

I SRN declare that I have completed this assignment completely and entirely on my own, without any consultation with others.  I understand that any breach of the UAB Academic Honor Code may result in severe penalties.

Srushti R Nayak

SP2024 CS Deep Learning

Hw3: 2-layer NN

**2-layer neural network for Image classification**

1. **Background and Method Introduction:**

**2-layer neural network is a supervised machine learning algorithm commonly used in classification tasks. It goes beyond the simplicity of a linear classifier by introducing non-linearity through an additional layer known as the hidden layer. This hidden layer enables the neural network to capture more complex relationships in the input data.**

**It can be used in image classification, a two-layer neural network takes features extracted from images as input and predicts predefined labels. The first layer, known as the input layer, receives the features, first layer’s output goes to the second layer, the second layer produces the final predictions. Both input and hidden layer accepts weights. Loss is calculated based on predicted output and actual output. During backward pass, gradients are calculated and weights are updated accordingly to minimize the loss function.**

**The non-linear activation function is typically applied to the hidden layer so model can learn more complex patterns in the data. Common activation functions include sigmoid, hyperbolic tangent (tanh), or rectified linear unit (ReLU).**

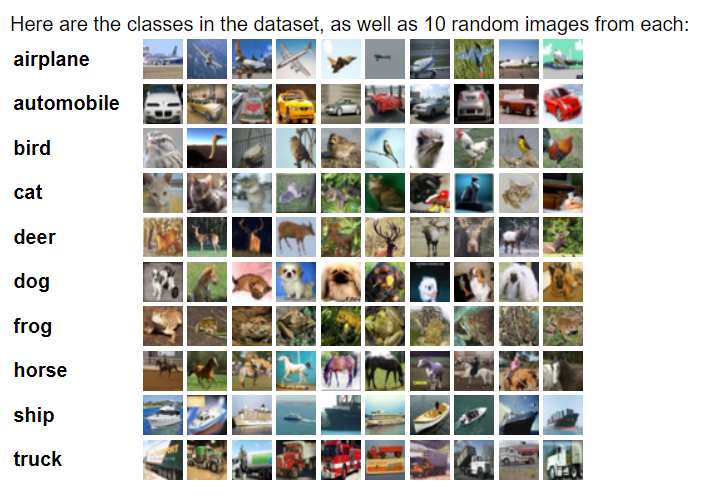
**Regularization techniques, such as L1 or L2 regularization, can be applied to the weights to prevent overfitting and improve the generalized performance of the neural network.**

1. **Dataset and Tasks Description**:

**2.1. CIFAR10 dataset:**

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.



