


# How to improve your image classifier with Google's AutoAugment





















Philip Popien

Follow

Jul 13, 2018 · 9 min read

*State-of-the-art results on CIFAR-10, CIFAR-100, SVHN and ImageNet by using optimized data augmentation techniques. Use them yourself from this repository.*

	Original	Sub-policy 1	Sub-policy 2	Sub-policy 3	Sub-policy 4	Sub-policy 5
Batch 1						
Batch 2						
Batch 3						
		Equalize, 0.4, 4 Rotate, 0.8, 8	Solarize, 0.6, 3 Equalize, 0.6, 7	Posterize, 0.8, 5 Equalize, 1.0, 2	Rotate, 0.2, 3 Solarize, 0.6, 8	Equalize, 0.6, 8 Posterize, 0.4, 6

Some of the best augmentations found for ImageNet. From <https://arxiv.org/abs/1805.09501v1>

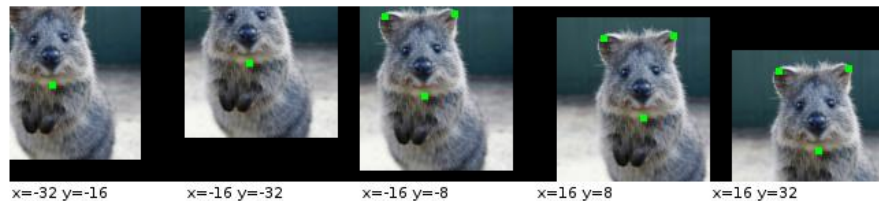
AutoML — the idea of using Machine Learning to improve Machine Learning design choices like architectures or optimizers — has reached the space of data augmentation. This article explains what Data Augmentation is, how Google's AutoAugment searches for the best augmentation policies and how you can transfer these policies to your own image classification problem.

. . .

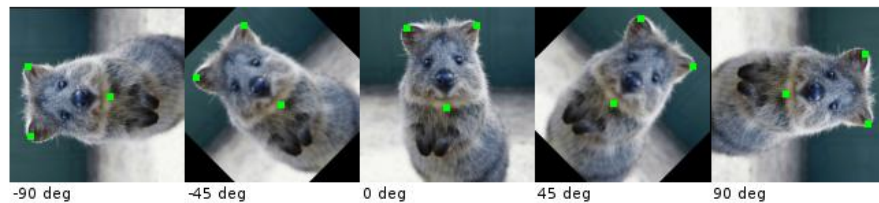
## Data Augmentation

Data augmentation means randomly applying various transformations to your inputs while training Machine Learning models. This artificially enlarges your training data as new possible input-output pairs are generated. It also helps prevent overfitting as the network almost never sees the exact same inputs twice and cannot just memorize them. Typical data augmentation techniques for images include randomly cropping parts out of input images, horizontally flipping them and applying affine transformations like translation, rotation or shearing.

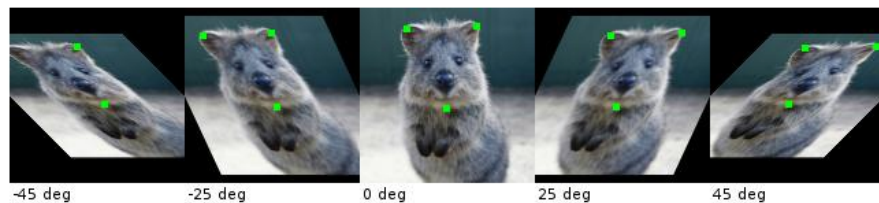
Affine: Translate



Affine: Rotate



Affine: Shear



From <https://github.com/aleju/imgaug>

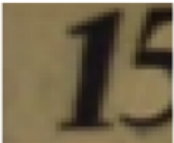

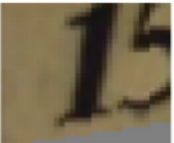


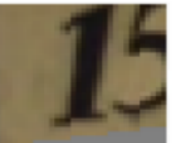
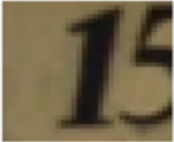

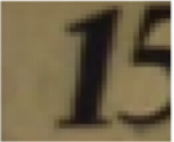
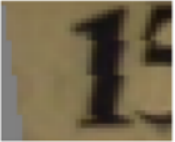


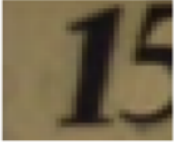
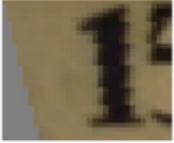
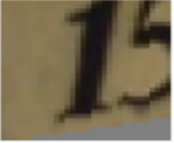

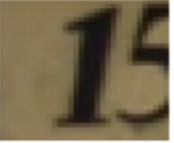
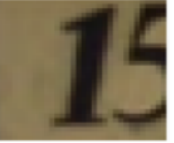
• • •

## Learning Augmentation Policies from Data

The idea of AutoAugment is to learn the best augmentation policies for a given dataset with the help of Reinforcement Learning (RL). Because the search space of all possible ways to apply and combine transformations on images is huge, they restrict it with several design choices. A policy consists of 5 sub-policies and each sub-policy applies 2 image operations in sequence. Each of those image operations has two parameters: The probability of applying it and the magnitude of the operation (e.g. rotate 30 degrees in 70% of cases).

How is such a policy applied to the images while training? For each image in our current batch, first one of the sub-policies is chosen uniformly at random and then this sub-policy is applied. Let's look at an example policy with 5 sub-policies that is applied to an image from the SVHN dataset:

Sub-policy 1 shears the image in 90% of cases with magnitude 7 in either x-direction. Then, with 20% probability, the colors of that image are inverted. Sub-policy 4 inverts the colors in 90% of cases and afterwards equalizes the color histograms 6 out of 10 times. The operation's magnitudes are fixed, but because of the randomness in which sub-policy is applied and the operation's probabilities there are a lot of possible augmentation outcomes for a single image.

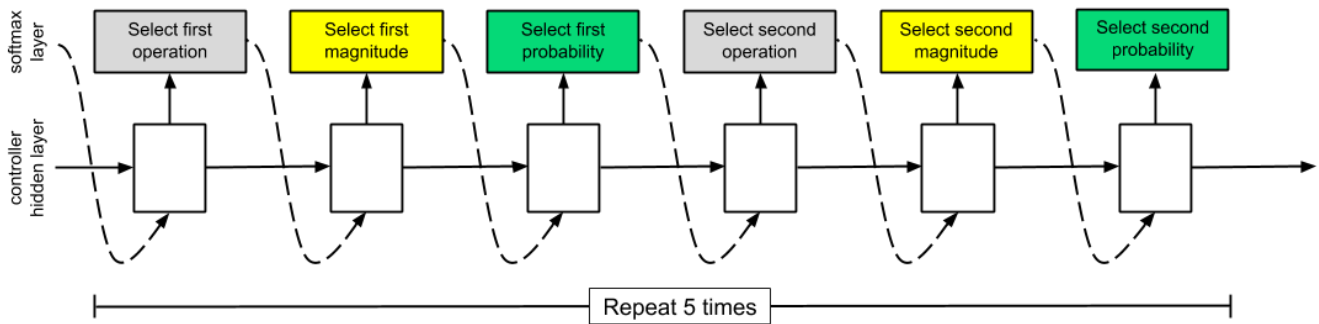
	Original	Sub-policy 1	Sub-policy 2	Sub-policy 3	Sub-policy 4	Sub-policy 5
Batch 1						
Batch 2						
Batch 3						
		ShearX, 0.9, 7 Invert, 0.2, 3	ShearY, 0.7, 6 Solarize, 0.4, 8	ShearX, 0.9, 4 AutoContrast, 0.8, 3	Invert, 0.9, 3 Equalize, 0.6, 3	ShearY, 0.8, 5 AutoContrast, 0.7, 3

Example of applying some of the best found augmentations to an SVHN image. From <https://arxiv.org/abs/1805.09501v1>

Let's look at the search space for AutoAugment's RL model. They consider 16 operations: 14 from the Python image library PIL like rotating, color inverting and lesser-known operations like posterizing (reducing pixel bits) and solarizing (inverting colors above a threshold) plus the newcomers in the field of data augmentation Cutout and SamplePairing (which is similar to mixup). Adding 11 discretized probability values (0.0, 0.1, ..., 1) and 10 evenly spaced magnitudes from 0 to 9 this amounts to  $(16 * 11 * 10)^2$  possibilities for one sub-policy and  $(16 * 11 * 10)^{10} \approx 2.9 * 10^{32}$  possibilities for a policy of 5 sub-policies. Reinforcement Learning to the rescue!

## How is AutoAugment trained?

AutoAugment is trained like NASNet — an earlier AutoML paper from Google where the RL system found state-of-the-art model architectures for image classification. Here is how: We have a controller that decides which data augmentation policy seems best at the moment and tests the generalization ability of that policy by running a child model experiment on a small subset of a particular dataset. After the child experiment is finished the controller is updated with the validation accuracy as the reward signal, using a policy gradient method called Proximal Policy Optimization algorithm (PPO). Let's look at the controller and the child model experiments in more detail as explaining PPO is beyond the scope of this post.



Illustrative controller model architecture. Adapted from <https://arxiv.org/abs/1707.07012>



In total there are 30 softmax predictions, as there are 5 sub-policies and each sub-policy needs two decisions regarding operation, magnitude and probability ( $5 * 2 * 3 = 30$ ) for its two operations applied in sequence.

## The child model

How do we tell the controller which policies were selected very well and which were not really improving the performance (think of putting brightness to zero)? For that we run a generalization experiment with a child neural network that uses the current data augmentation policy. After the experiment finishes, the weights of the RNN controller are updated with the validation accuracy as a reward signal. This is done 15,000 times when finally the overall best 5 policies (each with 5 sub-policies) are merged into the final policy (now with 25 sub-policies). This final policy is used for all results of that dataset.

## Results

As the sub-title already gave away, AutoAugment improves the state-of-the-art on datasets like CIFAR-10, CIFAR-100, SVHN, ImageNet and others. Additionally there were some particularly interesting details:

- The best policies for CIFAR-10 and ImageNet feature mostly

• • •

## Double Transfer Learning

If we want to solve our own image classification problem, we usually start by using pre-trained weights from ImageNet and fine-tune these weights to our problem. We just saw that using the best ImageNet policy from AutoAugment while training from scratch had a similar positive effect. What if we do both: Finetuning ImageNet weights while using the ImageNet AutoAugment policy? **Would the positive effects add up and provide us with the new best way to tackle any new image classification problem?**

To answer this question I took the same 5 FGVC datasets (Oxford 102 Flowers, Caltech-101, Oxford-IIIT Pets, FGVC Aircraft and Stanford Cars) and fine-tuned an Inception v4 with and without applying the ImageNet policy from AutoAugment.

### Experimental setting

- An Inception v4 is trained on one GPU with batch size 32, SGD with momentum 0.9 and learning rate 0.01.
- The learning rate is divided by 2 if the validation accuracy did not increase for 5 epochs. Training is stopped after  $3 \times 5 = 15$  epochs of no validation accuracy improvement.

~ . 1 2 3 4 5 6 7 8 9

• • •

## How to apply the AutoAugment policies to your problem

I have created a **repository** with the best ImageNet, CIFAR-10 and SVHN policies from the appendix of the paper. Some implementation details are still unclear, but I am in contact with the authors and will update the repo as soon as I know more.

With this basic syntax a random sub-policy of the ImageNet policy is applied to a PIL image:

```
from autoaugment import ImageNetPolicy
image = PIL.Image.open(path)
policy = ImageNetPolicy()
transformed = policy(image)
```

To apply it as a PyTorch transform you would do something like this:

```
from autoaugment import ImageNetPolicy
data = ImageFolder(rootdir, transform=transforms.Compose(
    [transforms.RandomResizedCrop(224),
     transforms.RandomHorizontalFlip(), ImageNetPolicy(),
     transforms.ToTensor(), transforms.Normalize(...)]))
loader = DataLoader(data, ...)
```

## Conclusion

Thanks to Avinash Mishra and Tom Voltz.

Machine Learning

Image Recognition

Deep Learning

Artificial Intelligence

Google

## Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. [Watch](#)

## Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. [Explore](#)

## Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just \$5/month. [Upgrade](#)

[About](#)

[Help](#)

[Legal](#)