**Background and Method Introduction:**

**Linear Classifier:**

Linear classifiers are a fundamental machine learning model used for classification tasks. They essentially draw a decision boundary in a high-dimensional feature space, separating different classes. In the context of image classification, each image is represented as a vector of features (e.g., pixel values), and the classifier aims to predict the class label (e.g., cat, dog) based on this feature vector.

**How it works:**

Input: An image represented as a feature vector (e.g., flattened pixels).

Decision boundary: A linear equation learned from the training data.

Prediction: The class label assigned based on which side of the boundary the feature vector falls on.

**Applications in image classification:**

* Simple and interpretable: Linear models are easy to understand and interpret, making them useful for tasks where understanding the decision-making process is important.
* Efficient for smaller datasets: They are computationally efficient, making them suitable for scenarios with limited data or resources.
* Good starting point: They often serve as a baseline model for more complex deep learning architectures.

**Limitations:**

Linear decision boundary: They struggle to capture complex relationships between features, especially for non-linear data.

Limited performance: Their accuracy may not be as high as deep learning models, especially for large and diverse datasets.

Overall, linear classifiers offer a simple yet effective approach for image classification, especially for smaller datasets and interpretability needs. However, their limitations in handling complex non-linearity should be considered when choosing a model for specific tasks.

**Dataset and Tasks Description:**

The CIFAR-10 dataset is a widely used benchmark dataset in the field of computer vision. It consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. The dataset is divided into 50,000 training images and 10,000 test images.

The specific classification task undertaken with the CIFAR-10 dataset is image classification. Given an input image, the task is to classify it into one of the following 10 classes {Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck}.

Each image in the dataset belongs to one of these classes, and the goal is to train a model that can accurately predict the correct class label for unseen images.

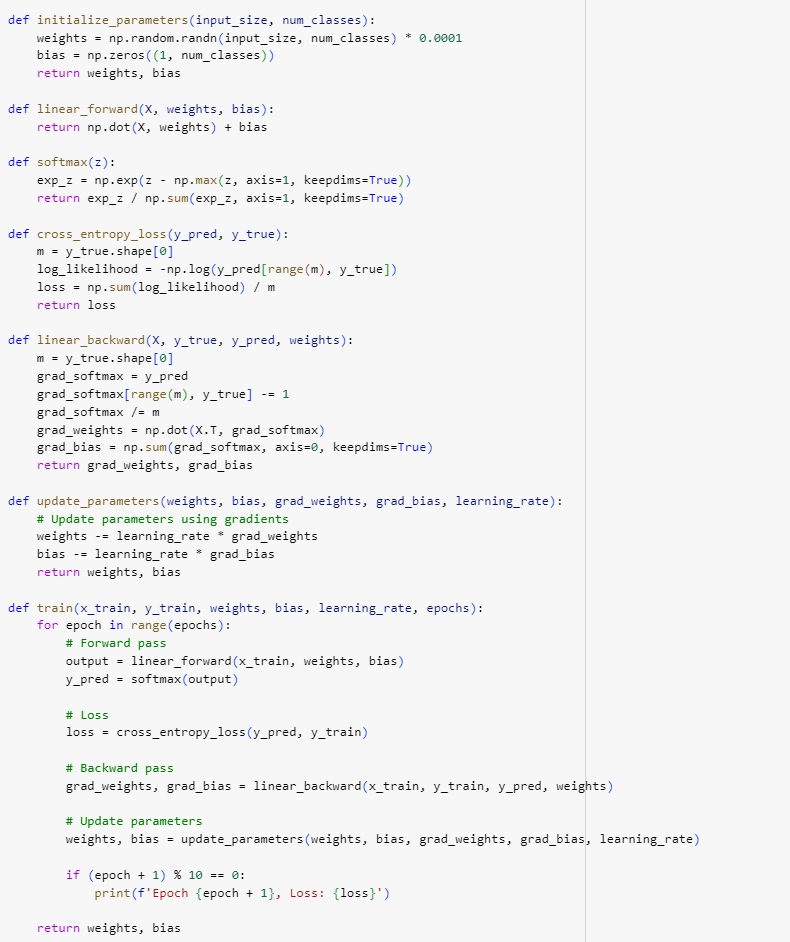
In the code, the CIFAR-10 dataset is loaded using torchvision and transformed to tensors. It is then divided into training and test sets. The images are normalised to have pixel values between -1 and 1.

The classification task involves training a linear classifier using the training data and evaluating its performance on the test data. The model is trained using the cross-entropy loss function and softmax activation for multi-class classification. First classification without regularisation is performed, then the algorithm is modified to include regularisation, then further modifications are done to obtain better performance. The optimization algorithm used is stochastic gradient descent (SGD).

The model's performance is evaluated based on its accuracy, which represents the percentage of correctly classified images in the test set. The best hyperparameters (learning rate, number of epochs, regularisation strength, and batch size) are determined through an exhaustive search over predefined ranges.

**Algorithms Used:**

1. **Linear Classifier:**

****

*Linear Classifier code snippet*

**Parameter Initialization (initialize\_parameters):**

* The initialize\_parameters function initialises the weights and biases of the linear classifier.
* The weights are initialised with small random values drawn from a normal distribution, scaled by a small factor (0.0001).
* Biases are initialised to zeros.
* The shape of the weight matrix depends on the input size and the number of classes.

**Forward Pass (linear\_forward):**

* The linear\_forward function computes the forward pass of the linear classifier.
* It performs a matrix multiplication of the input data (X) with the weights (weights) and adds the bias (bias) term.
* This operation produces raw scores for each class, which are not yet normalised.

