

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

Hw2: Linear classifier

**Linear classifier for Image classification**

1. **Background and Method Introduction:**

**A linear classifier is a supervised machine learning algorithm used in classification. It is simple and efficient which make it a good choice for various applications including image classification. Linear classifier model involves finding a linear decision boundary in the feature space. This linear boundary is used to distinguish between different classes.**

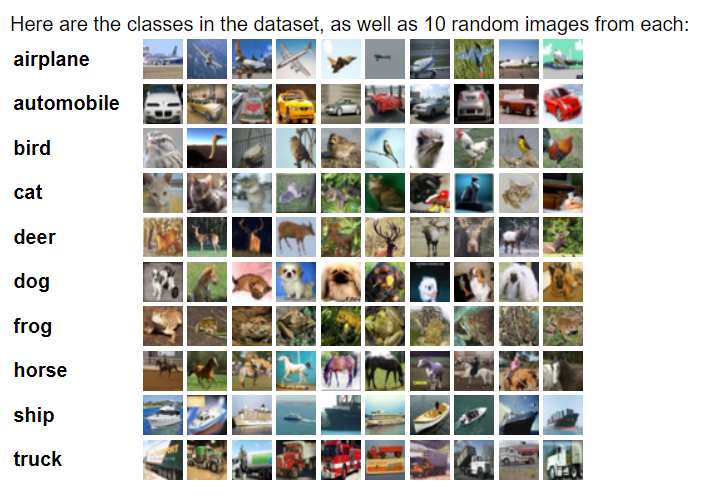
**While applying linear classifier to classify images, it takes features extracted from images as input and predicts predefined labels or classes to images. Concepts such as regularizations, gradient decent are used to improve performance of the classifier.**

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)

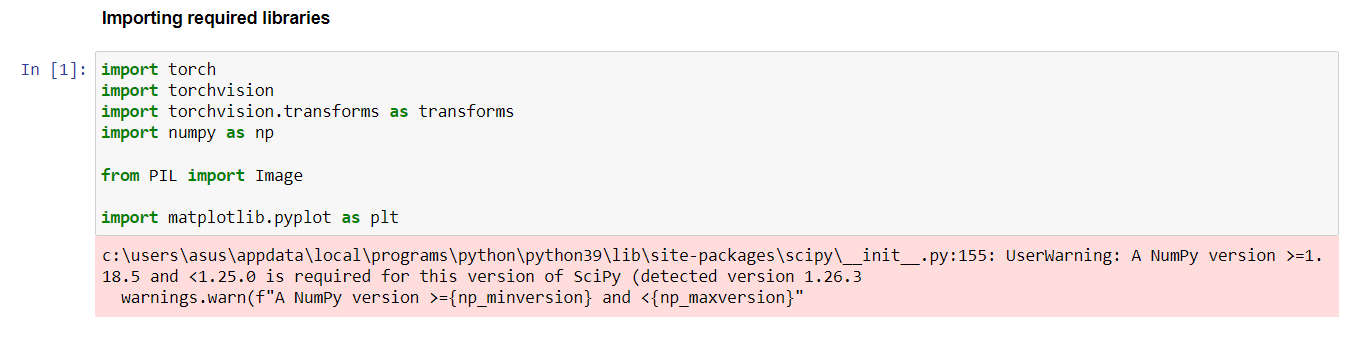
**2.2. Tasks Description:**

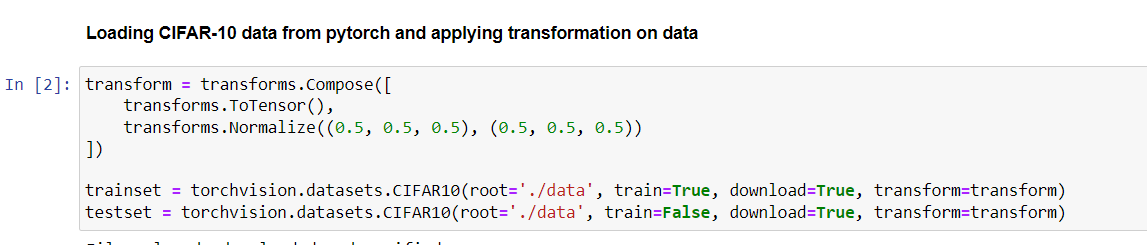
**This report details the construction and application of a Linear classifier on the CIFAR-10 dataset. Additionally, it examines the impact of incorporating regularization techniques such as L1 regularization, L2 regularization, and Elastic Net regularization to mitigate overfitting. Furthermore, the report provides insights into enhancing the training process of the linear classifier model through effective hyperparameter tuning.**

