Deep Learning 最終課題

名前: 田端諒大

1　はじめに

私は，VQA の課題を取り組んだ．以下に，ベースラインコードの順番を基にいくつかの段落に分けてそれぞれの工夫点を記述する．

2　データ準備

はじめに，google drive をマウントするコードの後に，google drive から google colaboratory のランタイムにすべてのデータをコピーした．これを行わないと，学習ごとに画像や文字データを google drive から取ってくることになり，学習時間が異常に長くなる．

ランタイムにコピーすることで，ベースラインコードのまま学習させると 1 エポック約 15 分程度で学習が可能となる．

3　乱数定義，文字処理，データ処理

乱数定義，文字処理，データローダーの作成を行うコードについて，ベースラインコードにおいては，これらは def set_seed，def process_text，class VQADataset で書かれている．

この三つに関しては，必要な内容が全て入っており，改善の必要があまりないと考えたため，変更を行わなかった．これらの処理を施されたデータをどのように学習させるかという所に主に焦点を当てたため，変更は行わなかった．

4　評価指数の実装

ベースラインの VQA の定義を基に def VQA_criterion を定義したため，このコードもベースラインの状態から変化を加えなかった．

5　モデルの実装

モデルに関しては様々な試行錯誤を行ったが，最終的にほとんどベースラインコードと同じモデルを用いることとなった．

ベースラインコードの class BottleneckBlock は変化を加えずに，class Resnet においては，dropout 層を含めた．最後に，ResNet50 を用いて，テキストの特徴量を学習する層と連結させた．

これらを考慮して，モデルとしては以下を用いた．

```
class ResNet(nn.Module):
    def __init__(self, block, layers):
        super().__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, layers[0], 64)
        self.layer2 = self._make_layer(block, layers[1], 128, stride=2)
        self.layer3 = self._make_layer(block, layers[2], 256, stride=2)
        self.layer4 = self._make_layer(block, layers[3], 512, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(p=0.3)
        self.fc = nn.Linear(512 * block.expansion, 512)

    def _make_layer(self, block, blocks, out_channels, stride=1):
        layers = []
        layers.append(block(self.in_channels, out_channels, stride))
        self.in_channels = out_channels * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.fc(x)

        return x

def ResNet50():
    return ResNet(BottleneckBlock, [3, 4, 6, 3])

class VQAModel_init50(nn.Module):
    def __init__(self, vocab_size: int, n_answer: int):
        super().__init__()
        self.resnet = ResNet50()
        self.text_encoder = nn.Linear(vocab_size, 512)

        self.fc = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
```

```
            nn.Linear(512, n_answer)
        )

    def forward(self, image, question):
        image_feature = self.resnet(image)  # 画像の特徴量
        question_feature = self.text_encoder(question)  # テキストの特徴量

        x = torch.cat([image_feature, question_feature], dim=1)
        x = self.fc(x)

        return x
```

最終的に採用はしなかったが，いくつかのモデルの変更を試みた．以下にその内容を示す．

- 事前学習モデルのファインチューニング

  事前学習済みの resnet モデルを外部から取り込み，それを ResNet50() の代わりに用いた．具体的には以下のようなモデルを組んだ．

```
class VQAModel_pretrained_resnet50(nn.Module):
    def __init__(self, vocab_size: int, n_answer: int):
        super().__init__()
        self.resnet = models.resnet50(pretrained=True)
        self.resnet = nn.Sequential(*list(self.resnet.children())[:-1])
        self.text_encoder = nn.Linear(vocab_size, 512)

        self.fc = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, n_answer)
        )

    def forward(self, image, question):
        image_feature = self.resnet(image)  # 画像の特徴量
        image_feature = image_feature.view(image_feature.size(0), -1)

        question_feature = self.text_encoder(question)  # テキストの特徴量

        x = torch.cat([image_feature, question_feature], dim=1)
        x = self.fc(x)

        return x
```

