CS480/680: Introduction to Machine Learning

Homework 2

Due: 11:59 pm, June 17, 2024, submit on LEARN.

NAME
student number

Submit your writeup in pdf and all source code in a zip file (with proper documentation). Write a script for each programming exercise so that the TA can easily run and verify your results. Make sure your code runs!
[Text in square brackets are hints that can be ignored.]

Exercise 1: Graph Kernels (6 pts)

One cool way to construct a new kernel from an existing set of (base) kernels is through graphs. Let $\mathcal{G} = (V, E)$ be a directed acyclic graph (DAG), where V denotes the nodes and E denotes the arcs (directed edges). For convenience let us assume there is a source node s that has no incoming arc and there is a sink node t that has no outgoing arc. We put a base kernel κ_e (that is, a function $\kappa_e : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$) on each arc $e = (u \to v) \in E$. For each path $P = (u_0 \to u_1 \to \cdots \to u_d)$ with $u_{i-1} \to u_i$ being an arc in E, we can define the kernel for the path P as the product of kernels along the path:

$$\forall \mathbf{x}, \mathbf{z} \in \mathcal{X},\ \kappa_P(\mathbf{x}, \mathbf{z}) = \prod_{i=1}^{d} \kappa_{u_{i-1} \to u_i}(\mathbf{x}, \mathbf{z}). \tag{1}$$

Then, we define the kernel for the graph $\mathcal{G}$ as the sum of all possible $s \to t$ *path kernels*:

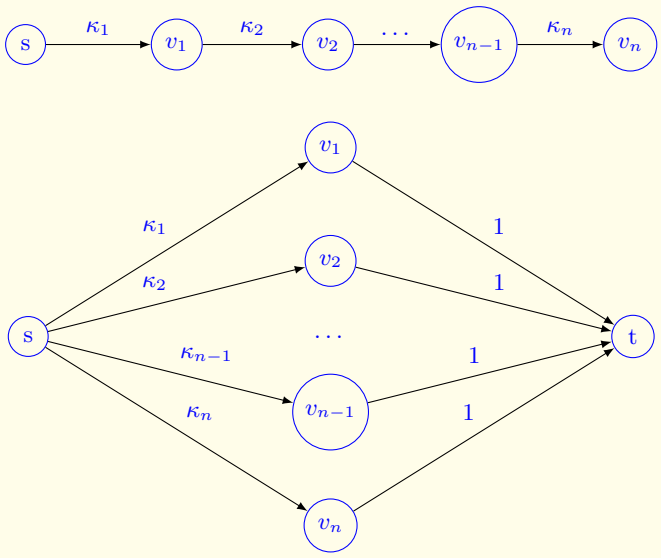
$$\forall \mathbf{x}, \mathbf{z} \in \mathcal{X},\ \kappa_{\mathcal{G}}(\mathbf{x}, \mathbf{z}) = \sum_{P \in \text{path}(s \to t)} \kappa_P(\mathbf{x}, \mathbf{z}). \tag{2}$$

1. (1 pt) Prove that $\kappa_{\mathcal{G}}$ is indeed a kernel. [You may use any property that we learned in class about kernels.]

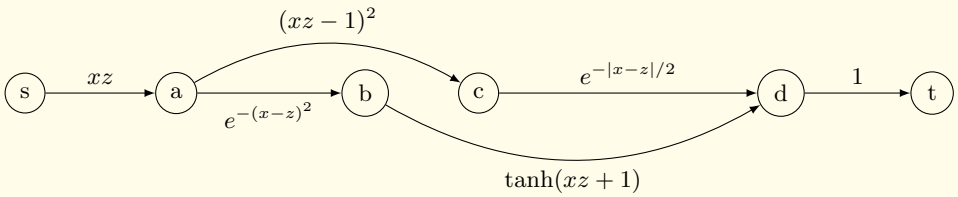
   Ans: From class we know sum and product of kernels remain to be a kernel.

