CSE 190 - HW 4

Xiuyuan Chen
xic145@ucsd.edu

Nikhilesh Sankaranarayanan
nsankara@ucsd.edu

Abstract

This paper describes the implementation and results of a Image Captioning Neural Network, built as a CNN-RNN model. The network trains on the Flick8k image dataset and generates sentences that describe a suite of test images.

Introduction

The neural network describe consists of two phases - an encoder and a decoder. The encoder is a Convolutional Network that extracts features from an input image. The decoder is a Recurrent Neural Network that decodes the features into a series of words, i.e. a caption for an image. The following sections detail the implementation and results of this network.

Methods

The main library used to design the networks was PyTorch. The dataset used is the Flickr8k dataset, and a custom Data Loader was used. The input is of the form of a list of image filenames, with 5 sentences that caption

each image. The Data Loader opens the images and converts them to PyTorch FloatTensors to pass into the Encoder CNN.

The training dataset was pooled from this dataset and comprises about 75% of the total dataset - the remaining is the validation set. Final performance is reported on a test dataset which is 25 images from the validation set.

The architecture of the network is as follows:

- An Encoder, which comprises of the pre-trained ResNet 152, with the output layer replaced by a Fully Connected Linear Layer that represents the feature vector of the input.
- An Embedding layer to connect it to the Decoder, and
- A Decoder, which is a Recurrent Neural Network built using the LSTM model from PyTorch. The output layer of the Decoder is a linear layer.

Regarding the architecture, we picked a linear final layer for the decoder instead of a softmax layer is because we find out that softmax layer was not training well. After replacing it with a linear layer the loss is reduced significantly.

The network is trained on 5 epochs, and used mini-batch sizes of 32 for the training set, and embedded layer of size 256 and the hidden layer in the RNN of size 3000. After every epoch, we validate our performance on the validation dataset.

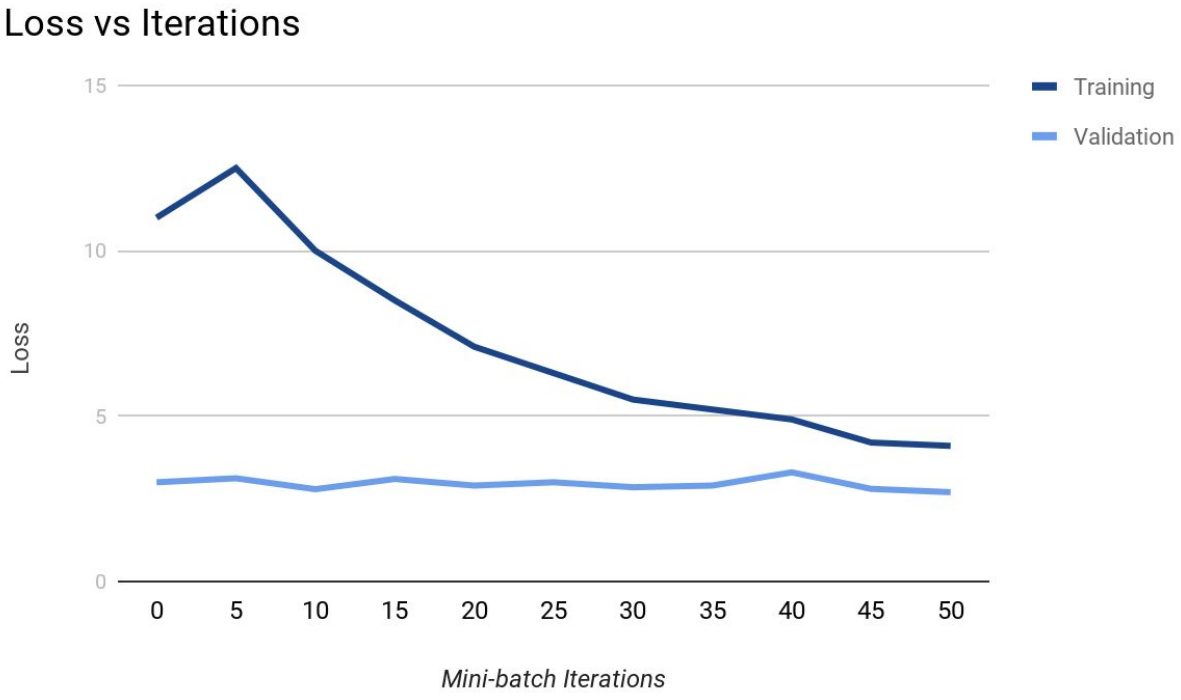
After the network is trained, we use the test data the generate sample image captions. During the testing phase, we don't concatenate features to the captions as the testing data should not have captions available. Then we pass the embedded feature vector to our network, use a sampling function that takes the highest 3 possible words, treat them as a probability distribution, generate a random number and then pick one of the 3 words as the input to the next timestep. Then we repeat this process until we have 20 words or we get an end token. This way, higher valued words get weighted more than others, to give us the correct word we need most of the time.

