Assignment 1. Music Century Classification

Submited By: Amit Efraim 034591552, Yuval Haitman 204087274

Deadline: Sunday, April 4th, by 9pm.

Submission: Submit a PDF export of the completed notebook as well as the ipynb file.

In this assignment, we will build models to predict which **century** a piece of music was released. We will be using the "YearPredictionMSD Data Set" based on the Million Song Dataset. The data is available to download from the UCI Machine Learning Repository. Here are some links about the data:

- https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd
- http://millionsongdataset.com/pages/tasks-demos/#yearrecognition

Note that you are note allowed to import additional packages **(especially not PyTorch)**. One of the objectives is to understand how the training procedure actually operates, before working with PyTorch's autograd engine which does it all for us.

Question 1. Data (21%)

Start by setting up a Google Colab notebook in which to do your work. Since you are working with a partner, you might find this link helpful:

- https://colab.research.google.com/github/googlecolab/colabtools/blob/master/notebooks/colab-github-demo.ipynb

The recommended way to work together is pair coding, where you and your partner are sitting together and writing code together.

To process and read the data, we use the popular `pandas` package for data analysis.

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
```

Now that your notebook is set up, we can load the data into the notebook. The code below provides two ways of loading the data: directly from the internet, or through mounting Google Drive. The first method is easier but slower, and the second method is a bit involved at first, but can save you time later on. You will need to mount Google Drive for later assignments, so we recommend figuring how to do that now.

Here are some resources to help you get started:

- http.://colab.research.google.com/notebooks/io.ipynb

```
load_from_drive = False

if not load_from_drive:
  csv_path = "http://archive.ics.uci.edu/ml/machine-learning-databases/00203/YearPredictic
else:
  from google.colab import drive
  drive.mount('/content/gdrive')
  csv_path = '/content/gdrive/My Drive/YearPredictionMSD.txt.zip' # TODO - UPDATE ME WITH

t_label = ["year"]
x_labels = ["var%d" % i for i in range(1, 91)]
df = pandas.read_csv(csv_path, names=t_label + x_labels)
```

Now that the data is loaded to your Colab notebook, you should be able to display the Pandas DataFrame `df` as a table:

```
df
```

	year	var1	var2	var3	var4	var5	var6	var7
0	2001	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202
1	2001	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630
2	2001	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872
3	2001	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097
4	2001	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207
...	...	...	...	...	...	...	...	...
515340	2006	51.28467	45.88068	22.19582	-5.53319	-3.61835	-16.36914	2.12652
515341	2006	49.87870	37.93125	18.65987	-3.63581	-27.75665	-18.52988	7.76108
515342	2006	45.12852	12.65758	-38.72018	8.80882	-29.29985	-2.28706	-18.40424
515343	2006	44.16614	32.38368	-3.34971	-2.49165	-19.59278	-18.67098	8.78428
515344	2005	51.85726	59.11655	26.39436	-5.46030	-20.69012	-19.95528	-6.72771

515345 rows × 91 columns

To set up our data for classification, we'll use the "year" field to represent whether a song was released in the 20-th century. In our case `df["year"]` will be 1 if the year was released after 2000, and 0 otherwise.

```
df["year"] = df["year"].map(lambda x: int(x > 2000))
```

```
df.head(20)
```

	year	var1	var2	var3	var4	var5	var6	var7	
0	1	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-1
1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-1
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928	20.67345	-
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206	-12.04282	-
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491	8.11871	-
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413	1.71392	-
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302	-6.95536	-1
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158	8.05718	-
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000	-0.10231	-
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439	5.51947	
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979	4.87374	-1
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916	-0.85322	-
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341	-1.31928	-
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377	-33.36843	-2
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169	11.75252	-
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457	1.02668	-
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028	-16.40421	-

20 rows × 91 columns

Part (a) -- 7%

The data set description text asks us to respect the below train/test split to avoid the "producer effect". That is, we want to make sure that no song from a single artist ends up in both the training and test set.

Explain why it would be problematic to have some songs from an artist in the training set, and other songs from the same artist in the test set. (Hint: Remember that we want our test accuracy to predict how well the model will perform in practice on a song it hasn't learned about.)

```
df_train = df[:463715]
df_test = df[463715:]

# convert to numpy
```

```
train_xs = df_train[x_labels].to_numpy()
train_ts = df_train[t_label].to_numpy()
test_xs = df_test[x_labels].to_numpy()
test_ts = df_test[t_label].to_numpy()

# Answer:
# Usualy, differnt songs from the same artist will have a messure of similarity, therofre
# may encourage overffiting in our trained model. Moreover, recalling the NN task is ident
# the same artist both on train and test we may harm the accuracy results on test when dea
```

Part (b) -- 7%

It can be beneficial to **normalize** the columns, so that each column (feature) has the *same* mean and standard deviation.

```
feature_means = df_train.mean()[1:].to_numpy() # the [1:] removes the mean of the "year" f
feature_stds  = df_train.std()[1:].to_numpy()

train_norm_xs = (train_xs - feature_means) / feature_stds
test_norm_xs = (test_xs - feature_means) / feature_stds
```

Notice how in our code, we normalized the test set using the *training data means and standard deviations*. This is *not* a bug.

Explain why it would be improper to compute and use test set means and standard deviations. (Hint: Remember what we want to use the test accuracy to measure.)

```
# Answer:
#Since the test dataset is unknown and will be presented to the model for in the inference
# dealing with real world application, it cannot be assumed that all the test data will be
# Therefore, it is invalid to normalized all the test dataset in advance.
# Under the reasonable assumption that the test will have similar distribution to the trai
# we can normalize the test dataset with the train mean and std which will be a good estim
```

Part (c) -- 7%

Finally, we'll move some of the data in our training set into a validation set.

Explain why we should limit how many times we use the test set, and that we should use the validation set during the model building process.

```
# shuffle the training set
reindex = np.random.permutation(len(train_xs))
train_xs = train_xs[reindex]
train_norm_xs = train_norm_xs[reindex]
train_ts = train_ts[reindex]
train_size = 50000 #50000
# use the first 50000 elements of `train_xs` as the validation set
```

```
train_xs, val_xs           = train_xs[train_size:], train_xs[:train_size]
train_norm_xs, val_norm_xs = train_norm_xs[train_size:], train_norm_xs[:train_size]
train_ts, val_ts           = train_ts[train_size:], train_ts[:train_size]

# Answer:
# Since the test dataset will be introduced to the model only on the inference step,
# we can not evaluate our model and test its accuracy on the test dataset.
# Checking accuracy in the train step is crucial for the evaluation and understanding of t
# that's why we take a random small portion from the train dataset which will be used as t
# The validation dataset will not be used for the weight training, instead it will be usec
# were never introduced to the model for accuracy check in the training step.
```

Part 2. Classification (79%)

We will first build a *classification* model to perform decade classification. These helper functions are written for you. All other code that you write in this section should be vectorized whenever possible (i.e., avoid unnecessary loops).

