A Convex Approach to Two-Layer Convolutional Neural Networks for Binary and Multi-Class Classification

Poojit Hegde, Ananya Karthik, Shreya Shubhangi, Burak Bartan
Stanford University
450 Serra Mall, Stanford, CA 94305
phegde8@stanford.edu, ananya23@stanford.edu, sshubhan@stanford.edu, bbartan@stanford.edu

Abstract

Convex optimization formulations for neural networks have strong potential. They promise high reproducibility and efficiency, faster convergence to a guaranteed global solution, and the decoupling of optimizer parameters from the study of the neural network model. In this paper, we expand on prior work and explore the use of convex simple semidefinite programs (SDP) for two-layer convolutional neural networks. For our preliminary experiments, we implement a convex CNN for binary classification as described by Bartan and Pilanci. The convex binary CNN achieves a test accuracy of 84.05% *on CIFAR-2, comparable to but slightly lower than the non-convex baseline of* 88.90%*. For our main experiments, we implement a convex multi-class CNN. The convex multiclass CNN achieves a test accuracy of* 28.5% *on CIFAR-10, somewhat lower but comparable to the non-convex baseline of* 36.50%*. We find that the Pytorch-based baselines for both binary and multi-class classification perform slightly to somewhat better than their convex counterparts, but the difference is not very significant. However, we note differences in built-in optimizations in Pytorch for standard non-convex methods versus the lack of such optimizations in our non-convex unconventional approach. Therefore, our results suggest that the use of convex SDPs is promising for both binary and multi-class classification tasks for image datasets like CIFAR-10. We hope to expand on this work by experimenting on deeper networks and different datasets in the future, and further deepen our understanding of the applicability of convex optimization to complex vision tasks and the mechanisms underlying deep learning algorithms broadly.*

[1]

[1]Burak Bartan, our project mentor, is a PhD student in Electrical Engineering and is not enrolled in CS231N. We have not submitted this paper to any conferences.

1. Introduction

Two layer neural networks with non-linear activation functions have broad applications in image classification. Current neural network structures generally rely on non-convex optimization, which carry a set of limitations. These limiations include the issue that internal hyperparameters, such as mini-batching, initialization, and step sizes, strongly influence the quality of the learned model (Bartan and Pilanci). Additionally, non-convex methods are only guaranteed to find local minima, not global minima.

In this work we explore the applicability of convex optimization, a more robust and reproducible technique, for image classification using neural networks. Convex optimization, considered previously for neural networks by Bengio et al. and Bach, has immense promise in deep learning – its advantages include the decoupling of optimizer parameters from the study of the neural network model as well as faster convergence to a guaranteed global solution. Prior work has developed exact convex optimization formulations for 2-layer neural networks based on semidefinite programming, and these formulations have achieved the same global optimal solution set as their non-convex versions (Bartan and Pilanci).

This study explores the use of convex programs for two-layer convolutional neural networks. We implement both a binary classification and multiclass classification model with convex optimization and obtain somewhat lower but comparable accuracies to their non-convex counterparts.

This study also helps further our understanding of the mechanisms underlying deep learning algorithms. For example, for our first experiment of binary classification, we try different polynomial activation functions, and the best choice concurs with that of previous literature, suggesting once again that polynomial activations can compare to ReLU in many cases. In addition, we also experiment with the use of the Pytorch framework for custom convex-based forward passes. These custom implementations with a vector-output convex SDP involved many design consid-

erations including how to constrain trainable parameters.

1.1. Related Work

The primary work that motivates our study is Bartan and Pilanci's 2021 paper "Neural Spectrahedra and Semidefinite Lifts: Global Convex Optimization of Polynomial Activation Neural Networks in Fully Polynomial-Time." Bartan and Pilanci achieve global optimization in fully polynomial time via convex semidefinite programming and also develop a matrix decomposition procedure called Neural Decomposition that extracts the optimal network parameters from the convex optimization solution. We build on their important work by implementing a convex SDP for two-layer convolutional neural networks with polynomial activation and average pooling for both binary classifcation and multiclass classification for CIFAR-10. In particular, for the binary classification of a subset of CIFAR-10, we use the Python-based convex optimization framework called cvxpy, and for multi-class classification, we implement the convex program using Pytorch.

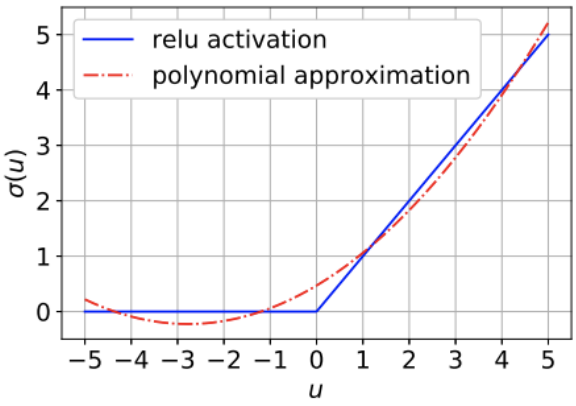
1.2. Polynomial Activation vs. ReLU

There are several types of non-linear activation functions that can be introduced to neural network layers. One of the most widely adopted ones is the ReLU (rectified linear unit) activation given by $\sigma(u) = \max(0, u)$. Another activation function is a polynomial activation: a scalar polynomial of a fixed degree. Most commonly used is are second degree polynomial activation functions, i.e., $\sigma(u) = a^2 + bu + c$, and previous work with convex program derivations have used polynomial activations that leverage convex duality and the S-procedure, and can be stated as a simple semidefinite program (SDP) (Yurtserver et. al). Several papers have explored optimization with quadratic activations, such as Mannelli et al. and Gamarnik et al., and ReLU, such as Lacotte and Pilanci and Lederer. Allen-Zhu and Li, and Ramachandran et al. find that polynomial activations are comparable to ReLU in deep networks. Researchers have studied convex formulations for ReLU neural networks (Pilanci and Ergen; Ergen and Pilanci, "Implicit"; Ergen and Pilanci, "Revealing"). Bartan and Pilanci build on these works' convex duality approach but harness a different mathematical structure to enable polynomial activations. In particular, the coefficients of $a = a = 0.09, b = 0.5, c = 0.47$ result in a second-degree polynomial closely approximating the ReLU activation function for relatively small u, as depicted in Figure 1.

1.3. Convex SDPs

The convex SDPs we use are different for the binary classification versus the multi-class classification, as the former requires a scalar output whereas the latter requires a vector output. Both are for two-layer convolutional neural networks with average pooling and polynomial activation.

Figure 1. ReLU activation: $(u) = max(0, u)$ and its polynomial approximation: $(u) = 0.09u^2 + 0.5u + 0.47$ (Bartan and Pilanci)



1.3.1 Scalar Output

We use the scalar output convex program for two-layer convolutional neural networks with polynomial activation and pooling from Bartan and Pilanci.