**Softmax Activation (softmax):**

* The softmax function takes the raw scores (logits) computed during the forward pass and applies the softmax activation function.
* Softmax converts the raw scores into a probability distribution over the classes, ensuring that the probabilities sum up to 1

**Loss Calculation (cross\_entropy\_loss):**

* The cross\_entropy\_loss function computes the cross-entropy loss between the predicted probabilities (y\_pred) and the true labels (y\_true).
* It penalises incorrect predictions more heavily, leading to faster convergence during training.

**Backward Pass (linear\_backward):**

* The linear\_backward function calculates the gradients of the loss with respect to the weights and biases.
* The gradients are used to update the parameters during training.

**Parameter Updates (update\_parameters):**

* The update\_parameters function updates the weights and biases using gradient descent.
* It adjusts the parameters in the direction that minimises the loss, scaled by the learning rate.

**Training Loop (train):**

* The train function iterates over the dataset for a fixed number of epochs.
* In each epoch, it performs a forward pass, computes the loss, performs a backward pass to calculate gradients, and updates the parameters.
* The loss is printed periodically to monitor the training progress.
* The time taken for training is measured using the time.time() function before and after training.
* This helps to monitor the training efficiency and compare different implementations or hyperparameters

**Testing (test):**

* The test function evaluates the trained model on the test dataset.
* It computes the accuracy of the model by comparing its predictions with the true labels.

1. **Linear Classifier with L2 regularisation:** Few additional codes are added to the previous code to implement L2 regularisation, the details about the code are as below:

****

*Linear Classifier with L2 regularisation*

**Regularization Parameter (lambda\_reg):**

The lambda\_reg parameter is introduced to control the strength of regularisation in the model. It determines how much the regularisation term influences the overall loss calculation during training.

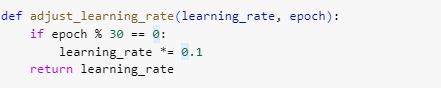
**Cross-Entropy Loss with L2 Regularization:**

* The cross\_entropy\_loss function is modified to incorporate L2 regularisation.
* In addition to calculating the cross-entropy loss, it now adds a regularisation term to the loss using the formula: 0.5 \* lambda\_reg \* sum(weights \*\* 2).
* This term penalises large weights and helps prevent overfitting by adding a constraint to the optimization process.

**Training Function Modification:**

* The train function is updated to accept the lambda\_reg parameter.
* During each iteration of training, the lambda\_reg parameter is passed to the cross\_entropy\_loss function to compute the loss with regularisation.
* Before starting training, the lambda\_reg parameter is set.

1. **Linear Classifier with hyperparameter tuning for better performance:** Few more lines of codes are added, which basically tunes the parameters and picks the parameters that give best test accuracy. The implementation details of the improvements are as below:



*Dynamic Learning rate calculation (w.r.t epoch)*



*Hyperparameter tuning of different parameter to obtain best model*

**Mini-Batch Training:**

* The train function now incorporates mini-batch training. Instead of updating weights after processing the entire dataset, it iterates through smaller batches of the dataset.
* This helps in making efficient use of memory and computation resources, especially for large datasets.
* The batch\_size parameter is introduced to control the number of samples processed in each mini-batch.
* It allows experimentation with different batch sizes to optimise training performance and convergence.

**Training Loop Modification:**

* The training loop now iterates over the number of batches (x\_train.shape[0] // batch\_size) instead of the number of epochs directly.
* Within each epoch, the training data is shuffled, and mini-batches are created and processed iteratively.

**Adjusting Learning Rate within Epochs:**

* The adjust\_learning\_rate function now adjusts the learning rate within each epoch instead of only at the start of training.
* This dynamic adjustment can help improve training stability and convergence.

**Hyperparameter Tuning:**

* The code now includes a loop for hyperparameter tuning, where different combinations of learning rates, epochs, regularisation parameters, and batch sizes are tested.
* For each combination, the model is trained and evaluated on the test dataset.
* The best combination of hyperparameters is recorded based on the highest accuracy achieved on the test set.

**Results:**

|  | Test Accuracy (%) | Time Taken |
| --- | --- | --- |
| Linear Classifier | 36.64 | 191.73 seconds |
| Linear Classifier with L2 | 36.64 | 183.572 seconds |
| Linear Classifier after tuning | 41.72 | 249.58 seconds |

Above table describes the test accuracy and time taken for training of the classifier. We can observe that a normal linear classifier took 191.73 seconds to obtain 36.63% accuracy, whereas a classifier with L2 regularisation obtained the same accuracy in a little less time i.e182.572 seconds. And after we have tuned the classifier for better performance, we were able to achieve an accuracy of 41.72% (at Learning rate of 0.01, 100 epochs, 0.01-regularisation strength and 32 batch size).

**Methods of Improvement:**

* Linear classifiers alone were a little slow and prone to overfitting when we train for longer epochs, so implementing **L2 regularisation** reduces overfitting and also here in this case it obtained the same accuracy in less time as well.
* We have implemented **mini-batch** training, by processing smaller batches at a time, the model can fit into the memory more easily, and computations can be distributed across available computing resources, such as CPUs or GPUs, leading to faster training times.
* Initially, during the early stages of training, a higher learning rate may help the model progress quickly towards a good solution. However, as training progresses, the same higher learning rate may overshoot the weights. Adjusting the learning rate can help prevent large fluctuations in the loss function during training. By gradually decreasing the learning rate, the optimization process can settle into a more stable trajectory, facilitating smoother training. So, I varied the **learning rate** with the number of epochs **dynamically**.
* As there are many hyperparameters involved like number of epochs, batch size, regularisation strength, it is difficult to use a single set to obtain a better model. So, some set of values were chosen and all possible combinations of the sets were tried and the model which gives best test accuracy was chosen (and the hyper parameters to obtain that model are recorded, this is **hyperparameter tuning**)