Syed Rizvi
Arno Dunstatter
Sevilay Ozturk
Elisa Martinez

Dr. Christopher Eick
CS 3337

Image Classification Project Final Report

A. Introduction

Image classification on large-scale open source datasets has been an area of intense interest and research within Computer Vision. Since the advent of large-scale deep learning and the resurgence of neural network popularity around 2008-2012, performance on computer vision tasks has rapidly improved, and many advanced techniques have been developed to improve the training and stochastic learning process. In the context of what we have learned this semester, our group perceived an opportunity to explore machine learning algorithms for doing image classification and educate our fellow peers about which techniques, processes, and frameworks performed well. More concretely, through our experiments detailed below we wanted to compare and contrast the image classification performance of various implementations Convolutional Neural Networks (CNNs) and Support Vector Machines (SVMs), and study how regularization and data augmentations techniques affect the overfitting tendencies of the CNN models.

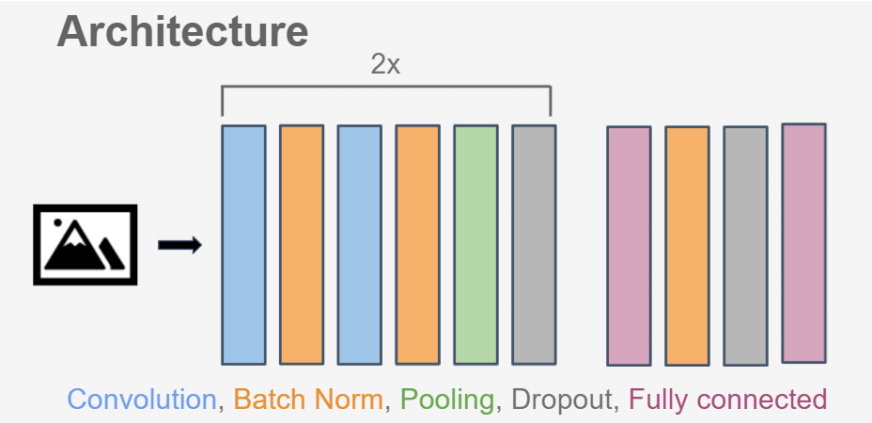
For our dataset, we chose the well-established cifar10 image dataset [1], organized by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton at the University of Toronto. This dataset is a 10-class classification problem of common objects, such as cats, boats, cars, and trucks. Our choice to use the Cifar 10 dataset and CNNs was a reflection of recent trends we see in high-performing neural networks [2] as well as many experiments and verifications done on public image datasets. Additionally, we were curious as to why CNNs seemed so dominant in image classification and decided to test the performance of SVMs on this data set to gain insight into their relative paucity. The reduced computational complexity of the dataset, which consists of 60,000 32x32 resolution images, also helped make these experiments feasible for our group to run in a realistic timeframe. We chose to divide the work among our four group members in a reasonable manner, with Syed and Elisa conducting neural network experiments and Arno and Sevilay conducting research and implementations of SVMs.

B. Methods

For CNNs we developed implementations in both Pytorch and Keras, two prominent deep learning libraries within Python. All experiments were run either in Google Colab or local Jupyter notebooks (using our laptop GPUs). For our Pytorch experiments, we started with a simple two-layer CNN network that Pytorch recommended on its beginner's tutorial. This network was composed of simple alternating convolutional and pooling layers to encode image features and downsize the 32x32 image, creating progressively deeper features in later layers of the CNN. The model ended with three fully-connected layers, forming a multilayer perception classifier at the end of the model to classify the deep CNN features into the final 10-class output. This two-layer model had a modest trainable parameter total of 62,006 parameters. Some hyperparameters that we used on this CNN (and on all subsequent CNN experiments in Pytorch, to introduce some level of consistency for testing independent regularization techniques) were a learning rate of 0.0003, a batch size of 16, ReLU activations, and the Adam optimizer. These choices of hyperparameters were chosen based on common best practices that we saw used in research papers as well as in many tutorial implementations of Convolutional Neural Networks.

Expanding from our two-layer CNN, we added two more convolutional layers for our next experiment to add more computational power in our network, resulting in a baseline four-layer CNN model with 591,914 trainable parameters. The experiments that followed was an ablation study of batch normalization [3], dropout [4], regular augmentation, and advanced image data augmentation techniques, where we added one element to our CNN to measure its impact on accuracy and overfitting before removing it and trying another technique. Batch normalization and dropout were implemented in the

network by adding batch normalization layers directly after convolutional layers and dropout layers after convolutional/pooling blocks. A reference architecture diagram with all layers included is shown below.



