HOMEWORK SET 3

CSCI5527 Deep Learning (Fall 2023)

Due 11:59 pm, Nov 13 2023

Instruction Your writeup, either typeset or scanned, should be a single PDF file. For problems requiring coding, organize all codes for each problem into a separate Jupyter notebook file (i.e., `.ipynb` file). Your submission to Gradescope should include the single PDF and all notebook files—**please DO NOT zip them**! No late submission will be accepted. For each problem, you should acknowledge your collaborators—**including AI tools**, if any.

About the use of AI tools You are strongly encouraged to use AI tools—they are becoming our workspace friends, such as ChatGPT (https://chat.openai.com/, which does not yet accept PDFs, but we provide the LaTeX source codes for our problems that you can copy and enter), Claude (https://claude.ai/chats, which accepts numerous forms of inputs, including PDFs), and Github Copilot (https://github.com/features/copilot), to help you when trying to solve problems. It takes a bit of practice to ask the right and effective questions/prompts to these tools; we highly recommend that you go through this popular free short course **ChatGPT Prompt Engineering for Developers** offered by https://learn.deeplearning.ai/ to get started.

If you use any AI tools for any of the problems, you should include screenshots of your prompting questions and their answers in your writeup. The answers provided by such AI tools often contain factual errors and reasoning gaps. **So, if you only submit an AI answer with such bugs for any problem, you will obtain a zero score for that problem**. You obtain the scores only when you explain the bugs and also correct them in your own writing. You can also choose not to use any of these AI tools, in which case we will grade based on the efforts you have made.

Notation We will use small letters (e.g., u) for scalars, small boldface letters (e.g., $\boldsymbol{a}$) for vectors, and capital boldface letters (e.g., $\boldsymbol{A}$) for matrices. $\mathbb{R}$ is the set of real numbers. $\mathbb{R}^n$ is the space of n-dimensional real vectors, and similarly $\mathbb{R}^{m\times n}$ is the space of $m \times n$ real matrices. The dotted equal sign $\doteq$ means defining.

Preparation Please carefully work through the following two PyTorch official tutorials (if you haven't done so for HW2) before attempting the following problems:

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html
https://pytorch.org/tutorials/beginner/basics/intro.html

You can also use TensorFlow or Jax, but you should figure out how to start by yourself. Basic torch built-in functions are listed at

https://pytorch.org/docs/stable/torch.html.

The tensor class and its built-in functions are listed at

https://pytorch.org/docs/stable/tensors.html.

Problem 1 (Stochastic optimization methods for MNIST digital recognition; 4.5/15) In this problem, we're going to train a shallow neural network based on different stochastic gradient descent (SGD) methods that we learned in class. **Neural networks modules and autograd are allowed in this problem, but no built-in optimizers in PyTorch are allowed.**

First, you should load the MNIST dataset into your workspace. Note that MNIST is a pre-loaded dataset in PyTorch. If you're not sure what this means, revisit this required tutorial page https://pytorch.org/tutorials/beginner/basics/data_tutorial.html, and also check out the pre-loaded image, text, and audio datasets from there.

(a) Design a neural network model **no deeper than** 3 **layers**. You're free to choose the architecture, i.e., number of hidden nodes, activation functions, layers (fully connected, or even convolutional, etc layers). Also, choose an appropriate loss for your training objective. (0.5/15)

(b) Implement the Adagrad algorithm. You're free to choose your hyperparameters (initialization, batch size, learning rate, learning rate scheduler, etc). Please include a plot of how the objective changes with respect to the epoch. (1/15)

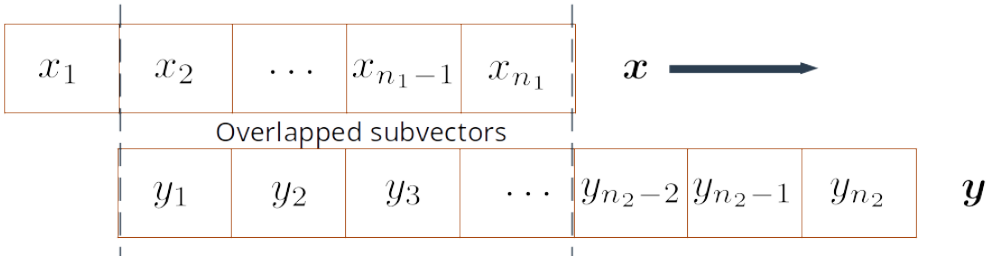
(c) Implement the RMSprob algorithm. The requirement is the same as (b). (0.5/15)

(d) Implement the Adam algorithm. The requirement is the same as (b). The version covered in our lecture is a reduced version of Algorithm 1 of the original paper (https://arxiv.org/abs/1412.6980) that also handles the initial instability. Please implement the original version. (1.5/15)

