BIG DATA ANALYSIS AND PROJECT

QUORA INSINCERE QUESTIONS CLASSIFICATION

QUESTION DESCRIPTION

- An existential problem for any major website today is how to handle toxic and divisive content. Quora wants to tackle this problem head-on to keep their platform a place where users can feel safe sharing their knowledge with the world.
- Quora is a platform that empowers people to learn from each other. On Quora, people can ask questions and connect with others who contribute unique insights and quality answers. A key challenge is to weed out insincere questions -- those founded upon false premises, or that intend to make a statement rather than look for helpful answers.
- In this competition, Kagglers will develop models that identify and flag insincere questions. To date, Quora has employed both machine learning and manual review to address this problem. With your help, they can develop more scalable methods to detect toxic and misleading content.
- Here's your chance to combat online trolls at scale. Help Quora uphold their policy of “Be Nice, Be Respectful” and continue to be a place for sharing and growing the world’s knowledge.

DEFINE THE TOXIC AND DIVISIVE CONTENT.

- An insincere question is defined as a question intended to make a statement rather than look for helpful answers. Some characteristics that can signify that a question is insincere:
  - Has a non-neutral tone
  - Is disparaging or inflammatory
  - Isn't grounded in reality
  - Uses sexual content (incest, bestiality, pedophilia) for shock value, and not to seek genuine answers

- The training data includes the question that was asked, and whether it was identified as insincere (target = 1). The ground-truth labels contain some amount of noise: they are not guaranteed to be perfect.

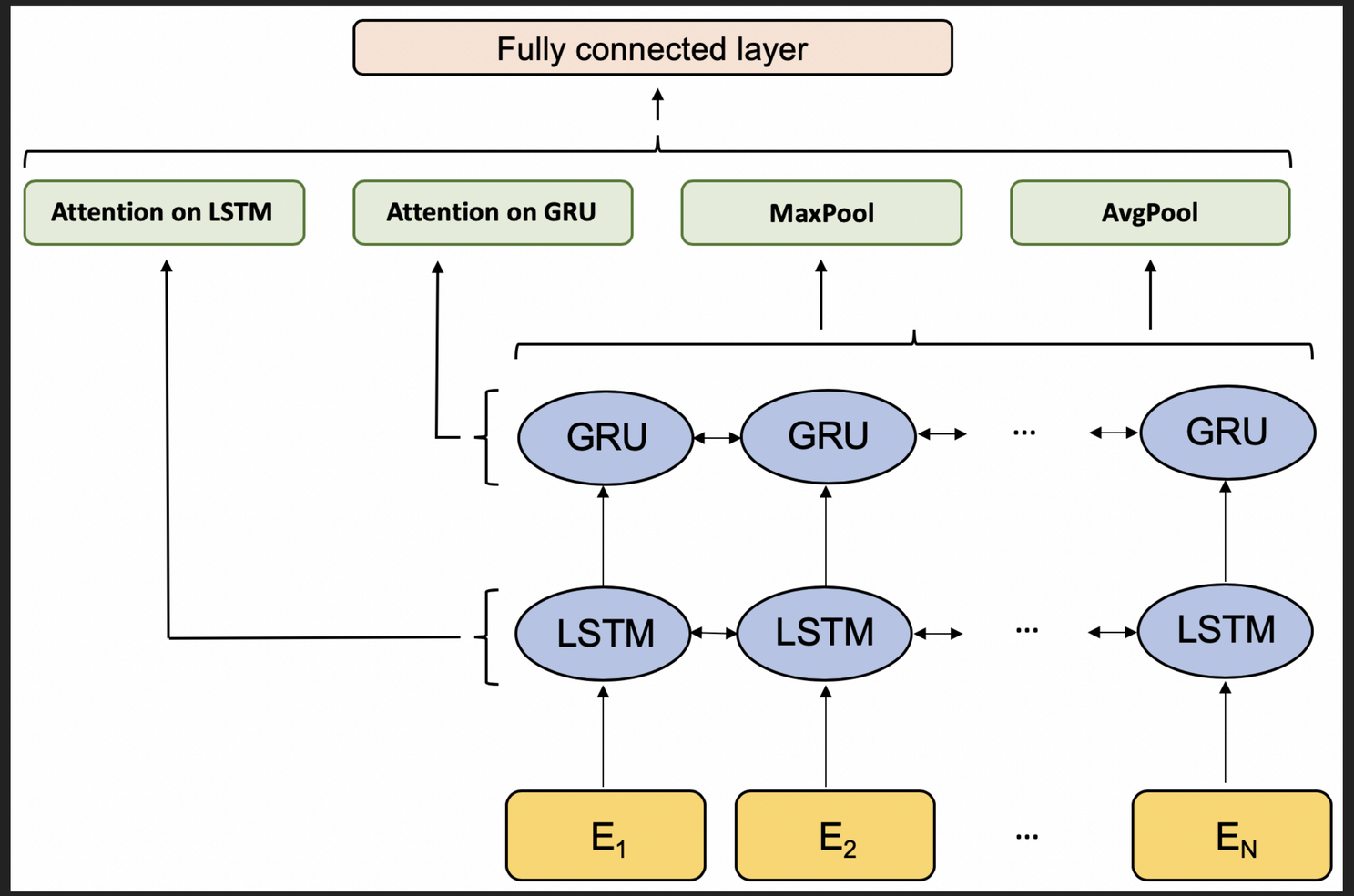
DATA

- The structure of the data is not complex, with a total of three columns of training data: qid, question_text and target (0-not insincere, 1-insincere). The training data was about 1.31 million. The organizers provided four kinds of word embeddings: GoogleNews, GloVe, Paragram, FastText.

MODLE STRUCTURE

- Bidirectional LSTM with attention on input sequence
- Bidirectional GRU with attention on LSTM output sequence
- Separate average-pool and max-pool layers on the GRU output sequence
- Linear layer (320, 16)
- ReLU layer over linear layer outputs
- Dropout layer with p = 0.1

MODLE STRUCTURE



EMBEDDING

- The official offers four embeddings for us to choose from.
- The most popular way is to using the mean of 4 embeddings. However, I find that using glove*0.6 + para*0.4 behaves better than using the average one.
- In fact, there should be a better proportion, but time is limit.

RNN： LSTM + GRU

- RNN can represent sequences of arbitrary length as fixed-length vectors while focusing on the structured properties of the input.
- The LSTM structure is designed to solve the gradient disappearance problem and is the structure of the first introduction mechanism.
- The complexity of the structure makes LSTM difficult to analyze, and the computational cost is also high. GRU is an alternative to LSTM. Similar LSTM, GRU is also based on the gate mechanism, but generally uses fewer doors and does not have separate memory components.

ATTENTION

- Allow the decoder to focus on different parts of the input sequence at each output.
- I ported the Attention mechanism many others have used in this competition to PyTorch. I am not sure where the Keras snippet originated from, so I am going to give credit to the kernel where I have first seen it.
- https://www.kaggle.com/bminixhofer/deterministic-neural-networks-using-pytorch

PRE-PROCESSING

- maxlen = 70
- max_features = 95000
- The symbol processing methods used in text preprocessing are as follows:

PRE-PROCESSING

```
puncts = [',', '.', '"', ':', ')', '(', '-', '!', '?', '|', ';', "'", '$', '&',
'/', '[', ']', '>', '%', '=', '#', '*', '+', '\\', '●', '~', '@', '£',
 '.', '_', '{', '}', '©', '^', '®', '`', '<', '→', '°', '€', '™', '›', '♥',
'←', '×', '§', '″', '′', 'Â', '■', '½', 'à', '…',
 '“', '★', '”', '–', '•', 'â', '►', '−', '¢', '²', '¬', '░', '¶', '↑', '±',
'¿', '▾', '═', '¦', '‖', '—', '¥', '▓', '—', '‹', '─',
 '▒', ': ', '¼', '⊕', '▼', '▪', '†', '▪', '’', '■', '¨', '▄', '♬', '☆', 'é',
'¯', '♦', '¤', '▲', 'è', '¸', '¾', 'Ã', '⋅', '‘', '∞',
 '•', ') ', '↓', '、', '|', ' (', '»', ',', '♪', '╩', '╚', '³', '．', '╦', '╣',
'╔', '╗', '━', '❤', 'ï', 'Ø', '¹', '≤', '‡', '√', ]
def clean_text(x):
    x = str(x)
    for punct in puncts:
        x = x.replace(punct, f' {punct} ')
    return x
```

PRE-PROCESSING

- In text preprocessing, some competitors replace words that are not in the word vector matrix, such as: {'Babchenko':' Arkady Arkadyevich Babchenko faked death',}.
- In fact, these may not be equivalent, so blunt word replacement work is not necessarily for the model. It is a wise decision to skip this step directly.

EMBEDDING

- Setting embedding_matrix [0] to full 0 vector is effective for getting a good score.

```
embedding_matrix = glove_embeddings*0.6 + paragram_embeddings*0.4
embedding_matrix[0] = np.zeros([300,])
```

EMBEDDING

- Use dropout after embedding will also increase the score.

```
class SpatialDropout(nn.Dropout2d):
    def forward(self, x):
        x = x.unsqueeze(2)
        x = x.permute(0, 3, 2, 1)
        x = super(SpatialDropout, self).forward(x)
        x = x.permute(0, 3, 2, 1)
        x = x.squeeze(2)
        return x
```

ATTENTION LAYER

```
class Attention(nn.Module):
    def __init__(self, feature_dim,
step_dim, bias=True, **kwargs):
        super(Attention,
self).__init__(**kwargs)

        self.supports_masking = True

        self.bias = bias
        self.feature_dim = feature_dim
        self.step_dim = step_dim
        self.features_dim = 0

        weight = torch.zeros(feature_dim,
1)
        nn.init.xavier_uniform_(weight)
        self.weight = nn.Parameter(weight)

        if bias:
            self.b =
nn.Parameter(torch.zeros(step_dim))
```

```
def forward(self, x, mask=None):
    feature_dim = self.feature_dim
    step_dim = self.step_dim

    eij = torch.mm(
        x.contiguous().view(-1, feature_dim),
        self.weight
    ).view(-1, step_dim)

    if self.bias:
        eij = eij + self.b

    eij = torch.tanh(eij)
    a = torch.exp(eij)

    if mask is not None:
        a = a * mask

    a = a / torch.sum(a, 1, keepdim=True) + 1e-10

    weighted_input = x * torch.unsqueeze(a, -1)
    return torch.sum(weighted_input, 1)
```

NEURAL NETWORK

- Now define the neural network. Defining a neural network in PyTorch is done by defining a class. This is almost as intuitive as Keras. The main difference is that you have one function (__init__) where it is defined which layers there are in the network and another function (forward) which defines the flow of data through the net.

NEURAL NETWORK

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

        hidden_size = 40

        self.embedding = nn.Embedding(max_features, embed_size)
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float32))
        self.embedding.weight.requires_grad = False

        self.embedding_dropout = SpatialDropout(0.1)
        self.lstm = nn.LSTM(embed_size, hidden_size, bidirectional=True, batch_first=True)
        self.gru = nn.GRU(hidden_size * 2, hidden_size, bidirectional=True, batch_first=True)

        self.lstm_attention = Attention(hidden_size * 2, maxlen)
        self.gru_attention = Attention(hidden_size * 2, maxlen)

        self.linear = nn.Linear(320, 16)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)
        self.out = nn.Linear(16, 1)
```

NEURAL NETWORK

```
def forward(self, x):
    h_embedding = self.embedding(x)
    h_embedding = self.embedding_dropout(h_embedding)

    h_lstm, _ = self.lstm(h_embedding)
    h_gru, _ = self.gru(h_lstm)

    h_lstm_atten = self.lstm_attention(h_lstm)
    h_gru_atten = self.gru_attention(h_gru)

    # global average pooling
    avg_pool = torch.mean(h_gru, 1)
    # global max pooling
    max_pool, _ = torch.max(h_gru, 1)

    conc = torch.cat((h_lstm_atten, h_gru_atten, avg_pool, max_pool), 1)
    conc = self.relu(self.linear(conc))
    conc = self.dropout(conc)
    out = self.out(conc)

    return out
```

TRAINING

- batch_size = 512
- n_epochs = 6

5-FOLD CROSS-VALIDATION

- First, define 5-Fold cross-validation. The `random_state` here is important to make sure this is deterministic.

```
splits = list(StratifiedKFold(n_splits=5, shuffle=True, random_state=10).split(x_train, y_train))
```

RESULTS AND FINDINGS

- Final score: 0.69178
- At first, I used the race more commonly used keras, but found that keras a fatal problem, the result can not repeat, and the keras speed is slow. At this point I decide to change the framework, pytorch is a very good choice.
- The problem lies within CuDNN. CuDNN's implementation of GRU and LSTM is much faster than the regular implementation but they do not run deterministically in TensorFlow and Keras. In this competition were speed is essential you can not afford to keep determinism by using the regular implementation of GRU and LSTM.

RESULTS AND FINDINGS

- In PyTorch, CuDNN determinism is a one-liner: torch.backends.cudnn.deterministic = True. This already solves the problem everyone has had so far with Keras. But that's not the only advantage of PyTorch. PyTorch is:
  - significantly faster than Keras and TensorFlow. Again, speed is important in this competition so this is great.
  - has a more pythonic API. I hate working with TensorFlow because there are seemingly tens of thousands of ways to do simple things. PyTorch has (in most cases) one obvious way and is by far not as convoluted as TensorFlow.
  - is executed eagerly. There is no such thing as an execution graph in PyTorch. That makes it much easier to try new things and interact with PyTorch in a notebook.
- Keras solves some of these problems with TensorFlow but it has a high-level API. I think that when doing research, it is often preferable to be able to interact with the model on a low-level. And you will see that the lower level API still doesn't make it complicated to work with PyTorch.

REFERENCE

- https://www.kaggle.com/suicaokhoailang/lstm-attention-baseline-0-652-lb
- https://www.kaggle.com/strideradu/word2vec-and-gensim-go-go-go
- https://www.kaggle.com/danofer/different-embeddings-with-attention-fork
- https://www.kaggle.com/ryanzhang/tfidf-naivebayes-logreg-baseline
- https://www.kaggle.com/bminixhofer/deterministic-neural-networks-using-pytorch