Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

Department of Physics and Center for Computing in Science Education, University of Oslo, Norway[1]

Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA[2]

April 16, 2024

Plans for the week April 15-19, 2024

Deep generative models

1. Finalizing discussion of Boltzmann machines, implementations using TensorFlow and Pytorch
2. Discussion of other energy-based models and Langevin sampling
3. Variational Autoencoders (VAE)
4. Generative Adversarial Networks (GANs)

Readings

1. Reading recommendation: Goodfellow et al, for VAEs and GANs see sections 20.10-20.11
2. To create Boltzmann machine using Keras, see Babcock and Bali chapter 4, see `https://github.com/PacktPublishing/Hands-On-Generative-AI-with-Python-and-TensorFlow-2/blob/master/Chapter_4/models/rbm.py`
3. See Foster, chapter 7 on energy-based models at `https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/07_ebm/01_ebm`

Reminder from last week and layout of lecture this week

1. We will present first a short reminder from last week, see for example the jupyter-notebook at https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week12/ipynb/week12.ipynb
2. We will then discuss codes as well as other energy-based models and Langevin sampling instead of Gibbs or Metropolis sampling.
3. Thereafter we start our discussions of Variational autoencoders and Generalized adversarial networks

Code for RBMs using PyTorch

```
import numpy as np
import torch
import torch.utils.data
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torchvision import datasets, transforms
from torchvision.utils import make_grid , save_image
import matplotlib.pyplot as plt


batch_size = 64
train_loader = torch.utils.data.DataLoader(
datasets.MNIST('./data',
    train=True,
    download = True,
    transform = transforms.Compose(
        [transforms.ToTensor()])
     ),
     batch_size=batch_size
)

test_loader = torch.utils.data.DataLoader(
datasets.MNIST('./data',
    train=False,
    transform=transforms.Compose(
    [transforms.ToTensor()])
```

RBM using TensorFlow and Keras

1. To create Boltzmann machine using Keras, see Babcock and Bali chapter 4, see `https://github.com/PacktPublishing/Hands-On-Generative-AI-with-Python-and-TensorFlow-2/blob/master/Chapter_4/models/rbm.py`

Energy-based models and Langevin sampling

See discussions in Foster, chapter 7 on energy-based models at https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/07_ebm/01_ebm
That notebook is based on a recent article by Du and Mordatch, **Implicit generation and modeling with energy-based models**, see https://arxiv.org/pdf/1903.08689.pdf.

Langevin sampling

Also called Stochastic gradient Langevin dynamics (SGLD), is sampling technique composed of characteristics from Stochastic gradient descent (SGD) and Langevin dynamics, a mathematical extension of the Langevin equation. The SGLD is an iterative optimization algorithm which uses minibatching to create a stochastic gradient estimator, as used in SGD to optimize a differentiable objective function.
Unlike traditional SGD, SGLD can be used for Bayesian learning as a sampling method. SGLD may be viewed as Langevin dynamics applied to posterior distributions, but the key difference is that the likelihood gradient terms are minibatched, like in SGD. SGLD, like Langevin dynamics, produces samples from a posterior distribution of parameters based on available data.

More on the SGLD

The SGLD uses the probability $p(\theta)$ (note that we limit ourselves to just a variable θ) and updates the **log** of this probability by initializing it through some random prior distribution, normally just a uniform distribution which takes values between $\theta \in [-1, 1]$, The update is given by

$$\theta_{i+1} = \theta_i + \eta \nabla_\theta \log p(\theta_i) + \sqrt{\eta} w_i,$$

where $w_i \sim N(0, 1)$ are normally distributed with mean zero and variance one and $i = 0, 1, \ldots, k$, with k the final number of iterations. The parameter η is the learning rate. The term $\sqrt{\eta} w_i$ introduces **noise** in the equation.

Code example of Langevin Samplig

In our calculations the gradient is calculated using the model we have for the probability distribution. For an energy-based model this gives us a derivative which involves the so-called positive and negative phases discussed last week.
Read more about Langevin sampling at for example `https://www.lyndonduong.com/sgmcmc/`. This site contains a nice example of a PyTorch code which implements Langevin sampling.