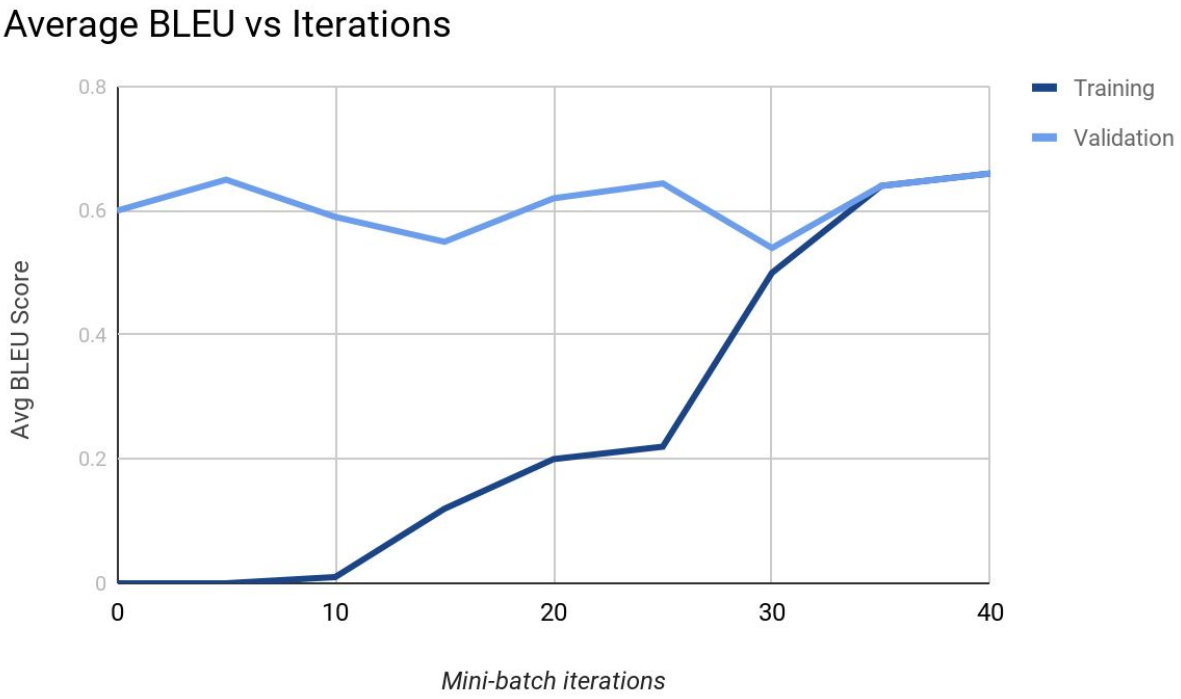
Results

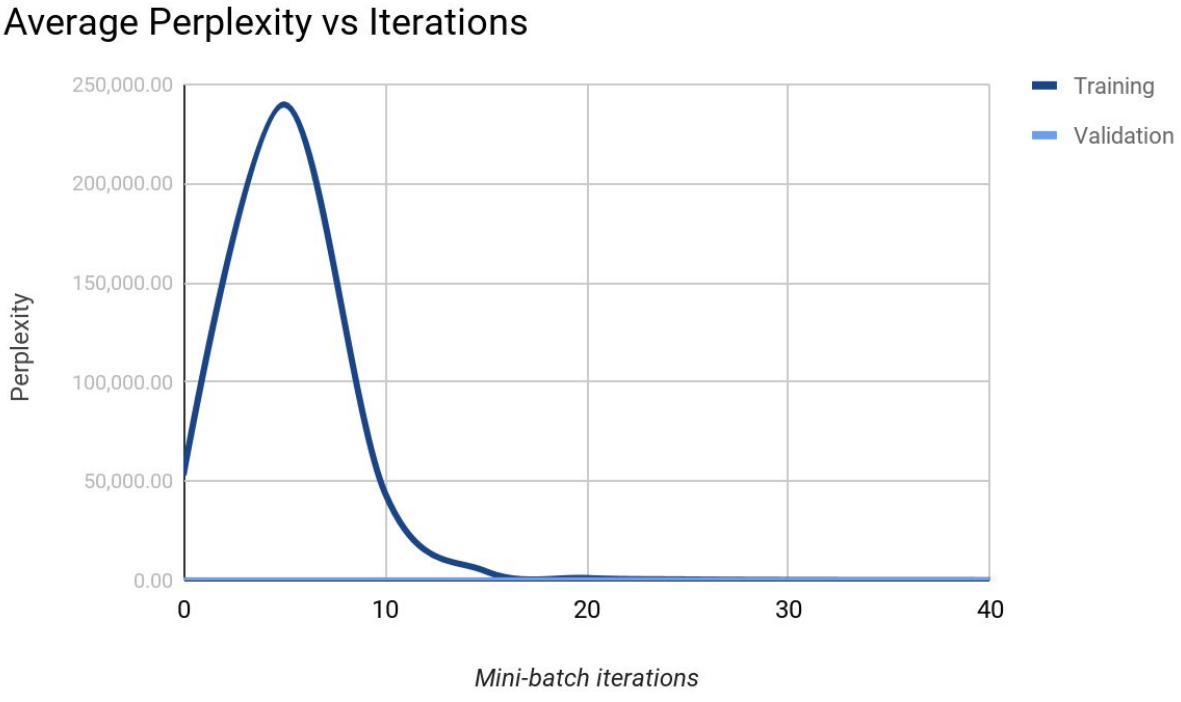
1. Using pre-trained CNN

   The network generally produces human-readable captions in full sentences. The final captions nearly always describe some features of the images. Some words may not match but the captions generally make sense and are grammatically correct sentences.

   Here are the following plots on training and validation sets.







Tabular values of the above graph:

	Training	Validation
0	53,091.00	19.61
5	239,913.31	22.87
10	43,928.69	16.42
15	4,588.30	22.30
20	1,225.23	18.22
25	524.07	21.54
30	258.32	17.54
35	185.20	18.22
40	136.11	27.88

Comments on plots:

The reason our validation loss and validation perplexity are always better is due to implementation reasons. We validate our network each epoch and what we are showing are the mini batches during