CS480/680: Introduction to Machine Learning

Homework 3

Due: 11:59 pm, November 19, 2022, 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: 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 ∼ 5)

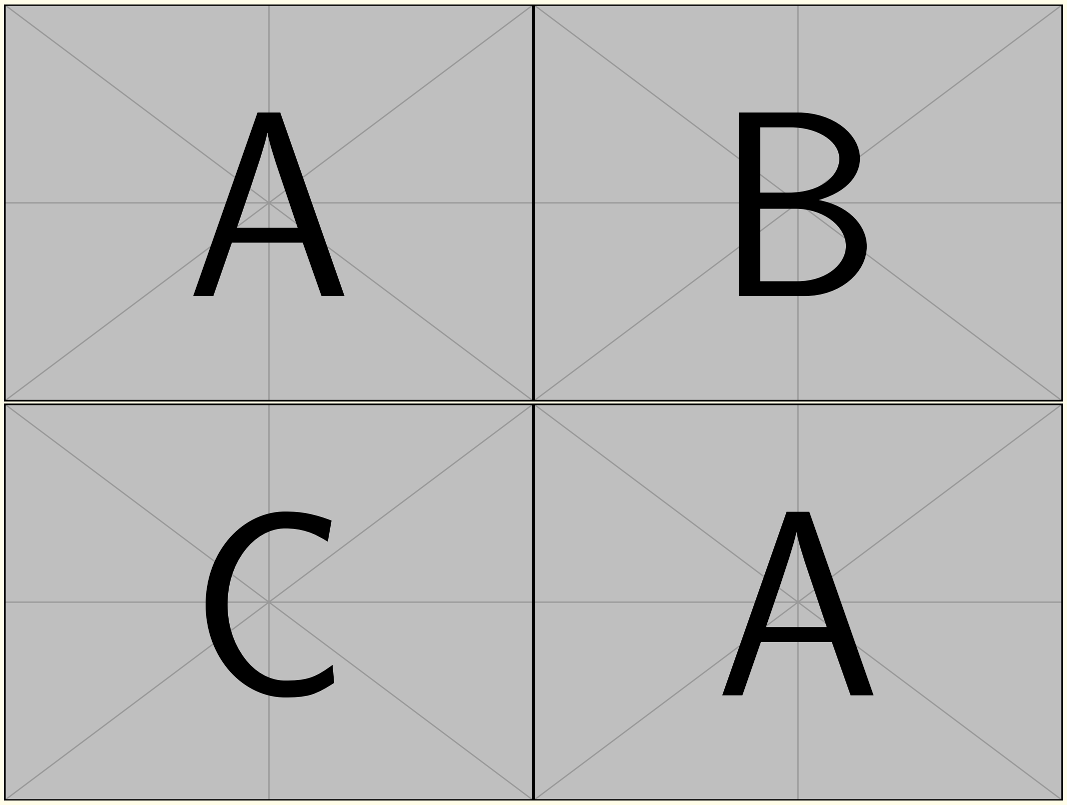
   (b) (1 pt) training accuracy vs the number of epochs

   (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:



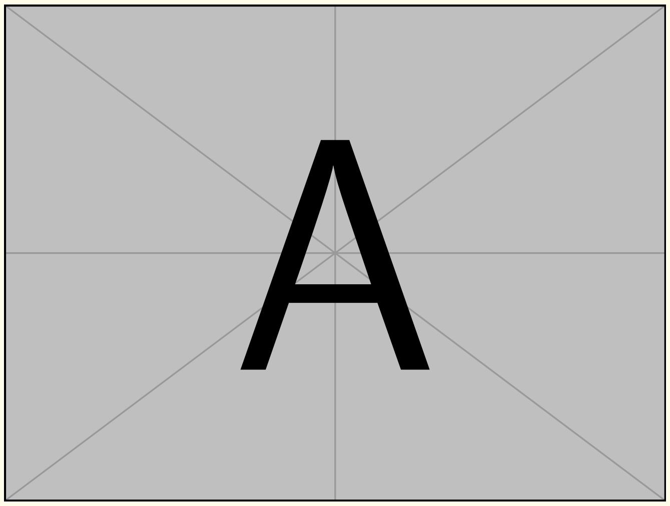
3. Then, it is time to inspect the generalization properties of your final model. Flip and blur the test set images using any python library of your choice, and complete the following:

   (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 can see that

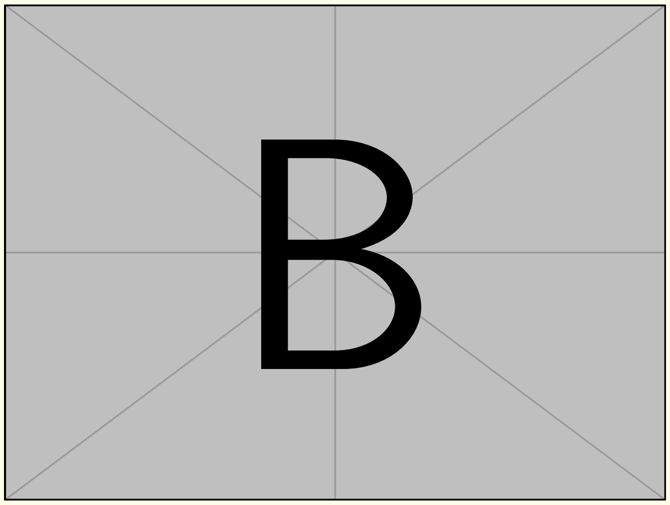


   (f) (1 pt) test accuracy vs Gaussian noise. Try adding standard Gaussian noise to each test image with variance 0.01, 0.1, 1 and report the test accuracies. What is the effect?

For instance, you may apply a user-defined lambda as a new transform t which adds Gaussian noise with variance say 0.01:

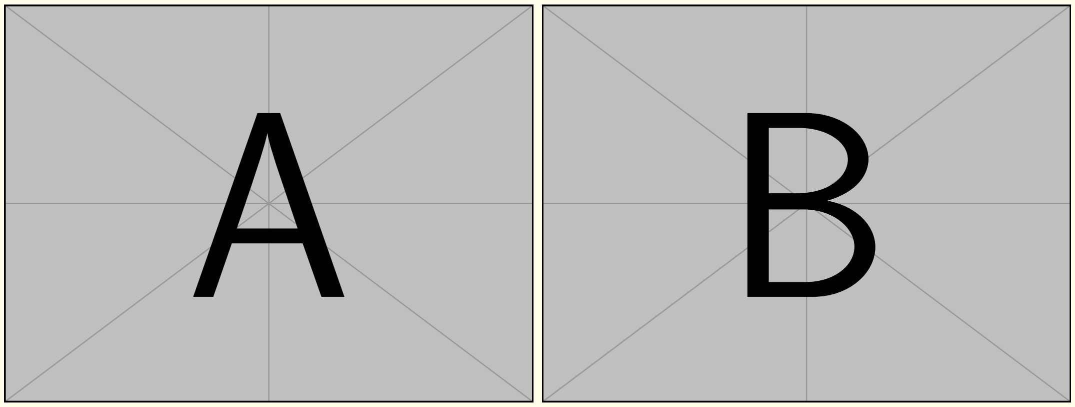
```
t = torchvision.transforms.Lambda(lambda x : x + 0.1*torch.randn_like(x))
```

Ans: We can see that



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: We can see that



Exercise 2: Graph Neural Networks (GNN) (8 pts)

You will need the following datasets to complete this exercise:

- Node classifications (small): **CiteSeer**

```
from torch_geometric.datasets import Planetoid
from torch_geometric.transforms import NormalizeFeatures
dataset = Planetoid(root='data/Planetoid', name='CiteSeer',
            transform=NormalizeFeatures())

print(f'Dataset: {dataset}:')
print('======================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')

data = dataset[0]
print(data)

## outputs:
# Dataset: CiteSeer():
# ======================
# Number of graphs: 1
# Number of features: 3703
# Number of classes: 6
# Data(x=[3327, 3703], edge_index=[2, 9104], y=[3327], train_mask=[3327],
                val_mask=[3327], test_mask=[3327])
```

- Graph classification. **TUDataset/MUTAG** (Training using GIN takes 2s on cpu for 20 epochs. Test Accuracy: 0.7895)

```
from torch_geometric.datasets import TUDataset
from torch_geometric.transforms import NormalizeFeatures
from torch_geometric.loader import DataLoader

# dataset = KarateClub(transform=NormalizeFeatures())

dataset = TUDataset(root='data/TUDataset', name='MUTAG',
                    transform=NormalizeFeatures())

print(f'Dataset: {dataset}:')
print('======================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')

train_dataset = dataset[: int(len(dataset) * 0.8)]
test_dataset = dataset[int(len(dataset) * 0.8): ]

print('==== train_dataset =====')
print(train_dataset)

print('==== test_dataset =====')
print(test_dataset)

## outputs:
# Dataset: MUTAG(188):
# ======================
# Number of graphs: 188
# Number of features: 7
# Number of classes: 2
# ==== train_dataset =====
# MUTAG(150)
# ==== test_dataset =====
# MUTAG(38)
```

