CS11-747 Neural Networks for NLP

Efficiency Tricks for Neural Nets

Graham Neubig

Carnegie Mellon University
Language Technologies Institute

Site
https://phontron.com/class/nn4nlp2020/

Glamorous Life of an AI Scientist

Perception

Photo Credit: Antoine Miech @ Twitter

Reality

```
neubig@itachi:~$ python nn-lm.py
[dynet] random seed: 3454201866
[dynet] allocating memory: 512MB
[dynet] memory allocation done.
--finished 500 sentences
--finished 1000 sentences
--finished 1500 sentences
--finished 2000 sentences
--finished 2500 sentences
--finished 3000 sentences
--finished 3500 sentences
--finished 4000 sentences
```

Waiting….

Why are Neural Networks Slow and What Can we Do?

- GPUs love big operations, but hate doing lots of them
    - → **Reduce the number of operations** through optimized implementations or batching
- Our networks are big, our data sets are big
    - → **Use parallelism** to process many data at once
- Big operations, especially for softmaxes over large vocabularies
    - → **Approximate operations or use GPUs**

GPU Training Tricks

GPUs vs. CPUs

CPU, like a motorcycle

Quick to start, top speed not shabby

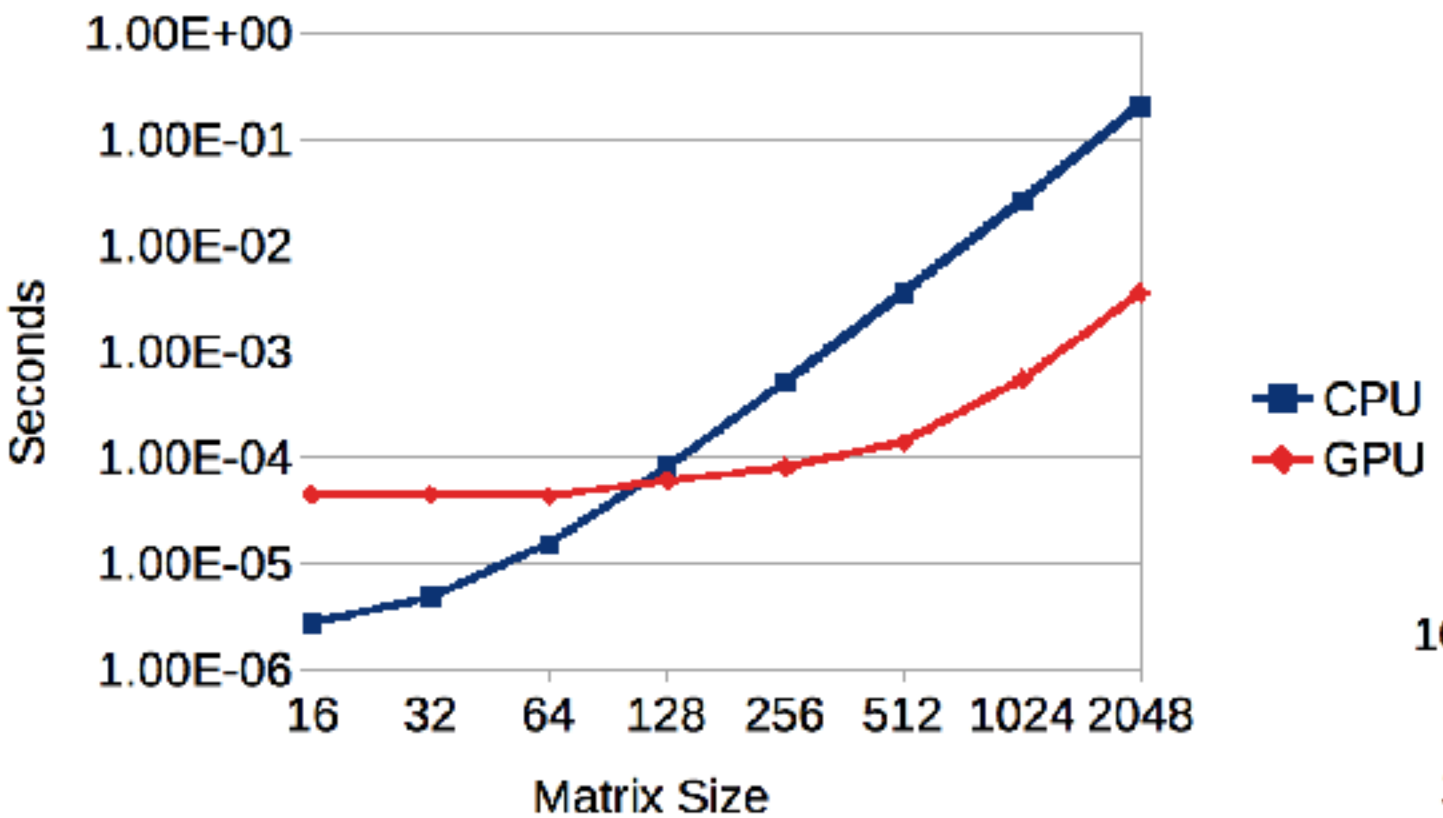
GPU, like an airplane

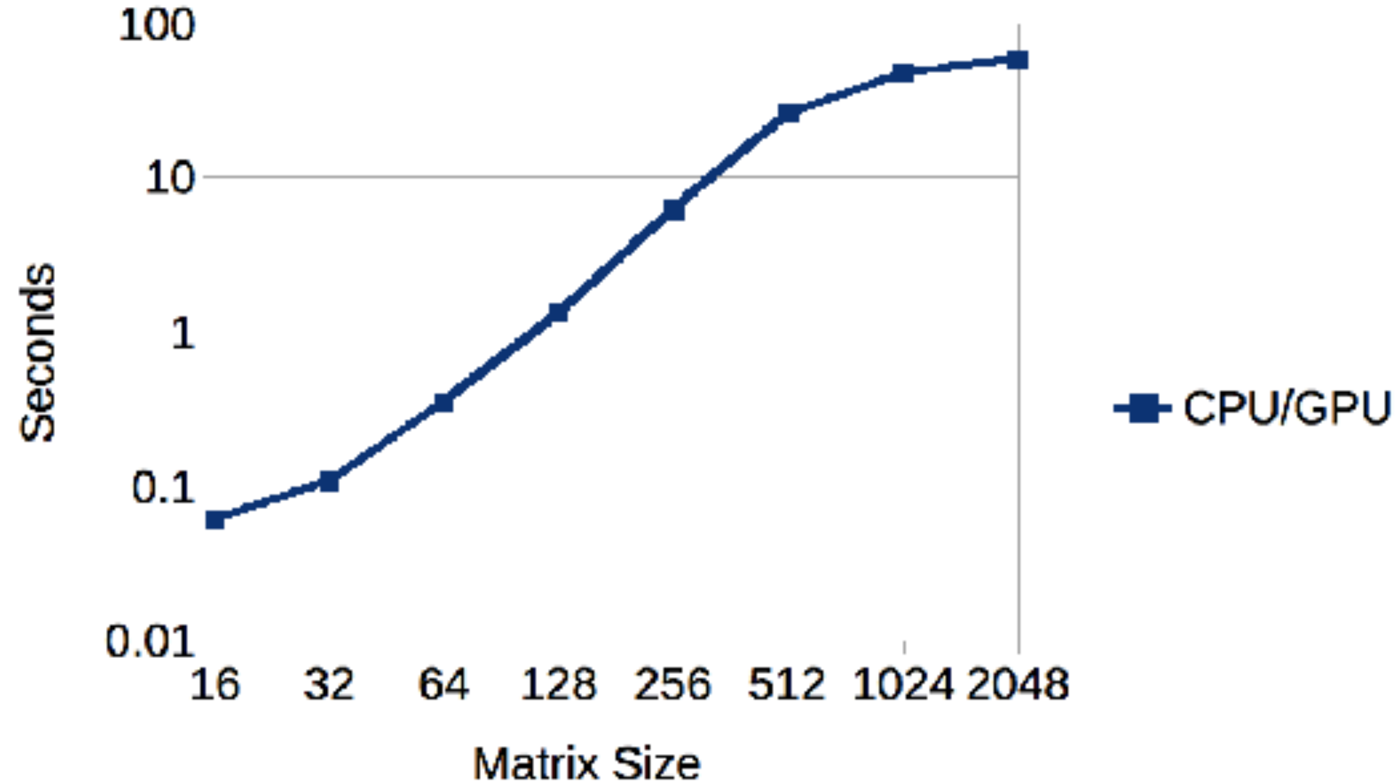
Takes forever to get off the ground, but super-fast once flying