Let P be the pooling size, and K be the patch size. Also, let $\hat{Y}$ be a vector of the predicted scores for an image, and Y be the true score. In addition, let the trainable parameters be matrices $Z_k, Z'_k \in \mathbb{S}^{(f+1)\times(f+1)}$, for k from 1 to K/P. We will use the following notation for describing partitions of the Z matrices. $Z_{k,1}, Z'_{k,1}$ denote the first $f \times f$ partition of Z_k, Z'_k. $Z_{k,2}, Z'_{k,2}$ denote the first f elements of the last row of Z_k, Z'_k, or alternatively the first f elements of the last column (as we will see shortly, a later constraint requires these to be the same as the Z matrices are symmetric). These partitions can be visually viewed as follows:

$$Z_k = \begin{bmatrix} Z_{k,1}\,(f \times f) & Z_{k,2}\,(f \times 1) \\ {Z_{k,2}}^T\,(1 \times f) & Z_{k,4}\,(1 \times 1) \end{bmatrix},$$

and

$$Z'_k = \begin{bmatrix} Z'_{k,1}\,(f \times f) & Z'_{k,2}\,(f \times 1) \\ {Z'_{k,2}}^T\,(1 \times f) & Z'_{k,4}\,(1 \times 1), \end{bmatrix}.$$

That being said, the convex program for a two-Layer convolutional network with average pooling, polynomial activation, and with scalar output is:

$$\min_{\{Z_k = Z_k^T, Z_k'^{(t)} = Z_k'^{(t)\,T}\}} \ell(\hat{Y}, Y) + \beta \sum_{k=1}^{K/P} \left(Z_{k,4} + Z'_{k,4}\right),$$

along with the following constraints. For image i from 1 to N,

$$\hat{Y}_i = a\frac{1}{P}\sum_{k=1}^{K/P}\sum_{l=1}^{P} x_{i,(k-1)P+l}^T * \left(Z_{k,1} - Z'_{k,1}\right) x_{i,(k-1)P+l} + b\frac{1}{P}\sum_{k=1}^{K/P}\sum_{l=1}^{P} x_{i,(k-1)P+l}^T \left(Z_{k,2} - Z'_{k,2}\right) + c\sum_{k=1}^{K/P}\left(Z^-_{k,4} Z'_{k,4}\right), \quad i \in [n], t \in [C],$$

In addition, we have the following constraints:

$$\operatorname{tr}(Z_{k,1}) = Z^{\cdot}_{k,4} \operatorname{tr}\left(Z'_{k,1}\right) = Z'_{k,4} \; . \quad Z_k \succeq 0, Z'_k \succeq 0.$$

These constraints indicate that the trace, or sum of the diagonal elements of the $Z_{k,1}$ partitions add to the last diagonal element of Z_k, namely $Z_{k,4}$ and same for the corresponding prime matrices. In addition, all the matrices must be positive semidefinite. A matrix Z is positive semidefinite if it is symmetric and if the scalar $z^T M Z$ is non-negative for all non-zero real column vectors z. As part of being positive semidefinite, this also means that the matrices must be symmetric, which ensures that the partition shown earlier holds.

1.3.2 Vector output

The convex problem formulation for multi-class classification requires a vectorized output to the formulation, since there are more than two classes and therefore it must output a set of scores rather than just a single value as is sufficient for binary classification. (Bartan, Pilanci)

Let C be the number of classes. Let P be the pooling size, and K be the patch size. Also, let $\hat{Y}$ be a vector of the predicted scores for an image, and Y be the true score. In addition, let the trainable parameters be matrices $Z_k^{(t)}, Z_k'^{(t)} \in \mathbb{S}^{(f+1)\times(f+1)}$, for k from 1 to K/P and t from 1 to C. This is all similar to the scalar output but instead with additional matrices for all classes. All these Z matrices are also partitioned as described above in an analagous way between the scalar and vector SDPs. Then, for multi-class vector-output applications, the convex program for a two-Layer convolutional network with average pooling is:

$$\min_{Z_k^{(t)}, Z_k'^{(t)}} \ell(\hat{Y}, Y) + \beta \sum_{t=1}^{C}\sum_{k=1}^{K/P}\left(Z_{k,4}^{(t)} + Z_{k,4}'^{(t)}\right),$$

with three sets of constraints. The first and most important constraint is for the predicted scores, $\hat{Y}$. Let there be N total images and C classes. If $i \in [n], t \in [C]$, then the first constraint is for $\hat{Y}_{it}$, which is the predicted score for image i and class t:

$$\hat{Y}_{it} = a\frac{1}{P}\sum_{k=1}^{K/P}\sum_{l=1}^{P} x_{i,(k-1)P+l}^T * \left(Z_{k,1}^{(t)} - Z_{k,1}'^{(t)}\right) x_{i,(k-1)P+l} + b\frac{1}{P}\sum_{k=1}^{K/P}\sum_{l=1}^{P} x_{i,(k-1)P+l}^T \left(Z_{k,2}^{(t)} - Z_{k,2}'^{(t)}\right) + c\sum_{k=1}^{K/P}\left(Z_{k,4}^{(t)} - Z_{k,4}'^{(t)}\right), \quad i \in [n], t \in [C],$$

Here, x refers to all the image pixel data, with the first iterative index referring to a specific image and the second iterative index referring to the specific patch. The structure of this constraint as analogous to $au^2 + bu + c$ indicates the polynomial activation within the convex formulation. To come up with the predicted score $\hat{Y}_{it}$, the corresponding differences between Z_k^t and $Z_k'^{(t)}$ are calculated for each partition. For the quadratic component, the difference between $Z_{k,1}^t$ and $Z_{k,1}'^{(t)}$ is left-multiplied with the transpose of a patch of the image data for image i and patch $(k-1)P+l$, and right-multiplied by the same patch of image pixel data. Then, this is multiplied by a/p. Similarly, the linear component following it is the difference of $Z_{k,2}^t$ and $Z_{k,2}'^{(t)}$ left-multiplied by the corresponding patch of image pixel data. This is then multiplied by b/p. Finally, the constant component consists of the difference between $Z_{k,4}^t$ and $Z_{k,4}'^{(t)}$ multiplied by the constant coefficient c. Together, this consists of the formulation to generate the predictions.

The convex program for vector-output two-layer convolutional neural networks also has a few more constraints similar to that for scalar output, namely, for k from 1 to K/P and t from 1 to C, we have:

$$\operatorname{tr}\left(Z_{k,1}^{(t)}\right) = Z_{k,4}^{(t)}, \operatorname{tr}\left(Z_{k,1}'^{(t)}\right) = Z_{k,4}'^{(t)} \quad Z_k^{(t)} \succeq 0, Z_k'^{(t)} \succeq 0.$$

.

2. Design and Technical Approach

2.1. Design Overview

We break down our objectives into 2 milestones: for our first task, we assess the performance of binary classification with convex optimization on the CIFAR-2 dataset (first 2 classes of CIFAR-10: plane and car). For our second task, we implement a multi-class convex implementation on CIFAR-10. Our baselines are non-convex versions of these models with all relevant parameters maintained. For our first task, we use cvxpy. For our second task, we use PyTorch. The main metrics we measure are training loss, training accuracy, validation accuracy, test accuracy, and convergence time. For each of these tasks, we have a baseline implementation using standard non-convex methods using Pytorch.

