Course: Deep Learning **(8 December 2021)**

Submission Assignment #3A

Name: Bruna Aguiar Guedes, Enrico Calleris, Romy Vos *VUNetID:* bas630, ecs820, rvs249

1 Introduction

Convolutional Neural Networks (CNN) have become a state-of-the art deep learning algorithm to solve image classification tasks. In this study, we first start from the basis, by building a simple non-vectorized CNN (without using any packages), to understand the mechanics of the model. Later on, we will use Pytorch to build and optimize a classifier that handles a great quantity of input images and uses several techniques (such as sophisticated optmizers and data augmentation) to achieve more than 99% accuracy in the test set. Lastly, we will also build a variable-resolution network to handle inputs of different sizes without having to change the network's architecture.

First, given and input tensor x of dimensions with dimensions (batch size, input channels, input width, input height), as well as a choice on the amount of padding, number of output channels, kernel size, and a stride, we produce an output tensor with dimensions (batch size, output channels, output width, output height). This is done in a non-vectorized way as seen in algorithm 1.

Algorithm 1: Non-Vectorized Convolution Operation

```
def forward(X):
    inp = X
    batch_size = X.shape[0]
    input_size = X.shape[1]
    input_channel = X.shape[3]

    filters = filters_w/_shape(output_channel, filter_size, filter_size, input_channel)

    output_dim = int(((inp.shape[1]-filter_size + 2 * pad)// stride)+1)
    output_image = np.zeros((batch_size, output_dim, output_dim, output_channel))

    X_padded = np.pad (X, ((0,0), (pad, pad), (pad, pad), (0,0)))

    for r in range(batch_size):                          #n batch sizes
        for k in range(output_channel):                  #n output channels
            filt = filters[k]
            for i in range(output_dim):                  #horizontal axis
                for j in range(output_dim):              #vertical axis
                    output_image[r,i,j,k] = np.multiply(filt, X_padded[r,i*
                                            stride + filter_size, j * stride: j *
                                            stride + filter_size, :]).sum()
```

Within this implementation, it can be seen that a convolution is a matrix operation, in which a filter (also known as kernel), is a reduced matrix of weights, that sweeps the input doing a element-wise multiplication with respect to the part of the input it is on (also known as patch). Finally results of each of those operations W*X is summed resulting in the output.

From this implementation, we can already observe a pattern on the size of the output for a given input tensor, kernel size, stride and padding. The output size of each Convolutional layer can be calculated as:

$$output\_size = [(input\_size + 2 * padding - kernel\_size)/stride] + 1 \tag{1.1}$$

where the size calculation can be applied for both output height or width. The padding is always multiplied by 2 due to symmetry of adding it for both sides. The kernel_size refers here to the dimensions of the filter (that is always a square matrix: i.e nxm where n=m). The stride indicates the number of units in which the kernel shifts over the input matrix; finally, the '+1' is to account for the initial position of the kernel.

The previous algorithm, although illustrative, is not practical for real applications. The use of nested loops is quite inefficient to implement the convolution. Therefore, a vectorization of both inputs (X) and weights (W) resulting in an also vectorized Y prediction are ideal for solving this issue. This is especially useful for if we have multiple channels and multiple inputs (for the MNIST dataset, each X is a different image of a number). We can see how that is applied in algorithm 2.

Algorithm 2: Vectorized Convolutional Layer

```
def X_flatten(X, window_h, window_w, window_c, out_h, out_w, stride=1, padding=0):
    X_padded = np.pad (X, ((0,0), (padding, padding), (padding, padding), (0,0)))
    windows = []

    for i in range(out_h):
        for j in range(out_w):
            window = X_padded[:,i *stride:i * stride +
            window_h, j * stride: j*stride + window_w, :]
            windows.append(window)
    stacked = np.stack(windows)
    return np.reshape(stacked, (-1, window_c * window_w * window_h))

def convolution(X, n_filters, kernel_size, padding, stride):

    global conv_activation_layer
    k_h = kernel_size[0]
    k_w = kernel_size[1]

    if padding == 'VALID':
        pad = 0
    else:
        pad = 1

    filters = []
    for i in range(n_filters):
        kernel = np.random.randn(k_h, k_w, X.shape[3])
        filters.append(kernel)
    kernel = np.reshape(filters, (k_h, k_w, X.shape[3], n_filters))

    n,h,w,c = X.shape[0], X.shape[1], X.shape[2], X.shape[3]
    filter_h, filter_w, filter_c, filter_n = kernel.shape[0], kernel.shape[1],
                                             kernel.shape[2], kernel.shape[3]

    out_h = (h + 2 * pad - filter_h) // stride + 1
    out_w = (w + 2 * pad - filter_w) // stride + 1

    X_flat = model.X_flatten(X, filter_h, filter_w, filter_c, out_h, out_w, stride, pad)
    W_flat = np.reshape(kernel, (filter_h * filter_w * filter_c, filter_n))

    z = np.matmul(X_flat, W_flat)
    z = np.transpose(np.reshape(z, (out_h, out_w, n, filter_n)), (2,0,1,3))
    conv_activation_layer = relu.activation(z)

    return conv_activation_layer
```

