Homework 1
CS 5787 Deep Learning
Spring 2020

Irene Font Peradejordi

Feb 2020

Instructions

Your homework submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Canvas.

Your homework solution must be typed. We urge you to prepare it in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free and can be accessed online.

Your programs must be written in Python. The relevant code to the problem should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution to that problem. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}. For this assignment, you may not use a neural network toolbox. **The algorithm should be implemented using only NumPy.**

If you have forgotten your linear algebra, you may find *The Matrix Cookbook* useful, which can be readily found online. You may wish to use the program *MathType*, which can easily export equations to AMS LaTeX so that you don't have to write the equations in LaTeX directly: `http://www.dessci.com/en/products/mathtype/`

If told to implement an algorithm, don't use a toolbox, or you will receive no credit.

Problem 1 - Softmax Properties

Part 1 (7 points)

Recall the softmax function, which is the most common activation function used for the output of a neural network trained to do classification. In a vectorized form, it is given by

$$\text{softmax}(\mathbf{a}) = \frac{\exp(\mathbf{a})}{\sum_{j=1}^{K} \exp(a_j)},$$

where $\mathbf{a} \in \mathbb{R}^K$. The exp function in the numerator is applied element-wise and a_j denotes the j'th element of $\mathbf{a}$.

Show that the softmax function is invariant to constant offsets to its input, i.e.,

$$\text{softmax}(\mathbf{a} + c\mathbf{1}) = \text{softmax}(\mathbf{a}),$$

where $c \in \mathbb{R}$ is some constant and $\mathbf{1}$ denotes a column vector of 1's.

Solution:

The key in understanding why these two expressions are equal is the following:

$$a^{b+c} = a^b * a^c$$

Therefore, when simplifying the equation the c term disappears. Let's look at an example where a is a vector of 3 elements:

$$softmax(\overline{a} + c) = \frac{e^{\overline{a}+c}}{e^{a1+c} + e^{a2+c} + e^{a3+c}}$$

$$= \frac{e^{\overline{a}} * e^c}{(e^{a1} * e^c) + (e^{a2} * e^c) + (e^{a3} * e^c)}$$

Then, you can extract e to the c as a common factor and delete it from both, numerator and denominator.

Part 2 (3 points)

In practice, why is the observation that the softmax function is invariant to constant offsets to its input important when implementing it in a neural network?

Solution:
It is important because the product of data and weights is very large. When this number is elevated to the exponent it becomes even larger and it can go to infinity, returning "nan" values as output.

To handle this problem we can subtract a constant from the product to bring it down to certain rates and ovoid the output to go to infinity. This procedure is what I have done to implement my function.

Problem 2 - Implementing a Softmax Classifier

For this problem, you will use the 2-dimensional Iris dataset. Download `iris-train.txt` and `iris-test.txt` from Canvas. Each row is one data instance. The first column is the label (1, 2 or 3) and the next two columns are features.

Write a function to load the data and the labels, which are returned as NumPy arrays.

```
def load (file):
    train = loadtxt(file, comments="#", delimiter=" ", unpack=False)
    x = train[:,1:3]
    y = train[:,0]
    return x, y
```

Part 1 - Implementation & Evaluation (20 points)

Recall that a softmax classifier is a shallow one-layer neural network of the form:

$$P\left(C=k|\mathbf{x}\right)=\frac{\exp\left(\mathbf{w}_k^T\mathbf{x}\right)}{\sum_{j=1}^{K}\exp\left(\mathbf{w}_j^T\mathbf{x}\right)}$$

where $\mathbf{x}$ is the vector of inputs, K is the total number of categories, and $\mathbf{w}_k$ is the weight vector for category k.

In this problem you will implement a softmax classifier from scratch. **Do not use a toolbox.** Use the softmax (cross-entropy) loss with L_2 weight decay regularization. Your implementation should use stochastic gradient descent with mini-batches and momentum to minimize softmax (cross-entropy) loss of this single layer neural network. To make your implementation fast, do as much as possible using matrix and vector operations. This will allow your code to use your environment's BLAS. Your code should loop over epochs and mini-batches, but do not iterate over individual elements of vectors and matrices. Try to make your code as fast as possible. I suggest using profiling and timing tools to do this.

Train your classifier on the Iris dataset for 1000 epochs. You should either subtract the mean of the training features from the train and test data or normalize the features to be between -1 and 1 (instead of 0 and 1). Hand tune the hyperparameters (i.e., learning rate, mini-batch size, momentum rate, and L_2 weight decay factor) to achieve the best possible training accuracy. During a training epoch, your code should compute the mean per-class accuracy for the training data and the loss. After each epoch, compute the mean per-class accuracy for the testing data and the loss as well. **The test data should not be used for updating the weights.**

After you have tuned the hyperparameters, generate two plots next to each other. The one on the left should show the cross-entropy loss during training for both the train and test sets as a function of the number of training epochs. The plot on the right should show the mean per-class accuracy as a function of the number of training epochs on both the train set and the test set.