2.2. Dataset Details

The CIFAR-10 dataset (Krizhevsky) is an image dataset containing 60,000 images, each size 32 by 32 by 3, for 10 different categories. For binary classification, we use 10,000 of the images as a training set with 2,000 of the images as a test set. Then, we expand to a multi-class implementation using the entire CIFAR-10 dataset. For multi-class classification, we use a randomly generated subset of 2000 images for training, and 200 each for test and validation. We scale the pixels in the images to [-1, 1] to make the training process more numerically stable.

2.3. Baseline 1: Non-Convex Binary-Class

Our first baseline is a non-convex standard implementation of classifying between the first two classes of CIFAR-10, namely aeroplanes and automobiles. We implement a two-layer convolutional neural network trained on the subsets of these two classes of CIFAR-10 constructed as described in the previous section.

The network architecture consists of a convolutional layer, an a polynomial activation layer, an average pooling layer, and finally a fully connected layer. We used parameters consistent with the corresponding experiment, such as a learning rate of 1e-3.

2.4. Experiment 1: Convex Binary-Class

For our convex implementation of binary-class classification, we again consider the task of classifying between the first two classes of CIFAR-10. This time, however, we use a convex SDP and implement it using cvxpy, a Python-based language specifically designed for solving convex optimization problems.

We split each image into patches of size $f \times f \times 3$, with the 3 being due to CIFAR-10 images being RGB 3-channel images.

We use the convex SDP as describes in Section 1.3.1, for scalar output two-layer convolutional neural networks with average pooling and polynomial activation. There is no parameter initialization for the Z matrices, as this is not an important consideration when using cvxpy. We implement each constraint using cvxpy, using a normalized $l2$ loss for the convex loss component of the minimization problem. We use $\beta = 10^{-6}$ for regularization, filter size $f = 4$, and stride = 4. We set the pooling size equal to the number of patches as well. In addition, we use $a, b, c = 0.09, 0.5, 0.47$ as the ideal coefficients for polynomial activation as this quadratic is close to ReLU. However, we experimented witha few different sets of a, b, c, specifically the three sets of $a = 0.08, b = 0.4, c = 0.4$, and $a = 0.09, b = 0.5, c = 0.47$, and $a = 0.10, b = 0.6, c = 0.5$, in order to see if there was a different polynomial activation which would perform any better.

2.5. Baseline 2: Non-Convex Multi-Class

Our first baseline is a non-convex standard implementation of classifying all ten classes of CIFAR-10. We implement a two-layer convolutional neural network trained on subsets of all ten classes of CIFAR-10, constructed as described in the previous section.

The network architecture consists of a convolutional layer, an a polynomial activation layer, an average pooling layer, and finally a fully connected layer. We used parameters consistent with the corresponding experiment, such as a learning rate of 1e-3.

2.6. Experiment 2: Convex Multi-Class

Finally, the primary experiment is a convex implementation of multi-class classification task for CIFAR-10. Here, we design a classifier using a convex formulation to classify between all ten classes of CIFAR-10, using the subset as described in the Dataset Details section. The convex SDP used is described in Section 1.3.2, for vector output two-layer convolutional neural networks with average pooling and polynomial activation.

We split each image into patches of size $f \times f \times 3$, with the 3 being due to CIFAR-10 images being RGB 3-channel images.

Our forward pass is based on the constraints as described in section 1.3.2. In particular, the first constraint is used to return the scores vector for each image sent into to the forward pass, where the scores vector consists of a C by 1 vector with a score for each class. In our case, $C = 10$, so it is a 10 by 1 vector. Since the forward pass only accepts one image at a time, the first constraint is adapted to only generate Y_{it} for the corresponding image i that is being sent in.

There are $2 \times c \times \frac{K}{P}$ different Z matrices, together constituting the trainable parameters for the SDP based network. Each matrix is size $3 * f^2 + 1$, because they are multiplied by the patches of size $3f^2$, so Z_1 matrices are size $3f^2 \times 3f^2$, Z_2 matrices are size $3f^2 \times 1$, and Z_4 are scalars when partitioning the Z matrices. Initializing the matrix parameters with these dimensions ensures that the matrix multiplication in the forward pass works out.

For parameter initialization, we used an Xavier uniform distribution. Usually, parameter initialization is an issue of minimal significance for convex formulations, but as our design relies on a Pytorch-based forward pass with insufficient optimizations and computation constraints, the initialization turns into a relevant factor. All the weights for the Z matrix parameters are initialized using the Xavier algorithm with a uniform distribution. Parameters are set uniformly to values from $[-a, a]$ where $a = \sqrt{\frac{6}{\text{fan\_in}+\text{fan\_out}}}$, where fan_in is the number of inputs to the hidden unit parameter and fan_out is the number of outputs.

In addition, $a = 0.09, b = 0.5, c = 0.47$ are the coefficients for the polynomial activation in the forward pass constraint, as these coefficients closely approximate the behavior of ReLU as shown in section 1.2.

Then, for the other constraints regarding trace and positive semidefiniteness, they are not as necessary but do affect the ability of convergence for the convex SDP. Enforcing these constraints on trainable parameters in Pytorch is tricky due to the nature of Pytorch parameters, but we implemented a design choice to force the Z matrices to be either symmetric, positive semidefinite, or otherwise.

Let $Z = \begin{bmatrix} Z_1 & Z_2 \\ Z_2^T & Z_4 \end{bmatrix}$ be any $3f^2 + 1 \times 3f^2 + 1$ matrix at the start of any iteration of the forward pass. Then, there is a mathematical result that shows that the product of any matrix and its transpose is positive semidefinite. Therefore, we have that

$$Z^* = ZZ^T = \begin{bmatrix} Z_1 Z_1^T + Z_2 Z_2^T & Z_1 Z_2 + Z_2 Z_4 \\ Z_2^T Z_1^T + Z_4 Z_2^T & Z_2^T Z_2 + Z_4 Z_4^T \end{bmatrix}$$

is a semidefinite matrix. Therefore at the beginning of every forward pass, we can apply this transformation to all Z parameters to ensure that they are positive semidefinite. Alternatively, if we just want them to be symmetric, we can apply the following transformation:

$$Z^* = 0.5 * (Z + Z^T)$$

, which ensures not only that Z^* is symmetric but also that the total sum of all its components stays consistent, something that our positive-semidefinite transformation does not do.

However, due to intractability due to our computational limitations on our devices, we were unable to train for more than a few epochs with the forced positive semidefinite constraints. Therefore, we only forced the matrices to be symmetric, and made a design concession by sacrificing some of our model convergence potential for efficiency.

Our loss function is also described by the convex SDP in section 1.3.2, as

$$\min_{Z_k^{(t)}, Z_k'^{(t)}} \ell(\hat{Y}, Y) + \beta \sum_{t=1}^{C} \sum_{k=1}^{K/P} \left(Z_{k,4}^{(t)} + Z_{k,4}'^{(t)} \right).$$

Here, ℓ refers to any convex loss function, and we chose normalized $l2$ loss, comparing $\hat{y}$ with a true value vector y which contains the actual class for each image. For the regularization coefficient β, we chose a small value of 10^{-7}.

