Mehmet Burak Sayıcı
21602940

EEE485 Phase III Report

Problem: Image classification problem is not a new problem, there are several algorithms. Although leading algorithm is Convolutional Neural Networks, we are able to solve the problems with multinomial logistic regression and multilayer perceptron. Methods will be explained in this report.

Three methods have loss function and there are two main weight regularisation methods, L1 and L2 regularisation. They reflect to the weights in a different way and it's observable with weight visualisation methods. It is analysed in this report.

Dataset: MNIST dataset contains handwritten digit images. There are 60000 training images, 10000 test images. Image sizes are 28x28 pixels. Images are grey coloured, means they have one channel between 0-255. Preparation of the dataset is explained in this paper.

Before deeper investigation, it should be noted that array shape is [60000,28,28] for training input set, [60000,1] for the training output set. However, outputs are converted to one hot encoding principles. Their processed shape is [60000,10]. It is explained in this report.

MNIST has also test set that has 10000 images.

After defining the model parameters, dataset is separated into three sets; training(%60), validation(%20), testing(%16.6) to investigate learning rate, L1 and L2 Loss parameters.

Mathematical Concepts Before Diving into The Algorithms:

Matrix Multiplication for weights,bias and input of the layer :

This simple formula is the core of the regression problems and neural networks. We have input *X*, weights *W*, and bias *B* that is responsible for sending inputs to some value. We want to find weights that are optimal or substantially good for our purposes. Although there are many notations and orders of multiplications, given *X*,*W* and *B* matrix, my base formulation is:

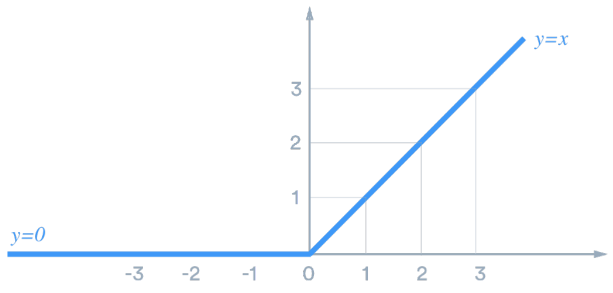
$$X \cdot W + B$$

Activation Functions: Activation functions are responsible for determining the output of the neurons. In neural networks, there are a lot of cases that we use stack of $X \cdot W + B$'s. Even for a single layer of neural network (or for multinomial logistic regression), bad design of weight matrix *W* will result in high values of $X \cdot W + B$ matrix. To squash the outputs, there are lots of activation functions such as sigmoid and tanh. Limit problem seems solvable for the forward propagation with squashing activation functions, however, the squashing effect is then responsible for vanishing gradient problem. One proposed activation function is ReLU, problems and benefits is discussed below.

Mehmet Burak Sayıcı
21602940

- ReLu Function: Rectified Linear Unit is basically a function that makes negative values of array zero.

$$ReLU(X_j) = max(0, X_j)$$



Here's the graph of ReLU, it's interpretable that it's non differentiable at x = 0, and differentiable at everywhere else. In practise, it's assumed that derivative at x = 0 is 0. It's unbounded, it does not squash the values and it can lead "inf" problems during training. Derivative is zero for x<0, that cause dying ReLU problem if weights or learning rates are not carefully designed. I'll discuss about the careful design at weight initialisation part. However, they're successful at diminishing the effects of vanishing gradient problem and it's computationally efficient. That's why I choose ReLU as an activation function.

- Softmax Function: It's basically normalised exponential function and very similar to the sigmoid function.

$$Softmax(X_j) = \frac{e^{X_j \cdot W + B}}{\sum_{j=1}^{n} e^{X \cdot W + B}}$$

- Softmax activation function helps us to represent probabilities because it's range is between [0,1], and $\sum_{j=1}^{n} \frac{e^{X_j \cdot W + B}}{\sum_{j=1}^{n} e^{X \cdot W + B}} = 1$. It helps us to model probability.

Loss Functions: Categorical Cross Entropy is the loss function for multi class outputs. That's why I'm using it for all methods.

- Categorical Cross Entropy: $CCE(y, \hat{y}) = -\sum y log(\hat{y})$
  - Categorical Cross Entropy with Weight Regularization: Regularization is for penalising the model complexity, as well as preserving the success on accuracy. Thus, it can be said that it prevents overfitting. Weight parameter and weight is added to the categorical cross entropy loss in two way. One is called L2 Regularization: $CCE(y, \hat{y}, W) = -\sum y log(\hat{y}) + \lambda \sum_{i=1} \sum_{j=1} W_{i,j}^2$, and the other is called L1 Regularization: $CCE(y, \hat{y}, W) = -\sum y log(\hat{y}) + \lambda \sum_{i=1} \sum_{j=1} \left| W_{i,j} \right|$. L1 Loss is unstable since surface that is around optima is flat due to the absolute value. L2 Loss is more stable since it increase the convexity with square term. L1 Loss has also one feature that we observe in this report,

feature selection. As I researched, there's no need to add regularisation to the bias since it has very limited effect.

Algorithms:

Multinomial Logistic Regression: Multinomial logistic regression is classification algorithm that has similar mathematical base with multinomial logistic regression but gives alternative solution to the multi class problems. There are one matrix of weights and one matrix of bias. Forward propagation of the process is as follows:

$X : [BatchSize, 28x28], W : [784,10], B : [1,10]$

Forward Propagation : $Softmax(X_j) = \frac{e^{X_j \cdot W+B}}{\sum_{j=1}^{n} e^{X \cdot W+B}}$.

This formula is the matrix form of the softmax function that is calculated for inputs multiplied by weights plus bias. For the loss, we have categorical cross entropy, $CCE(y, \hat{y}) = -\sum ylog(\hat{y})$. L1 and L2 loss is added to compare the weights via visualisation.

Multilayer Perceptron: In Multilayer Perceptron, one can stack more than one layers and add activation functions between them. For the sake of simplicity and to compare algorithms, I add one layer to the multinomial logistic regression and use ReLU for the activation function. Rest of the algorithm has the same idea with the multinomial logistic regression. Also, number of neurons for the first layer is chosen as 28x28 = 784 to plot a square image. This will allow us to see the difference between weights adjusted by CCE, CCE+L1, CCE+L2 loss in a 784x784 pixel images.

$X : [BatchSize, 28x28], W_1 : [784,784], B_1 : [1,784], W_2 : [784,10], B_2 : [1,10]$

Forward Propagation : $\delta j = max(X_j \cdot W + B, 0), Softmax(X_j) = \frac{e^{\delta_j}}{\sum_{j=1}^{n} e^{\delta_j}}$

For the loss, we have categorical cross entropy, $CCE(y, \hat{y}) = -\sum ylog(\hat{y})$. L1 and L2 is added to compare the weights via visualisation.

Convolutional Neural Network: Instead of having weight matrices that has [n_in,n_out] shape, CNN propose a solution with multiple weight matrices and convolution operation for the images. It can approach image locally and gives better solution than existing methods.

While regular nets do not scale well on full image, CNN reduce the number of parameters to be learned. MNIST has 28x28 images and neural network has 623290 parameters where CNN model has 34080.

Simulation Setup: Python and its libraries is used to implement and visualise algorithms. NumPy is used to implement matrix and array operations. MatPlotLib is used to visualise weights.

Data Preparation: Train and Test Data is normalised via min-max scaling method. The formula is $X_j = \frac{Xj - min(X)}{maxX - minX}$. Although standardisation is also another method (zero mean centering and dividing to the standard deviation), min-max scaling also works well.
Corresponding y values are also not in one hot encoding format originally, they're converted to one hot encoding format with algorithm that's available in appendix(in every code page). If an image is 1 and has label "1", it's one-hot encoding corresponder is [0,1,0,0,0,0,0,0,0,0,0].

For the training/test input data, input array shape is converted from [60000,28,28] to [60000,784]. Testing data has 1000 images.

Mathematical Function Implementations: Activation and loss functions are implemented with respect to formulas in this report, the only trick that I use for Categorical Cross Entropy is adding a small positive ϵ to avoid errors. Since $log(0)$ is not defined, Categorical Cross Entropy is now $CCE(y, \hat{y}) = -\sum ylog(\hat{y} + \epsilon)$ where $\epsilon = 1e - 15$.

Weight Initialisation: One of the important issues in modelling is weight initialisation. I thought my code for the models except multinomial logistic regression has some errors and searched for it for a very long time but I then realised that weights are not initialised properly. If there are more than one layers, weights should be initialised according to the activation function. ReLU is used as a activation function for the models except multinomial logistic regression(last layers always have softmax). According to "*Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*" paper, it can be seen that they proposed that standard deviation of the weights are formulated as

$$\sigma = \sqrt{\frac{2}{n_{in}}}$$

where n_{in} is the last matrix' output node. They advise that bias should have zero mean.

In addition, they indicate that the value for standard deviation for the first layer is not important, it should be set to 1. I follow this principle, for the multinomial logistic regression, first weight matrix has shape [784,784] and they're randomly created as $\mathcal{N}(\mu = 0.0, \sigma = 1, (784,784))$.
However, for the second weight initialisation sigma is calculated as

$$\sigma = \sqrt{\frac{2}{n_{in}}} = \sqrt{\frac{2}{784}} = 0.05050762723$$

Thus, second weight array is randomly created as $\mathcal{N}(\mu = 0.0, \sigma = 0.05050762723, (784,10))$.
It really affects the convergence and I would plan to add results as an extra work for the last report.

I know that it's not the focus of the report but the logic is keeping the input and output variances the same and this helps to the convergence due to the consistent values of outputs.

