Generating Hearthstone Cards with GANs - Documentation

Background:

In recent years, Generative Adverserial Nets have achieved outstanding results in generating data from noise. As GANs were mostly applied on images, we decided to experiment on one-dimensional data, which is the Heartstone game card dataset. In the following we want to describe basic idea, the preprocessing of the data, the network architecture and our experiments.

Hearthstone

Hearthstone is a Collectible Card Game (CCG) developed by Blizzard Entertainment. The knowledge of the rules of this game is not mandatory for understanding this paper. However, a basic understanding of the structure of a card and the basic design rules of cards are important to understand the challenge of data preprocessing and data generation.
The key feature of CCGs is the uniqueness of the cards.
First, all cards share a cost value, which is an positive integer, describing the cost to play this specific card. This value is strongly related the all other attributes of the card, since it resembles the card value in the game. Depending on the exact type of card (Minion, Hero, Spell, Secret or Weapon) a card can also have other positive integer attributes like health, durability and attack.
Additionally cards can have arbitrary many play conditions (each associated with a positive integer) and mechanics. Both are represented in the game mechanics by function pointers in the game code. Play Conditions are executed when the card is played and usually resemble more unique and complex features of the card. For example the Spell “mortal strike” deals different amounts of damage when the enemy is over or under 12 health. The mechanics resemble abilities of the card with more straightforward behaviour. For example the “forgetful”-mechanic is exclusive to minions and makes it attack the wrong enemy with a probability of 50%. There is no limit on how many mechanics and play conditions a card may have. However, there is currently no card with more than 5 mechanics and play conditions, since the designers likely wanted to limit the complexity of the cards.
Additionally a card is also attached with token-information, that has no direct mechanic attached to it, but sometimes interacts with the mechanics of other cards. For instance belong all cards to a certain “hero class”, “race”, “collection” and have a “rarity” value. Some cards may only interact with cards of a certain hero class, race or rarity.

Data Processing:

All hearthstone cards (except the latest expansion) are available on https://www.kaggle.com/jeradrose/hearthstone-cards as a csv file. The file consists of around 2819 unique cards with 18 different fields, some of which are redundant for our task (such as the card ID or the collection). Many of these fields still contain Strings, lists and dictionaries, so we decided to encode them as one-hot-vectors instead. For example, if there were ten possible names (e.g. Minion, Warrior, Mage…), then we converted it to a 10-dimensional vector. Fields that contain lists or dictionaries are represented as binary vectors with one dimension for a possible entries. For the fields that contained Integers on the other hand, we only normalized the values to values between zero and one. Finally we concatenated all One-Hot-vectors and other fields, which yielded a 134-dimensional vector. For decoding, we varied between taking the maximum value in the corresponding part of the vector or just thresholding (which might yield multiple values).

Network Architecture:

Because GANs were only rarely used for 1-dimensional data, we built our network from scratch instead of following a certain architecture. As input for the generator we used a ten-dimensional vector sampled from a standardized normal distribution. This vector is then fed forward through three dense layers of sizes 11, 268 and 134 respectively. We chose on that architecture in order to have a representational hidden layer of higher size and to create a vector of the same dimension as the real data in the end. Between the layers (not after the last one) we applied batch-normalization and dropout with a rate of 0.4 as this is highly recommended in training GAN to avoid local optima. For activation we used leaky Relu for the hidden layers and a sigmoid activation for the output, as we require values between 0 and 1 as output.
Regarding the discriminator network that receives the 134-dim-vector as input, we chose a similar structure of three fully connected layer of 256, 128 and 16 nodes, the first and the second followed by batch normalization and dropout again. These layer were activated by leaky Relu again. We added one layer with one node and sigmoid activation to get the scalar output indicating whether it is a real or a generated training example. For a graphical visualization of the structure see appendix 1.

Training

During training, we used the following procedure: Firstly, we sampled a batch of size 32 from the real data and a z-vector from the normal distribution. We fed z into the generator and the output of this (G_sample=generator(x)) again into the discriminator. We also calculated the discriminator output for the real data X, reusing the variables of the discriminator. Further with both outputs of the discriminator, we calculated the D_loss_real and the D_loss_fake by using the mean of the cross-entropy error between the discriminator(X) and soft labels (which will be described later on) around 1, and for D_loss_fake the cross entropy between D(G(z)) and soft labels around 0. Both losses were simply added and minimized using a Gradient Descent Optimizer with learning rate **0.001.** It is important to mention that the optimizer gets only the variables of the discriminator as variable scope, because otherwise it would also adjust the weights of the generator and decrease its performance by doing so.
Next, we needed to train the generator. We sampled from Z again and calculated D(G(Z)) in the forward step. This output of the discriminator was then compared to soft labels around 1 with a cross entropy loss function, because the generator aims on maximizing the output of the discriminator. Finally the loss is minimized by a Adam Optimizer with learning rate **0.01,** this time only regarding the variables of the generator. This procedure is the standard for GANs that we also often found in the literature. Of course there is room for adjustments which we want to explain in the next paragraph.

