Abstract

This project focused on implementing MLP models using NumPy, and we investigated the performance of various choices regarding MLP architecture, activation functions, and regularization. Moreover, a combined CNN-MLP model with 2 convolutional and 2 fully connected layers, and an altered ResNet-18 model with frozen CNN weights and new fully connected layers were also implemented and investigated. For training and testing, we used the CIFAR-10 dataset, which is a commonly used image set for benchmarking image classification models. During implementation, we found that more hidden layers and more hidden units increased MLP performance, but would often increase training time in custom coded MLP models. All investigated activation functions (ReLU, Leaky ReLU, Tanh) displayed roughly similar performance, with Leaky ReLU providing slightly better test accuracy and less training time. Tanh layers, on the other hand, increased model training times since the function and its derivative are computationally demanding. Furthermore, the addition of L1 and L2 regularization improved MLP performance, with L1 regularization also being able to decrease model training times. In contrast to the MLP models, the combined CNN-MLP model provided decent performance. Interestingly, the altered ResNet-18 model provided good training accuracy (~80%) but the test set accuracy was much lower (~50%), which has been outlined in some previous reports of the harmful effects of overfreezing in transfer learning frameworks. This ResNet-18 model also required an excessive amount of CPU time for training in comparison to the MLP models, making GPU training the temporally economic option for this project.

1. Introduction

For our image classification model, we used NumPy to implement a MLP (Multi-Layer Perceptron) and explored the effects of different hidden layer numbers (none, 1, 2), different activation functions (ReLU, Leaky ReLU, Tanh), and regularization (L1 and L2) on final performance. Additionally, for benchmarking, we used PyTorch to implement a combined CNN-MLP model and a model using pretrained ResNet-18 CNN layers with frozen weights and MLP layers. CNN and MLP models have been historically used in many fields, such as medicine ([1-3]). ResNet-18 was proposed as an example of a model architecture to alleviate several observed neural network problems, such as vanishing gradient, and performed quite well on several image datasets [4]. For the training and testing of all models in this project, we used the CIFAR-10 image dataset, which has been extensively used for benchmarking various neural network models [5, 6].

During implementation, we observed that adding hidden layers to the custom coded MLPs improved model performance but required more training time. We also observed that changing the number of units (e.g., from 256 to 512) also improved performance but required more training time. Among the different activation functions, Leaky ReLU showed slightly improved accuracy and training time, while the Tanh model had an inflated training time, possibly due to the computational cost of computing the Tanh function and its derivative. Moreover, the addition of regularization improved model accuracy, with L1 regularization also slightly decreasing training time, possibly due to some weights being shrunk to 0. In comparison to the MLP models, the CNN-MLP model displayed superior performance in line with previous reports, and the altered ResNet 18 model outperformed the MLP models while displaying very good performance. This observation supports the benefits of using pre-trained layers in tasks with similar data and objectives.

2. Datasets

The CIFAR-10 dataset is a common benchmarking image dataset composed of 60000 low resolution (32x32) images [7]. These images are split into 10 classes, each designated to either ‘plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', or 'truck'. The dataset is then split into one training dataset (N=50000) and one testing dataset (N=10000), and the distribution of classes within the five training datasets is equal, with exactly 5000 images in each class. This ratio also applies to the testing dataset, in which each class has 1000 images from the 10000 instances. Examining the dataset further, we can observe that every pixel within each image can be defined by an integer between 0 and 255 in the three channels representing the RGB color values. This can allow the dataset to be normalized and represented as a numerical array. Additionally, CIFAR-10 is a commonly used dataset to train and test various network architectures. For instance, a paper by Chenxi et al. explored the utilization of a sequential model-based optimization (SMBO) strategy for learning the structure of convolutional neural networks (CNNs) using the CIFAR-10 dataset [5].

3. Results

3.1 Data normalization and preprocessing.

The MLP model class was implemented from scratch using NumPy. We vectorized the RGB images (from 32*32*3 to a vector of 3072) and normalized the pixels by dividing them by 255. For the CNN sections, the images were normalized using PyTorch’s transform library. For the MLP models, a validation set using 10% of the initial CIFAR-10 training set was also used to monitor performance and avoid overfitting by a loss improvement threshold of 1e-5 and tolerance of 5. Mini-batch gradient descent (N=256) was used for training models as it can increase training speed. MLP models, unless stated otherwise, were trained using 500 epochs and with a learning rate of 0.01. It should be noted that the test results are not the best possible values, notably in comparison to the PyTorch MLP models, as we had to limit the training and architecture design based on limited available computation resources, and we implemented two PyTorch MLP models using GPU to illustrate this. Lastly, softmax function inputs were also rescaled for additional numerical stability (expz= exp(z-max(z)) => out=expz/(sum(expz))). Therefore, the calculated loss could be numerically different than those calculated using the PyTorch package.