Image Credit: Wikipedia

A Simple Example

- How long does a matrix-matrix multiply take?





Practically

- Use **CPU for prototyping**, it's often and you can run many more experiments
- For **many applications, CPU is just as fast** or faster than GPU: NLP analysis tasks with small or complicated data/networks
- You see **big gains on GPU when** you have:
  - Very big networks (or softmaxes with no approximation)
  - Do mini-batching
  - Optimize things properly

Speed Trick 1: Don't Repeat Operations

- Something that you can do once at the beginning of the sentence, don't do it for every word!

Bad

```
for x in words_in_sentence:
  vals.append( W * c + x )
```

Good

```
W_c = W * c
for x in words_in_sentence:
  vals.append( W_c + x )
```

Speed Trick 2: Reduce # of Operations

- e.g. can you combine multiple matrix-vector multiplies into a single matrix-matrix multiply? Do so!

Bad

```
for x in words_in_sentence:
  vals.append( W * x )
val = dy.concatenate(vals)
```

Good

```
X = dy.concatenate_cols(words_in_sentence)
val = W * X
```

Speed Trick 3: Reduce CPU-GPU Data Movement

- Try to **avoid memory moves** between CPU and GPU.
- When you do move memory, try to do it as early as possible (GPU operations are asynchronous)

Bad

```
for x in words_in_sentence:
  # input data for x
  # do processing
```

Good

```
# input data for whole sentence
for x in words_in_sentence:
  # do processing
```

What About Memory?

- Many GPUs only have up to 12GB, so **memory is a major issue**
- **Minimize unnecessary operations**, especially ones over big pieces of data
- If absolutely necessary, **use multiple GPUs** (but try to minimize memory movement)

Let's Try It!

