Contents

An Overview of Machine Learning

Dimitrios Komninos

2023, February

1 From AI to Deep Learning

1.1 Introduction

The concept of Artificial Intelligence, or AI for abbreviation, goes back to the 1930s since Alan Turing published his first work, one of the most important and influential papers in the history of computer science [1]. Research upon the subject grew rapidly in the following years and massively the last decade where we see various and complex techniques utilized in industry and influencing our lives every day.

More terms have been introduced throughout the years including *Machine Learning*, *Neural Networks* and *Deep Learning*. They should not be confused with one another. AI is a general concept and the others are just parts of it. The following picture and points help to clarify the matter.

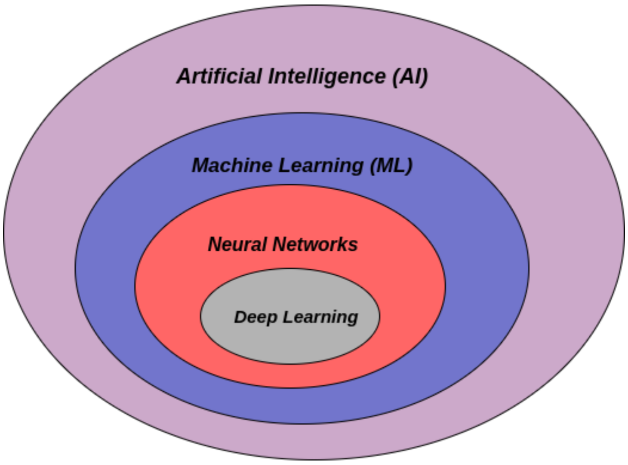

Figure 1.1. ML is not pure AI

- **Artificial Intelligence**: A machine accomplishes a task that requires human intelligence.
- **Machine Learning**: The art of designing an AI model based on data.
- **Neural Networks**: A family of architectures of ML algorithms.
- **Deep Learning**: Neural networks with multiple layers of computation.

Every machine learning model is categorized first and foremost based on feedback and then, based on the task it is intended to execute. The next graph provides a visual categorization of ML systems.

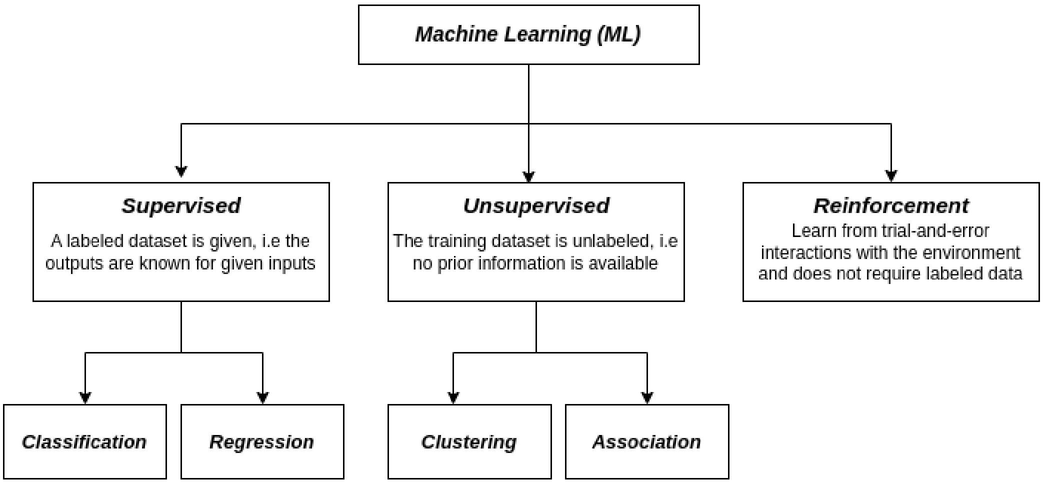
Machine Learning (ML)
Supervised
A labeled dataset is given, i.e the outputs are known for given inputs
Unsupervised
The training dataset is unlabeled, i.e no prior information is available
Reinforcement
Learn from trial-and-error interactions with the environment and does not require labeled data
Classification
Regression
Clustering
Association

Figure 1.2. ML Systems Categorization

Next, we head into neural networks and their basic architecture in terms of learning algorithms.

1.2 Artificial Neural Networks (ANNs)

As the name suggests, ANNs form a simplified model of the human brain. Generally, we can think of an ANN as a *nonlinear function* which transforms input data to output, based on the training on many data samples and parameters. In 1943, Warren McCulloch and Walter Pitts suggested the computational model of a neuron. For the sake of completeness, know that the human brain consists of approximately 100 billion neurons (brain cells) and 100 trillion synapses (permit interactions between neurons).

An ANN is a collection of connected processing units (neurons) which consist a network. Each of these units is able to execute a simple and specific mathematical operation. Although the simplicity, neural networks exhibit great processing potential. The figure below presents an ANN:

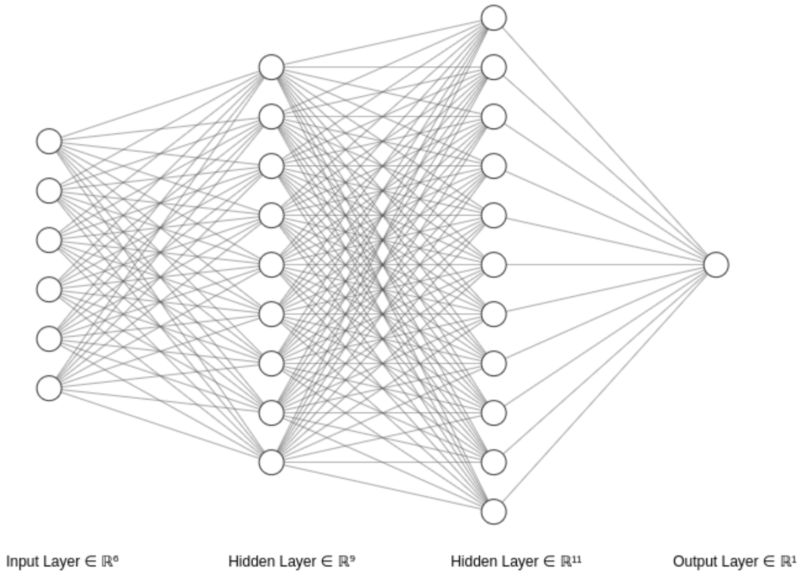
Input Layer ∈ $\mathbb{R}^6$
Hidden Layer ∈ $\mathbb{R}^9$
Hidden Layer ∈ $\mathbb{R}^{11}$
Output Layer ∈ $\mathbb{R}^1$