```
def sigmoid(z):
  return 1 / (1 + np.exp(-z))

def cross_entropy(t, y):
  return -t * np.log(y) - (1 - t) * np.log(1 - y)

def cost(y, t):
  return np.mean(cross_entropy(t, y))

def get_accuracy(y, t):
  acc = 0
  N = 0
  for i in range(len(y)):
    N += 1
    if (y[i] >= 0.5 and t[i] == 1) or (y[i] < 0.5 and t[i] == 0):
      acc += 1
  return acc / N
```

Part (a) -- 7%

Write a function `pred` that computes the prediction `y` based on logistic regression, i.e., a single layer with weights `w` and bias `b`. The output is given by:

$$y = \sigma(\mathbf{w}^T\mathbf{x} + b),$$

where the value of y is an estimate of the probability that the song is released in the current century, namely $\mathrm{year} = 1$.

```
def pred(w, b, X):
  """
  Returns the prediction `y` of the target based on the weights `w` and scalar bias `b`.
```

```
  Preconditions: np.shape(w) == (90,)
                 type(b) == float
                 np.shape(X) = (N, 90) for some N

  >>> pred(np.zeros(90), 1, np.ones([2, 90]))
  array([0.73105858, 0.73105858]) # It's okay if your output differs in the last decimals
  """

  return sigmoid(X @ w + b)
```

Test pred:

```
>>> pred(np.zeros(90), 1, np.ones([2, 90]))

    array([0.73105858, 0.73105858])
```

Part (b) -- 7%

Write a function `derivative_cost` that computes and returns the gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ and $\frac{\partial \mathcal{L}}{\partial b}$. Here, `X` is the input, `y` is the prediction, and `t` is the true label.

```
def derivative_cost(X, y, t):
  """
  Returns a tuple containing the gradients dLdw and dLdb.

  Precondition: np.shape(X) == (N, 90) for some N
                np.shape(y) == (N,)
                np.shape(t) == (N,)

  Postcondition: np.shape(dLdw) = (90,)
           type(dLdb) = float
  """
  # Based On analytical expression from text cell below
  # Dims
  N, d = X.shape

  phi = y-t
  dLdw = X.T @ phi / N
  dLdb = np.mean(phi)

  return dLdw, dLdb
```

Explenation on Gradients:

We will use the chain law to find $\frac{\partial \mathcal{L}}{db}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$. That is:

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{t})}{\partial y_n} \frac{\partial y_n}{\partial b}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{t})}{\partial y_n} \frac{\partial y_n}{\partial \mathbf{w}}$$

First we will find $\frac{\partial \mathcal{L}}{\partial y_n}$ where the cost function $\mathcal{L}$ is the empirical mean of the cross entropy funcion.

$$\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{t})}{\partial y_n} = \frac{\partial}{\partial y_n} \left(\frac{1}{N} \sum_{k=1}^{N} \left(-t_k \log(y_k) - (1 - t_k) \log(1 - y_k) \right) \right) = \frac{y_n - t_n}{N y_n (1 - y_n)}$$

Next we will find $\frac{\partial y_n}{\partial b}$, $\frac{\partial y_n}{\partial \mathbf{w}}$ using the chain rule agian:

$$\frac{\partial y_n}{\partial b} = \sigma'(z_n) \frac{\partial z_n}{\partial b}$$
$$\frac{\partial y_n}{\partial \mathbf{w}} = \sigma'(z_n) \frac{\partial z_n}{\partial \mathbf{w}}$$

where $z_k = \mathbf{w}^T \mathbf{x}_k + b$.

We know that the derivative of Sigmoid function satisfies:

$$\sigma'(z_n) = \sigma(z_n)(1 - \sigma(z_n)) = y_n(1 - y_n)$$

Recalling y_n:

$$y_n = \sigma(z_k)$$

The derivites of z_n are:

$$\frac{\partial z_n}{\partial b} = 1$$
$$\frac{\partial z_n}{\partial \mathbf{w}} = \mathbf{x}_n$$

Pulgining all together we get:

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N} \left(\frac{y_n - t_n}{N y_n (1 - y_n)} y_n (1 - y_n) \right) = \frac{1}{N} \sum_{n=1}^{N} y_n - t_n$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_{n=1}^{N} \left(\frac{y_n - t_n}{N y_n (1 - y_n)} y_n (1 - y_n) \mathbf{x}_n \right) = \frac{1}{N} \sum_{n=1}^{N} (y_n - t_n) \mathbf{x}_n$$

By defining $\boldsymbol{\varphi} = \mathbf{y} - \mathbf{t}$ and $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x}_n]$, we can write:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N} \boldsymbol{\varphi}^T \mathbf{1}$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N} \mathbf{X} \boldsymbol{\varphi}$$

Part (c) -- 7%

We can check that our derivative is implemented correctly using the finite difference rule. In 1D, the finite difference rule tells us that for small h, we should have

$$\frac{f(x+h) - f(x)}{h} \approx f'(x)$$

Show that $\frac{\partial \mathcal{L}}{\partial b}$ is implement correctly by comparing the result from `derivative_cost` with the

```
# Your code goes here

#Data, labels
bs = 100
X = train_norm_xs[:bs,:]
t = train_ts[:bs].flatten()
# Weights for example
w = np.zeros(90)
b = 1
#Create Cost
y = pred(w, b, X)
cost_func = cost(y, t)

#Create values with infinitesimal diff
delta = 1e-6
y_delta = pred(w, b + delta, X)
cost_delta = cost(y_delta, t)

#Create derivetives
_, r1 = derivative_cost(X, y, t)
r2 = (cost_delta - cost_func) / delta

#Prints
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
```

```
The analytical results is - 0.19105857863000486
The algorithm results is -  0.1910586772124745
```

Part (d) -- 7%

Show that $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ is implement correctly.

```
# Your code goes here. You might find this below code helpful: but it's
# up to you to figure out how/why, and how to modify the code

#Create values with infinitesimal diff
delta = 1e-6
ei = np.zeros(90); ei[0] = 1
cost_delta = np.array([cost(pred(w + delta * np.roll(ei,i),b,X),t) for i in range(90)])

#Create derivetives
r1, _ = derivative_cost(X, y, t)
```

```
r2 = (cost_delta - cost_func) / delta

#Prints
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
print("Norm Diff - ", np.linalg.norm(r1-r2))
```

