Image Creation Using Generative Adversarial networks

Table of Contents

Introduction

> "Adversarial training (also called GAN for Generative Adversarial Networks), and the variations that are now being proposed, is the most interesting idea in the last 10 years in ML, in my opinion.". Yann LeCun, 2016.

Figure 1 gives a preview of what you will learn to do in this lab.

Figure 1a: from left to right: original image (OI); OI + “young” attribute; OI - “blond hair” + “black hair”; OI - “smile”; OI + “male” + “bald”.

Figure 1b: animation showing interpolations across the images in Figure 1a.

What is a generative model?

In machine learning, a generative model learns to generate samples that have a high probability of being real samples like the samples from the training dataset. Let's take an example to understand what this means: consider a world composed of all possible 64x64 8-bit RGB images. There are 2^(3x8x64x64) possible arrangements of the pixel values in those images. This is a staggering number, much greater than the number of particles in the universe. Now consider the set of images that represent faces of people. Even though there are many, they represent a tiny fraction of all the images in the world. Learning a probability distribution of the set of face images means that the model needs to learn to assign a high probability (of being a real sample) to the images that represent a face and a low probability to other images. Finally, consider a dataset of a few thousand face images: learning to generate new samples from this dataset means that the model needs to learn the distribution from which the images in the dataset were drawn. Ultimately the model should be able to assign the right probability to any image—even those that are not in the dataset. A generative face model should be able to generate images from the full set of face images.

How does a GAN work?

Generative Adversarial Networks (GAN) were introduced by Ian Goodfellow in Generative Adversarial Networks (https://arxiv.org/abs/1406.2661), Goodfellow, 2014.

The typical GAN setup comprises two agents:

- a Generator `G` that produces samples, and
- a Discriminator `D` that receives samples from both `G` and the dataset.

`G` and `D` have competing goals (hence the term "adversarial" in Generative Adversarial Networks): `D` must learn to distinguish between its two sources while `G` must learn to make D believe that the samples it generates are from the dataset.

A commonly used analogy for GANs is that of a forger (`G`) that must learn to manufacture counterfeit goods and an expert (`D`) trying to identify them as such. But in this case — and this is a very important GAN feature — forgers have insider information into the police department: they have ways to know when and why their products are marked as fake (though they never get to see any of the real goods). Likewise, the police have access to a "higher authority" that lets them

know whether their guesses are correct. During training G and D keep playing this game, getting better as they play, until they both become so good that the samples produced by G are indistinguishable from the real stuff.

One important aspect to keep in mind here is the fact that since D is trained continuously, G cannot simply go about generating the same samples over and over. In particular, if D has ample capacity, it may be able to memorize all the samples that G ever produced and become expert at spotting fakes. How would G then be able to continuously handcraft novel samples? Going back to our dataset of face images, assuming your are G and already in possession of a few realistic images (or at least deemed as such by D), one reasonable way of coming up with an increasing number of realistic images would be to apply small perturbations to the images that you already have: crops, rotations, color transformations, etc. This way you could create a virtually infinite number of new images, but chances are D would start noticing patterns there and punish you.

In order to alleviate this issue, G learns to generate very different samples: given a set of random numbers, which is called the latent representation, or in short the z vector, and through a number of neural layers, G learns to generate new samples that depend on z sufficiently strongly that a different z will yield a very different sample.

It may be hard to believe at first, but through training, every factor in z, or combination thereof, will specialize towards one characteristic feature of the dataset. For example, in a dataset of face images, factors might represent the gender of the person, the color of their skin, their pose, etc.

Figure 2 summarises the GAN framework.

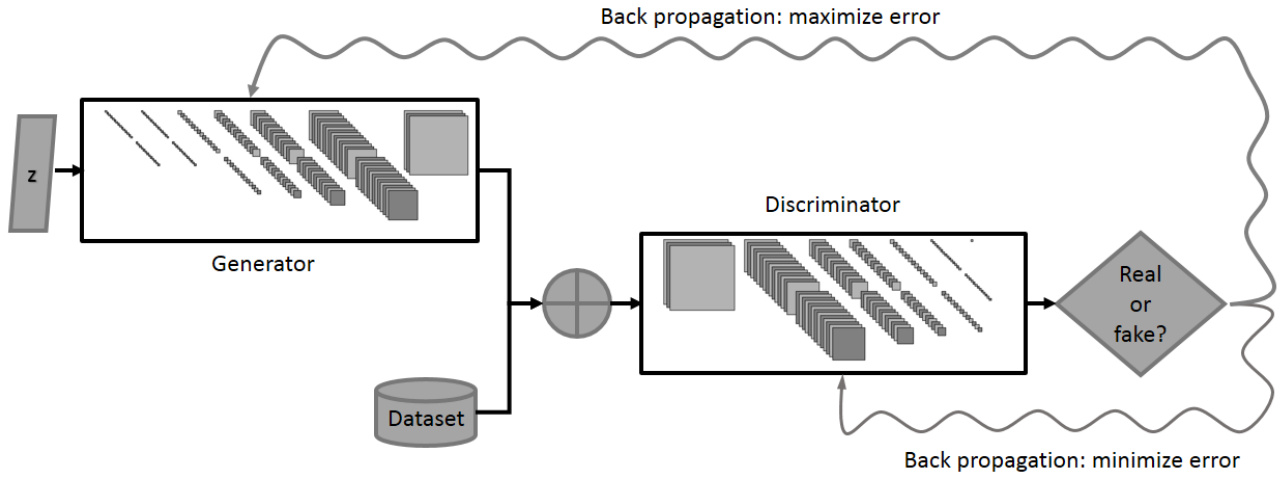


Figure 2: Illustration of the GAN framework: D (Discriminator) is alternately presented with images from G (Generator) and from the dataset. D is asked to distinguish between the two sources. The problem is formulated as a minimax game: D is trying to minimize the number of errors it makes. G is trying to maximize the number of errors D makes on generated samples. The squiggly arrows represent the back propagation of gradients into the target set of parameters.

Why are GANs so popular?

GANs are popular partly because they tackle the important unsolved challenge of unsupervised learning, and partly because the amount of available unlabelled data is considerably larger than the amount of labelled data. I can't resist another quote from our esteemed Deep Learning luminary:

> "If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake. We know how to make the icing and the cherry, but we don't know how to make the cake." Yann LeCun, 2016.

Another reason for their popularity is that GANs are considered to generate the most realistic images among generative models (Ian J. Goodfellow (2016). NIPS 2016 tutorial: Generative Adversarial Networks. arXiv:1701.00160). This is subjective, but it is an opinion shared by most practitioners.

Besides, the learned latent representation in a GAN is often very expressive: arithmetic operations in the latent space, that is to say the space of the z vectors, translate into corresponding operations in feature space. For example, you can see in Figure 1 that if you take the representation of a blond woman in latent space, subtract the "blond hair" vector and add back the "black hair" vector, you end up with a picture of a woman with black hair in feature space. That is truly amazing.

Generating Images of Hand-Written Numbers

In this part of the lab we will train a GAN on the popular MNIST dataset. Figure 3 shows a few sample images from this dataset:

Figure 3: samples from the MNIST dataset.

Here we will use the MNIST dataset. You don't need labels to train a GAN however if you do have labels, as is the case for MNIST, you can use them to train a **conditional** GAN. In this example, we will condition our GAN on the class labels. Conditioning a GAN in this way is useful because this allows us to dissociate classes from other learnable features that define the "style" of images. In practice, in our network we will concatenate a one-hot representation of labels to the activations of every layer in both the generator and the discriminator.

In this lab, the MNIST dataset for DIGITS has already been created for you but if you are curious to know how this was done you may refer to the DIGITS Getting Started Guide (https://github.com/NVIDIA/DIGITS/blob/digits-5.0/docs/GettingStarted.md).

Click here (/digits/) to open DIGITS.

On the DIGITS home page you may select the `Datasets` tab to view all existing datasets. You may then click `MNIST_Classification` to review the details of the MNIST dataset. In particular, you can click the `Explore` button to explore the dataset as can be seen on Figure 4.

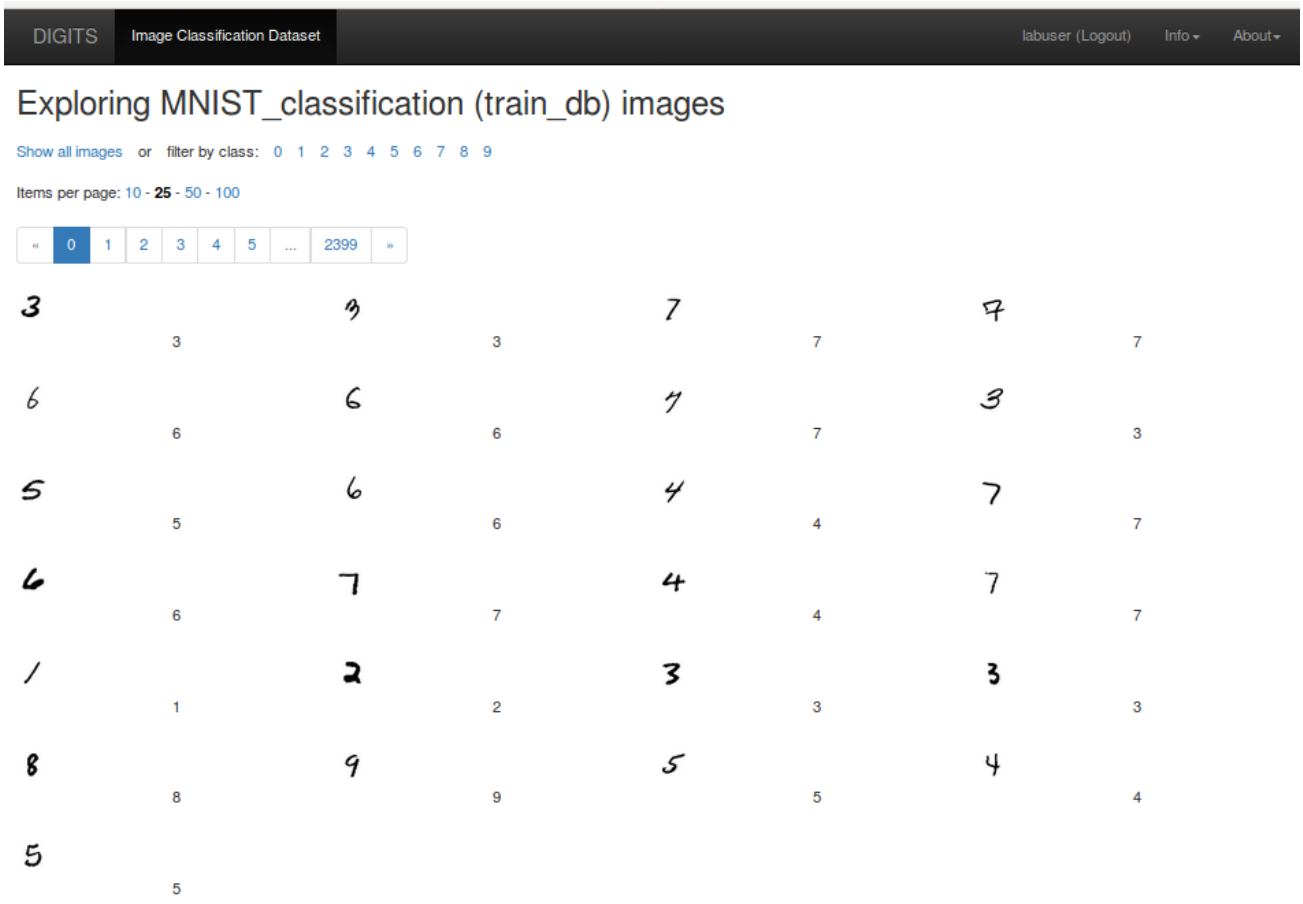


Figure 4: exploring the MNIST dataset

One technicality here is the fact that DIGITS supports several kinds of datasets:

- classification datasets, for classification models,
- generic datasets, for other models. In this lab, we will train a GAN, which falls into the category of "other" models, therefore we need to create a generic dataset. In the lab we have created a generic dataset using exactly the same data (the same LMDB database) as the MNIST classification dataset. This dataset is named `MNIST_Generic` in the lab and this is the one we are going to use.

Network topology

Note that labels aren't required to train a GAN, but if we do have labels (as is the case for MNIST) we can use them to train a conditional GAN. A conditional GAN is one that is conditioned to generate and discriminate samples based on a set of arbitrarily chosen attributes. On MNIST, we can condition the GAN on the class of the number we would like to generate. In practice, we concatenate a one-hot representation of the class to the activations of every layer. For fully-connected layers the one-hot representation is just a vector of length 10 (because we have 10 classes of digits) with zeros everywhere except when the index matches the class ID. This idea can be extended to convolutional layers too: in this case conditioning is materialized by a set of 10 feature maps with zeros everywhere except for the feature map whose index is the class ID, which is filled with ones. During training, the latent representation z is sampled from a 100-dimensional normal distribution; an arbitrary choice that yields acceptable results.

Network conditioning, and the topologies of our `G` and `D` networks - a slightly modified version of DCGAN (Alec Radford, Luke Metz, Soumith Chintala (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv:1511.06434) - are illustrated in Figure 5. Activation functions are not shown in this figure, but importantly the activation function for the last layer in `D` is a typical sigmoid.

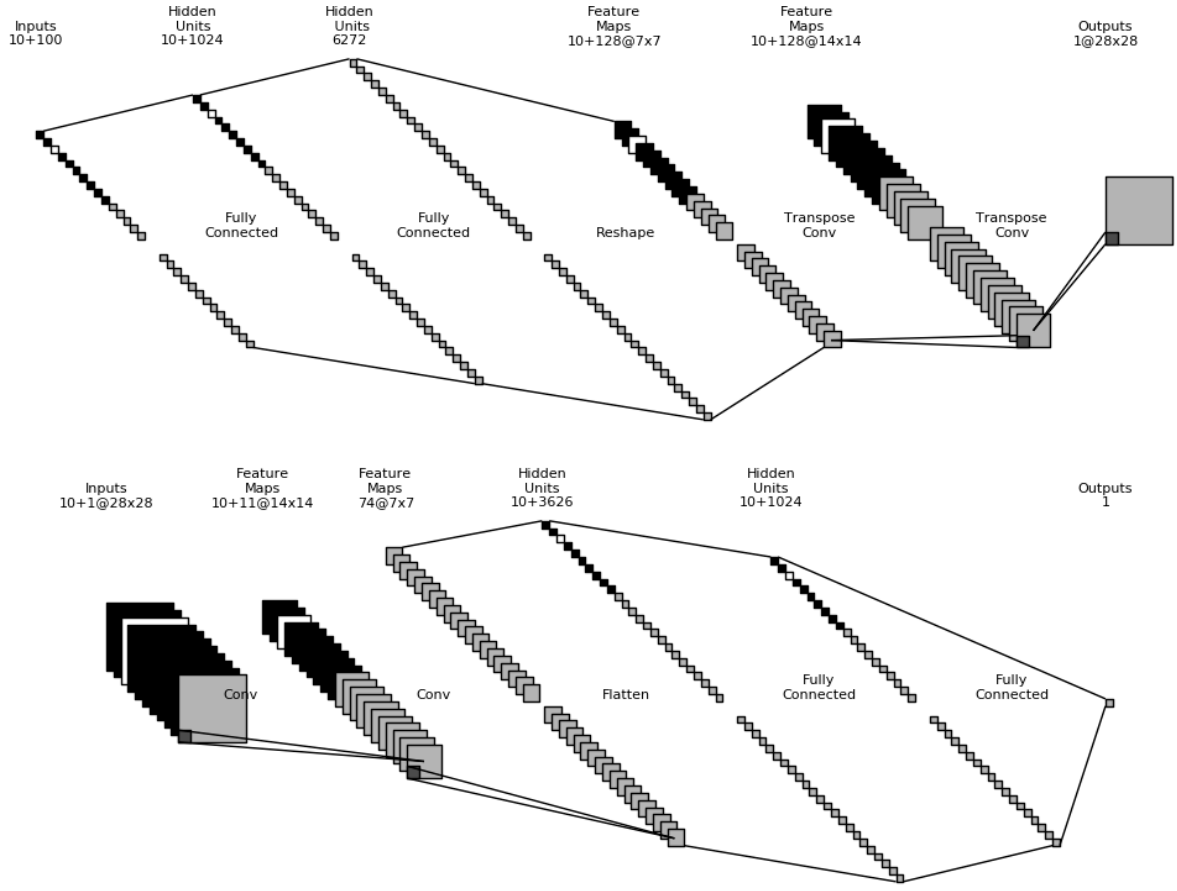


Figure 5: Top: the generator (G) network. Bottom: the discriminator (D) network. Layers are traversed from left to right. Network conditioning is illustrated by the black (zeros) and white (ones) feature maps/units. In the above example the third feature map is white, denoting the fact

that the corresponding label is assigned to the third class (digit "3"). Implementation based on [5]. Grey feature maps/units mark computed activations.

Objective functions

`D` is trained to predict a probability distribution: the probability that a sample xbelongs to each of the known classes. There are only two classes since a sample is either real or fake. In the original formulation of GAN, `D` is trained to maximise the probability of guessing the correct label by minimizing the corresponding cross-entropy loss `L=-p_i * log(q_i)` where `p` is a one-hot encoding of the label, `q` is the predicted probability distribution and `i` is the class index. If `D(x)` represents the probability that `x` is a data sample, then `L=-log(1-D(G(z))` for generated samples. When the GAN framework is formulated as a minimax game, `D` tries to minimize `L` and `G` tries to maximise it. That's rather unfortunate for `G` because during the early phases of training when the output of `G` is less than convincing, `D` easily assigns a very low probability to `G(z)`. This implies that the logit `x` before the last `sigmoid(x)=1/(1+exp(-x))` activation function in `D` is a large negative number and as the blue curve in Figure 6 shows, the loss saturates (it is flat and the gradient is tiny) as highlighted in the original GAN paper.

This makes `G` learn very slowly, if at all. Therefore it's common to reformulate the optimization objective for `G` as a maximisation of `L=log(D(G(z))` — the red curve — instead, which yields the same solution but exhibits much stronger gradients in the negative quadrant. Another common solution is to flip the labels: make `D` predict 1 when optimizing for `G`. One further refinement that we may implement to alleviate the penalty of an overconfident `D` is to use one-sided label smoothing and replace the hard label of 1 when optimizing `D` on real samples with a softer 0.9, as suggested in Ian J. Goodfellow (2016). NIPS 2016 tutorial: Generative Adversarial Networks. arXiv:1701.00160.

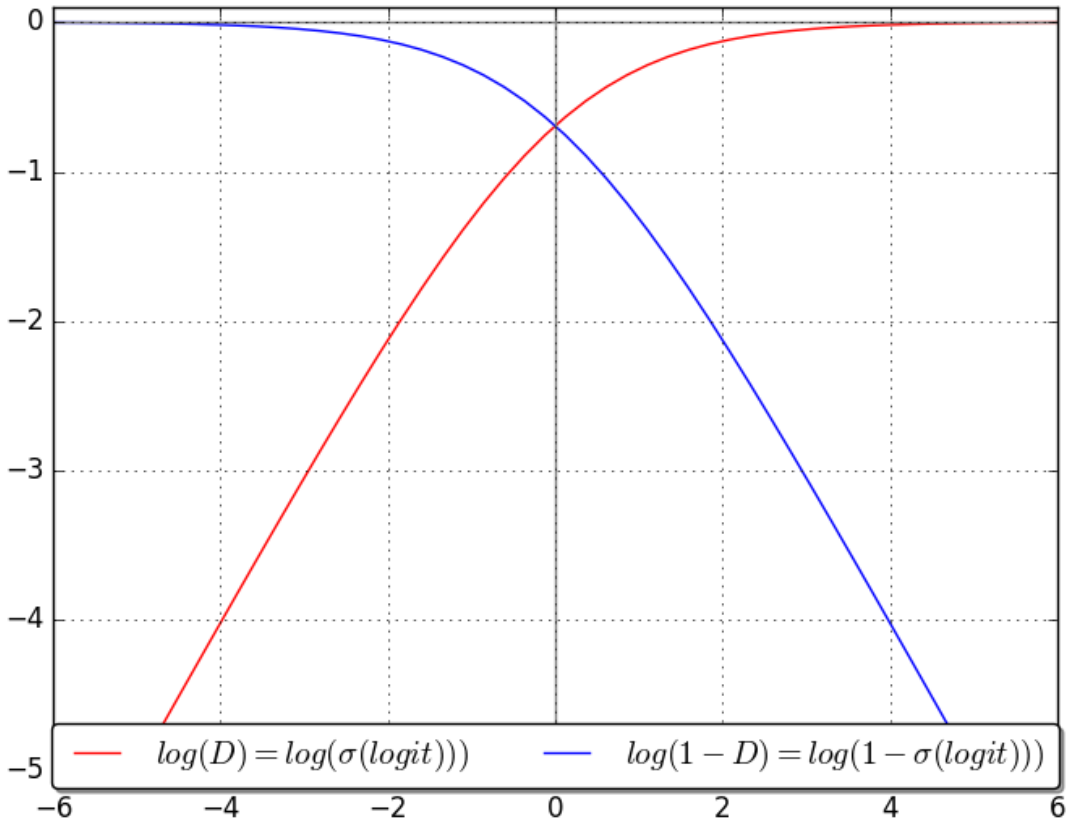


Figure 6: Illustration of vanishing gradient in the negative quadrant when using the original loss formulation log(1-D)(blue curve). The x-axis is D's logit (output of last layer before sigmoid activation). Minimising log(1-D)yields the same solution as maximising log(D) (red curve) but the red curve exhibits stronger gradients.

Implementation of the model in DIGITS

Credits: our implementation derives from DCGAN-tensorflow (https://github.com/carpedm20/DCGAN-tensorflow).

Take some time to review the DIGITS model definition (https://github.com/gheinrich/DIGITS-GAN/blob/DIGITS-GAN-v0.1/examples/gan/network-mnist.py). Read the code and comments carefully.

Some elements of the code should be given particular attention:

The core of the model is defined in the `UserModel` class, with which DIGITS interacts to perform training and inference. In the top-level constructor of the class, DIGITS sets some properties:

- `self.is_training`: whether the instance is created for training

- `self.is_inference`: whether the instance is created for inference
- `self.x`: input node. Shape: `[N, H, W, C]`
- `self.y`: label. Shape: `[N]` for scalar labels, `[N, H, W, C]` otherwise.

This class must define important properties which DIGITS will refer to:

- `inference`: TensorFlow operation to fetch during inference. This defines what the model outputs during inference. In our case, this is the output of `G`.
- `loss`: this returns an array of losses to use for optimization. In our case, this returns three losses:
  - `D` loss on real samples, optimizing over the variables of `D`,
  - `D` loss on fake samples, optimizing over the variables of `D`,
  - `G` loss, optimizing over the variables of `G`.

Make a good note of the fact that when we are optimizing the model to reduce the loss of `G`, we **must** specify which variables to update. By default, TensorFlow will update every trainable variable in the graph. In our case, we only want to modify the variables in `G`. The same applies to the losses of `D`.

Another aspect to keep in mind is the fact that we are creating to identical instances of `D` in this model:

- one instance receives samples from the dataset (`self.x`)
- the other instance receives samples from the generator (`self.G`) In order to create two instances of the same operations that share parameters (weights and biases) we are calling `scope.reuse_variables()` when creating the second instance.

Training the model

On the home page, click `New Model>Images>Other` to access the model creation page. On that page:

- select the `MNIST_Generic` dataset,
- set the number of epochs to 5,
- set mean subtraction to `None`,
- select `ADAM` solver,
- set the learning rate to `2e-4`.

In the `custom network` tab, select `Tensorflow` and copy-paste this network definition (https://raw.githubusercontent.com/gheinrich/DIGITS-GAN/DIGITS-GAN-v0.1/examples/gan/network-mnist.py).

You can click `Visualize` to pre-visualize the model graph. Sorry, the pre-visualization in DIGITS only works in Chrome. If you are not using Chrome, fret not. You will be able to visualize the graph later in Tensorboard when training is taking place.

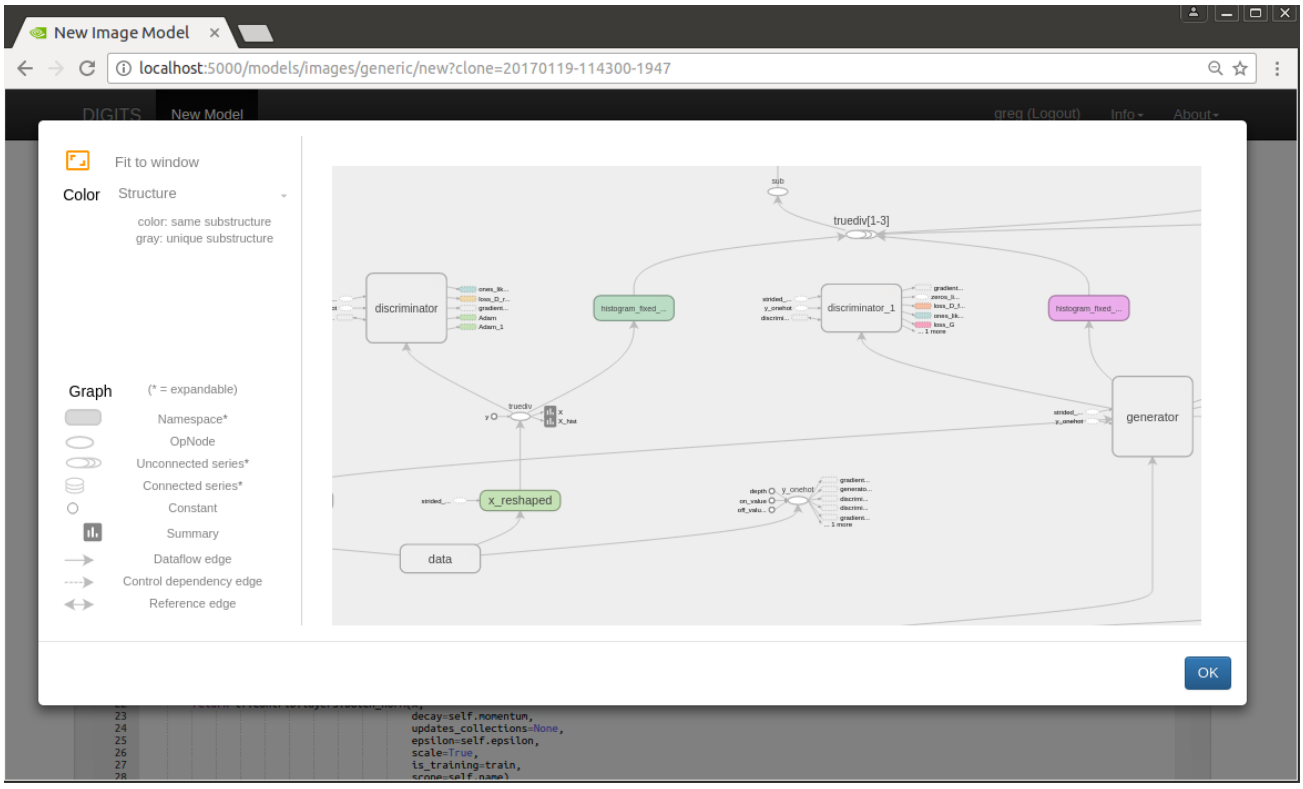


Figure7: visualization of the graph in DIGITS

Finally, to start training the model, name it `GAN-MNIST` and click `Create`.

Loss curves

The first thing that we usually check when training a neural network is whether the loss function is going down. However here there are two components of the total loss: `D`'s loss and `G`'s loss. By construction, the two losses cannot simultaneously go down. Otherwise D and G wouldn't be called adversaries! How do you determine whether training is successful? This turns out to be rather difficult in practice, and the subject of ongoing research.

For one thing, you can check whether the game is well balanced. How does that translate into meaningful loss values? If `G`'s loss is very low this means `D` is unable to recognize fake samples, possibly because it classifies everything as real. Remember that all the knowledge in `G` comes from `D` through back-propagation. `G` is therefore unlikely to generate good samples if `D` is doing a poor job at spotting fakes. At equilibrium you would expect D to not be able to do better than random guessing, meaning that it would assign probability 50% to all samples. Going back to the definition of the cross-entropy loss, this would yield a loss value of `-log(0.5)=69%`. That number should be easy enough to remember.

A well balanced game means that both players are equally good... or bad! How do you figure out how well players are doing? One solution is to stare at the generated samples. Tensorflow makes it easy to visualize generated samples by adding a variable to the collection of Tensorboard

summaries:

```
tf.summary.image("G",self.G))
```

Click here (/tensorboard/) to open Tensorboard.

This causes generated images to be added periodically to the Tensorboard log and we can then monitor them in real time as illustrated on Figure 8.

Note: on the main Tensorboard page you will notice several log directories. Make sure you pick the one that matches the job ID of your training job. Note that Tensorboard data are updated periodically so you might have to wait for a minute between each refresh.

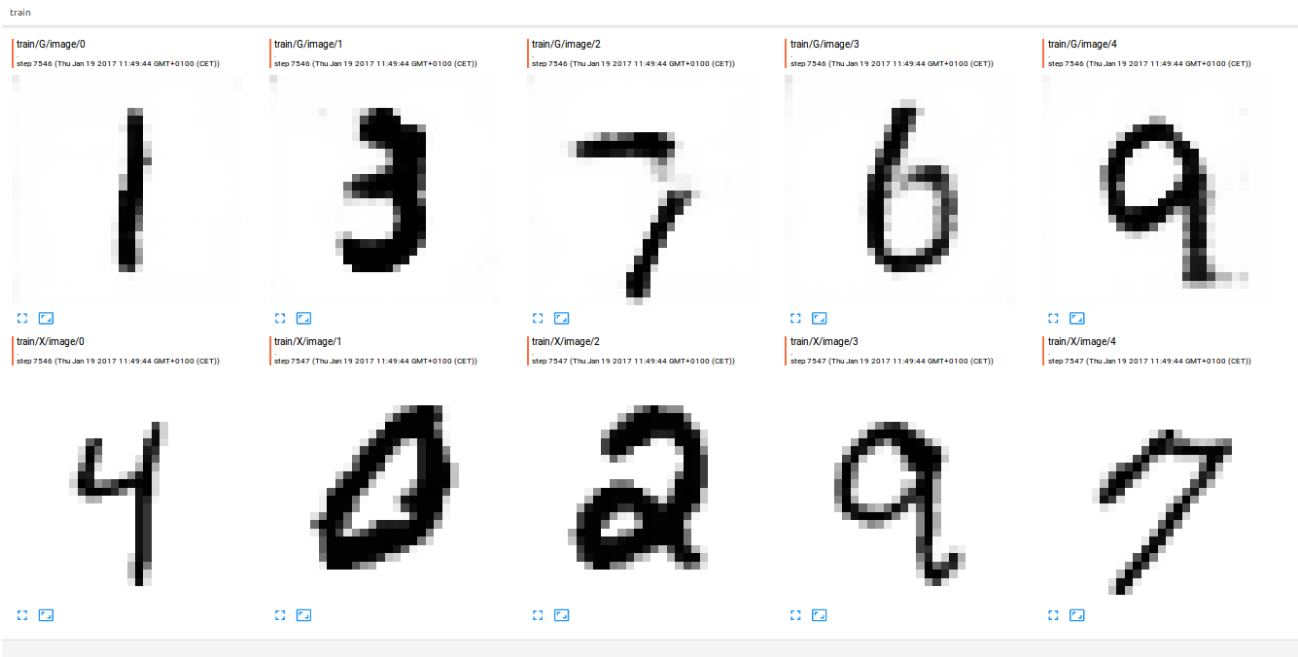


Figure 8: Image samples from an arbitrary iteration of the training loop. Tensorboard updates these images in real time during training. Top: output of G. Bottom: samples from the dataset.

Since it is not practical to keep looking at the generated samples, we implemented the following metric: remember that `G` is learning to approximate the probability distribution of the world from which our dataset samples are drawn. One simple proxy that we can use here is to compare the unconditional (non-spatial) distributions of the real and generated samples respectively. For every batch, we can calculate histograms of pixel values from each source and compare them using the Chi-square distance operator. A low Chi-square value, though not sufficient, is necessary for a good approximation of the data distribution and this metric is easy to implement:

```
value_range = [0.0, 1.0]
nbins = 100
hist_g = tf.histogram_fixed_width(self.G, value_range, nbins=nbin
s, dtype=tf.float32) / nbins
hist_x = tf.histogram_fixed_width(self.images, value_range, nbins
=nbins, dtype=tf.float32) / nbins
chi_square = tf.reduce_mean(tf.div(tf.square(hist_g - hist_x), hi
st_g + hist_x))
tf.summary.scalar("chi_square", chi_square))
```

Figure 9 shows the loss curve in DIGITS. As you can see, all losses are relatively close to 69%, an indication of a well balanced game. Also, the Chi_square curve (purple) goes down steadily, another reassuring sign.

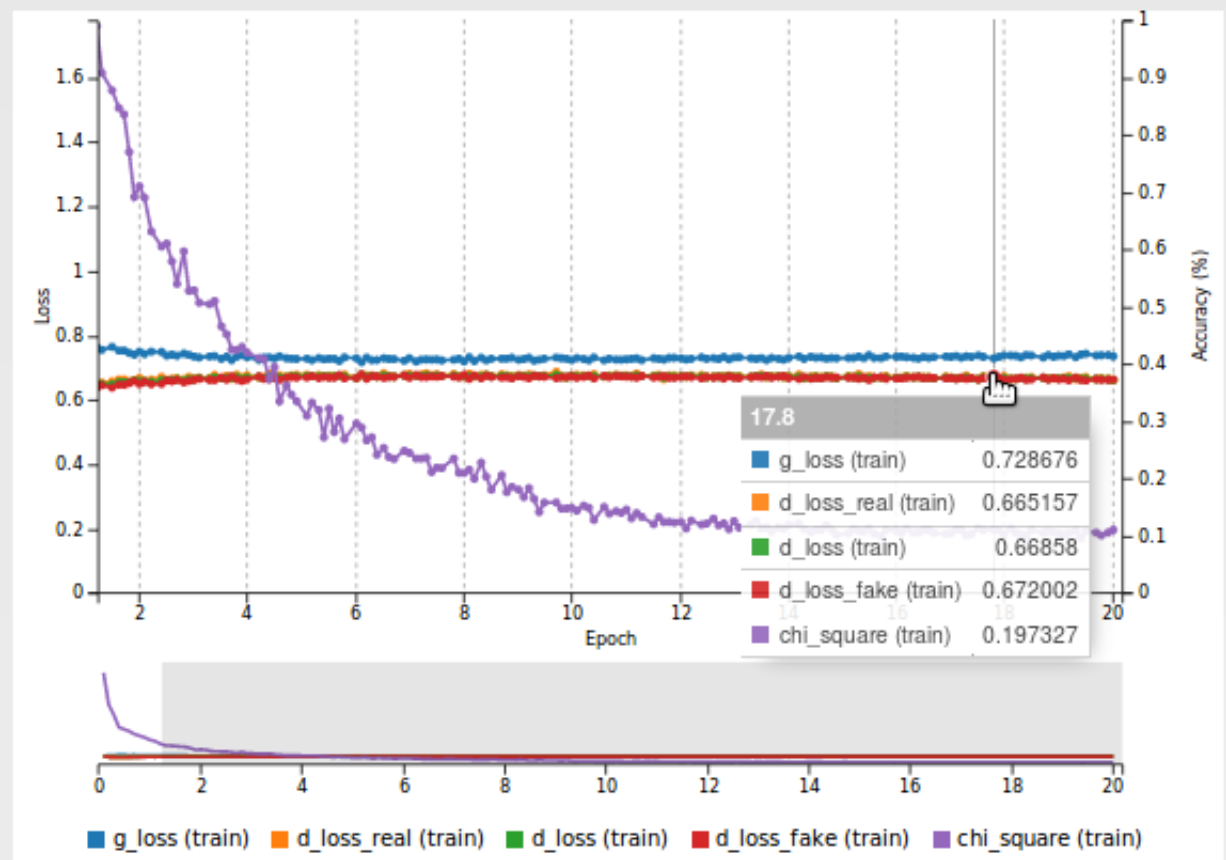


Figure 9: DIGITS visualization of the loss curve when training a GAN on MNIST.

Sampling the model

We could pick random values of `z` and generate random images but it would be great if we could more deterministically choose attributes of the images we create. Remember that in our conditional GAN, the latent representation (`z`) and the class labels are separate. Therefore, we

can pick one value of `z` and generate matching images for all classes. We can also do a class sweep: pick one `z` and slowly interpolate across classes to get smooth transitions between digits:

On the model page, select the `GAN` inference method and the `GAN` inference form. In the inference form, select the `MNIST Class sweep` task.

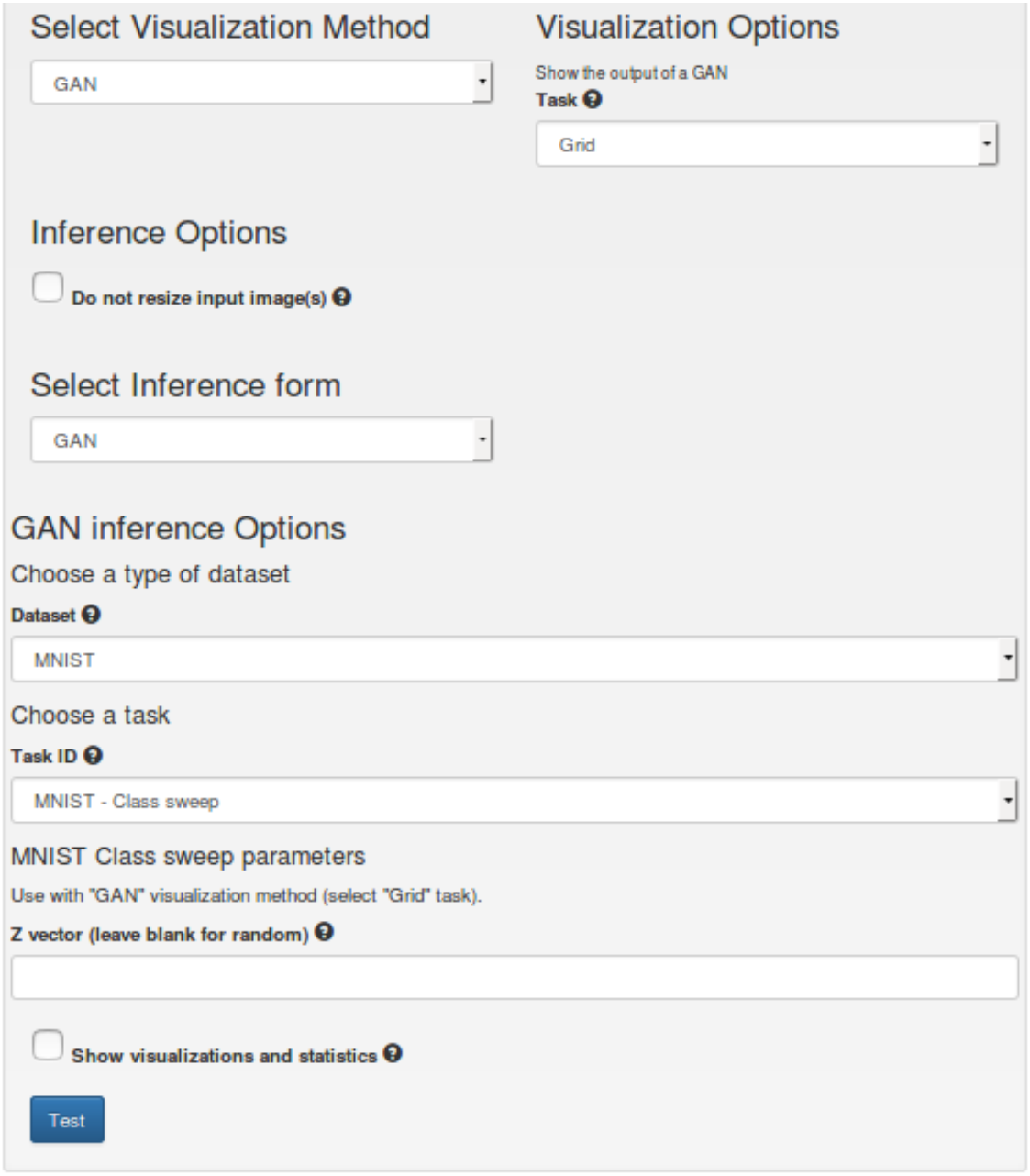


Figure 10: class sweep inference form

Click `Test`. This shows a grid of digits, all of them were sampled using the same randomly generated `z`. The `z` vector is then concatenated with various shades of labels, using spherical interpolation. Every column shows how a digit is slowly morphing into the next digit:

Figure 11: class sweep visualization

You can start to get a feeling of what is going on: style features like stroke width and curvature seem to be encoded in the latent representation. In other words, each grid seems to use a different style of handwriting.

This can also be visualized with an animated gif:

Figure 12: class sweep animation

You can also see on the animations on Figure 12 that transitions between digits are very smooth, an indication that the latent space yields a continuous space of real-looking images.

Now we can also do a style sweep: in the inference form, select `MNIST Style sweep` and click `Test`. This shows a new grid of digits. Every column shows how a digit is slowly morphing from one "style" (i.e. one randomly generated z vector) into another style:

Figure 13: style sweep animation

Now you might wonder how to do the interpolations in practice. Intuitively we would use a linear combination of my z vectors to perform interpolation in latent space. However intuition rarely works in a high-dimensional space. Tom White (Tom White (2016). Sampling Generative Networks. arXiv:1609.04468) pointed out that linear interpolation (taking the shortest path between points) leads to intermediate points that have very improbable norm and are therefore unlikely to be visited during training. That means the model may not perform well on these points. This led White to advocate the use of spherical interpolation: interpolating between two points as if walking on the surface of a high-dimensional sphere. See Figure 14 for an illustration of the difference between linear and spherical interpolation in 2D space. Spherical interpolation has become the standard way of performing interpolation in GANs. In this lab we are using spherical interpolation in all examples.

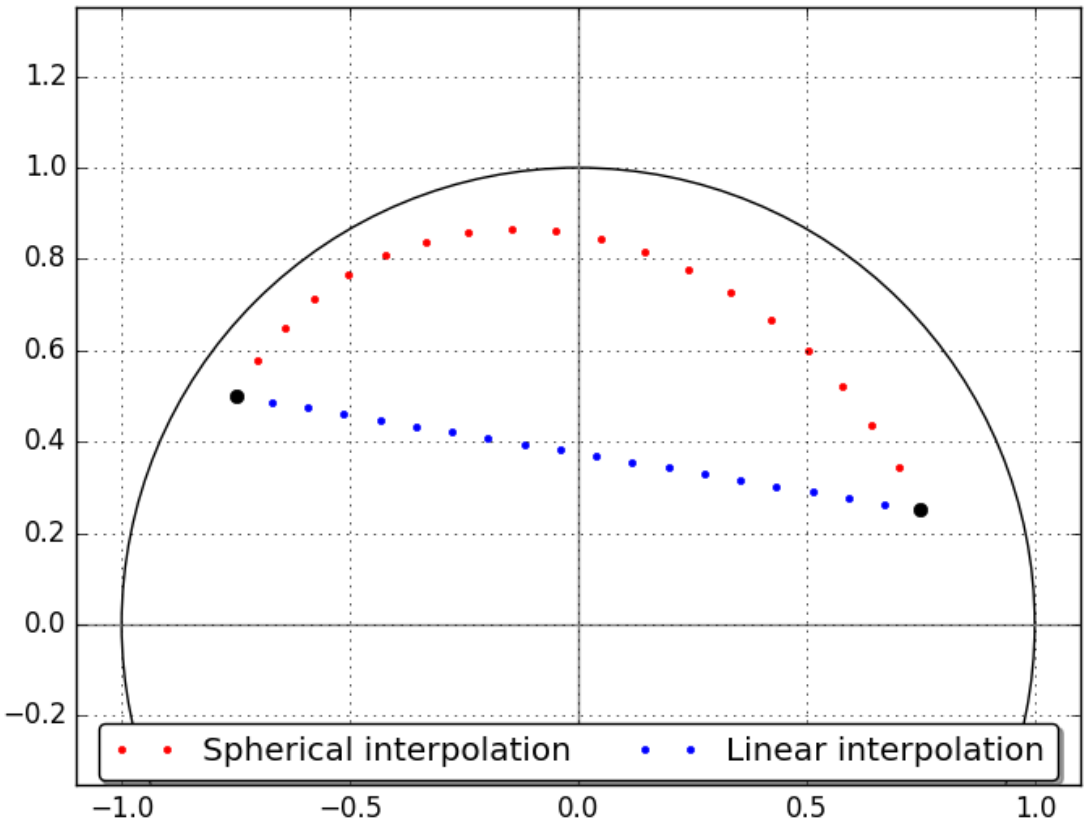


Figure 14: Illustration of the difference between spherical and linear interpolation between two points (black dots) in 2D space.

Spherical interpolation is easily implemented in Python:

```
def slerp(val, low, high):
    """Spherical interpolation. val has a range of 0 to 1."""
    if val <= 0:
        return low
    elif val >= 1:
        return high
    omega = np.arccos(np.dot(low/np.linalg.norm(low), high/np.lin
alg.norm(high)))
    so = np.sin(omega)
    return np.sin((1.0-val)*omega) / so * low + np.sin(val*omega)
/so * high
```

Training an encoder

We can keep sampling random `z` vectors until we find a style that we like. But is there a more direct approach? What if we have an image of a digit in the dataset that we really like and we would like to generate all other digits using the same style?

One limitation in the standard formulation of a GAN is that there is no mapping from feature space to latent space. In other words, given a sample, there is no direct way to find the corresponding z vector. We would like to separately train an encoder, that is to say a network that outputs a z vector when given an image as input. To verify that the `z` vector is correct we can feed it to `G` and then check whether the generated image is similar to the input image.

One question that arises is: how do we design a network that can extract features from images that are useful enough to produce a good `z` vector? Well, we already have such a network: `D`! We just need to change the number of output neurons in the last layer of `D` to 100 (the length of `z`), plug the output of the modified `D` (let's call it `E`) into `G` and use the L2 distance between the output of `G` and the input image as a loss. We have essentially reversed the order of things in GAN and created a GAN Auto-encoder. This is utterly simple but works well enough in practice. See Figure 15 for an illustration of the encoder topology and Figure 16 for an illustration of the new set-up.

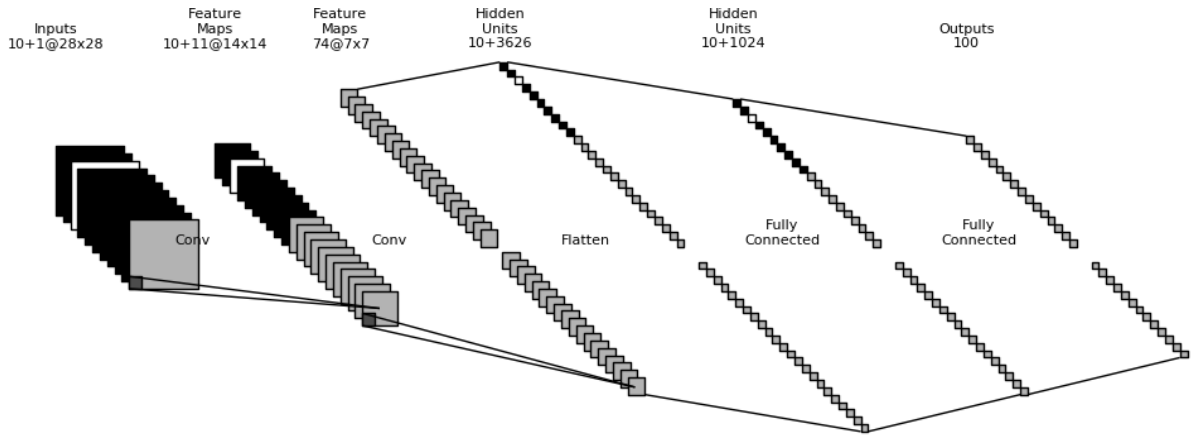


Figure 15: Encoder (`E`) network topology. `E` is the same as `D` with more output neurons in the final layer.

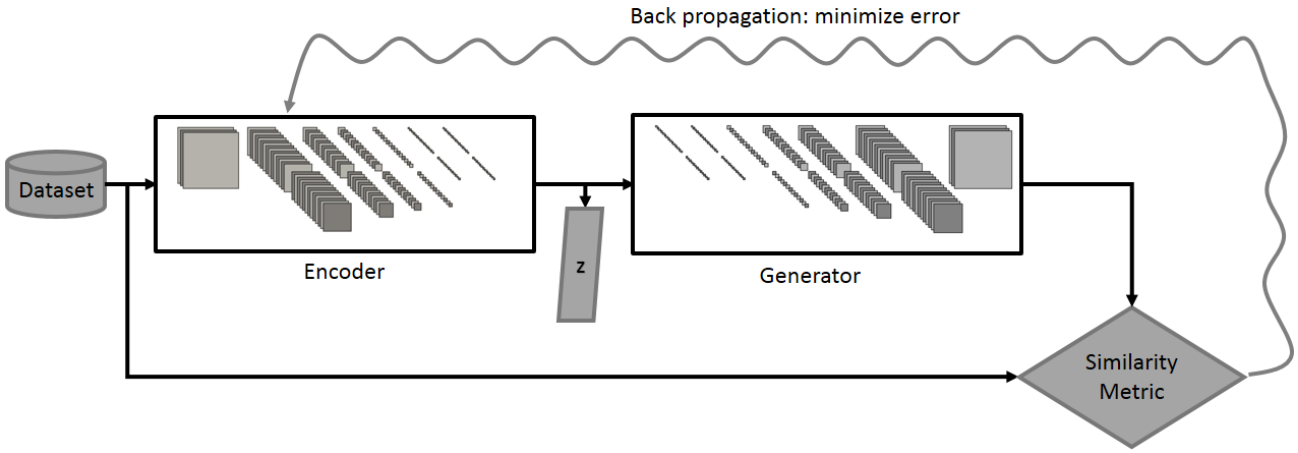


Figure 16: Encoder framework: the modified `D` is now an encoder `E` with 100 output neurons that plugs into `G`.

Importantly, in order to train `E` it is critical to initialize its weights from the weights of a trained `D`. In Deep Learning jargon, this is known as Transfer Learning.

To train the encoder, do this:

- clone the `GAN-MNIST` model
- in the `Previous networks` tab, select the GAN model, select the last epoch then click `Customize`
- use this network description (https://raw.githubusercontent.com/gheinrich/DIGITS-GAN/DIGITS-GAN-v0.1/examples/gan/network-mnist-encoder.py).

Name your model `GAN-MNIST-Encoder` and click `Create`.

Notice that the loss converges smoothly towards a low plateau.

Example digits "3"

Now that we have an encoder we can encode an image and find the corresponding `z` vector. On the `GAN-MNIST-Encoder` page,

- select the `GAN` visualization method and the `MNIST encoder` task,
- upload an image of a `3` from MNIST (you can use the above example: save it to your computer and upload the image using the form),
- click `Test One`

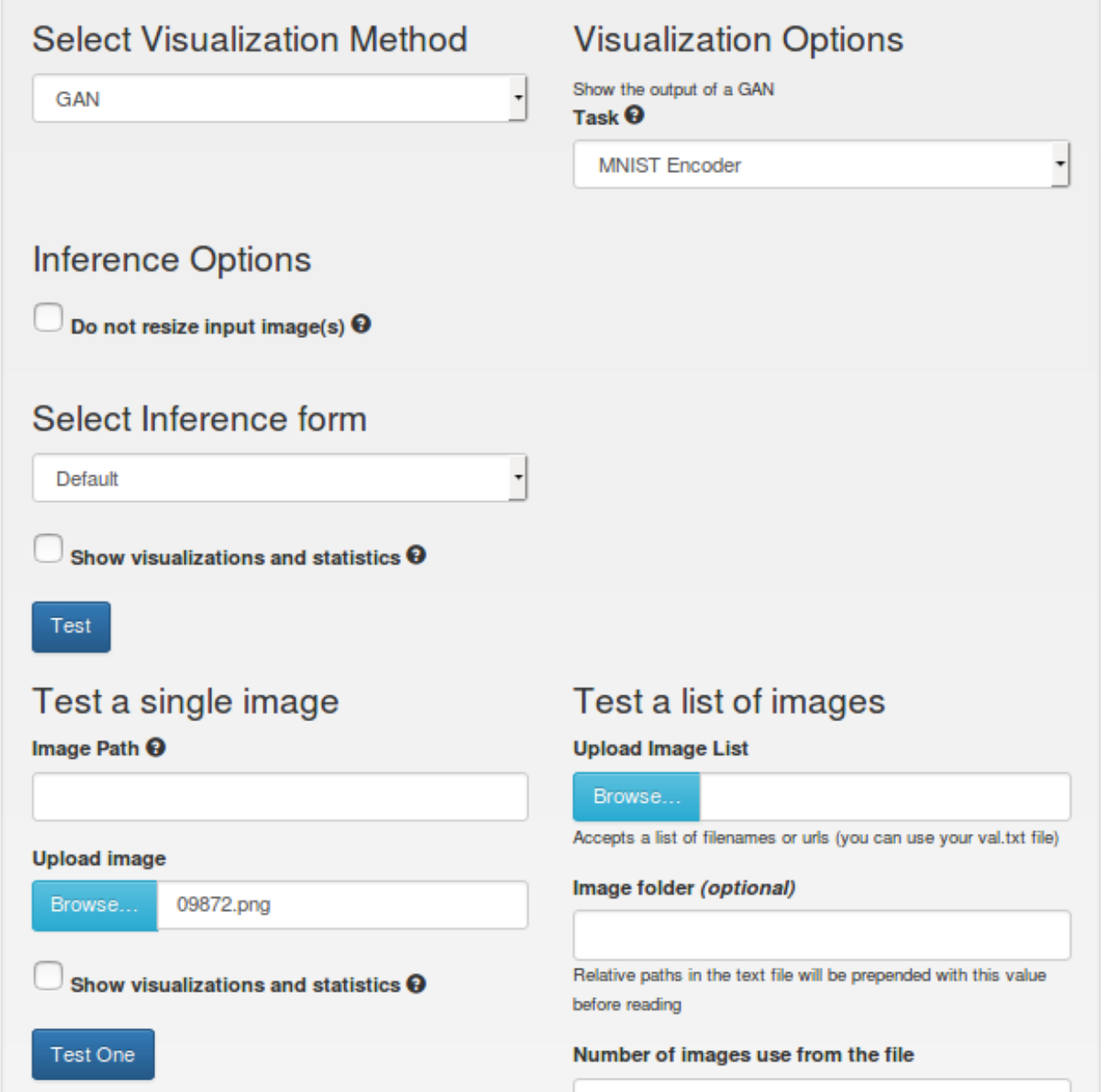


Figure 17

> Note: in this GAN-encoder, class "3" is hard-coded in the encoder model **during inference**. If you want to encode another class you will need to manually update this line in the model description: `self.y = tf.to_int32(3*tf.ones(shape=[self.batch_size]))`

On the resulting page, you can see the input image (left), the reconstructed image (middle), and the corresponding `z` vector (right) (Figure 18)

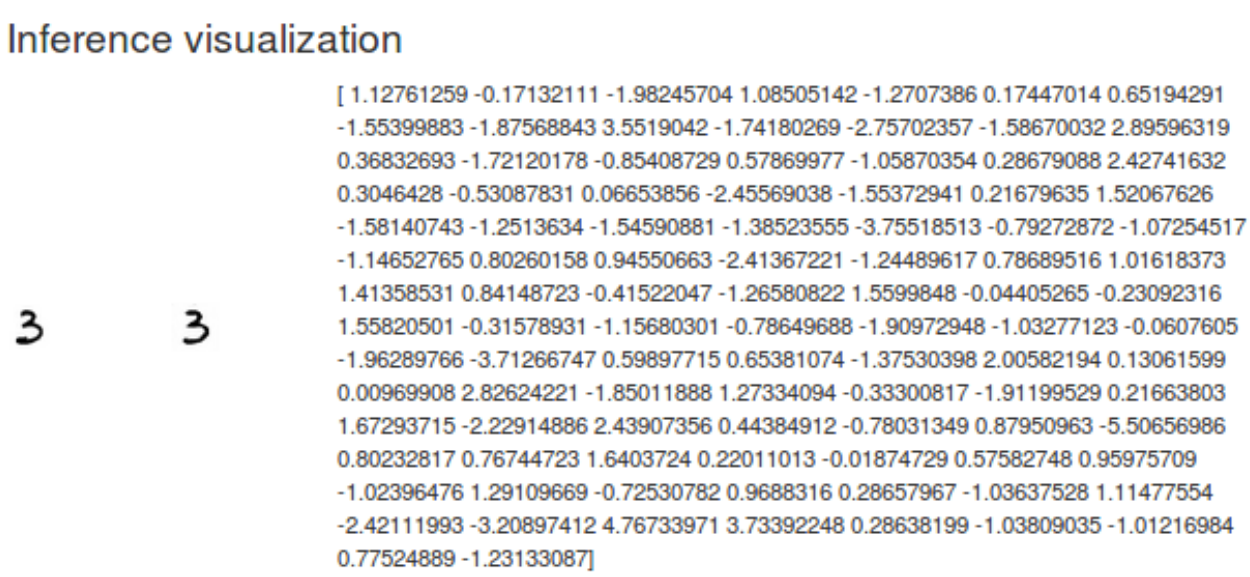


Figure 18

Now that we have a `z` vector, we can do a class sweep for that particular "style":

- copy the encoded `z` vector from the encoder.
- move to the `GAN-MNIST` model page.
- select the `GAN` visualization method and the `Grid` task,
- select the `GAN` inference form and the `Class sweep` task,
- paste the encoded `z` vector into the field:

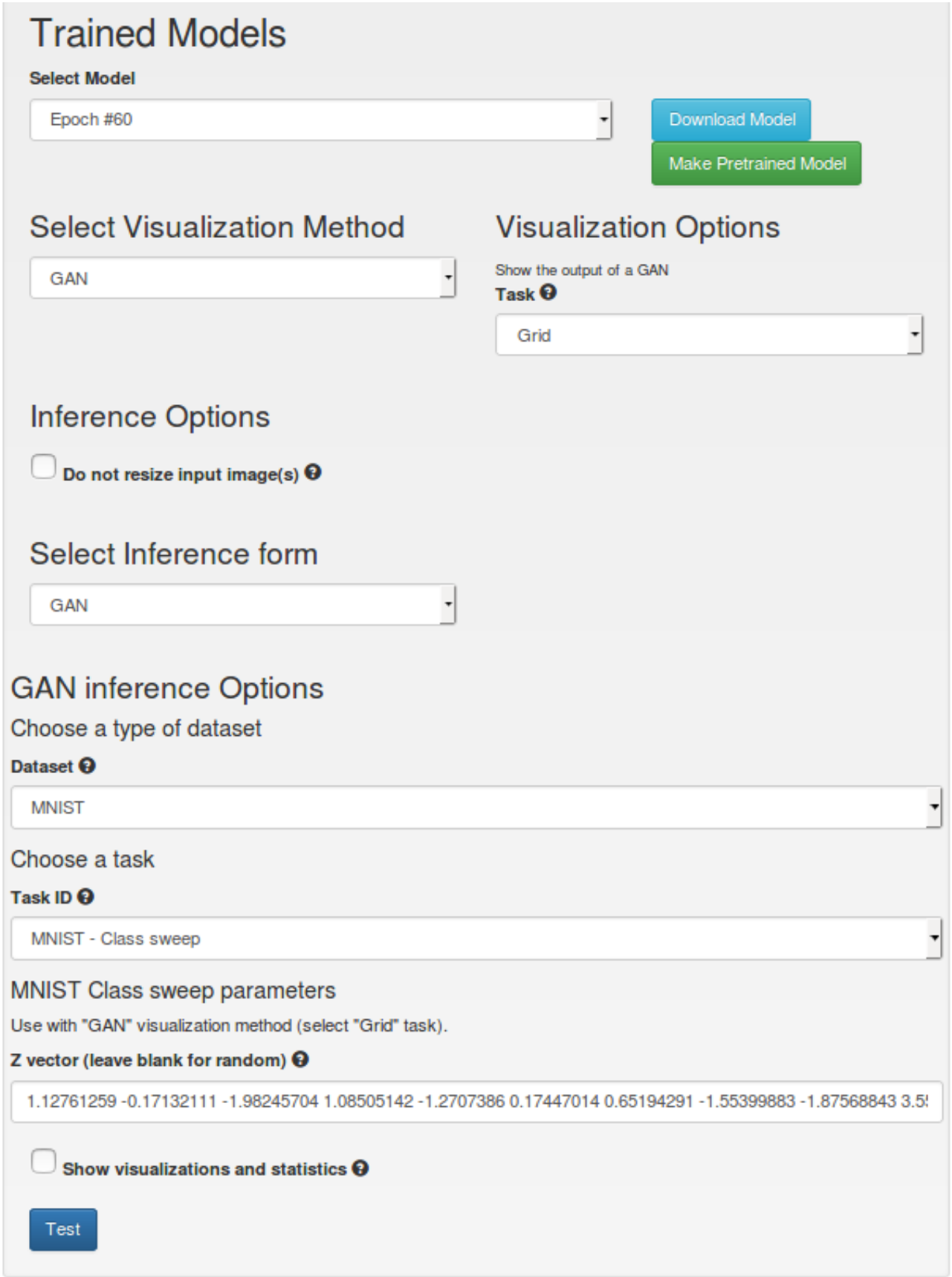

Trained Models
Select Model
Epoch #60
Download Model
Make Pretrained Model
Select Visualization Method
GAN
Visualization Options
Show the output of a GAN
Task
Grid
Inference Options
Do not resize input image(s)
Select Inference form
GAN
GAN inference Options
Choose a type of dataset
Dataset
MNIST
Choose a task
Task ID
MNIST - Class sweep
MNIST Class sweep parameters
Use with "GAN" visualization method (select "Grid" task).
Z vector (leave blank for random)
1.12761259 -0.17132111 -1.98245704 1.08505142 -1.2707386 0.17447014 0.65194291 -1.55399883 -1.87568843 3.5
Show visualizations and statistics
Test

Figure 19

Now click `Test` and you will see a class sweep using the particular style that you specified:

Celebrity faces

We can train a model on the CelebFaces dataset, a dataset of Celebrity Faces:

> Ziwei Liu and Ping Luo and Xiaogang Wang and Xiaoou Tang (2015). Deep Learning Face Attributes in the Wild. Proceedings of International Conference on Computer Vision (ICCV).

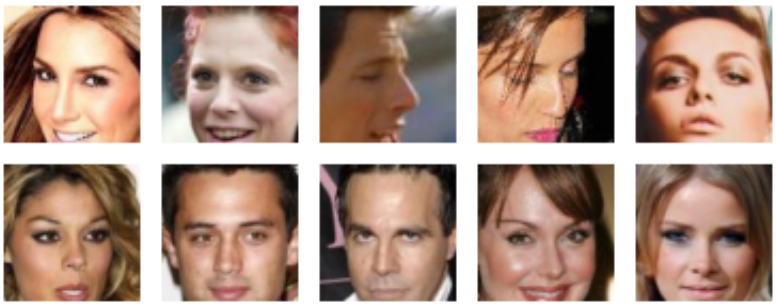


Figure 21: samples from CelebFaces

It would take too long to train a model on this dataset during the lab so we have provided pre-trained GAN and encoder models: `CelebFaces_GAN` and `CelebFaces_Encoder`.

The model topology is very similar to what we used in the previous sections. Most notable, this model does not use conditioning:

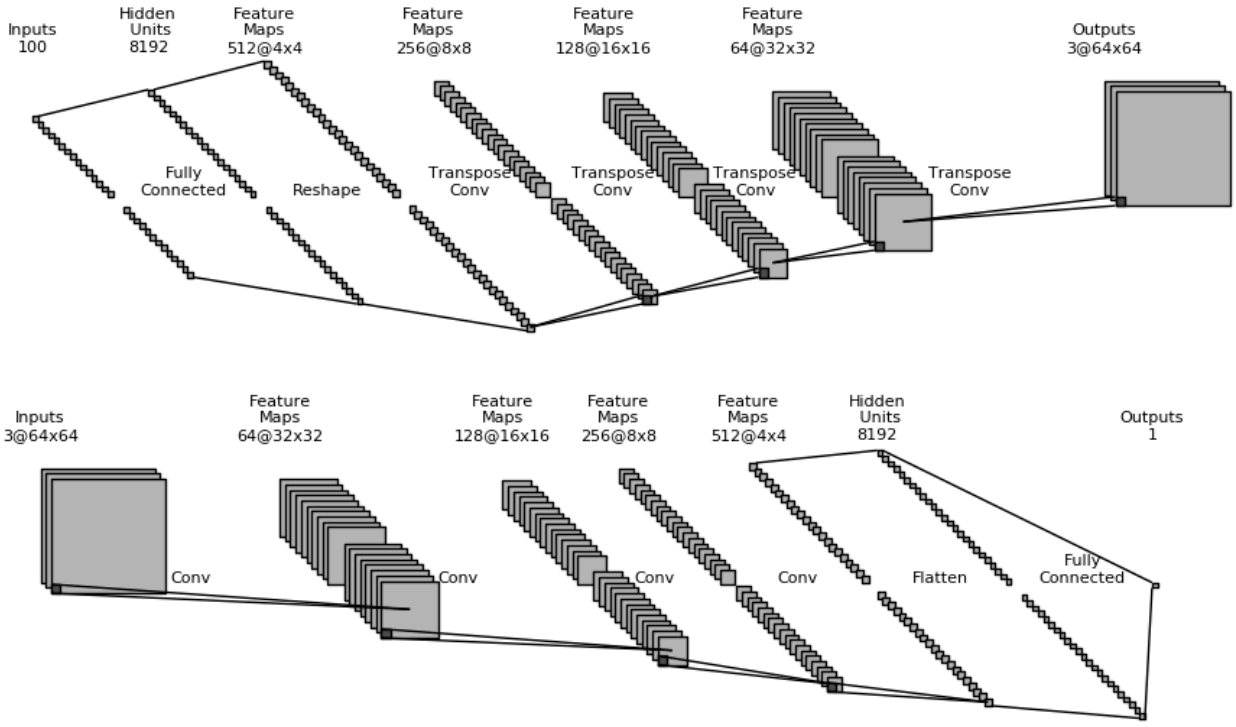
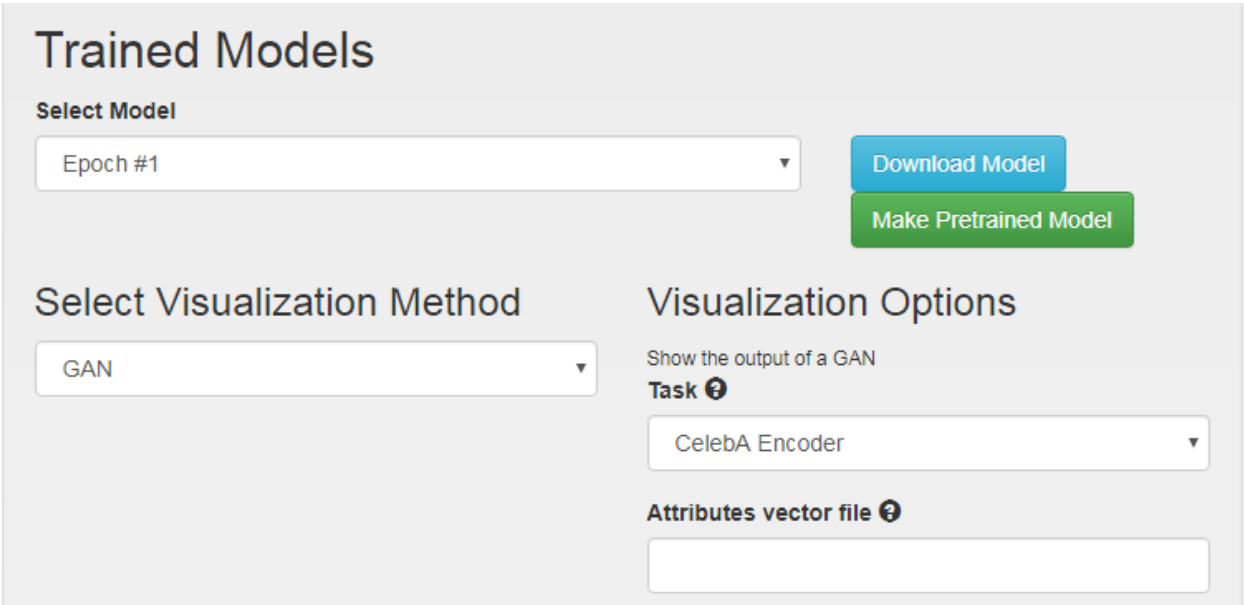


Figure 22: DCGAN topology

Using the encoder, you can find `z` vectors by encoding images from the dataset through the GAN-CelebA-Encoder model:

- move to the model page for `CelebFaces_Encoder`,
- select the `GAN` visualization method, select the `CelebA Encoder` task,
- select the `GAN` inference form, select the `CelebA - Encode list` task,
- specify the path to a file list (`/data/gan-data/list_attr_celebs.txt`) and the path to the image folder (`/data/gan-data/samples`).



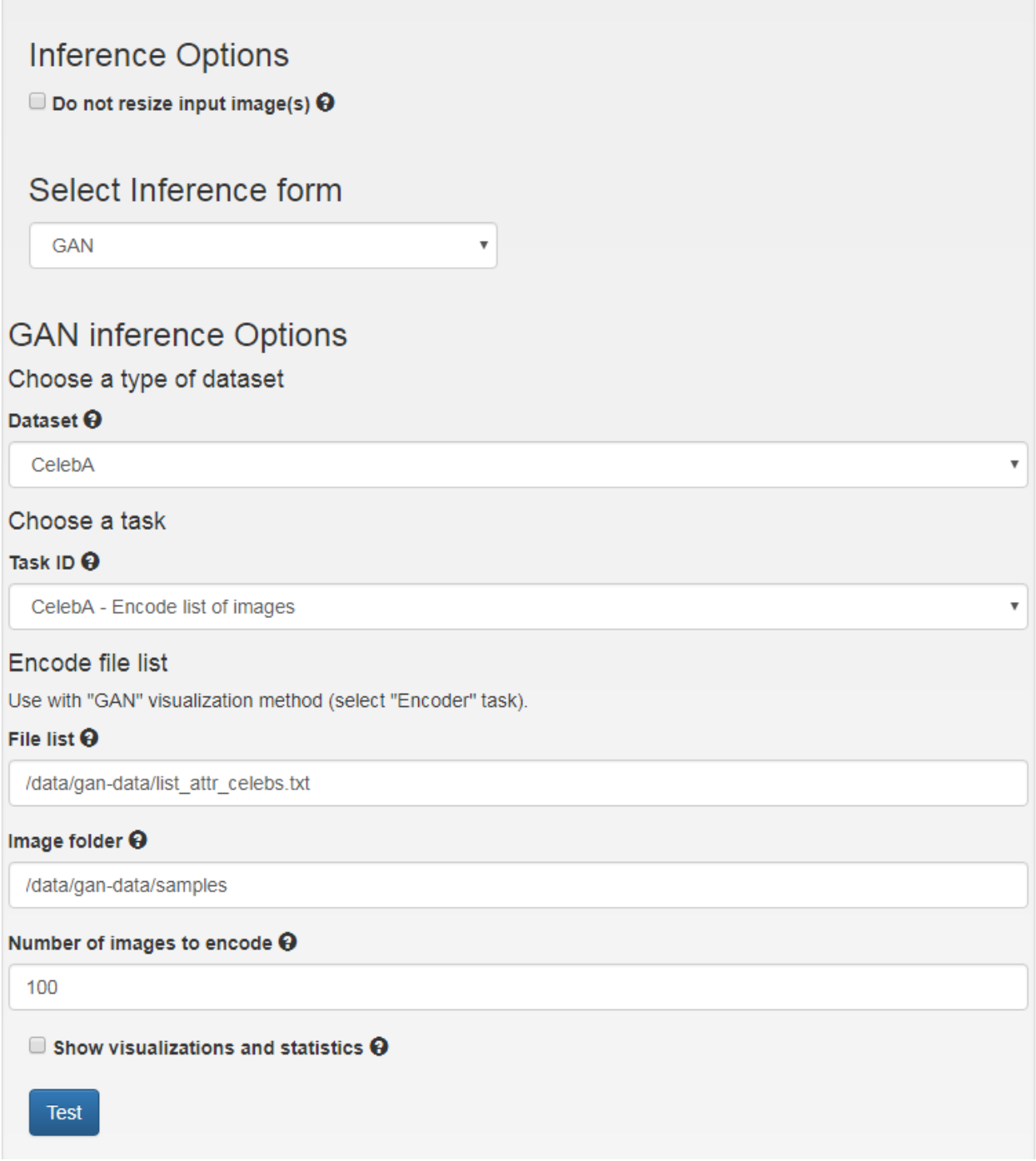
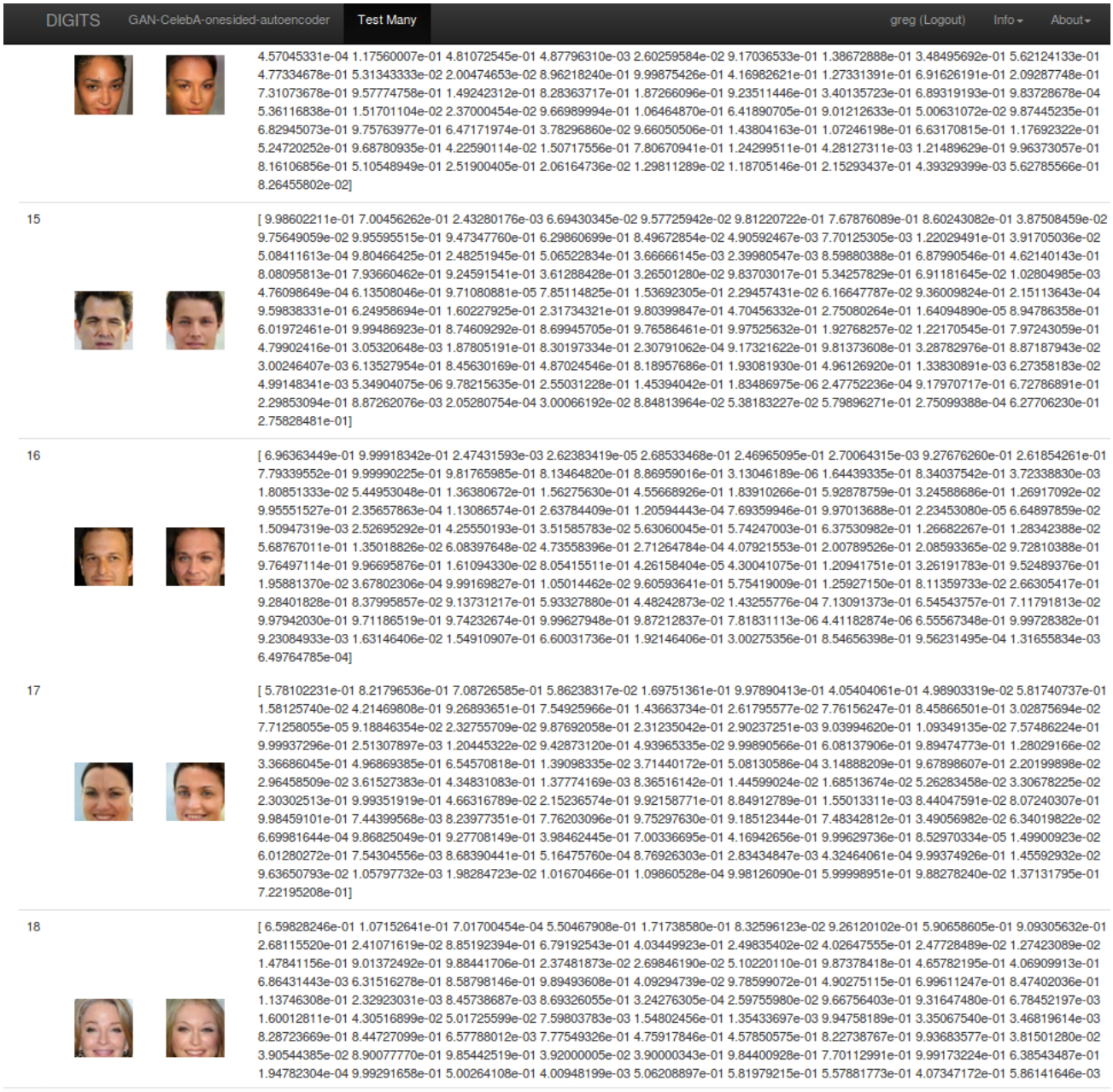


Figure 23: List encoding form

Click `Test`. You may see something like:



.

Figure 24: Every row is showing the input image (left), the reconstructed image (center) and the corresponding z vector (right).

Attribute vectors

Images in CelebA come with 40 binary attributes. It would be nice to be able to take an image of a face and modify it to make her look younger or change the color of her hair. Remember that one of the promises in GANs is that you can perform operations in latent space and have them reflected in feature space. In order to modify attributes we need to find a `z` vector for each of the attributes so what we did for you in this lab is we used `E` to find the `z` vector associated with every single image in the dataset. Then we calculated attribute vectors as follows: for example,

to find the attribute vector for “young” we subtracted the average z vector of all images that don’t have the “young” attribute from the average z vector of all images that have it. We ended up with a 40x100 matrix Zattr of characteristic z vectors, one for each of the 40 attributes in CelebA. We have stored this matrix in `/data/gan-data/attributes_z_20170202-000712-f3c3_20170314-225136-42aa.pkl` on the lab machine.

Sampling the CelebA model

Setting image attributes

Now if you pick a `z` vector, you can add/remove attributes to this image:

- open the `CelebFaces_GAN` model page,
- select the `Image output` visualization method and set the `Data order` to `HWC`:

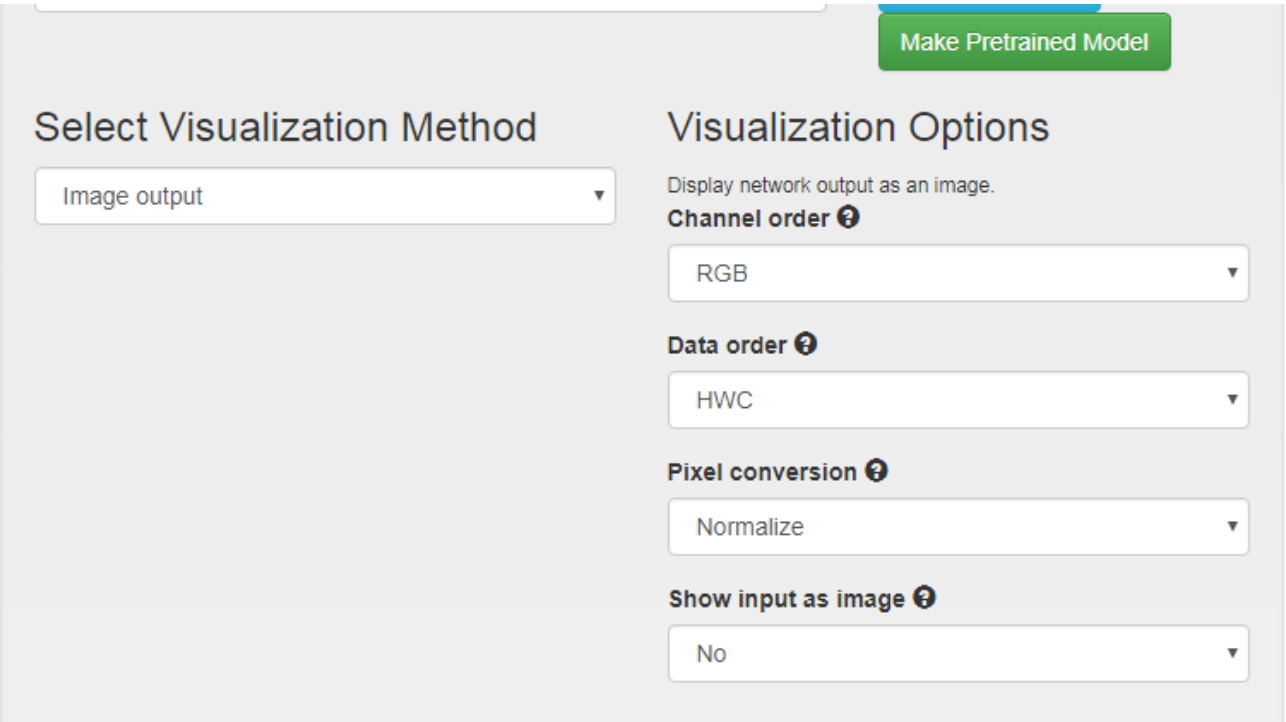


Figure 25

- select the `GAN` inference form and select the `CelebA - add/remove attributes` task,
- specify the location of the attributes file `/data/gan-data/attributes_z_20170202-000712-f3c3_20170314-225136-42aa.pkl`.
- paste the `z` vector (list without brackets) you found when using the `Encode list` task above.
- click `Add row` a number of times to create new rows. Each row will generate an image with the corresponding attributes. If you leave all cells in a row blank, you will get the original image. If you set `Black Hair` to `+1` and `Blond Hair` to `-1`, this will transform

a blond person into a person with dark hair. If you set `Smiling` to `-1` this will make a smiling person... not smile.

See for example:

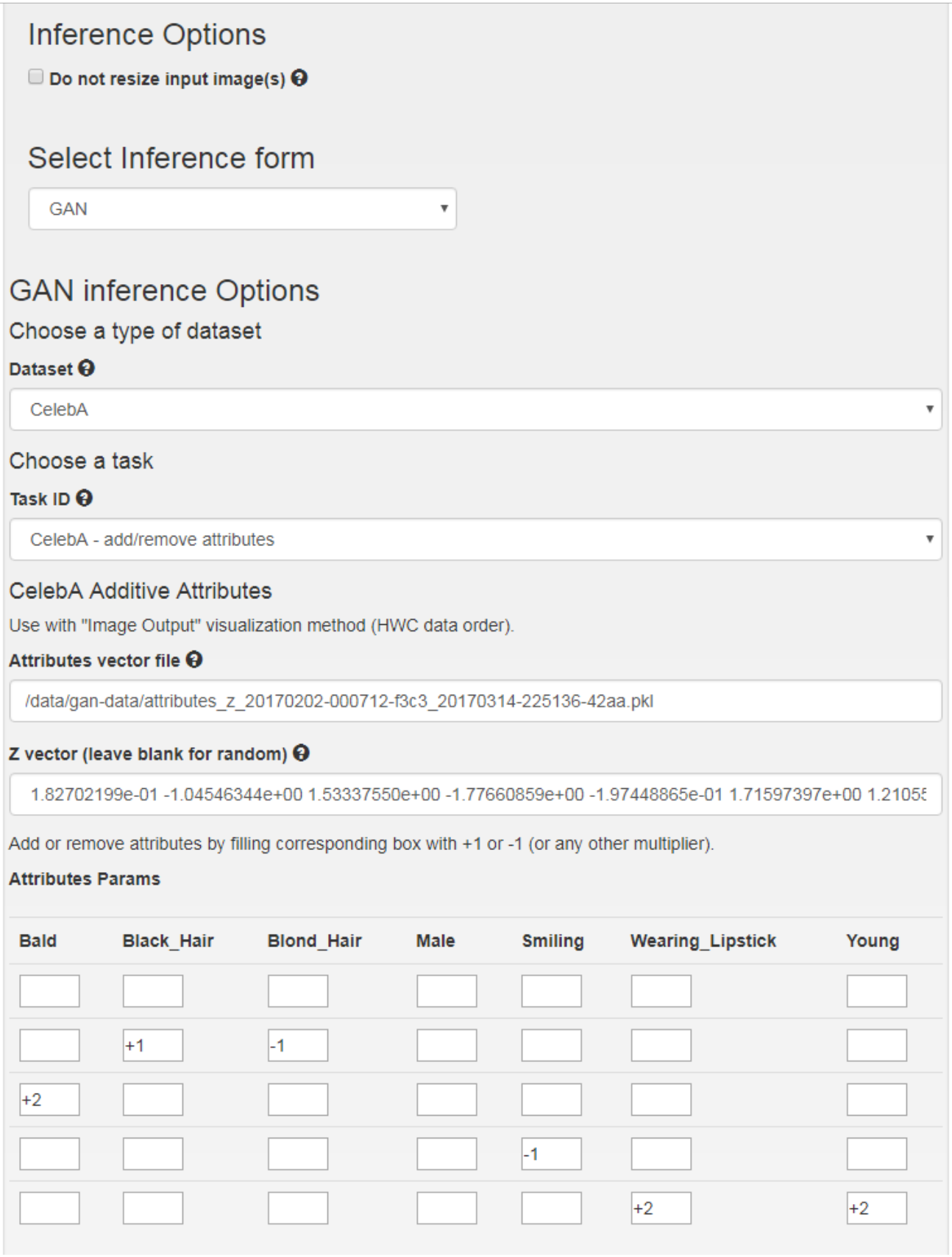

Inference Options
Do not resize input image(s)
Select Inference form
GAN
GAN inference Options
Choose a type of dataset
Dataset
CelebA
Choose a task
Task ID
CelebA - add/remove attributes
CelebA Additive Attributes
Use with "Image Output" visualization method (HWC data order).
Attributes vector file
/data/gan-data/attributes_z_20170202-000712-f3c3_20170314-225136-42aa.pkl
Z vector (leave blank for random)
1.82702199e-01 -1.04546344e+00 1.53337550e+00 -1.77660859e+00 -1.97448865e-01 1.71597397e+00 1.2105
Add or remove attributes by filling corresponding box with +1 or -1 (or any other multiplier).
Attributes Params
Bald Black_Hair Blond_Hair Male Smiling Wearing_Lipstick Young
+1 -1
+2
-1
+2 +2

Figure 26

This will generate these images:

Index	Data
1	
2	
3	
4	
5	

Figure 27

Analogy grid

Ever heard about the analogy “king - man + woman = queen”? The GAN-Auto-Encoder framework allows us to perform the same analogies on images, using simple arithmetics in latent space.

- open the `CelebFaces_GAN` model page,
- select the `GAN` visualization method, select the `Grid` task,
- select the `GAN` inference form, select the `CelebA - analogy` task,
- set the `source`, `sink 1` and `sink 2` `z` vectors (without brackets).

This will create a grid with the following analogy: `destination = sink 1 + sink 2 - source` with:

- `source` in top-left corner,
- `sink 1` in top-right corner,
- `sink 2` in bottom-left corner,
- `destination` in bottom-right corner.

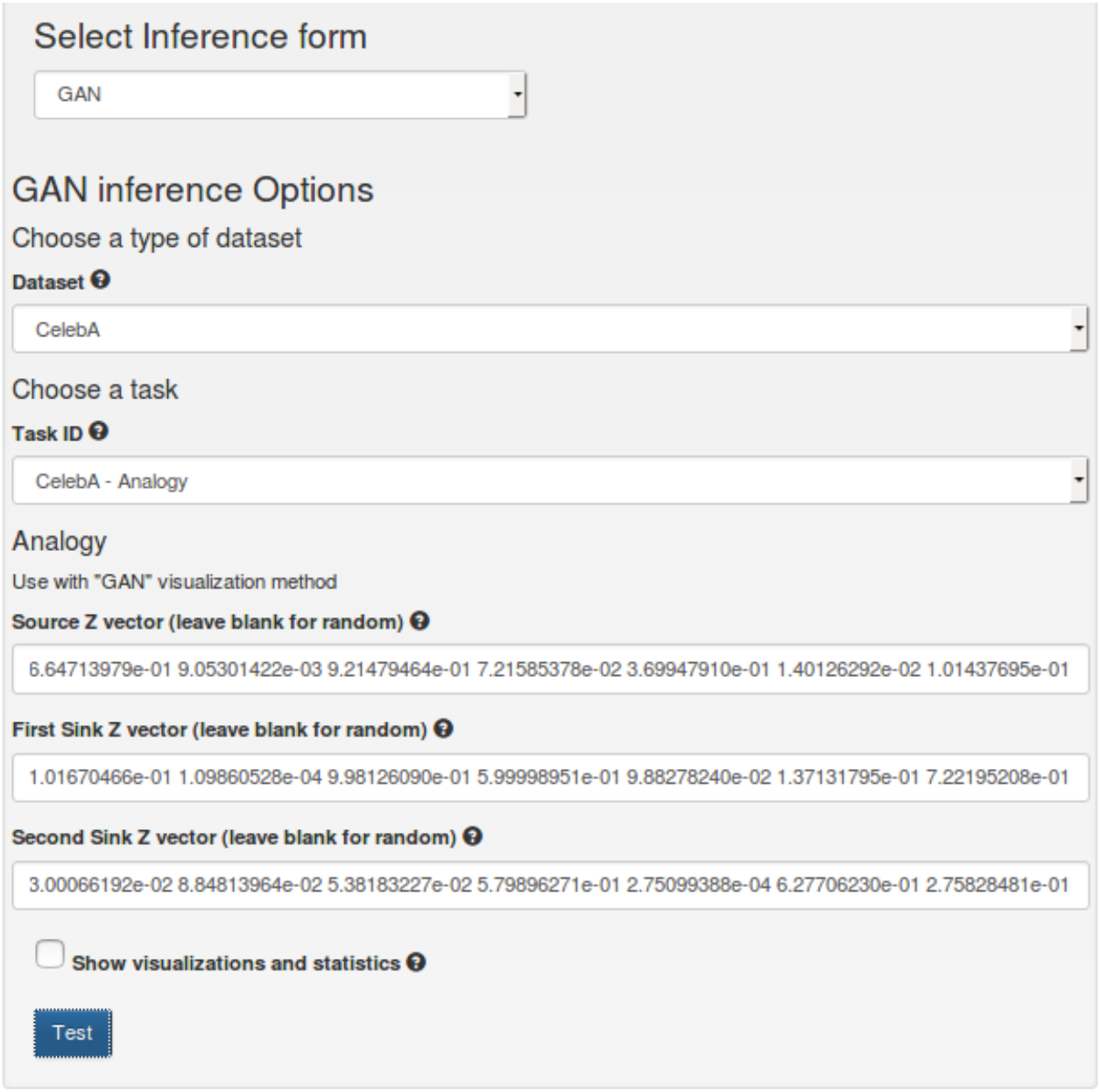


Figure 28: analogy inference form

For example you might see something like:

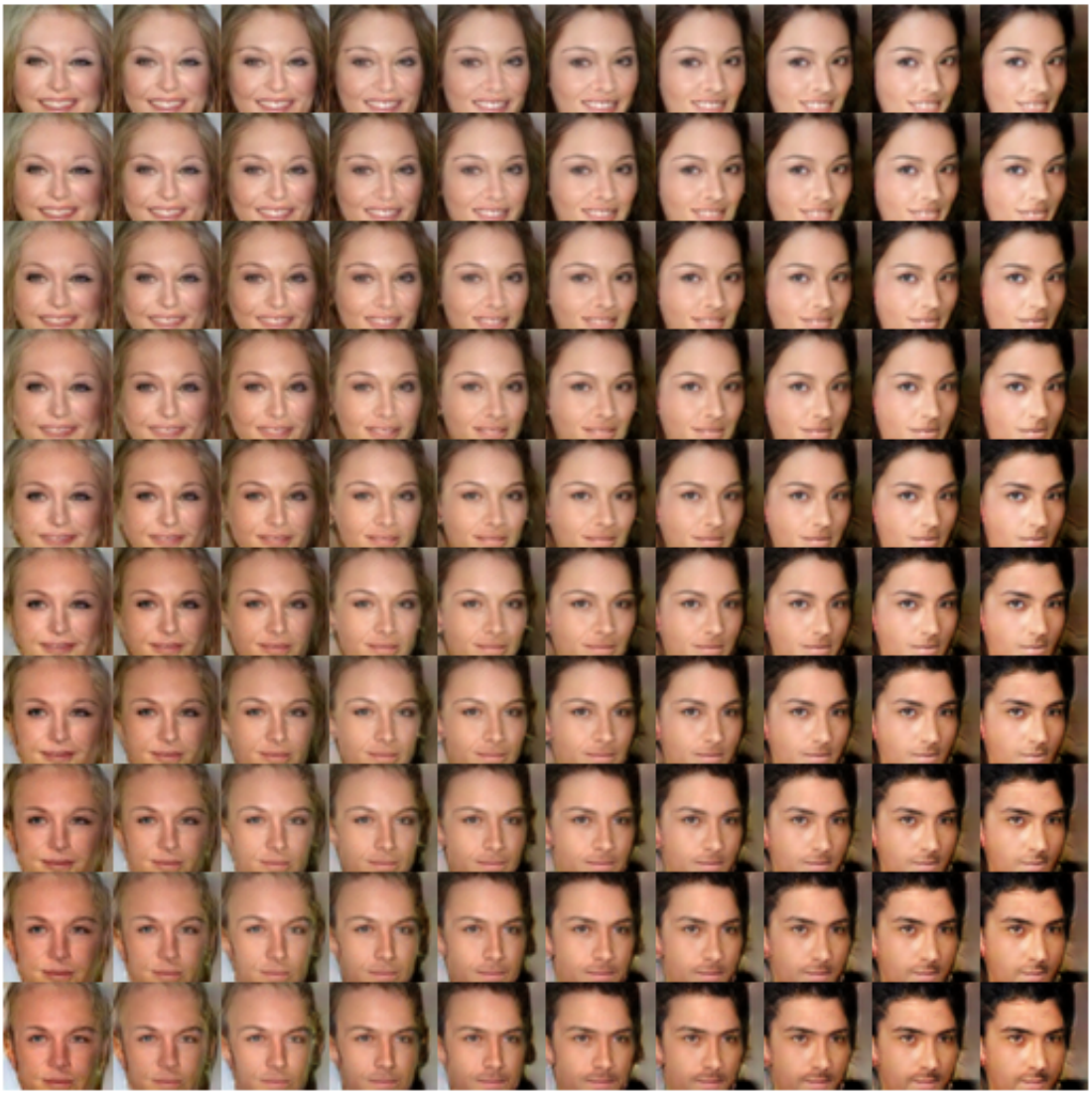


Figure 29: analogy visualization

Attribute extraction

Another fun use of face attributes is to let the model tell us what the main attributes in a face are. You can use `CelebFaces_Encoder` to generate the z vector for an image and then we can compute this inner product of this z vector with each of the attribute vectors to find the most prominent attributes.

- open the `CelebFaces_Encoder` model page,
- select the `GAN` visualization method, select the `CelebA get attributes` task, point to the location of the attributes matrix `/data/gan-data/attributes_z_20170202-000712-f3c3_20170314-225136-42aa.pkl`,
- select the default inference form
- in `image path` specify the path to an image, e.g. `/data/gan-data/lecun2.png` or upload your own image,

- click `Test One`.

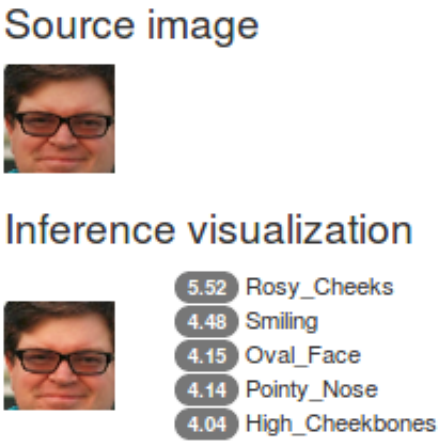


Figure 30: attribute extraction

With GANs, you can us AI to form any image you want based on the attributes you specifiy amongst many possibilties. Using the same base source code (https://github.com/carpedm20/DCGAN-tensorflow) that you used in this lab, projects like Neural Face (http://carpedm20.github.io/faces/) have become a reality: