DS-GA 1008: Deep Learning, Spring 2020
Homework Assignment 3

Tejaishwarya Gagadam, tg1779

`If you get, give. If you learn, teach. Maya Angelou (1928 - 2014)`

1. Fundamentals

1.1. Convolutions

For the following transformations from dimensionalities A to B, specify any combination of convolutions (with particular choices of kernels, stride), pooling, and other related modules that could get you from dimensionality A to B. [3 pts]

- Upsampling formula (with nearest neighbour): example with a factor 2 $(x, y, y) \rightarrow (x, 2y, 2y)$
- Deconvolution formula: (input to that layer - 1) x Stride - 2 x padding + (Deconv Kernel size - 1)
- Maxpooling forumal: (input to that layer - Max pool kernel size) / Stride + 1

1. A:$(64, 111, 111) \rightarrow$ B:$(4, 234, 234)$
   - Here we first perform Upsampling (using nearest neighbour) with a factor of 3 which gives $(64, 333, 333)$, then Deconvolution with 4 $(64, 7, 7)$ kernels and by zero padding the upsampled input by 3 (on each side), with stride 1 - which gives $(4, 332, 332)$, then Max-Pooling using kernel $(4, 99, 99)$ with stride 1 gives $(4, 234, 234)$
2. A:$(256, 56, 56) \rightarrow$ B:$(14, 126, 126)$
   - Here we first perform Upsampling (using nearest neighbour) with a factor of 3 which gives $(256, 168, 168)$, then Deconvolution with 14 $(256, 7, 7)$ kernels and by zero padding the upsampled input by 3 (on each side), with stride 1 - which gives $(14, 167, 167)$, then Max-Pooling using kernel $(4, 41, 41)$ with stride 1 gives $(14, 126, 126)$
3. A:$(256, 56, 56) \rightarrow$ B:$(42, 72, 72)$
   - Here we first perform Upsampling (using nearest neighbour) with a factor of 2 which gives $(256, 112, 112)$, then Deconvolution with 42 $(256, 7, 7)$ kernels and by zero padding the upsampled input by 3 (on each side), with stride 1 - which gives $(42, 111, 111)$, then Max-Pooling using kernel $(42, 39, 39)$ with stride 1 gives $(42, 72, 72)$

DS-GA 1008: Deep Learning, Spring 2020
Homework Assignment 3

Tejaishwarya Gagadam, tg1779

1.2. Dropout

Dropout is a very popular regularization technique.

(a) `torch.nn.Dropout2d` is the 2D dropout module in `torch.nn`.

(b) The following is a short explanation on **Dropout**:

- When we are dealing with Deep Neural Networks, we have a lot of moving parameters - which usually leads to over-fitting on the training set. We can observe this when we run the model on unseen/test data because the model cannot generalize on it. Dropout is one of the popular regularization methods that alleviates this.
- In Dropout, during the training phase, some of the co-adaptations learnt on the training data are randomly dropped. It takes a larger model, and repeatedly samples and trains smaller sub-models from it. Since we try to build different architectures every time we run through the data, it takes longer than usual - since the weight updates are noisy. Which is a trade-off between training time and over-fitting.
- Dropout can be implemented per-layer in a neural network, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer.
- It introduces a hyperparameter known as the dropout rate, which is the probability of training at a given node in a layer. That is if the rate was equal to 1 it means that there is no dropout, and conversely, 0 means no outputs from the layer.
- Dropout can result in the weights of the network to be larger. Hence, the weights are first scaled by the chosen dropout rate. Since Dropout is not used on the output layer, only on hidden/input layers.
- Using this regularization method on larger datasets, we can restrict over-fitting and improve the generalization error

1.3. Batch Norm

(a) **Mini-batch**:

- In the context of Deep Learning, mini-batch refers to a subset of the training data. Ideally, we want this mini-batch to represent our entire dataset in a much smaller scale. If we were dealing with a K Class Classification problem - we would want our min-batch to have a size of K or slightly more than K, which enforces this.

- When we are training our network, there are many optimization techniques we could use to reach the optimal condition. One of those is using the training data in mini-batches, where we only use this subset of the training data in that iteration.
- It usually results in a noisy path, which can be useful in over-shooting unwanted local optimas. Mini-batch approach converges faster because the weights are updates more frequently compared to full-batch approach.