For simplicity, we chose our pooling size P to be equal to the patch size K, which is determined through Pytorch when we create the patches of size $3 * f^2$. We ran our experiment with three different learning rates, 10^{-4}, 10^{-3}, and 10^{-2}. We also used Adam with AMSGrad as an optimizer for learning. Adam is an algorithm that uses both momentum and scaling to improve learning, and AMSGrad adjusts Adam by updating parameters via the maximum of past squared gradients (Reddy, Kale, et al.)

We trained the model for up to 10 epochs, keeping track of training and validation accuracies as well as training loss.

3. Results

Experiments	Test Accuracy
Baseline 1	88.90%
Experiment 1	84.05%
Baseline 2	36.50%
Experiment 2	28.55%

3.1. Baseline 1: Non-Convex Binary-Class

Figure 2. Training Loss for Baseline 1

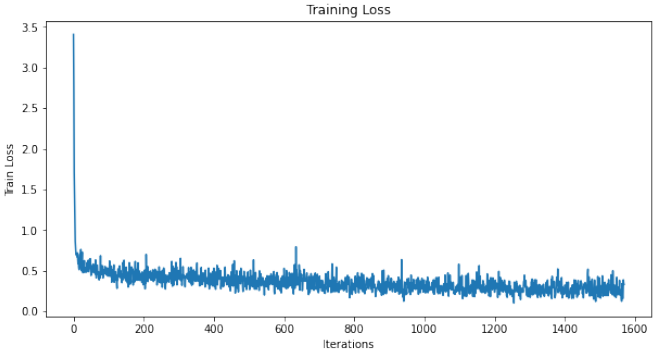
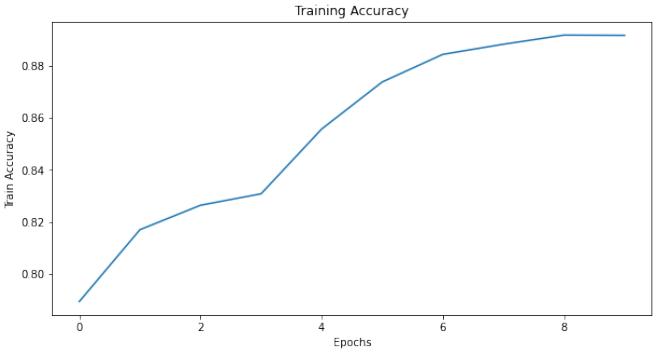


Figure 3. Training Accuracy for Baseline 1

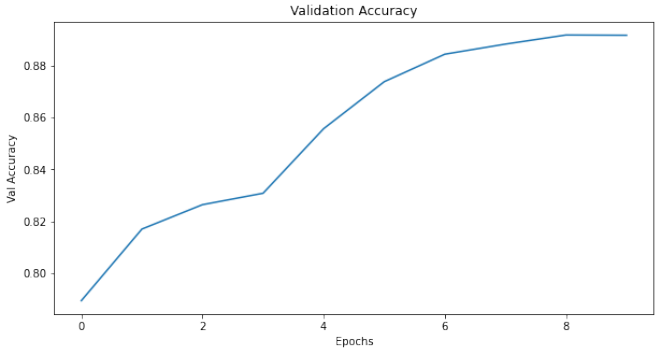


3.2. Experiment 1: Convex Binary-Class

a	b	c	iter	training acc	test acc
0.08	0.4	0.4	21000	0.8419	0.8
0.09	0.5	0.47	21000	0.8416	0.8405
0.1	0.6	0.5	20000	0.8415	0.84

a	b	c	iters	training cost	test cost
0.08	0.4	0.4	21000	645.671	128.473
0.09	0.5	0.47	21000	645.831	128.523
0.1	0.6	0.5	21000	646.126	128.579

Figure 4. Validation Accuracy for Baseline 1



Comparing Experiment 1 to Baseline 1 (see Figures 2-4), we see that the training accuracy of the baseline (greater than 88%) outperforms that of Experiment 1 (84%). The test accuracy of the baseline (88.90%) also is higher than the best test accuracy of Experiment 1 (84.05%).