What is the best test accuracy your model achieved? What hyperparameters did you use? Would early stopping have helped improve accuracy on the test data?

Solution:
Best accuracy achieved in the train set is 0.833, while in the 0.848. Early stopping is normally used to avoid training over fitting. In this case, it seems not to be over fitting and, on the contrary, the model performs better in the test than in the train set. Therefore, early stopping would not be helpful when trying to achieve higher test accuracy or lower test loss.

This is not at all a common outcome when training models, it is actually the opposite. I believe this is due to the simplicity of this specific data set.

The hyperparameters which happened to perform better are the following:

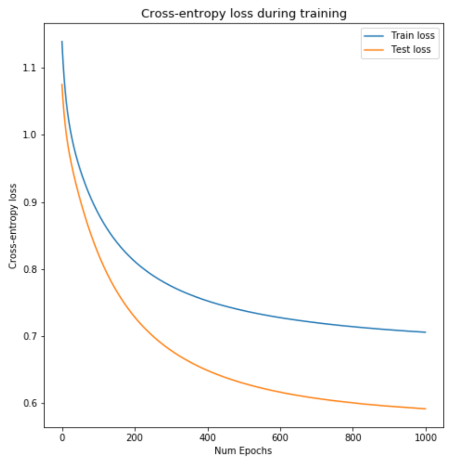
Momentum rate: 0.05

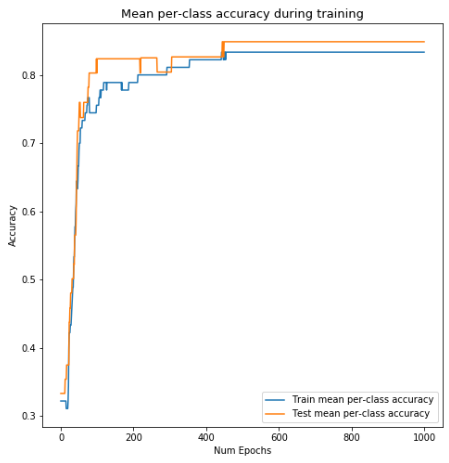
L2 wight decay regularization: 0.01

SGD with mini-batches of 10 examples each

For a total of 1000 epoch

With a learning rate of 0.01

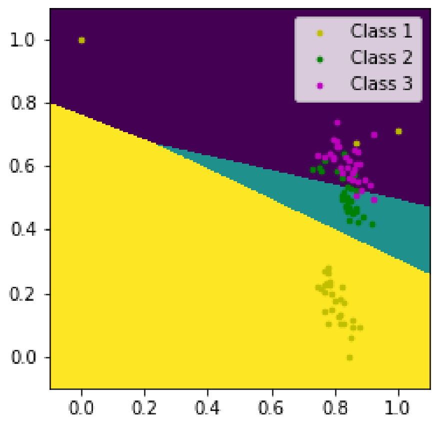




Part 2 - Displaying Decision Boundaries (10 points)

Plot the decision boundaries learned by softmax classifier on the Iris dataset, just like we saw in class. On top of the decision boundaries, generate a scatter plot of the training data. Make sure to label the categories.

Solution:



Problem 3 - Visualizing and Loading CIFAR-10 (5 points)

The CIFAR-10 dataset contains 60,000 RGB images from 10 categories. Download it from here: `https://www.cs.toronto.edu/~kriz/cifar.html`
Read the documentation.

Write a function to load the dataset, e.g.,
`trainLabels, trainFeat, testLabels, testFeat = loadCIFAR10()` where `trainLabels` has the categories for the training data, `trainFeat` has the 3072 dimensional image features from the training data, etc. Each of the returned variables should be NumPy arrays.

Using the first CIFAR-10 training batch file, display the first three images from each of the 10 categories as a 3×10 image array. The images are stored as rows, and you will need to reshape them into $32 \times 32 \times 3$ images if you load up the raw data yourself. It is okay to use the PyTorch toolbox for loading them or you can make your own.