We recommend the following hyperparameter setups:

```
For all tasks:

# hidden_dim: 64
# number of layers: 2
# activation: ReLU
# use Adam with learning rate = 0.01

For graph classification tasks, use # training epochs 30 would be sufficient

For node classification tasks, use # training epochs 200 would be sufficient
```

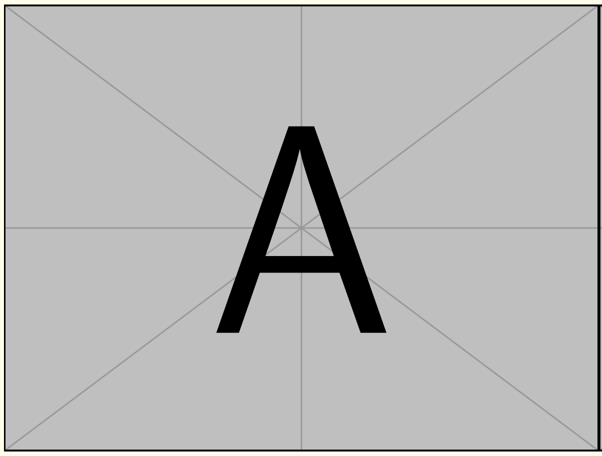
1. (node classification, 2 pts) Implement a GNN with backbone torch_geometric.nn.GCNConv and test your model on CiteSeer. Report the following:

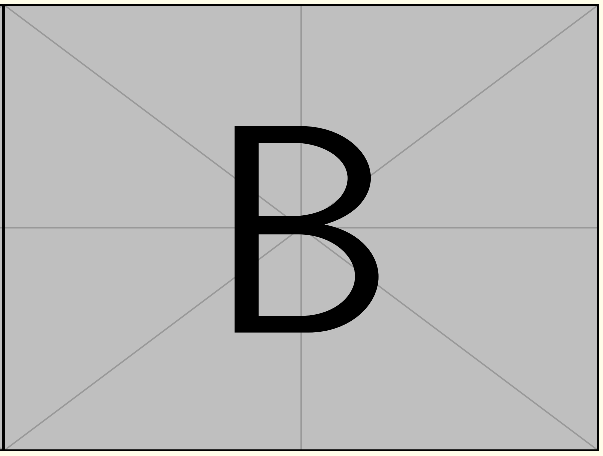
   - plot the training loss and classification error on training set w.r.t. iteration
   - plot the classification error on test set w.r.t. iteration
   - visualize the last layer node embeddings of the initialized model and the trained model. You may use the following code for visualization:

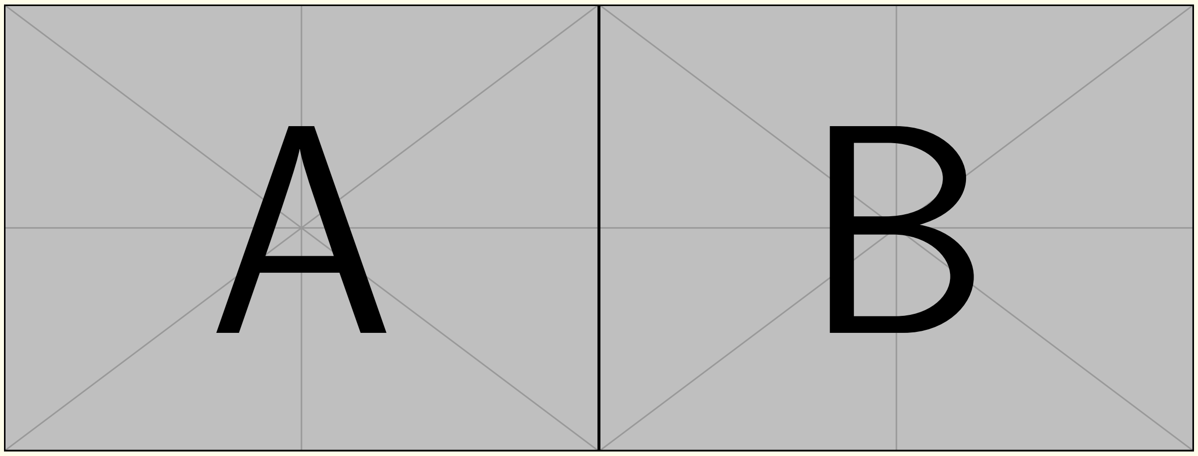
```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import torch

# emb: (nNodes, hidden_dim)
# node_type: (nNodes,). Entries are torch.int64 ranged from 0 to num_class - 1
def visualize(emb: torch.tensor, node_type: torch.tensor):
    z = TSNE(n_components=2).fit_transform(emb.detach().cpu().numpy())
    plt.figure(figsize=(10,10))
    plt.scatter(z[:, 0], z[:, 1], s=70, c=node_type, cmap="Set2")
    plt.show()
```