2. (2 pts) Let $\kappa_i, i = 1, \ldots, n$ be a set of given kernels. Construct a graph $\mathcal{G}$ (with appropriate base kernels) so that the graph kernel $\kappa_{\mathcal{G}} = \sum_{i=1}^{n} \kappa_i$. Similarly, construct a graph $\mathcal{G}$ (with appropriate base kernels) so that the graph kernel $\kappa_{\mathcal{G}} = \prod_{i=1}^{n} \kappa_i$.

   Ans:



3. (1 pt) Consider the subgragh of the figure below that includes nodes s, a, b, c (and arcs connecting them). Compute the graph kernel where s and c play the role of source and sink, respectively. Repeat the computation with the subgraph that includes s, a, b, c, d (and arcs connecting them), where d is the sink now.



Ans:

$$k_{s\to c}(x,z) = (xz)\cdot(xz-1)^2 \tag{3}$$

$$k_{s\to d}(x,z) = (xz)\cdot(xz-1)^2\cdot e^{-|x-z|/2} + xz\cdot e^{-(x-z)^2}\cdot\tanh(xz+1) \tag{4}$$

4. (2 pts) Find an efficient algorithm to compute the graph kernel $\kappa_{\mathcal{G}}(\mathbf{x},\mathbf{z})$ (for two fixed inputs $\mathbf{x}$ and $\mathbf{z}$) in time $O(|V|+|E|)$, assuming each base kernel κ_e costs $O(1)$ to evaluate. You may assume there is always at least one $s-t$ path. State and justify your algorithm is enough; no need (although you are encouraged) to give a full pseudocode.

   [Note that the total number of paths in a DAG can be exponential in terms of the number of nodes $|V|$, so naive enumeration would not work. For example, replicating the intermediate nodes in the above figure n times creates 2^n paths from s to t.]

   [Hint: Recall that we can use topological sorting to rearrange the nodes in a DAG such that all arcs go from a "smaller" node to a "bigger" one.]

   Ans: Let $\kappa_v(\mathbf{x},\mathbf{z}) := \sum_{P\in\text{path}(s\to v)}\kappa_P(\mathbf{x},\mathbf{z})$. Under this definition we have $\kappa_{\mathcal{G}} = \kappa_t$. Moreover, we have the following recursion (a.k.a. dynamic programming from CS341):

$$\kappa_v(\mathbf{x},\mathbf{z}) = \sum_{u:u\to v\in E}\kappa_u(\mathbf{x},\mathbf{z})\kappa_{u\to v}(\mathbf{x},\mathbf{z}). \tag{5}$$

   Therefore, if we topologically sort the DAG, with s being the smallest and t being the largest. Then, we can follow the topological order (with $\kappa_s \equiv 1$) and apply the above recursion repeatedly. The total time is $O(|V|+|E|)$ for topological sorting, plus an additional $O(|E|)$ for the recursions.

Exercise 2: CNN Implementation (8 pts)

Note: Please mention your Python version (and maybe the version of all other packages).

In this exercise you are going to run some experiments involving CNNs. You need to know Python and install the following libraries: Pytorch, matplotlib and all their dependencies. You can find detailed instructions and tutorials for each of these libraries on the respective websites.

For all experiments, running on CPU is sufficient. You do not need to run the code on GPUs, although you could, using for instance Google Colab. Before start, we suggest you review what we learned about each layer in CNN, and read at least this tutorial.

1. Implement and train a VGG11 net on the MNIST dataset. VGG11 was an earlier version of VGG16 and can be found as model A in Table 1 of this paper, whose Section 2.1 also gives you all the details about each layer. The goal is to get the loss as close to 0 as possible. Note that our input dimension is different from the VGG paper. You need to resize each image in MNIST from its original size 28×28 to 32×32 [why?].