Let the output of the neurons δ_l and $\delta_l = X_l \cdot W_l + B_l$ and l is index for layers. Let $\gamma = k^2c$ where k is the pixel size of the square image and c be the number of channels. Then γ corresponds to the number of connections in $X_l \cdot W_l$. B_l is zero mean centred and out of consideration for now.

$$Var[\delta_l] = \gamma * Var[X_l \cdot W_l].$$

If we let $W_{l-1} \sim \mathcal{N}(\mu = 0)$, $Var[\delta_{l-1}]$ is zero centered. Then, for $Var[X_l]$, we can say $Var[\delta_{l-1}] = \frac{1}{2}Var[X_l]$. I am not exactly sure but reason for adding $\frac{1}{2}$ multiplier is, ReLU divides the dataset into two and converts values smaller than zero into the zero. Thus, this formula is valid for ReLU function, but I still feel there's ambiguity for me. But it does not harm the logic. If we continue from that perspective,

$$Var[\delta_l] = \gamma * Var[X_l \cdot W_l] = \gamma * \frac{1}{2} * Var[\delta_{l-1}] \cdot Var[W_l]$$

If we want to keep the variances of γ_l for all l same, then $\frac{Var[\delta_l]}{Var[\delta_{l-1}]} = 1$ and lastly

$$Var[W_l] = \sigma = \sqrt{\frac{2}{n_{in}}}$$

It means that we'll use initialisation for CNN second weight as

$$\sqrt{\frac{2}{n_{in}}} = \sqrt{\frac{2}{20x3x3}} = 0.10540925533$$

$$\mathcal{N}(\mu = 0.0, \sigma = 0.10540925533, (784,10))$$

Backpropagation Implementation: There is only one backpropagation implementation for multinomial logistic regression with L2 regularisation that is added to this reports, since others have the same idea. Mathematical formulation for others will be added according to the feedback.

Learning Rate Decay is implemented but not used for the sake of simplicity at parameter selection. It's possible to use it according to the feedback.

It is assumed that parameters are defined. Parameters for logging accuracy and loss is not added here, but they're available in the code.

```
$W = \mathcal{N}(\mu = 0.0, \sigma = 0.01, (784,10))$
$B = \mathcal{N}(\mu = 0.0, \sigma = 0.01, (784,10))$

for epoch in range(0,numberofepoch) :
    if epoch == weightdecayperiod:
        $\eta = \eta * (wd)$ # $wd$ is some value between 0 and 1
    for a in range(600):
        $Xforminibatch = X[100 * a : (100 * (a+1))]$ # 100 is mini-batch size
        $Yforminibatch = Y[100 * a : (100 * (a+1))]$ # a is number of mini-batches for
                                                      #one epoch
        $prediction = Xforminibatch \cdot W1$
        $loss = CCE(Yforminibatch, prediction, W)$
        $W = W - \eta * Xforminibatch^T \cdot (prediction - Yforminibatch)$
        $B = B - \eta * mean(prediction - Yforminibatch)$ # Where mean corresponds
                                                          #column mean
```

Parameter Selection:

Model parameters are arranged with respect to computational simplicity.

- Logistic regression has weights shaped [784,10] that helps us to easily visualise the weights. There is no additional layer. Having one layer is providing intuitive visualisation.
- Neural network has one hidden layer. Visualised weights are the hidden-to-output weights. Reason behind one hidden layer is not to diminish the effect of L1 and L2 Regularization. If there's more than one hidden layers and if we visualise lasthidden-to-output weights , feature selection of these losses may not be visible.
- ReLU is the activation function of CNN and Neural Network models. One could use sigmoid, however, ReLU doesn't change the weight values except weights smaller than zero. Sigmoid and TanH are nonlinear function whereas ReLU is linear function after zero. Since ReLU is the least manipulative on the weights, I thought that feature selection of L1 and L2 losses will be more visible.
- CNN has kernel size (3,3) and has 20 kernels. CNN Architecture that is famous with mobility and fast training, MobileNet, has set of (3,3) and (1,1) of filters. Kernel size is adjusted with respect to this, filter size is reduced to increase the speed.
- Weights are initialised according to He initialisation.

Hyperparameter Selection:

After defining the model parameters, dataset is separated into three sets; training(%60), validation(%20), testing(%16.6). Training and validation datasets are used for the hyper parameter selection. Chosen hyper parameters are learning rate, L1 and L2 lambda parameters. Batch size is 32 for all models, it is not investigated and directly fixed. Epoch is 10.

Candidate learning rates are chosen after unfinished training for each algorithm, means that learning rate values that results in very low accuracy are not considered. They are trained on only training and validation dataset and their performance is compared. Best learning rate is chosen among them. Then, candidate lambda values are chosen separately for L1 and L2 loss. Again, they are trained on only training and validation dataset and their performance is compared. Then, training and validation dataset is combined for training. Test data is only used to investigate how well the algorithms are performed at their best conditions(best learning rate and lambda rate among the candidates).

Numba to Increase Speed of the CNN: Numba is just-in-time compiler that converts mathematical operations to the C Language. Roughly speaking, CNN with Numba is 26 times faster. CNN with Numba can finish one epoch at nearly 150 seconds but CNN without Numba can finish one epoch at 4000 seconds(more than one hour).

However, Numba is not helpful when I try to add L1 and L2 factors to the back propagation equations. One epoch requires more than one hour, thus, I couldn't give results of CNN with L1 and L2 loss. Results of CNN without L1 and L2 loss is provided in this report.

Hyperparameter Selection Results:

Multinomial Logistic Regression:

Multinomial Logistic Regression, Learning Rate Investigation

	1E-02	1E-03	1E-04
Training Accuracy	**%91.383**	%86.679	%70.860
Training Loss	**1.463**	3.127	9.031
Validation Accuracy	**%91.041**	%88.016	%73.241
Validation Loss	**0.355**	0.526	1.312

Multinomial Logistic Regression, LR =1E-2, L1 Loss Lambda Selection

	1E-03	1E-04	1E-05
Training Accuracy	%85.518	%90.9708	**%91.4875**
Training Loss	8.065	2.923	**1.659**
Validation Accuracy	%84.266	%90.350	**%91.091**
Validation Loss	0.723	0.409	**0.985**

Multinomial Logistic Regression, LR =1E-2, L2 Loss Lambda Selection

	1E-03	1E-04	1E-05
Training Accuracy	%90.110	%91.852	**%91.560**
Training Loss	4.548	2.551	**1.609**
Validation Accuracy	%88.991	%91.283	**%91.25**
Validation Loss	0.811	0.465	**0.387**

Multinomial Logistic Regression, Chosen Parameters are: Learning Rate = 1e-2, lambda = 1e-5 for L1 and L2 Loss.

Mehmet Burak Sayıcı
21602940

Neural Network, Learning Rate Investigation

	1E-02	1E-03	1E-04
Training Accuracy	%94.033	**%95.264**	%90.022
Training Loss	0.354	**0.102**	0.067
Validation Accuracy	%87.341	**%93.033**	%89.691
Validation Loss	1.403	**0.334**	0.491

Neural Network, LR =1E-3, L1 Loss Lambda Selection

	1E-04	1E-05	1E-06
Training Accuracy	%82.043	**%95.481**	%95.266
Training Loss	4.276	**4.665**	0.700
Validation Accuracy	%82.816	**%93.400**	%93.150
Validation Loss	4.447	**4.825**	0.923

Neural Network, LR =1E-3, L2 Loss Lambda Selection

	1E-04	1E-05	1E-06
Training Accuracy	%94.204	**%95.272**	%95.260
Training Loss	0.164	**0.098**	0.102
Validation Accuracy	%92.85	**%93.016**	%93.025
Validation Loss	0.252	**0.310**	0.332

Neural Network:
Multinomial Logistic Regression, Chosen Parameters are: Learning Rate = 1e-3, lambda = 1e-5 for L1 and L2 Loss**.**

Mehmet Burak Sayıcı
21602940

Results after Trained on Test Dataset:

Algorithms at Their Best Condition

	Multinomial L1	Multinomial L2	Neural Network L1	Neural Network L2	CNN
Training Accuracy	%91.853	%91.898	%93.011	**%93.785**	%87.4
Training Loss	2.195	2.092	0.018	**0.009**	0.116
Testing Accuracy	%90.78	%90.87	%92.416	**%95.375**	%85.691
Testing Loss	0.376	0.375	1.660	**0.849**	0.785

Algorithms are trained on combination of validation and training dataset, then tested on test dataset. This is not the same procedure when diagnosing which parameter is best among chosen ones, test dataset is not used at there.

Results show that Multilayer Perceptron works better than Multinomial Logistic Regression in most cases, when both training and testing accuracy is compared between algorithms. It should be noted that number of epoch is fixed to 10 and batch size 32 for all models.

It is hard to tell an exact reason for that. Multilayer Perceptron has more parameters to learn. Multinomial Logistic models has 784*10 + 10 = 7850 parameters to learn while Multilayer Perceptron models 784x784 + 784 + 784x10 + 10 = 623290 parameters to learn. CNN has 34080 parameters and it is good option.

CNN model is the worst, however, there can be a small implementation that I didn't realise but affects the training procedure negatively. Number of kernel should be increased to increase performance. It can be the reason for accuracy. L1 and L2 loss for CNN can not be applied due to the high waiting time. There's only CNN with CCE No Regularisation.

Better visualisation of the accuracy and loss for algorithms are appended to the appendix.

Visualising Weights: Weights are visualised directly in a grey colormap. MatPlotLib library gives black colour to the highest value of the array, besides the lowest value in array has white color.