**Fig1: Image classes in CIFAR10 database**

**Image from:** [**https://www.cs.toronto.edu/~kriz/cifar.html**](https://www.cs.toronto.edu/~kriz/cifar.html)

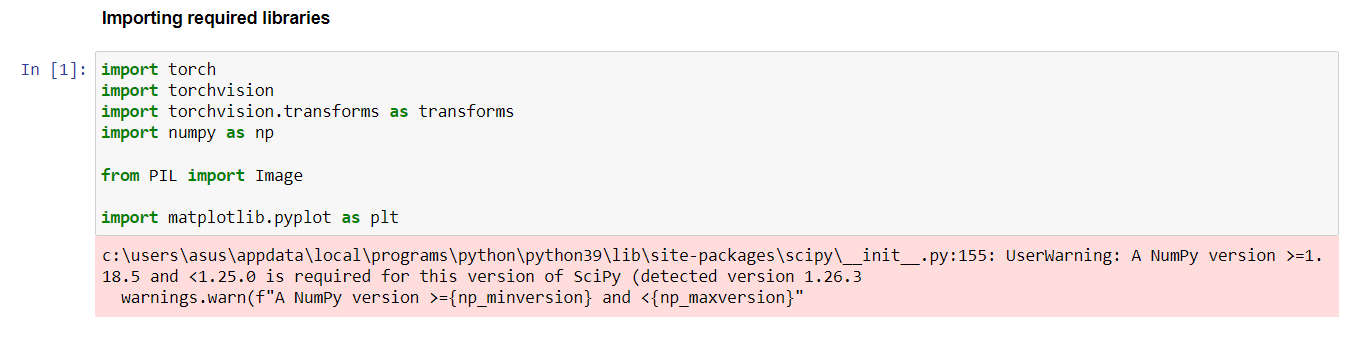
**2.2. Tasks Description:**

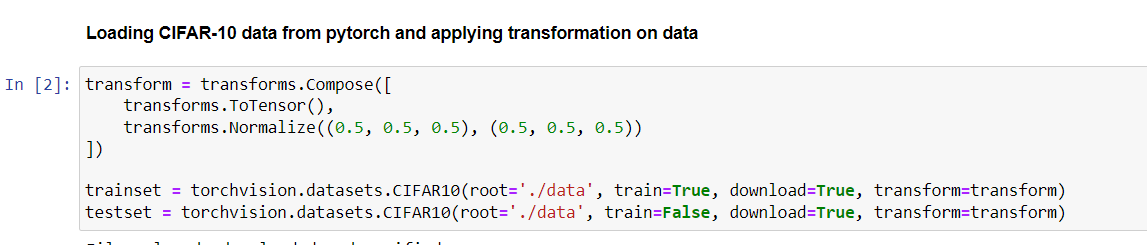
**This report details the construction and application of a 2-layer neural network on the CIFAR-10 dataset. Additionally, it examines the impact of incorporating L2 regularization to mitigate overfitting. Furthermore, the report provides insights into enhancing the training process of the neural network model through effective hyperparameter tuning.**

1. **Algorithms Used**:

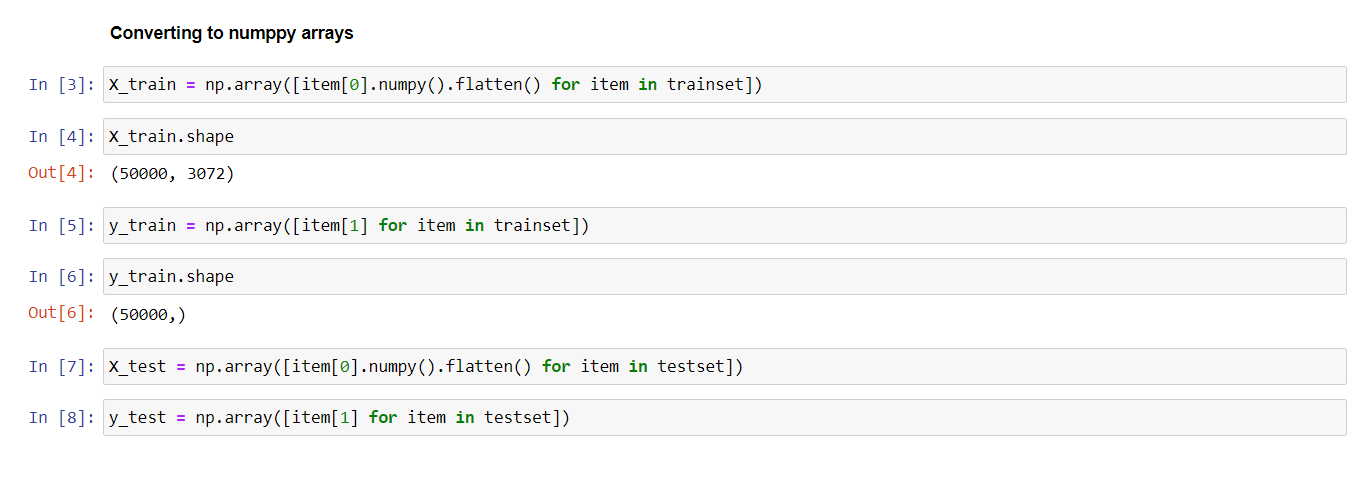
**3.0 Loading dataset and preparing data:**

- torch library is used to load the dataset.





* Converting tensors to flat numpy array which is suitable form for performing various operation supported by numpy



**3.1 Activation functions without regularization**

**3.1.1 2-layer neural network with sigmoid activation function without regularization:**

Sigmoid function f(x) =

If f(x) is sigmoid function, then it’s derivative will be

f’(x) = f(x) (1 – f(x))

So, sigmoid activation function will be used for forward computation.

And it’s derivative will be used in backward computation.

For forward computation, we can simply provide random weights w1 and w2 to input layer and hidden layer respectively.