   For your convenience, we list the details of the VGG11 architecture here. The convolutional layers are denoted as `Conv(number of input channels, number of output channels, kernel size, stride, padding)`; the batch normalization layers are denoted as `BatchNorm(number of channels)`; the max-pooling layers are denoted as `MaxPool(kernel size, stride)`; the fully-connected layers are denoted as `FC(number of input features, number of output features)`; the drop out layers are denoted as `Dropout(dropout ratio)`:

```
- Conv(001, 064, 3, 1, 1) - BatchNorm(064) - ReLU - MaxPool(2, 2)
```

```
- Conv(064, 128, 3, 1, 1) - BatchNorm(128) - ReLU - MaxPool(2, 2)
- Conv(128, 256, 3, 1, 1) - BatchNorm(256) - ReLU
- Conv(256, 256, 3, 1, 1) - BatchNorm(256) - ReLU - MaxPool(2, 2)
- Conv(256, 512, 3, 1, 1) - BatchNorm(512) - ReLU
- Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
- Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU
- Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
- FC(0512, 4096) - ReLU - Dropout(0.5)
- FC(4096, 4096) - ReLU - Dropout(0.5)
- FC(4096, 10)
```

You should use the cross-entropy loss `torch.nn.CrossEntropyLoss` at the end.

[This experiment will take up to 1 hour on a CPU, so please be cautious of your time. If this running time is not bearable, you may cut the training set to 1/10, so only have ∼600 images per class instead of the regular ∼6000.]

2. (4 pts) Once you've done the above, the next goal is to inspect the training process. Create the following plots:

   (a) (1 pt) test accuracy vs the number of epochs (say $3 \sim 5$)

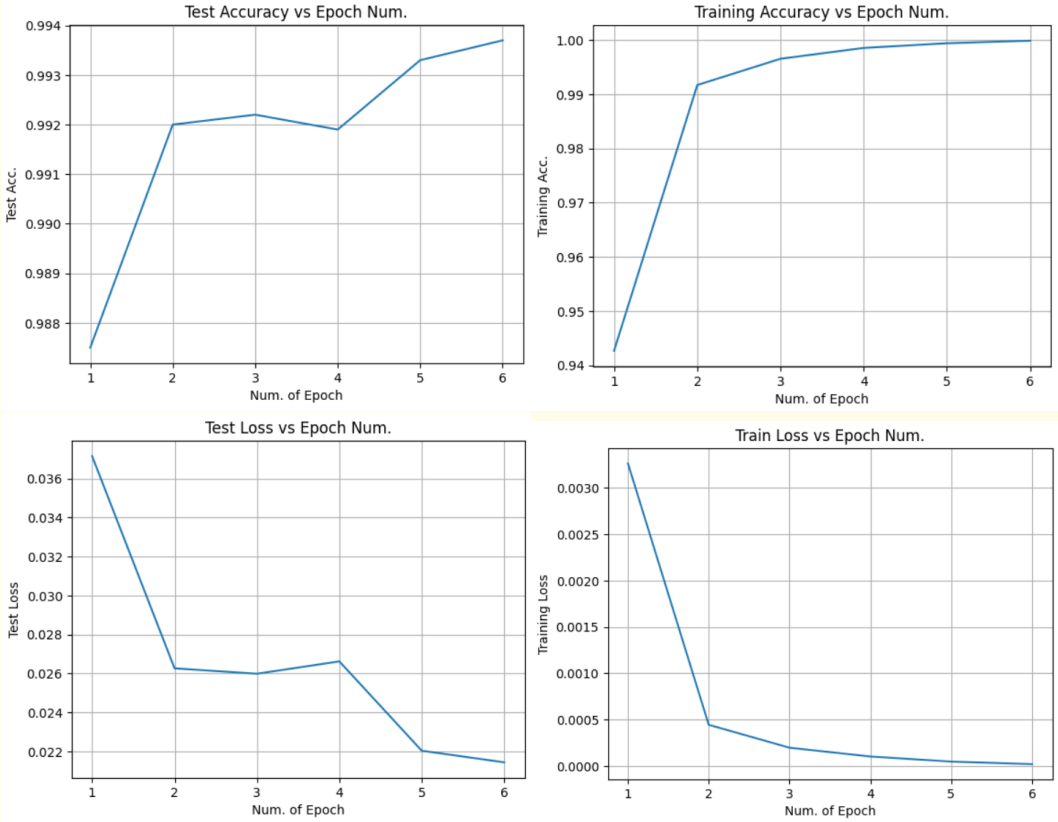
   (b) (1 pt) training accuracy vs the number of epochs