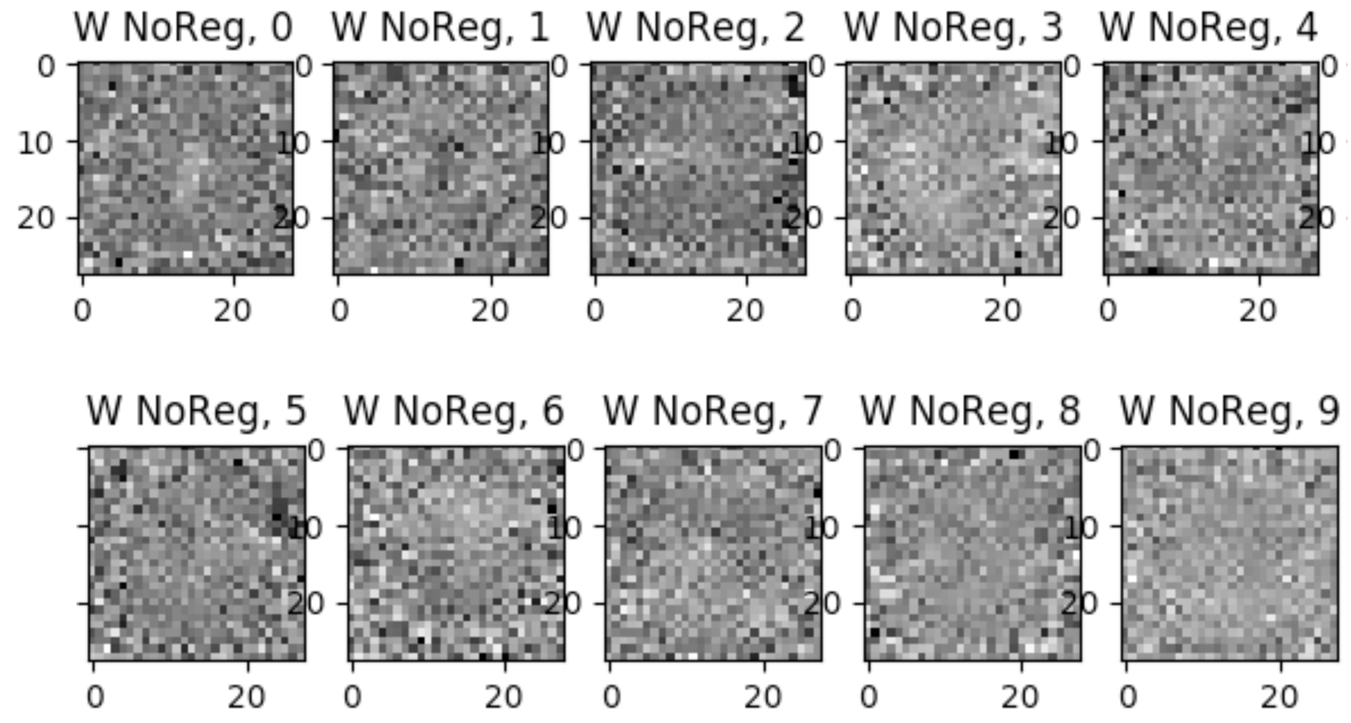
For multinomial logistic regression, it's very easy to see the effects of the loss functions. However, trial-and-error method for learning rate and lambda parameters are applied to obtain better visualisation. Learning rate is set to 1e-2 and lambda is set to 1e-2 after some trial.

For multilayer perceptron, it's hard to interpret the weights when it's compared to the multinomial logistic regression, but we can still comment on the effects of the loss functions. Multilayer perceptron weights are the same weights that results can be seen on the table. Last weights of the multilayer perceptron is visualised since L1 and L2 loss is only applied for the last weights at this work.

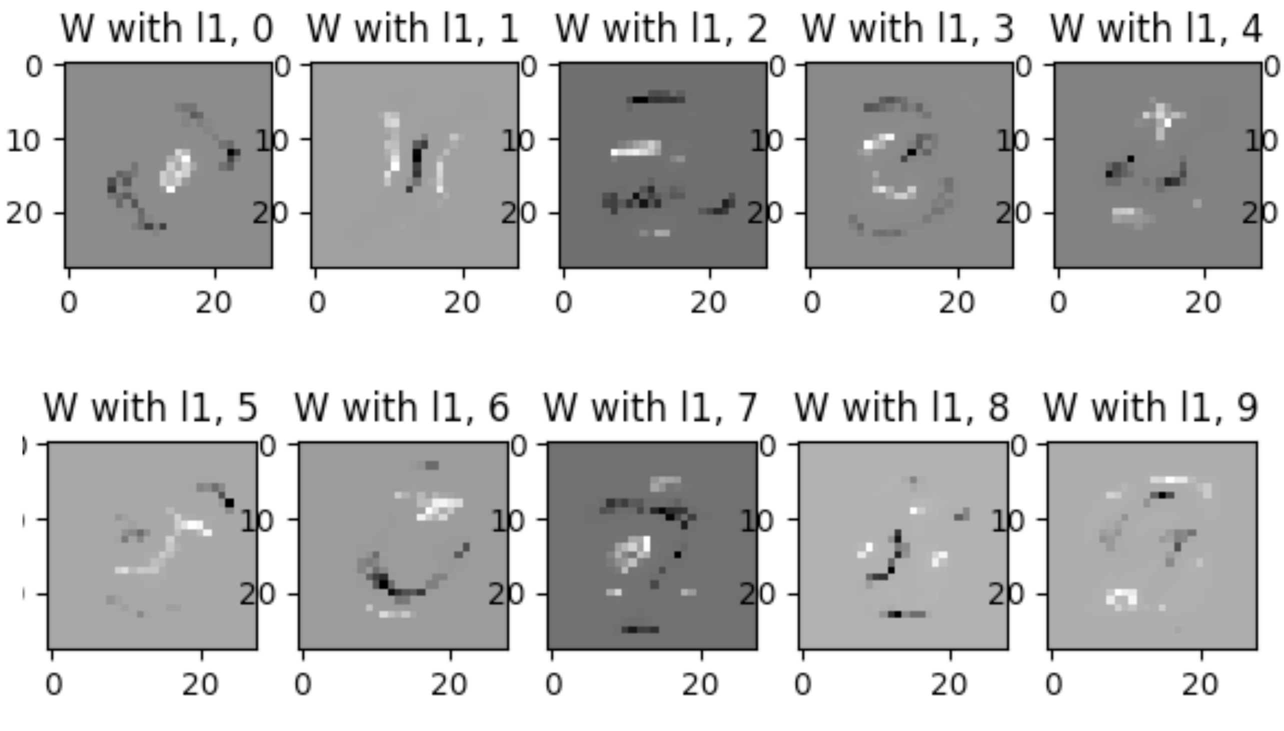
Multinomial logistic regression has weight with shape [784,10]. It's reshaped to [28,28,10] and plotted with respect to the order.
CNN visualisation can not be done due to the time. It takes 1 hour for one epoch for now. Interpretation part is after the images.

Mehmet Burak Sayıcı
21602940

Weight Visualisation with No Regularisation for Multinomial Logistic Regression:

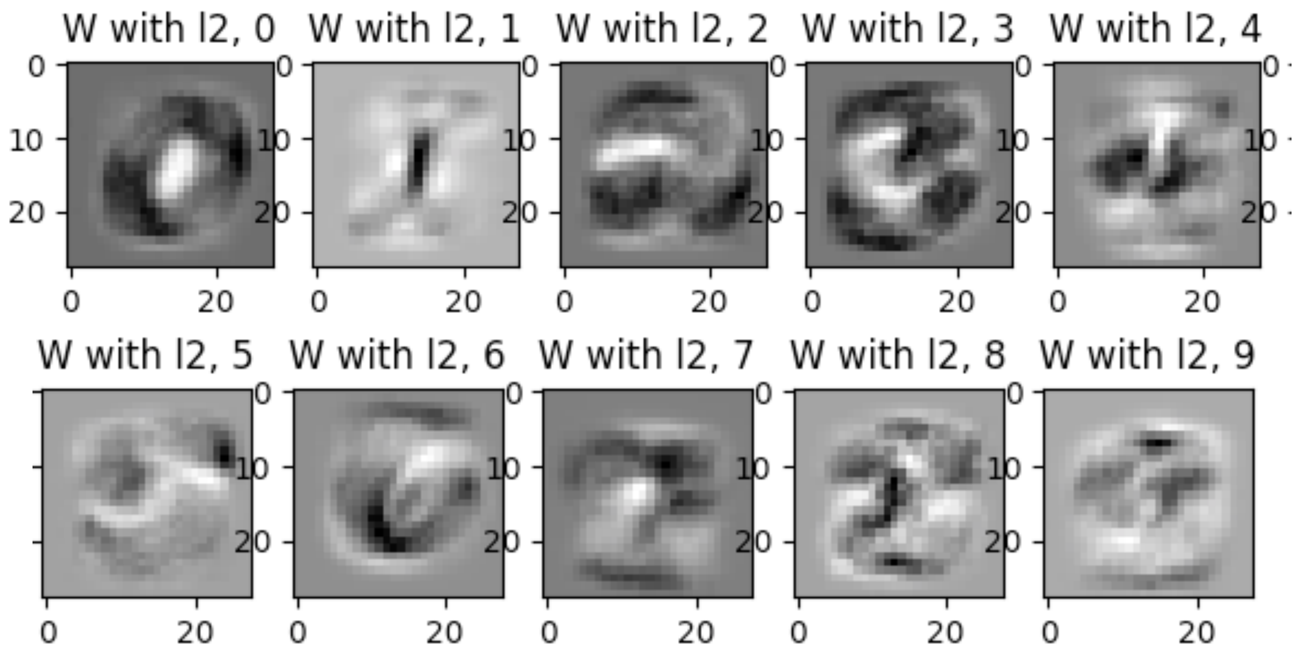


Weight Visualisation with L1 Loss for Multinomial Logistic Regression:

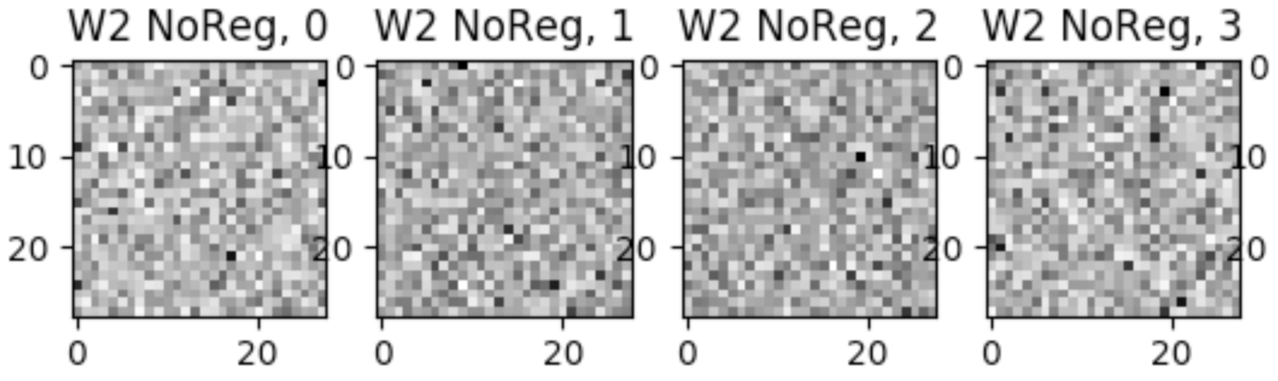


Mehmet Burak Sayıcı
21602940

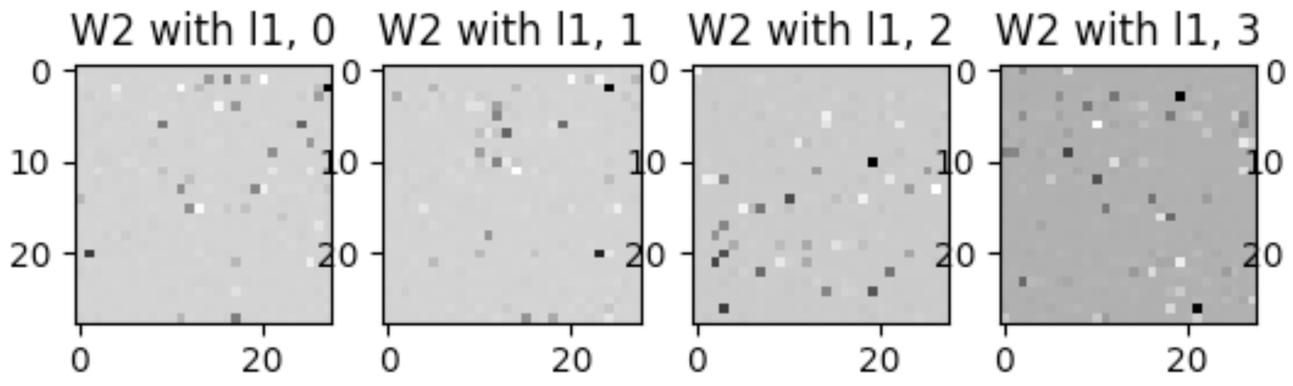
Weight Visualisation with L2 Loss for Multinomial Logistic Regression:



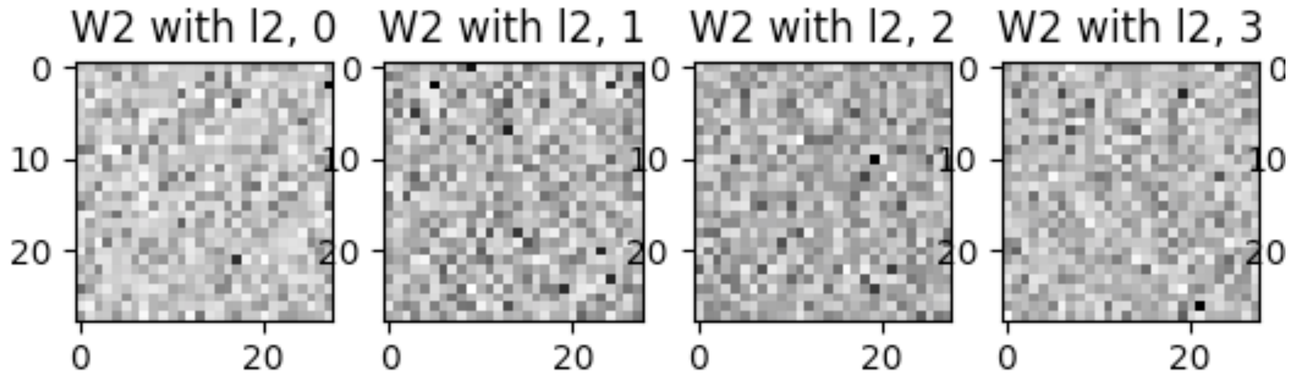
Weight Visualisation with No Regularization for Multilayer Perceptron(Only 4 digits here, full images are at appendix):



Weight Visualisation with L1 Regularization for Multilayer Perceptron(Only 4 digits here, full images are at appendix):



Weight Visualisation with L2 Regularization for Multilayer Perceptron(Only 4 digits here, full images are at appendix):



Interpretation of the Weight Visualisation: For all types of the loss functions, weights are penalised in a different way. We do know that L1 Loss is a feature selector. L1 Loss shrinks less important features to the zero. L1 Loss is a sparse model. It is easier for L1 Loss to set weights 0 than L2 Loss. In L1 Loss, most of the features are set to zero and some of the features has very high weights. On the other hand, it is hard to get 0 value for L2 Loss since it squares the weights and weights stay at the decimal form.

Let’s assume that images that can be seen in Multinomial Logistic Regression with No Loss is our base weights. It can be seen that weights are not penalised. There’s no exact feature that we can distinguish by looking at it.

L1 Loss acts as a feature selector for Multinomial Logistic Regression. It can be seen that it exposes some features and gives similar to the other weights. Black pixels indicates that Multinomial Logistic Regression is giving more importance to these points while classification. White means that these are the less important.

I said that L1 Loss is a feature selector, however, L2 Loss is more feature selector than no regularisation. All digits are identifiable with weights. The most significant result that I observe for Multinomial Logistic Regression with L2 Loss is the difference between 0 and 1 digit weights. White means that model doesn’t care much about these pixels, however, black means that model is looking for the black pixels (or weights) to understand what the input digit is. This is because black pixels have largest weights.

"0 digit" weight visualisation corresponds to the first element of reshaped weight

For "0 digit" weight visualisation at Multinomial Logistic Regression with L2 Loss, it can be seen that inside of the 0 is the whitest space. It means that, if input digit is 1, we'll observe low probability of being 0. Because the values of digit 1 image is concentrated on the middle area, and we know that weight for 0 has very small values for that.

On the other hand, for "1 digit" weight visualisation at Multinomial Logistic Regression with L2 Loss, we see that highest weights are concentrated on center and it looks like 1. Around of high weights is mostly white, because most of the images that has label 1 is easily distinguishable with the shape of 1.

For the weight visualisation for Multilayer Perceptron, we can't see good images as Multinomial Logistic Regression. However, it can be seen that, again, L1 Loss again works as a feature selector. It sets most of the weights to zero and highlights important features.

Expected Results of CNN: Although I couldn't produce images for CNN, my plan was to forward pass the image with the kernel that is trained on L1, L2 and No Regularisation. Since L1 Kernels are feature selector, they'll result in feature selected images such as weight with L1 Regularisation for Multinomial Logistic Regression. And same applies for l2

Advantages of CNN: CNN can reach very high accuracies with very low number of parameters. We know that
Multinomial Logistic models has 784*10 + 10 = 7850 parameters to learn while Multilayer Perceptron models 784x784 + 784 + 784x10 + 10 = 623290 parameters to learn. In CNN model, we have 20*3*3 + 3390*13 = 34080 parameters to learn.

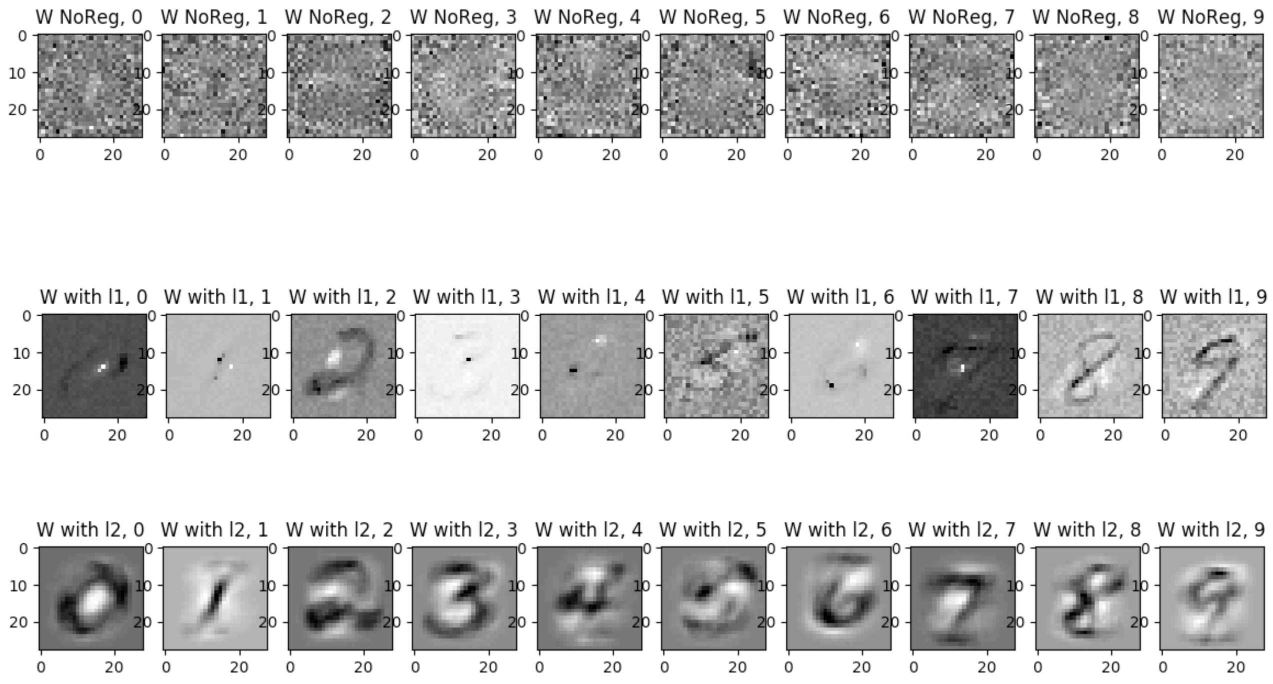
Why PyTorch is Faster: They do not produce differentiation formulas, they use AutoGrad system. Also, some systems use finite difference formula to estimate the derivative. It's more faster than traditional backpropagation calculation by hand.