Data augmentation techniques were implemented outside of the Pytorch CNN models in the data loader object that loaded images from the dataset before passing them through the model. For simple augmentations we used Pytorch's torchvision library to compose a chain of probabilities of applying random rotation, horizontal or vertical flipping, and color augmentation applying to a single image. For more advanced data augmentation techniques we implemented the AugMix algorithm [5] using open-source code from google research, which mixed branches of augmentations (implemented using Numpy and pillow Python libraries) to create numerous variations of images within the data loader.

The last Pytorch experiment we ran was a six-layer CNN model with batch normalization and dropout. The construction of this last experiment followed the best-performing four-layer experiment, and was done in the context of the upper limits of our computational resources (training this six-layer CNN with AugMix added in addition would take too long to run).

For visualization, all Pytorch experiments relied on the tensorboard library to do continuous logging during experiments and create accuracy and loss graphs. Accuracy and loss measurements were calculated throughout training through standard loss and output accuracy calculations, and graphs were created by accumulating these measurements over epochs and logging them to tensorboard scalar summary graphs. In addition, the random seeds for Python, Pytorch, Numpy, and Pytorch-CUDA backend were all manually set to 40 for all Pytorch experiments to ensure reproducible results.

Considering the results of the Pytorch experiments, we started with a simple custom 3-layer CNN using the Sequential class to group a linear stack of layers for our first **Keras** model. We used the following techniques for building the Model - CNN, Max Pooling and Dense Layers; for Activation Function - ReLU (in CNN layers for handling image pixels) and Softmax (for final classification); for handling Overfitting (Regularizing) - Dropout Layer; for normalizing/standardizing the inputs between the layers (within the network) and hence accelerating the training, providing regularization, and reducing the generalization error - Batch Normalization Layer. We also used data augmentation to avoid overfitting our data. We change the images position, inclination, zoom, color, focus, and other techniques to generate more images from one. This technique helps the computer to realize that an image is what the image is, no matter where it is located. For our experiment we finally used a shift in width and height, and horizontal and rotation given an angle.

We then applied a similar approach to what was seen in class where we tried different combinations of hyperparameters including: increasing number of epochs,, and batch sizes of 10, 32, 64, and 128. Finally we answered the following questions: how Keras implementations are different from Pytorch, how does image classifier trained on cifar10 predict/perform on random images of dogs/cats/cars taken from google images, does it transfer to new datasets well, what does it predict when you give image classifier an object that is not one of the 10 classes?

For SVMs, Scikit-learn libraries were implemented. Initially the SVC module from sklearn's SVM library was attempted to be implemented using the radial basis function as a kernel for nonlinear separation, however, once the training of the model, which relies on one big batch to perform gradient descent, took in excess of 2 hours it was realized that the training of SVMs often occurs in quadratic time [6] and therefore they are unsuitable to large data sets. As a result it was decided to train the SVC with only a fifth of the original training set which took only 5 minutes. Subsequently it was decided to implement the SGDClassifier module from sklearn's linear_model library because this module allowed the use of partial fitting so that the large cifar10 dataset could be split into smaller batches. Hyper-parameter tuning was explored, with the first round of tuning cycling through combinations of epoch counts of 1, 10, 20, 40, and 80 and batch counts of 5, 10, 25, 50, and 100, and the second round of tuning cycling through combinations of epoch counts of 5, 10, and 15, and batch counts of 100, 500, and 1000. Lastly, the original SVC module utilizing the radial basis function kernel was allowed to run overnight, despite the quadratic time complexity of its fitting coupled with the large training set (50,000 images), and was thereby trained in approximately 3.5 hours.

C. Results and Discussion

The results of our Pytorch CNN experiments mostly followed what we expected from the regularization and data augmentation techniques. In this section we will reference accuracy and loss measurements as well as trends that are present in the figures on our presentation slides. The initial two-layer CNN with no extra regularization or data augmentation techniques was able to achieve decent accuracy (80% on training set and 65% of validation set), however it was overfit after 5 epochs of training. The loss curve reflected this overfitting problem, with the validation loss increasing after 10 epochs while the training loss continued to decrease up to 30 epochs of training. To further study this divergence we analyzed the four-layer plain CNN Pytorch model; the accuracy improved (85% on training set, 70% on validation set after 30 epochs), however the overfitting was still present to the same degree (15% divergence in accuracy). From this we can interpret that adding more convolutional layers can improve overall accuracy to a point, since the CNN has more computational power to perform its classification task, but steps need to be taken to reduce the overfitting on the training set.

