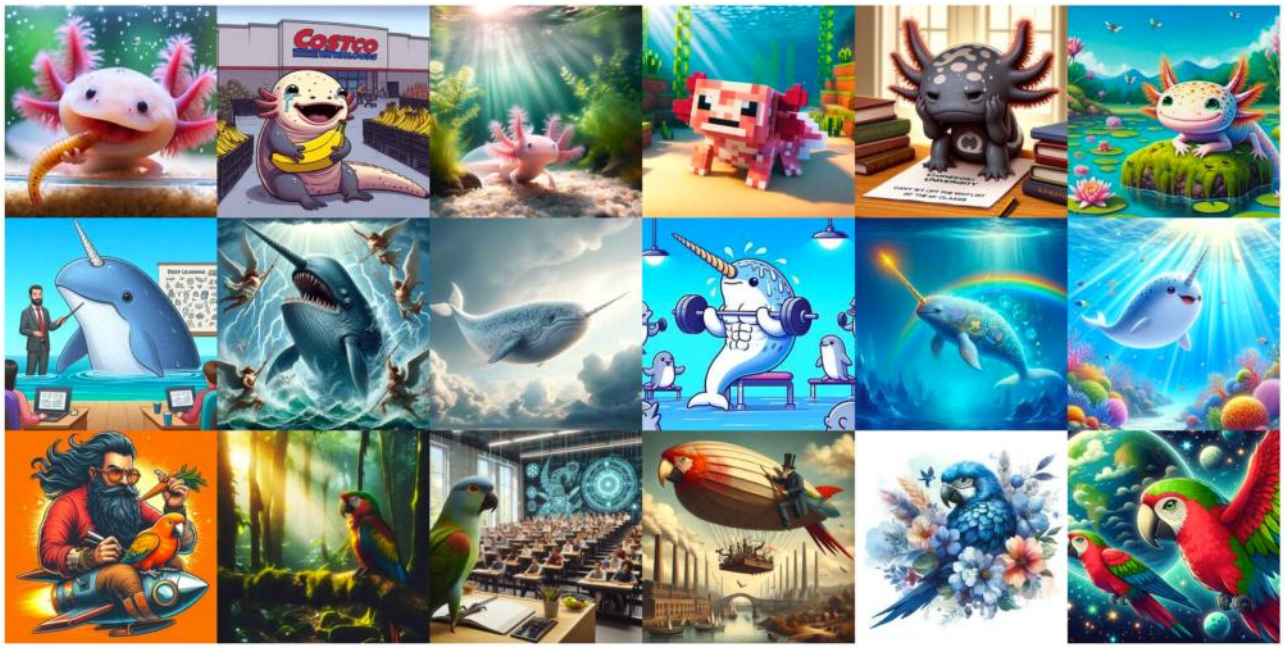


Figure 1: Examples images from Parrot/Narwhal/Axolotl dataset.

作业给了个起始脚本img_classifier.py（该文件很多残缺，甚者有些地方有问题，比如平均loss的计算方式，需要根据后续问题自己在相应位置新增代码段或修改原文件），第一步需要在里面添加wandb代码，要求如下：

- 使用 wandb.init() 设置项目名称并存储超参数。
- 使用 wandb.log() 记录每个批次的损失和已计算梯度的样本数。
- 在每个 epoch 结束时，使用一次 wandb.log() 调用来跟踪训练精度、训练损失、测试精度、测试损失以及当前的 epoch 编号。

完成上述代码新增后，完成并回答如下9个问题：（感觉需要先通读下面9个问题，然后一次性先把代码全部写好，不然要跑很多次代码）

2.1 (3 分)【完成】

使用默认的超参数运行图像分类器。使用wandb绘制每个batch的训练损失与迄今为止处理的batch总数的关系图。损失应为平均损失，即在每个batch内取平均值。

横轴约300多个batch（batchsize=8）。图中突然出现的波峰，估计是某个batch的数据特别异常，所以每个epoch轮到验证它时，loss就突然很高。

2.2 (3 分)【完成】

报告最终的训练/测试准确率和最终的训练/测试损失。（你不需要使用wandb来创建此表格。）

备注：以下的训练集精度和损失是在每个 epoch 完成后，将整个训练数据重新输入模型进行一次完整验证得出的。这与通常在训练过程中边训练边累计计算的精度和损失数据有所不同。前者的精度通常会略高一些，因为在整个训练完成后，模型已经经过了更多的优化，因此验证时的表现会更好。