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

def loadCIFAR10 ():
    trainLabels = []
    trainFeat = []
    testLabels = []
    testFeat = []

    for i in range(1,6):
        batch = unpickle("cifar-10-batches-py/data_batch_{}".format(i))
        trainLabels.extend(batch[b'labels'])
        trainFeat.extend(batch[b'data'])

    batch_te = unpickle("cifar-10-batches-py/test_batch")
    testLabels.extend(batch_te[b'labels'])
    testFeat.extend(batch_te[b'data'])

    return np.array(trainLabels), np.array(trainFeat), np.array(testLabels), np.array(testF


trainLabels, trainFeat, testLabels, testFeat = loadCIFAR10()
```

```
categories = {}
for i in range(len(trainFeat)):
    if trainLabels[i] in categories:
        categories[trainLabels[i]].append([(trainFeat[i])])
    else:
        categories[trainLabels[i]] = [(trainFeat[i])]


selected = []
for i in categories.keys():
    j = 0
    while j <= 2:
        selected.append(categories[i][j])
        j += 1


plt.figure(figsize=(17, 8))
columns = 10
order = np.array([a for a in range(1,31)])
index = order.reshape(3,10).T.flatten()
for i, image in enumerate(selected):
    new = np.array(image).reshape(3,32*32).T.reshape(32,32,3)
    plt.subplot(len(new) / columns + 1, columns, index[i])
    plt.imshow(new)
```

Problem 4 - Classifying Images (10 points)

Using the softmax classifier you implemented, train the model on CIFAR-10's training partitions. To do this, you will need to treat each image as a vector. You will need to tweak the hyperparmaters you used earlier.

Plot the training loss as a function of training epochs. Try to minimize the error as much as possible. What were the best hyperparmeters? Output the final test accuracy and a normalized 10×10 confusion matrix computed on the test partition. Make sure to label the columns and rows of the confusion matrix.

Solution:

The maximum test accuracy achieved is 0.412, while the maximum train accuracy is 0.426, indicating there is a certain degree of over fitting. We can also see it in the loss and the accuracy plots. When the lines separate then, over fitting comes in.

The hyperparameters which happened to perform better in terms of overfitting and maximized the test accuracy are the following:

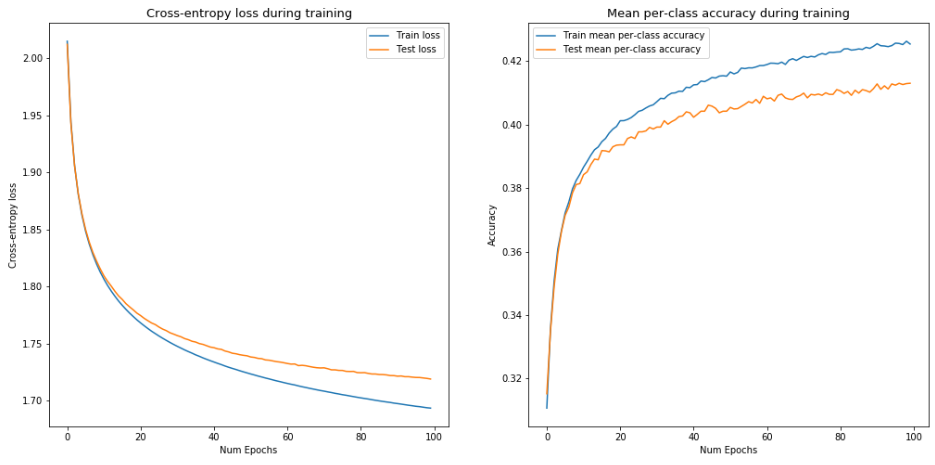
Momentum rate: 0.005

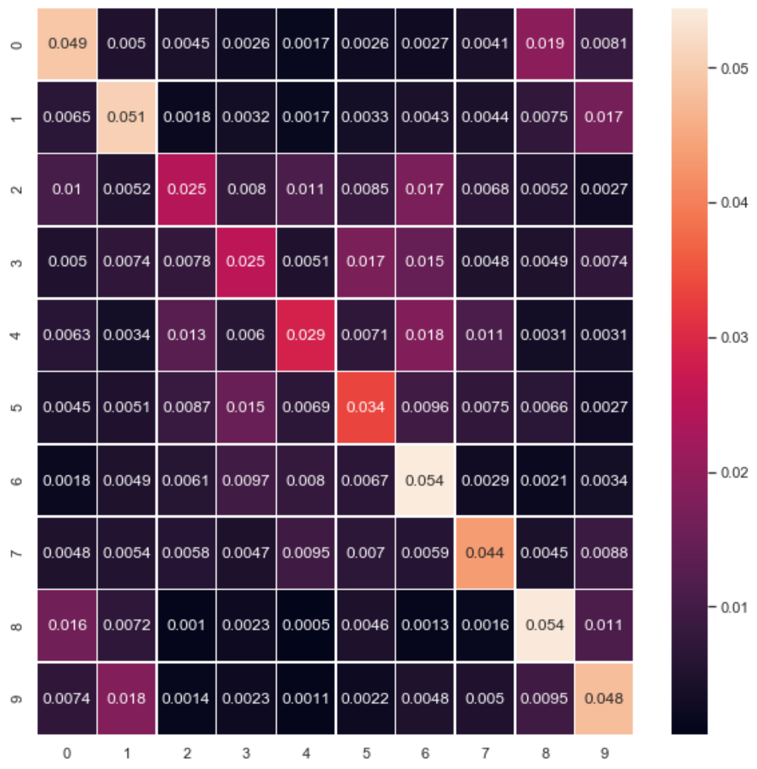
L2 wight decay regularization: 0.001

SGD with mini-batches of 10 examples each

For a total of 100 epoch

With a learning rate of 0.00001





	0	1	2	3	4	5	6	7	8	9
0	0.049	0.005	0.0045	0.0026	0.0017	0.0026	0.0027	0.0041	0.019	0.0081
1	0.0065	0.051	0.0018	0.0032	0.0017	0.0033	0.0043	0.0044	0.0075	0.017
2	0.01	0.0052	0.025	0.008	0.011	0.0085	0.017	0.0068	0.0052	0.0027
3	0.005	0.0074	0.0078	0.025	0.0051	0.017	0.015	0.0048	0.0049	0.0074
4	0.0063	0.0034	0.013	0.006	0.029	0.0071	0.018	0.011	0.0031	0.0031