validations. So they have the same performance. But just by looking at the loss (and other statistics) during training of the training data we know that our network is learning the patterns.

Generated Images and their Captions:

During the testing phase, we run our network using the testing data and generate captions.
The results are quite fantastic, and can be found in this album:

https://imgur.com/a/u0kIV

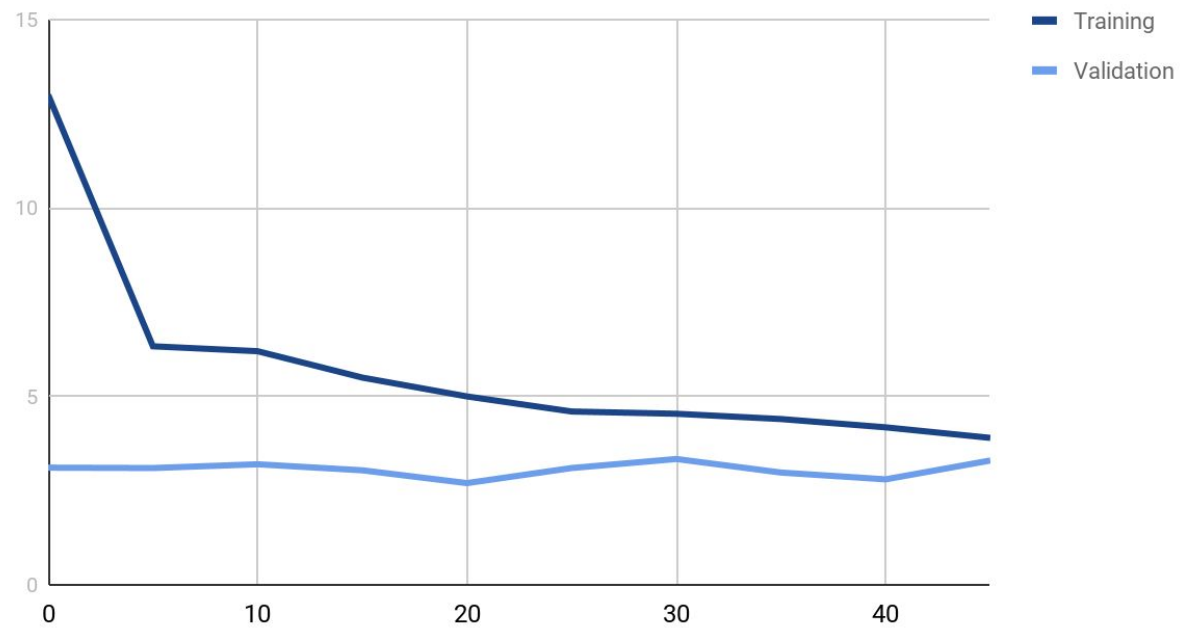
2. Fine-tuning the pre-trained network

For Fine Tuning our network, we could not train ResNet152 as the pods kept running out of memory. Therefore we switched to ResNet18. We STILL ran out of memory. So we reduced the embedding dimensions to 32 from 256. Finally, now the network runs without running out of memory. So the final outputs cannot be compared accurately with the original network's, and are likely of lower quality.