In our four-layer CNN + batch normalization experiment, we expected the overfitting to reduce somewhat and the performance to improve a little bit. Our intuition for this was in the discussion given in the original batch normalization paper by Sergey Ioffe and Christian Szegedy, where they inferred that by normalizing the activations of incoming signals to neural layers, neural network training could converge faster since the signal follows a more fixed distribution. Batch normalization is implemented by shifting incoming activations in each channel across a batch by its mean, and scaling by the standard deviation, resulting in a mean of 0 and unit standard deviation. It thus made sense to us that better gradient updates could occur during training with batch normalization, since the distribution of incoming activations was more constant and specific weights could be tuned to process more consistent distributions of input signals. In practice, we observed from our experiment that batch normalization did not have a strong regularization effect since overfitting still occurred, however there was an enormous gain in performance

accuracy (95% on training set, 80% on validation set after 30 epochs) due to the faster convergence provided by normalizing input distributions to neural layers. We concluded that implementing batch normalization provided faster training convergence and gave us leeway to use higher learning rates, since we could count on neural layers updating their weights based on more consistent distributions of inputs.

Stepping away from batch normalization, we tried adding dropout to our baseline four-layer CNN for our next experiment. Dropout zeros out the activations of certain neurons according to a certain probability, making the neural network predict and update its weights with only a subset of its total neurons. In our experiment, we found dropout to be an extremely strong regularizer. The gap between the training accuracy (68%) and the validation accuracy (70%) was basically completely closed, and the loss curves for the dropout experiment stayed close together as well. The discussion in the dropout paper by Srivastava et al. discusses the effect dropout has on neural networks, stating that the technique effectively samples from numerous "thinned" neural networks while training, which prevents specific neurons and units from adapting and relying on each other too much [4].We observed this in our results, and were surprised that adding dropout layers was already enough to eliminate overfitting from the CNN models on the Cifar classification task. We followed up our batch normalization and dropout experiments with a quick four-layer CNN experiment implementing both of the techniques, which resulted in a CNN model with high accuracy and no overfitting (80% training accuracy, 81% validation accuracy). We found this combination of regularizers to create good performance on the cifar10 dataset, without any data augmentation needed in the model.

For the next two four-layer CNN experiments we ran concerning data augmentation, we found that using data augmentation to increase our effective training dataset size was also an effective regularizer on the CNN models. Simple data augmentations such as flipping and rotations helped maintain accuracy generalization (60% training accuracy, 64% validation accuracy), and the loss curves also showed no sign of divergence in 30 epochs of training. It is worth mentioning however that doing simple data augmentation resulted in lower accuracies compared to using batch normalization + dropout. For our experiment using advanced AugMix augmentation branches, we noticed that it once again provided good regularization on the CNN (72% training accuracy, 74% validation accuracy), however the overall accuracy was higher than when we used simple augmentations. We attribute this to the specific Jenson-Shannon Divergence loss term proposed in the AugMix paper [5], which we added to our loss term in our training setup. This loss term, with roots in divergence mathematical theory, is beyond the scope of this project report to explain, however our understanding of it is that it forces the activations of the neural network to be similar for an image and its augmented version. The intuition behind this is that we as humans think about an image and its variations in a similar way, thus a CNN ought to also encode augmented versions of images in a similar way. This consistency loss term makes the CNN model more resistant to image corruptions, and can also help improve accuracy at times like in our experimental results. One thing worth noting, however, is the increased computational overhead of having to augment every image dynamically; each epoch while training with AugMix took 5 minutes to complete, versus one minute or less for every other four-layer CNN experiment we ran using Pytorch on our local GPU.

The final Pytorch CNN experiment we ran was a six-layer CNN model with batch normalization and dropout added, and no data augmentations. For this last experiment we wanted to see the highest accuracy we could obtain while keeping overfitting under control and not adding too much computational complexity with data augmentation. This experiment resulted in 82% validation accuracy and 80% training accuracy, which was a very small improvement over the four-layer CNN model. We hypothesize that in order to improve performance up to and beyond 90% accuracy without overfitting, we will need to

use more complex neural network architectures, such as recurrent networks or more advanced convolutional layers.

