Course: Deep Learning **([08/12/2021])**

Submission Assignment #[3]

Coordinator: Jakub Tomczak *Name:* James Bonello, Marton Fejer, Piotr Jastak*, Netid:* [jbo480], [mfr201], pjk600

1 Question answers

Question 1 *Write a pseudo-code for how you would implement this with a set of nested for loops. The convolution is defined by a set of weights/parameters which we will learn. How do you represent these weights?*

```
padded = pad(input_tensor)
filtered = zeros(batch_size, output_channels, output_width, output_height)

for instance in range(batch_size):
    for channel in range(0, input_channels - kernel.depth + 1, stride):
        for row in range(0, padded.height - kernel.height + 1, stride):
            for column in range(0, padded.width - kernel.width + 1, stride):
                value = 0
                for i in range(kernel.height):
                    for j in range(kernel.width):
                        for k in range(kernel.depth):
                            value += padded[channel + k][row + i][column + j] * kernel[k][i][j]
                filtered[instance][channel][row][column] = value
```

Question 2 *For a given input tensor, kernel size, stride and padding (no dilutions) work out a general function that computes the size of the output..*

The function output_shape() can be found in *utils.py* of the code.

Question 3 *Write a naive (non-vectorized) implementation of the unfold function in pseudocode. Include the pseudocode in your report.*

```
padded <- pad image with padding
batches = []
for image in range(batch_size):
    unfolded = []
    for row in range(0, image.height - kernel.height + 1, stride):
        for column in range(0, image.width - kernel.width + 1, stride):
            cross_section = []
            for channel in range(input_channels):
                for i in range(0, kernel.width):
                    for j in range(0, kernel.height):
                        append padded[channel][row + i][column + j] to cross_section
            append cross_section to unfolded
    append unfolded to batches
```

Question 4 *Work out the backward with respect to the kernel weights W.*

Let $\mathbf{X}$ be a $c \times n \times m$ matrix of inputs, $\mathbf{W}$ be a $j \times k \times l$ matrix of kernel weights. Let $\mathbf{X}'$ be the unfolded version of $\mathbf{X}$ and $\mathbf{W}'$ be the unfolded version of $\mathbf{W}$ with regards to a given padding and stride, such that we can calculate the unfolded outputs as:

$$\mathbf{Y} = \mathbf{X'}^T\mathbf{W}' \tag{1.1}$$

First, let us consider a 2D version of the convolution (that is, taking only a single channel out of c input channels and a single convolution kernel out of j output channels, or, let $c = 1$ and $j = 1$), where each row

of the output corresponds to the consecutive element of the output matrix after folding. For example, given a kernel

$$\mathbf{M} = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \tag{1.2}$$

and a matrix of inputs

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \tag{1.3}$$

, we have the unfolded versions (stride = 1, padding = 0):

$$\mathbf{M}' = \begin{pmatrix} M_{11} \\ M_{12} \\ M_{21} \\ M_{22} \end{pmatrix}, \mathbf{A}' = \begin{pmatrix} A_{11} & A_{12} & A_{21} & A_{22} \\ A_{12} & A_{13} & A_{22} & A_{23} \\ A_{21} & A_{22} & A_{31} & A_{32} \\ A_{22} & A_{23} & A_{32} & A_{33} \end{pmatrix} \tag{1.4}$$

Taking partial derivatives of $\mathbf{Z} = fold({\mathbf{A}'}^T\mathbf{M}')$:

$$\frac{\partial \mathbf{Z}}{\partial M'_{11}} = \frac{\partial Z_{11}}{\partial M_{11}} + \frac{\partial Z_{12}}{\partial M_{11}} + \frac{\partial Z_{21}}{\partial M_{11}} + \frac{\partial Z_{22}}{\partial M_{11}} \tag{1.5}$$

$$= \frac{\partial (A_{11}M_{11} + A_{12}M_{12} + A_{21}M_{21} + A_{22}M_{22})}{\partial M_{11}} + \frac{\partial (A_{12}M_{11} + A_{13}M_{12} + A_{22}M_{21} + A_{23}M_{22})}{\partial M_{11}} \tag{1.6}$$

$$+ \frac{\partial (A_{21}M_{11} + A_{22}M_{12} + A_{31}M_{21} + A_{32}M_{22})}{\partial M_{11}} + \frac{\partial (A_{22}M_{11} + A_{23}M_{12} + A_{32}M_{21} + A_{33}M_{22})}{\partial M_{11}} \tag{1.7}$$

