An Exploration of Current and Novel Deep Learning Approaches to Small Sample Learning for Computer Vision – With and Without External Data

Benjamin Therien [*] **Nigel Yong** [*]

Abstract

We explore two different approaches to each challenge in depth. For challenge 1 we find that augmenting our dataset with Auto Augment and using Wide Residual Networks elevates our performance above the baseline. We also explore image generation to further augment our training set and find that this further boosts our performance (35% accuracy + on challenge 1). We explore this generative procedure further and show evidence of its regularizing effects. Our approaches to challenge two test meta learning and transfer learning. Out of the settings we tried, transfer learning works best and we obtain accuracy scores above 65%.

1. Introduction

In the context of deep learning, small sample learning is the challenge of learning useful representations from a small dataset. [3] Historically, computer vision researchers used various handcrafted feature extraction techniques, notably SIFT[17], to extract meaningful features from images. In 2012, the Alex Net [13] breakthrough on the Imagenet challenge marked a paradigm shift for computer vision researchers. Instead of using handcrafted feature extractors, Alex net learns the parameters of superior convolutional filters during training. However, these deep learning architectures are not without their drawbacks. Imagenet is a dataset of over 1.2[13] million training images. On smaller datasets, learning useful convolutional feature extractors is challenging. As part of a project for the Deep Learning course at Concordia University, the following work explores different deep learning approaches to two restricted classification settings. Although handcrafted feature descriptors may have been viable approaches to the first challenge, we choose not to asses their performance and, instead, focus only on deep learning approaches.

[*]Equal contribution .
Benjamin Therien <benjtherien@gmail.com>,
Nigel Yong Sao Young <kenkenysy@gmail.com>

2. Project

The project description proposes that we train our models on a random class balanced 100-sample subset of the CIFAR-10 training dataset and evaluate them on a 2000-sample subset. [12]

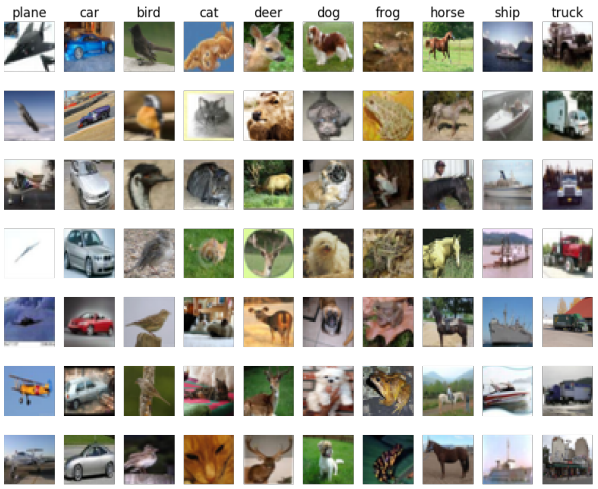

Figure 1. CIFAR-10 Classes

Here is a look of the CIFAR-10 classes and images.

After evaluating different approaches, we are asked to benchmark our best performing one in the same data arrangement, but using the CIFAR-10 test set instead.

- **Challenge 1: Learning with Limited data without external data**

  This challenge entails exclusively using the 100 training samples. Since no external data is permitted, we will explore different model architectures and different data augmentation techniques in the search of those best suited for the task. More specifically, we will focus on two approaches: traditional augmentation

and data augmentation through generative models.

- **Challenge 2: Learning with Limited data with External Data**

  This challenge is more permissive. In addition to the 100 samples, we can use any external data or pre-trained models, as long as they were not trained on or otherwise benefited from prior information involving CIFAR-10 samples other than the 100 prescribed. We consider various forms of finetuning, meta-learning, semi-supervised learning with an external dataset and other related ideas.

3. Conv Net Architectures

Since their introduction in the late nineties [14], convolutional neural networks have been building in popularity and evolving architectures as a consequence. Some these new architectures include experimenting with depth [21], introducing residual connections for more depth [8], combining the effects of depth and width [29], parameter efficient networks [11], and a series of more recent works that tries to optimally combine all these settings [24] [25]. We use many of the aforementioned architectures in the following work.

4. Literature review

4.1. AutoAugment

Data augmentation is very popular nowadays, especially when data is scarce, it provides the variance the model needs in training for a better generalization. Since we are working with only 100 training samples from CIFAR 10, it was pretty obvious that we needed to use data augmentation. Auto Augment is the state-of-the-art when it comes to this. [5]

Instead of the traditional manual augmentation, AutoAugment automates this procedure by automatically searching for the best augmentation policies for the task. It defines a search space where a policy contains multiple sub-policies, one of which is randomly chosen for each image in each mini-batch. A sub-policy consists of two operations, one being an image processing function such as invert, rotation or translation and the second one being the probability and magnitude to which the function is being applied. The search algorithm then tries out the different combinations of 5 sub-policies and finds the best policy: the one yielding the highest validation accuracy. It follows a reinforcement learning approach.

It has been observed that the augmentation policies are transferable between datasets: augmentation policies learned on ImageNet transfer well to other datasets such as Oxford Flowers and Stanford Cars, among others. [6] It is perfect for the challenge. In training the 100 images, we would get the best augmentation for more generalization, which is exactly what we wanted. Similarly, the paper trained on a smaller subset of CIFAR-10 but with 4000 random samples. We are taking it up a notch with 100 samples only.

An improvement to AutoAugment is Fast AutoAugment, which uses a more efficient search algorithm and uses fewer GPU resources. [15] We decided not to use it since the original one was already training fast and memory was not an issue due to the limited amount of training samples we had. We used the PyTorch implementation, found on GitHub. [1]

4.2. Generative models for data augmentation

In challenge 1's restricted setting, it is appealing to further augment the training set by generating new samples. However, popular generative models [7][19][26] require large training sets to obtain realistic results, but a recent alternative shows promise.

Generative Latent Optimization Generative latent optimization [4] is a generative model which is trained by jointly optimizing the latent vectors and the decoder model's parameters with respect to a reconstruction loss. Unlike the random noise vector that is fed to GAN, GLO assigns one latent vector to each image in the training set. It uses a DCGAN architecture to reconstruct an image from its latent vector. A reconstructed image is fed to the laplacian pyramid[16] loss 1[1]. Although the authors optimize their latent vectors, they constrain them to lie on the unit circle, by taking their projection onto the unit $\updownarrow_2$ sphere after parameters updates. The reconstruction results of latent vectors are on par if not better than those of other generative models. However, sampled images from the GLO model are not as good.

$$Lap(x, x') = \Sigma_j 2^{2j} |L^j(x) - L(x')|_1 \tag{1}$$

Building upon the work of GLO's authors, researchers propose Generative Latent Implicit Conditional Optimization[2] (GLICO), a framework to augment small datasets with images generated similarly to GLO. GLICO's authors point out the sampling flaws of the GLO model and attribute them manly to the sparsity of the latent space. GLICO's propose to mitigate this problem by adding a weak classifier of the generated images which feeds the latent vectors and the generator model's parameters error from a cross-entropy loss. They show that the addition of this classifier loss does indeed bring latent space vectors closer together, but does not fix the problem of meaningless

[1] taken from [4]

samples produced from random latent space vectors. In order to augment a dataset of small samples, the author's first train the GLICO model on the reconstruction and cross entropy losses. Once the reconstruction loss can no longer be optimized, they interpolate between known latent vectors of the same samples in order to produce generated realistic samples. The authors propose to use these interpolated images similarly to data augmentation where images from the actual training set are replaced by the interpolated images with a certain probability. They report state of the art performance on small sample settings from CIFAR-100, CUB-200, and CIFAR-10. Notably, they obtain just above 30% top-1 accuracy on the CIFAR-10 with 10 samples per class. I.E. the very same setting as challenge 1. In what follows, we re-create their results and provide analysis about the technique.