   (c) (1 pt) test loss vs the number of epochs

   (d) (1 pt) training loss vs the number of epochs

   [If running more than 1 epoch is computationally infeasible, simply run 1 epoch and try to record the accuracy/loss after every few minibatches.]

   Ans:

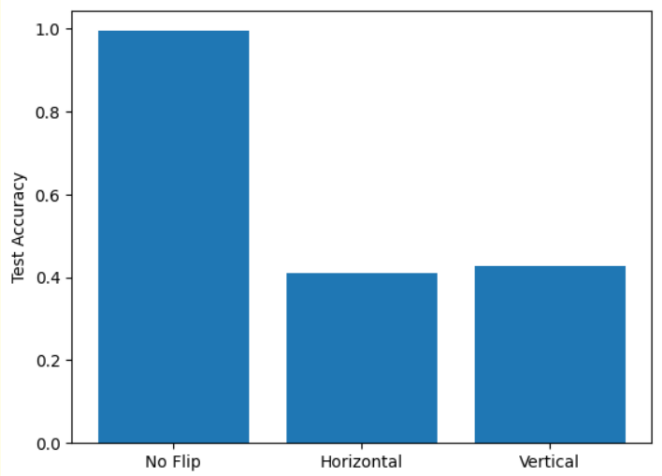


3. Then, it is time to inspect the generalization properties of your final model. Flip and blur the test set images using any python library of your choice, and complete the following:

   (e) (1 pt) test accuracy vs type of flip. Try the following two types of flipping: flip each image from left to right, and from top to bottom. Report the test accuracy after each flip. What is the effect?
   You can read this doc to learn how to build a complex transformation pipeline. We suggest the following command for performing flipping:

   ```
   torchvision.transforms.RandomHorizontalFlip(p=1)
   torchvision.transforms.RandomVerticalFlip(p=1)
   ```

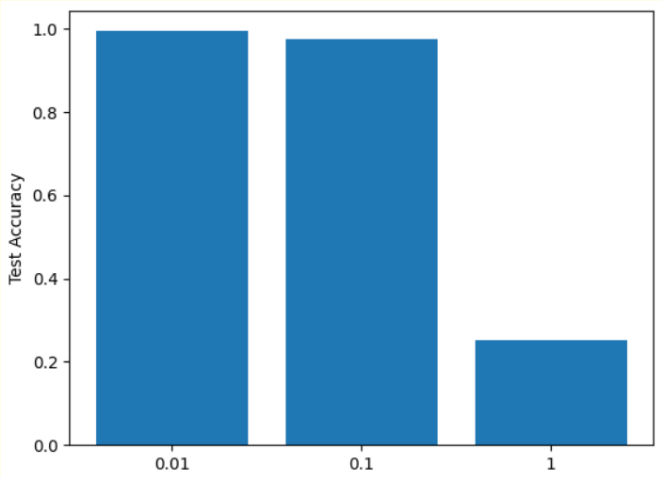
   Ans: We see that both types of flips lead to immense confusion in our model which decreases test accuracy by more than 60%!



