MNIST Digits Classification with Pytorch

Background

In this homework, we still focus on MNIST digits classification problem. After implementing the details about MLP and CNN, you might know more about them. Different from implementing all the things from scratch, this time you will use Pytorch framework to implement neural networks. Pytorch provides good abstraction for different modules, and its auto-differentiation feature can save you from the backward details.

In addition, you should implement an important technique discovered recently: **batch normalization** [1], which aims to deal with the *internal covariate shift* problem. Specifically, during the training process, the distribution of each layer's inputs will change when the parameters of the previous layers change. Researchers propose to apply batch normalization of the input before entering into activation function of each neuron, so that the input of each mini-batch has a mean of 0 and a variance of 1. To normalize a value x_i across a mini-batch,

$$BN_{initial}(x_i) = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}},$$

where μ_B and σ_B denote the mean and standard deviation of the mini-batch across C dimension (2D input (N, C), 4D input (N, C, H, W)). ϵ is a small constant to avoid dividing by zero. The transform above might limit the representation ability of the layer, thus we extend it to the following form:

$$BN(x_i) = \gamma \cdot \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta,$$

where γ and β are learnable parameters. For instance, the original output of the hidden layer in MLP is

$$y = \sigma(WX + b).$$

After we normalize the input to activation function σ, the output can be represented as

$$y = \sigma(BN(WX + b)).$$

NOTE: When we test our samples one by one, the normalization may not work because we don't have *mini-batch* concept this time. Therefore, during training we should keep running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.9, which can be mathematically expressed as

$$\hat{\mu}_{new} = \texttt{momentum} \times \hat{\mu} + (1 - \texttt{momentum}) \times \mu_t,$$

where where $\hat{\mu}$ is the estimated statistic and μ_t is the new observed value. The similar update rule can also be applied to σ_B.

Requirements

Currently we use python version 3.5, Pytorch version 1.0.0

Python Files Description

`layer.py` includes `BatchNorm1d` and `BatchNorm2d` layers, which are used for MLP and CNN respectively. For CNN, you should also use `Reshape` layer before final `Linear` to transform 4D inputs to 2D.

NOTE: The initialization parts have been provided in two BN layers. You *only* need to implement their forward functions (backward function can be automatically constructed by Pytorch autograd engine). To implement BN's different behaviors when training and testing, a module class variable `self.training` is used to determine what current phase is. It can be set by calling model's `train()` and `eval()` methods. See [2] for more details.

`run_mlp.py` and `run_cnn.py` are the main programs. For data loading part, we use `torchvision.datasets.MNIST` as default. For optimization part, we use `torch.optim.SGD` to handle the weights update. You should adjust hyperparameters of optimizer (learning rate, weight decay, momentum) to properly train the model. For model construction part, we use `torch.nn.Sequential` class to sequentially define a model, like what we have done in previous homeworks. As for loss layer, we directly use `torch.nn.functional.cross_entropy` as instead. So do **NOT** include any loss layers during model definition. As for other layers, we directly use `torch.nn.Conv2d`, `torch.nn.ReLU`, `torch.nn.AvgPool2d`, `torch.nn.Linear` as instead.

Report

We perform the following experiments in this homework:

1. plot the loss value against to every iteration during training
2. Construct MLP (`Linear–BN–ReLU–Linear–BN–ReLU–Linear`) and CNN (`Conv–BN–ReLU–AvgPool–Conv–BN–ReLU–AvgPool–Reshape–Linear`) with batch normalization.
3. Construct MLP and CNN without batch normalization, and compare the result diferences. (you can discuss the difference from the aspects of training time, convergence, numbers of parameters and accuracy)

Bonus: Can you construct a mini-version of ResNet [3] with pytorch? Try to train it on MNIST and achieve as high accuracy as possible.

Submission Guideline:

You need to submit both report and codes, which are:

- **report**: well formatted and readable summary including your results, discussions and ideas. Source codes should *not* be included in report writing. Only some essential lines of codes are permitted for explaining complicated thoughts.
- **codes**: organized source code files with README for extra modifications or specific usage. Ensure that others can successfully *reproduce* your results following your instructions. **DO NOT include model weights/raw data/compiled objects/unrelated stuff over 50MB**

Deadline: May. 5th

TA contact info: Yulong Wang (王宇龙）, wang-yl15@mails.tsinghua.edu.cn

[1] Ioffe S, Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, In Proceedings of the International Conference on Machine Learning, 2015: 448-456.
[2] https://pytorch.org/docs/stable/nn.html#torch.nn.Module.eval
[3] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).