Notes

Luka Skoric

February 4, 2017

Contents

1 Quantum algorithms **2**
- 1.1 Quantum Fourier transform [1] . . . 2
- 1.2 Phase estimation [1] . . . 2
- 1.3 Rejection sampling . . . 3
  - 1.3.1 Classical rejection sampling . . . 3
  - 1.3.2 Quantum rejection sampling [2] . . . 3
- 1.4 Quantum algorithm for linear systems of equations [3] . . . 5
  - 1.4.1 Overview of the algorithm . . . 5
  - 1.4.2 Details . . . 5
  - 1.4.3 Limitations [4] . . . 6

2 Classical machine learning **6**
- 2.1 Introduction . . . 6
- 2.2 Perceptrons . . . 7
- 2.3 Neural networks . . . 8
  - 2.3.1 Model . . . 8
  - 2.3.2 Learning . . . 8
- 2.4 Boltzmann machines [5] . . . 9

3 Second order optimisation **11**
- 3.1 Introduction . . . 11
- 3.2 Newton's method . . . 11
- 3.3 Linear conjugate gradient algorithm . . . 11
- 3.4 Gauss-Newton algorithm . . . 12
- 3.5 Hessian-free optimisation [6] . . . 13
- 3.6 AdaGrad [7] . . . 14
  - 3.6.1 Notation . . . 14
  - 3.6.2 Learning algorithms . . . 15
  - 3.6.3 AdaGrad algorithm . . . 15
- 3.7 Saddle-free Newton method . . . 16

4 Quantum Machine Learning **17**
- 4.1 Introduction . . . 17
- 4.2 Quantum deep learning [5] . . . 17
- 4.3 Quantum Neural Networks . . . 18
  - 4.3.1 Simulating a perceptron on a quantum computer [8] . . . 18

1 Quantum algorithms

1.1 Quantum Fourier transform [1]

The discrete Fourier transform in the usual notation takes an input vector of complex numbers $x_0, \dots, x_{N-1}$ and outputs the transformed data, a vector of complex numbers $y_0, \dots, y_{N-1}$ defined by

$$y_k \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi ijk/N} \tag{1}$$

The quantum Fourier transform is exactly the same transformation on quantum state. It is a linear operator with the following action on an orthonormal basis $|0\rangle, \dots, |N-1\rangle$:

$$|j\rangle \to \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi ijk/N} |k\rangle \tag{2}$$

Equivalently, the action on an arbitrary state is

$$\sum_{j=0}^{N-1} x_j |j\rangle \to \sum_{k=0}^{N-1} y_k |k\rangle \tag{3}$$

Now take $N = 2^n$ for some integer n and the basis $|0\rangle, \dots, |N-1\rangle$ to be the computational basis for an n qubit quantum computer. We can write j using the binary representation as $j = j_1 j_2 \dots j_n = j_1 2^{n-1} + \dots + j_n 2^0$. Also, adopt the notation $0.j_l j_{l-1} \dots j_m = j_l 2^{-1} + j_{l-1} 2^{-2} \dots + j_m 2^{m-l+1}$.

Then, the quantum Fourier transform can be given the product representation:

$$|j_1 \dots j_n\rangle \to \frac{(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle) \dots (|0\rangle + e^{2\pi i 0.j_1 j_2 \dots j_n}|1\rangle)}{2^{n/2}} \tag{4}$$

This representation allows us to construct an efficient quantum circuit for computing the Fourier transform and provides us with insight into algorithms based upon quantum Fourier transform.

1.2 Phase estimation [1]

Suppose a unitary operator U has an eigenvector $|u\rangle$ with eigenvalue $e^{2\pi i\varphi}$ where the value of φ is unknown. The phase estimation algorithm estimates φ. For the estimation, we assume we have oracles capable of preparing the state $|u\rangle$ and performing the controlled-U^{2^j} operation for suitable non-negative integers j.

The algorithm uses two registers. The first register contains t qubits initially in the state $|0\rangle$ where t is the number of digits we wish to have in our estimate for φ.

The second register begins in the state $|u\rangle$ and contains as many qubits as is necessary to store $|u\rangle$.

The algorithm is performed in two stages:

1. Apply Hadamard transform to the first register, followed by application of controlled-U operations on the second register with U raised to successive powers of two giving us the final state of the first register to be:

$$\frac{1}{2^{t/2}}(|0\rangle+e^{2\pi i 2^{t-1}\varphi}|1\rangle)(|0\rangle+e^{2\pi i 2^{t-2}\varphi}|1\rangle)\dots(|0\rangle+e^{2\pi i 2^{0}\varphi}|1\rangle) = \frac{1}{2^{t/2}}\sum_{k=0}^{2^t-1} e^{2\pi i\varphi k}|k\rangle \tag{5}$$

2. Apply inverse quantum Fourier transformation on the first register. If $\varphi = 0.\varphi_1\dots\varphi_t$, the resulting state is:

$$|\varphi_1\dots\varphi_t\rangle|u\rangle \tag{6}$$

and hence, measuring the first register gives us a good estimate for φ.

However, even if φ can not be written exactly with a t bit binary expansion, the procedure gives us a good approximation for φ with high probability.

1.3 Rejection sampling

1.3.1 Classical rejection sampling

In the classical case of rejection sampling we suppose that we have a distribution Q with a density function $q(x), x \in [a,b]$ and a function $\tilde{p}(x), x \in [a,b]$. Also, suppose we know κ s.t. $q(x) \geq \kappa\tilde{p}(x), \forall x \in [a,b]$. We want to sample from a distribution P with density function $p(x) \propto \tilde{p}(x)$.

