Machine Learning for Molecular Engineering
3/7/10/20.C01 (U) 3/7/10/20.C51 (G)
Spring 2025

Problem Set #2

Date: March 27, 2025
Due: April 14, 2025 @ 3 pm

Instructions

- **Important:** This problem set requires a GPU. Before you start in Google Colab, find `Notebook Settings` under the `Edit` menu. In the `Hardware accelerator` drop-down, select a GPU as your hardware accelerator.
- Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `*.ipynb`. We highly suggest using Google Colab for this course and using this template for the second problem set. If you have not used Google Colab, you might find this example notebook helpful. After opening this template link, you should select "Save a copy in Drive" from the File menu. When you've completed the assignment, you can download the `*.ipynb` file from the File menu to upload to Canvas. Your submission should contain *all* of the code that is needed to fully answer the questions in this problem set when executed from top to bottom and should run without errors (unless intentional). Please ensure that your plots are visible in the output file and that you are printing any additional comments or variables as needed. Commenting your code in-line and adding any additional comments in markdown (as "Text cells") will help the grader understand your approach and award partial credit.
- Note that collaboration between students is encouraged, however every student must be responsible for all the work in their individual submission. I.e., discussing solutions and debugging together should be considered fruitful teamwork, however physically taking the keyboard of another student and directly entering code for their submission would be intellectually dishonest and as such is prohibited. You should first read through and attempt every problem before discussing with others. Please list the names of all collaborators in your .ipynb submission file.

Background

In part 1, you will build a sequence-based deep learning model to classify DNA binding sites based on a DNA sequence. DNA binding sites are fragments in DNA sequence where other molecules/proteins may bind for biological functions. Binding activities in DNA are also associated with important biological process like transcription, a process to turn your DNA into mRNA which is later used to synthesize proteins in your cells. The dataset you will work with is from ChIP sequencing which is an experimental method to probe protein interactions with DNA.

In part 2, you will explore one task with computer vision techniques using PyTorch. Computer vision uses computation to automatically process and analyze images to perform visual tasks that humans can do. In Python, a digital image can be loaded into an `numpy.array` or `torch.Tensor` with 3 dimensions (color, width, height) (Fig. 1).

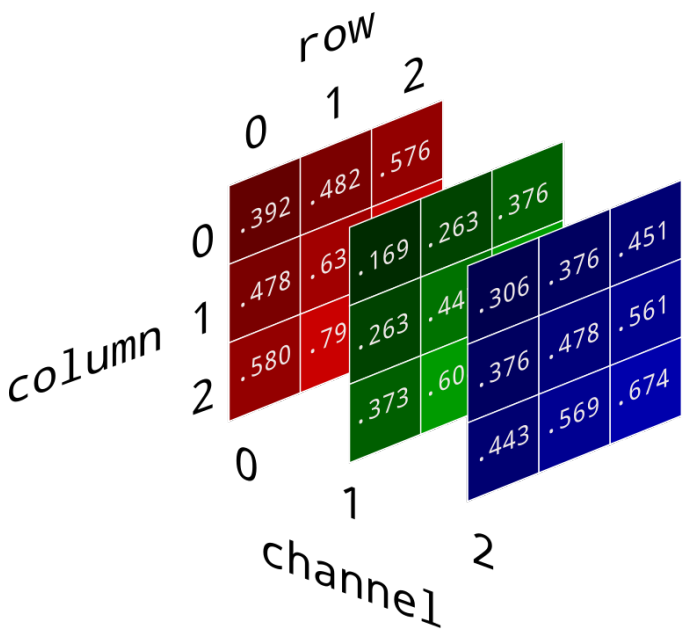

Figure 1: An image can be understood as a multi-dimensional array.

In materials and biological engineering, advanced imaging techniques using electron and optical microscopy can probe structures at multiple scales to better understand the underlying physical and chemical processes. When this information is processed in large batches, automated image processing programs using machine learning can improve efficiency and accuracy. You will use a U-net model to perform image segmentation on cell images to identify the locations and boundaries of nuclei in cells. For this task, you will be predicting an image filter. This will help scientists process images more efficiently to detect and understand different disease states of cells.

Part 1: Predicting DNA Binding Sites

In this part, you will write a pipeline that builds and trains a model to predict if a section of DNA sequence is a binding site for a protein. You will be provided a dataset of DNA sequences and a binary label 0/1 that indicates if the sequence binds to a protein or not. The sample data format is presented in Table 1.

