Big ML and GPUs
Parallel computing with map-reduce:
- Stream-and-sort in parallel
- Enormous datasets
- Tasks are i/o bound
- Many unreliable processors
  - which are basically commodity PCs
- Parallelize with mapreduce
  - loosely coupled, heavy-weight jobs
  - communicate via network/disk
- Don’t iterate (typically)
Streaming SGD:

- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly
Streaming SGD:

- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly

- You can speed it up by making the tasks in the stream bigger and doing them in parallel
- A GPU is a good way of doing that
Parallel computing with map-reduce:
- Stream-and-sort in parallel
- Enormous datasets
- Tasks are i/o bound
- Many unreliable processors
  - which are basically commodity PCs
- Parallelize with mapreduce
  - loosely coupled, heavy-weight jobs
  - communicate via network/disk
- Don’t iterate (typically)

Parallel ML computing with GPUs:
- Iterative streaming ML in parallel
- Big-but-not-too-big datasets
- Tasks are compute bound
- Many fast-but-simple processors
- Replace streaming operations with medium-sized computations that can be done in parallel
- Usually iterate many times
WHAT ARE GPUs?
What is a GPU?

A **graphics processing unit (GPU)** is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. [wikipedia]

The term GPU was popularized by [Nvidia](https://www.nvidia.com/) in 1999, who marketed the [GeForce 256](https://www.geforce.com/) as "the world's first …Graphics Processing Unit." It was presented as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines".[3]
GPUs are faster than CPUs: maximum FLOPS/clock cycle

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
GPUs compute faster because they are parallel.
GPUs compute faster because they are parallel.
GPUs have **more cores** than CPUs
- hi-end CPU: 32
- hi-end GPU: 2000

GPUs have **less memory** than CPUs
- hi-end CPU: 256 Gb
- hi-end GPU: 8 Gb

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
GPUs also **read memory faster** (memory bandwidth: bits/sec) because they are parallel.

[Theoretical Peak Memory Bandwidth Comparison graph]

[Link to the original source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/]
Summary of GPUs vs CPUs

- less total memory
- more cores and more parallelism

- Multicore CPUs are mostly multiple-instruction multiple-data (MIMD)
- GPUs are mostly single-instruction multiple-data (SIMD)
Summary of GPUs vs CPU clusters

- way way less total memory, and memory is RAM not disk
- SIMD vs MIMD
- Expensive and reliable vs cheap and fault-tolerant
Art, Science and GPU's
Adam Savage & Jamie Hyneman
Explain Parallel Processing

https://www.youtube.com/watch?v=-P28LKWzrI
HOW DO YOU USE A GPU?

Using GPUs for ML

• Programming parallel machines is complicated
• To use the parallelism of a GPU in an ML algorithm we almost always use matrix algebra as an abstraction layer – i.e. vectorize

```python
def logisticRegression(....):
    ....
    for X,Y in ....:
        Z = W.matrix_multiply(X)
        P = logistic(Z)
        W = W + learning_rate * (P - Y) * X
    ....
```
Using GPUs for ML

def logisticRegression(....):
    ....
    for X, Y in ....:
        Z = W.matrix_multiply(X)
        P = logistic(Z)
        W = W + learning_rate * (P - Y) * X
    ....
Using GPUs for ML

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);
    // Alloc space for host copies and setup values
    in  = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);
    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMalloc((void**)&d_out, size);
    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);
    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
SOME EXAMPLE CODE
Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the **host** (the CPU)

- NVIDIA compiler (nvcc)

- We can also write code for the **device** (the GPU)

Output:

```
$ nvcc hello_world.cu
$a.out
Hello World!
```

24
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}

- Two new things here...
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`
Hello World! with Device COde

mykernel<<<1,1>>>();

• Triple angle brackets mark a call from host code to device code
  – Also called a “kernel launch”
  – We’ll get to the parameters (1,1) soon
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- `mykernel()` does nothing at all in this example .... so let’s fix that.

Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```
Addition on the Device

• A simple kernel to add two integers (coming up: adding two arrays)

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

• As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host
Addition on the Device

• Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

• `add()` runs on the device, so `a`, `b` and `c` must point to device memory

• We need to allocate memory on the GPU
Memory Management

• Host and device memory are separate entities
  – *Device* pointers point to GPU memory
    May be passed to/from host code
    May *not* be dereferenced in host code
  – *Host* pointers point to CPU memory
    May be passed to/from device code
    May *not* be dereferenced in device code

• Simple CUDA API for handling device memory
  – `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  – Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- Let’s take a look at `main()`...
Addition on the Device: `main()`

```c
int main(void) {
    int a, b, c;  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
}```
We’re getting ready to do this...

1. Copy input data from CPU memory to GPU memory is **coming up next**!
Addition on the Device: `main()`

```c
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```
With `add()` running in parallel we can do vector addition

Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

By using `blockIdx.x` to index into the array, each block handles a different index
Vector Addition on the Device

__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

• On the device, each block can execute in parallel:

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
</tr>
</thead>
</table>
Vector Addition on the Device: \texttt{add()} \\

- Returning to our parallelized \texttt{add()} kernel

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

- Let’s take a look at \texttt{main()}...
Vector Addition on the Device: \texttt{main()}

```c
#define N 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int*)malloc(size); random_ints(a, N);
    b = (int*)malloc(size); random_ints(b, N);
    c = (int*)malloc(size);
```
Vector Addition on the Device: `main()`

```c
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

...a little more magic...
Coordinating Host & Device

• Kernel launches are asynchronous
  – Control returns to the CPU immediately
• CPU needs to synchronize before consuming the results

`cudaMemcpy()`  Blocks the CPU until the copy is complete
Copy begins when all preceding CUDA calls have completed

`cudaMemcpyAsync()`  Asynchronous, does not block the CPU

`cudaDeviceSynchronize()`  Blocks the CPU until all preceding CUDA calls have completed
MORE DETAILS ON GPU PROGRAMMING

Summary of GPUs vs CPUs

- less total memory
- more cores and more parallelism

- Multicore CPUs are *mostly* multiple-instruction multiple-data (MIMD)
- GPUs are *mostly* single-instruction multiple-data (SIMD)
Blocks, Grids, Threads, Warps

• Recall blocks are the things that work in parallel, and blocks are arranged in grids

• \( c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]; \)

• That would be SIMD (single instruction multiple data)

• It’s actually more complicated than that....
SIMD

Program
\[ color_{out} = f(color_{in}) \]

Fetch | Decode
--- | ---

CPU0

<table>
<thead>
<tr>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
</table>

7,0
6,0
5,0
4,0
SIMD: Zooming in

Program
\[ color_{out} = f(color_{in}) \]

Fetch Decode

CPU0

Execute Memory Writeback

Execute Memory Writeback

Execute Memory Writeback

Execute Memory Writeback

Register File
SIMT: Single Instruction Multiple Threads
SIMT: Single Instruction Multiple Threads

A thread can access its own block id and also thread id. Blocks and threads are in a grid, which is 2D or 3D (there’s a .x and a .y part).
Comparison
What’s in a GPU?

Threads (SIMT, synchronous threads) are grouped into cores (which are decoupled, like a MIMD machine)
IDs and Dimensions

– A kernel is launched as a grid of blocks of threads
  • blockIdx and threadIdx are 3D
  • We showed only one dimension (x)

• Built-in variables:
  – threadIdx
  – blockIdx
  – blockDim
  – gridDim
Thread and block parallelism

tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
bx = cuda.blockIdx.x
by = cuda.blockIdx.y
bw = cuda.blockDim.x
bh = cuda.blockDim.y
x = tx + bx * bw
y = ty + by * bh
array[x, y] = something(x, y)
How Do You REALLY Use a GPU?
Recap: Using GPUs for ML

• Programming parallel machines is complicated
• To use the parallelism of a GPU in an ML algorithm we almost always use **matrix algebra** as an abstraction layer – i.e. *vectorize*

