Mini-Project 2

Jozef Coldenhoff, Bonan Sun

I. Introduction

Neural networks are a class of models that attempt to approximate any function satisfying some constraints, by tuning model parameters using gradient methods. In particular, neural networks attempt to minimize an objective function by computing gradients, and making a step in the model parameter space in the direction of steepest descent. This is done by using the chain rule to propagate gradient information through the different layers of the model.

Over the years, many packages offering automatic differentiation have surfaced, with the currently most used being TensorFlow[1] and PyTorch[2].

Our implementation is based on the `Tensor` class offered by PyTorch, which offers many primitives allowing for numerical manipulations on the underlying data contained in the Tensor. Our implementation will work without the automatic differentiation engine offered by PyTorch.

II. Implementation details

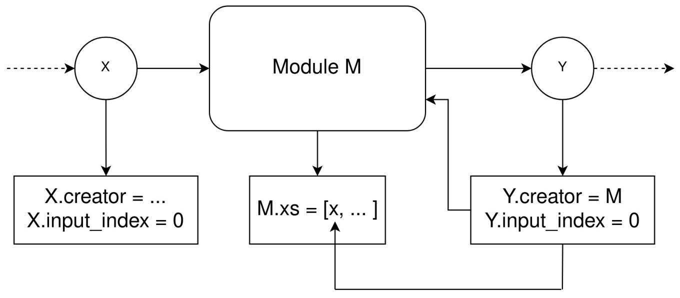

Fig. 1. Structure of a module in our framework. The module takes the input and applies some transformation on x generating output y. The module also sets references to the creator of the output y, and the index of the input that generated it.

Figure 1 shows the general structure of a module in our framework. As we can see, x and y denote the input and output respectively. Module M applies some operation on the input x and generates the output y. In the process of this transformation, the module M sets two attributes to the output y. In particular, y stores a reference to the module that generated it under the attribute y.creator. The output y also stores an index number of the input x used to generate the output y, which is needed to compute $\frac{\partial L}{\partial x}$ in order to propagate the corresponding gradient further down the network.

Note that we could also have chosen to attach a reference to the input `Tensor` x to the output `Tensor` y, we however chose to attach just an index, and store the input in the module. This was done, as the whole gradient computation is contained inside the module M itself, and y does not necessarily need to "know" about what input generated it for the forward pass. This choice also makes it easier to reference the order and actual inputs made to the module by looking at the `M.xs` attribute.

A. Forward pass

In the forward pass, the following happens. We assume that there is some `Sequential` container that contains a number of modules. When we input `Tensor` x, the `Sequential` container, iteritatively calls the `forward()` function of the contained modules which transform the input, and set the correct `.creator` and `.input_index` attributes.

Listing 1. Forward function of the Sequential container

```
def forward(self, input):
    x = input
    for m in self.modules:
        x = m(x)
    x.creator = self
    return x
```

B. Backward pass

Then in the backward pass, the `Sequential` container simply calls `backward()` on the last module in the container, where after the modules take care of the propagation of the gradient.

Listing 2. Backward function of the Sigmoid module

```
def backward(self, gradwrtoutput):
    g = gradwrtoutput
    x = self.xs[g.input_index]
    s = x.sigmoid()
    dldx = s * (1 - s) * g
    dldx.input_index = g.input_index
    if hasattr(x, 'creator'):
        x.creator.backward(dldx)
```

The above code shows the backward function of the `Sigmoid` module as an example to illustrate the idea. As can be seen in the code, during the backward pass, the gradient of the loss with respect to the output of the module is passed as an input. The module then calculates the gradient with respect to the input, and the gradient with respect to its parameters (in the case of the `Sigmoid` it does not have parameters). In the case of a module with parameters such as the convolution, and transpose convolution, we compute the gradient with respect to each of its parameters, and set it to its `grad` attribute, which can then be used later by the SGD optimizer to update the parameters. As can also be seen in the code above, it is possible for the input x to not have a `creator`, this means that we are at a leaf node of the computational graph, and as such does not need to propagate its gradient further.

A special case of the backward function can be found in the MSE loss. In this case, it is possible to omit the gradient with respect to the loss, as it will simply be a tensor of ones. We simply loop over the list of error vectors $e = \hat{y} - y$ computed by the MSE module, compute the gradient, and propagate them back to the appropriate module.

C. Optimizers

1) SGD optimizer: The last step of a full pass in the network, is to use the gradients to update the parameters of the network. The SGD algorithm takes the gradients obtained from a mini-batch of input-target samples, and updates the parameters by subtracting the gradient times some small constant γ from the parameter i.e. $\theta^{t+1} = \theta - \gamma \cdot \frac{\partial L}{\partial \theta^t}$. We achieve this by passing the model to the SGD optimizer, which loops over all the parameters in the network, updating them. This is done by calling `SGD.step()`. Finally, before starting the next forward/backward pass, we need to set the gradients of the parameters to 0, which is achieved by calling `SGD.zero_grad()`, this function in turn also clears all the tensors saved in the various modules by looping over the modules in the Sequential container, and calling the Module member function `Module.clear_saved_tensors()`.

