

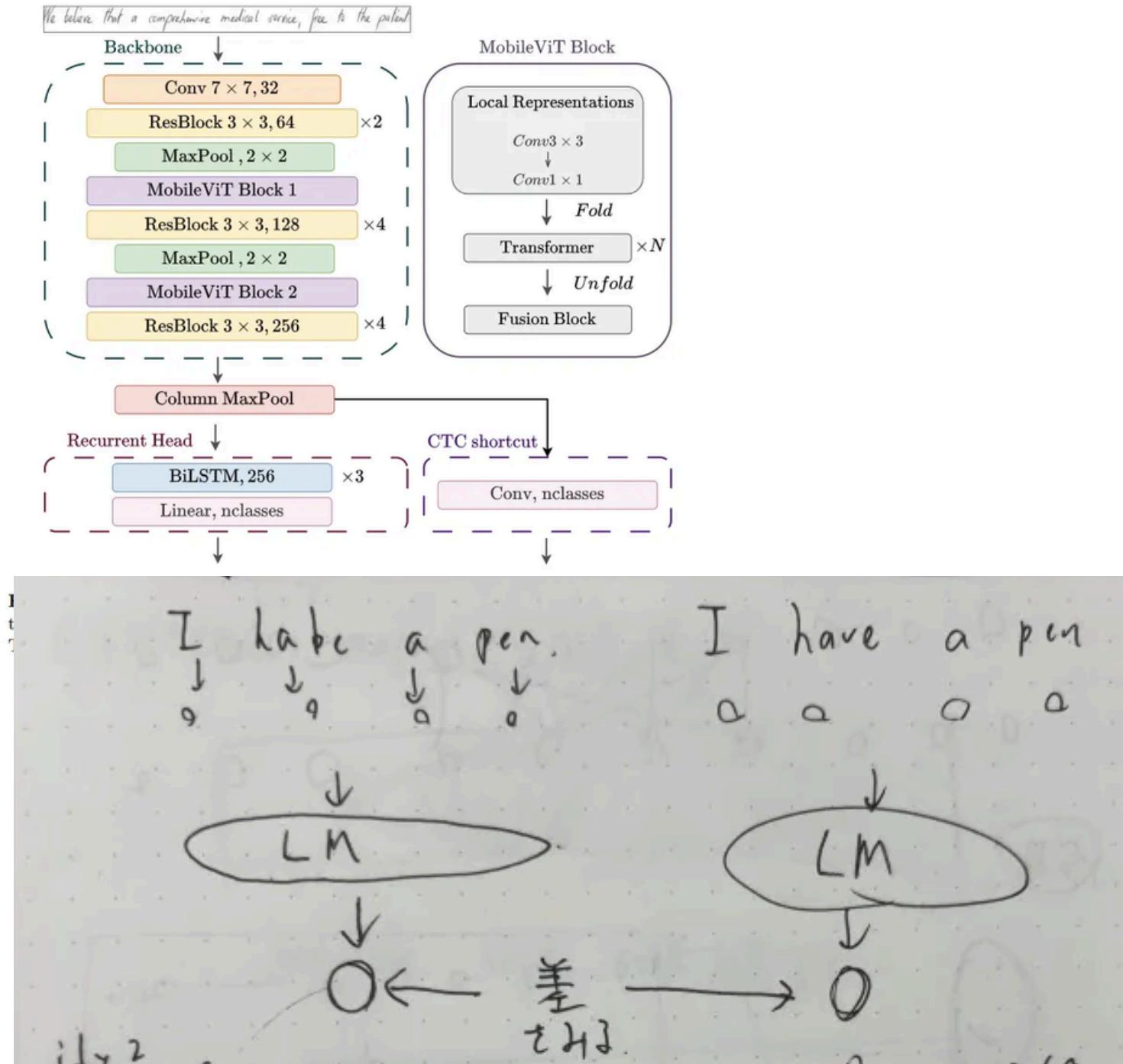
手書き文字認識における 言語モデルの有効性検証

既存手法でどんな誤りが多いのか



正解文字列: Become a success with a **disc** and hey presto! You're a **star**... Rolly sings with
CRNN予測 : Become a success with a **dise** and hey presto. You're a **starm**. Rolly sings with

