Homework1 Report

CMIU 11-775

Jooeon Kang

Q1. Introduce how do you improve the model. What's the performance? Why it works? If you have an extra time and GPU resource, what's your next steps?

A1.
Numerous experiments have been undertaken in an effort to enhance the model.
Initially, I modified the network, SoundNet, to extract optimal features by attempting to deepen the network through the addition of more layers. I also incorporated dropouts and regularizations to mitigate the overfitting issue. However, I later realized that since we were utilizing a pre-trained model, this approach was not conducive to improving the model.

Here is the command used to extract features with SoundNet:
$ python scripts/extract_soundnet_feats.py --feat_layer {the layer of the feature you wish to extract}

```
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--model_path', type=str, default='weights/sound8.npy',
                        help='Path to the .npy file with the SoundNet weights')
```

As you can see, we are using "weights/sound8.npy". It was evident that we were using a pre-trained model). Thus, even after enhancing the model with additional layers and dropouts, it would not be effective unless retrained.

Subsequently, my focus shifted towards determining which feature layer should be extracted. From the paper, "SoundNet: Learning Sound Representations from Unlabeled Video", I garnered insights that guided my trials with conv7, conv6, pool5, and conv5, among which pool5 demonstrated the best performance.

Model	conv4	conv5	pool5	conv6	conv7
8 Layer, AlexNet	84%	85%	84%	83%	78%
8 Layer, VGG	77%	88%	88%	87%	84%
8 Layer, AlexNet	66.0%	71.2%	74.2%	74%	63.8%
8 Layer, VGG	66.0%	69.3%	72.9%	73.3%	59.8%

To further optimize the model, I experimented with two different preprocessing methods involving manipulations on raw_audio. The objective was to scale the raw_audio to the desired range (-256~256). The methods used for preprocessing are as follows:

1) Normalize raw_audio into [0, 1], scale it up to [0, 512], then subtract 256 to adjust to the desired range of [-256, 256].

```
def preprocess(raw_audio, config=local_config):
    # Select first channel (mono)
    if len(raw_audio.shape) > 1:
        raw_audio = raw_audio[0]

    # Scale the audio to the desired range
    min_val = np.min(raw_audio)
    max_val = np.max(raw_audio)
    if max_val == min_val:
        raw_audio = np.zeros_like(raw_audio)
    else:
        # Scale to [0, 1]
        raw_audio = (raw_audio - min_val) / (max_val - min_val)
        #  Scale to [-256, 256]
        raw_audio *= 512
        raw_audio -= 256
```

2) Simply utilize np.clip to confine raw_audio within the desired range.

```
def preprocess(raw_audio, config=local_config):
    # Select first channel (mono)
    if len(raw_audio.shape) > 1:
        raw_audio = raw_audio[0]

    # Make range [-256, 256]
    raw_audio = np.clip(raw_audio * 256.0, -256, 256)
```

It was discerned that the second method yielded superior results.

Nevertheless, the score remained below 0.65, which was not in line with expectations. Consequently, I conducted numerous experiments to modify the MLP classifier's hyper-parameters. (Still working)

sklearn.neural_network.MLPClassifier

class `sklearn.neural_network.`**`MLPClassifier`**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) ¶ [source]

```
MLPClassifier(hidden_layer_sizes=(1024), activation="relu", solver="adam", alpha=1e-3,
              learning_rate="constant", learning_rate_init=1e-3, max_iter=200)
```

Upon encountering the MLP Classifier, I found its utilization to be challenging. Consequently, I constructed my own MLP using PyTorch. And I thought using three kinds of features would perform better so I made MultiHeadClassifier using conv5, pool5, conv6.

```
class MultiHeadClassifier(nn.Module):
    def __init__(self, conv5_dim, pool5_dim, conv6_dim, num_classes):
        super(MultiHeadClassifier, self).__init__()
        self.conv5_classifier = SingleHeadClassifier(conv5_dim, num_classes)
        self.pool5_classifier = SingleHeadClassifier(pool5_dim, num_classes)
        self.conv6_classifier = SingleHeadClassifier(conv6_dim, num_classes)

    def forward(self, x_conv5, x_pool5, x_conv6):
        out_conv5 = self.conv5_classifier(x_conv5)
        out_pool5 = self.pool5_classifier(x_pool5)
        out_conv6 = self.conv6_classifier(x_conv6)
        return out_conv5, out_pool5, out_conv6
```

```python
class SingleHeadClassifier(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(SingleHeadClassifier, self).__init__()
        self.block1 = nn.Sequential(
            nn.Linear(input_dim, 2048),
            nn.BatchNorm1d(2048),
            nn.ReLU(),
            nn.Dropout(0.5)
        )

        self.block2 = nn.Sequential(
            nn.Linear(2048, 2048),
            nn.BatchNorm1d(2048),
            nn.ReLU(),
            nn.Dropout(0.5)
        )

        self.block3 = nn.Sequential(
            nn.Linear(2048, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(),
            nn.Dropout(0.5)
        )

        self.block4 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.5)
        )

        self.block5 = nn.Sequential(
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.5)
        )

        self.block6 = nn.Sequential(
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.5)
        )

        self.classifier = nn.Linear(128, num_classes)
```

I used AdamW as an optimizer, and use learning scheduler starting with 5e-5.
I trained 200 epochs.

```
model = MultiHeadClassifier(conv5_dim, pool5_dim, conv6_dim, num_classes=len(set(label_list))).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=5e-5)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.9, patience=5, verbose=True)
```

If I had an extra time and GPU resource, my intention is to enhence the SoundNet model by incorporating deeper layers and dropouts, followed by retraining it. Moreover, I would explore the alternative activation functions, such as GeLU or SiLU, and consider employing a different optimizer.

Q2. What's the bottleneck of the current system?

1. **Pre-training Limitations:**
    - SoundNet is often used with pre-trained weights which were obtained from a specific training dataset (usually a large-scale dataset with various sounds and their corresponding images). The model might not perform as well when applied to different, particularly domain-specific, audio signals.
    - Transfer learning, while powerful, may not perfectly translate to all new tasks and adjustments or fine-tuning may be needed which can be computationally expensive.
2. **Model Architecture:**
    - The architecture of SoundNet may not be optimal for all kinds of audio signal processing tasks.
    - Some tasks may benefit from a different arrangement of layers or different types of layers, which may require a substantial amount of experimentation and computational resources to discover.
3. **Computational Resources:**
    - Training deep learning models, especially ones with additional layers or when using larger datasets, requires substantial computational resources (e.g., GPU power and memory).
    - In practical settings, the availability of computational resources could be a bottleneck for training and optimizing the SoundNet model.
4. **Data Requirement:**
    - Deep learning models typically require large datasets for training to achieve robust and generalized performance.
    - Acquiring and labeling such data can be labor-intensive and may also present a bottleneck, particularly in a domain-specific context where large labeled datasets are not readily available.
5. **Real-time Processing:**
    - Depending on the application, real-time processing might be required. The complexity of the SoundNet model, especially if additional layers are added, may introduce latency which can be a bottleneck in real-time applications.