$$= A_{11} + A_{12} + A_{21} + A_{22} \tag{1.8}$$

Factoring in the element-wise multiplication by the matrix of gradients w.r.t. $\mathbf{Z}$, and repeating the derivation for each element of $\mathbf{M}$ we get:

$$\frac{\partial L}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial M_{11}} = \delta_{11}A_{11} + \delta_{12}A_{12} + \delta_{21}A_{21} + \delta_{22}A_{22} \tag{1.9}$$

$$\frac{\partial L}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial M_{12}} = \delta_{11}A_{12} + \delta_{12}A_{13} + \delta_{21}A_{22} + \delta_{22}A_{23} \tag{1.10}$$

$$\frac{\partial L}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial M_{21}} = \delta_{11}A_{21} + \delta_{12}A_{22} + \delta_{21}A_{31} + \delta_{22}A_{32} \tag{1.11}$$

$$\frac{\partial L}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial M_{22}} = \delta_{11}A_{22} + \delta_{12}A_{23} + \delta_{21}A_{32} + \delta_{22}A_{33} \tag{1.12}$$

, where δ_{ij} are the entries of the matrix of gradients ($\frac{\partial L}{\partial \mathbf{Z}}$), which we assume is given by the previous steps of backpropagation. Note that we can unfold the matrix of gradients with the same parameters, and take its transpose, so that

$${\mathbf{A}'}^T unfold(\frac{\partial L}{\partial \mathbf{Z}}) \tag{1.13}$$

gives us the entries of the matrix of backwards of $\mathbf{M}$ as a column vector. In other words, knowing the parameters of the unfold function, we can calculate the backward of kernel weights $\mathbf{M}$ as:

$$\mathbf{M}^{\nabla} = fold({\mathbf{A}'}^T unfold(\frac{\partial L}{\partial \mathbf{Z}})) \tag{1.14}$$

Going back to the case of a convolution with multiple output channels and multiple input channels, we note that, since the columns of $\mathbf{W}'$ would each correspond to a separate output channel, we can generalize the 2D approach in a straightforward manner, yielding:

$$\mathbf{W}^{\nabla} = fold({\mathbf{X}'}^T unfold(\frac{\partial L}{\partial \mathbf{Y}})) \tag{1.15}$$

(Note that, in order for the dimensions to match, we would need to pad the unfolded weight matrices/gradient matrices with 0s corresponding to the parts of channels which are not multiplies. In other words, if column 1 corresponds to output channel 1, we need to put 0s to multiply out parts of the unfolded input which correspond

to other channels, i.e. if we have a kernel of size 2×2, 3 input and 3 output channels, the weight matrix $\mathbf{M}$ after unfolding would take the form:

$$M_{unfolded} = \begin{pmatrix} M_{111} & 0 & 0 \\ M_{112} & 0 & 0 \\ M_{121} & 0 & 0 \\ M_{122} & 0 & 0 \\ 0 & M_{211} & 0 \\ 0 & M_{212} & 0 \\ 0 & M_{221} & 0 \\ 0 & M_{222} & 0 \\ 0 & 0 & M_{311} \\ 0 & 0 & M_{312} \\ 0 & 0 & M_{321} \\ 0 & 0 & M_{322} \end{pmatrix} \tag{1.16}$$

Question 5 *If the input to the convolution is not the input to the network, but an intermediate value, we'll need gradients over the input as well. Work out the backward with respect to the input X.*

Question 6 *Implement your solution as a PyTorch Function. (If you didn't manage question 5, just set the backward for the input to None).*

Answer The code for the implementation of the solution can be found in the file `Conv2DFunc.py`, attached to this report.

Question 7 *Use the dataloaders to load both the train and the test set into large tensors: one for the instances, one for the labels. Split the training data into 50 000 training instances and 10 000 validation instances. Then write a training loop that loops over batches of 16 instances at a time.*

Answer: It is trivial to implement the dataloaders using PyTorch. However, it is important to note that the splitting of the training data had to be done prior to this since the dataloader objects are actually iterators over the data. With this being said, we have three dataloaders: a *training* one, a *validation one*, and a *testing* one. The dataloader and training loop implementation can be viewed on *part2.py* of the code.