Theory of Variational Autoencoders

Let us remind ourself about what an autoencoder is, see the jupyter-notebook at `https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week10/ipynb/week10.ipynb`.

The Autoencoder again

Autoencoders are neural networks where the outputs are its own inputs. They are split into an **encoder part** which maps the input $\boldsymbol{x}$ via a function $f(\boldsymbol{x}, \boldsymbol{W})$ (this is the encoder part) to a **so-called code part** (or intermediate part) with the result $\boldsymbol{h}$

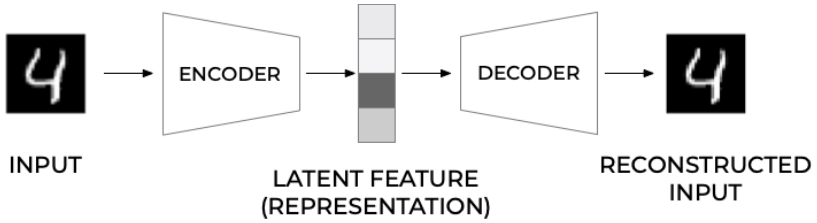
$$\boldsymbol{h} = f(\boldsymbol{x}, \boldsymbol{W})),$$

where $\boldsymbol{W}$ are the weights to be determined. The **decoder** parts maps, via its own parameters (weights given by the matrix $\boldsymbol{V}$ and its own biases) to the final ouput

$$\tilde{\boldsymbol{x}} = g(\boldsymbol{h}, \boldsymbol{V})).$$

The goal is to minimize the construction error, often done by optimizing the means squared error.

Schematic image of an Autoencoder



Mathematics of Variational Autoencoders

We have defined earlier a probability (marginal) distribution with hidden variables $\boldsymbol{h}$ and parameters $\boldsymbol{\Theta}$ as

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \int d\boldsymbol{h} p(\boldsymbol{x},\boldsymbol{h};\boldsymbol{\Theta}),$$

for continuous variables $\boldsymbol{h}$ and

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \sum_{\boldsymbol{h}} p(\boldsymbol{x},\boldsymbol{h};\boldsymbol{\Theta}),$$

for discrete stochastic events $\boldsymbol{h}$. The variables $\boldsymbol{h}$ are normally called the **latent variables** in the theory of autoencoders. We will also call then for that here.

Using the conditional probability

Using the the definition of the conditional probabilities $p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})$, $p(\boldsymbol{h}|\boldsymbol{x};\boldsymbol{\Theta})$ and and the prior $p(\boldsymbol{h})$, we can rewrite the above equation as

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \sum_{\boldsymbol{h}} p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})p(\boldsymbol{h},$$

which allows us to make the dependence of $\boldsymbol{x}$ on $\boldsymbol{h}$ explicit by using the law of total probability. The intuition behind this approach for finding the marginal probability for $\boldsymbol{x}$ is to optimize the above equations with respect to the parameters $\boldsymbol{\Theta}$. This is done normally by maximizing the probability, the so-called maximum-likelihood approach discussed earlier.

VAEs versus autoencoders

This trained probability is assumed to be able to produce similar samples as the input. In VAEs it is then common to compare via for example the mean-squared error or the cross-entropy the predicted values with the input values. Compared with autoencoders, we are now producing a probability instead of a functions which mimicks the input.
In VAEs, the choice of this output distribution is often Gaussian, meaning that the conditional probability is

$$p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta}) = N(\boldsymbol{x}|f(\boldsymbol{h};\boldsymbol{\Theta}), \sigma^2 \times \boldsymbol{I}),$$

with mean value given by the function $f(\boldsymbol{h};\boldsymbol{\Theta})$ and a diagonal covariance matrix multiplied by a parameter σ^2 which is treated as a hyperparameter.

Gradient descent

By having a Gaussian distribution, we can use gradient descent (or any other optimization technique) to increase $p(\boldsymbol{x}; \boldsymbol{\Theta})$ by making $f(\boldsymbol{h}; \boldsymbol{\Theta})$ approach $\boldsymbol{x}$ for some $\boldsymbol{h}$, gradually making the training data more likely under the generative model. The important property is simply that the marginal probability can be computed, and it is continuous in $\boldsymbol{\Theta}$..