   (f) (1 pt) test accuracy vs Gaussian noise. Try adding standard Gaussian noise to each test image with variance 0.01, 0.1, 1 and report the test accuracies. What is the effect?
   For instance, you may apply a user-defined lambda as a new transform t which adds Gaussian noise with variance say 0.01:

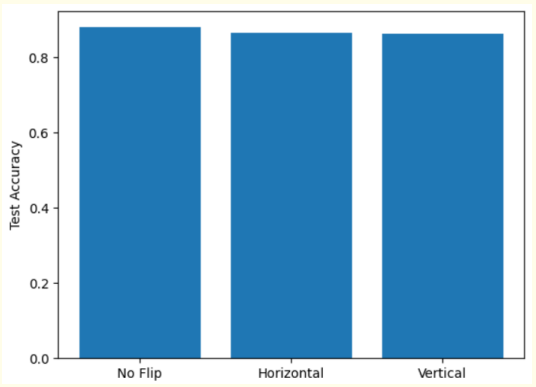
   ```
   t = torchvision.transforms.Lambda(lambda x : x + 0.1*torch.randn_like(x))
   ```

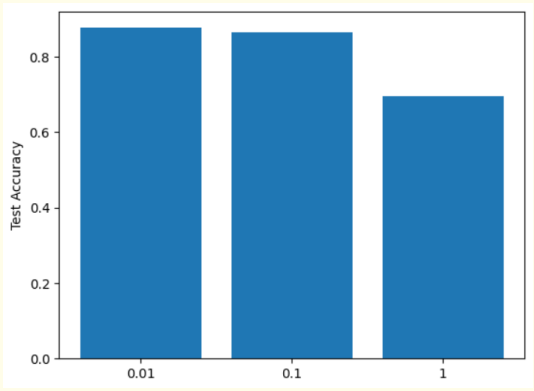
   Ans: We see that our model is very robust to the Gaussian noise for variance 0.01 and 0.1. However, for variance 1 our training accuracy is reduced to 16.58%, a meager 6% above random chance.



4. (2 pts) Lastly, let us verify the effect of regularization. Retrain your model with data augmentation and test again as in item 3 above (both e and f). Report the test accuracy and explain what kind of data augmentation you use in retraining.

Ans: By augmenting our training dataset with transforms.RandomAffine((-15,15)), transforms.RandomHorizontalFlip(p=0.5), transforms.Lambda(lambda x : x + 0.1*torch.randn_like(x)), and imagenet red channel normalization, our model builds great invariance to horizontal flips and Gaussian noise with variance 1. However, this comes at the cost of much poorer performance on vertical flips (a 20% decrease) and a smaller decrease in performance (7% each) for gaussian noise with variance 0.1 and 0.01.





Exercise 3: Regularization (4 pts)

Notation: For the vector $\mathbf{x}_i$, we use x_{ji} to denote its j-th element.

Overfitting to the training set is a big concern in machine learning. One simple remedy to avoid overfitting to any particular training data is through injecting noise: we randomly perturb each training data before feeding it into our machine learning algorithm. In this exercise you are going to prove that injecting noise to training data is essentially the same as adding some particular form of regularization. We use least-squares regression as an example, but the same idea extends to other models in machine learning almost effortlessly.

Recall that least-squares regression aims at solving:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2, \tag{6}$$

where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ are the training data. (For simplicity, we omit the bias term here.) Now, instead of using the given feature vector $\mathbf{x}_i$, we perturb it first by some independent noise $\boldsymbol{\epsilon}_i$ to get $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i)$, with different choices of the perturbation function f. Then, we solve the following **expected** least-squares regression problem:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n} \mathbf{E}[(y_i - \mathbf{w}^\top \tilde{\mathbf{x}}_i)^2], \tag{7}$$

where the expectation removes the randomness in $\tilde{\mathbf{x}}_i$ (due to the noise $\boldsymbol{\epsilon}_i$), and we treat $\mathbf{x}_i, y_i$ as fixed here. [To understand the expectation, think of n as so large that we have each data appearing repeatedly many times in our training set.]