Figure 1.3. A dense ANN with two hidden layers

Every neural network must obviously have an *input layer* as well as an *output layer*. The in-between are called the *hidden layers*, but the simplest form of a network is just a single neuron (or perceptron) with one or more inputs, some processing and a single output as shown below:

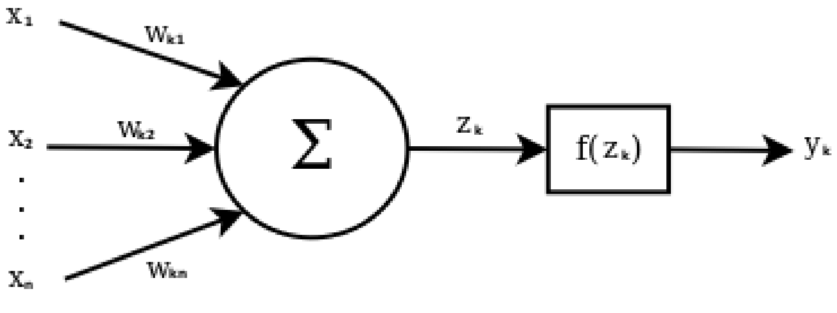

Figure 1.4. Basic Neuron Model (Perceptron)

The inputs are denoted as $x_j \in \mathbb{R}$ where $j = 1, ..., n$. The output of the neuron is a *nonlinear function* of weighted sums of inputs:

$$z_k = \sum_j w_{kj} x_j + b_k \tag{1}$$

where $w_{kj} \in \mathbb{R}$ are the *weights* and $b_k \in \mathbb{R}$ is some *offset* or *bias*. Then, (1) is the input of some nonlinear function which can be the same for all neurons in a network. Two examples of commonly used functions include the *rectified linear unit (RELU)*, where $y_k = f(z_k) = z_k$ and a constant derivative equal to one for $z_k \geq 0$ and zero elsewhere. Also, the *sigmoid function*

$$f(z_k) = \frac{1}{1 + e^{-z_k}} \tag{2}$$

provides a smooth variation in the output within the interval $[0, 1]$, which is useful when we want to model uncertainty. The derivative is equal to $(1 - y_k)y_k$. One may also consider other activation functions as well, as long as changes in the inputs do not cause big changes in the output.

For the sake of brevity, when dealing with neural networks we assume that a neuron node includes all the process we described above, meaning that each node we see at Figure 1.3 is a model as in Figure 1.4.

Let us consider a simple network with no hidden layers and introduce some indices that help avoid consufion when analyzing the underlying processes, as well as when scaling the network with more layers.

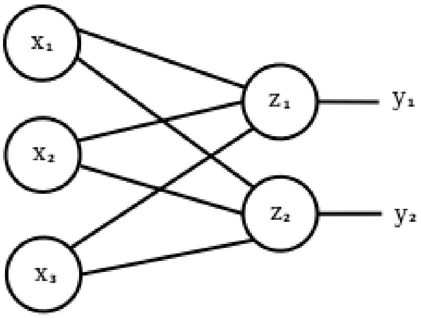

Figure 1.5. Simple ANN without hidden layers

We will use the index j for the input neurons and k for the output neurons. So, from (1):

$$z_k = \sum_{j=1}^{n} w_{kj} x_j + b_k \tag{3}$$

for $k = 1, 2$, hence

$$y_k = f(z_k) \tag{4}$$

By simple inspection on the indices and some linear algebra, we can write (3) in a more compact and computationally effective form as

$$\vec{\mathbf{z}} = \mathbf{W} \cdot \vec{\mathbf{x}} + \vec{\mathbf{b}} \tag{5}$$

where $\vec{\mathbf{z}} = \begin{bmatrix} z_1 & z_2 \end{bmatrix}^T$ is the output k-dimensional vector, $\mathbf{W}$ is the weight matrix of $k \times j$ dimensions acting with the dot product on the input j-dimensional vector $\vec{\mathbf{x}} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$ and $\vec{\mathbf{b}}$ is the bias vector of k dimensions. The number of rows of the weight matrix is equal to the number of output neurons and the number of columns is equal to the number of input neurons corresponding to that matrix. At last, we have a k-dimensional vector $\vec{\mathbf{y}}$ consisting of the nonlinear outputs of some activation function. It is also clear that there is a *feedforward process*, where the outputs of some layer are passed as inputs to the next layer.

1.2.1 Universality of ANNs and Training

At this point, we should introduce the *Universal Approximation Theorem*. It states that any arbitrary smooth function with vector input and vector output can be approximated as well as desired by an ANN with at least one hidden layer, as long as we allow for sufficiently many neurons.

All of the aforementioned may seem a little bit abstract in terms of the values of weights and biases. One must adjust these parameters in order to get the desired output value. This value should be of high accuracy and we can achieve this through *training* of the network at hand.

Training refers to the adjustment of those parameters based on some training data samples. This is a form of supervised learning. Assume that a neural network is denoted by F_w (contains weights and biases), its input vector by x^{in} and its output vector by y^{out}, such that

$$y^{out} = F_w(x^{in}) \tag{6}$$

Also, let the desired or target function be F. A training dataset consists of inputs x^{in} and respective outputs y^{target}, such that $y^{target} = F(x^{in})$. Obviously, we would like the network output to approximate with high accuracy the target output, hence

$$y^{out} \approx y^{target} = F(x^{in}) \tag{7}$$

We need a *cost function* that can measure the deviation between y^{out} and y^{target} with respect to the parameters. One may use various functions based on the underlying task, but one of the most common cost functions is the *least-squares* defined as

$$C(w) = \frac{1}{2} \left\langle \|F_w(x^{in}) - F(x^{in})\|^2 \right\rangle \tag{8}$$

where the vector *norm* is used and the *average over all samples* is taken.

There is process called *batch training*, where the respective model is trained on many samples in parallel. In this case, we would need a matrix of samples with dimensions $N_{samples} \times j$. That is the reason we use the average in expression (8). The 1/2 factor is used to ease derivative evaluation, but with modern computers this is not a matter to be concerned of.

1.2.2 Gradient Descent and the Backpropagation Algorithm

The purpose of training is to find the "best" weights and biases by minimizing the cost function with respect to its parameters. We can achieve this by using the *gradient descent method*. The gradient of the cost function is evaluated and then, the parameters should follow the path of the steepest descent, as the method's name suggests. Remember that the gradient is a vector pointing in the direction of the steepest ascent, so we need to find the negative gradient for minimization. One may also reverse the problem based on the underlying task and try to maximize the cost by using the so-called *gradient ascent*. We will stick to the minimization of the cost, meaning that if $C(w) \to 0$, the model tends toward better accuracy.

The problem is that evaluating $C(w)$ would mean averaging over all training samples. However, when we deal with lots of data we tend to average only a few samples and get an approximate cost. In each step, different samples are taken. This is called *stochastic gradient descent (SGD)*.