3.2 Models with different hidden layer architectures

We initially implemented the MLP model using three configurations; no hidden layers, 1 hidden layer with 256 units, and 2 hidden layers with 256 units. The model with no hidden layers displayed lower performance on the test set (Table 1) compared with the other two models. The model with 1 hidden layer performed worse than the model with 2 hidden layers, though the difference was small (Table 1). Training and validation loss dynamics over training epochs revealed that the model with 2 hidden layers displayed the sharpest drop in training and validation losses, while the model with no hidden layers displayed more gradual dynamics, especially after the first few epochs (Figure 1A, B). The observed results are in line with the observation that increasing the MLP model depth enables the model to learn non-linear patterns and increases the accuracy, especially when the task is not linear. It is also expected that adding hidden layers would increase the training time. Indeed, the training time for the model with 2 hidden layers was roughly 3 times that of the model with no hidden layers (Table 1).

We were also curious to see if adding additional layers beyond 2 (e.g. a model with 3 256-unit hidden layers) or increasing the size of the 2 hidden layers from 256 to 512 would further impact the model performance, and we observed a minor performance improvement when another 256 unit layer was added to the 2 layered model (Table 1). Interestingly, increasing the size of the hidden layers from 256 to 512 resulted in a noticeable increase in test accuracy from 39.16 to 42.67 (Table 1). The training and validation loss dynamics over training iterations also pointed to better performance of the model with 2 512-unit hidden layers compared with those with 2 and 3 256-unit hidden layers (Figure 1 C, D). The high dimension of the input data (i.e., 3072) could be a contributing factor to this observation by causing a certain degree of information loss when it is passed through smaller layers. However, a downside of using the 512-unit hidden layers is that the training time almost doubles compared with the model with 2 256-unit layers (Table 1).

Table 1: Performance characteristics of models with different hidden layers

Model	Loss	Test accuracy	Training time (s)
No hidden layers	0.339	20.12	2098
1 hidden layer	0.185	36.48	7077
2 hidden layers	0.172	39.16	7399
3 hidden layers	0.168	40.37	8206
2 hidden layers with 512 units	0.162	42.67	14890

Models were trained using a learning rate of 0.01 over 500 epochs.

