HOMEWORK SET 4

CSCI5527 Deep Learning (Fall 2022)

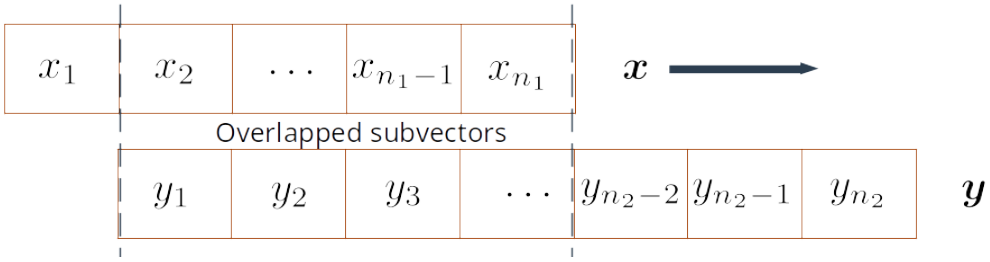
Due 11:59 pm, Nov 28 2022

Instruction Your writeup, either typeset or scanned, should be a single PDF file. For problem requiring coding, please organize all codes for each problem into a separate Jupyter notebook file (i.e., `.ipynb` file). Your submission into Canvas/Gradescope should include the single PDF and all the notebook files—**Please do not zip them**! No late submission will be accepted. For each problem, your should acknowledge your collaborators if any. For problems containing multiple subproblems, there are often close logic connections between the subproblems. So whenever possible, try to build on previous ones, rather than work from scratch.

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.

Problem 1 (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 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, as against $*$ 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 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 $[3, 2, 1] \star [4, 6, 3, 9]$. (0.5/15) For general $\boldsymbol{x}, \boldsymbol{y}$, 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? (0.5/15)

(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* 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 down $\boldsymbol{C}_{\boldsymbol{w}}$ explicitly in terms of elements of $\boldsymbol{w}$. (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

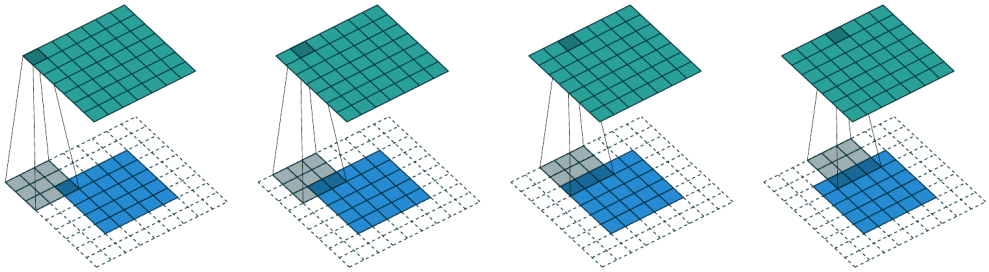


Figure 1: Illustration of 2D cross-correlation (image credit: https://arxiv.org/abs/1603.07285; check out https://github.com/vdumoulin/conv_arithmetic to see the dynamic demonstration under the **Full padding, no strides** setting. Note that padding zeros, as indicated as the additional dotted boxes, is equivalent to ignoring the out-of-boundary elements.)

and end at the bottom-right corner. We scan row by row and inner products are now taken between the overlapped submatrices. All the inner product values are naturally organized into a matrix. In the pictorial illustration above, we are considering $\boldsymbol{X} \star \boldsymbol{Y}$, 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 in 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'`) https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.correlate2d.html. (1/15)

(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

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

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 $\sqrt{\boldsymbol{G}_1^2 + \boldsymbol{G}_2^2}$, where the operations $(\cdot)^2$ and $\sqrt{(\cdot)}$ are applied pointwise. Display your result (i.e., `imshow` as in the online example). Does your result look alike 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 multi-channel cross-correlation. Let's consider the canonical 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 the cross-correlation:

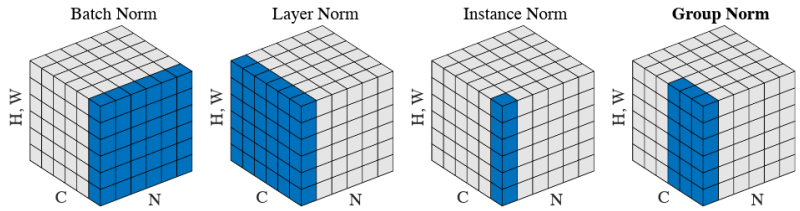
- **Summation of 2D cross-correlation.** We compute the 2D cross correlations of corresponding layers of the input and the filter, and then sum them up, i.e.,

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

  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 will assume the padding mode "valid" in PyTorch, so that the output shape should be $H \times W$ for both your implementation and PyTorch's. (1/15)



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

Problem 2 (The normalization zoo; 4/15) Normalization is crucial for practical optimization success in deep neural networks (DNNs). Simply put, it's about normalizing each slice of multidimensional data in tensor form into zero-mean, unit variance data,

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

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

(a) Generate a random 4-way tensor of size $H(50) \times W(50) \times C(20) \times N(20)$, and implement and apply batch norm, layer norm, and instance norm onto the tensor. Compare your results with those generated from PyTorch:

- 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 $\beta = 0$. Your results might be slightly different, due to their ε in the denominator. Why do they place that ε there? For your random data, it's fine you set their $\varepsilon = 0$. (2/15)

(b) Does normalization reduce the capacity of our DNN? Let's first think about learning a linear model, i.e., one-layer DNN with identity activation. Assume the ideal model we hope to learn is $\boldsymbol{W}^*\boldsymbol{x} + \boldsymbol{b}^*$ for a certain $(\boldsymbol{W}^*, \boldsymbol{b}^*)$ pair. Now batch normalization perform an affine transformation on all input $\boldsymbol{x} \in \mathbb{R}^n$:

$$\widehat{\boldsymbol{x}} = \frac{\boldsymbol{x} - \boldsymbol{u}}{\boldsymbol{\sigma}} = \operatorname{diag}\left(1/\sigma_1, \ldots, 1/\sigma_n\right)(\boldsymbol{x} - \boldsymbol{u}), \tag{3}$$

where $\operatorname{diag}(\cdot)$ reshapes the input vector into a diagonal matrix, and we assume $\sigma_i > 0$ for all i. Show that there exists a pair $(\widehat{\boldsymbol{W}}, \widehat{\boldsymbol{b}})$ so that

$$\widehat{\boldsymbol{W}}\widehat{\boldsymbol{x}} + \widehat{\boldsymbol{b}} = \boldsymbol{W}^*\boldsymbol{x} + \boldsymbol{b}^*, \tag{4}$$

and the pair is independent of the input $\boldsymbol{x}$. In other words, our capacity to learn the ideal model is effectively not affected. (1/15)

(c) Can you generalize the above argument to DNNs with multi-layers and nonlinear activations? (1/15)

Problem 3 (Transfer learning; 4/15) 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 sufficiently represent the domain distributions, the features learned 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 tutotial 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.

(a) 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. (1/15)

(b) 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 strategy, 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 trainable, but only iterate few steps? (3) or may be borrowing the model 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. (2/15)

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; lots of 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 check out our paper https://arxiv.org/abs/2210.12234.

(c) A "90%" test: you'll get 1 point if your classification accuracy exceeds 90%. But make sure to show all your work in (b) 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)