After flattening the input and applying the convolution operation, the last step that needs to be performed is reshaping the output to the desired size. This leaves us with a total of three steps: unfolding (U), matrix multiplication (Y'), and reshaping (Y). When given an input tensor X, the operations are defined as follows:

$$U = \text{Unfold(X)} \tag{1.2}$$

$$Y' = WU \quad \text{(Matrix Multiplication, W = Kernel Weights)} \tag{1.3}$$

$$Y = \text{Reshape(Y')} \tag{1.4}$$

The backward of this function with respect to the kernel weights W is computed as:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial Y'}\frac{\partial Y'}{\partial W} = Y^{\nabla}Y'^{\nabla}\frac{\partial (WU)}{\partial W} = Y^{\nabla}Y'^{\nabla}U \tag{1.5}$$

where Y^{∇} is the gradient of the loss with respect to the output, Y'^{∇} is the gradient of the output with respect to the reshape (which is equal to an unfold operation). The gradient of the convolution with respect to the kernel weights W is just the input to the convolution U (the unfolded tensor). The multiplication $Y'^{\nabla}U$ is basically another convolution, with U as the input and Y'^{∇} as the kernel. The backward with respect to the input X is defined as:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial Y'}\frac{\partial Y'}{\partial U}\frac{\partial U}{\partial X} = Y^{\nabla}Y'^{\nabla}\frac{\partial (WU)}{\partial U}\frac{\partial U}{\partial X} = Y^{\nabla}Y'^{\nabla}WU^{\nabla} \tag{1.6}$$

with Y^{∇} and Y'^{∇} defined as previously. The gradient of the convolution with respect to the input U returns the kernel weights W, while the gradient of the unfold operation U is the inverse function, which leaves us with a fold operation. The multiplication $Y'^{\nabla}W$ is yet another convolution, which is applied to the rotated foward kernel weights W with kernel Y'^{∇}.

A basic implementation of the steps just described can be done in PyTorch by defining a new function. This procedure is outlined in algorithm 3.

Algorithm 3: Custom Implementation of Convolution

```
class Conv2D(torch.autograd.Function):

        @staticmethod
        def forward(ctx, inputs, kernel):

                ctx.save_for_backward(inputs, kernel)
                unfold = nn.Unfold(kernel_size=(3, 3))
                u = unfold(inputs)
                yh = u.matmul(kernel)
                y = yh.reshape(-1)

                return y

        @staticmethod
        def backward(ctx, grad_output):

                inputs, kernel = ctx.saved_tensors
                grout = unfold(grad_output)
                input_batch_grad = torch.matmul(inputs, grout)
                kernel_grad = torch.matmul(kernel, grout)
                kernel_grad = torch.fold(kernel_grad)

                return input_batch_grad, kernel_grad
```

1.1 Dealing with Different Resolutions

Another characteristic of a CNN, apart from the vectorization - and therefore capacity to deal with a great input size as well as applying different kernels - is the flexibility to apply the same operation on images of different sizes. Let's take a convolution with a 5x5 kernel size, padding 1 and stride 2, with 3 input channels and 16 output channels. If we apply it to an image with size (3, 1024, 768) the output size will be (16, 511, 384), calculated through Equation 1.1. For an image of input size (3, 1920, 1080) the same operation yields (16, 960, 540), while for an image with size (8, 1920, 1080) we would have to modify the kernel to accept more input channels or remove some of them (e.g. through a grayscaling operation). Since images often have different resolutions, constructing a network that can deal with this issue becomes an important problem. A basic approach is to resize to a common resolution and then train on a fixed network, but this can cause problems of information loss. Downsizing images can lead to information loss while upsizing images adds noise and can negatively affect performances. The ideal approach would be to keep inputs to their own resolution, but this would require a network with variable size, since different resolutions result in different shapes of the tensor that gets fed to the fully-connected layer. To tackle this problem while still maintaining the original resolution of images, we could use a global pooling operation before the fully-connected layer, which computes a pooling operation over the whole image, removing height and width and therefore allowing different inputs to result in the same final tensor. Two of the most commonly used global pooling are Max Pooling and Mean Pooling, In PyTorch, these can be achieved through the following commands:

```
torch.flatten(F.adaptive_max_pool2d(x, (1, 1)), 1) (Global Max Pooling)
torch.flatten(F.adaptive_avg_pool2d(x, (1, 1)), 1) (Global Mean Pooling)
```