1. **Algorithms Used**:

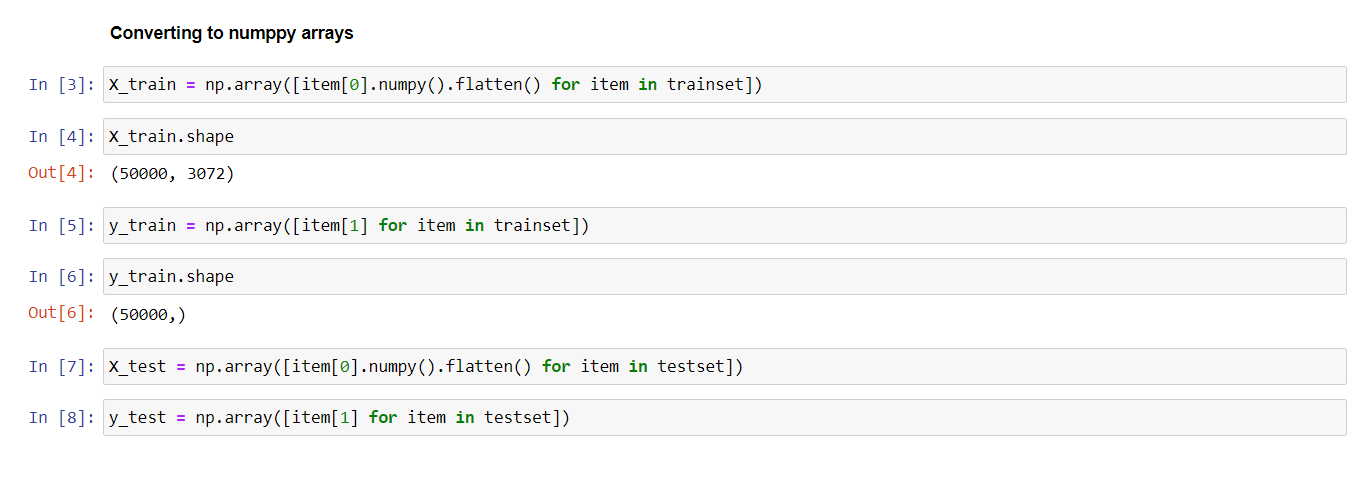
**3.0 Loading dataset and preparing data:**

- 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 Linear classifier without regularization:**

|  |
| --- |
| class Linear\_classifier:    def \_\_init\_\_(self, X\_train, y\_train):  self.X\_train = X\_train  self.y\_train = y\_train    def setHyperParameters(self, epochs, learning\_rate, total\_classes, input\_size):  self.epochs = epochs  self.learning\_rate = learning\_rate  self.input\_size = input\_size  self.weights = np.random.randn(input\_size, total\_classes) \* 0.0001  self.bias = np.zeros((1, total\_classes))    def forward\_pass(self, X, w, bias):  return np.dot(X, w) + bias    def softmax(self, z):  exp\_z = np.exp(z - np.max(z, axis=1, keepdims=True))  return exp\_z / np.sum(exp\_z, axis=1, keepdims=True)    def cross\_entropy\_loss(self, y\_pred, y\_train):  m = y\_train.shape[0]  log\_likelihood = -np.log(y\_pred[range(m), y\_train])  loss = np.sum(log\_likelihood) / m  return loss    def backward\_pass(self, X, y\_train, y\_pred):  m = y\_train.shape[0]  grad\_softmax = y\_pred  grad\_softmax[range(m), y\_train] -= 1  grad\_softmax /= m  grad\_weights = np.dot(X.T, grad\_softmax)  grad\_bias = np.sum(grad\_softmax, axis=0, keepdims=True)  return grad\_weights, grad\_bias  def update\_parameters(self, grad\_weights, grad\_bias):  self.weights = self.weights - learning\_rate \* grad\_weights  self.bias = self.bias - learning\_rate \* grad\_bias    def train(self):    for epoch in range(epochs):    output = self.forward\_pass(self.X\_train, self.weights,self.bias)  y\_pred = self.softmax(output)  cross\_entropy\_loss = self.cross\_entropy\_loss(y\_pred, self.y\_train)  grad\_weights, grad\_bias = self.backward\_pass(self.X\_train, self.y\_train, y\_pred)  self.update\_parameters(grad\_weights, grad\_bias)    if (epoch + 1) % 10 == 0:  print(f'Epoch {epoch + 1}, Loss: {cross\_entropy\_loss}')    def test(self, X\_test, y\_test):    output = self.forward\_pass(X\_test, self.weights, self.bias)  y\_pred = self.softmax(output)  predictions = np.argmax(y\_pred, axis=1)  accuracy = np.mean(predictions == y\_test) \* 100  print(accuracy) |

Forward pass:

f(X,w) = w\*X + bias, where X is the trains set of size 3072 x 1 (for CIFAR10 dataset)

w is the weight matrix of size 10 x 3072 (for CIFAR10 dataset)

bias is matrix of size 10 x 1

Backward pass:

These parameters weights and bias are generated randomly in starting phase and during backward pass their gradient decent is calculated such that it minimizes loss and these parameters are updated accordingly.