Are VAEs just modified autoencoders?

The mathematical basis of VAEs actually has relatively little to do with classical autoencoders, for example the sparse autoencoders or denoising autoencoders discussed earlier.
VAEs approximately maximize the probability equation discussed above. They are called autoencoders only because the final training objective that derives from this setup does have an encoder and a decoder, and resembles a traditional autoencoder. Unlike sparse autoencoders, there are generally no tuning parameters analogous to the sparsity penalties. And unlike sparse and denoising autoencoders, we can sample directly from $p(\boldsymbol{x})$ without performing Markov Chain Monte Carlo.

Training VAEs

To solve the integral or sum for $p(\boldsymbol{x})$, there are two problems that VAEs must deal with: how to define the latent variables $\boldsymbol{h}$, that is decide what information they represent, and how to deal with the integral over $\boldsymbol{h}$. VAEs give a definite answer to both.

Kullback-Leibler relative entropy (notation to be updated)

When the goal of the training is to approximate a probability distribution, as it is in generative modeling, another relevant measure is the **Kullback-Leibler divergence**, also known as the relative entropy or Shannon entropy. It is a non-symmetric measure of the dissimilarity between two probability density functions p and q. If p is the unkown probability which we approximate with q, we can measure the difference by

$$\mathrm{KL}(p||q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x}.$$

Kullback-Leibler divergence

Thus, the Kullback-Leibler divergence between the distribution of the training data $f(\boldsymbol{x})$ and the model distribution $p(\boldsymbol{x}|\boldsymbol{\Theta})$ is

$$\begin{aligned}
\mathrm{KL}(f(\boldsymbol{x})||p(\boldsymbol{x}|\boldsymbol{\Theta})) &= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{\Theta})} d\boldsymbol{x} \\
&= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log f(\boldsymbol{x}) d\boldsymbol{x} - \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log p(\boldsymbol{x}|\boldsymbol{\Theta}) d\boldsymbol{x} \\
&= \langle \log f(\boldsymbol{x}) \rangle_{f(\boldsymbol{x})} - \langle \log p(\boldsymbol{x}|\boldsymbol{\Theta}) \rangle_{f(\boldsymbol{x})} \\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \langle E(\boldsymbol{x}) \rangle_{data} + \log Z \\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \mathcal{C}_{LL}.
\end{aligned}$$

Maximizing log-likelihood

The first term is constant with respect to $\mathbf{\Theta}$ since $f(\boldsymbol{x})$ is independent of $\mathbf{\Theta}$. Thus the Kullback-Leibler Divergence is minimal when the second term is minimal. The second term is the log-likelihood cost function, hence minimizing the Kullback-Leibler divergence is equivalent to maximizing the log-likelihood.
To further understand generative models it is useful to study the gradient of the cost function which is needed in order to minimize it using methods like stochastic gradient descent.

More on the partition function

The partition function is the generating function of expectation values, in particular there are mathematical relationships between expectation values and the log-partition function. In this case we have

$$\langle \frac{\partial E(\boldsymbol{x};\Theta_i)}{\partial \Theta_i} \rangle_{model} = \int p(\boldsymbol{x}|\boldsymbol{\Theta}) \frac{\partial E(\boldsymbol{x};\Theta_i)}{\partial \Theta_i} d\boldsymbol{x} = -\frac{\partial \log Z(\Theta_i)}{\partial \Theta_i}.$$

Here $\langle \cdot \rangle_{model}$ is the expectation value over the model probability distribution $p(\boldsymbol{x}|\boldsymbol{\Theta})$.