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

Mehmet Burak Sayıcı
21602940

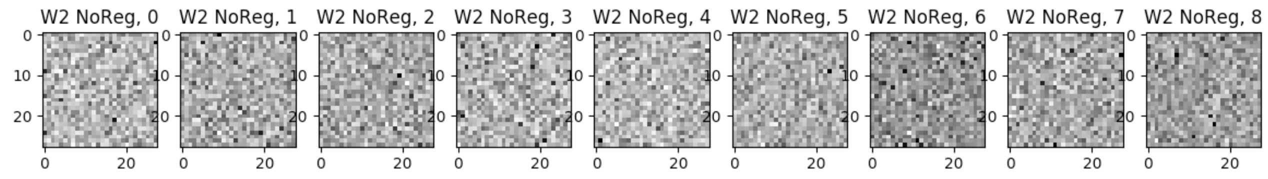
Appendix:

1)Visualisation of the weights of Logistic Regression(lr = 1e-2, lambda = 1e-2)

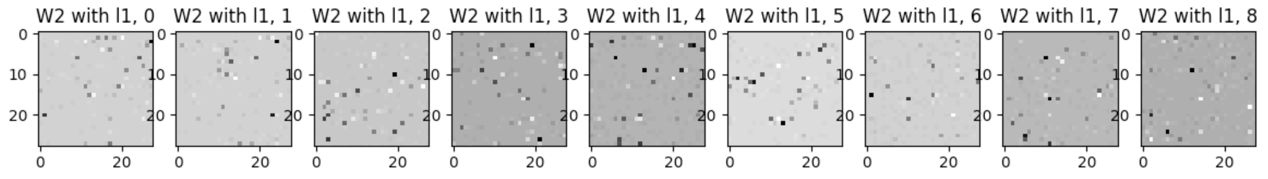


2)Visualisation of the weights of Multilayer Perceptron:

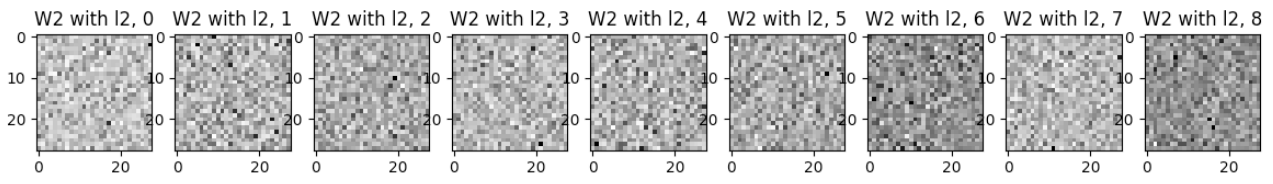
No Regularisation:



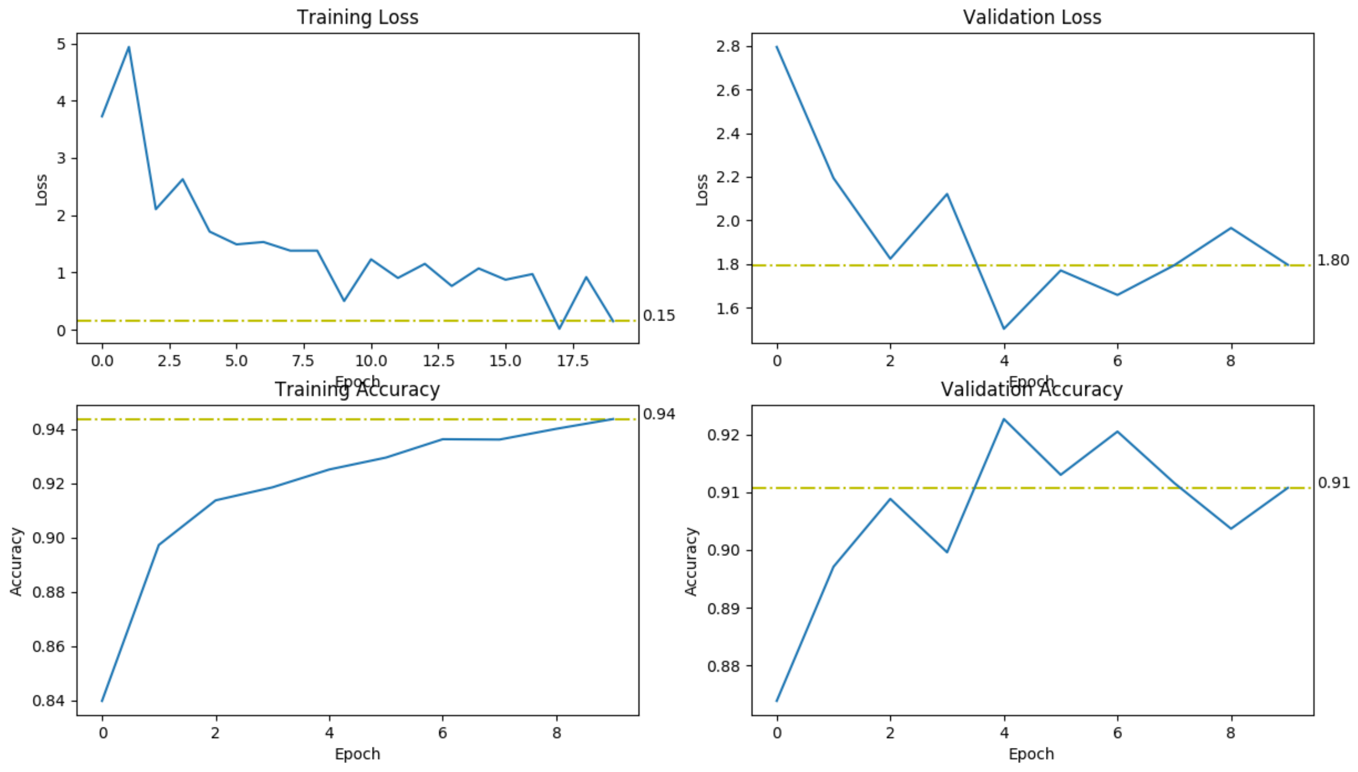
L1 Loss:



L2 Loss:



3) One Example Graph: Neural Network with Learning Rate = 1e-3, Lambda = 1e-5



4) Codes:

Neural Network:

```
import mnist
import numpy as np
import matplotlib.pyplot as plt
import time
import random
np.random.seed(42)
from numba import jit,njit


#Data Prep---
x_train, y_train, x_test, y_test = mnist.load()
x_train = x_train.reshape(-1,28*28)


numbofimages = 60000
mnistx = x_train/255
mnisty = y_train.reshape(1,numbofimages)
b = np.zeros((numbofimages,10)) # 10 is number of classes
b[np.arange(numbofimages), mnisty] = 1
mnisty = b
#---
```

```
mnistxtest = x_test/255
numbofimages = 10000
mnistytest = y_test.reshape(1,numbofimages)
b = np.zeros((numbofimages,10)) # 10 is number of classes
b[np.arange(numbofimages), mnistytest] = 1
mnistytest = b






def softmax(x):
    return np.exp(x)/np.sum(np.exp(x),axis=1,keepdims=True)



def relu(x):
    x.copy()
    out = x*(x > 0)
    return out



def relu_derivative(x):

    out = x.copy()
    out[out <= 0] = 0
    return out


print("oluyo")


def cceforward(x,y,w1,w2,lambdaa = 1e-5,reg = "CCE"):
    if reg == "CCE":
        return np.sum(-y * np.log(x +  1e-15) / x.shape[0]) # x + 1e-15
    elif  reg == "CCEL2":
        return np.sum(-y * np.log(x  + 1e-15) / x.shape[0])+ lambdaa*np.sum(np.square(w2))# +
lambdaa*np.sum(np.square(w1))
    elif reg == "CCEL1":
        return np.sum(-y * np.log(x  + 1e-15) / x.shape[0]) + lambdaa*np.sum(np.abs(w2))  # +
lambdaa*np.sum(np.abs(w1))





@jit
def train(losstype="CCEL1",lr=0.01, epoch= 10,  batchsize=32, weightdecay = 1,weightdecayperiod=
10,randomweights = True,
          node=700,testdatasetx=mnistx[48000:60000],testdatasety = mnisty[48000:60000]):
#testdatasetx=mnistxtest[0:10000],testdatasety = mnistytest[0:10000]):



    if randomweights == True:
        w1 = np.random.normal(0.0, 1 , (784, node))
        w2 = np.random.normal(0.0,  0.07216878364  , (node, 10))
        b1 = np.random.normal(0.0,1, (1, node))
        b2 = np.random.normal(0.0, 0.07216878364, (1, 10))



    traininglosstable = np.array([])
    testinglosstable = np.array([])
    trainingepochtable = np.array([])
    testingepochtable = np.array([])

    for i in range(epoch):
        beforeepoch = time.time()

        if i % weightdecayperiod == 0 and i > 1:
            lr = lr * weightdecay

        trainingepochaccuracy = 0
        testingepochaccuracy = 0

        lambdaa = 1e-5
        howmanyimagesfortraining = 30000 # 60000
        repet = int(howmanyimagesfortraining / batchsize)
        losssum = 0


        for a in range(repet):

            mnistxtrain = mnistx[batchsize * a:(batchsize * (a + 1))]
            mnistytrain = mnisty[batchsize * a:(batchsize * (a + 1))]

            layer1output = np.dot(mnistxtrain, w1)
            layer1outputrelu = relu(layer1output)
            #print(layer1outputrelu)

            pred = softmax(np.dot(layer1outputrelu, w2) + 0)

            # For computational efficiency
            w2temp = w2.copy()
            # layer1outputsigmoid = sigmoid(layer1output)
            predminusy = (pred-mnistytrain)
            reluder = relu_derivative(predminusy)

            reluderw2 = np.dot(reluder,w2temp.T)

            xreluderw2 = np.dot(mnistxtrain.T,reluderw2)
            #predminusyw2 = np.dot(predminusy , w2temp.T )




            #print("A:",a)
            if losstype == "CCE":
                loss = cceforward(pred, mnistytrain,w1,w2, reg=losstype, lambdaa=lambdaa)
                losssum = losssum + loss
                #print(losstype)
                w2 = w2 - lr *(np.dot(layer1outputrelu.T, predminusy))#ok for relu / batchsize  # - lambdaa*w2 #
    - lambdaa*abs(w2)/w2   LASSO


                b2 = b2 - lr * ((np.mean(predminusy, axis=0, keepdims=True))  / batchsize )# *0.001  # -
    lambdaa*abs(b2)/b2                # ok for relu


                w1 = w1-  lr *(xreluderw2/ batchsize ) # -lambdaa*w1


                b1 = b1 - lr * (np.mean( reluderw2, keepdims=True, axis=0) / batchsize  )#*0.001



            elif losstype == "CCEL2":
                loss = cceforward(pred,mnistytrain, w1,w2,   reg=losstype, lambdaa=lambdaa)
                losssum = losssum + loss
                w2 = w2 - lr *(np.dot(layer1outputrelu.T, predminusy)) - lambdaa*w2 #  - lambdaa*abs(w2)/w2
    LASSO

                b2 = b2 - lr * ((np.mean(predminusy, axis=0, keepdims=True))  / batchsize ) # - lambdaa*abs(b2)/
    b2

                w1 = w1 - lr *(xreluderw2/ batchsize ) #  -lambdaa*w1

                b1 = b1 - lr * (np.mean( reluderw2, keepdims=True, axis=0))

            elif losstype == "CCEL1":

                loss = cceforward(pred, mnistytrain, w1,w2, reg=losstype, lambdaa=lambdaa)
                losssum = losssum + loss

                w2 = w2 - lr *(np.dot(layer1outputrelu.T, predminusy)) - lambdaa*abs(w2)/w2

                b2 = b2 - lr * ((np.mean(predminusy, axis=0, keepdims=True))  / batchsize )

                w1 = w1-  lr *(xreluderw2/ batchsize )  # -lambdaa*abs(w1)/w1

                b1 = b1 - lr * (np.mean( reluderw2, keepdims=True, axis=0))
            row_maxes = pred.max(axis=1).reshape(-1, 1)
            pred[:] = np.where(pred == row_maxes, 1, 0)
            result = np.all(pred == mnistytrain, axis=1)

            trainingepochaccuracy += np.sum(1 * result, axis=0)

            now = time.time()

            #Training Loss
        traininglosstable = np.append(traininglosstable, losssum/repet)




        #Testing Loss and Accuracy
        layer1testoutput = np.dot(testdatasetx, w1)
        layer1testoutputrelu = relu(layer1testoutput)
        #print(layer1outputrelu)

        predtest = softmax(np.dot(layer1testoutputrelu, w2))
```

```
        losstest = cceforward(predtest, testdatasety, w1, w2, reg=losstype, lambdaa=lambdaa)



        row_maxes = predtest.max(axis=1).reshape(-1, 1)
        predtest[:] = np.where(predtest == row_maxes, 1, 0)


        resulttest = np.all(predtest == testdatasety, axis=1)
        testingepochaccuracy += np.sum(1 * resulttest, axis=0)





        print(i+1, "th epoch:")
        print(i+1,"th epoch took: ", now - beforeepoch, " seconds")
        now = 0
        beforeepoch = 0

        print("Training Epoch Accuracy/Testing Epoch Accuracy:", (trainingepochaccuracy / 
howmanyimagesfortraining),"-",testingepochaccuracy / 12000)
        print("Training Loss/Testing Loss:", losssum/repet,"-",losstest)

        testinglosstable = np.append(testinglosstable, losstest)
        traininglosstable = np.append(traininglosstable,loss)
        trainingepochtable = np.append(trainingepochtable,trainingepochaccuracy/howmanyimagesfortraining)
        testingepochtable = np.append(testingepochtable,testingepochaccuracy/12000)
        loss = 0
        losstest = 0
        losssum = 0



    #np.save("weight1fornnl1.npy",w1)
    np.save("weight2fornnl1.npy",w2)
    #np.save("bias1fornnl1.npy",b1)
    #np.save("bias2fornnl1.npy",b2)
    #np.save("1weight1.npy", w1)
    return traininglosstable, testinglosstable, trainingepochtable, testingepochtable

"""
    np.save("weight1fornnl2.npy",w1)
    np.save("weight2fornnl2.npy",w2)
    np.save("bias1fornnl2.npy",b1)
    np.save("bias2fornnl2.npy",b2)
    #np.save("1weight1.npy", w1)

"""


    #np.save("1bias1.npy", b1)
    #np.save("1weight2.npy", w2)
    #np.save("1bias2.npy", b2)
    # np.save("bias.npy",b)
# Train
learningrate = 1e-3
traininglosstable, testinglosstable, trainingepochtable, testingepochtable =
train(losstype="CCEL2",lr=learningrate, epoch= 10,
                                        batchsize=32, weightdecay = 1,weightdecayperiod= 10000,
                                        randomweights = True,node=784)


import matplotlib.pyplot as plt


print(traininglosstable)
plt.figure(figsize=(16,8))
titleforgraph = "Neural Network : CCE, lr = %0.02f, epoch = 10, "%learningrate
plt.title(titleforgraph)
plt.subplot(2,2,1)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.annotate('%0.2f' % traininglosstable[-1], xy=(1, traininglosstable[-1]), xytext=(2, 0),
            xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=traininglosstable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(traininglosstable)),traininglosstable)




plt.subplot(2,2,2)
plt.title("Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.annotate('%0.2f' % testinglosstable[-1], xy=(1, testinglosstable[-1]), xytext=(2, 0),
            xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=testinglosstable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(testinglosstable)),testinglosstable)

plt.subplot(2,2,3)
plt.title("Training Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.annotate('%0.2f' % trainingepochtable[-1], xy=(1, trainingepochtable[-1]), xytext=(2, 0),
            xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=trainingepochtable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(trainingepochtable)),trainingepochtable)

plt.subplot(2,2,4)
plt.title("Validation Accuracy") #Validation
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.annotate('%0.2f' % testingepochtable[-1], xy=(1, testingepochtable[-1]), xytext=(2, 0), xycoords=('axes
fraction', 'data'), textcoords='offset points')
plt.axhline(y=testingepochtable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(testingepochtable)),testingepochtable)

plt.savefig("nn1e-4lambda1e-5L2")
plt.show()
```

```
"""

# Weight init ---
w1 = np.random.normal(0.5,0.1,(784,784))
w2 = np.random.normal(0.5,0.1,(784,10))
#w = np.load("weight.npy")

b1 = np.random.normal(0.5,0.1,(1,784))
b2 = np.random.normal(0.5,0.1,(1,10))

b1 = np.random.normal(0.5,0.1,(1,784))
b2 = np.random.normal(0.5,0.1,(1,10))

w1 = np.load("1weight1.npy")
w2 = np.load("1weight2.npy")

b2 = np.load("1bias2.npy")
b1 = np.load("1bias1.npy")

"""
```

Mehmet Burak Sayıcı
21602940

Multinomial Logistic Regression:

```
import mnist
import numpy as np
import matplotlib.pyplot as plt
import time
import random
np.random.seed(42)



#Data Prep---
x_train, y_train, x_test, y_test = mnist.load()
x_train = x_train.reshape(-1,28*28)

#Normalizing train
numbofimages = 60000
mnistx = x_train/255
#One hot encoding for train
mnisty = y_train.reshape(1,numbofimages)
b = np.zeros((numbofimages,10)) # 10 is number of classes
b[np.arange(numbofimages), mnisty] = 1
mnisty = b
#---

#Normalizing
mnistxtest = x_test/255
numbofimages = 10000
#One hot encoding for train
mnistytest = y_test.reshape(1,numbofimages)
b = np.zeros((numbofimages,10)) # 10 is number of classes
b[np.arange(numbofimages), mnistytest] = 1
mnistytest = b


mnistxtest = mnistxtest[0:12000]
mnistytest = mnistytest[0:12000]


# Weight init ---
w = np.random.normal(0,1,(784,10))



b = np.random.normal(0,1,(1,10))
#---




def softmax(x):
    return np.exp(x)/np.sum(np.exp(x),axis=1,keepdims=True)
```