After specified number of epochs, weights and bias are set and model is ready to predict labels for new data. To evaluate the performance, model is used on test dataset.

**3.2 Linear classifier with L1 regularization:**

|  |
| --- |
| **class Linear\_classifier\_L1\_reg:**    **# initializing trainset**  **def \_\_init\_\_(self, X\_train, y\_train):**  **self.X\_train = X\_train**  **self.y\_train = y\_train**    **def setHyperParameters(self, epochs, learning\_rate, total\_classes, l1\_strength, input\_size):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.l1\_strength = l1\_strength**  **self.weights = np.random.randn(input\_size, total\_classes) \* 0.0001**  **self.bias = np.zeros((1, total\_classes))**    **def forward\_pass(self, X, w):**  **return np.dot(X, w) + self.bias**    **def softmax(self, z):**  **exp\_z = np.exp(z - np.max(z, axis=1, keepdims=True))**  **return exp\_z / np.sum(exp\_z, axis=1, keepdims=True)**    **def cross\_entropy\_loss(self, y\_pred, y\_train):**  **m = y\_train.shape[0]**  **log\_likelihood = -np.log(y\_pred[range(m), y\_train])**  **loss = np.sum(log\_likelihood) / m**  **return loss**    **def backward\_pass(self, X, y\_train, y\_pred):**  **m = y\_train.shape[0]**  **grad\_softmax = y\_pred**  **grad\_softmax[range(m), y\_train] -= 1**  **grad\_softmax /= m**  **grad\_weights = np.dot(X.T, grad\_softmax)**  **grad\_bias = np.sum(grad\_softmax, axis=0, keepdims=True)**  **return grad\_weights, grad\_bias**  **def update\_parameters(self, grad\_weights, grad\_bias):**  **self.weights = self.weights - (self.learning\_rate \* grad\_weights)**  **self.bias = self.bias - (self.learning\_rate \* grad\_bias)**    **def train(self):**    **for epoch in range(epochs):**    **output = self.forward\_pass(self.X\_train, self.weights)**  **y\_pred = self.softmax(output)**  **l1\_regularization = self.l1\_strength \* np.sum(np.abs(self.weights))**  **total\_loss = self.cross\_entropy\_loss(y\_pred, self.y\_train) + l1\_regularization**  **grad\_weights, grad\_bias = self.backward\_pass(self.X\_train, self.y\_train, y\_pred)**  **grad\_weights = grad\_weights + ((self.l1\_strength/self.X\_train.shape[0]) \* np.sign(self.weights))**  **self.update\_parameters(grad\_weights, grad\_bias)**    **if (epoch + 1) % 10 == 0:**  **print(f'Epoch {epoch + 1}, Loss: {total\_loss}')**    **def test(self, X\_test, y\_test):**    **output = self.forward\_pass(X\_test, self.weights)**  **y\_pred = self.softmax(output)**  **predictions = np.argmax(y\_pred, axis=1)**  **accuracy = np.mean(predictions == y\_test) \* 100**  **print(accuracy)** |