Train accuracy = 99.1%, Train avg loss = 0.125223

Val accuracy = 68.1%, Val avg loss = 0.659595

Test accuracy = 58.3%, Test avg loss = 0.902912

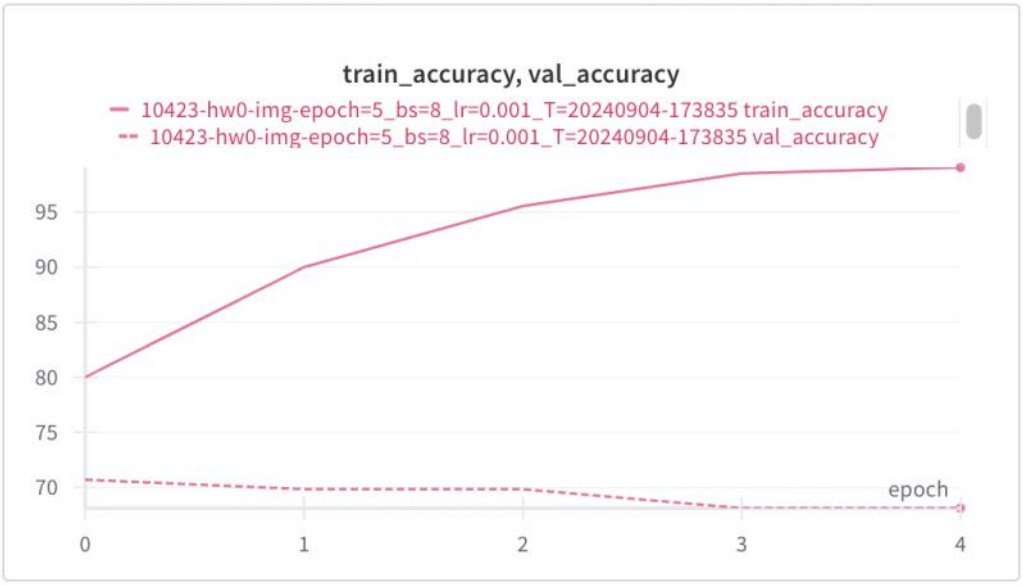
2.3 (4 分)【完成】

在同一图表上绘制训练损失和验证损失，横轴为epoch数。训练损失应在整个训练数据集上取平均值，验证损失同样在验证集上取平均值。

这种图表数据融合问题，可以训练完后，在wandb项目页面通过【add panel】来将不同数据融合在一张表里。结果如下：

2.4 (4 分)【完成】

在同一图表上绘制训练准确率和验证准确率，横轴为epoch数。



2.5【完成】

令人惊讶的是，这个简单的模型似乎能够区分（至少一部分）鹦鹉、独角鲸和墨西哥钝口螈的图像。或许，这种成功的原因是这些动物的颜色往往有很大的差异。将 `transform_img = T.Compose([...])` 中的转换顺序更改为使用 `torchvision.transforms.Grayscale` 将图像转换为单通道灰度图像。确保调整模型以适应新的输入尺寸！(回答完此问题后，恢复使用彩色图像。)

2.5.a (3 分)

使用彩色图像与灰度图像的测试准确率是多少?

数据集	图像类型	准确率
测试集	彩色	58.3%

测试集	灰度	54.8%

更详细的结果：

Train accuracy = 86.7%, Train avg loss = 0.490634

Val accuracy = 55.2%, Val avg loss = 0.900364

Test accuracy = 54.8%, Test avg loss = 0.983261

2.5.b (2 分)

这是否支持分类器除了学习动物的颜色之外，实际上并未学习其他东西的假设？

答：纯MLP还是学到了很多颜色之外的东西，test的精度并没下降太多。

2.6 (2 分)【完成】

接下来，考虑使用非常小的图像会发生什么情况。将每个图像的尺寸调整为(28×28)（MNIST数字的标准尺寸）。在这种情况下，测试准确率是多少？（回答完此问题后，恢复将图像调整为(256×256)大小。）

Train accuracy = 68.3%, Train avg loss = 0.892776

Val accuracy = 69.0%, Val avg loss = 0.861081

Test accuracy = 62.6%, Test avg loss = 0.946553

2.7【完成】

记录最后一个epoch时训练集、验证集、测试集的第一个batch的预测结果，并在其wandb标题中包含其预测结果（例如，鹦鹉、独角鲸、墨西哥钝口螈）和真实标签。标题格式为“<预测标签> / <真实标签>”。

2.7.a (2 分)

