(MINI) CINIC-10 Image Classification with Softmax Regression and Neural Networks

Carlos Avelar
168605
c168605@dac.unicamp.br

Tiago Chaves
187690
t187690@dac.unicamp.br

***Abstract*—In this project we explore the problem of image classification on a reduced version of the CINIC-10 dataset[1] by utilizing only 100,000 out of its 270,000 images, but still considering 10 labeled classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck).**

We compared the results of multinomial logistic regression (i.e. softmax regression) with the use of a 1-hidden-layer neural network. In an attempt to improve our classification network we also trained it with the Momentum [1] and Adam [2] optimization rules, besides the classic Gradient Descent.

Our best model was a neural network trained with the Momentum optimizer, that achieved 40% accuracy and 0.3687 combined F1 score on the validation set. We then used it with the test set, resulting in a combined F1 score of 0.3548, showing resilience and applicability to never seen data.

I. Introduction

Image classification consists on learning a mapping function from given images to their correct class/label.

The dataset used (MINI CINIC-10[2]) consists of 100,000 32×32 RGB images of 10 classes, already split into 80% training, 10% validation and 10% test sets. Thus, our objective is to find a good model for an object recognition system to predict (output) the labels of input images.

The CINIC-10 dataset aims to fill a benchmarking gap between ImageNet, which can be too difficult, and CIFAR-10, which can be too easy and small (having only 60,000 images in total, less than our training set). By knowing this we can better judge our results later.

II. Problem

The current state-of-the-art models for image classification on the CINIC-10 dataset achieve over 90% accuracy[3], applying Deep Convolutional Neural Networks.

The aim of this report is to evaluate the use of a shallow — one hidden layer — Neural Network (NN) and of a Softmax Regressor (which can be modeled as a NN with no hidden layers), and particularly how they compare to each other.

III. Proposed solution

In our implementation we define a **Computational Graph** class, since both softmax regressors and neural networks can be modelled as a special form thereof.

[1]github.com/BayesWatch/cinic-10
[2]Available at: tinyurl.com/y6fmvllh
[3]State-of-the-art leaderboard: paperswithcode.com/task/image-classification

A Computational Graph [3], [4] is a way of modeling relations between variables and mathematical operations, by having each graph node represent either a value — which will later be a layer in a NN — or a computation (i.e. an operation like the activation or cost functions).

The implemented class is composed of a cost function, an optimizer, and a sequence of **Layers** — another class — that individually perform feedforward and backpropagation based on its activation function (we omit the activation function of the input layer as its activation values are the model's input, so we can set them directly). Note that the optimizers, activation functions and cost functions are also classes to make it easier to customize the Computational Graph (CG), as we believe this way we've created a nice user-friendly interface for it all.

Thereby, we implemented the Softmax Regressor as a 2-layer CG with softmax activation and cross-entropy cost function, and the Neural Network as a 3-layer CG, with softmax activation on the output layer, sigmoid on the hidden, and cross-entropy cost function for evaluation as well.

A. Sigmoid activation

The Sigmoid function squashes the values passed to it between 0 and 1, often being used in binary classification.

$$sigmoid(t) = \frac{1}{1+e^{-t}} \tag{1}$$

B. Softmax activation

The Softmax function is a generalization of the Sigmoid function that gets an array instead of one scalar value as input, and squashes the sum of all array values to 1.

$$softmax(\mathbf{t}_j) = \frac{e^{\mathbf{t}_j}}{\sum_{\mathbf{t}_i \in \mathbf{t}} e^{\mathbf{t}_i}} \tag{2}$$

C. Cross entropy cost

Cross entropy is used to represent the ”distance” between what the model believes the output distribution should be, and what the original distribution really is [5].