Question 8 *Build this network and tune the hyperparameters until you get a good baseline performance you are happy with. You should be able to get at least 95% accuracy. If training takes too long, you can reduce the number of channels in each layer.*

Answer: The core hyperparameter tuned for the network is the *learning rate* set to 0.001. As specified on the assignment document, the network uses Adam for optimization and cross-entropy for the loss function. After 1 epoch, the network achieved an accuracy of 98%. The full implementation of this network, including compilation of the resultant accuracy, can be viewed on *part2.py* of the code.

Question 9 *Add some data augmentations to the data loader for the training set. Why do we only augment the training data? Play around with the augmentations available in torchvision. Try to get better performance than the baseline. Once you are happy with your choice of augmentations, run both the baseline and the augmented version on the test set and report the accuracies in your report.*

Answer: First of all, we only augment the training data because we are interested in creating more varied examples so we may get better training performance. The test data should not be augmented because it is the real, unseen data that is vital for measuring the true performance of our model. We can look at this from another point of view: if we can assume that the augmented targets are still correct, then we are implicitly assuming that we know the distribution of the target data, which is essentially what we are trying to learn, and the modeling step via a neural network would then be completely redundant.

Multiple augmentations were tested to see if the performance can be improved on top of the baseline. In fact, a visualizer script, *part2_aug_visualizer.py* was created to get a better idea of what a particular augmentation is really doing to the data. The augmentations were easily implemented by calling *transform.Compose* and passing

the desired augmentation transforms as a list. Amongst the augmentations tested were *random perspective*, *random affine*, *random rotation*, and also a few auto-augmentation policies (eg. CIFAR10. While no augmented model seemed to outperform the baseline model, the *random rotation* was observed to equal the baseline performance at times. The angle of 30 degrees was found to be optimal for this transform after some trial-and-error. The accuracy achieved for the baseline model was once again observed at 98%, while the augmented model achieved an accuracy of 97%.

Question 10 *Assume we have a convolution with a 5x5 kernel, padding 1 and stride 2, with 3 input channels and 16 output channels. We apply the convolution to an image with 3 channels and resolution 1024x768. What are the dimensions of the output tensor? What if we apply the same convolution to an image with 3 channels and resolution 1920x1080? Could we apply the convolution to an image with resolution 1920x1080 and 8 channels?*

Answer:

1. For an image with 3 input channels and a resolution of 1024×768, the output dimensions are $(16, 511, 383)$.
2. For an image with 3 input channels and a resolution of 1920×1080, the output dimensions are $(16, 959, 539)$.
3. We cannot apply the convolution to an image with 8 channels, as the dimensions of the input channels of the convolution and the input channels of the image don't match. In order for us to be able to apply the convolution, we would need a convolution with 8 input channels. (stride, padding and kernel width and height are irrelevant)

Question 11 *Let x be an input tensor with dimensions (b, c, h, w). Write the single line of PyTorch code that implements a global max pool and a global mean pool. The result should be a tensor with dimensions (b, c).*

Answer: The global max pool calculation can be implemented with

```
lambda t: torch.amax(t, (3,2))
```

the global mean pool calculation can be implemented with

```
lambda t: torch.mean(t, (3,2))
```

Putting those together, we can calculate a tuple of global max- and mean-pools with the function:

```
lambda t: torch.amax(t, (3,2)), torch.mean(t, (3,2))
```

Question 12 *Use an ImageFolder dataset to load the data, and pass it through the network we used earlier. Use a Resize transform before the ToTensor transform to convert the data to a uniform resolution of 28x28 and pass it through the network of the previous section. See what kind of performance you can achieve.*

Answer: The implementation can be found in *q12_13.py* of the code. The achieved performance is still quite high (95%), but lower than without the resize transform. However, this does not take into account the different datasets used. Perhaps in some instances, rather than resizing to a smaller size, the image could be cropped (wherever there is no information present, i.e. all zeros) to further improve performance.

Question 13 *We could, of course, resize everything to 64x64. Apart from the fact that running the network would be more expensive, what other downsides do you see?*

Answer: Resizing images (to a larger size) requires some form of interpolation, thus the image and its lower-level features might be distorted and depending on the type of interpolation, information may also be lost (whereas reducing size may also lead to information loss).