Hidden layer input will be w1x

Hidden layer output will be sig(hidden layer input)

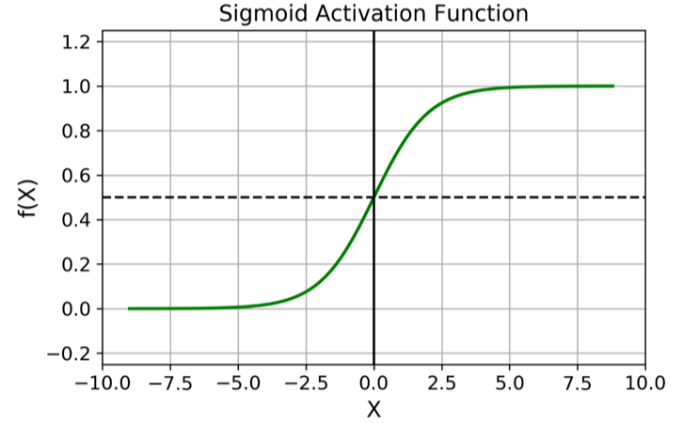
Weights w2 will be added to this output

After we get value in output layer we are calculating cross entropy loss and propagating backward the function’s derivative.

The values we get at the end of backpropagating that is gradients, we minus them from actual weights and bias(if included) to get updated weights and bias. This process continues for defined number of epochs.

After that the model will be trained and we will have bias and weights set properly.

We can first use them on test data to evaluate model performance.



**Figure 1: sigmoid Activation Function**

**https://insideaiml.com/blog/Sigmoid-Activation-Function-1031**

|  |
| --- |
| **# sigmoid withought regularization**  **class NNSig\_without\_reg:**    **def \_\_init\_\_(self, x\_train,y\_train):**  **self.x\_train = x\_train**  **self.y\_train = y\_train**  **self.batch\_size = 128**    **def setHyperParameters(self, input\_size, hidden\_size, output\_size, learning\_rate, epochs):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.w1 = np.random.randn(input\_size, hidden\_size)**  **self.w2 = np.random.randn(hidden\_size, output\_size)**    **def forward(self, x\_batch):**  **hidden\_layer\_input = x\_batch.dot(self.w1)**  **hidden\_layer\_output = 1 / (1 + np.exp(-hidden\_layer\_input))**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2)**  **scores = np.exp(output\_layer\_input - np.max(output\_layer\_input, axis=1, keepdims=True))**  **return hidden\_layer\_input, hidden\_layer\_output, scores**  **def loss(self,y\_batch,scores):**  **correct\_scores = scores[np.arange(len(scores)), y\_batch]**  **loss = -np.sum(np.log(correct\_scores / np.sum(scores, axis=1)))**  **return loss**    **def backward(self, hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch,x\_batch):**  **grad\_scores = scores / np.sum(scores, axis=1, keepdims=True)**  **grad\_scores[np.arange(len(grad\_scores)), y\_batch] -= 1**  **grad\_scores /= len(grad\_scores)**    **grad\_w2 = hidden\_layer\_output.T.dot(grad\_scores)**  **grad\_hidden = grad\_scores.dot(self.w2.T)**    **grad\_hidden\_layer\_input = grad\_hidden \* (hidden\_layer\_output \* (1 - hidden\_layer\_output))**  **grad\_w1 = x\_batch.T.dot(grad\_hidden\_layer\_input)**  **updated\_w1= self.w1 - self.learning\_rate \* grad\_w1**  **updated\_w2 = self.w2 - self.learning\_rate \* grad\_w2**    **return updated\_w1,updated\_w2**      **def train(self):**  **for epoch in range(self.epochs):**  **for i in range(0, len(self.x\_train), self.batch\_size):**  **x\_batch = self.x\_train[i:i+self.batch\_size]**  **y\_batch = self.y\_train[i:i+self.batch\_size]**    **hidden\_layer\_input, hidden\_layer\_output, scores = self.forward(x\_batch)**  **loss = self.loss(y\_batch, scores)**    **updated\_w1,updated\_w2 = self.backward(hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch,x\_batch)**  **self.w1 = updated\_w1**  **self.w2 = updated\_w2**    **print(f"Epoch {epoch+1}/{self.epochs}, Loss: {loss}")**  **def test(self,x\_test, y\_test):**  **hidden\_layer\_input = x\_test.dot(self.w1)**  **hidden\_layer\_output = 1 / (1 + np.exp(-hidden\_layer\_input))**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2)**  **predicted\_labels = np.argmax(output\_layer\_input, axis=1)**    **accuracy = np.mean(predicted\_labels == y\_test)**  **print(f"Accuracy on test set: {accuracy}")** |

