Assignment 2: A Multi-Layer Perceptron for Binary Classification on an Income Dataset

Deadline: October 4, 2018 at 11:59pm

This assignment must be done individually. This assignment is out of **100 points**. You can find the mark breakdown in each section. Grades will be based on the **correctness of the implementation, the plots, and the answers to the required questions in each section**.

Learning Objectives

In this assignment, you will learn how to use a Multi-Layer Perceptron (MLP) neural network to predict the level of income of a person based on specific information and data for each person. (The machine learning-word for that information is called the 'features' of the data). You'll learn some of the essential parts of the 'deep learning pipeline' of techniques. This work begins with a raw dataset and ends with a (hopefully) well-trained neural network, including the following tasks:

1. Load, clean and pre-process the dataset.
2. Obtain a basic understanding of the statistics of the dataset.
3. Converting the dataset into the right representation for training a neural network.
4. Train a multilayer perceptron to do binary classification.

This is also the first assignment in which you will write code to implement and run the network, based on *skeleton code* that is provided with this assignment.

What To Submit

You should hand in the following files:

- A PDF file `assignment2-writeup.pdf` containing your visualizations (plots) and answers to the written questions posed in this assignment. **Graded questions are located in each section**, indicated by the heading *Questions*. **Please answer these questions and include them in your report** under the appropriate section headings.
- Your code for parts 2, 3 and 4, which make use of the starter package files provided with this assignment. These are the files `main.py`, `dataset.py`, `model.py`, `util.py`and any other files that you created to make your code work. One note on this: this assignment asks you to explore different options with your code. As noted in Section 4.1, we require that you put all such options (which could be numerical parameters or flags that are strings) as *command line* options as described in that section.

You should upload all of these files **individually** to Quercus under this assignment.

1 The Deep Learning Work "Pipeline"

1.1 Introduction

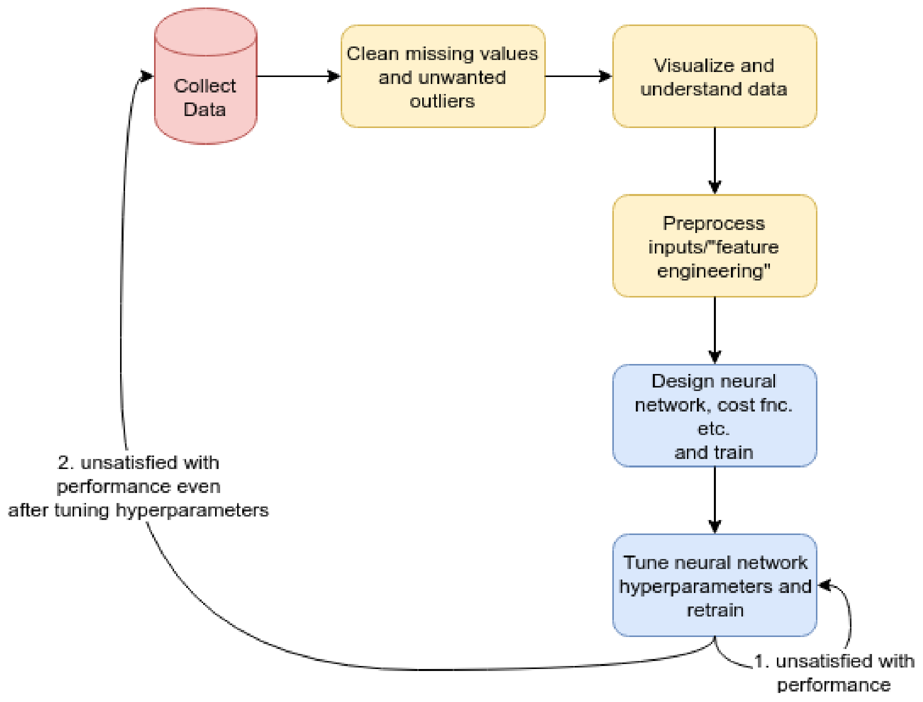
Collect Data
Clean missing values and unwanted outliers
Visualize and understand data
Preprocess inputs/"feature engineering"
Design neural network, cost fnc. etc. and train
Tune neural network hyperparameters and retrain
1. unsatisfied with performance
2. unsatisfied with performance even after tuning hyperparameters

Figure 1: The key tasks of the pipeline: data collection, data pre-processing, and neural network training.

