CMPUT 466 Miniproject

Name: Castor Shem   CCID: nshem   Student ID: 1666282   Section: A1

Introduction

Classification is among the most well-studied fields of computer vision. In this report I investigate several simple machine learning models for classifying small images from the CIFAR 10 dataset.

Neural networks originate from Rosenblatt's Perceptron in the 1950s, but did not reach their present popularity until the 2010s when more powerful computers were able to train networks for practical purposes [1]. Convolutional neural networks (CNNs) are now the standard machine learning model used for image classification tasks and first appeared in 1998 with Yang Le's LeNet. However, in this report I am using non-CNN algorithms from scikit-learn to experiment with a less intensive models for classifying small images: linear and logistic regression, the K-nearest-neighbours and Naive Bayes algorithms (chosen due to being very different from neural networks), and a multi-layer perceptron.

The CIFAR 10 dataset [2] is a relatively small image dataset (at least compared to other standard image datasets like ImageNet), with small 32x32 RGB-coloured images from 10 classes. It was collected in 2009 by Alex Krizhevsky, and in the years since has become a very popular machine learning dataset used to train and benchmark computer vision models.

The code to reproduce the results in this report is in a public Github repository.

Problem Formulation

I chose to work with the CIFAR 10 dataset [2] for this project and attempt to train a model to classify images into one of its 10 classes. I felt this would be a useful way for me to test different classification models we had learned about in the course.

The input to each model I tested was data from CIFAR 10 (preparation is described below), consisting of an M x 1024 data array (representing M samples with 1024 features each) and a M x 1 array of true labels (0 through 9, corresponding to one of the 10 classes). The output was a M x 1 array of predicted labels.

Dataset Preparation

CIFAR 10 provides 5 train batches of 10000 images, and 1 test batch of 10000 images. Across all 50000 train images there are 5000 images of each of the 10 classes in the dataset, and in the test batch there are 1000 images of each of the 10 classes.

To create the train and validation sets I loaded the provided 5 train batches, shuffled all 50000 images and split them into a train set of 40000 images and validation set of 10000 images.

I chose to perform some preprocessing to make the images easier to work with. The images in CIFAR 10 are 32x32 RGB images, which are represented in the data as $32 * 32 * 3 = 3072$ element arrays with elements that take on pixel values in the range $[0, 255]$. I averaged the three channels to convert the images to grayscale, creating $32 * 32 = 1024$ element arrays (I did this because I wanted to make the data smaller and faster to train on). I also normalized all the pixel values such that they took on the range $[-1, 1]$; this is a technique commonly used in computer vision tasks.

All the code to handle the dataset loading and preparation is in the `data.py` file in my submission.

Approaches and Baselines

Baselines

Chance accuracy on this problem (that is, the expected value of the accuracy if one were to choose one of the 10 classes for each image completely at random) is $1/10 = 0.1$.

Tuning

For each model I chose to use in this project, I defined a grid of hyperparameters and trained the model with each combination of hyperparameters from the grid. I obtained the accuracy of each trained model on the validation set. I then chose the best model (the one with the greatest validation accuracy) and tested it on the test set to obtain the 'final' accuracy I consider for that model.

Models

I tried several different machine learning algorithms from `scikit-learn` for this project, and tuned several hyperparameters for most of them.

Linear Regression: Training a linear regression algorithm with L2 regularization to act as a 10-class classifier. Hyperparameters I considered were "fit_intercept" (whether or not to include a bias), and the regularization strength.

Logistic Regression: Training a logistic regression algorithm with L2 regularization to act as a 10-class classifier. Hyperparameters I considered were "fit_intercept" (whether or not to include a bias), and the regularization strength (the same hyperparameters as those I considered for linear regression).

K-nearest-neighbours: An algorithm that classifies points by grouping them together (in "neighbourhoods") based on the principle that points closer together are more likely to be of the same class [4]. I tuned 'k' (the number of nearest neighbours to consider) and the 'weights' hyperparameter given by scikit-learn (either weighting all neighbours equally, or by the inverse of their distance).

Naive Bayes: Predicting classes from images using the Naive Bayes algorithm. I did not tune any hyperparameters for this method.

Multi-Layer Perceptron: A multi-layer perceptron trained for classification, with 2 hidden layers that have 10 neurons each. There are a lot of possible hyperparameters to tune for this model; I chose to try two different activation functions (logistic and ReLU), two different batch sizes, and several learning rates. I was limited in the number of hyperparameters I could try because of how long the model took to train on my laptop.

Model	Hyperparameters Considered
Linear Regression	bias, regularization strength
Logistic Regression	bias, regularization strength
K-Nearest-Neighbours	number of neighbours, weighting
Naive Bayes	
Multi-Layer Perceptron	activation, batch size, learning rate

Evaluation Metric

The metric of success for the models I used was accuracy: the ratio of samples whose classes were predicted correctly to total samples. This metric is implemented in a very simple function defined in the `utils.py` file. This is the primary goal of the problem (accurately classifying as many samples as possible), and I'm satisfied with using it as the only evaluation metric. However, in other implementations it might also be informative to calculate precision, recall, or other metrics.