(e) A "98%" test: MNIST is a relatively easy classification task, and state-of-the-art learning models can achieve near perfect recognition performance. If you get a test accuracy $\geq 98\%$ for **any two** of (b), (c), and (d), you get 1 point here. For this, you are free to adopt any strategy to avoid overfitting if necessary. You may also compare the performance of your implemented optimizers with the built-in ones (https://pytorch.org/docs/stable/optim.html), but the results you report must be obtained from your own implementations. (1/15)

Problem 2 (Correlation and template matching; 7/15) The word "convolutional" in convolutional neural networks is a misnomer. Cross-correlation, which is a close relative of convolution and is commonly used in signal processing, is actually used. In this problem, we explore some basic properties and applications of the cross-correlation operation. We use the standard notation $\star$ to denote cross-correlation, vs. $*$, which is often used to denote convolution.

(a) For two vectors $\boldsymbol{x} \in \mathbb{R}^{n_1}, \boldsymbol{y} \in \mathbb{R}^{n_2}$, the cross-correlation $\boldsymbol{x} \star \boldsymbol{y}$ is obtained as follows:



We fix the position of $\boldsymbol{y}$, and shift $\boldsymbol{x}$ to the left until $\boldsymbol{x}$ and $\boldsymbol{y}$ only have one overlapped element spatially, i.e., x_{n_1} with y_1—that's the starting point. We calculate the inner product of two overlapped subvectors—in the beginning, only two scalars actually. Then we repeatedly do this: shift $\boldsymbol{x}$ to the right by one element and calculate the corresponding inner product of the two overlapped subvectors (i.e., think of a sliding window). We end the process until $\boldsymbol{x}$ and $\boldsymbol{y}$ overlap only at one element, i.e., x_1 with y_{n_2}. The cross-correlation $\boldsymbol{x} \star \boldsymbol{y}$ is basically the

vector that collects all the inner product values that we have obtained in the left-to-right order. It is easy to see that $\boldsymbol{x} \star \boldsymbol{y} \in \mathbb{R}^{n_1+n_2-1}$.

Question: Calculate $[3, 2, 1] \star [4, 6, 3, 9]$. In general, is it true that $\boldsymbol{x} \star \boldsymbol{y} = \boldsymbol{y} \star \boldsymbol{x}$? If not, what relationship between $\boldsymbol{x} \star \boldsymbol{y}$ and $\boldsymbol{y} \star \boldsymbol{x}$ do you observe? (1/15)

(b) In convolutional neural networks, we have building blocks of the form $\boldsymbol{w} \star \boldsymbol{x}$, where $\boldsymbol{w}$ represents a group of learnable weights, often called *filter or kernel* following the signal processing convention. For simplicity, let's assume $\boldsymbol{w} \in \mathbb{R}^3$ and $\boldsymbol{x} \in \mathbb{R}^4$. Show that $\boldsymbol{w} \star \boldsymbol{x}$ can be written equivalently as $\boldsymbol{C}_{\boldsymbol{w}}\boldsymbol{x}$ for a certain matrix $\boldsymbol{C}_{\boldsymbol{w}} \in \mathbb{R}^{6\times 4}$ and write $\boldsymbol{C}_{\boldsymbol{w}}$ explicitly in terms of the elements of $\boldsymbol{w} = [w_1, w_2, w_3]^\intercal$. (1/15)

(c) To apply reverse-mode auto differentiation, we need to specify $\frac{\partial}{\partial \boldsymbol{w}} (\boldsymbol{w} \star \boldsymbol{x})$, i.e., the associated Jacobian. Assume again $\boldsymbol{w} \in \mathbb{R}^3$ and $\boldsymbol{x} \in \mathbb{R}^4$, can you derive the analytic form of the Jacobian? (1/15; Hint: Is it possible to write $\boldsymbol{w} \star \boldsymbol{x}$ as $\boldsymbol{C}_{\boldsymbol{x}}\boldsymbol{w}$ for a certain $\boldsymbol{C}_{\boldsymbol{x}}$?)

(d) The 2D cross-correlation is a natural generalization of the 1D cross-correlation to matrices, as illustrated in Fig. 1. Compared to the 1D version, now we start from the top-left corner and