Question 14 *Load the data into a memory as three tensors: one for each resolution. Then write a training loop with an inner loop over the three resolutions. In a more realistic scenario, we could have a dataset where every almost every image has a unique resolution. How would you deal with this situation?*

Answer: The implementation can be found in *q14.py* of the code. Rather than pre-define which size sets are present, the code finds all resolutions and creates a dataloader for each of these (for train, validation and test sets) In a situation where all images have unique sizes, two things can be done in general. First, one might rather do a stochastic gradient descent over the set of images. Another option would be to take a random batch of images and zero-pad each image to match the size of the largest image in that batch.

Question 15 *Note that if we set N=64, as we did for the fixed resolution network the last linear layer has fewer parameters here than it did in the first one. Either by trial and error, or through computing the parameters, find the value of N for which both networks have roughly the same number of parameters (this will allow us to fairly compare their performances).*

Answer: Note that we can easily calculate the number of parameters within our model by implementing the below code. The fixed model has 29066 parameters in total.

```
no_of_params = 0
for param in net.parameters():
    no_of_params += np.prod(np.array(list(param.shape)))

print('Number of parameters: ' + str(no_of_params))
```

Once we get the parameters for the network, we can use trial-and-error to compute N such that the number of parameters for the variable network are the closest to the fixed one. We found that setting $N = 81$ gave us the closest number at 29029 parameters.

Question 16 *Compare the validation performance of global max pooling to that of global mean pooling. Report your findings, and choose a global pooling variant.*

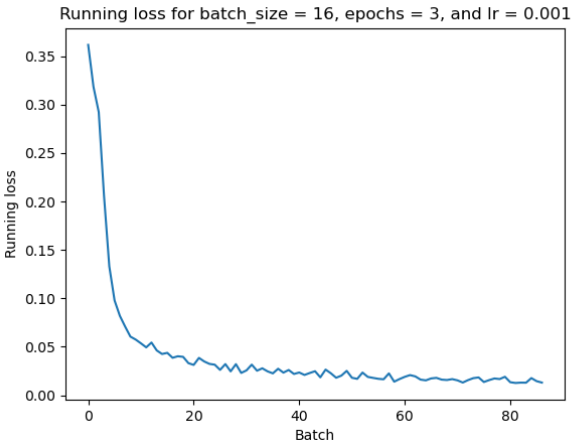
Answer: For this experiment, we ensure fairness by maintaining a batch size of 16 for both max and mean pool settings. We also set the number of epochs to 3 this time. This was done to ensure that both settings converge before we compare their performance. It is clear from Figure 1 and Figure 2 that global max-pooling performs better than global mean-pooling. Both running loss and accuracy for global max pooling seem to converge quicker than for global mean pooling. In fact, the max pooling achieved an accuracy of 98% while mean pooling achieved an accuracy of 95%. With this being said, the global max pooling model will be chosen for question 17. Please note that every point on the x-axis represents a batch checkpoint every 100 batches over all the epochs. Lastly, the code that was used to compare both networks can be found on *q14.py*.

Question 17 *Tune both networks (without data augmentation), and then compare the test set performance of the fixed-resolution network with 64x64 input resolution to the performance of the variable resolution network. Report your findings.*

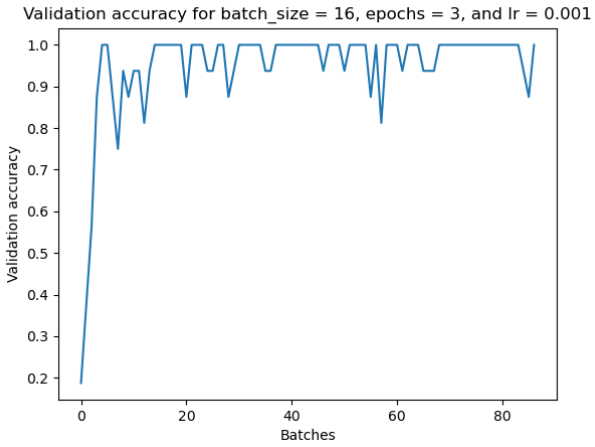
Answer: For question 17, there was some custom interpretation required, and we understood the question as follows: *Use the ImageFolder dataset on the fixed network that resizes varying resolutions to 28x28 (built for question 12), and on the variable network built for question 14 - and compare the performance achieved.*

The code that was used to compute the performances can be found on *q12_q13.py* and *q14.py*. The global max pooling network (with variable N), was not tuned further since it already achieved a high accuracy of 98%. On the other hand, the fixed network for question 12 was tuned by increasing the number of epochs to 3 because over 1, the accuracy was relatively low at 92%. The learning rate was set to 0.001 and batch size was kept at 16. As can be observed on Figure 3, the fixed network running loss seems to converge slower than the variable network as on Figure 1. In fact, the fixed network achieved an accuracy of 96%, meaning that the variable network seems to outperform the fixed network for varying resolutions. Indeed, we keep in mind that the fixed network is resizing the resolution to 28x28. On the other hand, the variable network accuracy is the same as the fixed network of part 2 which is trained on fixed resolution images.

