Machine Learning for Molecular Engineering

Problem Set 3

Date: March 22, 2021
Due: 10 pm ET on Thursday, April 8, 2021

Instructions 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 first 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. You are encouraged to ask for clarification on Piazza if you find any of the question statements unclear or ambiguous; chances are, another student feels the same way!

To get started, open your Google Colab or Jupyter notebook starting from the problem set template file pset3.ipynb (direct Colab link). You will need to use the data files here.

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 some unsupervised learning techniques to decode the hidden message your data is trying to tell you without any labels.

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.

Part 1.1 (0 points) Understand an example deep learning workflow in PyTorch

In previous psets, you built your models with the sklearn library. Sklearn is great for using common algorithms and predefined model types, but cannot be used to build a fully custom model for end-to-end training. PyTorch (and other frameworks like TensorFlow, JAX or CNTK) is built for

DNA sequence	binder or not
‘ATCGGGAA...’	1
‘TGCAGTAT...’	0
...	...

Table 1: Dataframe snapshot from DNA binding data

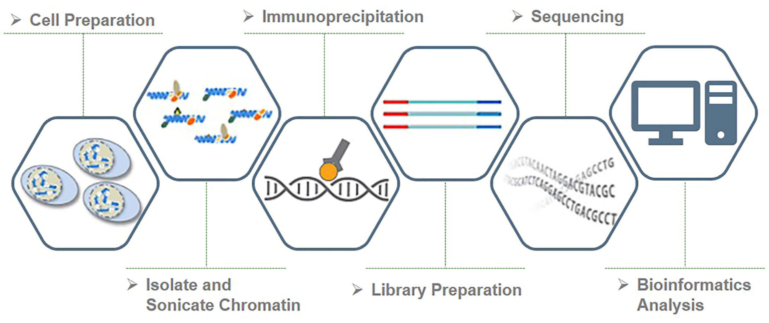

Figure 1: A typical ChIP workflow (source)

easily implementing deep learning models. The backbone of PyTorch is reverse-mode automatic differentation (AD), which is a programmatic framework to compute the gradients of your model output (a loss function) with respect to model parameters and inputs. Reverse-mode AD provides an easy way for machine learning researchers to construct models without having to think about explicit writing the gradient programs. These gradients are vital for optimizing (*i.e.* fitting) the weights efficiently. Please take a moment to go through this PyTorch tutorial here. PyTorch is pre-installed on Google Colab, so you can import it directly.

In this part, you are asked to classify if a DNA sequence is a protein binding site by building your our deep learning model using PyTorch's model and dataset APIs. You will also implement the necessary training and testing functions from scratch. To speed up training, you should be sure to use a graphical processing unit (GPU) on Colab. We've provided some skeleton code in the template file, but you will of course need to fill in the details. After this exercise, we hope you will be comfortable building your own machine learning workflow using PyTorch by adapting this and other sample codes.

Make sure you feel comfortable with the example Pytorch code before working on Part 1. There are plenty of examples available about how to build a deep learning pipeline in Pytorch. Try to follow an example code in the quick start guide here. We will go over this example in our clinic sessions.

Part 1.2 (5 points) Request a GPU on Google Colab

In Google Colab, find `Notebook Settings` under the `Edit` menu. In the `Hardware accelerator` drop-down, select GPU as your hardware accelerator. Print the number of GPUs using the code we provided to make sure you have successfully requested one GPU.

Part 1.3 (15 points) Build Datasets and DataLoaders in PyTorch

Code for processing data samples can get messy. From the perspective of good software engineering, it would good to modularize the dataloading procedure. In PyTorch, you will find two data-related classes, `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`, which are used to manage data when training a model. 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 Model takes time, so we won't perform cross validation or hyperparameter optimization as you did in previous psets. However, you might still want to do it in your final project if you decide to use a neural network based model.

Next, you need to construct your dataset and loader, with `torch.utils.data.Dataset` and `torch.utils.data.Dataloader`. A `Dataloader` is an object that iterates over your dataset and we provided an example of how to loop over your data. By using `Dataloder` you can utilize useful features such as memory pinning, multi-processing, and automatic batching when loading your data. You can also customize your data loading procedure to perform any on-the-fly calculations.

The center object in Pytorch is `torch.Tensor`. They are very similar to *np.array* objects you are already very familiar already. You can easily transform a `np.array` to `torch.Tensor`, and send `torch.Tensor` to a Nvidia GPU which is very fast at paralleled tensor operations. In the template notebook, we provided some examples of transforming `np.array` to `torch.Tensor` and back, and also moving `torch.Tensor` from CPUs to GPUs and back. A nice tutorial can be found here.