For sufficiently small steps, the sum over many steps approximates the true gradient. The following figure helps visualizing the difference between SGD and the true gradient:

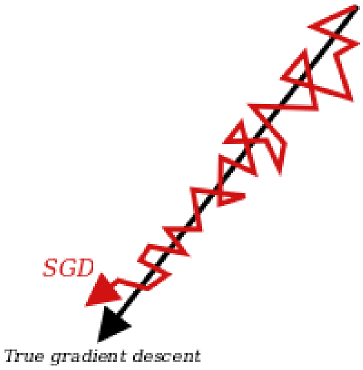

Figure 1.6. SGD vs True Gradient Descent

After evaluating the gradient of the cost function, we proceed to the update of the parameters as

$$w_* \; \to \; w_* - \eta \frac{\partial C(w)}{\partial w_*} \tag{9}$$

where η is the so-called *learning rate*, a small constant to ensure convergence to a local minimum. The learning rate can be also interpreted as a step size that shows how fast the gradient is moving and thus, how fast the learning process is. We assume that w_* is some weight somewhere in the network including the bias, as the introduction of this offset is the same as if we considered an extra input $x_0 = 1$ with weight $w_{k0} = b_k$.

If we look back at Figure 1.3 and for deep networks in general, the question is how do we calculate the gradient with respect to some inner weight? If a network consists of one million weights, we need to evaluate it one million times! Fortunately, the *chain rule* for derivatives is the holy grail here!

Let us consider a small network with two inputs and one output as shown below:

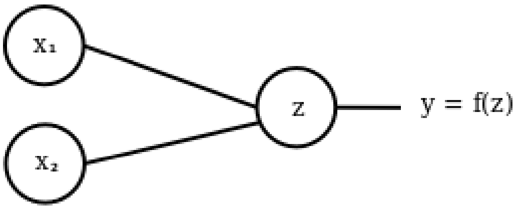

Figure 1.7. A simple ANN with two inputs and one output

From (3), we have

$$z = w_1 x_1 + w_2 x_2 + b \tag{10}$$

and the cost function reads

$$C(w) = \frac{1}{2} \left\langle (f(z) - F(x_1, x_2))^2 \right\rangle \tag{11}$$

where F is the target function. Using the last two expressions, the gradient reads

$$\nabla C(w) = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \\ \frac{\partial C}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \left\langle (f(z) - F) f'(z) \frac{\partial z}{\partial w_1} \right\rangle \\ \\ \left\langle (f(z) - F) f'(z) \frac{\partial z}{\partial w_2} \right\rangle \end{bmatrix} = \begin{bmatrix} \langle (f(z) - F) f'(z) x_1 \rangle \\ \\ \langle (f(z) - F) f'(z) x_2 \rangle \end{bmatrix}$$

and thus, we have all the values to evaluate it. We proceed with an update for both weights as in (9) and feed another input data until the cost function reaches a local minimum.

Next, we consider the general case of an ANN with n hidden layers:

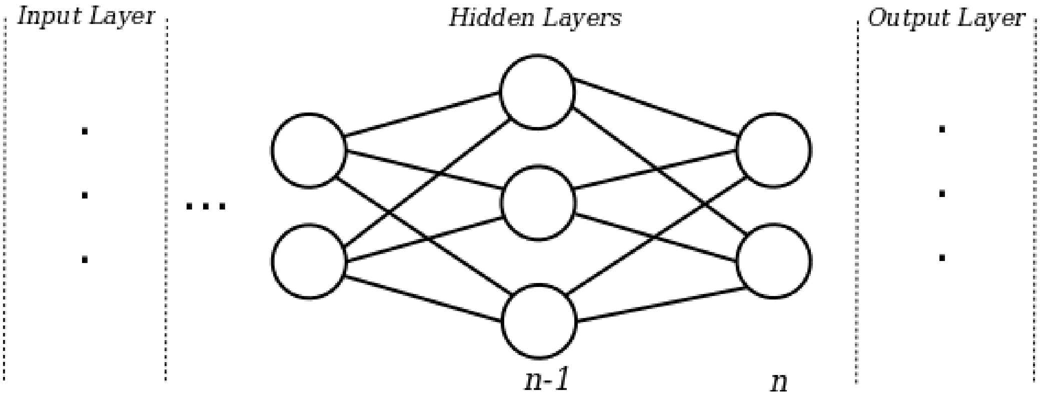

Figure 1.8. General ANN with n hidden layers

In order to work with the full network when it comes for the gradient layer-by-layer, we adopt the well-known *backpropagation algorithm.* The name will be clear later on. First, we need to keep track of the indices very carefully:

- x^{in}, the ANN input
- y_j^n, the value of neuron j in layer n
- z_j^n, the input value for $y_j^n = f(z_j^n)$
- $w_{jk}^{n,n-1}$, the weight from the k-th neuron in layer $n-1$ feeding the j-th neuron of layer n

The cost for one particular input is $C(w) = \langle C(w, x^{in}) \rangle$. The derivative with respect to some weight w_* somewhere in the network is

$$\frac{\partial C(w, x^{in})}{\partial w_*} = \sum_j (y_j^n - F_j(x^{in})) \frac{\partial y_j^n}{\partial w_*} \tag{12}$$

$$= \sum_j (y_j^n - F_j(x^{in})) f'(z_j^n) \frac{\partial z_j^n}{\partial w_*} \tag{13}$$

where F_j is the target function. Remember that $y_j^n = f(z_j^n)$, so if we apply the chain rule repeatedly for the term $\frac{\partial z_j^n}{\partial w_*}$, we get

$$\frac{\partial z_j^n}{\partial w_*} = \sum_k \frac{\partial z_j^n}{\partial y_k^{n-1}} \frac{\partial y_k^{n-1}}{\partial w_*} = \sum_k w_{jk}^{n,n-1} f'(z_k^{n-1}) \frac{\partial z_k^{n-1}}{\partial w_*} \tag{14}$$

This last expression runs over all the weights corresponding to the j-th neuron of layer n. Plugging this into (13), we must do the same for all neurons of layer n. Here comes an important insight to compute this efficiently. We can construct a matrix $\mathbf{M}$ whose element with index (j, k) is

$$M_{jk}^{n,n-1} = w_{jk}^{n,n-1} f'(z_k^{n-1}) \tag{15}$$

.

In this way, each pair of layers $(l, l-1)$ backwards through the network contributes with multiplication of a matrix $\mathbf{M}$ with dimensions $(l \times l-1)$. Instead of having to go through all the weights of some pair of layers, we can use repeated matrix multiplication starting from the last hidden layer up to the input as

$$\frac{\partial z_j^n}{\partial w_*} = \sum_{k,l,...,u,v} M_{jk}^{n,n-1} M_{kl}^{n-1,n-2} ... M_{uv}^{2,1} \frac{\partial z_v^1}{\partial w_*} \tag{16}$$

We see that in order to compute the derivative of the cost function, we need values from layer n, as well as values from the previous layer $n-1$. To continue with the evaluation of (14), we need the values of layer $n-1$ and layer $n-2$. In this way, we propagate the results starting from the n-th hidden layer backwards onto the first hidden layer. That is the backpropagation algorithm, which we declare below:

Algorithm 1 Backpropagation

```
Input: $y_j^n, F_j(x^{in}), f'(z_j^n)$                  ▷ n is the output layer
Output: $\frac{\partial C(w,x^{in})}{\partial w_*}$      ▷ gradient vector with respect to all weights

  $\Delta_j \leftarrow (y_j^n - F_j(x^{in}))f'(z_j^n)$   ▷ initialize vector from output layer
  $l \leftarrow n$
  repeat                                                  ▷ for each pair of layers
      for all $j \in l$ do                                ▷ neurons in $l$-th layer
          for all $k \in (l-1)$ do                        ▷ neurons in $(l-1)$-th layer
              $\mathbf{M}_{jk}^{n,n-1} \leftarrow w_{jk}^{n,n-1} f'(z_k^{n-1})$   ▷ construct layer matrix
          end for
      end for

      $\Delta_k^{new} \leftarrow \sum_j \Delta_j \mathbf{M}_{jk}^{n,n-1}$   ▷ multiply vector by matrix
      $dC \leftarrow$ store cost derivatives for all weights and biases in layer $l$
      $l \leftarrow l-1$
  until first hidden layer and input layer pair
```

Once we get the gradient with respect to all the parameters of the network, we proceed to the update as in equation (9). Then, the same procedure is followed with the updated parameters on some other subset of the training dataset. After all, everything reduces to matrix and vector multiplications which can be easily achieved through the efficiency of modern computer algebra. This is the power of the backpropagation algorithm, which combines basic aspects of calculus and linear algebra. Notice that we do not have to evaluate with respect to all possible weights w_* of the network. We just start from some weight in the output layer and via the chain rule, all weights contribute to the gradient vector, as it was intended to.

1.2.3 On Hyperparameters, Datasets and Training

The art of machine learning lies in the *fine-tuning* of the parameters and the structure of the respective model. In general, there is no standard procedure that one should follow in order to adjust the settings of a model, as it all comes down to the background and the experience of the designer.

Before heading towards implementation though, we have to deal with the *data* that the network will be trained on, especially in the case of supervised learning. The stage of *data pre-processing* has to do with collection, feature extraction and various techniques involving statistical analysis and pattern recognition, such as missing data issues, noisy samples, outliers removal and normalization. The following diagram portrays the various concerns that arise when designing a machine learning prototype:

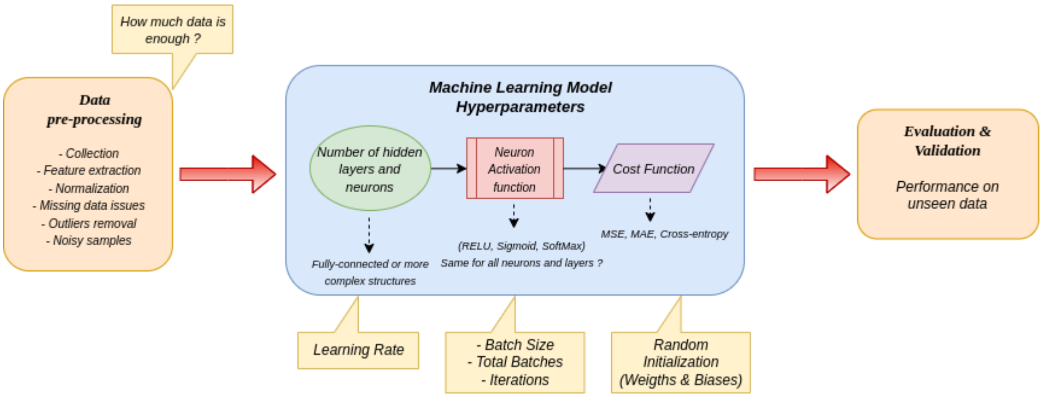
How much data is enough ?
Data pre-processing
- Collection
- Feature extraction
- Normalization
- Missing data issues
- Outliers removal
- Noisy samples
Machine Learning Model Hyperparameters
Number of hidden layers and neurons
Neuron Activation function
Cost Function
Fully-connected or more complex structures
(RELU, Sigmoid, SoftMax)
Same for all neurons and layers ?
MSE, MAE, Cross-entropy
Evaluation & Validation
Performance on unseen data
Learning Rate
- Batch Size
- Total Batches
- Iterations
Random Initialization (Weigths & Biases)

Figure 1.9. Learning is a stochastic and very complex nonlinear process

Basic machine learning model implementations with *Python*, as well as a summary of how parameter selection can influence the training process can be found in my personal *github* repository [2].

1.3 Generative Modeling

Generative modeling is a form of unsupervised learning where the taining dataset is unlabeled. This means that there is no prior information on what the output should be with respect to the input data. The goal of the model is to identify the relationships and interdependencies in intricate datasets and then produce similar data by drawing from the learned distribution. That is, it models an unknown probability distribution and generates synthetic data. Models of this kind have found several applications including computer vision, speech synthesis, inference of missing text, noise removal from images, chemical design and much more.

Assume that we have an unknown target distribution p and the distribution learned by the model $q_{\vec{\theta}}$. The objective is to minimize the *divergence* $\mathcal{D}$ of these distributions, i.e

$$\vec{\theta}^* = \arg\min_{\vec{\theta}} \mathcal{D}(p, q_{\vec{\theta}}) \tag{17}$$

where $\vec{\theta}^*$ are the parameters such that the distribution represented by the model approximately conforms to the targeted distribution as indicated by the training data in terms of divergence in the statistical manifold. As the actual distribution is not known a priori, it is estimated using a dataset $\{\vec{v}_i\}_{i=1}^N$ that is available to us and follows the distribution p.

In information theory and statistics, divergence $\mathcal{D}$ between two distributions is a kind of statistical distance that indicates how 'close' these distributions are. Consider the probability distributions p and q on a sample space $\mathcal{X}$. Of prime importance is the so-called *Kullback–Leibler* divergence or else, the *relative entropy* defined as

$$\mathcal{D}_{KL}(p, q) = \sum_{x \in \mathcal{X}} p(x) \log(\frac{p(x)}{q(x)}) = -\sum_{x \in \mathcal{X}} p(x) \log(\frac{q(x)}{p(x)}) \tag{18}$$

The relative entropy is always a non-negative value and it equals zero only when p and q are identical. Despite this fact, it cannot be considered as a true metric of distance between distributions because it fails to exhibit symmetry and does not adhere to the triangle inequality. Nevertheless, it is commonly viewed as a measure of "distance" between distributions for practical purposes, as in the case of training the generative models we discuss in this section.