Being $\hat{f}(x)$ the estimated value of $f(x)$ (which will be predicted by the computational graphs):

$$Cross\,entropy(f, \hat{f}) = -\sum_x f(x)\log\left(\hat{f}(x)\right) \tag{3}$$

It is worth noting that when using cross entropy as the cost function J, its derivative with respect to the inputs $\mathbf{z}$ of the

Softmax activation in the output layer result in the simple equation:

$$\frac{\partial J}{\partial \mathbf{z}_i} = p_i - y_i \tag{4}$$

where y_i is the target value and p_i is the prediction made by the output layer.

IV. Data Preprocessing

The dataset has 100,000 images divided in 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck (with 80:10:10 split for train, validation and test).

The 32x32 RGB images were flattened into one dimensional arrays of size 3072, with values in the $[0, 255]$ range.

The three sets were Z-Score [6] normalized, having each value be subtracted by the mean and divided by the standard deviation of the training set.

Finally, as the labels of the input images were represented by a scalar value from 0 to 9, we one-hot encoded them into arrays of size 10.

We also note that, as we used mini-batches, the data was shuffled before selecting the batches at each epoch of training to reduce overfitting and increase generalization.

V. Results

We have implemented and tested four CG models:

- Softmax Regressor with Gradient Descent
- Neural Network with Gradient Descent
- Neural Network with Momentum
- Neural Network with Adam

All of them use cross entropy [7] as the cost function and have an input layer size of 3072 and an output layer size of 10 with the softmax activation function. The NNs have one hidden layer with 200 units and sigmoid activation.

The tests were made training each model for 24 epochs with 0.005 learning rate and a batch size of 625 (which leads to 128 iterations per epoch).

A. Softmax Regressor

The Softmax Regressor was implemented as an instance of our Computational Graph class. It is composed of 2 Layers, one input layer and one output layer (with softmax activation). Cross entropy was used as the loss function, alongside the classic Gradient Descent optimization algorithm.

We were able to achieve 32.92% accuracy with this model on the validation set (see Figure 2).

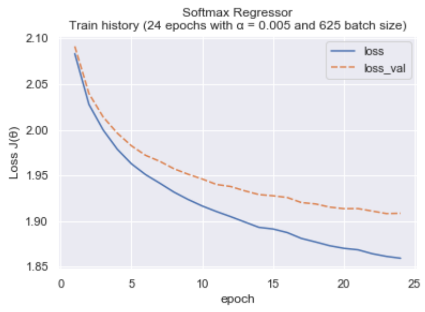


Figure 1: Loss *vs.* epoch for the Softmax Regressor

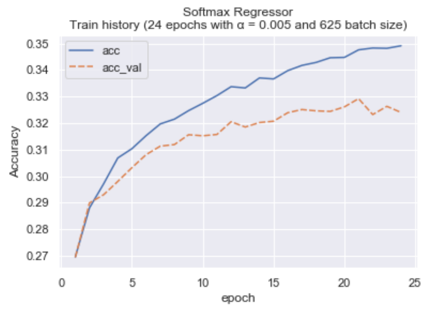


Figure 2: Accuracy *vs.* epoch for the Softmax Regressor

B. Neural Network

We have also used our Computational Graph class to implement the Neural Networks, composed of 3 Layers — an input layer, a hidden layer of 200 neurons and an output layer. The hidden layer uses sigmoid activation and the output layer uses the softmax activation, the loss is cross entropy and we've tested three optimizers: Gradient Descent, Momentum, and Adam.

When using gradient descent the network did pretty much the same as the regressor, with its best accuracy value on the validation set being equal to 32.88%.

However, the use of the two other optimizers did significantly improve classification performance. Adam resulted in 37.10% accuracy and Momentum in 40.00% — over 6 percentage points more than the models that used the classic gradient descent parameter update rule.

The variation of loss and accuracy values over training epochs is shown on Figures 3 and 4.

C. Optimizers

All optimizers work on the same fundamental idea, that is using the derivative of the cost/loss function to update the model parameters (weights and biases of each layer). Below we describe the three algorithms we have implemented, and how they try to accelerate convergence:

1) Gradient Descent: Uses only the gradient[4] and a learning rate to direct itself. Because of its simplicity it's by far the easiest to implement and the fastest of all algorithms, however it's prone to getting stuck at local minimums, and quite sensitive to the learning rate parameter, making it slow to converge at times — as seen by our results — or even completely diverging.