3.3. Baseline 2: Non-Convex Multi-Class

Figure 5. Training Loss for Baseline 2

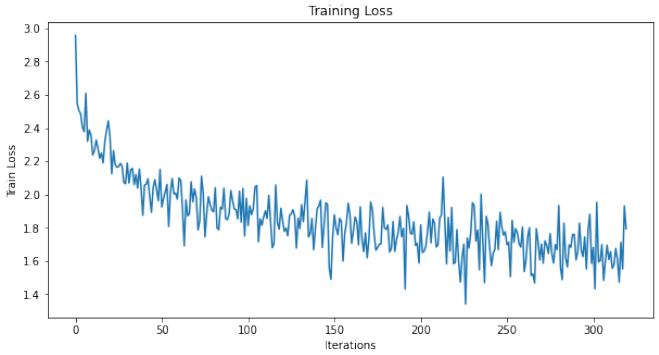


Figure 6. Training Time for Baseline 2

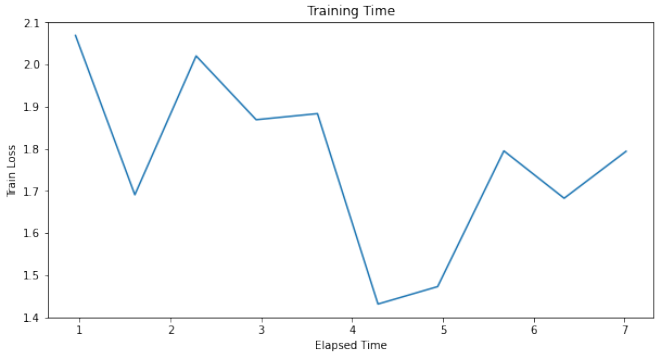


Figure 7. Training Accuracy for Baseline 2

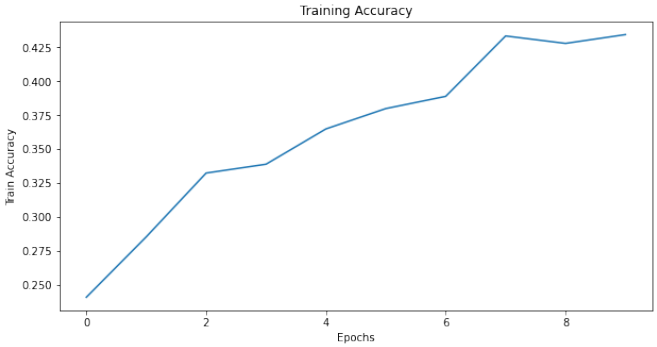


Figure 8. Validation Accuracy for Baseline 2

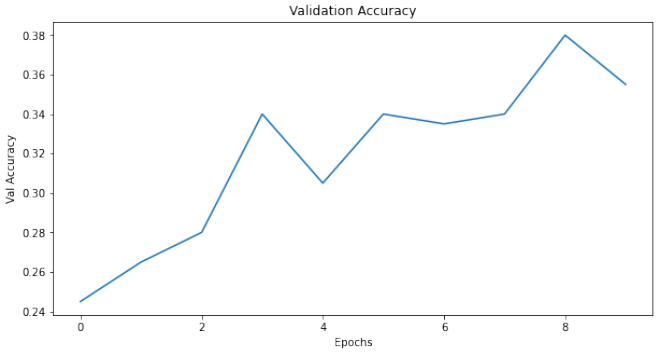


Figure 9. Training Loss for Experiment 2

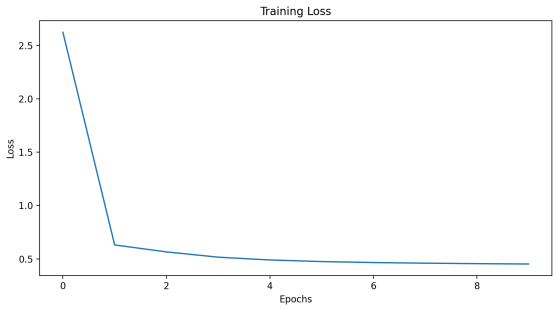
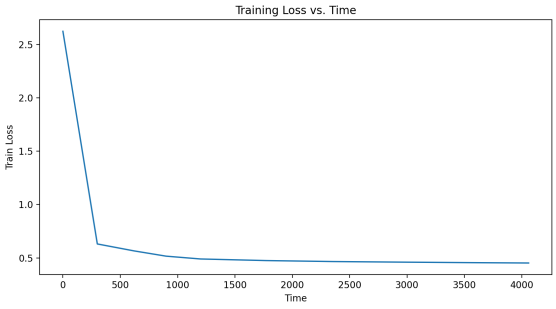


Figure 10. Training Time for Experiment 2



3.4. Experiment 2: Convex Multi-Class

Comparing Baseline 2 to Experiment 2 reveals two key findings. Firstly, the convergence time of the convex model is not comparable to that of the baseline, likely due to differences in built-in optimizations in Pytorch for standard

Figure 11. Training Accuracy for Experiment 2

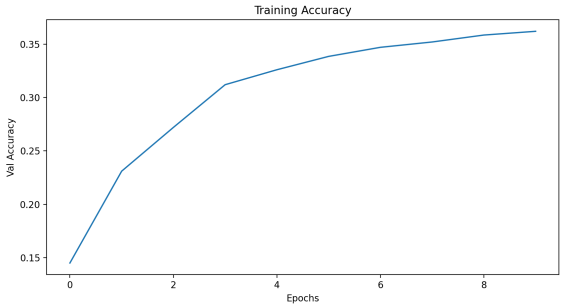
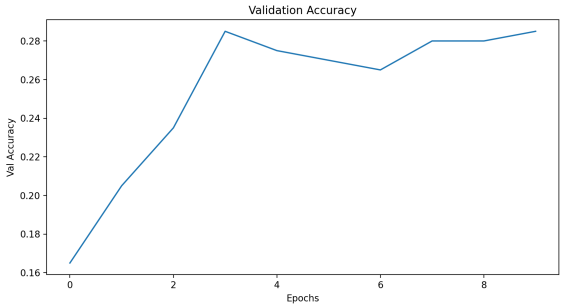


Figure 12. Validation Accuracy for Experiment 2



non-convex methods versus the lack of such optimizations in our non-convex unconventional approach (Figures 5, 6, 9, 10). Secondly, the baseline surpasses Experiment 2 in both training accuracy (Figures 7, 11) and validation accuracy (Figures 8, 12). We see the baseline test accuracy of 36.5% surpass that of Experiment 2 (28.55 %).

3.4.1 Hyperparameter Tuning

Our hyperparameter tuning experiments (see Figures 13-16) show that a learning rate of 1e-3 yields the highest validation accuracy out of the three learning rates tested: 1e-2, 1e-3, and 1e-4.

Figure 13. Training Loss, lr = 1e-2

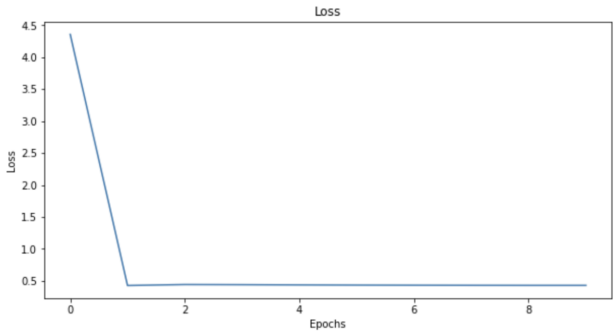


Figure 14. Validation Accuracy, lr = 1e-2

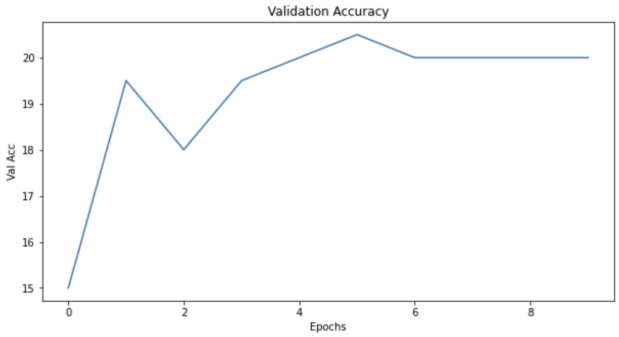


Figure 15. Training Loss, lr = 1e-4

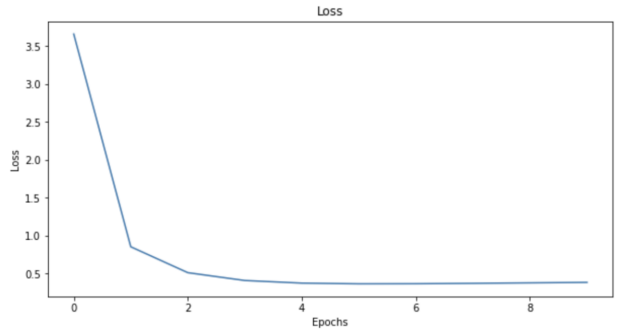
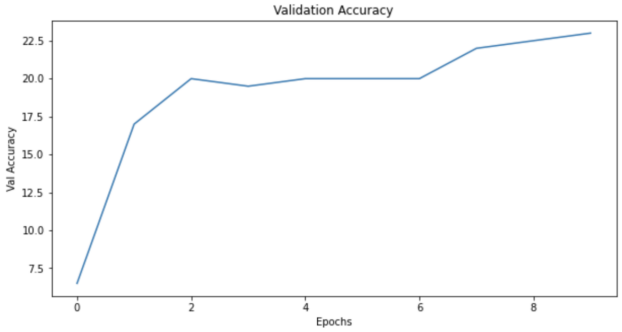


Figure 16. Validation Accuracy, lr = 1e-4



3.5. Limitations of Design Method

A major limitation that we had in our design method was due to Pytorch usually being equipped to deal with non-convex neural networks. Therefore, for our custom code with convex SDPs, the necessary optimizations were likely inadequate, therefore the training was slow. In addition, we ran our experiments on a CPU, so intractability was an issue and we could not train on a larger subset of CIFAR-10 or many epochs. We also were unable to include every single constraint for the vector-output convex SDP for two-layer convolutional neural networks. In particular, we primarily focused on the first constraint, and wrote implemen-

tations for the positive semidefinite constraint, but ended up only constraining the parameters to be symmetric due to intractability due to lack of sufficient optimizations in our positive semidefinite matrix calculations and our compute power. In addition, we did not include the constraints involving trace of the matrices. In the end, the training time being very high for our custom convex implementations resulted in convergence time also being high. If we were to explore further optimizations of the calculations in our forward pass based on the first constraint of the SDP, that would likely improve convergence time as well as accuracy.

Another important consideration relating to comparing our non-convex baselines to our convex experiments is that even with cvxpy, which is a framework specifically designed for convex optimization tasks, the accuracy for binary classification was lower than that for non-convex via Pytorch. Although we made efforts to control all the variables we could such as filter size and pooling size and so on, this must indicate that there are some non-controls which are positively skewing our baseline accuracies in comparison to our experiments, as the cvxpy problem solver should technically reach an optimal minimum of loss and consequently a equal or greater accuracy than our non-convex method. This potential discrepancy in our design comparisons may also contribute to the somewhat lower accuracy for non-convex multi-class classification between experiment 2. and baseline 2. Spending more effort in resolving all controls between the corresponding baselines and experiments and ensuring that the convex programs are optimized effectively may resolve this discrepancy and improve the efficacy of the convex versions.

4. Conclusion and Future Work

Our work explores the potential of convex optimization for image classification using neural networks. We implement convex binary and multiclass classification models and achieve accuracies somewhat lower but comparable to the baselines. While more work needs to be done to more fully formulate convex optimization as a promising technique for deep learning, our work contributes to the conversation around the promise of convex methods. Future steps for our project are to experiment with layerwise learning with convex optimization, building off the work of Allen-Zhu and Li, and to work with deeper architectures, drawing from studies like Ergen and Pilanci and Belilovsky et al. We also hope to work with different datasets to investigate generalizability and robustness.

5. Contributions and Acknowledgements

All three of us worked on the two baselines and two experiments together, as well as wrote the report and prepared the final presentation.

We would like to thank Dr. Burak Bartan, who was our primary mentor on this project. With his previous work on convex optimization for convolutional neural networks, Bartan provided us with starter code for implementing a convex SDP for a two-layer convolutional network for binary classification using cvxpy for CIFAR-10. In addition, Bartan helped us with resolving design-related issues for the convex multi-class Pytorch-based implementation. We relied on the results of his paper "Neural Spectrahedra and Semidefinite Lifts" paper to develop the convex formulation for both scalar-output and vector-output two-layer convolutional neural network with global average pooling.

Additionally, we would like to acknowledge the CS231N teaching team at Stanford for providing us with resources to complete our project. The teaching team has particularly helped us with providing us with feedback regarding our project and shaping the scope of our goals.
[2]

References

Bartan, B., & Pilanci, M. (2021). Neural Spectrahedra and Semidefinite Lifts: Global Convex Optimization of Polynomial Activation Neural Networks in Fully Polynomial-Time. arXiv preprint arXiv:2101.02429.

Bengio, Y., Roux, N. L., Vincent, P., Delalleau, O., and Marcotte, P. Convex neural networks. In Advances in neural information processing systems, pp. 123–130, 2006.

Bach, F. Breaking the curse of dimensionality with convex neural networks. The Journal of Machine Learning Research, 18(1):629–681, 2017.

Stefano Sarao Mannelli, Eric Vanden-Eijnden, and Lenka Zdeborov´a. Optimization and generalization of shallow neural networks with quadratic activation functions. arXiv preprint arXiv:2006.15459, 2020.

David Gamarnik, Eren C. Kızılda˘g, and Ilias Zadik. Stationary points of shallow neural networks with quadratic activation function. arXiv preprint arXiv:1912.01599, 2020.

Jonathan Lacotte and Mert Pilanci. All local minima are global for two-layer relu neural networks: The hidden convex optimization landscape. arXiv preprint arXiv:2006.05900, 2020.

Yurtsever, A., Tropp, J. A., Fercoq, O., Udell, M., Cevher, V. (2021). Scalable semidefinite programming. SIAM Journal on Mathematics of Data Science, 3(1), 171-200.

Johannes Lederer. No spurious local minima: on the optimization landscapes of wide and deep neural networks, 2020.

Zeyuan Allen-Zhu and Yuanzhi Li. Backward feature correction: How deep learning performs deep learning. arXiv preprint arXiv:2001.04413, 2020.

Prajit Ramachandran, Barret Zoph, and Quoc Le. Searching for activation functions. arXiv preprint arXiv:1710.05941, 2018.

Mert Pilanci and Tolga Ergen. Neural networks are convex regularizers: Exact polynomialtime convex optimization formu-

[2]Citation for starter code: Bartan, Burak. convexNN_poly_act_fullyconnected_v2 and convexNN_convolutional_binaryclass.

lations for two-layer networks. Proceedings of the International Conference on Machine Learning (ICML 2020), 2020.

Tolga Ergen and Mert Pilanci. Implicit convex regularizers of cnn architectures: Convex optimization of two- and three-layer networks in polynomial time. arXiv preprint arXiv:2006.14798, 2020.

Tolga Ergen and Mert Pilanci. Revealing the structure of deep neural networks via convex duality. arXiv preprint arXiv:2002.09773, 2020.

Belilovsky, E., Eickenberg, M., Oyallon, E. (2019, May). Greedy layerwise learning can scale to imagenet. In International conference on machine learning (pp. 583-593). PMLR.

Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009

Sashank J. Reddi and Satyen Kale and Sanjiv Kumar. On the Convergence of Adam and Beyond. arXiv: 1904.09237, 2019

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
#!pip install matplotlib
import matplotlib.pyplot as plt
import time
print(torch.__version__)
#!pip install torch===1.7.0 torchvision===0.8.1 torchaudio===0.7.0 -f https://downloac
import cv2
print(torch.__version__)

import torchvision.datasets as dset
import torchvision.transforms as T
from keras.datasets import cifar10
import numpy as np, numpy.linalg

USE_GPU = True
dtype = torch.float64 # We will be using float throughout this tutorial.