The deep learning pipeline [Fig. 1] is the series of steps we take between going out into the world to collect data and training a neural network to determine something meaningful about the data. It can be roughly broken down into these steps:

1. Collecting data from wherever it comes from.
2. Cleaning the data. Two examples of this are getting rid of entries with missing values or removing unwanted outliers in the dataset. There are many other possible "cleaning" sub tasks.
3. Visualizing and understanding the data. Often we can gain valuable intuition and insight on the dataset even before training a model, by, for example measuring the frequency distributions of the data.
4. Pre-processing the inputs. A machine learning engineer must determine the appropriate representation of the data presented to the neural network. In some cases, some human-inspired manipulation of the data is necessary, which is often referred to as 'feature engineering.' Both of these can have a dramatic effect on the outcome.
5. Design the neural network architecture, cost function, optimizer, etc, as discussed in class.
6. Tune the hyperparameters of the neural network such as batch size, learning rate, etc, as discussed in class.

7. Once the neural network has been trained, we can decide to collect more data if we are unsatisfied with the results, repeating the pipeline.

For this assignment, we will work with the "adult income dataset" which provides the annual income of $\approx$ 48,000 American adults. The goal is to try and predict whether a person is a high income earner or a low income earner based on several factors supplied in the dataset. A high income earner is defined to be someone who earns over \$50,000 per year and a low income earner as someone who earns less than or equal to \$50,000 per year. This makes the problem to be solved a *binary classification* problem (which means there is only two possible classes), in which the goal is to learn to distinguish between these two classes. The dataset consists of 48,842 samples and 16 factors (also known as *features* for each person. Six of the features are continuous (real) values and eight of them are *categorical* values. These features include education level, race, field of work, for example.

For more information on the dataset itself, please take a look at: `http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html`.

1.2 Installation

You should have PyTorch and Pycharm already setup on your computer from Assignments 0 and 1. We will make use of the following additional packages for this assignment:

- pandas: `http://pandas.pydata.org/pandas-docs/stable/10min.html`.
- sklearn: `http://scikit-learn.org/stable/index.html`.

These packages should be already installed because they come with Anaconda. If, however, there is a problem importing these packages, you can also install using the following: `pip install pandas sklearn`. We suggest that you quickly read over the documentation pages of the libraries to get a rough idea of what they do.

The Pandas package will be used for data representation and visualization for this assignment. The package has a DataFrame class which will store the dataset and has some useful helper functions for manipulating and visualizing the data.

The sklearn package is a general purpose data processing library. Its functionality ranges from pre-processing data to training models such as logistic regression which we will also cover in this course. In this assignment, we will use sklearn to pre-process our data into a representation suitable for our neural network.

2 Data Pre-processing and Visualization (30 Points)

2.1 Loading the dataset

In the files you downloaded with this assignment, you can find the dataset located in the folder `data` in the file named `adult.csv`. You will also find starter code in several python files that you will use throughout this assignment. The first step: modify the provided code in `main.py` to use the pandas libraries to load the dataset into a variable called "data" using the function `pd.read_csv`. (after reviewing the pandas documentation to figure out how to do that).

2.2 Understanding the dataset (4 Points)

Once the dataset has been loaded, we will first do a few "sanity" checks to ensure that the data makes sense. The `pd.read_csv` function stores the dataset in a class called DataFrame. The DataFrame class has some useful fields and methods that will allow you to inspect the data:

- `.shape` field. The first number provided by this field is the number of rows (which is the number of samples, in this case the number of people) in the dataset; the second number is the number of columns (which is the number of features in each sample).
- `.columns` field. This field gives the name of each of the columns of the dataset. For more information on what the columns are and the values they can take, refer to `http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html`
- `.head()` method. This method will print out the first 5 rows of the dataset. On some computers, the `print` method may limit the number of columns printed. In this case, use the function `verbose_print` provided in the `[util.py]` to do the printing.
- `["income"].value_counts()`. We can access a specific feature using indexing notation e.g `["income"]`. The .value_counts() method will return the number of occurrences of each value of the column. Use this method to determine the number of high and low income earners.