Mehmet Burak Sayıcı
21602940

```
def cceforward(x,y,w,reg,lambdaa):
    if reg == "CCE":

        return np.sum(-y * np.log(x +  1e-15) / x.shape[0]) # x + 1e-15

    elif  reg == "CCEL2":
        return np.sum(-y * np.log(x  + 1e-15) / x.shape[0])+ lambdaa*np.sum(np.square(w))
    elif reg == "CCEL1":
        return np.sum(-y * np.log(x  + 1e-15) / x.shape[0]) + lambdaa*np.sum(np.abs(w))
```

```
epoch = 10
traininglosstable = np.array([])
testinglosstable = np.array([])
trainingepochtable = np.array([])
testingepochtable = np.array([])

for i in range(epoch):

    losstype = "CCE"
    lr = 1e-2

    howmanyimagesfortraining = 60000
    batchsize = 32
    repet = int(howmanyimagesfortraining / batchsize)




    learningdecayperiod = 10
    learningdecay = 1
    losssum = 0

    beforeepoch = time.time()

    if i % learningdecayperiod == 0 and i > 1:
        lr = lr * learningdecay

    trainingepochaccuracy = 0
    testingepochaccuracy = 0

    lambdaa = 1e-2 # 1e-2 l2 için mthiş
    for a in range(repet):


        mnistxtrain = mnistx[batchsize * a:(batchsize * (a + 1))]
        mnistytrain = mnisty[batchsize * a:(batchsize * (a + 1))]

        pred = softmax(np.dot(mnistxtrain, w)+b)


        loss = cceforward(pred,mnistytrain,w=w,reg=losstype,lambdaa=lambdaa)

        if losstype == "CCEL1":

            w = w - lr*np.dot(mnistxtrain.T,(pred-mnistytrain)) - lambdaa*np.abs(w)/w  # L1
            b = b - lr*(np.mean(pred-mnistytrain,axis=0,keepdims=True))


        elif losstype == "CCEL2":
            w = w - lr*np.dot(mnistxtrain.T,(pred-mnistytrain)) -lambdaa*w  # L2
            b = b - lr*(np.mean(pred-mnistytrain,axis=0,keepdims=True))

        else:
            w = w - lr*np.dot(mnistxtrain.T,(pred-mnistytrain))
            b = b - lr*(np.mean(pred-mnistytrain,axis=0,keepdims=True))


        losssum = losssum + loss

        row_maxes = pred.max(axis=1).reshape(-1, 1)
        pred[:] = np.where(pred == row_maxes, 1, 0)
        result = np.all(pred == mnistytrain, axis=1)

        trainingepochaccuracy += np.sum(1 * result, axis=0)

        now = time.time()

#       epochaccuracy += np.sum(1*result,axis=0)





        # Testing Loss and Accuracy


    predtest = softmax(np.dot(mnistxtest, w)+b)

    losstest = cceforward(predtest,mnistytest, w, reg=losstype, lambdaa=lambdaa)

    row_maxes = predtest.max(axis=1).reshape(-1, 1)
    predtest[:] = np.where(predtest == row_maxes, 1, 0)

    resulttest = np.all(predtest == mnistytest, axis=1)
    testingepochaccuracy += np.sum(1 * resulttest, axis=0)

    print(i, "th epoch:")
    print(i, "th epoch took: ", now - beforeepoch, " seconds")
    now = 0
    beforeepoch = 0

    print("Training Epoch Accuracy/Testing Epoch Accuracy:", (trainingepochaccuracy /
howmanyimagesfortraining), "-",
          testingepochaccuracy / 10000)
    print("Training Loss/Testing Loss:", loss, "-", losstest)

    testinglosstable = np.append(testinglosstable, losstest)
    traininglosstable = np.append(traininglosstable, loss)
    trainingepochtable = np.append(trainingepochtable, trainingepochaccuracy / howmanyimagesfortraining)
    testingepochtable = np.append(testingepochtable, testingepochaccuracy / 10000)
    loss = 0
    losstest = 0
    losssum = 0



np.save("multilogcce",w)
#np.save("blogisticbestl1",b)





import matplotlib.pyplot as plt


print(traininglosstable)
plt.figure(figsize=(16,8))
plt.subplot(2,2,1)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.annotate('%0.2f' % traininglosstable[-1], xy=(1, traininglosstable[-1]), xytext=(2, 0),
             xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=traininglosstable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(traininglosstable)),traininglosstable)




plt.subplot(2,2,2)
plt.title("Testing Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.annotate('%0.2f' % testinglosstable[-1], xy=(1, testinglosstable[-1]), xytext=(2, 0),
             xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=testinglosstable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(testinglosstable)),testinglosstable)

plt.subplot(2,2,3)
plt.title("Training Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.annotate('%0.2f' % trainingepochtable[-1], xy=(1, trainingepochtable[-1]), xytext=(2, 0),
             xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=trainingepochtable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(trainingepochtable)),trainingepochtable)

plt.subplot(2,2,4)
plt.title("Testing Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.annotate('%0.2f' % testingepochtable[-1], xy=(1, testingepochtable[-1]), xytext=(2, 0), xycoords=('axes
fraction', 'data'), textcoords='offset points')
plt.axhline(y=testingepochtable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(testingepochtable)),testingepochtable)
plt.savefig("logl2bestvis")
plt.show()
```

Convolutional Neural Network:

```
import numpy as np
import mnist
import numpy as np
import matplotlib.pyplot as plt
import time
from numba import jit,njit,prange, optional
import numba




#Data Prep---
x_train, y_train, x_test, y_test = mnist.load()
x_train = x_train.reshape(-1,28*28)


numbofimages = 60000
mnistx = x_train/255
mnisty = y_train.reshape(1,numbofimages)
b = np.zeros((numbofimages,10)) # 10 is number of classes
b[np.arange(numbofimages), mnisty] = 1
mnisty = b
#---


#weightcnn1 = np.random.normal(0,0.01,(20,3,3))
#weightfcn = np.random.normal(0,0.01,(3380,10))
#bfcn = np.random.normal(0,0.2,(1,10))








@jit
def relu(x):
    x.copy()
    out = x*(x > 0)
    return out
```

```
@jit
def cceforward(x,y,w1,lambdaa = 1,reg = "CCE"):
    if reg == "CCE":
        return np.sum(-y * np.log(x + 1e-15) / x.shape[0])
    elif  reg == "CCEL2":
        return np.sum(-y * np.log(x  + 1e-150) / x.shape[0]) + lambdaa*np.sum(np.square(w1))
    elif reg == "CCEL1":
        return np.sum(-y * np.log(x  + 1e-15) / x.shape[0]) + lambdaa*np.sum(np.abs(w1))



@jit
def softmax(x):
    return np.exp(x)/np.sum(np.exp(x),axis=1,keepdims=True)




@jit
def ForwardConv(inputarray, stride, weights, flatten = True):

    filternumber = weights.shape[0]
    height = weights.shape[1]
    width = weights.shape[2]
    stride = stride
    numbofimagesinputarray = inputarray.shape[0]


    hoffmap = inputarray[0].shape[0]  # hoffmap -> height of feature map


    woffmap = inputarray[0].shape[1]  # woffmap -> width of feature map

    widthtour = int(1 + (woffmap - width) / stride)

    heighttour = int(1 + (hoffmap - height) / stride)
    # print(widthtour)
    # print(heighttour)
    featuremap = np.zeros((inputarray.shape[0], filternumber, widthtour, heighttour),dtype=np.float64)

    for i in range(inputarray.shape[0]):
        for j in range(filternumber):
            for k in range(int(heighttour)):
                for l in range(int(widthtour)):
                    # if channel is implemented  we need another for loop for channel
                    # print(stride*l,width+stride*(l),"---")
                    array_ = inputarray[i][stride * k:height + stride * k,
                             stride * l:width + stride * l]


                    # print(inputarray[i][l:height,width*l:width*(l+1)].shape)
                    # print(i, j, k, l)
                    featuremap[i, j, k, l] = np.sum(np.multiply(array_, weights[j]))
    if flatten== False:
        return featuremap.reshape(-1,k+1,l+1)
    else:
        return featuremap.reshape(numbofimagesinputarray,-1)
```

```
@jit
def BackwardConv(inputarray, stride, weights,derivweights,lr,losstype,lambdaa):
    filternumber = weights.shape[0]
    height = weights.shape[1]
    width = weights.shape[2]
    stride = stride
    numbofimagesinputarray = inputarray.shape[0]

    hoffmap = inputarray[0].shape[0]  # hoffmap -> height of feature map

    woffmap = inputarray[0].shape[1]  # woffmap -> width of feature map

    widthtour = int(1 + (woffmap - width) / stride)

    heighttour = int(1 + (hoffmap - height) / stride)
    # print(widthtour)
    # print(heighttour)
    featuremap = np.zeros((inputarray.shape[0], filternumber, widthtour, heighttour))

    for i in range(inputarray.shape[0]):
        for j in range(20):
            for k in range(int(heighttour)):
                for l in range(int(widthtour)):
                    # if channel is implemented  we need another for loop for channel
                    # print(stride*l,width+stride*(l),"---")
                    array_ = inputarray[i][stride * k:height + stride * k, stride * l:width + stride * l]


                    # print(inputarray[i][l:height,width*l:width*(l+1)].shape)
                    # print(i, j, k, l)
                    if losstype =="CCE":
                        weights[j] = weights[j]-lr*array_*np.sum(derivweights[i,j,:,:])
                    elif losstype == "CCEL2":
                        weights[j] = weights[j] - lr * array_ * np.sum(derivweights[i, j, :, :]) - lambdaa*weights[j]
                    elif losstype == "CCEL1":
                        weights[j] = weights[j] - lr * array_ * np.sum(derivweights[i, j, :, :]) - lambdaa
*abs(weights[j])/weights[j]

    return weights
```