1. (2 pts) Let $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i) = \mathbf{x}_i + \boldsymbol{\epsilon}_i$ where $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \lambda I)$ follows the standard Gaussian distribution. Simplify (7) as the usual least-squares regression (6), plus a familiar regularization function on $\mathbf{w}$.

   Ans: We clearly have the following identity:

$$\begin{aligned} \mathbf{E}[(y_i - \mathbf{w}^\top \tilde{\mathbf{x}}_i)^2] &= \mathbf{E}[(y_i - \mathbf{w}^\top \mathbf{x}_i - \mathbf{w}^\top \boldsymbol{\epsilon}_i)^2] && (8)\\ &= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 - \mathbf{E}[2(y_i - \mathbf{w}^\top \mathbf{x}_i)\mathbf{w}^\top \boldsymbol{\epsilon}_i] + \mathbf{E}[\mathbf{w}^\top \boldsymbol{\epsilon}_i]^2 && (9)\\ &= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 - 0 + \lambda \|\mathbf{w}\|_2^2, && (10) \end{aligned}$$

   where the last line follows from the assumption $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \lambda I)$. Thus, (7) is simply the usual ridge regression (with regularization constant $n\lambda$).

2. (2 pts) Let $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i) = \mathbf{x}_i \odot \boldsymbol{\epsilon}_i$, where $\odot$ denotes the element-wise product and $p\epsilon_{ji} \sim$ Bernoulli(p) independently for each j. That is, with probability $1-p$ we reset x_{ji} to 0 and with probability p we scale x_{ji} as x_{ji}/p. Note that for different training data $\mathbf{x}_i$, $\boldsymbol{\epsilon}_i$'s are independent. Simplify (7) as the usual least-squares regression (6), plus a different regularization function on $\mathbf{w}$. [This way of injecting noise, when applied to the weight vector $\mathbf{w}$ in a neural network, is known as Dropout (DropConnect).]

   Ans: As above, we have

$$\begin{aligned}
\mathbf{E}[(y_i - \mathbf{w}^\top \tilde{\mathbf{x}}_i)^2] &= \mathbf{E}[y_i - \mathbf{w}^\top \mathbf{x}_i - \mathbf{w}^\top (\mathbf{x}_i \odot (\boldsymbol{\epsilon}_i - 1))]^2 && (11)\\
&= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 - \mathbf{E}[2(y_i - \mathbf{w}^\top \mathbf{x}_i)\mathbf{w}^\top (\mathbf{x}_i \odot (\boldsymbol{\epsilon}_i - 1))] + \mathbf{E}[\mathbf{w}^\top (\mathbf{x}_i \odot (\boldsymbol{\epsilon}_i - 1))]^2 && (12)\\
&= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 - 0 + \mathbf{E}\left[\sum_j w_j x_{ij}(\epsilon_{ij} - 1)\right]^2 && (13)\\
&= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 + \sum_j \sum_k w_j w_k x_{ij} x_{ik} \mathbf{E}[(\epsilon_{ij} - 1)(\epsilon_{ik} - 1)] && (14)\\
&= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 + \sum_j w_j^2 x_{ij}^2 \mathbf{E}[(\epsilon_{ij} - 1)^2] && (15)\\
&= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 + \sum_j w_j^2 x_{ij}^2 [(1-p) + p(1/p - 1)^2] && (16)\\
&= (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 + \frac{1-p}{p} \sum_j w_j^2 x_{ij}^2. && (17)
\end{aligned}$$

   where we have used the fact that $p\epsilon_{ij} \sim$ Bernoulli(p) independently. Thus, (7) is simply the usual least squares regression, with the following regularization function:

$$\frac{1-p}{p} \sum_j \left(\sum_i x_{ij}^2\right) w_j^2, \tag{18}$$

   i.e., a weighted (squared) ℓ_2 regularizer. In particular, if we normalize the training data so that $\sum_i x_{ij}^2 \equiv 1$, then we reduce again to ridge regression (this time with regularization constant $\frac{1-p}{p}$).