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

**3.3 Linear classifier with L2 regularization:**

|  |
| --- |
| **class Linear\_classifier\_L2\_reg:**    **# initializing trainset**  **def \_\_init\_\_(self, X\_train, y\_train):**  **self.X\_train = X\_train**  **self.y\_train = y\_train**    **def setHyperParameters(self, epochs, learning\_rate, total\_classes, l2\_strength, input\_size):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.l2\_strength = l2\_strength**  **self.weights = np.random.randn(input\_size, total\_classes) \* 0.0001**  **self.bias = np.zeros((1, total\_classes))**    **def forward\_pass(self, X, w):**  **return np.dot(X, w) + self.bias**    **def softmax(self, z):**  **exp\_z = np.exp(z - np.max(z, axis=1, keepdims=True))**  **return exp\_z / np.sum(exp\_z, axis=1, keepdims=True)**    **def cross\_entropy\_loss(self, y\_pred, y\_train):**  **m = y\_train.shape[0]**  **log\_likelihood = -np.log(y\_pred[range(m), y\_train])**  **loss = np.sum(log\_likelihood) / m**  **return loss**    **def backward\_pass(self, X, y\_train, y\_pred):**  **m = y\_train.shape[0]**  **grad\_softmax = y\_pred**  **grad\_softmax[range(m), y\_train] -= 1**  **grad\_softmax /= m**  **grad\_weights = np.dot(X.T, grad\_softmax)**  **grad\_bias = np.sum(grad\_softmax, axis=0, keepdims=True)**  **return grad\_weights, grad\_bias**  **def update\_parameters(self, grad\_weights, grad\_bias):**  **self.weights = self.weights - learning\_rate \* grad\_weights**  **self.bias = self.bias - learning\_rate \* grad\_bias**    **def train(self):**    **for epoch in range(epochs):**    **output = self.forward\_pass(self.X\_train, self.weights)**  **y\_pred = self.softmax(output)**  **l2\_regularization = 0.5 \* self.l2\_strength \* np.sum(self.weights \*\* 2)**  **total\_loss = self.cross\_entropy\_loss(y\_pred, self.y\_train) + l2\_regularization**  **grad\_weights, grad\_bias = self.backward\_pass(self.X\_train, self.y\_train, y\_pred)**  **grad\_weights = grad\_weights + ((self.l2\_strength/self.X\_train.shape[0]) \* self.weights)**  **self.update\_parameters(grad\_weights, grad\_bias)**    **if (epoch + 1) % 10 == 0:**  **print(f'Epoch {epoch + 1}, Loss: {total\_loss}')**    **def test(self, X\_test, y\_test):**    **output = self.forward\_pass(X\_test, self.weights)**  **y\_pred = self.softmax(output)**  **predictions = np.argmax(y\_pred, axis=1)**  **accuracy = np.mean(predictions == y\_test) \* 100**  **print(accuracy)** |

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.