We can do this by taking samples from random variable $X \sim Q$ and uniformly distributed random variable $u \sim U(0,1)$ and accepting them if $uq(x) \leq \kappa\tilde{p}(x)$ and rejecting otherwise. Then we have:

$$\begin{aligned}
\mathbb{P}(x \leq y | x \text{ accepted}) &= \frac{\mathbb{P}(x \leq y, x \text{ accepted})}{\mathbb{P}(x \text{ accepted})} \\
&= \frac{\int_a^y \mathrm{d}x \int_0^1 \mathrm{d}u\ q(x)\mathbb{P}(uq(x) \leq \kappa\tilde{p}(x))}{\int_a^b \mathrm{d}x \int_0^1 \mathrm{d}u\ q(x)\mathbb{P}(uq(x) \leq \kappa\tilde{p}(x))} \\
&= \frac{\int_a^y \mathrm{d}x\ q(x)\frac{\kappa\tilde{p}(x)}{q(x)}}{\int_a^b \mathrm{d}x\ q(x)\frac{\kappa\tilde{p}(x)}{q(x)}} \\
&= \frac{\int_a^y \tilde{p}(x)\,\mathrm{d}x}{\int_a^b \tilde{p}(x)\,\mathrm{d}x}
\end{aligned}$$

Therefore, $\mathbb{P}(x \leq y | x \text{ accepted})$ has a required distribution.

1.3.2 Quantum rejection sampling [2]

Suppose we have vectors $\boldsymbol{\pi}, \tilde{\boldsymbol{\sigma}} \in \mathbb{R}^n$ with non negative entries and $\|\boldsymbol{\pi}\| = 1$. Also, have an oracle O with action on $|\bar{0}\rangle_{dn} \equiv |\underbrace{00\dots0}_{dn}\rangle$ defined by:

$$O|\bar{0}\rangle_{dn} = |\pi\rangle \equiv \sum_{k=1}^{n} \pi_k|\xi_k\rangle|k\rangle \tag{7}$$

and an arbitrary definition on the orthogonal subset. We want to generate the state $|\sigma\rangle = \sum_{k=1}^{n} \sigma_k |\xi_k\rangle|k\rangle$ s.t. $\boldsymbol{\sigma} \propto \tilde{\boldsymbol{\sigma}}$ and $\|\boldsymbol{\sigma}\| = 1$. Also, suppose we know κ s.t. $\kappa\pi_k \geq \tilde{\sigma}_k \ \forall k = 1, 2 \dots n$. Note that the states $|\xi_k\rangle$ are unknown which makes the problem non-trivial.

Define $\epsilon_k = \frac{\tilde{\sigma}_k}{\kappa\pi_k}$ and $\epsilon_k = 0$ for k s.t. $\pi_k = 0$. Note that $\epsilon_k \leq 1 \ \forall k = 1, 2 \dots n$. Also, have

$$R_\epsilon(k) := R_y(2\arcsin(\epsilon_k)) = \begin{pmatrix} \sqrt{1-\epsilon_k^2} & -\epsilon_k \\ \epsilon_k & \sqrt{1-\epsilon_k} \end{pmatrix} \quad \text{(in basis } \{|0\rangle, |1\rangle\}) \tag{8}$$

and

$$R_\epsilon := \sum_{k=1}^{n} |k\rangle\langle k| \otimes R_\epsilon(k) \tag{9}$$

The procedure goes as follows:

1. Start with $|\bar{0}\rangle_{dn}|0\rangle$ and apply $(O \otimes I_2)$:

$$(O \otimes I_2)|\bar{0}\rangle_{dn}|0\rangle = |\pi\rangle|0\rangle = \sum_{k=1}^{n} \pi_k |\xi_k\rangle|k\rangle|0\rangle \tag{10}$$

2. Apply $I_d \otimes R_\epsilon$:

$$(I_d \otimes R_\epsilon)|\pi\rangle|0\rangle = \sum_{k=1}^{n} \pi_k |\xi_k\rangle|k\rangle(\sqrt{1-\epsilon_k^2}|0\rangle + \epsilon_k|1\rangle) \tag{11}$$

3. Measure the last register and if 1 is measured, accept the state. Otherwise reject the state and repeat the procedure. The collapsed accepted state is then

$$|\Psi\rangle \propto \sum_{k=1}^{n} \pi_k \epsilon_k |\xi_k\rangle|k\rangle|1\rangle \tag{12}$$

Therefore, $|\Psi\rangle = \sum_{k=1}^{n} \sigma_k |\xi_k\rangle|k\rangle|1\rangle$

Using the definition of ϵ and the fact that $\boldsymbol{\sigma}$ is a unit vector we have that the acquired state is $|\Psi\rangle = |\sigma\rangle|1\rangle$.

The probability of successfully accepting a state is $q = \|\boldsymbol{\epsilon}\|^2$. In practice, we can use amplitude amplification algorithm to get the required state in $O(\frac{1}{\sqrt{q}})$. Operator used in the amplitude amplification is

$$S := \text{ref}_{|\Psi\rangle} \cdot \text{ref}_{|1\rangle} \tag{13}$$

where "ref" are reflections

$$\text{ref}_{|\Psi\rangle} := (I_d \otimes R_\epsilon)\text{ref}_{|\pi\rangle|0\rangle}(I_d \otimes R_\epsilon)^\dagger \tag{14}$$

$$\text{ref}_{|1\rangle} := I_{dn} \otimes Z \tag{15}$$

and $\text{ref}_{|\pi\rangle|0\rangle}$ can be obtained by the usage of oracle:

$$\text{ref}_{|\pi\rangle|0\rangle} = (O \otimes I_2)(I_{dn} \otimes I_2 - 2|\bar{0}\rangle_{dn}|0\rangle\langle\bar{0}|_{dn}\bar{0})(O \otimes I_2)^\dagger \tag{16}$$

1.4 Quantum algorithm for linear systems of equations [3]

Given a Hermitian $N \times N$ matrix A and a unit N dimensional vector b, suppose we would like to find x satisfying $Ax = b$.

1.4.1 Overview of the algorithm

1. Represent b as $|b\rangle = \sum_{i=1}^{N} b_i|i\rangle$
2. Use techniques of Hamiltonian simulation to apply e^{iAt} to $|b\rangle$
3. Use phase estimation algorithm (1.2) to decompose $|b\rangle$ in the basis of eigenstates of A and find the corresponding eigenvalues λ_j. The state of the system after this stage is close to $\sum_{j=1}^{N} \beta_j|u_j\rangle|\lambda_j\rangle$ where u_j is the eigenvector basis of A and $|b\rangle = \sum_{j=1}^{N} \beta_j|u_j\rangle$.
4. Perform the linear map $|\lambda_j\rangle \to C\lambda_j^{-1}|\lambda_j\rangle$ where C is a normalising constant. This is not unitary, so the operation has some probability of failing.
5. Uncompute the $|\lambda_j\rangle$ register and we are left with the state proportional to $\sum_{j=1}^{N} \beta_j\lambda_j^{-1}|u_j\rangle = A^{-1}|b\rangle$.