`slow-impl.py`

Parallelism in Computation Graphs

Three Types of Parallelism

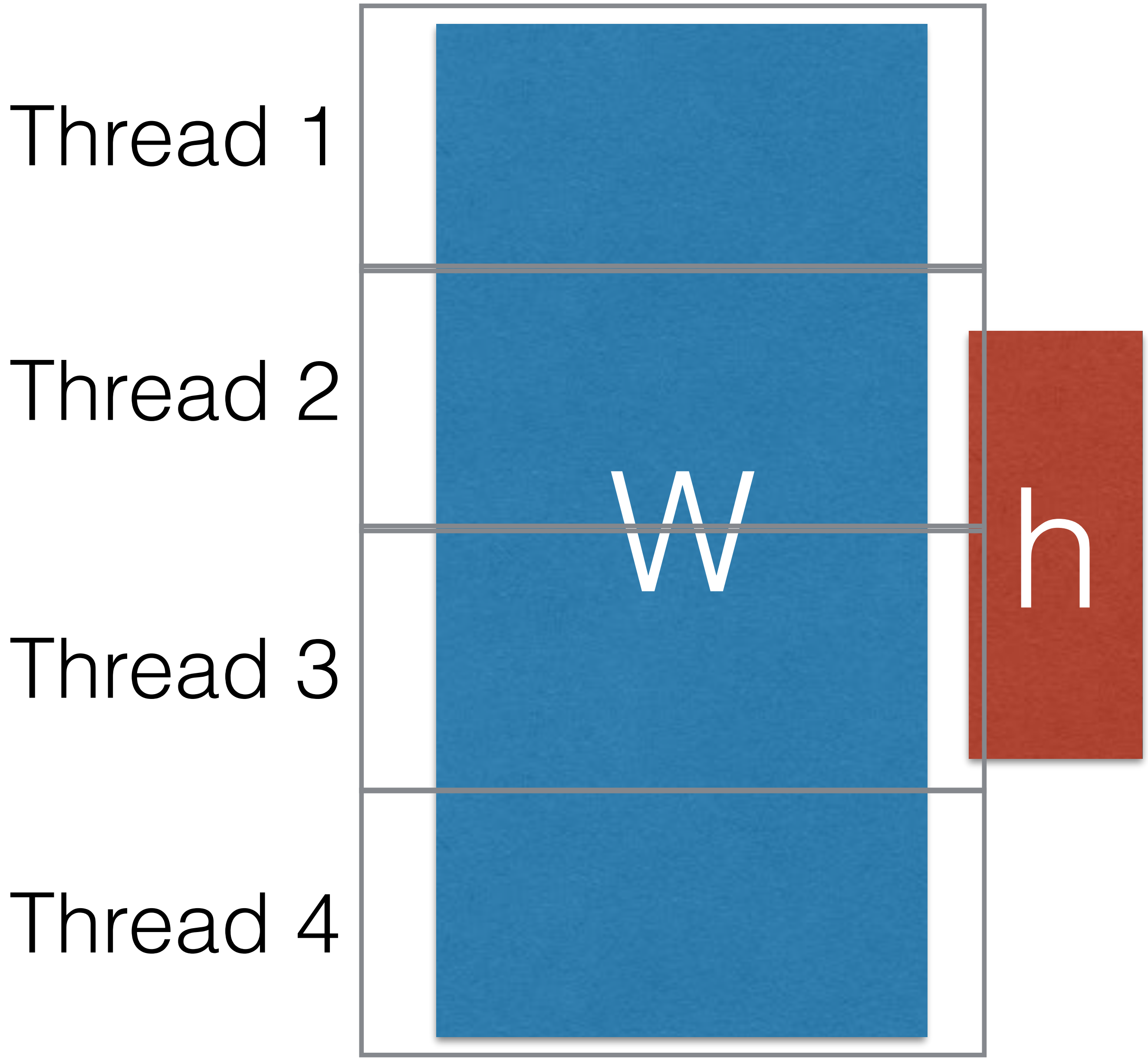
- Within-operation parallelism
- Operation-wise parallelism