2 Methodology

2.1 Simple Convolution Classifier

After developing an intuition on convolutions, we can now build a CNN to classify the MNIST data using Pytorch. The CNN will initially have three Convolutional layers activated with a ReLu function and Max pooled, followed by a fully connected layer that will generate the prediction. A loss will be calculated by using both a Cross Entropy Loss and a Binary Cross Entropy (encoded with one hot vectors), followed by the Adam optimization function with a parameter update. At this first moment, batch sizes of 16 examples will be used. All dataset is normalized when loaded.

For the experiments that will follow, to test the model as well as finetune it, we have divided the dataset into three parts: 50 000 instances were devoted to the training set, 10 000 to the validation (that will be used for testing our possible models and hyperparameters) and finally a test set of 10 000 instances is reserved for the test set, that will only generate the performance for the best model.

2.2 Variable-Resolution Classifier

A variable resolution classifier was built, using the same architecture as the previous network but adding a global pooling operation before the fully-connected layer. This type of network can take as input images with different resolution (64x64, 48x48, and 32x32 in this case), dealing with them through a global pooling layer before the final fully-connected nodes. In order to achieve a network with a similar number of parameters as the one previously built, a trial-and-error exploration was performed to determine the number N of output channel in the last convolution operation. The fixed resolution network has a total of 29066 parameters. The number of parameters is calculated as follows:

```
sum(parameter.number_elements() for parameter in model.parameters())
```

Therefore, the value of N that gives an amount of parameters closest to 29066 was chosen: N was then set to 81, which gives a total of 29029 parameters. Lastly, the global pooling operation was implemented, with two variants being global max pooling and global average pooling.

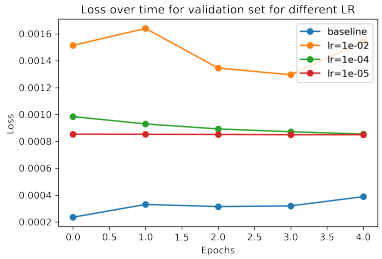
3 Experiments

3.1 Experiment 1 - Defining a Baseline

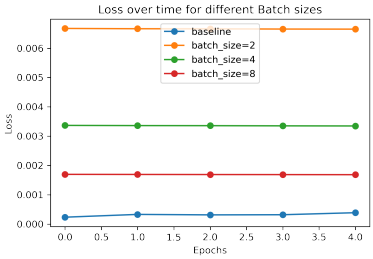
After building a simple Convolution Classifier, a hyper-parameter tuning was done in order to get a good baseline for future experiments. In order to investigate how training and validation behavior changes when network architecture is altered, several experiments were implemented. Due to computer power, a limited scope of parameters were chosen to be tested, not being possible to test all sets of parameter values against each other. The following aspects of the model were inspected:

- **Epochs:** First, a different number of epochs is tested over the model. This approach helps evaluating if more iterations can actually keep improving the learning, or if the algorithm will reach a plateau. While for 5 epochs, accuracy is 99.1% and the average loss is of 0.000513 , for 10 and 15 epochs the accuracy remains exactly the same (with a bigger computational effort). This indicates that for this model with the specific settings of learning rate 0.001 and batch size of 16, 5 epochs is enough to reach a maximum performance.
- **Learning Rate:** The number of epochs set to 5 was kept, as well as batch size of 16, and only learning rate was altered. After experimenting with the set of values [0.01, 0.001, 0.0001, 0.00001] it is seen that differences in loss can be clearly observed in Figure 1a). When using LR of 1e-2, results were worst within the set of values experimented, as well as less stable than other learning rates (in it's best performance it achieved a loss of 0.001296 and an accuracy of 97.7%). Smaller learning rates performed better, and 1e-3 was considered optimal for this set.
- **Batch size:** Using a different size of batches (as seen in Figure 1b) does affect the loss as well. The smaller the batch size was set, the poorer the loss, being 0.0066 for size 2, 0.0033 for size 4 and 0.0016 for size 8. Batch size 16, however was the only to reach a loss of 3 zeros in the decimal points after 5 epochs.