We set the pre-trained ResNet18 to be trained along with our modified CNN+RNN model. Setting the requires_grad = True flag in the ResNet, and adding its parameters to the parameters of the optimizer ensures that the ResNet network also gets trained. This network is significantly slower to train and therefore we only measure statistics for one epoch, with approximately 900 mini-batches of size 4 each.

Below is a plot for Loss over Mini-batch iterations:

Loss vs Iterations



Perplexity vs Iterations

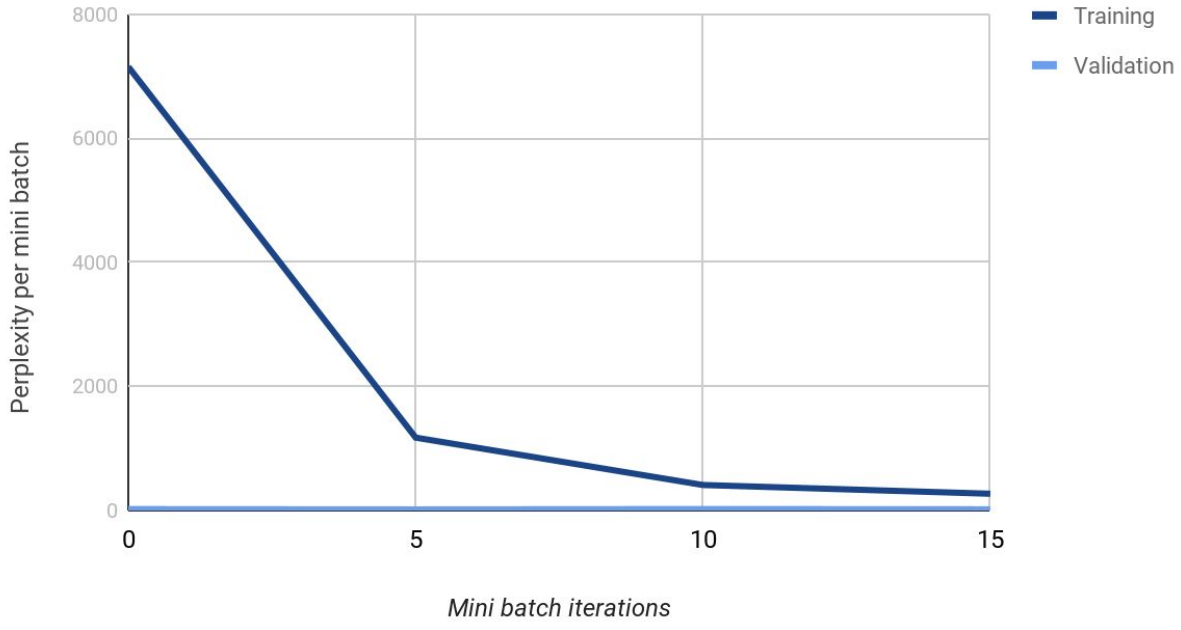


Table of the above values (values rounded to integers).

Minibatch iterations	Training	Validation
0	7155	22
5	1175	17
10	411	23
15	270	19
20	150	24
25	77	19
30	85	18
35	89	14
40	77	17

BLEU Score:

For the best caption we got a score of around 0.57914 which is pretty poor.

Comments on difference

The Fine tuned network, first off, is significantly slower. Structure of output captions are largely similar. However they make less sense and often contain keywords that have nothing to do with the image. Since we are using a shallower network, smaller embedding dimensions, and one single epoch, the network performs worse even though we finetune the pretrained network for our problem.

Teammate Contributions

Pair Programming was observed in every stage of the project.
The work was split evenly.

Xiyuan was responsible for deciding the architecture of the convolutional layers and constructing the Encoder and the embedding and hidden layers. He also implemented the iterative step to iterate over all mini of the program. He was also in charge of transportation and driving us to restaurants between coding sessions.

Nikhilesh worked on adapting the Data Loader and vocabulary to the network, and running the trained model on the test set and generating statistics and plots of loss, BLEU score and perplexity. Researched various blog posts and tutorials to get an idea of RNN implementation. Also in charge of getting food and finding places to eat between coding sessions.

References

- www.pytorch.org/docs
- www.pytorch.org/tutorials
- https://discuss.pytorch.org
- Lecture slides of Dr. Garrison Cottrell
- Pytorch tutorial by https://github.com/yunjey, reference only
- Bleu function and n-gram function by https://github.com/vikasnar, reference only
- Bleu score calculation from https://machinelearningmastery.com/calculate-bleu-score-for-text-python/