Results

The results of all my experiments can also be found in the CSV files generated by the code in `main.py`, and in the Appendix.

All Methods

The best hyperparameters I found for each model and their validation and test accuracies are:

Model	Hyperparameters	Val. Acc.	Test Acc.
Linear Regression	Bias included, regularization strength = 0.3	0.3963	0.3916
Logistic Regression	Bias included, regularization strength = 7.0	0.4076	0.3985
K-Nearest-Neighbours	k=5, weighting distance	0.3796	0.3728
Naive Bayes		0.2781	0.292
Multi-Layer Perceptron	ReLU activation function, batch size = 64, lr = 0.0001	0.5178	0.5123

Test accuracy in general was above the chance accuracy baseline (0.1), showing that the models are useful for this task.

Linear regression and logistic regression yielded very similar results, but the best result was obtained with dramatically different regularization strengths. K-nearest-neighbours performed similarly to the two regression algorithms, but slightly worse. The Naive Bayes method performed dramatically worse than the others (but still above the baseline). The multi-layer perceptron performed better than all other models; it has many more trainable parameters and a more complex structure (with two layers), and it is to be expected that it will have better performance.

Individual Methods

Linear Regression

alpha	fit_intercept	val_acc	test_acc
0.3	False	0.3907	
0.3	True	0.3963	0.3916
0.7	False	0.3908	
0.7	True	0.3962	
1.0	False	0.3908	
1.0	True	0.3957	
3.0	False	0.3921	
3.0	True	0.3958	
7.0	False	0.3918	
7.0	True	0.3956	

For linear regression, including a bias (fit_intercept = True) always improved accuracy; this is what I would expect, since it adds additional parameters. Increasing the regularization strength (alpha) very slightly decreased validation accuracy, and the best validation accuracy was obtained with the lowest value of regularization strength.

Logistic Regression

alpha	fit_intercept	val_acc	test_acc
0.3	False	0.3933	
0.3	True	0.3999	
0.7	False	0.3942	
0.7	True	0.4008	
1.0	False	0.395	
1.0	True	0.4007	
3.0	False	0.3966	
3.0	True	0.404	
7.0	False	0.4002	
7.0	True	0.4076	0.3985

The results for logistic regression are similar to linear regression. Including a bias (fit_intercept = True) always improved accuracy, as expected. Increasing the regularization strength (alpha) *increased* validation accuracy, and the best validation accuracy was obtained with the lowest value of regularization strength.

K Nearest Neighbours

k	weights	val_acc	test_acc
1	uniform	0.3671	
1	distance	0.3671	
3	uniform	0.3524	
3	distance	0.3739	
5	uniform	0.3572	
5	distance	0.3796	0.3728
7	uniform	0.3587	
7	distance	0.3772	

For K-nearest-neighbours typically the 'distance' weighting scheme worked better, and 5 neighbours (scikit-learn's default value) had the best overall validation performance.

Naive Bayes

val_acc	test_acc
0.2781	0.292

With no hyperparameters to tune, the train/validation/test split isn't terribly meaningful for this model, but is used for consistency. The single Naive Bayes model I used performed poorly compared to the other algorithms.

Multi-Layer Perceptron

activation	batch_size	lr	val_acc	test_acc
logistic	16	0.0001	0.4846	
logistic	16	0.001	0.4312	
logistic	16	0.01	0.3458	
logistic	64	0.0001	0.4948	
logistic	64	0.001	0.4455	
logistic	64	0.01	0.4271	
relu	16	0.0001	0.4995	
relu	16	0.001	0.4803	
relu	16	0.01	0.2757	
relu	64	0.0001	0.5178	0.5123
relu	64	0.001	0.4774	
relu	64	0.01	0.4027	

One clear trend in this table is that higher learning rate decreases performance; likely the model 'jumps around' and has a hard time nearing a minimum. The lowest learning rate performed best for all combinations of activation function and batch size. I tried batch sizes of 16 and 64, and it seems that using the larger batch size improves performance for most combinations of activation function and learning rate (with the exception of activation = relu, lr = 0.001, where batch size 16 performs better than 64).

The effect of the activation function varies. It seems like the combinations with the highest learning rate (lr = 0.01) perform better with the logistic activation function, but all other combinations perform better with the ReLU function.

The best performance overall came with the ReLU activation function, a batch size of 64, and a learning rate of 0.0001.

References

[1] N. Kushwaha, "A Brief History of the Evolution of Image Classification," Medium, Aug. 30, 2023. https://python.plainenglish.io/a-brief-history-of-the-evolution-of-image-classification-402c63baf50 (accessed Dec. 09, 2023).

[2] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," April 8, 2009. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf (accessed Dec. 07, 2023).

[3] "User Guide," Scikit-Learn. https://scikit-learn.org/stable/user_guide.html (accessed Dec. 07, 2023).

[4] "What is the k-nearest neighbors algorithm?" IBM. https://www.ibm.com/topics/knn (accessed Dec. 09, 2023).

[5] D. Nelson, "Overview of Classification Methods in Python with Scikit-Learn," Stack Abuse, Nov. 16th, 2023. https://stackabuse.com/overview-of-classification-methods-in-python-with-scikit-learn/ (accessed Dec. 09, 2023)