Data 598 (Winter 2022): Homework 3

1 The Effect of BatchNorm on a ConvNet

In this exercise, we will combine both the topics we covered in class this week. The goal of this exercise is to visualize the effective smoothness of a covolutional neural network with and without batch normalization.

Let $\varphi(\cdot; w) : \mathbb{R}^{28\times 28} \to \mathbb{R}^{10}$ denote a convolution neural network with parameters w which takes in an image of size 28×28 and returns a score for 10 output classes (All the MLPs and ConvNets we have considered so far fit this input-output description of φ, upto a reshaping of the images). Consider the objective function

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \varphi(x_i; w))$$

where ℓ is the multiclass logistic loss function. We define the **effective local smoothness** of this objective f at some w in a direction u as[1]

$$\widehat{L}(w; u) = \frac{\|\nabla f(w+u) - \nabla f(w)\|_2}{\|u\|_2}.$$

Observe that ∇f is the full gradient computed on the entire dataset. Here, we take u to be a minibatch stochastic gradient update at w, i.e.,

$$u = -\eta \frac{1}{B}\sum_{b=1}^{B} \nabla_w \ell(y_{i_b}, \varphi(x_{i_b}; w))$$

where $i_1, \cdots, i_B$ are random indices sampled from $\{1, \cdots, n\}$, B is the batch size and η is our learning rate.

Concretely, your task is as follows:

- Use the FashionMNIST dataset. Perform the same preprocessing as in previous homeworks.
- Code up a ConvNet module with two convolutional layers with the following structure (the input has 1 channel, so we write the image as $1 \times 28 \times 28$):

$$\begin{array}{c} \text{input} \\ 1\times 28\times 28 \end{array} \longrightarrow \begin{array}{c} \text{Conv} \\ k=5, \#\text{filters}=16, \text{padding=2} \end{array} \longrightarrow \text{ReLU} \longrightarrow \begin{array}{c} \text{MaxPool} \\ k=2 \end{array} \longrightarrow$$
$$\longrightarrow \begin{array}{c} \text{Conv} \\ k=5, \#\text{filters}=32, \text{padding=2} \end{array} \longrightarrow \text{ReLU} \longrightarrow \begin{array}{c} \text{MaxPool} \\ k=2 \end{array} \longrightarrow \begin{array}{c} \text{Linear} \\ \text{output}=10 \end{array}$$

  Here is how to interpret this specification:

  - k denotes the kernel/filter size and "#filters" denotes the number of filters

[1] Here, w represents all the parameters of a network concatenated into a single vector. Look at the hint at the end of this exercise for more details.

- In PyTorch, the convolutions and pooling operations on images are called "Conv2d" and "MaxPool2d" respectively.
- For the first conv layer, the specification asks you to use a kernel size of 5, and a padding of 2. The number of input channels is the same as the number of channels from the preceding layer (here, it is 1 since the preceding layer is just the image with 1 channel). Finally, the number of output channels is the same as the number of filters (here, 16). The second conv layer is constructed in a similarly; the number of input channels is the same as the number of outputs channels of the first conv layer (since ReLU and MaxPool do not change the number of channels). When not specified, we take the stride to be 1.
- The last "Linear" layer takes in the output of the second MaxPool and flattens it down to a vector of a certain size S. You are to figure out this size by running a dummy input through these layers and analyzing the output size, as we have done in the lab. The linear layer then maps this S-dimensional input to a 10-dimensional output, one for each class.

Hint: The ConvNet in the demo of week 2 matches this specification. You may use that code as a reference but you have to code up your own.

- Likewise, code up another ConvNet with a batch normalization layer inserted after each MaxPool. Note that use need to use "torch.nn.BatchNorm2d" for images, and this takes in the number of channels as an input.
- Train each ConvNet, with and without batch norm, with a learning rate of 0.04 and a batch size of 32 for 10 passes through the data.
- At the end of each pass, compute the effective local smoothness as we have defined it above.
- Make three plots: the train loss, the test accuracy and the effective local smoothness on the y-axis and the number of passes on the x-axis. Plot both models – with and without batch norm on the same plot.
- Comment how the local effective smoothness might explain why we can use larger learning rates with batch normalization. Recall for gradient descent on a L-smooth function, we can use a learning rate up to $1/L$.

Some hints:

- **Train/Eval Mode for BatchNorm 1**. Be careful with using the right mode for batch normalization – train or evaluation. You may use the lab as a reference.
- **Train/Eval Mode for BatchNorm 2**. In order to compute the effective local smoothness, compute the full gradient ∇f in evaluation model. However, compute the stochastic gradient for the direction u in train mode (since this is what the model will encounter during training).
- **Interpreting parameters as vectors**. We denote the parameters w or the direction u as vectors in $\mathbb{R}^d$. In code, these are stored as list of tensors (equivalently, lists of ndarrays), for instance via `list(model.parameters())`.

  For example, suppose parameters are represented by the list $[w_1, w_2]$ where

$$w_1 = \begin{pmatrix} 0.5 & 1 \\ 0 & -1 \end{pmatrix}, \qquad \text{and,} \qquad w_2 = \begin{pmatrix} 0 \\ -0.85 \end{pmatrix}.$$

  We can think of w as being obtained by concatenating the flattened components, e.g.,

$$w = \begin{pmatrix} 0.5 & 1 & 0 & -1 & 0 & -0.85 \end{pmatrix}^\top.$$

In this exercise, we only care about the ℓ_2 norm $\|w\|_2$ or a similar quantity for the gradients, which are also stored as a list of tensors. You may conveniently evaluate it (without explicitly constructing w) as

$$\|w\|_2 = \sqrt{\|\text{vec}(w_1)\|_2^2 + \|w_2\|_2^2},$$

where $\text{vec}(w_1)$ denotes w_1 flattened into a vector.

Note: we only run the above experiment for 10 passes because it is quite computationally expensive. In practice, we would run it until we observed interpolation.

2 (Bonus) Max pooling or convolution with stride?

In this exercise, we will compare two alternatives: convolution + max pooling, as considered in Exercise 1, versus a convolution with a stride greater than 1. Throughtout this exercise, we do **not** use batch norm. Consider the ConvNet

$$\begin{array}{c}\text{input}\\ 1\times 28\times 28\end{array} \longrightarrow \begin{array}{c}\text{Conv}\\ k=5, \#\text{filters}=16, \text{padding=2}, \text{stride=???}\end{array} \longrightarrow \text{ReLU} \longrightarrow$$

$$\longrightarrow \begin{array}{c}\text{Conv}\\ k=5, \#\text{filters}=32, \text{padding=2 stride=???}\end{array} \longrightarrow \text{ReLU} \longrightarrow \begin{array}{c}\text{Linear}\\ \text{output}=10\end{array}$$

Your tasks are as follows:

- Figure out the right stride so that the input to the second conv layer and the final linear layer are identical in shape to those in the previous exercise.
- Train each ConvNet, the one from Exercise 1 with a learning rate of 0.04 and a batch size of 32 for 10 passes through the data. The rest of the setup, including dataset preprocessing, is identical to Exercise 1.
- Make four plots: the train/test loss/accuracy. Plot both models on the same plot.
- Is there any difference in performance?
- Also report the average time per pass for each of the models. You may use Python's "time" package to keep track of the wallclock time.

Based on these observations, could you speculate why one of the two might be better or worse than the other?