Follow the same dataset example in the Quickstart guide to write a `dataset` class. (We have also provided some skeleton code.) 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. Ensure your implementation is correct by iterating over your dataloader to retrieve batches of data (We have provided the code). What is the shape of each batch? And how many batches are there in your dataset?

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 1.4 (15 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 maths 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 transform 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 procedures 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).

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, extract the paramterized 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 (we recommend you using `torch.nn.Sequential()` to construct a model, you can follow the example here)
- `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` and specify `hidden_size=16`. We provided an example computation of LSTM-based classifier in the template file, please make sure you understand the procedure.

In some way, this exercise is like playing with Legos, you can define and test the three operations separately and combine them into a single `torch.nn.module` object following the example shown in the quick start guide. We have also provided the skeleton code. Modern deep learning frameworks like PyTorch and Keras try to make the definition of models as modular as possible and encourage code reuse.

Part 1.5 (25 points) Implement functions for training and testing

Now that you have defined the model architecture and a dataloader, you can use them to train the model. First you need to choose an optimizer that takes the gradients computed from backpropagation to update model parameters through a variant of stochastic gradient descent. We recommend you use the Adam optimizer (`torch.optim.Adam`), which is a very popular and robust choice. The optimizer object takes the model parameters which you can retrieve with $model.parameters()$. You also need to specify a starting learning rate `lr`. We recommend you set `lr` to `1e-2`.

Following the example in the Quickstart guide, use the `dataloader` to loop over your training data, using your classifier to predict the binding probabilities for each minibatch. Then use the model output 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. After calculating the loss, we need to compute $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ to find the gradient step direction. This is done for you when you call `loss.backward()`. The optimizer receives the gradient and optimize your model by then calling `optimizer.step()`. Always remember to remove the accumulated gradient from a previous calculation (minibatch) by calling `optimizer.zero_grad()` at the beginning of the minibatch processing. Please take a moment to reflect on this procedure: all of the complicated gradient calculation/update is nicely wrapped by these function calls so that you do not need to think about the numerics behind the scenes.

Define a `train()` function which loops over the minibatches in the train dataloader to iteratively update your model over the detail training set. Record the loss of each batch and compute the average loss for all batches. Your `train()` function should return the average training loss. Similarly, you need implement a separate `validate()` function which uses the validation dataloader to iterate over validation data and return the mean validation loss averaged over all the batches. In the validation function, you should *not* call the optimizer because the test 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.

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. Additionally, we also include a learning scheduler `torch.optim.lr_scheduler.ReduceLROnPlateau` to orchestrate the training process. It monitors the progression of your validation loss, and reduce the learning rate whenever it sees the validation loss stop improving. We have implemented this scheduler for you.

Record the average train and validate loss for each epoch and plot these on a single graph. 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: Dimension Reductions for Molecular Representations

In this part, you will be performing dimensionality reduction analysis on a set of candidate molecules tested against the Mitogen-Activated Protein Kinase, also called MAPK1 or ERK2. The data is from the DUD-E database.

Converting the complex structure and topology of a molecule into a vector representation is not straightforward. The cheminformatics community has developed a variety of algorithms to numerically encode the structural information of molecules. Morgan fingerprints, also known as circular fingerprints, represent the molecular graph as a unique bit string that encodes the molecular topology. You can understand it as a bit-vector categorical encoding with 1 for substructures present in the molecule and 0 for those not present. Some implementations of fingerprints use counts of how many times a substructure is present, instead of just 0s and 1s for absence/presence. Morgan fingerprints are typically high dimensional to maximize their descriptive power and sizes of 512, 1024 or 2048 are common (you can choose the number of bits when generating these fingerprints with the Morgan Algorithm). Morgan fingerprints provide a vector representation of molecules that is nearly unique or each molecule (although fingerprint collisions for non-identical molecules are possible), on which we can apply machine learning algorithms.

In this question, you will apply dimensionality reduction algorithms on a high-dimensional molecular fingerprint data we provided.