(b) A short explanation on **Batch Normalization**:

- **Normalization** on input data is a way to bound the data. That is, we can fix a common scale for all the features so that we don't have inconsistencies between the them. If we have features that are in the range of 0.01 and 0.1, and another feature between 100 to 1000, we need to normalize the data to accelerate the training process. Similarly by extension, **Batch Normalization** does this on hidden layers. Since the parameters of these hidden layers are ever-changing, reducing the amount to which they can move around can help in faster learning.
- Stating the paper, Batch Normalization reduces the internal covariate shift, which in turn accelerates the training of a deep neural network. We achieve this by whitening the output of a previous activation layer - that is, we perform a linear transformation that eliminates the mean and gives us a unit variance. This will result in a fixed distribution which will reduce an internal covariate shift.
- Working of Batch Norm: In the case of mini-batch Stochastic Gradient Descent, where we consider a mini-batch of training samples. We will compute the mean and variance over this mini-batch.

$$\mu_B = \frac{1}{m}\sum_i x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_i (x_i - \mu_B)^2$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x_i} + \beta$$

where, μ_B is the mean of the mini-batch, σ_B^2 is the variance. We subtract each sample in the mini-batch with the batch mean, and divide this term with the batch standard deviation.

- Batch norm is useful to increase our learning process, make it more stable and improve the performance overall.

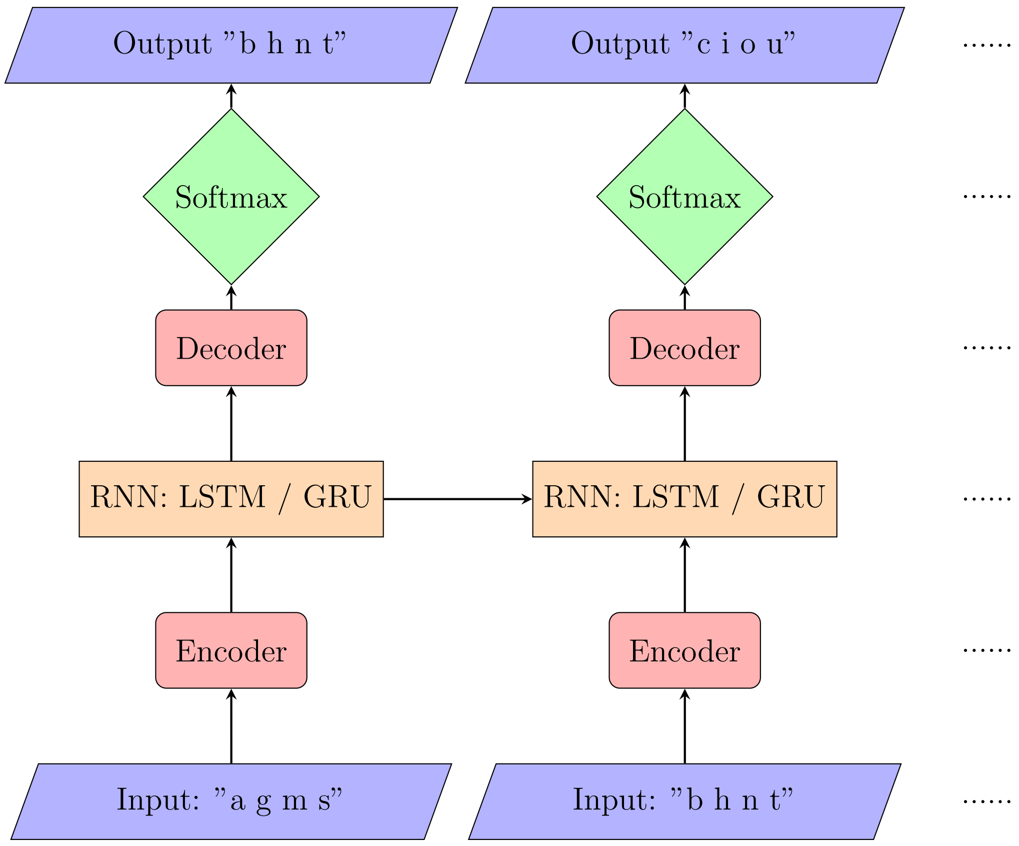
DS-GA 1008: Deep Learning, Spring 2020
Homework Assignment 3

Tejaishwarya Gagadam, tg1779

2. Language Modeling

This exercise explores the code from the `word_language_model` example in *PyTorch*.

