MACHINE LEARNING

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Abstract. Everything about Machine Learning.

From the beginning of 2016, I decided to cease all explicit crowdfunding for any of my materials on physics, math. I failed to raise *any* funds from previous crowdfunding efforts. I decided that if I was going to live in *abundance*, I must lose a scarcity attitude. I am committed to keeping all of my material **open-sourced**. I give all my stuff *for free*.

In the beginning of 2017, I received a very generous donation from a reader from Norway who found these notes useful, through *PayPal*. If you find these notes useful, feel free to donate directly and easily through PayPal, which won't go through a 3rd. party such as indiegogo, kickstarter, patreon. Otherwise, under the *open-source MIT license*, feel free to copy, edit, paste, make your own versions, share, use as you wish.
gmail : ernestyalumni
linkedin : ernestyalumni
tumblr : ernestyalumni
twitter : ernestyalumni
youtube : ernestyalumni

Contents

Date: 24 avril 2016.
Key words and phrases. Machine Learning, statistical inference, statistical inference learning.

Part 1. Data; Data Wrangling, Data cleaning, Web crawling, Data input

1. Sample, example data; input data

1.1. sklearn, from sci-kit learn, sample data, datasets. cf. `sampleinputdataX_sklearn.ipynb`

For $j = 0, 1, \dots d-1$, d = number of "features",

$$x_i^{(j)} \in (\mathbb{R}^N)^d = \underbrace{\mathbb{R}^N \times \mathbb{R}^N \times \dots \times \mathbb{R}^N}_{d}$$

e.g. $N = 442$ (number of given observations/data)

$y_i \in \mathbb{R}^N$ (represents target or result)

Given data $(x_i^{(j)}, y_i) \in (\mathbb{R}^N)^d \times \mathbb{R}^N$,

we can restrict data $(x_i^{(j)}, y_i)$ to subsets to train and test, for training and testing.

So let $I_{\text{train}}, I_{\text{test}} \subset \{0, 1, \dots N-1\}$ s.t. $I_{\text{train}} \cap I_{\text{test}} = \emptyset$.

Want:

$$\begin{gathered} (x_i^{(j)}, y_i)_{i \in I_{\text{train}}} \mapsto \theta_\alpha \\ (\mathbb{R}^{|I_{\text{train}}|})^d \times \mathbb{R}^{|I_{\text{train}}|} \to \mathbb{R}^{|d|} \end{gathered}$$

and so further, I think the idea is

$$\begin{gathered} (x_i^{(j)}, y_i)_{i \in I_{\text{test}}} \xrightarrow{L_{\theta_\alpha}} L_{\theta_\alpha}(\theta_\alpha(x_i^{(j)}, y_i)) \\ (\mathbb{R}^{|I_{\text{test}}|})^d \times \mathbb{R}^{|I_{\text{test}}|} \to \mathbb{R} \end{gathered}$$

Part 2. Introduction

1.1.1. *Terminology.*

inputs $\equiv$ independent variables $\equiv$ predictors (cf. statistics) $\equiv$ features (cf. pattern recognition)

outputs $\equiv$ dependent variables $\equiv$ responses

cf. Chapter 2 Overview of Supervised Learning, Section 2.1 Introduction of Hastie, Tibshirani, and Friedman (2009) [1]

cf. Chapter 2 Overview of Supervised Learning, Section 2.2 Variable Types and Terminology of Hastie, Tibshirani, and Friedman (2009) [1]

1.1.2. *FinSet.*
The category FinSet $\in$ Cat is the category of all finite sets (i.e. Obj(FinSet) $\equiv$ all finite sets) and all functions in between them; note that FinSet $\subset$ Set [1]

Recall that the FinSet *skeletal* is

1.2. **Supervised Learning.** cf. http://cs229.stanford.edu/notes/cs229-notes1.pdf

Consider data to belong to the category of all possible data:

$$\text{Data} \equiv \text{Dat} = (\text{Obj}(\text{Dat}), \text{MorDat}, 1, \circ), \qquad \text{Dat} \in \text{Cat}$$

Consider the **training set**:

$$\text{training set} := \{(x^{(i)}, y^{(i)}) | i = 1 \dots m, x^{(i)} \in \mathcal{X}, y^{(i)} \in \mathcal{Y}\}$$

where $\mathcal{X}$ is a manifold (it can be topological or smooth, EY:20160502 I don't know exactly because I need to check the topological and/or differential structure); $\mathcal{Y} \in \text{Obj(FinSet)}$, or ($\mathcal{Y} \in \text{Obj(Top)}$(or $\mathcal{Y} \in \text{Obj(Man)}$)).

So training set $\subset \mathcal{X} \times \mathcal{Y} \in \text{Obj(Dat)}$.

I propose that there should be a functor H that represents the "learning algorithm":

$$\text{Dat} \xrightarrow{H} \text{ML}$$

s.t.

$$H : \mathcal{X} \times \mathcal{Y} \to \text{Hom}(\mathcal{X}, \mathcal{Y})$$
$$H(\text{training set}) = H(\{(x^{(i)}, y^{(i)}) | i = 1 \dots m\}) = h$$

When $\mathcal{Y} \in \text{Obj(FinSet)}$, *classification*.
When $\mathcal{Y} \in \text{Obj(Top)}$ (or Obj(Man), *regression*.

1.2.1. *Linear Regression.* Keeping in mind

$$\text{Dat} \xrightarrow{H} \text{ML}$$

Consider

$$h : \mathbb{R}^p \to \text{Hom}(\mathcal{X}, \mathcal{Y})$$
$$h : \theta \mapsto h_\theta$$

s.t.

$$h_\theta : \mathcal{X} \to \mathcal{Y}$$

so (possibly) $h \in \text{Obj}ML$ (or is h part of the functor H?)

Consider the cost function J

$$J : \mathbb{R}^p \to \text{Hom}(\mathfrak{X} \times \mathfrak{Y}, \mathbb{R}) = C^\infty(\mathcal{X} \times \mathcal{Y})$$
$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

1.2.2. *LMS algorithm (least mean square (or Widrow-Hoff learning rule)).* Define **gradient descent** algorithm:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

with := being assignment (I'll use := for "define", in mathematical terms, use context to distinguish the 2), where α is the *learning rate*.

Rewriting the above,

$$\theta := \theta - \alpha \text{grad} J(\theta)$$

where grad : $C^\infty(M) \to \mathfrak{X}(M)$, with M being a smooth manifold.

[1] nlab FinSet https://ncatlab.org/nlab/show/FinSet

This is *batch gradient descent*:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) = \theta_j - \alpha \frac{\partial}{\partial \theta_j} \frac{1}{2} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 = \theta_j - \alpha \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \left(\frac{\partial h_\theta(x^{(i)})}{\partial \theta} \right)$$

Simply notice how the entire training set of m rows is used.

I will expound on the so-called distinguished object $1 \xrightarrow{P} X$ on pp. 8, in Section 2 The Category of Conditional Probabilities of Culbertson and Sturtz (2013) [2] because it wasn't clear to me in the first place (the fault is mine; the authors wrote a very lucid and very fathomable, pedagogically-friendly exposition).

$\forall\, Y$ with indiscrete σ-algebra $\Sigma_Y = \{Y, \emptyset\}$
(remember, $((Y, \Sigma_Y), \mu_Y)$, $\mu_Y(\phi) = 0$, $\mu_Y(Y) = 1$),

$\exists !$ unique morphism in Mor$\mathcal{P}$, $X \to Y$, since
$\forall\, P : X \to Y$, $P \in \text{Mor}\mathcal{P}$, P_x must be a probability measure on Y, because

$$(X, \Sigma_X) \xrightarrow{P} (Y, \Sigma_Y)$$
$$P : \Sigma_Y \times X \to [0, 1]$$
$$P(\cdot | x) : \Sigma_Y \to [0, 1] \equiv \begin{array}{l} P_x : \Sigma_Y \to [0,1] \text{ s.t.} \\ P_x(\emptyset) = 0,\, P_x(Y) = 1 \end{array}$$

i.e. EY: 20160503, Given $x \in X$ occurs, Y must occur.

By def. of terminal object ($\forall\, (X, \Sigma_X) \in \text{Obj}\mathcal{P}$, $\exists !$ morphism P s.t. $(X, \Sigma_X) \xrightarrow{P} (Y, \Sigma_Y)$,
Y *terminal* object, and denote unique morphism $!_X : X \to Y$, $!_X \in \text{Mor}\mathcal{P}$.

Up to isomorphism, canonical terminal object is 1-element set denoted by $1 = \{*\}$, with the only possible σ-algebra ($\mu(*) = 1$, $\mu(\emptyset) = 0$),

$$\forall\, P : 1 \to X,\ P \in \text{Mor}\mathcal{P},\ P \in \text{Hom}_{\mathcal{P}}(1, X),\ \forall\, X \in \text{Mor}\mathcal{P}$$

P is an "absolute" probability measure on X because "there's no variability (conditioning) possible within singleton set $1 = \{*\}$." [2]

Now

$$P : \Sigma_X \times 1 \to [0, 1]$$
$$P(\cdot | *) : \Sigma_X \to [0, 1]$$

where $P(\cdot | *) : \Sigma_X \to [0, 1]$ perfect probability measure on X, $P(\cdot | *) : \Sigma_X \to [0, 1] \equiv P_*$, i.e. $P(\cdot | *) = p(\cdot)$ (usual probability on X).

$\forall\, A \in \Sigma_X$, $P(A | \cdot) : 1 \to [0, 1]$, but $P(A | *) = P(A)$, $P(A | \emptyset) = 0$.

Refer to

$$1 \xrightarrow{P} X$$

morphism $P : 1 \to X \in \text{Mor}\mathcal{P}$ as probability measure or distribution on X.

2. Deep Learning

Deep Learning Tutorial [6]

3. Parallel Computing

3.1. **Udacity Intro to Parallel Programming : Lesson 1 - The GPU Programming Model.** Owens and Luebki pound fists at the end of this video. =)))) Intro to the class.

3.1.1. *Running CUDA locally.* Also, Intro to the class, in Lesson 1 - The GPU Programming Model, has links to documentation for running CUDA locally; in particular, for Linux: `http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html`. That guide told me to go download the NVIDIA CUDA Toolkit, which is the https://developer.nvidia.com/cuda-downloads.

For *Fedora*, I chose Installer Type `runfile (local)`.

Afterwards, installation of CUDA on Fedora 23 workstation had been nontrivial. Go see either my github repository ML-grabbag (which will be updated) or my wordpress blog (which may not be upgraded frequently).

$P = VI = I^2R$ heating.

3.1.2. *Definitions of Latency and throughput (or bandwidth).* cf. Building a Power Efficient Processor

Latency vs Bandwidth

latency [sec]. From the title "Latency vs. bandwidth", I'm thinking that throughput = bandwidth (???). throughput = job/time (of job).

Given total task, velocity v,
total task $/v$ = latency. throughput = latency/(jobs per total task).

Also, in Building a Power Efficient Processor. Owens recommends the article David Patterson, "Latency..."

cf. GPU from the Point of View of the Developer

$n_{\text{core}} \equiv$ number of cores
$n_{\text{vecop}} \equiv$ (n_{vecop}−wide axial vector operations/*core* core)
$n_{\text{thread}} \equiv$ threads/core (hyperthreading)

$$n_{\text{core}} \cdot n_{\text{vecop}} \cdot n_{\text{thread}} \text{ parallelism}$$

There were various websites that I looked up to try to find out the capabilities of my video card, but so far, I've only found these commands (and I'll print out the resulting output):

```
$ lspci -vnn | grep VGA -A 12
03:00.0 VGA compatible controller [0300]: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] [10de:17c8] (rev a1) (prog-if 00 [VGA controller])
        Subsystem: eVga.com. Corp. Device [3842:3994]
        Physical Slot: 4
        Flags: bus master, fast devsel, latency 0, IRQ 50
        Memory at fa000000 (32-bit, non-prefetchable) [size=16M]
        Memory at e0000000 (64-bit, prefetchable) [size=256M]
        Memory at f0000000 (64-bit, prefetchable) [size=32M]
        I/O ports at e000 [size=128]
        [virtual] Expansion ROM at fb000000 [disabled] [size=512K]
        Capabilities: <access denied>
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia

$ lspci | grep VGA -E
03:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] (rev a1)

$ grep driver /var/log/Xorg.0.log
[    18.074] Kernel command line: BOOT_IMAGE=/vmlinuz-4.2.3-300.fc23.x86_64 root=/dev/mapper/fedora-root ro rd.lvm.lv=fedora/root rd.lvm.lv=fedora/swap rhgb quiet LANG=en_US.UTF-8 nouveau.modeset=0 rd.driver.blacklist=nouveau nomodeset gfxpayloa
[    18.087] (WW) Hotplugging is on, devices using drivers 'kbd', 'mouse' or 'vmmouse' will be disabled.
[    18.087]    X.Org XInput driver : 22.1
[    18.192] (II) Loading /usr/lib64/xorg/modules/drivers/nvidia_drv.so
[    19.088] (II) NVIDIA(GPU-0): Found DRM driver nvidia-drm (20150116)
[    19.102] (II) NVIDIA(0):     ACPI event daemon is available, the NVIDIA X driver will
[    19.174] (II) NVIDIA(0): [DRI2]   VDPAU driver: nvidia
[    19.284]    ABI class: X.Org XInput driver, version 22.1
...

$ lspci -k | grep -A 8 VGA
03:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] (rev a1)
        Subsystem: eVga.com. Corp. Device 3994
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia
03:00.1 Audio device: NVIDIA Corporation GM200 High Definition Audio (rev a1)
        Subsystem: eVga.com. Corp. Device 3994
        Kernel driver in use: snd_hda_intel
        Kernel modules: snd_hda_intel
05:00.0 USB controller: VIA Technologies, Inc. VL805 USB 3.0 Host Controller (rev 01)
```

CUDA Program Diagram

$$\begin{tikzcd}
& \text{CUDA program in C with extensions} \arrow[dl] \arrow[dr] & \\
\text{CPU ``Host''} \arrow[rr, "\text{coprocessor}"] \arrow[d] & & \text{GPU ``Device''} \arrow[loop right, "4"] \\
\text{Memory} \arrow[rr, "1"] & & \text{Memory} \arrow[loop right, "3"] \arrow[ll, bend left, "2"]
\end{tikzcd}$$

CPU "host" is the boss (and issues commands) -Owen.

Coprocessor : CPU "host" $\to$ GPU "device"
Coprocessor : CPU process $\mapsto$ (co)-process out to GPU

With

1 data cpu $\to$ gpu
2 data gpu $\to$ cpu (initiated by cpu host)

1., 2., uses `cudaMemcpy`
3 allocate GPU memory: `cudaMalloc`
4 launch kernel on GPU

Remember that for 4., this launching of the kernel, while it's acting on GPU "device" onto itself, it's initiated by the boss, the CPU "host".

Hence, cf. Quiz: What Can GPU Do in CUDA, GPUs can respond to CPU request to receive and send Data CPU $\to$ GPU and Data GPU $\to$ CPU, respectively (1,2, respectively), and compute a kernel launched by the CPU (3).

A CUDA Program A typical GPU program

- `cudaMalloc` - CPU allocates storage on GPU
- `cudaMemcpy` - CPU copies input data from CPU $\to$ GPU
- *kernel launch* - CPU launches kernel(s) on GPU to process the data
- `cudaMemcpy` - CPU copies results back to CPU from GPU

Owens advises minimizing "communication" as much as possible (e.g. the `cudaMemcpy` between CPU and GPU), and do a lot of computation in the CPU and GPU, each separately.

Defining the GPU Computation

Owens circled this

BIG IDEA $\boxed{\text{This is Important}}$

Kernels look like serial programs

Write your program as if it will run on **one** thread

The GPU will run that program on **many** threads

Squaring A Number on the CPU

Note

(1) Only 1 thread of execution: ("thread" := one independent path of execution through the code) e.g. the `for` loop
(2) no explicit parallelism; it's serial code e.g. the `for` loop through 64 elements in an array

GPU Code A High Level View

CPU:

- Allocate Memory
- Copy Data to/from GPU
- Launch Kernel - species degree of parallelism

GPU:

- Express Out = In · In - says *nothing* about the degree of parallelism

Owens reiterates that in the GPU, everything looks serial, but it's only in the CPU that anything parallel is specified.
pseudocode: CPU code: square kernel <<< 64 >>> (outArray,inArray)

Squaring Numbers Using CUDA Part 3

From the example

```
// launch the kernel
square<<<1, ARRAY_SIZE>>>(d_out , d_in)
```

we're introduced to the "CUDA launch operator", initiating a kernel of 1 block of 64 elements (`ARRAY_SIZE` is 64) on the GPU. Remember that `d_` prefix (this is naming comvention) tells us it's on the device, the GPU, solely.

With CUDA launch operator ≡<<<>>>, then also looking at this explanation on `stackexchange` (so surely others are confused as well, of those who are learning this (cf. CUDA kernel launch parameters explained right?). From Eric's answer,

threads are grouped into blocks. all the threads will execute the invoked kernel function.

Certainly,

$$<<<>>>: (n_{\text{block}}, n_{\text{threads}}) \times \text{kernelfunctions} \mapsto \text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>> \in \text{End} : \text{Dat}_{\text{GPU}}$$

$$<<<>>>: \mathbb{N}^+ \times \mathbb{N}^+ \times \text{Mor}_{\text{GPU}} \to \text{EndDat}_{\text{GPU}}$$

where I propose that GPU can be modeled as a category containing objects Dat_{GPU}, the collection of all possible data inputs and outputs into the GPU, and Mor_{GPU}, the collection of all kernel functions that run (exclusively, and this *must* be the class, as reiterated by Prof. Owen) on the GPU.

Next,

$$\text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>>: \text{din} \mapsto \text{dout} \qquad \text{(as given in the "square" example, and so I propose)}$$

$$\text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>>: (\mathbb{N}^+)^{n_{\text{threads}}} \to (\mathbb{N}^+)^{n_{\text{threads}}}$$

But keep in mind that dout, din are pointers in the C program, pointers to the place in the memory.

`cudaMemcopy` is a functor category, s.t. e.g. $\text{Obj}_{\text{CudaMemcopy}} \ni$ cudaMemcpyDevicetoHost where

$$\text{cudaMemcopy}(-, -, n_{\text{thread}}, \text{cudaMemcpyDeviceToHost}) : \text{Memory}_{\text{GPU}} \to \text{Memory}_{\text{CPU}} \in \text{Hom}(\text{Memory}_{\text{GPU}}, \text{Memory}_{\text{CPU}})$$

Squaring Numbers Using CUDA 4

Note the C language construct *declaration specifier* - denotes that this is a kernel (for the GPU) and not CPU code. Pointers need to be allocated on the GPU (otherwise your program will crash spectacularly -Prof. Owen).

3.1.3. *What are C pointers?* Is ⟨ type ⟩∗, a pointer, then a mapping from the category, namely the objects of types, to a mapping from the specified value type to a memory address?

e.g.

$$\langle\rangle * : \text{float} \mapsto \text{float} *$$

$$\text{float} \ * : \text{din} \mapsto \text{ some memory address}$$

and then we pass in mappings, not values, and so we're actually declaring a square *functor*.

What is `threadIdx`? What is it mathematically? Consider that ∃ 3 "modules":

$$\text{threadIdx}.x$$
$$\text{threadIdx}.y$$
$$\text{threadIdx}.z$$

And then the line

```
int idx = threadIdx.x;
```

says that idx is an integer, "declares" it to be so, and then assigns idx to threadIdx.x which surely has to also have the same type, integer. So (perhaps)

$$idx \equiv \text{threadIdx}.x \in \mathbb{Z}$$

is the same thing.