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss.
print_every = 100
print('using device:', device)
```

```
 1.8.1+cu101
 1.8.1+cu101
 using device: cpu
```

```
#class for forward pass and loss

class custom_multiclass_torch(torch.nn.Module):
    def __init__(self, N, P, C, K, f,a,b,c,beta, x_patches_train, x_patches_val, x_pat
        super(custom_multiclass_torch, self).__init__()


        #initialize several variables
        # parameters, N = number of images, K = number of patches, P = pooling size, C

        self.N = N
        self.P = P
        self.C = C
        self.K = K
        self.f = f
```

```
        self.a = a
        self.b = b
        self.c = c
        self.beta = beta
        self.x_patches_train = x_patches_train
        self.x_patches_val = x_patches_val
        self.x_patches_test = x_patches_test


        #initialize tensor array variables as dictionaries

        self.Z_1_arr_train = nn.ParameterDict({  })
        self.Z_1_arr_prime_train = nn.ParameterDict({})
        self.Z_2_arr_train = nn.ParameterDict({})
        self.Z_2_arr_prime_train = nn.ParameterDict({})
        self.Z_4_arr_train = nn.ParameterDict({})
        self.Z_4_arr_prime_train = nn.ParameterDict({})
        self.Z_arr_train = {}
        self.Z_arr_prime_train = {}

        # fill in dictionaries with Z1 Z2 Z4 tensor parameters
        for i in range(1,self.K//self.P+1):
            for j in range(1,self.C+1):


                self.Z_1_arr_train[str(i)+','+str(j)] =  torch.nn.Parameter(data = tor
                self.Z_1_arr_prime_train[str(i)+','+str(j)] =  torch.nn.Parameter(data
                self.Z_2_arr_train[str(i)+','+str(j)] =  torch.nn.Parameter(data = tor
                self.Z_2_arr_prime_train[str(i)+','+str(j)] =  torch.nn.Parameter(data
                self.Z_4_arr_train[str(i)+','+str(j)] =  torch.nn.Parameter(data = tor
                self.Z_4_arr_prime_train[str(i)+','+str(j)] =  torch.nn.Parameter(data

                nn.init.xavier_uniform_(self.Z_1_arr_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_2_arr_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_4_arr_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_1_arr_prime_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_2_arr_prime_train[str(i)+','+str(j)])
                nn.init.xavier_uniform_(self.Z_4_arr_prime_train[str(i)+','+str(j)])







                self.Z_arr_train[str(i)+','+str(j)] = torch.vstack((torch.hstack((self
                self.Z_arr_prime_train[str(i)+','+str(j)] = torch.vstack((torch.hstack

    def forward(self, i, setting = "train"):

        Z_1_new = {}
        Z_2_new = {}
        Z_4_new = {}
```

```
Z_4_new = {}
Z_1_prime_new = {}
Z_2_prime_new = {}
Z_4_prime_new = {}
Z_new = {}
Z_new_prime = {}

if setting == "train":
    xdata = self.x_patches_train
elif setting == "val":
    xdata = self.x_patches_val
else:
    xdata = self.x_patches_test

#transform Z matrices into positve-semidefinite matrices

# #convertTo = "positive semidefinite"
convertTo = "other"
for key in self.Z_arr_train:
    if convertTo == "positive semidefinite":
        Z_new[key] = torch.matmul(self.Z_arr_train[key], self.Z_arr_train[key]
    elif convertTo == "symmetric:":
        Z_new[key] = 0.5 * (self.Z_arr_train[key]+self.Z_arr_train[key].T)
    else:
        Z_new[key] = self.Z_arr_train[key]
    Z_1_new[key] = Z_new[key][:3*f**2,:3*f**2]
    Z_2_new[key] = Z_new[key][3*f**2,:3*f**2]
    Z_4_new[key] = Z_new[key][3*f**2,3*f**2]

for key in self.Z_arr_prime_train:
    if convertTo == "positive semidefinite":
        Z_new_prime[key] = torch.matmul(self.Z_arr_prime_train[key], self.Z_ar
    elif convertTo == "symmetric":
        Z_new_prime[key] = 0.5 * (self.Z_arr_prime_train[key] + self.Z_arr_pri
    else:
        Z_new_prime[key] = self.Z_arr_prime_train[key]
    Z_1_prime_new[key] = Z_new_prime[key][:3*f**2,:3*f**2]
    Z_2_prime_new[key] = Z_new_prime[key][3*f**2,:3*f**2]
    Z_4_prime_new[key] = Z_new_prime[key][3*f**2,3*f**2]


#print(self.Z_1_arr_train[1,2])
#print(self.Z_4_arr_train[1,2])


ypred = torch.zeros((C))


# performing calculations for ypred scores
for t in range(1,self.C+1):

    constant_part = 0
```

```
            for k in range(1,self.K//self.P+1):
                Z4diff = self.Z_4_arr_train[str(k)+","+str(t)] - self.Z_4_arr_prime_tr
                constant_part += Z4diff
            constant_part *= self.c
            #print(constant_part)

            linear_part = 0
            for k in range(1,self.K//self.P+1):
                for l in range(1,self.P+1):
                    Z2diff = self.Z_2_arr_train[str(k)+","+str(t)] - self.Z_2_arr_prim
                    x_data_reshaped = xdata[i][(k-1)*P+ l-1].view(-1, 1)
                    linear_part += torch.matmul(x_data_reshaped.T, Z2diff)

                    #linear_part += float(torch.matmul(x_data_reshaped.T, Z2diff)[0])
            linear_part *= self.b/self.P

            quadratic_part = 0
            for k in range(1,self.K//self.P+1):
                for l in range(1,self.P+1):
                    Z1diff = self.Z_1_arr_train[str(k)+","+str(t)] - self.Z_1_arr_prim
                    x_data_reshaped = xdata[i][(k-1)*P+ l-1].view(-1, 1)
                    newpart = torch.matmul(x_data_reshaped.T, Z1diff)
                    newpart = torch.matmul(newpart, x_data_reshaped)
                    quadratic_part += newpart

                    #quadratic_part += float(newpart[0,0])
            quadratic_part *= self.a/self.P

            #print(quadratic_part, linear_part, constant_part)

            ypred[t-1] = quadratic_part + linear_part + constant_part

            #randnum = np.random.randint(10)
            #ypred *= randnum
        return ypred



    def customloss(self, Yhat, y):
        #convex L2 loss function
        objective1 = 0.5 * torch.norm(Yhat - y)**2


        # sum of Z4 scalars added to loss
        objective2 = 0

        for i in range(1,K//P+1):
            for j in range(1,C+1):
                objective2 += self.Z_4_arr_train[str(i)+","+str(j)] + self.Z_4_arr_pri

        objective = objective1+ self.beta * objective2
```

```
        return objective
```

```
#load cifar data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# shuffler = np.random.permutation(len(y_train))
# X_train = X_train[shuffler]
# y_train = y_train[shuffler]
# shuffler2 = np.random.permutation(len(y_test))
# X_test = X_test[shuffler2]
# y_test = y_test[shuffler2]

#subset data
X_train = X_train[:2000,:,:,:]
y_train = y_train[:2000]
X_test = X_test[:400,:,:,:]
y_test = y_test[:400]

#set up val/test split
X_val = X_test[:X_test.shape[0]//2,:,:,:]
X_test = X_test[X_test.shape[0]//2:,:,:,:]

y_val = y_test[:y_test.shape[0]//2]
y_test = y_test[:y_test.shape[0]//2]


#parameters: f is filter size, P is pooling size, C is class count, a,b,c are the poly

f = 4



train_images = X_train.astype(np.float64)
train_labels = y_train.astype(np.float64)
test_images = X_test.astype(np.float64)
test_labels = y_test.astype(np.float64)
val_images = X_val.astype(np.float64)
val_labels = y_val.astype(np.float64)
```