2) Adam optimizer: Given our heavy use of the Adam optimizer in the first Mini-Project, we chose to implement Adam as well [3]. We will compare the convergence speed of the different optimizers in section III

D. Differences with PyTorch

Our implementation differs from PyTorch in a number of ways. Firstly, as stated before, the Tensors used for computing $\frac{\partial L}{\partial x}$, are stored in the Modules themselves, as opposed to being their own entity in PyTorch. Furthermore, in PyTorch, the `.backward()` member function is defined directly on the `Tensor` class, this has a benefit that we do not need to know where the tensor came from in order to call `backward()`. In our implementation, we however have to call `Tensor.creator.backward()` in order to make the backward pass. We also lack the machinery to call backward on an arbitrary tensor automatically. To do this in our framework, we need to generate a `Tensor` with the same shape as the output filled with ones. Moreover, our implementation of SGD requires a reference to the full Sequential container in stead of just the parameters of the Modules, this is required, as we need to call `.clear_saved_tensors()` on all the sub-Modules in the Sequential, necessitating the SGD optimizer to know about the Module as a whole. Another difference with PyTorch is that when using MSE as a loss, we are able to do multiple forward passes followed by a single call to backward to accumulate gradients from both forward passes, while in PyTorch we need to specify $(loss_1 + loss_2)$`.backward()`. The benefit of being able to selectively backpropagating a single input-output pair, has as disadvantage, that the computational graph is not consumed during this backward pass, leading to a potentially larger memory footprint.

E. Enabling Cuda

In order to accelerate computations, we enable cuda computation by implementing a `.cuda()` method on the `Module` class. This method simply moves the parameters to cuda by calling `parameter.to('cuda')`.

F. Parameter initialization

In order to ensure numerical stability and avoid exploding and vanishing gradients, we initialize each parameter group used in 2d-convolution, and transposed convolution with the same method implemented in PyTorch, where the parameters are iid samples from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = 1/(c \cdot n)$, c is the number of input channels in the case of 2d-convolution, and the number of output channels in the transposed convolution case, and n the number of entries in the kernel.

G. 2d convolution

In order to implement the 2d convolution without native python loops, we notice that convolution is a linear transformation on the (of course, finite-dimensional) vector space of the images, which is isomorphic to a space of matrices with corresponding dimensions equipped with the standard matrix multiplication. This means we can compute a convolution by reshaping the kernels and performing a simple matrix multiplication. We leverage the `torch.unfold()` function, which unfolds the images into columns containing the values as they would have been seen by the 2d convolution.

Formally, let us denote the input of the layer by $x \in \mathbb{R}^{N\times C\times H\times W}$, output by $s \in \mathbb{R}^{N\times C'\times H'\times W'}$, convolution kernel weights by $w \in \mathbb{R}^{C'\times C\times K_1\times K_2}$, where $N, C, H, W, C', H', W', K_1, K_2$ are batch size, input channel size and height and width, output channel size and height and width, kernel height and width respectively. For simplicity and clarity, we ignore the bias now. Then we have the isomorphism illustrated in the following commutative diagram, where $\circledast$ and $*$ denote the convolution operation and the (batch) matrix multiplication respectively, leading to the implementation of the forward pass. Note that $K = CK_1K_2, P' = H'W'$ in this diagram.

$$\begin{tikzcd}
x \in \mathbb{R}^{N\times C\times H\times W} \arrow[r, "\circledast w \in \mathbb{R}^{C'\times C\times K_1\times K_2}"] \arrow[d, "\texttt{unfold}"'] & s \in \mathbb{R}^{N\times C'\times H'\times W'} \\
\tilde{x} \in \mathbb{R}^{N\times K\times P'} \arrow[r, "*\tilde{w} \in \mathbb{R}^{C'\times K}"'] & \tilde{s} \in \mathbb{R}^{N\times C'\times P'} \arrow[u, "\texttt{fold}"']
\end{tikzcd}$$

To implement the backward pass, we need to compute the gradient. Note that in the forward pass we have $\tilde{s} = \tilde{x} * \tilde{w}$, which can be written element-wisely as

$$\tilde{s}_{i,j,k} = \sum_{m=1}^{k} \tilde{w}_{j,m} \tilde{x}_{i,m,k}.$$

By chain rule we have

$$\frac{\partial l}{\partial w_{j,m}} = \sum_{i,k=1}^{N,P'} \frac{\partial l}{\partial s_{i,j,k}} \frac{\partial s_{i,j,k}}{\partial w_{j,m}} = \sum_{i,k=1}^{N,P'} \frac{\partial l}{\partial s_{i,j,k}} \tilde{x}_{i,m,k},$$

which is an Einstein sum and can be further written as a matrix multiplication with proper permuting and reshaping. Similarly, we have

$$\frac{\partial l}{\partial x_{i,m,k}} = \sum_{j=1}^{C'} \frac{\partial l}{\partial s_{i,j,k}} w_{j,m}.$$

H. *Transposed convolution*

By the definition of transposed convolution, the forward pass of it is simply the backward pass $\frac{\partial l}{\partial s_{i,j,k}} \to \frac{\partial l}{\partial x_{i,m,k}}$ of the convolution. Then we can compute the gradient based on this formula by similar derivations as in the last section.