**3.1.2 2-layer neural network with tanh activation function without regularization:**

* The algorithm is almost similar to previous one. But bias are added and tanh is used as activation function
* f(x) = tanh(x) =
* f’(x) = 1- f(x)2

similarly here also we use f(x) during forward pass and it’s derivative for backward pass

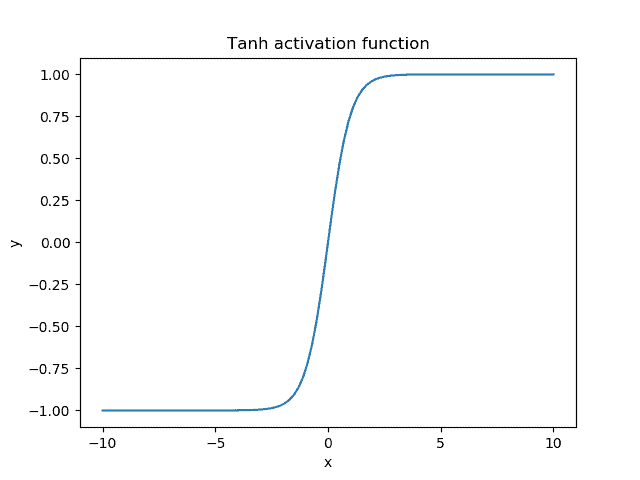


Figure 2

https://www.baeldung.com/cs/sigmoid-vs-tanh-functions

|  |
| --- |
| **#tanh without regularization**  **class NNtanh\_without\_reg:**    **def \_\_init\_\_(self, x\_train,y\_train):**  **self.x\_train = x\_train**  **self.y\_train = y\_train**  **self.batch\_size = 128**    **def setHyperParameters(self, input\_size, hidden\_size, output\_size, learning\_rate, epochs):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.w1 = np.random.randn(input\_size, hidden\_size)**  **self.w2 = np.random.randn(hidden\_size, output\_size)**  **self.bias1 = np.zeros((1, hidden\_size))**  **self.bias2 = np.zeros((1, output\_size))**    **def forward(self, x\_batch):**  **hidden\_layer\_input = x\_batch.dot(self.w1) + self.bias1**  **hidden\_layer\_output = np.tanh(hidden\_layer\_input)**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2) +self.bias2**  **scores = np.exp(output\_layer\_input - np.max(output\_layer\_input, axis=1, keepdims=True))**  **return hidden\_layer\_input, hidden\_layer\_output, scores**  **def loss(self,y\_batch,scores):**  **correct\_scores = scores[np.arange(len(scores)), y\_batch]**  **loss = -np.sum(np.log(correct\_scores / np.sum(scores, axis=1)))**  **return loss**    **def backward(self, hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch,x\_batch):**  **grad\_scores = scores / np.sum(scores, axis=1, keepdims=True)**  **grad\_scores[np.arange(len(grad\_scores)), y\_batch] -= 1**  **grad\_scores /= len(grad\_scores)**    **grad\_bias2 = np.sum(grad\_scores, axis=0, keepdims=True)**  **grad\_w2 = hidden\_layer\_output.T.dot(grad\_scores)**  **grad\_hidden = grad\_scores.dot(self.w2.T)**    **grad\_hidden\_layer\_input = grad\_hidden \* (1 - np.tanh(hidden\_layer\_output)\*\*2)**  **grad\_w1 = x\_batch.T.dot(grad\_hidden\_layer\_input)**  **grad\_bias1 = np.sum(grad\_hidden\_layer\_input, axis=0, keepdims=True)**  **updated\_w1= self.w1 - self.learning\_rate \* grad\_w1**  **updated\_w2 = self.w2 - self.learning\_rate \* grad\_w2**  **updated\_bias1 = self.bias1 - self.learning\_rate \* grad\_bias1**  **updated\_bias2 = self.bias2 - self.learning\_rate \* grad\_bias2**  **return updated\_w1,updated\_w2,updated\_bias1,updated\_bias2**    **def train(self):**  **for epoch in range(self.epochs):**  **for i in range(0, len(self.x\_train), self.batch\_size):**  **x\_batch = self.x\_train[i:i+self.batch\_size]**  **y\_batch = self.y\_train[i:i+self.batch\_size]**    **hidden\_layer\_input, hidden\_layer\_output, scores = self.forward(x\_batch)**  **loss = self.loss(y\_batch, scores)**    **updated\_w1,updated\_w2,updated\_bias1,updated\_bias2 = self.backward(hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch,x\_batch)**  **self.w1 = updated\_w1**  **self.w2 = updated\_w2**  **self.bias1 = updated\_bias1**  **self.bias2 = updated\_bias2**    **print(f"Epoch {epoch+1}/{self.epochs}, Loss: {loss}")**  **def test(self,x\_test, y\_test):**  **hidden\_layer\_input = x\_test.dot(self.w1) + self.bias1**  **hidden\_layer\_output = 1 / (1 + np.exp(-hidden\_layer\_input))**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2) + self.bias2**  **predicted\_labels = np.argmax(output\_layer\_input, axis=1)**    **accuracy = np.mean(predicted\_labels == y\_test)**  **print(f"Accuracy on test set: {accuracy}")** |