DNA sequence DNA sequence	binder or not binder or not
'ATCGGGAA...'	1
'TGCAGTAT...'	0
...	...

Table 1: Dataframe snapshot from DNA binding data

This part will require you to implement a deep neural network in PyTorch, just as you did in Problem Set 1. If you need to, please go through the PyTorch tutorial here and the quickstart guide here.

You'll want a GPU for this part. Request one now.

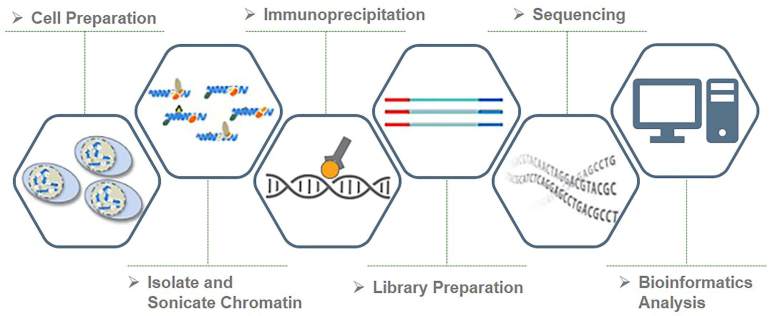

Figure 2: A typical ChIP workflow (source)

Part 1.1 (20 points) Build Datasets and DataLoaders in PyTorch

Now you will construct your data processing pipeline. First load the DNA sequence data with the code provided. Because the data is loaded as strings like "ATGTCA...", you will need to one-hot encode the DNA sequences into bit vectors of size 4 (corresponding to the 4 possible DNA bases A, T, G, and C) and split the data set into 80% train and 20% test. Further retrieve 10% of your training data as the validation set which will be used to check for training convergence. Because training a deep neural network takes time, so we won't perform cross validation or hyperparameter optimization.

Next, you need to construct your dataset and loader, with `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`. Mimic our code from Problem Set 1 and the Quickstart guide to write a `dataset` class. Essentially, you need to implement the `__getitem__()` method that return a sequence `x` and its corresponding label `y` given an `index`. Construct train, validation and test data loader with `batch_size=256`, using your train, validation and test datasets.

The concept of batching data might still be new to you, so take a moment to think about what it's doing and why (and optionally, do some brief research on your own about batching in deep learning). What is the benefit of batching your data into mini-batches versus using the entire dataset to optimize the model all at once?

Part 1.2 (20 points) Write an LSTM-based classifier

In this section, you will build a DNA sequence classifier with recurrent neural networks. In contrast to a standard feed forward neural network (i.e. a multi-layer perceptron), an RNN is a specialized network architecture that operates on sequential data such as text, DNA sequences, and stock market prices. When RNNs operate on an input at a current time point, it considers not just the value of the current input, but also what it has learned from the inputs it received previously through a hidden context vector. It is through this persistent context vector that the model architectures captures the sequential structure of our data.

Now, let us try to understand the math behind a RNN. First, recall that a single transformation of a neural network requires the following computation:

$$\sigma(Wx + b) \tag{1}$$

where x is the data, W is a learned matrix, σ is a element-wise nonlinear transformation, and b is a learned bias. If we applied this calculation to our sequential data as one vector, we will have ignored the sequential (e.g., time-varying) nature of our data. RNNs are designed to operate on

sequences one element at a time. Suppose our data has the form $\{x_0, x_1, x_2, ...x_t, ...x_T\}$; an RNN will sequentially construct a hidden embedding h_t for each x_t:

$$h_t = \sigma(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \tag{2}$$

In this equation t is the index for the element in your sequence. x_t is your sequence data, and the embedding h_t is the representation learned up to time t. At step t, $W_{ih}x_t + b_{ih}$ transforms the data x_t to a hidden state and $W_{hh}h_{(t-1)} + b_{hh}$ linearly transforms the hidden embedding of the *previous* state h_{t-1} to produce the next hidden state h_t. The two contributions (from h_{-1} and x_t) are added together and passed through a nonlinear activation function, σ. This procedure is applied to the sequence from x_0 all the way to x_T with the same set of parameters $W_{ih}, W_{hh}, b_{ih}, b_{hh}$ for $T + 1$ times.