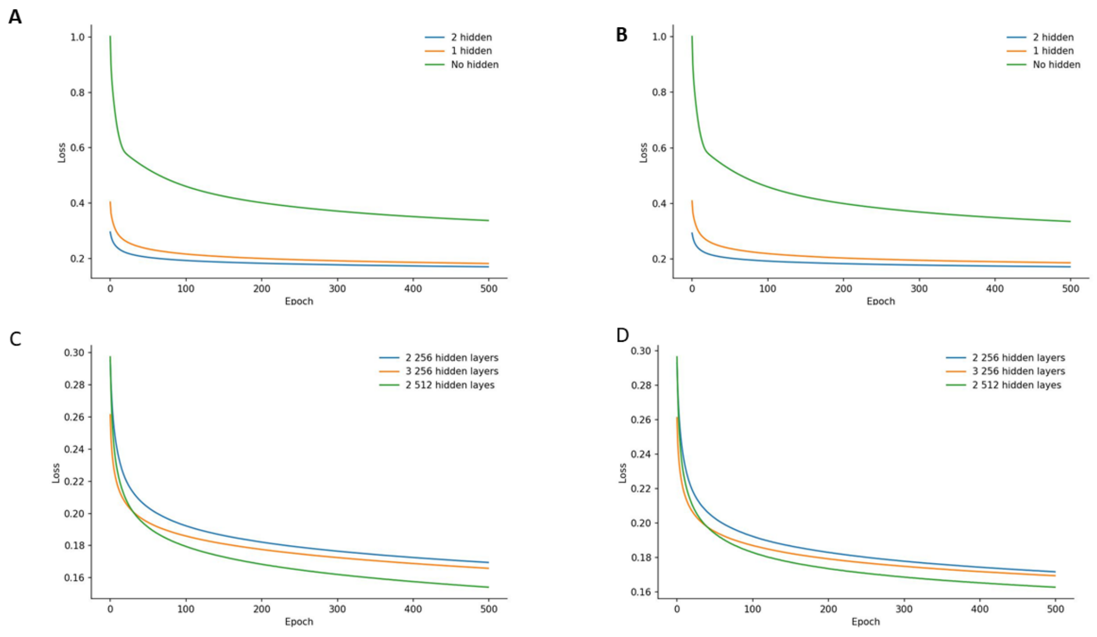

Figure 1. Training (**A**) and validation (**B**) loss dynamics of the models with 2, 1, and no hidden layers. Training (**C**) and validation (**D**) convey the same information but for the models with 2 256-unit, 3 256-unit, and 2 512-unit hidden layers. Plots are displayed over training epochs.

Finally, as a step beyond the project boundaries, we opted to validate our results via training MLP models using PyTorch and GPU. The PyTorch model with 2 256-unit layers and without momentum attained accuracy very similar to ours (~40%) over 500 epochs, while another model trained with 1024-unit and 512-unit layers reached accuracies around 51% on the test set over 200 epochs using SGD with momentum. These results are in line with our observations.

3.3 Effect of different activation functions

Next, we opted to investigate the effect of different hidden layer activation functions on model performance and training times. We observed that the performance of the models on the test set were significantly close to each other (Table 2). Furthermore, the training and validation loss dynamics over training epochs were also close with Leaky ReLU (alpha=0.01) displaying slightly faster convergence (Figure 2A, B). Significant differences were observed regarding the training times, with models using the Tanh activation function taking approximately twice as much training time as ReLU and Leaky ReLu (Table 2). This observation is expected, as the Tanh derivative is much more computationally expensive than those of the ReLU and Leaky ReLU functions.

Table 2: Performance characteristics of models with different activation functions

Model	Loss	Test accuracy	Training time (s)
Tanh	0.172	39.83	11145
Leaky ReLU	0.172	39.82	6379
ReLU	0.172	39.16	7399

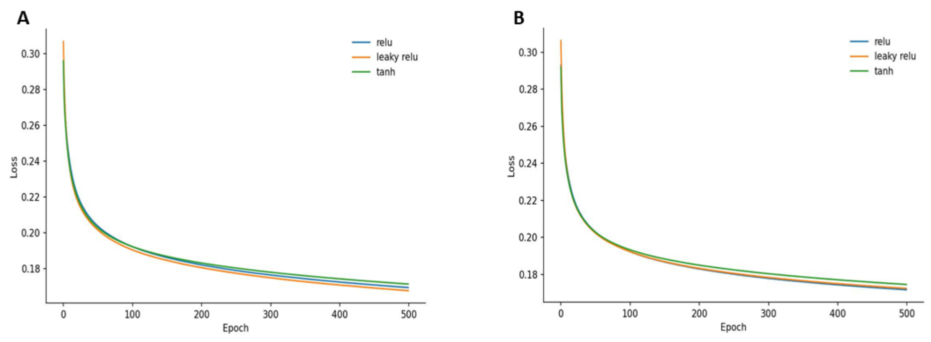

Figure 2. Training (**A**) and validation (**B**) loss dynamics of models with different activation functions.

3.4 Effect of regularization on MLP model performances.

A regularization strength of 0.1 was used for the models with L1 and L2 regularizations. Models with regularization displayed slightly better performance on the test set (Table 3), possibly due to better model generalization from regularizations. Training and validation loss dynamics also showed a somewhat close trend for models with and without regularization (Figure 3 A,B). Interestingly, the model with L1 regularization displayed a lower training time in comparison to the models with L2 and no regularizations (Table 3). This could stem from the nature of L1 regularization, which tends to drive some weights to 0, possibly decreasing the computation time and complexity.