**3.1.3 2-layer neural network with ReLU activation function without regularization:**

* ReLU stands for rectified linear unit
* It is simply 0 for x< 0 and x for x>=0
* f(x) = 0, x<0

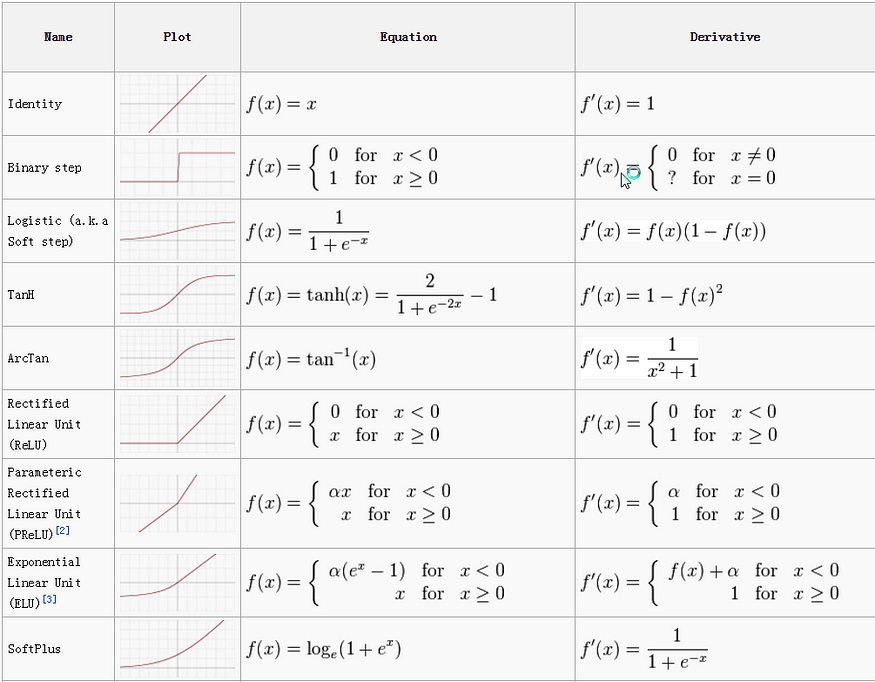
x, x>=0

* f’(x) = 0, x<0

1, x>=0

|  |
| --- |
| **class NNReLU\_without\_reg:**    **def \_\_init\_\_(self, x\_train, y\_train):**  **self.x\_train = x\_train**  **self.y\_train = y\_train**  **self.batch\_size = 128**    **def setHyperParameters(self, input\_size, hidden\_size, output\_size, learning\_rate, epochs):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.w1 = np.random.randn(input\_size, hidden\_size)**  **self.w2 = np.random.randn(hidden\_size, output\_size)**  **self.bias1 = np.zeros((1, hidden\_size))**  **self.bias2 = np.zeros((1, output\_size))**  **def forward(self, x\_batch):**  **hidden\_layer\_input = x\_batch.dot(self.w1) + self.bias1**  **hidden\_layer\_output = np.maximum(0, hidden\_layer\_input)**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2) + self.bias2**  **scores = np.exp(output\_layer\_input - np.max(output\_layer\_input, axis=1, keepdims=True))**  **return hidden\_layer\_input, hidden\_layer\_output, scores**  **def loss(self, y\_batch, scores):**  **correct\_scores = scores[np.arange(len(scores)), y\_batch]**  **loss = -np.sum(np.log(correct\_scores / np.sum(scores, axis=1)))**  **return loss**  **def backward(self, hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch, x\_batch):**  **grad\_scores = scores / np.sum(scores, axis=1, keepdims=True)**  **grad\_scores[np.arange(len(grad\_scores)), y\_batch] -= 1**  **grad\_scores /= len(grad\_scores)**  **grad\_bias2 = np.sum(grad\_scores, axis=0, keepdims=True)**  **grad\_w2 = hidden\_layer\_output.T.dot(grad\_scores)**  **grad\_hidden = grad\_scores.dot(self.w2.T)**  **grad\_hidden\_layer\_input = grad\_hidden \* (hidden\_layer\_output > 0).astype(int)**  **grad\_w1 = x\_batch.T.dot(grad\_hidden\_layer\_input)**  **grad\_bias1 = np.sum(grad\_hidden\_layer\_input, axis=0, keepdims=True)**  **updated\_w1 = self.w1 - self.learning\_rate \* grad\_w1**  **updated\_w2 = self.w2 - self.learning\_rate \* grad\_w2**  **updated\_bias1 = self.bias1 - self.learning\_rate \* grad\_bias1**  **updated\_bias2 = self.bias2 - self.learning\_rate \* grad\_bias2**  **return updated\_w1, updated\_w2, updated\_bias1, updated\_bias2**  **def train(self):**  **for epoch in range(self.epochs):**  **for i in range(0, len(self.x\_train), self.batch\_size):**  **x\_batch = self.x\_train[i:i+self.batch\_size]**  **y\_batch = self.y\_train[i:i+self.batch\_size]**  **hidden\_layer\_input, hidden\_layer\_output, scores = self.forward(x\_batch)**  **loss = self.loss(y\_batch, scores)**  **updated\_w1, updated\_w2, updated\_bias1, updated\_bias2 = self.backward(hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch, x\_batch)**  **self.w1 = updated\_w1**  **self.w2 = updated\_w2**    **def test(self,x\_test, y\_test):**  **hidden\_layer\_input = x\_test.dot(self.w1) + self.bias1**  **hidden\_layer\_output = 1 / (1 + np.exp(-hidden\_layer\_input))**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2) + self.bias2**  **predicted\_labels = np.argmax(output\_layer\_input, axis=1)**    **accuracy = np.mean(predicted\_labels == y\_test)**  **print(f"Accuracy on test set: {accuracy}")** |

* **Three activation functions are used in this report. But there are many kind of activation functions**



**Figure 3 : various activation functions and their derivatives**

**https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6**

* **3.2 Regularization**

Mainly three types of regularizations are there: Lasso(L1), Ridge(L2) and Elastic net

L1 regularization also known as lasso regression

It is used to prevent overfitting of a model to train dataset. Overfitted model performs best on train dataset. However, when it comes to test dataset or any other new dataset it might result in high variance.

L1 regularization adds penalty to the loss function and prevent the model from overfitting which is proportional to the absolute values of the coefficients (in our case it is weights) of the model.

Now model tries to minimize total loss = cross entropy loss + L1 regularization.

L1 regularization = λ \* , where λ= L1 regularization strength

L2 regularization also known as ridge regression

It is also used to prevent overfitting of a model to train dataset.

L2 regularization adds penalty to the loss function and prevent the model from overfitting which is proportional to the square of the coefficients (in our case it is weights) of the model.

Now model tries to minimize total loss = cross entropy loss + L2 regularization.

L2 regularization = λ \* , where λ= L2 regularization strength

* **Difference between L1 and L2 regularization:**

Main difference between L1 and L2 regularization is that L2 can penalize more large coefficients but cannot make them totally zero. But L1 regularization can make irrelevant features exactly zero. This scenario we can visualize from figure2. Left side graph represents L1 regularization and right-side graph represents L2 regularization.

Because of which L1 regularization can be used for feature selection.

L1 regularization is useful in cases where there are too many features and only small set from it is useful. L2 regularization is useful for cases where all the features contribute but some features have larger coefficients, L2 regularization stop them from dominating the model.