```
# #meaning out the images
# mean_image = np.mean(train_images, axis = 0)
# train_images -= mean_image
# test_images -= mean_image
# val_images -= mean_image

# # RGB 0 to 255
# #-127.5 to +127.5
# #-1 to 1

# # scale to [-1, 1]

# train_images /= 127.5
# test_images /= 127.5
# val_images /= 127.5


train_images /= 255
train_images *= 2
train_images -= 1

test_images /= 255
test_images *= 2
test_images -= 1

val_images /= 255
val_images *= 2
val_images -= 1

train_images_v2 = np.swapaxes(train_images.reshape(train_images.shape[0], 3, 32, 32),
test_images_v2 = np.swapaxes(test_images.reshape(test_images.shape[0], 3, 32, 32), 2,
val_images_v2 = np.swapaxes(val_images.reshape(val_images.shape[0],3, 32, 32), 2, 3)

train_images_v3 = torch.tensor(train_images_v2, device = device, dtype = dtype)
test_images_v3 = torch.tensor(test_images_v2, device = device, dtype = dtype)
val_images_v3 = torch.tensor(val_images_v2, device = device, dtype = dtype)

#set s_val = torch.nn.functional.unfold(val_images_v3, kernel_size=(f,f), stride=f, pa
#setting up patches
patches_train = torch.nn.functional.unfold(train_images_v3, kernel_size=(f,f), stride=
patches_test = torch.nn.functional.unfold(test_images_v3, kernel_size=(f, f), stride=1
patches_val = torch.nn.functional.unfold(val_images_v3, kernel_size=(f,f), stride=f, p



patches_train = patches_train.permute(0,2,1)
patches_test = patches_test.permute(0,2,1)
patches_val = patches_val.permute(0,2,1)

N = X_train.shape[0]
```

```
K = patches_train.shape[1]

Yhat_train =  None
Yhat_test = None

P = K
C = 10
a=0.09
b=0.5
c=0.47
beta = 1e-7

print("K: ", K)
print("N: ", N)
print("patches_train size: ", patches_train.size())
print("patches_val size: ", patches_val.size())
print("patches_test size: ", patches_test.size())
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 4s 0us/step
170508288/170498071 [==============================] - 4s 0us/step
K:  64
N:  2000
patches_train size:  torch.Size([2000, 64, 48])
patches_val size:  torch.Size([200, 64, 48])
patches_test size:  torch.Size([200, 64, 48])
```

```
def train(model, loss_fn, optimizer, epochs=1):
    model = model.to(device=device) # move the model parameters to CPU/GPU

    loss_train = []
    accuracies_train = []
    accuracies_val = []
    start = time.time()
    times = []
    times.append(time.time)
    zzz = 0
    #accuracies_train.append(check_accuracy(X_train, y_train, model, segment = "train'

    for e in range(epochs):
        #idx = torch.randperm(train_images.shape[0])
        #sample_images = train_images[idx].view(train_images.size())
        #sample_images = sample_images[]
        for t, (x, y) in enumerate(zip(train_images, train_labels)):
            model.train() # put model to training mode

            scores = model(t, setting = "train")
```

```
            y_hot = torch.zeros(scores.size(), dtype = dtype)

            y_hot[y] = 1


            #print("Scores before: ", scores)
            loss = loss_fn(scores, y_hot)
            loss_train.append(loss)

            # Zero out all of the gradients for the variables which the optimizer
            # will update.
            optimizer.zero_grad()

            # backwards pass
            loss.backward()

            # update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()
            #print("Scores after: ", scores)
            #if t % 100 == 0:
            if t==0:
              times.append(time.time)
              print('Iteration %d, loss = %.4f' % (t, loss.item()))
              accuracies_train.append(check_accuracy(X_train, y_train, model, segment
              accuracies_val.append(check_accuracy(X_val, y_val, model, segment = "val


         # times.append(time.time)
          #print('Iteration %d, loss = %.4f' % (t, loss.item()))
          #accuracies_train.append(check_accuracy(X_train, y_train, model, segment = "tr

    for i in range(len(times)):
        times[i] -= start


    return loss_train, accuracies_train, accuracies_val, times
```

```
def check_accuracy(X, Y, model, segment = "train"):
    if segment=='train':
        print('Checking accuracy on train set')
    elif segment == "val":
        print('Checking accuracy on val set')
    else:
        print('Checking accuracy on test set')
```

```
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for t, (x, y) in enumerate(zip(X, Y)):
            scores = model(t, setting = segment)
            max_score_idx = torch.argmax(scores)
            y = torch.tensor(y)
            #if t % 1000 == 0:
              #print(t, scores, y)
              #for name, param in model.named_parameters():
               #   if param.requires_grad:
                #      if name == "Z_2_arr_train.1,1" or name == "Z_2_arr_prime_train.
                 #       print(name, param.data.T)
                      #break


            addvalue = 1 if (max_score_idx == y) else 0
            num_correct += addvalue
            num_samples += 1

        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)'% (num_correct, num_samples, 100 * acc))
        return acc


model = custom_multiclass_torch(N, P, C, K, f,a,b,c,beta, patches_train, patches_val,




count = 0
for param in model.parameters():
    #print(param.shape)
    count += 1
print(count)
```

```
60
```

```
loss_fn = model.customloss

optimizer = optim.Adam(model.parameters(), lr=1e-3, amsgrad=True)

#optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

loss_train, accuracies_train, accuracies_val, times = train(model, loss_fn, optimizer,
```

```
Iteration 0, loss = 2.1351
Checking accuracy on train set
Got 220 / 2000 correct (11.00)
Checking accuracy on val set
```

```
Got 22 / 200 correct (11.00)
Iteration 0, loss = 0.5060
Checking accuracy on train set
Got 486 / 2000 correct (24.30)
Checking accuracy on val set
Got 44 / 200 correct (22.00)
```

```
plt.figure(figsize=(10,5))
plt.title('Training Accuracy')
#print(accuracies_train)
plt.plot(accuracies_train)
plt.xlabel('Epochs')
plt.ylabel('Val Accuracy')
#plt.legend()
plt.show()
```

```
plt.figure(figsize=(10,5))
plt.title('Validation Accuracy')
#print(accuracies_val)
plt.plot(accuracies_val)
plt.xlabel('Epochs')
plt.ylabel('Val Accuracy')
#plt.legend()
plt.show()
```

```
plt.figure(figsize=(10,5))
plt.title('Training Loss')
#print(loss_train)
plt.plot(loss_train)
plt.xlabel('Epochs')
plt.ylabel('Train Loss')
#plt.legend()
plt.show()
```

```
plt.figure(figsize=(10,5))
plt.title('Training Loss')
#print(loss_train)
plt.plot(loss_train)
plt.xlabel('Epochs')
plt.ylabel('Train Loss')
#plt.legend()
plt.show()
```

```
plt.figure(figsize=(10,5))
plt.title('Training Time')
#print(loss_train)
```

```
#print(loss_train)
plt.plot(times)
plt.xlabel('Epochs')
plt.ylabel('Time (seconds)')
#plt.legend()
plt.show()


check_accuracy(X_test, y_test, model, segment = "test")
```