Setting up for gradient descent calculations

Using the previous relationship we can express the gradient of the cost function as

$$
\begin{aligned}
\frac{\partial \mathcal{C}_{LL}}{\partial \Theta_i} =& \langle \frac{\partial E(\boldsymbol{x};\Theta_i)}{\partial \Theta_i} \rangle_{data} + \frac{\partial \log Z(\Theta_i)}{\partial \Theta_i} \\
=& \langle \frac{\partial E(\boldsymbol{x};\Theta_i)}{\partial \Theta_i} \rangle_{data} - \langle \frac{\partial E(\boldsymbol{x};\Theta_i)}{\partial \Theta_i} \rangle_{model}
\end{aligned}
$$

Difference of moments

This expression shows that the gradient of the log-likelihood cost function is a **difference of moments**, with one calculated from the data and one calculated from the model. The data-dependent term is called the **positive phase** and the model-dependent term is called the **negative phase** of the gradient. We see now that minimizing the cost function results in lowering the energy of configurations $\boldsymbol{x}$ near points in the training data and increasing the energy of configurations not observed in the training data. That means we increase the model's probability of configurations similar to those in the training data.

More observations

The gradient of the cost function also demonstrates why gradients of unsupervised, generative models must be computed differently from for those of for example FNNs. While the data-dependent expectation value is easily calculated based on the samples $\boldsymbol{x}_i$ in the training data, we must sample from the model in order to generate samples from which to caclulate the model-dependent term. We sample from the model by using MCMC-based methods. We can not sample from the model directly because the partition function Z is generally intractable.

Adding hyperparameters

As in supervised machine learning problems, the goal is also here to perform well on **unseen** data, that is to have good generalization from the training data. The distribution $f(x)$ we approximate is not the **true** distribution we wish to estimate, it is limited to the training data. Hence, in unsupervised training as well it is important to prevent overfitting to the training data. Thus it is common to add regularizers to the cost function in the same manner as discussed for say linear regression.

Using the KL divergence

In practice, for most $\boldsymbol{h}$, $p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})$ will be nearly zero, and hence contribute almost nothing to our estimate of $p(\boldsymbol{x})$.
The key idea behind the variational autoencoder is to attempt to sample values of $\boldsymbol{h}$ that are likely to have produced $\boldsymbol{x}$, and compute $p(\boldsymbol{x})$ just from those.
This means that we need a new function $Q(\boldsymbol{h}|\boldsymbol{x})$ which can take a value of $\boldsymbol{x}$ and give us a distribution over $\boldsymbol{h}$ values that are likely to produce $\boldsymbol{x}$. Hopefully the space of $\boldsymbol{h}$ values that are likely under Q will be much smaller than the space of all $\boldsymbol{h}$'s that are likely under the prior $p(\boldsymbol{h})$. This lets us, for example, compute $E_{\boldsymbol{h}\sim Q}p(\boldsymbol{x}|\boldsymbol{h})$ relatively easily. Note that we drop $\boldsymbol{\Theta}$ from here and for notational simplicity.

Kullback-Leibler again

However, if $\boldsymbol{h}$ is sampled from an arbitrary distribution with PDF $Q(\boldsymbol{h})$, which is not $\mathcal{N}(0, I)$, then how does that help us optimize $p(\boldsymbol{x})$?
The first thing we need to do is relate $E_{\boldsymbol{h}\sim Q}P(\boldsymbol{x}|\boldsymbol{h})$ and $p(\boldsymbol{x})$. We will see where Q comes from later.
The relationship between $E_{\boldsymbol{h}\sim Q}p(\boldsymbol{x}|\boldsymbol{h})$ and $p(\boldsymbol{x})$ is one of the cornerstones of variational Bayesian methods. We begin with the definition of Kullback-Leibler divergence (KL divergence or $\mathcal{D}$) between $p(\boldsymbol{h}|\boldsymbol{x})$ and $Q(\boldsymbol{h})$, for some arbitrary Q (which may or may not depend on $\boldsymbol{x}$):

$$\mathcal{D}\left[Q(z)\|P(z|X)\right] = E_{z\sim Q}\left[\log Q(z) - \log P(z|X)\right].$$