For this pset, we will use a long short-term memory (LSTM) model, which is a variant of the vanilla RNN described above. LSTMs have been very successful in deep learning applications for natural language processing. In 2017, Facebook switched to using LSTMs to perform billions of language translations on user comments. In contrast to regular RNNs described above, LSTMs contain a gated 'memory cell' to store long-range correlation along the sequence. You are not required to understand the inner-workings of an LSTM for this assignment (although you are encouraged to read about it on your own here).

Now, let us talk about how to construct a classifier using LSTMs in PyTorch. You will first apply an LSTM on the one-hot encoded sequential data you prepared in the previous part using the `torch.nn.LSTM` class to construct a learned representation. Then, you will extract the parameterized hidden state after the final input, h_T, as an input to an MLP. The input dimension of the MLP should have the same dimension as the h_T and should output a scalar for each sample. Finally, a sigmoid activation function on the scalar output will provide a probability between 0 and 1 corresponding to the probability that this sequence is a DNA binding site. To summarize, there are three elements you need to construct:

- `lstm`: a LSTM module (`torch.nn.LSTM()`)
- `mlp`: a neural net (`torch.nn.Sequential()`, as in Problem Set 1). Use 2 hidden layers, each with tanh activation, and a final output layer with 1 output dimension.
- `sigmoid`: an element-wise sigmoid transformation (`torch.nn.Sigmoid()`)

In terms of mathematical equations:

$$\begin{aligned} \{h_t\}, h_T, c_T &= \texttt{lstm}(\{x_t\}) \\ \texttt{prob} &= \texttt{sigmoid}(\texttt{mlp}(h_T)) \end{aligned} \tag{3}$$

In this equation, x_t is the one-hot encoded DNA sequence, h_t are the hidden embeddings of the sequence, h_T is the embedding of the last element of the sequence. c_T is the cell state of the last element in the sequence. h_T is parameterized to encode the information of the entire sequence, and is the representation you need to use to feed into an MLP regressor.

For `torch.nn.LSTM`, please set `batch_first=True`, `num_layers=1`, `bidirectional=True`, and specify `hidden_size=hidden_dim` (the input to the `LSTMSeq` class). Since we're using a bidirectional LSTM, you'll have two hidden states produced (one for each direction); use the one corresponding to the backwards direction as input for the MLP, which will be $h_T[1]$ or $h_T[-1]$. For the `mlp`, use hidden dimensionality of `hidden_dim` in both layers. We have provided an example computation of LSTM-based classifier in the template file to help guide you, as well as some skeleton code.

Part 1.3 (20 points) Implement functions for training and testing

As in Problem Set 1, set up the Adam optimizer to train your model, with a learning rate `lr` of `1e-2`. In this problem, we will also be using a learning rate scheduler, specifically the scheduler `torch.optim.lr_scheduler.ReduceLROnPlateau`. This is a common strategy to prevent overfitting or overshooting a loss minimum by gradually reducing the learning rate during training. We have provided the scheduler for you in the skeleton code.

Next, write your training loop, following the Quickstart guide or the example we provided in Problem Set 1. Unlike before, this is a classification problem, so you will want to compute the binary cross entropy loss (`nn.functional.binary_cross_entropy()`) which requires you to input classification probabilities and the ground truth labels from your data. This should be implemented in the provided `train()` function, which already loops over minibatches for you. Record the loss of each batch and compute the average loss for all batches. Your `train()` function should return the average training loss.

Similarly, complete the `validate()` function to iterate over validation data and return the mean validation loss; remember to ensure that the validation function does not call the optimizer because the validation dataset is not used for training. You can make sure this is the case by calling `model.eval()` and turn on the `torch.no_grad()` context manager at the beginning of your validation function (we have implemented that for you). You will need to call `model.train()` to turn the gradients and optimizer updates back on in the `train()` function.

Finally, complete the `evaluate()` function. This function takes in data and produces the model's predictions, along with the true values for that dataset. You'll use it to evaluate your model's predictions on the test data and measure model performance.

Looping over the entire training data set once is called an *epoch*. The `train()` and `test()` function you will implement is only for one epoch, and should return a train loss and validation loss computed as averages over all the batches. Train and validate your model for 500 epochs; we have provided you with the code for this. This will take a few minutes.

