コードの改善点

モデルの構造に関して

1. `ResidualBlock`の導入

変更前コードでは、`ConvBlock`のみで畳み込み層を積み重ねていましたが、変更後コードでは新たに`ResidualBlock`を導入しています。 これにより、残差接続が可能になり、勾配消失問題の緩和や、より深いネットワークの学習が容易になります。

```
- class BasicConvClassifier(nn.Module):
+ class ImprovedConvClassifier(nn.Module):
    def __init__(
        self,
        num_classes: int,
        seq_len: int,
        in_channels: int,
        hid_dim: int = 128
+       hid_dim: int = 512,  # 隠れ層の次元数を増やす
+       p_drop: float = 0.4,
    ) -> None:
        super().__init__()

        self.blocks = nn.Sequential(
            ConvBlock(in_channels, hid_dim),
            ConvBlock(hid_dim, hid_dim),
+           ResidualBlock(hid_dim),  # Residual Blockを追加
+           ConvBlock(hid_dim, hid_dim * 2),  # チャンネル数を増やす
+           ResidualBlock(hid_dim * 2),
+           ConvBlock(hid_dim * 2, hid_dim * 2),
        )
```

追加した`ResidualBlock`の役割は、主に以下の2点です。

- 勾配消失問題の緩和: 深いニューラルネットワークでは、誤差逆伝播法によって勾配が浅い層に伝わるにつれて勾配が小さくなり、学習がうまく進まなくなる勾配消失問題が発生しやすいです。`ResidualBlock`では、入力xをそのまま出力に加えるスキップ接続を導入しました。 これにより、勾配が直接浅い層に伝わる経路が確保され、勾配消失問題が緩和されます。

- 表現能力の向上: スキップ接続によって、ネットワークは恒等写像を学習することが容易になります。これにより、層を深くしても学習が安定し、より複雑な表現を獲得できる可能性を高くしています。また、`ResidualBlock`内には、畳み込み層、バッチ正規化層、ReLU活性化関数といった要素を含めており、これらが組み合わさることで、非線形性や正規化の効果をもたらし、モデルの表現能力が向上するようにしています。

```
class ResidualBlock(nn.Module):
  def __init__(self, channels):
    super().__init__()
```

```
    self.conv1 = nn.Conv1d(channels, channels, kernel_size=3, padding="same")
    self.bn1 = nn.BatchNorm1d(channels)
    self.conv2 = nn.Conv1d(channels, channels, kernel_size=3, padding="same")
    self.bn2 = nn.BatchNorm1d(channels)

  def forward(self, x):
    residual = x  # 入力xをresidualとして保持
    x = F.relu(self.bn1(self.conv1(x)))  # 畳み込み、バッチ正規化、ReLU
    x = self.bn2(self.conv2(x))  # 畳み込み、バッチ正規化
    x += residual  # residualとの和
    return F.relu(x)  # ReLU
```

2. 隠れ層の次元数の増加

変更前コードでは、`hid_dim=128`でしたが、変更後コードでは`512`に増加させています。 これにより、モデルの表現能力が向上し、より複雑な特徴を捉えられるようになります。

```
-       hid_dim: int = 128
+       hid_dim: int = 512,  # 隠れ層の次元数を増やす
```

3. `Dropout`の追加

変更後コードでは、`head`モジュールの全結合層の前に`Dropout`を追加しています。 これにより、過学習を抑制し、モデルの汎化性能を向上させる効果が期待できます。

```
        self.head = nn.Sequential(
            nn.AdaptiveAvgPool1d(1),
            Rearrange("b d 1 -> b d"),
+           nn.Dropout(p_drop),  # 全結合層の前にDropoutを追加
            nn.Linear(hid_dim * 2, num_classes),  # チャンネル数に合わせて変更
        )
```

4. 活性化関数と`Dropout`率の変更

変更前コードでは、活性化関数に`F.gelu`、`Dropout`率に`0.1`を使用していましたが、 変更後コードでは、`ResidualBlock`内で`F.relu`を使用し、`Dropout`率を`0.4`に変更しています。 これにより、学習の安定化や過学習の抑制が期待できます。

```
-       p_drop: float = 0.1,
+       p_drop: float = 0.4,

-       X = F.gelu(self.batchnorm0(X))
+       x = F.relu(self.bn1(self.conv1(x)))
```

5. チャンネル数の増加

変更後コードでは、`ConvBlock`を積み重ねる中で、チャンネル数を`hid_dim`から`hid_dim * 2`に増加させています。 これにより、モデルの表現能力がさらに向上します。

```
+            ConvBlock(hid_dim, hid_dim * 2),  # チャンネル数を増やす
```

データの前処理に関して

データの前処理は、今回は特に実施しませんでした。

リサンプリングやフィルタリング・スケーリング・ベースライン補正といった`Readme`に記載されているものを実装しても、自分の実装方法では精度が向上しなかったことが理由です。 別の実装方法を試せばまた違う結果になったのかもしれませんが、今回はモデルの改善だけでもベースライン基準を超えられました。

GitHubレポジトリのリンク

https://github.com/SeiichiTake/dl_lecture_competition_pub/tree/MEG-competition