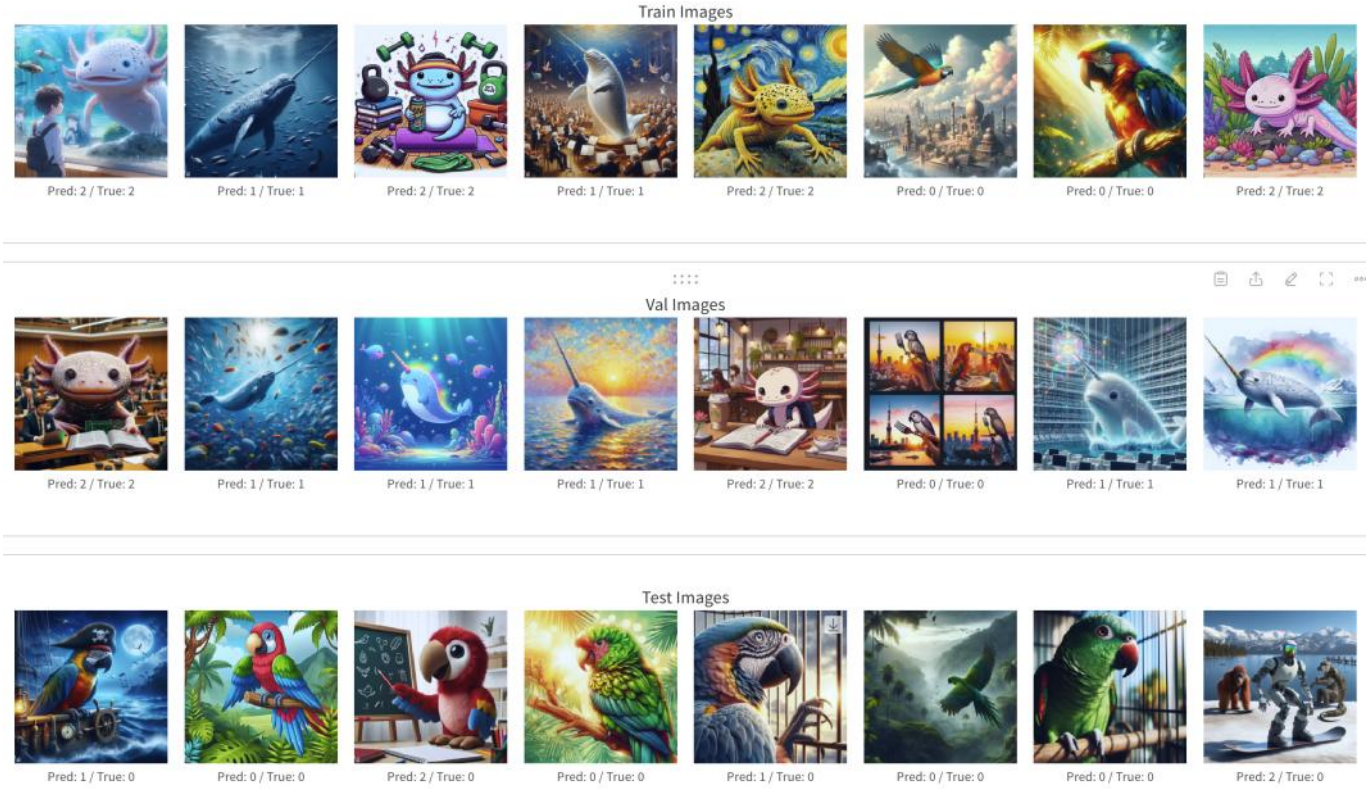
在wandb中显示训练数据集的第一个batch，并适当地为其加上标题。

2.7.b (2 分)

在wandb中显示验证数据集的第一个batch，并适当地为其加上标题。

2.7.c (2 分)

在wandb中显示测试数据集的第一个batch，并适当地为其加上标题。



2.7.d (2 分)

选择模型出现错误的几张图像，试着解释为什么模型会在这些图像上遇到困难。答案用几句话概述即可。

答:

- test第1个：估计背景与海豚的样本很相似，所以预测为海豚。
- 其他的看不出原因。。。

2.8【完成】

我们当前的优化器是随机梯度下降（torch.optim.SGD）。更换为torch.optim中的其他优化器，并调整超参数，看看是否能比SGD获得更好的性能。如果性能没有提升，你仍然可以获得满分。（回答完此问题后，恢复使用SGD。）

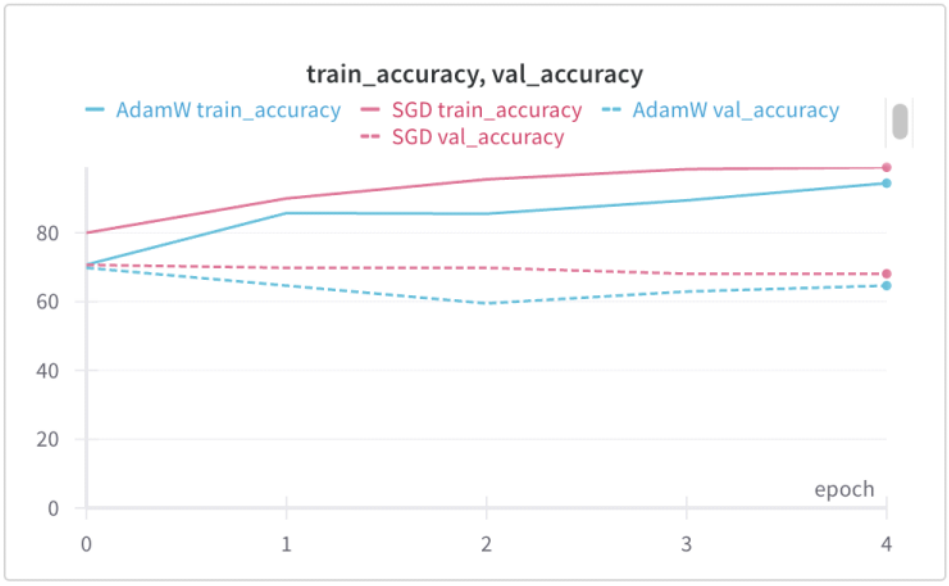
2.8.a (2 分)

报告你选择的优化器和超参数。

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

2.8.b (3 分)