1.3.1 Generative Adversarial Networks (GANs)

A generative adversarial network (GAN) is a machine learning framework consisting of two ANNs that compete with each other in a zero-sum game. These two sub-networks are commonly seen as the *generator* and the *discriminator* respectively.

The following picture represents the general structure of a GAN:

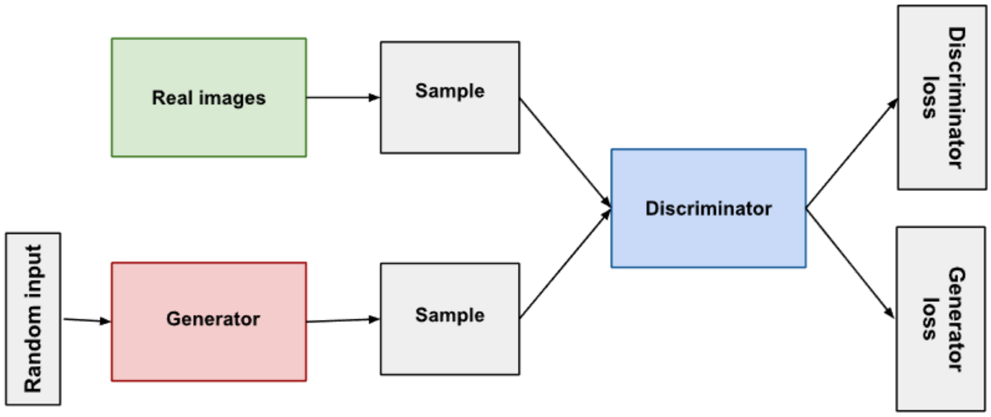

Figure 1.10. General structure of a GAN. Source: Google

The generator's task is to generate conceivable data. The discriminator is mainly a classifier and it is trained using the generated instances as negative examples and the real data as positive examples. By differentiating between the generator's synthetic data and real data, the discriminator penalizes the generator for creating unrealistic outputs. At the start of training, the generator's output is evidently artificial and the discriminator learns to identify it as such. With time, the generator's performance improves and it generates more realistic data, leading to the discriminator making more classification errors and ultimately decreasing its accuracy. If the generator is trained effectively, the discriminator's ability to distinguish between real and fake data diminishes, causing it to misclassify fake data as real and lower its accuracy. As we can see, the generator tries to maximize the error of the discriminator, whereas the latter wants to minimize the generator's error by providing information feedback to update its parameters.

Neural networks typically require input data to operate. However, when a network generates novel instances as output, we must determine an appropriate form of input to supply to the network. At a fundamental level, a GAN utilizes random noise as its input. Subsequently, the generator processes this noise into a relevant output, allowing the GAN to produce a diverse range of data instances by drawing from distinct locations in the desired distribution. Empirical studies indicate that the nature of the noise distribution is not significantly relevant. Hence, we can use a simple distribution, such as a uniform distribution, for noise generation. Furthermore, the noise space is typically of smaller dimension than the output space for ease of use.

Training a GAN is a complex process because its training algorithm must address two complications. The first is that we have to train two different ANNs. Thus, there is a need of scheduling two procedures. This leads to the second difficulty, the identification of the convergence of the GAN training as a whole. A method of alternating training is adopted during the design of a GAN. First, the discriminator trains for one or more iterations. Then, the generator trains for one or more iterations as well. These two steps are repeated to train the two respective networks. During the discriminator training phase, we keep the generator constant, meaning that we do not proceed to the update of its parameters. Respectively, we keep the discriminator constant during the generator's training.

As the generator's proficiency improves through training, the discriminator's performance deteriorates since distinguishing between real and fake instances becomes more challenging. In the best case scenario where the generator performs exceptionally well, the discriminator would have a 50% accuracy, indicating that it is no better than random guessing at classification. The evolution of the generator and discriminator relationship presents a challenge for the convergence of the GAN. As training progresses, the feedback from the discriminator to the generator becomes less relevant and if training persists beyond this point, the generator might receive spurious feedback, ultimately resulting in a decline in quality.

1.4 Boltzmann Machines

A Boltzmann machine is a unique category of neural network that serves as essential component in deep learning structures. Even today, they hold significant relevance in the field of both practical and theoretical machine learning. The idea for this type of network goes back to 1982 from J.J. Hopfield's work, where he presents a completely connected network comprising interdependent, deterministic units and possess the capability to store and recall binary patterns [3]. A Boltzmann machine [4] is a modified version of the Hopfield network consisting of stochastic units. Every unit modifies its state with time, relying on a probabilistic approach based on the state of neighboring units. This proposal addresses various problems of Hopfield networks that are presented in his work.

The general structure of a Boltzmann machine consists of the so-called *visible units* that play the role of the input layer and the *hidden units*, serving as underlying variables that shape a conditional and concealed representation of the data.

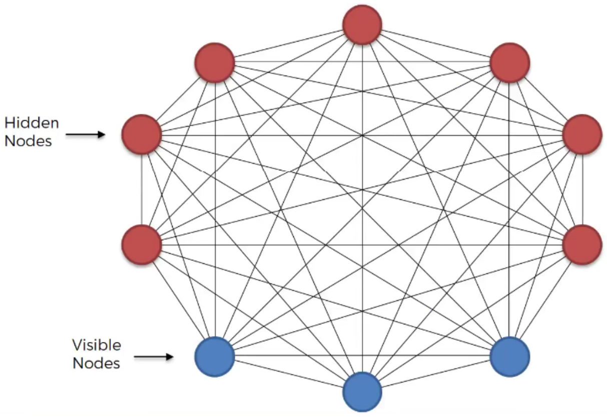

Figure 1.11. A Boltzmann machine of 3 visible units and 7 hidden units. Source: Google

1.4.1 Restricted Boltzmann Machine (RBM)

Interestingly, Boltzmann machines can be trained for generative modeling by using an alternative structure called *restricted Boltzmann machine* [5–7]. This version tackles with the problem of learning the parameters of the model, which is computationally intensive due to full connectivity of the network. A restricted Boltzmann machine, on the other hand, retains a similar structure with the difference that the units in the same layer are not interconnected, resulting in a bipartite graph as presented below:

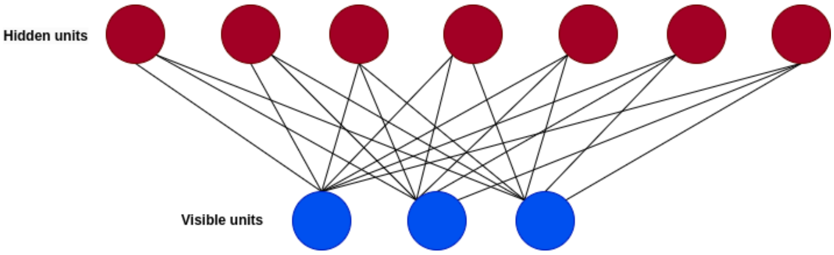

Figure 1.12. A restricted Boltzmann machine of 3 visible units and 7 hidden units.

Restricted Boltzmann machine units are binary stochastic units that exhibit probabilistic behavior, taking on a value of either 0 or 1. The joint state of the visible layer VL is a bitstring $\mathbf{v} \in \{0,1\}^V$, where V is the number of units. Same goes for the hidden layer HL as $\mathbf{h} \in \{0,1\}^H$. In a restricted Boltzmann machine, every unit possesses a bias and every connection has a weight. Functioning as a generative model, the machine characterizes a probability distribution. The model determines the joint distribution of each conceivable pair of visible and hidden vectors as

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z}\, e^{-E(\mathbf{v},\mathbf{h})} \tag{19}$$

Also, the marginal probability distribution of the visible layer can be evaluated by summing over all possible vectors of the hidden layer as

$$p(\mathbf{v}) = \sum_{\mathbf{h}} \frac{1}{Z}\, e^{-E(\mathbf{v},\mathbf{h})} \tag{20}$$

and that of the hidden layer by doing the opposite as

$$p(\mathbf{h}) = \sum_{\mathbf{v}} \frac{1}{Z}\, e^{-E(\mathbf{v},\mathbf{h})} \tag{21}$$

The energy value $E(\mathbf{v}, \mathbf{h})$ of a given joint configuration $(\mathbf{v}, \mathbf{h})$ in a restricted Boltzmann machine depends on the biases and pairwise interactions of the units as

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i \in VL} a_i v_i - \sum_{j \in HL} b_j h_j - \sum_{i,j} w_{ij} v_i h_j \tag{22}$$

which can be written in a more concrete way by defining the VL bias vector $\mathbf{a}$, the HL bias vector $\mathbf{b}$ and the weight matrix $\mathbf{W}$ between the layers, so

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{a}^T\mathbf{v} - \mathbf{b}^T\mathbf{h} - \mathbf{v}^T\mathbf{W}\mathbf{h} \tag{23}$$

In equation (19), Z refers to the so-called *partition function* which is calculated by summing over all possible pairs of visible and hidden vectors as

$$Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{24}$$

which can be interpreted as a normalizing constant to ensure that (19) is a valid probability distribution, i.e probabilities sum to 1. However, it is clear that the computational cost in order to calculate Z is of exponential time and feasible for problems with low dimensionality. This translates to the evaluation of the joint distribution as well.

A common technique to overcome this problem is by using an algorithm called *Gibbs Sampling*. Gibbs sampling is a type of Markov Chain Monte Carlo (MCMC) algorithm used to generate samples from a probability distribution. The method is commonly used in machine learning and statistical inference. A starting value is chosen for each variable in the model, and then, in each iteration, one variable is updated by sampling from its conditional distribution given the values of the other variables. The process repeats, and after a sufficiently large number of iterations, the samples converge to the true underlying distribution. One of the advantages of Gibbs sampling is that it can be used to sample from complex distributions where direct sampling is not possible. However, convergence to the true distribution can be slow and the method may require a large number of iterations to produce accurate results.

By utilizing Bayes theorem and some simple mathematical operations, we can derive

$$p(\mathbf{h}|\mathbf{v}) = \prod_j p(h_j|\mathbf{v}) \tag{25}$$

and

$$p(\mathbf{v}|\mathbf{h}) = \prod_i p(v_i|\mathbf{h}) \tag{26}$$

hence, it implies conditional independence of visible units conditioned on all hidden units and vice-versa, as it is a product of probabilities.

At last, we can express (25), (26) using the sigmoid function σ as

$$p(h_j = 1|\mathbf{v}) = \sigma\left(b_j + \sum_i w_{ij} v_i\right) \tag{27}$$

and

$$p(v_i = 1|\mathbf{h}) = \sigma\left(a_i + \sum_j w_{ij} h_j\right) \tag{28}$$

respectively.

1.4.2 Learning with a restricted Boltzmann Machine

In order to train a restricted Boltzmann machine, we recall what we said in the beginning of this subsection on generative modeling. Consider a training dataset with N samples $\mathcal{S} = \{\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, ..., \mathbf{v}^{(N)}\}$ following the distribution $p_{data}(\mathbf{v})$, where $\mathbf{v} \in \mathcal{S}$. We denote the model distribution as $q_\theta(\mathbf{v})$.

We are interested in minimizing the relative entropy between the model and the data distribution

$$\mathcal{D}_{KL}(p_{data}, q_\theta) = \sum_{\mathbf{v}\in\mathcal{S}} p_{data}(\mathbf{v}) \log(\frac{p_{data}(\mathbf{v})}{q_\theta(\mathbf{v})}) \tag{29}$$

as

$$\arg\min_\theta \mathcal{D}(p_{data}, q_\theta) \tag{30}$$

If we work this out, we can see that minimizing the relative entropy is equal to maximizing the *log-likelihood*

$$\arg\min_\theta \mathcal{D}(p_{data}, q_\theta) = \arg\min_\theta \sum_{\mathbf{v}\in\mathcal{S}} (p_{data}(\mathbf{v}) \log p_{data}(\mathbf{v}) - p_{data}(\mathbf{v}) \log q_\theta(\mathbf{v})) \tag{31}$$

$$= \arg\max_\theta \sum_{\mathbf{v}\in\mathcal{S}} p_{data}(\mathbf{v}) \log q_\theta(\mathbf{v}) \tag{32}$$

Gradient-based optimization methods are typically employed to maximize the log-likelihood. The log-likelihood for a particular vector $\mathbf{v}$ and parameter θ can be obtained by using (20) as

$$\ln q_\theta(\mathbf{v}) = \ln\left(\sum_{\mathbf{h}} \frac{1}{Z}\, e^{-E(\mathbf{v},\mathbf{h})}\right) \tag{33}$$

$$= \ln \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} - \ln \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{34}$$

The gradient with respect to the parameter θ is

$$\nabla_\theta \ln q_\theta(\mathbf{v}) = -\sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \nabla_\theta E(\mathbf{v},\mathbf{h}) + \sum_{\mathbf{v},\mathbf{h}} p(\mathbf{v},\mathbf{h}) \nabla_\theta E(\mathbf{v},\mathbf{h}) \tag{35}$$

which comes by differentiation of (34). For the given dataset of N samples, the log-likelihood gradient is

$$\frac{1}{N}\sum_{l=1}^{N} \nabla_\theta \ln q_\theta(\mathbf{v}^{(l)}) = \mathbb{E}_q[\nabla_\theta E(\mathbf{v},\mathbf{h})] - \mathbb{E}_{p_{data}}[\nabla_\theta E(\mathbf{v},\mathbf{h})] \tag{36}$$

where $\mathbb{E}$ denotes the expectation value with respect to the corresponding distribution.

For the restricted Boltzmann machine, the parameter θ corresponds to all weights and biases of the model. From the last equation, we derive the following

$$\Delta w_{ij} = \mathbb{E}_{p_{data}}[v_i h_j] - \mathbb{E}_q[v_i h_j] \tag{37}$$

$$\Delta a_i = \mathbb{E}_{p_{data}}[v_i] - \mathbb{E}_q[v_i] \tag{38}$$

$$\Delta b_j = \mathbb{E}_{p_{data}}[h_j] - \mathbb{E}_q[h_j] \tag{39}$$

hence, we update the parameters as

$$w_{ij}^* = w_{ij} + \eta \Delta w_{ij} \tag{40}$$

$$a_i^* = a_i + \eta \Delta a_i \tag{41}$$

$$b_j^* = b_j + \eta \Delta b_j \tag{42}$$

where η is the learning rate.

The expectation value of the probabilities of the hidden layer with respect to the training data $\mathbb{E}_{p_{data}}$ can be easily obtained and it's called the *positive phase*. The second term $\mathbb{E}_q$ is called the *negative phase* and calculates the joint probability of the visible and the hidden layer. As we said previously, the evaluation of this expression is exponential. It depends on the size of the smallest layer because the joint probability $p(\mathbf{v}, \mathbf{h})$ can be expressed with respect to both conditional probabilities $p(\mathbf{h}|\mathbf{v})$ or $p(\mathbf{v}|\mathbf{h})$.

A Markov chain Monte Carlo (MCMC) algorithm for sampling is once again used to obtain the negative phase of our model. The samples are collected from the Markov chain at the point of reaching the steady-state distribution. However, if the chain is allowed to converge at every iteration, it would result in a high computational cost. To circumvent this issue, we employ a technique called *Contrastive Divergence (CD)* [7], which helps us avoid the computational overhead. The intuition behind CD is that the positive phase computes the activations that the RBM expects to see when the input is a training sample, while the negative phase computes the activations that the RBM generates when the input is generated from the RBM itself. By minimizing the difference between these activations, CD encourages the RBM to generate samples that are similar to the training samples.

To reduce the computational burden, one can initialize the Markov chain with dataset samples, thereby bringing it closer to the target distribution. Additionally, instead of computing the expectation over the entire converged distribution, we can obtain a single sample $\mathbf{v}^k$ by running the Markov chain for k steps. This enables us to bypass most of the computational expense involved in obtaining samples from the fully converged distribution.

Hinton has explained that 'contrastive divergence' can be viewed as the disparity between two Kullback-Leibler divergences. He also argues that we should not focus on minimizing the relative entropy between the original data distribution and the fully converged distribution of the Markov chain, but we can minimize the following:

$$\mathcal{D}_{KL}(p_{data}(\mathbf{v}), q_\theta(\mathbf{v})) = \mathcal{D}_{KL}(p_{data}(\mathbf{v}), q_\theta(\mathbf{v})) - \mathcal{D}_{KL}(p_k(\mathbf{v}), q_\theta(\mathbf{v})) \tag{43}$$

where $p_k(\mathbf{v})$ is the distribution after k steps of the Markov chain. For the majority of problems, a single iteration of Gibbs sampling is adequate in practice.

1.4.3 Generative Modeling with RBMs

All things considered, a restricted Boltzmann machine (RBM) is a generative stochastic artificial neural network that can be used for unsupervised learning tasks, such as dimensionality reduction, feature learning, collaborative filtering and topic modeling. RBMs are particularly useful for modeling complex distributions of high-dimensional data, such as images, speech signals, genomic data and have been successfully applied in a wide range of fields, including computer vision, speech recognition, natural language processing and bioinformatics. RBMs are also used as building blocks for more complex deep learning models, such as deep belief networks and deep autoencoders, which are used for tasks like image and text generation and anomaly detection.

Here, we will use the *Fashion-MNIST dataset* in order to train a RBM to generate synthetic data that resembles the samples of this dataset. Some original samples are presented below:

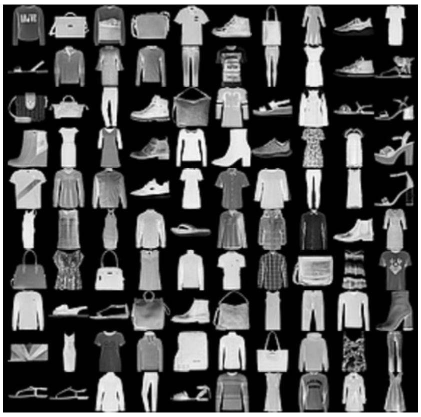

Figure 1.13. Random samples from the Fashion-MNIST dataset

The Fashion-MNIST dataset is a benchmark dataset commonly used in the field of computer vision for image classification tasks. It was created as a more challenging alternative to the popular MNIST dataset, which consists of grayscale images of handwritten digits. It consists of 70,000 grayscale images of size 28x28 pixels, which are divided into 60,000 training images and 10,000 testing images. Each image in the dataset belongs to one of 10 different classes representing different types of clothing and accessories, including T-shirts/tops, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags and ankle boots.

The values of each pixel range from 0 to 255. However, a RBM has binary units, and thus, we convert the grayscale images with values $[0, 255]$ to binary with values $[0, 1]$. If one wants to feed the machine with continuous inputs, a different approach should be taken by using a variation of the contrastive divergence algorithm. The mapping is done by setting a threshold at 128. If the pixel value is below 128 we set the value to 0, otherwise we set the value to 1. Note that the threshold of 128 used in the original method is equivalent to sampling from a Bernoulli distribution with probability of success 0.5. The advantage of using the Bernoulli distribution is that it allows for a more flexible threshold, since the probability of success can vary across different pixels. Additionally, using the Bernoulli distribution ensures that the binary pixel values are statistically independent of each other, which can be beneficial for some machine learning applications.

The input layer of the RBM has 784 units, corresponding to the 28x28 pixel values of each image in the Fashion-MNIST dataset. The hidden layer has 100 units, which were found to be sufficient for general image reconstructions.

To train the RBM, we used the mini-batch version of the Contrastive Divergence algorithm, following the practical guide by Hinton [8]. Specifically, we divided the Fashion-MNIST training set into batches of 10 images. We used a learning rate of 0.1 divided by the mini-batch size to ensure the consistency across different batch sizes. During training, we updated the weights and biases of the RBM using the positive and negative phases of the Contrastive Divergence algorithm. In the positive phase, we clamped the input values to the visible layer and computed the probabilities of the hidden units. In the negative phase, we sampled the hidden units based on their probabilities and then used these samples to compute the probabilities of the visible units. We then sampled from the visible units to obtain a reconstruction of the input, which was used to update the weights and biases of the RBM. We repeated this process for a fixed number of iterations and then used the trained model to generate reconstructions of the input images.

The first figure below shows 3 random samples from the test data, the ones that were not used in training, while the second presents the reconstructed images from the RBM:

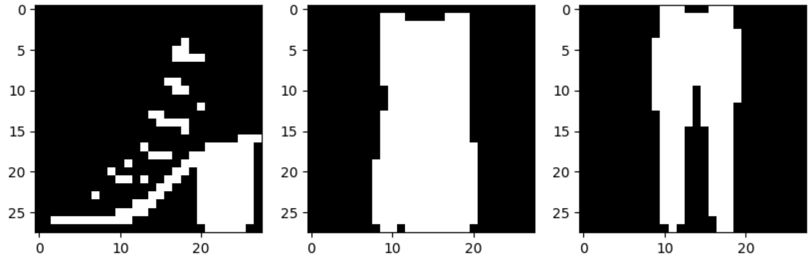

Figure 1.14. Random samples from the Fashion-MNIST test dataset

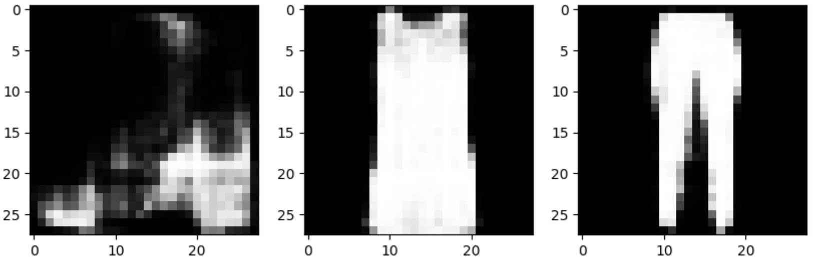

Figure 1.15. Synthetic Data produced by the RBM trained with 20 epochs, 10 Markov steps and learning rate of 0.01

As we can see, images with more detail are difficult to reconstruct. We found that 20-30 training iterations are more than enough for convergence.

Finally, we present the reconstruction error progress throughout training with different learning rates and number of hidden units. As per Hinton's practical guide, we used the momentum method. Instead of using the estimated gradient times the learning rate to increment the values of the parameters, the momentum method uses this quantity to increment the velocity of the parameters and the current velocity is then used as the parameter increment. The recipe for the momentum states that we should start with a value of 0.5. Once the large initial progress in the reduction of the reconstruction error has settled down to gentle progress (after roughly 5 iterations in our implementation), we increased the momentum to 0.9. This shock may cause a transient increase in the reconstruction error, as Hinton states, and this is obvious from the graphs below. This causes a more lasting instability, so we keep reducing the learning rate by factors of 2 until the instability disappears.

This is clear when monitoring the training process:

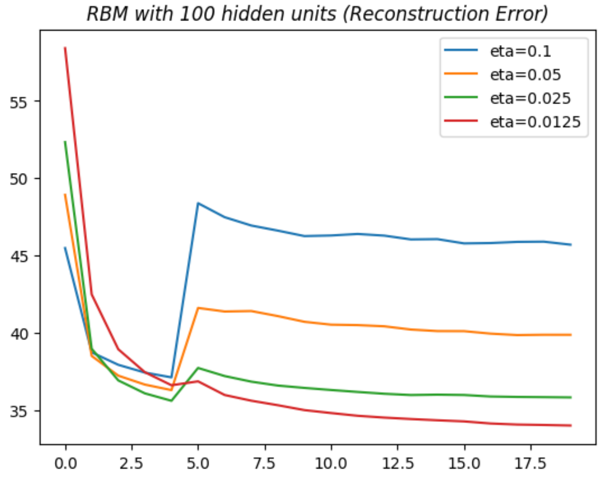

Figure 1.16. Average training time for 20 epochs and 100 hidden units is 5.2 minutes.

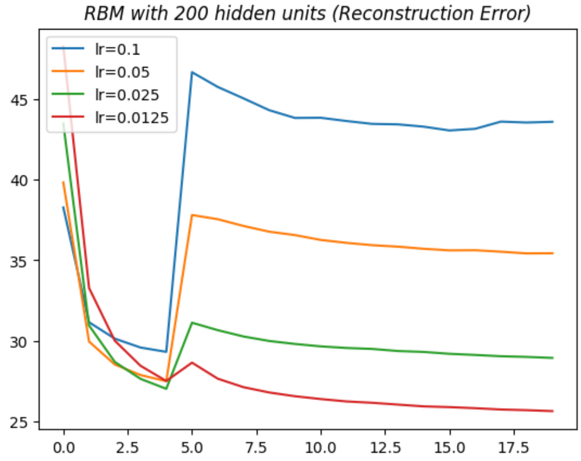

Figure 1.17. Average training time for 200 hidden units is 11 minutes.

References

[1] A. M. Turing, On Computable Numbers, with an Application to the ENTSCHEIDUNGSPROBLEM, 1936

[2] Komninos Dimitrios, Machine Learning with Python and Keras and generative modeling on the Fashion-MNIST dataset with a restricted Boltzmann Machine. github.com/dkomni/machine-learning-overview

[3] John J Hopfield. “Neural networks and physical systems with emergent collective computational abilities.” In: Proceedings of the national academy of sciences 79.8 (1982), pp. 2554–2558.

[4] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. “A learning algorithm for Boltzmann machines”. In: Cognitive science 9.1 (1985), pp. 147– 169.

[5] Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Tech. rep. Colorado Univ at Boulder Dept of Computer Science, 1986.

[6] Y Freund and D Haussler. “Unsupervised learning of distributions on binary vectors using two layer networks (Technical Report UCSC-CRL-94-25)”. In: University of California, Santa Cruz (1994).

[7] Geoffrey E Hinton. “Training products of experts by minimizing contrastive divergence”. In: Neural computation 14.8 (2002), pp. 1771–1800.

[8] Geoffrey E Hinton. “A practical guide to training restricted Boltzmann machines”. In: Neural networks: Tricks of the trade. Springer, 2012, pp. 599–619.