Appendix

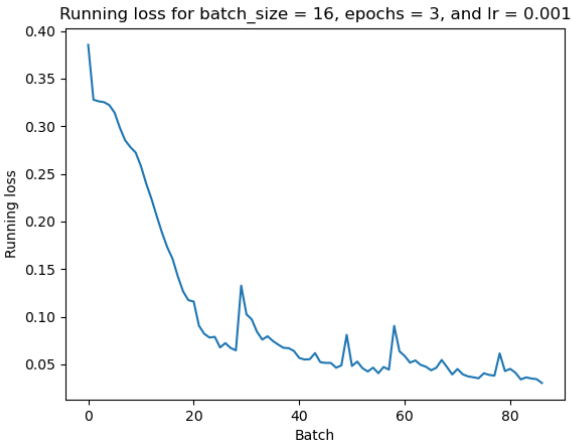


(a) Running loss for max pool

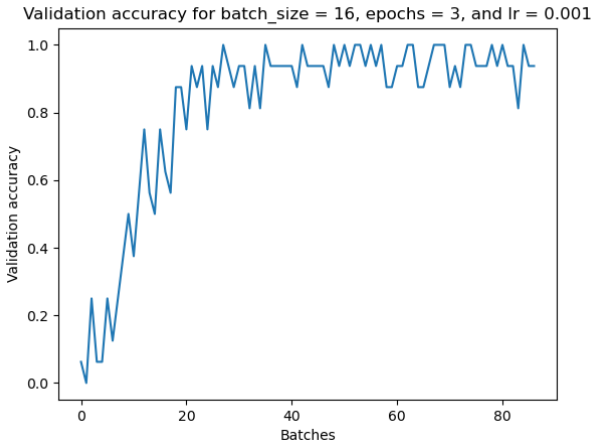


(b) Validation accuracy for max pool

Figure 1: Max pooling results

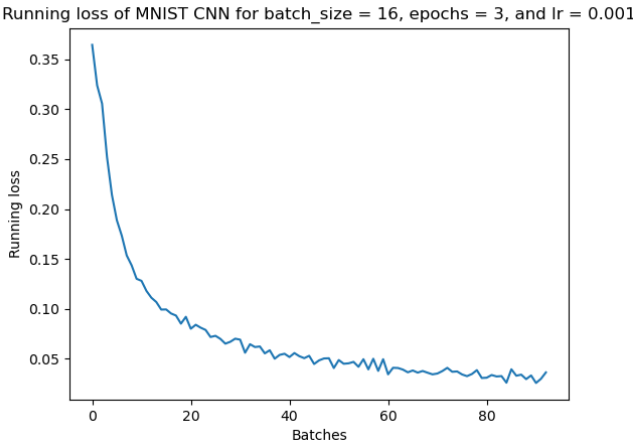


(a) Running loss for mean pool

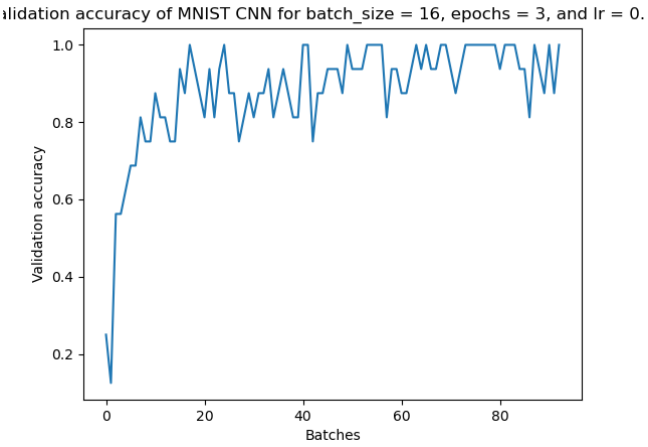


(b) Validation accuracy for mean pool

Figure 2: Mean pooling results



(a) Running loss for mean pool



(b) Validation accuracy for mean pool

Figure 3: Mean pooling results