The algorithm has runtime of $O(\log(N)s^2\kappa^2/\epsilon)$ where matrix A has at most s nonzero entries per row, κ is the ratio between the largest and the smallest eigenvalue of A and ϵ is the error achieved in the output state $|x\rangle$. In comparison, the best known classical algorithm has a runtime of $O(Ns\sqrt{\kappa}\log(1/\epsilon))$

1.4.2 Details

First, we want to transform a given Hermitian matrix A into a unitary operator e^{iAt} which we can then apply as necessary. If A has at most s nonzero entries per row and given a row index, these entries can be computed in time $O(s)$ then we can use an algorithm for simulating sparse Hamiltonians which simulates e^{iAt} in time $O(\log(N)s^2t)$.

Also, suppose we have an efficient procedure for preparing $|b\rangle = \sum_{i=1}^{N} b_i|i\rangle$. Denote by $|u_j\rangle$ eigenvectors of A and by λ_j the corresponding eigenvalues. Let

$$|\Psi_0\rangle = \sqrt{\frac{2}{T}}\sum_{\tau=0}^{T-1}\sin(\frac{\pi(\tau+\frac{1}{2})}{T})|\tau\rangle \tag{17}$$

for some large T.

We apply the conditional Hamiltonian evolution $\sum_{\tau=0}^{T-1}|\tau\rangle\langle\tau| \otimes e^{iAt_0/T}$ on $|\Psi_0\rangle \otimes |b\rangle$ where $t_0 = O(\kappa/\epsilon)$ and then perform the quantum Fourier transform (1.1) on the first register. The acquired state is:

$$\sum_{j=1}^{N}\sum_{k=0}^{T-1}\alpha_{kj}\beta_j|k\rangle|u_j\rangle \tag{18}$$

where $|k\rangle$ are the Fourier basis states and $|\alpha_{kj}|$ is large iff $\lambda_j \approx \frac{2\pi k}{t_0}$. Define $\tilde{\lambda_k} \equiv \frac{2\pi k}{t_0}$ and relabel the first register to $\tilde{\lambda_k}$. If the phase estimation was perfect,

we would have $\alpha_{kj} = 1$ if $\lambda_k = \lambda_j$ and 0 otherwise. Assuming that, we have the state:

$$\sum_{j=1}^{N} \beta_j |\lambda_j\rangle |u_j\rangle \tag{19}$$

Now we do rejection sampling by adding an ancilla qubit and rotating conditioned on $|\lambda_j\rangle$

$$\sum_{j=1}^{N} \beta_j |\lambda_j\rangle |u_j\rangle \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_k} |1\rangle \right) \tag{20}$$

We can now perform a measurement on the ancilla qubit and conditioned on seeing 1, we have the final state

$$|x\rangle \propto \sum_{j=1}^{N} \beta_j \lambda_j^{-1} |u_j\rangle \tag{21}$$

as required.

1.4.3 Limitations [4]

1. We have to be able to load the vector $|b\rangle$ sufficiently quickly. If it is being loaded from the classical data, it is essential that b is relatively uniform.
2. Quantum computer has to be able to apply unitary transformations e^{iAt} for various values of t. For this, it is important that A is sparse.
3. The matrix A needs to be robustly invertible as the performance of the algorithm depends linearly on $\kappa = |\lambda_{max}/\lambda_{min}|$
4. The algorithm outputs the state $|x\rangle$. Hence, learning the value of a specific entry x_i requires repeating the algorithm roughly n times. However, the state can be used to reveal some limited statistical information about x instead.

2 Classical machine learning

2.1 Introduction

Machine learning explores the construction and study of algorithms that can learn from and make predictions on data. In general, machine learning is divided in three categories:

- **Supervised learning**: The computer is presented with example inputs and their outputs. The goal is to generate a map which will map the inputs to the outputs.
- **Unsupervised learning**: The computer is given unlabelled data and is required to find structure in the input.
- **Reinforced learning**: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle), without a teacher explicitly telling it whether it has come close to its goal or not.

We will mostly focus on supervised learning here. Given a set of N training examples of the form $\{(x_1, y_1), \dots, (x_N, y_N)\}$ such that x_i is the vector containing features of the i-th example and y_i is the given output, a learning algorithm seeks a 'hypothesis' function $h : X \to Y$ where X is an input and Y output space and h belongs to H, some set of possible function. To measure how well $h \in H$ fits some given labelled example, we use the loss function $J : Y \times Y \to \mathbb{R}^{\geq 0}$. The risk $R(h)$ is defined as the expected loss of h. This can be estimated from the training data as:

$$R(h) = \frac{1}{N} \sum_{i=1}^{N} J(h(x_i), y_i) \tag{22}$$

Usually, we pick the hypothesis to depend on some weights: $H = \{h(x; w) | w \in W \subset \mathbb{R}^n\}$. We can then update the weight vector based on the training set to reduce our risk R. This can be done by e.g. gradient descent:

$$w_i(t+1) = w_i(t) - \eta \frac{1}{N} \sum_{i=1}^{N} \frac{\partial J(h(x_i; w(t)), y_i)}{\partial w_i} \tag{23}$$

There are two most common problems we have to overcome to end up with a good hypothesis function: underfitting and overfitting. Underfitting happens when our hypothesis set is too 'restricted'. I.e. there is no function in G which will perform well on the training data. On the other side, overfitting is the problem we have when the hypothesis performs too well on the training set modeling the errors training set data has as well as the required function making its generalisation on further examples poor.

2.2 Perceptrons

Perceptrons are the most basic type of a linear classifier. Although limited in power, they form a basis to full artificial neural networks used in machine learning today.

