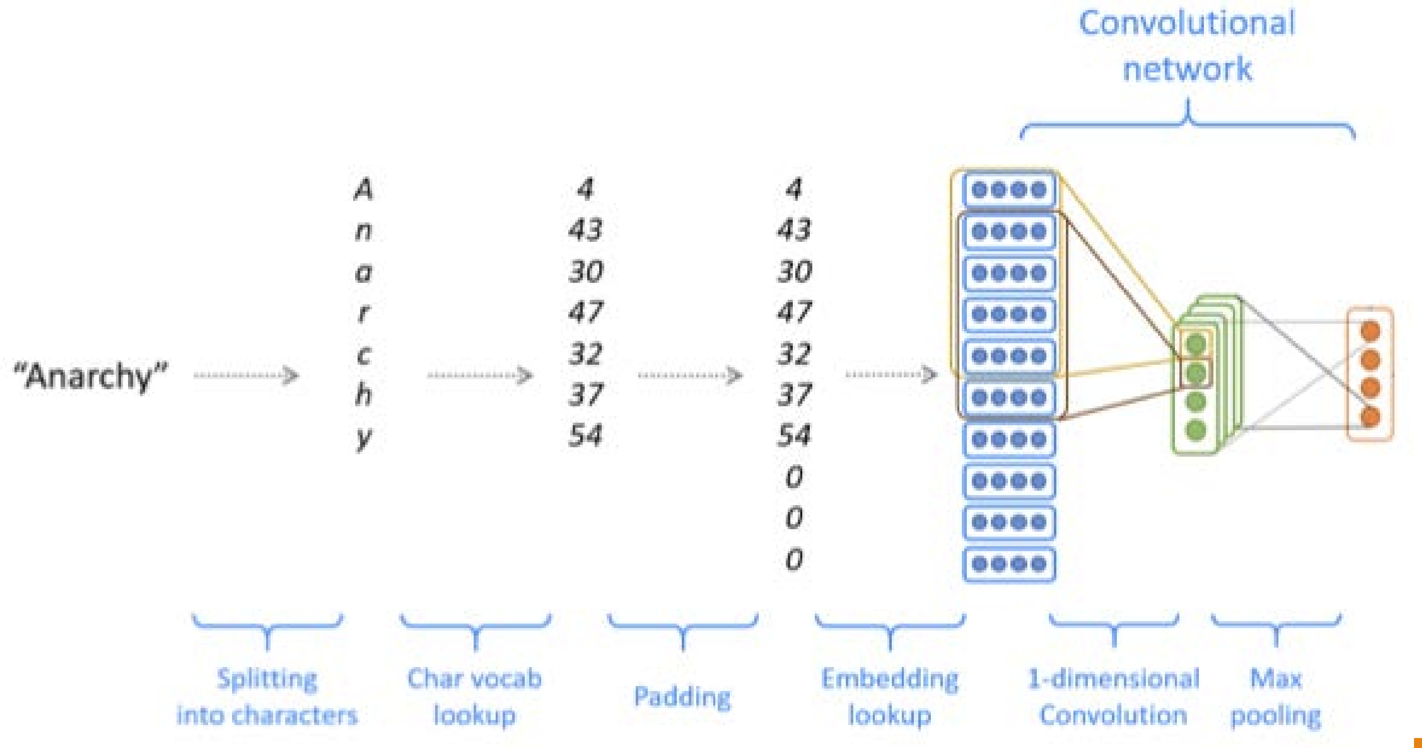
Deep Neural Networks for Natural Language Processing (AI6127)

JUNG-JAE KIM

TUTORIAL 7: SUBWORD MODELS

Question 1: Modify the CNN model of document classification of Tutorial 4 to use character embeddings of "Character-aware neural language models"

- Figure 1. Character-based convolutional encoder, which ultimately produces a word embedding of length eword.



Question 1: Modify the CNN model to use character embeddings of the method "Character-aware neural language models"

- **Convert word to character indices**:
- Assume we have a pre-defined 'vocabulary' of characters (e.g. all lowercase letters, uppercase letters, numbers and punctuations).
- By looking up the index of each character, we can represent the length-*l* word x as a vector of integers: $\boldsymbol{x} = [c_1, c_2, \cdots, c_l] \in \mathbb{Z}^l$
  - where each is c_i an integer index into the character vocabulary.

m_{word}: max. word len.
e_{char}: char. emb. size

Question 1: **Padding and embedding lookup**

- Using a special <PAD> 'character', we pad (or truncate) every word so that it has length m_{word} (pre-defined hyper parameter, representing maximum word length):
  - $\boldsymbol{x}_{padded} = [c_1, c_2, \cdots, c_{m_{word}}] \in \mathbb{Z}^{m_{word}}$
- For each of these characters c_i, we lookup a dense character embedding (which has shape e_{char}; $e_{char} = 50$). This yields a tensor $\boldsymbol{x}_{emb}$:
  - $\boldsymbol{x}_{emb} = \mathrm{CharEmbedding}(\boldsymbol{x}_{padded}) \in \mathbb{R}^{m_{word} \times e_{char}}$
- We will reshape $\boldsymbol{x}_{emb}$ to obtain $\boldsymbol{x}_{reshaped} \in \mathbb{R}^{e_{char} \times m_{word}}$ before feeding into the convolutional network of the Convolutional network below.
  - Necessary because PyTorch Conv1D performs only on the last dimension of the input

m_{word}: max. word len.
e_{char}: char. emb. size
k: kernel size
f: out. channel size

Question 1: **Convolutional network**

- To combine these character embeddings, we'll use 1-dimensional convolutions.
- The convolutional layer has two hyper-parameters:
  - the kernel size *k* (also called window size; *k*=5), which dictates the size of the window used to compute features, and
  - the number of filters *f* (also called number of output features or number of output channels)
  - Assume no padding is applied and the stride is 1
- The convolutional layer has a weight matrix $\boldsymbol{W} \in \mathbb{R}^{f \times e_{char} \times k}$ and a bias vector $\boldsymbol{b} \in \mathbb{R}^{f}$.

Question 1: Modify the model to use character embeddings of the method Character-aware neural language models

m_{word}: max. word len.
e_{char}: char. emb. size
k: kernel size
f: out. channel size

- To compute the i^{th} output feature (where $i \in \{1, . . . , f\}$) for the t^{th} window of the input, the convolution operation is performed between the input window $(\boldsymbol{x}_{reshaped})_{[:,t:t+k-1]} \in \mathbb{R}^{e_{char} \times k}$ and the weights $\boldsymbol{W}_{[i,:,:]} \in \mathbb{R}^{e_{char} \times k}$, and the bias term $b_i \in R$ is added:
  - $(\boldsymbol{x}_{conv})_{i,t} = \mathrm{sum}\left(\boldsymbol{W}_{[i,:,:]} \odot (\boldsymbol{x}_{reshaped})_{[:,t:t+k-1]}\right) + b_i \in \mathbb{R}$
- where $\odot$ is element-wise multiplication of two matrices with the same shape and sum is the sum of all the elements in the matrix. This operation is performed for every feature *i* and every window t, where $t \in \{1, \cdots, m_{word} - k + 1\}$. Overall this produces output $\boldsymbol{x}_{conv}$:
  - $\boldsymbol{x}_{conv} = \mathrm{Conv1D}(\boldsymbol{x}_{reshaped}) \in \mathbb{R}^{f \times (m_{word} - k + 1)}$

Question 1: Modify the model to use character embeddings of the method Character-aware neural language models

- For our application, we'll set *f* to be equal to e_{word}, the size of the final word embedding for word *x* (the rightmost vector in Figure 1). Therefore,
  - $\boldsymbol{x}_{conv} \in \mathbb{R}^{e_{word} \times (m_{word}-k+1)}$
- Finally, we apply the ReLU function to $\boldsymbol{x}_{conv}$, then use max-pooling to reduce this to a single vector $\boldsymbol{x}_{conv-out} \in \mathbb{R}^{e_{word}}$, which is the final output of the Convolutional Network:
  - $\boldsymbol{x}_{conv-out} = \mathrm{MaxPool}\big(\mathrm{ReLU}(\boldsymbol{x}_{conv})\big) \in \mathbb{R}^{e_{word}}$

m_{word}: max. word len.
e_{char}: char. emb. size
k: kernel size
f: out. channel size
e_{word}: word emb. size

Question 1: Modify the model to use character embeddings of the method Character-aware neural language models

- Here, MaxPool simply takes the maximum across the second dimension.
- Given a matrix $M \in \mathbb{R}^{a\times b}$, then $\mathrm{MaxPool}(M) \in \mathbb{R}^{a}$ with $\mathrm{MaxPool}(M)_i = \max_{1\le j\le b} M_{ij}$ for $i \in \{1, \cdots, a\}$
- This output is fed into the Convolutional Network of Q1.

Hands-on

Answer 1: Convert word to character indices

```
# The Vocabulary
class CharacterSequenceVocabulary(Vocabulary):
    def __init__(self, token_to_idx=None, pad_token="<PAD>"):
        super(CharacterSequenceVocabulary, self).__init__(token_to_idx)
        self._pad_token = pad_token
        self.pad_index = self.add_token(self._pad_token)
    def to_serializable(self):
        contents = super(CharacterSequenceVocabulary, self).to_serializable()
        contents.update({'pad_token': self._pad_token})
        return contents
    def lookup_token(self, token):
        return self._token_to_idx[token]
```

Answer 1: Convert word to character indices

```
# The Vectorizer
class NewsVectorizer(object):
    def __init__(self, title_char_vocab, category_vocab):
        self.title_char_vocab = title_char_vocab
        self.category_vocab = category_vocab

    def vectorize(self, title, max_seq_length,
max_word_length):
        words = title.split(" ")
        # cut off title words after max length
        if len(words) > max_seq_length:
            words = words[:max_seq_length]
        out_vectors = []
        for word in words:
            chars = list(word)
            # cut off word characters after max length
            if len(chars) > max_word_length:
                chars = chars[:max_word_length]
            # retrieve character embeddings
            char_indices =
                [self.title_char_vocab.lookup_token(token)
                 for char in chars]
            out_vector = np.zeros(max_word_length,
dtype=np.int64)
            out_vector[:len(char_indices)] = char_indices
            # fill up with <PAD> embeddings
            if len(char_indices) < max_word_length:
                out_vector[len(char_indices):] =
                    self.title_char_vocab.pad_index
            out_vectors.append(out_vector)
        if len(words) < max_seq_length:
            null_word_emb =
np.array([self.title_char_vocab.pad_index] *
max_word_length, dtype=np.int64)
            for _ in range(max_seq_length - len(words)):
                out_vectors.append(null_word_emb)
        out_vectors = np.array(out_vectors, dtype=np.int64)
        return out_vectors
```

Answer 1: Convert word to character indices

```
# The Vectorizer
    def from_dataframe(cls, news_df):
        …
        title_char_vocab =
          CharacterSequenceVocabulary()
        for title in news_df.title:
            for token in title.split(" "):
                title_char_vocab. \
                  add_many(list(token))

    def from_serializable(cls, contents):
        title_char_vocab =
CharacterSequenceVocabulary.from_serializable(contents['title_char_vocab'])
        category_vocab =
Vocabulary.from_serializable(contents['category_vocab'])
        return cls(
            title_char_vocab=title_char_vocab,
            category_vocab=category_vocab)

    def to_serializable(self):
        return {'title_char_vocab':
          self.title_char_vocab.to_serializable(),
                'category_vocab':
          self.category_vocab.to_serializable()}
```

Answer 1: Convert word to character indices

```
# The Dataset
class NewsDataset(Dataset):
    def __init__(self, news_df, vectorizer):
        …
        self._max_seq_length = max(map(measure_len, news_df.title))
        self._max_word_length = 0
        for title in news_df.title
            for token in title.split(" "):
                if len(token) > self._max_word_length:
                    self._max_word_length = len(token)
    def __getitem__(self, index):
        …
        title_vector = self._vectorizer.vectorize(row.title, self._max_seq_length,
            self._max_word_length)
        …
```

m_{word}: max. word len.

Answer 1: Convert word to character indices

```
# The Model: NewsClassifier
class NewsClassifier(nn.Module):
    def __init__(self, char_embedding_size, word_embedding_size, char_num_embeddings,
        word_num_channels, char_kernel_size, hidden_dim, num_classes, dropout_p,
        char_pretrained_embeddings=None, padding_idx=0):
        super(NewsClassifier, self).__init__()
        if char_pretrained_embeddings is None:
            self.char_emb = nn.Embedding(embedding_dim=char_embedding_size,
                                   num_embeddings=char_num_embeddings,
                                   padding_idx=padding_idx)
        else:
            char_pretrained_embeddings =
                torch.from_numpy(char_pretrained_embeddings).float()
            self.char_emb = nn.Embedding(embedding_dim=char_embedding_size,
                                   num_embeddings=char_num_embeddings,
                                   padding_idx=padding_idx, _weight=char_pretrained_embeddings)
```

e_{char}: char. emb. size

Answer 1: Convert word to character indices

```
# The Model: NewsClassifier
 def forward(self, x_in, apply_softmax=False):
    # x_in: (batch_size, max_seq_size, max_word_size)
    # x_emb: (batch_size, max_seq_size, max_word_size, char_embedding_size)
    x_emb = self.char_emb(x_in)
    batch_size = x_emb.size(dim=0)
    max_seq_size = x_emb.size(dim=1)
    max_word_size = x_emb.size(dim=2)
    char_embedding_size = x_emb.size(dim=3)
    # x_reshaped: (batch_size * max_seq_size, char_embedding_size, max_word_size)
    x_reshaped = x_emb.view(batch_size * max_seq_size, max_word_size,
      char_embedding_size).permute(0, 2, 1)
```

Answer 1: Convolutional network

```
# The Model: NewsClassifier
    def __init__(…):
        …
        self.char_convnet = nn.Sequential(
            nn.Conv1d(in_channels=char_embedding_size,
                out_channels=word_embedding_size,
                kernel_size=char_kernel_size),
            nn.ReLU())

    def forward(self, x_in, apply_softmax=False):
        # x_conv: (batch_size * max_seq_size, word_embedding_size, max_word_size-char_kernel_size+1)
        x_conv = self.char_convnet(x_reshaped)
        # x_conv_out: (batch_size * max_seq_size, word_embedding_size)
        word_embedding_size = x_conv.size(dim=1)
        remaining_size = x_conv.size(dim=2)
        x_conv_out = F.max_pool1d(x_conv, remaining_size).squeeze(dim=2)
```

m_{word}: max. word len.
e_{char}: char. emb. size
k: kernel size
f: out. channel size

Answer 1: This output is fed into the Convolutional Network of Q1

```
# The Model: NewsClassifier
def forward(self, x_in, apply_softmax=False):
    …
    features = self.word_convnet(x_conv_out.view(
        batch_size, max_seq_size, word_embedding_size).permute(0, 2, 1))
```

Answer 1: Training

```
# Settings and some prep work
args = Namespace(...,
    word_embedding_size=100, char_embedding_size=50,
    char_kernel_size=5, word_num_channels=100, ...)
# Initializations
args.use_glove = False
classifier = NewsClassifier(char_embedding_size=args.char_embedding_size,
                            word_embedding_size=args.word_embedding_size,
                            char_num_embeddings=len(vectorizer.title_char_vocab),
                            word_num_channels=args.word_num_channels,
                            char_kernel_size=args.char_kernel_size,
                            char_pretrained_embeddings=embeddings, ...)
```