5	0.0045	0.0051	0.0087	0.015	0.0069	0.034	0.0096	0.0075	0.0066	0.0027
6	0.0018	0.0049	0.0061	0.0097	0.008	0.0067	0.054	0.0029	0.0021	0.0034
7	0.0048	0.0054	0.0058	0.0047	0.0095	0.007	0.0059	0.044	0.0045	0.0088
8	0.016	0.0072	0.001	0.0023	0.0005	0.0046	0.0013	0.0016	0.054	0.011
9	0.0074	0.018	0.0014	0.0023	0.0011	0.0022	0.0048	0.005	0.0095	0.048

Figure: Test accuracy as a normalized confusion matrix

Problem 5 - Regression with Shallow Nets

Tastes in music have gradually changed over the years, and our goal is to predict the year of a song based on its timbre summary features. This dataset is from the 2011 Million Song Challenge dataset: `https://labrosa.ee.columbia.edu/millionsong/`

We wish to build a linear model that predicts the year. Given an input $\mathbf{x} \in \mathbb{R}^{90}$, we want to find parameters for a model $\hat{y} = \text{round}\left(f\left(\mathbf{x}\right)\right)$ that predicts the year, where $\hat{y} \in \mathbb{Z}$.

We are going to explore three shallow (linear) neural network models with different activation functions for this task.

To evaluate the model, you must round the output of your linear neural network. You then compute the mean squared error.

Part 1 - Load and Explore the Data (5 points)

Download the music year classification dataset from Canvas, which is located in `music-dataset.txt`. Each row is an instance. The first value is the target to be predicted (a year), and the remaining 90 values in a row are all input features. Split the dataset into train and test partitions by treating the first 463,714 examples as the train set and the last 51,630 examples as the test set. The first 12 dimensions are the average timbre and the remaining 78 are the timbre covariance in the song.

Write a function to load the dataset, e.g.,
`trainYears, trainFeat, testYears, testFeat = loadMusicData(fname, addBias)`
where `trainYears` has the years for the training data, `trainFeat` has the features, etc. `addBias` appends a '1' to your feature vectors. Each of the returned variables should be NumPy arrays.

Write a function `mse = musicMSE(pred, gt)` where the inputs are the predicted year and the 'ground truth' year from the dataset. The function computes the mean squared error (MSE) by rounding `pred` before computing the MSE.

Load the dataset and discuss its properties. What is the range of the variables? How might you normalize them? What years are represented in the dataset?

Generate a histogram of the labels in the train and test set and discuss any years or year ranges that are under/over-represented.

What will the test mean squared error (MSE) be if your classifier always outputs the most common year in the dataset?

What will the test MSE be if your classifier always outputs 1998, the rounded mean of the years?

Solution:

Function to load the dataset:

```
def normalize (x, mean, sd):
    return (x - mean)/sd

def loadMusicData(fname, addBias):
    df = pd.read_csv(fname, header=None, )

    train = df[:463715]
    test = df[463715:]

    x_train = train[range(1,91)].to_numpy()
    y_train = train[0].to_numpy()

    x_test = test[range(1,91)].to_numpy()
    y_test = test[0].to_numpy()

    mean_x = x_train.mean(axis=0)
    sd_x = x_train.std(axis=0)
    mean_y = y_train.mean(axis=0)
    sd_y = y_train.std(axis=0)

    x_train = normalize(x_train, mean_x, sd_x)
    x_test = normalize(x_test, mean_x, sd_x)
    y_train = normalize(y_train, mean_y, sd_y)
    y_test = normalize(y_test, mean_y, sd_y)

    x_train = np.clip(x_train, -3,3)
    x_test = np.clip(x_test, -3,3)

    x_train = np.c_[x_train, np.ones(len(x_train))*addBias]
    x_test = np.c_[x_test, np.ones(len(x_test))*addBias]

    x_train, y_train = shuffle(x_train, y_train)
    x_test, y_test = shuffle(x_test, y_test)

    return y_train, x_train, y_test, x_test, mean_y, sd_y
```

Mean-Square-Error function:

```
def musicMSE(y, pred):
    y1 = y * sd_y + mean_y
    pred1 = pred * sd_y + mean_y
    return np.sum((y1 - pred1)**2)/len(y)
```

Max value train: 2011

Min value train: 1922

Std train: 10.939755150678018

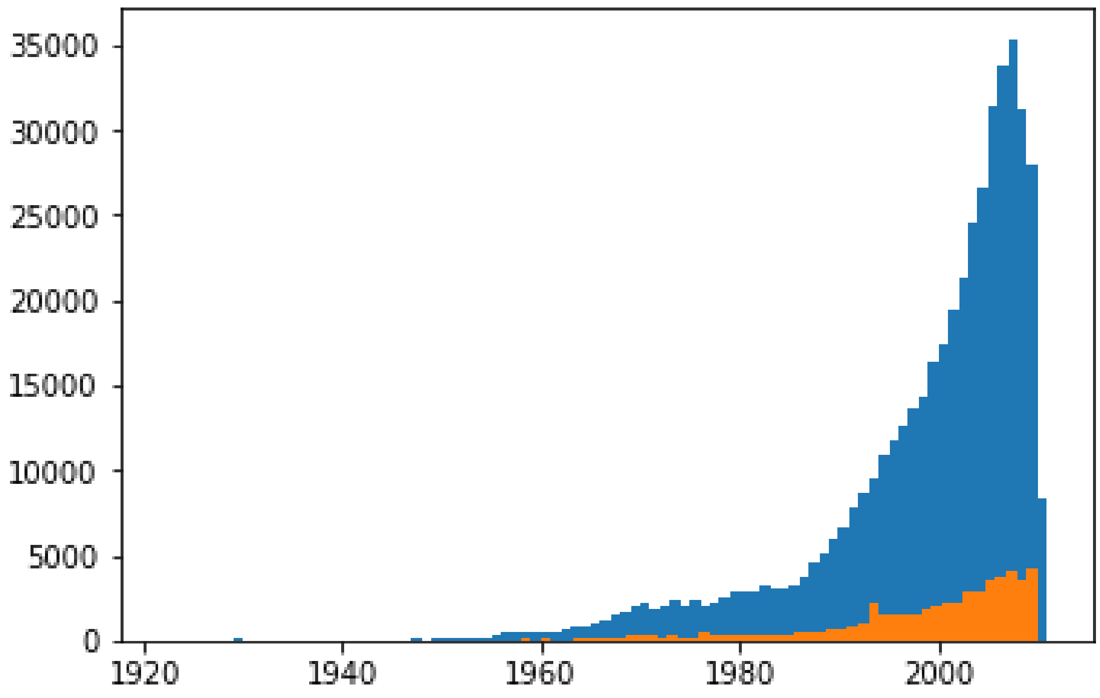
Median train: 2002.0

Max value test: 2010

Min value test: 1927

Std test: 10.851909820717683

Median test: 2002.0



Years range between 1922 and 2010. However, later years are highly over resented, representing a big percentage of the total sample.

As we can see in the plot, both test and train set have similar distributions even though not equal. In the train set the later years are exponentially over represented while in the test set they are linearly over represented. What will MSE output if the classifier always predict the most common year?

The mode of both the train set and the test set is 2007. In the test set with a count of 35375 and in the test set with a count of 4029.

After computing the MSE in the train we get the following:

```
pred = [2007 for i in range(len(y_train))]

def MSE (y, pred):
    return np.sum((y-pred)**2)/len(y)
```

```
MSE (y_train, pred)
```

```
Out: 193.87760370054883
```

And in the test the following:

```
pred = [2007 for i in range(len(y_test))]

def MSE (y, pred):
    return np.sum((y-pred)**2)/len(y)