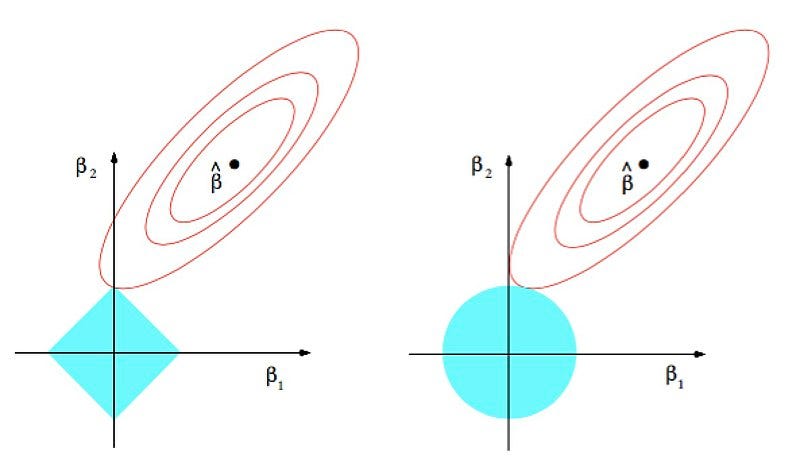


Figure2: visual difference between L1 and L2 regularization

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

**3.4: Elastic-net regularization:**

|  |
| --- |
| **class Linear\_classifier\_Elastic\_net\_reg:**    **# initializing trainset**  **def \_\_init\_\_(self, X\_train, y\_train):**  **self.X\_train = X\_train**  **self.y\_train = y\_train**    **def setHyperParameters(self, epochs, learning\_rate, total\_classes, l1\_strength, l2\_strength, input\_size):**  **self.epochs = epochs**  **self.learning\_rate = learning\_rate**  **self.input\_size = input\_size**  **self.l2\_strength = l2\_strength**  **self.l1\_strength = l1\_strength**  **self.weights = np.random.randn(input\_size, total\_classes) \* 0.0001**  **self.bias = np.zeros((1, total\_classes))**    **def forward\_pass(self, X, w):**  **return np.dot(X, w) + self.bias**    **def softmax(self, z):**  **exp\_z = np.exp(z - np.max(z, axis=1, keepdims=True))**  **return exp\_z / np.sum(exp\_z, axis=1, keepdims=True)**    **def cross\_entropy\_loss(self, y\_pred, y\_train):**  **m = y\_train.shape[0]**  **log\_likelihood = -np.log(y\_pred[range(m), y\_train])**  **loss = np.sum(log\_likelihood) / m**  **return loss**    **def backward\_pass(self, X, y\_train, y\_pred):**  **m = y\_train.shape[0]**  **grad\_softmax = y\_pred**  **grad\_softmax[range(m), y\_train] -= 1**  **grad\_softmax /= m**  **grad\_weights = np.dot(X.T, grad\_softmax)**  **grad\_bias = np.sum(grad\_softmax, axis=0, keepdims=True)**  **return grad\_weights, grad\_bias**  **def update\_parameters(self, grad\_weights, grad\_bias):**  **self.weights = self.weights - learning\_rate \* grad\_weights**  **self.bias = self.bias - learning\_rate \* grad\_bias**    **def train(self):**    **for epoch in range(epochs):**    **output = self.forward\_pass(self.X\_train, self.weights)**  **y\_pred = self.softmax(output)**  **l1\_regularization = self.l1\_strength \* np.sum(np.abs(self.weights))**  **l2\_regularization = 0.5 \* self.l2\_strength \* np.sum(self.weights \*\* 2)**  **total\_loss = self.cross\_entropy\_loss(y\_pred, self.y\_train) + l2\_regularization + l1\_regularization**  **grad\_weights, grad\_bias = self.backward\_pass(self.X\_train, self.y\_train, y\_pred)**  **grad\_weights = grad\_weights + (self.l2\_strength \* self.weights) + ((self.l1\_strength/self.X\_train.shape[0]) \* np.sign(self.weights))**  **self.update\_parameters(grad\_weights, grad\_bias)**    **if (epoch + 1) % 10 == 0:**  **print(f'Epoch {epoch + 1}, Loss: {total\_loss}')**    **def test(self, X\_test, y\_test):**    **output = self.forward\_pass(X\_test, self.weights)**  **y\_pred = self.softmax(output)**  **predictions = np.argmax(y\_pred, axis=1)**  **accuracy = np.mean(predictions == y\_test) \* 100**  **print(accuracy)** |

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.

1. **Results**

**Epochs: 100, learning\_rate = 0.01**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **L1\_strength** | **L2\_strength** | **Accuracy** |
| Linear\_classifier | NA | NA | **36.59%** |
| Linear\_classifier\_L1\_reg | 0.001 | NA | **36.63%** |
| Linear\_classifier\_L2\_reg | NA | 0.001 | **36.65%** |
| Linear\_classifier\_Elastic\_net\_reg | 0.001 | 0.001 | **36.68%** |

1. **Methods of improvement**

To improve performance, try changing values for hyperparameters (hyperparameter tuning). Here learning rate and regularization strength are gradually increased. Input size will be 32\*32\*3. As for in CIFAR10 dataset, all images are with 32\*32 pixels and with 3 channnels.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Epochs** | **Learning\_rate** | **L1\_strength** | **L2\_strength** | **Accuracy** |
| Linear\_classifier\_L2\_reg | 100 | 0.05 | NA | 0.001 | **39.6%** |
| Linear\_classifier\_L2\_reg | 200 | 0.05 | NA | 0.001 | **40.6%** |
| Linear\_classifier\_L2\_reg | 200 | 0.05 | NA | 0.008 | **40.61%** |
| Linear\_classifier\_L2\_reg | 200 | 0.08 | NA | 0.01 | **40.92%** |
| Linear\_classifier\_L2\_reg | 200 | 0.1 | NA | 0.05 | **39.51%** |
| Linear\_classifier\_L2\_reg | 200 | 0.08 | NA | 0.09 | **40.93%** |
| Linear\_classifier | 200 | 0.08 | NA | NA | **40.94%** |
| Linear\_classifier\_L1\_reg | 200 | 0.08 | 0.09 | NA | **40.89%** |

* As we can see from above table, increasing number of epochs results in increases accuracy.
* In terms of learning\_rate, as we gradually increase learning\_rate, accuracy increases. But after certain value increasing learning\_rate results in decreases accuracy. It is happening because due to high learning rate we are crossing the minima while calculating and updating gradient decent for weights and bias. That is the reason why learning\_rate 0.1 has less accuracy of 39.51% then of 40.92% at learning\_rate 0.08.
* Also it is noticeable that adding regularization can improve accuracy from average around 36.60% to 40%.

1. **Conclusion:**

* Increasing epochs boosts accuracy even without regularization.
* A moderate learning rate aids convergence, while very high rates may lead to overshooting.
* Regularization, especially L2, positively impacts accuracy by preventing overfitting.
* Balancing regularization strength is crucial, but even without regularization, good accuracy can be achieved.
* Optimal configuration: Linear classifier without regularization, trained for 200 epochs with a learning rate of 0.08, achieves competitive accuracy.

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/)

[**https://www.tensorflow.org/api\_docs/python/tf/keras/datasets/cifar10/load\_data**](https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar10/load_data)

**Lecture slides**