Then suppose threadIdx ⊂ FinSet, a subcategory of the category of all (possible) finite sets, s.t. threadIdx has 3 particular morphisms, $x, y, z \in \text{Mor}threadIdx$,

$$x : \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}}$$
$$y : \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}}$$
$$z : \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}}$$

Configuring the Kernel Launch Parameters Part 1

$n_{\text{blocks}}, n_{\text{threads}}$ with $n_{\text{threads}} \geq 1024$ (this maximum constant is GPU dependent). You should pick the $(n_{\text{blocks}}, n_{\text{threads}})$ that makes sense for your problem, says Prof. Owen.

3.1.4. *Memory layout of blocks and threads.* $\forall (n_{\text{blocks}}, n_{\text{threads}}) \in \mathbb{Z} \times \{1 \dots 1024\}$, $\{1 \dots n_{\text{block}} \times \{1 \dots n_{\text{threads}}\}$ is now an ordered index (with lexicographical ordering). This is just 1-dimensional (so possibly there's a 1-to-1 mapping to a finite subset of $\mathbb{Z}$).

I propose that "adding another dimension" or the 2-dimension, that Prof. Owen mentions is being able to do the Cartesian product, up to 3 Cartesian products, of the block-thread index.

Quiz: Configuring the Kernel Launch Parameters 2

Most general syntax:

Configuring the kernel launch

```
kernel<<<grid of blocks, block of threads >>>(...)

// for example

square<<<dim3(bx,by,bz), dim3(tx,ty,tz), shmem>>>(...)
```

where `dim3(tx,ty,tz)` is the grid of blocks $bx \cdot by \cdot bz$
`{dim3}(tx,ty,tz)` is the block of threads $tx \cdot ty \cdot tz$
`shmem` is the shared memory per block in bytes

Problem Set 1 "Also, the image is represented as an 1D array in the kernel, not a 2D array like I mentioned in the video."

Here's part of that code for squaring numbers:

```
__global__ void square(float *d_out, float *d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f*f;
  }
```

3.1.5. *Grid of blocks, block of threads, thread that's indexed; (mathematical) structure of it all.* Let

$$\text{grid} = \prod_{I=1}^N (\text{block})^{n_I^{\text{block}}}$$

where $N = 1, 2, 3$ (for CUDA) and by naming convention $\begin{aligned} I &= 1 \equiv x \\ I &= 2 \equiv y \\ I &= 3 \equiv z \end{aligned}$

Let's try to make it explicity (as others had difficulty understanding the grid, block, thread model, cf. colored image to greyscale image using CUDA parallel processing, Cuda gridDim and blockDim) through commutative diagrams and categories (from math):

$$\begin{tikzcd} \prod_{I=1}^N \mathbb{Z}^+ \arrow[d, "\text{dim3}"] & \ni (N_x^{\text{blocks}}, N_y^{\text{blocks}}, N_z^{\text{blocks}}) \arrow[d, mapsto, "\text{dim3}"'] \\ \text{grid} \arrow[u, bend left, "\text{gridDim}"] & \ni \text{gridSize}(N_x^{\text{blocks}}, N_y^{\text{blocks}}, N_z^{\text{blocks}}) \arrow[u, bend right, "(\text{gridDim}.x,\ \text{gridDim}.y,\ \text{gridDim}.z)"'] \end{tikzcd}$$

$$\begin{tikzcd} \text{grid} \arrow[d, "\text{blockIdx}"] & \ni \texttt{d\_rgbaImage} \arrow[d, mapsto, "(\text{blockIdx}.x\text{,blockIdx}.y\text{,blockIdx}.z)"] \\ \prod_{I=1}^N \mathbb{Z} \supset \prod_{I=1}^N \{1 \dots N_I^{\text{blocks}}\} & \ni (i^{\text{blocks}}, j^{\text{blocks}}, k^{\text{blocks}}) \end{tikzcd}$$

and then similar relations (i.e. arrows, i.e. relations) go for a block of threads:

$$\begin{tikzcd} \prod_{I=1}^N \mathbb{Z}^+ \arrow[d, "\text{dim3}"] & \ni (N_x^{\text{threads}}, N_y^{\text{threads}}, N_z^{\text{threads}}) \arrow[d, mapsto, "\text{dim3}"'] \\ \text{block} \arrow[u, bend left, "\text{blockDim}"] & \ni \text{blockSize}(N_x^{\text{threads}}, N_y^{\text{threads}}, N_z^{\text{threads}}) \arrow[u, bend right, "(\text{blockDim}.x,\ \text{blockDim}.y,\ \text{blockDim}.z)"'] \end{tikzcd}$$

$$\begin{tikzcd} \text{block} \arrow[d, "\text{threadIdx}"] & \ni \text{block} \arrow[d, mapsto, "(\text{threadIdx}.x\text{,threadIdx}.y\text{,threadIdx}.z)"] \\ \prod_{I=1}^N \mathbb{Z} \supset \prod_{I=1}^N \{1 \dots N_I^{\text{threads}}\} & \ni (i^{\text{threads}}, j^{\text{threads}}, k^{\text{threads}}) \end{tikzcd}$$

gridsize help assignment 1 Pp explains how threads per block is variable, and remember how Owens said Luebki says that a GPU doesn't get up for more than a 1000 threads per block.

3.1.6. *Generalizing the model of an image.* Consider vector space V, e.g. $\dim V = 4$, vector space V over field $\mathbb{K}$, so $V = \mathbb{K}^{\dim V}$.

Each pixel represented by $\forall\, v \in V$.

Consider an image, or space, M. $\dim M = 2$ (image), $\dim M = 3$. Consider a local chart (that happens to be global in our case):

$$\begin{aligned} &\varphi : M \to \mathbb{Z}^{\dim M} \supset \{1 \dots N_1\} \times \{1 \dots N_2\} \times \dots \times \{1 \dots N_{\dim M}\} \\ &\varphi : x \mapsto (x^1(x), x^2(x), \dots, x^{\dim M}(x)) \end{aligned}$$

$$\begin{tikzcd} E \arrow[r, "\varphi"] \arrow[d, "\pi"'] & M \times V \arrow[dl] \\ M & \end{tikzcd} \qquad \begin{tikzcd} E \arrow[r, "\varphi"] \arrow[d, "\pi"'] & \text{grid} \times \text{ block of threads} \arrow[dl] \\ \text{grid} & \end{tikzcd}$$

Consider a "coarsing" of underlying M:

$$\begin{tikzcd} M \times V \arrow[rr, "\text{proj}"] \arrow[d, "\pi"'] & & \text{proj}(M) \times \text{proj}(V) \arrow[d, "\text{proj}(\pi)"] \\ M = \{1 \dots N_1\} \times \{1 \dots N_2\} \times \dots \times \{1 \dots N_{\dim M}\} \arrow[rr, "\text{proj}"] & & \text{proj}(M) = \{1 \dots \frac{N_1}{N_1^{\text{threads}}}\} \times \{1 \dots \frac{N_2}{N_2^{\text{threads}}}\} \times \dots \times \{1 \dots \frac{N_{\dim M}}{N_{\dim M}^{\text{threads}}}\} \end{tikzcd}$$

e.g. $N_1^{\text{thread}} = 12$

$N_2^{\text{thread}} = 12$

Just note that in terms of syntax, you have the "block" model, in which you allocate blocks along each dimension. So in

$$\begin{aligned} &const\ dim3\ blockSize(n_x^b, n_y^b, n_z^b) \\ &const\ dim3\ gridSize(n_x^{\text{gr}}, n_y^{\text{gr}}, n_z^{\text{gr}}) \end{aligned}$$

Then the condition is $n_x^b/\dim V, n_y^b/\dim V, n_z^b/\dim V \in \mathbb{Z}$ (condition), $\quad (n_x^{\text{gr}} - 1)/\dim V, n_y^{\text{gr}}/\dim V, n_z^{\text{gr}}/\dim V \in \mathbb{Z}$

Transpose Part 1

Now

$$\begin{aligned} &\text{Mat}_{\mathbb{F}}(n,n) \xrightarrow{T} \text{Mat}_{\mathbb{F}}(n,n) \\ &A \mapsto A^T \text{ s.t. } (A^T)_{ij} = A_{ji} \\ &\text{Mat}_{\mathbb{F}} \xrightarrow{T} \mathbb{F}^{n^2} \\ &A_{ij} \mapsto A_{ij} = A_{in+j} \end{aligned}$$

$$\begin{tikzcd} \text{Mat}_{\mathbb{F}}(n,n) \arrow[r] \arrow[d, "T"'] & \mathbb{F}^{n^2} \arrow[d, "T"] \\ \text{Mat}_{\mathbb{F}}(n,n) \arrow[r] & \mathbb{F}^{n^2} \end{tikzcd} \qquad \begin{tikzcd} A_{ij} \arrow[r, mapsto] \arrow[d, mapsto, "T"'] & A_{in+j} \arrow[d, mapsto, "T"] \\ (A^T)_{ij} = A_{ji} \arrow[r, mapsto] & A_{jn+i} \end{tikzcd}$$

Transpose Part 2

Possibly, transpose is a functor.

Consider struct as a category. In this special case, $\text{Obj}\,\text{struct} = \{\text{arrays}\}$ (a struct of arrays). Now this struct already has a hash table for indexing upon declaration (i.e. "creation"): so this category struct will need to be equipped with a "diagram" from the category of indices J to struct: $J \to \text{struct}$.

So possibly

$$\begin{array}{ccc} \text{struct} & \xrightarrow{T} & \text{array} \\ \text{ObjStruct} = \{ \text{ arrays } \} & \xrightarrow{T} & \text{Objarray} = \{ \text{ struct } \} \\ J \to \text{ struct } & \xrightarrow{T} & J \to \text{ array} \end{array}$$

Quiz: What Kind Of Communication Pattern This quiz made a few points that clarified the characteristics of these so-called communication patterns (amongst the memory?)

- map is bijective, and map : Idx $\to$ Idx
- gather - not necessarily surjective
- scatter - not necessarily surjective
- stencil - surjective
- transpose (see before)

Parallel Communication Patterns Recap

- map - bijective
- transpose - bijective
- gather - not necessarily surjective, and is many-to-one (by def.)
- scatter - one-to-many (by def.) and is not necessarily surjective
- stencil - several-to-one (not injective, by definition), and is surjective
- reduce - all-to-one
- scan/sort - all-to-all

Programmer View of the GPU

thread blocks: group of threads that cooperate to solve a (sub)problem

Thread Blocks And GPU Hardware

CUDA GPU is a bunch of SMs:

Streaming Multiprocessors (SM)s

SMs have a bunch of simple processors and memory.

Dr. Luebki:

$$\boxed{\begin{array}{c}\text{Let me say that again because it's really important} \\ \text{GPU is responsible for allocating blocks to SMs}\end{array}}$$

Programmer only gives GPU a pile of blocks.

Quiz: What Can The Programmer Specify

I myself thought this was a revelation and was not intuitive at first:

Given a single kernel that's launched on many thread blocks include X, Y, the programmer cannot specify the sequence the blocks, e.g. block X, block Y, run (same time, or run one after the other), and which SM the block will run on (GPU does all this).

Quiz: A Thread Block Programming Example

Open up `hello blockIdx.cu` in Lesson 2 Code Snippets (I got the repository from github, repo name is cs344).

At first, I thought you can do a single file compile and run in Eclipse without creating a new project. No. cf. Eclipse creating projects every time to run a single file?.

I ended up creating a new CUDA C/C++ project from File -¿ New project, and then chose project type Executable, Empty Project, making sure to include Toolchain CUDA Toolkit (my version is 7.5), and chose an arbitrary project name (I chose cs344single). Then, as suggested by Kenny Nguyen, I dragged and dropped files into the folder, from my file directory program.

I ran the program with the "Play" triangle button, clicking on the green triangle button, and it ran as expected. I also turned off Build Automatically by deselecting the option (no checkmark).

GPU Memory Model

$$\begin{tikzcd} \text{thread} & \text{local memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"] \end{tikzcd}$$

Then consider threadblock $\equiv$ thread block

$$\text{Objthreadblock} \supset \{ \text{ threads } \}$$
$$\text{FinSet} \xrightarrow{\text{threadIdx}} \text{thread } \in \text{Morthreadblock}$$

$$\begin{tikzcd} \text{threadblock} & \text{shared memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"] \end{tikzcd}$$

$\forall$ thread,

$$\begin{tikzcd} \text{thread} & \text{global memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"] \end{tikzcd}$$

Synchronization - Barrier

Quiz: The Need For Barriers

3 barriers were needed (wasn't obvious to me at first). All threads need to finish the write, or initialization, so it'll need a barrier.

While

```
array[idx] = array[idx+1];
```

is 1 line, it'll actually need 2 barriers; first read. Then write.

So *actually* we'll need to *rewrite* this code:

```
int temp = array[idx+1];
__syncthreads();
array[idx] = temp;
__syncthreads();
```

kernels have implicit barrier for each.

Writing Efficient Programs

(1) Maximize *arithmetic intensity* arithmetic intensity $:= \frac{\text{math}}{\text{memory}}$

video: Minimize Time Spent On Memory

local memory is fastest; global memory is slower

$$\text{local} > \text{ shared} >> \text{global} >> \text{CPU}$$

kernel we know (in the code) is tagged with `__global__`

quiz: A Quiz on Coalescing Memory Access

Work it out as Dr. Luebki did to figure out if it's coalesced memory access or not.

Atomic Memory Operations

Atomic Memory Operations

atomicadd atomicmin atomicXOR atomicCAS Compare And Swap

4. Pointers in C; Pointers in C categorified (interpreted in Category Theory)

Suppose $v \in \text{ObjData}$, category of data **Data**,
e.g. $v \in \text{Int} \in \text{ObjType}$, category of types Type.

$$\text{Data} \xrightarrow{\&} \text{Memory}$$
$$v \overset{\&}{\mapsto} \&v$$

with address $\&v \in \text{Memory}$.
With
assignment $pv = \&v$,

$$pv \in \text{Objpointer}, \quad \text{category of pointers, pointer}$$
$$pv \in \text{Memory} \qquad (\text{i.e. not } pv \in \text{Dat, i.e. } pv \notin \text{Dat})$$

$$\text{pointer } \ni pv \overset{*}{\mapsto} *pv \in \text{Dat}$$

$$\begin{tikzcd} v \arrow[r, mapsto, "\&"] & \&v \arrow[d, "="] \\ *pv \arrow[u, "=="] & pv \arrow[l, mapsto, "*"] \end{tikzcd} \qquad \begin{tikzcd} \text{Data} \arrow[r, "\&"] & \text{Memory} \arrow[d, "="] \\ \text{Data} \arrow[u, "=="] & \text{pointer} \arrow[l, "*"] \end{tikzcd}$$

Examples. Consider `passfunction.c` in Fitzpatrick [5].
Consider the type `double`, `double` $\in$ ObjTypes.
fun1, fun2 $\in$ MorTypes namely
fun1, fun2 $\in \text{Hom}(\text{double}, \text{double}) \equiv \text{Hom}_{\text{Types}}(\text{double}, \text{double})$
Recall that

$$\text{pointer} \xrightarrow{*} \text{Dat}$$
$$\text{pointer} \xrightarrow{\&} \text{Memory}$$

$*, \&$ are functors with domain on the category pointer.
Pointers to functions is the "extension" of functor $*$ to the codomain of MorTypes:

$$\begin{aligned} &\text{pointer} && \xrightarrow{*} \text{MorTypes} \\ &\text{fun1} && \overset{*}{\mapsto} *\text{fun1} \in \text{Hom}_{\text{Types}}(\text{double}, \text{double}) \end{aligned}$$

$$\begin{tikzcd} \text{double} \arrow[r, "\&"] & \text{Memory} \arrow[d, "\cong"] \\ \text{double} \arrow[d, "\text{cube}"'] & \text{pointer} \arrow[l, "*"] \arrow[dl, "*"] \\ \text{double} & \end{tikzcd} \qquad \begin{tikzcd} \text{res1} \arrow[r, mapsto, "\&"] & \&\text{res1} \arrow[d, "\cong"] \\ *\text{res1} \arrow[d, "\text{cube}"'] & \text{res1} \arrow[l, mapsto, "*"] \arrow[dl, mapsto, "*"] \\ *\text{res1} = y^3 & \end{tikzcd}$$

It's unclear to me how `void cube` can be represented in terms of category theory, as surely it cannot be represented as a mapping (it acts upon a functor, namely the $*$ functor for pointers). It doesn't return a value, and so one cannot be confident to say there's explicitly a domain and codomain, or range for that matter.
But what is going on is that

$$\text{pointer}\ ,\ \text{double}\ ,\ \text{pointer} \xrightarrow{\text{cube}} \text{pointer}\ ,\ \text{pointer}$$
$$\text{fun1}, x, \text{res1} \overset{\text{cube}}{\mapsto} \text{fun1}, \text{res1}$$

s.t. $*\text{res1} = y^3 = (*\text{fun1}(x))^3$
So I'll speculate that in this case, `cube` is a functor, and in particular, is acting on $*$, the so-called deferencing operator:

$$\begin{matrix} \text{pointer} \xrightarrow{*} \text{float} \in \text{Data} \\ \text{res1} \overset{*}{\mapsto} *\text{res1} \end{matrix} \xrightarrow{\text{cube}} \begin{matrix} \text{pointer} \xrightarrow{\text{cube}(*)} \text{float} \in \text{Data} \\ \text{res1} \overset{\text{cube}(*)}{\mapsto} \text{cube}(*\text{res1}) = y^3 \end{matrix}$$

cf. Arrays, from Fitzpatrick [5]

$$\text{Types} \xrightarrow{\text{declaration}} \text{arrays}$$

If $x \in \text{Objarrays}$,

$$\&x[0] \in \text{Memory} \xrightarrow{==} x \in \text{ pointer (to 1st element of array)}$$

cf. Section 2.13 Character Strings from Fitzpatrick [5]

```
char word[20] = ``four''
char *word = ``four''
```

cf. C++ extensions for C according to Fitzpatrick [5]

- simplified syntax to pass by reference pointers into functions
- inline functions
- variable size arrays

```
int n;
double x[n];
```

- complex number class

4.0.1. *Need a CUDA, C, C++, IDE? Try Eclipse!* This website has a clear, lucid, and pedagogical tutorial for using Eclipse: Creating Your First C++ Program in Eclipse. But it looks like I had to pay. Other than the well-written tips on the webpage, I looked up stackexchange for my Eclipse questions (I had difficulty with the Eclipse documentation).

Part 3. Machine Learning with Deep Learning

cf. Machine Learning - Introduction, from Coursera. Dr. Andrew Ng.

