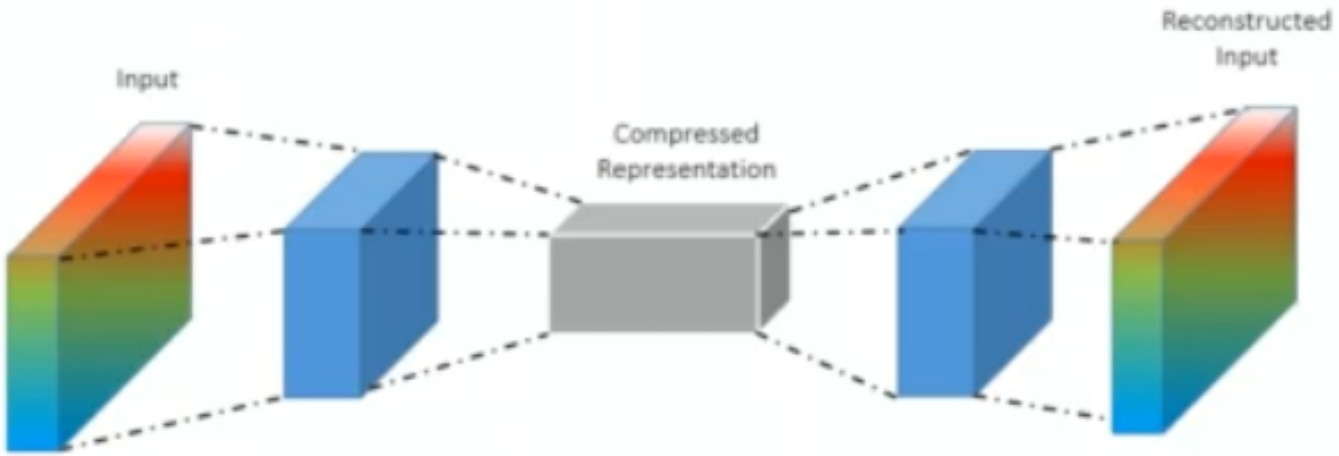
Machine Learning L+Pr

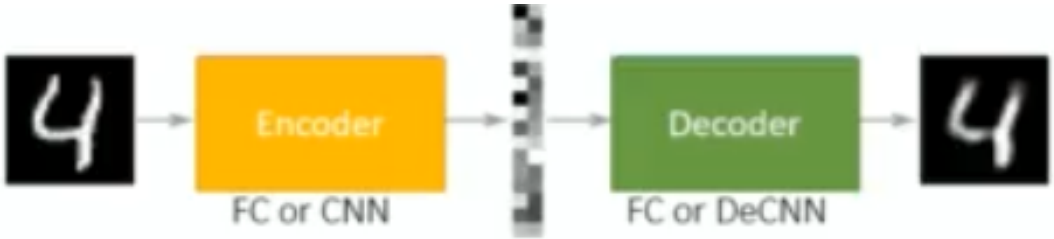
Béla J. Szekeres, PhD

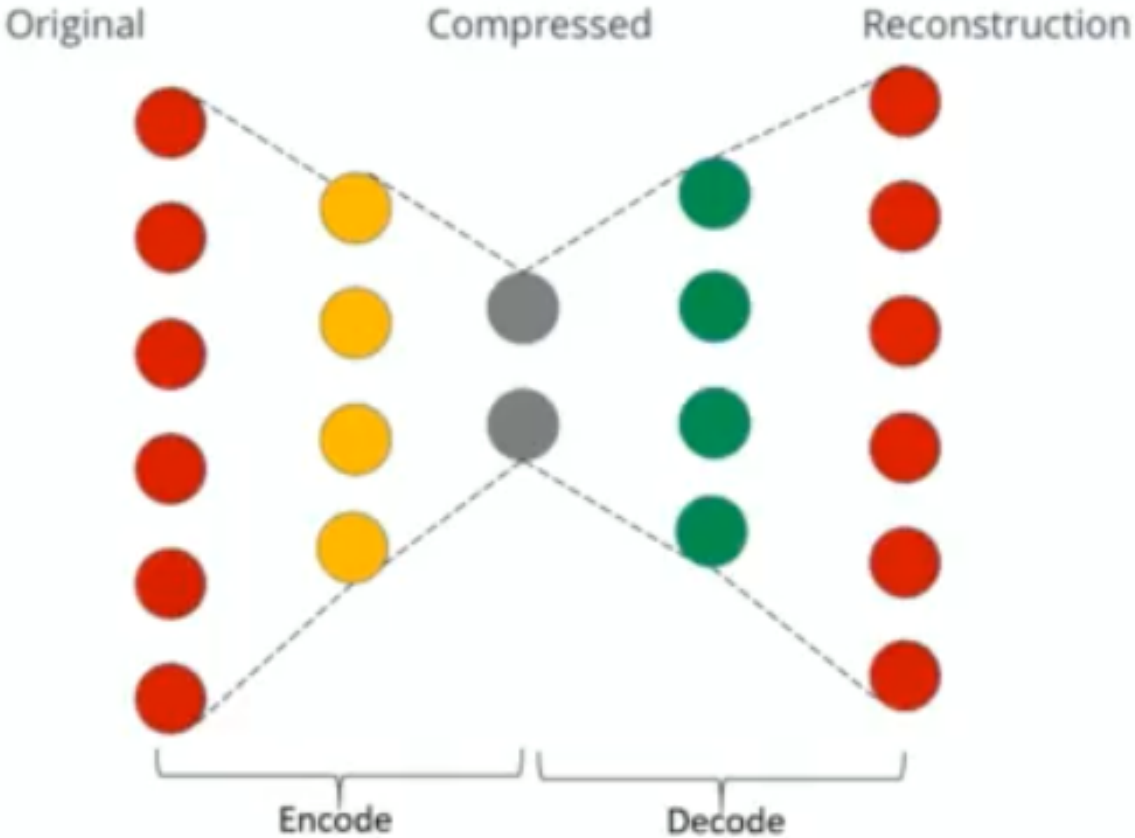
Lecture 12

ELTE Faculty of Informatics, Szombathely, Hungary

Autoencoders and Variational Autoencoders



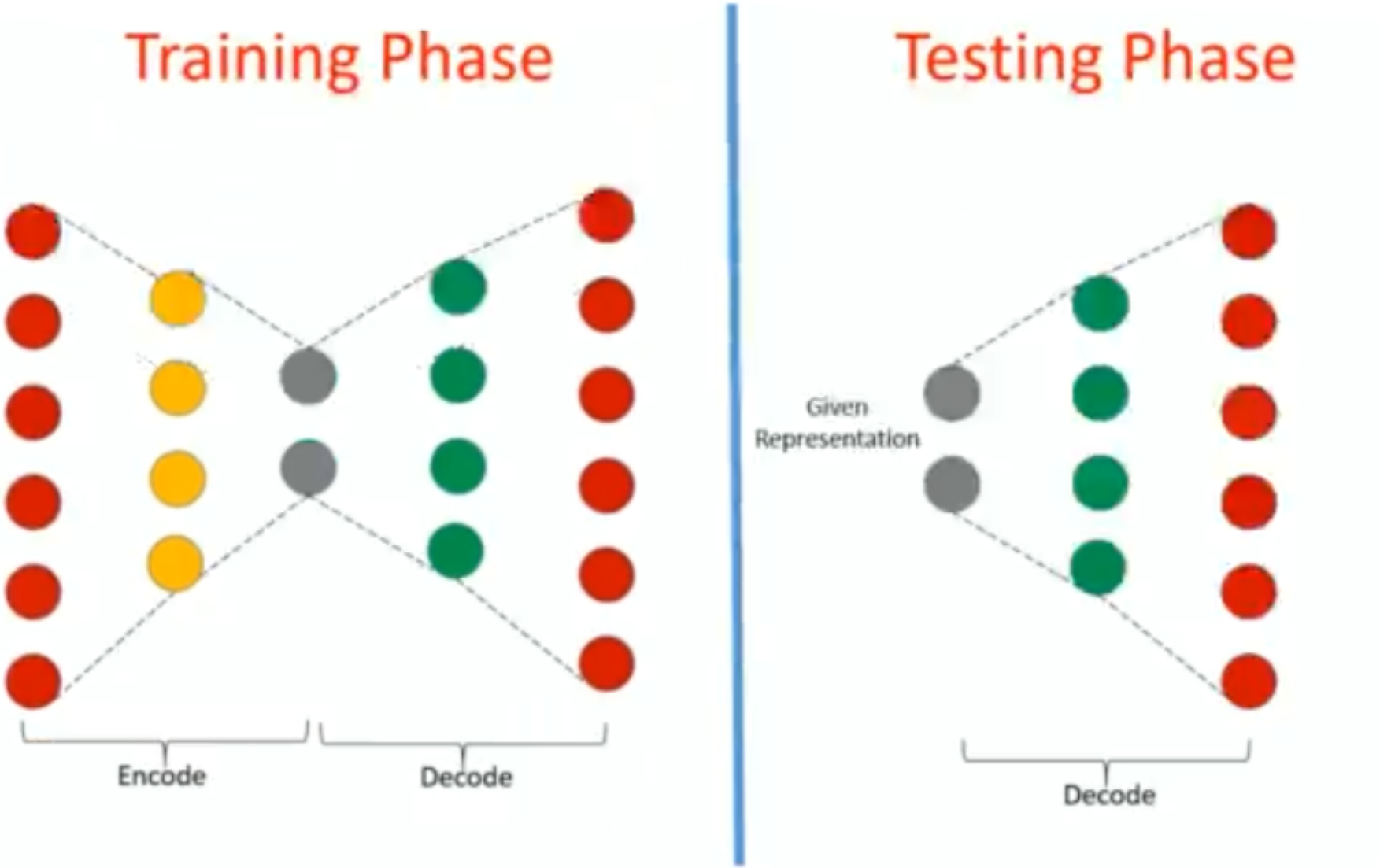




Autoencoders

They encode the input into a smaller representation, and decode it back again. In other words, it learns to reconstruct the input from a compressed representation of it (which is what the decoder does).

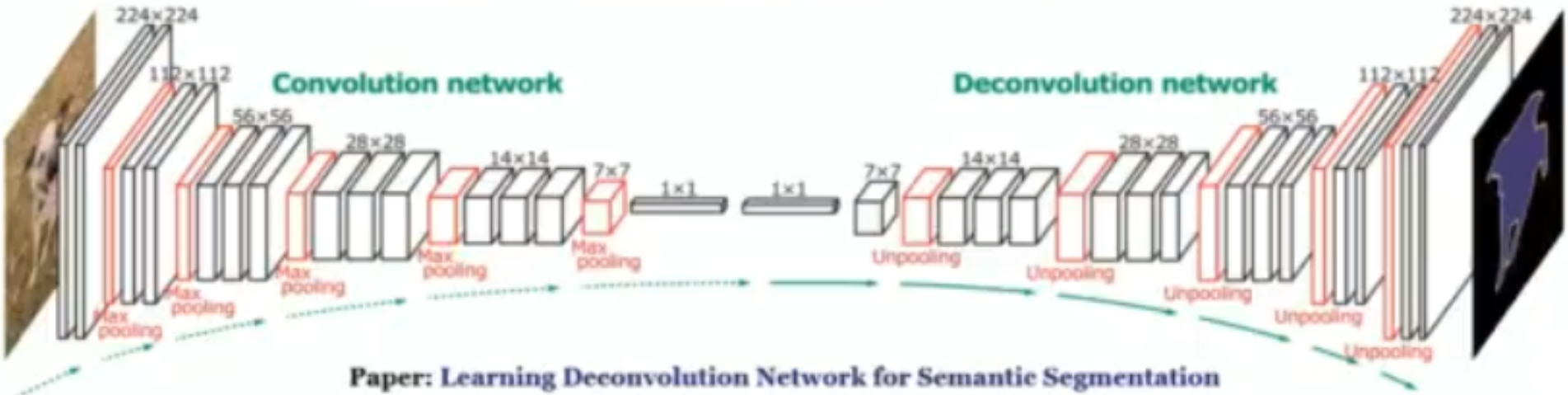
The output is expected to be the input (therefore the output label is just the input itself!)


Training Phase
Encode
Decode
Testing Phase
Given
Representation
Decode

Example: Image Compression

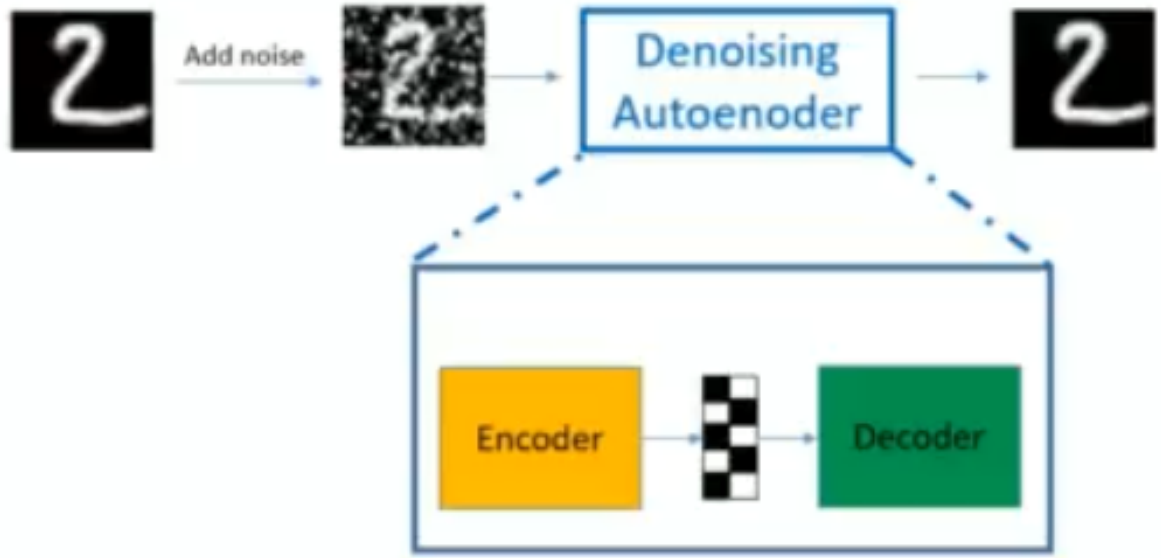
- Downloading a compressed image to your device (phone), and reconstructing it back from the decoder in your device.

Example: Image Segmentation



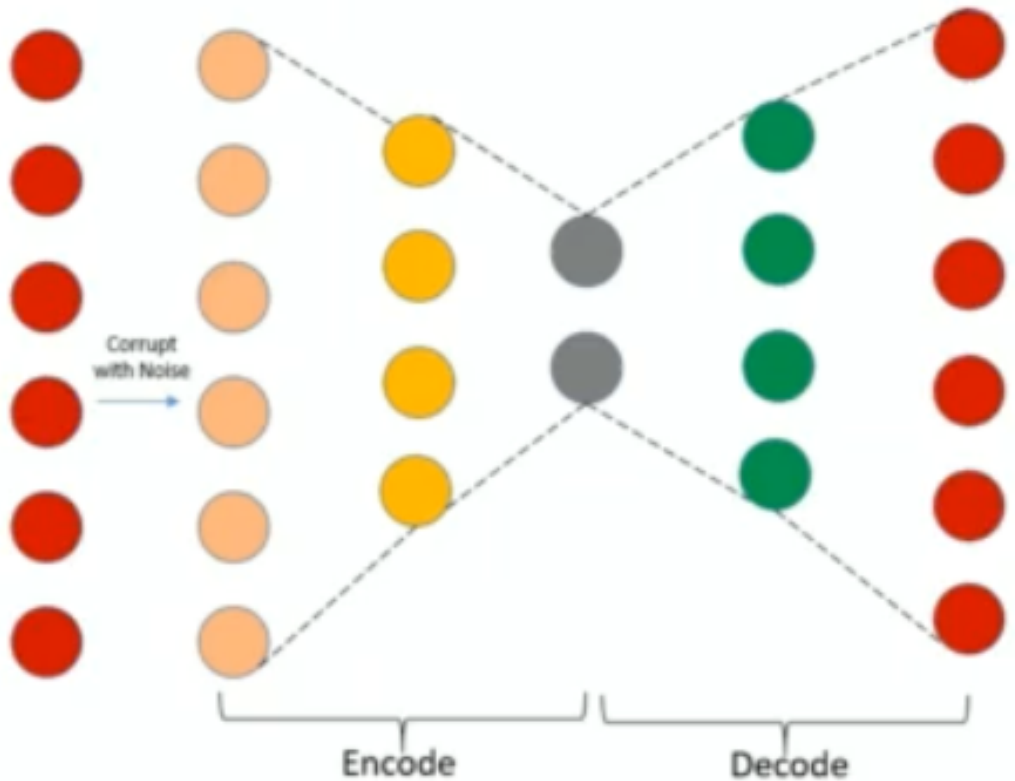
Paper: Learning Deconvolution Network for Semantic Segmentation

Denoising Autoencoders: Input Reconstruction

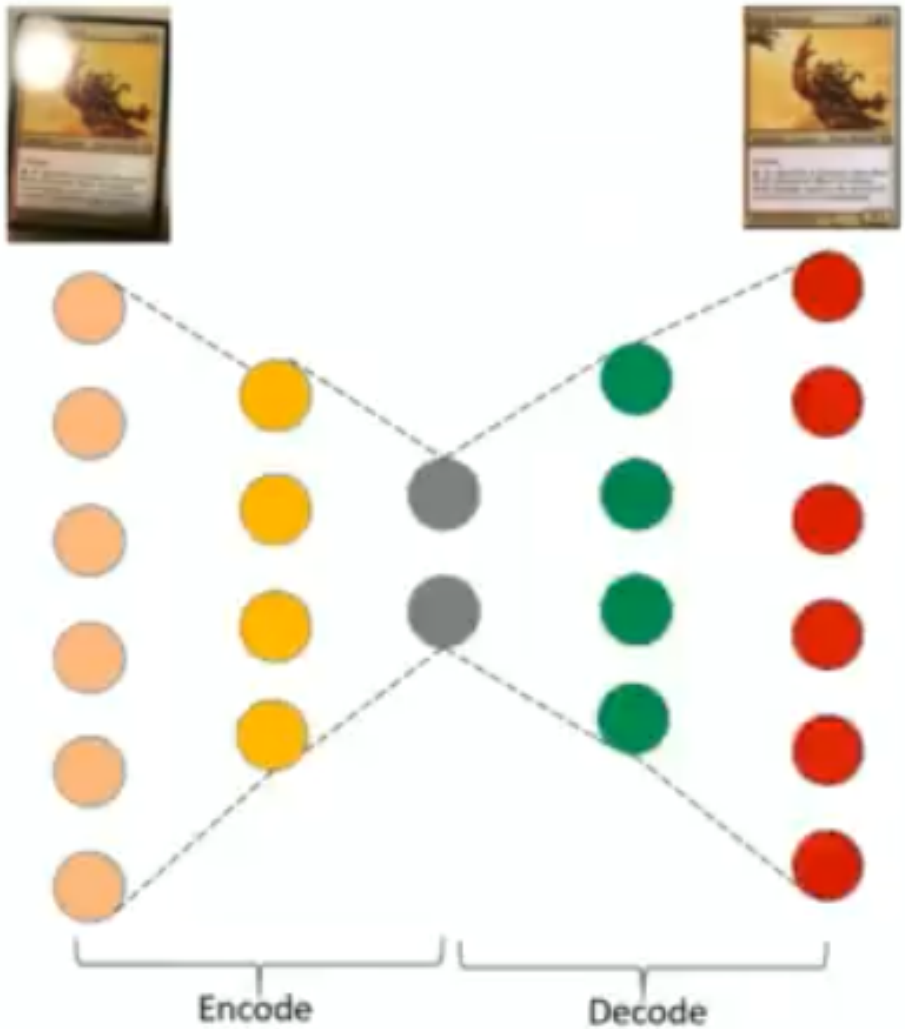


Through training, the encoder learns how to get rid of the noise

Denoising Autoencoders: Input Reconstruction



Same Concept, but remove image parts rather than adding noise: **Image Inpainting**


Encode
Decode

Loss Function for Autoencoders

- We can either use the binary cross entropy (even when our output labels are not binary → output labels has values between 0 and 1)

$$BCE = -\frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$$

- Or MSE loss

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

What's the Problem in Autoencoders?

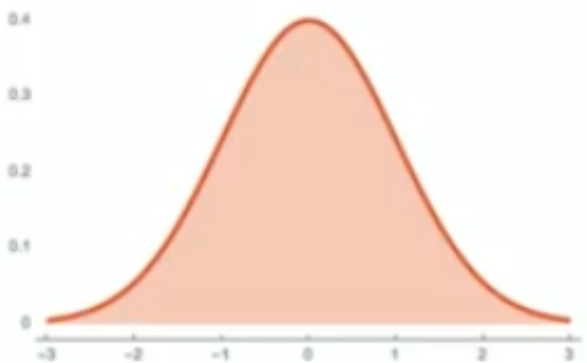
- The compressed representation is not know to which distribution it belongs to.

- At testing time, if we give random numbers to the decoder, we might get a random image:

Therefore, it is necessary to know which distribution to take the random numbers from and supply them to the decoder.

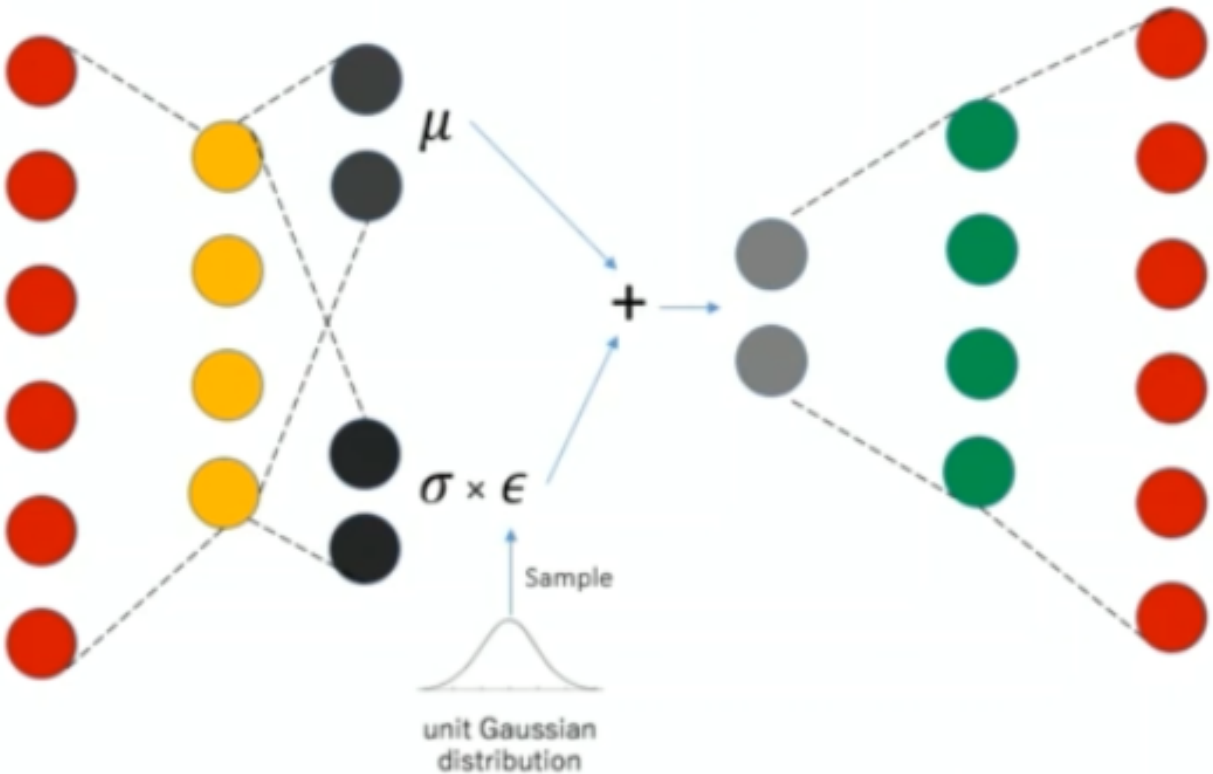
Solving the problem

- Instead of forcing the encoder to produce a single encoding (as a normal autoencoder) with random numbers ***not known to which distribution they belong to***, we want to force the encoder to produce a **specific probability distribution** (in practice, a Gaussian/normal distribution) over encodings. The decoder will then sample an encoding from that probability distribution, and try to reconstruct the original input. → VAE



Variational Autoencoders

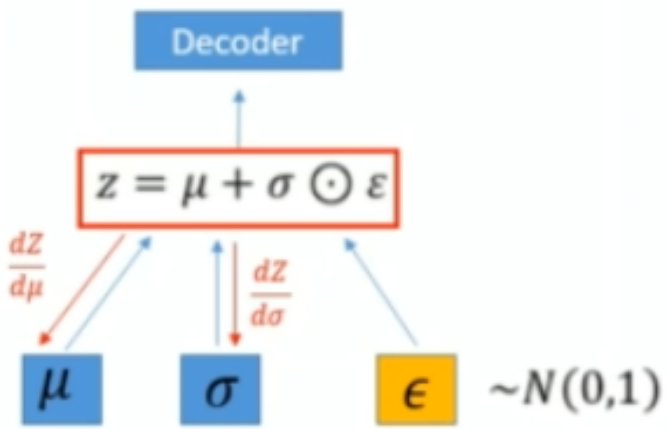
Autoencoding Variational Bayes, ICLR 2013



In VAE, the last encoder layer predicts two vectors: the mean μ and the variance σ. Remember that we want to learn a distribution similar to a normal/Gaussian distribution. We expect μ and σ to be from a normal distribution. We can sample from the mean and standard deviation we computed. However, sampling operation is **not differentiable** and we can't do backpropagation.

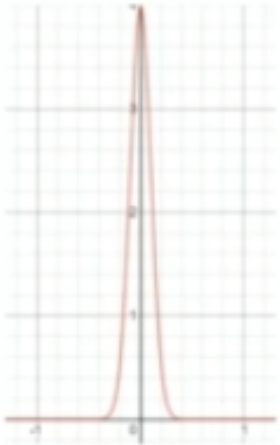
We can solve this using the *reparameterization trick*

We can randomly sample ϵ from a unit Gaussian (a constant), and then shift it by the latent distribution's mean μ and scale it by the latent distribution's variance σ. Now we can perform backpropagation.

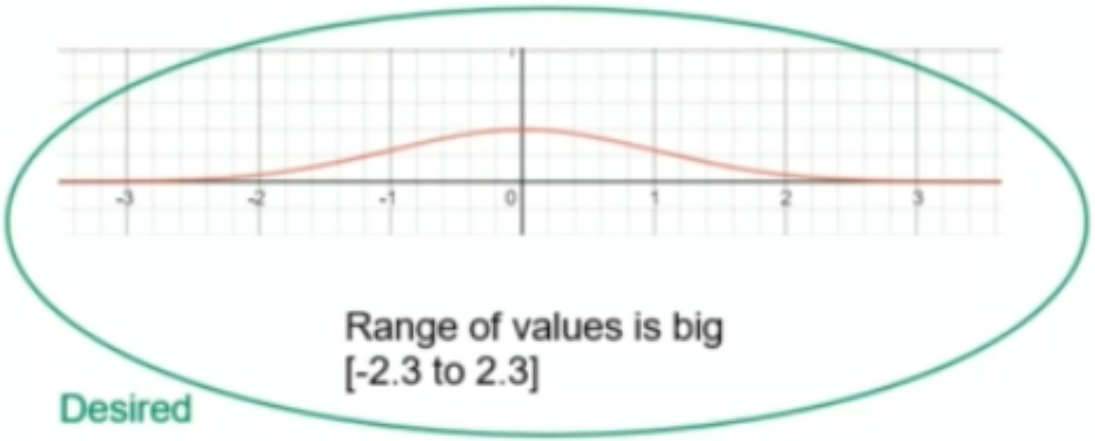


What if the encoder produces a very narrow distribution?

This is desirable from the perspective of the model, since the decoder would no longer have to care about covering a wide range of inputs. This defeats the purpose of VAE

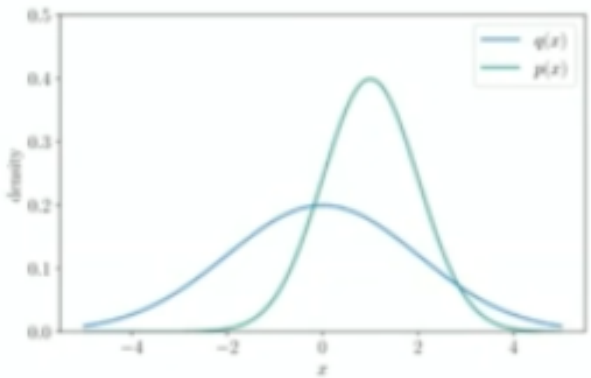


Range of values is small
[-0.25 to 0.25]



So how can we solve this problem?

The VAE introduces a loss other than the reconstruction loss: **the KL divergence between the distribution produced by the encoder and a unit Gaussian distribution**. Adding the KL divergence term to the loss forces the model to make a trade-off between ease of reproduction and covering the unit Gaussian. This allows us to safely create new images from encodings sampled from the unit Gaussian.



Therefore, the KL divergence is included in the loss function to *improve the similarity between the distribution of latent variables from the network and the normal distribution*, in order to sample from a larger range.

In practice, however, it's better to predict $\Sigma(X)$ as $\log\Sigma(X)$, as it is more numerically stable. Therefore, the encoder output (for the variance) is $\log\Sigma(X)$

Hence, our final KL divergence term is:

$$D_{KL}[N(\mu(X), \Sigma(X)) \| N(0,1)] = \frac{1}{2} \sum_k \left(\exp(\Sigma(X)) + \mu^2(X) - 1 - \Sigma(X) \right)$$

PyTorch Code:

```
reconst_loss = F.binary_cross_entropy(reconstructed, images)
kl_div = torch.mean(0.5 * torch.sum(log_var.exp() + mean.pow(2) - 1.0 - log_var, 1))
loss = reconst_loss + kl_div
```

Deep Fake

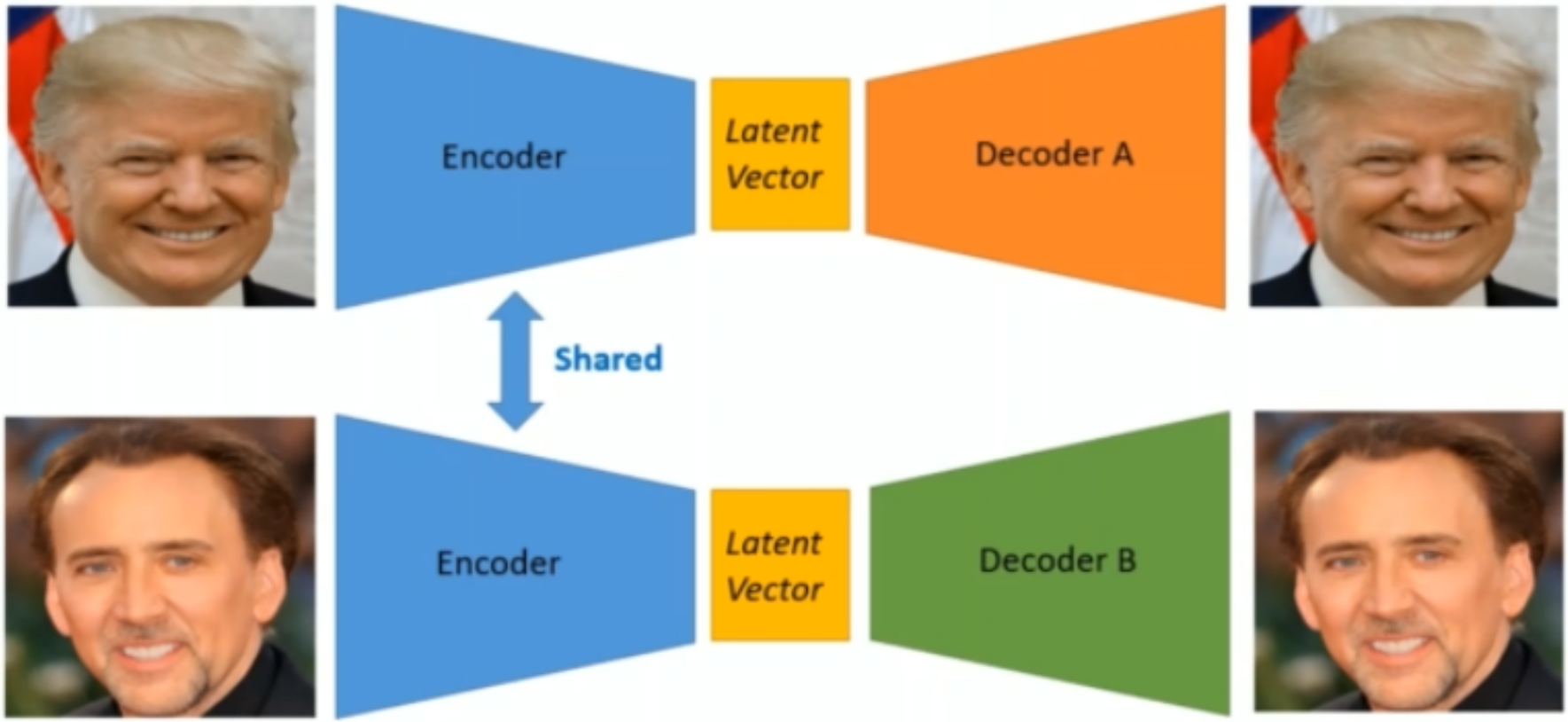
Donald Trump → Mr. Bean

Home Alone → Home Stallone

Gather Dataset

Training Phase

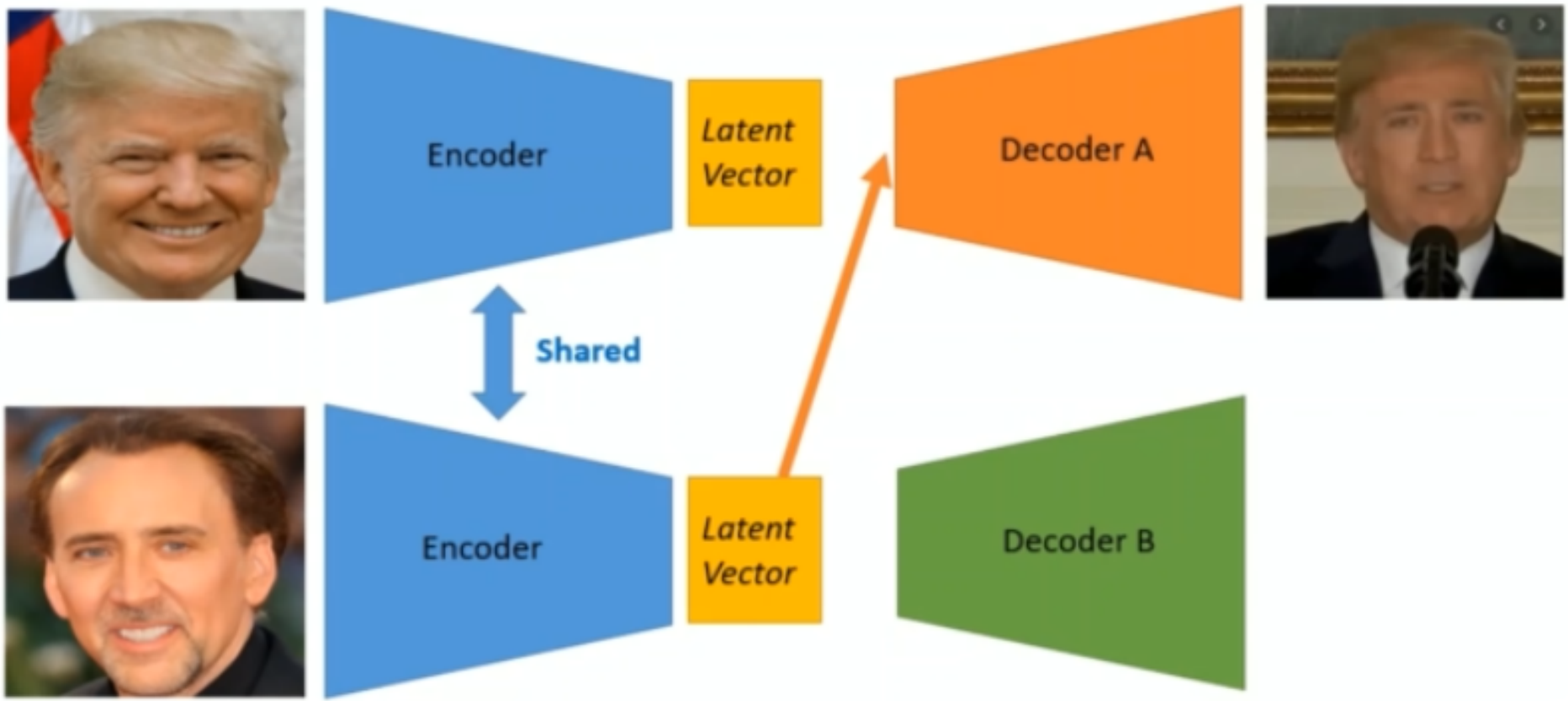
The **Decoder A** is only trained with faces of A; the **Decoder B** is only trained with faces of B. However, all latent faces are produced by the **same Encoder**.



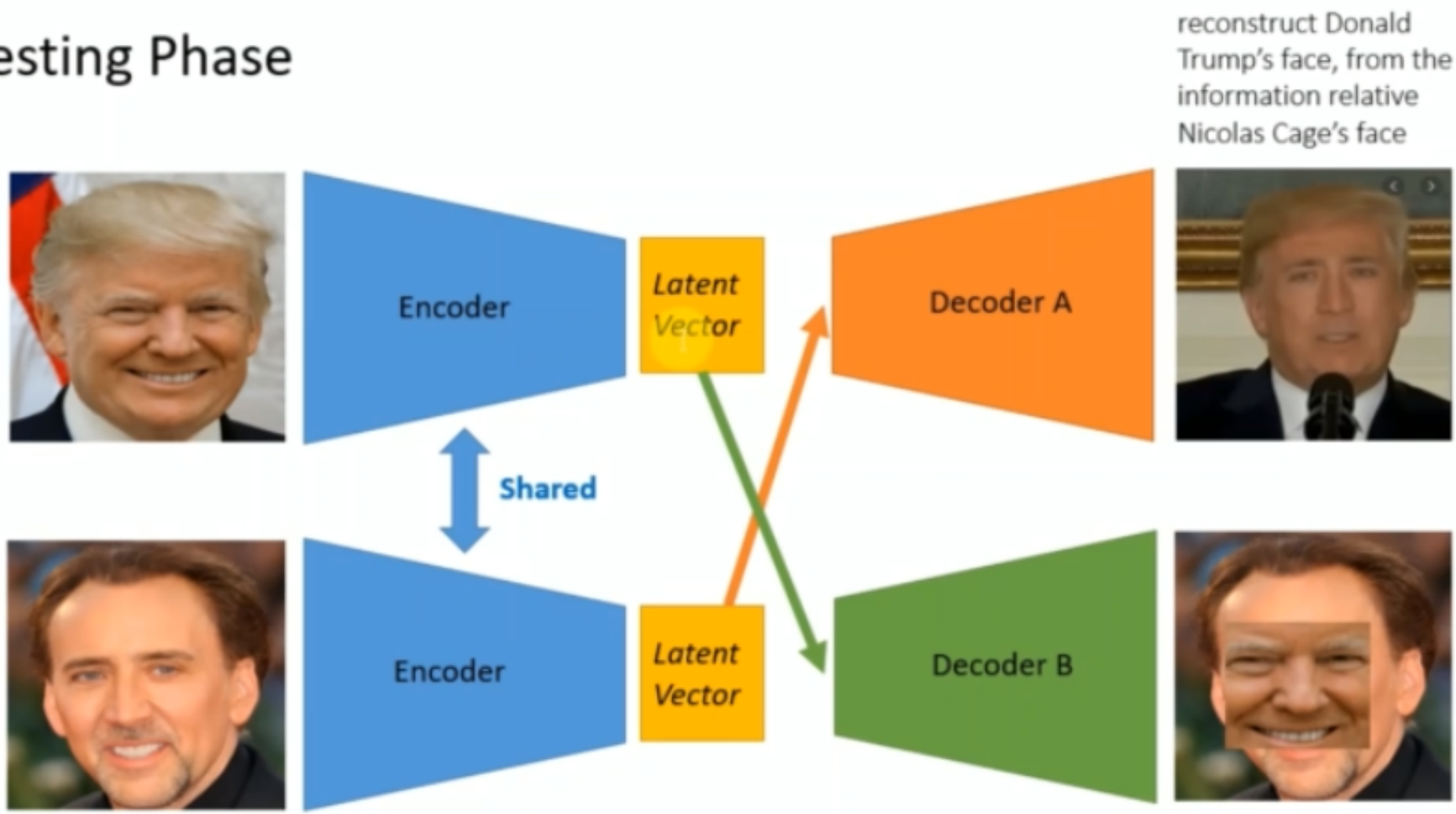
Why shared Encoder?

We don't want the encoder (latent representation) to represent different features, we want them to learn *important shared features (e.x face)* from both images → The encoder learns common features in both faces.

Testing Phase



Testing Phase



Summary

- https://arxiv.org/pdf/2201.03898.pdf