MSE (y_test, pred)
```

```
Out: 190.08607398799148
```

What if the classifier always outputs 1998?

Train:

```
pred = [1998 for i in range(len(y_train))]

def MSE (y, pred):
    return np.sum((y-pred)**2)/len(y)

MSE (y_train, pred)
```

```
Out: 119.82731203433143
```

Test:

```
pred = [1998 for i in range(len(y_test))]

def MSE (y, pred):
    return np.sum((y-pred)**2)/len(y)

MSE (y_test, pred)
```

```
Out: 118.00972302924656
```

Part 2 - Ridge Regression (10 points)

Possibly the simplest approach to the problem is linear ridge regression, i.e., $\hat{y} = \mathbf{w}^T\mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^d$ and we assume the bias is integrated by appending a constant to $\mathbf{x}$. The 'ridge' refers to L_2 regularization, which is closely related to L_2 weight decay.

Minimize the loss using gradient descent, just as we did with the softmax classifier to find $\mathbf{w}$. The loss is given by

$$L = \sum_{j=1}^{N} \left\| \mathbf{w}^T \mathbf{x}_j - y_j \right\|_2^2 + \alpha \left\| \mathbf{w} \right\|_2^2 ,$$

where $\alpha > 0$ is a hyperparameter, N is the total number of samples in the dataset, and y_j is the j-th ground truth year in the dataset. Differentiate the loss with respect to $\mathbf{w}$ to get the gradient descent learning rule and give it here. Use stochastic gradient descent with mini-batches to minimize the loss and evaluate the train and test MSE. Show the train loss as a function of epochs.

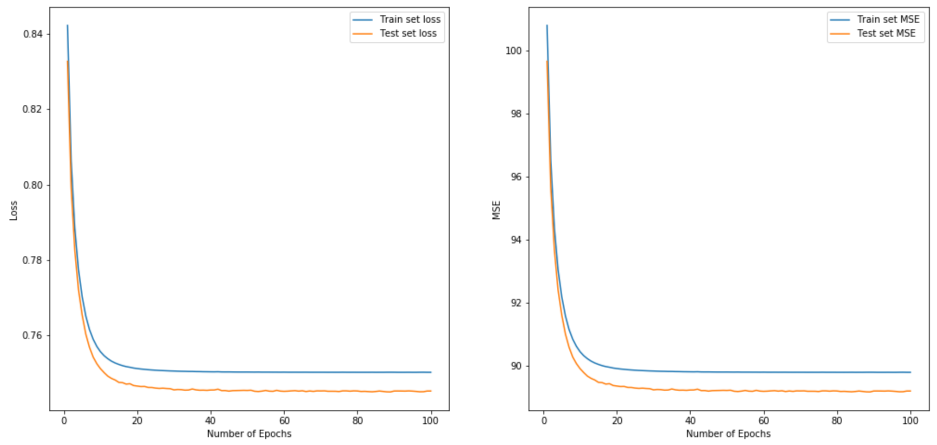
As you probably learned in earlier courses, this problem can be solved directly using the pseudoinverse. Compare both solutions.

Tip: Debug your models by using an initial training set that only has about 100 examples and make sure your train loss is going down.

Tip: If you don't use a constant (bias), things will go very bad. If you don't normalize your features by 'z-score' normalization of your data then things will go very badly. This means you should compute the training mean across feature dimensions and the training standard deviation, and then normalize by subtracting the training mean from both the train and test sets, and then divide both sets by the train standard deviation.

Solution:

Ridge Regression Performance using SGD with mini batches:



```
def pseudoinverse(xs, ys):
    return np.linalg.pinv(xs.T.dot(xs)).dot(xs.T).dot(ys)

