AutoEncoder (AE): High Level

TL;DR

- The Deep Learning analog of Principal Components (PCA)
    - Most of the lessons of AE apply equally to PCA
- Unsupervised: no labels (really semi-supervised)
- Create "synthetic features" from the original set of features
- May be able to use reduced set of synthetic features (dimensionality reduction)
- **Generative (vs Discriminative)**

Autoencoder

An Autoencoder network has two parts

- **An Encoder, which takes input $\mathbf{x}$ and "encodes" it into $\mathbf{z}$**
- **A Decoder, which takes the encoding $\mathbf{z}$ and tries to reproduce $\mathbf{x}$**

Each part has its own weights, which can be discovered through training, with examples

- $\langle \mathbf{X}, \mathbf{y} \rangle = \langle \mathbf{X}, \mathbf{X} \rangle$

That is: we are asking the output to be identical to the input.

$\mathbf{z}$ is an alternative latent representation of $\mathbf{x}$.

- **Encoded by the Encoder**
- **Inverted by the Decoder**

But when the dimension of $\mathbf{z}$ is less than the dimension of $\mathbf{x}$.

- **$\mathbf{z}$ is a *bottle-neck***
- **the inversion by the Decoder will be imperfect**

$\mathbf{z}$ becomes a *reduced-dimensionality* approximation of $\mathbf{x}$.

This is reminiscent of the dimensionality reduction of Principal Components Analysis (PCA).

The *main difference* from PCA

- **PCA uses a *linear* transformation**
- **NN can use *non-linear* transformations too**
    - **PCA as a special case of AE**

Autoencoders: Uses

Dimension reduction

After training

- **we can discard the Decoder**
- **use the Encoder output (synthetic features) as reduced dimension inputs to a *new* task**
    - **Encoder weights are frozen: non-learnable when training new task**
        - **It may be easier to solve the new task given $\mathbf{z}$ rather than $\mathbf{x}$**
            - **have already discovered "structure" of $\mathbf{x}$**
        - ***Transfer Learning***

In PCA, we eliminated original features that were "less important"

- **i.e., explained variation among only a small fraction of the training set**
  - **recall how we re-denominated explained variance in terms of "number of features"**

There is no direct similar concept of feature importance in AE

- **other than minimizing a Loss function, which *may* wind up focusing on "important" features**

Layer-wise pre-training with Autoencoders

Autoencoders played a vital role in the development of Deep Learning:

- **They made it possible to train otherwise untrainable NN's.**
- **Other innovations supplanted the need for AE's to assist training**
    - **better initialization**
    - **better activations functions**
    - **normalization**

Although they are no longer needed for that purpose, it is interesting to see how (and why) they were used.

For a NN with L layers that solves a Supervised Learning Problem

- **Training attempts to learn the weights of all layers simultaneously**
- ***Layer wise pre-training* was an attempt**
    - **to *initialize* the weights of each layer**
    - **in succession**
    - **so that the task of simultaneously solving for optimal weights had a better chance of succeeding**

The idea was to learn an initialization of $\mathbf{W}_{(l)}$, the weights of layer l.

- After having learned the weights $\mathbf{W}_{(l')}$ for all layers $l' < l$.

To initialize $\mathbf{W}_{(l)}$:

- Train an AE that takes $\mathbf{x}^{(\mathbf{i})}$ as input
- Using initialized weights $\mathbf{W}_{(l')}$ for all layers $l' < l$
- Produces $\tilde{\mathbf{x}}^{(\mathbf{i})}$ at layer l's output $\mathbf{y}_{(l)}$

So weight initializations were learned layer by layer.

Note that the labels $\mathbf{y}^{(\mathbf{i})}$ *were not used* !

- wouldn't be useful for the shallow NN

It was thought

- to be easier to learn the structure of the input $\mathbf{x}$ independent of the labels
- to be easier to learn $\mathbf{W}_{(l)}$ incrementally

One the weights $\mathbf{W}_{(l)}$ were initialized via AE's

- training of the Supervised Learning task had a better chance of succeeding
- compared to any other initialization

Autoencoders and Transfer Learning

Today, autoencoders are useful for another purpose: Transfer Learning.

If we can train an AE network to create features that are useful for reconstruction

- **it is possible that these features are useful for solving more complicated tasks.**

This was in essence what

- **Our dimension reduction example (replace the head) was doing**
- **Layerwise Pre-training was attempting.**

So it is not uncommon to approach a complicated problem

- **by first constructing an autoencoder to come up with an alternate (and smaller) representation of the input space.**

Note that Autoencoders are *unsupervised*: they don't take labels.

So the encodings they produce stress syntactic similarity, rather than semantic similarity.

Their use in Transfer Learning depends on the hope that inputs that are syntactically similar also have the same labels.

Denoising

Very much like dimension reduction but with the assumption that

- **"less important" features are just random noise that is added to the true example**

Generative Artificial Intelligence

A less obvious use of AE (using the Decoder rather than the Encoder) is to *generate* examples.

Most of the Machine Learning we have studied thus far is *discriminative*

- $p(\hat{\mathbf{y}}^{(\mathbf{i})}|\mathbf{x}^{(\mathbf{i})}))$
    - **e.g., classifier: discriminate among the possible classes $\mathbf{y}^{(\mathbf{i})}$, given example $\mathbf{x}^{(\mathbf{i})}$**

We can use the Decoder on *arbitrary* $\mathbf{z}$ to *generate* a completely new $\mathbf{x}$:

- $p(\mathbf{x}^{(i')}|\mathbf{z}^{(i')})$ **for some i' not in training**
- ***generate* a new example i', in the domain of $\mathbf{x}$, that was not encountered during training**

Autoencoder (AE): Details

The *task* that trains an Autoencoder

- **Given input $\mathbf{x}^{(i)}$**
- **Output of Encoder: $\mathbf{z}^{(i)} = E(\mathbf{x}^{(i)})$**
- **Output of Decoder: $\tilde{\mathbf{x}}^{(i)} = D(\mathbf{z}^{(i)})$**
- **"Target": $\mathbf{x}^{(i)}$**

Both the Encoder and Decoder are parameterized (learnable parameters)

- **Goal: find the parameters such that**

$$\tilde{\mathbf{x}}^{(i)} = D(E(\mathbf{x})) \quad \approx \quad \mathbf{x}$$

$\mathbf{z}^{(i)} = E(\mathbf{x}^{(i)})$ is the latent representation of $\mathbf{x}^{(i)}$.

Loss function

The obvious loss functions compare the original $\mathbf{x}^{(\mathbf{i})}$ and reconstructed $\tilde{\mathbf{x}}^{(\mathbf{i})}$ feature by feature:

Mean Squared Error (MSE)

$$\mathcal{L}^{(\mathbf{i})} = \sum_{j=1}^{|\mathbf{x}|} (\mathbf{x}_j^{(\mathbf{i})} - \tilde{\mathbf{x}}_j^{(\mathbf{i})})^2$$

Binary Cross Entropy

For the special case where *each* original feature is in the range $[0, 1]$ (e.g., an image)

$$\mathcal{L}^{(\mathbf{i})} = \sum_{j=1}^{|\mathbf{x}|} \left(\mathbf{x}_j^{(\mathbf{i})} \log(\tilde{\mathbf{x}}_j^{(\mathbf{i})}) + (1 - \mathbf{x}_j^{(\mathbf{i})}) \log(1 - \tilde{\mathbf{x}}_j^{(\mathbf{i})}) \right)$$

Variational Autoencoder (VAE): Generative ML

Observe that the Decoder part of the "vanilla" AE $D(\mathbf{z}^{(\mathbf{i})})$

- **has been trained to produce "realistic" $\tilde{\mathbf{x}}^{(\mathbf{i})}$ *only* for a $\mathbf{z}^{(\mathbf{i})} = E(\mathbf{x}^{(\mathbf{i})})$**
  - **i.e., "realistic": appears to come from the distribution of training $\mathbf{X}$**
- **there is no guarantee that $D(\mathbf{z}^{(i')})$ for some i' not in training is realistic**

That is: the AE has not been trained to *extrapolate* beyond the training inputs.

A VAE is able generate outputs

- **that *could have* come from the training distribution from a latent representation $\mathbf{z}^{(i')}$**
- **but that *did not* come from $\mathbf{X}$.**

Our goal is constructing a Decoder that can extrapolate.

Variational Autoencoder (VAE)

The Decoder will take a *latent vector* $\mathbf{z}$ and produce $D(\mathbf{z})$, just as in a vanilla AE.

The difference is that $\mathbf{z}$ will be sampled from a *distribution* rather than being a unique mapping of a training example.

This will be done by modifying the Encoder

- **It will *indirectly* create $\mathbf{z}^{(i)}$**
- **It will compute *variables* $\boldsymbol{\mu}^{(i)}$ and $\boldsymbol{\sigma}^{(i)}$**
    - **$\mathbf{z}^{(i)}$ will be *sampled* from a distribution with mean $\boldsymbol{\mu}^{(i)}$ and standard deviations $\boldsymbol{\sigma}^{(i)}$**

As long as $\mathbf{z}$ is sampled from this distribution, the decoder will produce a "realistic" output.

Note

μ and σ are

- **vectors**
- **computed values (and hence, functions of $\mathbf{x}$) and not parameters**
- **so training learns a *function* from $\mathbf{x}^{(i)}$ to $\mu^{(i)}$ and $\sigma^{(i)}$**

To train a VAE:

- **pass input $\mathbf{x}^{(i)}$ through the Encoder, producing $\boldsymbol{\mu}^{(i)}$, $\boldsymbol{\sigma}^{(i)}$**
  - **use $\boldsymbol{\mu}^{(i)}$, $\boldsymbol{\sigma}^{(i)}$ to sample a latent representation $\mathbf{z}^{(i)}$ from the distribution**
- **pass the sampled $\mathbf{z}^{(i)}$ through the decoder, producing $D(\mathbf{z}^{(i)})$**
- **measure the reconstruction error $\mathbf{x}^{(i)} - D(\mathbf{z}^{(i)})$, just as in a vanilla AE**
- **back propagate the error, updating all weights and $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$**

Essentially, each input $\mathbf{x}^{(i)}$ has *many* latent representations (with different probabilities): any sample from the distribution.

Training

Encoder produces

$$E(\mathbf{x}) \quad = \quad q_\phi(\mathbf{z}|\mathbf{x}) \quad \approx \quad p_\theta(\mathbf{z}|\mathbf{x})$$

We sample from

$$\hat{\mathbf{z}} \quad \sim \quad q_\phi(\mathbf{z}|\mathbf{x})$$

Decoder produces

$$D(\hat{\mathbf{z}}) \quad = \quad p_\theta(\mathbf{x}|\mathbf{z})$$

Each time (epoch) that we encounter the same training example, we select another random element from the distribution.

So the VAE learns to represent the same example from multiple latents.

Generative

- **sample $\hat{\mathbf{z}} \sim \hat{p}(\mathbf{z})$**
- **use decoder to produce output $p_\theta(\mathbf{x}|\mathbf{z})$**

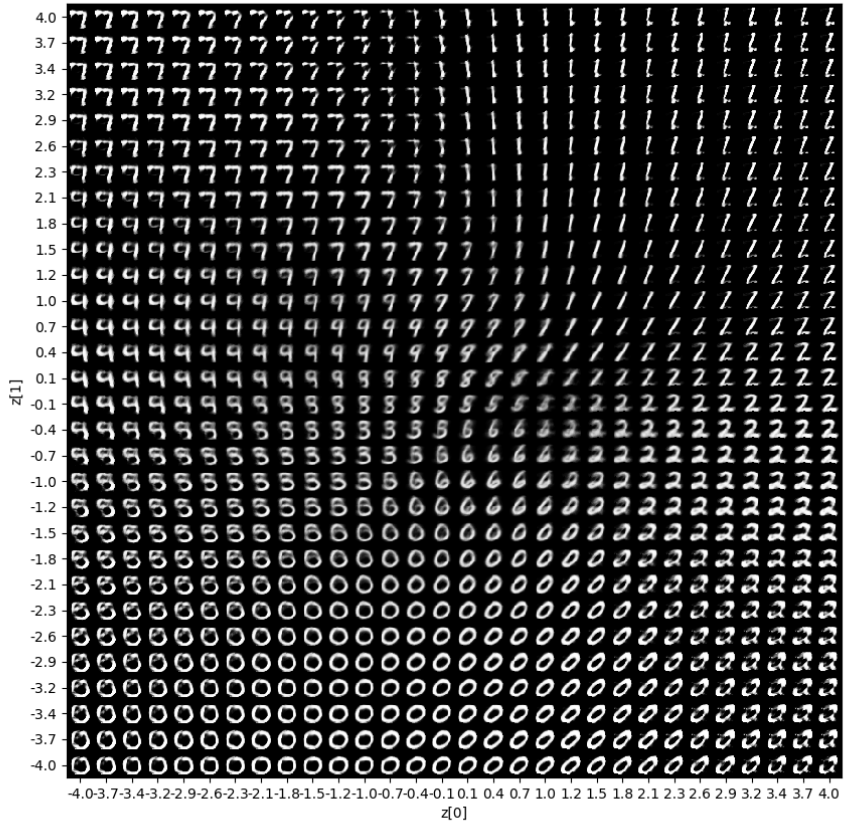
This means we can feed in a $\mathbf{z}$

- **that doesn't correspond to any training example**
- **and perhaps get an output that *resembles* something from the training set, rather than noise.**

To give you an idea of the generative nature of the VAE, consider

- **Creating latent vectors $\mathbf{z}$ from scratch**
  - **not as the output of the Encoder**
- **Varying these latent vectors systematically and examining the output created by the Decoder**

MNIST clustering produced by a VAE



Note that the outputs

- are not instances of any examples
- There was no guarantee that a random $\mathbf{z}$ would produce something that looked like a digit !

We may even be able to interpret the elements of $\mathbf{z}$

- $\mathbf{z}_0$: control slant ?
    - See the bottom row of $\mathbf{0}$'s
- $\mathbf{z}_1$: control "verticality" ?
    - See right-most column

Conditional VAE

Once a VAE is trained, $D(\mathbf{z})$ should produce a realistic output, for any $\mathbf{z}$ from the distribution.

However, if the distribution of $\mathbf{X}$ includes examples from many classes

- **Assuming we have labels as auxilliary information (not used in training)**
  - **e.g., the 10 digits**
- **The VAE can't control *which class* the output will come from**

A *Conditional VAE* allow our generator (Decoder) to control the class $\boldsymbol{c}$ of the output $\tilde{\mathbf{x}}$.

The class label c

- **is given as part of *training***
  - **So the Encoder produces a distribution that is conditioned on *both* $\mathbf{x}$ and c.**
- **is an *additional parameter* of the Decoder**
  - **So the output class can be controlled**

$$\tilde{\mathbf{x}}^{(\mathrm{i})} = D(\mathbf{z}^{(\mathrm{i})}, c)$$

So now we

- **create a latent $\mathbf{z}$**
- **append a class label c**
- **and presumably have the decoder produce an output from the desired class.**
- **The encoding distribution is now conditional on class label c: $q_\phi(z|x, c)$**
- **So is the decoding distribution $p_\theta(x|z, c)$**

Again, by restricting the functional form of the prior distribution $\hat{p}$ we can simplify the math.

Detour: Autoencoder notebook on Colab

Let's examine some Keras code that implements several types of Autoencoders

- **Vanilla**
- **Denoising**
- **VAE**

We will write our AE's using the Keras *Functional* API rather than the *Sequential* model

- **We *could* write the complete AE using the Sequential API**
- **But**
    - **we want to extract the Encoder and Decoder parts as *separate models***
    - **we can do this with the Functional API**

We will now switch to a notebook running on Google Colab Autoencoder example from github (https://colab.research.google.com/github/kenperry-public/ML_Fall_2021/blob/master/Autoencoder_example.ipynb)

VAE derivation: Probabilistic formulation

Note: Advanced material

Let's pretend: we don't already know that we will represent $\mathbf{z}$ as a function of $\mu_\theta(\mathbf{x})$ and $\sigma_\theta(\mathbf{x})$

- **this derivation will show why we made that choice**

The mathematical derivation of a VAE is quite detailed

- **it is interesting but not absolutely necessary to understand**
- **this is where we define the Loss function**

The interested reader is refered to a highly recommended VAE tutorial (https://arxiv.org/pdf/1606.05908.pdf).

We will try to give the essence in the following slides.

TL;DR

- The VAE has a very interesting two part Loss Function
    - Reconstruction Loss, as in the Vanilla AE
    - Divergence Loss
- The Reconstruction Loss is not sufficient
    - Issues of intractability arise
    - The Divergence Loss skirts intractability
        - By constraining the Encoder to produce a tractable distribution

We can state our goal as

- **Producing $\hat{\mathbf{X}}^{(i')}$ that comes from the same distribution as training examples $\mathbf{X}$**
- **But are non-deterministic**
    - **some $\hat{\mathbf{X}}^{(i')}$ are not exactly equal to $\mathbf{X}^{(\mathrm{i})}$ for any $1 \leq i \leq m$**

That is: we want to generate new examples that are similar (but not identical) to the training examples $\mathbf{X}$.

Let $p(\mathbf{X})$ denote the probability distribution of training examples $\mathbf{X}$.

Note that $p(\mathbf{X})$ is an *empirical* distribution define by the finite set $\mathbf{X}$

- We do not have a *closed form* expression for the distribution

One way to conceive of achieving the goal is to condition $p(X)$ on a random variable $\mathbf{z}$ and generate and synthetic example based on the random value

$$p(\mathbf{X}|\mathbf{z})$$

We will use a Neural Network with weights Θ to compute

$$p_{\Theta}(\mathbf{X}|\mathbf{z})$$

Let's try to create an optimization objective function against which we choose our model weights.

We will use Maximum Likelihood as the objective

- **Given the weights: how likely is the model to produce the training distribution $\mathbf{X}$?**
- **Recall that the likelihood of the set $\mathbf{X}$ is the product of the probabilities that the model produces each $\mathbf{x} \in \mathbf{X}$**
- **So the log likehood is the sum of the log probabilities**

Since our practice is to minimize Loss (rather than maximize an objective function) we write our loss as (negative of log) likelihood

$$\mathcal{L} = -\log(p(\mathbf{X}))$$

Minimizing $\mathcal{L}$ is equivalent to maximizing likelihood.

Because the generation of $\hat{\mathbf{x}}$ depends on a random variable $\mathbf{z}$ we marginalize $\mathbf{x}$ over $\mathbf{z}$

$$p(\mathbf{x}) = \int_{\mathbf{z} \in Q} p(\mathbf{x}|\mathbf{z})\, p(z)$$

where random $\mathbf{z}$ comes from (as of yet unknown) distribution Q.

That is: there are potentially many (and at least one) random choice that produces approximations of any training example.

We want to train a Neural Network to produce randomized examples that are similar to examples in $\mathbf{X}$.

This means that, for each example in $\mathbf{X}$, we have to sample from $p(\mathbf{z})$

Some obvious concerns

- **It may be very expensive to draw many samples from $\mathbf{z}$ for each training $\mathbf{x}$**
- **Moreover: It is likely that that there are many random choices from $\mathbf{z}$**
  - **for which the generated $p_{\Theta}(\mathbf{x}|\mathbf{z})$**
  - **is *unlike* and example in $\mathbf{X}$**

Let's address these concerns by considering the joint distribution of $\mathbf{X}$ and $\mathbf{z}$

$$p(\mathbf{x}, \mathbf{z})$$

(from which we can compute $p(\mathbf{z}|\mathbf{x})$ via Bayes formula)

We can improve our sampling by considering only those choices of $\mathbf{z}$ that could generate $\mathbf{x}$ and re-write the objective as

$$p(\mathbf{x}) = \int_{\mathbf{z} \in p(\mathbf{z}|\boldsymbol{x})} p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})$$

The problem is that $p(\mathbf{z}|\mathbf{x})$ is intractable !

- The examples in $\mathbf{X}$ are *deterministic*: they were not produced using a random $\mathbf{z}$
  - So we have *no data* on which to infer a joint distribution
- So we don't know the joint distribution $p(\mathbf{x}, \mathbf{z})$ or the conditional distribution $p(\mathbf{z}|\mathbf{x})$

The solution is *approximate* the intractable $p(\mathbf{z}|\mathbf{x})$ with a tractable $q_\Phi(\mathbf{z}|\mathbf{x})$

- **That is computed by a Neural Network (the "Encoder")**
- **That is parameterized by Φ**

We use KL divergence as a measure of the difference between two distributions.

So we want to minimize

$$\mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x}))$$

VAE derivation: 2

Note that the Encoder in the VAE arises out of necessity

- **the need to compute $p(\mathbf{z}|\mathbf{x})$**
- **rather than a priori design considerations**

We add the KL divergence to our loss function

$$\begin{aligned}\mathcal{L} &= -\log(p_\Theta(\mathbf{x})) + \mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})) \\ &= \mathcal{L}_R + \mathcal{L}_D\end{aligned}$$

which now has two objectives

- Reconstruction loss $\mathcal{L}_R$: maximize the likelihood (by minimizing the negative likelihood)
- Divergence constraint $\mathcal{L}_D$: $q_\phi(\mathbf{z}|\mathbf{x})$ must be close to $p_\theta(\mathbf{z}|\mathbf{x}))$

$$\begin{aligned}\mathcal{L}_R &= -\log(p_\theta(\mathbf{x})) \\ \mathcal{L}_D &= \mathbf{KL}\left(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})\right)\end{aligned}$$

That is, our new loss $\mathcal{L}$ function has two components

- $\mathcal{L}_R$
  - **the reconstruction loss, as before**
- $\mathcal{L}_D$
  - **the "KL divergence" loss which constrains the approximate $q_\phi(\mathbf{z}|\mathbf{x})$**

We will show (in the next section: lots of algebra !) that the loss can be re-written as

$$\mathcal{L} = -\mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\left(\log(p_\Theta(\mathbf{x}|\mathbf{z}))\right) + \mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}))$$

This is *almost* identical to our original express for $\mathcal{L}$ except

- **Re-write**
  $\log(p_\theta(\mathbf{x})) = \mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\left(\log(p_\Theta(\mathbf{x}|\mathbf{z}))\right)$
- **the KL term becomes**
  $$\mathbf{KL}\left(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z})\right)$$
  rather than the original
  $$\mathbf{KL}\left(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})\right)$$

The purpose of re-writing: replace intractable $p(\mathbf{z}|\mathbf{x})$ with a tractable $p(\mathbf{z})$!

- **So we can an Loss function with which we can train !**

Advanced: Obtain $\mathcal{L}$ by rewriting $\mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})$

Let's derive a simpler expression for $\mathcal{L}$ by manipulating $\mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p_\Theta(\mathbf{z}|\mathbf{x}))$:

$$
\begin{aligned}
\mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})) &= \textstyle\sum_z q_\Phi(\mathbf{z}|\mathbf{x})(\log(q_\Phi(\mathbf{z}|\mathbf{x}) - \log(p(\mathbf{z}|\mathbf{x})) \\
&= \mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\left(\log(q_\Phi(\mathbf{z}|\mathbf{x}) - \log(p(\mathbf{z}|\mathbf{x}))\right) \\
&= \mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\big(\ \log(q_\Phi(\mathbf{z}|\mathbf{x})) \\
&\quad - (\ \log(p(\mathbf{x}|\mathbf{z})) + \log(p(\mathbf{z})) - \log(p(\mathbf{x}))\ )\ \ \big)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})) \\
&- \log(p(\mathbf{x})) &= \mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\big(\ \log(q_\Phi(\mathbf{z}|\mathbf{x})) - (\log(p_\Theta(\mathbf{x}|\mathbf{z})) + \log(p \\
&&= \mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\big(\ -\log(p_\Theta(\mathbf{x}|\mathbf{z})) + (\ \log(q_\Phi(\mathbf{z}|\mathbf{x})) - \log \\
&&= -\mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\left(\log(p_\Theta(\mathbf{x}|\mathbf{z}))\right) + \mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z})) \\
&\mathcal{L} &= -\mathbb{E}_{z\sim q_\Phi(\mathbf{z}|\mathbf{x})}\left(\log(p_\Theta(\mathbf{x}|\mathbf{z}))\right) + \mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}))
\end{aligned}
$$

The LHS cannot be optimized via SGD (recall the tractability issue with $p(\mathbf{z}|\mathbf{x})$).

But the RHS can be made tractable giving a tractable choice of $p(\mathbf{z})$.

Choosing $p(\mathbf{z})$

So what distribution should we use for the prior $p(\mathbf{z})$?

- **It should be differentiable, since we use Gradient Descent for optimization**
- **It should be tractable with a closed form (such as a normal)**
- **If we choose $p(\mathbf{z})$ as normal, we can require $q_\phi(\mathbf{z}|\mathbf{x})$ to be normal too**
    - **The KL divergence between two normals is an easy to compute function of their means and standard deviations.**
    - **See [VAE tutorial (https://arxiv.org/pdf/1606.05908.pdf)](https://arxiv.org/pdf/1606.05908.pdf) Section 2.2**

Re-parameterization trick

There is still one impediment to training.

It involves the random choice of $z \sim q_\Phi(\mathbf{z}|\mathbf{x})$ in

$$\mathcal{L}_R = \mathbb{E}_{z \sim q_\Phi(\mathbf{z}|\mathbf{x})}\left(\log(p_\Theta(\mathbf{x}|\mathbf{z}))\right)$$

This is not a problem in the forward pass.

But in the backward pass we need to compute

$$\frac{\mathcal{L}_R}{\partial \Theta}$$

How do we back propagte through a random choice ?

The "reparameterization trick" redefines the random choice $\mathbf{z}$ as

$$\begin{aligned} \mathbf{z} &= \mu_\theta(\mathbf{x}) + \sigma_\theta(\mathbf{x}) * \epsilon \\ \epsilon &\sim p(\mathbf{z}) \end{aligned}$$

That is

- $\mathbf{z}$ has been made a function of (trainable parameters) $\mu_\Theta, \sigma_\Theta$
- the random variable ϵ appears in a product term
    - we can differentiate the product with respect to Θ
    - ϵ can be treated as a constant in $\frac{\partial \epsilon}{\partial \Theta}$

Reparameterization trick

This gets us to the final picture of the VAE:

Variational Autoencoder (VAE)

ELBo (Evidence-based Lower Bound)

By re-writing the Loss, we removed the intractable term $p(\mathbf{z}|\mathbf{x})$

It turns out that even this may not be necessary.

For the truly interested reader:

- **The derivation uses a method known as *Variational Inference*. See this blog (https://mbernste.github.io/posts/variational_inference/) for a summary.**
- **One can show that loss $\mathcal{L}$ is equal to -1 times the *ELBo* (Evidence Based Lower Bound)**

So if one knows how to maximize the ELBo (https://mbernste.github.io/posts/elbo/), one can minimize the loss.

Loss function: discussion

Let's examine the role of $\mathcal{L}_R$ and $\mathcal{L}_D$ in the loss function $\mathcal{L}$.

- What would happen if we dropped $\mathcal{L}_D$?
  - We would wind up with a deterministic $\mathbf{z}$ and collapse to a vanilla VAE
- What would happen if we dropped $\mathcal{L}_R$?
  - the encoding approximation $q_\Phi(\mathbf{z}|\mathbf{x})$ would be close to the empirical $p(\mathbf{z}|\mathbf{x})$ *in distribution*
  - but two variables with the same distribution are not necessarily the same ?
    - e.g., get a distribution p by flipping a coin
      - let distribution q be a relabelling of p by changing Heads to Tails and vice-versa
      - p and q are equal in distribution but clearly different !

In [3]:
```
print("Done")
```

```
Done
```