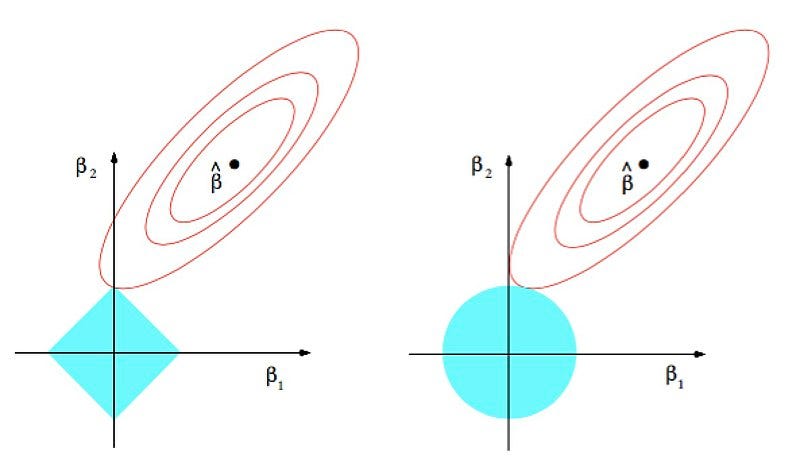


Figure2: visual difference between L1 and L2 regularization

Source: https://commons.wikimedia.org/wiki/File:Regularization.jpg

Elastic net regularization is combination of L1 and L2 regularization

Now total loss = cross entropy loss + L1 regularization + L2 regularization

Different strengths for L1 and L2 can be defined.

* **In this report L2 regularization is used**

**3.2.1: 2-layer neural network with sigmoid activation function with l2 regularization:**

* In regularization, regularization penalty Is added in calculation of loss.
* And the entire term with regularization penalty is tried to reduced.
* In this code just l2 regularization is added during calculating loss and while calculating gradients also l2 regularization is taken into consideration.

|  |
| --- |
| **class NNSigmoid\_with\_reg:**    **def \_\_init\_\_(self, x\_train, y\_train):**  **self.x\_train = x\_train**  **self.y\_train = y\_train**  **self.batch\_size = 128**    **def setHyperParameters(self, input\_size, hidden\_size, output\_size, learning\_rate, epochs, l2\_reg):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.w1 = np.random.randn(input\_size, hidden\_size)**  **self.w2 = np.random.randn(hidden\_size, output\_size)**  **self.bias1 = np.zeros((1, hidden\_size))**  **self.bias2 = np.zeros((1, output\_size))**  **self.l2\_reg = l2\_reg**  **def sigmoid(self, x):**  **return 1 / (1 + np.exp(-x))**  **def forward(self, x\_batch):**  **hidden\_layer\_input = x\_batch.dot(self.w1) + self.bias1**  **hidden\_layer\_output = self.sigmoid(hidden\_layer\_input)**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2) + self.bias2**  **scores = np.exp(output\_layer\_input - np.max(output\_layer\_input, axis=1, keepdims=True))**  **return hidden\_layer\_input, hidden\_layer\_output, scores**  **def loss(self, y\_batch, scores):**  **correct\_scores = scores[np.arange(len(scores)), y\_batch]**  **data\_loss = -np.sum(np.log(correct\_scores / np.sum(scores, axis=1)))**  **reg\_loss = 0.5 \* self.l2\_reg \* (np.sum(self.w1\*\*2) + np.sum(self.w2\*\*2))**  **loss = data\_loss + reg\_loss**  **return loss**  **def backward(self, hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch, x\_batch):**  **grad\_scores = scores / np.sum(scores, axis=1, keepdims=True)**  **grad\_scores[np.arange(len(grad\_scores)), y\_batch] -= 1**  **grad\_scores /= len(grad\_scores)**  **grad\_bias2 = np.sum(grad\_scores, axis=0, keepdims=True)**  **grad\_w2 = hidden\_layer\_output.T.dot(grad\_scores)**  **grad\_hidden = grad\_scores.dot(self.w2.T)**  **grad\_hidden\_layer\_input = grad\_hidden \* (hidden\_layer\_output \* (1 - hidden\_layer\_output))**  **grad\_w1 = x\_batch.T.dot(grad\_hidden\_layer\_input)**  **grad\_bias1 = np.sum(grad\_hidden\_layer\_input, axis=0, keepdims=True)**  **# Add L2 regularization terms to gradients**  **grad\_w1 += self.l2\_reg \* self.w1**  **grad\_w2 += self.l2\_reg \* self.w2**  **updated\_w1 = self.w1 - self.learning\_rate \* grad\_w1**  **updated\_w2 = self.w2 - self.learning\_rate \* grad\_w2**  **updated\_bias1 = self.bias1 - self.learning\_rate \* grad\_bias1**  **updated\_bias2 = self.bias2 - self.learning\_rate \* grad\_bias2**  **return updated\_w1, updated\_w2, updated\_bias1, updated\_bias2**  **def train(self):**  **for epoch in range(self.epochs):**  **for i in range(0, len(self.x\_train), self.batch\_size):**  **x\_batch = self.x\_train[i:i+self.batch\_size]**  **y\_batch = self.y\_train[i:i+self.batch\_size]**  **hidden\_layer\_input, hidden\_layer\_output, scores = self.forward(x\_batch)**  **loss = self.loss(y\_batch, scores)**  **updated\_w1, updated\_w2, updated\_bias1, updated\_bias2 = self.backward(hidden\_layer\_input, hidden\_layer\_output, scores, y\_batch, x\_batch)**  **self.w1 = updated\_w1**  **self.w2 = updated\_w2**  **self.bias1 = updated\_bias1**  **self.bias2 = updated\_bias2**  **print(f"Epoch {epoch+1}/{self.epochs}, Loss: {loss}")**  **def test(self, x\_test, y\_test):**  **hidden\_layer\_input = x\_test.dot(self.w1) + self.bias1**  **hidden\_layer\_output = self.sigmoid(hidden\_layer\_input)**  **output\_layer\_input = hidden\_layer\_output.dot(self.w2) + self.bias2**  **predicted\_labels = np.argmax(output\_layer\_input, axis=1)**  **accuracy = np.mean(predicted\_labels == y\_test)**  **print(f"Accuracy on test set: {accuracy}")** |