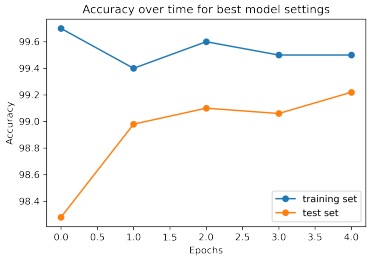
After experiments with the aspects, the best performed model was chosen, and results for its accuracy can be seen in Figure 1c. The use of 5 epochs, a learning rate of 1e-03 and a batch size of 16 will become the baseline for comparing next models. Final results were of 99.22% accuracy, and a cost of 0.000415.



(a) Validation set with baseline settings and different Learning rates



(b) Validation set with baseline settings and different batch sizes



(c) Accuracy of best performing model on test set

3.2 Experiment 2 - Using Data Augmentation

As a way of improving the training, data augmentation techniques were applied to the best model settings. These random manipulations in the input images help reducing overfitting, as well as it makes the algorithm learn invariance that did not appear originally. For those reasons, it should be only applied when training. To serve this purpose, different data augmentation were combined and tested with the validation set, to understand how much would the model improve for unknown datasets. The following masks were applied: 'CollorJitter', which randomly change the brightness, contrast, saturation and hue of an image; the 'RandomRotation', which rotate the image by 30 degrees angle; and 'RandomAffine', that transforms the image keeping the center invariant.

Results can be seen in Figure 2. When comparing the baseline with the training developed over a dataset with augmented data results slightly improve, achieving 99.30% accuracy and an average loss of 0.000367 in epoch 5. Since the baseline had already learnt with a high accuracy, improvements are seen in a smaller scale, however this effect can still be appreciated after applying image transformation techniques.

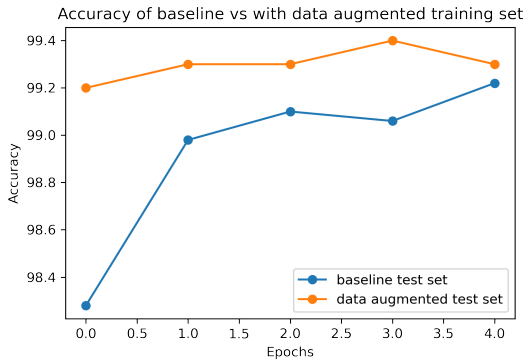


Figure 2: Accuracy comparison for Fixed Resolution network.

3.3 Experiment 3 - Variable Resolution Network

After constructing the Variable-Resolution network, a variable-resolution MNIST dataset was loaded (with resolutions 64x64, 48x48, and 32x32) and its input was resized to a 28x28 resolution and fed through the previously-tuned fixed resolution network, to establish a baseline for training loss and validation set accuracy. Figure 3 shows the comparison of the training and validation loss when using the fixed resolution network. The obtained accuracy on the validation set without data augmentation was 97.2% after 5 epochs, a higher value than the 91.5% after 5 epochs obtained when using data augmentation. Both performances are lower than the ones obtained in (99.30% for the best model), likely due to the resizing operation which leads to an information loss. To obtain different resolutions in the dataset, a padding of black pixels was added around the original image, so resizing leads to information loss, especially for the higher resolution images (e.g. 64x64), explaining the drop in performance. Using data augmentation further decreases the accuracy, as this operation adds more noise on top of the already lower-resoluted resized images.

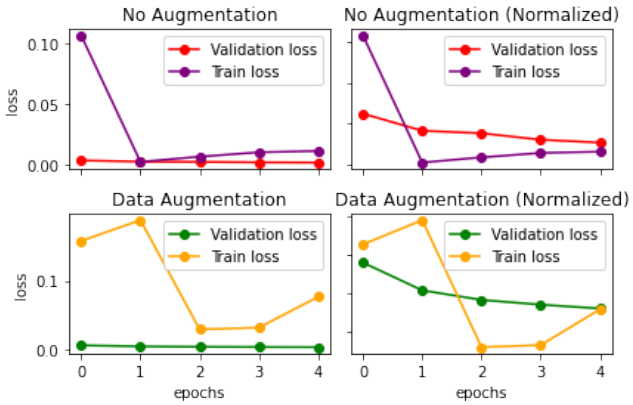
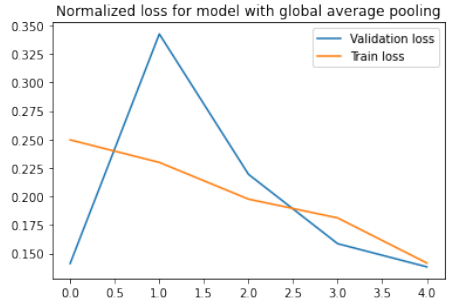


Figure 3: Losses and Normalized Losses with and without Data Augmentation.