The model of a perceptron is inspired by signal processing between neural cells in brain. Neurons can be in either of the two states: 'active' or 'resting'. In a perceptron we have $n+1$ input nodes $x_i \in \{-1, , 1\}, i = 0, 1, \dots n$ (where x_0 is constantly active bias node) which are connected to the output node y with a connection of strength w_i (2.2). The output is then calculated via an activation function

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq 0, \\ -1 & \text{otherwise} \end{cases} \tag{24}$$

Perceptrons can be used to learn how to classify an input based on examples by initialising $x_1, \dots x_n$ with a number of example inputs, comparing the acquired output with the target output and adjusting accordingly by e.g gradient descent:

$$w_i(t+1) = w_i(t) + \eta \left(d^{(j)} - y^{(j)}(t) \right) x_i^{(j)} \tag{25}$$

where $x_i^{(j)}$ is a j-th example with the target value $d^{(j)}$ and $y^{(j)}(t)$ is the output calculated with the weights $w_i(t)$.

It can be shown that a single perceptron can only classify the linearly separable functions. However, combining several layers of perceptrons gives us a much more powerful artificial neural network.

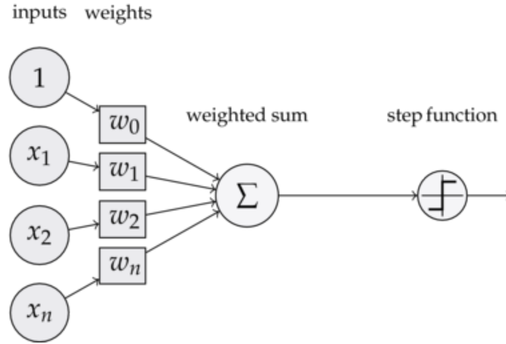

Figure 1: Illustration of a perceptron

2.3 Neural networks

2.3.1 Model

Neural networks are a powerful classification algorithm with wide applications in I.T. They are based on multiple perceptrons with a sigmoidal activation function. A typical neural network consist of given input neurons $x \in \mathbb{R}^n$, a number of hidden layers and an output layer (2.3.1) We denote the activation of i-th unit in the l-th layer by $a_i^{(l)}$ with input $a_i^{(1)} = x_i$ and output $y_i = a_i^{(L)}$. Moreover, the connection strength between i-th unit in l-th layer and j-th unit in layer $l+1$ is $\Theta_{ij}^{(l)}$. Then, the activation $a_i^{(l+1)}$ is calculated by

$$a_i^{(l+1)} = \sigma\left(\sum_j \Theta_{ij}^{(l)} a_j^{(l)}\right) \tag{26}$$

where $\sigma = \frac{1}{1+e^{-x}}$ is the sigmoidal function.

The output of the neural network is going to be the composition of sigmoidal function which gives us a highly non-linear behaviour making the hypothesis space very versatile.

We model our examples in a such way that the classification output in the training set is a vector of binary units ($y \in Y = \{0,1\}^K$). Our hypothesis function gives values in $[0,1]^K$ which we can then use to predict a vector in Y by e.g. taking the entry i with $h_\Theta(x)_i$ maximal to be 1 and all others 0.

2.3.2 Learning

True power of the neural network lies in the ability to train them easily. For the cost function we choose

$$J(h_\Theta(x), y) = -\sum_{i=1}^{K} (y_i \log(h_\Theta(x))_i + (1-y_i)\log(1-(h_\Theta(x))_i) \tag{27}$$

We learn the parameters $\Theta_{ij}^{(l)}$ using the gradient descent. Although derivative of the cost function may seem complicated considering that the hypothesis is a large composition of sigmoidal functions, there is a simple recurrence relation which lets us calculate all derivatives by the method of backpropagation:

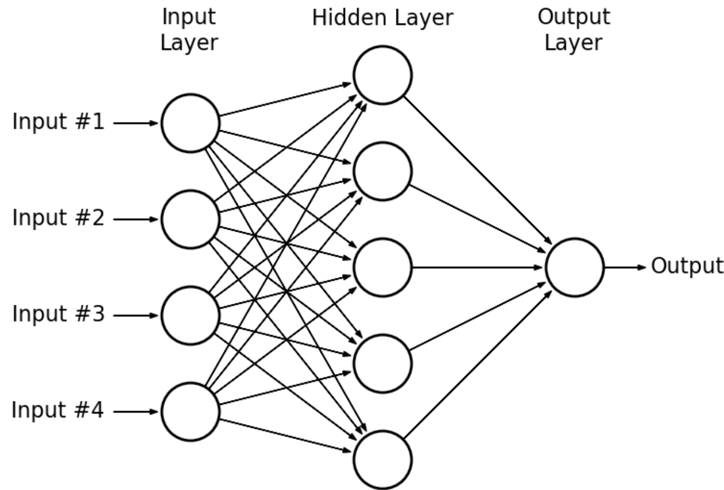

Figure 2: A typical neural network

1. We start by the forward propagation to calculate the hypothesis function:

$$a^{(1)} = x$$
$$a_i^{(l+1)} = \sigma(\Theta_{ij}^{(l)} a_j^{(l)})$$

2. Having a hypothesis $h_\Theta(x) = a^{(L)}$, the activation units in all layers and the target output y we use the following recurrence relation to compute $\delta_i^{(l)}$ which loosely saying assign a local error to each weight:

$$\delta^{(L)} = a^{(L)} - y$$
$$\delta^{(l-1)} = ((\Theta^{(l-1)})^T \delta^{(l)}) .* a^{(l-1)} .* (1 - a^{(l-1)})$$

   where $.*$ denotes element-wise multiplication.

3. Finally, having now $\delta^{(l)}$ for $l = 2, 3, \ldots, L$, the derivative of the cost function we need for the gradient descent (and most other more advanced optimisation algorithms) can be shown to be:

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = \delta_i^{(l+1)} a_j^{(l)} \tag{28}$$

2.4 Boltzmann machines [5]

Boltzmann machines (BMs) are a class of deep networks which provide a generative model for the data, meaning that they provide the probability of each possible visible vector appearing. The set of nodes that encode the observed data and the output are called the visible units (v) and the nodes used to model the feature space are called hidden units (h) (2.4).

