towards
data science

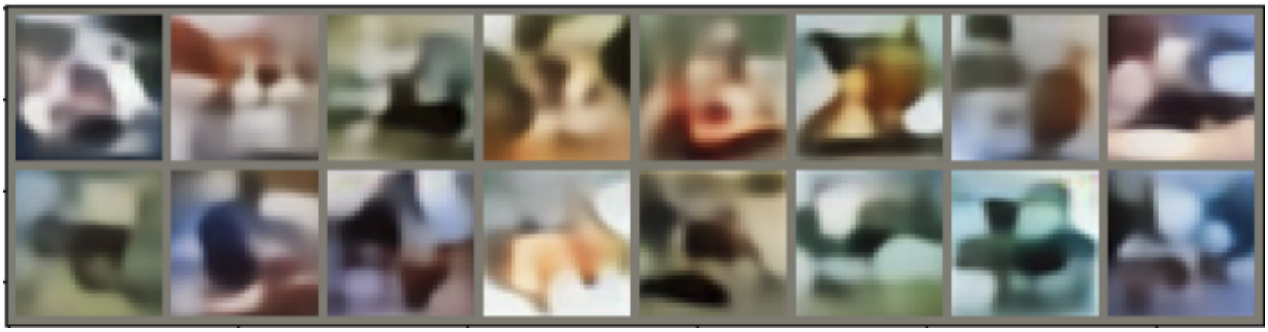
Follow 522K Followers

HANDS-ON TUTORIALS

Variational Autoencoder Demystified With PyTorch Implementation.

This tutorial implements a variational autoencoder for non-black and white images using PyTorch.

William Falcon Dec 5, 2020 · 9 min read



Generated images from cifar-10 (author's own)

It's likely that you've searched for VAE tutorials but have come away empty-handed. Either the tutorial uses MNIST instead of color images or the concepts are conflated and not explained clearly.

You're in luck!

This tutorial covers all aspects of VAEs including the matching math and implementation on a realistic dataset of color images.

1. Resources (github code, colab).
2. ELBO definition (optional).
3. ELBO, KL divergence explanation (optional).
4. ELBO, reconstruction loss explanation (optional).
5. PyTorch implementation

. . .

Resources

Follow along with this colab.

Code is also available on Github here (don't forget to star!).

For a production/research-ready implementation simply install pytorch-lightning-bolts

```
pip install pytorch-lightning-bolts
```

and import and use/subclass

```
from pl_bolts.models.autoencoders import VAE

model = VAE()
trainer = Trainer()
trainer.fit(model)
```

. . .

ELBO loss

In this section, we'll discuss the VAE loss. If you don't care for the math, feel free to skip this section!

distribution you want like Normal, etc… In this tutorial, we don't specify what these are to keep things easier to understand.

So, when you see p, or q, just think of a blackbox that is a distribution. Don't worry about what is in there.

VAE loss: The loss function for the VAE is called the ELBO. The ELBO looks like this:

ELBO loss — Red=KL divergence. Blue = reconstruction loss. (Author's own).

The first term is the KL divergence. The second term is the reconstruction term.

Confusion point 1 MSE: Most tutorials equate reconstruction with MSE. But this is misleading because MSE only works when you use certain distributions for p, q.

Confusion point 2 KL divergence: Most other tutorials use p, q that are normal. If you assume p, q are Normal distributions, the KL term looks like this (in code):

```
kl = torch.mean(-0.5 * torch.sum(1 + log_var - mu ** 2 -
log_var.exp(), dim = 1), dim = 0)
```

But in our equation, we DO NOT assume these are normal. We do this because it makes things much easier to understand and keeps the implementation general so you can use any distribution you want.

Let's break down each component of the loss to understand what each is doing.

. . .

ELBO Loss — KL Divergence term

Let's first look at the KL divergence term.

The first part (min) says that we want to minimize this. Next to that, the E term stands for *expectation* under q. This means we draw a sample (z) from the q distribution.

Notice that in this case, I used a Normal(0, 1) distribution for q. When we code the loss, we have to specify the distributions we want to use.

Now that we have a sample, the next parts of the formula ask for two things: 1) the log probability of z under the q distribution, 2) the log probability of z under the p distribution.

Notice that z has almost **zero** probability of having come from p. But has 6% probability of having come from q. If we visualize this it's clear why:

z has a value of 6.0110. If you look at the area of q where z is (ie: the probability), it's clear that there is a non-zero chance it came from q. But, if you look at p, there's basically a zero chance that it came from p.

If we look back at this part of the loss

You can see that we are **minimizing** the difference between these probabilities.

So, to maximize the probability of z under p, we have to shift q closer to p, so that when we sample a new z from **q**, that value will have a much higher probability.

and now our new kl divergence is:

Now, this z has a single dimension. But in the real world, we care about n-dimensional zs. To handle this in the implementation, we simply sum over the last dimension. The trick here is that when sampling from a univariate distribution (in this case Normal), if you sum across many of these distributions, it's equivalent to using an n-dimensional distribution (n-dimensional Normal in this case).

Here's the kl divergence that is *distribution agnostic* in PyTorch.

This generic form of the KL is called the monte-carlo approximation. This means we sample z many times and estimate the KL divergence. (in practice, these estimates are really good and with a batch size of 128 or more, the estimate is very accurate).

. . .

ELBO loss — Reconstruction term

The second term we'll look at is the reconstruction term.

In the KL explanation we used p(z), q(z|x). For this equation, we need to define a third distribution, P_rec(x|z). To avoid confusion we'll use P_rec to differentiate.

Tip: DO NOT confuse P_rec(x|z) and p(z).

So, in this equation we again sample z from q. But now we use that z to calculate the probability of seeing the input x (ie: a color image in this case) given the z that we sampled.

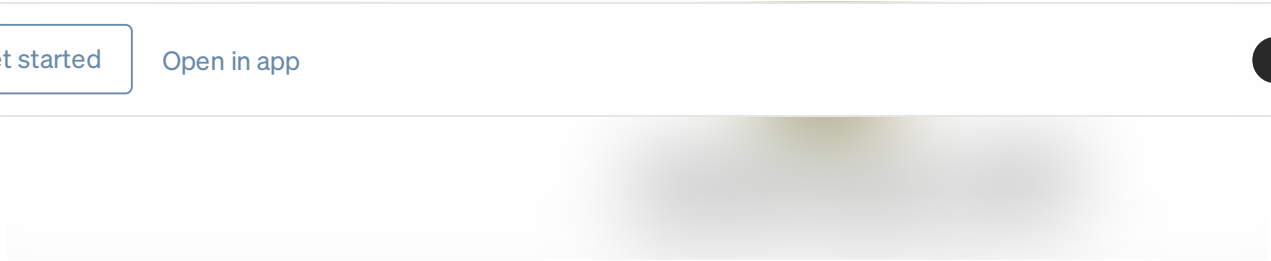
First we need to think of our images as having a distribution in image space. Imagine a very high dimensional distribution. For a color image that is 32x32 pixels, that means this distribution has (3x32x32 = 3072) dimensions.

So, now we need a way to map the z vector (which is low dimensional) back into a super high dimensional distribution from which we can measure the probability of seeing this particular image. In VAEs, we use a *decoder* for that.

Confusion point 3: Most tutorials show x_hat as an image. However, this is wrong. x_hat **IS NOT** an image. These are **PARAMETERS** for a distribution. But because these tutorials use MNIST, the output is already in the zero-one range and **can** be interpreted as an image. But with color images, this is not true.

To finalize the calculation of this formula, we use x_hat to parametrize a likelihood distribution (in this case a normal again) so that we can measure the probability of the input (image) under this high dimensional distribution.

ie: we are asking the same question: Given P_rec(x|z) and this image, what is the probability?



Since the reconstruction term has a negative sign in front of it, we minimize it by **maximizing** the probability of this image under P_rec(x|z).

. . .

ELBO summary

Some things may not be obvious still from this explanation. First, **each** image will end up with its own q. The KL term will push all the qs towards the **same** p (called the prior). But if all the qs, collapse to p, then the network can cheat by just mapping everything to zero and thus the VAE will collapse.

The reconstruction term, forces each q to be unique and spread out so that the image can be reconstructed correctly. This keeps all the qs from **collapsing** onto each other.

As you can see, both terms provide a nice balance to each other. This is also why you may experience instability in training VAEs!

. . .

PyTorch Implementation

Now that you understand the intuition behind the approach and math, let's code up the VAE in PyTorch. For this implementation, I'll use PyTorch Lightning which will

If you skipped the earlier sections, recall that we are now going to implement the following VAE loss:

This equation has 3 distributions. Our code will be agnostic to the distributions, but we'll use Normal for all of them.

The first distribution: q(z|x) needs parameters which we generate via an encoder.

The second distribution: p(z) is the **prior** which we will fix to a specific location (0,1). By fixing this distribution, the KL divergence term will force **q(z|x)** to move closer to **p** by updating the parameters.

The optimization start out with two distributions like this (q, p).

and over time, moves q closer to p (p is fixed as you saw, and q has learnable parameters).

The third distribution: $p(x|z)$ (usually called the reconstruction), will be used to measure the probability of seeing the image (input) given the z that was sampled.

Think about this image as having 3072 dimensions (3 channels x 32 pixels x 32 pixels).

So, we can now write a full class that implements this algorithm.

What's nice about Lightning is that all the hard logic is encapsulated in the training_step. This means everyone can know exactly what something is doing when it is written in Lightning by looking at the training_step.

Data: The Lightning VAE is fully decoupled from the data! This means we can train on imagenet, or whatever you want. For speed and cost purposes, I'll use cifar-10 (a much smaller image dataset).

Lightning uses regular pytorch dataloaders. But it's annoying to have to figure out transforms, and other settings to get the data in usable shape. For this, we'll use the optional abstraction (Datamodule) which abstracts all this complexity from me.

And we'll see training start…

Even just after 18 epochs, I can look at the reconstruction.

Even though we didn't train for long, and used no fancy tricks like perceptual losses, we get something that kind of looks like samples from CIFAR-10.

. . .

Next post

In the next post, I'll cover the derivation of the ELBO!

williamFalcon/pytorch-lightning-vae

Dismiss GitHub is home to over 50 million developers working together to host and review code, manage projects, and...

github.com

Deep Learning | Data Science | Machine Learning | Hands On Tutorials

Finding it difficult to learn programming? Here's why.

Natassha Selvaraj in Towards Data Science

Apple's New M1 Chip is a Machine Learning Beast

Daniel Bourke in Towards Data Science

A Complete 52 Week Curriculum to Become a Data Scientist in 2021

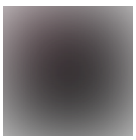
Terence Shin in Towards Data Science

How to Become Fluent in Multiple Programming Languages

Madison Hunter in Towards Data Science

Three Functions to Know in Python

Luay Matalka in Towards Data Science



10 Must-Know Statistical Concepts for Data Scientists

Soner Yıldırım in Towards Data Science

How to create dashboard for free with Google Sheets and Chart.js

Rafał Rybnik in Towards Data Science

Pylance: The best Python extension for VS Code

Dimitris Poulopoulos in Towards Data Science