Model parallelism

- Example-wise parallelism

Data parallelism

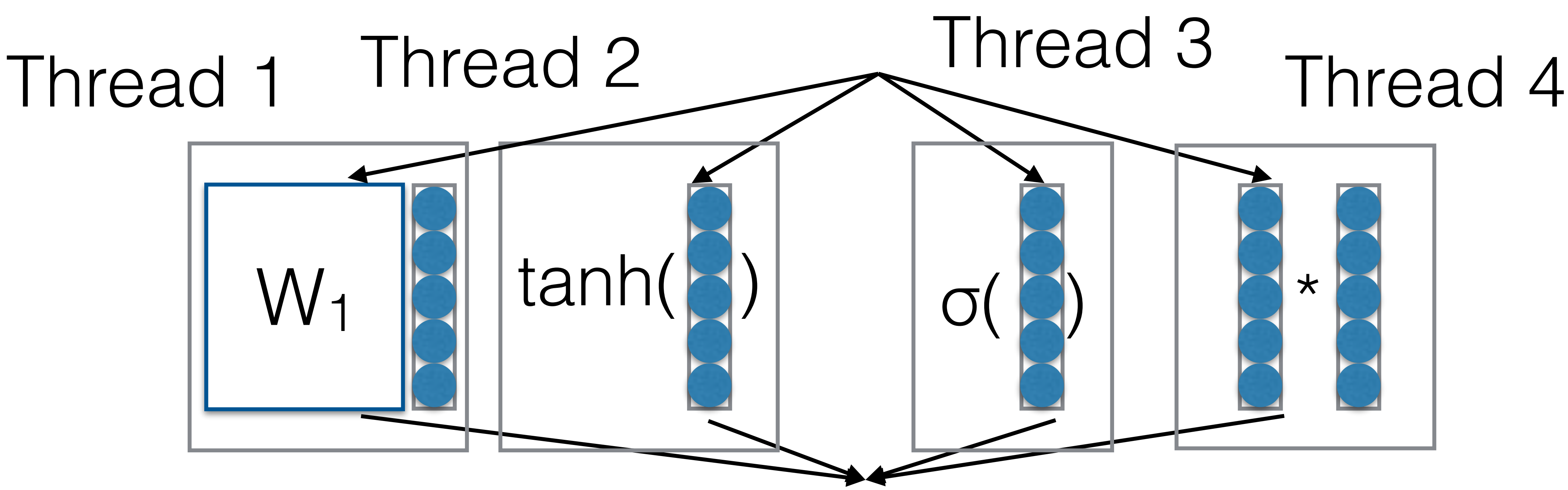
Within-operation Parallelism



- GPUs (and TPUs) excel at this!
- Libraries like MKL implement this on CPU, but gains less striking.
- Thread management overhead is counter-productive when operations small.

Operation-wise Parallelism

- Split each operation into a different thread, or different GPU device



- **Difficulty:** How do we minimize dependencies and memory movement?

Example-wise Parallelism

- Process each training example in a different thread or machine

this is an example	Thread 1
this is another example	Thread 2
this is the best example	Thread 3
no, i’m the best example	Thread 4

- **Difficulty:** How do we implement, accumulate gradients, keep parameters fresh across machines?

Implementing Data Parallelism

- Many modern libraries make data parallelism relatively easy, e.g. PyTorch DistributedDataParallel

```
def demo_basic(rank, world_size):
    setup(rank, world_size)

    # setup devices for this process, rank 1 uses GPUs [0, 1, 2, 3] and
    # rank 2 uses GPUs [4, 5, 6, 7].
    n = torch.cuda.device_count() // world_size
    device_ids = list(range(rank * n, (rank + 1) * n))

    # create model and move it to device_ids[0]
    model = ToyModel().to(device_ids[0])
    # output_device defaults to device_ids[0]
    ddp_model = DDP(model, device_ids=device_ids)

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(device_ids[0])
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()
```

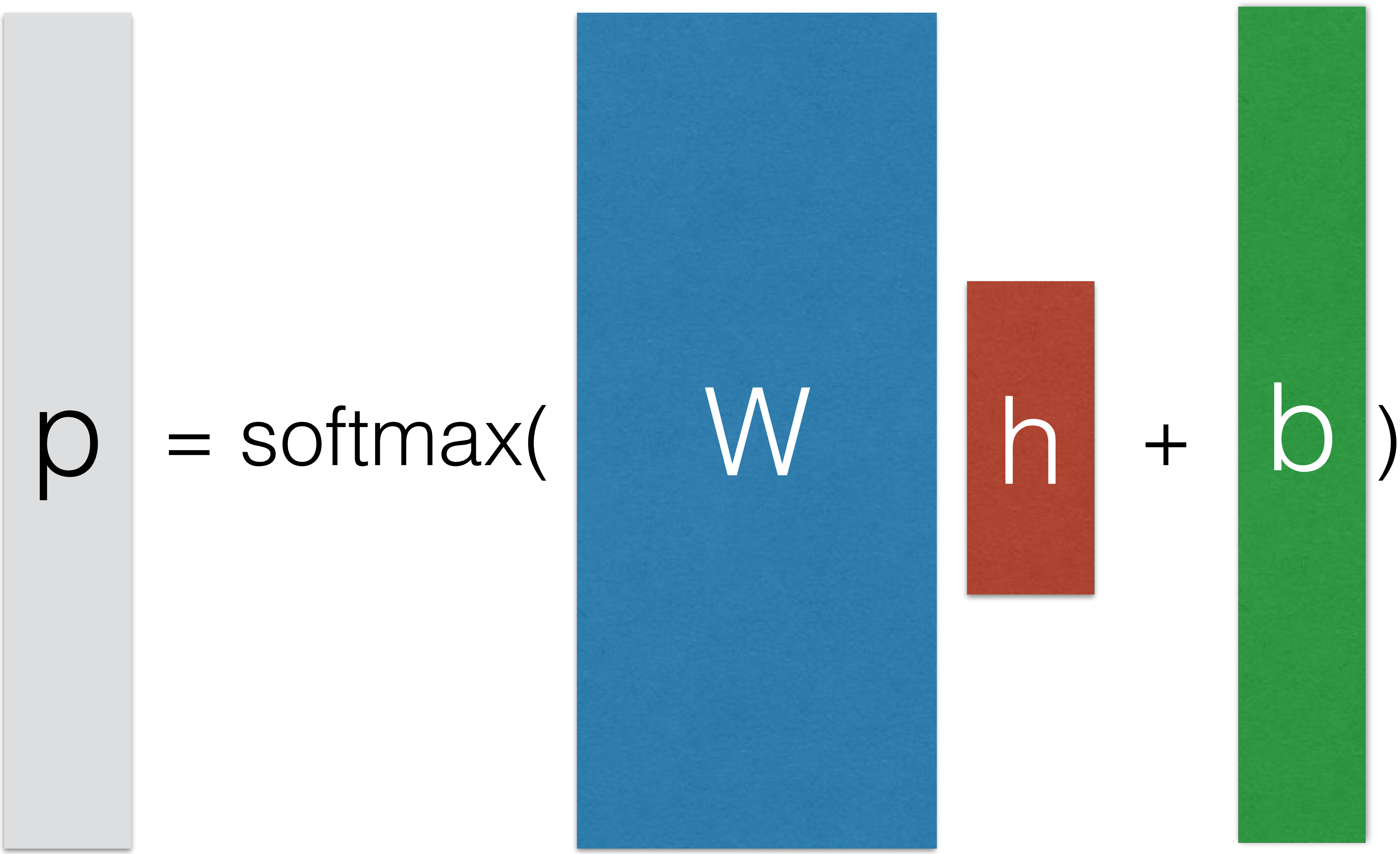
```
def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)
```

