Automating Route Setting in Climbing via Deep Generative Models

Baris Sevilmis, Berk Mandiracioglu, Onur Veyisoglu
baris.sevilmis@epfl.ch, berk.mandiracioglu@epfl.ch, onur.veyisoglu@epfl.ch
Department of Computer Science, EPFL, Switzerland

***Abstract*—MoonBoards [1] are becoming a more and more common rock climbing training tools in gyms around the world. Through a dedicated app, climbers can share climbing "problems" with fellow users, and with thousands of boards available in climbing gyms around the world, a large quantity of graded routes (>8k) is freely available online. In this project, we propose to make use of this data to automate climbing-route generation. To achieve this goal, we started by investigating the quality of our data by training supervised learning models on it, such as classifiers and regressors. Assessed this, we moved to route climbing generation. Specifically, we have implemented Conditionally Generative Adversarial networks [2]. As assessing the quality of generated routes is challenging for inexperienced users, we propose to make use of the supervised models above to assess the quality of generated problems. This project was offered by CVLAB at EPFL under the supervision of PhD candidate Edoardo Remelli.**

I. Introduction

After being introduced into the upcoming Olympic Games, rock climbing is becoming more and more popular around the world. In order to keep the customers of climbing gyms challenged, climbing routes need to be changed often. In this project, we investigate the feasibility of automating route-setting by making use of unsupervised learning techniques.
As a first step towards this goal, we considered a simplified setting where the topology of both the wall and the climbing holds is fixed, such as MoonBoards [1]. Plotted against a grid of lettered and numbered coordinates (see Figure 1), each board hold is rotated and set in a specific location, allowing us to represent Moonboards conveniently as tensors.
Exploiting data freely available online, we implemented Conditional Generative Adversarial Networks (CGAN) and Deep Convolutional Adversarial Networks(DCGAN) to learn a generative model capable of, given a user specified grade, generate a climbing route of that difficulty. Furthermore, as validating our models was also crucial for the purpose of determining how well our GAN models were performing, we have exploited available data to implement classifiers and regressors to observe the difficulty level of newly generated climbing routes.
Implementation of the project is done mainly in Python[3]. In particular, Numpy[4], Scikit Learn[5] and PyTorch[6] libraries are utilized to build learning models and validate them.

II. Dataset

In our project we have used 2017 Moonboard data, which contains climbing problems labelled within 16 classes of difficulty. Data was directly obtained from the Moonboard website [1]. Difficulty values are entered by users when logging a climb into the Moonbard application and are as follows: 6A, 6A+,6B, 6B+, 6C, 6C+, 7A, 7A+, 7B, 7B+, 7C, 7C+, 8A, 8A+, 8B, 8B+ where 6A is the easiest level and 8B+ is the hardest level. We used available meta-data (number of repetition of the route) to filter out invalid samples. In total, we have obtained 8397 climbing problems. A sample from the Moonboard dataset is given in Figure 1.

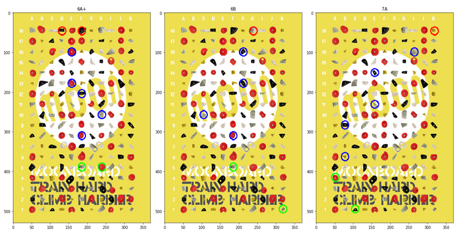

Fig. 1. Samples from 2017 MoonBoard Dataset. Climbers begin by holding on to the green circled holds. They are then allowed to use all blue circled holds and have to reach the red circled holds to complete the climb.

2017 Moonboard Dataset is unbalanced with respect to the difficulty levels. To be more precise, easier levels such as 6A to 6C are more common in comparison to higher difficulty levels. In this case, undersampling of the easier boards would be possible. However, undersampling some of the classes would result in insufficient amount of data to train our model, as harder difficulty climbs are merely existing in the dataset. As a consequence, we decided to use all available data to train our models.
As Moonboards consist of a regular 11×18 grid of holds, Euclidean grids are the most natural way to represent them within a Machine Learning framework. Specifically, as we have 3 types of holds (start, climb and top, refer to Figure 1 for more details), we simply represent climb x as

$$x = \{0, 1, 2, 3\}^{11\times18}. \quad (1)$$

To cope with the fact that moonboard data is unbalanced, three different ways of representing labels have been attempted, that are described by the distinct *MODE*'s in following subsections:

A. *MODE 0*

In *MODE 0*, existing moonboards have been concatenated according to the first character within their complete difficulty label. For instance, in this mode, 6A, 6A+, 6B, 6B+, 6C, 6C+ are all demonstrated as the same difficulty level. Objective of MODE 0 is to compensate the level of imbalance in dataset by decreasing difficulty accuracy. Moreover, observing the impact of reduced amount of classes on classification, regression and generation accuracy has crucial importance.

B. MODE 1

In *MODE 1*, moonboards have been aggregated by their first two characters with regards to their difficulty label. In other words, labels as 6A & 6A+ or 7B & 7B+ are considered as the same difficulty level. Underlying reason is to observe effects of the trade-off between the difficulty levels of moonboard routes and the possible degree of classification, regression and generation of moonboard routes.

C. MODE 2

Lastly, *MODE 2* is used as the original variation of dataset, in which authentic difficulty representations are being utilized as each character of difficulty label combination depicts a distinct challenge level. Underlying reason is to observe influences of unique labeling in the provided moonboard dataset. To put it in a different way, 16 distinct class labels are ensured in to the learning algorithms.

III. Models and Methods

As the moonboard dataset has been transformed into requested formats mentioned in Section II, development of learning algorithms are prioritized. Progress occur in two main phases, and the very first applied machine learning methodology is based on various classification and regression models. Second phase mainly concentrates on the route setting generation, in which particular Generative Adversarial Networks are being utilized. In the following subsections, these phases are explained in details.

A. Classification and Regression

To begin with, Following 5 particular classifier models have been trained on the moonboard dataset:

Classifier Models: $MODE \in [0, 2]$
- Pytorch Dense Network(PYT)
- Multilayer Perceptron(MLP)
- K Nearest Neighbours(KNN)
- Support Vector Machine (SVC)
- Decision Tree(DT)

In addition, following 4 distinctive regression models have been applied on the moonboard dataset:

Regressor Models: $MODE \in [0, 2]$
- Multilayer Perceptron(MLP)
- Random Forest Regressor(RF)
- Linear Regression(LR)
- Ada Boost Regressor(AB)

Both classifier and regressor models have been trained on moonboard dataset for each $MODE(\in [0, 2])$. PYT has been implemented with Pytorch [6], and rest of the models are implemented with help of Scikit-learn [5]. Hyperparameter tunning of each model is explained in the further subsections.

B. Cross Validation (K-fold) & Hyperparameter Tuning

Cross validation is not only used as a method to tune hyperparameters of classifiers and regressors, but also is applied to the classifiers as a validation method. K-fold size is chosen as 3-fold for the sake of faster training, otherwise for larger k-fold amount ratio of robust accuracy score to training time drops exponentially. In other words, using 3-fold cross validation for classifiers ensured the most optimal accuracy-time trade-off.
Hyperparameter tuning is applied to all classifiers and regressors. Apart from PYT, each scikit-learn based classifier and regressor have been tuned by built-in function, namely *GridSearchCV*, that benefits from cross validation method. On the other hand, PYT has been tuned by using GridSearch as well, but tuning was written manually. To avoid further complexity for PYT, best set of parameters were handpicked after GridSearch and GridSearch was removed from its implementation. Set of hyperparameters for each algorithm are depicted in the following Table I and Table II.

TABLE I
Classifier Hyperparameters

Classifier	*MODE 0*	*MODE 1*	*MODE 2*
MLP	ReLU	ReLU	ReLU
	Hidden: 100	Hidden: 50	Hidden: 100
	LR: 0.005 Adaptive	LR: 0.005 Adaptive	LR: 0.005 Adaptive
	SGD	SGD	SGD
KNN	Manhattan	Manhattan	Gini
	N: 15	N: 15	N: 15
	Distance	Distance	Distance
DT	Gini	Entropy	Gini
	Depth: 15	Depth:10	Depth:5
SVC	C:1	C:10	C:10
	Linear	RBF	RBF
PYT	Epoch: 100	Epoch: 100	Epoch: 100
	ReLU	ReLU	ReLU
	(1024,256)	(1024,256)	(1024,512)
	Adam	Adam	Adam
	LR: 1e-6 Adaptive	LR: 1e-6 Adaptive	LR: 1e-6 Adaptive

TABLE II
Regressor Hyperparameters

Regressor	*MODE 0*	*MODE 1*	*MODE 2*
MLP	ReLU	ReLU	ReLU
	Hidden: 50	Hidden: 100	Hidden: 100
	LR: 0.005 Adaptive	LR: 0.001 Adaptive	LR: 0.001 Adaptive
	SGD	SGD	SGD
LR	Normalize	Normalize	Normalize
	Fit_Intercept	Fit_Intercept	Fit_Intercept
AB	LR: 0.05	LR: 0.1	LR: 0.1
	Loss: Exp.	Loss: Exp.	Loss: Exp.
	N_Estim: 50	N_Estim: 50	N_Estim: 50
RF	Depth: 50	Depth: 50	Depth: 50
	Feat: Sqrt	Feat: Sqrt	Feat: Sqrt
	Min_leaf: 1	Min_leaf:1	Min_leaf:1
	Min_split: 5	Min_split: 5	Min_split: 5

C. *Generative Adversarial Networks (GAN)*

Generative Adversarial Network(GAN) are mainly used to generate a set of requested objects. A regular GAN model consists of a generator and discriminator, which are both deep neural networks.
For the generator part, input consists of latent features. Latent features denote low dimensional feature vectors, and these features are fed into conditional generator as well as the actual training labels. These low dimensional latent features are mainly generated from random distributions and, for the sake of the project, Gaussian distribution is utilized. In our case, activation function for the generators is selected as ReLU and generator architecture is mainly built upon upscale layers. In the end, output of the generator is produced by the TanH activation function, where the output size needs to be set to the same size as the actual input size.
Objective of the discriminator model is to distinguish between the real moonboard inputs and fake ones. Therefore, generator has the purpose of deceiving the discriminator by producing realistic outputs. In our case, activation function for the discriminators selected as Leaky ReLU ($\alpha = 0.2$). In the end, output of the discriminator is produced by the Sigmoid activation function. Discriminator model is trained with real and fake inputs beforehand by the usage of valid and fake labels. Real inputs are fed into discriminator network with a feature vector consisting of ones and fake inputs are fed into discriminator network with a feature vector consisting of zeros. Training both of the generator and discriminator models under sufficient conditions, namely computational resources and data, should produce distinct and opposite loss and accuracy values for the generator and discriminator. Loss of generator model needs to decrease every epoch, whereas loss of discriminator model needs increase every epoch. This would indicate that generated results from the generator model are able to deceive discriminator model.

1) Conditional GAN (CGAN): For CGAN, generator and discriminator network architectures are mainly built upon upscaling fully connected layers and 1D batch normalization layers. Figure 2 depicts mainly the architecture of CGAN used within the project.

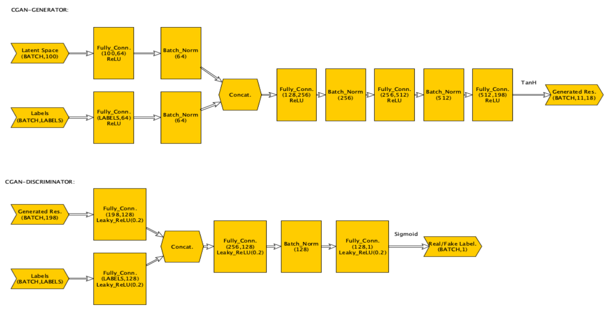
CGAN-GENERATOR:
Latent Space (BATCH,100)
Fully_Conn. (100,64) ReLU
Batch_Norm (64)
Labels (BATCH,LABELS)
Fully_Conn. (LABELS,64) ReLU
Batch_Norm (64)
Concat.
Fully_Conn. (128,256) ReLU
Batch_Norm (256)
Fully_Conn. (256,512) ReLU
Batch_Norm (512)
Fully_Conn. (512,198) ReLU
TanH
Generated Res. (BATCH,11,18)
CGAN-DISCRIMINATOR:
Generated Res. (BATCH,198)
Fully_Conn. (198,128) Leaky_ReLU(0.2)
Labels (BATCH,LABELS)
Fully_Conn. (LABELS,128) Leaky_ReLU(0.2)
Concat.
Fully_Conn. (256,128) Leaky_ReLU(0.2)
Batch_Norm (128)
Fully_Conn. (128,1) Leaky_ReLU(0.2)
Sigmoid
Real/Fake Label. (BATCH,1)

Fig. 2. CGAN Model Architecture

2) Deep Convolutional GAN (DCGAN): For DCGAN, generator and discriminator network architectures are mainly built upon upscaling 2D convolutional and 2D batch normalization layers. Figure 3 depicts mainly the architecture of DCGAN used within the project.

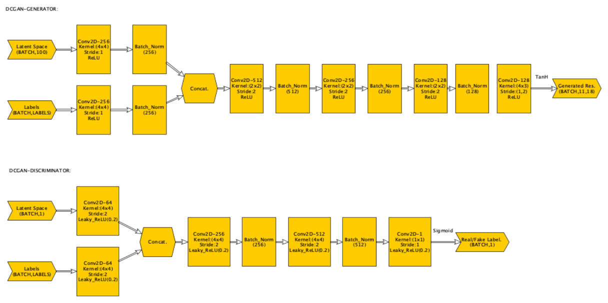
DCGAN-GENERATOR:
Latent Space (BATCH,100)
Conv2D-256 Kernel:(4x4) Stride:1 ReLU
Batch_Norm (256)
Labels (BATCH,LABELS)
Conv2D-256 Kernel:(4x4) Stride:1 ReLU
Batch_Norm (256)
Concat.
Conv2D-512 Kernel:(2x2) Stride:2 ReLU
Batch_Norm (512)
Conv2D-256 Kernel:(2x2) Stride:2 ReLU
Batch_Norm (256)
Conv2D-128 Kernel:(2x2) Stride:2 ReLU
Batch_Norm (128)
Conv2D-128 Kernel:(4x3) Stride:(1,2) ReLU
TanH
Generated Res. (BATCH,11,18)
DCGAN-DISCRIMINATOR:
Latent Space (BATCH,1)
Conv2D-64 Kernel:(4x4) Stride:2 Leaky_ReLU(0.2)
Labels (BATCH,LABELS)
Conv2D-64 Kernel:(4x4) Stride:2 Leaky_ReLU(0.2)
Concat.
Conv2D-256 Kernel:(4x4) Stride:2 Leaky_ReLU(0.2)
Batch_Norm (256)
Conv2D-512 Kernel:(4x4) Stride:2 Leaky_ReLU(0.2)
Batch_Norm (512)
Conv2D-1 Kernel:(1x1) Stride:1 Leaky_ReLU(0.2)
Sigmoid
Real/Fake Label. (BATCH,1)

Fig. 3. DCGAN Model Architecture

3) Tuning of CGAN & DCGAN: As mentioned before, a successful GAN model should produce a decreasing generator loss and an increasing discriminator loss for each proceeding epoch. Unfortunately, there are cases where loss values progress in vice versa directions. This issue is called a *Collapsing GAN* problem, in which discriminator is trained much better in comparison to generator. CGAN and DCGAN faced similar issues, therefore various tunings on both models have been performed.
Firstly, uniform noise values are added to valid feature vector consisting of ones and fake feature vector consisting of zeros. In other words, valid feature vector is transformed into a vector of features having values $\in [0.9, 1]$ and fake feature vector is transformed into a feature vector with values of $\in [0, 0.1]$
Secondly, generator models are trained more than discriminator models, such that both models are on a similar training level. Difference in training has to be tuned, otherwise generator may overfit and discriminator may perform poorly. Therefore, in our case discriminator model is trained only once at every 40 steps, meanwhile generator model is trained at every step of an epoch.

IV. Results and Discussion

A. *Classification and Regression Results*

Classification and regression results for each learning algorithm and *MODE* combination are depicted in Tables III,IV,V. For *MODE 0* & *MODE 1*, scikit MLP-classifier end up with highest accuracy score. On the other hand, PYT model has the worst performance. However, for *MODE 2* PYT model has the highest accuracy score, and scikit MLP classifier results in the worst accuracy score. Other classifier models are always performing accuracy scores inbetween MLP and PYT classifiers. In fact, regression results have the same trends as the classifier results as well. Regression scores are converted to accuracy scores for the sake of the comparison, and resulting accuracy scores are very similar.
For the sake of comparing different *MODE*'s, obviously *MODE 2* and *MODE 0* have a huge difference in terms of accuracy scores. In other words, working on aggregated classes

solves the issue of unbalanced dataset on a high level. *MODE 1* demonstrates similar behaviour in terms of accuracy scores to *MODE 2* and therefore, *MODE 1* could still be labeled with the problem of unbalanced dataset.

TABLE III
CLASSIFICATION AND REGRESSION RESULTS FOR *MODE* 0

3	Classification		3	Regression	
Class	Acc.	Train Time	Class	Acc.	Train Time
MLP	0.8732	378.1811s	MLP	0.8625	195.3364s
KNN	0.8494	102.6362s	LR	0.8464	0.1451s
DT	0.8315	0.8982s	AB	0.7982	41.0183s
SVC	0.8678	50.4439s	RF	0.8607	104.4642s
PYT	0.8109	90.242s			

TABLE IV
CLASSIFICATION AND REGRESSION RESULTS FOR *MODE* 1

8	Classification		8	Regression	
Class	Acc.	Train Time	Class	Acc.	Train Time
MLP	0.5273	302.7172s	MLP	0.4351	160.9884s
KNN	0.4976	103.3942s	LR	0.4107	0.1203s
DT	0.4761	0.9434s	AB	0.3386	37.8598s
SVC	0.4958	134.9227s	RF	0.5023	100.8990s
PYT	0.4763	90.341s			

TABLE V
CLASSIFICATION AND REGRESSION RESULTS FOR *MODE* 2

16	Classification		16	Regression	
Class	Acc.	Train Time	Class	Acc.	Train Time
MLP	0.3720	163.1302s	MLP	0.2583	178.0572s
KNN	0.4048	108.5340s	LR	0.2136	1.5557s
DT	0.3892	1.6635s	AB	0.1767	45.4603s
SVC	0.3869	214.0254s	RF	0.3208	165.5315s
PYT	0.4291	90.8781s			

B. CGAN & DCGAN Results

Table VI indicates a quantitative metric for generation results. Generated routes are conditioned by specific labels during their training, therefore comparison of the actual label and classified label should demonstrate a quantitative outcome for the success level of the generation process. As expected, precision of labeling reduces as *MODE* increases for both CGAN and DCGAN and strikingly, DCGAN acquires highly accurate metric scores in comparison to CGAN.

TABLE VI
ACTUAL LABELING VS. DIFFICULTY LABELING OF GENERATED ROUTES

Quantitive Metric.	*MODE 0*	*MODE 1*	*MODE 2*
CGAN	0.3388	0.0909	0.0909
DCGAN	0.8281	0.4531	0.0937

Figure 4 depicts loss curves of CGAN models for each *MODE* distinctly. Additionally, generated moonboard route settings for each *MODE* are demonstrated in Figure 5. These generated routes are classified by the most successful classifiers as well. As expected, loss of generator is decreasing and loss of discriminator is increasing for each *MODE*. Strict changes in loss are occurring until around 5th epoch.

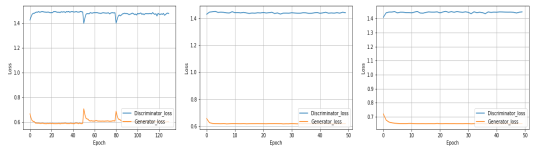

Fig. 4. Discriminator and Generator Losses of CGAN

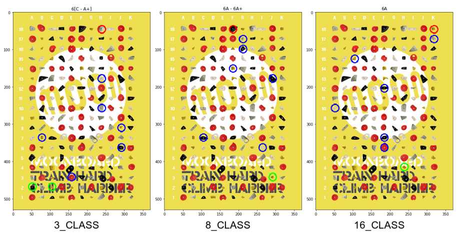

Fig. 5. Sample MoonBoards Generated by CGAN

Figure 6 depicts loss curves of DCGAN models for each *MODE* distinctly. Additionally, generated moonboard route settings for each *MODE* are demonstrated in Figure 7. These generated routes are classified by the most successful classifiers as well. Considering the losses, for *MODE 1* & *MODE 2* losses for both generator and discriminator model are as expected. However, for *MODE 0* discriminator loss is decreasing as well. Although collapsing GAN problem is solved overall, there are still some exiting issues.

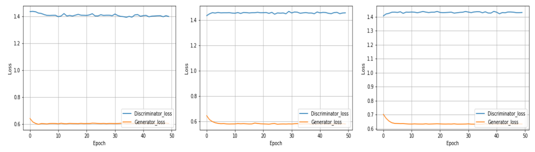

Fig. 6. Discriminator and Generator Losses of DCGAN

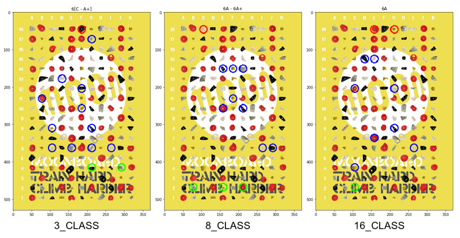

Fig. 7. Sample MoonBoards Generated by DCGAN

V. FUTURE WORK

In this project, object was to generate unique climbing routes from the moonboard climbing route dataset by using Generative Adversarial Networks. A certain generation accuracy was achieved, however due to imbalance of dataset itself, classification of these generated routes were not truly accurate. Given a larger and balanced dataset, more precise routes could be generated with more accurate class labels assignments.

References

[1] "Welcome to training on the moonboard, climb on the same problems as other climbers from around the world." [Online]. Available: https://www.moonboard.com/

[2] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014, cite arxiv:1411.1784. [Online]. Available: http://arxiv.org/abs/1411.1784

[3] G. van Rossum, "Python tutorial," Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Tech. Rep. python, May 1995.

[4] T. Oliphant, "NumPy: A guide to NumPy," USA: Trelgol Publishing, 2006–, [Online; accessed ¡today¿]. [Online]. Available: http://www.numpy.org/

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.