```
The analytical results is - [-2.11649199e-01 -2.08733952e-02 -3.69623855e-02  5.0452
 -3.76471495e-02  9.68148633e-02 -6.37978378e-02 -2.38459479e-02
  1.72800249e-02 -3.51873603e-02  5.18420421e-02  5.81311206e-02
  5.12311733e-02  7.73467152e-02  3.91936985e-02  4.33201632e-02
  1.33228627e-02  5.52143215e-02 -3.95551728e-02  2.93020295e-02
 -2.72728213e-02  4.56871222e-02  3.50012742e-02 -3.81186187e-02
  5.91461674e-02 -6.94042398e-03 -3.85251613e-02 -3.64564662e-02
 -4.91181859e-02 -1.79888765e-04  1.56053014e-02  2.83284261e-02
 -7.93320863e-02  8.37688931e-02 -8.53057336e-02  1.43665252e-03
  1.09576364e-01 -9.02929914e-02  4.11417888e-02  3.25112308e-02
  6.46056049e-02 -2.35806176e-02  1.89232183e-02  9.25183396e-02
 -2.73815299e-03  4.87312094e-02 -1.09476867e-02  2.08169083e-02
 -7.46459782e-02 -5.55522956e-02  1.03577926e-01 -3.20708918e-02
 -4.69506001e-02  2.60342547e-02 -4.11367027e-02  1.44800903e-01
 -3.39261413e-02 -4.27422806e-02 -4.11413711e-03 -8.05943054e-02
  3.29495183e-02  7.15901623e-04  1.10826804e-01 -1.10689742e-02
  3.33509463e-03  6.32812529e-02 -1.15361769e-01  7.56094989e-02
  5.17794101e-02 -4.89123172e-02 -7.43903980e-02  4.02512760e-02
  1.69040978e-02 -7.52168867e-02  1.57259685e-02 -4.69660998e-02
  5.00617188e-02  4.14059745e-02 -1.52524200e-03 -4.21724950e-02
 -1.11330502e-02 -1.40069600e-01  4.41602816e-02  3.26435334e-02
 -6.86837832e-02  6.78355977e-02  5.45679979e-02  5.81415412e-02
  9.00324414e-02  9.83914692e-02]
The algorithm results is -  [-2.11649091e-01 -2.08733186e-02 -3.69622855e-02  5.0452
 -3.76470624e-02  9.68149623e-02 -6.37977621e-02 -2.38458844e-02
  1.72801315e-02 -3.51872781e-02  5.18421470e-02  5.81312262e-02
  5.12312366e-02  7.73467792e-02  3.91937861e-02  4.33202476e-02
  1.33229178e-02  5.52144296e-02 -3.95550980e-02  2.93021756e-02
 -2.72727467e-02  4.56872568e-02  3.50013513e-02 -3.81185430e-02
  5.91462437e-02 -6.94034086e-03 -3.85250898e-02 -3.64564013e-02
 -4.91180837e-02 -1.79822268e-04  1.56054020e-02  2.83285300e-02
 -7.93319731e-02  8.37689808e-02 -8.53056649e-02  1.43671952e-03
  1.09576448e-01 -9.02929245e-02  4.11418501e-02  3.25112945e-02
  6.46056898e-02 -2.35805543e-02  1.89233897e-02  9.25184293e-02
 -2.73805156e-03  4.87312793e-02 -1.09476277e-02  2.08170061e-02
 -7.46459139e-02 -5.55521835e-02  1.03578013e-01 -3.20707966e-02
 -4.69505405e-02  2.60343280e-02 -4.11366328e-02  1.44801001e-01
 -3.39260656e-02 -4.27421685e-02 -4.11407497e-03 -8.05941912e-02
  3.29495837e-02  7.15972837e-04  1.10826900e-01 -1.10688922e-02
  3.33518135e-03  6.32813222e-02 -1.15361658e-01  7.56095975e-02
  5.17794781e-02 -4.89122151e-02 -7.43903025e-02  4.02513447e-02
  1.69041564e-02 -7.52167969e-02  1.57260385e-02 -4.69660126e-02
  5.00618006e-02  4.14060776e-02 -1.52517077e-03 -4.21724401e-02
 -1.11329729e-02 -1.40069502e-01  4.41603566e-02  3.26436133e-02
 -6.86836998e-02  6.78356603e-02  5.45680591e-02  5.81416121e-02
  9.00325482e-02  9.83915411e-02]
Norm Diff -  8.265910369784957e-07
```

Part (e) -- 7%

Now that you have a gradient function that works, we can actually run gradient descent. Complete the following code that will run stochastic: gradient descent training:

```
def run_gradient_descent(train_norm_xs, train_ts, val_norm_xs, val_ts, w0, b0, mu=0.1, bat
  """Return the values of (w, b) after running gradient descent for max_iters.
  We use:
    - train_norm_xs and train_ts as the training set
    - val_norm_xs and val_ts as the test set
    - mu as the learning rate
    - (w0, b0) as the initial values of (w, b)

  Precondition: np.shape(w0) == (90,)
                type(b0) == float

  Postcondition: np.shape(w) == (90,)
                 type(b) == float
  """
  w = w0
  b = b0
  iter = 0
  train_loss_arr = []
  val_loss_arr = []
  val_acc_arr = []
  trian_acc_arr = []
  best_acc = 0.0
  best_b = 0.0
  best_w = 0


  while iter < max_iters:
    # shuffle the training set (there is code above for how to do this)
    reindex = np.random.permutation(len(train_norm_xs))
    train_norm_xs = train_norm_xs[reindex]
    train_ts = train_ts[reindex]


    for i in range(0, len(train_norm_xs), batch_size): # iterate over each minibatch
      # minibatch that we are working with:
      X = train_norm_xs[i:(i + batch_size)]
      t = train_ts[i:(i + batch_size), 0]

      # since len(train_norm_xs) does not divide batch_size evenly, we will skip over
      # the "last" minibatch
      if np.shape(X)[0] != batch_size:
        continue

      # compute the prediction
      predictions = pred(w, b, X)

      # update w and b
      grad_w, grad_b = derivative_cost(X, predictions, t)
      w = w - mu * grad_w
      b = b - mu * grad_b
```

```
    # increment the iteration count
    iter += 1
    # compute and print the *validation* loss and accuracy
    if (iter % 10 == 0):
      pred_val = pred(w, b, val_norm_xs)
      pred_train = pred(w, b, X)
      val_cost = cost(pred_val, val_ts)
      val_acc = get_accuracy(pred_val, val_ts)
      train_acc = get_accuracy(pred_train, t)
      train_loss = cost(pred_train, t)

      train_loss_arr.append(train_loss)
      trian_acc_arr.append(train_acc)
      val_loss_arr.append(val_cost)
      val_acc_arr.append(val_acc)

      if val_acc >= best_acc:
        best_b = b
        best_w = w
        best_acc = val_acc


      if verbose:
        print("Iter %4d. [Train Loss %f, Val Loss %f, Train Acc %.0f%%, Val Acc %.0f%%]"
              iter, train_loss, val_cost, train_acc*100, val_acc * 100))
      else:
        if (iter % (max_iters//4) == 0):
          print("Iter %4d. [Train Loss %f, Val Loss %f, Train Acc %.0f%%, Val Acc %.0f%%
              iter, train_loss, val_cost, train_acc*100, val_acc * 100))




    if iter >= max_iters:
      break


  analysis_dict = {'Train_Loss': np.array(train_loss_arr),
                   'Val_Loss':np.array(val_loss_arr),
                   'Train_acc':np.array(trian_acc_arr),
                   'Val_acc':np.array(val_acc_arr),
                    'best_b': best_b,
                    'best_w': best_w,
                    'best_acc': best_acc}