Table 3: Performance characteristics of models with different regularizations

Model	Loss	Test accuracy	Training time (s)
L1 regularization	0.169	40.22	6880
L2 regularization	0.169	40.47	7218

No regularization	0.172	39.16	7399

Regularization strengths (alpha and beta) were set to 0.1

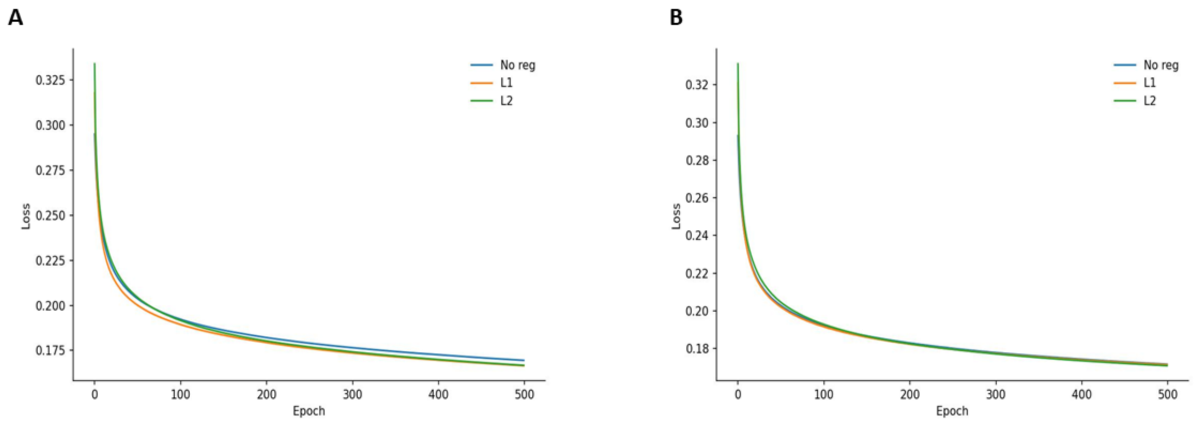

Figure 3. Training (**A**) and validation (**B**) of models with different states of regularization. A regularization strength of 0.1 was used for L1 and L2 models.

3.5 Training MLP model on unnormalized data

Using unnormalized pixel values between 0 and 255 caused numerical instability and large values that resulted in calculation overflow and an untrainable model. With the model not learning from the data, it had terrible test performance. Furthermore, altering the softmax function by adding small values (epsilon) to the denominator did not resolve this issue. This observation is possibly due to the accumulative effect of large pixel values (e.g., 250) on calculated loss and gradient, causing NaN values in the process.

3.6 Convolutional Neural Network

Using PyTorch, we created a hybrid model with 2 convolutional and 2 fully connected layers and trained it on the CIFAR-10 dataset using cross entropy loss and stochastic gradient descent (lr=0.005, momentum=0.9). For the hyperparameters of the convolutional layers, both have a kernel size of 3x3, a stride of 1, and a padding of 1, while the first convolutional layer has 3 channels in and 16 out, giving the second convolutional layer 16 in (since it must be the same size) and 32 out. Both fully connected layers have 256 channels out, while the first has 2048 in and the second has 256 out. For regularization, we added dropout (with p=0.25), since a high initial train accuracy and lower test accuracy was evident, suggesting overfitting. This gave us a test accuracy of 65.05% after 40 epochs, so we added additional regularization methods from torchvision.transforms like RandomHorizontalFlip and RandomCrop which we used for our final results. During the previous implementations, the train accuracy stayed around 80%. Finally after 40 epochs with GPU implementation, the model reached a training accuracy of 81.1% and a test accuracy of 68.45. Unfortunately due to time constraints the training results are not completely visible as output on the ipynb file, but are available after running the program. Therefore, the combined CNN-MLP model was vastly superior to MLP for this task. However, the model training, even on GPU let alone CPU, was time consuming. The following figure displays the CNN-MLP loss over training epochs.

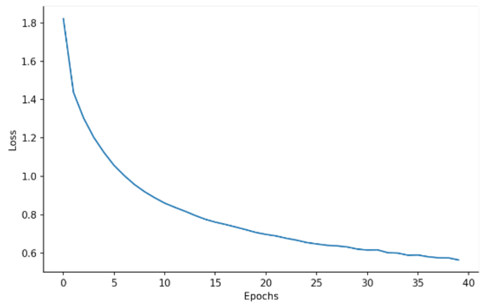

Figure 4. Training loss dynamics of the CNN-MLP model over 40 training epochs.