Negative Sampling

Computation Across Large Vocabularies

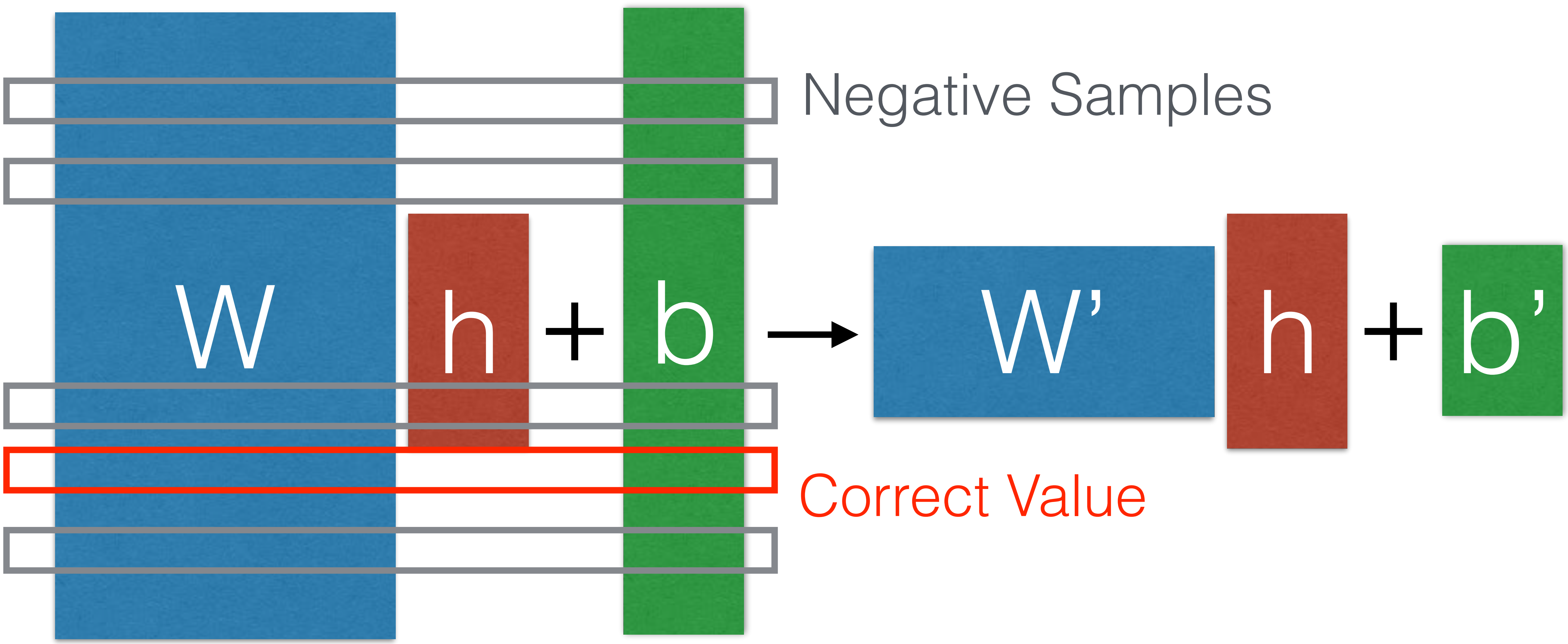
- All the words in the English language (e.g. language modeling)
- All of the examples in a database (e.g. search or retrieval)
- Too many to calculate each every time!