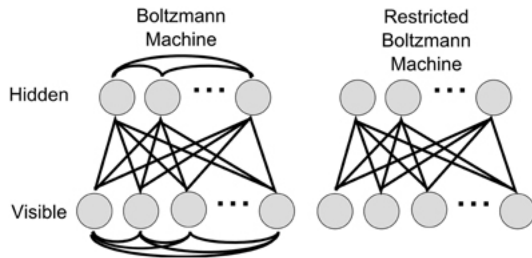

Figure 3: Full and restricted Boltzmann machine

In a Boltzmann machine, each node has assigned an energy function $E(v,h)$:

$$E(v,h) = -\sum_i v_i b_i - \sum_j h_j d_j - \sum_{i,j} w_{ij}^{vh} v_i h_j - \sum_{i,j} w_{ij}^{v} v_i v_j - \sum_{i,j} w_{ij}^{h} h_i h_j \quad (29)$$

where b and d are biases and $w_{ij}^{vh}, w_{ij}^{v}, w_{ij}^{h}$ are weights assigning penalties for correlations between units. Define $w = [w_{ij}^{vh}, w_{ij}^{v}, w_{ij}^{h}]$ and let n_v and n_h be the numbers of visible and hidden units, respectively.

A Boltzmann machine models the probability of a given configuration of visible and hidden units by the Gibbs distribution:

$$P(v,h) = \frac{e^{-E(v,h)}}{Z} \quad (30)$$

where Z is the normalising factor, also called the partition function.

When training a BM we are trying to make the model we are generating fit the given data as well as possible. It can be proven, using Bayesian statistics, that that is equivalent to maximising the objective function

$$O := \frac{1}{N_{train}} \sum_{v \in X_{train}} \log(\sum_h P(v,h)) - \frac{\lambda}{2} w^T w \quad (31)$$

where N_{train} is the size of the training set, x_{train} is the set of training vectors, and λ is a regularization term. We do this by using gradient ascent where the gradient is:

$$\frac{\partial O}{\partial w_{ij}} = \langle x_i x_j \rangle_{Data} - \langle x_i x_j \rangle_{model} - \lambda w_{ij} \quad (32)$$

where $\langle x_i x_j \rangle_{Data}$ is the expected correlation when visible units are fixed to the training examples and $\langle x_i x_j \rangle_{model}$ is the expected correlation with free visible and hidden units.

Note that the partition function Z has exponential number of terms and hence, is hard to calculate directly. As a consequence of that, we need to employ different methods to approximate the terms in the gradient. Classically, this can be done by either the Monte Carlo method or rejection sampling (1.3.1).

3 Second order optimisation

3.1 Introduction

Standard gradient descent tends to progress very slowly in deep neural networks, most often because flatter regions with low curvature are explored much more slowly than they should be. Regions like that create false impressions of local minima halting the progress of the algorithm. To improve our optimisation we have adjust the gradient descent. In this sections we discuss different ways this can be done.

3.2 Newton's method

Newton's method, like gradient descent, is an optimization algorithm which iteratively updates the parameters $\theta \in \mathbb{R}^N$ of an objective function f by computing search directions p and updating θ as $\theta + \alpha p$ for some learning rate α.

The central idea is to approximate f up to the 2nd order:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^T p + \frac{1}{2} p^T B p \tag{33}$$

where $B = H(\theta)$ is the Hessian matrix of f at θ. The good search direction is then the one that minimises $q_\theta(p)$

The approximation is only good for small values of p and this quadratic might not have a minimum (if Hessian indefinite). Hence, in practice H is re-conditioned to $B = H + \lambda I$ for some constant $\lambda \geq 0$.

The quadratic $q_\theta(p)$ as defined in the equation 33 has a unique stationary point at

$$p = -B^{-1} \nabla f(\theta) \tag{34}$$

This inspires the generalised gradient descent where we update the parameters θ by

$$\theta(t+1) = \theta(t) - \eta A^{-1} \nabla f(\theta) \tag{35}$$

for some matrix A and learning rate η. Note that by taking A to be the identity matrix we recover the standard gradient descent.

3.3 Linear conjugate gradient algorithm

For Newton's method to work in practice, we have to have an efficient way of finding p which is equivalent to solving the equation $Ax = b$ where $x, b \in \mathbb{R}^N$ and A is $N \times N$ symmetric, positive definite real matrix. We denote the unique solution to the equation by x_*.

We say that two vectors are conjugate if they are orthogonal w.r.t. the inner product $\langle ., . \rangle_A$. Then, we can find a basis of conjugate vectors $P = \{p_k, k = 1, 2 \ldots n | \langle p_i, p_j \rangle = 0 \; \forall i \neq j\}$ in which we can expand x_*:

$$x_* = \sum_{i=1}^{n} \alpha_i p_i \tag{36}$$

Plugging into the equation and taking an inner product with p_k, we get:

$$\alpha_k = \frac{\langle p_k, b \rangle}{\|p_k\|_A^2} \tag{37}$$

We choose the vectors p_k carefully, so we don't need all of them to obtain a good approximation to x_*.

Start with some initial guess x_0. This means we will take $p_0 = b - Ax_0$. Let r_k be the residual at the kth step:

$$r_k = b - Ax_k \tag{38}$$

Similar to the Gram-Schmidt orthogonalisation, we then take

$$p_k = r_k - \sum_{i<k} \frac{\langle p_i, r_k \rangle_A}{\langle p_i, p_i \rangle_A} p_i \tag{39}$$

Then, the next optimal direction is

$$x_{k+1} = x_k + \alpha_k p_k \tag{40}$$

with

$$\alpha_k = \frac{\langle p_k, b \rangle}{\langle p_k, p_k \rangle_A} = \frac{\langle p_k, r_{k-1} \rangle}{\langle p_k, p_k \rangle_A} \tag{41}$$

Moreover, it can be proven that r_{k+1} is conjugate to all p_i with $i < k$. Therefore, the algorithm we acquire is the repetition of the following steps:

$$\alpha_k = \frac{\|r_k\|^2}{\|p_k\|_A^2}$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k A p_k$$

$$\beta_k = \frac{\|r_{k+1}\|^2}{\|r_k\|^2}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k$$

The algorithm produces the exact solution in at most N steps. However, the method provides a monotonically improving approximations x_k to the exact solution, which may reach the required tolerance after a relatively small (compared to the problem size) number of iterations.

3.4 Gauss-Newton algorithm

The Hessian matrix might be indefinite and also hard to calculate. Hence, we often don't use the full Hessian for A in the generalised gradient descent as defined in the equation 35. Instead, we use different approximations which attempt to capture the main characteristics of the Hessian.