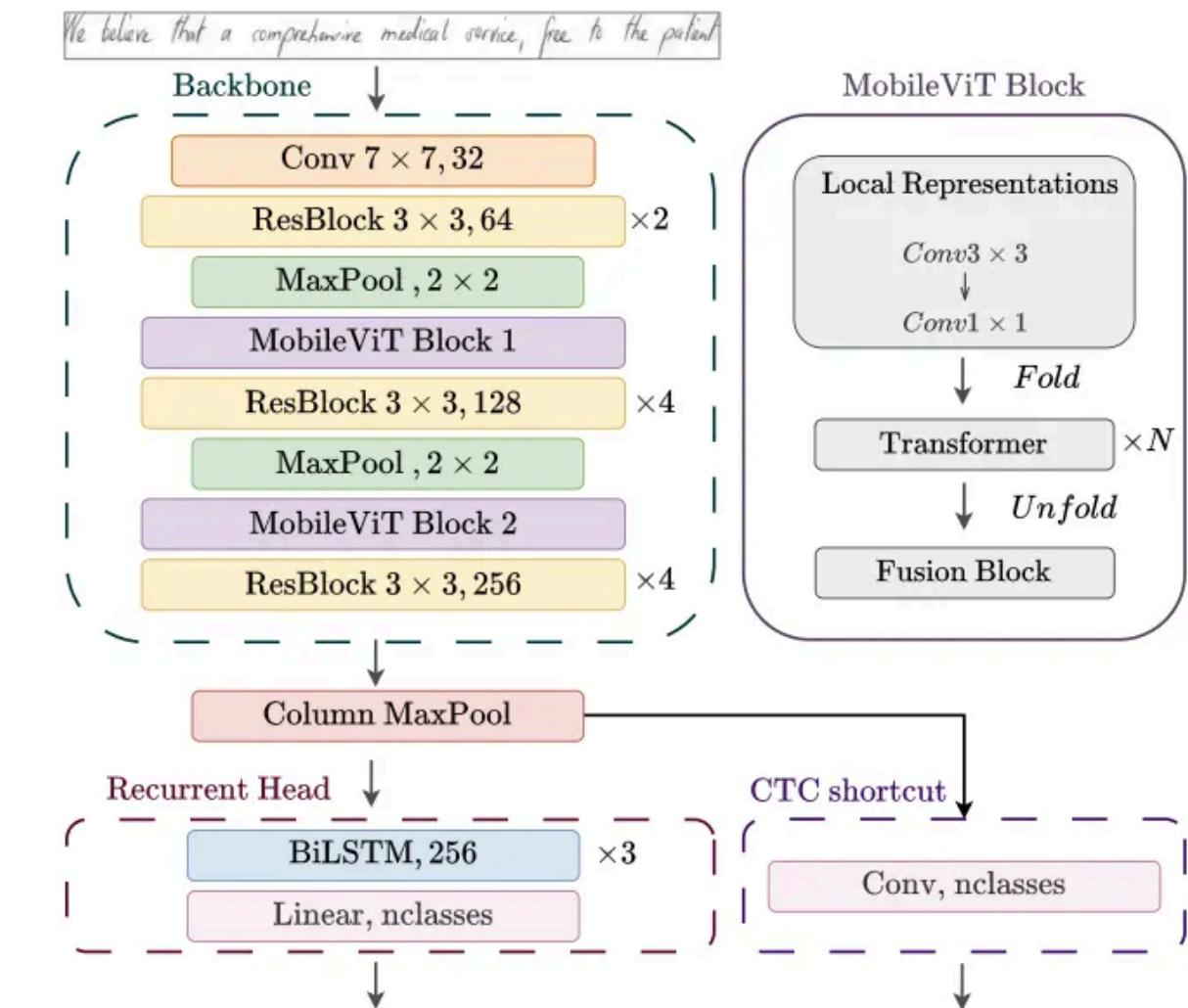
言語タスク



index	CTC出力Loss	正解文字列Loss
0	6.2707	5.2461
1	6.9152	5.2664
2	4.6952	3.3369
3	6.104	3.4619
4	5.0421	4.5322
5	5.228	4.8255
6	6.0486	4.6419
7	8.1825	5.8097
8	6.8154	5.6124
9	6.6855	4.0256
10	5.5432	5.1879
11	5.8678	5.8678
12	5.0739	5.0739
13	5.2923	4.7248
14	6.583	5.6528
15	5.3517	4.7984
16	4.647	3.7389
