Assignment 1
COL 870: Deep Learning. Semester II, 2020-21.
Due Date: Wed April 15, 2021. 11:50 pm.

April 4, 2021

1 ResNet over Convolutional Networks and different Normalization schemes

Residual Networks (ResNet) [He et al., 2016] present a very simply 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 [He et al., 2016] architecture, and study the effect of different normalisation schemes, viz. Batch Normalization [Ioffe and Szegedy, 2015], Instance Normalization [Ulyanov et al., 2016], Batch-Instance Normalization [Nam and Kim, 2018], Layer Normalization [Ba et al., 2016], and Group Normalization [Wu and He, 2020] within ResNet. We will experiment with CIFAR 10 dataset as described in the ResNet paper.

1.1 Image Classification using Residual Network

This sub-part will implement ResNet for Image Classification.

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 32×32. This is followed by n layers with feature map size 32×32, followed by n layers with feature map size 16×16, n layers with feature map size given by 8×8, 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 [Simonyan and Zisserman, 2015]. 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 CIFAR 10 dataset. Use a batch size of 128 and train for 100 epochs. For CIFAR 10, $r = 10$. Use SGD optimizer with initial learning rate of 0.1. Decay or schedule the learning rate as appropriate. Feel free to experiment with different optimizers other than SGD.

3. The train data has $50,000$ images. Randomly select any $10,000$ of these as validation data. Use validation data for early stopping. NOTE: DO NOT use test data split at all during the training process. Report the following statistics / analysis:

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

1.2 Impact of Normalization

The standard implementation of ResNet uses Batch Normalization [Ioffe and Szegedy, 2015]. 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 nn.Module.

   (a) Batch Normalization (BN) [Ioffe and Szegedy, 2015]

   (b) Instance Normalization (IN) [Ulyanov et al., 2016]

   (c) Batch-Instance Normalization (BIN) [Nam and Kim, 2018]

   (d) Layer Normalization (LN) [Ba et al., 2016]

   (e) Group Normalization (GN) [Wu and He, 2020]

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 CIFAR 10 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 the six models.

6. **Impact of Batch Size:** [Wu and He, 2020] 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 100 epochs and early stop based on validation accuracy.

7. **Evolution of feature distributions:** Pretrained models are commonly used as feature extractors for transfer learning. In this sub-part, we will qualitatively analyze the features extracted from different variants of the ResNet model. Plot the 1^{st}, 20^{th}, 80^{th}, and 99^{th} quantile of the features as a function of the training epoch. You may compute these statistics over the val data at the end of each epoch. You may use the activations at the end of penultimate layer as features.

2 LSTM with Layer Normalization and CRF

2.1 NER Tagging with Bi-LSTM

Task: Recurrent architectures, like LSTM, are commonly used in sequence tagging tasks, like Named Entity Recognition (NER) in Natural Language Processing ([Lample et al., 2016]). In NER task, we are given a sentence as input, and the objective is to identify named entities in it, e.g., 'location', 'person' etc. Each word in the sentence is assigned a tag from a given set of NER tags. The words which are not a part of a named entity are classified as 'Other'.

Architecture: In this part of the assignment, we will closely follow the architectures proposed in section 2 of [Lample et al., 2016], but use a different dataset.

Dataset: We will use the publicly available GMB dataset[1]. It contains about 62 thousand sentences, 24 different NER tags. We have randomly split it into 60/20/20 train/dev/test sets respectively. The NER tags in the original dataset are fine-grained and contain a hierarchy, which we will not use in this assignment. After removing the hierarchy among NER tags (e.g., mapping 'person-title' and 'person-family-name' to a single tag 'person'), we are left with 9 high-level NER tags. We will use the BIO encoding for the labels in our modeling. Note that it is different from the tagging scheme used by Lample et al., which uses IOBES encoding. This link contains a short description of the BIO scheme.

[1] https://gmb.let.rug.nl/data.php

Download the cleaned version of the data from this link.
Each sentence is represented by a sequence of tokens, where a token represents the normalized form of each word (e.g., after stemming). Each line in the input file represents the details of a single token belonging to a sentence. Each line has 4 columns: token, POS Tag, original word, NER tag. We will ignore the POS tag for this assignment. Tokens appear (each in a new line) in the order they appear in a sentence. There is a new line separating the tokens belonging to two different sentences.