(1) Week 1
- Linear Regression with One Variable
  - Model and Cost Function
    - Model Representation
    - Cost Function
    - Cost Function - Intuition I
    - Cost Function - Intuition II
  - Parameter Learning
    - Gradient Descent
    - Gradient Descent Intuition
    - Gradient Descent For Linear Regression

cf. Linear Regression with One Variable

cf. Model Representation; Week 1 Linear Regression with 1 Variable, Coursera Machine Learning, Ng

For hypothesis h,

$$h_\theta : \mathbb{R}^d \to \mathbb{R}$$
$$h_\theta : x \mapsto h_\theta(x) \qquad \text{(prediction of } y \text{ for } x)$$

$h_\theta \in L(\mathbb{R}^d, \mathbb{R})$

$$\begin{aligned} h_\theta &: \mathbb{R}^{|\theta|} \to L(\mathbb{R}^d, \mathbb{R}) \\ \theta &\mapsto h_\theta \end{aligned}$$

Cost Function; Week 1, Coursera, Machine Learning, Ng

So for parameters

$$\theta \in \mathbb{R}^{|\theta|}$$

define a *cost function*

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x_i) - y_i)^2 \tag{1}$$

Find

$$\min_\theta J(\theta) = ?(???)$$

for

$$J : \mathbb{R}^{|\theta|} \to \mathbb{R}$$

Actually,

$$\begin{aligned} &J(\theta, (x_i, y_i)_{i \in I_{\text{train}}}) \\ &J : \mathbb{R}^{|\theta|} \times (\mathbb{R}^d)^m \times \mathbb{R}^m \to \mathbb{R} \end{aligned} \tag{2}$$

$m =$ number of training examples $= |I_{\text{train}}|$.

Considering

$$H(\theta + \Delta\theta) \approx J(\theta) + \text{grad}J(\theta) \cdot \Delta\theta + \frac{1}{2t}\|\Delta\theta\|^2$$

Suppose $\Delta\theta \equiv \Delta\theta(t) = t\Delta\theta$

$\Delta\theta \approx -\gamma \text{grad}J(\theta)$ is an ansatz, γ small enough.

Then assume J convex, use this ansatz by plugging in, with Lipshitz condition

$$\|\text{grad}J(\theta + \Delta\theta) - \text{grad}J(\theta)\| \leq L\|\Delta\theta\|$$

some constant $L > 0$,

$$\begin{aligned} \theta^i_{n+1} &= \theta^i_n - \gamma_n(\text{grad}J(\theta))^i \\ \gamma_n &= \frac{(\theta^i_n - \theta^i_{n-1})(\text{grad}_\theta J(x_n) - \text{grad}_\theta J(x_{n-1}))^i}{\|\text{grad}_\theta J(x_n) - \text{grad}_\theta J(x_{n-1})\|^2} = \frac{(\theta_n - \theta_{n-1}) \cdot (\text{grad}_\theta J(x_n) - \text{grad}_\theta J(x_{n-1}))}{\|\text{grad}_\theta J(x_n) - \text{grad}_\theta J(x_{n-1})\|^2} \end{aligned} \tag{3}$$

or as Ng points out in the Gradient Descent lesson recap, the correct way is to store in temporary variables first:

$$\begin{aligned} \text{temp} &= \theta^i_n - \gamma_n(\text{grad}J(\theta))^i \\ \theta^i_{n+1} &= \text{temp} \end{aligned} \tag{4}$$

where temp $\in \mathbb{R}^{|\theta|}$

In the lesson recap for Gradient Descent Intuition, Ng denotes the learning rate $\alpha \in \mathbb{R}$ with α, but note that it's denoted as γ or `gamma` for `sci-kit learn`. So be aware of different notations. Nevertheless, the learning rate can be a constant, but even then, choosing it is nontrivial.

4.0.2. *Testing many hypotheses at the same time, via refactoring the matrix.* In Linear Algebra Review of Week 1, Matrix Matrix Multiplication, Ng provided a useful tip in refactoring the matrix of hypotheses h_θ so to test multiple number of hypotheses at the same time on the same input data, X.

Mathematically, beginning with

$$\begin{aligned} h : \mathbb{R}^{|\theta|} &\longrightarrow L(\mathbb{R}^d, \mathbb{R}) \\ \theta &\longmapsto h_\theta \end{aligned}$$

Consider testing H different hypotheses, $\underbrace{\mathbb{R}^{|\theta|} \times \cdots \times \mathbb{R}^{|\theta|}}_{H} \equiv \otimes^H_{i=1}\mathbb{R}^{|\theta|}$,

so treat

$$\otimes^H_{i=1}\mathbb{R}^{|\theta|} = \text{Mat}_{\mathbb{R}}(|\theta|, H)$$

and so

$$\begin{aligned} h : \otimes^H_{i=1}\mathbb{R}^{|\theta|} = \text{Mat}_{\mathbb{R}}(|\theta|, H) &\longrightarrow \otimes^H_{i=1} L(\mathbb{R}^d, \mathbb{R}) \\ \theta^{(i)} &\longmapsto h_{\theta^{(i)}} \end{aligned}$$

cf. Week 4, Non-linear Hypotheses video of Motivations for Coursera's Machine Learning by Ng

For a sigmoid function g, consider

$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \dots)$$

If n large (Ng's notation), $d = \dim\mathbb{R}^d$, number of features for training (data) set, for including quadratic features,

$$\begin{aligned} &x_1^2, x_1x_2, x_1x_3, x_1x_4 \dots x_1x_{100} \\ &x_2^2, x_1x_3, \dots \\ &\approx \mathcal{O}(n^2) \approx \frac{n^2}{2} \qquad (\mathcal{O}(d^2) \approx \frac{d^2}{2}) \end{aligned}$$

e.g. computer vision,
e.g. 50×50 pixel images,
$n = 2500$
pixel intensity $\in [0, 255]$
rgb $\in [0, 255]^3$

$$\begin{aligned} g : \mathbb{R}^{|\theta|} &\to L(\mathbb{R}^d, \mathbb{R}) \\ \theta &\mapsto g(\theta) \equiv g_\theta \end{aligned}$$

$n \equiv d = 2$.

Consider

$$\sum_{\substack{a_1, a_2 = 0 \\ i = a_1 + 2a_2}} \theta^{(i)} x_1^{a_1} x_2^{a_2}$$

and so for this example

$$g(\theta)(x_1, x_2) = g\left(\sum_{\substack{a_1, a_2 = 0 \\ i = a_1 + 2a_2}} \theta^{(i)} x_1^{a_1} x_2^{a_2} \right)$$

For computer vision, consider

$$x \in \mathbb{R}^d \text{ with } d = n^x \times n^y$$

and in particular, given pixel intensity or rgb range,

$$x \in [0,255]^d$$
$$x \in [0,255]^{3d}$$

cf. Model Representation I of Week 4, Coursera's Machine Learning Introduction with Ng

The notes at the end of each video segment **help very much**.

For input

$$\mathbf{x} \in \mathbb{R}^d$$

e.g. $d = 1,2,3, \text{ or } 4, \dots$

$x_0 =$ "bias unit", input node 0, $x_0 = 1$ always (Ng).

Sigmoid (logistic) activation function $\equiv a$.

$$a_i^{(j)} \equiv \text{ "activation" of unit } i \text{ in layer } j$$

$j \in \{2, \dots, N-1\}$, $j = 1$ is input layer, $j = N$ is output layer.

$$a_i^{(j)} = g(\Theta_{ik}^{(j-1)} x_k)$$

$$j \xrightarrow{\Theta^{(j)}} j+1$$

$\Theta^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$.

$$h_\Theta(x) = a_1^{(N)} = g(\Theta_{1k}^{(N-1)} a_k^{(N-1)})$$

$\forall$ layer j, $\exists$ matrix of weights $\Theta^{(j)}$.

If s_j units in layer j, s_{j+1} units in layer $j+1$, $\dim \Theta^{(j)} = s_{j+1} \times (s_j + 1)$

If $N = 2$, (1 neuron or only 1 hidden layer)

$$x = (x_i)_{i=1\dots d} \in \mathbb{R}^d, \qquad y \in \mathbb{R}, \, x_0 = 1$$
$$y = h(\Theta_{1k}^{(1)} x_k^{(1)}) = h(\Theta_{1k}^{(1)} x_k) = h(\Theta^{(1)})(x)$$

e.g. $h(z) = \frac{1}{1+e^z}$ logistic function.

Neural Network, input layer, output layer, and hidden layers.

$$\Theta_{ik}^{(j)} x_k \mapsto g a_i^{(j+1)} \qquad \begin{aligned} & k = 0, 1, \dots s_j \\ & i = 1, 2, \dots s_{j+1} \end{aligned} \tag{5}$$

Note that y can be $y \in \mathbb{R}^M$, not just $M = 1$.

Model Representation II

$z_i^{(j)}$, $i = 1, \dots s_j$, layer $j = 1, \dots N$.

$$g : z_i^{(j)} \mapsto a_i^{(j)} \tag{6}$$

e.g. $z_i^{(j)} = \Theta_{ik}^{(j-1)} x_k$, $k = 0, 1 \dots d$.

Set $x = a^{(1)}$ for input layer.

$$\begin{gathered} \Theta^{(j-1)} \in \mathrm{Mat}_{\mathbb{R}}((d+1), s_j) \\ \Theta^{(j-1)} : a^{(j-1)} \in \mathbb{R}^{d+1} \mapsto z^{(j)} \in \mathbb{R}^{s_j} \xrightarrow{g} a^{(j)} \in \mathbb{R}^{s_j} \xrightarrow{a_0^{(j)}=1} a^{(j)} \in \mathbb{R}^{s_j+1} \end{gathered} \tag{7}$$

For the $j = N$ case, "output" layer,

$$\Theta^{(N-1)} : a^{(N-1)} \mapsto z^N \in \mathbb{R} \xrightarrow{g} g(z^N) = a^N = h_\Theta(x) \in \mathbb{R} \qquad \Theta^{(N-1)} \in \mathrm{Mat}_{\mathbb{R}}(s_{N-1}+1, 1) \tag{8}$$

In general,

$$\Theta^{(N-1)} : a^{(N-1)} \mapsto z^N \in \mathbb{R} \xrightarrow{g} g(z^N) = a^N = h_\Theta(x) \in \mathbb{R}^M \qquad \Theta^{(N-1)} \in \mathrm{Mat}_{\mathbb{R}}(s_{N-1}+1, M)$$

cf. Learning With Large Datasets, Quiz of Week 10, Gradient Descent with Large Datasets; Learning with Large Datasets.

Suppose you are facing a supervised learning problem and have a very large dataset ($m = 100,000,000$). How can you tell if using all of the data is likely to perform much better than using a small subset of the data (say $m = 1,000$)?

Plot a learning curve ($J_{\text{train}}(\theta)$ and $J_{CV}(\theta)$, plotted as a function of m) for a range of values of m and verify that the algorithm has high variance when m is small.

cf. 1.4 Regularized cost function

$$\begin{aligned} J(\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[-y_k^{(i)} \log ((h_\theta(x^{(i)}))_k) - (1 - y_k^{(i)}) \log (1 - (h_\theta(x^{(i)}))_k) \right] + \\ + \frac{\lambda}{2m} \left[\sum_{j=1}^{s_2} \sum_{k=1}^{d} (\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{K} \sum_{k=1}^{s_2} (\Theta_{j,k}^{(2)})^2 \right] \end{aligned}$$

5. Feedforward; Feedforward Propagation and Prediction

Given ordered sequence of linear transformations L, $L \geq 2$,

$$\begin{gathered} \Theta^{(l)} \in \mathrm{Mat}_{\mathbb{R}}(s_l + 1, s_{l+1}) \text{ i.e. } s_{l+1} \times (s_l + 1) \text{ matrix size }, \forall \, l = 1, 2, \dots L-1 \\ \Theta^{(l)} : \mathbb{R}^{s_l + 1} \to \mathbb{R}^{s_{l+1}} \\ \Theta^{(l)} : a^{(l)} \mapsto z^{(l+1)} = \Theta^{(l)} a^{(l)} = \Theta_{ij}^{(l)} a_j^{(l)} = z_i^{(l+1)} \end{gathered} \tag{9}$$

$a^{(l)} \equiv$ "activation" of layer l.

$s_l \equiv$ "layer size" of layer l, number of units or nodes in layer l

$$\begin{aligned} & g : \mathbb{R}^{s_l} \to \mathbb{R}^{s_l} \\ & g : z^{(l)} \mapsto g(z^{(l)}) \end{aligned} \tag{10}$$

e.g. g sigmoid function.

Remember to add $a_0^{(l)} = 1$, $\forall \, l = 1, \dots L-1$, i.e. $\forall$ input layer and hidden layers.

For $l = 1$, the so-called *input layer*, is such that

$$(a_0^{(1)} = 1, x) = a^{(1)} \tag{11}$$

For $l = 1, 2, \dots L-1$,

$$\mathbb{R}^{s_l} \xrightarrow{a_0^{(l)} = 1} \mathbb{R}^{s_l + 1} \xrightarrow{\Theta^{(l)}} \mathbb{R}^{s_{l+1}} \xrightarrow{g} \mathbb{R}^{s_{l+1}}$$

$$a^{(l)} \xmapsto{a_0^{(l)} = 1} (a_0^{(l)} = 1, a^{(l)}) \xmapsto{\Theta^{(l)}} z^{(l+1)} \xmapsto{g} g(z^{(l+1)}) = a^{(l+1)} \tag{12}$$

6. Backpropagation; Backpropagation algorithm

First, do feedforward on *each* training example t, i.e.

$$\forall\, t = 1, 2 \dots m$$

$$\begin{gathered} \mathbb{R}^d \xrightarrow{(g \circ \Theta^{(l)} \circ (a_0^{(l)} = 1) \times \cdot)^{L-1}} \mathbb{R}^K \\ x^{(t)} \xmapsto{(g \circ \Theta^{(l)} \circ (a_0^{(l)} = 1) \times \cdot)^{L-1}} a^{(L)} \end{gathered} \tag{13}$$

For $K = 1$ or $K > 1$ e.g. $K = 10$ for multi-class logistic regression.

In fact, we obtain an ordered sequence of "activation" vectors:

$$\forall\, t = 1, 2 \dots m$$

$$\begin{gathered} \mathbb{R}^d \xrightarrow{(g \circ \Theta^{(l)} \circ (a_0^{(l)} = 1) \times \cdot)^{L-1}} \mathbb{R}^{s_2} \times \mathbb{R}^{s_2} \times \mathbb{R}^{s_3} \times \mathbb{R}^{s_3} \times \dots \times \mathbb{R}^K \\ x^{(t)} \xmapsto{(g \circ \Theta^{(l)} \circ (a_0^{(l)} = 1) \times \cdot)^{L-1}} z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)}, \dots, a^{(L)} \end{gathered} \tag{14}$$

From Backpropagatio¡n algorithm, Cost Function and Backpropagation, Week 5 of Coursera's Machine Learning Introduction by Ng, Backpropagation algorithm notes, and `ex4.pdf`

Calculate

$$\begin{aligned} \delta^{(L)} &:= a^{(L)} - y \\ \delta_k^{(L)} &:= a_k^{(L)} - y_k \qquad \forall\, k = 1, 2, \dots K \end{aligned} \tag{15}$$

For the term $((\Theta^{(L-1)})^T \delta^{(L)})$, the matrix size dimensions of the $(\Theta^{(L-1)})^T$ are $\dim(\Theta^{(L-1)})^T = (s_{L-1} + 1) \times s_L$.

It seems that the element-wise or component-wise multiplication that seems obvious in Matlab/Octave or numpy is called the *Hadamard product*, denoted $\circ$ or $\odot$. There ought to be a homomorphism that maps this operation onto "vectorized" forms of these vectors that allows for, or is equipped with the operation, Hadamard product.

For $m = 1$,

$$\delta^{(L-1)} := \left((\Theta^{(L-1)})^T \delta^{(L)} \right) \odot g'(z^{(L-1)}) \in \mathbb{R}^{s_{L-1}+1} \qquad \forall\, k = 0, 1, \dots s_{L-1}$$

i.e.

$$\begin{gathered} \mathrm{vec}(\delta^{(L-1)}) = \mathrm{vec}((\Theta^{(L-1)})^T \delta^{(L)}) \odot \mathrm{vec}(g'(z^{(L-1)})) \mapsto \delta^{(L-1)} \in \mathbb{R}^{s_{L-1}+1} \\ (s^{(L-1)})_K := ((\Theta^{(L-1)})^T \delta^{(L)})_K (g'(z^{(L-1)}))_K \end{gathered}$$

Then add this term to the so-called "accumulator matrix" $\Delta^{(l)}$:

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$$

Note that prior, skip or *remove* $\delta_0^{(l+1)}$ entry:

$$\delta^{(l+1)} \in \mathbb{R}^{s_{l+1}+1} \xrightarrow{r} \delta^{(l+1)} \in \mathbb{R}^{s_{l+1}}$$

The whole purpose is to obtain

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} = a_j^{(l)} ((\Theta^{(l+1)})^T \delta^{(l+2)} \odot g'(z^{(l+1)}))_i$$

which can be shown.

So first we had set

$$\Delta_{ij}^{(l)} = 0$$

for

$$\Delta^{(l)} \in \mathrm{Mat}_{\mathbb{R}}(s_l, s_{l+1}) \in \mathbb{R}^{s_l} \otimes \mathbb{R}^{s_{l+1}}$$

Again, it can be shown that

$$\Delta_{ij}^{(l)} = \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \tag{16}$$

and so

$$\begin{aligned} \Delta^{(l)} &:= \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T \\ \Delta_{ij}^{(l)} &:= \Delta_{ij}^{(l)} + \delta_j^{(l+1)} (a^{(l)})_i \end{aligned}$$

and so for $\forall\, t$,

$$\begin{cases} D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$$

$$D^{(l)} = \frac{1}{m} \sum_{t=1}^m (\Delta^{(l)})^{(t)} + \lambda \Theta^{(l)} \in \mathrm{Mat}_{\mathbb{R}}(s_l, s_{l+1})$$

In summary, we have, for the first step,

$$\delta^{(L)} := a^{(L)} - y \in \mathbb{R}^K \tag{17}$$

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} \odot g'(z^{(l)}) \in \mathbb{R}^{s_l+1}, \qquad l = L-1, L-2, \dots 2, \qquad (L-2) \text{ steps} \tag{18}$$

and so for

$$(\Delta^{(l)})^{(t)} := (\delta^{(l+1)} (a^{(l)})^T)^{(t)} \tag{19}$$

$$D^{(l)} = \frac{1}{m} \sum_{t=1}^m (\Delta^{(l)})^{(t)} + \lambda \Theta^{(l)} \in \mathrm{Mat}_{\mathbb{R}}^l(s_l, s_{l+1}) \tag{20}$$

with $D^{(l)} \sim \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$.

And so

$$\begin{gathered} (\mathbb{R}^{s_2})^2 \times (\mathbb{R}^{s_3})^2 \times \dots \times (\mathbb{R}^K)^2 \longrightarrow \mathrm{Mat}_{\mathbb{R}}(s_1, s_s) \times \mathrm{Mat}_{\mathbb{R}}(s_2, s_3) \times \dots \times \mathrm{Mat}_{\mathbb{R}}(s_{L-1}, s_L) \\ z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)}, \dots, z^{(L)}, a^{(L)} \longmapsto (\Delta^{(1)})^{(t)}, (\Delta^{(2)})^{(t)}, \dots, (\Delta^{(L-1)})^{(t)} \end{gathered} \tag{21}$$

$\forall\, t = 1, \dots m$, then obtaining

$$D^{(l)} \sim \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \in \mathrm{Mat}_{\mathbb{R}}(s_l, s_{l+1}) \qquad \forall\, l = 1, 2, \dots L-1 \tag{22}$$

To collect our facts, consider that we're given $x \in (\mathbb{R}^d)^m$, with $x_i^{(t)}$, $\begin{matrix} i = 1 \dots d \\ t = 1 \dots m \end{matrix}$, with $\begin{matrix} y \in (\mathbb{R}^K)^m & \\ y \in \{1, 2, \dots K\}^m & \text{(classifier)} \end{matrix}$.

