K-NEAREST NEIGHBORS

**Algorithm analysis:** The algorithm takes the labels of the k nearest samples and predicts the label from the majority of them. As an implementation it is quite simple but the big problem is that the algorithm "assumes" that all samples of the same type are in close proximity to each other (as if creating clusters). This means its accuracy depends directly on this property of the dataset we choose and because it is obviously not something that is mainly true, usually its accuracy is quite low and unsatisfactory.

**POSITIVES:**

* Easy implementation
* Sufficient accuracy for dataset that has clustered samples

**NEGATIVE:**

* Large runtime and complexity especially for large samples and/or large dimensions
* Generally bad accuracy

**Parameters: (sklearn)**

* **k:** The number of neighbors who "vote" to get the sample in their class. The bigger the k, the more the noise effect decreases but the boundaries of the classification become inconspicuous.
* **algo:** "auto" chooses the best algorithm among 3 options based on the dimensions and size of the dataset. These 3 algorithms are KD-Tree, Ball Tree and Brute Force where each is most efficient depending on the previous 2 parameters.
* **weights:** the default of the algorithm is the uniform i.e. each neighbor has the same "value" of the vote as the others. If we put it to be distance then the value of the vote of each neighbor will be determined by their distance to the sample. First of all, it is easy to see that we cannot apply it for k =1 (it's the same) and that we might expect better results, of course reinforcing the algorithm's big deal for clustering class samples.
* **metric:** we choose from a range of options the metric we want to use to measure distances. In this task, 3 metrics were used for comparison, euclidean , minkowski and manhattan

NEAREST CENTROID

**Algorithm**

Each class is represented by its centroid and all test samples are classified based on the class of the nearest centroid. Centroid is the mean of each class. Like K-Nearest Neighbors the algorithm is very easily implemented but there is the problem of assuming that the elements of the same class of the dataset they are in close proximity so that it does not generally have good accuracy. On the other hand, of course, due to the centroid, complexity and runtime are dramatically reduced.

**POSITIVES:**

* Easy implementation
* Sufficient accuracy for dataset that has clustered samples

**NEGATIVE:**

* It relies a lot on data clustering
* Generally bad accuracy

**Parameter:**

* **metric:** Corresponding to KNN
* **shrink\_threshold:** We choose whether we want to "shorten" the centroids to get elements that cause noise and make classification difficult

CIFAR-10

The dataset used for this work is cifar-10. It consists of 60000, 32pixel x 32pixel images in 10 classes. These classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship , truck. The dataset was chosen firstly because it meets the requirements of the job and secondly because it includes a sufficient degree of difficulty to become classified, resulting in theoretically seeing more changes in accuracy by testing different techniques and parameters. The dataset data is easily fetched from the module of tensorflow.

IMPLEMENTATION

The program was implemented in **Python** language. 4 main packages were used:

1. "keras": to load the dataset CIFAR-10 with ease
2. "sklearn": to get the algorithms K-Nearest neighbors and Nearest Centroid
3. "numpy": for operations with arrays
4. "cv2": to turn the sample into grayscale

Results



The implementations of the 3 algorithms with different parameters have presented important discoveries. First of all, looking at the table we can see that among the 3 algorithms the best accuracy has been achieved by the 1-Nearest neighbor with some specific parameters. 2nd in the predictions we see to be the 3-Nearest neighbor so we conclude that although with more neighbors theoretically you should have a better picture of where your sample belongs because more neighbors vote for the right label, in our sample we conclude that our data is more evenly distributed resulting in the 1 neighbor who is the closest to be more effective, and we can "confirm" this from the fact that in cases where we get weighted distances the predictions are better.

At the same time, we see that the choice of metric does not present a substantial difference in prediction but gives us a little better runtime.

Nearest Centroid at an initial glance seems to have the worst odds, but we notice that in the end its deviations with the other algorithms are in the range of 4-5%, in an already bad prediction to emphasize, and we achieve minimal runtime. From where we were running for an original sample of 3-NN for 57 minutes, we run the same sample momentarily with a deviation of 5%.