pseudoinverse_w = np.array([pseudoinverse(x_train, y_train)]).T

pred = x_test @ pseudoinverse_w
MSE = musicMSE(y_test, pred)
```

MSE = 89.17

Following both procedures (Stocastic Gradient Descent or pseudoinverse), the resulting MSE is the same.

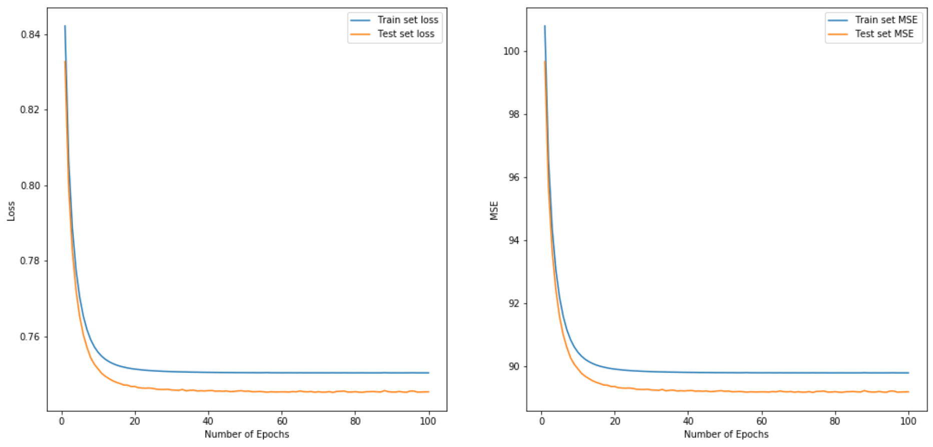
Part 3 - L_1 Weight Decay (10 points)

Try modifying the model to incorporate L_1 regularization (L_1 weight decay). The new loss is given by

$$L = \sum_{j=1}^{N} \left\| \mathbf{w}^T \mathbf{x}_j - y_j \right\|_2^2 + \alpha \left\| \mathbf{w} \right\|_1^1 .$$

Tune the weight decay performance and discuss results. Plot a histogram of the weights for the model with L_2 weight decay (ridge regression) compared to the model that uses L_1 weight decay and discuss.

Solution:



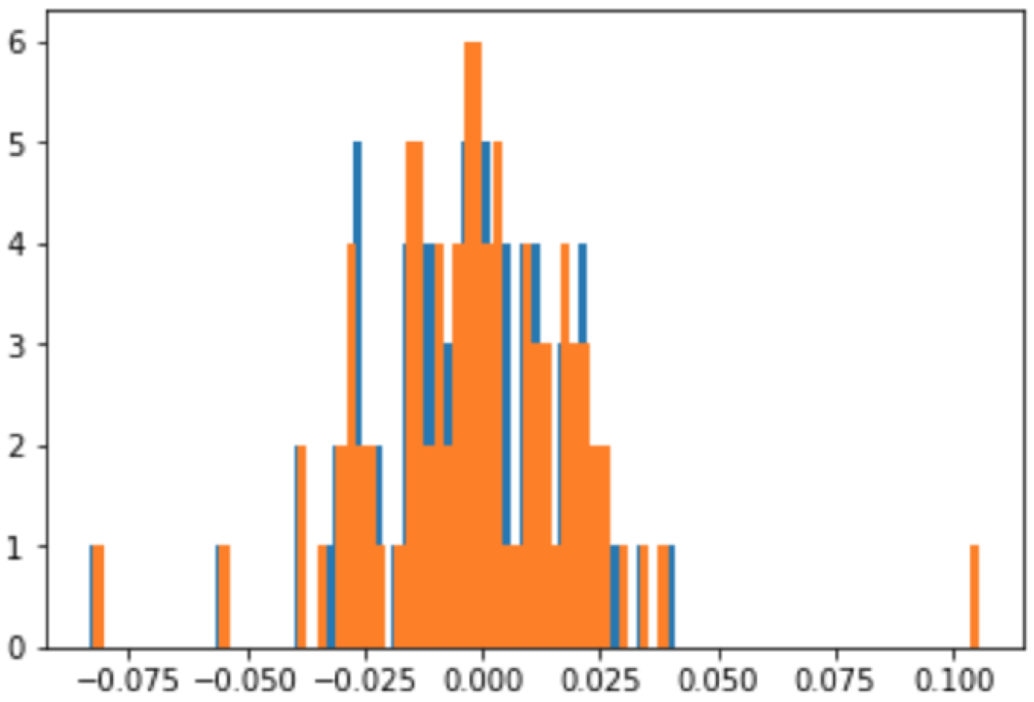


Figure: L2 vs L1 weights histogram. There is barely a difference in terms of weights distribution.

Part 4 - Poisson (Count) Regression (10 points)

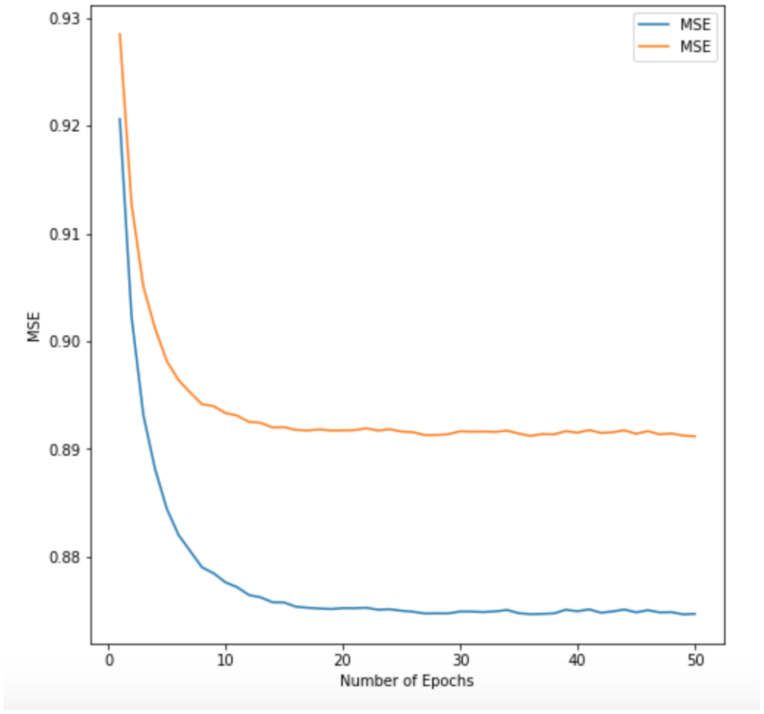
A potentially interesting way to do this problem is to treat it as a counting problem. In this case, the prediction is given by $\hat{y} = \exp\left(\mathbf{w}^T\mathbf{x}\right)$, where we again assume the bias is incorporated using the trick of appending a constant to $\mathbf{x}$.

The loss is given by

$$L = \sum_{j=1}^{N} \left(\exp\left(\mathbf{w}^T\mathbf{x}_j\right) - y_j\mathbf{w}^T\mathbf{x}_j\right),$$

where we have omitted the L_2 regularization term. Minimize it with respect to parameters/weights $\mathbf{w}$ using SGD with mini-batches. Plot the loss. Compute the train and test MSE using the function we created earlier.

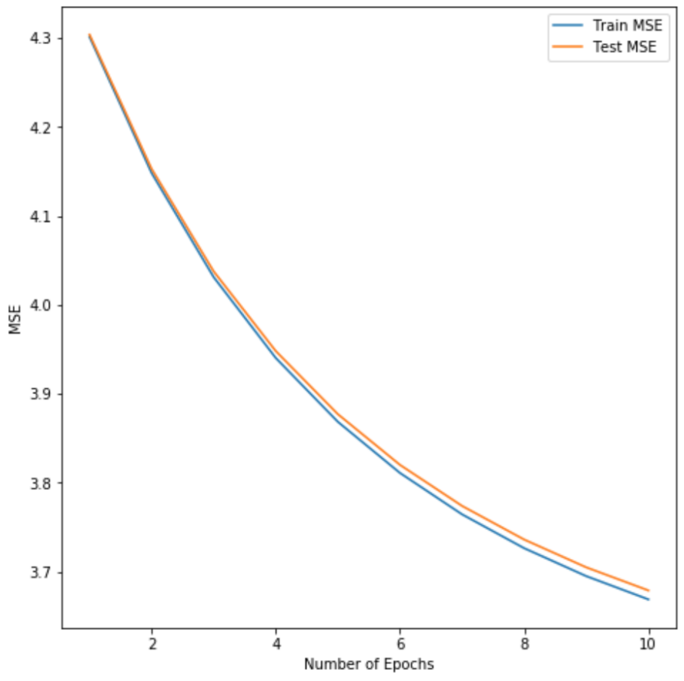
Solution:



Part 5 - Classification (5 points)