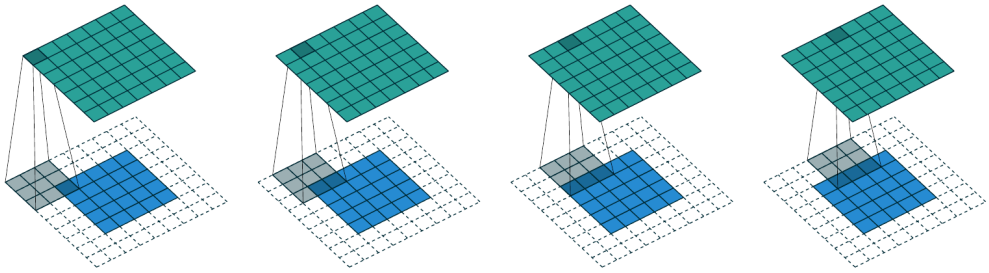


Figure 1: Illustration of 2D cross-correlation (image credit: https://arxiv.org/abs/1603.07285; check out https://github.com/vdumoulin/conv_arithmetic to see the dynamic demonstration under the **full padding, no strides** setting. The additional dotted boxes indicate padded zeros.

end at the bottom-right corner. We scan row by row, and now the inner products are taken between the overlapped submatrices. All the inner product values are naturally organized into a matrix. In the above pictorial illustration, we consider $\boldsymbol{X} \star \boldsymbol{Y}$, where $\boldsymbol{X} \in \mathbb{R}^{3\times 3}$ is the gray matrix, and $\boldsymbol{Y} \in \mathbb{R}^{5\times 5}$ is the blue matrix, the resulting green matrix $\boldsymbol{X} \star \boldsymbol{Y} \in \mathbb{R}^{7\times 7}$, where $7 = 3 + 5 - 1$.

Question: In Numpy, implement a 2D cross-correlation function. The function should take two general matrices $\boldsymbol{Z}_1 \in \mathbb{R}^{n_1\times n_2}$ and $\boldsymbol{Z}_2 \in \mathbb{R}^{m_1\times m_2}$ and return the resulting cross-correlation matrix. To debug your implementation, please generate a couple of random cases and benchmark against the Scipy built-in function `scipy.signal.correlate2d` (remember to set `mode = 'full'`) (1/15)
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.correlate2d.html .

(e) Most basic image processing algorithms are implemented as cross-correlation of a small filter $\boldsymbol{X}$ with the image of interest $\boldsymbol{Y}$. Check out the examples at the bottom of the page https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html. Use their image `ascent`, let's test your implementation of 2D cross-correlation. Try two filters

$$\boldsymbol{X}_1 = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} \quad \text{and} \quad \boldsymbol{X}_2 = \boldsymbol{X}_1^\intercal.$$

Let's say they generate two resulting matrices $\boldsymbol{G}_1 = \boldsymbol{X}_1 \star \boldsymbol{Y}$ and $\boldsymbol{G}_2 = \boldsymbol{X}_2 \star \boldsymbol{Y}$. Calculate $(\boldsymbol{G}_1^2 + \boldsymbol{G}_2^2)^{1/2}$, where the operations $(\cdot)^2$ and $(\cdot)^{1/2}$ are applied elementwise. Display your result (i.e., `imshow` as in the online example). Does your result look like the gradient magnitude plot, except for the image boundaries? (1/15)

(f) Another way of thinking about cross-correlation is template matching. Imagine that $\boldsymbol{X}$ is a 2D pattern of interest. During the cross-correlation process, the inner product measures the agreement of the 2D pattern and local patches in $\boldsymbol{Y}$. If the value is relatively large, very likely we find a match. After we finish the cross-correlation calculation, we can spot the locations of the largest values in the cross-correlation matrix as candidate matching locations. Study the example here https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.correlate2d.html and compare the performance of your implementation with that of the example using `scipy.signal.correlate2d`. (1/15)

(g) In practice, we often have multichannel cross-correlation. Let's consider the following setting: for an input $\mathcal{Y}$ (we use script font to denote tensors) of size H(height) $\times$ W(width) $\times$ D(depth), we consider a filter $\mathcal{X}$ of size $h \times w \times D$, and note that depth of the filter matches the depth of the input. There are two equivalent ways of thinking about cross-correlation:

- **Summation of 2D cross-correlation.** We compute the 2D cross correlations of corresponding layers of the input and the filter, and then sum them up, i.e.,

$$\sum_{d=0}^{D-1} \mathcal{X}[:, :, d] \star \mathcal{Y}[:, :, d] + b, \tag{1}$$

  where b is the bias term;

- **Restricted/degenerated 3D cross-correlation.** We can generalize the previous 2D cross-correlation to 3D cases—that will generate a 3D tensor in principle. But here we do not shift the filter $\mathcal{X}$ in the depth direction and only shift it in the height and width directions. In other words, at each position, we take the inner product of two overlapped 3D "tubes".