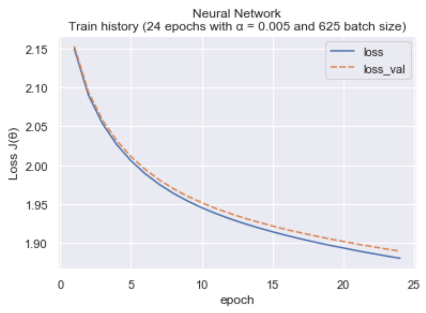
2) Momentum: Momentum is another relatively simple algorithm that, on top of the current gradient, stores and uses the last gradient values to accelerate convergence. It basically stores the last gradient and sums it to the update amount, weighted by a parameter γ that usually defaults to 0.9.

Being η the learning rate and v_t stored at time step t[5]:

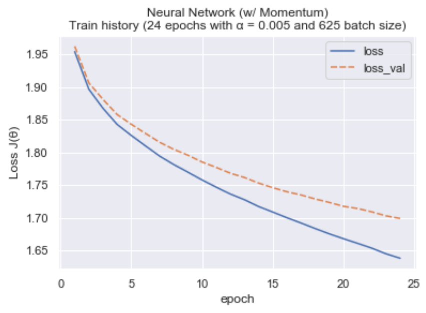
$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta) \implies \theta := \theta - v_t \tag{5}$$

[4]The derivative of the cost function J w.r.t. the parameters, that is, $\nabla_W J$ and $\nabla_b J$ (where W, b refers to the weights and biases)

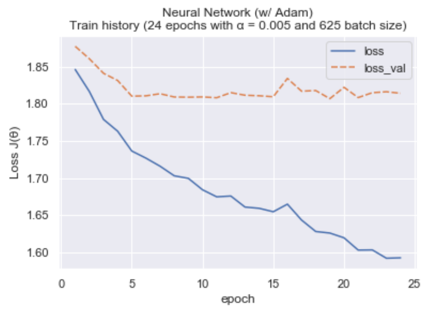
[5]$\theta = \{W, b\}$, i.e. the parameters



(a) NN with Gradient Descent (1.8892)

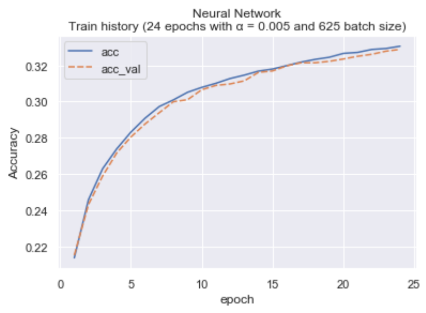


(b) NN with Momentum (1.6993)

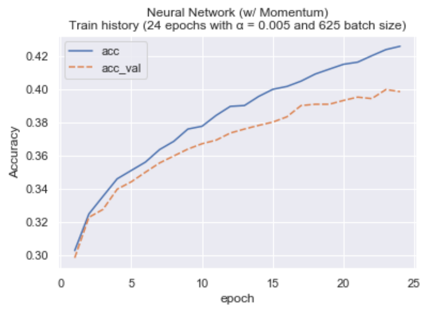


(c) NN with Adam (1.8070)

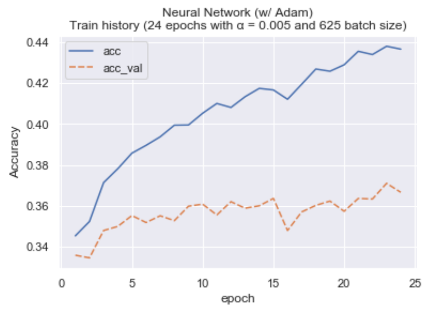
Figure 3: Loss *vs.* epochs for each Neural Network model (with best scores on the validation set)



(a) NN with Gradient Descent (0.3288)



(b) NN with Momentum (0.4000)



(c) NN with Adam (0.3710)

Figure 4: Accuracy *vs.* epochs for each Neural Network model (with best scores on the validation set)

With this small modification to the gradient descent rule it is able to converge much faster, because the momentum accelerates the descent in dimensions that keep their sign the same and slows in alternating sign dimensions. In other words, when the gradient changes directions it slows down and when it keeps its direction it accelerates.