"layer" $l = 1, 2, \dots L-1$ For input layer $\begin{matrix} \Theta^{(1)} : \mathbb{R}^{d+1} \to \mathbb{R}^{s_2} \\ \Theta^{(1)} : a^{(1)} \mapsto \Theta^{(1)} a^{(1)} = z^{(1)} \end{matrix}$, with $a^{(1)} = (1, x^{(t)})$.

Instead of thinking of separate "layers", one should really think of encapsulating the relation, or arrows, or mappings between "layers":

$$\begin{gathered}\mathbb{R}^d \xrightarrow{a_0^{(1)}=1} \mathbb{R}^{d+1} \xrightarrow{\Theta^{(1)}} \mathbb{R}^{s_2} \xrightarrow{g} \mathbb{R}^{s_2} \\ x \xmapsto{a_0^{(1)}=1} (a_0^{(l)}=1,x) \xmapsto{\Theta^{(1)}} z^{(2)} \xmapsto{g} g(z^{(2)}) = a^{(2)}\end{gathered} \tag{23}$$

$$\begin{gathered}\mathbb{R}^{s_l} \xrightarrow{a_0^{(l)}=1} \mathbb{R}^{s_l+1} \xrightarrow{\Theta^{(l)}} \mathbb{R}^{s_{l+1}} \xrightarrow{g} \mathbb{R}^{s_{l+1}} \\ a^{(l)} \xmapsto{a_0^{(l)}=1} (a_0^{(l)}=1,a^{(l)}) \xmapsto{\Theta^{(l)}} z^{(l+1)} \xmapsto{g} g(z^{(l+1)}) = a^{(l+1)}\end{gathered} \tag{24}$$

I found that Theano wasn't like the '`.stack`' method, the "addition" of adding the $a_0 = 1$ component to a vector or matrix, as a shared variable, very much on the GPU (it indeed is a bug, Merge fails on GPU but passes on CPU #152 , and so I rewrote the mathematical formulation to fit in with separating the intercepts from the "weights" or Θ.

For

$$\Theta^{(l)}, b^{(l)} : \mathbb{R}^{s_l} \to \mathbb{R}^{s_l+1} \tag{25}$$

where

$$\begin{gathered}\Theta^{(l)} \in \text{Mat}_{\mathbb{R}}(s_l, s_{l+1}) = \mathbb{R}^{s_{l+1}} \otimes (\mathbb{R}^{s_l})^* \\ b^{(l)} \in \mathbb{R}^{s_{l+1}}\end{gathered} \tag{26}$$

$$\begin{gathered}\mathbb{R}^{s_l} \xrightarrow{\Theta^{(l)}, b^{(l)}} \mathbb{R}^{s_{l+1}} \xrightarrow{g} \mathbb{R}^{s_{l+1}} \\ a^{(l)} \xmapsto{\Theta^{(l)}, b^{(l)}} z^{(l+1)} \xmapsto{g} g(z^{(l+1)}) = a^{(l+1)}\end{gathered} \tag{27}$$

6.1. **Cost functional.** The cost function J is really a cost *functional*, to first input in the output values y. So

$$\begin{gathered}J : (\mathbb{R}^K)^m \to L((\mathbf{\Theta}, \mathbf{b}), \mathbb{R}) \\ J : y \mapsto J_y \equiv J\end{gathered} \tag{28}$$

for a "vector-valued" regression, with the usual linear regression being the case of $K = 1$.

For y taking on discrete values,

$$\begin{gathered}J : \{1, 2, \dots K\}^m \to L((\mathbf{\Theta}, \mathbf{b}), \mathbb{R}) \\ J : y \mapsto J_y \equiv J\end{gathered} \tag{29}$$

Then, we can find the cost $J((\Theta, b))$, for a particular choice of the parameters, $(\Theta, b) \in (\mathbf{\Theta}, \mathbf{b})$:

$$\begin{gathered}J_y \equiv J : (\mathbf{\Theta}, \mathbf{b}) \to \mathbb{R} \\ J : (\Theta, b) \to J(\Theta, b)\end{gathered} \tag{30}$$

i.e. $J \in C^\infty((\Theta, b))$ (hopefully J is smooth or at least C^2 differentiable, so that a Hessian can be obtained).

For the above $(\mathbf{\Theta}, \mathbf{b})$ was notation or shorthand as follows:

$$(\mathbf{\Theta}, \mathbf{b}) \equiv (\text{Mat}_{\mathbb{R}}(s_1, s_2) \times \mathbb{R}^{s_2}) \times (\text{Mat}_{\mathbb{R}}(s_2, s_3) \times \mathbb{R}^{s_3}) \times \dots \times (\text{Mat}_{\mathbb{R}}(s_{L-1}, s_L) \times \mathbb{R}^{s_L})$$

7. Universal approximation theorem

Wikipedia: Universal Approximation Theorem

From Hornik (1991) [7], pp. 252, Section 2. Results,

$$\mathcal{N}_k^{(n)}(\psi) = \{h : \mathbb{R}^k \to \mathbb{R} | h(x) = \sum_{j=1}^n \beta_j \psi(a_j' x - \theta_j)\} \tag{31}$$

with $\begin{aligned} a &= (\alpha_1, \dots \alpha_k) \\ x &= (\xi_1, \dots, \xi_k)\end{aligned}$ with $a' \equiv a^T \equiv$ transpose of a.

For arbitrary number of hidden layers,

$$\mathcal{N}_k(\psi) = \bigcup_{n=1}^\infty \mathcal{N}_k^{(n)}(\psi)$$

The 2 very important theorems from Hornik (1991) are the following:

Theorem 1. *If ψ unbounded and nonconstant, then $\mathcal{N}_k(\psi)$ dense in $L^p(\mu)$, $\forall$ finite measure μ on $\mathbb{R}^k$*

Theorem 2. *If ψ cont., bounded, nonconstant, then $\mathcal{N}_k(\psi)$ dense in $C(X)$, $\forall$ compact subsets X of $\mathbb{R}^k$, i.e. $\forall f \in C(X)$, $\exists$ sequence, (h_n) s.t. $h_n \xrightarrow{n} f$ uniformly i.e. $\forall$ given $\epsilon > 0$, $\exists N = N(\epsilon)$ (independent of $x \in X \subset \mathbb{R}^k$), s.t. $|h_n(x) - f(x)| < \epsilon \quad \forall x \in X, \quad \forall n \geq N(\epsilon)$*

I will write now a dictionary between Hornik's notation and my notation (take note, Hornik's notation $\equiv$ my notation).

$f \in C(X)$, $f : \mathbb{R}^k \to \mathbb{R}$, $k \equiv d$, so $f : \mathbb{R}^d \to \mathbb{R}$

$\psi \equiv g$, e.g. $g(z) = \frac{1}{1+\exp{(-z)}}$ or $g(z) = \tanh{(z)}$, but equip g with element-wise (component-wise) action, i.e. g as a functor,

$g : \mathbb{R}^k \to \mathbb{R}^k$, i.e. $g : \mathbf{Vec} \to \mathbf{Vec}$.

$g : x_j \mapsto g(x_j)$

Now $a \equiv \Theta \in \text{Mat}_{\mathbb{R}}(d, n)$,

$$g(\Theta x + b) = g(z)$$

i.e. $z \in \mathbb{R}^n$,

$$z := \Theta x + b \text{ i.e. } z_j = \Theta_{jk} x_k + b_j \qquad g(z) \in \mathbb{R}^n$$

and so, notation-wise,

$$\sum_{j=1}^n \beta_j \psi(a_j' x - \theta_j) \equiv \sum_{j=1}^n \beta_j g(\Theta_{jk} x_k + b_j)$$

Consider

$$\begin{gathered}\Theta^{(1)} \in \text{Mat}_{\mathbb{R}}(d, s_2),\ b^{(1)} \in \mathbb{R}^{s_2} \\ \Theta^{(2)} \in \text{Mat}_{\mathbb{R}}(s_2, 1),\ b^{(2)} \in \mathbb{R} \\ h(x) = \Theta^{(2)} g(\Theta^{(1)} x + b^{(1)}) + b^{(2)} \in \mathcal{N}_d^{(s_2)}(g)\end{gathered}$$

so the neural net of L total layers $d = 1$ "input layer", $l = L$ is "output layer" is a tuple $((\Theta, b), g) \in \mathcal{N}_d^{(L)}(g)$

Hornik, Stinchcombe, and White (1989) [8] deals with multi-(hidden) layer networks on pp. 363, on and after Corollary 2.6.

Given training data,

$$\begin{gathered}(X, y) : \mathbb{R} \to (\mathbb{R}^d \times \mathbb{R}^k)^m \\ (X, y)(t) \mapsto (X(t), y(t))\end{gathered} \tag{32}$$

discretize time $t \in \mathbb{R}$,

$$\begin{aligned} &\mathbb{R} \xrightarrow{\text{discretize}} \mathbb{Z} \\ &[0,T] \text{ where } T \in \mathbb{R}^+ \to \{0,1,\dots T-1\} \text{ where } T \in \mathbb{Z}^+ \end{aligned} \tag{33}$$

Consider 4 different feedforwards. Note $y(-1) = 0$.

8. LSTM; Long Short Term Memory

LSTM (Long Short Term Memory), according to Christian Herta

Rewriting Herta's formulation of LSTM, which actually puts in the "cell" memory into some of the input, forget gates, that's different from a "traditional" LSTM (see Wikipedia),

$$\begin{aligned} \text{input gates } i_t &= \psi_{(i)}(\Theta^{(i)}X_t + b^{(i)} + \theta^{(i)}h_{t-1} + W^{(i)}c_{t-1}) \\ f_t &= \psi_{(f)}(\Theta^{(f)}X_t + b^{(f)} + \theta^{(f)}h_{t-1} + W^{(f)}c_{t-1}) \\ c_t &:= f_t \odot c_{t-1} + i_t \odot g_t \\ \text{output gates } o_t &= \psi_{(0)}(\Theta^{(0)}X_t + b^{(0)} + \Theta^{(0)}g_{t-1} + W^{(0)}c_t) \end{aligned} \tag{34}$$

and then finally, not predict yet (I was mistaken) but h here denotes some other "hidden" variable,

$$h_t = o_t \odot \psi_h(c_t) \tag{35}$$

$o_t, c_t \in \mathbb{R}^H$, and so $W^{(i)}, W^{(f)}, W^{(0)} \in \text{Mat}_{\mathbb{R}}(H, s_2)$.

$$\begin{aligned} y_t &= \psi_{(y)}(\Theta^{(y)}h_t + b^{(y)}) \\ \Theta^{(y)} &: \mathbb{R}^{s_L} \to \mathbb{R}^K \end{aligned} \tag{36}$$

$$\begin{array}{ccccccc} (\mathbb{R}^d \times \mathbb{R}^H)^m \times (\mathbb{R}^H) & \xrightarrow{\{\psi_\alpha \circ ((\Theta^{(\alpha)}, b^{(\alpha)}), \theta^{(\alpha)}, W^{(\alpha)})\}_{\alpha=i,f,g}} & (\mathbb{R}^H \times \mathbb{R}^H \times \mathbb{R}^H)^m & \longrightarrow & (\mathbb{R}^H \times \mathbb{R}^H)^m & \longrightarrow & (\mathbb{R}^H)^m \\ X_t, h_{t-1}, c_{t-1} & \longmapsto & (i_t, f_t, g_t) & \longmapsto & c_t, o_t & \longmapsto & h_t \end{array} \tag{37}$$

Consider what we're essentially doing at time step t:

$$\begin{array}{ccccc} ((\mathbb{R}^d \times \mathbb{R}^H) \times (\mathbb{R}^H))^m & \xrightarrow{\{\psi_\alpha \circ ((\Theta^{(\alpha)}, b^{(\alpha)}), \theta^{(\alpha)}, W^{(\alpha)})\}_{\alpha=i,f,g}} & (\mathbb{R}^H \times \mathbb{R}^H)^m & \xrightarrow{(\cdot,\cdot,(\Theta^{(y)}, b^{(y)})} & (\mathbb{R}^H \times \mathbb{R}^H \times \mathbb{R}^H)^m \\ X_t, h_{t-1}, c_{t-1} & \longmapsto & c_t, h_t & \longmapsto & c_t, h_t, y_t \end{array} \tag{38}$$

The recurrence relation is essentially this:

$$X(t), h(t-1), c(t-1) \longmapsto c(t), h(t), y(t) \qquad \forall\, t = 0, 1, \dots T-1 \tag{39}$$

This is the recurrence relation that changes with time t and in the language of `theano`, for `theano.scan` it is the argument value for argument `sequences`.

Notice how h, c change over time. These are the sequences we want to take in as input and output. $X(t)$ is the sequences we want to "iterate over." $X(t)$ doesn't get modified by our operations over time t (than what is given). $y(t)$ is an output we desire. So in the language of `theano`, for `theano.scan`, $X(t)$ goes to the argument value for argument `sequences`, as it's part of the "list of Theano variables or dictionaries describing the sequences `scan` has to iterate over" and since $X = X(t)$ for time t, the "`taps`" is `[0]`. h, c, y is expected to be the return value of the Python function describing a single time step, and "the order of the outputs is the same as the order of `outputs_info`.

Look at the parameters:

$$\begin{aligned} &(\Theta^{(g)}, b^{(g)}), \theta^{(g)} \in (\text{Mat}_{\mathbb{R}}(d,H) \times \mathbb{R}^H \times \text{Mat}_{\mathbb{R}}(H,H) \\ &(\Theta^{(\alpha)}, b^{(\alpha)}), \theta^{(\alpha)}, W^{(\alpha)} \in (\text{Mat}_{\mathbb{R}}(d,H) \times \mathbb{R}^H \times \text{Mat}_{\mathbb{R}}(H,H) \times \text{Mat}_{\mathbb{R}}(H,H) \\ &(\Theta^{(y)}, b^{(y)}) \in (\text{Mat}_{\mathbb{R}}(H,K) \end{aligned} \tag{40}$$

These parameters are what you put into, in the language of `theano`, for the argument value of `non_sequences` of `theano.scan`.

8.1. How to choose the number of hidden layers and nodes in a neural net.

Part 4. Support Vector Machines (SVM)

The clearest and most mathematically rigorous (and satisfying) introductory exposition on support vector machines (SVM) comes out of a Bachelor's thesis from Nowak (2008) [9]. There is a lot of material that tries to talk about SVM, but the implementation either boils down to showing how to turn the crank on a black-box solution, or is too verbose without saying anything substantial. I'll include references and links of the material I looked at and didn't find as helpful as Nowak (2008) [9].

Lecture12 pdf slides for Ng's Machine Learning Intro. for coursera

Support Vector Machine (and Statistical Learning Theory) Tutorial by Jason Weston, NEC Labs America

Wikipedia page for Support Vector Machine

Support Vector Machines and Generalisation in HEP Not much real generalization going on here other than a recap of literally what's exactly in Shawe-Taylor and Cristianini (2000) [10].

`https://www.cs.cornell.edu/people/tj/publications/joachims_99a.pdf`

9. From linear classifier as a hyperplane, (big) margin, to linear support vector machine (SVM), and Lagrangian dual (i.e. conjugate variables, conjugate momenta)

Intuitively, we seek to find a boundary line that'll draw a line that separates the data points into distinct K (usually $K = 2$) classes to classify the data points. Then, this boundary line will help to predict what class a new data point would fall into, be classified to be. For a linear model, i.e. "linear discriminator", what we're trying to do is

find

$$\theta \in \mathbb{R}^d \backslash \{0\},\ b \in \mathbb{R}$$

s.t.

$$y^{(i)}(\langle \theta, x^{(i)} \rangle + b) - 1 \geq 0 \qquad \forall\, i = 1, \dots, m \tag{41}$$

where $\|\theta\|$ is minimal. It is minimal because, since the distance between 2 hyperplanes,

$$\langle \theta, x \rangle - b = \pm 1 \qquad \text{(defining equations for hyperplanes)}$$

is

$$\frac{2}{\|\theta\|} \qquad \text{(distance between 2 hyperplanes)}$$

Thus, we want the "margins", that distance between hyperplanes separating the input data points, to be as big as possible, and so we want $\|\theta\|$ small.

Consider this cost functional, called "Lagrangian", that we want to minimize:

$$\mathcal{L}((\theta, b), \lambda) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle \theta, x^{(i)} \rangle - b - 1) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle \theta, x^{(i)} \rangle - b) + \sum_{i=1}^m \lambda_i \tag{42}$$

Note that

$$f_0((\theta, b)) := \frac{1}{2}\|\theta\|^2 (\text{objective function (slightly modified)}) \tag{43}$$

is the objective function, what we want to minimize.

The KKT condition tells us that (θ, b) makes $\mathcal{L}$ a minimum for a certain λ:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta_j} &= 0 = \theta_j - \sum_{i=1}^m \lambda_i y^{(i)} x_j^{(i)} \\ \frac{\partial \mathcal{L}}{\partial b} &= 0 = -\sum_{i=1}^m \lambda_i y^{(i)} \end{aligned} \tag{44}$$

Note that this step in taking the partial derivatives of $\mathcal{L}$ in Eq. 44 is analogous to the construction/computation of dual "conjugate" variables, conjugate momentum, in physics.

Notice then that

$$\begin{gathered} \frac{1}{2}\|\theta\|^2 = \frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \text{ and} \\ \sum_{i=1}^m \lambda_i y^{(i)} (\langle \theta, x^{(i)}\rangle - b) = \sum_{i=1}^m \lambda_i y^{(i)} (\sum_{j=1}^m \lambda_j y^{(j)} \langle x^{(j)}, x^{(i)}\rangle) \end{gathered} \tag{45}$$

$$\Longrightarrow \mathcal{L}((\theta, b), \lambda) = -\frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)}\rangle + \sum_{i=1}^m \lambda_i \tag{46}$$

10. So-called "Kernel trick"; feature space is a Hilbert space

The so-called "feature space" F is a Hilbert space H, $\Phi : \mathbb{R}^d \to H$, equipped with inner product

$$\langle \Phi(x), \Phi(y)\rangle = K(x,y) \tag{47}$$

with $K : \mathbb{K}^d \times \mathbb{K}^d \to \mathbb{K}^K$ being called the kernel function. Recall that the feature space F had been introduced to represent the process of preprocessing input data X. For example, given a single input data example, $X = (X_1, \dots, X_d) \in \mathbb{R}^d$, maybe we'd want to consider polynomial features, linear combinations of various orders of monomials X_iX_j or $X_i^2X_j$, and so on. Then Φ represents the map from X to all these features.

The essense of the kernel trick is this: the explicit form of Φ need not be known, nor even the space H. Only the kernal function K form needs to be guessed at.

And so even if we now have to modify our Eq. 42 to account for this preprocessing map Φ, applied first to our training data $x^{(i)} \equiv X^{(i)}$ (Novak's notation vs. Andrew Ng's notation), we essentially still have the same form, formally.

Keep in mind the whole point of this nonlinear preprocessing map Φ - we want to keep the linear discrimination procedure with the weight, or parameter θ, and intercept b, being this linear model on the feature space (Hilbert space) F. We're linear in F. But we're nonlinear in the input data $X = \{X^{(1)}, \dots X^{(m)}\}$.