然后，在同一wandb图表上绘制训练准确率和验证准确率，横轴为epoch数。图表上要同时包括SGD和新优化器结果。结果曲线各自以优化器的名称命名。



adamw优化器的精度：（估计是没用学习器，导致效果还不如SGD）

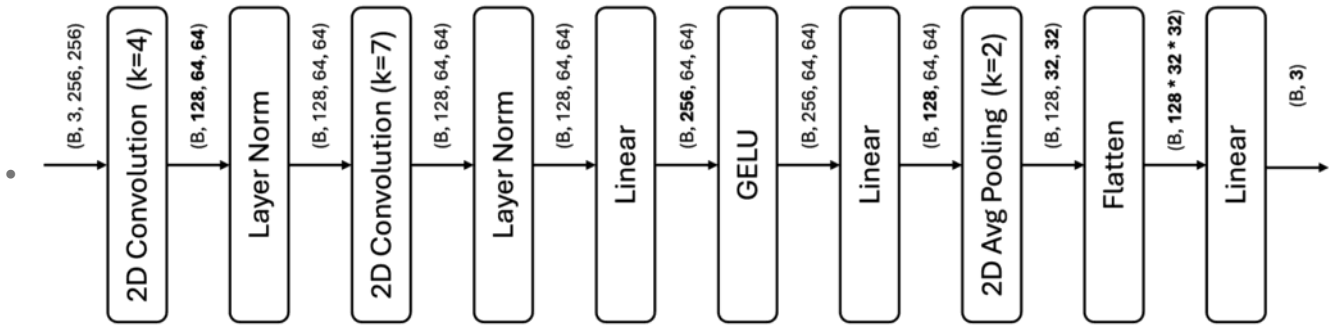
Train accuracy = 94.4%, Train avg loss = 0.186344

Val accuracy = 64.7%, Val avg loss = 2.593634

Test accuracy = 59.1%, Test avg loss = 2.827169

2.9【完成】

我们使用的模型非常简单：一个具有两个隐藏层（每层512个单元）和ReLU激活函数的前馈神经网络。定义一个实现以下计算图的新模型：



- 每个元组表示上一层的输出尺寸（批大小、通道数、图像高度、图像宽度），`k`表示卷积核的大小。

你应当阅读torch.nn的文档，并为计算图的每一层选择对应的模块。

备注：上图架构有明显的错误。。。架构图中间的2个线性层应该是卷积层而不是全连接，因为输入输出维度并没拉平，而且如果接了2层全连接层后又接个2D平均池化，这是什么奇葩操作。。。维度根本输入不进去。。。估计后续官方会修正作业讲义里的错误。。。

上图中间2层的linear我换用卷积（k=3），并且LN改用BN。

2.9.a (1 分)

报告新模型的最终训练/测试准确率。

数据集	准确率
训练集	
测试集	

还测试了resnet18（1000万参数）模型精度：

Train accuracy = 67.6%, Train avg loss = 0.796635

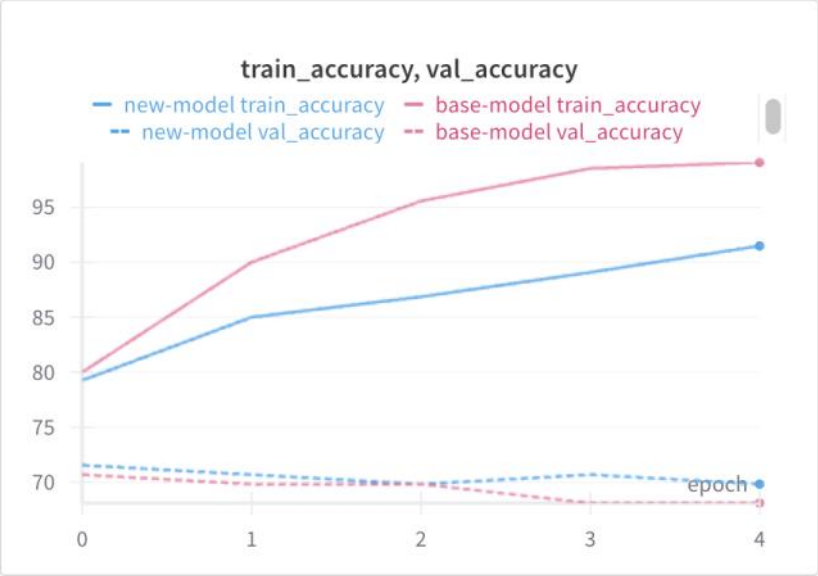
Val accuracy = 66.4%, Val avg loss = 0.767702

Test accuracy = 52.2%, Test avg loss = 0.997682

S24届人的代码里显示精度也较低，估计是数据集太少太少，不能用大参数模型，而且没用学习器等原因导致。

2.9.b (4 分)

使用wandb，在同一图表上绘制训练准确率和验证准确率，横轴为epoch数。包括原始模型和新模型的结果。将你的结果分别命名为“new-model”和“base-model”，并包含图例。



2.9.c (2 分)

原始模型和新模型中各有多少个参数？（请参考recitation手册了解如何计算这些数字。）

模型	参数数量
原始模型	100,928,003
新模型	1,793,155

2.9.d (2 分)

你的新模型在测试集上的表现是否优于或劣于原始模型？哪些因素可能解释了模型之间的准确率差异？

答：val精度更好了点，估计是过拟合降低了导致，以及使用了卷积特征提取。至于精度差距很小，很大原因估计在于数据集太少，只有200来张，并且类内差距较大，CNN模型难以收敛，epoch又少，也没有学习器、数据增强等等原因。

3 文本分类（24分）

你将使用一个包含真实和虚假新闻文章的数据集，这些新闻文章的来源包括全球的新闻媒体，尤其是小镇本地新闻。而虚假文章是通过一个大型语言模型生成的，生成时使用了对应真实新闻文章的标题作为提示词。

