Assignment 1
COL775: Deep Learning. Semester II, 2023-2024.
Due Date: March 30, 2024

March 16, 2024

PART 1

1 ResNet over Convolutional Networks and different Normalization schemes

Residual Networks (ResNet) [5] present a very simple idea to introduce identity mappings via residual connections. They are shown to significantly improve the quality of training (and generalization) in deeper networks. We covered the core idea in class. Before starting this part of the assignment, you should thoroughly read the ResNet paper. Specifically, we will implement the ResNet [5] architecture, and study the effect of different normalisation schemes, viz. Batch Normalization [6], Instance Normalization [12], Batch-Instance Normalization [12], Layer Normalization [2], and Group Normalization [13] within ResNet. We will experiment with a dataset on Indian Birds species classification.

1.1 Image Classification using Residual Network

This sub-part will implement ResNet for Image Classification.
Dataset: link

1. You will implement a ResNet architecture from scratch in PyTorch. Assume that the total number of layers in the network is given by 6n+2. This includes the first hidden (convolution) layer processing the input of size 256×256. This is followed by n layers with feature map size 256×256, followed by n layers with feature map size 128×128, n layers with feature map size given by 64 × 64, and finally a fully connected output layer with r units, r being number of classes. The number of filters in the 3 sets of n hidden layers (after the first convoluational layer) are 16, 32, 64, respectively. There are residual connections between each block of 2 layers, except for the first convolutional layer and the output layer. All the convolutions use a filter size of 3 × 3 inspired by the VGG net architecture [10].

Whenever down-sampling, we use the convolutional layer with stride of 2. Appropriate zero padding is done at each layer so that there is no change in size due to boundary effects. The final hidden layer does a mean pool over all the features before feeding into the output layer. Refer to Section 4.2 in the ResNet paper for more details of the architecture. Your program should take n as input. It should also take r as input denoting the total number of classes.

2. Train a ResNet architecture with n = 2 as described above on the given dataset. Use a batch size of 32 and train for 50 epochs. For dataset, r = 25. Use SGD optimizer with initial learning rate of 10^{-4}. Decay or schedule the learning rate as appropriate. Feel free to experiment with different optimizers other than SGD.

3. Train data has 30,000 images while validation has 7500 images each image is of different resolution. Report the following statistics / analysis:

   - Accuracy, Micro F1, Macro F1 on Train, Val and Test splits.
   - Plot the error, accuracy and F1 (both macro and micro) curves for both train and val data

2 Impact of Normalization

The standard implementation of ResNet uses Batch Normalization [6]. In this part of the assignment, we will replace Batch Normalization with various other normalization schemes and study their impact.

1. Implement from scratch the following normalization schemes. They should be implemented as a sub-class of torch.nn.Module.

   (a) Batch Normalization (BN) [6]

   (b) Instance Normalization (IN) [12]

   (c) Batch-Instance Normalization (BIN) [7]

   (d) Layer Normalization (LN) [2]

   (e) Group Normalization (GN) [13]

2. In your implementation of ResNet in Section 1.1, replace the Pytorch's inbuilt Batch Normalization (nn.BatchNorm2d) with the 5 normalization schemes that you implemented above, giving you 5 new variants of the model. Note that normalization is done immediately after the convolution layer. For comparison, remove all normalization from the architecture, giving you a No Normalization (NN) variant as a baseline to compare with. In total, we have 6 new variants (BN, IN, BIN, LN, GN, and NN).

3. Train the 6 new variants on the dataset, as done in Section 1.1

4. As a sanity check, compare the error curves and performance statistics of the model trained in Section 1.1 with the BN variant trained in this part. It should be identical (almost).

5. Compare the error curves and performance statistics (accuracy, micro F1, macro F1 on train / val / test splits) of all six models.

6. **Impact of Batch Size:** [13] claim that one of the advantages of GN over BN is its insensitivity to batch size. Retrain the BN and GN variants of the model with Batch Size 8 and compare them with the same variants trained with Batch Size 128. Note that reducing the batch size will significantly increase the training time. To reduce the time taken, run it for 50 epochs and early stop based on validation accuracy.

7. **Visualize model's thinking:** An important part of any model training is analysing why a model predicts any particular label for a given input. In this sub-part we'll use Grad-CAM [9] for producing visual-explanations from our ResNet models. Grad-CAM computes the gradient of input feature map with respect to the specified output class. Higher the gradient value at particular position indicates it's importance for predicting particular class. For more information please read Grad-CAM paper[9]. For this assignment use the Grad-CAM's implementation from this package.

    - Consider following 7 classes for visualization: Cattle Egret , Coppersmith Barbet, Indian Peacock, Red Wattled Lapwing, Ruddy Shelduck, White Breasted Kingfisher, White Breasted Waterhen. Use the best performing model from all the variants trained above, visualize the gradient maps (superimposed on image) of 5 correctly and 5 incorrectly classified images for each of the above class from the validation set. Gradient needs to be computed with respect to the correct class and for the output feature map from last $2n$ layers (where feature map size is 64x64). These gradient maps need to be mean aggregated to give final 64x64 gradient map. The given package does all of these by default and you just need to mention the model layers.
    - Analyse the gradient maps and report your observations.