  return w, b, analysis_dict
```

Part (f) -- 7%

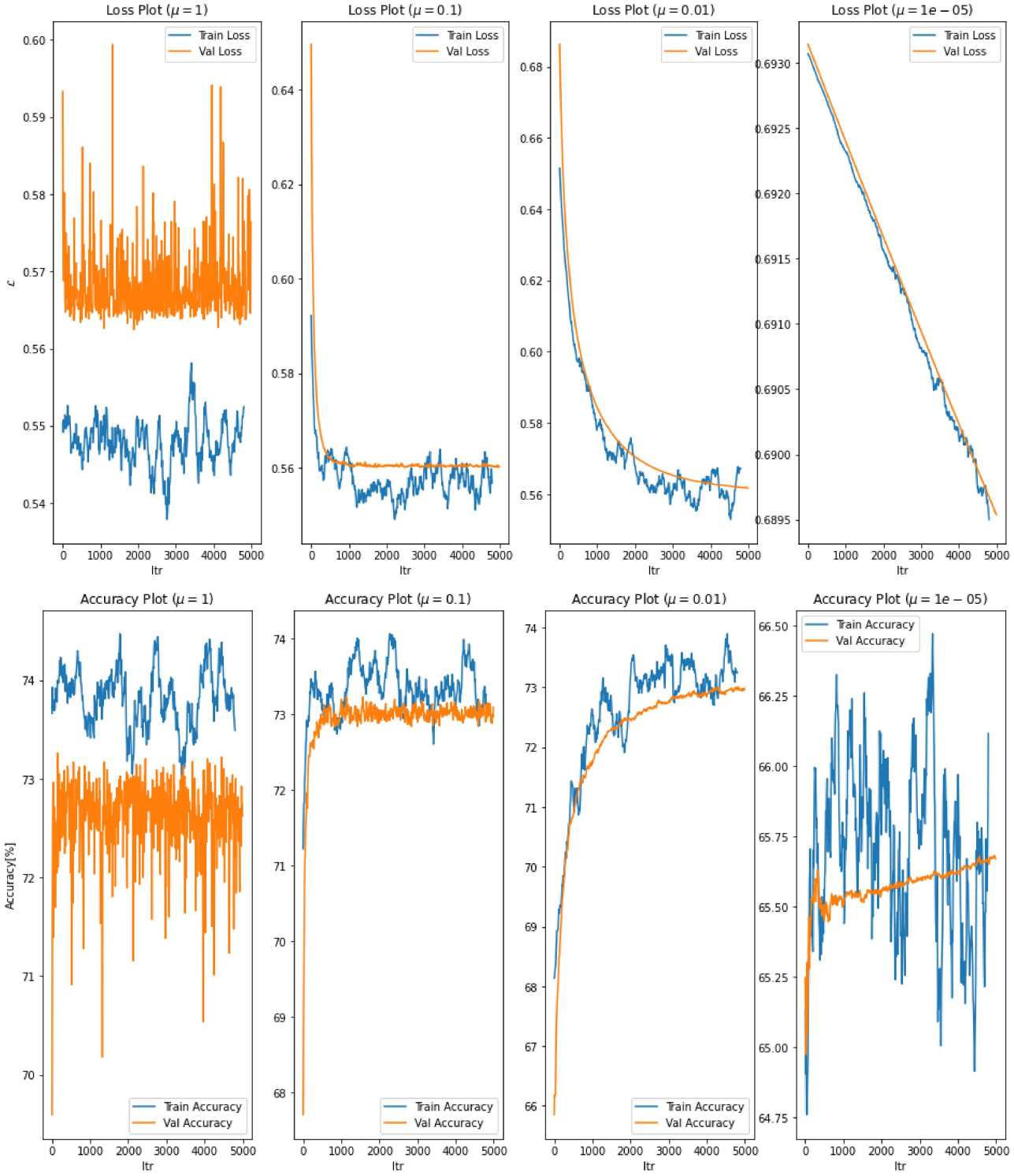
Call `run_gradient_descent` with the weights and biases all initialized to zero. Show that if the learning rate μ is too small, then convergence is slow. Also, show that if μ is too large, then the optimization algorirthm does not converge. The demonstration should be made using plots showing these effects.

```
w0 = np.zeros(90)
b0 = 0

mu_list = [1,0.1,0.01,1e-5]
max_itr = 5000
batch_size = 1000
res_dict = {'w':[],'b':[],'analysis_dict':[]}
for mu in mu_list:
  print('mu: {}'.format(mu))
  w, b, analysis_dict = run_gradient_descent(train_norm_xs, train_ts, val_norm_xs,
                                             val_ts.flatten(), w0, b0,
                                             max_iters=max_itr, mu=mu,
                                             batch_size=batch_size)
  res_dict['w'].append(w)
  res_dict['b'].append(b)
  res_dict['analysis_dict'].append(analysis_dict)
```

```
Iter   50. [Train Loss 0.546977, Val Loss 0.580244, Train Acc 73%, Val Acc 71%]
Iter   60. [Train Loss 0.523711, Val Loss 0.570123, Train Acc 76%, Val Acc 73%]
Iter   70. [Train Loss 0.556281, Val Loss 0.566214, Train Acc 74%, Val Acc 73%]
Iter   80. [Train Loss 0.539786, Val Loss 0.564707, Train Acc 74%, Val Acc 73%]
Iter   90. [Train Loss 0.566624, Val Loss 0.568557, Train Acc 72%, Val Acc 73%]
Iter  100. [Train Loss 0.534212, Val Loss 0.574985, Train Acc 74%, Val Acc 72%]
Iter  110. [Train Loss 0.545232, Val Loss 0.570941, Train Acc 74%, Val Acc 73%]
Iter  120. [Train Loss 0.564780, Val Loss 0.567843, Train Acc 72%, Val Acc 72%]
Iter  130. [Train Loss 0.528321, Val Loss 0.566473, Train Acc 74%, Val Acc 72%]
Iter  140. [Train Loss 0.557564, Val Loss 0.572304, Train Acc 72%, Val Acc 72%]
Iter  150. [Train Loss 0.533719, Val Loss 0.565083, Train Acc 76%, Val Acc 73%]
Iter  160. [Train Loss 0.554804, Val Loss 0.565712, Train Acc 74%, Val Acc 73%]
Iter  170. [Train Loss 0.539213, Val Loss 0.573262, Train Acc 74%, Val Acc 72%]
Iter  180. [Train Loss 0.557319, Val Loss 0.565674, Train Acc 74%, Val Acc 73%]
Iter  190. [Train Loss 0.557643, Val Loss 0.567572, Train Acc 74%, Val Acc 73%]
Iter  200. [Train Loss 0.548624, Val Loss 0.564149, Train Acc 75%, Val Acc 73%]
Iter  210. [Train Loss 0.547851, Val Loss 0.566005, Train Acc 77%, Val Acc 73%]
Iter  220. [Train Loss 0.586607, Val Loss 0.569585, Train Acc 70%, Val Acc 72%]
Iter  230. [Train Loss 0.555529, Val Loss 0.564542, Train Acc 74%, Val Acc 73%]
Iter  240. [Train Loss 0.548562, Val Loss 0.566722, Train Acc 74%, Val Acc 73%]
Iter  250. [Train Loss 0.566480, Val Loss 0.565551, Train Acc 74%, Val Acc 73%]
Iter  260. [Train Loss 0.539490, Val Loss 0.568022, Train Acc 74%, Val Acc 72%]
Iter  270. [Train Loss 0.548129, Val Loss 0.567568, Train Acc 74%, Val Acc 73%]
Iter  280. [Train Loss 0.541276, Val Loss 0.567106, Train Acc 74%, Val Acc 73%]
Iter  290. [Train Loss 0.536401, Val Loss 0.567699, Train Acc 76%, Val Acc 73%]
Iter  300. [Train Loss 0.546014, Val Loss 0.563622, Train Acc 74%, Val Acc 73%]

Iter  310. [Train Loss 0.563140, Val Loss 0.576919, Train Acc 73%, Val Acc 72%]
Iter  320. [Train Loss 0.542390, Val Loss 0.563742, Train Acc 73%, Val Acc 73%]
Iter  330. [Train Loss 0.581747, Val Loss 0.565053, Train Acc 72%, Val Acc 73%]
Iter  340. [Train Loss 0.533666, Val Loss 0.570271, Train Acc 76%, Val Acc 72%]
Iter  350. [Train Loss 0.508337, Val Loss 0.568655, Train Acc 76%, Val Acc 72%]
Iter  360. [Train Loss 0.567273, Val Loss 0.571125, Train Acc 73%, Val Acc 73%]
```

```
Iter  370. [Train Loss 0.545904, Val Loss 0.567872, Train Acc 74%, Val Acc 72%]
Iter  380. [Train Loss 0.560056, Val Loss 0.566471, Train Acc 75%, Val Acc 73%]
Iter  390. [Train Loss 0.546395, Val Loss 0.567658, Train Acc 74%, Val Acc 73%]
Iter  400. [Train Loss 0.528609, Val Loss 0.565729, Train Acc 77%, Val Acc 73%]
Iter  410. [Train Loss 0.528490, Val Loss 0.565476, Train Acc 74%, Val Acc 73%]
Iter  420. [Train Loss 0.541923, Val Loss 0.565958, Train Acc 73%, Val Acc 73%]
Iter  430. [Train Loss 0.554598, Val Loss 0.565086, Train Acc 74%, Val Acc 73%]
Iter  440. [Train Loss 0.571027, Val Loss 0.566024, Train Acc 71%, Val Acc 73%]
Iter  450. [Train Loss 0.560938, Val Loss 0.565503, Train Acc 74%, Val Acc 73%]
Iter  460. [Train Loss 0.546113, Val Loss 0.567452, Train Acc 74%, Val Acc 72%]
Iter  470. [Train Loss 0.553229, Val Loss 0.565807, Train Acc 72%, Val Acc 73%]
Iter  480. [Train Loss 0.525598, Val Loss 0.569842, Train Acc 77%, Val Acc 73%]
Iter  490. [Train Loss 0.562944, Val Loss 0.563391, Train Acc 73%, Val Acc 73%]
Iter  500. [Train Loss 0.557229, Val Loss 0.567988, Train Acc 74%, Val Acc 72%]
Iter  510. [Train Loss 0.558683, Val Loss 0.568841, Train Acc 74%, Val Acc 72%]
Iter  520. [Train Loss 0.546621, Val Loss 0.563797, Train Acc 74%, Val Acc 73%]
Iter  530. [Train Loss 0.558860, Val Loss 0.586090, Train Acc 74%, Val Acc 71%]
Iter  540. [Train Loss 0.531431, Val Loss 0.566844, Train Acc 74%, Val Acc 73%]
Iter  550. [Train Loss 0.559182, Val Loss 0.565432, Train Acc 72%, Val Acc 73%]
Iter  560. [Train Loss 0.523798, Val Loss 0.573522, Train Acc 77%, Val Acc 72%]
Iter  570. [Train Loss 0.548471, Val Loss 0.567171, Train Acc 74%, Val Acc 73%]
Iter  580. [Train Loss 0.534515, Val Loss 0.568357, Train Acc 74%, Val Acc 72%]
Iter  590. [Train Loss 0.549842, Val Loss 0.567766, Train Acc 74%, Val Acc 72%]
Iter  600. [Train Loss 0.539256, Val Loss 0.571360, Train Acc 74%, Val Acc 72%]
Iter  610. [Train Loss 0.548620, Val Loss 0.565711, Train Acc 74%, Val Acc 72%]
Iter  620. [Train Loss 0.517984, Val Loss 0.566947, Train Acc 75%, Val Acc 72%]
Iter  630. [Train Loss 0.562360, Val Loss 0.564062, Train Acc 73%, Val Acc 73%]
Iter  640. [Train Loss 0.547497, Val Loss 0.565456, Train Acc 74%, Val Acc 72%]
```

```
running_avg_size = 20
#Cost Plots
fig, axs = plt.subplots(1, 4,figsize=(16,9))
for i,mu in enumerate(mu_list):
  running_avg_train_loss = np.convolve(res_dict['analysis_dict'][i]['Train_Loss'], np.ones

  axs[i].plot(range(0,max_itr - (running_avg_size-1)*10,10), running_avg_train_loss)
  axs[i].plot(range(0,max_itr,10),res_dict['analysis_dict'][i]['Val_Loss'])
  axs[i].legend(['Train Loss','Val Loss'])
  axs[i].set_title('Loss Plot ($\mu={}$)'.format(mu))
  axs[i].set_xlabel('Itr')
  if i == 0:
    axs[i].set_ylabel('$\mathcal{L}$')

  #ACC Plots
fig, axs = plt.subplots(1, 4,figsize=(16,9))
for i,mu in enumerate(mu_list):
  running_avg_train_acc = np.convolve(res_dict['analysis_dict'][i]['Train_acc'], np.ones(r

  axs[i].plot(range(0,max_itr - (running_avg_size-1)*10,10), 100*running_avg_train_acc)
  axs[i].plot(range(0,max_itr,10),100*res_dict['analysis_dict'][i]['Val_acc'])
  axs[i].legend(['Train Accuracy','Val Accuracy'])
  axs[i].set_title('Accuracy Plot ($\mu={}$)'.format(mu))
  axs[i].set_xlabel('Itr')
  if i == 0:
    axs[i].set_ylabel('Accuracy[%]')
```


Loss Plot ($\mu = 1$)
Loss Plot ($\mu = 0.1$)
Loss Plot ($\mu = 0.01$)
Loss Plot ($\mu = 1e-05$)
Train Loss
Val Loss
$\mathcal{L}$
Itr
Accuracy Plot ($\mu = 1$)
Accuracy Plot ($\mu = 0.1$)
Accuracy Plot ($\mu = 0.01$)
Accuracy Plot ($\mu = 1e-05$)
Train Accuracy
Val Accuracy
Accuracy[%]

Explain and discuss your results here:

First, we can see that the results for the train loss and train acc are much noisier than the results for the validation set. That is beacuse we use mini-batch with size of 100 samples in the train step, i.e. we are not using all the train data in each iteartion but a small portion of it, what adds a notion of stochtisity with higer varinces to the results. In the other hand, for the validtion calcultion we used *all* the valdiation dataset, whice yileds some noise reduction.

We can see that the value of the step size has termndeous impcat on the results. When using huge step size of $\mu = 1$, we get "jumping" values and we are missing the local minima. From the other hand, when using very small step size of $\mu = 10^{-5}$, we are not reaching convergence. For the midterme values we get fine results where the best results are achieved by using $\mu = 0.01$.

Part (g) -- 7%

Find the optimial value of $\mathbf{w}$ and b using your code. Explain how you chose the learning rate μ and the batch size. Show plots demostrating good and bad behaviours.

```
# We will swipe over differnt parmutaion for the hyper-params (init_type,bs,mu)
max_itr = 5000
mu_list = [1,0.1,0.01,1e-5]
batch_size_list = [100,1000,10000]
weight_init = ['zeros','rand']
params_tuples_list = [(init_type, batch_size,mu) for init_type in weight_init
                      for batch_size in batch_size_list
                      for mu in mu_list ]
analysis_dict_list = []

for init_type in weight_init:
  for batch_size in batch_size_list:
    for mu in mu_list:
      print('Init Type: {}, Batch Size: {}, Mu: {}'.format(init_type, batch_size, mu))
      if init_type == 'zeros':
        w0 = np.zeros(90)
        b0 = 0
      else:
        w0 = np.random.randn(90)
        b0 = np.random.randn(1)[0]

      _, _, analysis_dict = run_gradient_descent(train_norm_xs, train_ts, val_norm_xs,
                                                 val_ts.flatten(), w0, b0,
                                                 max_iters=max_itr, mu=mu,
                                                 batch_size=batch_size, verbose=False)

      analysis_dict_list.append(analysis_dict)
```

```
#Find Best Model
best_model_idx = np.argmax([analysis_dict_list[i]['best_acc'] for i in range(len(analysis_
best_b = analysis_dict_list[best_model_idx]['best_b']
best_w = analysis_dict_list[best_model_idx]['best_w']
best_model_best_val_acc = analysis_dict_list[best_model_idx]['best_acc']
print('Best Model: {}, Best Val ACC: {}'.
      format(params_tuples_list[best_model_idx], best_model_best_val_acc ))
```

```
Iter 1250. [Train Loss 0.515268, Val Loss 0.621536, Train Acc 72%, Val Acc 70%]
Iter 2500. [Train Loss 0.494752, Val Loss 0.759790, Train Acc 83%, Val Acc 67%]
Iter 3750. [Train Loss 0.481672, Val Loss 0.621715, Train Acc 78%, Val Acc 71%]
Iter 5000. [Train Loss 0.463615, Val Loss 0.632331, Train Acc 80%, Val Acc 71%]
Init Type: rand, Batch Size: 100, Mu: 0.1
Iter 1250. [Train Loss 0.522084, Val Loss 0.568428, Train Acc 76%, Val Acc 73%]
Iter 2500. [Train Loss 0.507056, Val Loss 0.569150, Train Acc 80%, Val Acc 73%]
Iter 3750. [Train Loss 0.564936, Val Loss 0.564841, Train Acc 71%, Val Acc 73%]
Iter 5000. [Train Loss 0.708059, Val Loss 0.563888, Train Acc 60%, Val Acc 73%]
Init Type: rand, Batch Size: 100, Mu: 0.01
Iter 1250. [Train Loss 1.590744, Val Loss nan, Train Acc 56%, Val Acc 57%]
Iter 2500. [Train Loss 0.706705, Val Loss nan, Train Acc 64%, Val Acc 65%]
Iter 3750. [Train Loss 0.551696, Val Loss nan, Train Acc 73%, Val Acc 69%]
Iter 5000. [Train Loss 0.762079, Val Loss 0.628958, Train Acc 64%, Val Acc 70%]
Init Type: rand, Batch Size: 100, Mu: 1e-05
Iter 1250. [Train Loss 4.795432, Val Loss nan, Train Acc 42%, Val Acc 50%]
Iter 2500. [Train Loss 2.985893, Val Loss nan, Train Acc 51%, Val Acc 50%]
Iter 3750. [Train Loss 3.633941, Val Loss nan, Train Acc 42%, Val Acc 50%]
Iter 5000. [Train Loss 3.305234, Val Loss nan, Train Acc 48%, Val Acc 50%]
Init Type: rand, Batch Size: 1000, Mu: 1
Iter 1250. [Train Loss 0.555203, Val Loss 0.565021, Train Acc 73%, Val Acc 73%]
Iter 2500. [Train Loss 0.532556, Val Loss 0.568420, Train Acc 75%, Val Acc 73%]
Iter 3750. [Train Loss 0.539162, Val Loss 0.569049, Train Acc 76%, Val Acc 73%]
Iter 5000. [Train Loss 0.597755, Val Loss 0.576601, Train Acc 70%, Val Acc 72%]
Init Type: rand, Batch Size: 1000, Mu: 0.1
Iter 1250. [Train Loss 0.547721, Val Loss 0.564386, Train Acc 75%, Val Acc 73%]
Iter 2500. [Train Loss 0.557894, Val Loss 0.560312, Train Acc 73%, Val Acc 73%]
Iter 3750. [Train Loss 0.559000, Val Loss 0.560218, Train Acc 73%, Val Acc 73%]
Iter 5000. [Train Loss 0.585439, Val Loss 0.560781, Train Acc 72%, Val Acc 73%]
Init Type: rand, Batch Size: 1000, Mu: 0.01
Iter 1250. [Train Loss inf, Val Loss nan, Train Acc 62%, Val Acc 61%]
Iter 2500. [Train Loss 0.922541, Val Loss nan, Train Acc 66%, Val Acc 65%]
Iter 3750. [Train Loss 0.717722, Val Loss nan, Train Acc 68%, Val Acc 68%]
Iter 5000. [Train Loss 0.691009, Val Loss 0.646004, Train Acc 67%, Val Acc 70%]
Init Type: rand, Batch Size: 1000, Mu: 1e-05
Iter 1250. [Train Loss nan, Val Loss nan, Train Acc 53%, Val Acc 55%]
Iter 2500. [Train Loss nan, Val Loss nan, Train Acc 54%, Val Acc 55%]
Iter 3750. [Train Loss nan, Val Loss nan, Train Acc 53%, Val Acc 55%]
Iter 5000. [Train Loss nan, Val Loss nan, Train Acc 55%, Val Acc 55%]
Init Type: rand, Batch Size: 10000, Mu: 1
Iter 1250. [Train Loss 0.565156, Val Loss 0.560562, Train Acc 73%, Val Acc 73%]
Iter 2500. [Train Loss 0.553958, Val Loss 0.560327, Train Acc 74%, Val Acc 73%]
Iter 3750. [Train Loss 0.558289, Val Loss 0.560706, Train Acc 73%, Val Acc 73%]
Iter 5000. [Train Loss 0.560912, Val Loss 0.560483, Train Acc 72%, Val Acc 73%]
Init Type: rand, Batch Size: 10000, Mu: 0.1
Iter 1250. [Train Loss 0.552831, Val Loss 0.564370, Train Acc 74%, Val Acc 73%]
Iter 2500. [Train Loss 0.556406, Val Loss 0.560339, Train Acc 74%, Val Acc 73%]
Iter 3750. [Train Loss 0.552446, Val Loss 0.560094, Train Acc 74%, Val Acc 73%]
Iter 5000. [Train Loss 0.554274, Val Loss 0.560050, Train Acc 74%, Val Acc 73%]
Init Type: rand, Batch Size: 10000, Mu: 0.01
```

```
Init Type: rand, Batch Size: 10000, Mu: 0.01
Iter 1250. [Train Loss nan, Val Loss nan, Train Acc 58%, Val Acc 59%]
Iter 2500. [Train Loss nan, Val Loss nan, Train Acc 63%, Val Acc 63%]
Iter 3750. [Train Loss 0.796717, Val Loss nan, Train Acc 67%, Val Acc 66%]
Iter 5000. [Train Loss 0.695961, Val Loss nan, Train Acc 67%, Val Acc 69%]
Init Type: rand, Batch Size: 10000, Mu: 1e-05
Iter 1250. [Train Loss nan, Val Loss nan, Train Acc 52%, Val Acc 53%]
Iter 2500. [Train Loss nan, Val Loss nan, Train Acc 52%, Val Acc 53%]
Iter 3750. [Train Loss nan, Val Loss nan, Train Acc 53%, Val Acc 53%]
Iter 5000. [Train Loss nan, Val Loss nan, Train Acc 52%, Val Acc 53%]
```

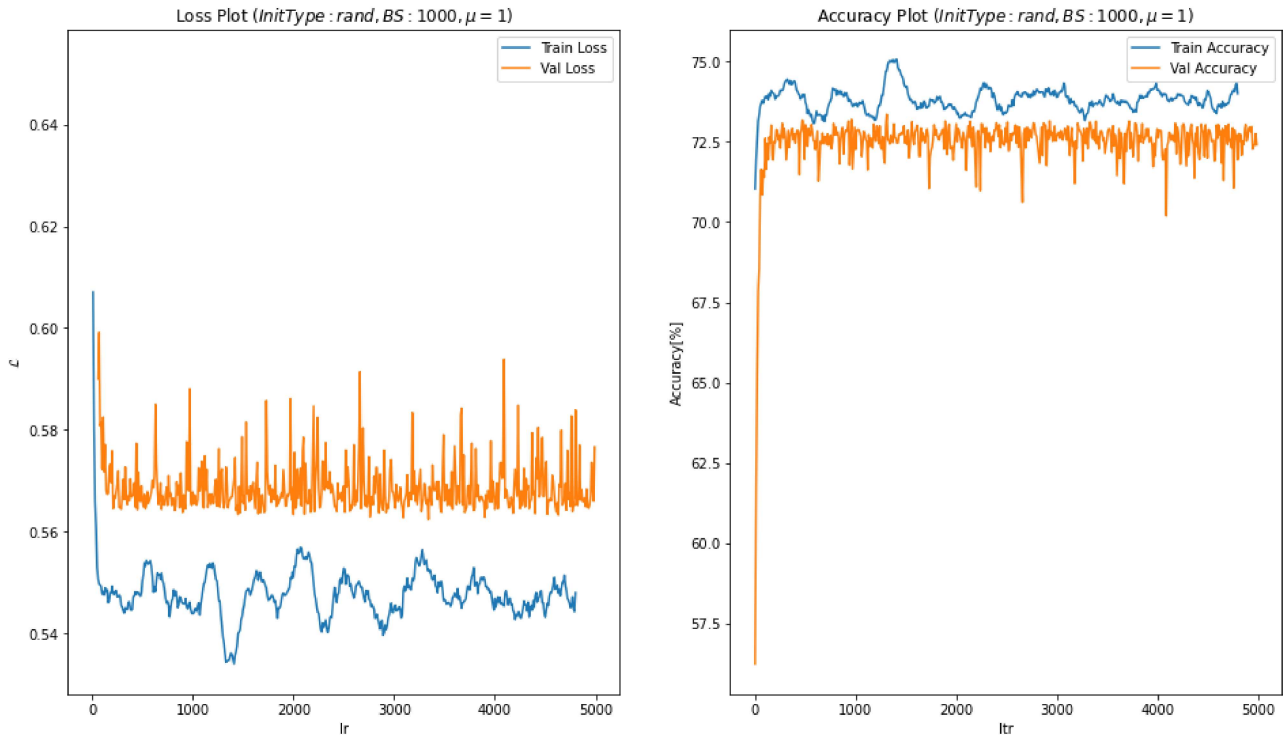
```
#We will sort by the bes val_acc values for each model

acc_best_vals_arr = [analysis_dict_list[i]['best_acc'] for i in range(len(analysis_dict_li
sorted_idxs = np.argsort([-analysis_dict_list[i]['best_acc'] for i in range(len(analysis_d
acc_val_sorted = [(acc_best_vals_arr[idx],params_tuples_list[idx],idx) for idx in sorted_i
print(acc_val_sorted)
params_tuples_list_sorted = [params_tuples_list[i] for i in sorted_idxs]

def Loss_Acc_Plots(idx,supTitle, params_tuples_list,analysis_dict_list,running_avg_size=20
  """ Plot function for Loss and ACC values for a given model idx"""
  init_type, batch_size, mu = params_tuples_list[idx]
  fig, axs = plt.subplots(1, 2,figsize=(16,9))
  fig.suptitle(supTitle, fontsize=16)

  running_avg_train_loss = np.convolve(analysis_dict_list[idx]['Train_Loss'],
                                       np.ones(running_avg_size)/running_avg_size,mode='val
  axs[0].plot(range(0,max_itr - (running_avg_size-1)*10,10), running_avg_train_loss)
  axs[0].plot(range(0,max_itr,10),analysis_dict_list[idx]['Val_Loss'])
  axs[0].legend(['Train Loss','Val Loss'])
  axs[0].set_title('Loss Plot ($Init Type: {}, BS: {}, \mu={}$)'.format(init_type, batch_s
  axs[0].set_xlabel('Itr')
  axs[0].set_ylabel('$\mathcal{L}$')

  running_avg_train_acc = np.convolve(analysis_dict_list[idx]['Train_acc'], np.ones(runnir

  axs[1].plot(range(0,max_itr - (running_avg_size-1)*10,10), 100*running_avg_train_acc)
  axs[1].plot(range(0,max_itr,10),100*analysis_dict_list[idx]['Val_acc'])
  axs[1].legend(['Train Accuracy','Val Accuracy'])
  axs[1].set_title('Accuracy Plot ($Init Type: {}, BS: {}, \mu={}$)'.format(init_type, bat
  axs[1].set_xlabel('Itr')
  axs[1].set_ylabel('Accuracy[%]')
```

