Contextual word representations: Practical fine-tuning

Christopher Potts

Stanford Linguistics

CS224u: Natural language understanding

Guiding idea

1. Your existing architecture can benefit from contextual representations.

2. `finetuning.ipynb` shows you how to bring in Transformer representations:
   - Simple featurization
   - Fine-tuning

3. By extending existing PyTorch modules for this course, you can create *customized* fine-tuning models with just a few lines of code.

4. This is possible only because of the amazing work that the Hugging Face team has done!

Standard RNN dataset preparation

Embedding			
1	−0.42	0.10	0.12
2	−0.16	−0.21	0.29
3	−0.26	0.31	0.37

Examples

[a, b, a]
[b, c]

$\Downarrow$

Indices

[1, 2, 1]
[2, 3]

$\Downarrow$

Vectors

$\Big[[-0.42\ 0.10\ 0.12], [-0.16\ -0.21\ 0.29], [-0.42\ 0.10\ 0.12]\Big]$

$\Big[[-0.16\ -0.21\ 0.29], [-0.26\ 0.31\ 0.37]\Big]$

RNN contextual representation inputs

Examples

[a, b, a]
[b, c]

$$\Downarrow$$

Vectors

$$\Big[[-0.41\ -0.08\ 0.27], [0.17\ -0.22\ 0.78][-0.46\ 0.24\ 0.12]\Big]$$

$$\Big[[-0.02\ -0.56\ 0.11][-0.45\ 0.43\ 0.32]\Big]$$

Code snippet: BERT RNN inputs

```
[1]: import torch
     from transformers import BertModel, BertTokenizer
     import os
     from torch_rnn_classifier import TorchRNNClassifier
     import sst

[2]: SST_HOME = os.path.join("data", "sentiment")

[3]: weights_name = 'bert-base-cased'

[4]: bert_tokenizer = BertTokenizer.from_pretrained(weights_name)

[5]: bert_model = BertModel.from_pretrained(weights_name)

[6]: def bert_phi(text):
         input_ids = bert_tokenizer.encode(text, add_special_tokens=True)
         X = torch.tensor([input_ids])
         with torch.no_grad():
             reps = bert_model(X)
             return reps.last_hidden_state[0].squeeze(0).numpy()

[7]: def fit_prefeaturized_rnn(X, y):
         mod = TorchRNNClassifier(
             vocab=[],  # No notion of a vocab; the model deals only with vectors.
             early_stopping=True,
             use_embedding=False)  # Feed in vectors directly.
         mod.fit(X, y)
         return mod

[8]: experiment = sst.experiment(
         sst.train_reader(SST_HOME),
         bert_phi,
         fit_prefeaturized_rnn,
         assess_dataframes=sst.dev_reader(SST_HOME),
         vectorize=False)  # Pass in the BERT hidden states directly!
```

Simple custom models

[7]:
```
class TorchSoftmaxClassifier(TorchShallowNeuralClassifier):

    def build_graph(self):
        return nn.Sequential(
            nn.Linear(self.input_dim, self.n_classes_))
```

tutorial_pytorch_models.ipynb

Simple custom models

[14]:
```
class TorchDeeperNeuralClassifier(TorchShallowNeuralClassifier):
    def __init__(self, hidden_dim1=50, hidden_dim2=50, **base_kwargs):
        super().__init__(**base_kwargs)
        self.hidden_dim1 = hidden_dim1
        self.hidden_dim2 = hidden_dim2
        # Good to remove this to avoid confusion:
        self.params.remove("hidden_dim")
        # Add the new parameters to support model_selection using them:
        self.params += ["hidden_dim1", "hidden_dim2"]

    def build_graph(self):
        return nn.Sequential(
            nn.Linear(self.input_dim, self.hidden_dim1),
            self.hidden_activation,
            nn.Linear(self.hidden_dim1, self.hidden_dim2),
            self.hidden_activation,
            nn.Linear(self.hidden_dim2, self.n_classes_))
```

tutorial_pytorch_models.ipynb

Simple custom models

[24]:
```
class TorchLinearRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.input_dim = input_dim
        self.w = nn.Parameter(torch.zeros(self.input_dim))
        self.b = nn.Parameter(torch.zeros(1))

    def forward(self, X):
        return X.matmul(self.w) + self.b
```

tutorial_pytorch_models.ipynb

Simple custom models

```
[25]: class TorchLinearRegresson(TorchModelBase):
          def __init__(self, **base_kwargs):
              super().__init__(**base_kwargs)
              self.loss = nn.MSELoss(reduction="mean")

          def build_graph(self):
              return TorchLinearRegressionModel(self.input_dim)

          def build_dataset(self, X, y=None):
              """
              This function will be used in training (when there is a `y`)
              and in prediction (no `y`). For both cases, we rely on a
              `TensorDataset`.
              """
              X = torch.FloatTensor(X)
              self.input_dim = X.shape[1]
              if y is None:
                  dataset = torch.utils.data.TensorDataset(X)
              else:
                  y = torch.FloatTensor(y)
                  dataset = torch.utils.data.TensorDataset(X, y)
              return dataset

          def predict(self, X, device=None):
              """
              The `_predict` function of the base class handles all the
              details around data formatting. In this case, the
              raw output of `self.model`, as given by
              `TorchLinearRegressionModel.forward` is all we need.
              """
              return self._predict(X, device=device).cpu().numpy()

          def score(self, X, y):
              """
              Follow sklearn in using `r2_score` as the default scorer.
              """
              preds = self.predict(X)
              return r2_score(y, preds)
```

tutorial_pytorch_models.ipynb

Code: BERT fine-tuning with Hugging Face

[31]:
```
class HfBertClassifierModel(nn.Module):
    def __init__(self, n_classes, weights_name='bert-base-cased'):
        super().__init__()
        self.n_classes = n_classes
        self.weights_name = weights_name
        self.bert = BertModel.from_pretrained(self.weights_name)
        self.bert.train()
        self.hidden_dim = self.bert.embeddings.word_embeddings.embedding_dim
        # The only new parameters -- the classifier:
        self.classifier_layer = nn.Linear(
            self.hidden_dim, self.n_classes)

    def forward(self, indices, mask):
        reps = self.bert(
            indices, attention_mask=mask)
        return self.classifier_layer(reps.pooler_output)
```

Code: BERT fine-tuning with Hugging Face

[32]:
```
class HfBertClassifier(TorchShallowNeuralClassifier):
    def __init__(self, weights_name, *args, **kwargs):
        self.weights_name = weights_name
        self.tokenizer = BertTokenizer.from_pretrained(self.weights_name)
        super().__init__(*args, **kwargs)
        self.params += ['weights_name']

    def build_graph(self):
        return HfBertClassifierModel(self.n_classes_, self.weights_name)

    def build_dataset(self, X, y=None):
        data = self.tokenizer.batch_encode_plus(
            X,
            max_length=None,
            add_special_tokens=True,
            padding='longest',
            return_attention_mask=True)
        indices = torch.tensor(data['input_ids'])
        mask = torch.tensor(data['attention_mask'])
        if y is None:
            dataset = torch.utils.data.TensorDataset(indices, mask)
        else:
            self.classes_ = sorted(set(y))
            self.n_classes_ = len(self.classes_)
            class2index = dict(zip(self.classes_, range(self.n_classes_)))
            y = [class2index[label] for label in y]
            y = torch.tensor(y)
            dataset = torch.utils.data.TensorDataset(indices, mask, y)
        return dataset
```