So,

$$\begin{gathered} \mathcal{L}((\theta, b), \lambda) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)} (\langle \theta, \Phi(x^{(i)})\rangle - b - 1) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)} (\langle \theta, x^{(i)}\rangle - b) + \sum_{i=1}^m \lambda_i \text{ and so} \\ \frac{\partial \mathcal{L}}{\partial \theta_j} = 0 = \theta_j - \sum_{i=1}^m \lambda_i y^{(i)} \Phi(x)_j^{(i)} \\ \Longrightarrow \mathcal{L}((\theta, b), \lambda) = -\frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)} y^{(j)} \langle \Phi(x^{(i)}), \Phi(x^{(j)})\rangle + \sum_{i=1}^m \lambda_i \end{gathered} \tag{48}$$

10.1. Dealing with Errors, (non-negative) slack variables, dealing with not-necessarily perfectly separable data.

First, loosen the strict constraint $y^{(i)}(\langle \theta, x^{(i)}\rangle - b) \geq 1$ by introducing *non-negative* slack variables ξ_i, $i = 1 \dots m$,

$$y^{(i)}(\langle \theta, x^{(i)}\rangle - b) \geq 1 - \xi_i, \qquad \forall\, i = 1, 2, \dots m \tag{49}$$

Simply add ξ to the objective function to implement penalty (for "too much slack"):

$$f_0(\theta, b, \xi) = \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^m \xi_i \tag{50}$$

So then the total Lagrangian becomes

$$\mathcal{L}(\theta, b, \xi, \lambda, \mu) = \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^m \xi_i - \sum_{i=1}^m \lambda_i (y^{(i)}(\langle \theta, x^{(i)}\rangle - b) - 1 + \xi_i) - \sum_{i=1}^m \mu_i \xi_i \tag{51}$$

where the constraint is turned into a Lagrange-multiplier type relation:

$$\xi_i \geq 0 \Longrightarrow \mu_i(\xi_i - 0) \qquad \forall\, i = 1, \dots m \tag{52}$$

$-\mu_i\xi_i$ is indeed a valid cost (penalty) functional (if $\xi_i < 0$, $-\mu_i\xi_i > 0$, and there's more penalty as ξ_i gets more negative. Note that I understood this cost or penalty accounting, given an *inequality constraint*, from reading notes from here, .

11. Dual Formulation

$$\begin{aligned} \text{min.} \quad & W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)}) \\ \text{s.t.} \quad & \sum_{i=1}^m \lambda_i y^{(i)} = 0 \\ & 0 \leq \lambda_i \leq C \end{aligned} \tag{53}$$

At this point, Eq. 82 is what I could consider the "theoretical gold" version. Further modification of this formulation are really to efficiently implement this on the computer (or microprocessor!). But the schemes should respect this "gold" version and compute what this is and say.

11.1. Implementation.

Wotao Yin's notes had a terse, but to-the-point, survey/summary of optimization, in particular non-linear optimization with inequality constraints, for his courses 273a and Math 164, Algorithms for constrained optimization. In both course notes, the material is "taken from the textbook Chong-Zak, 4th. Ed." So we'll refer to Chong and Zak (2013) [11].

From Ch. 22 "Algorithms for Constrained Optimization", 2nd. Ed., from pp. 439, Sec. 22.2 Projections, consider $\Omega \subset \mathbb{R}^d$,

$$\Omega = \{\mathbf{x} | l_i \leq x_i \leq u_i,\ i = 1 \dots d\}$$

Let $\Pi \equiv$ projection operator. Define the above case as such:

$$\begin{gathered} \forall\, \mathbf{x} \in \mathbb{R}^d,\ y := \Pi[x] \in \mathbb{R}^d \\ y_i \equiv \begin{cases} u_i & \text{if } x_i > u_i \\ x_i & \text{if } l_i \leq x_i \leq u_i \\ l_i & \text{if } x_i < l_i \end{cases} \end{gathered}$$

11.1.1. *Projected Gradient descent.* .

Implement $\sum_{i=0}^m \lambda_i y^{(i)} = 0$, consider the orthogonal projector matrix (operator)

$$\mathbf{P} := \mathbf{1}_{\mathbb{R}^d} - A^T (AA^T)^{-1} A \tag{54}$$

If $m=1$, then

$$\text{Proj}_\Omega(\mathbf{y}) = \mathbf{y} - \frac{\mathbf{a}_1^T \mathbf{y} - b}{\|\mathbf{a}_1\|^2}\mathbf{a}_1$$

If $m>1$, then

$$\text{Proj}_\Omega(\mathbf{y}) = (1_{\mathbb{R}^d} - A^T(AA^T)^{-1}A)\mathbf{y} + A^T(AA^T)^{-1}\mathbf{b}$$

For the linear (but it's an equality) constraint

$$\sum_{i=1}^m \lambda_i y^{(i)} = 0$$

so

$$\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\mathbf{y}) = \left(\mathbf{y} - \frac{\sum_{i=1}^m y^{(i)}(\mathbf{y})_i}{\sum_{i=1}^m (y^{(i)})^2}(y^{(i)})\mathbf{e}_i \right) \tag{55}$$

Narasimhan's Optimization Tutorial 3, Projected Gradient Descent, Duality had some concrete pseudocode for the projected gradient descent [12].

In summary,

$$\begin{gathered} \text{we seek to minimize} \\ W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)}y^{(j)}K(X^{(i)},X^{(j)}) \qquad \forall\, i=1,2,\dots m \\ \text{by iterating } t=0,1,\dots, \text{ as such:} \\ \lambda_i'(t+1) := \lambda_i(t) - \alpha \text{grad} W(\lambda) \\ \lambda_i''(t+1) := \mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\lambda_i'(t+1)) \\ \lambda_i(t+1) := \Pi_{0\leq \lambda_i \leq C}(\lambda_i''(t+1)) \end{gathered} \tag{56}$$

where

$$\begin{gathered} \mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\lambda_i'(t+1)) = \lambda_i'(t+1) - \frac{\sum_{i=1}^m y^{(i)}\lambda_i'(t+1)}{\sum_{i=1}^m (y^{(i)})^2} y^{(i)} \\ \Pi_{0\leq\lambda_i\leq C}(\lambda_i''(t+1)) = \begin{cases} C & \text{if } \lambda_i''(t+1) > C \\ \lambda_i''(t+1) & \text{if } 0 \leq \lambda_i''(t+1) \leq C \\ 0 & \text{if } \lambda_i'(t+1) < 0 \end{cases} \end{gathered} \tag{57}$$

11.1.2. *Computing b, the intercept, with a good algebra tip: multiply both sides by the denominator.* Bishop (2007) [13], on pp. 330 of Ch. 7, Sparse Kernel Machines, gave a very good (it resolved possible numerical instabilities) prescription on how to compute the intercept b, given λ, which would then give us the function that can make predictions $\widehat{y}$ on input data example $X^{(i)} \in X$. It's worth expounding upon here.

For any support vector (Bishop called it a support vector; what I think it's equivalent to is that we've trained on our training set $(X,y)^{\text{train}}$, and this is 1 of the training examples) $X^{(i)}$, $i=1\dots m$,

$$y^{(i)} f(X^{(i)}) = 1 \tag{58}$$

. Then using

$$\begin{gathered} f(x) := \sum_{i=1}^m y^{(i)}\lambda_i^* K(X^{(i)},x) + b \\ \implies y^{(i)}\left(\sum_{j=1}^m y^{(j)}\lambda_j^* K(X^{(j)},X^{(i)}) + b \right) = 1 \end{gathered} \tag{59}$$

Although we can solve this equation for b with algebra/arithmetic for our arbitrarily chosen support vector, it's numerically more stable to 1st. multiply through by $y^{(i)}$, using $(y^{(i)})^2 = 1$, and then averaging over all support vectors.

$$\begin{gathered} \sum_{j=1}^m y^{(j)}\lambda_j^* K(X^{(j)},X^{(i)}) + b = y^{(i)} \\ \implies b = \frac{1}{m}\left(\sum_{i=1}^m y^{(i)} - \sum_{i,j=1}^m y^{(j)}\lambda_j^* K(X^{(j)},X^{(i)}) \right) \end{gathered} \tag{60}$$

11.1.3. *Prediction (with SVM).*

$$\begin{gathered} \widehat{y}(X) = \sum_{i=1}^m y^{(i)}\lambda_i^* K(X^{(i)},X) + b^* \\ \widehat{y} : \mathbb{R}^d \to \{0,1,\dots,K-1\} \end{gathered} \tag{61}$$

Clarke, Fokoue, and Zhang (2009) [14]

12. Support Vector Machines (SVM) natively implemented in theano, entirely on the GPU with the CUDA backend, with constrained gradient descent

12.1. **Executive Summary.** I implemented SVM natively in theano, and can run entirely on the GPU(s) (through the CUDA C/C++ backend). Solving the *constrained optimization* problem to train a SVM here used *parallel reduce* algorithms; in fact a parallel reduce nested in side a parallel reduce. The work complexity achieved in this case should be of $O(2\log m)$ where m is the total number of training examples, as opposed to $O(m^2)$ for Quadratic Programming (QP), such as Sequential Minimal Optimization (SMO). Training on the same data set for vehicles as previous work for C/C++ library `libsvm` that uses SMO, `SVM_parallel` (what I call the described implementation here) achieves an accuracy of 95.1% on test data, as opposed to 87.8% for `libsvm`. What I'd like to do in the future is to train and test on larger datasets ($m > 10000$) to test `SVM_parallel` for its promising scalability, and to implement it as the outer layer of a deep neural network (DNN) for "Deep SVM".

12.2. **Motivations and Introductions.** Support Vector Machines (SVM) can be used for (binary) classification in supervised learning on labeled data, being able to learn non-linear, higher-dimensional features and to predict a boundary line or discriminator between classes, through the so-called *kernel trick*, which is really to presume a higher-dimensional Hilbert space to represent feature space $\mathcal{F}$ (i.e. $\mathcal{F}$ is a Hilbert space).

I considered the proposition of having, as a final, outer "layer," of a deep neural network (DNN) to be a support vector machine. Could it outperform the same DNN with a sigmoid or softmax function at the final layer?

To my knowledge, there was not a native implementation of SVM on theano, a Python framework for deep learning/DNNs. Being that I sought to speed up learning/computation on the GPU(s) through the theano CUDA backend, it would seem to defeat the GPU speedup advantages if from the very last layer, a large global memory transfer had to occur from the GPU (with the last DNN layer), to a host CPU, *serial*, implementation of SVM. Global memory transfers are prohibitive expensive, for latency, i.e. time-wise, between host CPU and GPUs.[3]

The outline/plan/highlights for this (short) paper is as follows: in

- 12.3, I review or summarize points in the theory for SVM, give derivations, etc., in which (this has all been done before; I sought to give a concise review)

  - 12.3.1, I recap basic, elementary concepts motivating the linear discriminator concept and what hyperplanes are, and how they're defined by a *linear* function
  - 12.3.2 - I continue to review and present derivations for the *Lagrangian*, a Lagrange multiplier problem, we want to minimize, and apply the usual Karush-Kuhn-Tucker (KKT) condition to make progress in deriving the constrained optimization problem we seek to solve,
  - 12.4 kernel trick, with the feature space $\mathcal{F}$ as a Hilbert space, 12.4.1 slack variables to deal with non-perfectly-separable data, and more derivation that was done before, but made explicitly here
- 12.5, the *constrained optimization* problem we wish to solve for SVM
- 12.6, I translate how our constrained optimization problem is to be solved with *projected gradient descent* or "constrained gradient descent" (as the projection operators enforce constraint equalities and inequalities). As noted, this method/algorithm was chosen to utilize the (very useful) `grad` method in the `theano` software package.
  The Eqns. 86, 87 at the end of 12.6.1 is the crux of the training method considered in this paper and code for SVM and was directly referred to when implemented in code.
  - I also show how to compute, in a numerically stable manner, the intercept b, after λ_i Lagrange multipliers are found, and how to compute predictions $\widehat{y}$, in 12.6.2, 12.6.3
- 12.7 the *constrained gradient descent* to solve our constrained optimization problem is implemented and I detail its implementation using software package `theano`, and especially its CUDA backend, so to run solely on the *GPU*. I give the rationale in deploying this constrained gradient descent as opposed to the Sequential Minimal Optimization (SMO) usually used in the predominant SVM software package (in C/C++) `libsvm`. Noteworthy, I also show that **work complexity goes from $O(m^2)$ to $O(2\log(m))$.**
- 12.7.1 briefly tells where the code is made available and the 1-to-1 correspondence between the code and mathematical formulation. I also note that *novel use of theano's reduce within reduce.*
- 12.8 has *results* that I try to compare with sample datasets used previously, that I've trained as quickly as possible. Look here for the results; better yet, feel free to try the code and jupyter notebook and share benchmarks.[2]

12.3. Concise Mathematical Review/Summary of the theory for SVM.

12.3.1. *Hyperplanes and distances to motivate the linear discriminator concept; Support Vector Machine name.* I'll recap basic, elementary concepts, from Clarke, Fokoue, and Zhang (2009) [14], that motivate the concept of a linear discriminator classifying input data X.

Consider $\theta \in \mathbb{R}^d$, and a linear function y,

$$\begin{aligned} & y : \mathbb{R}^d \to \mathbb{R} \\ & y(x) := \langle \theta, x \rangle + b \end{aligned} \tag{62}$$

Consider a "level set" at real number value $c \in \mathbb{R}$, $H_c(\theta,b)$:

$$H_c(\theta, b) := \{ x | y(x) = \langle \theta, x\rangle + b = c \} \tag{63}$$

where $\dim H_c(\theta,b) = d-1$ is a *hyperplane*.

$\theta \in \mathbb{R}^d$ is the normal vector to this hyperplane, since,

$$\begin{gathered} \forall\, x^{(i)}, x^{(j)} \in H_c(\theta,b), \text{ then} \\ \langle \theta, x^{(i)} \rangle + b = c = \langle \theta, x^{(j)} \rangle + b \Longrightarrow \langle \theta, x^{(i)} - x^{(j)} \rangle = 0 \end{gathered}$$

and since $x^{(i)} - x^{(j)} \in TH_c(\theta,b)$, i.e. $x^{(i)} - x^{(j)}$ belongs in the tangent space to $H_c(\theta,b)$, $TH_c(\theta,b)$, then θ, in general, is normal to the hyperplane ($\langle \theta, x^{(i)} - x^{(j)} \rangle = 0$).

[2] github:ernestyalumni/MLgrabbag/ML, github:ernestyalumni/MLgrabbag SVM theano.ipynb

Given $z \in \mathbb{R}^d$, what is the distance from z to this hyperplane $H_c(\theta,b)$, $d(z, H_c(\theta,b))$? Consider $z* = z + t\theta \in H_c(\theta,b)$. Then

$$\begin{gathered} \langle \theta, z^* \rangle + b = c = \langle \theta, z \rangle + t\langle \theta, \theta \rangle + b = c \Longrightarrow t = \frac{c - b - \langle \theta, z\rangle}{\|\theta\|^2} \\ \text{and so } d(z, H_c(\theta,b)) = \|t\theta\| = \frac{|\langle \theta, z \rangle + b - c|}{\|\theta\|} \end{gathered}$$

Thus, for the perpendicular distance between 2 parallel hyperplanes, $H_c(\theta,b)$, $H_{c'}(\theta,b)$, can be found: choose a pt. from $H_c(\theta,b)$, without loss of generality, s.t. $z = \left(\frac{c-b}{\theta_1}, 0, \dots 0 \right)$, so that

$$\langle \theta, z \rangle + b = c \Longrightarrow \theta_1 z^1 = c - b$$

Then

$$d(H_c(\theta,b), H_{c'}(\theta,b)) = \frac{|\langle \theta, z\rangle + b - c'|}{\|\theta\|} = \frac{|c - b + b - c'|}{\|\theta\|} = \frac{|c - c'|}{\|\theta\|} \tag{64}$$

Given an input (data) domain $\mathcal{X} \subseteq \mathbb{R}^d$, for the case of binary classification, with total number of classes $K = 2$, we can consider representing the outcomes y for each input data example, $X \in \mathbb{R}^d$, in 2 ways:

$$y \in \{-1, 1\} \textbf{ or } \; y \in \{0,1\} \text{ for } y \in \{0, 1, \dots K-1\}\, (K=2) \tag{65}$$

What ends up happening is that the distance between 2 hyperplanes, $c = -1, c' = 1$ vs. $c = 0, c' = 1$, respectively, changes, as $d(H_c(\theta,b), H_{c'}(\theta,b)) = \frac{|c-c'|}{\|\theta\|}$, but its absolute value doesn't matter. What matters is the form of $y : \mathbb{R}^d \to \mathbb{R}$, of Eq. 62 which defines the hyperplane in Eq. 63, notably in θ, b. The lesson is to *be consistent with what the value of y is to define what class X belongs to.* For instance, Bishop (2007) [13] and Clarke, Fokoue, and Zhang (2009) [14] chooses to consider $y \in \{-1, 1\}$, $\forall\, X$ and I'll do the same here.

The name "support vectors" seems to come from this intuitive notion: $\theta \in \mathbb{R}^d$,b are determined from m input data examples $X^{(i)} \in \mathbb{R}^d$, $\forall\, i = 1, 2, \dots d$, and $\forall\, i$, the corresponding class label $y \in \mathbb{Z}$. $\forall\, X^{(i)} \in \mathbb{R}^d$, imagine attaching normal vectors of the form $t\theta$, $t \in \mathbb{R}$ that extend out to the respective hyperplane, determined by $y^{(i)}$. These imagined vectors "support" the respective hyperplane.

An important takeaway is that the equation defining the hyperplane $H_c(\theta,b)$ in Eq. 63 is *linear.*

12.3.2. *Margins, cost functional or "Lagrangian", dual formulation.* With output, outcome $y \in \{-1, 1\}$, $\forall\, X$ input data example, the distance between the 2 hyperplanes, which are level sets of $c = -1, c' = 1$, is

$$\frac{2}{\|\theta\|}$$

The method of SVM seeks to maximize this distance, also known as "margin", to make margins as big as possible.

Clearly, this is equivalent to minimizing $\frac{1}{2}\|\theta\|^2$, with $\frac{1}{2}$ multiplication factor chosen, without loss of generality, to make taking derivatives of θ easier.

But we also have the following constraints. We want to have a "margin" of $\frac{2}{\|\theta\|}$ between the hyperplanes that'll separate the input data examples $X^{(i)}, \forall\, i = 1, 2, \dots m$, for different classes, in this binary classification class, of those with $y^{(i)} \in \{-1, 1\}$, and so those $X^{(i)}$'s will "fall far away" from this "margin" and remain within its corresponding hyperplane $H_{c'=1}(\theta,b)$ or $H_{c=1}(\theta,b)$, thus defining these inequalities:

$$y^{(i)}(\langle \theta, x^{(i)} \rangle + b) - 1 \geq 0 \qquad \forall\, i = 1, \dots, m \tag{66}$$

So we want to find

$$\theta \in \mathbb{R}^d \backslash \{0\},\, b \in \mathbb{R}$$

s.t.

$$y^{(i)}(\langle \theta, x^{(i)} \rangle + b) - 1 \geq 0 \qquad \forall\, i = 1, \dots, m$$

where $\frac{2}{\|\theta\|}$ is maximized, or equivalently, defining the so-called *objective function* $f_\theta(\theta, b)$, minimize $f_\theta(\theta, b)$:

$$f_\theta(\theta, b) := \frac{1}{2}\|\theta\|^2 \qquad \text{(objective function)} \tag{67}$$

Consider then this cost functional, also known as the "Lagrangian", which we want to *minimize*.

$$\mathcal{L}((\theta, b), \lambda) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle + b - 1) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle + b) + \sum_{i=1}^m \lambda_i \tag{68}$$

Of note, we introduced *Lagrangian multipliers* λ_i, $\forall i \in 1, 2 \dots m$, $\lambda_i \in \mathbb{R}$, to account for each of the constraints given in Eq. 66.

The Karush-Kuhn-Tucker (KKT) condition tells us that (θ, b) makes $\mathcal{L}$ a minimum for a certain λ (and that these λ_i's exist), and that these relations hold:[9], [11]:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta_j} &= 0 = \theta_j - \sum_{i=1}^m \lambda_i y^{(i)} x_j^{(i)} \qquad j = 1, 2 \dots d \\ \frac{\partial \mathcal{L}}{\partial b} &= 0 = -\sum_{i=1}^m \lambda_i y^{(i)} \end{aligned} \tag{69}$$

and

$$\lambda_i \geq 0 \qquad \forall\, i = 1, 2, \dots m \tag{70}$$

,

$$\sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle) + b - 1) = 0 \tag{71}$$

$\forall\, i = 1, 2, \dots m$, we want input data example $X^{(i)}$ to be "far away" from the boundary line, or, i.e. to give enough "margin" from the other class's hyperplane, and so in general, $(\langle\theta, x^{(i)}\rangle) - b - 1)$ will be non-zero in Eq. 71. So this condition is equivalently

$$\sum_{j=1}^m \lambda_i y^{(i)} = 0 \tag{72}$$

It's interesting to see that the step in taking the partial derivatives of $\mathcal{L}$ in Eq. 69 is analogous to the construction/computation of dual "conjugate" variables, conjugate momentum, in physics.

Notice then that

$$\begin{aligned} &\frac{1}{2}\|\theta\|^2 = \frac{1}{2}\sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)}\rangle \text{ and} \\ &\sum_{i=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle + b) = \sum_{i=1}^m \lambda_i y^{(i)}(\sum_{j=1}^m \lambda_j y^{(j)} \langle x^{(j)}, x^{(i)}\rangle) \end{aligned} \tag{73}$$

$$\Longrightarrow \mathcal{L}((\theta, b), \lambda) = -\frac{1}{2}\sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)}\rangle + \sum_{i=1}^m \lambda_i \tag{74}$$

12.4. **So-called "Kernel trick"; feature space is a Hilbert space.** The so-called "feature space" $\mathcal{F}$ is a Hilbert space $\mathcal{H}$, $\Phi : \mathbb{R}^d \to \mathcal{H}$, equipped with inner product

$$\langle\Phi(x), \Phi(y)\rangle = K(x, y) \tag{75}$$

with $K : \mathbb{K}^d \times \mathbb{K}^d \to \mathbb{K}^K$ being called the kernel function. Recall that the feature space $\mathcal{F}$ had been introduced to represent the process of preprocessing input data X. For example, given a single input data example, $X = (X_1, \dots, X_d) \in \mathbb{R}^d$, maybe we'd want to consider polynomial features, linear combinations of various orders of monomials $X_i X_j$ or $X_i^2 X_j$, and so on. Then Φ represents the map from X to all these features.

As both a pedantic remark and academic question, I had denoted $\mathbb{K}$ to be, in general, a *field* - (very familiar) examples of fields are $\mathbb{K} = \mathbb{R}, \mathbb{C}, \mathbb{Z}$, the real numbers, complex numbers, integers, respectively. Many times, input data that we receive could take on discrete values, meaning $X^{(i)} \in \mathbb{Z}$. Would there be issues, in proving existence, continuity, and differentiability, throughout derivations for these SVM algorithms, if the underlying field $\mathbb{K}$ is not the real number line $\mathbb{R}$?

Nevertheless, the essense of the kernel trick is this: the explicit form of Φ need *not be known*, nor even the space $\mathcal{H}$. Only the kernal function K form needs to be guessed at.

And so even if we now have to modify our Eq. 68 to account for this preprocessing map Φ, applied first to our training data $X^{(i)}$, we essentially still have the same form, formally.

Keep in mind the whole point of this nonlinear preprocessing map Φ - we want to keep the linear discrimination procedure with the weight, or parameter θ, and intercept b, being this linear model on the feature space (Hilbert space) F. We're *linear* in $\mathcal{F}$. But we're *nonlinear* in the input data $X = \{X^{(1)}, \dots X^{(m)}\}$'s domain.

So,

$$\begin{aligned} &\mathcal{L}((\theta, b), \lambda) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta, \Phi(x^{(i)})\rangle + b - 1) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle + b) + \sum_{i=1}^m \lambda_i \text{ and so} \\ &\frac{\partial \mathcal{L}}{\partial \theta_j} = 0 = \theta_j - \sum_{i=1}^m \lambda_i y^{(i)} \Phi(x)_j^{(i)} \\ &\Longrightarrow \mathcal{L}((\theta, b), \lambda) = -\frac{1}{2}\sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} \langle\Phi(x^{(i)}), \Phi(x^{(j)})\rangle + \sum_{i=1}^m \lambda_i = \mathcal{L}(X, y, \lambda) \end{aligned} \tag{76}$$

Note that we'll now want to *maximize* this dual formulation $\mathcal{L}(X, y, \lambda)$.

12.4.1. *Dealing with Errors, (non-negative) slack variables, dealing with not-necessarily perfectly separable data.* First, "loosen the strict constraint" $y^{(i)}(\langle\theta, x^{(i)}\rangle + b) \geq 1$ by introducing *non-negative* slack variables ξ_i, $i = 1 \dots m$,

$$y^{(i)}(\langle\theta, x^{(i)}\rangle - b) \geq 1 - \xi_i, \qquad \forall\, i = 1, 2, \dots m \tag{77}$$

Simply add ξ to the objective function to implement penalty (for "too much slack"), with a "regularization" constant C (in analogy to regularization in the linear regression or logistic regression classifier methods):

$$f_0(\theta, b, \xi) = \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^m \xi_i \tag{78}$$

So then the total Lagrangian becomes

$$\mathcal{L}(\theta, b, \xi, \lambda, \mu) = \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^m \xi_i - \sum_{i=1}^m \lambda_i(y^{(i)}(\langle\theta, x^{(i)}\rangle + b) - 1 + \xi_i) - \sum_{i=1}^m \mu_i \xi_i \tag{79}$$

where the constraint is turned into a Lagrange-multiplier type relation:

$$\xi_i \geq 0 \Longrightarrow \mu_i(\xi_i - 0) \qquad \forall\, i = 1, \dots m \tag{80}$$

$-\mu_i \xi_i$ is indeed a valid cost (penalty) functional (if $\xi_i < 0$, $-\mu_i \xi_i > 0$, and there's more penalty as ξ_i gets more negative. I understood this cost or penalty accounting, given an *inequality constraint*, from reading notes from here, http://www.pitt.edu/~jrclass/opt/notes4.pdf).

If we "turn the crank" and take partial derivatives of $\mathcal{L}$, with respect to ξ_i, finding its "conjugate momentum dual", we'll actually see that $\mathcal{L}$ has *no* dependence on ξ_i:

$$
\begin{gathered}
\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \lambda_i - \mu_i = 0 \\
\Longrightarrow \\
\text{since} \quad C - \lambda_i = \mu_i \quad \Longrightarrow C \geq \lambda_i \\
\mu_i \geq 0 \text{ is given} \\
\mathcal{L}(\theta, b, \xi, \lambda, \mu) = \frac{1}{2}\|\theta\|^2 - \sum_{i=1}^m \lambda_i (y^{(i)}(\langle \theta, \Phi(x^{(i)})\rangle)) + \sum_{i=1}^m \lambda_i = \mathcal{L}((\theta, b), \lambda)
\end{gathered} \tag{81}
$$

ξ, μ no longer appear in the dual Lagrangian, $\mathcal{L}(X, y, \lambda)$, which we want to *maximize*, nor in the so-called "primal" Lagrangian, $\mathcal{L}((\theta, b), \lambda)$.

12.5. **Dual Formulation.** Denoting $W(\lambda) := -\mathcal{L}(X, y, \lambda)$,

$$
\boxed{
\begin{aligned}
&\text{minimize.} \quad W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2} \sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)}) \\
&\text{s.t.} \quad \sum_{i=1}^m \lambda_i y^{(i)} = 0 \\
&\qquad 0 \leq \lambda_i \leq C \qquad \forall \, i = 1, 2 \dots m
\end{aligned}
} \tag{82}
$$

At this point, Eq. 82 is what I could consider the "theoretical gold" version. Further modification of this formulation are really to efficiently implement this on the computer (or microprocessor!). But the schemes should respect this "gold" version and compute what this is and say.

12.6. **Constrained Optimization.** Wotao Yin's notes had a terse, but to-the-point, survey/summary of optimization, in particular nonlinear optimization with inequality constraints, for his courses 273a and Math 164, Algorithms for constrained optimization. In both course notes, the material is "taken from the textbook Chong-Zak, 4th. Ed." So we'll refer to Chong and Zak (2013) [11].

From Ch. 22 "Algorithms for Constrained Optimization", 2nd. Ed., pp. 439, Sec. 22.2 "Projections", consider $\Omega \subset \mathbb{R}^d$, with

$$
\Omega = \{ \mathbf{x} | l_i \leq x_i \leq u_i, \, i = 1 \dots d \}
$$

Let us denote $\Pi \equiv$ projection operator. Let us mathematically formulate how projection operator Π maps a point $\mathbf{x} \in \mathbb{R}^d$ onto the subset $\Omega \subset \mathbb{R}^d$ defined above:

$$
\begin{gathered}
\forall \, \mathbf{x} \in \mathbb{R}^d, \, y := \Pi[x] \in \mathbb{R}^d \\
y_i \equiv \begin{cases} u_i & \text{if } x_i > u_i \\ x_i & \text{if } l_i \leq x_i \leq u_i \\ l_i & \text{if } x_i < l_i \end{cases}
\end{gathered} \tag{83}
$$

12.6.1. *Projected Gradient descent.* We want to minimize $W(\lambda)$ in Eq. 82. The software package `theano` provides the graph-generating method `grad`, which automatically computes the symbolic gradient of a scalar-valued function of symbolic (theano) variables. This `grad` has been very useful for automating the computation of the so-called "back-propagation" step of machine learning/deep learning.

We would like to reuse this useful theano method for SVM. Therefore I sought out a solution to our constrained optimization problem that'll involve computing gradients at each iteration, but subject to our constraint equality and inequalities.

We already know how to deal with constraint *inequalities* via the projection operator in Eq. 83. And note that this can be simply implemented in Python/theano with a `if/else` statement(s) and `theano.tensor.switch`, respectively.

To implement the constraint *equality*, $\sum_{i=0}^m \lambda_i y^{(i)} = 0$, consider the orthogonal projector matrix (operator)

$$
\mathbf{P} := \mathbf{1}_{\mathbb{R}^d} - A^T (AA^T)^{-1} A \tag{84}
$$

with A being a transformation from $\mathbb{R}^d$ to $\mathbb{R}^m$, i.e. $A : \mathbb{R}^d \to \mathbb{R}^m$, and where $Ax = b$ is the constraint equality (written in its most general form) [11].

So for where $\Omega = \{X | AX = b\}$, if $m = 1$, then

$$
\text{Proj}_\Omega(\mathbf{y}) = \mathbf{y} - \frac{\mathbf{a}_1^T \mathbf{y} - b}{\|\mathbf{a}_1\|^2} \mathbf{a}_1
$$

If $m > 1$, then

$$
\text{Proj}_\Omega(\mathbf{y}) = (1_{\mathbb{R}^d} - A^T (AA^T)^{-1} A) \mathbf{y} + A^T (AA^T)^{-1} \mathbf{b}
$$

For the linear (equality) constraint

$$
\sum_{i=1}^m \lambda_i y^{(i)} = 0
$$

we have

$$
\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)} = 0}(\mathbf{y}) = \left(\mathbf{y} - \frac{\sum_{i=1}^m y^{(i)} (\mathbf{y})_i}{\sum_{i=1}^m (y^{(i)})^2} (y^{(i)}) \mathbf{e}_i \right) \tag{85}
$$

In summary,

$$
\boxed{
\begin{gathered}
\text{we seek to minimize} \\
W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2} \sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} K(X^{(i)}, X^{(j)}) \\
\text{by iterating } t = 0, 1, \dots, \text{ as such:} \\
\lambda_i'(t+1) := \lambda_i(t) - \alpha \text{grad} W(\lambda) \\
\lambda_i''(t+1) := \mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)} = 0}(\lambda_i'(t+1)) \\
\lambda_i(t+1) := \Pi_{0 \leq \lambda_i \leq C}(\lambda_i''(t+1))
\end{gathered}
} \tag{86}
$$

where

$$
\boxed{
\begin{aligned}
&\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)} = 0}(\lambda_i'(t+1)) = \lambda_i'(t+1) - \frac{\sum_{i=1}^m y^{(i)} \lambda_i'(t+1)}{\sum_{i=1}^m (y^{(i)})^2} y^{(i)} \\
&\Pi_{0 \leq \lambda_i \leq C}(\lambda_i''(t+1)) = \begin{cases} C & \text{if } \lambda_i''(t+1) > C \\ \lambda_i''(t+1) & \text{if } 0 \leq \lambda_i''(t+1) \leq C \\ 0 & \text{if } \lambda_i'(t+1) < 0 \end{cases}
\end{aligned}
} \tag{87}
$$

The α parameter is the analogue to the *learning rate* of gradient descent and will need to be tuned.

12.6.2. *Computing b, the intercept, with a good algebra tip: multiply both sides by the denominator.* Bishop (2007) [13], on pp. 330 of Ch. 7, Sparse Kernel Machines, gave a very good (it resolved possible numerical instabilities) prescription on how to compute the intercept b, given λ, which would then give us the function that can make predictions $\widehat{y}$ on input data example $X^{(i)} \in X$. It's worth expounding upon here.

For any support vector (Bishop called it a support vector; what I think it's equivalent to is that we've trained on our training set $(X,y)^{\text{train}}$, and this is 1 of the training examples) $X^{(i)}$, $i = 1 \dots m$,

$$y^{(i)} f(X^{(i)}) = 1 \tag{88}$$

. Then using

$$\begin{aligned} f(x) &:= \sum_{i=1}^m y^{(i)} \lambda_i^* K(X^{(i)}, x) + b \\ \Longrightarrow y^{(i)} &\left(\sum_{j=1}^m y^{(j)} \lambda_j^* K(X^{(j)}, X^{(i)}) + b \right) = 1 \end{aligned} \tag{89}$$

Although we can solve this equation for b with algebra/arithmetic for our arbitrarily chosen support vector, it's numerically more stable to 1st. multiply through by $y^{(i)}$, using $(y^{(i)})^2 = 1$, and then averaging over all support vectors.

$$\begin{aligned} &\sum_{j=1}^m y^{(j)} \lambda_j^* K(X^{(j)}, X^{(i)}) + b = y^{(i)} \\ &\Longrightarrow b = \frac{1}{m} \left(\sum_{i=1}^m y^{(i)} - \sum_{i,j=1}^m y^{(j)} \lambda_j^* K(X^{(j)}, X^{(i)}) \right) \end{aligned} \tag{90}$$

12.6.3. *Prediction (with SVM).* Compute predictions with this formula: [14]

$$\begin{aligned} &\widehat{y}(X) = \sum_{i=1}^m y^{(i)} \lambda_i^* K(X^{(i)}, X) + b^* \\ &\widehat{y} : \mathbb{R}^d \to \{0, 1, \dots, K-1\} \quad \text{(with } K = 2 \text{ for binary classification)} \end{aligned} \tag{91}$$

12.7. **Constrained Gradient Descent (Implementation).** From Eqns. 86, 87, with the algorithm or iterative, computational steps that we should take mathematically formulated (clearly), I had sought out to implement these steps using `theano` and on the GPU, in the hopes of speeding up computation and developing a method that can scale with m input data examples.

Take a look at this double summation term in Eqn. 86 for $W(\lambda)$:

$$f_1(\lambda) := \frac{1}{2} \sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} K(X^{(i)}, X^{(j)}) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \lambda_i y^{(i)} K(X^{(i)}, X^{(j)}) \lambda_j y^{(j)} = (\mathbf{q}^{(i)})^T K(X^{(i)}, X^{(j)}) \mathbf{q}^{(j)} \tag{92}$$

Quadratic programming (denoted "QP" in computer science literature) is essentially trying to put the calculation of double sums, such as the above, into quadratic form, as in the very last equality. Techniques for efficient calculation, on the CPU, of this quadratic form, after reformulation of the original problem, make up QP.

The prevailing software package used for SVM, written in C/C++, that also underlies SVM module for sci-kit learn (`sklearn`) [16] is `libsvm` [17]. `libsvm` employs the method of Sequential Minimal Optimization (SMO) [18]. The main advantage of SMO is that only 2 λ_i's, Lagrange multipliers, are considered in the working set at each stage in time and the optimal solution is computed analytically at this point.

For instance, suppose we are considering 2 Lagrange multipliers λ_1 and λ_2. We first compute the optimal value changing λ_2 only. Then, using the inequality constraints $0 \leq \lambda_1, \lambda_2 \leq C$, and equality constraint $\sum_{i=1}^m \lambda_i y^{(i)} = 0$ (but adapted to the fact that we're only changing 2 λ_i's), we can analytically compute the other λ_1.

The (serial) computation in C/C++ of this analytical problem at this single step for SMO is fast. However, for the fitting for large data sets (large m), the fit time complexity is more than quadratic with the number of examples m, which makes it difficult to scale to datasets of more than a couple of 10000 examples ($m > 10000$) [3] [16].

Instead, I considered the idea behind *All-pairs N-body* algorithm of Nyland, Harris, and Prins (2007) in Ch. 31 of **GPU Gems 3** [15]. It was also explained in Udacity's CS344 with Owens and Luebke [3] [4].

Look again at Eq. 92, f_1, which clearly requires m^2 fetches, or reads, for $\lambda_i \lambda_j y^{(i)} y^{(j)} K(X^{(i)}, X^{(j)})$ term and m^2 computations, for each (i,j) pairs (and there are m^2 total pairs). It could also help to imagine a $m \times m$ matrix:

$$\begin{matrix} i=1, j=1 & i=1, j=2 & \dots & i=1, j=m \\ i=2, j=1 & i=2, j=2 & \dots & i=2, j=m \\ \vdots & \vdots & \ddots & \dots \\ i=m, j=1 & i=m, j=2 & \dots & i=m, j=m \end{matrix}$$

and observing that $\forall\, i = 1, 2, \dots m$, we're doing m computations for j and needing to fetch m values for each λ_j, y_j, $X^{(j)}$, and so on.

Consider this computation: for a given, single $i \in \{1, 2, \dots m\}$, define

$$f_{1i}(\lambda) := \frac{1}{2} \sum_{j=1}^m \lambda_j y^{(j)} K(X^{(i)}, X^{(j)}) \tag{93}$$

For this step, we'll only need to do m fetches for the $\lambda_j, y^{(j)}, X^{(j)}$ values, and $X^{(i)}$ value will be fetched once. As this is a summation over a potentially large vector (m can be big), this looks like a good case/candidate for the usage of parallel *reduce* algorithm. The work complexity of parallel reduce is $O(\log m)$ [3][5]. Theano has an implementation of reduce in `theano.reduce`.

Once all m f_{1i}'s are obtained, for $i = 1, 2, \dots m$, then parallel reduce can be used again (especially if m is large!). Also, empirically, I found that using `theano.reduce` again helped to circumvent the problem of the maximum recursion limit for Python [6], which is inherent with Python (cf. `import sys  sys.getrecursionlimit()`). In practice, above about 10000 recursions, the Python script fails with run-time errors.

Nevertheless, in this second (parallel) reduce step, we are doing

$$f_1 = \sum_{i=1}^m \lambda_i y^{(i)} f_{1i}(\lambda)$$

with m fetches of values for $\lambda_i, y^{(i)}$. The work complexity here for this reduce step is again $O(\log(m))$

Thus, we are doing, for 2 (parallel) reduces, $2m$ fetches (or reads), for each λ_i or $y^{(i)}$ or $X^{(i)}$.

The total work complexity is $O(2\log(m))$.

Likewise, for the computation of the intercept b in Eqn. 90, after minimizing $W(\lambda)$ by varying λ, I also employed parallel reduce via `theano.reduce` (but only once for the single sum) and for the prediction step for $\widehat{y}$ in Eq. 91

[3] sklearn.svm.SVC
[4] Quiz: All Pairs N-Body
[5] Step Complexity of Parallel Reduce - Intro to Parallel Programming, Udacity
[6] max recusion limit #689

12.7.1. *Code (theano/Python script), jupyter notebook accompanying code.* SVM is implemented as described above, in particular Eqns. 86, 87, in the Python class `SVM_parallel`. The default kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ is the radial basis function, which takes the form of a gaussian, is first implemented and I can implement other kernel functions easily, as a Python function object and Python class member, in the future.

Take note that for the (currently) implemented radial basis function, Python function (object) `rbf` in `SVM.py` of github:ernestyalumni/MLgrabbag/ML, what's formulated is this:

$$K(X^{(i)}, X^{(j)}) = \exp\left(- \frac{ \| X^{(i)} - X^{(j)} \|^2 }{2\sigma^2} \right) \tag{94}$$

with $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

Take a look at the $\sigma \in \mathbb{R}$ parameter in Eq. 94. σ is analogous to the variance of a Gaussian (normal) distribution. For other implementations, notably `libsvm` and sci-kit learn, they use this form of the radial basis function:

$$K(X^{(i)}, X^{(j)}) = \exp\left(-\gamma \| X^{(i)} - X^{(j)} \|^2 \right)$$

So γ parameter here is equivalent to σ:

$$\gamma = \frac{1}{2\sigma^2}$$

While this redefinition makes no change to the formulation above, this is something to note when when using `libsvm`, sci-kit learn, or `SVM.py` here when manually inputting the parameters to train models.

The theano/Python code follows directly from Eqns. 86, 87 and is in the `/ML` subfolder of the github repository `MLgrabbag` [7], in `SVM.py`. Wherever a summation is seen in the mathematical formulation, `theano.reduce` is used.

In the `SVM_parallel` Python class method `build_W`, I code a `theano.reduce` inside a `theano.reduce` and show it's possible to be done. This represents, both formally and the parallel reduction on the GPU, the double summation that we sought to compute in 86 for $W(\lambda)$.

The jupyter notebook `SVM_theano.ipynb` in the same github repository steps through how I developed and used `SVM_parallel`, training it on a number of sample datasets. Because of the interactivity of jupyter notebook, I invite others to explore and play with the notebook if further clarification on `SVM_parallel`, or how to use it, is needed [8]

12.8. **Immediate Results from training on sample datasets.**

12.8.1. *Real-World Examples.* I trained a SVM on 2 of the real-world data sets provided by Hsu, Chang, and Lin [19], one for astroparticles and another for vehicles, using, for hardware, a NVIDIA GeForce GTX 980 Ti. Checking the computational graph generated by theano (using `theano.function.maker.fgraph.toposort()`), `nvidia-smi -l 2` (monitoring real-time GPU usage), and the (usual, in Utilities) CPU resources System Monitor.

I will copy the results from Hsu, Chang, and Lin [19] for comparison. The accuracy measure is determined from the given *test* data, *not* on the training data (which is part of good machine learning and scientific practice).

Applications	# training data	# testing data	# features	# classes	$C =$	$\gamma =$	Accuracy by `libsvm`
Astroparticle[9]	3089	4000	4	2	2.0	2.0	96.9%
Vehicle[10]	1243	41	21	2	128.0	0.125	87.8%

Table 1: Sample Dataset Problem characteristics and accuracy performance [19].

Applications	$C =$	$\sigma =$	$\alpha =$	# iterations	Time to train (on GTX 980Ti)	Accuracy by `SVM_parallel`
Astroparticle	2.0	0.30	0.001	15	1h 7min 18s	96.1%
Vehicle	128.0	2.0	0.001	20	14min 54s	95.1%

Table 2: Results of training on Sample Datasets with `SVM_parallel`

The very last result testing on the test data for vehicles is promising for `SVM_parallel`. At this point, I would invite others to suggest sample and real-world datasets to train and test on, using `SVM_parallel`, as I also try to find other datasets, and add onto the jupyter notebook SVM theano.ipynb on github. It'd be interesting to vary the *number of training examples*, to find a dataset with more than 10000 ($m > 10000$) examples and see how `SVM_parallel` can scale with large data sets (indeed, for $m > 10000$, the SVM would have $m > 10000$ support vectors in the model), and vary the *number of features* (whether SVM does better with large or small number of features, relative to m).

[7] github:ernestyalumni/MLgrabbag

[8] github:ernestyalumni/MLgrabbag SVM theano.ipynb

12.9. **Conclusions/Summary/Dictionary between Math and Code.** I had reviewed the motivation and derivations for SVM.

What's novel is that, given the GPU(s), I implemented *constrained gradient descent* or *projected gradient descent*, for training models, instead of Quadratic Programming, that computes a quadratic form (to tackle the double summation in the dual formulation), through SMO, as used before (e.g. `libsvm`, sci-kit learn). Its (*constrained gradient descent* or its implementation here `SVM_parallel`) work complexity is $O(2 \log m)$, as opposed to $O(m^2)$. This was achieved by using theano's reduce, inside a reduce.

Its (i.e. `SVM_parallel`) promising to be scalable to large datasets ($m > 10000$). I seek to find large datasets to train and test on and are appropriate for binary classification, and invite others to make suggestions or play with the code and jupyter notebook itself.

I'll provide a 1-to-1 dictionary here between the mathematical formulation and the Python code. As a note on software engineering, object-oriented programming (OOP) and how to code classes, I had sought to identify (make isomorphisms) and design Python classes and function objects with 1-to-1 correspondence to the mathematical formulation. The hope is that it would allow other developers to rapidly make progress in improving upon the code or to rapidly understand its usage and apply it as they'd like to see fit.

$$\begin{aligned} & \text{we seek to minimize} & & \\ & W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2} \sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} K(X^{(i)}, X^{(j)}) & & \texttt{SVM\_parallel.build\_W} \\ & \text{by iterating } t = 0, 1, \dots, \text{ as such:} & & \texttt{SVM\_parallel.train\_mode\_full(max\_iters=250)} \\ & \lambda_i'(t+1) := \lambda_i(t) - \alpha \text{grad} W(\lambda) & & \\ & \lambda_i''(t+1) := \mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)} = 0}(\lambda_i'(t+1)) & & \texttt{SVM\_parallel.build\_update} \\ & \lambda_i(t+1) := \Pi_{0 \leq \lambda_i \leq C}(\lambda_i''(t+1)) & & \end{aligned}$$

where

$$\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)} = 0}(\lambda_i'(t+1)) = \lambda_i'(t+1) - \frac{\sum_{i=1}^m y^{(i)} \lambda_i'(t+1)}{\sum_{i=1}^m (y^{(i)})^2} y^{(i)}$$

$$\Longleftrightarrow$$

`updatelambda_mult=updatelambda_mult-T.dot(y,updatelambda_mult)/T.dot(y,y)*y` in `SVM.build_update`

$$\Pi_{0 \leq \lambda_i \leq C}(\lambda_i''(t+1)) = \begin{cases} C & \text{if } \lambda_i''(t+1) > C \\ \lambda_i''(t+1) & \text{if } 0 \leq \lambda_i''(t+1) \leq C \\ 0 & \text{if } \lambda_i'(t+1) < 0 \end{cases}$$

$\Longleftrightarrow$ `updatelambda_mult=T.switch(T.lt(C,updatelambda_mult),C,updatelambda_mult)` in `SVM.build_update`
`updatelambda_mult=T.switch(T.lt(updatelambda_mult,lower_bound),lower_bound,updatelambda_mult)` in `SVM.build_updat`

Finally, to tie it back into my original motivation, now that SVM is natively implemented in theano, it would be interesting to try to develop (and of course find appropriate datasets to train and test on) a DNN that will have as its "outer" or last layer to be a SVM. Since SVM is now part of the theano computational graph, optimization (the so-called "backpropagation" step) will be done automatically and simply with theano's `grad`, on all the parameters or "weights" of the entire model.

Part 5. Image Preprocessing; Image Classification

13. Links, Reading, Online Searches

- Day and night: an image classifier with scikit-learn, Giuseppe Cardone, GCardone

14. Deep Support Vector Machines (SVM)

14.1. **Right R-modules.** Consider, as a start, the total given (training) input data, consisting of $m \in \mathbb{Z}^+$ (training) examples, each example, say the ith example, being represented by a "feature" vector of d features, $X^{(i)} \in \mathbb{K}^d$, where $\mathbb{K}$ is a field or (categorical) classes, i.e. as examples of fields, the real numbers $\mathbb{R}$, or integers $\mathbb{Z}$, so that $\mathbb{K} = \mathbb{R}, \mathbb{Z}$ or $\mathbb{K} = \{0, 1, \dots K-1\}$, where K is the total number of classes that a feature could fall into. Note that for this case, the case of $\mathbb{K} = \{0, 1, \dots K-1\}$, for K classes, though labeled by integers, this set of integer labels is *not* equipped with ordered field properties (it is meaningless to say $0 < 1$, for example), nor the usual field (arithmetic) operations (you cannot add, subtract, multiply, or even take the modulus of these integers). How can we "feed into" our machine such (categorical) class data? Possibly, we should intuitively think of the Kronecker Delta function:

$$\delta_{iJ} = \begin{cases} 0 & \text{if } i \neq J \\ 1 & \text{if } i = J \end{cases}$$

for some (specific) class J, represented by an integer. So perhaps our machine can learn kronecker delta, or "signal"-like functions that will be "activated" if the integer value of a piece (feature) of data is exactly equal to J and 0 otherwise.

Onward, supposing $\mathbb{K}$ is a field, consider the total given input data of m examples:

$$\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m$$

One can arrange such input data into a $m \times d$ matrix. We want to do this, for one reason, to *take advantage of the parallelism afforded by GPU(s).* Thus we'd want to act upon the entire input data set $\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m$.

We'd also want to do *parallel reduce* in order to do a *summation*, $\sum_{i=1}^m$, over all (training) examples, to obtain a cost function(al) J.

For `theano`, parallel `reduce` and `scan` operations can only be done over the first dimension of a `theano` tensor. Thus, we write the total input data as such:

$$\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m \mapsto X_j^{(i)} \in \mathrm{Mat}_{\mathbb{K}}(m,d) \tag{95}$$

i.e. $X_j^{(i)}$ is a $m \times d$ matrix of $\mathbb{K}$ values, with each ith row corresponding to the $i = 1, 2, \dots m$th example, and jth column corresponding to the $j = 1, 2, \dots d$th feature (of the feature vector $X^{(i)} \in \mathbb{K}^d$).

Let's, further, make the following abstraction, in that the input data $\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m$ is really an element of a right R-module $\mathbf{X}$, in the category of right R-modules $\mathbf{Mod}_R$ with ring R, R not necessarily being commutative.

A reason for this abstraction is that if we allow the underlying ring R to be a field $\mathbb{K}$, (e.g. $\mathbb{K} = \mathbb{R}, \mathbb{Z}$), then the "usual" scalar multiplication by scalars is recovered. But we also need to equip $X \in \mathbf{Mod}_R$ with a *right action*, where ring R is *noncommutative*, namely

$$R = \mathrm{Mat}_{\mathbb{K}}(d,s) \cong L(\mathbb{K}^d, \mathbb{K}^s)$$

where $\mathrm{Mat}_{\mathbb{K}}(d,s)$ denotes the ring of all matrices over field $\mathbb{K}$ of matrix (size) dimensions $d \times s$ (it has d rows and s columns), $\cong$ is an isomorphism, $L(\mathbb{K}^d, \mathbb{K}^s)$ is the space of all (linear) maps from $\mathbb{K}^d$ to $\mathbb{K}^s$. If $\mathbb{K}$ is a field, this isomorphism exists.

Thus, for

$$\begin{aligned} &X \in \mathbf{X} \in \mathrm{Mod}_R \\ &R = \mathrm{Mat}_{\mathbb{K}}(d,s) \cong L(\mathbb{K}^d, \mathbb{K}^s) \end{aligned} \tag{96}$$

Let $\Theta \in R$. Θ is also known as the "parameters" or "weights" (and is denoted by w or W by others).

Consider, as a first (pedagogical) step, only a single example ($m = 1$). X is only a single feature vector, $X \in \mathbb{K}^d$. Then for basis $\{e_\mu\}_{\mu=1\dots d}$ of $\mathbb{K}^d$, corresponding dual basis $\{e^\mu\}_{\mu=1\dots d}$ (which is a basis for dual space $(\mathbb{K}^d)^*$), then

$$X\Theta = X^\mu e_\mu (\Theta_\nu{}^j e_j \otimes e^\nu) = \qquad \begin{aligned} &\mu, \nu = 1 \dots d \\ &j = 1 \dots s \end{aligned}$$
$$X^\mu \Theta_\nu{}^j e_j \otimes e^\nu (e_\mu) = X^\mu \Theta_\nu{}^j e_j \delta_\mu^\nu = X^\mu \Theta_\mu{}^j e_j$$

In this case where X is simply a vector, one could think of X as a "row matrix" and Θ is a matrix, acting on the right, in matrix multiplication.

Now suppose, in general, $X \in \mathbf{X} \in \mathbf{Mod}_R$, where X could be a $m \times d$ matrix, or higher-dimensional tensor. For a concrete example, say $\mathbf{X} = \mathrm{Mat}_{\mathbb{K}}(m,d)$. We not only have to equip this right R-module with the usual scalar multiplication, setting ring $R = \mathbb{K}$, but also the right action version of matrix multiplication, so that $R = \mathrm{Mat}_{\mathbb{K}}(d,s)$. This R is *non-commutative*, thus, necessitating the abstraction to right R-modules.

Indeed, for

$$\Theta \in L(\mathrm{Mat}_{\mathbb{K}}(m,d), \mathrm{Mat}_{\mathbb{K}}(m,s)) \cong (\mathrm{Mat}_{\mathbb{K}}(m,d))^* \otimes \mathrm{Mat}_{\mathbb{K}}(m,s) \cong \mathrm{Mat}_{\mathbb{K}}(d,s), \text{ and so}$$
$$X\Theta \in \mathrm{Mat}_{\mathbb{K}}(m,s)$$

Further

$$X\Theta \in \mathrm{Mat}_{\mathbb{K}}(m,s) \in \mathbf{Mod}_R$$

with ring R in this case being $R = \mathrm{Mat}_{\mathbb{K}}(s, s_2) \cong L(\mathbb{K}^s, \mathbb{K}^{s_2})$.

Since $X\Theta$ is an element in a R-module, it is an element in an (additive) abelian group. We can add the "intercept vector" b (in theano, it'd be the usual theano vector, but with its dimensions "broadcasted" for all m examples, i.e. for all m rows).

$$X\Theta + b \in \mathrm{Mat}_{\mathbb{K}}(m,s)$$

Considering these 2 operatons on X, the "matrix multiplication on the right" or right action Θ, and addition by b together, through *composition*, (Θ, b), we essentially have

$$X \in \mathbf{X} \in \mathbf{Mod}_{R_1} \xrightarrow{(\Theta,b)} X\Theta + b \in \mathbf{X_2} \in \mathbf{Mod}_{R_2} \tag{97}$$

where

$$\begin{aligned} R_1 &= \mathrm{Mat}_{\mathbb{K}}(d, s_1) \cong L(\mathbb{K}^d, \mathbb{K}^{s_1}) \\ R_2 &= \mathrm{Mat}_{\mathbb{K}}(s_1, s_2) \cong L(\mathbb{K}^{s_1}, \mathbb{K}^{s_2}) \end{aligned}$$

14.2. **Deep Neural Networks (DNN).** Consider a(n artificial) neural network (NN) of $L + 1 \in \mathbb{Z}^+$ "layers" representing $L + 1$ neurons, with each layer or neuron represented by a vector $a^{(l)} \in \mathbb{K}^{s_l}$, $s_l \in \mathbb{Z}^+$, $l = 1, 2, \dots L+1$ (or, counting from 0, $l = 0, 1, \dots L$). Again, $\mathbb{K}$ is either a field (e.g. $\mathbb{K} = \mathbb{R}, \mathbb{Z}$), or categorical classes (which is a subset of $\mathbb{Z}^+$, but without any field properties, or field operations).

Nevertheless, for this pedagogical example, currently, let $\mathbb{K} = \mathbb{R}$. Recall the usual (familar) NN, accepting that we do right action multiplications (matrices act on the right, vectors are represented by "row vectors", which, actually, correspond 1-to-1 with `numpy`/`theano` arrays, exactly). Recall also that the sigmoidal or (general) *activation* function, $\psi^{(l)}$, acts element-wise on a vector. An "axon" between 2 layers, such as layer l and layer $l + 1$, is mathematically computed as follows:

$$\begin{aligned} z^{(l+1)} &:= a^{(l)}\Theta^{(l)} + b^{(l)} \\ a^{(l+1)} &:= \psi^{(l)}(z^{(l)}) \end{aligned} \tag{98}$$

where $\Theta^{(l)}, b^{(l)}$ is as above, except there will be a total of L of these tuples ($l = 0, 1, 2, \dots L-1$).

With $(\Theta^{(l)}, b^{(l)})$ representing the (right action) linear transformation

$$(\Theta^{(l)}, b^{(l)})(a^{(l)}) = a^{(l)}\Theta^{(l)} + b^{(l)}$$

essentially,

$$a^{(l)} \xmapsto{(\Theta^{(l)},b^{(l)})} z^{(l+1)} \xmapsto{\psi^{(l)}\odot} a^{(l+1)}$$

$$(\mathbb{R}^{s_l})^m \xrightarrow{(\Theta^{(l)},b^{(l)})} (\mathbb{R}^{s_{l+1}})^m \xrightarrow{\psi^{(l)}\odot} (\mathbb{R}^{s_{l+1}})^m$$

$$\mathbf{Mod}_{R^{(l)}} \xrightarrow{(\Theta^{(l)},b^{(l)})} \mathbf{Mod}_{R^{(l+1)}} \xrightarrow{\psi^{(l)}\odot} \mathbf{Mod}_{\mathbb{R}^{(l+1)}} \quad (99)$$

Since we need to operate with the activation function $\psi^{(l)}\odot$ elementwise, we (implicitly) equip $\mathbf{Mod}_{R^{(l+1)}}$ with the Hadamard product. In fact, with composition, we can represent the lth axon as

$$a^{(l)} \xmapsto{\psi^{(l)}\odot(\Theta^{(l)},b^{(l)})} a^{(l+1)}$$

$$(\mathbb{R}^{s_l})^m \xrightarrow{\psi^{(l)}\odot(\Theta^{(l)},b^{(l)})} (\mathbb{R}^{s_{l+1}})^m$$

$$\mathbf{Mod}_{R^{(l)}} \xrightarrow{\psi^{(l)}\odot(\Theta^{(l)},b^{(l)})} \mathbf{Mod}_{\mathbb{R}^{(l+1)}} \quad (100)$$

The lesson is this: instead of thinking of layers, each separately, think of or focus on the relationship, the relations, between each layers, the axon, as one whole entity.

Suppose we "feed in" input data X into the first or 0th layer of this NN. This means that for $a^{(0)} \in \mathbb{R}^d$,

$$a^{(0)} = X^{(i)}$$

for the ith (training) example.

The "output" layer, layer L, should output the *predicted* value, given X. So

$$a^{(L)} \in \mathbb{R} \text{ or } \{0,1,\dots K-1\} \text{ or } [0,1]$$

for regression, or classification (so it takes on discrete values) or the probability likelihood of being in some class k, respectively.

The entire NN can mathematically expressed as follows:

$$X^{(i)} \xmapsto{\prod_{l=0}^{L-1}\psi^{(l)}\odot(\Theta^{(l)},b^{(l)})} a^{(L)}$$

$$(\mathbb{R}^d)^m \xrightarrow{\prod_{l=0}^{L-1}\psi^{(l)}\odot(\Theta^{(l)},b^{(l)})} (\mathbb{R}^{s_L} \text{ or } \mathbb{R} \text{ or } \{0,1,\dots K-1\} \text{ or } [0,1])^m \quad (101)$$

Part 6. Natural Language Processing (NLP)

15. TextRank

TextRank: Bringing Order into Texts, Rada Mihalcea and Paul Tarau

Mihalcea and Tarau [20].

Let $G=(V,E)$ be a directed graph with set of vertices V, set of edges E, $E\subset V\times V$.

$\forall$ given vertex V_i, let $\text{In}(V_i)\subset V \equiv$ set of vertices that point to it $\equiv$ "predecessors",
let $\text{Out}(V_i)\subset V\equiv$ set of vertices that vertex V_i points to "successors"

Let score $S: V\to\mathbb{R}$,

$$S(V_i) := (1-d) + d\sum_{j\in\text{In}(V_i)}\frac{1}{(\text{Out}(V_j))}S(V_j) \quad (102)$$

where $0\leq d\leq 1$.

Usually $d=0.85$.

Let $t\in\mathbb{Z}^+$. Let $t=0$. $\forall V_i\in V$, $S(V_i)(t=0)\in\mathbb{R}$, randomly assigned.

Weighted graphs.

$$WS(V_i) = (1-d) + d*\sum_{V_j\in\text{In}(V_i)}\frac{w_{ji}}{\sum_{V_k\in\text{Out}(V_j)}w_{jk}}WS(V_j) \quad (103)$$

$0\leq w_{ij}\leq 10$

15.1. Keyword Extraction.

2 vertices *connected* if corresponding lexical units co-occur within window of max. N words. $2\leq N\leq 10$.

Topic Ranking

Part 7. Notes

Restricted Boltzmann machine - estimate a probability distribution

Recurrent neural network - creates an internal state of the network which allows it to exhibit dynamic temporal behavior

How to choose the number of hidden layers and nodes in a feedforward neural network?

"In sum, for most problems, one could probably get decent performance (even without a second optimization step) by setting the hidden layer configuration using just two rules: (i) number of hidden layers equals one; and (ii) the number of neurons in that layer is the mean of the neurons in the input and output layers."

Part 8. Unsupervised Learning

16. PageRank

cf. Gleich (2015) [21]

Let $i=1,2\dots N$; $N=$ total number of "states". If "states" are represented as vertices $v_i\in V\in$ (Finite)Set, then i are some choice of labels for v_i's.

Let $P_{ij}:=$ probability of transitioning from j to i.

Clearly $P(i)\equiv$ probability of state i in next iteration $=P_{ij}P(j)$.

Let $\mathbf{v}:\{1,2,\dots N\}\to\mathbb{R}$, $0\leq\mathbf{v}(i)\leq 1$.

$\mathbf{v}(i)$ = (normalized) "probability" (likelihood) $i\in\{1,2,\dots N\}$ of *teleportation distribution* of randomly transitioning or teleporting for state i.

Note that this $\mathbf{v}$ is also represented as vector:

$$\mathbf{v}\xrightarrow{\text{vectorize}}\mathbf{v}\in\mathbb{R}^N$$

$\alpha\in\mathbb{R}^+$, $0\leq\alpha\leq 1$ is a tuned parameter.

Then Gleich considers this as the (vectorized) PageRank algorithm:

$$\alpha P_{ij}x_j + (1-\alpha)v_i = 0 \quad (104)$$

with $\mathbf{x}$ being what Gleich denotes as the PageRank vector.

So in Gleich's notation,

$$(\alpha\mathbf{P}+(1-\alpha)\mathbf{v}\mathbf{e}^T)\mathbf{x}=\mathbf{x} \text{ or } (\mathbf{1}-\alpha\mathbf{P})\mathbf{x}=(1-\alpha)\mathbf{v} \quad (105)$$

cf. Eq. (2.1) and (2.2) of Gleich (2015) [21], respectively.

Following Page, Brin, Motwani, and Winograd (1998) [22], and their notation now:

let u be a webpage. $u\equiv x\in X$, where X is a set (that could represent the set of vertices X). Let $Y=$ set of all edges of this graph.

Let $F_u :=$ set of pages that u points to, i.e.

$$F_u = \{x | x \in X, \begin{matrix} u = o(y) \\ x = t(y) \end{matrix} \text{ for some } y \in Y\} \quad (106)$$

Let $B_u :=$ set of pages that point to u, i.e.

$$B_u = \{x | x \in X, \begin{matrix} x = o(y) \\ u = t(y) \end{matrix} \text{ for some } y \in Y\} \quad (107)$$

$N_u = |F_u| =$ number of links from u. Let $c \in \mathbb{R}$ normalization factor (so total rank of all webpages constant).

Define simple ranking R,

$$\begin{aligned} & R : X \to \mathbb{R} \\ & R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} \end{aligned} \quad (108)$$

Consider

$$\sum_{u \in X} R(u) = c \sum_{u \in X} \sum_{v \in B_u} \frac{R(v)}{N_v} \Longrightarrow \frac{\sum_{u \in X} R(u)}{\sum_{u \in X} \sum_{v \in B_u} \frac{R(v)}{N_v}} = c$$
$$c = \frac{R(u)}{\sum_{v \in B_u} \frac{R(v)}{N_v}}$$

Consider $u \in X$. $\forall v \in B_u$, $N_v \geq 1$ (since connected graph **must** be connected, by definition).

$c < 1$ since there's the case of $F_u = \emptyset$ and "their weight is lost from system (cf. Sec. 2.7 of Page, Brin, Motwani, and Winograd (1998) [22])".

There's a problem if we have the case of circuits of size $n = 1$, $n = 2$. To overcome existence of circuits (rank sinks),

Definition 1. *Let $E(u) :=$ source of ranks.*
$R' \equiv$ PageRank,

$$\begin{aligned} & R' : X \to \mathbb{R} \\ & R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u) \end{aligned} \quad (109)$$

s.t. c maximized, $\|R'\|_1 = 1$ ($\|R'\|_1$ denotes the L_1 norm of R')

Generalizing this, for $A_{uv} \equiv$ "likelihood of transition **from** v **to** u, so that $A_{uv} = \frac{1}{N_v}$ for this special case.

$$R'(u) = c \sum_{v \in B_u} A_{uv} R'(v) + cE(u)$$

Both Page, Brin, Motwani, and Winograd and Gleich rewrites this as

$$R' = c(A + E\mathbf{e}^T)R'$$

with $\mathbf{e}$ being a column of 1's.

Indeed,

$$E_{i1}(\mathbf{e}^T)_{1k} R'_{k1} = E_{i1} \|R'\|_1 = E_{i1}$$

since $\|R'\|_1 = 1$ by (defined) normalization.

E is a user-defined parameter, possibly uniform $\forall u \in X$.

$$R'(u) - c \sum_{v \in B_u} A_{uv} R'(v) = cE(u) \quad (110)$$

16.0.1. *PageRank algorithm by Page, Brin, Motwani, and Winograd (1998)* [22]. cf. 12.6 *Computing Page Rank*, pp. 6 of Page, Brin, Motwani, and Winograd (1998) [22].

Let S be almost any vector over X; $\begin{matrix} S : X \to \mathbb{R}, \\ S(u) \geq 0 \end{matrix}$ (e.g. E). Then, for iterations $t = 0, 1, \dots \in \mathbb{Z}^+$, and so for

$$R : X \times \mathbb{Z}^+ \to \mathbb{R}$$

Then

$$R(u, 0) \equiv R_0(u) = S(u) \qquad (R_0 \leftarrow S)$$

$\forall t \in \mathbb{Z}^+$,

$$\begin{aligned} \forall u \in X, \, R(u, t+1) &= A_{uv} R(v, t) \\ d &= (\|R(u,t)\|_1 - \|R(u,t+1)\|_1) \\ R(u, t+1) &= R(u, t+1) + dE \\ \delta &= \|R(u, t+1) - R(u,t)\|_1 \end{aligned}$$

while $\delta > \epsilon$.

Compare this to the iteration by Gleich (2015) [21]:

$$\mathbf{x}^{(k+1)} = \alpha \mathbf{P} \mathbf{x}^{(k)} + (1 - \alpha)\mathbf{v} \quad (111)$$

where

$$\mathbf{x}^{(0)} = \mathbf{v} \text{ or } \mathbf{x}^{(0)} = 0$$

Indeed,

$$R(u) - R(u; t+1) = (\alpha A_{uv} R(v) + (1-\alpha)E) - (\alpha A_{uv} R(v,t) + (1-\alpha)E) = \alpha A_{uv}(R(v) - R(v,t)) \quad (112)$$

Indeed, $R(u,t)$ converges to $R(u)$.

16.0.2. *PageRank Vector Iteration Implementation* [23]. cf. Nov. 14. Dwarf No. 2 - Sparse Linear Algebra lecture by Dr. Bader [23], http://www5.in.tum.de/lehre/vorlesungen/hpc/WS16/sparseLA.pdf

Define

$$A_{ij} \in \text{Mat}_{\mathbb{R}}(N, N) \quad (113)$$

where $N =$ total number of webpages (vertices) $= |X|$, with A_{ij} defined as

$$A_{ij} = \begin{cases} 1 & \text{if } \exists \text{ edge } y \in Y \text{ **from** } j \text{ **to** } i \text{ (i.e. } \exists y \in Y \text{ s.t. } \begin{matrix} o(y) = x_j \\ t(y) = x_i \end{matrix}) \\ 0 & \text{otherwise} \end{cases} \quad (114)$$

with

$$N_j = \sum_{i=1}^N A_{ij} = \text{ total number of links from } j\text{th webpage (vertex) } = |F_j| \quad (115)$$

and so *define* B_{ij}

$$\begin{aligned} & B_{ij} \in \text{Mat}_{\mathbb{R}}(N, N) \\ & B_{ij} := \frac{1}{N_j} A_{ij} \end{aligned} \quad (116)$$

Compute PageRank vector via vector iteration

$$\begin{aligned} \mathbf{x}^{(m)} &= \alpha B \mathbf{x}^{(m-1)} + (1-\alpha)\frac{1}{N}\mathbf{e} \text{i.e.} \\ R(u; t+1) &= \alpha B_{uv} R(v; t) + (1-\alpha)E(u) \end{aligned} \quad (117)$$

B is a sparse matrix, so use SpMV, i.e. sparse matrix vector multiplication.

Now, we should talk about *Sparse Linear Algebra*.

17. Data Structures for Sparse Matrices

cf. Part II: Data Structures for Sparse Matrices in Nov. 14. Dwarf No. 2 - Sparse Linear Algebra lecture by Dr. Bader [23], http://www5.in.tum.de/lehre/vorlesungen/hpc/WS16/sparseLA.pdf.

17.1. **Coordinate Scheme (aka Triple Scheme).** We want

$$A_{ij} \in \text{Mat}_{\mathbb{F}}(N_i, N_j) \mapsto (a_{ij}, i, j)\mathbb{F} \times \mathbb{Z}^+ \times \mathbb{Z}^+ \quad (118)$$

Let $K \in \mathbb{Z}^+ =$ total number of **nonzero** entries in A_{ij}.

The coordinate scheme is implemented either as an array of struct (of size K), i.e.

$$\begin{aligned} & M : \mathbb{Z}^+ \to \mathbb{F} \times \mathbb{Z}^+ \times \mathbb{Z}^+ \\ & M(I) = (A(I), i(I), j(I)) \end{aligned} \quad (119)$$

or struct of array, i.e.

$$\begin{aligned} A_{ij} \in \text{Mat}_{\mathbb{F}}(N_i, N_j) \mapsto M \in (\mathbb{Z}^+ \to \mathbb{F}) \times (\mathbb{Z}^+ \times \mathbb{Z}^+)^2 \text{ with} \\ M_1(I) = A(I), M_2(I) = i(I), M_3(I) = j(I) \end{aligned} \quad (120)$$

used, e.g. in Matlab, or as input format (format to input in). Note also that it's possibly not sorted, i.e. $i = i(I), j = j(I)$ may not follow any lexicographic order, depending on I.

17.2. **Compressed Row Storage (CRS).** 2 arrays of size K with a_{ij} and j, i.e. consider $a : \{1, 2, \dots K\} \to \mathbb{F}$.

$$A_{ij} \in \text{Mat}_{\mathbb{F}}(N_i, N_j) \mapsto \begin{aligned} & a : \{1, 2, \dots K\} \to \mathbb{F}, \\ & j : \{1, 2, \dots K\} \to \mathbb{Z}^+, \\ & IA : \{0, 1, \dots N_i\} \to \mathbb{Z}^+ \in \text{Hom}(\mathbb{Z}^+, \mathbb{F}), \text{Hom}(\mathbb{Z}^+, \mathbb{Z}^+)^2 \end{aligned} \quad (121)$$

Note that we are using left-to-right, top-to-bottom "row-major" order, which is amenable to so-called (thread) warp coalescing for memory usage/optimization in CUDA C/C++.

So, to reiterate, we have

$a(k) = a_{ij}$ for some surjective $(i, j)] \mapsto k$

$j(k) \in \mathbb{Z}^+$

$$IA(i) = \begin{cases} 0 & \text{if } i = 0 \\ & IA(i-1) + (\text{number of nonzero elements of } (i-1)\text{th row in original matrix}) \end{cases}$$

Let's take a look at $IA : \{0, 1, \dots N_i\} \to \mathbb{Z}^+$ in greater detail. Consider these simple cases:

$IA(0) = 0$

$IA(i) = IA(i-1)+$(number of nonzero elements of $(i-1)$th row in original matrix, with $i = 0, 1, \dots N_i - 1$, in this particular case)

References

[1] Trevor Hastie, Robert Tibshirani, Jerome Friedman. **The Elements of Statistical Learning: Data Mining, Inference, and Prediction**, Second Edition (Springer Series in Statistics) 2nd ed. 2009. Corr. 7th printing 2013 Edition. ISBN-13: 978-0387848570. https://web.stanford.edu/~hastie/local.ftp/Springer/OLD/ESLII_print4.pdf

[2] Jared Culbertson, Kirk Sturtz. *Bayesian machine learning via category theory.* arXiv:1312.1445 [math.CT]

[3] John Owens. David Luebki. *Intro to Parallel Programming. CS344.* **Udacity** http://arxiv.org/abs/1312.1445 Also, https://github.com/udacity/cs344

[4] CS229 Stanford University. http://cs229.stanford.edu/materials.html

[5] Richard Fitzpatrick. "Computational Physics." http://farside.ph.utexas.edu/teaching/329/329.pdf

[6] LISA lab, University of Montreal. Deep Learning Tutorial. http://deeplearning.net/tutorial/deeplearning.pdf September 2015.

[7] Kurt Hornik. "Approximation Capabilities of Muitilayer Feedforward Networks." **Neural Networks**, Vol. 4, pp. 251-257. 1991

[8] Kurt Hornik. Maxwell Stinchcombe and Halbert White. "Multilayer Feedforward Networks are Universal Approximators." **Neural Networks**, Vol. 2, pp. 359-366, 1989.

[9] Thomas Nowak. "Implementation and Evaluation of a Support Vector Machine on an 8-bit Microcontroller." Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich Institut für Technische Informatik Fakultät für Informatik Technische Universität Wien. Juli 2008. https://www.lri.fr/~nowak/misc/bakk.pdf

[10] J. Shawe-Taylor and N. Cristianini, Support Vector Machines and other kernel-based learning methods, Cambridge University Press (2000).

[11] Edwin K. P. Chong and Stanislaw H. Zak. **An Introduction to Optimization**. 4th Edition. Wiley. (January 14, 2013). ISBN-13: 978-1118279014

[12] Lecture by Harikrishna Narasimhan. *Optimization Tutorial 3: Projected Gradient Descent, Duality.* **E0 270 Machine Learning**. Jan 23, 2015. http://drona.csa.iisc.ernet.in/~e0270/Jan-2015/Tutorials/lecture-notes-3.pdf

[13] Christopher M. Bishop. **Pattern Recognition and Machine Learning** (Information Science and Statistics). Springer (October 1, 2007). ISBN-13: 978-0387310732

[14] Bertrand Clarke, Ernest Fokoue, Hao Helen Zhang. **Principles and Theory for Data Mining and Machine Learning** (Springer Series in Statistics) Springer; 2009 edition (July 30, 2009). ISBN-13: 978-0387981345

[15] Hubert Nguyen. **GPU Gems 3**. Addison-Wesley Professional (August 12, 2007). ISBN-13: 978-0321515261. Also made available in its entirety online at https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_pref01.html

[16] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, **JMLR 12**, pp. 2825-2830, 2011.

[17] C.-C. Chang and C.-J. Lin. *LIBSVM : a library for support vector machines.* **ACM Transactions on Intelligent Systems and Technology**, 2:27:1–27:27, 2011.

[18] J. Platt. *Fast training of support vector machines using sequential minimal optimization.* In A. Smola B. Schölkopf, C. Burges, editor, **Advances in Kernel Methods: Support Vector Learning**. MIT Press, Cambridge, MA, 1998.

[19] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification.* http://www.ee.columbia.edu/~sfchang/course/spr/papers/svm-practical-guide.pdf

[20] Rada Mihalcea and Paul Tarau. "TextRank: Bringing Order into Texts." https://web.eecs.umich.edu/~mihalcea/papers/mihalcea.emnlp04.pdf

[21] David F. Gleich. *PageRank Beyond the Web.* **SIAM Review**. Vol. 57, No. 3, pp. 321-363 2015

[22] Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry. "The PageRank Citation Ranking: Bringing Order to the Web." Technical Report. Stanford InfoLab. 1998. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[23] Prof. Dr. Michael Bader, with Alexander Pöppl, Valeriy Khakhutskyy (Tutorials). High Performance Computing (HPC) - Algorithms and Applications - Winter 16/17. Informatics V - Scientific Computing. Technical University of Munich (TUM) https://www5.in.tum.de/wiki/index.php/HPC_-_Algorithms_and_Applications_-_Winter_16