Homework 4 Part 1

Bootcamp

Overview

- Train Recurrent Neural Networks to generate text with language modeling
- Regularization techniques for recurrent networks
- WikiText-2 Language Modeling Dataset
  - 33,278 sized vocabulary
  - The train and validation files each contain an array of articles. Each article is an array of integers, corresponding to words in the vocabulary. There are **579** articles in the training set
  - As an example, the first article in the training set contains 3803 integers. The first 6 integers of the first article are [1420 13859 3714 7036 1420 1417]. Looking up these integers in the vocabulary reveals the first line to be: = Valkyria Chronicles III = <eol>

What all do we have to implement?

1. LanguageModelDataLoader
2. LanguageModel
3. LanguageModelTrainer (Optimizer & Criterion)
4. TestLanguageModel

LanguageModelDataLoader

- __iter__
  - You can shuffle the dataset here before concatenating
  - Concatenate all the articles in our dataset
  - Calculate the number of batches based on concatenated size and batch size
  - At a single time-step, the input is a word, and the output (ground truth) is the next word
  - Iterate through the concatenated data, and yield (input, label) pairs
  - Ex. for sequence length 3 - INPUT: (I eat a), TARGET: (eat a banana)
  - You can do variable length BPTT here (covered later)

LanguageModel

- Embedding layer that maps vocab size dim input to a pre-defined dim vector (Refer the paper for recommended values of output dim)
- Implement LSTM layer(s), just as you have been doing so far
- Linear classifier which maps from LSTM hidden dim back to vocab size dim
- Keeping the shapes consistent allows you to use weight tying here between embedding layer and the linear classifier

LanguageModelTrainer

- We model the problem as a classification task, so which loss is appropriate?
- Define an appropriate optimizer
- Follow the standard forward pass, backward pass flow in *train_batch* method

TestLanguageModel

- In *prediction* method, you pass the given input through the model and return the **last time-step output**
- In the *generation* method, predict the next word given an input using the *prediction* method
- In a loop, use the previously generated output as the next input to your model, and generate the prediction for the next word
- Parameter *forward* will tell you the number of times you need to run this loop

Important Regularization Techniques

- Variable Length BPTT
- Locked and Embedding Dropout
- Weight Decay
- Weight Tying
- Activity Regularization
- Temporal Activity Regularization

Variable Length BPTT

- We are using a fixed sequence length for breaking up our data. (We saw sequence length 3 in our example)
- No matter how many times we iterate over the dataset, the first element in the sequence will never have anything to backprop into. So, given a sequence length of N, we are not utilizing the information from 1/N of the dataset - **Highly inefficient**
- To prevent this, randomly select sequence length instead of keeping it constant
- First select the base sequence length to be bptt (pre-defined sequence length) with probability p and bptt/2 with probability 1-p. Typically, use p = 0.95
- Additionally, the learning rate is scaled with the sequence length

Locked & Embedding Dropout

- Typically, we apply a different dropout mask to each example in the batch
- Applying a dropout on an RNN is dangerous. In the sense that the purpose of an RNN is to keep a memory of events over the long term. But classical dropout methods are not efficient since they create a noise that prevents these models from keeping a memory on the long term
- In Locked Dropout, the same dropout mask is applied across sequences, essentially we repeat the same dropout mask at each time step for inputs
- Embedding Dropout is equivalent to performing dropout on the embedding matrix at a word level, where the dropout is broadcast across all the word vectors embedding
- A good reference: https://towardsdatascience.com/12-main-dropout-methods-mathematical-and-visual-explanation-58cdc2112293

Activity & Temporal Activity Regularization

- L2-regularization is often used on the weights to reduce overfitting. L2 decay can also be used on the individual unit activations. This is termed activation regularization. AR penalizes activations that are significantly larger than 0 as a means of regularizing the network.

  loss = loss + alpha * rnn_h.pow(2).mean()

- In Temporal Activity Regularization, L2 decay is applied to the difference in outputs of an RNN at different time steps. It penalizes the model from producing large changes in the hidden state.

  loss = loss + beta * (rnn_h[1:] - rnn_h[:-1]).pow(2).mean()