17	6.9486	5.0523
18	7.5657	6.2751
19	6.7299	6.7299
20	6.335	6.335
21	7.357	4.3485
22	5.581	5.6425
23	6.0287	6.0287
24	6.0911	5.4226
25	7.6662	6.3018
26	6.8882	6.9832
27	7.1863	6.1951
28	6.9063	6.4734
29	6.3067	4.1144

ここ

実装



文字列デコード

正解文字列

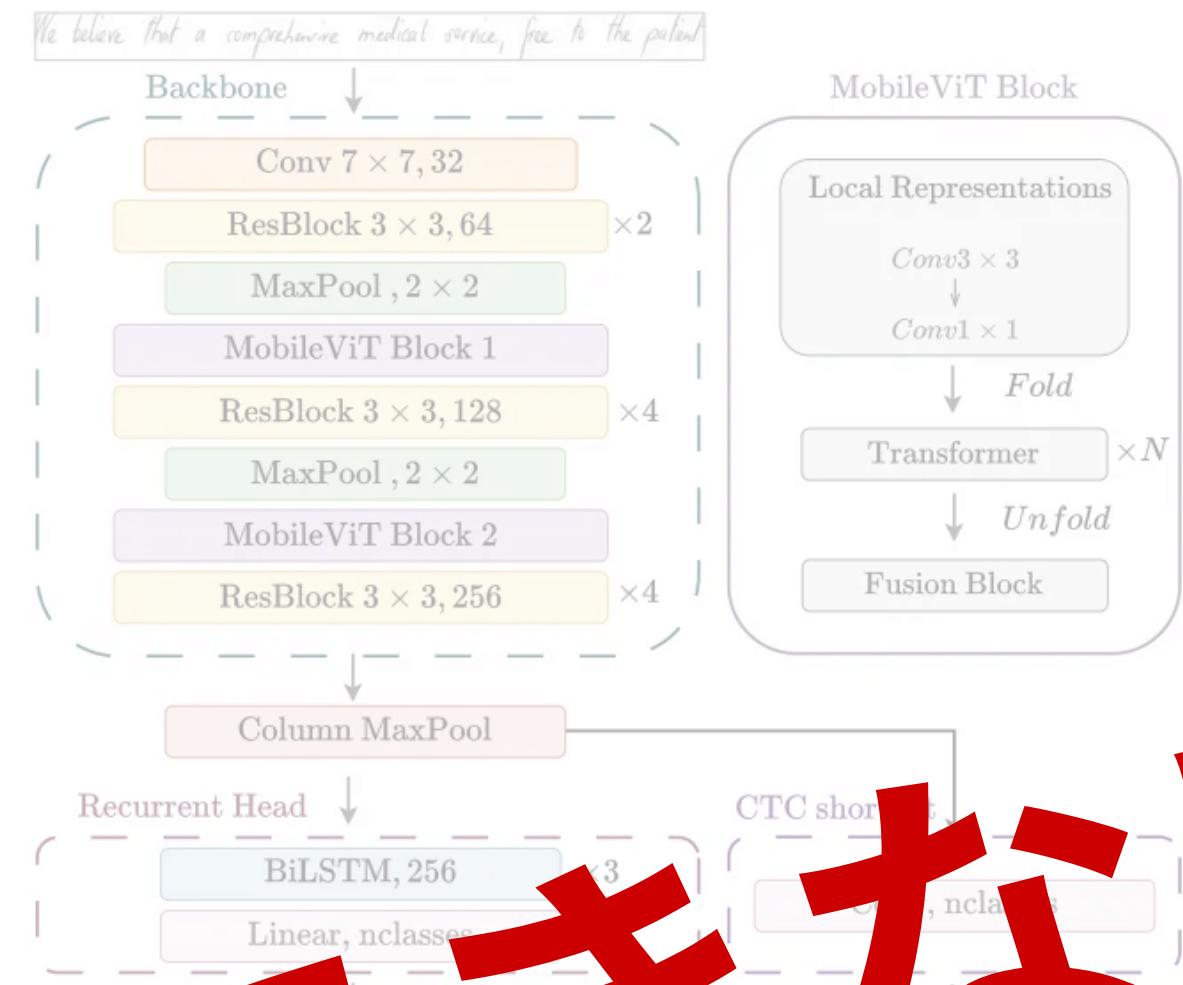
LM (GPT2-small)

LM (GPT2-small)

差を口スとする

実装

BPできない!