One way to do this problem is to treat it as a classification problem by treating each year as a category. Use your softmax classifier from earlier with this dataset and compute the MSE for the train and test dataset. Discuss the pros and cons of treating this as a classification problem.

Solution:



Using classification instead of regression for a problem like this is dangerous. By using the years as classes we can lose the sense of separation. Imagine we have only three possible outcomes: 1990, 1991, and 2000. The model will only understand that there are 3 different classes but it will not understand that there is a much higher difference between 1991 and 2000 than between 1990 and 1991. Therefore, the final MSE can not be a good metric of success.

Explained in another way, the model can be consistently predicting one year off (above or below) the true year. In a regression model the error would not be big, as one year off is not much and it is almost predicting it right every time. However, in a classification, this means that the model is predicting a wrong class every time, so the MSE will be huge.

Part 6 - Model Comparison (10 points)

Discuss and compare the behaviors of the models. Are there certain periods (ranges of years) in which models perform better than others? Where are the largest errors across models. Did L_2 regularization help for some models but not others?

Solution:
An identified difference between softmax and the linear models, is that L2 regularization has an impact in linear, while it does not really matter in the softmax case.

In general, the data set is very unbalanced meaning that some years (early ones) have few examples while the later ones are over represented. In this conditions, we have to find a model that works better than the softmax in unbalanced datasets. The model predicts better the oldest years than the early years.