And applying Bayes rule

We can get both $p(\boldsymbol{x})$ and $p(\boldsymbol{x}|\boldsymbol{h})$ into this equation by applying Bayes rule to $p(\boldsymbol{h}|\boldsymbol{x})$

$$\mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{x}|\boldsymbol{h}) - \log p(\boldsymbol{h})\right] + \log p(\boldsymbol{x}).$$

Here, $\log p(\boldsymbol{x})$ comes out of the expectation because it does not depend on $\boldsymbol{h}$. Negating both sides, rearranging, and contracting part of $E_{\boldsymbol{h}\sim Q}$ into a KL-divergence terms yields:

$$\log p(\boldsymbol{x}) - \mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{x}|\boldsymbol{h})\right] - \mathcal{D}\left[Q(\boldsymbol{h})\|P(\boldsymbol{h})\right].$$

Rearraning

Using Bayes rule we obtain

$$E_{\boldsymbol{h}\sim Q}\left[\log p(y_i|\boldsymbol{h}, x_i)\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{h}|y_i, x_i) - \log p(\boldsymbol{h}|x_i) + \log p(y_i|x_i)\right]$$

Rearranging the terms and subtracting $E_{\boldsymbol{h}\sim Q} \log Q(\boldsymbol{h})$ from both sides gives

$$\begin{aligned} \log P(y_i|x_i) - E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|x_i, y_i)\right] = \\ E_{\boldsymbol{h}\sim Q}\left[\log p(y_i|\boldsymbol{h}, x_i) + \log p(\boldsymbol{h}|x_i) - \log Q(\boldsymbol{h})\right] \end{aligned}$$

Note that $\boldsymbol{x}$ is fixed, and Q can be *any* distribution, not just a distribution which does a good job mapping $\boldsymbol{x}$ to the $\boldsymbol{h}$'s that can produce X.

Inferring the probability

Since we are interested in inferring $p(\boldsymbol{x})$, it makes sense to construct a Q which *does* depend on $\boldsymbol{x}$, and in particular, one which makes $\mathcal{D}\left[Q(\boldsymbol{h}) \| p(\boldsymbol{h}|\boldsymbol{x})\right]$ small

$$\log p(\boldsymbol{x}) - \mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x}) \| p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{x}|\boldsymbol{h})\right] - \mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x}) \| p(\boldsymbol{h})\right].$$

Hence, during training, it makes sense to choose a Q which will make $E_{\boldsymbol{h}\sim Q}[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|x_i, y_i)]$ (a $\mathcal{D}$-divergence) small, such that the right hand side is a close approximation to $\log p(y_i|y_i)$.

Central equation of VAEs

This equation serves as the core of the variational autoencoder, and it is worth spending some time thinking about what it means.

1. The left hand side has the quantity we want to maximize, namely $\log p(\boldsymbol{x})$ plus an error term.
2. The right hand side is something we can optimize via stochastic gradient descent given the right choice of Q.

Setting up SGD

So how can we perform stochastic gradient descent?
First we need to be a bit more specific about the form that $Q(\boldsymbol{h}|\boldsymbol{x})$ will take. The usual choice is to say that $Q(\boldsymbol{h}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{h}|\mu(\boldsymbol{x};\vartheta), \Sigma(;\vartheta))$, where μ and Σ are arbitrary deterministic functions with parameters ϑ that can be learned from data (we will omit ϑ in later equations). In practice, μ and Σ are again implemented via neural networks, and Σ is constrained to be a diagonal matrix.

More on the SGD

The name variational "autoencoder" comes from the fact that μ and Σ are "encoding" $\boldsymbol{x}$ into the latent space $\boldsymbol{h}$. The advantages of this choice are computational, as they make it clear how to compute the right hand side. The last term—$\mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x})\|p(\boldsymbol{h})\right]$—is now a KL-divergence between two multivariate Gaussian distributions, which can be computed in closed form as:

$$\mathcal{D}[\mathcal{N}(\mu_0,\Sigma_0)\|\mathcal{N}(\mu_1,\Sigma_1)] =$$
$$\frac{1}{2}\Big(\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1-\mu_0)^\top\Sigma_1^{-1}(\mu_1-\mu_0) - k + \log\Big(\frac{\det\Sigma_1}{\det\Sigma_0}$$

where k is the dimensionality of the distribution.

Simplification

In our case, this simplifies to:

$$\mathcal{D}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)] =$$
$$\frac{1}{2}\left(\mathrm{tr}\left(\Sigma(X)\right) + \left(\mu(X)\right)^\top \left(\mu(X)\right) - k - \log\det\left(\Sigma(X)\right)\right).$$

Terms to compute

The first term on the right hand side is a bit more tricky. We could use sampling to estimate $E_{z\sim Q}[\log P(X|z)]$, but getting a good estimate would require passing many samples of z through f, which would be expensive. Hence, as is standard in stochastic gradient descent, we take one sample of z and treat $\log P(X|z)$ for that z as an approximation of $E_{z\sim Q}[\log P(X|z)]$. After all, we are already doing stochastic gradient descent over different values of X sampled from a dataset D. The full equation we want to optimize is:

$$\begin{aligned} E_{X\sim D}\left[\log P(X) - \mathcal{D}\left[Q(z|X)\|P(z|X)\right]\right] = \\ E_{X\sim D}\left[E_{z\sim Q}\left[\log P(X|z)\right] - \mathcal{D}\left[Q(z|X)\|P(z)\right]\right]. \end{aligned}$$

Computing the gradients

If we take the gradient of this equation, the gradient symbol can be moved into the expectations. Therefore, we can sample a single value of X and a single value of z from the distribution $Q(z|X)$, and compute the gradient of:

$$\log P(X|z) - \mathcal{D}\left[Q(z|X) \| P(z)\right]. \tag{1}$$

We can then average the gradient of this function over arbitrarily many samples of X and z, and the result converges to the gradient. There is, however, a significant problem $E_{z\sim Q}\left[\log P(X|z)\right]$ depends not just on the parameters of P, but also on the parameters of Q. In order to make VAEs work, it is essential to drive Q to produce codes for X that P can reliably decode.

$$E_{X\sim D}\left[E_{\epsilon\sim\mathcal{N}(0,I)}[\log P(X|z = \mu(X) + \Sigma^{1/2}(X) * \epsilon)] - \mathcal{D}\left[Q(z|X) \| P(z)\right]\right]$$

Code examples using Keras

See `https://keras.io/examples/generative/vae/`

Code in PyTorch for VAEs

```
import torch
from torch.autograd import Variable
import numpy as np
import torch.nn.functional as F
import torchvision
from torchvision import transforms
import torch.optim as optim
from torch import nn
import matplotlib.pyplot as plt
from torch import distributions

class Encoder(torch.nn.Module):
    def __init__(self, D_in, H, latent_size):
        super(Encoder, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, H)
        self.enc_mu = torch.nn.Linear(H, latent_size)
        self.enc_log_sigma = torch.nn.Linear(H, latent_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        mu = self.enc_mu(x)
        log_sigma = self.enc_log_sigma(x)
        sigma = torch.exp(log_sigma)
        return torch.distributions.Normal(loc=mu, scale=sigma)
```

What is a GAN?

A GAN is a deep neural network which consists of two networks, a so-called generator network and a discriminating network, or just discriminator. Through several iterations of generation and discrimination, the idea is that these networks will train each other, while also trying to outsmart each other.

What is a generator network?

A generator network is often a deep network which uses existing data to generate new data (from for example simulations of physical systems, imagesm video, audio and more) from randomly generated inputs, the so-called latent space. Training the network allows us to generate say new data, images etc. As an example a generator network could for example be a Boltzmann machine as discussed earlier. This machine is trained to produce for example a quantum mechanical probability distribution.
It can a simple neural network with an input layer and an output layer and a given number of hidden layers.

And what is a discriminator network?

A discriminator tries to distinguish between real data and those generated by the abovementioned generator.

Appplications of GANs

There are exteremely many applications of GANs

1. Image generation
2. Text-to-image analysis
3. Face-aging
4. Image-to-image translation
5. Video synthesis
6. High-resolution image generation
7. Completing missing parts of images and much more

Generative Adversarial Networks

Generative Adversarial Networks are a type of unsupervised machine learning algorithm proposed by Goodfellow et. al, see https://arxiv.org/pdf/1406.2661.pdf in 2014 (Read the paper first it's only 6 pages). The simplest formulation of the model is based on a game theoretic approach, *zero sum game*, where we pit two neural networks against one another. We define two rival networks, one generator g, and one discriminator d. The generator directly produces samples

$$x = g(z; \theta^{(g)}).$$

Discriminator

The discriminator attempts to distinguish between samples drawn from the training data and samples drawn from the generator. In other words, it tries to tell the difference between the fake data produced by g and the actual data samples we want to do prediction on. The discriminator outputs a probability value given by

$$d(x; \theta^{(d)}).$$

indicating the probability that x is a real training example rather than a fake sample the generator has generated.

Zero-sum game

The simplest way to formulate the learning process in a generative adversarial network is a zero-sum game, in which a function

$$v(\theta^{(g)}, \theta^{(d)}),$$

determines the reward for the discriminator, while the generator gets the conjugate reward

$$-v(\theta^{(g)}, \theta^{(d)})$$

Maximizing reward

During learning both of the networks maximize their own reward function, so that the generator gets better and better at tricking the discriminator, while the discriminator gets better and better at telling the difference between the fake and real data. The generator and discriminator alternate on which one trains at one time (i.e. for one epoch). In other words, we keep the generator constant and train the discriminator, then we keep the discriminator constant to train the generator and repeat. It is this back and forth dynamic which lets GANs tackle otherwise intractable generative problems. As the generator improves with training, the discriminator's performance gets worse because it cannot easily tell the difference between real and fake. If the generator ends up succeeding perfectly, the the discriminator will do no better than random guessing i.e. 50%.

Progression in training

This progression in the training poses a problem for the convergence criteria for GANs. The discriminator feedback gets less meaningful over time, if we continue training after this point then the generator is effectively training on junk data which can undo the learning up to that point. Therefore, we stop training when the discriminator starts outputting 1/2 everywhere. At convergence we have

$$g^* = \operatorname*{argmin}_{g} \max_{d} v(\theta^{(g)}, \theta^{(d)}),$$

Deafault choice

The default choice for v is

$$v(\theta^{(g)}, \theta^{(d)}) = \mathbb{E}_{x \sim p_{\mathrm{data}}} \log d(x) + \mathbb{E}_{x \sim p_{\mathrm{model}}} \log(1 - d(x)).$$

Design of GANs

The main motivation for the design of GANs is that the learning process requires neither approximate inference (variational autoencoders for example) nor approximation of a partition function. In the case where

$$\max_{d} v(\theta^{(g)}, \theta^{(d)})$$

is convex in $\theta^{(g)}$ then the procedure is guaranteed to converge and is asymptotically consistent (Seth Lloyd on QuGANs). This is in general not the case and it is possible to get situations where the training process never converges because the generator and discriminator chase one another around in the parameter space indefinitely.

More references

A much deeper discussion on the currently open research problem of GAN convergence is available from https://www.deeplearningbook.org/contents/generative_models.html. To anyone interested in learning more about GANs it is a highly recommended read. Direct quote: **In this best-performing formulation, the generator aims to increase the log probability that the discriminator makes a mistake, rather than aiming to decrease the log probability that the discriminator makes the correct prediction.** Another interesting read can be found at https://arxiv.org/abs/1701.00160.

Writing Our First Generative Adversarial Network

This part is best seen using the jupyter-notebook.

Let us implement a GAN in tensorflow. We will study the performance of our GAN on the MNIST dataset. This code is based on and adapted from the Google tutorial at
https://www.tensorflow.org/tutorials/generative/dcgan

First we import our libraries

```
import os
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model
```

Next we define our hyperparameters and import our data the usual way

```
BUFFER_SIZE = 60000
BATCH_SIZE = 256
EPOCHS = 30

data = tf.keras.datasets.mnist.load_data()
(train_images, train_labels), (test_images, test_labels) = data
train_images = np.reshape(train_images, (train_images.shape[0],
                                        28,
```