Record the average train and validation loss for each epoch and plot these on a single graph, using our provided code. Finally report your test AUC score of your trained model on the test data. You would need to transform your prediction from `torch.Tensor` to `np.array` to compute ROC/AUC scores. As usual, you should use `sklearn.metrics.roc_auc_score`.

Part 2: Image Segmentation

In computer vision, image segmentation is a computational technique that partitions images into different parts for easier analysis and processing. This technique has been very useful for photo editing and videoconferencing software. For example, Zoom uses image segmentation to add virtual backgrounds to your video stream.

Image segmentation is also useful for analyzing cell images. Accurate and efficient segmentation of the cell nuclei improves the quality of disease states assessment for cells. In this question, you are given a dataset of cell images, each labeled with a pixel-wise mask to indicate if it belongs to a nucleus or not. The dataset comes from the 2018 Kaggle data science bowl.

Part 2.1 (20 points) Build Datasets and DataLoaders

Download the image datasets with code we provided. Make your own image dataset with a `__getitem__()` method that loads and processes images (`X`) and segmentation masks (`y`). Similar to what you did in Part 1, make an `ImageDataset` object as a subclass of `torch.utils.data.dataset`.

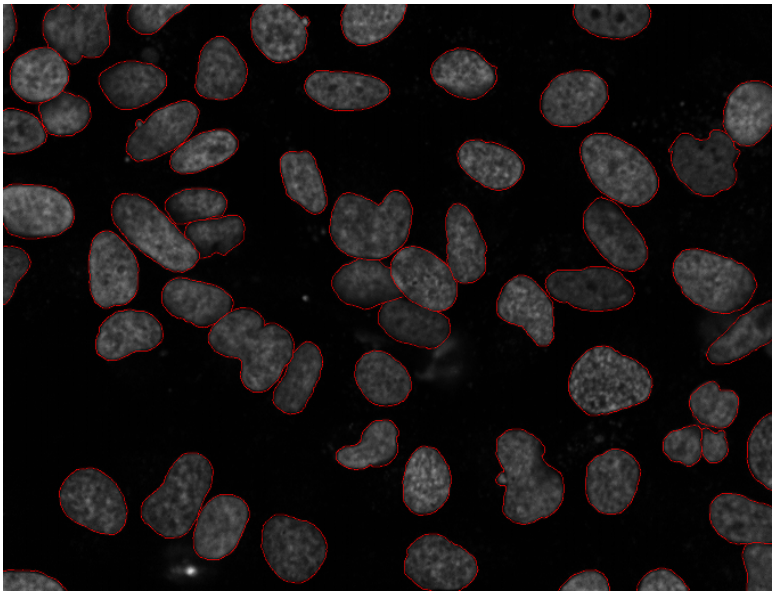

Figure 3: Applying image segmentation to cell images. The red lines marks the boundaries of cells.

We provided a function `load_img()` that reads the image and its mask given a path to the image file. The cell images you will load are gray scale. `load_img()` returns a tuple of images as `np.array`s with dimensions of 256×256 (cell image) and 256×256 (mask image). Like in part 1, you can convert them to 256×256 `torch.Tensor`s using `transforms.ToTensor()`.

Build your Datasets (70% train, 10% validation, and 20% test) and DataLoaders with a batch size of 4. To split your data, apply the split to the `paths` list generated in our code, and feed in the resulting `paths` to your `ImageDataset` class.

Ideally, your model predictions should respect rotational and translational invariance. While CNNs are invariant to translations, rotational symmetry is not embedded in a CNN architecture. [1]. To implicitly make your model adapt to different possible rotations, you should apply random rotation transformations to your training image samples. This technique is called **data augmentation**. Do you think random rotation data augmentation was necessary for our specific task of cell image segmentation? Also, why are CNNs invariant to translations? Briefly justify your answer.

Part 2.2 (20 points) Train a U-Net Model that Performs Image Segmentation

The model you will use is called U-Net which is a popular method for image segmentation[2]. Unlike a CNN model that successively contracts image information into a scalar output for classification, a U-Net model uses an encoder and a decoder to parameterize pixel-wise output to predict pixel-wise labels (Fig. 4). A final pixel-wise sigmoid transformation is used to ensure that the pixel-wise outputs are from 0 to 1. Mathematically, the model takes an image and outputs an segmentation filter ($f : \mathbb{R}^{256\times256} \rightarrow \mathbb{R}^{256\times256}$).