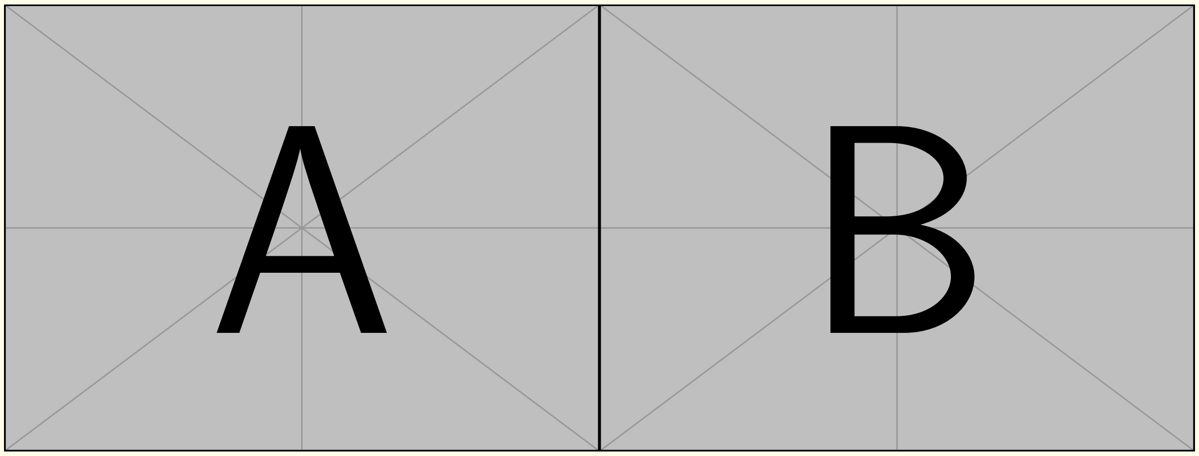
Ans:

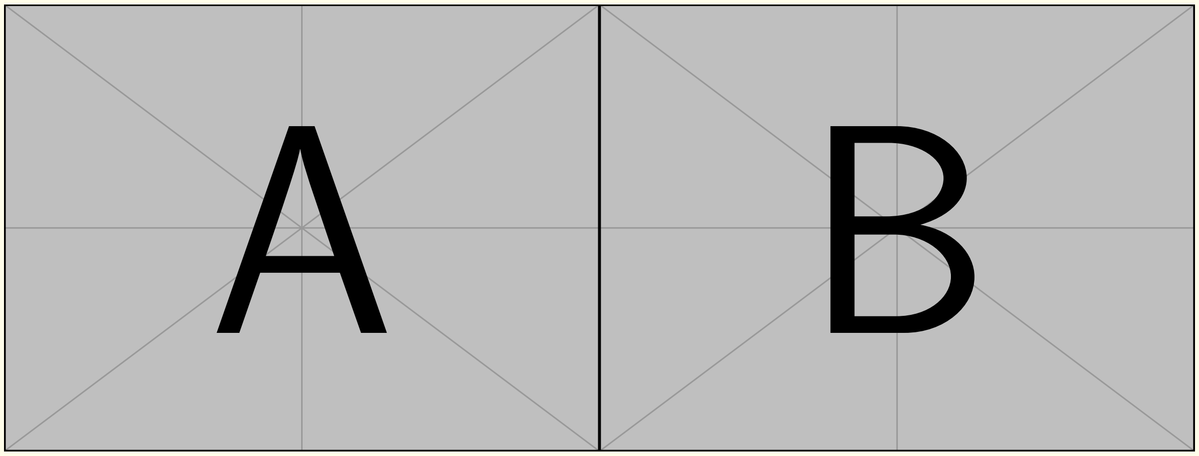




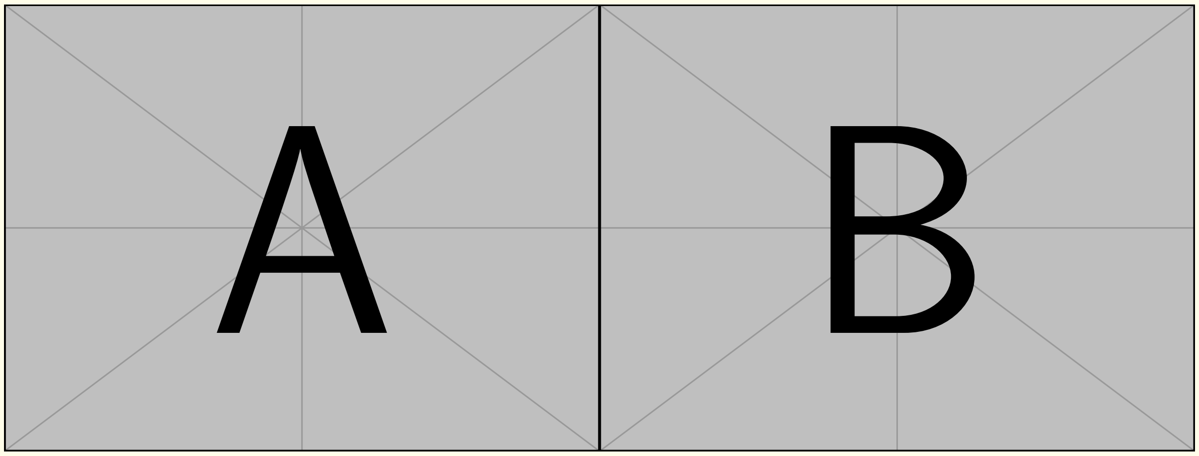


2. (node classification, 2 pts) Repeat the above task with backbone torch_geometric.nn.GINConv.
   [**Note**: Training over CiteSeer does not take much time (less than 1 min on cpu).]
   Ans:



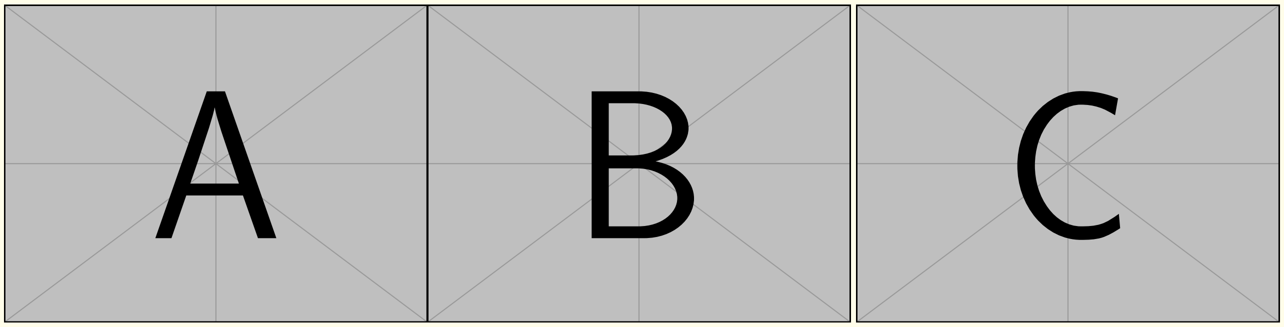


3. (graph classification, 2 pts) Implement a GNN with backbone torch_geometric.nn.GINConv. Plot the training loss, classification error on the training set and classification error on the test set w.r.t. iteration. (It is sufficient to implement the model with batch size one.) Include 1 figure for the two curves on training set and 1 figure for the curve on test set.
   Ans:



4. (graph classification with mini-batching, 2 pts) The graphs of a dataset may have different numbers of nodes, which makes it difficult to implement mini-batch by stacking node features of different graph samples. One workaround is to create a giant graph consisting of multiple isolated subgraphs, each associated to a graph sample. (See ADVANCED MINI-BATCHING for details.) In this question, you need to modify your implementation to support mini-batching. Plot the training loss, classification error on training set and classification error on test set w.r.t. iteration, for mini-batch size 4, 8 and 16, respectively. Include 1 figure for each mini-batch size.

Ans:



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 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:

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

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:

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

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 (2) as the usual least-squares regression (1), plus a familiar regularization function on $\mathbf{w}$.

   Ans:

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 (2) as the usual least-squares regression (1), plus a different regularization function on $\mathbf{w}$ (that may also depend on $\mathbf{x}$). [This way of injecting noise, when applied to the weight vector $\mathbf{w}$ in a neural network, is known as Dropout (DropConnect).]

   Ans: