April 24-28: Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway
[2]Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA

April 24-28, 2023

Plans for the week April 24-28

Deep generative models and Restricted Boltzmann machines.

1. Restricted Boltzmann machines
2. Generative Adversarial Networks (GANs)
3. Reading recommendation: Goodfellow et al chapter 20.10-20-14
4. Video of lecture
5. Whiteboard notes

Boltzmann Machines

Why use a generative model rather than the more well known discriminative deep neural networks (DNN)?

- Discriminitave methods have several limitations: They are mainly supervised learning methods, thus requiring labeled data. And there are tasks they cannot accomplish, like drawing new examples from an unknown probability distribution.
- A generative model can learn to represent and sample from a probability distribution. The core idea is to learn a parametric model of the probability distribution from which the training data was drawn. As an example

1. A model for images could learn to draw new examples of cats and dogs, given a training dataset of images of cats and dogs.
2. Generate a sample of an ordered or disordered Ising model phase, having been given samples of such phases.
3. Model the trial function for Monte Carlo calculations

- Both use gradient-descent based learning procedures for minimizing cost functions
- Energy based models don't use backpropagation and automatic differentiation for computing gradients, instead turning to Markov Chain Monte Carlo methods.
- DNNs often have several hidden layers. A restricted Boltzmann machine has only one hidden layer, however several RBMs can be stacked to make up Deep Belief Networks, of which they constitute the building blocks.

History: The RBM was developed by amongst others Geoffrey Hinton, called by some the "Godfather of Deep Learning", working with the University of Toronto and Google.

A BM is what we would call an undirected probabilistic graphical model with stochastic continuous or discrete units.

It is interpreted as a stochastic recurrent neural network where the state of each unit(neurons/nodes) depends on the units it is connected to. The weights in the network represent thus the strength of the interaction between various units/nodes.

It turns into a Hopfield network if we choose deterministic rather than stochastic units. In contrast to a Hopfield network, a BM is a so-called generative model. It allows us to generate new samples from the learned distribution.

A standard BM network is divided into a set of observable and visible units $\hat{x}$ and a set of unknown hidden units/nodes $\hat{h}$.

Additionally there can be bias nodes for the hidden and visible layers. These biases are normally set to 1.

BMs are stackable, meaning they cwe can train a BM which serves as input to another BM. We can construct deep networks for learning complex PDFs. The layers can be trained one after another, a feature which makes them popular in deep learning

However, they are often hard to train. This leads to the introduction of so-called restricted BMs, or RBMS. Here we take away all lateral connections between nodes in the visible layer as well as connections between nodes in the hidden layer.

The network

The network layers:

1. A function $\mathbf{x}$ that represents the visible layer, a vector of M elements (nodes). This layer represents both what the RBM might be given as training input, and what we want it to be able to reconstruct. This might for example be the pixels of an image, the spin values of the Ising model, or coefficients representing speech.

2. The function $\mathbf{h}$ represents the hidden, or latent, layer. A vector of N elements (nodes). Also called "feature detectors".

The goal of the hidden layer is to increase the model's expressive power. We encode complex interactions between visible variables by introducing additional, hidden variables that interact with visible degrees of freedom in a simple manner, yet still reproduce the complex correlations between visible degrees in the data once marginalized over (integrated out).

Examples of this trick being employed in physics:

1. The Hubbard-Stratonovich transformation

2. The introduction of ghost fields in gauge theory

3. Shadow wave functions in Quantum Monte Carlo simulations

The network parameters, to be optimized/learned:

1. $\mathbf{a}$ represents the visible bias, a vector of same length as $\mathbf{x}$.

2. $\mathbf{b}$ represents the hidden bias, a vector of same lenght as $\mathbf{h}$.

3. W represents the interaction weights, a matrix of size $M \times N$.

Joint distribution. The restricted Boltzmann machine is described by a Boltzmann distribution

$$P_{rbm}(\mathbf{x},\mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})}, \tag{1}$$

where Z is the normalization constant or partition function, defined as

$$Z = \int \int e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})} d\mathbf{x} d\mathbf{h}. \tag{2}$$

It is common to ignore T_0 by setting it to one.

Network Elements, the energy function. The function $E(\mathbf{x}, \mathbf{h})$ gives the **energy** of a configuration (pair of vectors) $(\mathbf{x}, \mathbf{h})$. The lower the energy of a configuration, the higher the probability of it. This function also depends on the parameters $\mathbf{a}$, $\mathbf{b}$ and W. Thus, when we adjust them during the learning procedure, we are adjusting the energy function to best fit our problem.

Defining different types of RBMs. There are different variants of RBMs, and the differences lie in the types of visible and hidden units we choose as well as in the implementation of the energy function $E(\mathbf{x}, \mathbf{h})$. The connection between the nodes in the two layers is given by the weights w_{ij}.

Binary-Binary RBM: RBMs were first developed using binary units in both the visible and hidden layer. The corresponding energy function is defined as follows:

$$E(\mathbf{x}, \mathbf{h}) = -\sum_{i}^{M} x_i a_i - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j, \tag{3}$$

where the binary values taken on by the nodes are most commonly 0 and 1.

Gaussian-Binary RBM: Another varient is the RBM where the visible units are Gaussian while the hidden units remain binary:

$$E(\mathbf{x}, \mathbf{h}) = \sum_{i}^{M} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}. \tag{4}$$

1. RBMs are Useful when we model continuous data (i.e., we wish $\mathbf{x}$ to be continuous)
2. Requires a smaller learning rate, since there's no upper bound to the value a component might take in the reconstruction

Other types of units include:

1. Softmax and multinomial units
2. Gaussian visible and hidden units
3. Binomial units
4. Rectified linear units

Cost function. When working with a training dataset, the most common training approach is maximizing the log-likelihood of the training data. The log likelihood characterizes the log-probability of generating the observed data using our generative model. Using this method our cost function is chosen as the negative log-likelihood. The learning then consists of trying to find parameters that maximize the probability of the dataset, and is known as Maximum Likelihood Estimation (MLE). Denoting the parameters as $\boldsymbol{\theta} = a_1, ..., a_M, b_1, ..., b_N, w_{11}, ..., w_{MN}$, the log-likelihood is given by

$$\mathcal{L}(\{\theta_i\}) = \langle \log P_\theta(\boldsymbol{x}) \rangle_{data} \tag{5}$$

$$= -\langle E(\boldsymbol{x}; \{\theta_i\}) \rangle_{data} - \log Z(\{\theta_i\}), \tag{6}$$

where we used that the normalization constant does not depend on the data, $\langle \log Z(\{\theta_i\}) \rangle = \log Z(\{\theta_i\})$ Our cost function is the negative log-likelihood, $\mathcal{C}(\{\theta_i\}) = -\mathcal{L}(\{\theta_i\})$

Optimization / Training. The training procedure of choice often is Stochastic Gradient Descent (SGD). It consists of a series of iterations where we update the parameters according to the equation

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla \mathcal{C}(\boldsymbol{\theta}_k) \tag{7}$$

at each k-th iteration. There are a range of variants of the algorithm which aim at making the learning rate η more adaptive so the method might be more efficient while remaining stable.

We now need the gradient of the cost function in order to minimize it. We find that

$$\frac{\partial \mathcal{C}(\{\theta_i\})}{\partial \theta_i} = \langle \frac{\partial E(\boldsymbol{x}; \theta_i)}{\partial \theta_i} \rangle_{data} + \frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i} \tag{8}$$

$$= \langle O_i(\boldsymbol{x}) \rangle_{data} - \langle O_i(\boldsymbol{x}) \rangle_{model}, \tag{9}$$

where in order to simplify notation we defined the "operator"

$$O_i(\boldsymbol{x}) = \frac{\partial E(\boldsymbol{x}; \theta_i)}{\partial \theta_i}, \tag{10}$$

and used the statistical mechanics relationship between expectation values and the log-partition function:

$$\langle O_i(\boldsymbol{x}) \rangle_{model} = \mathrm{Tr} P_\theta(\boldsymbol{x}) O_i(\boldsymbol{x}) = -\frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i}. \tag{11}$$

The data-dependent term in the gradient is known as the positive phase of the gradient, while the model-dependent term is known as the negative phase of the gradient. The aim of the training is to lower the energy of configurations that are near observed data points (increasing their probability), and raising the energy of configurations that are far from observed data points (decreasing their probability).

The gradient of the negative log-likelihood cost function of a Binary-Binary RBM is then

$$\frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial w_{ij}} = \langle x_i h_j \rangle_{data} - \langle x_i h_j \rangle_{model} \tag{12}$$

$$\frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial a_{ij}} = \langle x_i \rangle_{data} - \langle x_i \rangle_{model} \tag{13}$$

$$\frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial b_{ij}} = \langle h_i \rangle_{data} - \langle h_i \rangle_{model}. \tag{14}$$

$$\tag{15}$$

To get the expectation values with respect to the *data*, we set the visible units to each of the observed samples in the training data, then update the hidden units according to the conditional probability found before. We then average over all samples in the training data to calculate expectation values with respect to the data.

Kullback-Leibler relative entropy. 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

$$\mathrm{KL}(p||q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x}. \tag{16}$$

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

$$\mathrm{KL}(f(\boldsymbol{x})||p(\boldsymbol{x}|\boldsymbol{\theta})) = \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{\theta})} d\boldsymbol{x} \tag{17}$$

$$= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log f(\boldsymbol{x}) d\boldsymbol{x} - \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} \tag{18}$$

$$= \langle \log f(\boldsymbol{x}) \rangle_{f(\boldsymbol{x})} - \langle \log p(\boldsymbol{x}|\boldsymbol{\theta}) \rangle_{f(\boldsymbol{x})} \tag{19}$$

$$= \langle \log f(\boldsymbol{x}) \rangle_{data} + \langle E(\boldsymbol{x}) \rangle_{data} + \log Z \tag{20}$$

$$= \langle \log f(\boldsymbol{x}) \rangle_{data} + \mathcal{C}_{LL}. \tag{21}$$

The first term is constant with respect to $\boldsymbol{\theta}$ since $f(\boldsymbol{x})$ is independent of $\boldsymbol{\theta}$. 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.

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

$$\langle\frac{\partial E(\boldsymbol{x};\theta_i)}{\partial\theta_i}\rangle_{model} = \int p(\boldsymbol{x}|\boldsymbol{\theta})\frac{\partial E(\boldsymbol{x};\theta_i)}{\partial\theta_i}d\boldsymbol{x} = -\frac{\partial\log Z(\theta_i)}{\partial\theta_i}. \tag{22}$$

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

$$\begin{aligned}
\frac{\partial\mathcal{C}_{LL}}{\partial\theta_i} =&\langle\frac{\partial E(\boldsymbol{x};\theta_i)}{\partial\theta_i}\rangle_{data} + \frac{\partial\log Z(\theta_i)}{\partial\theta_i} && (23)\\
=&\langle\frac{\partial E(\boldsymbol{x};\theta_i)}{\partial\theta_i}\rangle_{data} - \langle\frac{\partial E(\boldsymbol{x};\theta_i)}{\partial\theta_i}\rangle_{model} && (24)\\
& && (25)
\end{aligned}$$

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.

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.

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 we discussed for say linear regression.

Mathematical details. Because we are restricted to potential functions which are positive it is convenient to express them as exponentials, so that

$$\phi_C(\boldsymbol{x}_C) = e^{-E_C(\boldsymbol{x}_C)} \tag{26}$$

where $E(\boldsymbol{x}_C)$ is called an *energy function*, and the exponential representation is the *Boltzmann distribution.* The joint distribution is defined as the product of potentials.

The joint distribution of the random variables is then

$$\begin{aligned} p(\boldsymbol{x}) =& \frac{1}{Z}\prod_C \phi_C(\boldsymbol{x}_C) \\ =& \frac{1}{Z}\prod_C e^{-E_C(\boldsymbol{x}_C)} \\ =& \frac{1}{Z} e^{-\sum_C E_C(\boldsymbol{x}_C)} \\ =& \frac{1}{Z} e^{-E(\boldsymbol{x})}. \end{aligned} \tag{27}$$

$$p_{BM}(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z_{BM}} e^{-\frac{1}{T}E_{BM}(\boldsymbol{x},\boldsymbol{h})}, \tag{28}$$

with the partition function

$$Z_{BM} = \int\int e^{-\frac{1}{T}E_{BM}(\tilde{\boldsymbol{x}},\tilde{\boldsymbol{h}})} d\tilde{\boldsymbol{x}} d\tilde{\boldsymbol{h}}. \tag{29}$$

T is a physics-inspired parameter named temperature and will be assumed to be 1 unless otherwise stated. The energy function of the Boltzmann machine determines the interactions between the nodes and is defined

$$\begin{aligned} E_{BM}(\boldsymbol{x}, \boldsymbol{h}) = &- \sum_{i,k}^{M,K} a_i^k \alpha_i^k(x_i) - \sum_{j,l}^{N,L} b_j^l \beta_j^l(h_j) - \sum_{i,j,k,l}^{M,N,K,L} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j) \\ &- \sum_{i,m=i+1,k}^{M,M,K} \alpha_i^k(x_i) v_{im}^k \alpha_m^k(x_m) - \sum_{j,n=j+1,l}^{N,N,L} \beta_j^l(h_j) u_{jn}^l \beta_n^l(h_n). \end{aligned} \tag{30}$$

Here $\alpha_i^k(x_i)$ and $\beta_j^l(h_j)$ are one-dimensional transfer functions or mappings from the given input value to the desired feature value. They can be arbitrary functions of the input variables and are independent of the parameterization (parameters referring to weight and biases), meaning they are not affected by training of the model. The indices k and l indicate that there can be multiple transfer functions per variable. Furthermore, a_i^k and b_j^l are the visible and hidden bias. w_{ij}^{kl} are weights of the **inter-layer** connection terms which connect visible and hidden units. v_{im}^{k} and u_{jn}^{l} are weights of the **intra-layer** connection terms which connect the visible units to each other and the hidden units to each other, respectively.

We remove the intra-layer connections by setting v_{im} and u_{jn} to zero. The expression for the energy of the RBM is then

$$E_{RBM}(\boldsymbol{x},\boldsymbol{h}) = -\sum_{i,k}^{M,K} a_i^k \alpha_i^k(x_i) - \sum_{j,l}^{N,L} b_j^l \beta_j^l(h_j) - \sum_{i,j,k,l}^{M,N,K,L} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j). \tag{31}$$

resulting in

$$\begin{aligned}
P_{RBM}(\boldsymbol{x}) = &\int P_{RBM}(\boldsymbol{x},\tilde{\boldsymbol{h}}) d\tilde{\boldsymbol{h}} \\
=&\frac{1}{Z_{RBM}} \int e^{-E_{RBM}(\boldsymbol{x},\tilde{\boldsymbol{h}})} d\tilde{\boldsymbol{h}} \\
=&\frac{1}{Z_{RBM}} \int e^{\sum_{i,k} a_i^k \alpha_i^k(x_i) + \sum_{j,l} b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,j,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{\boldsymbol{h}} \\
=&\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \int \prod_{j}^{N} e^{\sum_{l} b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{\boldsymbol{h}} \\
=&\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \bigg(\int e^{\sum_{l} b_1^l \beta_1^l(\tilde{h}_1) + \sum_{i,k,l} \alpha_i^k(x_i) w_{i1}^{kl} \beta_1^l(\tilde{h}_1)} d\tilde{h}_1 \\
&\times \int e^{\sum_{l} b_2^l \beta_2^l(\tilde{h}_2) + \sum_{i,k,l} \alpha_i^k(x_i) w_{i2}^{kl} \beta_2^l(\tilde{h}_2)} d\tilde{h}_2 \\
&\times ... \\
&\times \int e^{\sum_{l} b_N^l \beta_N^l(\tilde{h}_N) + \sum_{i,k,l} \alpha_i^k(x_i) w_{iN}^{kl} \beta_N^l(\tilde{h}_N)} d\tilde{h}_N \bigg) \\
=&\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \prod_{j}^{N} \int e^{\sum_{l} b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{h}_j
\end{aligned} \tag{32}$$

Similarly

$$\begin{aligned}
P_{RBM}(\boldsymbol{h}) =&\frac{1}{Z_{RBM}} \int e^{-E_{RBM}(\tilde{\boldsymbol{x}},\boldsymbol{h})} d\tilde{\boldsymbol{x}} \\
=&\frac{1}{Z_{RBM}} e^{\sum_{j,l} b_j^l \beta_j^l(h_j)} \prod_{i}^{M} \int e^{\sum_{k} a_i^k \alpha_i^k(\tilde{x}_i) + \sum_{j,k,l} \alpha_i^k(\tilde{x}_i) w_{ij}^{kl} \beta_j^l(h_j)} d\tilde{x}_i
\end{aligned} \tag{33}$$

Using Bayes theorem

$$
\begin{aligned}
P_{RBM}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{P_{RBM}(\boldsymbol{x},\boldsymbol{h})}{P_{RBM}(\boldsymbol{x})} \\
=& \frac{\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i) + \sum_{j,l} b_j^l \beta_j^l(h_j) + \sum_{i,j,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j)}}{\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \prod_j^N \int e^{\sum_l b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{h}_j} \\
=& \prod_j^N \frac{e^{\sum_l b_j^l \beta_j^l(h_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j)}}{\int e^{\sum_l b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{h}_j}
\end{aligned} \tag{34}
$$

Similarly

$$
\begin{aligned}
P_{RBM}(\boldsymbol{x}|\boldsymbol{h}) =& \frac{P_{RBM}(\boldsymbol{x},\boldsymbol{h})}{P_{RBM}(\boldsymbol{h})} \\
=& \prod_i^M \frac{e^{\sum_k a_i^k \alpha_i^k(x_i) + \sum_{j,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j)}}{\int e^{\sum_k a_i^k \alpha_i^k(\tilde{x}_i) + \sum_{j,k,l} \alpha_i^k(\tilde{x}_i) w_{ij}^{kl} \beta_j^l(h_j)} d\tilde{x}_i}
\end{aligned} \tag{35}
$$

The original RBM had binary visible and hidden nodes. They were showned to be universal approximators of discrete distributions. It was also shown that adding hidden units yields strictly improved modelling power. The common choice of binary values are 0 and 1. However, in some physics applications, -1 and 1 might be a more natural choice. We will here use 0 and 1.

$$
E_{BB}(\boldsymbol{x},\mathbf{h}) = -\sum_i^M x_i a_i - \sum_j^N b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j. \tag{36}
$$

$$
p_{BB}(\boldsymbol{x},\boldsymbol{h}) = \frac{1}{Z_{BB}} e^{\sum_i^M a_i x_i + \sum_j^N b_j h_j + \sum_{ij}^{M,N} x_i w_{ij} h_j} \tag{37}
$$

$$
= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}} \tag{38}
$$

with the partition function

$$
Z_{BB} = \sum_{\boldsymbol{x},\boldsymbol{h}} e^{\boldsymbol{x}^T \boldsymbol{a} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}}. \tag{39}
$$

Marginal Probability Density Functions. In order to find the probability of any configuration of the visible units we derive the marginal probability density function.

$$
\begin{aligned}
p_{BB}(\boldsymbol{x}) =& \sum_{\boldsymbol{h}} p_{BB}(\boldsymbol{x}, \boldsymbol{h}) \qquad (40)\\
=& \frac{1}{Z_{BB}} \sum_{\boldsymbol{h}} e^{\boldsymbol{x}^T \boldsymbol{a} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}}\\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \sum_{\boldsymbol{h}} e^{\sum_j^N (b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j}\\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \sum_{\boldsymbol{h}} \prod_j^N e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j}\\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \left(\sum_{h_1} e^{(b_1 + \boldsymbol{x}^T \boldsymbol{w}_{*1}) h_1} \times \sum_{h_2} e^{(b_2 + \boldsymbol{x}^T \boldsymbol{w}_{*2}) h_2} \times \right.\\
& \left. ... \times \sum_{h_2} e^{(b_N + \boldsymbol{x}^T \boldsymbol{w}_{*N}) h_N} \right)\\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \prod_j^N \sum_{h_j} e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j}\\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \prod_j^N (1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}). \qquad (41)
\end{aligned}
$$

A similar derivation yields the marginal probability of the hidden units

$$
p_{BB}(\boldsymbol{h}) = \frac{1}{Z_{BB}} e^{\boldsymbol{b}^T \boldsymbol{h}} \prod_i^M (1 + e^{a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}}). \qquad (42)
$$

Conditional Probability Density Functions. We derive the probability of the hidden units given the visible units using Bayes' rule

$$
\begin{aligned}
p_{BB}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{p_{BB}(\boldsymbol{x},\boldsymbol{h})}{p_{BB}(\boldsymbol{x})} \\
=& \frac{\frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}+\boldsymbol{b}^T\boldsymbol{h}+\boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}})} \\
=& \frac{e^{\boldsymbol{x}^T\boldsymbol{a}} e^{\sum_j^N (b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}})} \\
=& \prod_j^N \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} \\
=& \prod_j^N p_{BB}(h_j|\boldsymbol{x}).
\end{aligned} \tag{43}
$$

From this we find the probability of a hidden unit being "on" or "off":

$$
\begin{aligned}
p_{BB}(h_j=1|\boldsymbol{x}) =& \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} && (44)\\
=& \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} && (45)\\
=& \frac{1}{1+e^{-(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})}}, && (46)
\end{aligned}
$$

and

$$
p_{BB}(h_j=0|\boldsymbol{x}) = \frac{1}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}}. \tag{47}
$$

Similarly we have that the conditional probability of the visible units given the hidden are

$$
\begin{aligned}
p_{BB}(\boldsymbol{x}|\boldsymbol{h}) =& \prod_i^M \frac{e^{(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})x_i}}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}} && (48)\\
=& \prod_i^M p_{BB}(x_i|\boldsymbol{h}). && (49)
\end{aligned}
$$

$$
\begin{aligned}
p_{BB}(x_i=1|\boldsymbol{h}) =& \frac{1}{1+e^{-(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})}} && (50)\\
p_{BB}(x_i=0|\boldsymbol{h}) =& \frac{1}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}}. && (51)
\end{aligned}
$$

Gaussian-Binary Restricted Boltzmann Machines. Inserting into the expression for $E_{RBM}(\boldsymbol{x}, \boldsymbol{h})$ in equation results in the energy

$$
\begin{aligned}
E_{GB}(\boldsymbol{x}, \boldsymbol{h}) =& \sum_{i}^{M} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j}^{N} b_j h_j - \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2} \\
=& || \frac{\boldsymbol{x} - \boldsymbol{a}}{2\boldsymbol{\sigma}} ||^2 - \boldsymbol{b}^T \boldsymbol{h} - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \boldsymbol{h}.
\end{aligned} \tag{52}
$$

Joint Probability Density Function.

$$
\begin{aligned}
p_{GB}(\boldsymbol{x}, \boldsymbol{h}) =& \frac{1}{Z_{GB}} e^{-|| \frac{\boldsymbol{x} - \boldsymbol{a}}{2\boldsymbol{\sigma}} ||^2 + \boldsymbol{b}^T \boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \boldsymbol{h}} \\
=& \frac{1}{Z_{GB}} e^{- \sum_{i}^{M} \frac{(x_i - a_i)^2}{2\sigma_i^2} + \sum_{j}^{N} b_j h_j + \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}} \\
=& \frac{1}{Z_{GB}} \prod_{ij}^{M,N} e^{- \frac{(x_i - a_i)^2}{2\sigma_i^2} + b_j h_j + \frac{x_i w_{ij} h_j}{\sigma_i^2}},
\end{aligned} \tag{53}
$$

with the partition function given by

$$
Z_{GB} = \int \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-|| \frac{\tilde{\boldsymbol{x}} - \boldsymbol{a}}{2\boldsymbol{\sigma}} ||^2 + \boldsymbol{b}^T \tilde{\boldsymbol{h}} + (\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \tilde{\boldsymbol{h}}} d\tilde{\boldsymbol{x}}. \tag{54}
$$

Marginal Probability Density Functions. We proceed to find the marginal probability densitites of the Gaussian-binary RBM. We first marginalize over the binary hidden units to find $p_{GB}(\boldsymbol{x})$

$$
\begin{aligned}
p_{GB}(\boldsymbol{x}) =& \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} p_{GB}(\boldsymbol{x}, \tilde{\boldsymbol{h}}) \\
=& \frac{1}{Z_{GB}} \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-|| \frac{\boldsymbol{x} - \boldsymbol{a}}{2\boldsymbol{\sigma}} ||^2 + \boldsymbol{b}^T \tilde{\boldsymbol{h}} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \tilde{\boldsymbol{h}}} \\
=& \frac{1}{Z_{GB}} e^{-|| \frac{\boldsymbol{x} - \boldsymbol{a}}{2\boldsymbol{\sigma}} ||^2} \prod_{j}^{N} (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}).
\end{aligned} \tag{55}
$$

We next marginalize over the visible units. This is the first time we marginalize over continuous values. We rewrite the exponential factor dependent on $\boldsymbol{x}$ as a Gaussian function before we integrate in the last step.

$$
\begin{aligned}
p_{GB}(\boldsymbol{h}) =& \int p_{GB}(\tilde{\boldsymbol{x}}, \boldsymbol{h}) d\tilde{\boldsymbol{x}} \\
=& \frac{1}{Z_{GB}} \int e^{-||\frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}} d\tilde{\boldsymbol{x}} \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \int \prod_i^M e^{-\frac{(\tilde{x}_i - a_i)^2}{2\sigma_i^2} + \frac{\tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{\sigma_i^2}} d\tilde{\boldsymbol{x}} \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \left(\int e^{-\frac{(\tilde{x}_1 - a_1)^2}{2\sigma_1^2} + \frac{\tilde{x}_1 \boldsymbol{w}_{1*}^T \boldsymbol{h}}{\sigma_1^2}} d\tilde{x}_1 \right. \\
& \times \int e^{-\frac{(\tilde{x}_2 - a_2)^2}{2\sigma_2^2} + \frac{\tilde{x}_2 \boldsymbol{w}_{2*}^T \boldsymbol{h}}{\sigma_2^2}} d\tilde{x}_2 \\
& \times ... \\
& \left. \times \int e^{-\frac{(\tilde{x}_M - a_M)^2}{2\sigma_M^2} + \frac{\tilde{x}_M \boldsymbol{w}_{M*}^T \boldsymbol{h}}{\sigma_M^2}} d\tilde{x}_M \right) \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{(\tilde{x}_i - a_i)^2 - 2\tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}} d\tilde{x}_i \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{\tilde{x}_i^2 - 2\tilde{x}_i(a_i + \tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}) + a_i^2}{2\sigma_i^2}} d\tilde{x}_i \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{\tilde{x}_i^2 - 2\tilde{x}_i(a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}) + (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})^2 - (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})^2 + a_i^2}{2\sigma_i^2}} d\tilde{x}_i \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{(\tilde{x}_i - (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}))^2 - a_i^2 - 2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} - (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2 + a_i^2}{2\sigma_i^2}} d\tilde{x}_i \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \int e^{-\frac{(\tilde{x}_i - a_i - \boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} d\tilde{x}_i \\
=& \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}.
\end{aligned}
\tag{56}
$$

Conditional Probability Density Functions. We finish by deriving the conditional probabilities.

$$\begin{aligned}
p_{GB}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{x})} \\
=& \frac{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2} \prod_j^N (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}})} \\
=& \prod_j^N \frac{e^{(b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}) h_j}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\
=& \prod_j^N p_{GB}(h_j|\boldsymbol{x}).
\end{aligned} \tag{57}$$

The conditional probability of a binary hidden unit h_j being on or off again takes the form of a sigmoid function

$$\begin{aligned}
p_{GB}(h_j = 1|\boldsymbol{x}) =& \frac{e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\
=& \frac{1}{1 + e^{-b_j - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}
\end{aligned} \tag{58}$$

$$p_{GB}(h_j = 0|\boldsymbol{x}) = \frac{1}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}. \tag{59}$$

The conditional probability of the continuous $\boldsymbol{x}$ now has another form, however.

$$
\begin{aligned}
p_{GB}(\boldsymbol{x}|\boldsymbol{h}) =& \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{h})} \\
=& \frac{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{(x_i-a_i)^2}{2\sigma_i^2} + \frac{x_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + 2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i - b_i - \boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \\
=& \prod_i^M \mathcal{N}(x_i | b_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}, \sigma_i^2) \qquad (60) \\
\Rightarrow p_{GB}(x_i|\boldsymbol{h}) =& \mathcal{N}(x_i | b_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}, \sigma_i^2). \qquad (61)
\end{aligned}
$$

The form of these conditional probabilities explains the name "Gaussian" and the form of the Gaussian-binary energy function. We see that the conditional probability of x_i given $\boldsymbol{h}$ is a normal distribution with mean $b_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}$ and variance σ_i^2.

Generative Models

Generative models describe a class of statistical models that are a contrast to **discriminative models**. Informally we say that generative models can generate new data instances while discriminative models discriminate between different kinds of data instances. A generative model could generate new photos of animals that look like 'real' animals while a discriminative model could tell a dog from a cat. More formally, given a data set x and a set of labels / targets y. Generative models capture the joint probability $p(x, y)$, or just $p(x)$ if there are no labels, while discriminative models capture the conditional probability $p(y|x)$. Discriminative models generally try to draw boundaries in the data space (often high dimensional), while generative models try to model how data is placed throughout the space.

Generative Adversarial Networks

Generative Adversarial Networks are a type of unsupervised machine learning algorithm proposed by Goodfellow et. al 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

$$x = g(z; \theta^{(g)}) \tag{62}$$

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

$$d(x; \theta^{(d)}) \tag{63}$$

indicating the probability that x is a real training example rather than a fake sample the generator has generated. The simplest way to formulate the learning process in a generative adversarial network is a zero-sum game, in which a function

$$v(\theta^{(g)}, \theta^{(d)}) \tag{64}$$

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

$$-v(\theta^{(g)}, \theta^{(d)}) \tag{65}$$

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%. 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

$$g^* = \underset{g}{\text{argmin}}\, \underset{d}{\text{max}} v(\theta^{(g)}, \theta^{(d)}) \tag{66}$$

The default choice for v is

$$v(\theta^{(g)}, \theta^{(d)}) = \mathbb{E}_{x \sim p_{\text{data}}} \log d(x) + \mathbb{E}_{x \sim p_{\text{model}}} \log(1 - d(x)) \tag{67}$$

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

$$\underset{d}{\text{max}} v(\theta^{(g)}, \theta^{(d)}) \tag{68}$$

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. A much deeper discussion on the currently open research problem of GAN convergence is available here. 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

Writing Our First Generative Adversarial Network

Let us now move on to actually implementing 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

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,
```

```
                                             28,
                                             1)).astype('float32')

# we normalize between -1 and 1
train_images = (train_images - 127.5) / 127.5
training_dataset = tf.data.Dataset.from_tensor_slices(
                      train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Let's have a quick look

```
plt.imshow(train_images[0], cmap='Greys')
plt.show()
```

Now we define our two models. This is where the 'magic' happens. There are a huge amount of possible formulations for both models. A lot of engineering and trial and error can be done here to try to produce better performing models. For more advanced GANs this is by far the step where you can 'make or break' a model.

We start with the generator. As stated in the introductory text the generator g upsamples from a random sample to the shape of what we want to predict. In our case we are trying to predict MNIST images (28×28 pixels).

```
def generator_model():
    """
    The generator uses upsampling layers tf.keras.layers.Conv2DTranspose() to
    produce an image from a random seed. We start with a Dense layer taking this
    random sample as an input and subsequently upsample through multiple
    convolutional layers.
    """

    # we define our model
    model = tf.keras.Sequential()


    # adding our input layer. Dense means that every neuron is connected and
    # the input shape is the shape of our random noise. The units need to match
    # in some sense the upsampling strides to reach our desired output shape.
    # we are using 100 random numbers as our seed
    model.add(layers.Dense(units=7*7*BATCH_SIZE,
                           use_bias=False,
                           input_shape=(100, )))
    # we normalize the output form the Dense layer
    model.add(layers.BatchNormalization())
    # and add an activation function to our 'layer'. LeakyReLU avoids vanishing
    # gradient problem
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, BATCH_SIZE)))
    assert model.output_shape == (None, 7, 7, BATCH_SIZE)
    # even though we just added four keras layers we think of everything above
    # as 'one' layer

    # next we add our upscaling convolutional layers
    model.add(layers.Conv2DTranspose(filters=128,
                                     kernel_size=(5, 5),
                                     strides=(1, 1),
                                     padding='same',
```

```python
                                    use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    assert model.output_shape == (None, 7, 7, 128)

    model.add(layers.Conv2DTranspose(filters=64,
                                    kernel_size=(5, 5),
                                    strides=(2, 2),
                                    padding='same',
                                    use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    assert model.output_shape == (None, 14, 14, 64)

    model.add(layers.Conv2DTranspose(filters=1,
                                    kernel_size=(5, 5),
                                    strides=(2, 2),
                                    padding='same',
                                    use_bias=False,
                                    activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

And there we have our 'simple' generator model. Now we move on to defining our discriminator model d, which is a convolutional neural network based image classifier.

```python
def discriminator_model():
    """
    The discriminator is a convolutional neural network based image classifier
    """

    # we define our model
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(filters=64,
                            kernel_size=(5, 5),
                            strides=(2, 2),
                            padding='same',
                            input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    # adding a dropout layer as you do in conv-nets
    model.add(layers.Dropout(0.3))


    model.add(layers.Conv2D(filters=128,
                            kernel_size=(5, 5),
                            strides=(2, 2),
                            padding='same'))
    model.add(layers.LeakyReLU())
    # adding a dropout layer as you do in conv-nets
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

Let us take a look at our models. **Note**: double click images for bigger view.

```
generator = generator_model()
plot_model(generator, show_shapes=True, rankdir='LR')
```

```
discriminator = discriminator_model()
plot_model(discriminator, show_shapes=True, rankdir='LR')
```

Next we need a few helper objects we will use in training

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

The first object, $cross_entropyisourlossfunctionandthetwoothersareouroptimizers.Noticeweusethesamele$

```
def generator_loss(fake_output):
    loss = cross_entropy(tf.ones_like(fake_output), fake_output)

    return loss


def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_liks(fake_output), fake_output)
    total_loss = real_loss + fake_loss

    return total_loss
```

Next we define a kind of seed to help us compare the learning process over multiple training epochs.

```
noise_dimension = 100
n_examples_to_generate = 16
seed_images = tf.random.normal([n_examples_to_generate, noise_dimension])
```

Now we have everything we need to define our training step, which we will apply for every step in our training loop. Notice the @tf.function flag signifying that the function is tensorflow 'compiled'. Removing this flag doubles the computation time.

```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dimension])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
```

```
    gradients_of_generator = gen_tape.gradient(gen_loss,
                                               generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                               discriminator.trainable_variables)
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                               generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
                                               discriminator.trainable_variables))

    return gen_loss, disc_loss
```

Next we define a helper function to produce an output over our training epochs to see the predictive progression of our generator model. ***Note****: I am including this code here, but comment it out in the training loop.*

```
def generate_and_save_images(model, epoch, test_input):
    # we're making inferences here
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig(f'./images_from_seed_images/image_at_epoch_{str(epoch).zfill(3)}.png')
    plt.close()
    #plt.show()
```

Setting up checkpoints to periodically save our model during training so that everything is not lost even if the program were to somehow terminate while training.

```
# Setting up checkpoints to save model during training
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, 'ckpt')
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                            discriminator_optimizer=discriminator_optimizer,
                            generator=generator,
                            discriminator=discriminator)
```

Now we define our training loop

```
def train(dataset, epochs):
    generator_loss_list = []
    discriminator_loss_list = []

    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)
            generator_loss_list.append(gen_loss.numpy())
```

```
            discriminator_loss_list.append(disc_loss.numpy())

        #generate_and_save_images(generator, epoch + 1, seed_images)

        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)

        print(f'Time for epoch {epoch} is {time.time() - start}')

    #generate_and_save_images(generator, epochs, seed_images)

    loss_file = './data/lossfile.txt'
    with open(loss_file, 'w') as outfile:
        outfile.write(str(generator_loss_list))
        outfile.write('\n')
        outfile.write('\n')
        outfile.write(str(discriminator_loss_list))
        outfile.write('\n')
        outfile.write('\n')
```

To train simply call this function. ***Warning****: this might take a long time so there is a folder of a pretrained network already included in the repository.*

```
train(train_dataset, EPOCHS)
```

And here is the result of training our model for 100 epochs

Movie 1: `images_from_seed_images/generation.gif`

Now to avoid having to train and everything, which will take a while depending on your computer setup we now load in the model which produced the above gif.

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
restored_generator = checkpoint.generator
restored_discriminator = checkpoint.discriminator

print(restored_generator)
print(restored_discriminator)
```

Exploring the Latent Space

So we have successfully loaded in our latest model. Let us now play around a bit and see what kind of things we can learn about this model. Our generator takes an array of 100 numbers. One idea can be to try to systematically change our input. Let us try and see what we get

```
def generate_latent_points(number=100, scale_means=1, scale_stds=1):
    latent_dim = 100
    means = scale_means * tf.linspace(-1, 1, num=latent_dim)
    stds = scale_stds * tf.linspace(-1, 1, num=latent_dim)
    latent_space_value_range = tf.random.normal([number, number],
                                                 means,
                                                 stds,
```

```
                                           dtype=tf.float64)
    return latent_space_value_range

def generate_images(latent_points):
    # notice we set training to false because we are making inferences
    generated_images = restored_generator(latent_space_value_range,
                                          training=False)

    return generated_images


def plot_result(generated_images, number):
    # obviously this assumes sqrt number is an int
    fig, axs = plt.subplots(int(np.sqrt(number)), int(np.sqrt(number)),
                            figsize=(10, 10))

    for i in range(int(np.sqrt(number))):
        for j in range(int(np.sqrt(number))):
            axs[i, j].imshow(generated_images[i*j], cmap='Greys')
            axs[i, j].axis('off')

    plt.show()


generated_images = generate_images(generate_latent_points())
plot_result(generated_images, number)
```

Interesting! We see that the generator generates images that look like MNIST numbers: $1, 4, 7, 9$. *Let's try to tweak it a bit more to see if we are able to generate a similar plot where we generate every MNIST number. Let us now try to 'move' a bit around in the latent space.* ***Note****: decrease the plot number if these following cells take too long to run on your computer.*

```
plot_number = 225

generated_images = generate_images(generate_latent_points(number=plot_number,
                                                          scale_means=5,
                                                          scale_stds=1))
plot_result(generated_images, plot_number)

generated_images = generate_images(generate_latent_points(number=plot_number,
                                                          scale_means=-5,
                                                          scale_stds=1))
plot_result(generated_images, plot_number)

generated_images = generate_images(generate_latent_points(number=plot_number,
                                                          scale_means=1,
                                                          scale_stds=5))
plot_result(generated_images, plot_number)
```

Again, we have found something interesting. Moving *around using our means takes us from digit to digit, while* moving *around using our standard deviations seem to increase the number of different digits! In the last image above, we can barely make out every MNIST digit. Let us make on last plot using this information by upping the standard deviation of our Gaussian noises.*

```
plot_number = 400
generated_images = generate_images(generate_latent_points(number=plot_number,
                                                          scale_means=1,
                                                          scale_stds=10))
```

A pretty cool result! We see that our generator indeed has learned a distribution which qualitatively looks a whole lot like the MNIST dataset.

Interpolating Between MNIST Digits

Another interesting way to explore the latent space of our generator model is by interpolating between the MNIST digits. This section is largely based on "this excellent blogpost": https://machinelearningmastery.com/how-to-interpolate-and-perform-vector-arith *by Jason Brownlee.*

So let us start