Our first experiment with the Keras library yielded a 83.46% validation accuracy and a 80.9% training accuracy. This experiment had the following hyperparameters: epochs: 50, batch size: 32, 3-layer CNN + a final Dense Softmax layer for the final classification, with 0.2, 0.3, 0.4, and 0.5 dropout rates, respectively. We also used the 'Adam' optimizer. For the next three experiments, we increased the number of epochs and tried different batch sizes. The results of the last three were:

84.49% validation accuracy and a 82.2% training accuracy with 100 epochs and batch size of 64;
82.39% validation accuracy and a 81.37% training accuracy with 150 epochs and batch size of 100;
85.62% validation accuracy and a 82.55% training accuracy with 200 epochs and batch size of 128;

We also wanted to study how Keras implementations are different from Pytorch. Keras is an Open Source library written in Python which is capable of running on the top of TensorFlow, R, Theano etc. Keras it's a high-level API focused Library which has a bigger community than Pytorch, but the latter is easier to debug, has better training speeds, and gives you more insights on how the science of it works. Pytorch is more like doing python coding and if you want to implement your own neural network layers, PyTorch is the tool for you. Researchers favor Pytorch over Keras.

The Keras experiments and simulations alone do not give students a full appreciation of the full power and complexity of neural network-based controls, but this project opened the door to exploration of these powerful machine learning models. The state-of-the-art algorithms can give us nearly 99% of accuracy [7]; this project also pointed out the problem a neural network can have when there is insufficient training data and the drawback it has regarding lengthy training time, but it also gave us insights on the different adjustments based on the dataset we can perform on these.

The SVMs which were trained were of two main types, linear and nonlinear. The linear SVMs achieved accuracies between 27% and 39%, taking about 1.5 hours to train all 34 altogether. The nonlinear SVMs trained with 1/5th of the training set data and 100% of the training set data achieved accuracies of 47% and 54% and had training times of 5 minutes and 3 hours, respectively. Given the fact that at any given pixel location, for any color value, there is a high degree of overlap between multiple classes out of the 10, these results are what should be expected since the classes are not linearly separable in their original dimensions, but become more separable through the use of the radial basis function kernel which thereby enables the use of hyper-planes in higher dimensions in order to separate the classes.

D. Conclusion

From our experimental results, we have come to the conclusion that neural networks outperform SVMs in image classification tasks due to their higher model complexity and their more effective utilization of minibatch stochastic gradient descent learning for non-linearly separable data. Neural network performance, however, is greatly dependent on the training setup, hyperparameters, and architecture of the model. Our CNN models with optimal layer configurations (i.e. batch normalization layers after every convolution, dropout layers present) outperformed less optimal configurations by as much as 15% on the cifar10 validation set. When comparing CNN to SVM techniques, our group found through our varying implementations and experiments that neural network architectures were very flexible and open for experimentation and improvement, while the predetermined kernels that SVMs

relied on left little room for us to try and improve performance. SVM accuracy on the validation set of Cifar 10 was limited by the fact that SVM parameter counts increase linearly with the size of input, and also because SVMs accuracies do not significantly improve even when expanding the size of the training set by a factor of 5. This can be seen from the difference of the accuracies between SVMs trained on 1/5th of the dataset versus the entire training data. Overall, our project serves as a confirmation of the classification prowess of CNN models when set up and architected correctly, as well as a study of the impact and side effects of different regularization and data augmentation techniques on CNNs.

E. Python Libraries Used

[Pytorch Experiments]
Pytorch==1.8.0
Torchvision==0.9.0
Tensorboard==1.15.0
Numpy==1.21.4
Matplotlib==3.4.3
Sklearn == 1.0.1

A full list of requirements for the Pytorch experiments will be provided in the code submission in the form of a requirements.txt file.

[Keras CNN & SVM Experiments]
The requirements for Keras CNN and SVM experiments mostly overlap with Pytorch experiment requirements, with the exception of needing the Keras and Scikit-learn libraries.
Keras-Preprocessing==1.1.2
Keras==2.6.0

F. References

Citations

1. Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images." (2009): 7.
2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25, 1097-1105.
3. Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456). PMLR.
4. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.
5. Hendrycks, Dan, et al. "Augmix: A simple data processing method to improve robustness and uncertainty." arXiv preprint arXiv:1912.02781 (2019).
6. Yangguang Liu, ; Qinming He, ; Qi Chen, (2004). [IEEE Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788) - Hangzhou, China (June 15-19, 2004)] Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788) - Incremental

batch learning with support vector machines. , 2(), 1857–1861. doi:10.1109/WCICA.2004.1340997

7. "Papers with Code - CIFAR-10 Benchmark (Image Classification)". *Paperswithcode.com*, 2021. Online. Internet. 16 Nov. 2021. . Available: https://paperswithcode.com/sota/image-classification-on-cifar-10.

Code Repositories
https://github.com/google-research/augmix