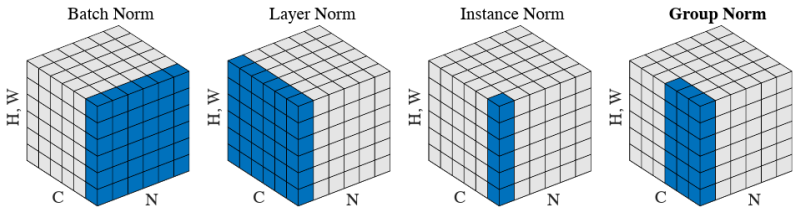
Implement multichannel cross correlation (let's assume $b = 0$), and test your implementation with $H = W = D = 50$ and $h = w = 3$ by generating a pair of random $\mathcal{X}$ and $\mathcal{Y}$ and then comparing your result with that generated by PyTorch from https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html. We take the output size as $(H + h - 1) \times (W + w - 1)$. So, you need to perform proper padding in PyTorch's version to ensure the output size is correct. (1/15)

Problem 3 (The normalization zoo; 1.5/15) Normalization is a crucial building component of deep neural networks (DNNs) that boosts the numerical stability. Simply put, it's about normalizing each slice of multidimensional data in tensor form into zero-mean, unit variance data,

$$\widehat{x_i} = \frac{x_i - \mu}{\sigma} \quad \forall\ x_i \text{ inside the slice}, \quad \text{where } \mu : \text{slice mean } \sigma : \text{slice standard deviation} \tag{2}$$

and feeding these normalized data into the next layer. Different normalization methods differ in how they form the slices, as illustrated in Fig. 2.

Generate a random 4-way tensor of size $H(50) \times W(50) \times C(64) \times N(64)$, and implement and apply batch norm, layer norm, and instance norm onto the tensor. Compare your results with those generated from PyTorch:



Normalization methods. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Figure 2: Illustration of different normalization methods; figure taken from the paper https://arxiv.org/abs/1803.08494

- https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html
- https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html
- https://pytorch.org/docs/stable/generated/torch.nn.InstanceNorm2d.html

You can set their trainable scaling factor $\gamma = 1$ and shift factor $\beta = 0$. To compare two tensors $\mathcal{X}$ and $\mathcal{Y}$ of the same size, we can take the relative difference between them:

$$\frac{\|\mathrm{vec}(\mathcal{X}) - \mathrm{vec}(\mathcal{Y})\|_2}{\max\left(\|\mathrm{vec}(\mathcal{X})\|_2, \|\mathrm{vec}(\mathcal{Y})\|_2\right)}, \tag{3}$$

where vec means vectorization or flattening. If $\mathcal{X}$ and $\mathcal{Y}$ are close, the relative difference between them should be close to 0. Why do the built-in functions place an ε in the denominators? For your random data, it's fine you set their $\varepsilon = 0$.

Problem 4 (Transfer learning; 2/15) In computer vision and natural language processing, large-scale datasets are available and high-performing deep models that are already trained on these datasets, called **pretrained models**, are readily usable. For example, in Pytorch, a list of pretrained models on the renowned ImageNet dataset is available here https://pytorch.org/docs/stable/torchvision/models.html. Since these datasets are large-scale and believed to adequately represent the domain distributions, the learned features tend to be shareable across tasks. For example, in computer vision, when coming to a new image classification task, it is rare that people will train a model from scratch. Instead, a pretrained model will be taken and only finetuning of the model on the new task will be performed.

The different possibilities of finetuning have been explained in class; this webpage provides an excellent summary https://cs231n.github.io/transfer-learning/. A Pytorch tutorial on implementing transfer learning for vision tasks can be found here https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html. In this problem, we will perform transfer learning for classifying pneumonia from chest x-rays. Read the instruction for this Kaggle competition https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia, download (you'll need a kaggle account) and setup the dataset.

(a) Set up an appropriate transfer learning pipeline to perform the classification. Feel free to choose pretrained models you like (or can fit into your resource constraint—large models can be more powerful but need powerful GPUs). You may want to play with different transfer learning strategies and think about the following factors: (1) do you want to freeze all or only some of convolutional layers? (2) or do you want to make all layers trainable, but only

iterate a few steps? (3) or may be borrowing the architecture is sufficient and training can be done from scratch? You may also want to check out our truncated transfer learning paper https://arxiv.org/abs/2106.05152. (1/15)

Hint on training: the two classes are not balanced; it may be helpful to put different weights on the positive and negative samples—e.g., weighting the minority class slightly more than the dominant class—when constructing the training objective; many PyTorch functions already implement the weighting mechanism, e.g., the weight input in `torch.nn.CrossEntropyLoss` https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html. To learn more about the complicated issues around learning from imbalanced data, you may want to refer to our paper https://arxiv.org/abs/2210.12234 for a quick review of these.

(b) A "90%" test: you'll get 1 point if your classification accuracy exceeds 90%. But make sure to show all your work in (a) even if you don't make it 90%. Optionally, you're also encouraged to report the balanced accuracy (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.balanced_accuracy_score.html), and also average precision (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html), but we won't grade you based on the latter two metrics. (1/15)