Tags

VQAプロジェクトにおいて、特に効果的だったデータ拡張とPyTorchの高速化に関して、まとめます。

1. データ拡張と前処理

データ拡張は、モデルの汎化性能を向上させるために非常に重要な役割を果たしました。データ拡張を行うことで、モデルが多様なデータに対しても強くなる。つまり、汎化性能が向上することが期待されます。

- **訓練データ用のデータ拡張と前処理**
  訓練データに対しては、以下のようなデータ拡張を実施しました：
  - **リサイズ**: 全ての画像を224x224にリサイズし、入力サイズを統一しました。
  - **ランダム水平反転**: 画像をランダムに水平反転させることで、データの多様性を増やしました。
  - **カラージッター**: 明るさ、コントラスト、彩度、色相をランダムに調整し、画像の色合いの多様性を確保しました。
  - **テンソル化**: 画像をPyTorchテンソルに変換しました。
  - **正規化**: 各チャンネルごとに画像を正規化し、学習を安定させました。

```
pythonコードをコピーする
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2,
hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])
])
```

- **評価/テストデータ用の前処理**
  評価およびテストデータに対しては、データ拡張を行わずに以下の前処理のみを行いました：
  - **リサイズ**: 画像を224x224にリサイズしました。
  - **テンソル化**: 画像をPyTorchテンソルに変換しました。
  - **正規化**: 学習データと同様に各チャンネルごとに画像を正規化しました。

```
pythonコードをコピーする
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])
```

```
])
```

2. PyTorchの高速化

デフォルトのコードでは、PyTorchにおいてデータの読み込みやバッチ処理を高速化するために、以下の工夫を行いました：

- **DataLoaderの設定**
  - **バッチサイズの最適化**: 訓練データに対してはバッチサイズを16、テストデータに対してはバッチサイズを1とし、メモリの使用量と計算速度のバランスを最適化しました。
  - **複数スレッドの使用**: `num_workers`を4に設定し、データ読み込みを並列化することで、I/O待ち時間を短縮しました。
  - **ピンメモリの使用**: `pin_memory=True`を設定することで、CPUからGPUへのデータ転送を高速化しました。
  - **シャッフルの有効化**: 訓練データに対してはシャッフルを有効にし、エポックごとにデータの順序をランダムに変更しました。
  - **drop_lastの設定**: `drop_last=True`により、最後の不完全なバッチをドロップし、一貫したバッチサイズを維持しました。

```
pythonコードをコピーする
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16,
num_workers=4, pin_memory=True, shuffle=True, drop_last=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1, nu
m_workers=4, pin_memory=True, shuffle=False)
```

3. モデル

本プロジェクトでは、BERTとResNetを組み合わせたモデルを使用し、画像とテキストの特徴量を統合する工夫を行いました。

- **BERTの活用**: 自然言語処理で高い性能を示すBERTを用いることで、テキストから高品質な特徴量を抽出しました。
- **ResNetの活用**: 画像認識で実績のあるResNetを用いて画像特徴量を抽出しました。
- **特徴量の統合**: Multihead Attentionを用いて、テキスト特徴量と画像特徴量を効果的に統合し、最終的な予測精度を向上させました。

```
class VQAModel(nn.Module):
    def __init__(self, num_labels: int):
        super().__init__()
        self.bert = BertModel.from_pretrained("bert-base-uncased")
        self.visual_projection = nn.Linear(2048, self.bert.config.hidden_si
ze)
        self.attention = nn.MultiheadAttention(self.bert.config.hidden_siz
```

```
e, num_heads=8)
        self.classifier = nn.Linear(self.bert.config.hidden_size, num_label
s)

    def forward(self, input_ids, attention_mask, visual_embeds):
        bert_outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        text_features = bert_outputs.last_hidden_state  # [batch_size, seq_
len, hidden_size]

        visual_features = self.visual_projection(visual_embeds)  # [batch_s
ize, hidden_size]
        visual_features = visual_features.unsqueeze(1).expand(-1, text_feat
ures.size(1), -1)  # [batch_size, seq_len, hidden_size]

        # Attention機構を使用してテキストと画像特徴を結合
        combined_features, _ = self.attention(text_features, visual_feature
s, visual_features)

        # 分類のために[CLS]トークンの特徴量を使用
        logits = self.classifier(combined_features[:, 0, :])
        return logits
```

結論

本プロジェクトでは、データ拡張、PyTorchの高速化、およびBERTとResNetを組み合わせたモデルの工夫により、モデルの予測精度とトレーニング効率を向上させることができました。