Part 2.1 (10 points) Principal Component Analysis on Molecular Fingerprints

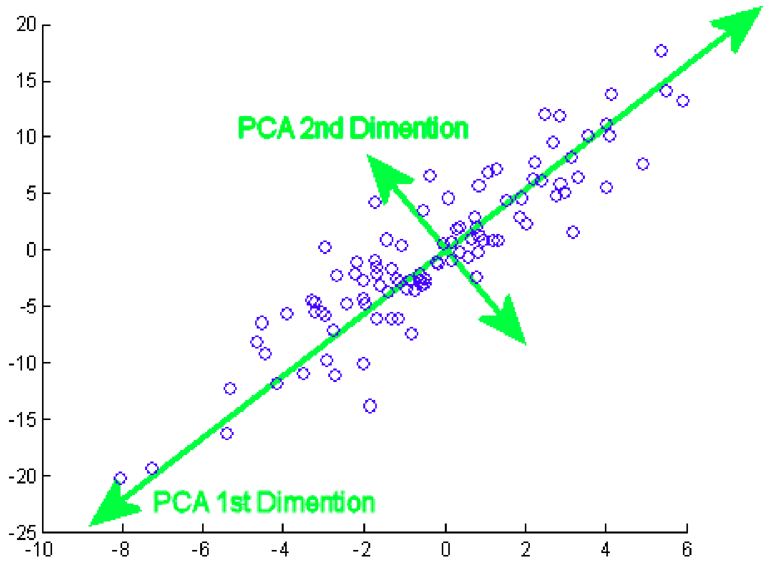

Figure 2: An example PCA analysis on a 2D dataset. The principal components are marked in green arrows. (source)

First load the fingerprint data arrays of 512 bit size and the molecule dataframe with the code we provided to you (The row in the fingerprint array is mapped one-to-one to the row of the molecule dataframe). The fingerprint is computed with the Morgan algorithm implemented in the RDkit package. In the dataframe, we also provided a binary label to indicate if the molecule tested active against the target protein.

Each molecular fingerprint spans a 512-dimensional space, so it is hard to make sense of it. We encourage you to think about the the size of the space that can be encoded in such representation.

Principal Component Analysis (PCA) is a dimensionality-reduction method that is often used to tackle high-dimensional, large data sets. In PCA, a large set of variables is transformer into a smaller one that contains as much of the information in the larger set as possible. PCA performs a linear transformation on your data by re-projecting in onto a new orthogonal basis. These new basis is chosen such that the fewest basis vectors best explain the variance of your data. Because of ease of visualization, it is common to just interrogate the first two dimensions (also called eigenvectors or components) of the PCA, but its possible to select additional dimensions.

You should perform PCA on the fingerprint data with `sklearn.decomposition.PCA()`. Use 100 components by setting `n_components=100`. This will reduce your original 512 dimensional space onto just a space of just 100 dimensions. Visualize the first two components of your data in a 2D scatter plot and color the active and inactive molecules differently. What is the total explained variance ratio of the first 100 principal components? (This is an estimate of how much information is preserved after this dimensionality reduction procedure) You can retrieve the ratios of explained variances by checking the `.explained_variance_ratio_` attribute of a `PCA()` object.

Inspect the scatter plot, do you observe any patterns in the distribution of active drugs in this 2D plot?

Part 2.2 (10 points) T-SNE analysis on Molecular Fingerprints

PCA applies linear transformations on your data, but may struggle to tease out subtle underlying patterns in your data. Non-linear transformations can distort or compress the dimensions much more effectively and can uncover much more complex structure in the dataset. There are two popular choices for non-linear dimension reduction; UMAP and t-SNE. T-Distributed Stochastic Neighbor Embedding(T-SNE) performs a stochastic non-linear transformation on your data and it is implemented in sklearn ((`sklearn.manifold.TSNE()`)). In general, t-sne is a random algorithm so you will have different looking embeddings every time; it is also sensitive to *perplexities* which balance attention between local and global aspects of your data during the optimization. Read the documentation for `TSNE()` to understand these options.

Apply T-SNE on your the 100-dimensional embedding you obtained from previous part with `n_components=2` with a perplexity value of 2, 50 and 500 separately and produce three labeled t-SNE plots with active and inactive drugs in different colors. Do you notice any differences in the clustering? Focus on the plots obtained with `perplexity=50`. In your 2D scatter plot, do you observe any patterns for the active drugs?

Part 2.3 (15 points) Are the low dimensional embeddings meaningful?

In this question, you will be asked to make sense of the extracted low dimensional embeddings and how the may relate to prediction performance in a supervised learning task. You will need to train a Random Forest classifier of active vs. inactive compounds and then relate the performance of this supervised to the embeddings you learned in the previous part.

Split your data into 10 splits using `sklearn.model_selection.KFold()` and train a Random Forest Classifier with 100 trees (`n_estimators=100`). As what you have done in pset 1, use `sklearn.ensemble.RandomForestClassifier`. Validate each split with models trained on the other 9 splits, and record your prediction in your dataframe or an array. Classify your predictions in 4 categories: True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). How powerful is the classifier you have trained? We can now interrogate the reasons behind its performance.

Make a scatter plot with the t-sne embeddings, and color each of the for categories above with different colors. Inspect your plot, do you observe any patterns in the prediction categories (TP, TN, FP, FN) on your t-sne plots?

Part 3: Feedback Survey

Part 3.1 (5 points) Please finish the feedback survey on our Canvas site