  Resnet18，Resnet34，Resnet50，Resnet101 などの様々な resnet の事前学習モデルを用いて試した．Resnet モデルは画像分類に特化しており，モデルの学習が済んでいるため学習を始める前から高い精度で結果を出力することができる．

  今回の VQA のタスクにおいても画像分類と似たような画像特徴量を取ると考え，事前学習モデルを導入することに決めた．このモデルは VQA にも適応できると考えていた．

  しかしながら，結果としては，どの事前学習モデルを用いても精度が 0.45 程度で学習率の伸びもいまいちであった．ベースラインコードで実装した場合とほとんど精度が変わらない，もしくは低くなるという結果に至った．また，実装時間も 20 分程度と少し長くなったため，この手法は諦めた．

  学習モデルとしては画像の特徴量を抽出する resnet 層とテキストの特徴量を抽出する text_encoder 層に分かれている．事前学習した Resnet モデルを用いれば画像特徴量においては非常に高い精度で抽出が可能になり，精度が飛躍的に上がると考えたが，実際はそうではなかった．これは，画像分類において必要とされる特徴量と VQA で必要とされる特徴量が全く異なるものであり，画像分類用に学習したモデルを用いても VQA には使えないためでないかと考えている．

また，resnet 以外にも vgg16 や Inception-Resnet などのファインチューニングも試みたが，こちらもあまりうまくいかなかった．

- 双方向 LSTM の導入

  ベースラインコードでは，テキストの特徴量を捉えるモデルが少し単純すぎると感じたため，変更を検討した。具体的には，文章中の文脈をより正確に捉えるために双方向 LSTM を導入しようと考えた．以下のコードを作成した．

```
class VQAModel_lstm_50(nn.Module):
    def __init__(self, vocab_size: int, emb_dim: int, hid_dim: int, n_answer: int):
        super().__init__()
        self.resnet = ResNet50()
        self.resnet.fc = nn.Identity()
        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.bilstm = nn.LSTM(emb_dim, hid_dim, batch_first=True, bidirectional=True)

        self.fc = nn.Sequential(
            nn.Linear(2048 + 2 * hid_dim, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, n_answer)
        )

    def forward(self, image, question):
        image_feature = self.resnet(image)

        question = question.long()
        embedded = self.embedding(question)

        pad_token = 0
        question_lengths = (question != pad_token).sum(dim=1).cpu()
        question_lengths = question_lengths.to(torch.int64)
        packed = nn.utils.rnn.pack_padded_sequence(embedded, question_lengths, batch_first=True, enforce_sorted=False)
        packed_output, (hidden, cell) = self.bilstm(packed)

        question_feature = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)

        x = torch.cat([image_feature, question_feature], dim=1)
        x = self.fc(x)

        return x
```

  このコードは，テキスト特徴量抽出において文章の文脈に焦点を当て，より複雑なモデルを適用することで全体の精度向上を目指していた．具体的には双方向 LSTM モデルを導入し，前後の文脈から単語の意味をより高精度で予測することを期待していた．

  しかしながら，実際には，1 エポックの学習時間が一時間弱かかる一方で，1 エポック当たりの学習率が平均 0.02 程度しか向上しなかった．あまりにもコストパフォーマンスが悪いと考えたため，このモデルも諦めた．

最終的に，モデルを複雑化しても学習時間がかかる一方で精度の向上があまり見られないということがわかった．そのため，モデルはベースラインコードからあまり複雑にせずに，学習回数をとにかく増やして最大の学習精度になるまで学習させるという方針に転換した．

それに際して，過学習への心配もあるために，dropout 層を resnet モデルに加えた．また，後にも記述するが，データセットの前処理においても過学習を防ぐための処置を施した．

6　学習モデル，評価モデル

学習，評価を行うモデルとしては，ベースラインコードの def train，def eval と同じものを用いた．双方向 LSTM を導入した際には若干の変更を加えたが，このモデルは結果的に使わなかったために，最終的にベースラインコードの定義と同じとなった．

7　データ前処理

得られた画像データに対して次のような変換を行う前処理を行った．

```
transform = transforms.Compose([
    transforms.RandomResizedCrop(256),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Normalize で学習の安定化を行った．また，RandomResizedCrop や，RandomHorizontalFlip，ColorJitter を用いて，モデルの汎化性能やロバスト性の向上を目指した．これによって過学習や異常値に対してもモデルの精度を安定化させることができる．

また，訓練データローダーについて，バッチサイズを 128 にすると，ResNet50() では容量不足となってしまうため，64 にした．

8　学習

実際に学習を行うコードは以下のようにした．

前述の通り，学習を多数行う必要性があったためにチェックポイントパスに学習中の情報を記録し，自由なタイミングで学習を中断，続行させることができた．

エポック数を多くし，学習精度が最大となった時のみ，モデル，エポック，オプティマイザー，スケジューラー，精度の最大値を記録した．

これによって，精度向上が見込まれる．さらに，学習が途中で終了した場合でも，そのモデルを用いて再度学習を継続することが可能となる．この工夫により，学習プロセスが大幅に便利になり，効率的に進めることができた．

```
model = VQAModel_init50(vocab_size=len(train_dataset.question2idx) + 1, n_answer=len(
    train_dataset.answer2idx)).to(device)

num_epoch = 100
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)

steps_per_epoch = len(train_loader)
print(f"batch_size: {steps_per_epoch}")
scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.01, epochs=
    num_epoch, steps_per_epoch=steps_per_epoch)

checkpoint_path = "/content/drive/MyDrive/deep learning/models/QVA_params_model_renew
    .tar"

checkpoint = torch.load(checkpoint_path)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
scheduler.load_state_dict(checkpoint["scheduler_state_dict"])
epoch = checkpoint["epoch"]
best = checkpoint.get("best")

```

```python
for epoch in range(num_epoch):
    train_loss, train_acc, train_simple_acc, train_time = train(model, train_loader,
        optimizer, criterion, device)
    print(f"【{epoch + 1}/{num_epoch}】\n"
          f"train time: {train_time:.2f} [s]\n"
          f"train loss: {train_loss:.4f}\n"
          f"train acc: {train_acc:.4f}\n"
          f"train simple acc: {train_simple_acc:.4f}")

    if train_acc > best:
        torch.save(
            {
                "epoch": epoch,
                "model_state_dict": model.state_dict(),
                "optimizer_state_dict": optimizer.state_dict(),
                "scheduler_state_dict": scheduler.state_dict(),
                "best": best,
            },
            "/content/drive/MyDrive/deep learning/models/QVA_params_model_renew.tar",
        )

        best = train_acc

print(best)
```

はじめ，エポックを 100 として学習を行った．学習後にまだ精度向上の兆しが見えたため，再度学習を行うことを何度か行った．最終的に，合計 300 回ほど学習を行ったところで精度の向上が止まったため，学習を終了した．

9　学習のやり直し

300 回ほど学習させたモデルを用いて予測データを出したところ，精度が 0.41 程度であり，あまりよくなかった．これは学習過程で過学習を起こしてしまっていたということが考えられる．

そのため，計測途中で獲得していたモデルを用いていくつか予測データを作り，最も精度が高かったものを最終的な結果として提出した．

10　最後に

github へのステージングの際に画像，文書データの容量が大きすぎてステージングができないため，別のディレクトリに移し，github にアップロードしなかった．

また，作成したモデルについても 500 MB 程度あり，いくつか試行錯誤をしたがプッシュすることができなかった．そのため，作成したモデルも git に上げることを諦めた．(学習コードは全て github に上がっているため，用いたモデルはコード内で定義されるようなモデルであるということをご承知おきください)．

以上より，github においては今回の学習において用いたすべてのコード，この pdf のみである．