A common loss function for training an image segmentation model is the Dice loss [3] (from S**o**rensen–Dice coefficient). Let p_i be your predicted mask and t_i be your target mask, the Dice loss is defined as :

[1] It is possible to incorporate rotational equivariance in images, please see [1]

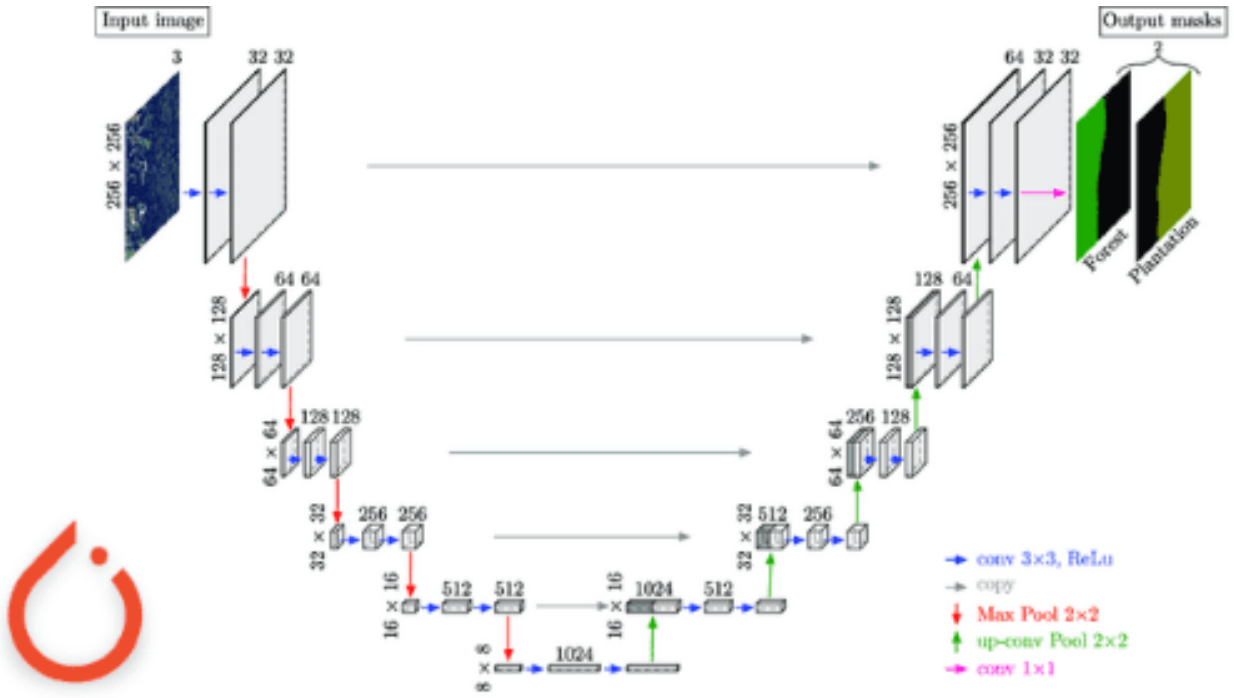

Figure 4: The U-Net architecture. The name stems from its U-shaped architecture. [2]

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_i p_i t_i + 1}{\sum_i p_i + \sum_i t_i + 1} \tag{4}$$

Please implement the loss function `dice_loss()` which takes the predicted mask and the labeled mask as arguments. +1 is included in the numerator and the denominator to deal with edge cases when there are no predicted mask in an image.

We have implemented a untrained U-Net model for you to use. Train your model for 20 epochs with a learning rate of `1e-3` to obtain a model that predicts the segmentation mask for each image. You will need to convert your images and masks to float tensors using `.to(float)` for them to train properly. We have provided an function `plot_seg()` for you to visually compare the predicted segmentation and ground truth segmentation; check the dimensions requirements of `plot_seg()` carefully. When your training is finished, show the segmentation results for three images in the test data using `plot_seg()`.

References

[1] Weiler, M., Geiger, M., Welling, M., Boomsma, W. & Cohen, T. 3d steerable cnns: Learning rotationally equivariant features in volumetric data. *arXiv preprint arXiv:1807.02547* (2018).

[2] Ronneberger, O., Fischer, P. & Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, 234–241 (Springer, 2015).

[3] Jadon, S. A survey of loss functions for semantic segmentation. In *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, 1–7 (IEEE, 2020).