Use the above fields and methods to print out each of the fields or to perform the method calls as described directly above. As you can see, there are 15 columns in the dataset: 14 features and 1 label called "income". In this label, a high income earner is represented by the string "$> 50K$" and a low income earner is represented by the string "$<= 50K$".

Questions

1. (2 points) How many high income earners are there in the dataset? How many low income earners?
2. (2 points) Is the dataset balanced (i.e. does it have roughly the same number of each of the classes/labels - in this case high income vs. low income)? What are some possible problems with training on an unbalanced dataset?

2.3 Cleaning (4 Points)

Datasets are often "dirty" when they are first collected, meaning that there could be missing or erroneous values or unwanted bias. In your final project and in this assignment, we will pre-process the data by removing missing values and unwanted outliers. Not doing so could be detrimental to the performance of the model. For the adult income dataset, missing values are indicated with the "?" string. Let's begin by figuring out how many missing values there are. Iterate over all the columns of the dataset and print out the number of "?" entries for each column. You will find the method .isin(["?"]).sum() helpful. We will need to remove any sample/row which has at least one "?" value. To do this, take advantage of the DataFrame class's indexing syntax: [data[data[column] != value]] returns a DataFrame with all the rows which do not contain the value "value" for column "column". Once you have done this, print out the shape of the dataset. How many rows are left?

Questions

1. (4 points) How many samples did we throw out? How many are left? Do you think this is a reasonable number of samples to throw out?

Sometimes there's a trade-off between the cleanliness of the dataset and the amount of data we have to train on. In an ideal world, we would have a very large dataset with no missing values or unwanted samples. However, collecting data from the "wild" is often time consuming and/or expensive so sometimes we might keep the samples with missing values in the dataset so at least there's more data to train on (but we would have to be very careful how we handle those samples in our model). For this assignment, our dataset is relatively large and the number of samples with missing values is small enough that we can just discard them.
A note on terminology: We will refer to rows/samples/entries and columns/features/field interchangeably.

2.4 Balancing the dataset (4 Points)

As indicated above in Section 2.2 (question 2) our dataset is unbalanced. There are a few ways to deal with this:

- Do nothing. We might be okay with the imbalance in the dataset as long as the final model does what we want, i.e. has high enough accuracy. Note we often care about per-class accuracy.

If ignoring the imbalance leads to a model that does poorly on the under-represented class, then we can try the following:

- Randomly drop data from the over-represented class so that all classes have the same number of samples.
- Enforce an equal number of samples from each class in each minibatch. This means either you train on each sample in the under-represented class more than once every training epoch; or train on only a subset of the samples in the over-represented class.
- Have a class-based-weight in your loss function. This would mean reducing the weight of samples from the over-represented class.
- Collect more data.

To keep things simple, in this lab we will simply drop data from the over-represented class.

1. (4 points) Use the DataFrame.sample (`https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sample.html`) method to balance the dataset. If there are n samples in class A and m samples in class B to begin with, you should be left with $\min(n, m)$ samples in both classes. **Hint: use the random_state argument to make sure the samples dropped are consistent between runs. Otherwise you will get slightly different results each run which can make debugging difficult.**

2.5 Visualization and understanding (10 Points)

Even before we train a model, there's a lot we can learn about the dataset. By understanding the characteristics of the data beforehand, we are better prepared to design better models to fit the data. For continuous valued features, it's often helpful to check basic statistics such as the mean or variance. Use the `.describe()` method of the DataFrame class to check the statistics of our data. Use the `verbose_print` method again. It will provide the following information about each feature:

- count: the number of samples with non-null value for this field.
- mean: the average value of this field.
- std: the standard deviation of this field.
- min: the minimum value of this field.
- 25%: the lowest 25 percentile of this field.

Questions

1. (1 point) What is the minimum age of individuals and the minimum number of hours worked per week for the dataset?

For the *categorical* features, it is useful to know the frequency of occurrence of each feature as one way to understand the dataset. Print the number of times each value of the feature occurs in the dataset. Notice that the most common features are: Private, White, HS-grad, etc. Pie charts and bar graphs are often helpful for visualizing categorical data. A `pie_chart` method has been provided for you in `util.py`. Visualize the first 3 categorical features using pie charts. **Include the plots in your report**. If you are interested, create the pie charts for the other categorical features as well.