Also configuring the Nearest Centroid with the shrink threshold we see that whether we get a small price or a big one does not help us at all in accuracy

In the implementations it was used due to the large runtime in KNN mainly the dataset with grayscale as it reduced it significantly without a large loss of accuracy (Order of 2%)

Normalization was also tested in the sample but the results were the same for all implementations in both runtime and accuracy

Finally, we note that finally none of the 3 models with its different parameters can model the dataset correctly as the hypothesis of the algorithms we have mentioned above does not apply to our sample.

Neural Networks

We are trying for a start to model our dataset using neural networks. The simplest way to do this is by using perceptrons that are fully connected to a dense network.

The experiments started from very simple networks for a start with a single layer with few neurons (64) that could not adapt well to the data (~30%) and gradually increased neurons and hidden layers. Of course, various learning parameters such as different loss functions (MSE, Crossentropy), optimizers (Adam, SGD, with momentum and without), different activation functions, etc. were also tested throughout the duration. One thing worth noting is that the learning rate remained for most of the experiment stable at 10-3 because in previous tests where it was changing even a little, no network did train or some brought about very bad accuracy.

Some conclusions that can come out of these experiments are that Adam and Categorical Crossentropy seem to have much better results than SGD and MSE. Also, according to [research](https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/) , the kernel initializer was tested: he instead of glorot (the default) in layers with activation functions ReLU and there were slightly better results. A change in the initialization of bias to 1 instead of 0 was also tested with slightly worse results than 0 as initialization.

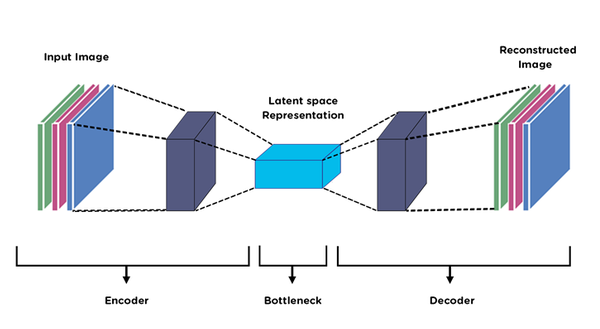
The problem that arises is that with dense neural networks there is no good modeling of the problem as the best accuracy achieved is 52% in a network with 4 layers 1000 (ReLU, glorot)-300 (ReLU, glorot) -50 (ReLU, glorot) -10 (softmax), adam and crossentropy. This accuracy is not sufficient to say that we solved the problem of categorizing the dataset. We realize that by "blindly" searching for a multitude of layers and neurons and teasing the hyperparameters we are not going to be led to our goal. Studying how we can push the dense multilayer perceptrons to the next level, we are led to 4 basic ideas ([source](https://openreview.net/pdf/1WvovwjA7UMnPB1oinBL.pdf) for the first 2).

* **Introduction of linearity:** reduces the number of parameters (hence the runtime) and mainly has output that follows the Gauss distribution and helps to better pass the information to more neurons.
* Pretrain with **Autoencoders:** we manage to compress the information in a much smaller dimension and do feature extraction. Ideally our encoder will have "learned" the important characteristics for him and will be able to do classification more easily and quickly (due to compression the dimensions, and therefore the actions and the runtime, are dramatically reduced).
* **Data augmentation:** we modify slightly our sample (e.g. Rotation 90o) and we do retrain. So in a sense we have increased the amount of data in the train and make it easier to extract the feature extraction for our network and therefore learning.
* **Grayscaling**: As in KNN and NC we reduce dimensions, eliminating RGB, assuming that color is not as important to classification and making learning much more efficient.

Linearity

In the continuation of the experiment we use architecture 4000-1000-4000 with different activation functions and parameters and now this network is the basis for experimentation. We confirm the findings of the research and observe that indeed the network with intermediate linearity carries better results than the rest, not by much difference of course. After several experiments, the results are more or less the same as the ones we got from simple multilayer networks. We managed accuracy 54.1% with 4000(ReLU,he\_uniform)-1000(linear)-4000 (ReLU,he\_uniform)-10(softmax), Adam, Categorical Crossentropy after 179 seasons and overfit. What was also basically observed is that in almost all models the overfit actually improved the network's performance by a small degree.