PART 2

Text to Math program

In Math Word Problem solving[3], given a mathematical problem specified in text, the goal is to find the solution to the problem by applying mathematical reasoning on the input text. See Figure 1 for an example.

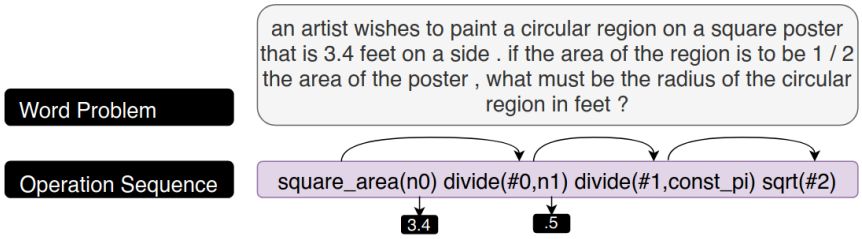

Figure 1: Sample data

Answering math word problems is a difficult task, and requires logical reasoning over implicit or explicit quantities expressed in text. In this assignment, our goal is develop an automatic *converter*, which would convert the textual narrative into executable program. One of the ways to generate executable programs from their textual description is to make use of **Encoder-Decoder** architecture. For this part of assignment, we will make use of seq2seq architectures to convert input text into the corresponding symbolic form. We will be using *operation representation language* provided in [1] to represent the executable program.

Note that training recurrent architectures like LSTMs can be slightly tricky. Therefore, look into various tricks that people have employed for this purpose. Some of them can be found in [11]. Try to optimize your training pipeline as much as possible.

2.1 Data

Dataset and checker file link
The folder contains the train, test and dev json files along with a checker file. Each json contains a $Program$, $linear\_formula$ and $answer$ fields for each example. The train split contains 19791 samples, dev contains 2961 and test contains 1924 samples. We would be generating the $linear\_formula$ given the $Problem$, the example of same is given in 1.

2.2 Metrics

Evaluation metrics are critical for any system to be evaluated. For this task we will use 2 different metrics.

- **Exact Match :** If all the predicted sequence matches the ground-truth sequence exactly it's 1 else 0.
- **Execution Accuracy :** If the executed answer of your predicted sequence is within 2% of true answer then the score is 1 else 0. You are provided

with the an evaluation script in the dataset. You must run this script on generated outputs to get the evaluation scores. The script will do the substitution of variables in the predicted sequence and will also execute the sequence to generate answer. Please read the checker file for more details.

2.3 Models

For your experiments you are supposed to train the following models.

2.3.1 Architectural experiments

1. A Seq2Seq model with GloVe embeddings [8], using an Bi-LSTM encoder and an LSTM decoder. For details of an LSTM refer [4]
2. A Seq2Seq+Attention model with GloVe embeddings, using an Bi- LSTM encoder and an LSTM decoder.
3. A Seq2Seq+Attention model using a pre-trained frozen BERT-base-cased encoder and an LSTM decoder
4. A Seq2Seq+Attention model with pre-trained BERT-base-cased encoder and an LSTM decoder, where the BERT encoder can now be fine-tuned along with the remaining network.

   You must report the following:

   - Loss curves on the training and dev set
   - Exact Match Accuracy and Execution Accuracy (see evaluation metrics) on the dev/test set.
   - All hyper-parameter settings used in your experiments.
   - Any findings observed from any potential grid search performed
   - Any other insights that you gained from the training procedure and the outputs of your model.

In this part, we will implement the models with **teacher forcing** with a ratio of **0.6** during training. During inference we will use **beam search** with **k = 10** to generate output. You must implement beam search in python/pytorch from scratch.

2.3.2 Effect of Teacher Forcing probability

Train the second model (Seq2Seq+Attention BiLSTM-LSTM) again with teacher forcing probabilities in $\{0.3, 0.6, 0.9\}$. Report the exact match accuracy and token match accuracy on test data with beam size $k = 10$. Comment your observations.

2.3.3 Effect of Beam Size

For the fourth model (Seq2Seq+Attention with fine-tuned Bert) trained with teacher forcing probability 0.6, generate the output for test data with beam sizes $\{1, 10, 20\}$. Report the exact match accuracy and token match accuracy on test data. Comment your observations.

References

[1] Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms, 2019.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[3] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[7] Hyeonseob Nam and Hyo-Eun Kim. Batch-instance normalization for adaptively style-invariant neural networks, 2019.

[8] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[9] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, October 2019.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[11] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[12] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization, 2017.

[13] Yuxin Wu and Kaiming He. Group normalization, 2018.