3.7 ResNet18

Next, we loaded ResNet18 as our pre-trained model using the PyTorch library, then froze the convolutional layers and added one fully connected layer (N=512) and one output layer. A low number of fully connected layers was chosen to avoid overfitting and reduce training time. The results revealed a higher accuracy than the multilayer perceptron model, especially on the training set, with 77% training accuracy after 50 iterations, compared with 40% from manual MLP after 500 iterations. Interestingly, model performance on the test set remained much lower at around 50%. The observed disparity persisted with different hidden layer depths and widths. We hypothesize that 'overfreezing', as reported previously, could harm model generalization by providing a high training performance but low test performance [8]. The train set losses across the training epochs can be seen in the graph below:

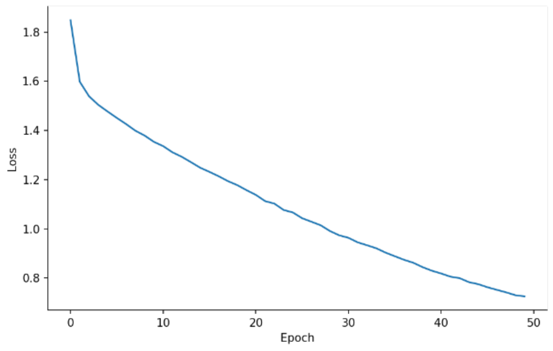

Figure 5: Train set loss dynamics of altered ResNet 18 model over 50 epochs

To compare the running time of the operation, a set of 10 epochs was implemented on CPU to evaluate the average time taken to run a single iteration without the GPU, which allows for an equivalent comparison to the MLP models. As the running time required to complete 10 iterations on the CPU was 5177 seconds, translating to over 86 minutes, the running time for the pre-trained model was much higher than the MLP without the implementation of the GPU. Thus, overall, an increase in accuracy could be observed when using a pre-trained model, at the price of computation speeds which can become quite time-consuming with larger numbers of iterations, especially without GPU.

4. Conclusion

We observed that implementing MLP models from scratch on Python and training them on CPU quickly becomes computationally expensive and time consuming, especially as the depth and width of the model increase. Models with more hidden layers performed better, especially compared with the no hidden layer model. Furthermore, using hidden layers with unit numbers proportionate to the input dimension (eg., D=3000; 512 layer vs 256) could also increase accuracy by preventing information loss caused by a forced dimension reduction. Regarding different activation units, Leaky ReLU performed better as it slightly improved accuracy and decreased training time. On the other hand, the Tanh model considerably inflated the training time possibly due to the function and its derivative being more computationally expensive to compute. The addition of L1 and L2 regularization improved model accuracy, with L1 regularization also decreasing training time, potentially due to simplifying computation by driving some weights to 0.

We observed that a combination of CNNs and MLPs performs much better than MLPs alone in line with previous reports. Furthermore, using pre-trained CNN layers from ResNet18 can jumpstart a model and provide superior performance, but training using CPU is very time consuming. GPU, on the other hand, significantly boosted model training speed compared with CPU models.

Future projects can unfreeze CNN layers of ResNet and compare training time and performance with those extracted in this project. Moreover, other optimizers, such as Adam, and regularization techniques, such as dropout, could also be explored to compare the effect of various training and generalization approaches.

Statement of Contributions

Yu Cheng: Implementation and testing ResNet model. Report drafting and revision.

Taylor: Implementation and testing CNN-MLP model. Report drafting and revision.

Mahdi Mahdavi: Implementation and testing MLP models. Report drafting and revision.

References

1. Yan, H., et al., *A multilayer perceptron-based medical decision support system for heart disease diagnosis.* Expert Systems with Applications, 2006. **30**(2): p. 272-281.
2. Bikku, T., *Multi-layered deep learning perceptron approach for health risk prediction.* Journal of Big Data, 2020. **7**(1): p. 1-14.
3. Kayalibay, B., G. Jensen, and P. van der Smagt, *CNN-based segmentation of medical imaging data.* arXiv preprint arXiv:1701.03056, 2017.
4. He, K., et al. *Deep residual learning for image recognition*. in *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
5. Liu, C., et al. *Progressive neural architecture search*. in *Proceedings of the European conference on computer vision (ECCV)*. 2018.
6. Xu, J., et al., *RegNet: self-regulated network for image classification.* IEEE Transactions on Neural Networks and Learning Systems, 2022.
7. Krizhevsky, A. and G. Hinton, *Learning multiple layers of features from tiny images.* 2009.
8. Dar, Y., L. Luzi, and R.G. Baraniuk, *Overfreezing Meets Overparameterization: A Double Descent Perspective on Transfer Learning of Deep Neural Networks.* arXiv preprint arXiv:2211.11074, 2022.