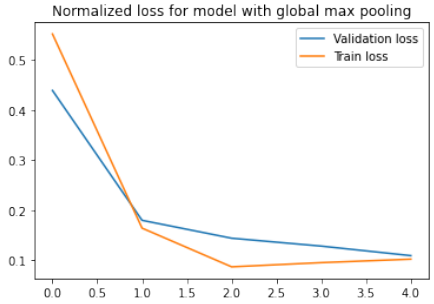
Another option could be to resize all images to 64x64 resolution, however this would cause a number of issues. First of all, computations would be more expensive, leading to higher training times for the network. In addition, more noise would be introduced in the images, as some would become more blurry, possibly leading to a decrease in performances. Lastly, the network would need to be changed: currently the fixed resolution NN is built to deal with 28x28 images, accepting a tensor of size (batch, 576) for the fully-connected layer, however it would now need to receive an input size of (batch, 4096) (the higher dimensionality previously mentioned can also be appreciated by the differences in shapes).

The images with different resolutions were then loaded into three separate tensors and then fed through the variable resolution network. The performance of the network with global max pooling and global average pooling were compared and, after tuning it to the optimal pooling method, it was compared to the classifier previously built when resizing the input to 28x28. The global max pooling and global mean pooling have been implemented before the linear layer, with intricacies of the model described in section 3.3. The two methods have been trained over 5 epochs for comparison purposes. Results are seen in 4a, 4b, 4c, 4d. The figures indicate

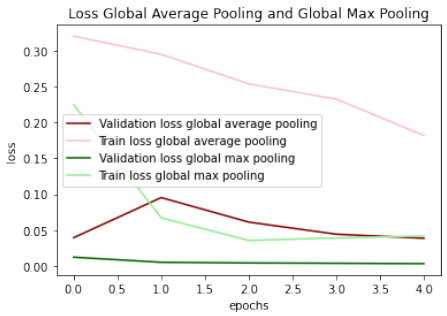
that global max pooling reaches a higher validation accuracy within 5 epochs, and a lower validation loss. The validation loss of global max pooling decreases in a more stable way than global average pooling. Global max pooling is therefore chosen for the remaining experiments.



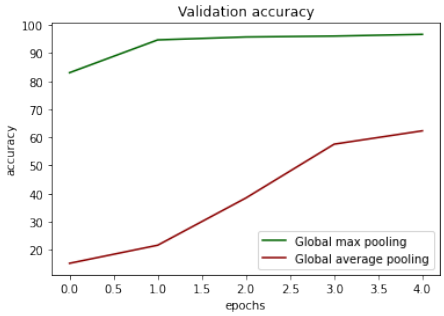
(a) Loss for global average pooling, normalized for visualization purposes.



(b) Loss for global max pooling, normalized for visualization purposes.



(c) Comparison of validation loss for global mean and max pooling.



(d) Comparison of validation accuracy for global mean and max pooling.

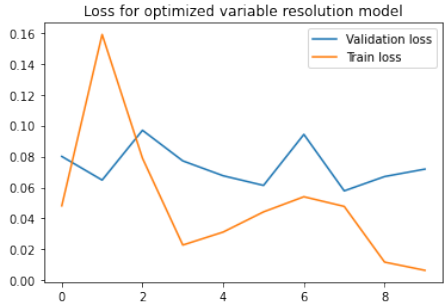
Next, both networks - the variable resolution network with global max pooling and the fixed resolution network - were compared to each other. For the fixed resolution network, the images were all resized to 28x28 so that they could be given as input to the model. The hyperparameters (batch size and learning rate) of both models were tuned using 5 epochs. This leads to the following settings:

	Variable resolution model	**Fixed resolution model**
Batch size	2	8
Learning rate	1e-2	1e-3

Table 1: Optimized hyperparameters.

To show the training and validation losses of the final models, the models with tuned hyper parameters were trained for 10 epochs (figure 5a and figure 5b). The validation loss is lower for the fixed resolution model. Both seem to have an unstable training loss, which is not consistently decreasing. However, this can be due to the training on only 10 epochs.

Finally, the two optimized models were tested on the test set. The accuracy of the fixed resolution model was 97.7%, and the accuracy of the variable resolution model 91.13%. The fixed resolution model thus seems to perform better on this data. A possible explanation for this event could be that the global max pooling will lead to a loss of information, more than regular max pooling (if the images are equal in size). This effect is stronger when the input images are larger and could thus be the reason for the worse performance.



(a) Loss for optimized variable resolution model.

(b) Loss for optimized fixed resolution model.