Improvements:

In the first paper about GANs of Goodfellow et al. it was already proposed in the algorithm to vary the rate of updating the discriminator and the generator. We found that often the error of the discriminator decreases quickly, while the generator loss increases. This can be prevented by training the discriminator only every k steps, or only if the generator loss reaches a certain level. We achieved the best results by training the discriminator only every **0.001** steps.
To improve our architecture we looked at the advices from Soumith Chintala's GitHub repository "Ganhacks" (https://github.com/soumith/ganhacks) and a presentation of Goodfellow[1]. As advised, we used batch normalization and dropout, applied leaky relu for the hidden layers, sampled z from a Gaussian instead of a uniform distribution and used Gradient Descent Optimizer for the discriminator. We also wanted to use tanh as activation for the generator output, but as we need values between 0 and 1, this worked out better with a sigmoid function.
Furthermore, as recommended by Soumith et al., we used soft labels. This means, instead of labeling all real data with a 1 and fake data with 0, we sampled from a uniform distribution between 0.8 and 1.0, or 0 and 0.2 respectively. This improved our results as our model generated different vectors

afterwards instead of overfitting on one example. However, all these advices did not help the model to result in realistic Hearthstone cards.

Evaluation:

Unfortunately we were not able to generate useful data in the sense of getting realistic values for one card, fitting to each other. To find out why our model collapses, we used the visualization tool Tensorboard. We found that the loss functions decrease dependently on each other as expected, and that both the D_loss_real and D_loss_fake decrease in a similar matter, only that D is not as good in classifying the generated data as in classifying the real data (which is also to be expected). However, looking at the histograms of our variables, we found that many values are very close to zero (see appendix 2). Apparently our One-Hot-encoding leads to very small weights which are not able to generate different data any more.

