Conditional GANs

Pokemon's type generation from scratch - Deep Learning course's exam

Matteo Robbiati

July 2022

GANs

Generator and discriminator

➔ Generative Adversal Networks (aka GANs) are models composed of two neural networks:

- a **generator** G, which goal is to understand the data distribution ρ_{ds}. In order to do that, it takes random vector z from a latent space $\mathcal{Z}$[1] with a probability density function ρ_z[2] and maps it into an element with the same dimensionality of the data sample, building a fake sample space $\hat{\Omega}$, e.g. images in the form $(64, 64, 3)$:
$$G : z \in \mathcal{Z} \rightarrow G(z|\vec{\theta}_g) \in \hat{\Omega}.$$
- a **discriminator**, D, which goal is to classify an input variable x as *generated* (0) or *real* (1):
$$D : x \in \{(64, 64, 3)\} \rightarrow D(x|\vec{\theta}_d) \in \{0, 1\}$$

[1] with a lower dimensionality than the sample space

[2] typically in a simple way, e.g. through an uniform or a normal distribution

The loss function to optimize

➔ G wants D classifies all its purposes as 1, instead D wants to be able to classify correctly every input it receives. To summing up this conflict between the two network we build a **min-max game**:

$$\min_{\vec{\theta}_g} \max_{\vec{\theta}_d} \left\{ \left\langle \log D(x|\vec{\theta}_d) \right\rangle_{x\sim\rho_{ds}} + \left\langle \left[1 - \log D\left(G(z|\vec{\theta}_g)|\vec{\theta}_d\right)\right] \right\rangle_{z\sim\rho_z} \right\} \tag{1}$$

```
# the pseudo-algorithm
for n epochs do:
    for k steps do:
        let D classify an hybrid batch of data
        update the discriminator weights with a B-P
    generate a new fake batch using G
    freeze D weights and classify the elements
    update the generator weights with a B-P
```

➔ This idea of optimization is applied also in the case of conditional generative adversal nets.

Conditional GANs

A conditioned problem

- Conditional GANs were born for implementing a model that was able to generate conditioned data.
- The idea is to feed both G and D with an extra information y, which can be arbitrarily implemented[3].

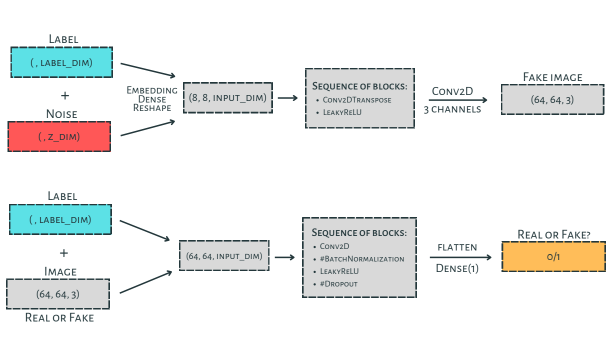


Figure 1: Above, G diagram. Below, D diagram.

[3] I treated two methods. In method 👆 i used functional API mode for passing G and D two inputs, with method ✌ i merged all inputs into a single array and built sequential models

This problem

My problem

➔ I want to generate 64×64 **RGB images** of pokemon of three different types: *grass*, *water* and *fire*.

➔ i managed this dataset, which contained initially $\sim$ 17.3k images:

- **deleted** all folders containing pokemon **different** from *water, grass* or *fire* (from 149 to 50 directories);
- **standardized** the images to a 64×64 RGB and `*png` format (removed 208 images incompatible with the RGB-png conversion);
- **removed** all the images **without a white background**. For this step I've written a function which evaluate the percent value of *whiteness* of an image (removed 2450 images);

➔ At the end of the pre-processing i found a dataset of 1064 images divided into three type-classes, which i put into a `tf.keras.Dataset`.

A random sampling

How i built the dataset

➔ Since `cgan_ds` contains only a few elements, i have decided to apply a data augmentation strategy during the process of ds definition:

```python
#a bit of data augmentation
data_augmentation = tf.keras.models.Sequential(
  [
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
  ]
)

#prepare function
def prepare(ds, shuffle=True, augment=True):
  ds.cache()
  if shuffle:
    ds = ds.shuffle(6000)
  ds = ds.batch(batch_size)
  if augment:
    ds = ds.map(lambda x, y: (data_augmentation(x), y), num_parallel_calls=BUFF_SIZE)
  return ds.prefetch(buffer_size=BUFF_SIZE)

#prepare my dataset as decided
dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
dataset = prepare(dataset)
```

A random sampling

Conditioning the generator - Method

With the first way we want to embed all infos into **sheets**, from which we will get a (64,64,3) image.

1. the label is passed as integer end embedded into a vector with an arbitrary dimension which we call **embed_label**. Then, using a `Dense` layer composed of 8×8 nodes and a `Reshape` layer we transform the **embed_label** into a (8,8,1) sheet;
2. **noise** z is passed as a random vector of dimension 100, from which we build N_s sheets using a `Dense` with $N_s \times 8 \times 8$ nodes and a `Reshape` layer;
3. with a `Concatenate` layer we **combine the sheets** composing all the amount of input information, in the form $(8, 8, N_s + 1)$.

```
#embedding the label
input_label = Input(shape=(1,))
embed_label = Embedding(num_classes, 100)(input_label)
nodes = 8 * 8
embed_label = Dense(nodes)(embed_label)
embed_label = Reshape(8, 8, 1))(embed_label)
#embedding the noise
latent_input = Input(shape=(latent_dim,))
nodes = 64 * 8 * 8
gen = Dense(nodes)(latent_input)
gen = Reshape((8, 8, 64))(gen)
```

Conditioning the generator - Method ✌

We encode all the information (both noise and class label) into a single input vector, from which we get the same sheets.

1. Once calculated the **input dimension**: $N_{in,G} = N_{classes} + z_{dim}$;
2. we transform the input information using a `Dense` layer with $8 \times 8 \times N_{in,G}$ nodes and reshape the output into sheets: $(8, 8, N_{in,G})$.

```
generator = keras.Sequential(
    [
        #embedding the input information
        #generator_in_channels is our N_in,G
        InputLayer((generator_in_channels,)),
        Dense(8 * 8 * generator_in_channels),
        LeakyReLU(alpha=0.2),
        Reshape((8, 8, generator_in_channels)),
        #(...)
    ]
)
```

Conditioning the discriminator - Method

We operate in terms of sheets again:

1. the **label** is passed as first input and embedded in the same way of G, with the difference that it is transformed into a sheet of the form $(64, 64, 1)$;
2. the **image** is provided as second input and pasted with the *label sheet* through a `Concatenate` layer.

```
# embedding the label
input_label = Input(shape=(1,))
embed_label = Embedding(1, 100)(input_label)
nodes = 64 * 64
embed_label = Dense(nodes)(embed_label)
embed_label = Reshape((64, 64, 1))(embed_label)
# passing the second input (the RGB image)
input_image = Input(shape=input_shape)
# concatenating the two inputs
merged_image = Concatenate()([input_image, embed_label])
```

Conditioning the discriminator - Method ✌

We build up a sequence of sheets in the form $(64, 64,)$ cointaining the image and the label information.

1. the **label** compose three sheets for a total dimension $(64, 64, 3)$;
2. the **image** compose other three sheets in the same form $(64, 64, 3)$;
3. all inputs are concatenated into six sheets as $(64, 64, 6)$.

```
discriminator = keras.Sequential(
    [
        #discriminator_in_channels is (n_channels + n_classes)
        #so it is an input (64, 64, 6) in this case
        keras.layers.InputLayer((64, 64, discriminator_in_channels)),
        # Convolutions (...)
    ]
)
```

G and D models

- `Conv2D` and `Conv2DTranspose` operations respectively for reducing and increasing the data size

```
#discriminator's core
layers.Conv2D(512, (4, 4), strides=(1, 1), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.Conv2D(256, (4, 4), strides=(1, 1), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.Conv2D(128, (4, 4), strides=(1, 1), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.Conv2D(64, (4, 4), strides=(1, 1), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.GlobalMaxPooling2D(),
layers.Dense(1)

#generator's core
layers.Conv2DTranspose(64, (3, 3), strides=(2, 2), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.Conv2DTranspose(128, (3, 3), strides=(2, 2), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.Conv2DTranspose(256, (3, 3), strides=(2, 2), padding="same"),
layers.LeakyReLU(alpha=0.2),
layers.Conv2D(3, (4, 4), padding="same", activation="sigmoid")
```

Optimization

Optimization with model - metrics

- From now on I will refer to **method two**, with which I obtained the best results.
- I've discovered the possibily **to customize** the `tf.keras.Model.fit` module and I used this approach for training the conditional GAN.
- For doing that, in a `cond_GAN` class, I **customized** three methods `metrics`, `compile` and `train_step`:

```
#metrics
#the mean value of the loss-fz obtained predicting the samples
self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

def metrics(self):
    return [self.gen_loss_tracker, self.disc_loss_tracker]
```

Optimization - compile

- Implementing an **exponential decay** of the learning rate [2] using `keras.optimizers.schedules.ExponentialDecay`;
- using a Binary Crossentropy as loss function.

```
#an exponential decay for the learning rate
initial_learning_rate = 0.0005

lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate, decay_steps=500, decay_rate=0.99, staircase=True)

#the decay's expression
def decayed_learning_rate(step):
  return initial_learning_rate * decay_rate ^ (step / decay_steps)

cond_gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=lr_schedule, beta_1=0.5),
    g_optimizer=keras.optimizers.Adam(learning_rate=lr_schedule, beta_1=0.5),
    loss_fn=keras.losses.BinaryCrossentropy(from_logits=True),
)

# Compile method is defined into the cGAN class
def compile(self, d_optimizer, g_optimizer, loss_fn):
    super(ConditionalGAN, self).compile()
    self.d_optimizer = d_optimizer          #Adam
    self.g_optimizer = g_optimizer          #Adam
    self.loss_fn = loss_fn                  #BinaryCrossentropy
```

Learning rate decay

➔ My schedule performed a decay step each 50 optimization steps (after using 50 batches, so about every 2 epochs).

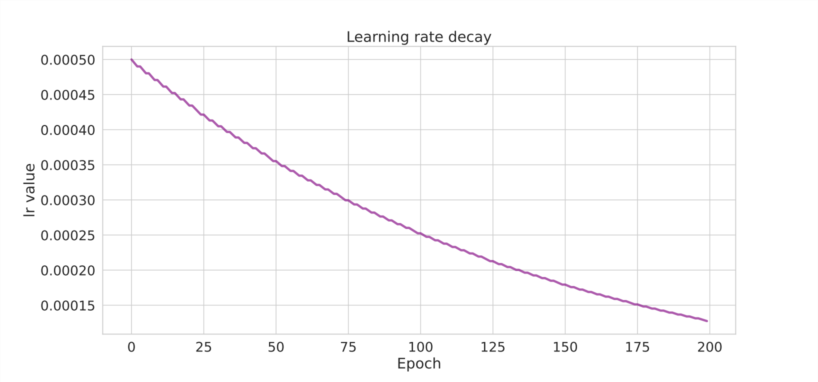


Figure 2: First 200 values of the learning rate used during the train in function of the epochs

Optimization - train step

The Algorithm's steps involve an **alternating training** of D and G. This procedure is done informing the optimizer about the correct label of the batches (0 real, 1 fake)[4]:

1. select a **real batch** of size n_{batch};
2. generate n_{batch} **latent points** (with both noise and label information inside) and use it for **generating fake images**;
3. use these **concatenated batches** for training only D.

```python
def train_step(self, data):
    #once selected the real batch, we prepare the fake one
    random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
    random_vector_labels = tf.concat([random_latent_vectors, one_hot_labels], axis=1)
    generated_images = self.generator(random_vector_labels)

    #let's concatenate images with their labels and create a big mixed batch
    comb_images = tf.concat([fake_image_and_labels, real_image_and_labels], axis=0)
    labels = tf.concat([tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0)

    #training only the discriminator
    with tf.GradientTape() as tape:
        predictions = self.discriminator(comb_images)
        d_loss = self.loss_fn(labels, predictions)
    grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
    self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_weights))
```

[4] This is the opposite of the above, but this choice is conventional and due to the fact that `apply_gradients` performs a descent and not an ascent

4. generate other n_{batch} latent points and labels;
5. let G use them for generating a new **fake** sample;
6. let's train G on the cGAN predictions (here we *freeze* the discriminator)

```
#then we create another set of inputs for the generator
#associating them with 0 labels, because we want to deceive D
    misleading_labels = tf.zeros((batch_size, 1))

    with tf.GradientTape() as tape:
        fake_images = self.generator(random_vector_labels)
        fake_image_and_labels = tf.concat([fake_images, image_one_hot_labels], -1)
        predictions = self.discriminator(fake_image_and_labels)
        g_loss = self.loss_fn(misleading_labels, predictions)
    grads = tape.gradient(g_loss, self.generator.trainable_weights)
    self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

    #updating the trackers state
    self.gen_loss_tracker.update_state(g_loss)
    self.disc_loss_tracker.update_state(d_loss)

    return {
        "g_loss": self.gen_loss_tracker.result(),
        "d_loss": self.disc_loss_tracker.result(),
    }
```

Results

Results

- I runned my code on `Google Colab Pro`, because such a problem requires a long lead time, even if well optimized. It provides NVIDIA Tesla K80 GPUs for a maximum single run of 12h;
- I also tried the `kaggle` GPUs, provided for a maximum of 37h of weekly usage. It provides NVIDIA Tesla P100 PCI GPUs for a maximum single run of 9h;

hyper-param	**choice**
n_{batch}	32
n_{epochs}	500
η_0	0.0005
γ_η	0.99
T_η	50
β_1	0.5
opt	Adam
J	Binary Crossentropy
z_{dim}	128

- the following result took $\sim 80'$ time.

Epoch 0

- Fire line 1, Water line 2, Grass line 3.

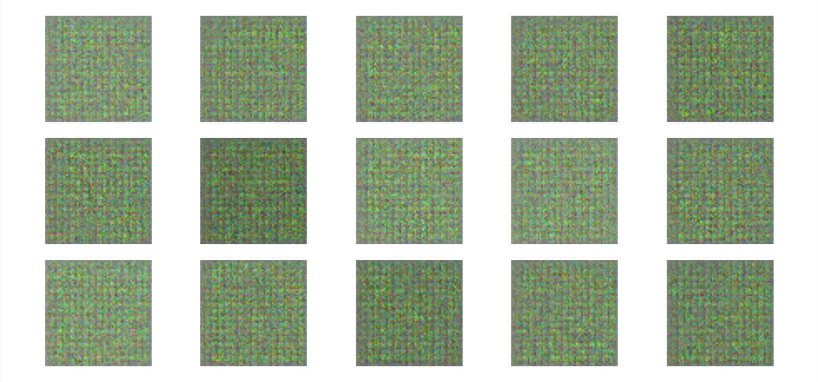


Figure 3: Images before the training

Epoch 10

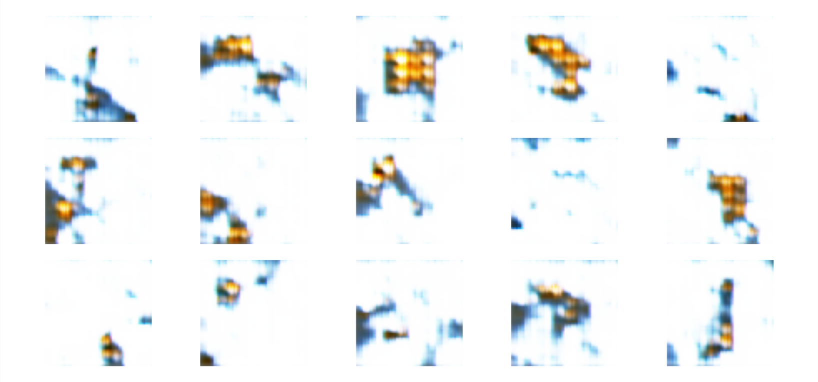


Figure 4: Images after 10 epochs of training

Epoch 100

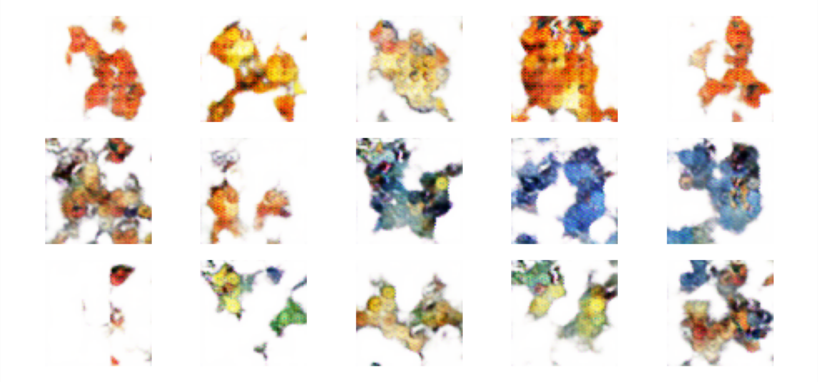


Figure 5: Images after 100 epochs of training

Epoch 200

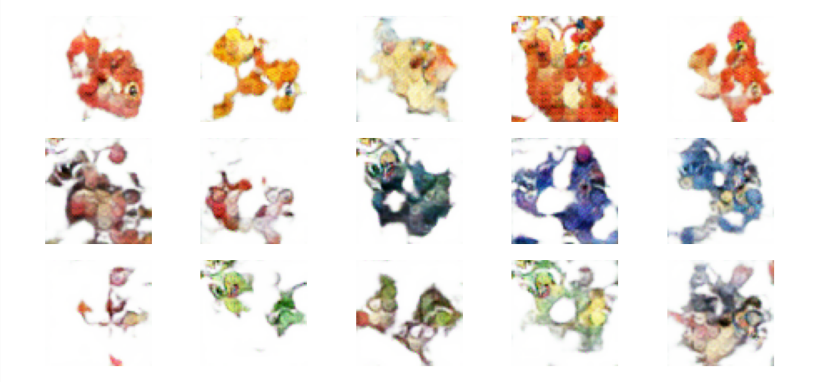


Figure 6: Images after 200 epochs of training

Epoch 500

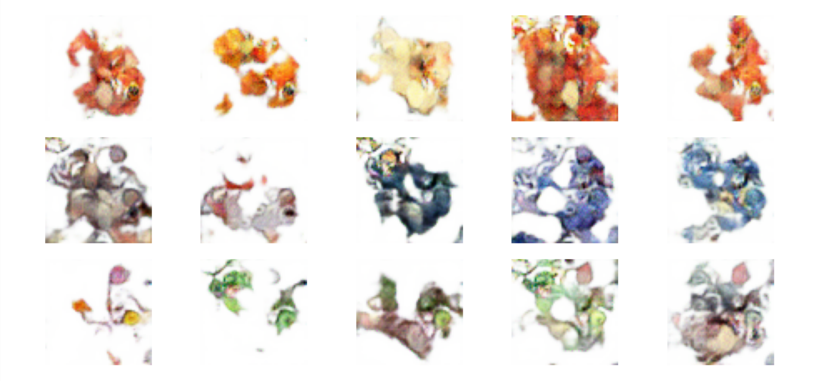


Figure 7: Images after 500 epochs of training

Movie results

- cGAN learns to represent the three classes

Thanks to [3] i've learnt that if we feed G with a series of *intermediate state* labels, we are able to generate *hybrid-classes' pokemon*!

```
#fire to grass label's transformation
#in this case we have 10 interpolations
tf.Tensor(
[[1.         0.         0.        ]
 [0.8888889  0.11111111 0.        ]
 [0.7777778  0.22222222 0.        ]
 [0.6666666  0.33333334 0.        ]
 [0.5555556  0.44444445 0.        ]
 [0.44444442 0.5555556  0.        ]
 [0.3333333  0.6666667  0.        ]
 [0.22222221 0.7777778  0.        ]
 [0.1111111  0.8888889  0.        ]
 [0.         1.         0.        ]], shape=(10, 3), dtype=float32)
```

- Fire to Grass transformation
- Fire to Water transformation

A difficult training

- To train GANs, and cGANs too, is a difficult task
- Pokemon are very heterogeneous, even those belonging to the same type;
- during my job I performed several attempts desiring to achieve better and better results;
- I report the history of the loss functions evaluated during the last training:

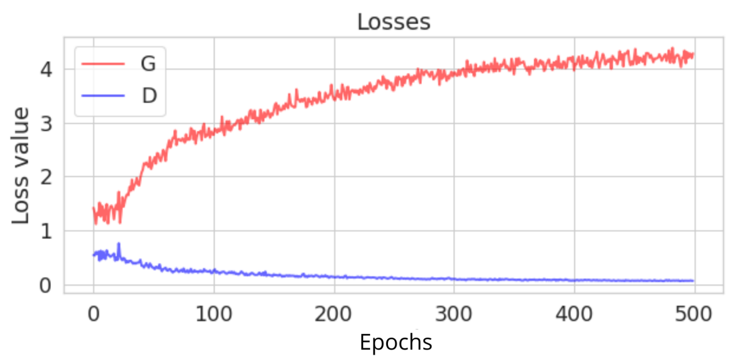


Figure 8: Value of loss function as a function of epochs

How to do better?

Data:

- searching for new data, maybe involving more than three classes and increasing the poor amount of images of my DS;
- building a dataset more balanced: in this case I had 351 fire pokes, 288 grass pokes and 425 water pokes.

Model:

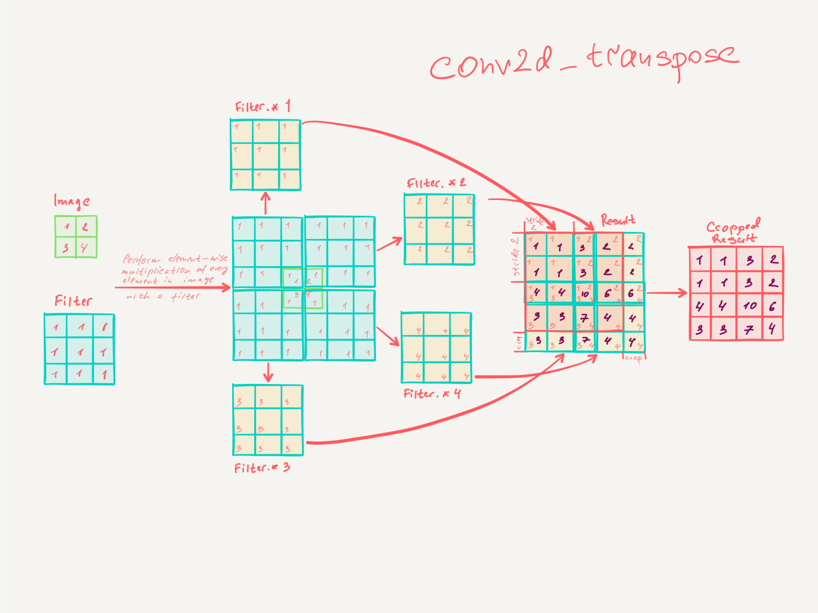
- implement an hyper-optimization method, instead of doing that manually;
- test more complex models, in order to keep under control the training of G and D: for example, it's possible to use more than one encoder and more than one decoder;
- exploit verified models, such as the NVIDIA Research Projects StyleGAN, using the power of the transfer learning;

Training:

- writing a more balanced train step method, for example training k times D and one time G, as said in [1], or training G with a batch of the same size of the one used with D.

Bibliography

- NIan J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, *Generative Adversarial Networks* (2014), arXiv:1406.2661.
- Mehdi Mirza, Simon Osindero, *Conditional Generative Adversarial Nets* (2014), arXiv:1411.1784.
- Sayak Paul, *conditional GAN* (2021), official Keras' tutorial with MNIST.
- Mikolaj Kolman, Initial state of the pokemon's dataset


conv2d_transpose
Image
Filter
Perform element-wise multiplication of every element in image with a filter
Filter. * 1
Filter. * 2
Filter. * 3
Filter. * 4
Result
Cropped Result