文字列デコ

LM (GPT2-small)

正解文字列

LM (GPT2-small)

差を口スとする

実装

```
if use_lm_loss and indices_lm is not None and self.lm_model is not None:
    from models import calculate_lm_loss_batch

    # サンプリングされたサンプルのCTC出力を取得
    lm_output = output[:, indices_lm, :] # (width, lm_batch, nclasses)

    # CTCデコードして予測文字列を取得
    pred_texts = self.decode_batch(lm_output, self.classes['i2c'], blank_id=0)

    # LM lossを計算 (pred_loss - label_lossの差分)
    lm_loss_raw = calculate_lm_loss_batch(
        pred_texts, transcr_lm,
        self.lm_model, self.lm_tokenizer, device
    )

    lm_weight = config.train.get('lm_weight', 0.1)
    lm_loss_val = lm_loss_raw * lm_weight
    loss_val = loss_val + lm_loss_val

    tloss_val = loss_val.item()

print("☆"*60)
print(loss_val)
print(loss_val.requires_grad) # True/False
print("☆"*60)
```

```
def decode(self, tdec, tdict, blank_id=0):
    tt = [v for j, v in enumerate[Any](tdec) if j == 0 or v != tdec[j - 1]]
    dec_transcr = ''.join([tdict[t] for t in tt if t != blank_id])

    return dec_transcr

def decode_batch(self, logits, tdict, blank_id=0):
    """
    パッチのCTCロジットをデコードして文字列リストを返す
    """

    Args:
        logits: CTC logits (width, batch, nclasses)
        tdict: クラスインデックスから文字へのマッピング辞書
        blank_id: ブランクトークンのID (デフォルト0)

    Returns:
        List[str]: デコードされた文字列のリスト
    """

    # argmaxで最も確率の高いクラスを選択
    tdec = logits.argmax(2).permute(1, 0).cpu().numpy() # (batch, width)

    decoded_texts = []
    for i in range(tdec.shape[0]):
        decoded_text = self.decode(tdec[i], tdict, blank_id)
        decoded_texts.append(decoded_text)

    return decoded_texts
```

```
tensor(0.2137, device='cuda:0')
False
```

実装

```
def calculate_lm_loss_diff(pred_text, label_text, lm_model, tokenizer, device):
    """
    CTC予測と正解文字列のLM loss差を計算

    Args:
        pred_text: CTC予測文字列
        label_text: 正解文字列
        lm_model: 言語モデル
        tokenizer: トークナイザ
        device: デバイス

    Returns:
        torch.Tensor: max(0, pred_loss - label_loss) (予測が正解より悪い場合のみペナルティ)
    """
    pred_loss = calculate_lm_loss_single(pred_text, lm_model, tokenizer, device)
    label_loss = calculate_lm_loss_single(label_text, lm_model, tokenizer, device)

    # 両方とも有効な場合のみ差分を計算
    # infチェック (Tensorの場合)
    if torch.isinf(pred_loss) or torch.isinf(label_loss):
        return torch.tensor(0.0, device=device)

    # 予測が正解より悪い場合のみペナルティ
    diff = pred_loss - label_loss
    return torch.clamp(diff, min=0.0) # max(0, diff) の微分可能版

def calculate_lm_loss_batch(pred_texts, label_texts, lm_model, tokenizer, device):
    """
    バッチ全体のLM loss差の平均を計算

    Args:
        pred_texts: CTC予測文字列のリスト
        label_texts: 正解文字列のリスト
        lm_model: 言語モデル
        tokenizer: トークナイザ
        device: デバイス

    Returns:
        torch.Tensor: バッチ平均LM loss差 (微分可能)
    """
    total_loss = torch.tensor(0.0, device=device)
    valid_count = 0

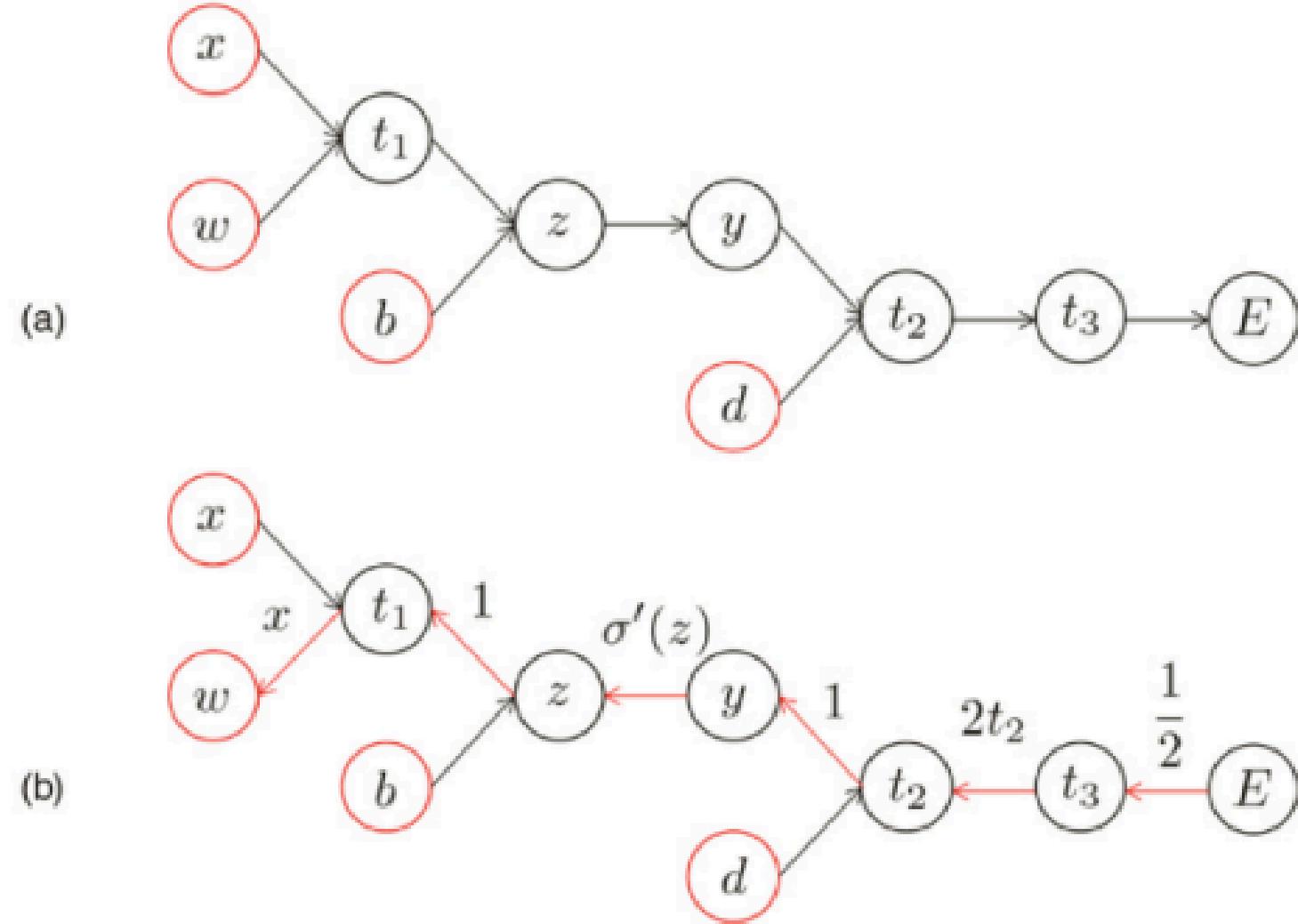
    for pred, label in zip(tuple(pred_texts, label_texts)):
        diff = calculate_lm_loss_diff(pred, label, lm_model, tokenizer, device)
        if diff.item() > 0:
            total_loss = total_loss + diff
            valid_count += 1

    # 有効なサンプルがない場合は0を返す
    if valid_count == 0:
        return torch.tensor(0.0, device=device)

    return total_loss / valid_count
```

```
tensor(0.2137, device='cuda:0')
False
```