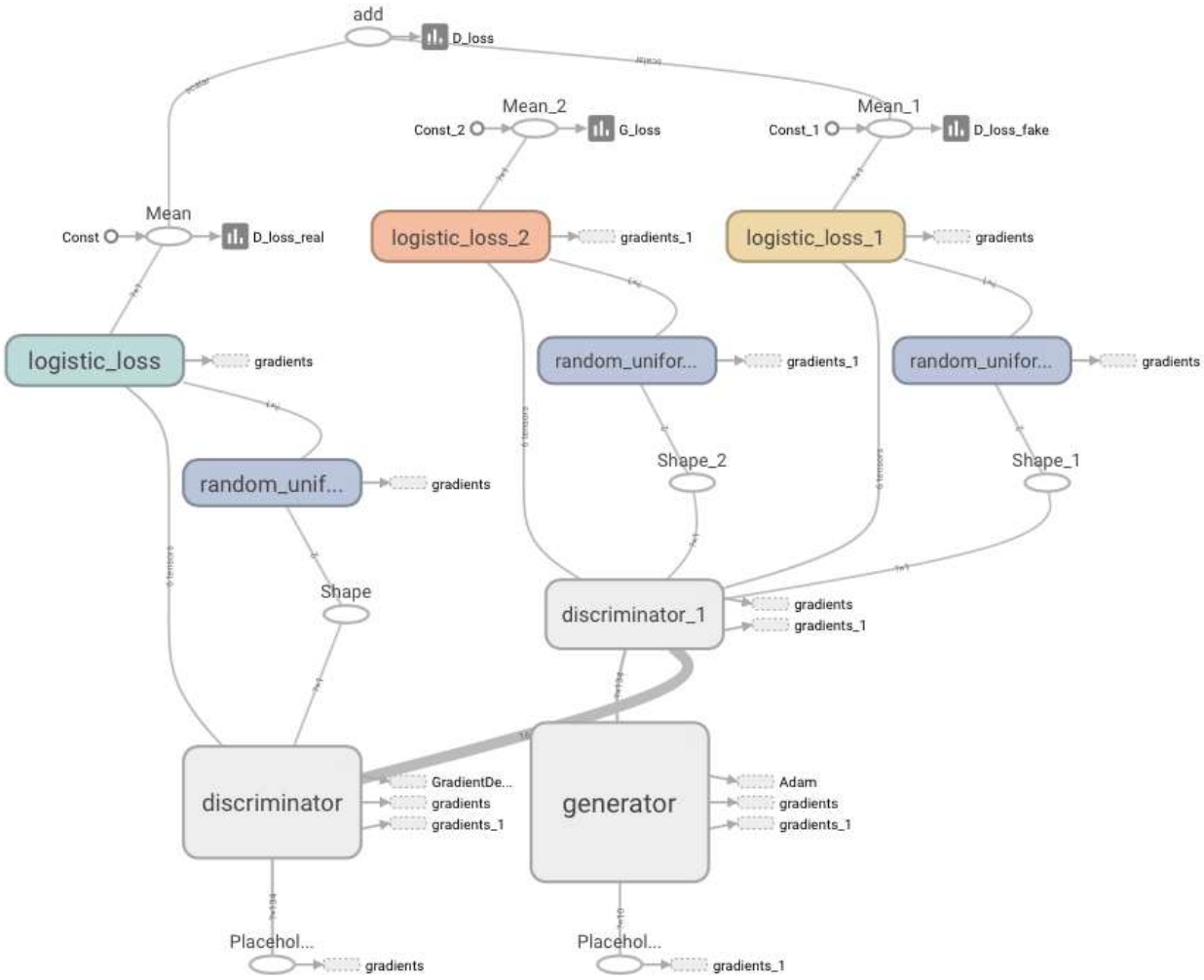
We assume that the biggest problem for our model actually lies in the data. Firstly, one problem could be the complexity of the data itself. Successful demonstrations of card generation (for Magic the Gathering and Hearthstone alike) were usually conducted with LSTMs and char-by-char generation of strings. The advantage of this approach is that it focusses only on the generation of text, not on the rules. Since the cards have only short and very samish text-structures it is fairly easy for a LSTM to extract clear pattern from the text structure of the cards. Therefore, the mechanical perspective we choose for this project might be too complex for the amount of data and the network topologies we experimented with.
Furthermore there are lots of missing values in the dataset. Except for the cost-value, all values of the cards have some exclusive relationship to other values. For instance, only minion and hero-cards feature some specific mechanics and play conditions, also they are the only cards to feature attack and health values. Secrets are exclusive to mages, paladins and hunters. Weapons are the only cards to feature a durability value and all cards with the “legendary” rarity-value have the strong tendency to have very odd compositions of mechanics. Even excluding some card type, for example only focussing on minion-cards, didn’t improve the predictive qualities, since also they have design-specific sub-categories. For instance feature only minion cards for the class “shaman” and with a mana cost less than 5 the play condition “overload”. This is bypassed by a LSTM-system since the card text is usually short and does not resemble the inner complexity of the card on a mechanical level. Our model though needs to represent this relation somehow and apparently our approach with One-Hot-vectors was not able to resemble the structure of the card. Our model is obviously trying to fake the data by filling in many zeros into the vector, which in return leads to completely unrealistic cards in our view, as a card would never consist of missing values exclusively.

All in all, after working on our model structure for a long time, we are quite confident it would work for a one-dimensional dataset, but assume that the data was not appropriate for the quite unstable Generative Adversarial Nets or would have to be preprocessed in another way.

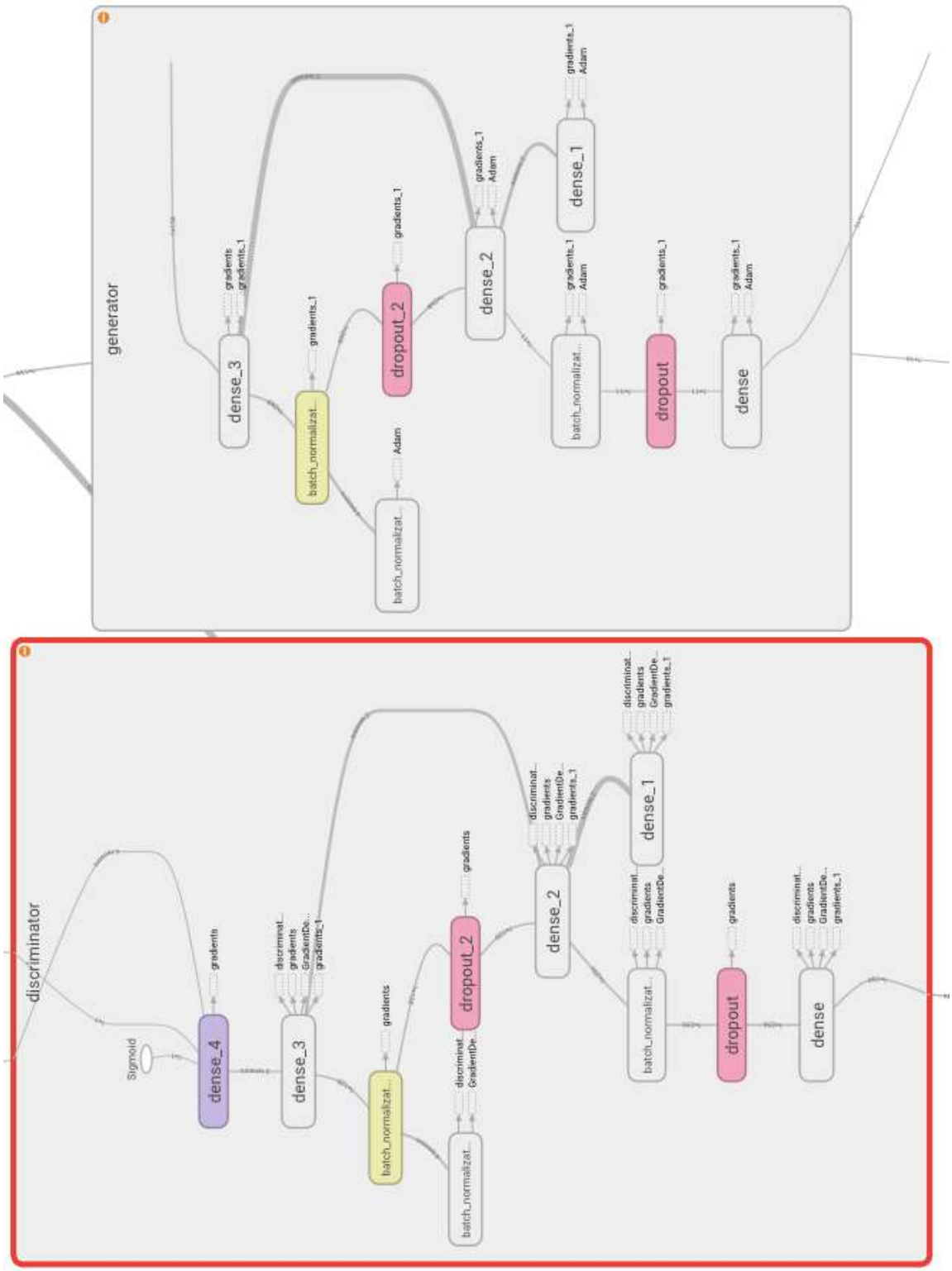
Sources:

[1] Ian Goodfellow, “NIPS 2016 Tutorial: Generative Adversarial Networks”, 2017 (from: https://media.nips.cc/Conferences/2016/Slides/6202-Slides.pdf)

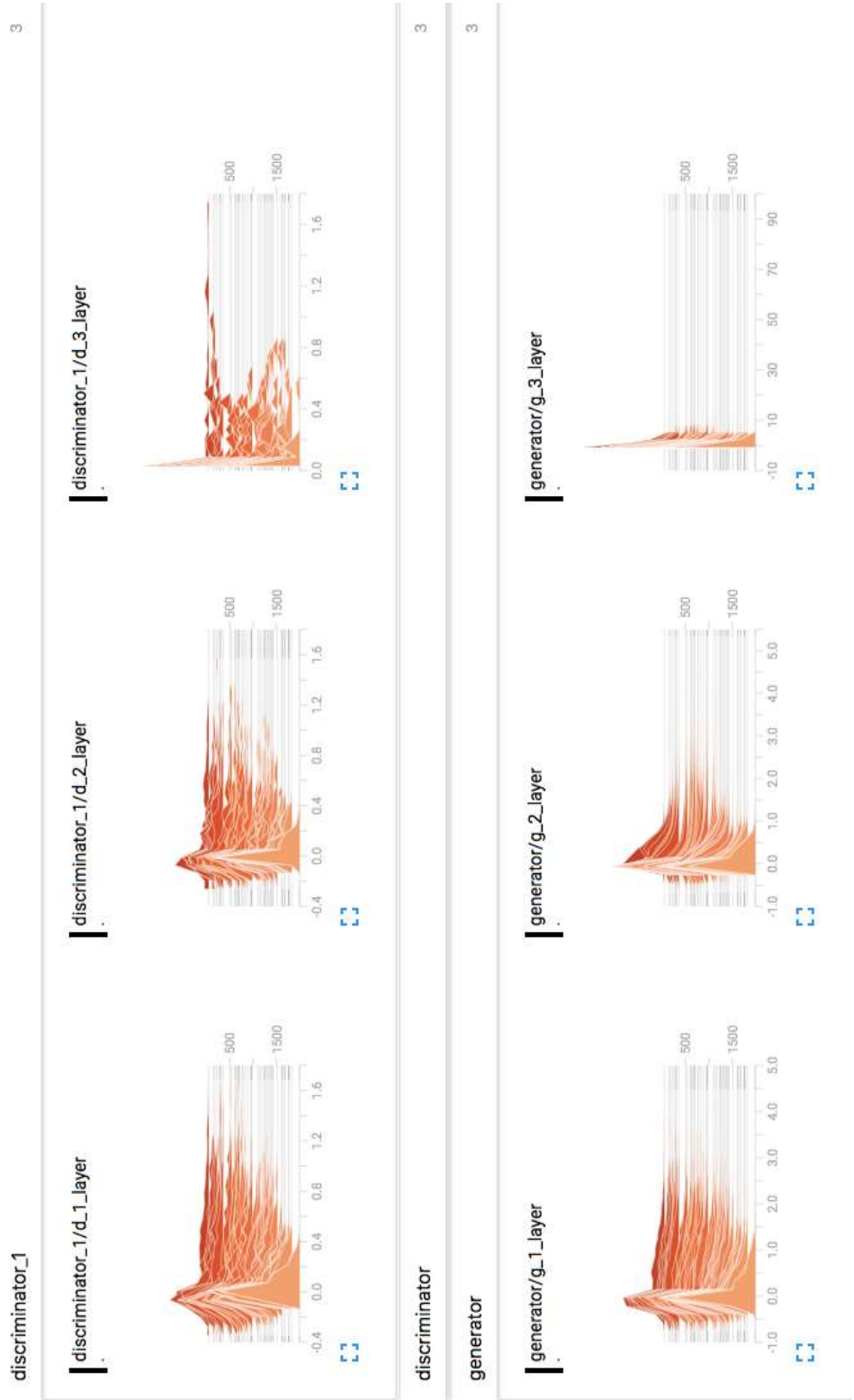
Appendix 1: Tensorflow Graph - a) General structure



Appendix 1: Tensorflow Graph - b) Detailed architecture of G and D



Appendix 2: Weights of the discriminator and generator networks:



Appendix 2: Usage of the implementation

File Structure: The top level of the zip folder stores “restore.py”, this is the final version, alongside auxiliary files, like two csv-files to store the results and the weight vectors.
The code folder contains older or rejected versions of the GAN to illustrate the learning process, we went through.
The SeqGan subfolder, contains the LSTM-GAN hybrid. It is fully functional but is not giving good results for this task.
The “Old GAN versions (Hearthstone)”-subfolder contains legacy versions of the final GAN. The “hearthstone_deeper_softlabel.py” is functionally equivalent to the final version without the save feature.

Step 1: Unpack the zip-file from StudIP

Step 2: the main script is called “restore.py”, you can either train the model new or load the existing weights

Step 2a: If you want to use the stored weights, just execute the script, the script will then output a plot (ignore it in this case). 10 sample results are stored in “results_clean.csv” in the same format as the original (untokenized) cards.
The raw-data-vectors produced by the generator of the same vectors are stored in “results.csv” in the exact same order

Step 2b: If you want to train the model from ground up (takes 2-3 minutes on a laptop gpu), you have to replace the value in line 21 in “restore.py” with None. In this case, the model will start training 2000 batches of cards with 128 real and 128 fake cards per batch.
After the end of the training process, the system will print the losses of generator and discriminator. It will also plot the evolution of the discriminator and generator loss over over the batches of training process.
As described in 2a, after the training process samples are generated and stored in “results_clean.csv” and “results.csv” (for the raw vector).
The weight vector is saved at multiple checkpoints, these are stored in multiple files in the executing directory of “restore.py”.