```
def logisticRegression(....):
    ....
    for X,Y in ....:
        Z = W.matrix_multiply(X)
        P = logistic(Z)
        W = W + learning_rate * (P - Y) * X
    ....
```
def logisticRegression(....):
    ....
    for X,Y in ....:
        Z = W.matrix_multiply(X)
        P = logistic(Z)
        W = W + learning_rate * (P - Y) * X
    ....
Recap: Using GPUs for ML

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int* in, int* out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int* x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int* in, *out;
    fill_ints(in, N + 2*RADIUS);
    fill_ints(out, N + 2*RADIUS);
    cudaMalloc((void**)&d_in, size);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```
How Do You REALLY Use a GPU?

Example: Vectorizing Logistic Regression with Stochastic Gradient Descent
Vectorizing ML Algorithms

• We want to specify an algorithm so that
  – It’s concise
  – It’s easy to verify correctness
  – It reveals to a compiler what can be done in parallel on a GPU

  – Matrix-vector computations!
Vectorizing logistic regression

- Conditional probability of an example:

\[
P(Y = y | X = x, w) = \begin{cases} 
\frac{1}{1 + e^{-x \cdot w}} & \text{if } y = 1 \\
\frac{1}{1 - \frac{1}{1 + e^{-x \cdot w}}} & \text{if } y = 0
\end{cases}
\]

- Conditional log likelihood of an example:

\[
\log P(Y = y | X = x, w) = \begin{cases} 
\log p & \text{if } y = 1 \\
\log(1 - p) & \text{if } y = 0
\end{cases}
\]

\[
p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}
\]

\[
\frac{\partial}{\partial w^j} \log P(Y = y | X = x, w) = (y - p) x^j
\]
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each $\mathbf{x}$ in the minibatch, compute

$$p \equiv \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_{j} x^j w^j)}$$

• For each feature $j$: update $w^j$ using

$$\frac{\partial}{\partial w^j} \log P(Y = y|X = \mathbf{x}, \mathbf{w}) = (y - p) x^j$$
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each $x$ in the minibatch $X_{batch}$, compute

$$p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}$$

Note: $x \cdot w$ can be partially computed in parallel...
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each \( \mathbf{x} \) in the minibatch \( X_{batch} \), compute

\[
p = \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x_j w_j)}
\]

Note: \( X_{batch} \cdot \mathbf{w} \) can be computed in parallel...

\[
X_{batch} = \begin{bmatrix}
  x_1^1 & \cdots & x_J^1 \\
  \vdots & \ddots & \vdots \\
  x_B^1 & \cdots & x_J^B
\end{bmatrix}
\]
Vectorizing logistic regression

- Computation we’d like to parallelize:
  
  - For each \( \mathbf{x} \) in the minibatch \( X_{\text{batch}} \), compute

    \[
    p \equiv \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}
    \]

    \[
    X_{\text{batch}} \mathbf{w} = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w^1 \\ \vdots \\ w^J \end{bmatrix} = \begin{bmatrix} \mathbf{w} \cdot \mathbf{x}_1 \\ \vdots \\ \mathbf{w} \cdot \mathbf{x}_B \end{bmatrix}
    \]

    We’re most of the way now...
Streaming SGD:

- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly
Streaming SGD:

- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly
- You can speed it up by making the tasks in the stream bigger and doing them in parallel
- A GPU is a good way of doing that
Vectorizing logistic regression

• Computation we’d like to parallelize:

$\text{For each } x \text{ in the minibatch } X_{\text{batch}}, \text{ compute}$

$$p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}$$

$$X_{\text{batch}} w = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w^1 \\ \vdots \\ w^J \end{bmatrix} = \begin{bmatrix} w \cdot x_1 \\ \vdots \\ w \cdot x_B \end{bmatrix}$$

We’re most of the way now...
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each $x$ in the minibatch $X_{batch}$, compute

$$p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}$$

$$\begin{bmatrix}
w \cdot x_1 \\
\vdots \\
w \cdot x_B
\end{bmatrix} + 1$$

We can overload the operator for the vector/matrix class so that adding 1 will ”work”
Vectorizing logistic regression

```
class Matrix(object):
    ...
    def __add__(self, addend):
        result = Matrix()
        ....
        return result
```

```
m = Matrix(...)  # m
z = m + 1         # z = m + 1
```

\[ \begin{bmatrix}
  w \cdot x_1 \\
  \vdots \\
  w \cdot x_B 
\end{bmatrix} + 1 \]

We can overload the operator for the vector/matrix class so that adding \( c \) will create a copy with \( c \) added component-wise.
Vectorizing logistic regression

class Matrix(object):
    ...
    def __add__(self, addend):
        result = Matrix()
        ....
        return result

m = Matrix(...)  
z = (m+1).exp() 

exp(\begin{bmatrix}
w \cdot x_1 \\
\vdots \\
w \cdot x_B 
\end{bmatrix} + 1) ....and also define operations like exp() to work component-wise...
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each \( \mathbf{x} \) in the minibatch \( X_{\text{batch}} \), compute

\[
p \equiv \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x_j w_j)}
\]

\[
X_b = \text{Matrix}(...)
\]
\[
w = \text{Matrix}(...)
\]

```python
def logistic(X):
    return (X+1).exp().reciprocal()

p = logistic(Xb.dot(w))  # B rows, 1 column
```
Vectorizing logistic regression

- Computation we’d like to parallelize:
  - For each \( x \) in the minibatch, compute

\[
p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}
\]

\[
\frac{\partial}{\partial w^j} \log P(Y = y|X = x, w) = (y - p)x^j
\]

\[
X_{\text{batch}} w \quad = \quad \begin{bmatrix} w \cdot x_1 \\ \vdots \\ w \cdot x_B \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} p_1 \\ \vdots \\ p_B \end{bmatrix} \quad \begin{bmatrix} y_1 - p_1 \\ \vdots \\ y_B - p_B \end{bmatrix}
\]
Vectorizing logistic regression

- Computation we’d like to parallelize:
  - For each \( x \) in the minibatch, compute

\[
p = \frac{1}{1 + e^{-x \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}
\]

\[
\frac{\partial}{\partial w^j} \log P(Y = y | X = \mathbf{x}, \mathbf{w}) = (y - p)x^j
\]