Word Embeddings: At each time step, the input to an LSTM model is a vector representing a token. One naïve way of converting a word (token) to a vector is to use a $|V|$ sized binary vector representation, where V is the entire vocabulary and each word is represented as a one-hot vector. However such a representation may not be very effective and will result in a very large recurrent matrix in the LSTM cell. Another alternative is to use a low dimensional dense representation of the words, called word embeddings. Here, each word is embedded in a low dimensional (typically 256 or less) continuous vector space. There are many advantages of using a low dimensional dense representation. You can read this tutorial for more details about word embeddings. In this assignment, we will use 100 dimensional word embeddings. We will experiment with both pre-trained word embeddings as well as randomly initialized ones.

1. **Simple Bi-LSTM with and without pre-trained word embeddings:** Train a simple bi-directional LSTM model for the task of NER tagging. Use randomly initialized 100 dimensional word embeddings and compare it with the same model trained using pre-trained 100 dimensional GloVe embeddings [Pennington et al., 2014]. You can download the pre-trained GloVe embeddings from this link. Use 100 dimensional hidden state for both forward and backward LSTMs. Note that, as in the original paper, the pretrained embeddings are fine-tuned during training. We have not yet covered LSTMs in class, but you can read about them in Section 10.10 of the deep learning book if you are starting early. You do not have to implement an LSTM cell - you can directly use the in-built implementation, i.e., you can use nn.LSTMCell module available in Pytorch. Use a batch size of 128 and train till the learning saturates. Use SGD optimizer with initial learning rate of 0.01 and gradient clipping at 5. As in the 1st part, decay or schedule the learning rate as appropriate and feel free to experiment with other optimizers. For regularization, use a dropout of 0.5 as in [Lample et al., 2016].

2. **Performance metrics:** Report the accuracy, micro F1 and macro F1 on train/dev/test splits. Plot various performance statistics (accuracy and micro/macro F1) for both the train and the val data, as a function of number of learning updates.

3. **Using Character Level Features:** To capture the orthographic features in the the words, [Lample et al., 2016] creates word embeddings

using information about characters in the corresponding word. In this sub-part, we will train a Bi-LSTM using embeddings created using character information (these are typically concatenated with pre-trained word embeddings). Refer to section 4.1 in Lample et al. Report the performance metrics as in Part (2) above. Also, compare the performance obtained in this case with that obtained using pre-trained word embeddings. Comment on your observations.

4. **Layer Normalization in LSTM:** Implement a variant of an LSTM cell with layer normalization built on top of it. Use layer normalized LSTM Cell to train a Layer normalized Bi-LSTM model for NER tagging. Compare the performance statistics with the un-normalized version. Use pre-trained word embeddings as well as char embeddings in this part.

2.2 Linear chain CRF

CRF is an effective way of modeling the relationship between the labels of the different words in a sentence. See section 2.2 in [Lample et al., 2016].

1. Implement a linear chain CRF layer in Pytorch. For the speed of training and efficiency, ensure that you do not loop over sentences in a mini-batch and use tensor operations effectively. Note that you will have to implement Viterbi algorithm for decoding and finding the best assignment of tags for a sentence.

2. Construct a BiLSTM-CRF model using a combination of your Bi-LSTM architecture and the CRF layer implemented above (see Lample et al.). Train this BiLSTM-CRF model for NER tagging using the dataset provided.

   Initialization: CRF layer is very sensitive to the initialization of the transition matrix. What would be an appropriate initialization for sequence tagging tasks?

   Loss function for CRF: You can't use standard cross entropy loss separately for each word with CRF layer. See Eqn 1 in [Lample et al., 2016] for the appropriate loss function to be used with CRF layer.

3. Report the accuracy, micro F1 and macro F1 on train/dev/test splits using your Bi-LSTM CRF model as done in the previous parts. Also plot various performance statistics (accuracy and micro/macro F1) for both the train and the val data, as a function of number of learning updates. How do these compare with those obtained using Bi-LSTM variations? Comment on your observations.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. 2016. URL `http://arxiv.org/abs/1607.06450`.

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 2016, Las Vegas, NV, USA, June 27-30, 2016, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 32nd International Conference on Machine Learning, ICML 2015, 1:448–456, 2015.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 260–270, San Diego, California, June 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1030. URL https://www.aclweb.org/anthology/N16-1030.

Hyeonseob Nam and Hyo-Eun Kim. Batch-instance normalization for adaptively style-invariant neural networks. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pages 2563–2572, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/018b59ce1fd616d874afad0f44ba338d-Abstract.html.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Empirical Methods in Natural Language Processing (EMNLP), pages 1532–1543, 2014. URL http://www.aclweb.org/anthology/D14-1162.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015. URL http://arxiv.org/abs/1409.1556.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance Normalization: The Missing Ingredient for Fast Stylization. (2016), 2016. URL http://arxiv.org/abs/1607.08022.

Yuxin Wu and Kaiming He. Group Normalization. International Journal of Computer Vision, 128(3):742–755, 2020.