3) Adam (Adaptive Moment Estimation): The Adam optimizer computes adaptive learning rates for each parameter. For this it stores an exponentially decaying average of the past gradients (m_t) and of the past squared gradients (v_t).

$$\theta := \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{6}$$

where $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected values (cf. [8])

The method has three parameters, with default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

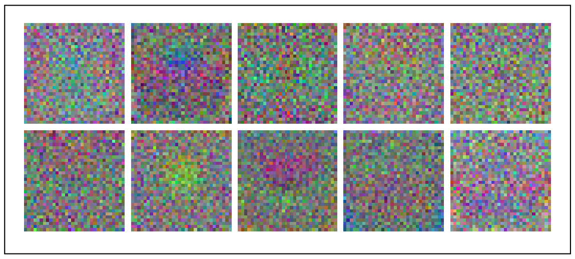


Figure 5: Visualization of the Softmax Regressor weights for each class (0 to 4 in the top row, 5 to 9 in the bottom row)

D. Test set results

The best values of loss and accuracy, together with the combined F1 score — calculated as the harmonic mean of the F1 score [9] average of each class — for every model on the validation set are shown on Table I.

Model	Best Loss	Best Accuracy	Combined F1 score
Softmax Regressor	1.9082	0.3292	0.2940
NN	1.8892	0.3288	0.2945
NN (Momentum)	1.6993	0.4000	0.3687
NN (Adam)	1.8070	0.3710	0.3144

Table I: Validation set statistics for the implemented models

As observed in Figures 2 and 4, the Neural Network with Momentum optimizer was the model that ended with the best accuracy values, and also with the highest combined F1 score (see Table I). So finally we used it to classify the images on the MINI CINIC-10 test set (composed of 10,000 unseen images), achieving a combined F1 score of 0.3548.

The F1 score for each class is shown in Table II, and the confusion matrix in Figure 6.

Class	F1 score	Class	F1 score
0 (airplane)	0.532	5 (dog)	0.233
1 (automobile)	0.435	6 (frog)	0.453
2 (bird)	0.309	7 (horse)	0.455
3 (cat)	0.278	8 (ship)	0.474
4 (deer)	0.265	9 (truck)	0.385

Table II: F1 scores for each class on the test set

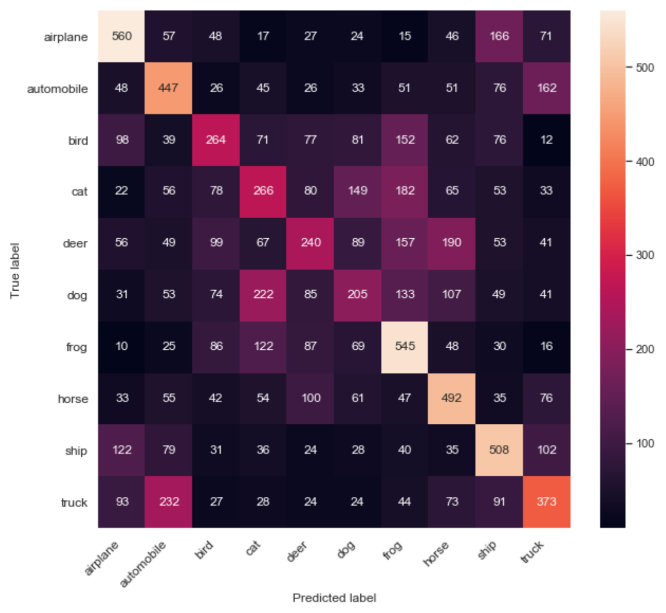


Figure 6: Confusion matrix of the test set image classification, made by our best model (1-hidden-layer NN with Momentum)

VI. Discussion

A. *Softmax Regression vs. Neural Network*

The Softmax Regressor achieved 32.40% and the best Neural Network got up to 40.00% accuracy on the validation set, altough it only did better than softmax regression when using more elaborate optimizers (note that we considered the same number of epochs and learning rate for all models).

B. *Keras implementations*

To get a basis of comparison for our early implementations, we have also created both models on Keras using its SGD optimizer, which performs gradient descent.

The results obtained were very similar to ours: 32.40% *vs.* Keras' 32.78% accuracy on softmax regression, and 32.88% against 33.09% on Keras for the neural network.

The processing time for each epoch was close when performing softmax regression: 2s for Keras *vs.* 3s in ours. However on the neural network our implementation was taking 7s, while Keras' only took 3s. This is probably due to better optimization of the backpropagation process on multiple layers, and was already expected of a popular library.

C. *Optimizers*

The results obtained with the different optimizers weren't what we had anticipated, being Momentum the best optimizer for the chosen parameters, followed by Adam (which we expected to do better) and, lastly, by the Gradient Descent.

Probably the small number of epochs and slightly low learning rate gave the Momentum optimizer an edge as it accelerates the descent without much consideration.

Another interesting fact is that our implementation of the Adam optimizer was much slower than the other optimizers when using a smaller batch size (of 80 on early tests). And, although the Momentum was a bit slower than the GD, its speed of convergence compensates for it, producing the best model we have used for the problem.

D. *Confusion Matrix*

It is interesting to see that the Neural Network learned a few of the classes fairly well, like airplane, automobile, frog, horse and ship (which also seem less blurry on Figure 5). However, it also tended to predict those classes way more than the others, confusing automobile and truck, airplane and ship, and bird, cat, deer and dog with frog.

VII. Conclusions and Future Work

Our best results were achieved with a 1-hidden-layer Neural Network using the Momentum optimizer, scoring 40% accuracy on the validation set.

Although we expected more from the Neural Network, the models would overfit the data when using many more epochs or higher learning rates. We think that by adding another hidden layer, implementing an early stop callback and using a smaller learning rate coupled with more epochs, the model would achieve better results.

Also, using convolutional layers instead of fully connected ones generally yield superior results on image related tasks, as they better exploit the spacial features of the image. Another possible way of getting better results would be by doing some extra pre-processing of the data, beyond just normalization (e.g. using visual descriptors like LBP or HOG, or even PCA to reduce dimensionality [10]–[12]).

It's possible to see in the confusion matrix that there's a lot of miss-classification happening, and although confusing so many classes like that isn't great, it's good to see that at least it tends to mess up on similar classes.

The notebook with the source code and the datasets used can be seen on github.com/laurelkeys/machine-learning (inside the `assignment-2` folder).

References

[1] "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145 – 151, 1999. 1
[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2015. [Online]. Available: http://arxiv.org/abs/1412.6980 1
[3] Christopher Olah, "Calculus on Computational Graphs," 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Backprop/ 1
[4] Stanford CS231n, "Backpropagation, Intuitions." [Online]. Available: http://cs231n.github.io/optimization-2/ 1
[5] P. Dahal, "Classification and Loss Evaluation." [Online]. Available: https://deepnotes.io/softmax-crossentropy#cross-entropy-loss 1
[6] "Standard score — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Standard_score 2
[7] "Cross entropy — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Cross_entropy#Definition 2
[8] S. Ruder, "An overview of gradient descent optimization algorithms." [Online]. Available: http://ruder.io/optimizing-gradient-descent/ 3
[9] "F1 score — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/F1_score 3
[10] "LBP — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Local_binary_patterns 4
[11] "HOG — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients 4
[12] "PCA — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Principal_component_analysis 4
Links accessed on October 2019.