**Stress testing with GANs | Yields.io | Ilias Aarab**

**PROGRESS**

**Phase 1 – GAN with MNIST**

* Use [eriklindernoren/Keras-GAN](https://github.com/eriklindernoren/Keras-GAN) Github repo to train a vanilla GAN on MNIST data
  + Copy main class to Jupyter notebook
  + Create GAN object and train with MNIST data -> OK
* Use [eriklindernoren/Keras-GAN](https://github.com/eriklindernoren/Keras-GAN) Github repo to train a vanilla GAN on dummy dataset sampled from a Gaussian distribution
  + Tedious to change the code to accommodate a non-image format dataset
  + Write a new notebook with a more modular [GAN](https://arxiv.org/abs/1406.2661) setup to more easily accommodate different datasets. -> OK

**Phase 2 – GAN with continuous and nominal data**

* Use GAN notebook to generate dummy datasets
  + Univariate Gaussian distribution -> OK
  + Multivariate Gaussian distribution -> results are a bit erratic, with sometimes suboptimal results
  + Nominal distributions -> no convergence
* Modify GAN to accommodate nominal distributions
  + Use a multistep process to convert nominal data into probability sets
  + Train GAN to generate probabilities -> what about multivariate case?
  + Add [parallel output dense layers](https://arxiv.org/pdf/1807.01202.pdf) to Generator architecture
  + Generate multivariate nominal datasets -> OK
* Modify GAN to handle mixed datasets (continuous + nominal)
  + Combine architectures of continuous GAN setup with nominal GAN setup
  + Train on mixed datasets -> OK, but semi erratic convergence behavior of the model

**Phase 3 – GAN with credit risk dataset**

* Use GAN notebook to train GAN on a credit risk dataset (7000 instances, 24 features, contains both continuous and nominal data)
* GAN fails to convergence -> [vanishing gradient problem](https://arxiv.org/pdf/1910.00927.pdf)
* Try to calibrate hyperparameters to solve problem -> semi-improvements
* Try to change architectures of both Generator and Discriminator -> semi-improvements (especially from utilizing *Tanh* activation functions and *Batch normalization* as recommended in literature)
* Results still subpar -> Change GAN setup

**Phase 4 – Wasserstein GAN with Gradient Penalty**

* Change GAN setup to accommodate vanishing gradient problems -> [Wasserstein GAN with Gradient Penalty](https://arxiv.org/abs/1704.00028) most robust version in literature together with [DRAGAN](https://arxiv.org/abs/1705.07215)
* Implement WGAN-GP as DRAGAN is more optimal to deal with mode collapse issues than vanishing gradients
* Coding issues with implementing the gradient penalty from Eq. 3 of the paper -> Use TensorFlow to compute gradient penalty
* Follow [official Keras example](https://keras.io/examples/generative/wgan_gp/) to modify the *train*\_*step* of the keras.Model Class to include new gradient penalty
* Test out WGAN-GP on univariate Gaussian dataset -> OK
* Test out WGAN-GP on Multivariate Gaussian dataset -> OK + stable results
* Test out WGAN-GP on Multivariate nominal dataset -> Issues with architecture
* Change WGAN-GP setup to easily accommodate different network architectures -> OK
* Test out modified WGAN-GP on Multivariate nominal dataset -> OK

**Phase 4 – WGAN-GP with credit risk dataset (WIP)**

* Modify WGAN-GP to accommodate mixed distributions
* Train on credit risk dataset -> suboptimal
* Make changes on architecture -> remove batch normalization of Discriminator (!), [alternative](https://arxiv.org/abs/1704.00028) is available when dealing with correlated instances
* Train on credit risk dataset -> OK, but
  + Fails to generate truncated distributions
  + Fails to generate highly imbalanced nominal features
  + One-hot encoding process create possibly large Generator architectures -> implement Keras embedding layers (?)
* Change architecture to better incorporate nominal data -> OK, including imbalanced data OK
* Calibrate hyperparameters -> WIP

**Phase 5 – Quantitative evaluation metrics (WIP)**

* Create more generic code to test models out on different datasets (WIP)
* Implement Kullback-Leibler divergence to quantitatively assess performance
  + Use empirical cumulative distribution to approximate probabilities
  + Extend measure to multivariate universe (WIP)
* Implement alternative quantitative metrics
  + Odds ratio (WIP)
* Implement more reliable statistics
  + Confidence Intervals
* Evaluate (marginal) effects of hyperparameters and architecture on performance

**TO DO LIST**

* **GAN Functions**: Modify the WGAN-GP compilation/training process to make it easier to use (ILIAS)
* **GAN Functions:** Solve save model issue (ILIAS)
* **GAN Functions**: Add performance tracking functions to WGAN-GP (ILIAS)
* **Evaluation:** Add quantitative evaluation metrics (e.g. KL-Divergence metric) (ILIAS)
* **Evaluation:** Add reliable statistics (e.g. Confidence Intervals)
* **GENERAL:** Solve Imbalanced data generation issue -> SOLVED
* **GENERAL:** Solve Truncated data generation issue
* **GENERAL:** Evaluate/understand marginal effects of all hyperparameters of GAN (e.g. adding different types of noise to one-hot encoded variables)
* **GENERAL:** Evaluate/understand the effect of changing the architecture of the Generator/Discriminator. Which architecture is the right one and why (?)
* **GENERAL:** One-hot encoding creates additional columns in our dataset which might lead to a big dataset which leads to a big Generator architecture. Does this affect speed of training (?) Are there better alternatives (?) Current idea is to look at the embedding Layers option from Keras and see whether an embedding of categorical variables is possible? Alternative is to train an autoencoder and use its latent space.

**Q&A SESSIONS**

* **What is the meaning of MV Gaussian? Multivariate?**

Yes, MV stands for multivariate.

* **Why is Batch Normalization not allowed?**

Batch normalization is not recommended within the Discriminator of the WGAN-GP, as it influences the Gradient Penalty attached to the Discriminator. See *Gulrajani et al., Improved Training of Wasserstein GANs pg.5* for a detailed explanation.

* **When you use the Generator of the latent space what exactly are you referring to?**

The Generator draws from a latent space and makes the necessary changes to modify the latent space into a generated distribution that looks as similar as possible to the true distribution that we’re trying to approximate. In other words, the input to the Generator is a latent space (e.g. a multivariate standard Gaussian distribution) and the output is the generated distribution that we want to optimize. See *I. Goodfellow et al, Generative Adversarial Nets* or [GANs Google](https://developers.google.com/machine-learning/gan/summary) for a detailed explanation.

* **There is a comment related to the WGAN-GP saying that it is subclassed from the Keras.Model, what does that mean?**

Keras provides three main ways to create a Neural Network:

* Sequential API: Straightforward, but doesn’t allow much flexibility.
* Functional API: Still Easy to use, and allows to create more flexible architectures (e.g. parallel dense layers)
* Model Subclassing: More complex, but allows us to create a neural network from scratch in a well-defined way.

The different Discriminators, Generators and basic GAN are all built using the Functional API from Keras. However the WGAN-GP requires more complex changes thus we need to use the Model Subclassing method, which means that the implementation of the WGAN-GP is a bit different compared to the other neural networks. See [Keras Official](https://keras.io/api/models/) for more details.

* **Why is the WGAN-GP a better model and how can you see this?**

The basic GAN suffers from a vanishing gradient of the Discriminator which results in the Discriminator failing to provide useful gradient information (during backpropagation) to the Generator as soon as the Discriminator is well-trained. In contrast, the WGAN-GP Discriminator does not saturate and converges to a linear function with clean gradients.

An easy way to see this is to run both the basic GAN and WGAN-GP on the PPNR dataset and evaluate the training process.

A more detailed discussion can be found in *Arjovsky et al., Wasserstein GAN*.

* **Why are continuous data normalized to [-1,1] ?**

Most papers recommend using a bounded activation function to train a GAN model as it leads to more stable results. As mentioned in Radford et al., *Unsupervised representation learning with DCGAN “*The ReLU activation (Nair & Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution.*”*

Thus we use the *Tanh* activation function when generating continuous data with the Generator. To make the output of the Generator match with the input dataset to the Discriminator we preprocess the continuous data to [-1,1].

However, it might be the case that other bounded activation functions yield better results and thus this should be regarded as a hyperparameter that can be calibrated.

* **What does the “WIP” stand for within the notebooks and documentation?**

*WIP* is an abbreviation for *Work In Progress,* meaning that I am still working on the specific task and that the current version is only a semi-prototype for test purposes

* **Can you explain the performance metric you use?**

It’s notoriously difficult to evaluate the performance of GANs. Most papers (which are mostly concerned with image processing) simply look at generated samples from the trained Generator to evaluate whether the quality of the samples is satisfactory. Although some papers that are mentioned in the notebook provide alternatives.

The current version of the *GAN\_functions* notebook(28/08/2020) depends on evaluating performance by looking at the marginal distributions of the generated dataset and comparing those with the ones of the original dataset.

I am working on integrating a multivariate version of the KL-divergence within the notebook, in order to have a quantitative metric to evaluate the performance of the model. Another addition is to have a multivariate Kolmogorov-Smirnov test. I am also looking at using the loss functions as a way to measure performance.

* **At the end of the main notebook, there is another *wgan.fit()* function running. Are you training again? Could you explain?**

In case the current results are unsatisfactory, we can run the last cell to continue training the model. Afterwards, we can go back to “STEP 4: EVALUATE PERFORMANCE “ and sample a new set of generated data with the newly trained generator and re-evaluate the output.

* **How are the loss functions of WGAN-GP computed?**

We have the following minimax problem with the WGAN-GP model (ignoring the GP for now):

**min\_G max\_D ( 𝔼 x∼p X [ D ( x ) ] - 𝔼 z∼p Z [ D ( G ( z ) ) ] )**

with: p(X) the true distribution, p(Z) the latent distribution, G(z) the Generator, D(x) the Discriminator

Which results in the following loss functions:

**Discriminator**: min (**𝔼 z∼p Z [ D ( G ( z ) ) ] - 𝔼 x∼p X [ D ( x ) ]** )

**Generator**: min (**- 𝔼 z∼p Z [ D ( G ( z ) ) ] )**

We approximate the Expected values by taking the arithmetic means.

Since the Discriminator has an output space of [-inf, +inf] we can expect quite erratic behaviour which would result in problems during the backpropagation phase. To ensure a well-behaved function we constrain the Discriminator to be 1-Lipschitz continuous. We can do this by either weightclipping (WGAN) or adding a gradient penalty (WGAN-GP). Weightclipping is suboptimal so we use the Gradient Penalty.

The Gradient Penalty ensures that the norm of the gradient of the Discriminator (with the input being an interpolation of the real and fake distribution) is at most one, which ensures that the Discriminator is 1-Lipschitz continuous (see Proof of Proposition 1, page 12, Improved WGANs). The Discriminator loss function then becomes:

**Discriminator**: min (**𝔼 z∼p Z [ D ( G ( z ) ) ] - 𝔼 x∼p X [ D ( x ) ]**

**+ c𝔼 w∼pW[ square( norm(gradient(D(w))) - 1) ]** )

with: c a penalty coefficient, p(W) an interpolated distribution.

During backpropagation the weights of the Discriminator will be adjusted to minimize the added Gradient Penalty, ensuring that the weights and Discriminator conform to the 1-Lipschitz requirement in a more natural way than directly clipping the weights.

Within the current notebooks (07/09) the two loss functions are defined in the main notebook (STEP3), whereas the Gradient Penalty is defined within the functions notebook within the WGAN class by utilizing Tensorflow to create interpolated distributions. The loss function of the Discriminator and the Gradient Penalty are then added together during the training (when calling the fit() function).

* **How should I interpret the loss functions of WGAN-GP?**

The loss function of the **discriminator** is an approximation of the Earth-Mover distance (i.e. the minimum transportation cost when transporting mass of probability Q in order to approximate probability P) up to a constant scaling factor dependent on the Discriminator’s architecture and the Gradient Penalty. The constant scaling factor makes it hard to compare the loss functions between different model specifications. Note also, that the loss function is simply an approximation of the true EM-distance, so it’s hard to know how closely the loss function is to the true EM-distance.

However, we can use the loss function as an indication of how well the training relatively improves the performance of our model. A lower loss would indicate an improvement in the generated distribution. In our case the loss function is inverted so an increase in the loss would indicate an improvement in the generated distribution. Examples of plots of the loss function can be found back in the *Wasserstein GAN* and *Improved Training of WGANs* papers.

The **Generator** is dependent on the Discriminator during the backpropagation phase and thus the loss function of the Generator does not provide a meaningful absolute metric. However, we should expect the generator to oscillate between a certain range during convergence or to gradually increase/decrease, without any erratic behaviour.

The **KL-divergence** quantifies how much one probability distribution differs from another probability distribution. The range of the KL-divergence is [0, +inf] with a value of zero indicating that the two distributions are identical. Thus we should expect the KL-divergence to decrease during the training phase and converge towards zero.

Additional questions:

* **In the def generator\_setup\_WGANGP\_GENERIC function, why do you create dense layers for every column in the dataset?, the same question for the new output layers, why are they created for every column of the dataset?.**