Questions

1. (2 point) Are certain groups over or under-represented? For example, do you expect the results trained on this dataset to generalize well to different races?
2. (2 point) What other biases in the dataset can you find?

Next let's see how well we can classify salaries by looking at the features ourselves (human performance is often a useful benchmark for deep learning algorithms!). The `binary_bar_chart` method plots the number of occurrences of each value of a categorical feature. Furthermore, it plots the number of times (`feature==value and income` $> 50K$) in green and the number of times (`feature==value and income` $<= 50K$) in red. Plot the binary bar graph for the first 3 categorical features. **Include the plots in your report**. Investigate any other features you are curious about.

Questions

1. (2 points) Are there any features that are very useful for distinguishing between high and low salary earners? For example, is education a strong indicator of salary? How about race?

2. (3 points) Suppose you're told that someone has high school-level of education. How likely would they be to earn above $50K$? Given their education is "Bachelors" and nothing else about a person what is the best assumption as to whether they earn above or below $50K$?

3. (0 points) Now suppose you're given a new sample, think about how you would classify (by eye) the person as a high or low income earner. You don't need to come up with an explicit algorithm or submit anything for this question (but you can if you want to think about this problem).

2.6 Pre-processing (10 + 2 Points)

As discussed in class, the performance of neural networks can depend heavily on the representation of the input. Data that has continuous values (i.e. is a decimal number) should typically be normalized by the mean and standard deviation of the feature. For categorical data (i.e. where the feature is one of several specific *categories*), we should represent the feature as a 1-hot vector, as discussed in class.

1. (1 point) Extract the continuous features into a separate variable. To normalize the continuous features, subtract the average (computed with `.mean()`) and divide by the standard deviation (computed with `.std()`). Return a numpy representation of the data using the `.values` field.

2. (2 points) Currently the categorical features in our dataset are represented as strings. We will use the `LabelEncoder` class of sklearn to turn them into integers and use the `OneHotEncoder` class to convert the integers into one-hot vectors. For each categorical feature, call the `.fit_transform` to convert it into integer representation. Note this includes the "income" column.

3. (2 points) Next, extract the "income" column and store it as a separate variable (and convert it into a numpy array as well). We don't want to convert "income" into 1-hot. Remove the "income" field from the feature DataFrame completely to separate the features from the label. Use the `OneHotEncoder` class to convert each categorical feature from integer to one-hot.

4. (1 point) Finally, stitch the categorical and continuous features back together.

Questions

1. (2 points) What are some disadvantages of using an integer representation for categorical data?

2. (2 points) What are some disadvantages of using un-normalized continuous data?

Bonus: (2 points) create a separate dataset where continuous features are un-normalized and categorical features are represented as integers. Compare the performance of the neural network on this dataset versus the one created as above. Report any differences in your report. Note that the input size of the neural network will have to change with this different representation.

2.7 Train-validation split (3 Points)

This section has a function called `train_test_split`, which we use to create a **validation** set. Recall from class that the difference between a validation and test dataset is we are allowed to tune hyperparameters on the validation set but not the test set. We will tune hyperparameters on the dataset we create, therefore it's a "validation" set. (Don't think too much into the name of the function).

We will train on a fraction of our dataset and evaluate our model on the other portion. Make sure both features and labels are now numpy arrays. Then use the sklearn `train_test_split` method to divide the dataset into train and validation portions. `train_test_split` takes an argument `test_size` which controls the percentage of data allocated to the validation split (0 means all data will be in the train split and 1 means all data will be in the validation split). What value should we use for `test_size`? If we wanted as much data as possible to train on, then we would make `test_size` small. However, having a small validation set means our evaluation is not very reliable since we are evaluating only on a few samples. On the other hand, making `test_size` big gives us very reliable evaluation but worsens our model because there is less data to train on. Hence, there is an inherent trade off between data for training and the reliability of our evaluation. For larger datasets, we might be able to get away with making `test_size` a smaller percentage of the total dataset since the absolute number of samples in the validation split is reasonably large. For smaller datasets, we should probably use a larger percentage for `test_size`. In this assignment, we'll use a value of `test_size == 0.2`. Finally don't forget to provide a random seed to the `train_test_split` method to make sure your results are deterministically reproducible!

