Automatic Augmentation of Vision Datasets

John Carter, Maxim Ramsay King, Sun Jin Kim, Mia Wang
{jac202, mrj2717, sk2521, yw21218}@imperial.ac.uk

Supervisor: Dr Pancham Shukla
Course: COMP70048, Imperial College London

May 3rd, 2022

1 Introduction

1.1 Introduction to the Problem Setting of Image Data Augmentation

Image classification is a type of classification task set within the realm of machine learning, whereby a model is fed images and tasked with categorising them in some way[5]. One of the simplest and most famous examples is the MNIST handwritten digit classification task [6], which consists of feeding 28x28 pixel greyscale images of handwritten digits to a machine learning model.

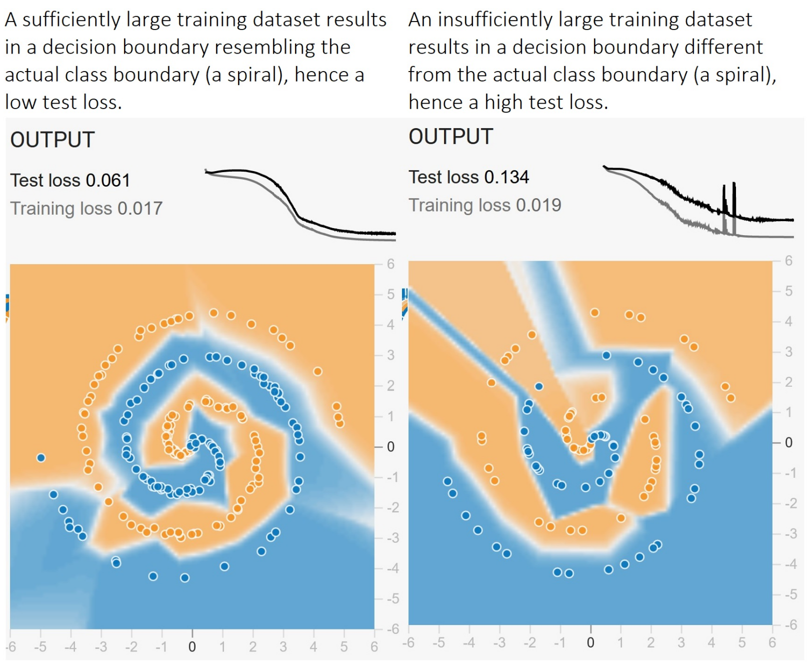


Figure 1: Demonstration of the impact a small training dataset has on the predictive power of a classification model. We see the predictions of a model with a sufficiently large training dataset (**Left**) against an insufficient dataset (**Right**).

Neural network classifiers can generalise to unseen data better if the size of our training dataset(the number of images that were manually labelled by humans) they were trained on was bigger. This is because it has seen seen more varied examples of the training data. The lack of training labelled images can lead to overfitting (Figure 1). The number of labelled images we have is often one of the biggest economic bottlenecks when developing machine learning pipelines, since we need humans to label images manually. Data augmentation, instead, provides a cheap solution to this problem for image classification [23].

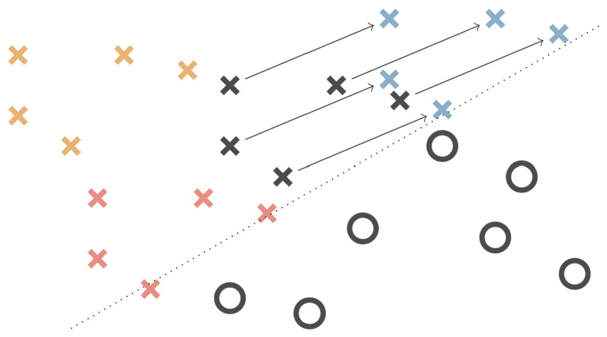


Figure 2: "Visualisation of a 2D dataset, with two classes (crosses and circles), separated by a decision boundary represented by the dotted line. Coloured crosses represent the effects of different augmentations applied to the 'cross' class." Taken from [22].

Data augmentation involves generating new, varied images from the original data, so that a model has a wider range of inputs to train from. Data augmentation for images involve applying image functions such as flipping, rotating, cropping, skewing, shifting, and adjusting colours [12]. This can be interpreted as generating additional data points within the decision boundary (Figure 2). These image functions can all be processed relatively quickly, compared to training the models, and so provides some cheap extra data which has shown to increase model accuracy in various settings [25].

What then becomes an interesting problem in machine learning is deciding which augmentations are likely to increase accuracy and which are redundant. Using MNIST as an example again, clearly, flipping training examples of the number 3 along the vertical axis will create augmented images that look nothing like a real number, and therefore will not add any value to a model. Similarly, flipping a number 4 along the horizontal axis also won't add any value. However, shifting, cropping, rotating (by a small number of degrees) and skewing may well add value and increase generalisability. In other words, the augmentation functions we choose need to be *class preserving*. In the context of Figure 2, this can be interpreted as finding a set of functions that map the cross class to the space above the dotted line, but not below it.

1.2 Introduction to the AutoAugment (AA) Methodology

The trick then becomes training a model to find such an appropriate set of augmentation functions, rather than having a human laboriously hand design data augmentation strategies for each dataset it comes across. This is where auto-augmentation [15] comes in. The idea being that a model can learn from a dataset which augmentations add the most value, and which are redundant. This is a hugely important area of research and there have been some excellent advances in image classification due to this particular idea [7, 30, 19, 20].

Cubuk et al. [15] was the first major paper in this area. I believe it is important for us to elaborate on this paper since it introduces us to:

- The search space. (The space of of image augmentation policies). The same search space or a modified version were used in most following papers in this subfield[7]. We have used a slightly smaller version of the original search space as well.
- Certain terminology that will be used throughout this report.

1.2.1 The AutoAugment Search Space

In this work there were 16 image functions available to apply on the dataset images. These include functions such as 'ShearX', 'Rotate', and 'Invert'.

The search space consists of around 10^{31} policies. A *policy* is a set(where ordering doesn't matter) of five subpolicies. A *subpolicy* is an tuple (where ordering *does* matter) of two operations. An *operation* is a 3-tuple of $(image\_function\_name, probability, magnitude)$. We set the $probability (= p)$ of application to be $0 \leq p \leq 1$ in increasing increments of 0.1, and the $magnitude (= M)$ to range from $0 \leq M \leq 9$.

An example of a policy would be:

```
[
(("Invert", 0.8, None), ("Contrast", 0.2, 6)),
(("Rotate", 0.7, 2), ("Invert", 0.8, None)),
(("Sharpness", 0.8, 1), ("Sharpness", 0.9, 3)),
(("ShearY", 0.5, 8), ("Invert", 0.7, None)),
(("AutoContrast", 0.5, None), ("Equalize", 0.9, None))
]
```

Note: The policy is represented as a list (which is order-dependent) on Python for convenience.

Also note: some image functions do not have a magnitude value, since it is not applicable. In these situations the magnitude is given as 'None'.

How the policy is applied to a dataset Before each image is fed into the convolutional neural network(CNN) classifier, one of the subpolicies of the policy is chosen uniformly randomly. Each operation is applied *in order*. Each operation is applied with the specified magnitude, but it is only applied with probability which is specified. This also means the identity operator can be expressed within this search space by letting $probability = 0$ for all operations.

The complicated policy-subpolicy structure adds expressivity of the search space. The element of randomness introduced by the random subpolicy and the p value ensures a greater diversity of images. In the context of Figure 2, this can be interpreted as the fact that we have more than one type of color of group of crosses, hence making our data point space more full, hence preventing overfitting as seen in Figure 1.

We used the same search space except with 14 kinds of image functions instead of the 16, since only 14 were available on the torchvision library.

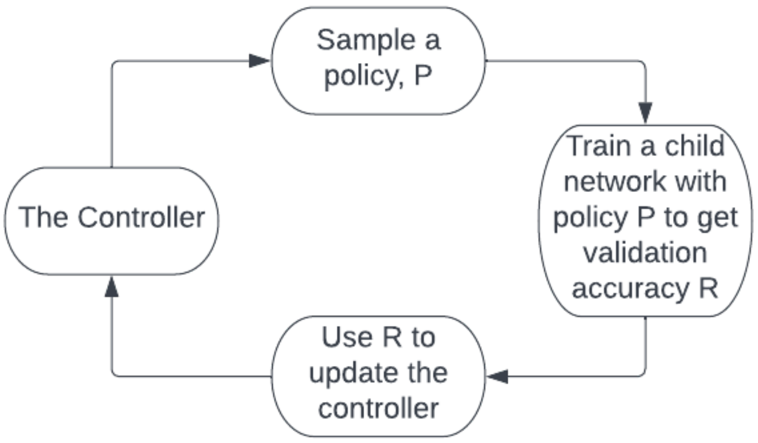

Sample a policy, P
Train a child network with policy P to get validation accuracy R
Use R to update the controller
The Controller

Figure 3: How a usual AutoAugment solution algorithm looks like.

1.2.2 Further AutoAugment terminology

The AutoAugment[15] paper also introduces terminology which is used throughout this project. See Figure 3

The algorithm which generates which policy to test is called the *controller*. The CNN classifier that we are training the training dataset and validation on is called the *child network*.

In the AutoAugment[15] paper, they ran the loop in Figure 3 1500 times. Then, they gathered the top 5 best policies and concatenated (set union) them into one big policy. This can be interpreted as increasing the number of colours we see in Figure 2.

In our project, we call this final large policy the *mega policy*.

1.3 Summary of our Contributions

There are several contributions made from our team in this project. Firstly, we have implemented new learners including: two reinforcement learning learners, one evolutionary strategy learner and one genetic algorithm learner. Secondly, we have implemented a web app which allows non-technical users to run these learners on predefined or their own custom datasets, where users can choose various hyperparameters instead of the default options. Thirdly, we have implemented a python library which allows technical users to get more hands on with the learners and make bigger changes such as using custom child networks, and will include adding their own custom learners in at a future date.

2 Specification

The role of our project is to provide easy-to-use methods for users to generate augmentations to datasets that improve the accuracy of their machine learning models. We began by considering the target userbase for our product.

2.1 Identifying Two Groups of Target Users

We identified two types of user: non-technical and technical/professional.

A non-technical user may have some images that they need to classify, and are looking to increase the accuracy through augmentation, but perhaps don't understand how deep learning works, or even what types of data augmentations are available, and so won't have expertise in this area to be able to apply expert knowledge to their dataset. There is no point allowing them to change the internal child network, or giving them unrestricted access to change hyperparameters, and instead will want to upload a dataset and then be given some augmentations which, when applied, will improve the accuracy.

A technical user, however, would want much more freedom to explore and apply their own expert knowledge. This would include using their own child network that they know works well/is best for their chosen dataset, adjusting default hyperparameters, removing augmentations they know won't add value (such as flips for MNIST), or even creating their own learner using the parent classes and methods available in the package.

2.2 How our product aims to meet their demands

In order to meet both these users' needs we decided on creating two products. A web app that would be used to meet the requirements of the non-technical user, and a package that would be used to meet the requirements of the technical user.

The web app meets the non-technical user's needs by allowing them to find data augmentations that increase the accuracy of their image classification models, without having to use any expert knowledge or get too heavily involved in how everything works. The web app is easy to use and self-explanatory.

The package consists of lots of classes and functions available to the technical user, which they can either adapt in its current format to include their own functions, changes and expert knowledge, or incorporate into their own code and add their own learners, make permanent changes, add meta layers and so on. The package forms the basis for the web app, so it is all consistent, but allows much more flexibility and is written in such a way that it will be easily readable and understandable for a technical user.

The package meets the technical user's needs by providing a raw base of readable code that offers sufficient flexibility for a technical user to adapt and change to suit their needs.

3 System Architecture

Figure 4 demonstrates the relationship between data and algorithms. Our product Auto-augmentation has interaction with both web app users and library users. And it is consist of three parts: the library, where our main algorithms are stored, which technical users can install on pip and customise their training hyperparameters for more personalised use; the react front-end, which enables less-technical users to experiment the impact of data auto-augmentation and allows them to generate a data augmentation policy to boost their CNN training performance; and the flask back-end, which is the interconnection between the front-end and the library.

The implementation steps for library and web app users are similar. There are four main steps that consist of: preparing the dataset they want to train and deciding on training parameters (e.g. child network, auto-augmentation learners); initiating the auto-augmentation training process; collecting the output policy after the training completes; and finally, applying the policy on the their original dataset to obtain an augmented dataset. The difference for these two uses is that the first three steps can be implemented on our web app with more user-friendly interface.

4 Python library – autoaug

4.1 Our starting point

We have built upon the neural network library provided by PyTorch, and the subsidiary computer vision library TorchVision.

For the auto augmentation learners in particular, we made use of a functionally incomplete module in the TorchVision library, `torchvision/transforms/autoaugment.py`. In particular we used the `AutoAugment` class within the module. An `AutoAugment` object is capable of applying a given policy (as defined in AutoAugment[15]) to a torchvision `VisionDataset`. However, it is not capable of deciding which policies to generate nor is it capable of updating its controller to make better decisions. It merely stores the good megapolicies found by the original authors of AutoAugment[15]. We used the `AutoAugment` class in all our auto-augmentation learners to apply the policies our controllers have popped out to the given dataset.

4.2 Structure

4.2.1 Overall structure

The library consists of three submodules: autoaugment_learners, child_networks, and controller_networks. child_networks is used only for demonstration and testing purposes, whereas the other two are core functionalities of our library.

autoaugment_learners contains the auto-augmentation learners: evolutionary learner, genetic learner, GRU learner, random search learner, and UCB1 learner.

child_networks contains various PyTorch written CNN classifiers(`torch.nn.Module`'s)

controller_networks contains neural networks which are used as the controllers in some of the auto-augmentation learners, namely: evolutionary learner and GRU learner.

4.2.2 Refactorisation

The auto-augmentation learners are refactored as to allow easier usage and easier development and contribution. See Figure 5.

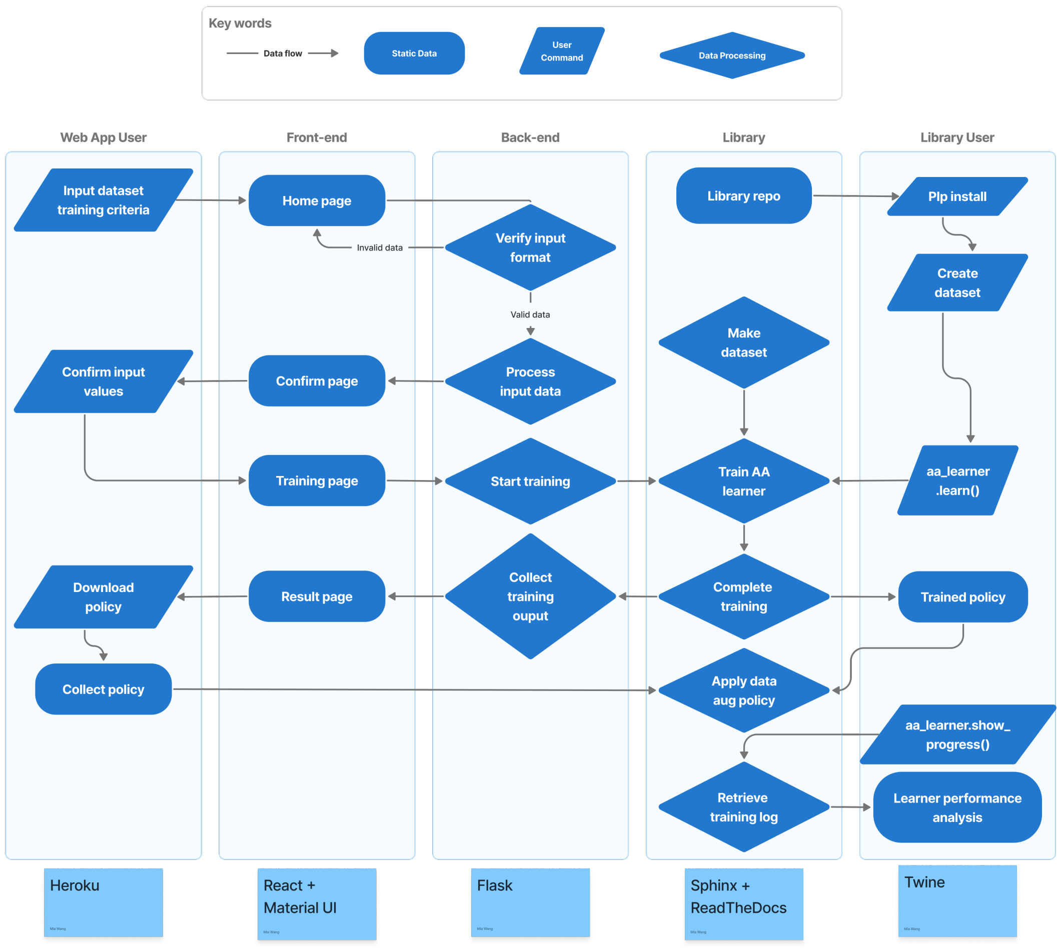


Figure 4: Architectural diagram of Auto-Augmentation web app and library. Note that 'AA learner' is an abbreviation of auto-augmentation learner.

We were inspired by the system adopted by SciKit-Learn[11] where for example, all regression models would have the methods `fit`, `predict`, and `score`, which means that in a user's code, one would only need to change one line of code if one were to change the model their script was using without worrying about whether any of the method calls of the object would break.

Our `AaLearner`'s (see Figure 5) share certain user friendly methods such as `.learn()`, `.get_n_best_policies()` `.get_mega_policy()`, as well as methods that make implementing new `AaLearner`'s easier such as `._test_autoaugment_policy()` and `._translate_operation_tensor()`.

4.3 Documentation using Sphinx and ReadTheDocs

"It doesn't matter how good your software is, because if the documentation is not good enough, people will not use it."

(Daniele Procida)

The documentation for this project was written in RestructuredText, generated by Sphinx, and hosted on the free documentation hosting website, readthedocs.org. The specific documentation for this project can be found here [2] .

The documentation has three of the four different types of documentation articles specified by Daniele Procida [4]: Tutorials (practical guides), explanations (theoretical explanations), and refer-

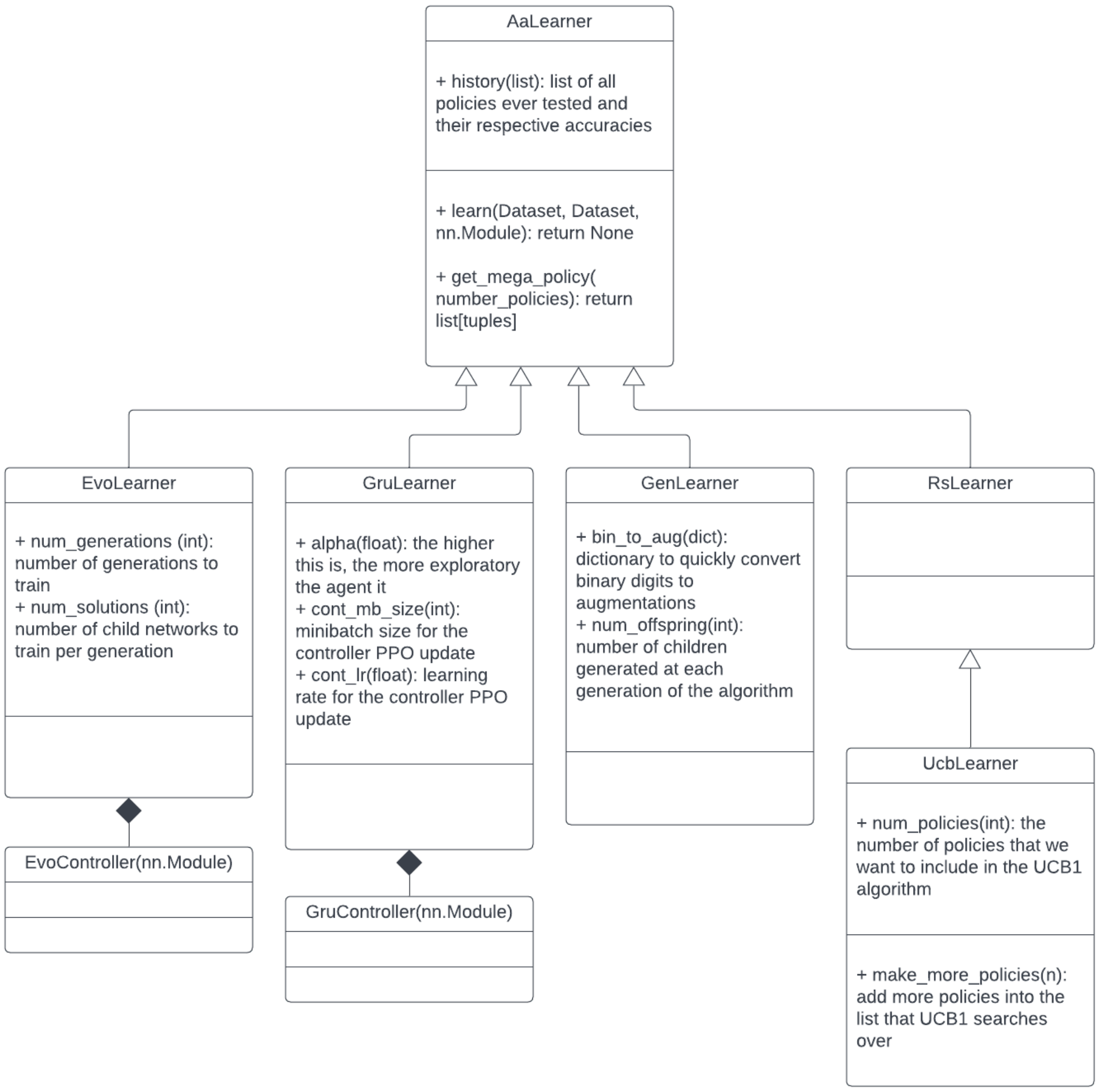

AaLearner
+ history(list): list of all policies ever tested and their respective accuracies
+ learn(Dataset, Dataset, nn.Module): return None
+ get_mega_policy(number_policies): return list[tuples]
EvoLearner
+ num_generations (int): number of generations to train
+ num_solutions (int): number of child networks to train per generation
EvoController(nn.Module)
GruLearner
+ alpha(float): the higher this is, the more exploratory the agent it
+ cont_mb_size(int): minibatch size for the controller PPO update
+ cont_lr(float): learning rate for the controller PPO update
GruController(nn.Module)
GenLearner
+ bin_to_aug(dict): dictionary to quickly convert binary digits to augmentations
+ num_offspring(int): number of children generated at each generation of the algorithm
RsLearner
UcbLearner
+ num_policies(int): the number of policies that we want to include in the UCB1 algorithm
+ make_more_policies(n): add more policies into the list that UCB1 searches over

Figure 5: A UML diagram illustrating the refactorisation done in `autoaug/autoaugment_learners/`.

ences (technical specifications of the library). It is designed to be both friendly enough for new users and thorough enough for potential contributors.

The code for the tests can be seen in `./docs/`

4.4 Theory behind the AutoAugment Learners

For the pseudocode for each learner, see our documentation.

Upper Confidence Bound (UCB1) learner UCB1 is often described as optimism in the face of uncertainty. We are trying to find the action with the highest reward given some uncertainty of the current expected rewards (q-values). As the number of times an action has been taken increases (relative to the total number of actions taken), its outcome becomes more certain. Instead of looking at raw q-values, the algorithm takes the action counts into account, which can be seen in Eqn. 1;

$$q_i + \sqrt{\frac{2ln(n)}{n_i}}, \tag{1}$$

where i represents an action, q_i is the associated q-value, n is the total action count, and n_i is the action count. UCB1 is a typical algorithm used in multi-arm bandit problems [17] and, since the auto-augmentation problem can be rephrased as a multi-arm bandit problem (where each arm is applying an augmentation and each return is an accuracy score on a validation set), it is simple to use UCB1 as a starting case before moving to more complex algorithms. UCB1 is easy to understand and easy to implement but does not use neural networks, and so offers no real way to generalise between different types of augmentations.

It works by keeping track of how many times each augmentation has been attempted, and adjusting the raw q-values (mean accuracy on validation set) by these counts (seen in the formula above). This new adjusted q-value is then used to decide which augmentation is applied in the next iteration (that is, which bandit arm is pulled).

Due to the fact UCB1 doesn't generalise between types of augmentation, it is quite slow compared to methods that involve neural networks (for example GRU). For this reason, we used a similar methodology to the AutoAugment paper[15] however we limit the search space to 5 policies each with 5 subpolicies (the original paper has 14,000 policies each with 5 subpolicies). The raw accuracy on a fixed validation set is fed in as the reward.

UCB1 is very good at balancing the exploration-exploitation trade-off in an efficient manner that minimises regret, which is why it's often the go-to algorithm for multi-arm bandit problems.

Random Search learner This learner is a purely randomised searcher. Despite this, it is a hard baseline to beat, as it is in many automatic hyperparameter tuning problem settings[13]. Each subpolicy is found by the following process; an augmentation is chosen from the available selection with an equal probability, followed by a probability from 0.0 to 1.0 in incrementations of 0.1, which, again, are equally likely. If the selected augmentation requires a magnitude, we use Python's random module to select a random integer between 0 and 9 inclusive, otherwise the magnitude is set to None. This is repeated to generate a complete subpolicy, and can be joined into a policy to be tested. The corresponding accuracy of the child network is then stored with the policy for logging.

Genetic learner The genetic learner has similar elements to the Random Search learner, but uses information from previous sub-policies when generating new ones to more efficiently search for optimal augmentation parameters.

In this algorithm each subpolicy is represented as a binary string. For example the subpolicy ((ShearX, 0.9, 3), (TranslateY, 0.1, 7)) would be represented as '000010010011001100010111' where each augmentation, probability, and magnitude is parsed as a 4-bit long digit. We then rank the effectiveness of a policy by the accuracy of the trained child network implementing it, and carry this by storing tuples of (policy, accuracy). We randomly select parents as previous policies, where the probability of selection is weighted by the corresponding accuracy. A child is then produced using the random crossover method, and this is taken as the next policy to test. In the case that the binary number produced from this method did not correspond to an available augmentation, probability, or magnitude, or if edge cases were found (e.g. an augmentation with required magnitude None was assigned a float value), such cases were resolved with uniform random selection.

GRU with PPO updating Agent (gru_learner.py) This model nearly identical to what was used in Cubuk et. al [16], where the authors borrowed the model from this Zoph et. al [31]. In their works an LSTM controller was used, which output a policy in the form of a length 10 sequence of vectors, each vector representing an operation. The LSTM controller was updated using proximal policy optimization[24](PPO), using the accuracy of the child network as the reward value.

In the context of the PPO update, which was developed in the reinforcement learning literature, the subpolicies are the 'actions' of our RL agent.

We use GRU instead of LSTM because it's faster while empirically being functionally equivalent[14].

For more details on how an RNN (LSTM or GRU) network can be updated using PPO see Section 3.2 of Zoph et. al [31]. For more details on the hyperparameters of the LSTM they used during training, see 'Training Details' in the same paper.

Evolutionary learner The Evolutionary learner is similar in principle to the Genetic learner except that instead of the subpolicy being expressed as genes, the genes are expressed via the weights of a neural network. In this work PyGad [8] was used, which allows an easy implementation of such an evolutionary approach that is compatible with PyTorch networks. For this learner a child network and controller are required. The architecture of the child network must be able to pass images and classify them, whereas the controller network must be able to take the same input images and output a policy. In this work, this output consisted of the number of functions used in the search, in addition to 2 nodes for the probability and magnitudes respectively, and then doubling this to account for a subpolicy consisting of pairs of transformations.

In order to calculate which augmentations were optimal, we considered the strength of pairs of augmentations, as well as the presence of those exact pairs in the network output. We first found the covariance of the outputs for the first and second augmentations, and chose the most strongly correlated pair of augmentations. We then returned to the batched network output to find all instances where these pairs of augmentations were chosen, and found the mean magnitude and probability from this. This could be repeated to generate multiple different subpolicies, at which point it was passed to the child network for training.

4.5 Testing with pytest

Both unit tests and integration tests were made with pytest[9], and run automatically on GitLab CI/CD. The code for the tests can be seen in `./test/`

4.6 Upload onto PyPip

To make the experience of the technical user smoother, we uploaded our augmentation library to PyPip [1], so that the library can be easily installed and used. The reader is welcomed to install this themselves to test any aspects of the product. The library can be installed simply by

```
pip install auto-augmentation
```

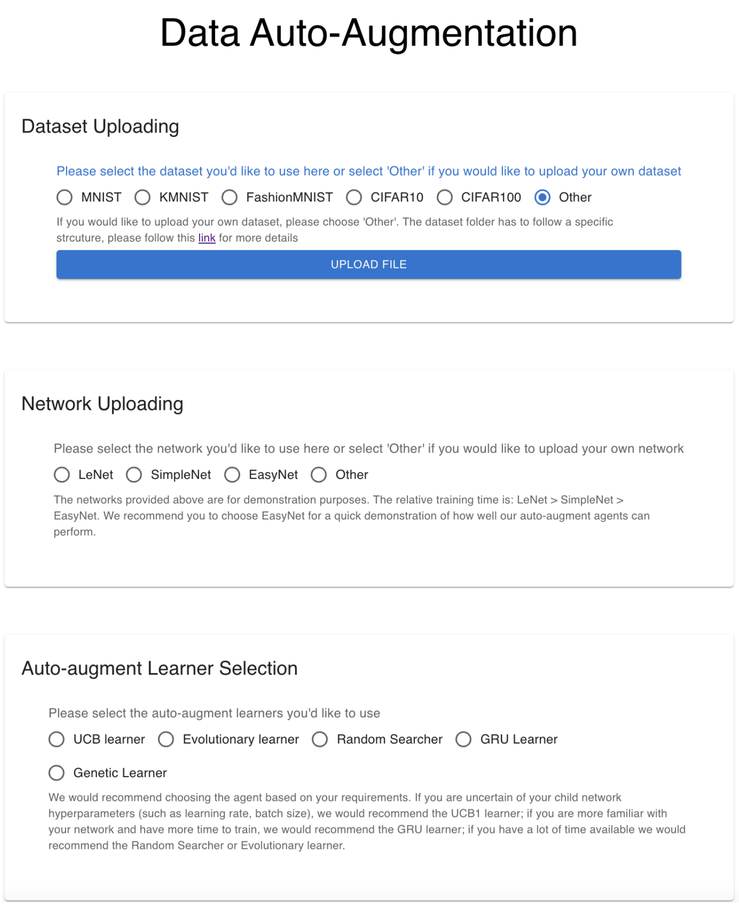
5 Web App

5.1 Frontend

Originally vanilla HTML was used to generate the web app, but we updated this using the React [10] Javascript framework in order to improve user experience and usability, as well as the aesthetic appearance of the application.

Figure 6 shows the homepage of the Web App. This gives the user the option to use some default datasets (available through PyTorch) or upload their own. The child network that is used to train on the dataset can also be selected, or uploaded by the user. LeNet is a standard deep learning network used for image classification and performs well on the default datasets, although is slow relative to EasyNet, which is our own network with just one layer. They can then select which of our learners they want to use. We also provide the option to change some hyperparameters that could greatly effect the ability of the learners. These are the 'Advanced Settings' that can be seen in Figure 6 **Top** and include changing the batch size, the learning rate, the number of iterations the learner trains for, the proportion of the dataset which is used for training and, perhaps most importantly, which type of augmentations they want to exclude.

After the options have been selected, the user is redirected to a confirmation page to ensure the correct settings have been selected, which can be seen in Figure 7 **Bottom**. Once confirmed, the model



Data Auto-Augmentation

Dataset Uploading

Please select the dataset you'd like to use here or select 'Other' if you would like to upload your own dataset

MNIST KMNIST FashionMNIST CIFAR10 CIFAR100 [✓ Other]

If you would like to upload your own dataset, please choose 'Other'. The dataset folder has to follow a specific strcuture, please follow this link for more details

UPLOAD FILE

Network Uploading

Please select the network you'd like to use here or select 'Other' if you would like to upload your own network

LeNet SimpleNet EasyNet Other

The networks provided above are for demonstration purposes. The relative training time is: LeNet > SimpleNet > EasyNet. We recommend you to choose EasyNet for a quick demonstration of how well our auto-augment agents can perform.

Auto-augment Learner Selection

Please select the auto-augment learners you'd like to use

UCB learner Evolutionary learner Random Searcher GRU Learner

Genetic Learner

We would recommend choosing the agent based on your requirements. If you are uncertain of your child network hyperparameters (such as learning rate, batch size), we would recommend the UCB1 learner; if you are more familiar with your network and have more time to train, we would recommend the GRU learner; if you have a lot of time available we would recommend the Random Searcher or Evolutionary learner.

Figure 6: WebApp Homepage showing the non-advanced, required settings. Note that upon selecting 'Other' in the Dataset or Network Uploading sections an upload file button appears.

begins training and, once finished, returns results and the option to download the augmentation policy to the user. These can then be used in their own models (see Figure 8).

5.2 Backend

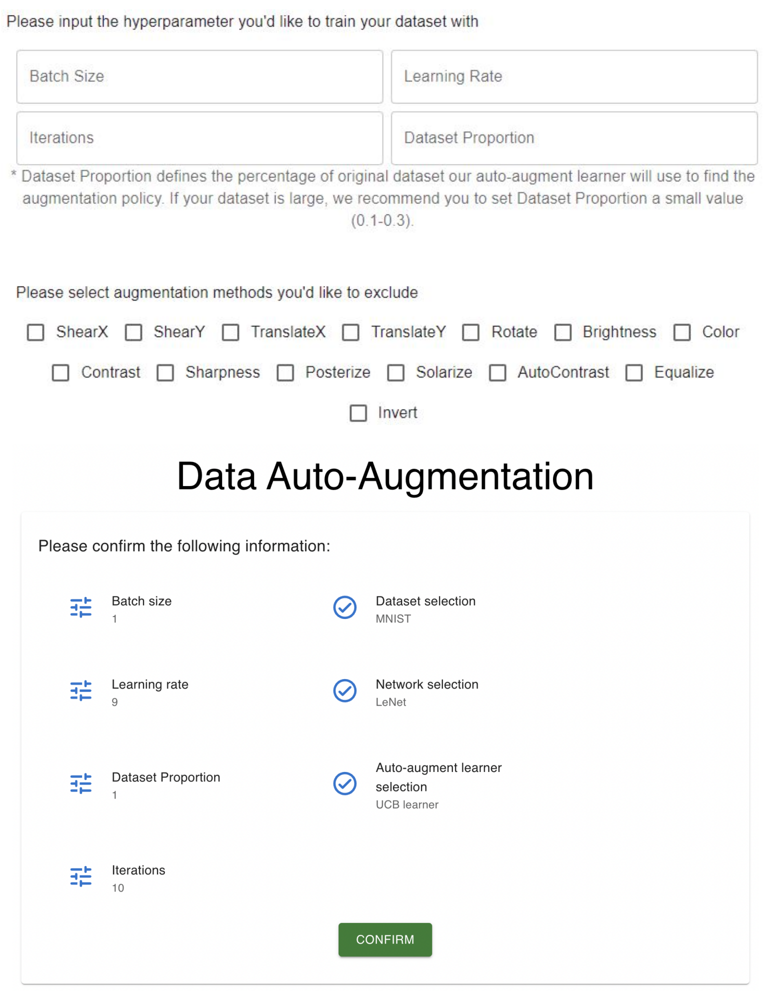
For this project, Flask [3] was used as a Python backend to run the user queries, and was beneficial for multiple reasons. Firstly, it allowed us to easily create the app and run get requests from the website, along with simple debugging tools that made the development process easier. Secondly, it was the most seamless way to interact with our Python library to run our learners.

6 Evaluation

6.1 Performance against baseline

Figure 4 shows the final results of running the various learners on the FashionMNIST dataset with EasyNet as the child network (EasyNet is our own simple neural network with just one layer). The baseline is the mean of 10 iterations of running EasyNet with no data augmentations on the full dataset, that is, a very simple classification problem. As can be seen from the figure, UCB1 outperforms the baseline by 1 standard deviation (equivalent to around 9%) however GRU (based on [15] but swapping the Long-Short-Term Memory (LSTM) for GRU) and RandomSearcher outperform the baseline by 2 standard deviations (equivalent to around 20%).

Figure 5 shows the results when comparing learners against the baseline for the CIFAR10 dataset, using LeNet as the child network. Here the results are even more impressive. UCB1 beats the baseline



Please input the hyperparameter you'd like to train your dataset with

Batch Size	Learning Rate
Iterations	Dataset Proportion

* Dataset Proportion defines the percentage of original dataset our auto-augment learner will use to find the augmentation policy. If your dataset is large, we recommend you to set Dataset Proportion a small value (0.1-0.3).

Please select augmentation methods you'd like to exclude

☐ ShearX ☐ ShearY ☐ TranslateX ☐ TranslateY ☐ Rotate ☐ Brightness ☐ Color
☐ Contrast ☐ Sharpness ☐ Posterize ☐ Solarize ☐ AutoContrast ☐ Equalize
☐ Invert

Data Auto-Augmentation

Please confirm the following information:

Batch size: 1	✓ Dataset selection: MNIST
Learning rate: 9	✓ Network selection: LeNet
Dataset Proportion: 1	✓ Auto-augment learner selection: UCB learner
Iterations: 10	

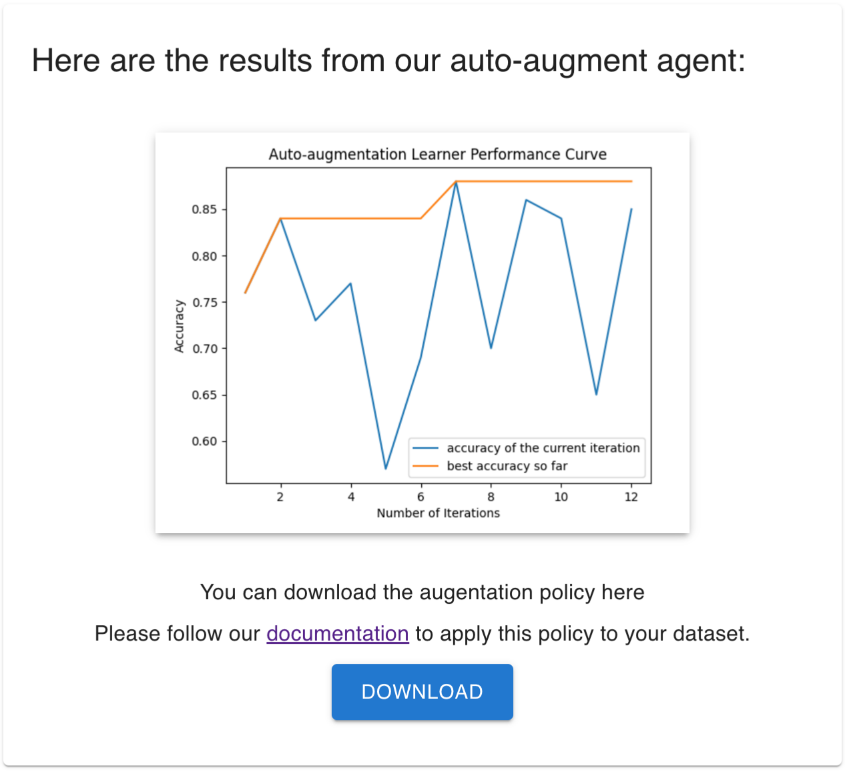
Figure 7: **Top**: Advanced settings section of the WebApp Homepage. **Bottom**: Example confirmation page once the initial settings have been selected.

by 2 standard deviations (equivalent to around 3%) whereas GRU and RandomSearcher beat the baseline by 4 standard deviations (equivalent to around 6%).

UCB1 is great for multi-arm bandit settings however that means it doesn't offer much ability to generalise between policies. For example, it can't tell that a rotation of 10 degrees and a rotation of 11 degrees are similar and might yield similar increases in performance. As a result, this makes it very slow compared to a neural network approach, like GRU. Only 5 policies were tested in the above results and yet 14000 were tested in the original AutoAugment paper[15]. It would be infeasible to increase UCB1 to anywhere near this number and so, although it did increase accuracy versus the baseline, its performance will always lag a neural network approach. That said, UCB1 is very easy to understand and explain the results, and is a tried and tested method for bandit problems that can be explained without a 'black box'.

The GRU approach allows memory to play a part and so allows better generalisation between policies. In this setting, the learner will realise that rotating by an extra degree yields similar results. This greatly increases the number of policies that can be searched over and therefore increases the chances of finding better policies. GRU is more complex than UCB1 to explain and understand, however a technical user who has familiarity with recurrent neural networks and understands reinforcement learning should easily be able to understand this learner. It would be more difficult for a non-technical user to understand.

We also show the performance of child networks when utilising the 5 best subpolicies of our learners. These can be seen in Figure 11, where we have used all learners to generate policies for the simpler FasionMNIST (**Left**) dataset, and more complex CIFAR10 (**Right**) dataset and compared


Auto-augmentation Learner Performance Curve
Accuracy
Number of Iterations
accuracy of the current iteration
best accuracy so far
DOWNLOAD

Here are the results from our auto-augment agent:

You can download the augentation policy here

Please follow our documentation to apply this policy to your dataset.

Figure 8: Example Results page after the training of the learner has completed.

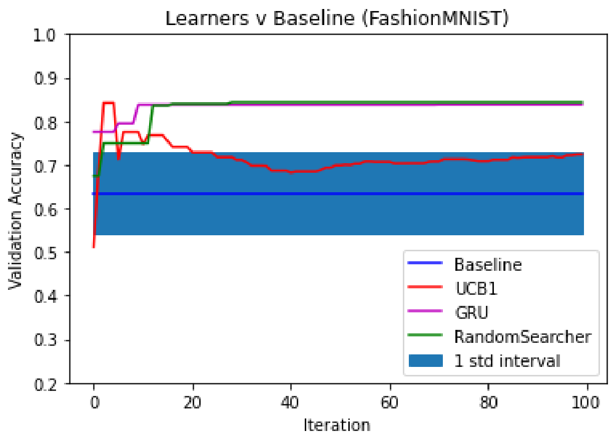


Figure 9: Learners versus baseline on FashionMNIST with EasyNet child network

these to their respective baselines. We see that for the FashionMNIST, all learners improve the average accuracy, apart from the random searcher, but the Evolutionary and GRU learners (which both use neural networks) introduce more variance to these accuracies. Conversely, in the case of CIFAR10, all learners improve the accuracy compared to the baseline. We also see that, due to the complexity of the dataset, each learner still retains uncertainty in their accuracies.

6.2 Novelty of methodologies

The `autoaug` library introduces several novel auto-augmentation algorithms to the literature: UCB1 and evolutionary algorithm. It also implements already tried algorithms in different ways: we implement our own form of the genetic algorithm and we made our own form of the RNN learner using a GRU network instead of the LSTM in [15]. We also implemented the random search learner as

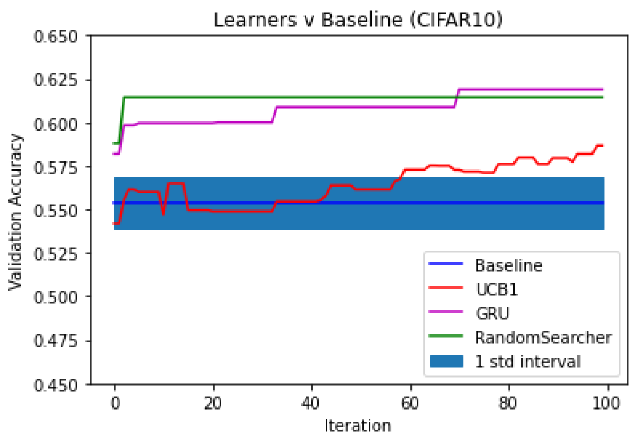


Figure 10: Learners versus baseline on CIFAR10 with LeNet child network

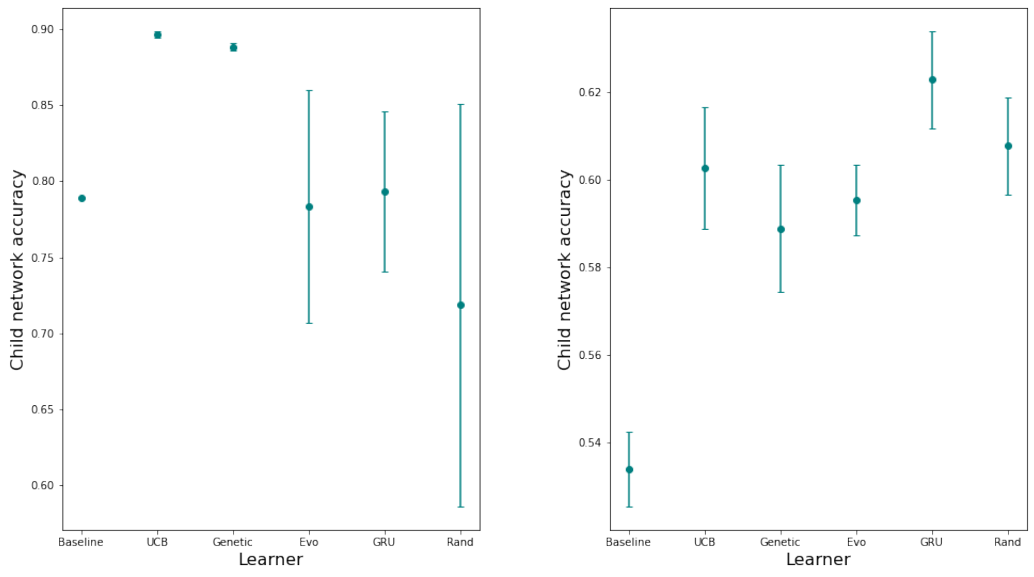


Figure 11: Performers of various learners against the baseline for the FashionMNIST dataset **Left** and CIFAR10 **Right**, averaged over 10 repetitions. EasyNet was used for all learners in FashionMNIST, and LeNet for CIFAR10. Note that for both plots, the error bars represent the $\pm 1\sigma$ range, which may be too small to be seen.

suggested by [15] to act as a strong baseline against other learners.

Forms of UCB1 have been used and combined with other methods for auto-augmentation in very few settings[18, 21]. Our product has shown that it can be used in its raw form to improve performance on classification problems versus non-augmented datasets.

6.3 Meeting user's needs and testing robustness

Working closely with the external supervisor from start to finish, we ensured at each stage that what we were delivering was something that, firstly, they wanted and, secondly, would be useful to them. This involved several iterations of the product, adding features, stopping dead avenues and finessing the product. We have ended up with two very useful products that fit both type of user's needs.

The products have been tested thoroughly using unit testing where possible, and end-user style testing thereafter. GitLab was used so that each time a new feature was added, it could be tested thoroughly and then easily go back to a previous version if any problems arose. This allowed smooth integration as the project progressed.

6.4 Future work

The most immediate extension to our work that we can implement is to add other types of learners. Some progress was made with an actor-critic RL learner, however, it wasn't fully tested before the final product release. Another potential improvement would be the introduction of a learner that does not prioritise the accuracy of the trained network as the optimal solution, but instead on the variance of policies, such as a learner based on quality-diversity. Other state-of-the-art methodologies [30, 19, 20] could also be incorporated into the library, as they offer significantly quicker training times and improved accuracies.

One of the original goals of the project was to apply meta reinforcement learning [26] to the auto-augmentation problem[28, 27, 29]. In our current framework we have a child network that classifies images, and augmentation learners that, based on the dataset images, derives optimal augmentations to improve the accuracy of the child network. A MetaRL learner would be taking a further step back in abstraction and retain information about seen datasets and their augmentations to improve learning on unseen examples. There are many possible avenues on how this could be approached, including: having a neural network sit above the learners that optimises their hyperparameters, using a meta-learner that uses previous experience to pick better augmentations based on new images shown to the learner, or a meta-learner that decides which of available learners would be best for the problem shown. This is definitely something that we would've liked to explore however we felt it would have involved such a large amount of computation time that it wouldn't be feasible for this short-term project.

There is also potential for rolling this type of software out to not just images but also videos, sound data and other time series data, as well as augmenting text for natural language processing. These are further avenues the team would have liked to explore if time permitted.

Lastly, the algorithms on the WebApp currently only run on CPU not GPU, due to Heroku only supporting this option. This would be something to address in the future to greatly reduce the training times for the child networks, allowing for a more enjoyable user experience. As we were not able to receive GPU credits in time for deployment, we could gain access to GPUs either through paying for access or through a specialised server. Note that this is only a problem for the WebApp, as the library currently uses a GPU if available.

References

[1] Auto-augmentation pypi website. `https://pypi.org/project/auto-augmentation/0.1/`. Accessed: 2022-05-02.

[2] Auto-augmentation readthedocs documentation. `https://autoaug.readthedocs.io/en/latest/index.html#`. Accessed: 2022-05-02.

[3] Flask framework. `https://flask.palletsprojects.com/en/2.1.x/`. Accessed: 2022-05-02.

[4] The four kinds of documentation and why you need to ...

[5] Image Classification Explained: An Introduction.

[6] MNIST classification | TensorFlow Quantum.

[7] Papers with code - imagenet benchmark (data augmentation).

[8] Pygad. `https://pygad.readthedocs.io/`. Accessed: 2022-04-30.

[9] Pytest. `https://docs.pytest.org/en/7.1.x/`. Accessed: 2022-05-02.

[10] React framework. `https://reactjs.org/`. Accessed: 2022-05-02.

[11] Scikit. `https://scikit-learn.org/stable/`. Accessed: 2022-05-02.

[12] Data Augmentation | How to use Deep Learning when you have Limited Data, May 2021.

[13] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.

[14] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[15] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. AutoAugment: Learning Augmentation Policies from Data. *arXiv:1805.09501 [cs, stat]*, April 2019. arXiv: 1805.09501.

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

[17] kexugit. Test Run - The UCB1 Algorithm for Multi-Armed Bandit Problems.

[18] Byungchan Ko and Jungseul Ok. Adaptive Scheduling of Data Augmentation for Deep Reinforcement Learning. page 11.

[19] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast AutoAugment. *arXiv:1905.00397 [cs, stat]*, May 2019. arXiv: 1905.00397 version: 2.

[20] Tom Ching LingChen, Ava Khonsari, Amirreza Lashkari, Mina Rafi Nazari, Jaspreet Singh Sambee, and Mario A. Nascimento. UniformAugment: A Search-free Probabilistic Data Augmentation Approach. *arXiv:2003.14348 [cs]*, March 2020. arXiv: 2003.14348 version: 1.

[21] Zhengying Liu. Automated Deep Learning: Principles and Practice. page 241.

[22] Samuel G. Müller and Frank Hutter. Trivialaugment: Tuning-free yet state-of-the-art data augmentation, 2021.

[23] Yuji Roh, Geon Heo, and Steven Euijong Whang. A Survey on Data Collection for Machine Learning: a Big Data – AI Integration Perspective. *arXiv:1811.03402 [cs, stat]*, August 2019. arXiv: 1811.03402.

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[25] Connor Shorten. A survey on Image Data Augmentation for Deep Learning. page 48, 2019.

[26] Joaquin Vanschoren. Meta-learning: A survey, 2018.

[27] Joaquin Vanschoren. Meta-Learning: A Survey. *arXiv:1810.03548 [cs, stat]*, October 2018. arXiv: 1810.03548.

[28] Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv:1611.05763 [cs, stat]*, January 2017. arXiv: 1611.05763.

[29] Lilian Weng. Meta Reinforcement Learning, June 2019. Section: posts.

[30] Yu Zheng, Zhi Zhang, Shen Yan, and Mi Zhang. Deep AutoAugment. *arXiv:2203.06172 [cs]*, March 2022. arXiv: 2203.06172 version: 2.

[31] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.