III. Testing

A. *Numerical issues*

During the testing phase of our implementation, we found that the computed gradients slightly differed from the ones computed by PyTorch when propagated from module to module, with the first gradient in the chain (last layer) being exactly the same as the PyTorch one. The order of magnitude of the errors are around 10^{-18}. These errors are likely the case due to small errors accumulating across multiple computations. We do find however that these small errors do not hurt convergence rate as compared to the native PyTorch implementation.

B. *Speed*

In order to benchmark our implementation, we measure the time it takes to run for one epoch on the train data. As expected, the native PyTorch implementation is much faster than ours. This is of course due to the speedup gained from native C implementations. We are however content with this speed, as it is not that much slower than PyTorch.

		Epoch in Seconds	Standard Deviaton
Ours	SGD no momentum	3.846	0.187
	SGD with momentum	3.932	0.131
	Adam	3.986	0.143
PyTorch	SGD no momentum	0.844	0.070
	SGD with momentum	0.770	0.022
	Adam	0.856	0.060

TABLE I
Table showing the timings of training for one epoch. One epoch consists of 50000 training examples with batch size of 128. Shown times are computation times using CUDA capability on a laptop RTX 3060.

C. *Best Model*

Following our findings from the first project, and the constraints put on the network in the project descriptions, we converged on the following network architecture:

Listing 3. Implemented model

```
Sequential(Conv2d(3, 16, 4, stride=2, padding=1),
    ReLU(),
    Conv2d(16, 32, 4, stride=2),
    ReLU(),
    TransposeConv2d(32, 16, 4, stride=2),
    ReLU(),
    TransposeConv2d(16, 3, 4, stride=2, padding=1),
    Sigmoid())
```

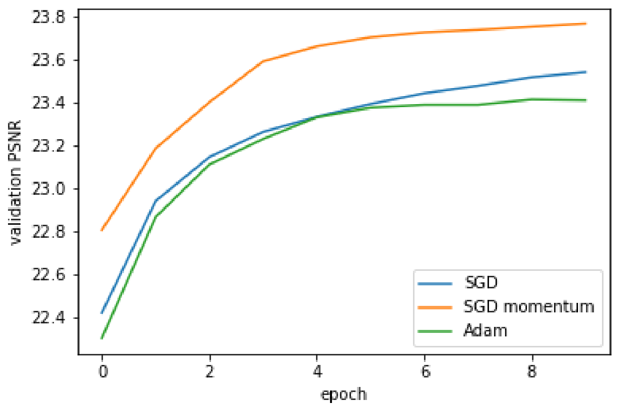

Fig. 2. Validation PSNR per epoch for different optimizers using batch size 16 and 50000 training samples. SGD used a learning rate of 0.5, and momentum of 0.9 when used. Adam had a learning rate of $1e-2$, and betas of (0.9, 0.999).

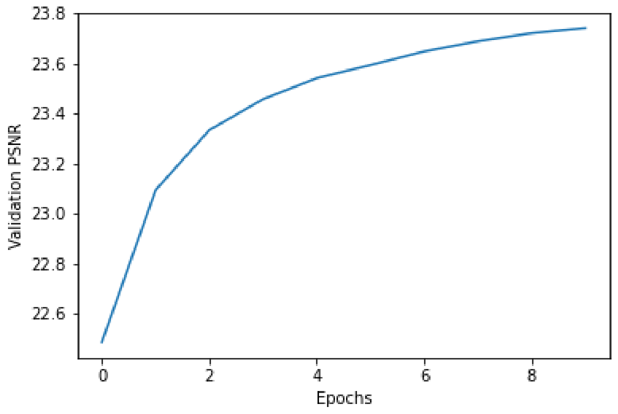

Fig. 3. Final model trained for 10 epochs using SGD optimizer with learning rate of 0.5 and momentum of 0.9, batch size 16.

D. *Convergence speed*

As can be seen in Fig. 2, SGD with momentum greatly outperforms both Adam and vanilla SGD. We will thus use SGD with learning rate of 0.5 and momentum of 0.9 for the training of the final model.

IV. Final model

Thus our final model is trained using the SGD optimizer with learning rate of 0.5 and momentum of 0.9. We use a batch size of 16, without data augmentation to train the network. We obtain a final PSNR of 23.74 Db on the validation set. Figure 3 shows the final training curve of the model.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.