Autoencoder:



The basic principle is to insert an image, compress it into fewer dimensions by holding the "important" features and decompress it back to the original one. In order to help the training in the classification, it is enough to take the information from the bottleneck layer and train with it. (Target autoencoder = the training sample itself, x\_train--> x\_train)

Various experiments were carried out to find a good autoencoder and a fully dense neural network and a fully convolutional neural network with deconvolution in the decoder as well as a mix of 2 was used. The results were unexpected, as with a fully dense network the autoencoder achieved a performance of ~73% and then with convolutional while we expected there to be improvement, the performance was the same or much worse in some cases.

The experiments that were done are similar to those in the dense neural network for classification. For a start, compression was tested in 128 dimensions, a network without extra layer (flattened->bottleneck->3072->(32,32,3)) and we did train with adam and mse and sigmoid in the bottleneck layer and got accuracy 75%.

A picture containing text

Description automatically generated

Then we went into compression at 512 dimensions in the same simple model and achieved an accuracy of 80.21%.

A picture containing calendar

Description automatically generated

(obvious difference between them, in the 2nd the images are quite clear, they lose details)

Tests with further layers and neurons, different activations or other parameters had worse results and larger runtimes(Typically the model (flattened->2048 ReLU -> 1024-> bottleneck Sigmoid -> 1024 ReLU -> 2048 ReLU -> 3072 sigmoid -> (32.32.3)) had accuracy 71% with 20s/epoch.)

Migrating to a convolution-deconvolution architecture we expected the results to be significantly better but in reality the results were a little worse than the dense autoencoder. Changes were made to the kernel size, convolution size, strides, activation functions, loss, layers, neurons, etc. a. without achieving a performance of more than 78%. The exact same results (with similar tests) we got for the convolution-dense autoencoder.

Finally there was an implementation of U-Net which is a convolutional network for image segmentation. The network did train and very long time (46 minutes for 100 seasons) and achieved accuracy of only 64%.

So we conclude that the best model is the simplest dense autoencoder that we can make with compression in 512 dimensions (80% autoencoding accuracy). So by encodeing our dataset and train with the best model format 4000ReLU-1000Lin-4000ReLU we do not receive a substantial increase in performance (an increase of 1-2%). By pretrain and with a convolution-dense model (which had 78% accuracy) we get worse results than the dense network that was not pretrained with an autoencoder. The responsibility for these results may lie in the relatively low accuracy we received from autoencoders, in their ability to correctly compress the information and perform the feature extraction.

Grayscaling-Data Augmentation

Grayscaling with minimal testing on already tested, dense network structures seems to have worse performance (43% accuracy) with much faster implementations of course.

In the same context, data augmentation showed worse results (~45% accuracy) with image rotation and a small fluctuation of their dimensions (0.1). It is noted that the training time was 35s/epoch. This shows that probably even with more data, our network would not be able to do good learning.

As we can see from the results, of course, the use of dense networks does not model our dataset to a good extent (~50%), no matter how much we "tweak" the parameters and apply tricks. This pushes us to look for another architecture to get better results. The disadvantage of using dense fully connected networks is that when we reshape the input and "stack stack" we remove" the vectors, then the information we get from neighboring pixels is lost. As is easily understood this information is very important (i.e. what value the pixel gets definitely depends to a certain extent (possibly large) on the values of its neighbors) and we force with the dense fully connected network to relearn it. Finally, we end up training a network which is very difficult to train and has too many parameters making it extremely time consuming.

Convolutional Neural Network

Diagram

Description automatically generated

So, in order to succeed in keeping the information of neighboring pixels and to reduce the number of parameters, and therefore the training time, we can implement a synodal architecture. In essence we will create a square kernel that scans the image, multiply these 2 element wise, add all the values and divide them by the length of the kernel. Thus, the network succeeds in doing much easier feature extraction and significantly reduces the runtime.

We begin the search for architecture with networks of the following form:

* Block : Convolutional Layer – Maxpooling
* Flatted layer
* Dense layer
* Classification layer

Optimizer is adam, in the classification layer we use softmax, in the remaining layers ReLU with he uniform for kernel initializer.