```python
def logistic(X): return (X+1).exp().reciprocal()
p = logistic(Xb.dot(w))  # B rows, 1 column
grad = Xb.dot(y - p).rowsum() * 1/B
w = w + grad*rate
```
EXAMPLE: NUMPY PACKAGE
```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```
Binary to softmax logistic regression

\[ p \equiv \frac{1}{1 + e^{-x \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)} \]

\[
X_{batch} \mathbf{w} = \begin{bmatrix}
\begin{array}{ccc}
  x_1^1 & \cdots & x_1^J \\
  \vdots & \ddots & \vdots \\
  x_B^1 & \cdots & x_B^J
\end{array}
\end{bmatrix}
\begin{bmatrix}
  \mathbf{w}^1 \\
  \vdots \\
  \mathbf{w}^J
\end{bmatrix}
= \begin{bmatrix}
  \mathbf{w} \cdot x_1 \\
  \vdots \\
  \mathbf{w} \cdot x_B
\end{bmatrix}
\]
Binary to **softmax** logistic regression

\[ p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)} \]

\[ p^y \equiv \frac{\exp(x \cdot w^y)}{\sum_y \exp(x \cdot w^y')} \]

\[
XW = \begin{bmatrix}
  x^1_1 & \cdots & x^J_1 \\
  \vdots & \ddots & \vdots \\
  x^1_B & \cdots & x^J_B
\end{bmatrix}
\begin{bmatrix}
  w^1_1 \\
  \vdots \\
  w^1_J
\end{bmatrix}
= \begin{bmatrix}
  w \cdot x_1 \\
  \vdots \\
  w \cdot x_B
\end{bmatrix}
\]

\[
XW = \begin{bmatrix}
  x^1_1 & \cdots & x^J_1 \\
  \vdots & \ddots & \vdots \\
  x^1_B & \cdots & x^J_B
\end{bmatrix}
\begin{bmatrix}
  w^{y_1}_1 & \cdots & w^{y_K}_1 \\
  \vdots & \ddots & \vdots \\
  w^{y_1}_J & \cdots & w^{y_K}_J
\end{bmatrix}
= \begin{bmatrix}
  w^{y_1} \cdot x_1 & \cdots & w^{y_K} \cdot x_1 \\
  \vdots & \ddots & \vdots \\
  w^{y_1} \cdot x_B & \cdots & w^{y_K} \cdot x_B
\end{bmatrix}
\]
Matrix multiply, \(\text{then exponentiate component-wise}\)

prob will have B rows and K columns, and each row will sum to 1

\[
p^y \equiv \frac{\exp(x \cdot w^y)}{\sum_y, \exp(x \cdot w^{y'})}
\]

\[
XW = \begin{bmatrix}
  x_1^1 & \cdots & x_J^1 \\
  \vdots & \ddots & \vdots \\
  x_1^B & \cdots & x_J^B
\end{bmatrix}
\begin{bmatrix}
  w_1^{y1} & \cdots & w_1^{yK} \\
  \vdots & \ddots & \vdots \\
  w_J^{y1} & \cdots & w_J^{yK}
\end{bmatrix}
= \begin{bmatrix}
  w_1^{y1} \cdot x_1 & \cdots & w_1^{yK} \cdot x_1 \\
  \vdots & \ddots & \vdots \\
  w_J^{y1} \cdot x_B & \cdots & w_J^{yK} \cdot x_B
\end{bmatrix}
\]

... that this line will work correctly even though \('a' and 'a_sum' have different shapes\)
```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.

def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```
Error on each example $x$ in batch and each class $y$

Python bug: should be $x.T$ (transpose)

The gradient step!

$$
x.T \, dy = \begin{bmatrix} x_1^1 & \cdots & x_B^1 \\
\vdots & \ddots & \vdots \\
x_1^J & \cdots & x_B^J \\
\end{bmatrix} \begin{bmatrix} dy_{x1}^{y1} & \cdots & dy_{xB}^{yK} \\
\vdots & \ddots & \vdots \\
dy_{x1}^{y1} & \cdots & dy_{xB}^{yK} \\
\end{bmatrix}
$$

$$
\frac{\partial}{\partial w^j} \log P(Y = y|X = x, w) = (y - p)x^j
$$

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_clusters

# Predict the class using multinomial logistic regression
def predict(w, x):
    preds = np.dot(x, w)
    return np.argmax(preds)

# Using gradient descent to get the correct weights.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if loss < 0.0:
            break
        dw = np.dot(data.T, dy) / num_samples
        # Update parameters; fixed learning rate
        w -= 0.1 * dw

# Initialize training data.
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```
So this will run in parallel on a GPU?


No: we’re not there yet….
So this will run in parallel on a GPU? Not yet....


Option 1: switch from numpy (old package) to cupy (new GPU-oriented package)

Option 2: switch to a package that will compile to a GPU and also compute the gradients for you (like Theano, Tensorflow, Torch, ...)

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial Logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```