1. (2 points) Use `train_test_split` from sklearn to split the data into train and validation splits.
2. (1 point) Set a random seed for `train_test_split` so the data split can be reproducible.

3 Model Training (30 Points)

3.1 PyTorch Dataset (3 Points)

At this point we've cleaned our data, transformed it into the correct representations, and created the train and validation splits. It's time to start implementing and training our model. But before we do that, recall that in training with the batch stochastic gradient descent method, a random batch of data is sampled in every iteration. Luckily, we don't need to implement this data sampler ourselves; PyTorch has a class to do most of the work. We will use the `DataLoader` class to manage batch sampling in our training loop. First let's define a dataset class that extends PyTorch `data.Dataset`. Refer to `https://pytorch.org/tutorials/beginner/data_loading_tutorial.html` for an example of creating a dataset class and dataloading in PyTorch in general. This will be our container class that stores the data which `DataLoader` will retrieve.

1. (3 points) The code has been started for you in `dataset.py`; complete the implementation for the `AdultDataset` class.

3.2 DataLoader (4 Points)

1. (2 points) In `main.py` create a function called `load_data`. Instantiate two AdultDatasets for train and validation. Subsequently create two instances of the DataLoader class by passing

in the train and validation datasets. Specify the batch size using the batch_size argument. Make sure shuffle=True for the train loader!

Questions

1. (2 points) Why is it important to shuffle the data during training? What do you think would happen to the neural network during training if the dataset was collected in a particular order (e.g. ordered by income) and we didn't shuffle the data?

3.3 Model (4 Points)

We will be training a multilayer perceptron to do binary classification. A "Multilayer perceptron" or MLP for short, as described in class, is the original name for a basic neural network with only linear layers. In `model.py` complete the `MultiLayerPerceptron` class.

1. (2 points) The `init` method takes as input the dimension of the feature vector and specifies the layers that will be used in the model. Let's begin by using a single hidden layer with `Tanh` activation function. Make a guess for for a good size for the hidden layer – that is what the number of neurons in the hidden layer should be. What should the size of the output layer be?

2. (2 points) In the `forward` method, you define the complete architecture of the neural net, which is the sequence of operations to compute the output from the input. The output is the thing that goes into the loss function. You will be using the cross entropy loss function; since it expects as input a probability value between 0 and 1, you should apply a `Sigmoid` activation function after the last linear layer.

3.4 Loss Function and Optimizer (3 Points)

1. (3 Points) In `main.py` define a method called `load_model` which instantiates the MLP model and defines a loss function and an optimizer. For the loss function, make use of PyTorch's binary cross entropy function `BCELoss`. For the optimizer, we will use `optim.SGD` (stochastic gradient descent). `load_model` should take as input the learning rate which should be passed to the optimizer instantiation function. Make a guess at a good learning rate.

3.5 Training (8 Points)

In the `main` method call `load_data` and `load_model`.

1. (4 Points) Write a training loop that iterates over the total number of epochs to train for. For each epoch, iterate over the train loader and perform a gradient step. Some examples can be found here `https://pytorch.org/tutorials/beginner/data_loading_tutorial.html` and examples for training a network here `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`. Don't forget to zero your gradients at the beginning of every step!

2. (4 Points) In the inner loop, print the loss and number of correct predictions every N steps ($N = 10$ is a reasonable choice). Our MLP model outputs a value between 0 and 1. Therefore, if the output is greater than 0.5, we will declare that the model predicts a label of 1.

3.6 Validation (8 Points)

1. (4 points) Add a method called `evaluate` to `main.py`. `evaluate` should take as input the model as trained so far and validation loader and evaluate the accuracy of the model on the validation dataset. Call `evaluate` every N steps inside the training loop and print out the validation accuracy.

If you did everything correctly, we are now ready to train our neural network. A working model should have validation accuracy greater than 80%. If your model doesn't work quite yet, it may be because of the hyperparameters. Play around with the batch size, MLP hidden size, and learning rate to maximize your validation accuracy.

2. (3 points) Using Matplotlib, make a plot of the train accuracy and a plot of the validation accuracy as a function of the number of gradient steps. Refer to `train.py` and `plot_train.py` from Assignment 1 to see how to plot using Matplotlib. For the training accuracy, you don't need to report the accuracy of the entire train dataset (this will take too long to run), just report the accuracy on the last N steps. **Include the plots in your report**. Report the batch size, MLP hidden size, and learning rate that you used. If you aren't getting over 80% validation accuracy, don't worry we will tune the hyperparameters in the next section. What validation accuracy does your model achieve?

3. (1 point) The performance of your network may oscillate. Add smoothed plots to your graphs better visualize the results. You can use any smoothing algorithm you wish; one possibility is the `savgol_filter` from `scipy.signal`. Play around with the smoothing algorithm's parameters to get a plot that is less noisy but maintain the essence of the original plot. If N is small, the training error may oscillate a lot. In this case, you can apply high smoothing to the train plot or increase N and batch size.

4 Hyperparameters (30 Points)

The performance of neural networks is highly dependent on hyperparameters: these are parameters that are manually set by the ML engineer rather than learned by the model. Note that hyperparameters can be both real-valued or categorical variables. For our model, the hyperparameters include: learning rate, batch size, activation function, number of layers, loss function, and regularization. It is often stated in deep learning that choosing good hyperparameters is an art more than a science. ML engineers typically use their prior experience to narrow down the hyperparameters to a reasonable range before conducting some sort of search. This assignment, together with the last, is likely the beginning of that experience for you! Two commonly used search policies are:

- Random: we simply choose a random value within some range $[a, b]$ for continuous hyperparameters or from some set $\{a, b, \dots\}$ for categorical hyperparameters. We repeat until the model achieves satisfactory performance.

- Grid search: for continuous hyperparameters, we start at the lower bound a and increase the value of the hyperparameter by some increment i until upper bound b. Finally we take the value that gave us the best performance. For categorical hyperparameters, we try all possible values and take the best one. This may not be feasible, given the number of possibilities and the time available.

4.1 Learning rate (7 Points)

Here you will try to understand the effect of learning rate. Use a hidden layer size of 64, and a batch size of 64. Use a Grid search to find the best learning rate. **You will notice that in main.py there is a –lr flag to set the learning rate. To keep things organized, please use flags for any hyperparameters you introduce to the model. For discrete parameters like activation function you can use a string flag, which model.py will use.** Vary the learning rate from 1e-3 to 1e3 (in steps that vary by a factor of 10, i.e. 1e-3, 1e-2, 1-e1, ...)., retraining the model each time. This is because models are typically only sensitive to a multiplicative change in the learning rate (be careful: that this might not apply to other hyperparameters!). One could also use a smaller multiplication constant than 10 (e.g. 2, 5) but it might not make much of a difference for this particular model.

1. (1 point) Report the highest validation accuracy for each learning rate in a table. Include this table in your report.
2. (3 points) Make a plot of training accuracy (achieved on the training data) and validation accuracy (achieved on the validation data) as a function of the number of steps for learning rate equal to 0.01, 1, and 100. **Include the plots in your report**.

Questions

1. (1 point) Which learning rate works the best?
2. (2 points) What happens if the learning rate is too low? Too high?

4.2 Number of epochs (2 Points)

The number of epochs to train for might be the easiest hyperparameter to set. If we set the number of epochs too small, the model won't have the opportunity obtain a good accuracy. On the other hand, it doesn't hurt to train for too long; the only downside is a waste of time and electrical energy. In `main.py` the default number of epochs has been set to 100. Throughout the rest of the assignment, please adjust the number of epochs yourself to find a good setting. For example, if the learning rate is small, then we probably need to train for longer to compensate. In general:

- if your validation accuracy is still increasing when the run finishes, the training process has too few epochs
- if your validation accuracy is flat for a long time near the end of training or going down, the training process has too many epochs

Generally it's a good idea to lean on the side of training too many epochs rather than too few. You can never be completely sure your model won't suddenly do something interesting!

1. (2 points) Make sure in the plots you're submitting, you've run the model for long enough.