(a) Flowchart:



- **Input:** We input the first row of the data, and enforce the model to produce the next row as the output - so that the model learns language predictions.
- **Encoder:** Here the words/letters are converted to vectors that the model can be trained with - as known as text embedding.
- **RNN:** The Recurrent Neural Network is where the training happens. It would comprise of either Long-Short Term Memory/ Gated Recurrent Unit, where the model carries the ouput of this layer to the next RNN. This is important for sequential data like languages.
- **Decoder:** Here, the embedded output from the RNN which is in the vector form is converted to a text.
- **Softmax:** Since language is a discrete form of data, we run a softmax on it. Which outputs the highest possible letters associated to that vector input. **This is essentially a part of the Decoder.**

- **Output:** This is the output, which will be the input of the next segment.

(b) Find and explain where and how the back-propagation through time (BPTT) takes place (you may need to delve into *PyTorch* source code). [5 pts]

- Back-propagation Through Time (BPTT) is used to calculate gradients in Recurrent Neural Networks (RNNs). `loss.backward()` calculates the partial derivatives of the loss function with respect to the input. However, unlike traditional feed-forward neural networks where these derivatives can be calculated all the way to the start, in RNNs we need to detach hidden states after every batch - to avoid vanishing/exploding gradient problems. That is the gradients will propagate through all possible connections and could either vanish or explode depending on the weight matrices. If we use a gated network like LSTM (Long-Short Term Memory) or GRU (Gated Recurrent Units), where some connections are blocked and others are open, the back propagation is more streamlined, which will prevent the gradients from vanishing or exploding. `clip_grad_norm` is used to clip the gradient norm to prevent the exploding gradient problem.

(c) Describe why we need the `repackage_hidden(h)` function, and how it works. [5 pts]

- `repackage_hidden(h)` is defined to detach previous hidden states from being carried over. `h.detach()` performs this action. We do this, to prevent the gradients from vanishing/exploding while back-propagating. This way the gradients cannot back-propagate all the way to the start.

(d) Why is there a `--tied (tie the word embedding and softmax weights)` option? [5 pts]

- This option exists to enable weight sharing. The weight matrix between the input to the embedding layer is shared with the output to the softmax layer. Weight sharing prevents overfitting in deep neural network architectures like these.
- As we learned, natural data has repeating patterns/motifs which is termed as stationarity, which is why sharing weight matrix across the data works.

(e) Compare LSTM and GRU performance (validation perplexity and training time) for different values of the following parameters: number of epochs, number of layers, hidden units size, input embedding dimensionality, BPTT temporal interval, and nonlinearities (pick just 3 of these parameters and experiment with 2-3 different values for each). [5 pts]

-

(f) Why do we compute performance on a test set as well? What is this number good for? [5 pts]

- The performance of the model on a test set gives us an estimate to how the model performs on unseen data. This is important because training a model using training data and computing the performance of the model on this train set, doesn't show the robustness of the model. When we run the same model on data that the model hasn't see before, we can fairly judge it's working.
- This will show how our model generalizes on new data. In some cases (especially in the case of Neural Networks which have a lot of parameters) the model over-fits on the training data, and develops very specific co-adaptations. This model will generalize poorly on data it hasn't seen, so measuring the performance on test data is key.

DS-GA 1008: Deep Learning, Spring 2020
Homework Assignment 3

Tejaishwarya Gagadam, tg1779

3. PyTorch

If you haven't already, install most recent versions of Python (3.6 or higher), PyTorch (we recommend using conda for the installation), and Jupyter.

Complete the programming exercises provided in the homework 3 directory of the course Google Drive folder (link).

4. Evaluation

Homework is worth a total of 100 points.

- Part 1 - 50 points
- Part 2 - 50 points

5. Submission

You are required to write up your solutions to Part 1 using markdown or LaTeX.
Please submit the homework on the NYU classes assignment page. Please upload the following:

- `First-name_Last-name_netID_A3.tex` (or `.md`) file for Part 1
- `First-name_Last-name_netID_A3.pdf` file for Part 1
- `First-name_Last-name_netID_A3_code.ipynb` file for Part 2
- `First-name_Last-name_netID_A3_code.pdf` file for Part 2
  (you can use Jupyter Notebook's "File → Download as → PDF" feature)

6. Disclaimers

You are allowed to discuss problems with other students in the class but have to write up your solutions on your own.
As feedback might be provided during the first days, the current homework assignment might be undergoing some minor changes. We'll notify you if this happens.