数据集分为三个部分：训练集、验证集和测试集，每个部分都以相同的格式存储在CSV文件中。每行对应一个新闻文章样本，包括文章文本、来源、标签（真实或虚假）以及标签的整数表示。训练集占总数据的70%，验证集和测试集各占15%。

txt_ classifier. py 中的启动代码是基于 PyTorch 文本分类教程的一个功能齐全（尽管简单）的文本分类器。在继续之前，您应该先阅读本教程，因为它不同于 PyTorch 的主要教程。

起始笔记本中模型文本类别分类的原理

初始数据形态：一段段的英文句子。每个段落的句子对应一个标签0或1，代表是否是AI生成的新闻。

数据集制作和处理过程：首先根据训练集构建出一个词汇表，然后将train、val、test集中的文本转成的单词索引号，并截断超过1024个词汇样本的后续信息。每个batch中数据，都填充至每个batch中最长样本的长度（目的是为了后续在模型里能批处理）。

模型结构：就2层：

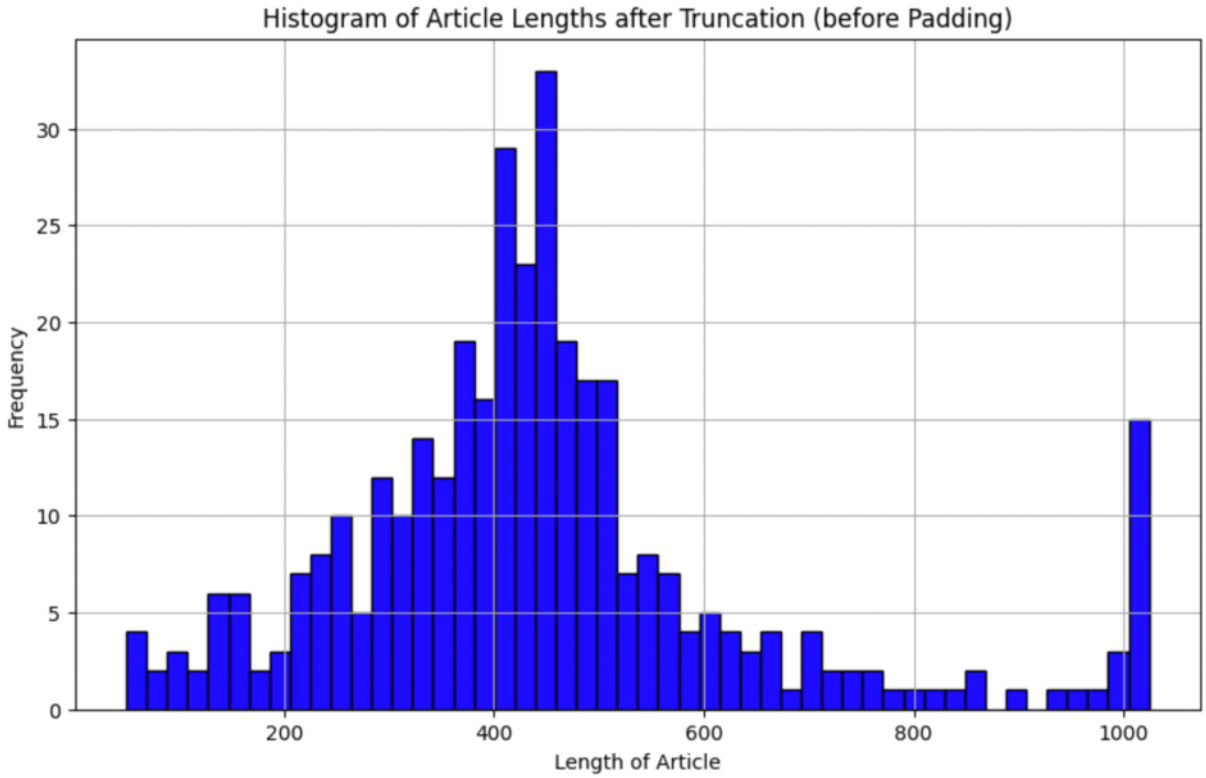
- nn.EmbeddingBag(vocab_size, embed_dim, sparse=False)【EmbeddingBag相当于把每个序列样本信息浓缩成一个 embed_dim维度向量，这跟Embedding不一样。】
- nn.Linear(embed_dim, num_class)

损失函数、学习器、优化器：交叉熵、StepLR、SGD

精度的评估标准：预测正确数/总数

3.1 (3 分)

生成在应用 `MAX_LEN` 截断后但在对序列进行填充之前的新闻文章长度的直方图。你可以使用任意方法来完成此任务。（一个选择是使用wandb。如果使用wandb，你应该按照"在总结中使用直方图"的说明进行操作。你需要点击运行，然后点击"概览"标签，然后滚动到页面底部找到直方图。）



3.2 (3 分)

运行代码三次，并报告验证集和测试集的准确率。（如果这些数据被记录下来，你可以在wandb的"运行"标签上轻松创建这样一个表格。）

运行	验证准确率	测试准确率
运行1	0.684	0.654
运行2	0.737	0.769
运行3	0.684	0.705

3.3 (2 分)

将优化器切换为 `torch.optim.Adam`，然后运行代码三次，并报告验证集和测试集的准确率。（不要恢复此更改，并在后续问题中继续使用Adam优化器。）

运行	验证准确率	测试准确率

运行1	0.921	0.885
运行2	0.921	0.897
运行3	0.947	0.923

3.4

仔细阅读 `nn.Embedding` 和 `nn.EmbeddingBag` 的文档，特别注意 `padding_idx` 参数和相关示例。然后阅读 `txt_classifier.py` 中的 `PadSequence` 代码，以便理解其工作原理。（回答完此问题后，恢复你在解答过程中所做的任何更改。）

3.4.a (1 分)

修改代码，使得不再向 `DataLoader` 传递 `collate_fn` 参数。报告此时出现的错误。

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[16], line 275
    272     print("test accuracy {:8.3f}".format(accu_test))  # 打印测试准确率
    274 if __name__ == '__main__':
--> 275     main()  # 执行主函数

Cell In[16], line 241
    240 def main():
--> 241     corpus_info, train_dataloader, val_dataloader, test_dataloader = get_data()
# 获取数据
    243     model = TextClassificationModel(corpus_info.vocab_size, EMBED_DIM,
corpus_info.num_labels).to(device)  # 初始化模型
    244     criterion = torch.nn.CrossEntropyLoss()  # 使用交叉熵损失函数

Cell In[16], line 167
    161 plt.show()
    166 # 打印一个批次的形状
--> 167 for X, y in train_dataloader:
    168     print(f"Shape of X [B, N]: {X.shape}")
    169     print(f"Shape of y: {y.shape} {y.dtype}")

File e:\soft_big\anaconda3\envs\10423\lib\site-packages\torch\utils\data
\dataloader.py:631, in _BaseDataLoaderIter.__next__(self)
    628 if self._sampler_iter is None:
    629     # TODO(https://github.com/pytorch/pytorch/issues/76750)
    630     self._reset()  # type: ignore[call-arg]
--> 631 data = self._next_data()
...
--> 169     raise RuntimeError('each element in list of batch should be of equal size')
    170 transposed = list(zip(*batch))  # It may be accessed twice, so we use a list.
```

```
    172 if isinstance(elem, tuple):

RuntimeError: each element in list of batch should be of equal size
```

3.4.b (2 分)

为什么会出现这个错误?

1. **数据加载器的要求**:

   PyTorch 的 `DataLoader` 在创建批次数据时，通常期望每个批次中的所有数据项具有相同的形状。这样的设计使得在使用 GPU 加速计算时能够有效地并行处理数据。如果数据项（在此案例中为文本序列）长度不一，会违反这一要求。

2. **批次组合的默认行为**:

   如果没有提供自定义的 `collate_fn` 来明确如何将数据项组合成批次，`DataLoader` 将尝试将单独的数据项简单地堆叠（stack）起来。由于文本序列长度不同，直接堆叠会导致维度不一致，无法形成合法的张量。

3. **维度不匹配的错误触发**:

   当 `DataLoader` 试图堆叠不同长度的序列时，会因为无法对这些序列进行对齐而触发 `RuntimeError`。错误信息 `"each element in list of batch should be of equal size"` 直接指出了问题所在，即列表中的每个元素（序列）必须具有相等的尺寸。

总结来说，由于没有在数据预处理中执行填充操作，导致试图加载批次数据时，由于序列长度不一致，无法成功构造统一形状的批次数据，因此触发了运行时错误。

3.4.c (1 分)

现在将batch大小更改为1（而不是8）。报告此时出现的错误。

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[19], line 275
    272     print("test accuracy {:8.3f}".format(accu_test))  # 打印测试准确率
    274 if __name__ == '__main__':
--> 275     main()  # 执行主函数

Cell In[19], line 241
    240 def main():
--> 241     corpus_info, train_dataloader, val_dataloader, test_dataloader = get_data()
# 获取数据
    243     model = TextClassificationModel(corpus_info.vocab_size, EMBED_DIM,
corpus_info.num_labels).to(device)  # 初始化模型
    244     criterion = torch.nn.CrossEntropyLoss()  # 使用交叉熵损失函数

Cell In[19], line 168
```

```
    166 # 打印一个批次的形状
    167 for X, y in train_dataloader:
--> 168     print(f"Shape of X [B, N]: {X.shape}")
    169     print(f"Shape of y: {y.shape} {y.dtype}")
    170     break

AttributeError: 'list' object has no attribute 'shape'
```

3.4.d (2 分)

为什么会发生这个新错误?

出现这个错误的原因是当 `batch_size` 被设置为 1 时， `DataLoader` 返回的批次实际上是一个包含单个元素的列表，而不是一个形状固定的张量。具体来说，当 `batch_size=1` 时，返回的数据结构是一个包含单个样本的列表，而不是像 `batch_size>1` 时返回的那样是一个堆叠好的张量。

在你的代码中， `X` 和 `y` 是从 `train_dataloader` 中取出的批次数据。当 `batch_size` 设置为 1 时， `X` 变成了一个列表（包含单个张量），而不是一个二维张量，因此访问 `X.shape` 会报错，因为列表对象没有 `shape` 属性。

3.5

将简单的 `TextClassificationModel` 替换为基于LSTM的分类器。这个模型可以是任意架构，只要它以合理的方式使用 `torch.nn.LSTM` （例如，单向LSTM、双向LSTM、深度LSTM）。对于那些不熟悉LSTM的人，可以使用HW0讨论课中展示的代码。请阅读PyTorch文档中的顺序模型和LSTM部分，了解更多示例和实现细节。

3.5.a (2 分)

用数学、伪代码或计算图描述你的新模型的结构。

```
# 文本分类模型类，使用Embedding、LSTM和全连接层
class TextClassificationModel(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_class, lstm_hidden_dim=128,
lstm_layers=2):
        super(TextClassificationModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)  # 创建嵌入层

        # LSTM层
        self.lstm = nn.LSTM(embed_dim, lstm_hidden_dim, num_layers=lstm_layers,
                            bidirectional=True, batch_first=True)

        # 全连接层
        self.fc = nn.Linear(lstm_hidden_dim * 2, num_class)  # *2 是因为双向LSTM


    def forward(self, text):
        embedded = self.embedding(text)  # 将文本转化为嵌入向量
```

```
        # 通过LSTM层
        lstm_out, _ = self.lstm(embedded)

        # 获取LSTM输出的最后一个时间步的隐状态（双向LSTM会输出两个方向的隐状态，需要拼接）
        out = lstm_out[:, -1, :]

        return self.fc(out)  # 将最后的隐状态传递给全连接层并输出
```

3.5.b (4 分)

绘制你新模型与原始模型的训练和验证准确率对比图。你的结果不需要比原始模型好。

新模型精度：

valid accuracy　0.605

test accuracy　0.603

这结果比用nn.EmbeddingBag差的是真多。。。

3.6 (2 分)

查看一些真实/虚假新闻文章的对比。在几句话中推测模型是如何成功训练的。

备注1：关于nn.Embedding和nn.EmbeddingBag

`nn.EmbeddingBag` 和 `nn.Embedding` 都是 PyTorch 中用于处理文本数据的嵌入层，但它们之间存在一些关键的区别，主要涉及它们的用途和处理方式：

1. 基本功能和用途

- **`nn.Embedding`**：
  - 这是一个简单的查找表，用于将离散的数字索引（代表词或其他类型的标记）转换为固定大小的密集向量。
  - 它通常用于处理序列数据，其中每个索引可以获得一个单独的词向量，非常适合用于序列模型如 RNN 和 LSTM。
  - 输出的维度为 `(N, L, E)`，其中 `N` 是批次大小，`L` 是序列长度，`E` 是嵌入维度。
- **`nn.EmbeddingBag`**：
  - `EmbeddingBag` 计算嵌入的“包”（或汇总），无需实例化每个嵌入向量。它包括对嵌入的加和或平均（通过 `mode` 参数控制），可以在一步中处理整个序列。

- 这种层特别适合于那些需要快速获得整个文本序列的单个嵌入表示的应用场景，如简单的文本分类任务。
- 输出的维度为 `(N, E)`，其中 `N` 是批次大小，`E` 是嵌入维度，它直接返回经过汇总的结果，不保留序列的时间步信息。

2. 性能和优化

- `nn.EmbeddingBag` 通常比 `nn.Embedding` 更高效，因为它直接计算整个序列的汇总表示而无需额外步骤。这在处理包含大量相似文本片段（如词袋模型）的大型数据集时特别有用。

3. 用例差异

- 如果你的任务涉及到理解序列中每个元素的独立贡献或顺序信息（如在语言模型或机器翻译中），则应使用 `nn.Embedding`。
- 如果任务只需从整个序列中抽取一种固定表示（如文本分类），且计算效率是关键考虑因素，那么 `nn.EmbeddingBag` 可能更合适。

4. 模式选择

- `nn.EmbeddingBag` 允许选择 `sum`、`mean` 或 `max` 模式来聚合嵌入，这使得它可以灵活地适应不同的需求，如不同类型的文本表示。

总结

选择 `nn.Embedding` 还是 `nn.EmbeddingBag` 取决于你的具体需求——是否需要序列中的每个元素的详细嵌入向量，或者你需要的是整个序列的快速且简洁的表示。在设计模型架构时，理解这两种嵌入方法的差异是非常重要的。

备注2：nn.EmbeddingBag模型一般如何分辨真假文章的?

使用 `nn.EmbeddingBag` 的模型在分辨真假文章（如假新闻检测或文本真实性分类）时，通常会基于以下几个步骤和原理进行：

1. 文本预处理

- **文本分词**：首先将文章拆分为单词或子词（token）。
- **索引映射**：使用词汇表（vocabulary）将每个单词或子词映射为一个唯一的整数索引。

- **嵌入表示**：这些整数索引随后会通过 `nn.EmbeddingBag` 层映射为相应的嵌入向量。

2. EmbeddingBag 汇总操作

- **汇总嵌入**：`nn.EmbeddingBag` 会将整个文章的词嵌入向量进行汇总操作（例如通过求和、平均或最大化），从而生成一个固定长度的向量表示。这种汇总表示了整篇文章的语义信息。
  - **Sum** 模式：对文章中所有词的嵌入向量进行求和，生成一个表示整篇文章的向量。
  - **Mean** 模式：对文章中所有词的嵌入向量进行平均，得到一个表示整篇文章的平均语义的向量。
  - **Max** 模式：取每个维度上的最大值，生成一个表示关键特征的向量。

3. 后续分类层

- **全连接层**：汇总后的向量表示会传递给一个或多个全连接层，这些层负责进一步处理该向量并提取出更高层次的特征。
- **分类层**：最后一个全连接层输出的向量通常会通过一个 softmax 层（或其他合适的激活函数），生成每个类别的概率分布。例如，模型可能会输出两个概率值，分别表示“真”或“假”的可能性。

4. 训练过程

- **目标函数**：使用交叉熵损失（`CrossEntropyLoss`）等适当的损失函数，模型在训练过程中通过调整参数来最小化预测值与真实标签之间的差异。
- **训练数据**：模型需要大量标注为真或假的文章作为训练数据。通过这些数据，模型可以学习到与真或假文章相关的语义特征。

5. 特征学习

- **词汇和模式**：模型通过学习哪些单词、短语或模式更可能出现在假新闻中，哪些则更可能出现在真实新闻中。例如，假新闻中可能会更多地使用夸张的形容词、误导性的标题或情绪化的词汇。
- **上下文理解**：尽管 `EmbeddingBag` 本身不保留词汇的顺序信息，但它可以通过多个单词的组合（词袋模型）学习到这些词汇的共同语义特征，从而捕捉到一些具有区分能力的上下文模式。

6. 推理和预测

- **输入新文章**：在推理阶段，当一篇新文章输入模型时，模型会通过与训练过程类似的步骤生成一个固定长度的嵌入表示，并通过分类层输出预测结果，判断文章的真实性。

关键点

- **词汇统计特征**：`EmbeddingBag` 汇总后的表示类似于传统的词袋模型（Bag-of-Words），它能捕捉到文章中词汇的出现频率和组合特征，虽然不考虑词序，但对于识别假新闻中的典型词汇模式仍然有效。
- **模型训练的关键**：模型的成功依赖于大量的标注数据以及足够复杂的分类器来捕捉假新闻中微妙的词汇模式

或语义特征。

优势和局限

- **优势**：`nn.EmbeddingBag` 简洁高效，适合处理大规模文本数据，尤其是在文本长度不一致的情况下；它能够快速生成文本的整体表示。
- **局限**：由于 `EmbeddingBag` 丢失了序列信息，它在捕捉上下文依赖的复杂模式时可能不如序列模型（如 RNN、LSTM、Transformer）强大。因此，它通常适用于较为简单或规则的文本分类任务。

总结来说， nn.EmbeddingBag 模型通过学习文章中的词汇特征，能够有效地对文章进行真假分类，尤其是当假新闻在词汇选择或语义表达上有明显特征时。

备注3：为什么EmbeddingBag + 全连接层效果更好?

在你的项目中，`EmbeddingBag + 全连接层` 模型的精度达到 0.9，而 `Embedding + BiLSTM + 全连接层` 模型的精度只有 0.6，这种结果可能由多种原因导致。下面是一些可能的解释：

1. 数据特征与模型匹配度

- **数据的性质**：如果你的数据集更倾向于通过词汇频率或简单的词汇组合来区分真假新闻，那么 `EmbeddingBag` 模型可能更适合这种任务。对于某些文本分类任务，词序和上下文可能并不如单词的整体统计特征重要。
- **数据量与复杂性**：如果数据集相对较小或者训练数据中没有足够的复杂模式，`BiLSTM` 可能过拟合或难以学习到有效的模式。相反，`EmbeddingBag` 模型简单直接，能够更有效地从数据中提取特征。

2. 模型训练与超参数调优

- **训练过程**：`BiLSTM` 模型由于其复杂性，可能更难以训练。如果没有足够的训练时间、适当的学习率、优化器以及其他超参数的调优，模型可能无法充分学习到数据中的模式。
- **过拟合**：`BiLSTM` 模型由于其更高的复杂度，可能更容易过拟合，特别是在数据量不足或正则化不足的情况下。这可能导致在训练集上表现良好，但在验证或测试集上表现较差。
- **初始化与收敛**：复杂模型如 `BiLSTM` 对参数初始化和训练初期的收敛非常敏感，如果没有正确的初始化或合适的训练策略，模型可能无法达到理想的性能。

3. 特征表示与信息丢失

- **BiLSTM 中的信息处理**：虽然 `BiLSTM` 能捕捉序列信息，但如果输入数据中的特征主要依赖于词汇的整体统计分布（如某些关键词的频繁出现），而不是词序，那么 `BiLSTM` 可能没有利用好这些信息。
- **`EmbeddingBag` 的优势**：`EmbeddingBag` 能快速聚合并简化数据，这可能对你特定的数据集更为有效，尤其是在词汇统计特征已经足够区分类别的情况下。

4. 模型复杂性与数据适应性

- **模型复杂性**：对于较简单或规则化的分类任务，过于复杂的模型反而可能带来不必要的噪音和过拟合问题。`EmbeddingBag` 这种较为简单的模型可能在这种情况下更为鲁棒。
- **数据和模型的适配**：数据集本身可能更适合简单的模型，复杂模型在捕捉到更多数据中的噪声后，可能会导致整体性能下降。

总结

在你的项目中，`EmbeddingBag + 全连接层` 模型表现更好，可能是因为数据集的特征更适合通过简单的词汇统计特征进行分类，而不依赖于序列信息和上下文依赖。`BiLSTM` 这种更复杂的模型可能因为数据量、模型训练、超参数调优等因素，无法充分发挥其潜力，甚至可能因为过拟合等问题导致性能下降。

这提醒我们，选择模型时需要充分考虑数据的特性和任务的需求，简单的模型有时在特定场景中反而能取得更好的效果。