```
[(0.73362, ('rand', 1000, 1), 16), (0.73314, ('zeros', 10000, 1), 8), (0.73252, ('ze
```

We will generate some examples for good and bad models:

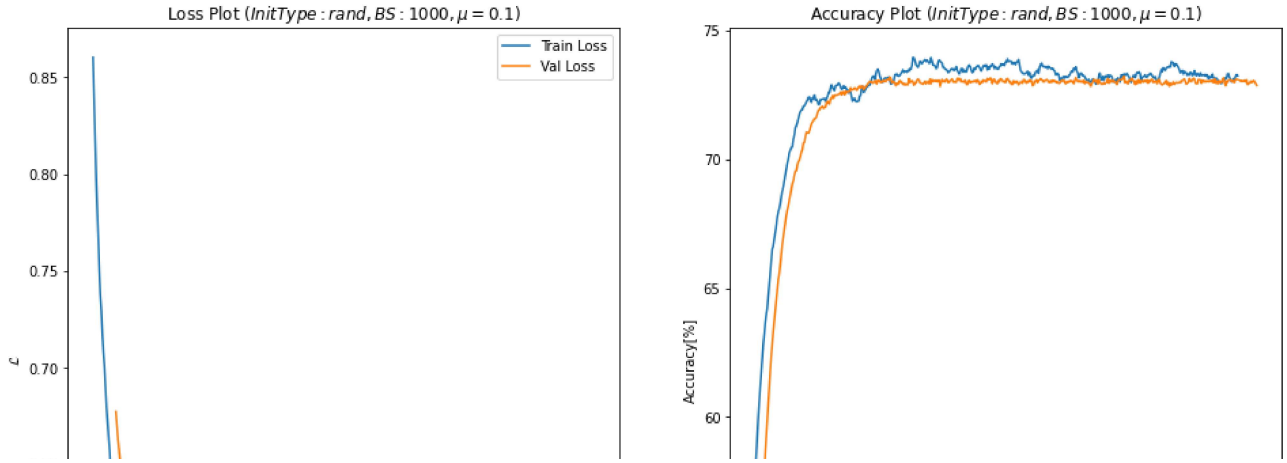
```
#Good but "noisy"
Loss_Acc_Plots(idx=16, supTitle='Fig1: Good But Noisy', params_tuples_list=params_tuples_l
```

Fig1: Good But Noisy



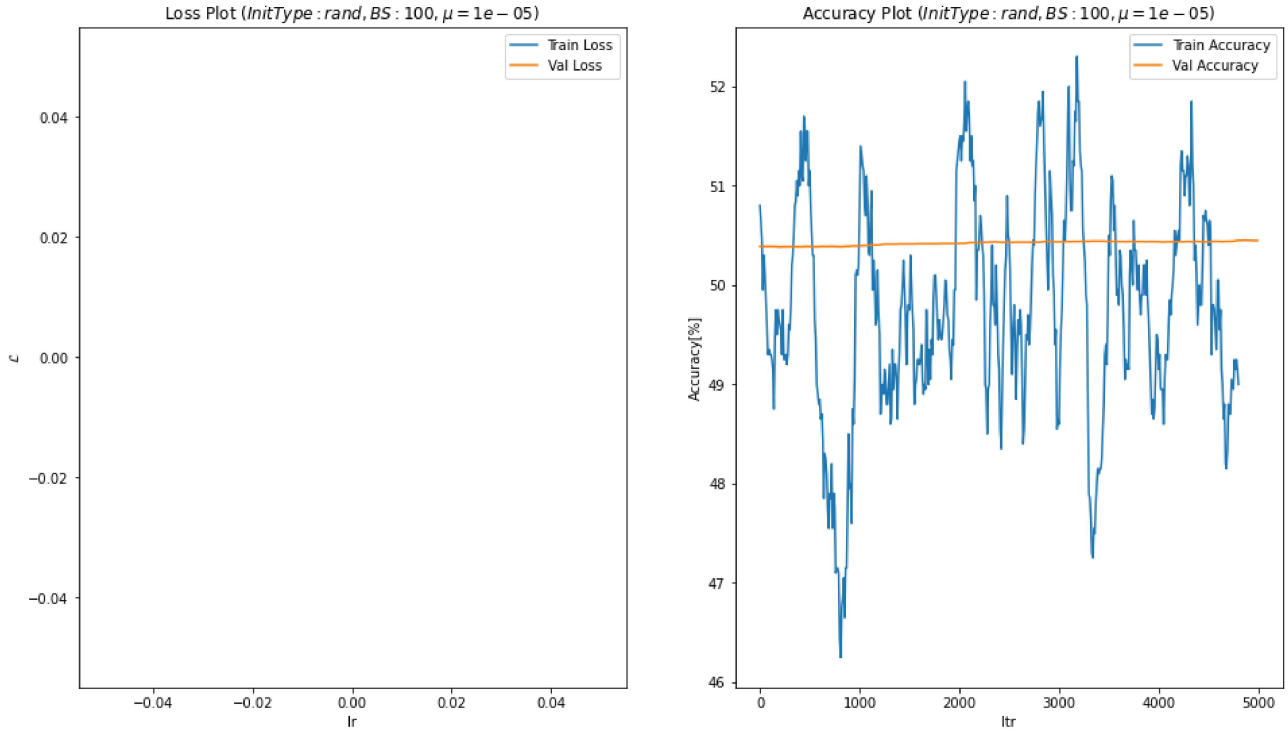

```
#Good but and stable
Loss_Acc_Plots(idx=17, supTitle='Fig2: Good And Stable', params_tuples_list=params_tuples_
```

Fig2: Good And Stable




```
#Bad Results
Loss_Acc_Plots(idx=15, supTitle='Fig3: Bad Results', params_tuples_list=params_tuples_list
```

Fig3: Bad Results



Explain and discuss your results here:

In Figures 1-3 we can see different results for different hyper-params. In Figure 1 we can see that although we got the highest acc_val of 0.73362 we got noisy loss and acc plots, that is due to the large step size of $\mu = 1$, we will preferer to take as best result the hyper params that will yield similar acc_val but with more stable plots. Therefore we choose as best model the one shown in Figure 2 that achieved very close best acc of 0.73222 and more stable plots we the values of:

```
mu = 0.1, batch_size = 1000, init_type = 'rand'
```

In Figure 3 we can see very bad results when we used small step size μ=1e−5, we can see no values on the loss plots because all the values are nan (we got zero or negative values in the log argument), the values for the acc plot are also very bad.

Part (h) -- 15%

Using the values of `w` and `b` from part (g), compute your training accuracy, validation accuracy, and test accuracy. Are there any differences between those three values? If so, why?

```
# Write your code here
best_idx = 17
best_b = analysis_dict_list[best_idx]['best_b']
best_w = analysis_dict_list[best_idx]['best_w']

pred_val = pred(best_w, best_b, val_norm_xs)
pred_train = pred(best_w, best_b, train_norm_xs)
pred_test = pred(best_w, best_b, test_norm_xs)

train_acc = get_accuracy(pred_train, train_ts)
val_acc = get_accuracy(pred_val, val_ts)
test_acc = get_accuracy(pred_test, test_ts)

print('train_acc = ', train_acc, ' val_acc = ', val_acc, ' test_acc = ', test_acc)
```

```
train_acc =  0.7337152387513143  val_acc =  0.73222  test_acc =  0.7277358125121054
```

Explain and discuss your results here:

We can see the acc results for each set is different. Although the values are different, they close enough to each other in a reasonable sense. The best result was achieved to the train set because those are the values we trained on. The next best value is for the validation set because the validation set has the same distribution as the train set. The test set is a different set from the train (and so from the validation) as was explained in the "producer effect" and thats why it has the lowest value. Although the test acc is the lowest, it is still close enough to the validation results which indicates that we are not overfitted.

Part (i) -- 15%

Writing a classifier like this is instructive, and helps you understand what happens when we train a model. However, in practice, we rarely write model building and training code from scratch. Instead, we typically use one of the well-tested libraries available in a package.

Use `sklearn.linear_model.LogisticRegression` to build a linear classifier, and make predictions about the test set. Start by reading the API documentation here.

Compute the training, validation and test accuracy of this model.

```
import sklearn.linear_model

model = sklearn.linear_model.LogisticRegression(random_state=0).fit(train_norm_xs,train_ts
train_acc = model.score(train_norm_xs,train_ts.flatten())
val_acc = model.score(val_norm_xs,val_ts.flatten())
test_acc = model.score(test_norm_xs,test_ts.flatten())

print('train_acc = ', train_acc, ' val_acc = ', val_acc, ' test_acc = ', test_acc)
```

```
train_acc =  0.7331762203449235  val_acc =  0.73046  test_acc =  0.7267286461359674
```

This parts helps by checking if the code worked. Check if you get similar results, if not repair your code