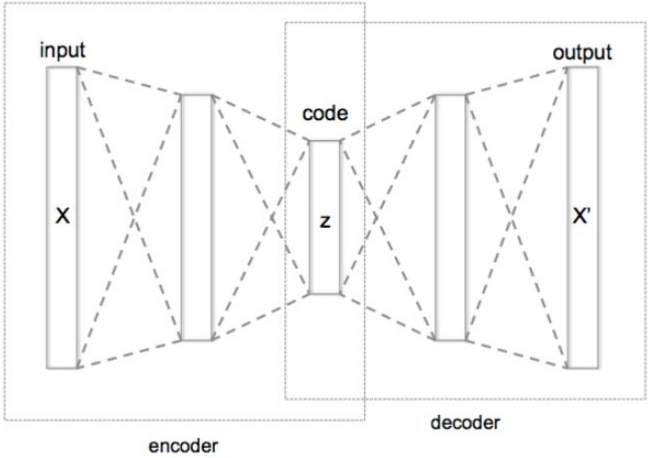
Autoencoders y GANs

Aprendizaje Auto Supervisado / Manifold Learning/
Adversarial Training

Introducción



- Copian la entrada a la salida
- Encoder: h=f(x)
- Decoder: r=g(h)
- Encontrar un autoencoder tal que g(f(x))=x para todo x no es interesante en cuanto a la tarea que realiza, pero resolver esta tarea suele brindar subproductos interesantes si se aplican algunas restricciones adicionales.
  - Ej: Si la dimensionalidad de h es menor que la de la entrada/salida y la reconstrucción es válida, entonces el autoencoder encontró una representación más eficiente que la original.
- Para poder entrenar el modelo hay que definir una medida de error entre la entrada y la salida. Dicha medida debe ser derivable para poder entrenar el modelo.

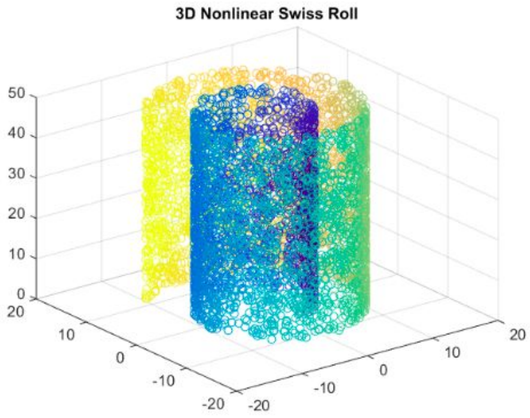
Autoencoders: Aplicaciones

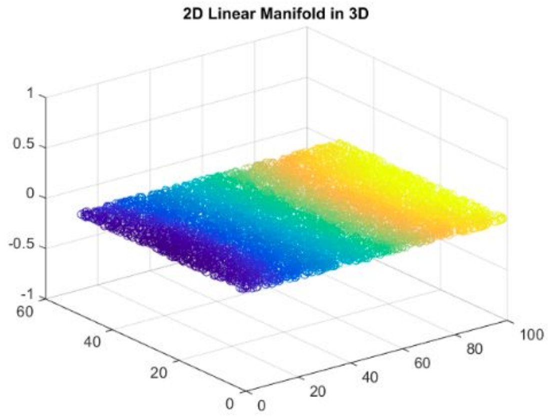
- Reducción de la dimensionalidad (Aprendizaje Semi-Supervisado)
- Detección de anomalías
- Denoising

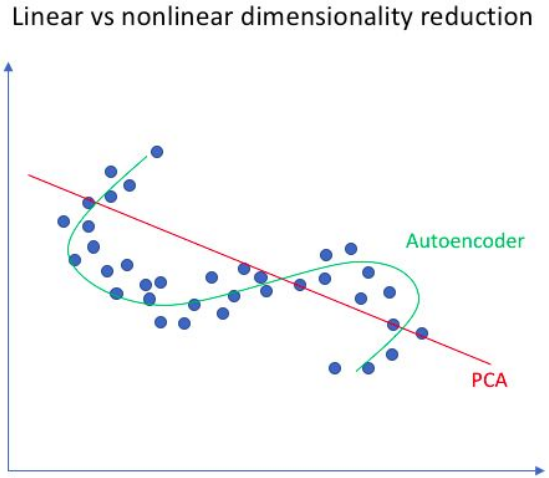
Undercomplete Autoencoder

- La idea es que la dimensión de **h** sea menor que la de **x**.
- Aprenden una representación incompleta de los datos de entrada, guiados por una función de costo.
- El objetivo es encontrar una representación que minimice L(x,g(f(x)).
  - Ej:
    - Lineales + MSE: PCA
    - Alineales: Redes Neuronales
- Las varianzas del encoder y del decoder también tienen que ser acotadas ya que en idealmente podrían mapear todo el espacio de entrada en una sola dimensión y de ahí volver a obtener el dato original. Esto no pasa en la práctica pero es un ejercicio interesante para pensar.

Undercomplete Autoencoder (Interpretación)





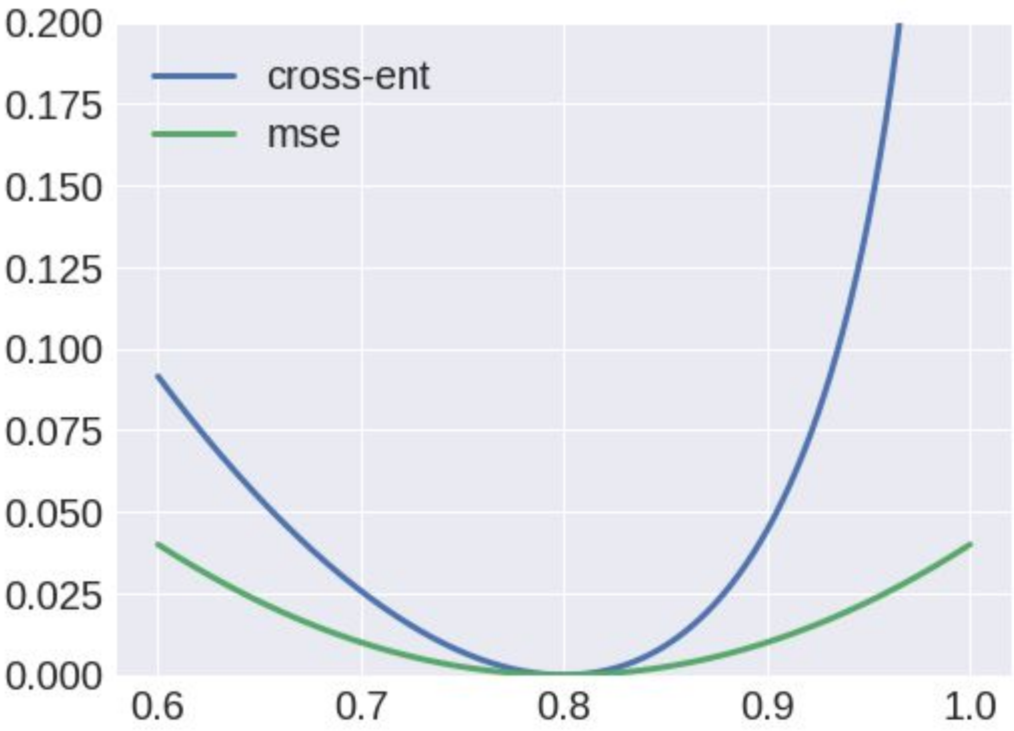


[Fuente]

Funciones de costo (Discusión)

- MSE
- MAE
- Binary Crossentropy
- Categorical Crossentropy
- Loss para imágenes?

[Perceptual Loss]



Perceptual Loss

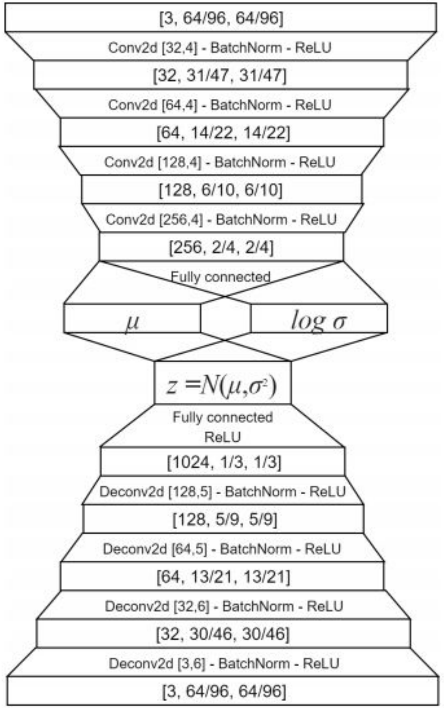


Fig. 4. The convolutional variational autoencoder used in this work.

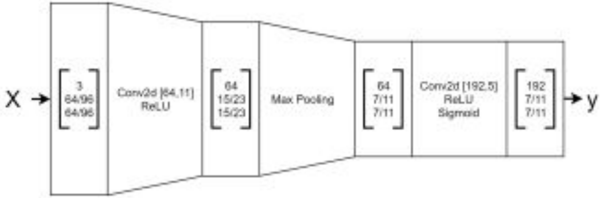


Fig. 3. The parts of a pretrained AlexNet that were used for calculating and backpropagating the perceptual loss.

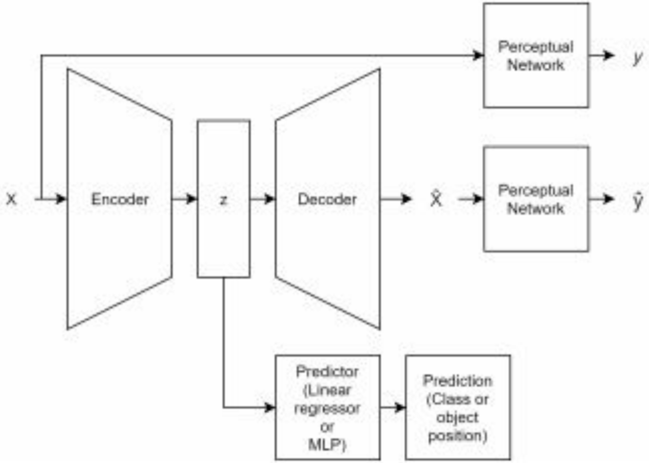


Fig. 5. The system used in this work including both the autoencoding and prediction pathways.

Loss Function

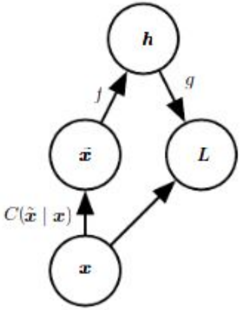
$$E = \sum_{k=1}^{n} f(X_k, a(X)_k)$$

$$E = \sum_{k=1}^{m} f(p(X)_k, p(a(X))_k)$$

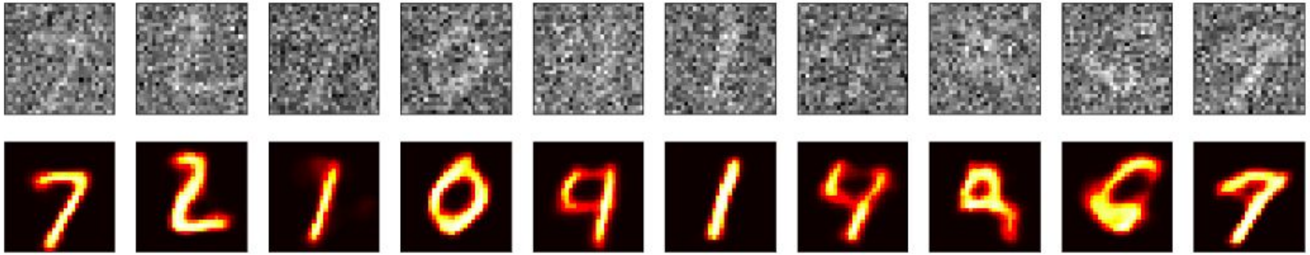
Denoising Autoencoders

- Un DA en vez de minimizar el error de reconstrucción de la salida con respecto a la entrada, minimiza una los dada por:

$$L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}})))$$

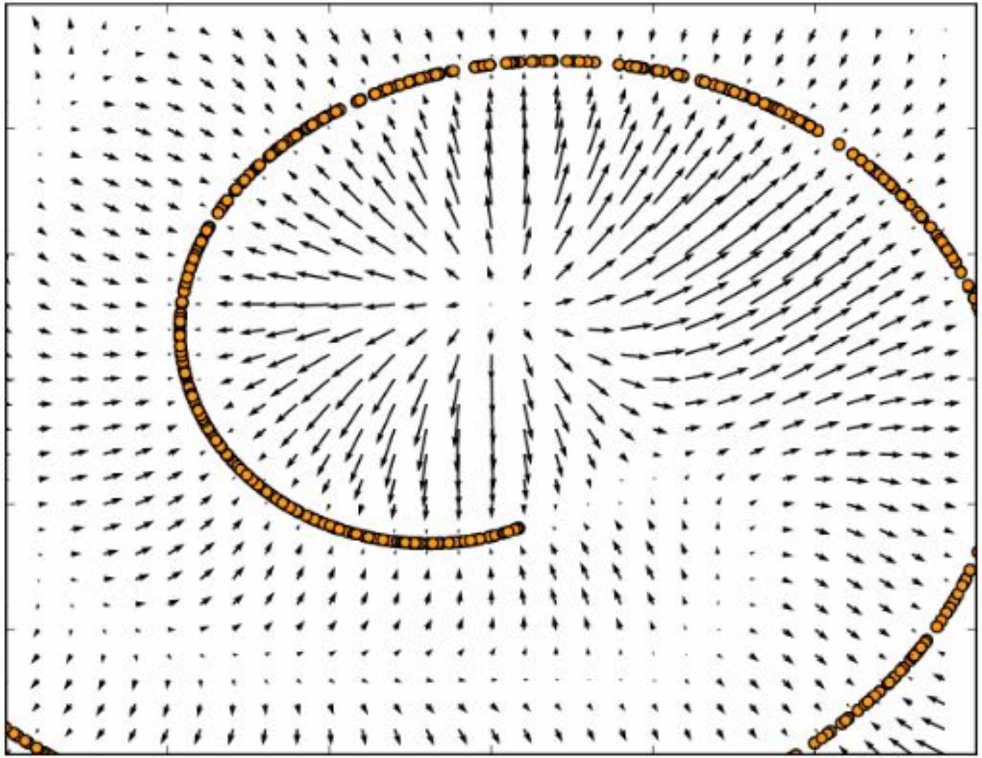


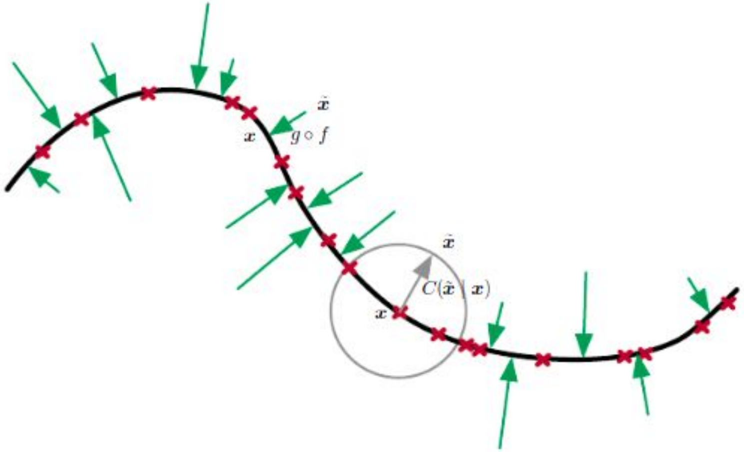
- En este caso la entrada es una versión modificada con ruido del dato original.
- En este caso se pueden utilizar overcompletes autoencoders.



[Github]

Denoising Autoencoders (Interpretación)



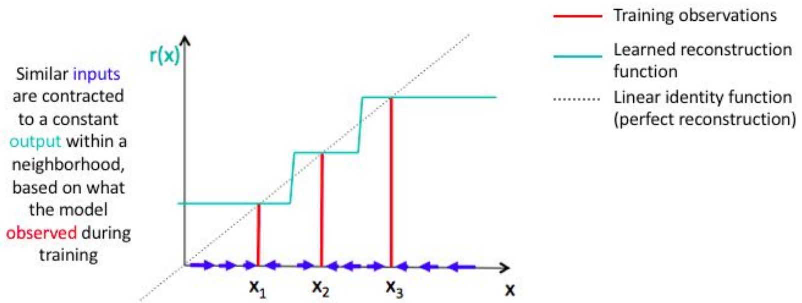


Contractive Autoencoders

- Penalizan altos valores de derivadas de las activaciones de h, con respecto a las entradas.
- Equivale a hacer denoising pero con apartamientos infinitesimales.

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\delta a_1^{(h)}(x)}{\delta x_1} & \cdots & \frac{\delta a_1^{(h)}(x)}{\delta x_m} \\ \vdots & \ddots & \vdots \\ \frac{\delta a_n^{(h)}(x)}{\delta x_1} & \cdots & \frac{\delta a_n^{(h)}(x)}{\delta x_m} \end{bmatrix}$$



$$\mathcal{L}(x, \hat{x}) + \lambda \sum_i \left\|\nabla_x a_i^{(h)}(x)\right\|^2$$

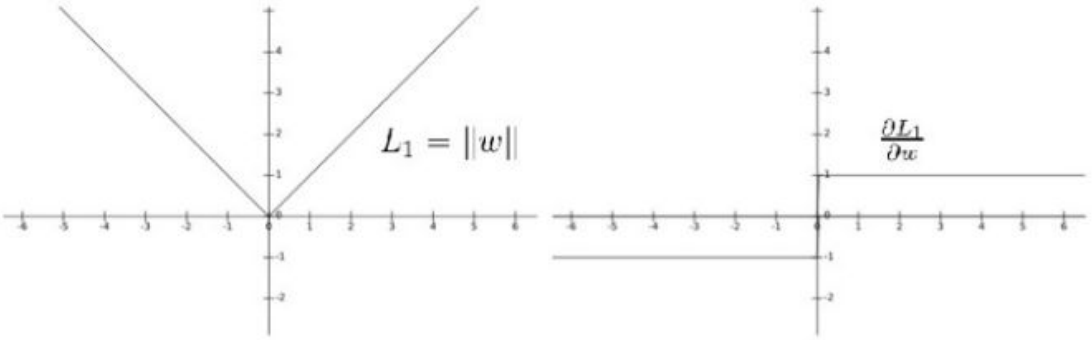
Sparse Autoencoders

- Así como en otro contexto regularizábamos con respecto a los pesos, podemos regularizar con respecto a las activaciones.
- Al agregar un término de regularización L1 con respecto a las activaciones de **h** podemos asegurarnos una representación sparsa (rala) en dimensionalidad reducida.
- En algunos casos las representaciones sparsas representan mejor un problema.
- En el caso de aprendizaje semi-supervisado se ve una mejora en la performance de distintas tareas utilizando representaciones sparsas. Per ejemplo: clasificación de imágenes.

$$Obj = L(x, \hat{x}) + regularization + \lambda \sum_i |a_i^{(h)}|$$

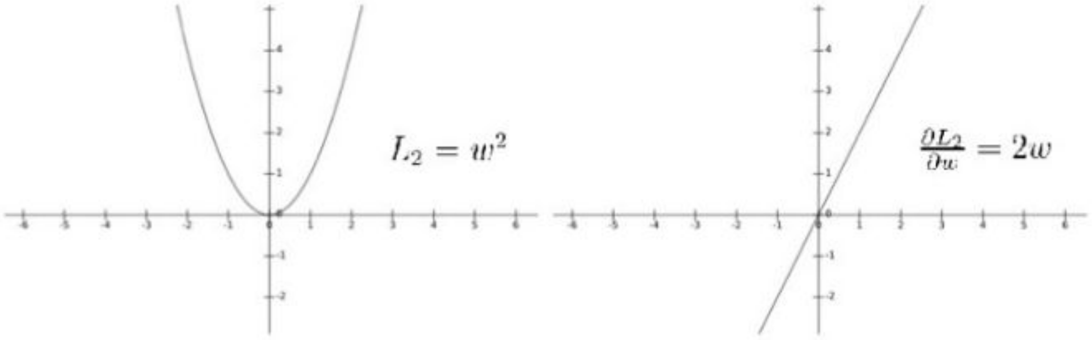
Sparse Autoencoders

- [Notas de clase de Andrew Ng]()



L1 regularization and its derivative

```
...
model.add(Conv2D(32, (3,3), activity_regularizer=l1(0.001)))
...
```

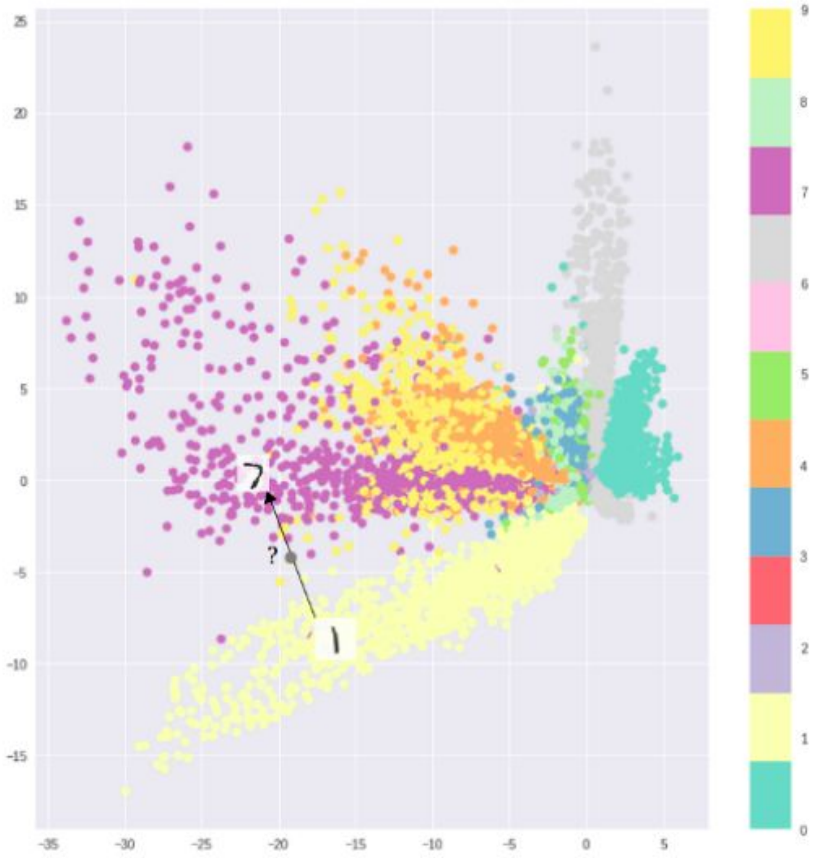


L2 regularization and its derivative

Variational Autoencoders (Link)

- Los autoencoders vistos hasta ahora resuelven el problema de representación y denoising.
- ¿Podría modelarse la distribución del espacio latente y a partir de esto generar datos nuevos?
  - No hay garantías de la continuidad ni la interpolabilidad del espacio latente.
- El problema son las discontinuidades entre los clusters.
- Durante el entrenamiento no hay datos que se generen desde esas regiones del espacio latente. Por lo tanto no se puede estimar qué va a hacer el decoder.

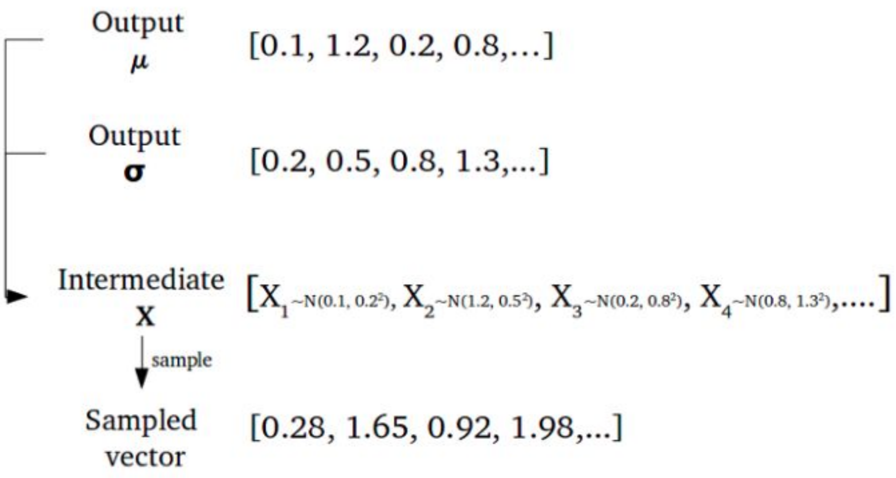
Los VAE tienen por diseño un espacio latente continuo (regularizado) y por lo tanto permiten la interpolación y sampleo para la generación de datos. Link a Keras.js



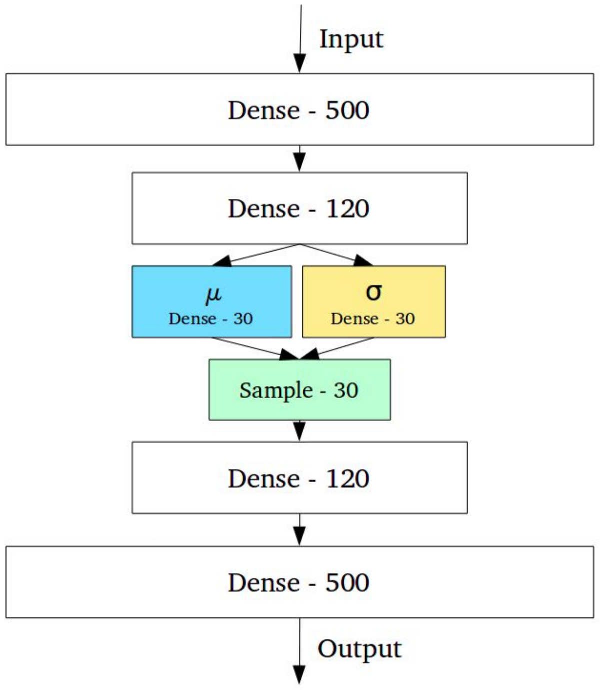
Optimizing purely for reconstruction loss

Variational Autoencoders

- Los VAE no dan como subproducto un vector en el espacio latente, sino que dan una distribución.
- De esa distribución se muestrea un vector y éste ingresa al autoencoder para generar el dato.



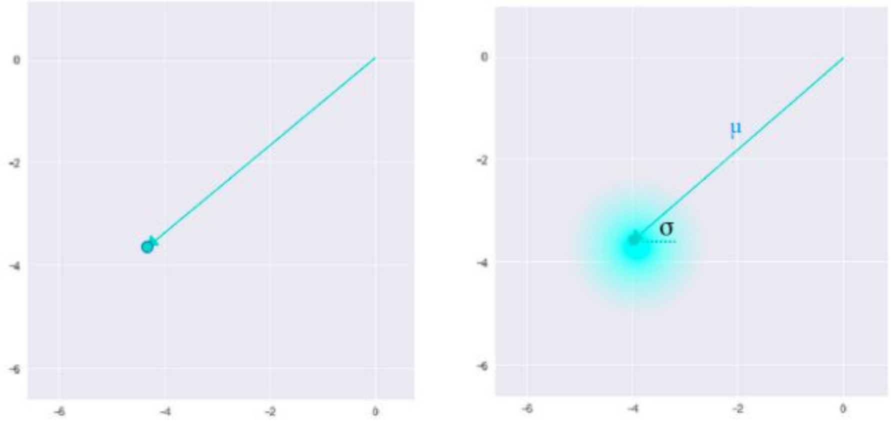
Stochastically generating encoding vectors



Variational Autoencoder

Variational Autoencoders

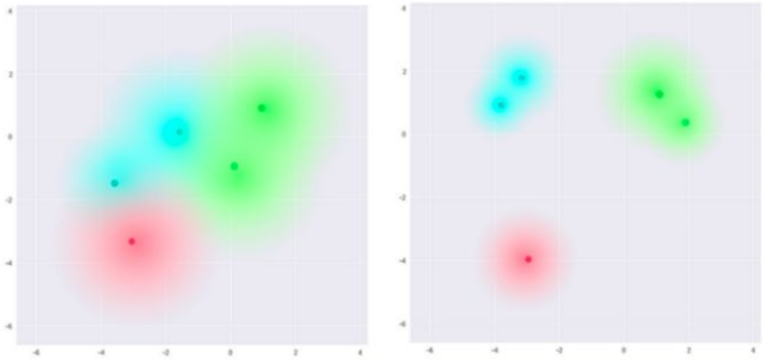
- Esto implica que si bien para la misma entrada el encoder nos va a dar a la salida la misma distribución, el decoder va a recibir dos vectores distintos producto del muestreo.



Standard Autoencoder
(direct encoding coordinates)

Variational Autoencoder
(μ and σ initialize a probability distribution)

- Como el dato debe ser reconstruído correctamente no desde un punto en particular sino que ahora debe ser reconstruído desde un zona del espacio de embeddings, las representaciones van a tener cierto grado de continuidad local.
- También es deseable que haya continuidad entre puntos del espacio de embeddings correspondientes a clases distintas.
- Sin embargo con este tipo de modelo, las muestras tienden a agruparse de la siguiente forma:

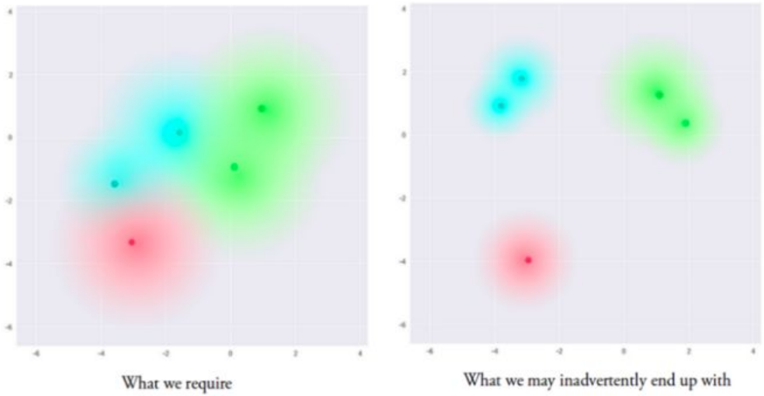


What we require

What we may inadvertently end up with

Variational Autoencoders

- Esto se debe a que no hay restricciones con respecto a los valores que pueden tomar mu y sigma.
- Datos muy distintos pueden ir a parar a mu’s muy distintos y eso hace que no tenga control sobre la transición entre clases, o modos de la distribución de entrada.
- Por este motivo, condiciones adicionales deben ser impuestas a los valores de mu y sigma.
- Por forzar a que los puntos estén cerca, podemos pedir que las distribuciones a la salida del encoder se parezcan lo más posible a una distribución conocida. Por ejemplo: Normal (0,1)



Variational Autoencoders

- Una forma de obligar a que los puntos estén aglomerados en el centro del espacio latente es utilizando la divergencia KL entre las distribuciones obtenidas y una distribución de referencia, como la Normal (0,1).
- Para medir distancias entre distribuciones podemos utilizar la DKL entre los valores a la salida del encoder y la Normal (0,1):

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

KL Divergence for Continuous Probability Distributions — Wikipedia

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

Variational Autoencoders (DKL entre gaussianas. Link)

$$\int \left[\log(p(x)) - log(q(x))\right] p(x)dx$$

$$= \int \left[-\tfrac{1}{2}\log(2\pi) - \log(\sigma_1) - \tfrac{1}{2}\left(\tfrac{x-\mu_1}{\sigma_1}\right)^2 + \tfrac{1}{2}\log(2\pi) + \log(\sigma_2) + \tfrac{1}{2}\left(\tfrac{x-\mu_2}{\sigma_2}\right)^2\right]$$

$$\times \frac{1}{\sqrt{2\pi}\sigma_1}\exp\left[-\tfrac{1}{2}\left(\tfrac{x-\mu_1}{\sigma_1}\right)^2\right]dx$$

$$= \int \left\{\log\left(\tfrac{\sigma_2}{\sigma_1}\right) + \tfrac{1}{2}\left[\left(\tfrac{x-\mu_2}{\sigma_2}\right)^2 - \left(\tfrac{x-\mu_1}{\sigma_1}\right)^2\right]\right\} \times \frac{1}{\sqrt{2\pi}\sigma_1}\exp\left[-\tfrac{1}{2}\left(\tfrac{x-\mu_1}{\sigma_1}\right)^2\right]dx$$

$$= E_1\left\{\log\left(\tfrac{\sigma_2}{\sigma_1}\right) + \tfrac{1}{2}\left[\left(\tfrac{x-\mu_2}{\sigma_2}\right)^2 - \left(\tfrac{x-\mu_1}{\sigma_1}\right)^2\right]\right\}$$

$$= \log\left(\tfrac{\sigma_2}{\sigma_1}\right) + \tfrac{1}{2\sigma_2^2}E_1\left\{(X-\mu_2)^2\right\} - \tfrac{1}{2\sigma_1^2}E_1\left\{(X-\mu_1)^2\right\}$$

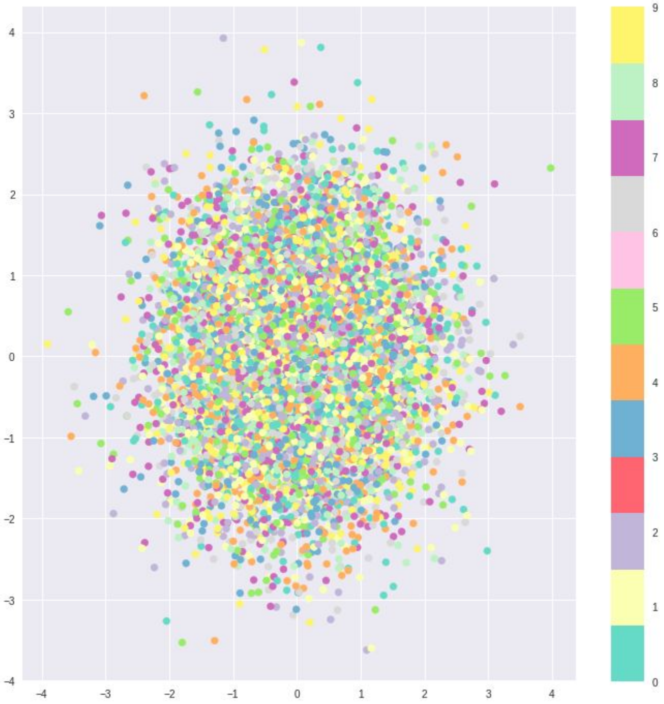
$$= \log\left(\tfrac{\sigma_2}{\sigma_1}\right) + \tfrac{1}{2\sigma_2^2}E_1\left\{(X-\mu_2)^2\right\} - \tfrac{1}{2}$$

(Now note that
$(X-\mu_2)^2 = (X-\mu_1+\mu_1-\mu_2)^2 = (X-\mu_1)^2 + 2(X-\mu_1)(\mu_1-\mu_2) + (\mu_1-\mu_2)^2)$

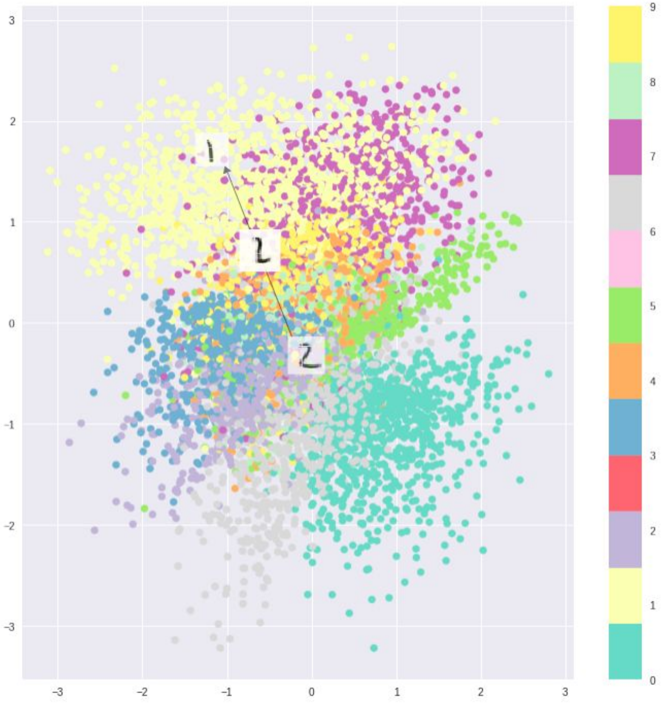
$$= \log\left(\tfrac{\sigma_2}{\sigma_1}\right) + \tfrac{1}{2\sigma_2^2}\left[E_1\left\{(X-\mu_1)^2\right\} + 2(\mu_1-\mu_2)E_1\left\{X-\mu_1\right\} + (\mu_1-\mu_2)^2\right] - \tfrac{1}{2}$$

$$= \log\left(\tfrac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \tfrac{1}{2}$$

Variational Autoencoders



Optimizing using pure KL divergence loss



Optimizing using both reconstruction loss and KL divergence loss

Variational Autoencoders: Pytorch - Lightning

```
class VAE(pl.LightningModule):
    def __init__(self,alpha = 1):
        #Autoencoder only requires 1 dimensional argument since input and output-size is the same

        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(784,196),nn.ReLU(),nn.BatchNorm1d(196,momentum = 0.7),
                                     nn.Linear(196,49),nn.ReLU(),nn.BatchNorm1d(49,momentum = 0.7),
                                     nn.Linear(49,28),nn.LeakyReLU())
        self.hidden2mu = nn.Linear(28,28)
        self.hidden2log_var = nn.Linear(28,28)
        self.alpha = alpha
        self.decoder = nn.Sequential(nn.Linear(28,49),nn.ReLU(),
                                     nn.Linear(49,196),nn.ReLU(),
                                     nn.Linear(196,784),nn.Tanh())
        self.data_transform = transforms.Compose([
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean=(0.5,),std=(0.5,))])

    def encode(self,x):
        hidden = self.encoder(x)
        mu = self.hidden2mu(hidden)
        log_var = self.hidden2log_var(hidden)
        return mu,log_var

    def reparametrize(self,mu,log_var):
        #Reparametrization Trick to allow gradients to backpropagate from the
        #stochastic part of the model
        sigma = torch.exp(0.5*log_var)
        z = torch.randn(size = (mu.size(0),mu.size(1)))
        z= z.type_as(mu)
        return mu + sigma*z

    def decode(self,x):
        x  = self.decoder(x)
        return x
```

Variational Autoencoders: Implementación

```python
def configure_optimizers(self):
    return Adam(self.parameters(),lr = 1e-3)

def forward(self,x):
    batch_size = x.size(0)
    x = x.view(batch_size,-1)
    mu,log_var = self.encode(x)
    hidden = self.reparametrize(mu,log_var)
    return self.decoder(hidden)

# Functions for dataloading
def train_dataloader(self):
    mnist_train  = MNIST('data/',download = True,train = True,transform=self.data_transform)
    return DataLoader(mnist_train,batch_size=64, num_workers=12)

def val_dataloader(self):
    mnist_val  = MNIST('data/',download = True,train = False,transform=self.data_transform)
    return DataLoader(mnist_val,batch_size=64,num_workers=12)

def scale_image(self,img):
    out = (img + 1) / 2
    return out
```

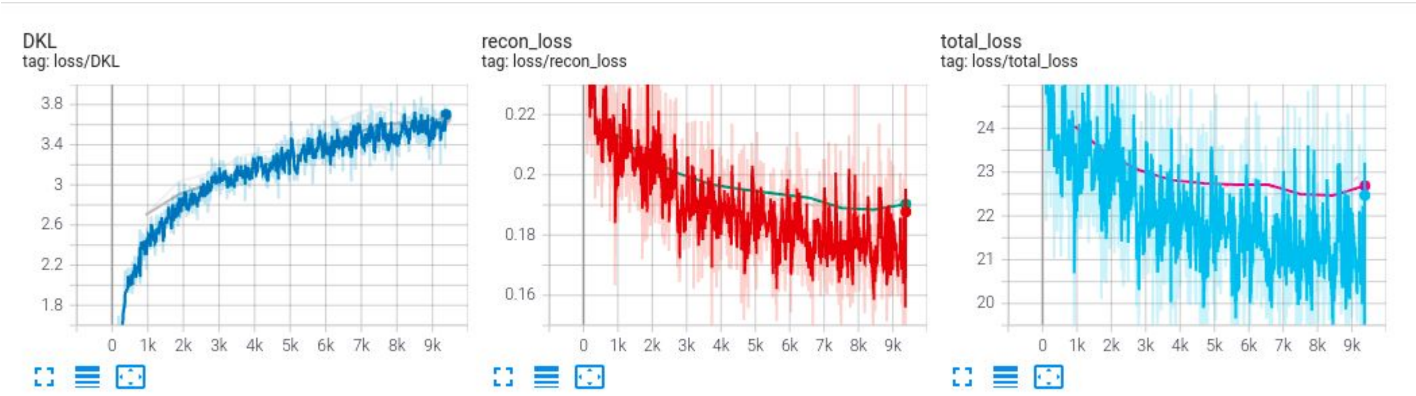
Variational Autoencoders: Implementación

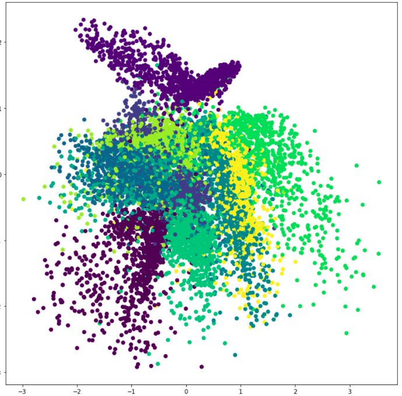
```python
def training_step(self,batch,batch_idx):
    x,_ = batch
    batch_size = x.size(0)
    x = x.view(batch_size,-1)
    mu,log_var = self.encode(x)

    kl_loss =  (-0.5*(1+log_var - mu**2- torch.exp(log_var)).sum(dim = 1)).mean(dim =0)
    hidden = self.reparametrize(mu,log_var)
    x_out = self.decode(hidden)
    recon_loss_criterion = nn.MSELoss()
    #print(kl_loss.item(),recon_loss.item())
    recon_loss_criterion = nn.MSELoss()
    recon_loss = recon_loss_criterion(x,x_out)
    loss = recon_loss*self.alpha + kl_loss
    self.logger.experiment.add_scalars('loss/DKL',{'train':kl_loss},global_step=self.global_step)
    self.logger.experiment.add_scalars('loss/recon_loss',{'train':recon_loss},global_step=self.global_step)
    self.logger.experiment.add_scalars('loss/total_loss',{'train':loss},global_step=self.global_step)
    return loss
```

Variational Autoencoders: Implementación

```
def validation_step(self,batch,batch_idx):
    x,_ = batch
    batch_size = x.size(0)
    x = x.view(batch_size,-1)
    mu,log_var = self.encode(x)

    kl_loss =  (-0.5*(1+log_var - mu**2- torch.exp(log_var)).sum(dim = 1)).mean(dim =0)
    hidden = self.reparametrize(mu,log_var)
    x_out = self.decode(hidden)

    recon_loss_criterion = nn.MSELoss()
    recon_loss = recon_loss_criterion(x,x_out)
    #print(kl_loss.item(),recon_loss.item())
    loss = recon_loss*self.alpha + kl_loss
    return x_out,{"loss": loss, "rec": recon_loss, "kl_loss": kl_loss, "batch": batch}

def validation_epoch_end(self,outputs):
    avg_loss = torch.stack([x[1]['loss'] for x in outputs]).mean()
    avg_rec = torch.stack([x[1]['rec'] for x in outputs]).mean()
    avg_dist = torch.stack([x[1]['kl_loss'] for x in outputs]).mean()
    self.logger.experiment.add_scalars(
        "loss/total_loss",
        {"val": avg_loss},
        global_step=self.global_step
    )
    self.logger.experiment.add_scalars(
        "loss/recon_loss",
        {"val": avg_rec},
        global_step=self.global_step
    )
    self.logger.experiment.add_scalars(
        "loss/DKL",
        {"val": avg_dist},
        global_step=self.global_step
    )
    if not os.path.exists('vae_images'):
        os.makedirs('vae_images')
    choice = random.choice(outputs)
    output_sample = choice[0]
    output_sample = output_sample.reshape(-1, 1, 28, 28)
    output_sample = self.scale_image(output_sample)
    save_image(output_sample, f"vae_images/epoch_{self.current_epoch+1}.png")
```

Variational Autoencoder: Entrenamiento

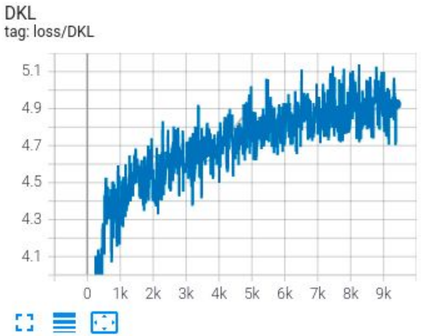


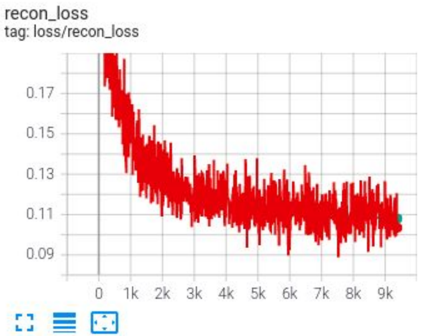


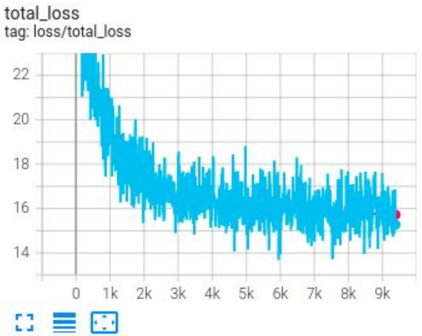
Epoch 1 — Reconstrucción — Epoch 10

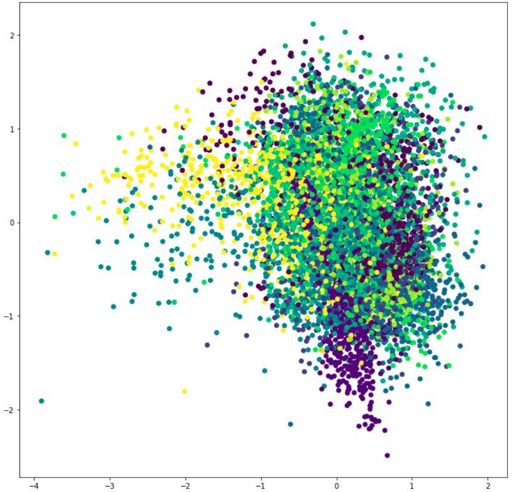
Epoch 1 — Generación — Epoch 10

Variational Autoencoder: Entrenamiento (hidden=2)

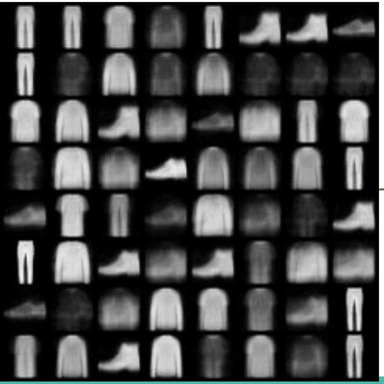




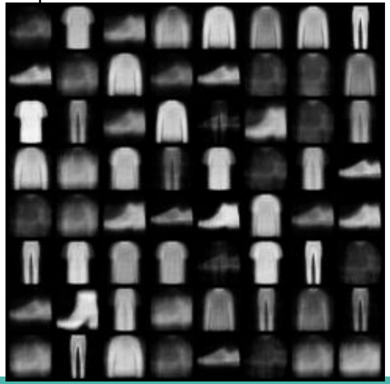


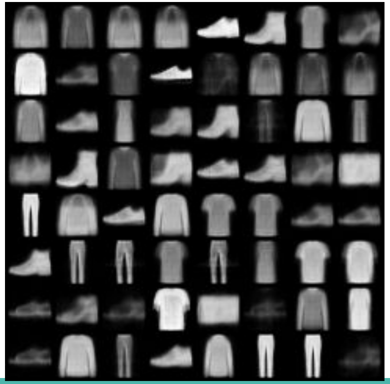


Epoch 1 Reconstrucción Epoch 10

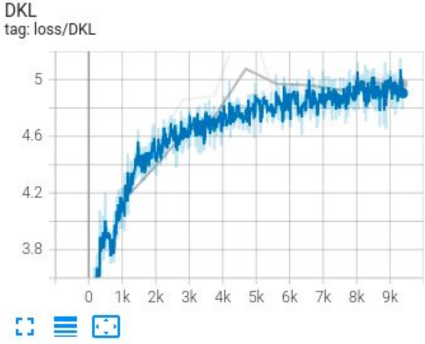


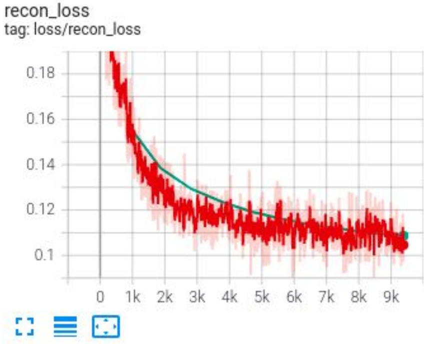
Epoch 1 Generación Epoch 10

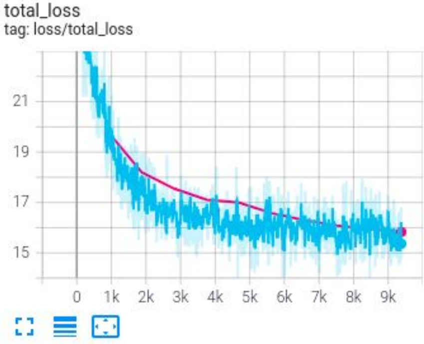


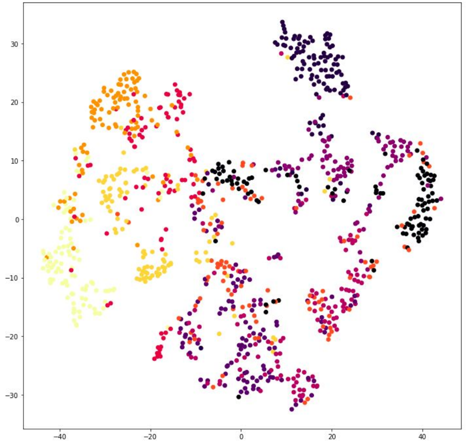


Variational Autoencoder: Entrenamiento (hidden=28)



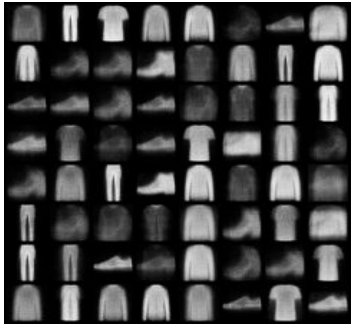






Reconstrucción

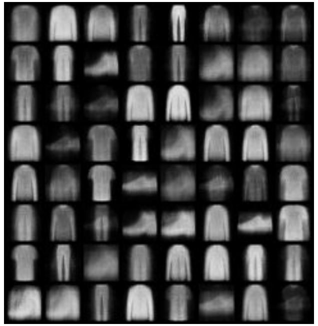
Epoch 1



Epoch 10

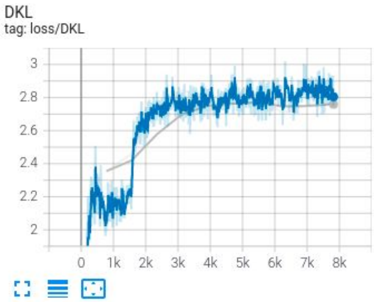
Generación

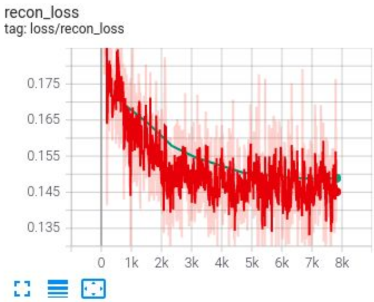
Epoch 1

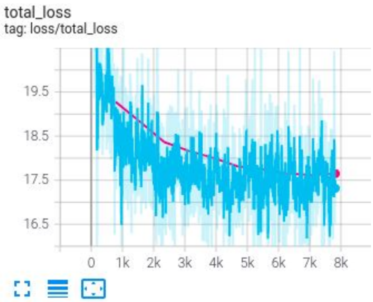


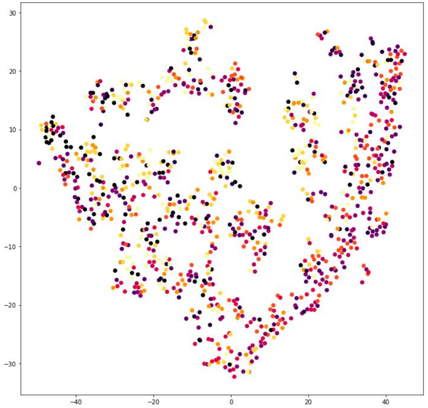
Epoch 10

VAE: Entrenamiento (hidden=28). CIFAR10

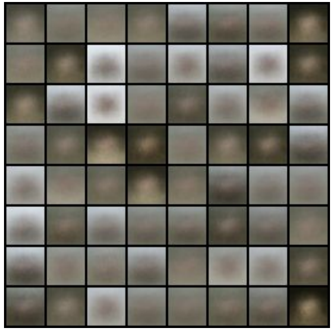


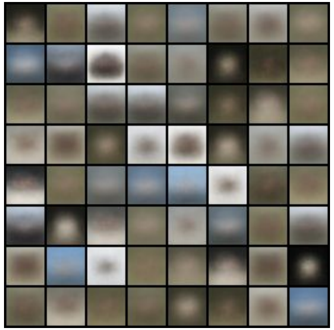




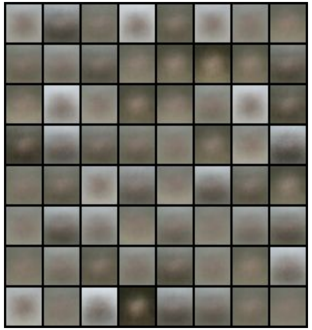


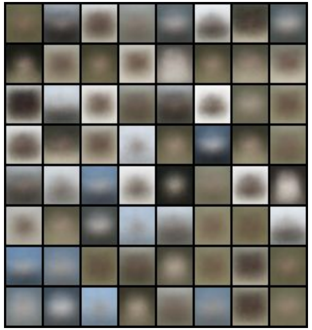
Epoch 1 Reconstrucción Epoch 10





Epoch 1 Generación Epoch 10



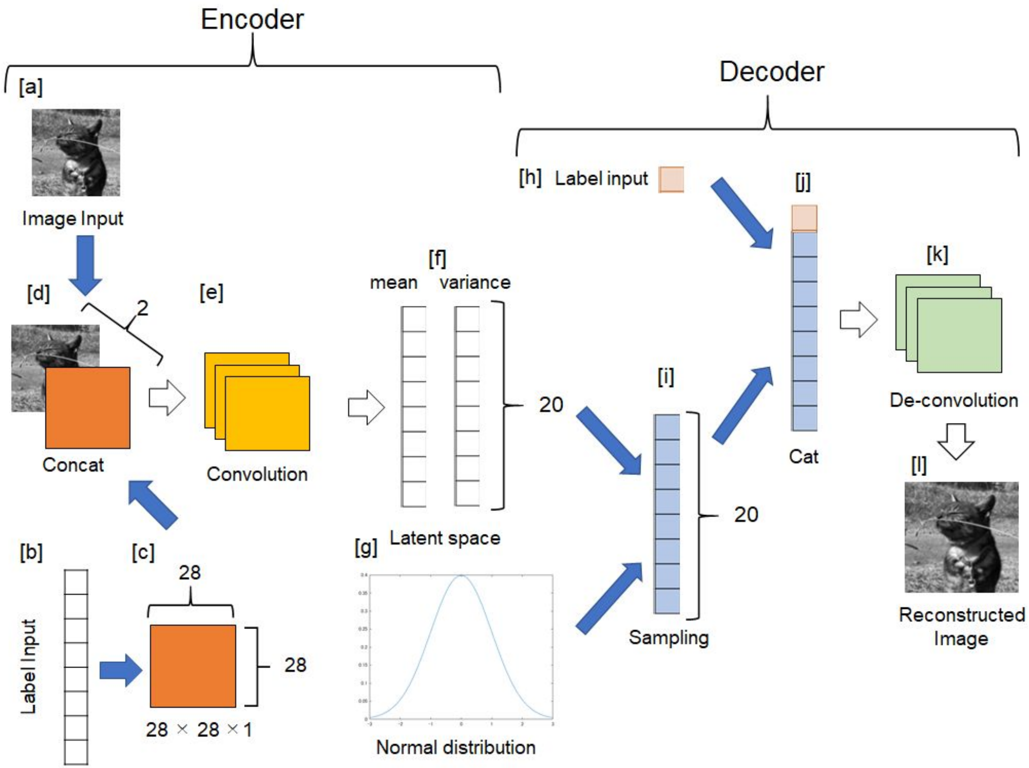


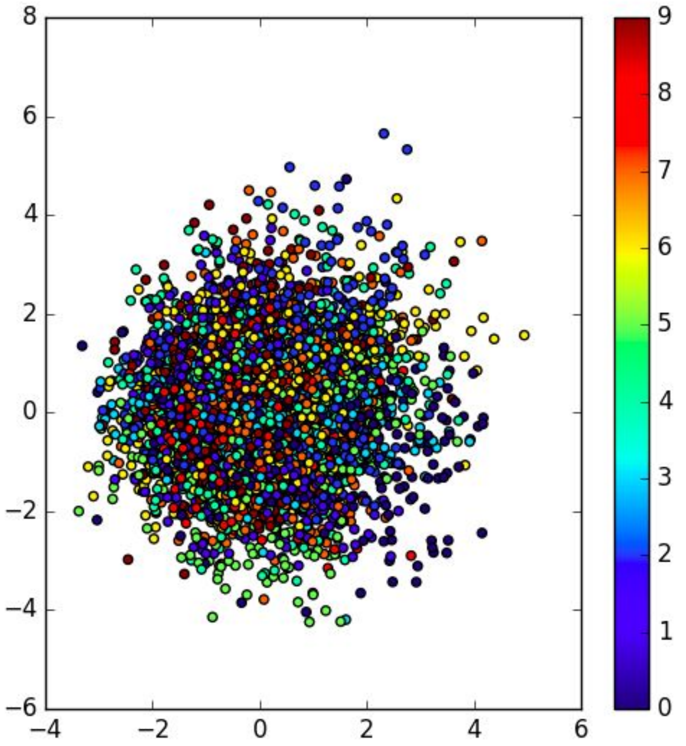
Soluciones a CIFAR10

- CVAEs
- DC-VAEs
- DC-CVAEs
- GANs
- DC-GANs
- DC-CGANs

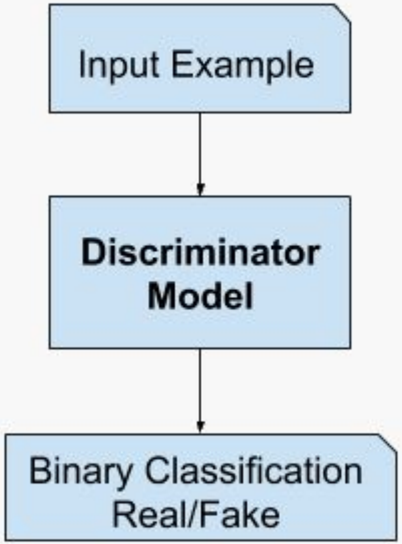
VAE-DCGANs (Blog de VAE)

CVAEs (Link, Link)

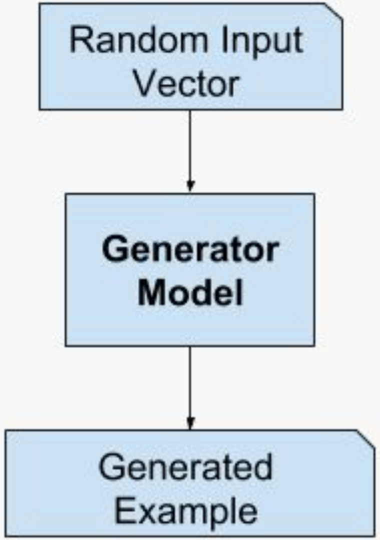




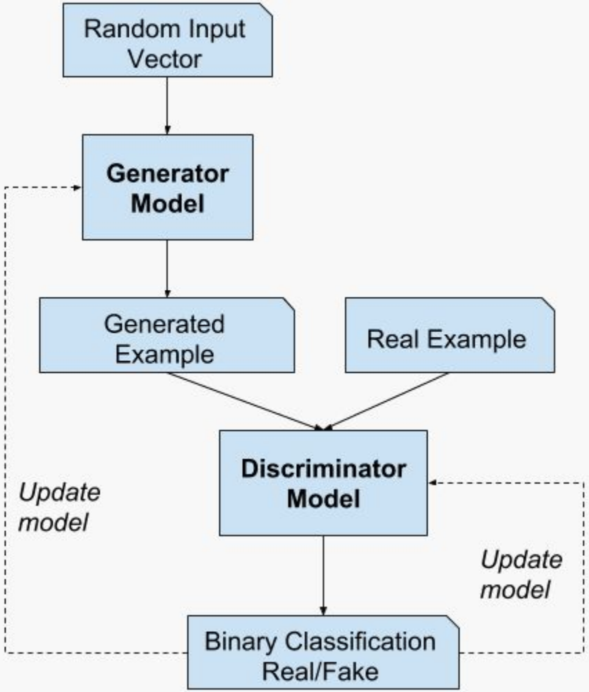
GANs (Generative Adversarial Networks) (Link)



Example of the GAN Discriminator Model

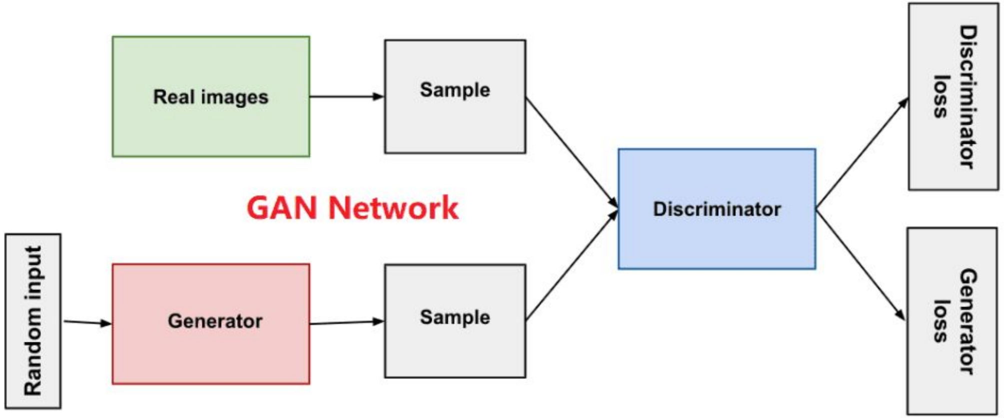


Example of the GAN Generator Model



Example of the Generative Adversarial Network Model Architecture

SimpleGAN: Implementación




```python
class Generator(nn.Module):
    def __init__(self, latent_dim, dim):
        super().__init__()
        self.dim = 2

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat))
            layers.append(nn.ReLU(inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(latent_dim, 128, normalize=False),
            *block(128, 256, normalize=True),
            *block(256, 512, normalize=True),
            *block(512, 1024, normalize=True),
            nn.Linear(1024, self.dim),
        )

    def forward(self, z):
        data = self.model(z)
        return data
```

```python
class Discriminator(nn.Module):
    def __init__(self, dim):
        super().__init__()

        self.model = nn.Sequential(
            nn.Linear(dim, 16),
            nn.ReLU(inplace=True),
            nn.Linear(16, 32),
            nn.ReLU(inplace=True),
            nn.Linear(32, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, 1),
            nn.Sigmoid(),
        )

    def forward(self, data):
        validity = self.model(data)
        return validity
```

Simple GAN: Implementación

```python
class GAN(pl.LightningModule):

    def __init__(
        self,
        dim,
        latent_dim: int = 100,
        lr: float = 0.0002,
        b1: float = 0.5,
        b2: float = 0.9,
        **kwargs
    ):
        super().__init__()
        self.save_hyperparameters()

        # networks
        self.generator = Generator(latent_dim=self.hparams.latent_dim, dim=dim)
        self.discriminator = Discriminator(dim=dim)

        self.validation_z = torch.randn(8, self.hparams.latent_dim)

    def forward(self, z):
        return self.generator(z)

    def adversarial_loss(self, y_hat, y):
        return F.binary_cross_entropy(y_hat, y)
```

```python
def configure_optimizers(self):
    lr = self.hparams.lr
    b1 = self.hparams.b1
    b2 = self.hparams.b2

    opt_g = torch.optim.Adam(self.generator.parameters(), lr=lr, betas=(self.hparams.b1, self.hparams.b2))
    opt_d = torch.optim.Adam(self.discriminator.parameters(), lr=lr, betas=(self.hparams.b1, self.hparams.b2))
    return [opt_g, opt_d], []

def on_epoch_end(self):
    z = self.validation_z.type_as(self.generator.model[0].weight)
```

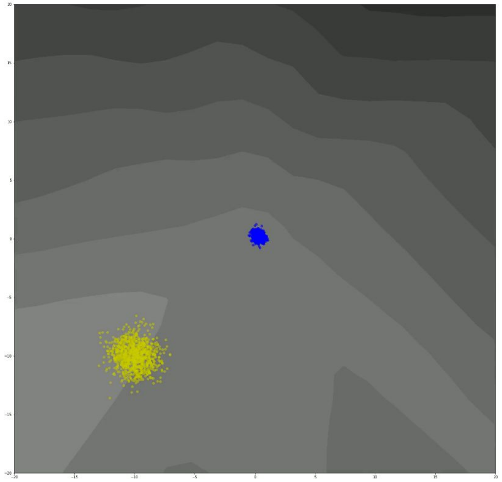
SimpleGAN: Implementación

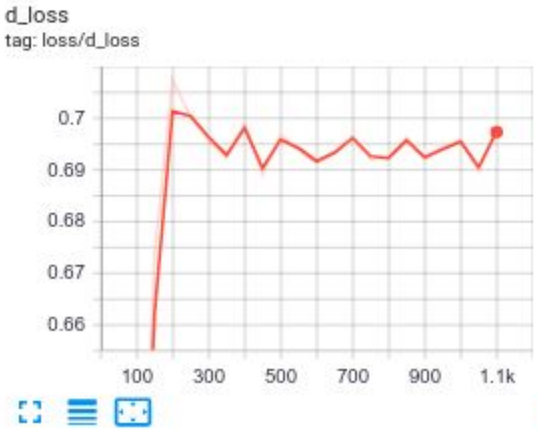
```python
def training_step(self, batch, batch_idx, optimizer_idx):
    data, _ = batch

    # sample noise
    z = torch.randn(data.shape[0], self.hparams.latent_dim)
    z = z.type_as(data)

    # train generator
    if optimizer_idx == 0:
        for param in self.discriminator.parameters():
            param.requires_grad = False
        for param in self.generator.parameters():
            param.requires_grad = True
        # generate images
        self.generated_data = self(z)

        # ground truth result (ie: all fake)
        # put on GPU because we created this tensor inside training_loop
        valid = torch.ones(data.size(0), 1)
        valid = valid.type_as(data)

        # adversarial loss is binary cross-entropy
        g_loss = self.adversarial_loss(self.discriminator(self.generated_data), valid)
        tqdm_dict = {'g_loss': g_loss}
        output = OrderedDict({
            'loss': g_loss,
            'progress_bar': tqdm_dict,
            'log': tqdm_dict
        })
        return output

    # train discriminator
    if optimizer_idx == 1:
        # Measure discriminator's ability to classify real from generated samples
        for param in self.discriminator.parameters():
            param.requires_grad = True
        for param in self.generator.parameters():
            param.requires_grad = False
        # how well can it label as real?
        valid = torch.ones(data.size(0), 1)
        valid = valid.type_as(data)

        real_loss = self.adversarial_loss(self.discriminator(data), valid)

        # how well can it label as fake?
        fake = torch.zeros(data.size(0), 1)
        fake = fake.type_as(data)

        fake_loss = self.adversarial_loss(
            self.discriminator(self.generated_data.detach()), fake)

        # discriminator loss is the average of these
        d_loss = (real_loss + fake_loss) / 2
        tqdm_dict = {'d_loss': d_loss}
        output = OrderedDict({
            'loss': d_loss,
            'progress_bar': tqdm_dict,
            'log': tqdm_dict
        })
        return output
```

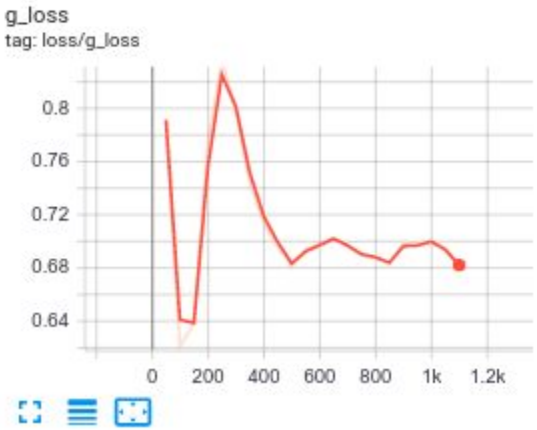
SimpleGAN: Entrenamientos

Dataset 1: Gaussiana 2D mu=(-10,-10), sigma=1

Caso: Discriminador y Generador de Alta Varianza



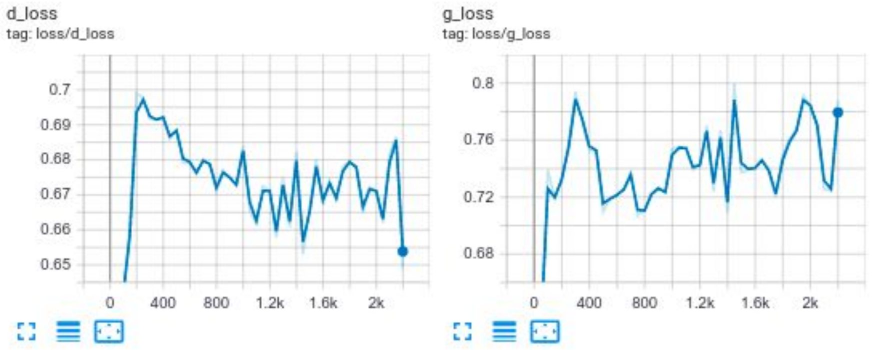


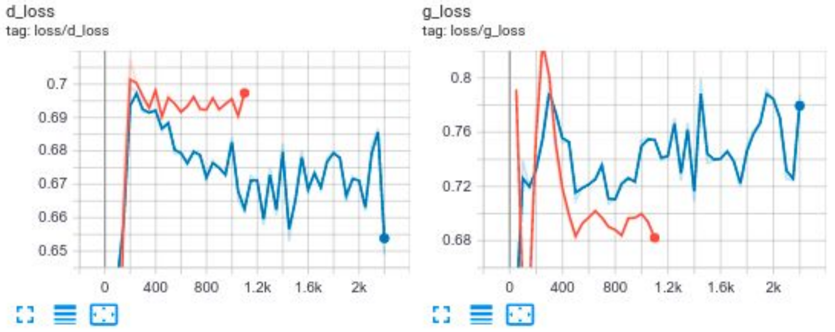


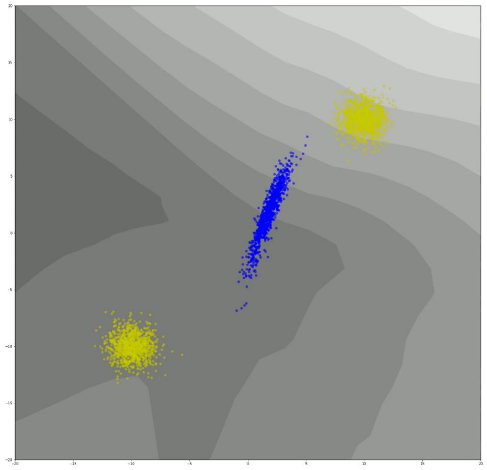
SimpleGAN: Entrenamientos

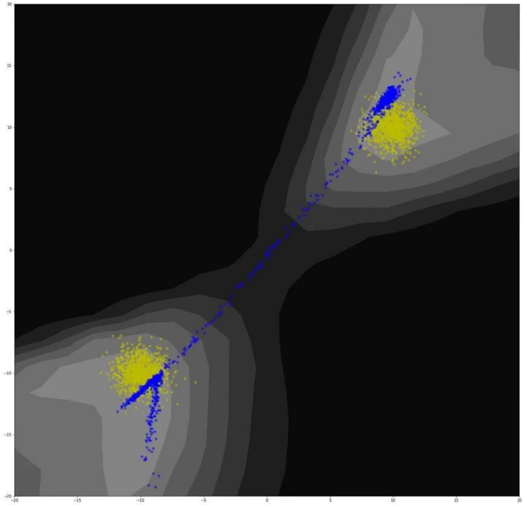
Dataset 2: GMM 2D mu=[(-10,-10), (10,10)], sigma=1

Caso: Discriminador y Generador de Alta Varianza





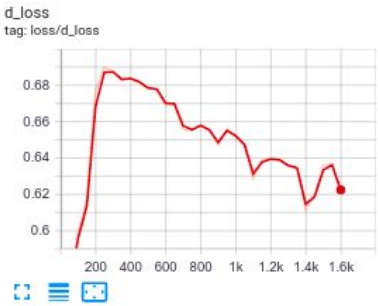


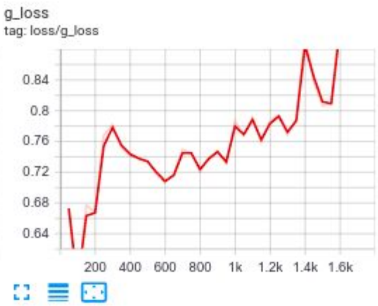


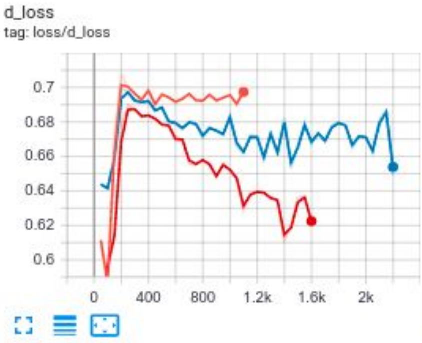
SimpleGAN: Entrenamientos

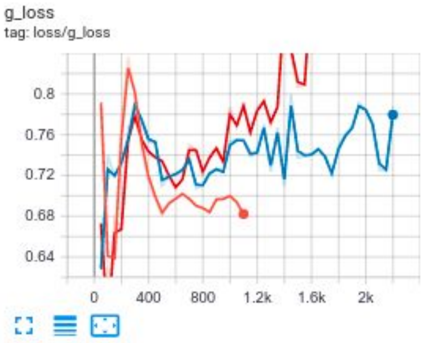
Dataset 3: GMM 2D mu=[(-10,-10), (10,10), (10,-10)], sigma=1

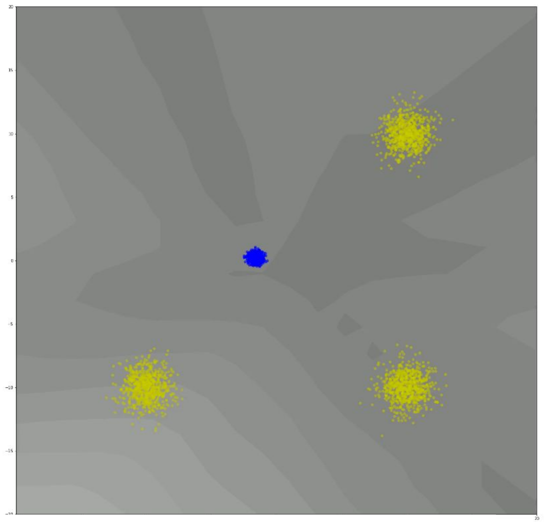
Caso: Discriminador y Generador de Alta Varianza

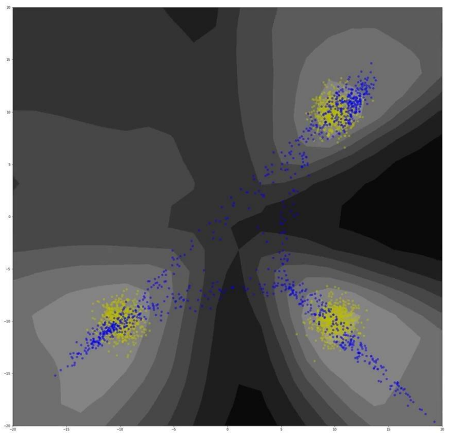








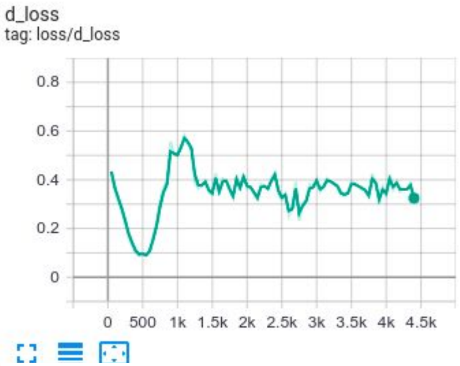


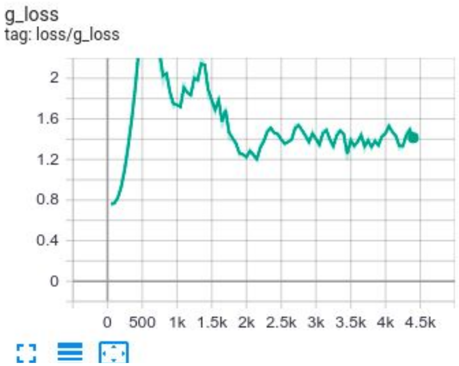


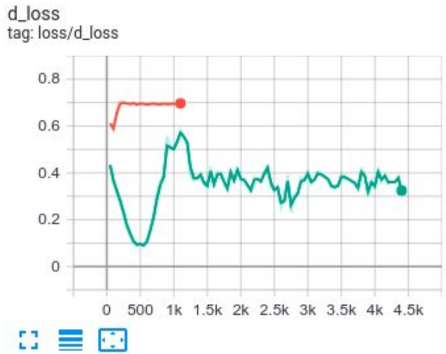
SimpleGAN: Entrenamientos

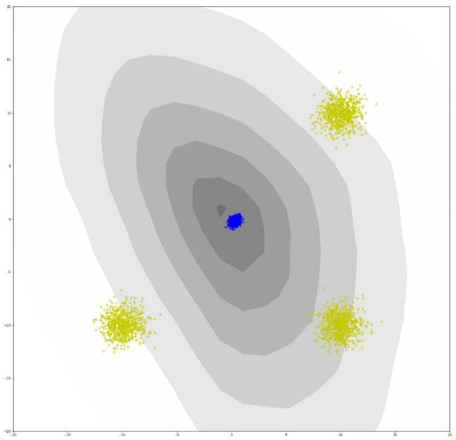
Dataset 3

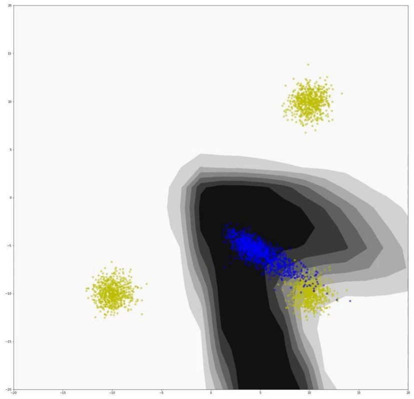
Caso: Discriminador AV y Generador BV







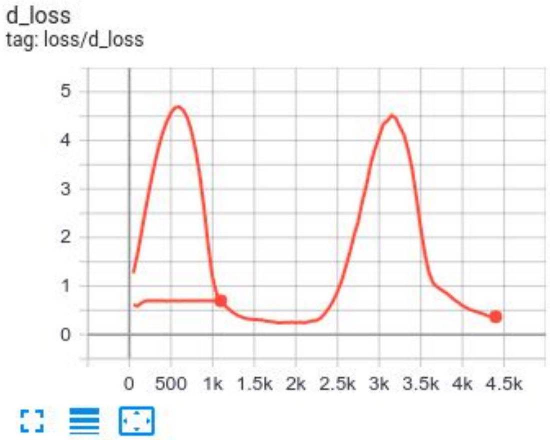


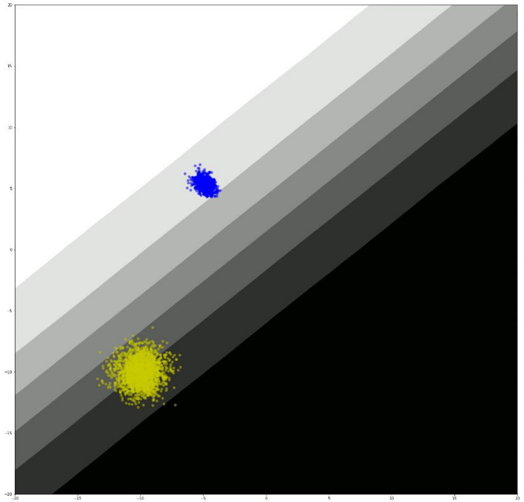


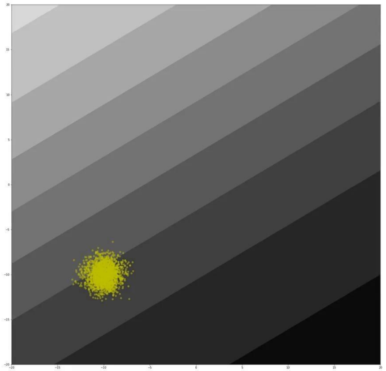
SimpleGAN: Entrenamientos

Dataset 1

Caso: Discriminador BV y Generador AV



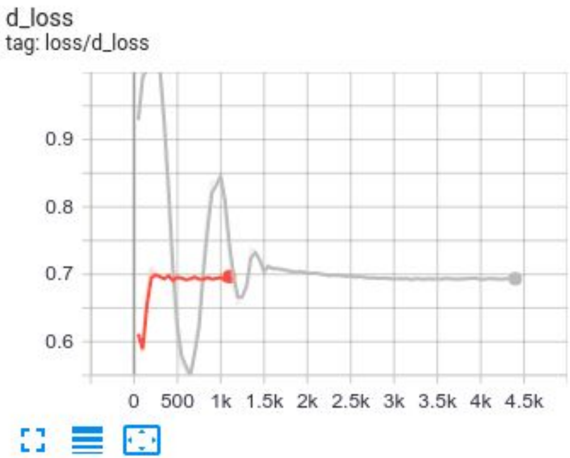


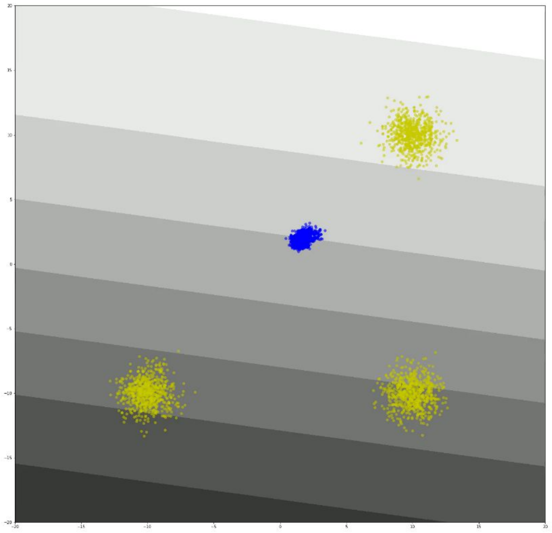


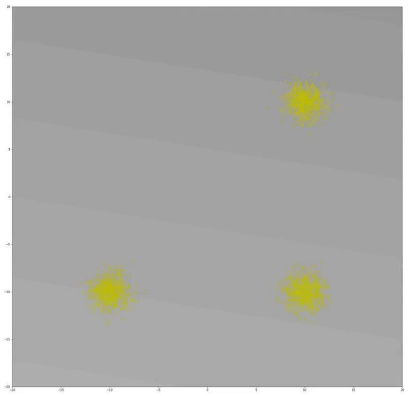
SimpleGAN: Entrenamientos

Dataset 3

Caso: Discriminador BV y Generador AV



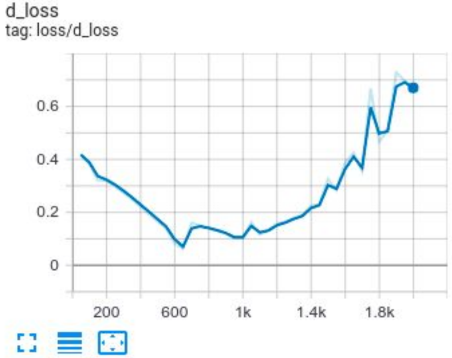


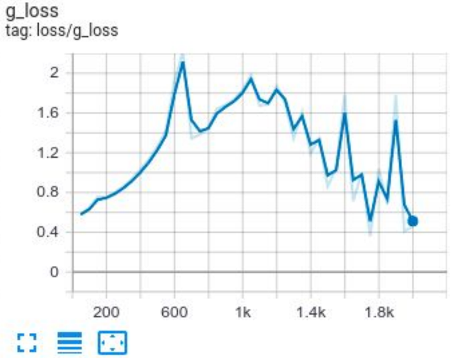


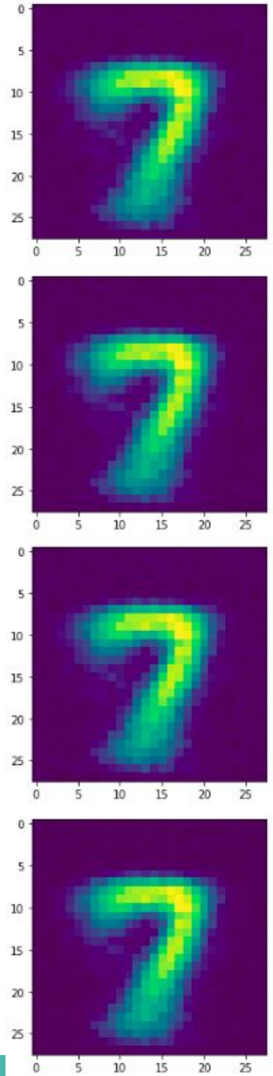
SimpleGAN: Entrenamientos

Dataset MNIST-7

Caso: Discriminador AV y Generador AV



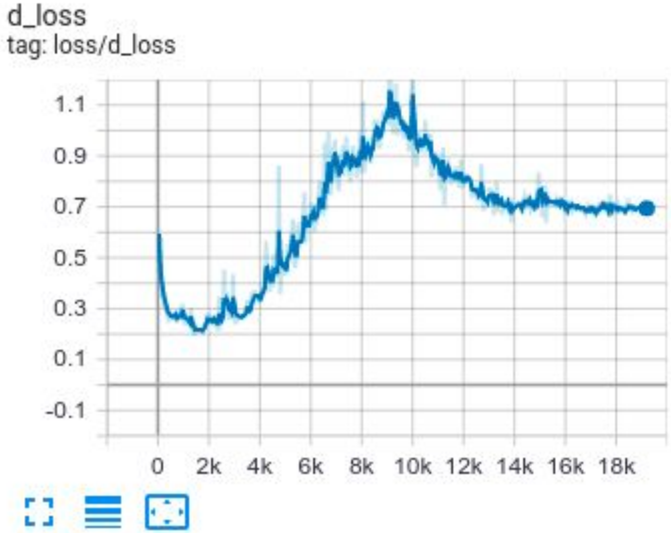


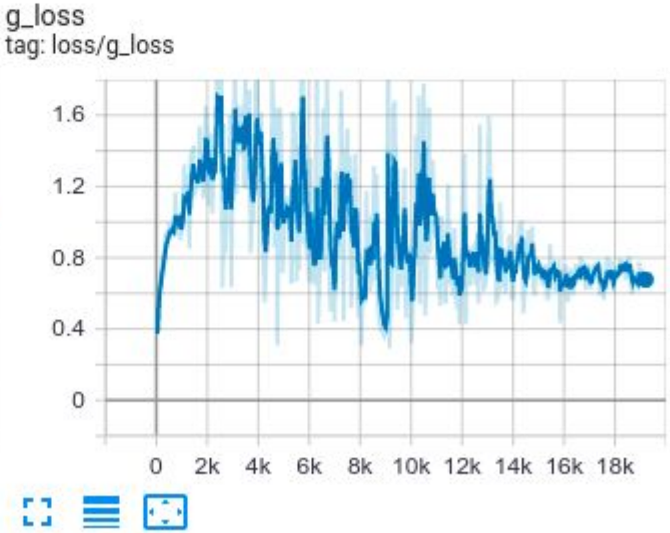


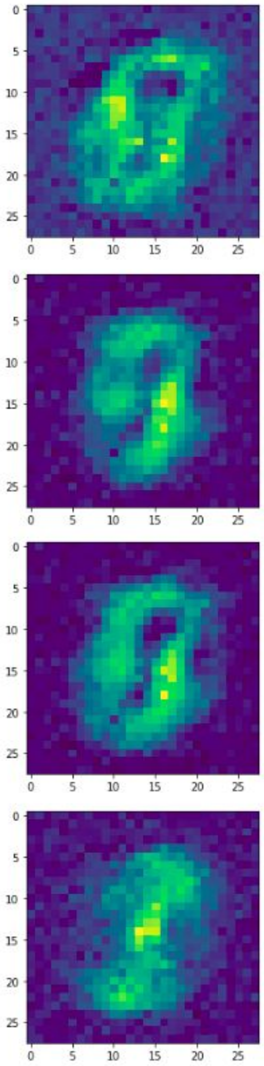
SimpleGAN: Entrenamientos

Dataset MNIST

Caso: Discriminador ?? y Generador ?? (Según lo visto anteriormente, le falta varianza al discriminador)



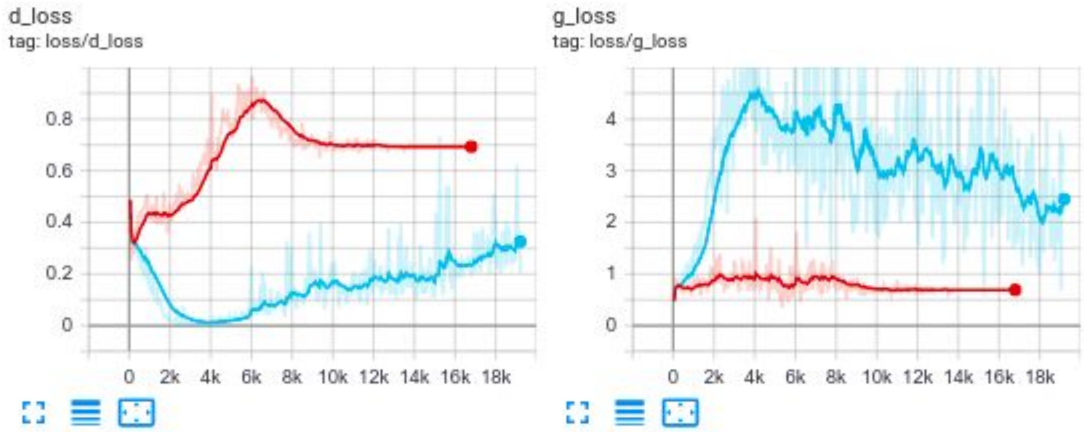


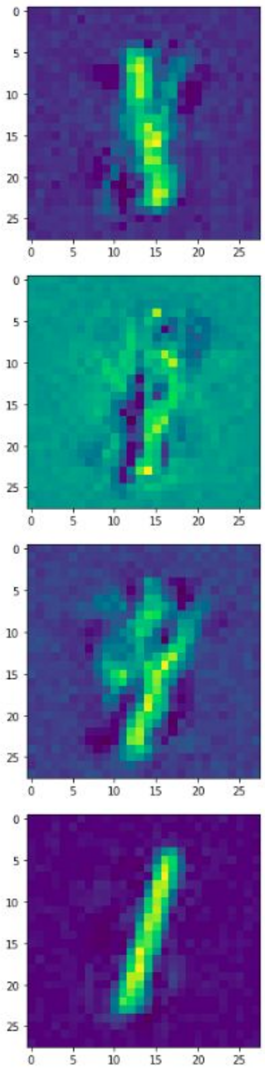


SimpleGAN: Entrenamientos

Dataset MNIST

Caso: Discriminador ?? y Generador ?? (Según lo visto anteriormente, le falta varianza al discriminador)

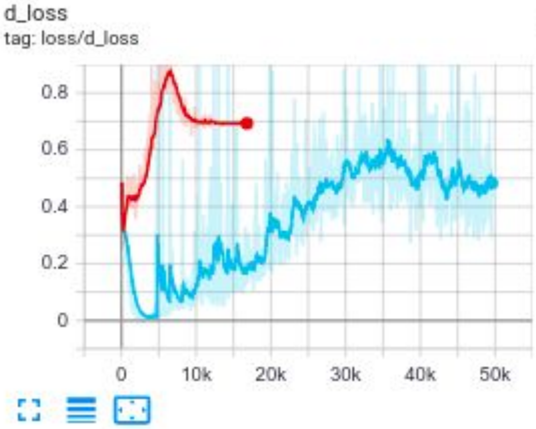


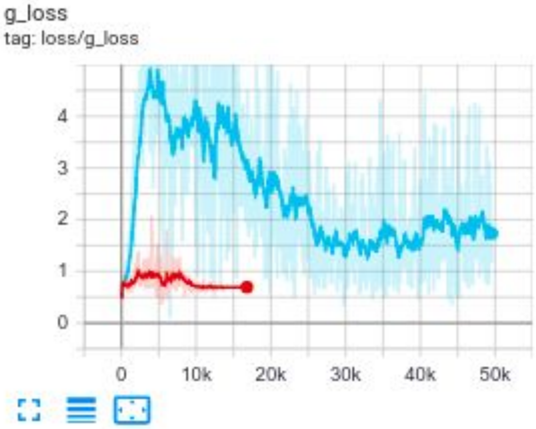


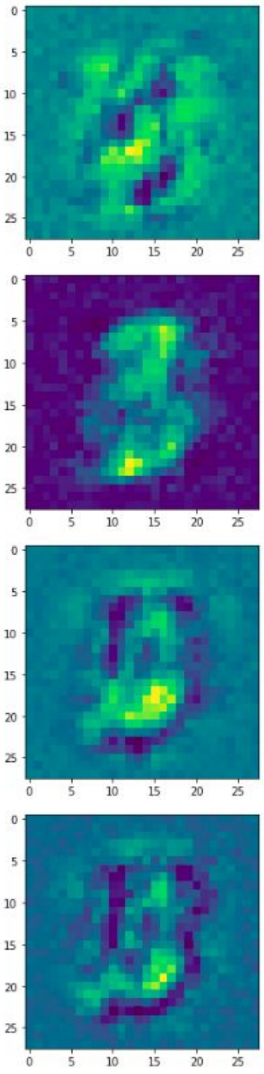
SimpleGAN: Entrenamientos

Dataset MNIST

Caso: Discriminador ?? y Generador ?? (El discriminador está ok, le falta varianza al generador)



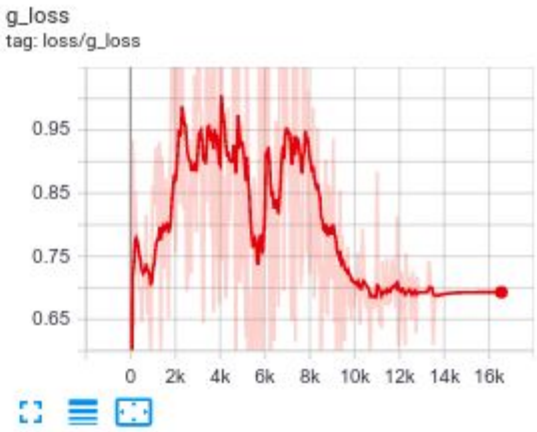


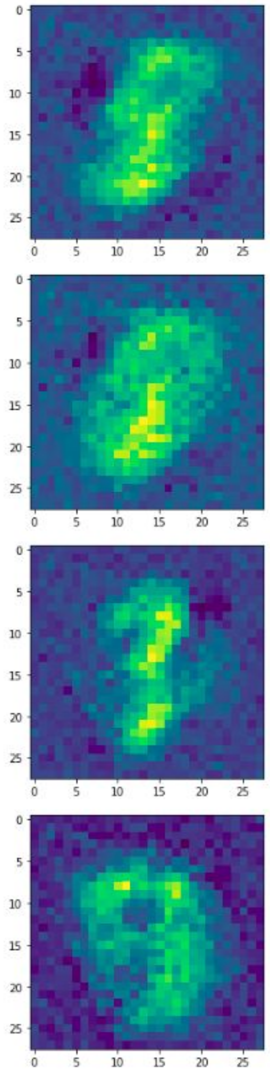


SimpleGAN: Entrenamientos

Dataset MNIST

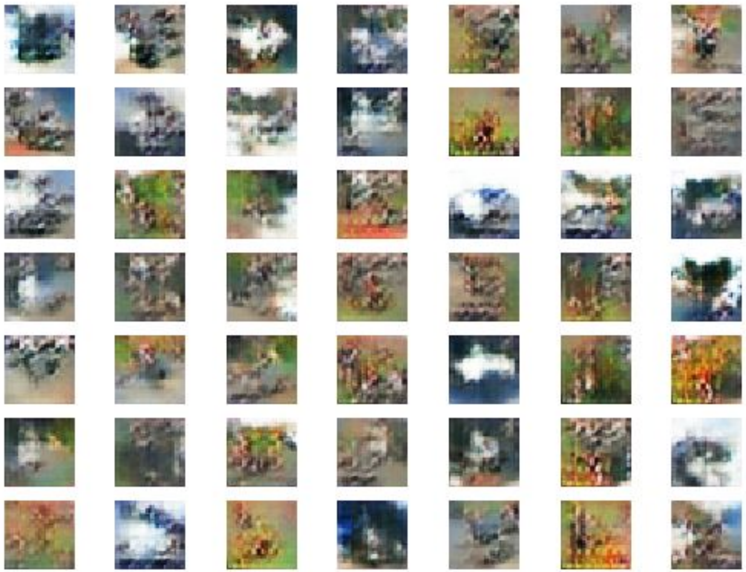
Caso: Discriminador == y Generador +V



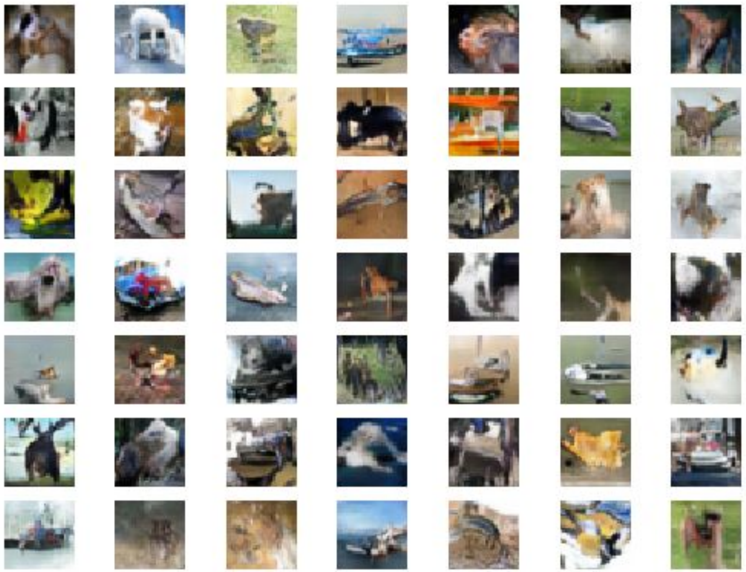


DCGAN: CIFAR-10

[How to Develop a GAN to Generate CIFAR10 Small Color Photographs]



10 epochs



200 epochs