Convolutional Neural Network for Natural Language Processing

(IMDB Reviews Rating)

This neural network defines CNN's model for NLP and trains it on a set of IMDB movie review data from PyTorch.

Definition of CNN for NLP

Convolutional neural networks (CNN) are a type of neural network commonly used for image classification tasks. However, they can also be used in natural language processing (NLP) for tasks such as text classification and language modeling. In NLP, CNN accepts a sequence of words as input (eg. sentence or document) and applies convolution to the inserted words using filters of different sizes. The output of the convolution layers is a set of flag maps, which then pass through a layer of maximum pooling to reduce their dimensionality. The resulting output is then inserted into a linear layer to form the final prediction.

CNN are suitable for NLP tasks where the local context of words is important, for example, when analyzing moods or classifying texts. They are also faster to train and more memory-efficient than more complex models such as RNN (recurrent neural networks), making them a good choice for larger files data.

CNN Functionality and Training

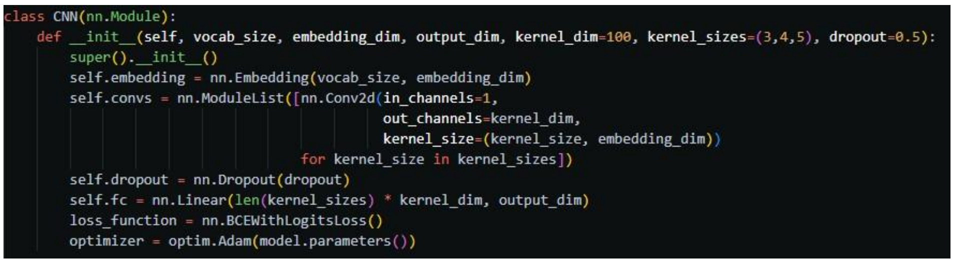
We use **the torchtext** package to easily load and process common NLP datasets. We use the **torchtext.datasets** to classify movie reviews as positive or negative**. IMDB.**

CNN consists of the following layers: • Embed layer: This layer converts input text into dense

fixed-size vectors.

- A set of convolution layers: These layers apply convolution to the embedded text using filters of different sizes. The output of these layers is a set of flag maps.
- Max pooling layer: This layer applies max pooling to the symptom maps, reducing their dimensionality.
- Omitting layer: This layer randomly erases some elements of the tensor to avoid over-fitting.
- Linear layer: This layer maps the output from the omission layer to the output dimensions of the model.

To train the model , we define the lossy funk (binary loss of cross-entropy) and the optimizer **(Adam).**



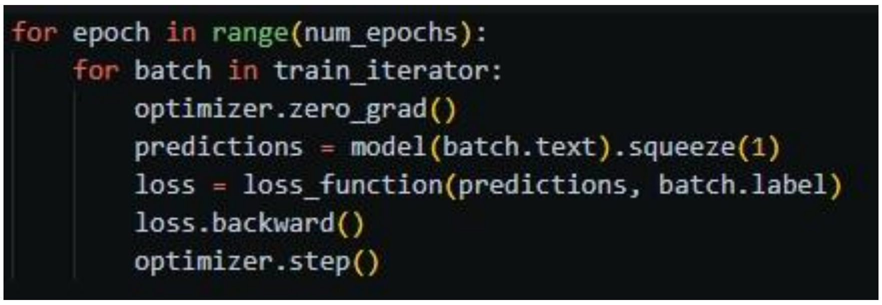
```
class CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, kernel_dim=100, kernel_sizes=(3,4,5), dropout=0.5):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.convs = nn.ModuleList([nn.Conv2d(in_channels=1,
                                              out_channels=kernel_dim,
                                              kernel_size=(kernel_size, embedding_dim))
                                    for kernel_size in kernel_sizes])
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(len(kernel_sizes) * kernel_dim, output_dim)
        loss_function = nn.BCEWithLogitsLoss()
        optimizer = optim.Adam(model.parameters())
```

Adam (Adaptive Moment Estimation) is an optimization algorithm that can be used to update the parameters of the neural network. It is a variant of stochastic gradient descent that uses moving averages of parameters to ensure an intermediate estimate of the second gross moments of the gradients; The term adaptive in the title refers to the fact that the algorithm
It "adjusts" the learning rates of each parameter based on historical information about the Nte gradie.

We then iterate through the trained data, calculate the loss, and update the model parameters using the reverse spread and optimizer. This process is repeated for each mini-dose in the trained data and for a specified number of epoch (i.e. transitions through the entire trained set). After training, the model is evaluated on the test set by calculating the accuracy of the predictions. Finally, we evaluate the model on the test set by calculating the accuracy.

It iterates the code through the trained data in mini-batches using an iterator and performs the following for each mini-batch:

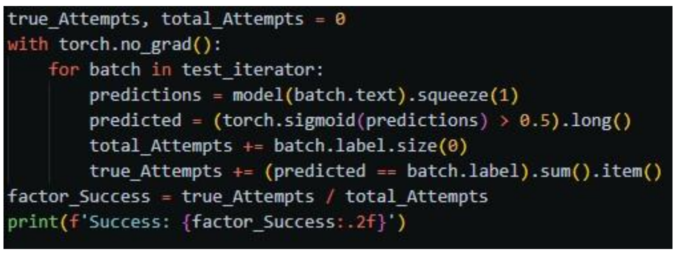
1. **optimizer.zero_grad()** - This sets the gradients of the model parameters to zero to prepare for the reverse transition.
2. **predictions = model(batch.text).squeeze(1)** - The model is applied to the mini-batch input **text** to create a set of predictions.
3. **loss = loss_**function**(predictions, batch.label)** - The loss function is applied to forecasts and actual labels to calculate the loss.
4. **loss.backward()** - The loss gradients relative to the parameters of the model are calculated using re-propagation.
5. optimizer.step**()** - Theop timalizer is used to update the parameters of the model based on the calculated gradients and the learning rate.



```
for epoch in range(num_epochs):
    for batch in train_iterator:
        optimizer.zero_grad()
        predictions = model(batch.text).squeeze(1)
        loss = loss_function(predictions, batch.label)
        loss.backward()
        optimizer.step()
```

The model is applied to the input text of the mini-batch to create a set of predictions. Forecasts shall be converted into labels (i.e. 0 or 1) using a threshold of 0,5. This is done by using the sigmoid function for predictions and by scoringing the resulting values.

The number of correct predictions is calculated by comparing the predicted labels with the actual labels and adding up the number of matches. The total number of predictions is the size of the minidas. The accuracy is then calculated as a proportion of the number of correct predictions and the total number of forecasts.



```
true_Attempts, total_Attempts = 0
with torch.no_grad():
    for batch in test_iterator:
        predictions = model(batch.text).squeeze(1)
        predicted = (torch.sigmoid(predictions) > 0.5).long()
        total_Attempts += batch.label.size(0)
        true_Attempts += (predicted == batch.label).sum().item()
factor_Success = true_Attempts / total_Attempts
print(f'Success: {factor_Success:.2f}')
```