4.3. Meta-Transfer Learning for Few-Shot Learning

This paper combines a few shots learning and meta-learning: meta-transfer learning. [23] Few shots learning is known to be an effective method to achieve high performance from a small data sample.[27] In a nutshell, we use a set of support data, support labels and query data. A real-life example would be to show a kid cards of different animals with their name on top of it and then show the kid a new card without its name. He will intuitively find which card is more similar to the new card and will conclude they are the same animal. It is a very efficient method in itself. To further optimize it, the paper applies meta-learning principles to learn to learn. [10] If the kid is repeatedly given different support cards and query cards and learns from it, he will get better at finding which cards is the most similar.

The main idea is to leverage a large number of similar few-shot tasks to learn how to adapt a base-learner to a new task for which only a few labelled samples are available. It is ideal for challenge 2, as we have only 100 training samples of CIFAR-10 to train on. Also, the 100 samples are class balanced, with 10 samples of each 10 classes. This imposes a 10 shots 10-way problem. Since we will train a base learner, we can pre-train on a dataset like Mini Imagenet first and then evaluate on CIFAR-10 with our 100 training samples as a support set. The model not learning in our 10 classes is not an issue.

As deep neural networks (DNN) tend to overfit using a few samples only, meta-learning usually uses shallow neural networks (SNN) but that limits its efficiency. [20][18] This paper proposes a novel few-shot learning method called meta-transfer learning (MTL), which learns to adapt a DNN for few shots learning. How? Meta refers to training multiple tasks and transfer is achieved by learning to scale and shifting functions of DNN weights for each task. This makes the model robust for better generalization. A hard task (HT) meta-batch scheme is also introduced as an effective learning curriculum for MTL. [22] The paper achieves state-of-the-art benchmarks on (5-class, 1-shot) and (5-class, 5-shot) learning with miniImageNet and CIFAR-100.[23]

In terms of implementation, it has 3 phases: Pre-training, Meta-training and Meta-evaluation. On the miniImageNet, the paper split the 100 classes to 64, 16 and 20 classes respectively to the different phases. For the challenge, we will keep pre-training and meta-training similar but with CIFAR-10 10 classes when evaluating. We used the PyTorch implementation found on GitHub. [28]

5. Validation Strategy

Given the very limited training data, it is possible to randomly choose a sample of 100 test images that is representative of the sample of validation images that we test on, but not representative of the real data generating distribution. To mitigate this risk, we always evaluate our models on 5×2000 training samples during the model evaluation phase. In the last section 11, as indicated in the final submission instructions, we present our results using the CIFAR-10 test set for 3 random permutations of 100 train samples and 2000 test samples.

6. Challenge 1: Data Augmentation

Our first approach to this challenge was to establish a working baseline using data augmentation techniques, better deep learning architectures, and appropriate hyper-parameter tuning. For reference, we train the 'Net' architecture from the initial test bed on the same data folds (ensured by using the same seed) as our other models discussed below and plot its training curve (see Fig. 2).

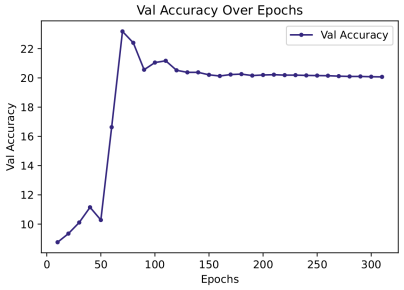

Figure 2. Seed:301, Net, no augmentation

6.1. Methodology

An efficient development environment To efficiently measure our performance, we redesign the testbed to display and save figures for training and testing accuracies, and losses over epochs. Furthermore, we also track the hyper-

parameters of each model trained and its performance over epochs using a latex table format which is saved after every run. Our new and improved testbed was the backbone used to assess the performance of most of our models.

Comparing Models Choosing the best performing model is, perhaps, the most important part of effective hyperparameters tuning and architecture search. Our models were chosen by out of sample performance assessment on a test set 10000 class balanced CIFAR-10 images. Furthermore, we endeavour to train the models well beyond what might be considered their early stopping point to assess the extent to which the method is robust to overfitting.

Architecture search We compared different deepening and widening factors for Wide Residual Networks[29] and found that using a depth of 28 layers and a widening factor of 10 work best for our task. We refer to this network as WRN-28-10 in the rest of this work.

Hyperparameter Tuning We experiment with different optimizers, different learning rates, and different batch sizes. We were interested in optimizing the fastest while still maintaining reasonable optimization performance when the loss becomes smaller. Using adam with a learning rate of 0.01 and a batch size of 128 robustly outperformed all other settings.

Data Augmentation We chose to use the data augmentation scheme proposed by AutoAugment. They use a sequence of random crop, Horizontal flip, AutoAugment, and cutout. We will refer to this sequence as the AutoAugment for simplicity.

Computational and Memory efficiency Using WRN-28-10 is much less parameters efficient then the 'net' model from the original baseline 1. However, the performance gain is considerable for such an already low performing setting.

6.2. Results

When training WRN-28-10 with and without data augmentation, we can see an overwhelming difference in the training accuracy over epochs (figures 4 and 3), where the model trained with AutoAugment takes almost 20 $\times$ as long to reach 100% performance on the training data. When looking at the validation performance, we notice that the non augmented set reaches a stable maximum of approximately 25%, while the model trained with AutoAugment reaches highs of 32.5% and lows of below 25%. However, the model does not see these validation fluctuations before it reaches 500+ epochs of training.

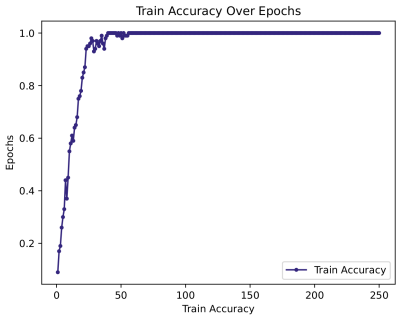

Figure 3. Seed:301, WideResNet28-10, No Augmentation, Adam, lr=0.001, test=10000 samples

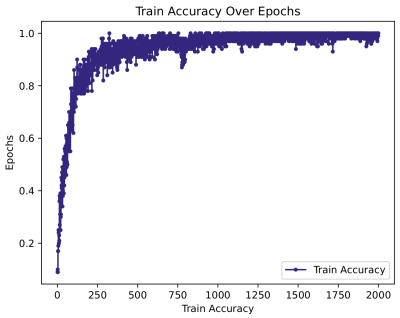

Figure 4. Seed:301, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=10000 samples

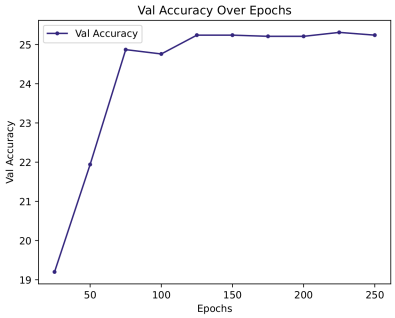

Figure 5. Seed:301, WideResNet28-10, No Augmentation, Adam, lr=0.001, test=10000 samples

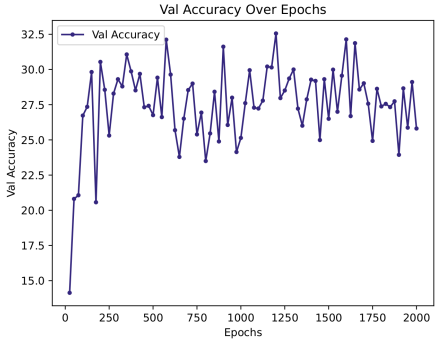

Figure 6. Seed:301, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=10000 samples

6.3. Discussion

The augmented data is richer than its non-augmented counterpart, as it yields better performance. However one must be careful not to overfit the training set when using augmented data, as we see that overfitting the augmented noise of the training set sometimes yields poorer results than not using it at all. If we were to use this as our final model, we would choose to use early stopping at 375-425 epochs as we have done with out codalab submission.

7. Challenge 1: Generative Data Augmentation with GLICO

In this section we explain our recreation of the results from the GLICO paper [2] and explore our most interesting finding. We pieced together the code that the GLICO authors made available online [2] to augment our training set with generated data.

7.1. Methodology

GLICO GLICO proposes to jointly train an embedding matrix, a generator network, and a weak classifier. Their trainable parameters are optimized by backpropagating error from a reconstruction loss and a cross-entropy loss. The goal of the network is to learn a mapping from learnable noise embeddings to train set images. To this end, each image in the training set is mapped to a learnable embedding (of size 512). During training, a Gaussian noise vector is concatenated to the learnable embedding associated with some training image. The result of this concatenation is passed to a generator network that attempts to reconstruct the image. The reconstruction is then passed to a classifier which attempts to determine its class and to the perceptual[9] reconstruction loss. As part of the GLO framework, embedding vectors are projected onto the unit sphere after each parameter update. New images are generated by spherically interpolating between the latent vectors of samples from the same class.

Our implementation To be clear, we did no implement GLICO ourselves, but we did rework it to fit into our code. In our code, we train a GLICO model using the 100 training samples allocated. Then, we take combinations of the latent vectors of each class obtaining $c(10, 2) = 45$ distinct pairs for each class. For each pair, we spherically interpolate between the points in the latent space for 5 ratios between the. This yields $45 \cdot 10 \cdot 5 = 2250$ generated images total and 225 per class. We store all the generated images in a lookup table. During training, we replace images in the training batch with generated images of the same class according to some probability.

Hyperparameter Tuning We use exactly the same settings as discussed before: WRN-28-10, adam, $lr = 0.001$, and batch size 128. However, this time we also have the option of tuning GLICO's parameters and its probability of replacing train images. Default settings yield good results for training glico (see fig 10 and 11). When augmenting the test set we find our best results are achieved when we set the glico replace probability to 0.05. I.E. when we replace train images with generated images 5% of the time.

Computational and Memory efficiency Glico is relatively fast to train on only 100 samples, thus adding relatively little overhead compared to training WRN-28-10. However, to benefit from its regularizing effect one must train WRN-28-10 for a longer time, so it does add overhead in this sense.

Data Augmentation We use the same Auto Augment data augmentation scheme.

7.2. Results

When compared to training WRN-28-10 on augmented data only, we see that the addition of GLICO's generated samples increase our score substantially. Our model now acheives maximal accuracy above 35% and lows of just below 30%. The network's valication accuracy is generally trending upwards even after 1000 epochs of training. Furthermore, we see that the model's training accuracy never fully reaches 100% (figure 8). Lastly, note that training the model with replace probability 20% (figure 9). yields lower validation accuracy then the 5% setting.

[2] glico repository https://github.com/IdanAzuri/glico-learning-small-sample

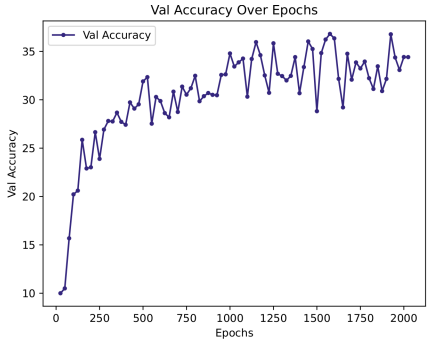

Figure 7. Validation Accuracy. Seed:301, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=10000 samples , 0.05 replace prob for GLICO

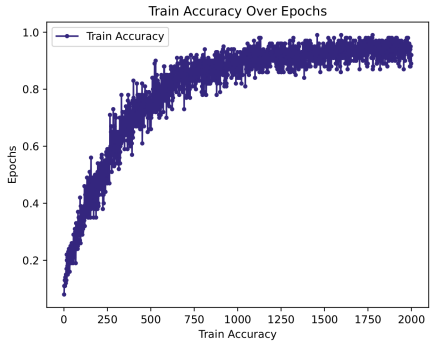

Figure 8. Train Accuracy.Seed:301, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=10000 samples, 0.05 replace prob for GLICO

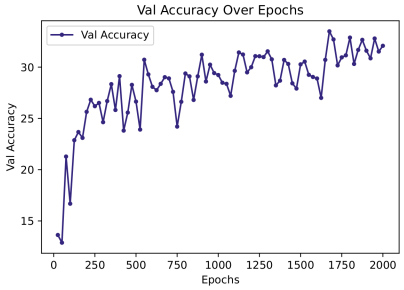

Figure 9. Validation Accuracy, Seed:301, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=10000 samples, 0.2 replace prob for GLICO

7.3. Discussion

The samples generated by GLICO clearly have a strong regularizing effect on our network. Not only do we obtain a higher validation accuracy, but this accuracy remains stable as epochs increase, unlike our previous network trained exclusively with AutoAugment. Furthermore, the performance of our network suffers when we introduce more generated data 9 indicating that the positive effect GLICO cannot entirely be equated to generating new images. It is possible that the combination of features in images that results from the interpolation between latent vectors allows the model to learn good associations that it otherwise could not. Although the samples generated appear quite faint and noisy 10 they do have a positive effect. Given these good results, we choose to use GLICO as our final approach to challenge 1. If we had to cache the model at a certain epoch range we would choose $1800 - 1850$ epochs.

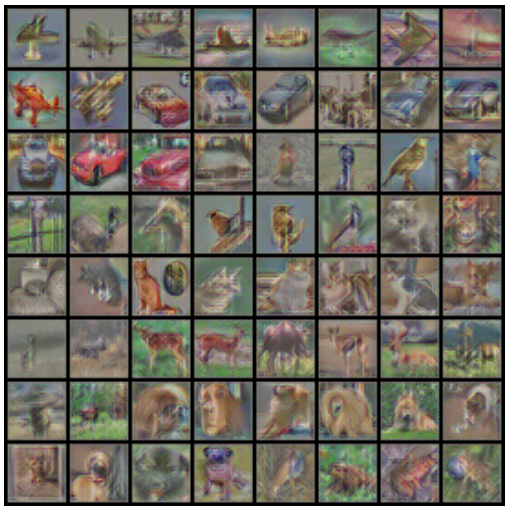

Figure 10. Reconstruction of training examples from latent vectors at 700 epochs trained on 100 samples

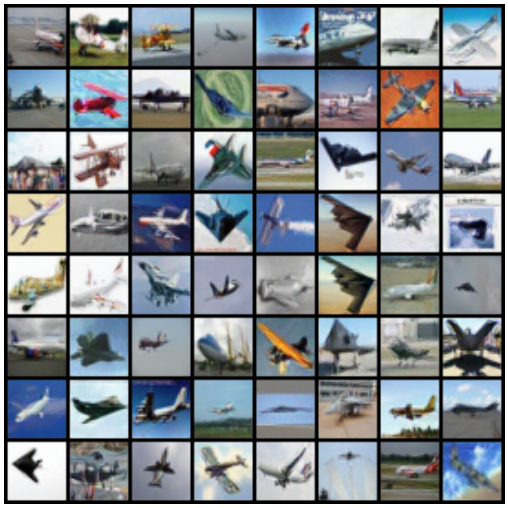

Figure 11. Reconstruction of training examples from latent vectors at 700 epochs trained on 1000 samples

8. Challenge 2: Transfer Learning

The first approach for challenge 2 was Transfer Learning. Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. In our case, many image classification models have been pre-trained already on datasets other than CIFAR-10. These models have learned the features necessary to recognize the difference in images so will be useful once adapted to our dataset.

8.1. Methodology

We have tried to tune various models offered by PyTorch vision library, as well as external Open Source models such as Efficient Nets. We would then evaluate them on 5 different validation sets which are part of the Train set.

Tuning To tune the models, we experimented with different learning rates and different optimizers such as Stochastic Gradient Descent (SGD) and Adam Optimizer. The epochs varied. We tried to train the models until they overfit in training. Depending on the model, we optimized their classification layer mostly, while keeping the features exactly the same.

Computational and Memory efficiency The Alexnet in the test bed runs really fast, converging in only 10 epochs. Among all the models we tried, it was the fastest. The slowest was definitely Efficient Nets, which took thousands of epochs to get a good accuracy. The model which gave us the best accuracy converges really fast in around 200 epochs.

In terms of memory, AlexNet in the test bed only contained 3.39M trainable parameters, which is very light, compared to other the models. Efficient Nets were the most memory demanding. Our 24 GB GPU would only allow us to run up to EfficientNet-b5, given that we don't resize the images to a too high resolution.

Models	$10^6 \times$Parameters
Net	0.02
AlexNet	14.4
VGG16	134.3
ResNet50	25.5
WideResNet28-10	36.48
Inception_V3	23.8
EfficientNet-b5	30
EfficientNet-b6	43
EfficientNet-b7	66

Table 1. Model Parameters Comparison table.

While VGG16 had the most number of parameters, it was surprisingly really quick to converge. Similar to AlexNet, we resized our image to 224x224.

8.2. Results

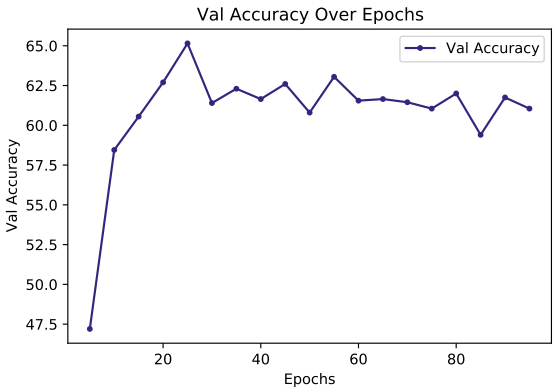

Figure 12. Seed:1620017187, VGG16 + Autoaugment + Horizontal flip + cutout

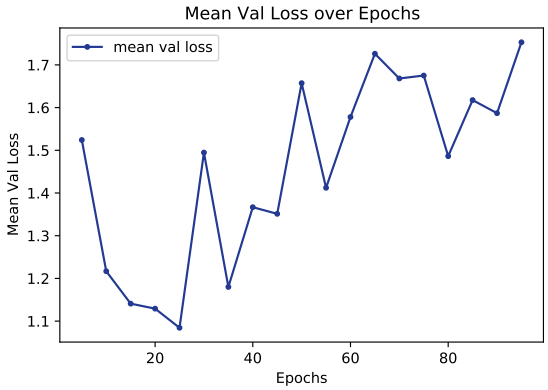

Figure 13. Seed:1620017187, VGG16 + Autoaugment + Horizontal flip + cutout

Here is are the results. The maximum accuracy observed is 65.15%. As we can see, the model is fast to converge. Around epochs 25, it will quickly start to overfit the training data and the validation loss will become smaller and smaller.

One surprising observation we found was that sometimes, not changing the output features to 10 but keeping it as 1000 yield a similar or better results. With 1000 target size, the cross entropy still optimizes and the prediction class is still obtained using the max-log probability.

Key aspects of the code is explained in the appendix. 16

Here are the hyper-parameters of our final VGG16 model:

- SGD optimizer with 0.01 learning rate, 0.9 momentum and 0.0005 weight decay
- Optimized on the classifier and last convolutional layer.
- 1000 output features
- Images resized to 224
- Using Random Crop + Horizontal Flip + AutoAugment + Cutout

2 other seed runs as well as one full run can be found in the appendix. 4 15.3

9. Challenge 2: Meta-Transfer Learning

In the hope to improve the accuracy we achieved using only Transfer Learning, we experimented with Meta-Transfer Learning (MTL). It combines Transfer Learning, Meta Learning and few shots methodology. As discussed extensively in the literature review, in our case, we working on solving a 10 shots 10 way problem.

9.1. Methodology

MTL has 3 steps: pretraining, meta-training and meta-evaluation. Usually all 3 steps would be done on the same dataset, but with different classes. For example, miniImageNet would be split to 64 classes, 16 classes and 20 classes respectively. Since were only able to use 100 training samples from CIFAR-10, we decided to pre-train and meta-train on another dataset and then meta-evaluate on CIFAR-10, hoping that the base learner would learn to differentiate between classes well.

We decided to pre-train and meta-train on miniImageNet since it requires fewer resources and infrastructure than running on the full ImageNet dataset. Instead of then using the test set, we would set up the dataloaders so that the support set uses the training set of 100 samples and the query set would be the test set.

Computational and Memory efficiency This approach requires a lot of memory. The author of the PyTorch implementation recommends a minimum of 16GB of GPU. The pre-train phase takes around 6 hours and the meta-train takes around 2 hours.

As the models were saved, the meta-evaluation can be run almost instantly.

9.2. Results

CIFAR-10 set	Mean Accuracy (3 seeds)
Train	49.9 +- 0.01
Test	52.0 +- 0.01

Table 2. Accuracy on 3 set splits

The code to obtain the results can be found in the appendix. 15 We achieved a better result evaluating the test set compared to the train set.

10. Challenge 2 Discussion

Out of all the methods attempted, the MTL few shots learning is the most complex to implement. For a company, it would not be worth it to choose this approach, due to the time/ benefit ratio - hours more of training, with final accuracy almost similar to AlexNet, which can be achieved in minutes. Since it requires a 16 GB GPU, a company might need to purchase a more expensive GPU. It would not be feasible to implement this model on a mobile platform or embedded architecture either, due to it's memory constraint.

Having only 100 training samples on complex images, the MTL did not deliver the expected results. In other settings, it would still be worth trying, since it has been shown to produce high accuracy with 4000 CIFAR-10 images. We believe it could have had better results given we had more training samples, or less complex images, like numbers or characters.

Transfer Learning, on the other hand, is a very simple but effective method. In minutes, we can receive similar accuracy to the MTL approach. While some models require a lot of memory, some models like AlexNet or VGG16 can be trained on Google Colab, making it easy for deep learning enthusiasts of all levels.

The Meta-Transfer Learning for a few shots implementation uses an adaptation of ResNet-12 architecture. It would be interesting to see what kind of results we would get if the MTL was adapted with other networks, such as VGG or AlexNet. This is something that has not been attempted by the paper either. It could be interesting future work.

11. Final results on test set

A table with the maximum test set accuracies for both challenges can be found on the next page. 3

Our best accuracy for challenge 1 and challenge 2 is **35.85**% and **65.15**% respectively. Furthermore, the appendix includes graphs of our training of 3 different sets of the CIFAR-10 test data for challenge 1 and challenge 2.

Table 3. Maximum Test set Accuracies obtained for challenge 1 and challenge 2

NETWORK	EPOCHS	TEST ACCURACY	OPTIMIZER	LEARNING RATE	TEST SAMPLES	DATA AUGMENTATION	SEED
WRN-28-10	1875	34.5	ADAM	0.001	2000	AUTOAUGMENT	1620014211
WRN-28-10	1950	34.6	ADAM	0.001	2000	AUTOAUGMENT	1620014879
WRN-28-10	**2000**	**35.85**	**ADAM**	**0.001**	**2000**	**AUTOAUGMENT**	**1620014643**
VGG16	300	63.4	SGD	0.01	2000	AUTOAUGMENT	1620017995
VGG16	**25**	**65.15**	**SGD**	**0.01**	**2000**	**AUTOAUGMENT**	**1620017187**
VGG16	100	64.45	SGD	0.01	2000	AUTOAUGMENT	1620017687

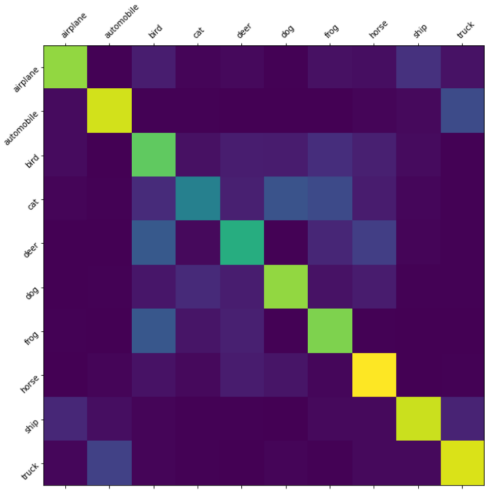

Figure 14. Confusion Matrix of VGG16

To have a look at the predictions of the VGG16, we plot a confusion matrix, depicting the count of prediction to actual target for each class. On x axis we have the predictions and on the y axis we have the targets.

We have a diagonal line for the correct predictions while the other spots are misclassifications. The most common errors are frogs and bird, deer and birds and cats and dogs.

11.1. Final discussion on test set

Our results from the CIFAR-10 training set generalize well to the test set. For challenge 1, our best scores are obtain around the epoch range we predicted from the test set. GLICO's strong regularizing effect allows WRN-28-10 to be trained for a large amount of epoch without overfitting the test set (see 22,24, and 26).

Finally, we found that the best model was VGG16. It converges quick and does not need as much memory as for Meta-Transfer Learning. For different seeds, we consistently get over 63%.

11.2. Possible Future Work

- The current MTL implementation uses ResNet-12. It would be interesting to experiment a similar approach with different models. A pre-trained model might do better than another, which is the case for VGG16 and ResNet50 for CIFAR-10. The same might apply for MTL.
- MTL trains a base learner model from scratch. An exciting possible work would be to be able to use a pre-trained model, like VGG16 to then start pre-training and meta-training specifically for few shots.
- GLICO's performance on small samples could possible be ameliorated by utilizing metric learning principles. Using a triplet loss as opposed to the cross entropy classification loss would be an interesting experiment.

12. Conclusion

Deep Neural Networks have been able to surpass humans in some cases like image recognition and classification. However, learning from limited samples still remains a challenge. In this paper, we experimented with techniques such as Data Augmentation, Generative Models, Meta-learning and Transfer Learning.

Here are some things we learned:

- Meta-Transfer Learning pre-trained on miniImageNet did not perform as well as expected. With a few more samples, we might have been able to use some training samples to fine-tune the model for CIFAR-10 dataset. It surely would have performed better.
- Transfer Learning is really powerful. We can achieve a high accuracy in a short amount of time.
- When using pre-trained models, we don't necessarily need to decrease the output features to the number of classes. With log max probability, we can still get a prediction and it can have better performance.
- Data Augmentation is key to introduce variance in the dataset when we have only a limited amount of samples. It prevents the model over-fitting easily.

- GLICO can act as a regularizer when sampled in low probability. It prevents the model from completely overfitting and keeps the model at a good accuracy.

All in all, we are quite satisfied with our results and the knowledge we learned throughout. We achieved an accuracy of 35% for Challenge 1 and 65% for Challenge 2.

Our code used can be found here.

13. Contribution

Benjamin Therien

- Mostly challenge 1
- Generative Latent Implicit Conditional Optimization
- Data Loaders, Test classes, and Graphs automation

Nigel Yong Sao Young

- Mostly challenge 2
- AutoAugmentation
- Meta-Transfer Learning
- Transfer Learning

Both of us worked on the report.

14. Notes

References

[1] 4uiiurz1. Pytorch autoaugment, 2019. 2

[2] Idan Azuri and Daphna Weinshall. Generative latent implicit conditional optimization when learning from small sample, 2020. 2, 5

[3] Nihar Bendre, Hugo Terashima-Marín, and Peyman Najafirad. Learning from few samples: A survey. *CoRR*, abs/2007.15484, 2020. 1

[4] Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks, 2019. 2

[5] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501, 2018. 2

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. 2

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. 2

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. 2

[9] Yedid Hoshen and Jitendra Malik. Non-adversarial image synthesis with generative latent nearest neighbors, 2018. 5

[10] Timothy M. Hospedales, Antreas Antoniou, Paul Micaelli, and Amos J. Storkey. Meta-learning in neural networks: A survey. *CoRR*, abs/2004.05439, 2020. 3

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. 2

[12] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012. 1

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. 1

[14] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998. 2

[15] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast autoaugment. *CoRR*, abs/1905.00397, 2019. 2

[16] Haibin Ling and Kazunori Okada. Diffusion distance for histogram comparison. In *CVPR (1)*, pages 246–253, 2006. 2

[17] David G. Lowe. Distinctive image features from scale-invariant keypoints, 2004. 1

[18] Youngjae Min and Hye Won Chung. Shallow neural network can perfectly classify an object following separable probability distribution. *CoRR*, abs/1904.09109, 2019. 3

[19] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016. 2

[20] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014. 3

[21] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. 2

[22] Qianru Sun, Yaoyao Liu, Zhaozheng Chen, Tat-Seng Chua, and Bernt Schiele. Meta-transfer learning through hard tasks. *CoRR*, abs/1910.03648, 2019. 3

[23] Qianru Sun, Yaoyao Liu, Tat-Seng Chua, and Bernt Schiele. Meta-transfer learning for few-shot learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 3

[24] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020. 2

[25] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training, 2021. 2

[26] Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. *CoRR*, abs/1711.00937, 2017. 2

[27] Yaqing Wang and Quanming Yao. Few-shot learning: A survey. *CoRR*, abs/1904.05046, 2019. 3

[28] yaoyao liu. Pytorch meta-transfer learning, 2019. 3

[29] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2017. 2, 4

15. Appendix

15.1. Meta-Evaluation code

```
1  def eval(self):
2      """The function for the meta-eval phase."""
3      # Load the logs
4      trlog = torch.load(osp.join(self.args.save_path, 'trlog'))
5
6      # Load meta-test set
7      # test_set = Dataset('test', self.args)
8      # sampler = CategoriesSampler(test_set.label, 600, self.args.way, self.args.shot +
           self.args.val_query)
9      # loader = DataLoader(test_set, batch_sampler=sampler, num_workers=8, pin_memory=True)
10
11     # Set test accuracy recorder
12     test_acc_record = np.zeros((600,))
13
14     # Load model for meta-test phase
15     if self.args.eval_weights is not None:
16         self.model.load_state_dict(torch.load(self.args.eval_weights)['params'])
17     else:
18         self.model.load_state_dict(torch.load(osp.join(self.args.save_path, 'max_acc' + '.pth'))
   ['params'])
19     # Set model to eval mode
20     self.model.eval()
21
22     # Set accuracy averager
23     ave_acc = Averager()
24
25     use_cuda = torch.cuda.is_available()
26     device = torch.device("cuda" if use_cuda else "cpu")
27
28     normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
29                                      std=[0.229, 0.224, 0.225])
30
31
32     # We resize images to allow using imagenet pre-trained models, is there a better way?
33     resize = transforms.Resize(80)
34
35     transform_val = transforms.Compose([resize, transforms.ToTensor(), normalize]) #careful to keep
   this one same
36     transform_train = transforms.Compose([resize, transforms.ToTensor(), normalize])
37
38
39
40     ##### Cifar Data
41     cifar_data = datasets.CIFAR10(root='.', train=False, transform=transform_train, download=True)
42     cifar_data_val = datasets.CIFAR10(root='.', train=False, transform=transform_val,
                                         download=True)
43
44     mean_acc = []
45
46     for seed in range(3):
47
48         prng = RandomState(seed)
49         # random_permute = prng.permutation(np.arange(0, len(cifar_data_val)/10))
50         random_permute = prng.permutation(np.arange(0, 1000))
51
52
53
54         indx_train = np.concatenate(
55             [np.where(np.array(cifar_data.targets) == classe)[0][random_permute[0:10]] for classe
   in range(0, 10)])
56
57         indx_val = np.concatenate(
58             [np.where(np.array(cifar_data.targets) == classe)[0][random_permute[10:210]] for classe
   in range(0, 10)])
59
60         train_data = Subset(cifar_data, indx_train)
61         val_data = Subset(cifar_data_val, indx_val)
62
63         train_loader = torch.utils.data.DataLoader(train_data,
64                                                    batch_size=128,
65                                                    shuffle=True)
66
67         val_loader = torch.utils.data.DataLoader(val_data,
68                                                  batch_size=128,
69                                                  shuffle=False)
70
71         # Generate labels
72         # label = torch.arange(self.args.way).repeat(self.args.val_query)
73         # if torch.cuda.is_available():
74         #     label = label.type(torch.cuda.LongTensor)
75         # else:
76         #     label = label.type(torch.LongTensor)
77         # label_shot = torch.arange(self.args.way).repeat(self.args.shot)
78         # if torch.cuda.is_available():
79         #     label_shot = label_shot.type(torch.cuda.LongTensor)
80         # else:
81         #     label_shot = label_shot.type(torch.LongTensor)
82
83         dataloader_iterator = iter(train_loader)
84         support_data, support_target = next(dataloader_iterator)
85         support_data, support_target = support_data.to(device), support_target.to(device)
86
87         # Start meta-test
88         for i, batch in enumerate(val_loader, 1):
89             if torch.cuda.is_available():
90                 data, label = [_.cuda() for _ in batch]
91             else:
92                 data, label = batch
93
94             # k = self.args.way * self.args.shot
95             data_query = data
96             # data_shot, data_query = data[:k], data[k:]
97             logits = self.model((support_data, support_target, data_query))
98             acc = count_acc(logits, label)
99             ave_acc.add(acc)
100            test_acc_record[i-1] = acc
101            if i % 100 == 0:
102                print('batch {}: {:.2f}({:.2f})'.format(i, ave_acc.item() * 100, acc * 100))
103
104        # Calculate the confidence interval, update the logs
105        m, pm = compute_confidence_interval(test_acc_record)
106        print('Val Best Epoch {}, Acc {:.4f}, Test Acc {:.4f}'.format(trlog['max_acc_epoch'],
   trlog['max_acc'], ave_acc.item()))
107        print('Test Acc {:.4f} + {:.4f}'.format(m, pm))
108
109        mean_acc.append(ave_acc.item())
110
111    mean_acc = np.array(mean_acc)
112
113    print(f"Mean accuracy for 3 seeds: {mean_acc.mean():.2f} +- {mean_acc.std():.2f}")
```

Figure 15. Meta-Transfer Learning Evaluation Code

On line 15-18, the meta-trained model with the maximum accuracy is loaded.

On line 41-42, we load the train and test split from the CIFAR test set.

For 3 seeds, the data is loaded similar to the test bed.

On line 83-85, we iterate on the train data once to use as support data and support labels. They are kept fixed.

On line 88, we then iterate on the test set and consider the data as query data supplied to the few shot model. The accuracy is then evaluated.

This is repeated for 3 seeds. The mean accuracy is then calculated and printed.

15.2. Transfer Learning code

```
1  torch.cuda.empty_cache()
2  gc.collect()
3
4  #     resnet18 = models.resnet18()
5  #     alexnet = models.alexnet()
6  #     vgg16 = models.vgg16()
7  #     squeezenet = models.squeezenet1_0()
8  #     densenet = models.densenet161()
9  #     inception = models.inception_v3()
10 #     googlenet = models.googlenet()
11 #     shufflenet = models.shufflenet_v2_x1_0()
12 #     mobilenet = models.mobilenet_v2()
13 #     resnext50_32x4d = models.resnext50_32x4d()
14 #     wide_resnet50_2 = models.wide_resnet50_2()
15 #     mnasnet = models.mnasnet1_0()
16
17 OPTIM = "sgd"
18 MODEL = "vgg16"
19 EPOCH_NUM = 1000
20 TRAIN_SAMPLE_NUM = 100
21 VAL_SAMPLE_NUM = 2000
22 BATCH_SIZE = 64
23 VALIDATION_SET_NUM = 1
24 AUGMENT = True
25 VAL_DISPLAY_DIVISOR =5
26 CIFAR_TRAIN = False
27
28 #cifar-10:
29 #mean = (0.4914, 0.4822, 0.4465)
30 #std = (0.247, 0.243, 0.261)
31
32 normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
33                                  std=[0.229, 0.224, 0.225])
34 if AUGMENT:
35     dataAugmentation = [
36         transforms.RandomCrop(32, padding=4),
37         transforms.RandomHorizontalFlip(),
38         AutoAugment(),
39         Cutout()
40     ]
41     augment = "Crop,Flip,AutoAugment,Cutout"
42 else:
43     dataAugmentation = []
44     augment = "Nothing"
45
46
47 # We resize images to allow using imagenet pre-trained models, is there a better way?
48 resize = transforms.Resize(224)
49
50 transform_train = transforms.Compose(dataAugmentation + [resize, transforms.ToTensor(), normalize])
51 transform_val = transforms.Compose([resize, transforms.ToTensor(), normalize]) #careful to keep this one same
52
53 cifar_train = datasets.CIFAR10(root='.',train=CIFAR_TRAIN, transform=transform_train, download=True)
54 cifar_val = datasets.CIFAR10(root='.',train=CIFAR_TRAIN, transform=transform_val, download=True)
55
56 ss = SmallSampleController(numClasses=10,trainSampleNum=TRAIN_SAMPLE_NUM, # abstract the data-loading procedure
57                            valSampleNum=VAL_SAMPLE_NUM, batchSize=BATCH_SIZE,
58                            multiplier=VALIDATION_SET_NUM, trainDataset=cifar_train,
59                            valDataset=cifar_val)
60
61
62 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
63 train_data, valSets, seed = ss.generateNewSet(device,valMultiplier = VALIDATION_SET_NUM) #Sample from datasets
64
65
66
67 eNet = getModel(MODEL).cuda()
68
69 # for param in eNet.parameters():
70 #     param.requires_grad = False
71
72 # model = featureExtractor(eNet).cuda()
73 model = eNet
74
75 print(model.parameters)
76
77 parameters = list(model.features[28:].parameters()) +  list(model.classifier.parameters())
78
79 optimizer,LR = getOptimizer128(OPTIM, parameters)
80
81 # print(model.features[28:].parameters() + model.classifier.parameters())
82 print(' => Total trainable parameters: %.2fM' % (sum(p.numel() for p in model.parameters()) / 1000000.0))
83
84 trainTracker = {"meanLoss":[],"accuracy":[]}
85 valTracker = {"allLoss":[],"allAcc":[],"meanLoss":[],"meanAcc":[],"stdAcc":[]}
86 latexTracker = []
87
88 print("Begin Train for {} epochs".format(EPOCH_NUM))
89 for epoch in range(EPOCH_NUM):
90     acc, loss = train(model, device, train_data[0], optimizer, epoch+1, display=True)
91     trainTracker["meanLoss"].append(loss)
92     trainTracker["accuracy"].append(acc)
93
94     if (epoch+1) % VAL_DISPLAY_DIVISOR == 0:
95         checkTest(model,device,valSets,valTracker,latexTracker,epoch+1,
96                   model_name=MODEL,optim_name=OPTIM,lr=LR,totalTestSamples=VAL_SAMPLE_NUM*VALIDATION_SET_NUM,
97                       seed=seed,verbose=True)
98
99
100 # print(parameters)
```

Figure 16. Transfer Learning Code

In here, we change the model and optimizer name to the one we need.

On line 77, for the VGG, we are optimizing on the last convolutional layer and the classifier.

```python
def getModel(model_name):
    if "wide" in model_name.lower():
        return WideResNet(28, 10, num_classes=10)
    elif "efficient" in model_name.lower():
        return EfficientNet.from_pretrained(model_name,num_classes = 10) # change to not be pretrained
    elif "vgg16" in model_name.lower():
        model = models.vgg16(pretrained=True)
        # model.classifier[6] = nn.Linear(4096, 10)
        return model
    elif "alexnet" in model_name.lower():
        model = models.alexnet(pretrained=True)
        model.classifier = nn.Linear(256 * 6 * 6, 100)
        return model
    elif "resnet18" in model_name.lower():
        model = models.resnet18(pretrained=True)
#         model.fc.out_features = 10
        return model
    elif "resnet50" in model_name.lower():
        model = models.resnet50(pretrained=True)
#         model.fc.out_features = 10
        return model
    elif "densenet161" in model_name.lower():
        model = models.densenet161(pretrained=True)
#         model.fc.out_features = 10
        return model
    elif "wideresnet" in model_name.lower():
            model = models.wide_resnet50_2(pretrained=True)
            return model
    elif "resnext101" in model_name.lower():
            model = models.resnext101_32x8d(pretrained=True)
            return model
    elif "inception_v3" in model_name.lower():
            model = models.inception_v3(pretrained=True,aux_logits=False)
            return model
    t


def getOptimizer128(optimizer_name,parameters):
    if "sgd" in  optimizer_name.lower():
        LR = 0.01
        optim = torch.optim.SGD(parameters,
                                lr=LR, momentum=0.9,
                                weight_decay=0.0005)
        return optim, LR
    elif "adam" in optimizer_name.lower():
        LR = 0.00005
        optim = torch.optim.Adam(parameters,
                                 lr=LR, weight_decay=0)
        return optim, LR
```

Figure 17. Model and Optimize configuration Code

On the code snippet above, we abstract choosing the model and optimizer.

15.3. VGG run on additional 2 seeds

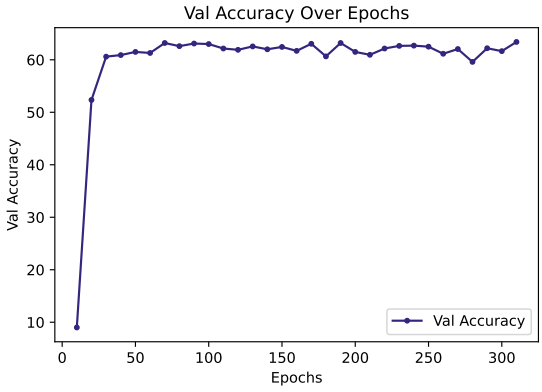

Figure 18. Seed:1620017995, VGG16 + Autoaugment + Horizontal flip + cutout

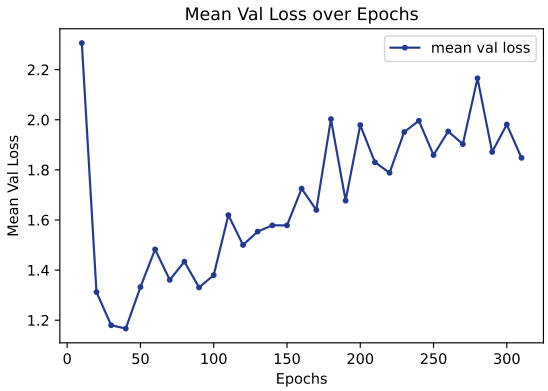

Figure 19. Seed:1620017995, VGG16 + Autoaugment + Horizontal flip + cutout

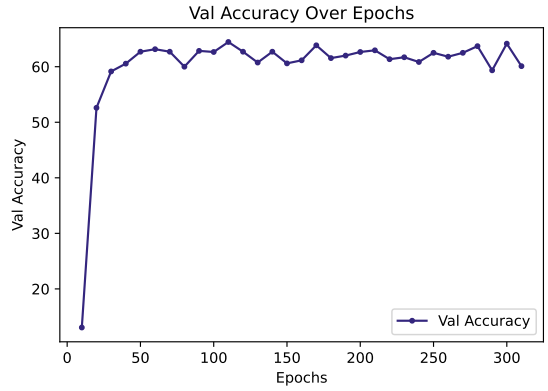

Figure 20. Seed:1620017687, VGG16 + Autoaugment + Horizontal flip + cutout

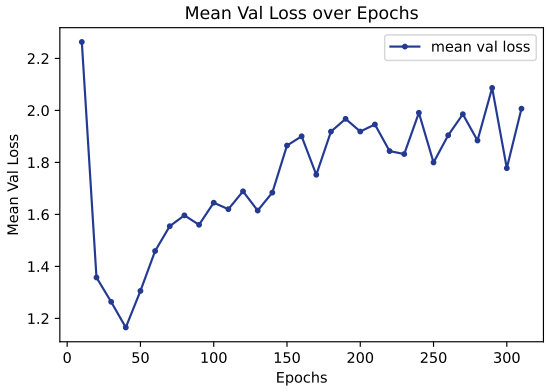

Figure 21. Seed:1620017687, VGG16 + Autoaugment + Horizontal flip + cutout

15.4. GLICO WRN-28-10 run on 3 seeds of CIFAR-10 test

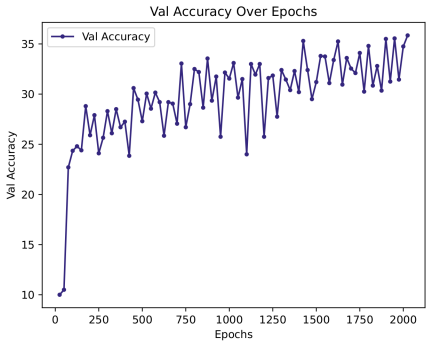

Figure 22. Test Accuracy. Seed:1620014643, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=2000 samples , 0.05 replace prob for GLICO

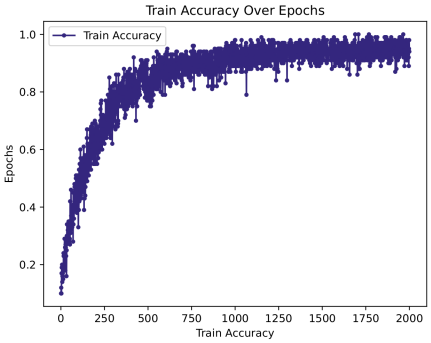

Figure 23. Train Accuracy. Seed:1620014643, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=2000 samples, 0.05 replace prob for GLICO

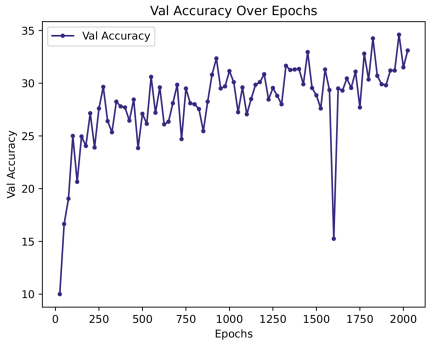

Figure 24. Test Accuracy. Seed:1620014879, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=2000 samples , 0.05 replace prob for GLICO

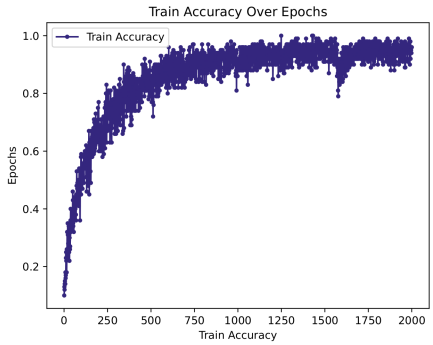

Figure 25. Train Accuracy. Seed:1620014879, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=2000 samples, 0.05 replace prob for GLICO

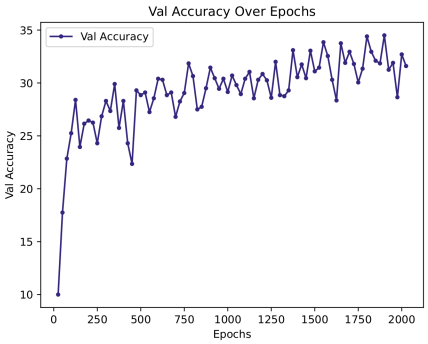

Figure 26. Test Accuracy. Seed:1620014211, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=2000 samples , 0.05 replace prob for GLICO

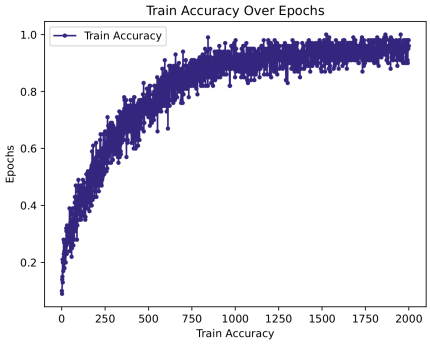

Figure 27. Train Accuracy. Seed:1620014211, WideResNet28-10,AutoAugment , Adam, lr=0.001, test=2000 samples, 0.05 replace prob for GLICO

Table 4. Classification Accuracies - VGG16 + AutoAugment on seed 1620048211

NETWORK	EPOCHS	ACCURACY	OPTIMIZER	LEARNING RATE	TESTING SAMPLES
VGG16	5	47.2	SGD	0.01	2000
VGG16	10	58.45	SGD	0.01	2000
VGG16	15	60.55	SGD	0.01	2000
VGG16	20	62.7	SGD	0.01	2000
VGG16	**25**	**65.15**	**SGD**	**0.01**	**2000**
VGG16	30	61.4	SGD	0.01	2000
VGG16	35	62.3	SGD	0.01	2000
VGG16	40	61.65	SGD	0.01	2000
VGG16	45	62.6	SGD	0.01	2000
VGG16	50	60.8	SGD	0.01	2000
VGG16	55	63.05	SGD	0.01	2000
VGG16	60	61.55	SGD	0.01	2000
VGG16	65	61.65	SGD	0.01	2000
VGG16	70	61.45	SGD	0.01	2000
VGG16	75	61.05	SGD	0.01	2000
VGG16	80	62.0	SGD	0.01	2000
VGG16	85	59.4	SGD	0.01	2000
VGG16	90	61.75	SGD	0.01	2000
VGG16	95	61.05	SGD	0.01	2000
VGG16	100	61.20	SGD	0.01	2000