With only 1 block( filter =64, kernel size = 3 and pool size = (2,2)) and only with the classification layer we go directly to accuracy 64%. After experiments where we fluctuated both convolutional and dense layers and their parameters we saw no substantial improvement.

For a start we decide to go to architectures from very deep convolutional networks where each block consists of 2 convolutional layers (with the same filter size that doubles in each block) and 1 maxpooling layer.

We define as a base the model that consists of 3 such blocks and 1 dense layer. (filters from 32 to 128 , kernel 3x3, ReLU and he uniform everywhere except the last layer, padding = same, 128 neurons in the dense layer, optimizer adam, loss = categorical crossentropy)

With this model we have an accuracy of 76.54% which shows a very significant improvement compared to the previous ones. An important observation is that education seems to start directly from the "solution", may be because of adam, and does not improve from then on (1.5% improvement). The most important, however, is that the network overfits to a very large extent as only after 20 seasons it achieves 99% accuracy training.

Chart, line chart

Description automatically generated

Bearing in mind the overfitting of our network and the fact that in the training set it learns quickly the data and to a good extent, we can assume that it has the ability to learn the data correctly and we are pushed to explore techniques that slow down the convergence of our network. We can achieve this in various ways such as data augmentation (which we also did for dense networks), changes in batch size, regularization or dynamic changes in the learning rate.

For starters as a regularization method we test the introduction of dropout between layers. We start with dropout 0.2 between each block and see that the accuracy becomes 80%.

We are also introducing weight decay to further assist regularizing. We use the L2(0.001) method in each layer, without the dropout layers. We notice that there is an improvement of 1%, not significant, and perhaps we should strengthen the weight decay and make it 0.01 instead of 0.001.

Using both L2 and Dropout together didn't work better.

**Data augmentation.**  We apply the same technique we did to the Dense network. Accuracy rises to 83% in 30 seasons but training is very slow (37s/epoch).

We see that data augmentation and dropout are showing a noticeable improvement, so it makes sense to try to combine and optimise them.

Starting out we can make a variation of dropout regularization, try targeted patterns that may bring us better results such as increasing the dropout in each block (0.20->0.30->0.40->0.50). Accuracy does not change dramatically (78.98%). In combination with data augmentation, however, we end up with accuracy 84%, the highest so far.

Other techniques that could help modeling are batch normalization and pixel scaling.

Confusion Matrix of baseline model with Dropout:

A picture containing chart

Description automatically generated

We notice that the network has difficulty recognizing the 3 classes of bird, cat, deer. In particular, we see that it often confuses a dog with a cat, a deer with a horse and a bird with an airplane. These pairs of animals obviously have common characteristics and can bottleneck the network. For example, the plane with the birds have a similar shape, a feature that may have been extracted from the network and through neighboring pixels as well as the sky as a background in all 2 categories. We could for better separation perhaps create a classifier that does a better job in distinguishing these categories from each other and make it implement on a wider network that makes classify all classes of the dataset.

Conclusions

Looking at the findings so far we can easily state that the convolutional architecture is the most suitable to model the dataset of cifar-10 as it achieves the highest accuracy with 84%. This of course is not the only reason we consider this architecture better as it also has fewer parameters, lower runtimes, it quickly converge and even a very simple model manages to adapt adequately to the data. Consequently, we observe that with good practices we can optimize even more their capabilities and theoretically reach an even better accuracy.

The algorithms k nearest neighbors and nearest centroid are obviously easier to implement but they are poor in performance and lag behind even very simple dense neural networks models.

As for dense neural networks, the performance that can be achieved for dataset modeling is limited. We have seen that as many changes to hyperparameters and clever techniques to help our network learn better and more easily have not had the desired results and did not reach the performance of convolutional networks. We obviously confirmed that ultimately the information lost from neighboring pixels it is useful for our network and to be "relearned" by our network is not something that is easily done with this architecture.

For more information in experiments you can refer to the other files.

The code exists in the form of a jupyter notebook and has some optimizations for tpu testing on google colab which is optional.