4.3 Batch size (9 Points)

Next, consider the effects of batch size. Use the best learning rate you found in the previous section.

1. (1 point) For the following batch sizes: 1, 64, and 17932 (the size of the entire training dataset), make plots of the train and validation accuracies versus the number of steps. **Include the plots in your report**. Note you will need to change N, the frequency at which you record train and val error. Note[2] running batch size 1 will take a while, you can try reducing the number of epochs.

2. (1 point) Measure the time of your training loop and make a plot of the train and validation accuracy versus time instead of steps. **Include the plots in your report**.

Questions

1. (1 point) Which batch size gives the highest validation accuracy?

2. (2 point) Which batch size is fastest in reaching a high validation accuracy in terms of the number of steps? Which batch size is fastest in reaching its maximum validation accuracy in terms of time?

3. (2 point) What happens if the batch size is too low? Too high?

4. (2 points) State the advantages and disadvantages of using a small batch size. Do the same for large batch size. Make a general statement about the value of batch size to use (relative to 1 and the size of the dataset).

Hint: The difference between using a large and small batch size may be more apparent when you reduce the amount of smoothing in your graphs (you may need to vary the smoothing to see the effects).

4.4 Under-fitting (4 Points)

What happens when the model is too small, that is has too few parameters/neurons? This might be kind of obvious but let's check. Use the best learning rate and batch size you have found so far (in terms of validation accuracy) to do the following:

1. (2 points) Change your MLP model to have no hidden layers and only one linear layer that maps the input directly to output (but still apply the `Sigmoid` function before the output). Make a plot of the train and validation accuracy versus the number of steps. **Include the plots in your report**.

Questions

1. (2 points) What validation accuracy does the small model achieve? How does this compare to the best model you've trained so far? Is this model underfitting? (Note you don't have to answer "yes" just because this section is called "under-fitting", think!)

In general the term “under-fitting” refers to making the model smaller than it should be. If a model has underfit, you could increase its size to obtain higher validation accuracy. Usually you can tell you are underfitting because the training and validation accuracy measurements are about the same. It turns out you usually want the training accuracy to be just slightly higher than the validation accuracy.

4.5 Over-fitting (4 Points)

What happens when the model is too big? Use the best learning rate and batch size you have found so far (in terms of validation accuracy).

1. (2 points) Change your MLP model to have 3 hidden layers, each of size 64 (note that training might take a while longer). Make a plot of the train and validation accuracy versus the number of steps. **Include the plots in your report**.

Questions

1. (2 points) What validation accuracy does the large model achieve? How does this compare to the best model you’ve trained so far? Is this model over-fitting?

In general the term “over-fitting” refers to making the model larger than it should be. If a model has overfit, the train accuracy is significantly greater than the validation accuracy. We say the model has bad “generalization.” This means the model can do well on the training set but will not generalize to data not seen during training. The most extreme example would be to use a lookup table in place of a neural network as the model and simply match samples in the validation set to samples in the training set and assign a prediction that way (if the sample doesn’t appear in the training set, the model assign a random sample). You would not expect the lookup table to generalize well. When people talk about over-fitting, they often mean you could decrease the size of the model to obtain higher validation accuracy.

4.6 Activation function (4 Points)

1. (3 points) Take the best model you’ve trained so far. Replace the `Tanh` activation function in your model with a `ReLu`, then a `Sigmoid` activation function. Plot the train and the validation accuracy of all three models on the same graph. Do you notice any qualitative or quantitative differences? **Include the plots in your report**.
2. (1 point) Measure the time of each of the training runs with each activation function. Include a table of the times in your report. Is there a difference between the activation functions in terms of how long they take to run?

5 Hyperparameter search (4 Points)

In this last part of the assignment, use what you’ve learned about hyperparameter tuning to find the best configuration for training your network. Make sure to write down the value of the random seed that you used and the hyperparameter values so the results are reproducible. You are allowed to use any techniques you might have read about such as other loss functions, optimization functions, regularization, etc.

6 Feedback (2 Points)

1. Describe your experience with assignment 2:
   (a) How much time did you spend on assignment 2?
   (b) What did you find challenging?
   (c) What did you enjoy?
   (d) What did you find confusing?
   (e) What was helpful?