Mehmet Burak Sayıcı
21602940

```
def train(losstype="CCE",lr=0.0001, epoch=10,  batchsize=32, weightdecay = 1,weightdecayperiod=
10,lambdaa=1e-5,
        testdatasetx=mnistx[480:].reshape(-1,28,28),testdatasety = mnisty[480:]):

    traininglosstable = np.array([])
    testinglosstable = np.array([])
    trainingepochtable = np.array([])
    testingepochtable = np.array([])

    weightcnn1 = np.random.normal(0, 0.01, (20, 3, 3))
    weightfcn = np.random.normal(0, 0.02432521277, (3380, 10))
    bfcn = np.random.normal(0, 0.2, (1, 10))

    for i in range(epoch):
        beforeepoch = time.time()

        if i % weightdecayperiod == 0 and i > 1:
            lr = lr * weightdecay

        trainingepochaccuracy = 0
        testingepochaccuracy = 0
        lambdaa = 1e-5
        howmanyimagesfortraining = 320 # 60000
        losssum = 0
        repet = int(howmanyimagesfortraining / batchsize)



        for a in range(repet):

            mnistxtrain = mnistx[batchsize * a:(batchsize * (a + 1))].reshape(-1,28,28)
            mnistytrain = mnisty[batchsize * a:(batchsize * (a + 1))]


            cnnout = ForwardConv(mnistxtrain, 2, weightcnn1, flatten=True)
            #relucnnout = relu(cnnout)
            cnntopred =  np.dot(cnnout,weightfcn) #+ bfcn
            pred = softmax(cnntopred)

            weightfcntemp = weightfcn.copy() # For grad
            predminusy = pred-mnistytrain # For grad

            loss = cceforward(pred, mnistytrain, w1=weightcnn1,  reg=losstype, lambdaa=lambdaa)

            losssum = losssum + loss

            weightfcn = weightfcn - lr * np.dot(cnnout.T, predminusy)
            derivuntilcnn = np.dot(weightfcntemp, predminusy.T).reshape(-1, 20, 13, 13)

            weightcnn1 = BackwardConv(mnistxtrain, 2, weightcnn1, derivuntilcnn, lr,losstype,lambdaa)

            traininglosstable = np.append(traininglosstable, losssum)




            row_maxes = pred.max(axis=1).reshape(-1, 1)
            pred[:] = np.where(pred == row_maxes, 1, 0)
            result = np.all(pred == mnistytrain, axis=1)

            trainingepochaccuracy += np.sum(1 * result, axis=0)


            print(100 * a / repet, "% of Epoch",i+1," is finished")
            print("Batch Accuracy:",np.sum(1 * result, axis=0)/32)

            now = time.time()

        cnnout = ForwardConv(testdatasetx, 2, weightcnn1, flatten=True)
        # relucnnout = relu(cnnout)
        cnntopred = np.dot(cnnout, weightfcn)  # + bfcn
        predtest = softmax(cnntopred)


        losstest = cceforward(predtest, testdatasety, w1=0, reg=losstype, lambdaa=lambdaa)



        row_maxes = predtest.max(axis=1).reshape(-1, 1)
        predtest[:] = np.where(predtest == row_maxes, 1, 0)


        resulttest = np.all(predtest == testdatasety, axis=1)
        testingepochaccuracy += np.sum(1 * resulttest, axis=0)

        print(i+1, "th epoch:")
        print(i+1,"th epoch took: ", now - beforeepoch, " seconds")
        now = 0
        beforeepoch = 0

        print("Training Epoch Accuracy/Testing Epoch Accuracy:", (trainingepochaccuracy /
howmanyimagesfortraining),"-",testingepochaccuracy / 12000)
        print("Training Loss/Testing Loss:", losssum/repet,"-",losstest)

        testinglosstable = np.append(testinglosstable, losstest)
        traininglosstable = np.append(traininglosstable,loss)
        trainingepochtable = np.append(trainingepochtable,trainingepochaccuracy/howmanyimagesfortraining)
        testingepochtable = np.append(testingepochtable,testingepochaccuracy/12000)
        loss = 0
        losstest = 0
        losssum = 0

    np.save("cnnweightl1",weightcnn1)
    np.save("cnnfcnweightl1",weightfcn)
    return traininglosstable, testinglosstable, trainingepochtable, testingepochtable

    print("Epoch : ",epoch)



learningrate=1e-3
traininglosstable, testinglosstable, trainingepochtable, testingepochtable  =
train(losstype="CCE",lr=learningrate, epoch=10,  batchsize=32, weightdecay = 1,weightdecayperiod= 10,
lambdaa=1e-6,testdatasetx=mnistx[48000:].reshape(-1,28,28),testdatasety = mnisty[48000:])
```

```
import matplotlib.pyplot as plt


print(traininglosstable)
plt.figure(figsize=(16,8))
titleforgraph = "Neural Network : CCE, lr = %0.02f, epoch = 10, "%learningrate
plt.title(titleforgraph)
plt.subplot(2,2,1)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.annotate('%0.2f' % traininglosstable[-1], xy=(1, traininglosstable[-1]), xytext=(2, 0),
        xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=traininglosstable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(traininglosstable)),traininglosstable)




plt.subplot(2,2,2)
plt.title("Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.annotate('%0.2f' % testinglosstable[-1], xy=(1, testinglosstable[-1]), xytext=(2, 0),
        xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=testinglosstable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(testinglosstable)),testinglosstable)

plt.subplot(2,2,3)
plt.title("Training Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.annotate('%0.2f' % trainingepochtable[-1], xy=(1, trainingepochtable[-1]), xytext=(2, 0),
        xycoords=('axes fraction', 'data'), textcoords='offset points')
plt.axhline(y=trainingepochtable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(trainingepochtable)),trainingepochtable)

plt.subplot(2,2,4)
plt.title("Validation Accuracy") #Validation
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.annotate('%0.2f' % testingepochtable[-1], xy=(1, testingepochtable[-1]), xytext=(2, 0), xycoords=('axes
fraction', 'data'), textcoords='offset points')
plt.axhline(y=testingepochtable[-1], color='y', linestyle='-.')
plt.plot(np.arange(len(testingepochtable)),testingepochtable)
plt.savefig("cnnNOLOSSlr1e-5")
plt.show()
```

Visualize Weights for Multinomial Logistic Regression:

```
import mnist
import numpy as np
import matplotlib.pyplot as plt
from forward2 import *
import matplotlib
wl1 = np.load("multilogccel1.npy")


wl2 = np.load("multilogccel2.npy")
w = np.load("multilog.npy")


w = w.reshape(28,28,10)
wl2 = wl2.reshape(28,28,10)
wl1 = wl1.reshape(28,28,10)




plt.figure(figsize=(16,8))

for i in range(10):
    print(i)

    titl = str("W NoReg, %d"%i)

    plt.subplot(1, 10,i+1)
    plt.title(titl)
    plt.imshow(w[:, :, i], cmap=matplotlib.cm.binary, interpolation="nearest")
plt.show()
plt.figure(figsize=(16,8))

for i in range(10):
    print(i)
    plt.subplot(1,10,i+1)
    titl = str("W with l1, %d"%i)
    plt.title(titl)
    plt.imshow(wl1[:, :, i], cmap=matplotlib.cm.binary, interpolation="nearest")
plt.show()
plt.figure(figsize=(16,8))

for i in range(10):
    print(i)
    plt.subplot(1,10,i+1)
    titl = str("W with l2, %d"%i)
    plt.title(titl)
    plt.imshow(wl2[:, :, i], cmap=matplotlib.cm.binary, interpolation="nearest")
plt.show()
```

```
"""

for i in range(10):
    plt.imshow(w[:,:,i],cmap = matplotlib.cm.binary,interpolation = "nearest") #matplotlib.cm.binary
    plt.show()
“""
```

Visualize Weights for Neural Network:

```
import mnist
import numpy as np
import matplotlib.pyplot as plt
from forward2 import *
import matplotlib
#w1l2 = np.load("weight1fornnl2.npy")
w2l2 = np.load("weight2fornnl2.npy")

#w1 = np.load("weight1fornn.npy")
w2 = np.load("weight2fornn.npy")

#w1l1 = np.load("weight1fornnl1.npy")
w2l1 = np.load("weight2fornnl1.npy")

import mnist
import numpy as np
import matplotlib.pyplot as plt



def relu(x):
    x.copy()
    out = x*(x > 0)
    return out


w2 = (w2.reshape(-1,28,28))
w2l2 = (w2l2.reshape(-1,28,28))
w2l1 = (w2l1.reshape(-1,28,28))


plt.figure(figsize=(16,8))

for i in range(9):
    print(i)

    titl = str("W2 NoReg, %d"%i)

    plt.subplot(1, 9,i+1)
    plt.title(titl)
    plt.imshow(w2[i],cmap="Greys")
plt.show()
plt.figure(figsize=(16,8))
for i in range(9):
    print(i)
    plt.subplot(1,9,i+1)
    titl = str("W2 with l2, %d"%i)
    plt.title(titl)
    plt.imshow(w2l2[i],cmap="Greys")
plt.show()
plt.figure(figsize=(16,8))


for i in range(9):
    print(i)
    plt.subplot(1,9,i+1)
    titl = str("W2 with l1, %d"%i)
    plt.title(titl)
    plt.imshow(w2l1[i],cmap="Greys")
plt.show()










# finding mean img
"""

# Data Prep---
x_train, y_train, x_test, y_test = mnist.load()
x_train = x_train.reshape(-1, 28 * 28)

numbofimages = 60000
mnistx = x_train / 255

mnisty = y_train.reshape(1, numbofimages)
"""

"""
b = np.zeros((numbofimages, 10))  # 10 is number of classes
b[np.arange(numbofimages), mnisty] = 1
mnisty = b
"""
"""

a,b= np.where(mnisty==2)

summ = np.array([])
for elements in b[0:200]:

    summ = np.append(summ,mnistx[elements])


summ = summ.reshape(-1,28,28)
print(summ.shape)
```

```
summean = summ.mean(axis=0)

plt.subplot(1,2,1)
plt.imshow(summean)
plt.subplot(1,2,2)

w2 = w2.reshape(-1,28,28)

w2[w2>0.5] =1

w2[w2<0.5] =0
plt.imshow(w2[2])
plt.show()
"""
```