なぜできないか



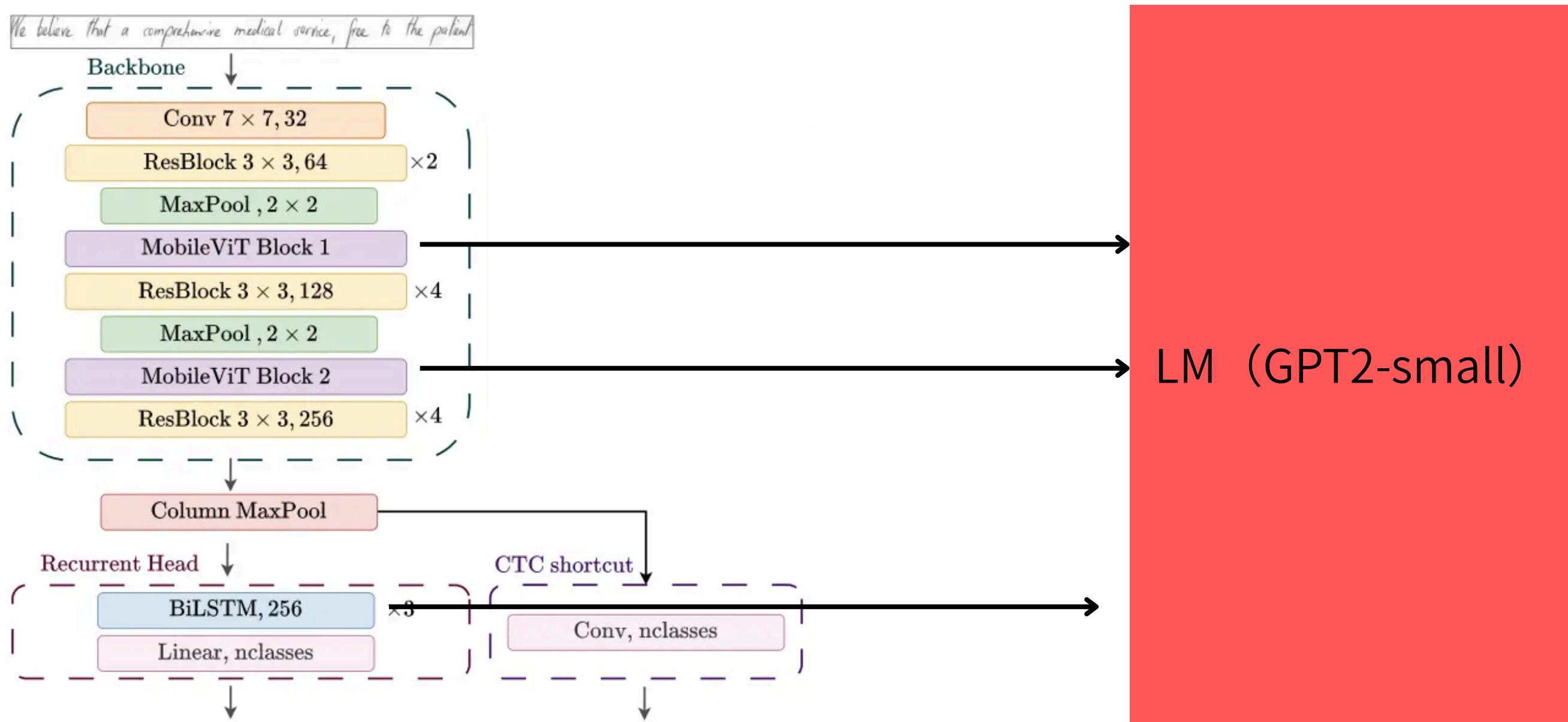
4.3 自動微分
p123

requires_grad のデバック

True : テンソルから計算された結果は 計算グラフ に記録され、.backward() を呼ぶとそのテンソルに対して勾配が計算される

False : 自動微分の対象外になり、勾配は計算できない。

実装 2



実装 2