Here we develop Gauss-Newton method to find a simple approximation to the Hessian which exploits the fact that the objective functions are often sums of squares. The approximation is always positive definite and can be calculated directly from the derivatives of the cost function.

As before, the Newton's method for minimising a function F of parameters θ is

$$\theta(t+1) = \theta(t) - H^{-1}g \tag{42}$$

where g is the gradient of F and H denotes the Hessian matrix of F. Suppose F is the sum of squares: $F = \sum_{i=1}^{m} f_i(\theta)^2$. Then the gradient is given by:

$$g_j = 2\sum_{i=1}^{m} f_i \frac{\partial f_i}{\partial \theta_j} \tag{43}$$

and the Hessian is

$$H_{jk} = 2\sum_{i=1}^{m} \left(\frac{\partial f_i}{\partial \theta_j}\frac{\partial f_i}{\partial \theta_k} + f_i \frac{\partial^2 f_i}{\partial \theta_j \partial \theta_k} \right) \tag{44}$$

The Gauss–Newton method is obtained by ignoring the second-order derivative terms (the second term in this expression). That is, the Hessian is approximated by

$$H_{jk} \approx 2\sum_{i=2}^{m} J_{ij} J_{ik} \tag{45}$$

where $J_{ij} = \frac{\partial f_i}{\partial x_j}$ is the Jacobian matrix.

3.5 Hessian-free optimisation [6]

Optimising $q_\theta(p)$ by solving the equation $Bp = -\nabla f(\theta)$ directly is computationally expensive. Even storing N^2 elements of the matrix B in the working memory can take significant resources. Hence, this is not reasonable even for modestly sized neural networks. However, we can exploit properties of the objective function to make this substantially faster.

Firstly, note that for any N-dimensional vector d, Hd can be easily computed using single extra gradient evaluation via:

$$Hd = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon d) - \nabla f(\theta)}{\epsilon} \tag{46}$$

and taking $B = H + \lambda I$ instead is equivalent to adding λd to the above equation.

Secondly, the linear conjugate gradient algorithm (CG) (3.3) is a very effective algorithm for optimizing quadratic objectives and it requires only matrix-vector products with B. In the worst case, the algorithm takes N iterations to completely converge. However, it makes a significant progress even after a much more practical number of iterations.

Moreover, it is important to choose the appropriate value of λ which controls how conservative the approximation is preventing us to go out of the region where approximation (46) is valid. A simple way of updating λ is given by:

$$\textbf{If} \quad \rho < \frac{1}{4} \quad \text{then} \quad \lambda \leftarrow \frac{3}{2}\lambda \quad \textbf{elseif} \quad \rho > \frac{3}{4} \quad \text{then} \quad \lambda \leftarrow \frac{2}{3}\lambda \tag{47}$$

where ρ is the reduction ratio which measures the accuracy of the quadratic approximation:

$$\rho = \frac{f(\theta + p) - f(\theta)}{q_\theta(p) - q_\theta(0)} \tag{48}$$

For some learning models such as neural networks there is an efficient procedure for computing the product Hd exactly using the algorithm similar to back-propagation in implementation and complexity. Also, Gauss-Newton approximation to the Hessian yields very good results in cases where it can be applied.

3.6 AdaGrad [7]

3.6.1 Notation

- $\partial f(x)$ denotes a subdifferential set of a function f at x and a particular vector in the set is $g_t \in \partial f(x)$
- $\langle x, y \rangle$ is the inner product between x and y. The Bregmann divergence associated with the strongly convex and differentiable function ψ is

$$B_\psi(x, y) = \psi(x) - \psi(y) - \langle \nabla \psi(y), x - y \rangle \tag{49}$$

- Define the outer product matrix: $G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^T$
- $\|.\|$ is the Euclidean norm while $\|.\|_A = \langle ., A. \rangle$ is the Mahalanobis norm
- $\prod_{\mathcal{X}} a \equiv \text{argmin}_{x \in \mathcal{X}} \|x - a\|$ is the projection onto $\mathcal{X}$ using Euclidean norm while $\prod_{\mathcal{X}}^{A} a \equiv \text{argmin}_{x \in \mathcal{X}} \|x - a\|_A$ is the projection onto $\mathcal{X}$ according to A

In online learning, in each step the learner predicts a weight vector $x_t \in \mathcal{X} \subseteq \mathbb{R}^d$ and is then given an error function $f_t(x)$ (e.g. $f_t(x) = (h_x(z_t) - y_t)^2$ where z_t is the next data vector, y_t it's label and $h_x(z)$ is the hypothesis function). The learner's goal is to achieve a low regret $R(T)$:

$$R(T) = \sum_{t=1}^{T} f_t(x_t) - \inf_{x \in \mathcal{X}} \sum_{t=1}^{T} f_t(x) \tag{50}$$

At every timestep the learner also receives the gradient information $g_t \in \partial f(x)$ which it can then use to update x_t.

We consider online learning with the sequence of composite functions $\phi_t(x) = f_t(x) + \varphi(x)$ where f_t is the instantaneous loss and φ is the regularization function and both are convex.

The regret w.r.t fixed (optimal) predictor x^* is then:

$$R_\phi(T) = \sum_{t=1}^{T} (\phi_t(x_t) - \phi(x^*)) \tag{51}$$

The goal is to devise algorithms which have asymptotically sub-linear regret.

3.6.2 Learning algorithms

The standard gradient descent employs an update of the predictor which moves it in the opposite direction of the gradient g_t while maintaining $x_{t+1} \in \mathcal{X}$:

$$x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} \|x - (x_t - \eta g_t)\|^2 \tag{52}$$

Note that the norm in the equation above can be written as

$$\|(x - x_t) + \eta g_t\|^2 = \|x - x_t\|^2 + 2\eta \langle g_t, x \rangle + M_t \tag{53}$$

where M_t is independent of x and can hence be ignored. Hence, the gradient descent becomes:

$$x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} \left(\|x - x_t\|^2 + 2\eta \langle g_t, x \rangle \right) \tag{54}$$

Which we can consider as a trade-off between going down the direction of the gradient of the cost function and staying close to x_t regularised by the learning rate.

- **Composite mirror descent** [9] is the generalisation of the gradient descent which attempts to take into account the geometry of the space and also treats regularisation separately. The measure of distance is generalised by the Bregmann divergence to take the geometry of the space into the account:

  $$x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} \left(B_\psi(x, x_t) + \eta \langle g_t, x \rangle \right) \tag{55}$$

  Where ψ is some strictly convex proximal function which has to be chosen.

  Moreover, if we also have regularisation term φ in the objective function: $\phi_t(x) = f_t(x) + \varphi(x)$, just ignoring the composite structure and using the equation above, can produce some undesirable effects. Hence, the composite objective mirror descent treats regularisation term separately by:

  $$x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} (B_\psi(x, x_t) + \eta \langle \nabla(f_t(x_t), x \rangle + \eta \varphi(x)) \tag{56}$$

  This algorithm can be equally applied to both online learning and batch methods.

- **Primal-dual subgradient method** is similar to the composite mirror descent and employs the update:

  $$x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} \left(\frac{1}{t} \psi_t(x) + \eta \langle \frac{1}{t} \sum_{\tau=1}^{t} g_\tau, x \rangle + \eta \varphi(x) \right) \tag{57}$$

  Where ψ_t is some strictly convex proximal function

3.6.3 AdaGrad algorithm

AdaGrad is an algorithm for online learning which takes into account the geometry of the data observed in earlier iterations to emphasise infrequently occurring features which tend to be highly informative.

The AdaGrad generalization to the gradient descent (52) employs the update:

$$x_{t+1} = \prod_{\mathcal{X}}^{G_t^{1/2}} (x_t - \eta G_t^{-1/2} g_t) \tag{58}$$

This is not practically very useful as it requires calculation of the root of the outer product matrix. However, a good approximation with linear computation time is given by:

$$x_{t+1} = \prod_{\mathcal{X}}^{\text{diag}(G_t)^{1/2}} (x_t - \eta \text{diag}(G_t)^{-1/2} g_t) \tag{59}$$

AdaGrad method adapts the proximal function in algorithms such as composite mirror descent (56) and primal-dual subgradient (57). We use Mahalanobis norm to define the proximal function $\psi_t(x) = \langle x, H_t x \rangle$. H_t is then updated by:

$$H_t = \delta I + \text{diag}(G_t)^{1/2} \text{ and } H_t = \delta I + G_t^{1/2} \tag{60}$$

for some small fixed $\delta \geq 0$ which can in practice be set to 0 (more on δ)

3.7 Saddle-free Newton method

Although the regular Newton's method (3.2) provides a significant improvement when trying to explore large, flat regions, it doesn't work well when the Hessian matrix is not positive definite which can lead to the method getting stuck in the saddle points. Intuitively, if we are in the basis of eigenstates, then the step $p = -H^{-1}\nabla f(\theta)$ (33) is equivalent to dividing each component of the gradient with it's eigenvalue, making the step in the direction of low curvature large. However, if an eigenvalue is negative, we are also changing the sign of the gradient, making it choose the wrong direction altogether. We can take $B = H + \alpha I$ as discussed earlier. However, then this can limit our movement in the important directions as well.

From the results of random matrix theory, it can be proven that the expected number of saddle points increases exponentially with the dimension of the space. Hence, the problem of getting stuck in a saddle point becomes increasingly important as the dimension of the network increases.

The natural thing to do to solve this problem is to divide by the absolute values of the eigenvalues instead. We can prove why this works by taking a generalised version of the Newton's method. Define $\mathcal{T}_k(f, \theta, \delta\theta)$ to be the k-th order Taylor series expansion of f around θ evaluated at $\theta + \delta\theta$. Then we take the Newton update, but restrict to keeping our new weight vector within some trust region:

$$\begin{aligned} \delta\theta \; &= \; \underset{\delta\theta}{\text{argmin}} \, \mathcal{T}_k(f, \theta, \delta\theta) \text{ with } k \in \{1, 2\} \\ &\text{s.t. } d(\theta, \theta + \delta\theta) \leq \Delta \end{aligned} \tag{61}$$

Note that we can recover the regular Newton method by taking $k = 2$ and $d(\theta, \theta + \delta\theta) = \|\theta\|_2^2$, where Δ is implicitly function of α.

For the Saddle-free Newton method, we take $k = 1$ and d to be the difference between the first and the second order Taylor expansions of f:

$$d(\theta, \theta + \delta\theta) = \frac{1}{2}|\delta\theta^T H \delta\theta| \leq \Delta \tag{62}$$

It can be proven that $|x^T A x| \leq x^T |A| x$ where $|A|$ is the matrix obtained by taking the absolute value of each eigenvalue of A.

We can get an approximation of $|H|$ by finding Krylov subspace vectors through Lanczos iteration of the Hessian.

maybe a few more details

This method allows us to keep advantages of the Hessian free method, but doesn't get stuck in the saddle points which becomes an important advantage in bigger networks.

4 Quantum Machine Learning

4.1 Introduction

There are two different approaches to implementing and, hopefully, speeding up machine learning algorithms using quantum computers:

1. Using the quantum algorithms to perform optimisation tasks to speed up learning or some other type of data processing within the algorithm, but keeping the model classical (e.g. neural network or Boltzmann machine). The example of this is the Quantum Deep Learning paper (4.2).

2. Use quantum computers to devise learning models which might have different properties from the know algorithms. There have been a few attempts to construct quantum neurons, but it has no yet been proven that those models offer improvement over the classical neural networks.

4.2 Quantum deep learning [5]

As discussed in the Boltzmann machines section (2.4) the gradient of the loss function of a Boltzmann machine is:

$$\frac{\partial O}{\partial w_{ij}} = \langle x_i x_j \rangle_{Data} - \langle x_i x_j \rangle_{model} - \lambda w_{ij} \tag{63}$$

This can be roughly approximated by the Monte Carlo method. However, acquiring the exact value of the gradient is computationally expensive as the partition function has exponential number of terms. The Quantum Deep Learning paper proposes a method which speeds up the process of finding the probabilities

$$P(v, h) = \frac{e^{-E(v,h)}}{Z} \tag{64}$$

The goal of the algorithm is to create the state $|\Psi\rangle = \sum_{v,h} \sqrt{P(v, h)} |v\rangle |h\rangle$ which we can measure and by repeating the process we can approximate the expectation values required for the gradient.

We take the Mean-field(MF) approximation $Q(v,h)$ that minimizes Kullback-Leiber divergence $KL(Q\|P) = \sum_{v,h} Q(v,h)\log(\frac{Q(v,h)}{P(v,h)})$:

$$Q(v,h) = (\prod_i \mu_i^{v_i}(1-\mu_i)^{1-v_i})(\prod_j \nu_j^{h_j}(1-\nu_j^{1-h_j}) \quad (65)$$

with

$$\mu_i = \sigma(-b_i - \sum_j w_{ij}\nu_j)$$
$$\nu_j = \sigma(-d_j - \sum_i w_{ij}\mu_i)$$

where $\sigma = \frac{1}{1+e^{-x}}$. Note that this is not necessary for the algorithm to work and virtually any distribution could work, however, the closer $Q(v,h)$ is to the $P(v,h)$ the higher the success rate of the algorithm

The Mean field state can be efficiently prepared using single qubit rotations. Say that this is preformed by the oracle O.

The estimate of the partition function Z is Z_Q s.t.

$$\log(Z_Q) = \sum_{v,h} Q(v,h)\log(\frac{e^{-E(v,h)}}{Q(v,h)}) \quad (66)$$

Note that $Z_Q \leq Z$ with equality iff $KL(Q\|P) = 0$ (follows from Gibbs inequality).

Assume κ is known such that

$$P(v,h) \leq \frac{e^{-E(v,h)}}{Z_Q} \leq \kappa Q(v,h) \quad (67)$$

Note that using these quantities, we can now use quantum rejection sampling (1.3.2) with oracle O and

$$\pi_{v,h} \to \sqrt{Q(v,h)}, \quad \tilde{\sigma}_{v,h} \to \sqrt{\frac{e^{-E(v,h)}}{Z_Q}}, \quad \sigma_{v,h} \to \sqrt{P(v,h)}, \quad \kappa \to \sqrt{\kappa} \quad (68)$$

.

Note that here we do not have states $|\xi_k\rangle$ which makes the algorithm slightly simpler, but otherwise the same.

Using this algorithm we can acquire the required state$|\Psi\rangle = \sum_{v,h} \sqrt{P(v,h)}|v\rangle|h\rangle$. Moreover, the algorithm works faster than any of the known classical algorithms for estimating the gradient in BM learning.

4.3 Quantum Neural Networks

4.3.1 Simulating a perceptron on a quantum computer [8]

A perceptron can be simulated on a quantum computer using the phase estimation algorithm. First, note that rescaling the weights so that $w_k \in [-1, 1)$ doesn't change the output. From now on assume this rescaling.

Suppose we have a state $|x_1, \ldots, x_n\rangle$. Take a unitary operator $U(w)$ which can be decomposed into a single qubit operators as

$$U(w) = U_n(w_n) \ldots U_1(w_1)U_0 \quad \text{with} \quad U_k(w_k) = \begin{pmatrix} e^{-2\pi i w_k \delta\phi} & 0 \\ 0 & e^{2\pi i w_k \delta\phi} \end{pmatrix} \tag{69}$$

where U_k acts on the input register 's qubit x_k, $\delta\phi = \frac{1}{2n}$ and U_0 adds a global phase of πi. The resulting phase of state $|x_1, \ldots, x_n\rangle$ is then given by $\exp(2\pi i(\delta\phi(w^T x) + 0.5)) = \exp(2\pi i\varphi)$.

Note that $\varphi \in [0, 1)$ which is important as now we can run the phase estimation algorithm to determine approximate value of φ. We only need to know weather φ is greater or smaller than a half as it corresponds directly to the activation rule in a classical perceptron. This can be done using only a small number of qubits [8].

Also, the algorithm can be performed while keeping the weights in the quantum register. The initial state is then:

$$|x_1, \ldots, x_n, W_1^{(1)}, \ldots W_1^{(M)}, \ldots, W_n^{(1)}, \ldots, W_n^{(M)}\rangle = |x; w\rangle \tag{70}$$

where $W_k^{(m)}$ is the m-th digit of the binary representation of w_k s.t.

$$w_k = W_k^{(1)} \frac{1}{2}, \ldots W_k^{(M)} \frac{1}{2^M} \tag{71}$$

with a precision of M.

Then, we can replace $U(w)$ with $\tilde{U} = U_0 \prod_{k=1}^{n} \prod_{m=1}^{M} U_{W_k^{(m)}, x_k}$ where we introduce the controlled two-qubit operator

$$U_{W_k^{(m)}, x_k} = \text{diag}\left(1, 1, e^{-2\pi i \delta\phi \frac{1}{2^m}}, e^{2\pi i \delta\phi \frac{1}{2^m}}\right) \tag{72}$$

It is thought that quantum versions of classical learning algorithms might be used to construct superposition-based learning algorithms in which the training set is represented as a superposition of input data. The superposition could then be leveraged to process the entire training set at once. However, no such algorithms have been found.

References

[1] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2007.

[2] Martin Roetteler Maris Ozols and Jeremie Roland. Quantum rejection sampling. *ACM Transactions on Computation Theory (TOCT)*, 5(3), 2013.

[3] Avinatan Hassidim Aram W. Harrow and Seth Lloyd. Quantum algorithm for linear systems of equations, 2009. arXiv:0811.3171 [quant-ph].

[4] S. Aaronson. Quantum machine learning algorithms: Read the fine print. *Nature Physics*, 2015.

[5] Ashish Kapoor Nathan Wiebe and Krysta M. Svore. Quantum deep learning, 2015. arXiv:1412.3489 [quant-ph].

[6] James Martens. Deep learning via hessian-free optimization. Proceedings of the 27th International Conference on Machine Learning (ICML), 2010.

[7] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. COLT 2010.

[8] Ilya Sinayskiy Maria Schuld and Francesco Petruccione. Simulating a perceptron on a quantum computer, 2014. arXiv:1412.3635 [quant-ph].

[9] Yoram Singer John C. Duchi, Shai Shalev-Shwartz and Ambuj Tewari. Composite objective mirror descent. *COLT*, 2010.