L2 regularization implementation is similar also in tanh and ReLU activation function.

Just need to add regularization term according to regularization strength while calculating loss and during backward pass, you'll need to compute the gradients of the regularization term with respect to the weights.

1. **Results**

Rest of the hyper parameters are kept same.

|  |  |  |
| --- | --- | --- |
| **Activation Function** | **Regularization** | **Accuracy** |
| Sigmoid | No | 28% |
| Tanh | No | 23.8% |
| ReLU | No | 23.37% |
| Sigmoid | Yes | 31.02% |
| Tanh | Yes | 37.7% |
| ReLu | Yes | 31.51% |

* It can be seen that accuracy is increased for all the activation function after adding regularization terms.
* Almost 10% increase is apparent after adding regularization.

1. **Methods of improvement**

To improve performance, try changing values for hyperparameters (hyperparameter tuning). Here epochs value increased, regularization is added, regularization strength is changed and learning\_rate is changed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Activation function** | **Epochs** | **Regularization** | **L2\_strength** | **Learning\_rate** | **Accuracy** |
| Sigmoid | 25 | No | 0.001 | 0.08 | 36.1% |
| Sigmoid | 25 | Yes | 0.001 | 0.08 | 37.72% |
| Tanh | 25 | No | 0.001 | 0.08 | 31.16% |
| Tanh | 25 | Yes | 0.001 | 0.08 | 39.16% |
| ReLU | 50 | Yes | 0.001 | 0.04 | 40.39% |
| Sigmoid | 50 | Yes | 0.008 | 0.04 | 41.98% |

* As we can see from above table, if epochs are increased, accuracy will also increase
* Also adding regularization increase model performance.
* Highest accuracy in above configurations is achieved when l2 strength is 0.008 and learning rate is 0.04 and number of epochs is 50.
* Adding regularization really boosts accuracy of tanh activation function from 31.16% to 39.16%.
* ReLU also achieves good accuracy with decreased learning rate and added regularization.
* Adding regularization can improve performance even after decreasing learning rate

1. **Conclusion:**

* Increasing epochs boosts accuracy even without regularization.
* Adding regularization also improves accuracy even with lower learning rate
* Regularization positively impacts accuracy by preventing overfitting.
* Tanh activation function is very sensitive to regularization
* Increasing learning rate too much might result in fail to find minima and overshooting can happen and decreasing learning rate to lower value can result in gradient to stuck in local minima.
* Optimal configuration: Sigmoid function with 50 epochs, added regularization, l2 strength as 0.008, learning rate = 0.04

1. **References:**

[**https://www.cs.toronto.edu/~kriz/cifar.html**](https://www.cs.toronto.edu/~kriz/cifar.html)

[**https://numpy.org/**](https://numpy.org/)

[**https://pytorch.org/**](https://pytorch.org/)

**Lecture slides**