A Visual Example of the Softmax



Negative Sampling

- Calculate the denominator over a subset



- Sample negative examples according to distribution *q*

Softmax

- Convert scores into probabilities by taking the exponent and normalizing (softmax)

$$P(x_i \mid \boldsymbol{h}_i) = \frac{e^{s(x_i|\boldsymbol{h}_i)}}{\boxed{\sum_{\tilde{x}_i} e^{s(\tilde{x}_i|\boldsymbol{h}_i)}}}$$

This is expensive, would like to approximate

$$Z(\boldsymbol{h}_i) = \sum_{\tilde{x}_i} e^{s(\tilde{x}_i|\boldsymbol{h}_i)}$$

Importance Sampling
(Bengio and Senecal 2003)

- Sampling is a way to approximate a distribution we cannot calculate exactly
- **Basic idea:** sample from arbitrary distribution Q (uniform/unigram), then re-weight with e^s/Q to approximate denominator

$$Z(\boldsymbol{h}_i) \approx \frac{1}{N} \sum_{\tilde{x}_i \sim Q(\cdot|\boldsymbol{h}_i)} \frac{e^{s(\tilde{x}_i|\boldsymbol{h}_i)}}{Q(\tilde{x}_i \mid \boldsymbol{h}_i)}$$

- This is a biased estimator (esp. when N is small)

Noise Contrastive Estimation

(Mnih & Teh 2012)

- **Basic idea:** Try to guess whether it is a true sample or one of N random noise samples. Prob. of true:

$$P(d=1 \mid x_i, \boldsymbol{h}_i) = \frac{P(x_i \mid \boldsymbol{h}_i)}{P(x_i \mid \boldsymbol{h}_i) + N * Q(x_i \mid \boldsymbol{h}_i)}$$

- Optimize the probability of guessing correctly:

$$\mathbb{E}_P[\log P(d=1 \mid x_i, \boldsymbol{h}_i)] + N * \mathbb{E}_Q[\log P(d=0 \mid x_i, \boldsymbol{h}_i)]$$

- During training, approx. with unnormalized prob.

$$\tilde{P}(x_i \mid \boldsymbol{h}_i) = P(x_i \mid \boldsymbol{h}_i)/e^{c_{\boldsymbol{h}_i}} \quad \text{(set } c_{\boldsymbol{h}_i} = 0\text{)}$$

Simple Negative Sampling
(Mikolov 2012)

- Used in `word2vec`
- Basically, sample one positive *k* negative examples, calculate the log probabilities

$$P(d = 1 \mid x_i, \boldsymbol{h}_i) = \frac{P(x_i \mid \boldsymbol{h}_i)}{P(x_i \mid \boldsymbol{h}_i) + 1}$$

- Similar to NCE, but biased when k != |V| or Q is not uniform

Mini-batch Based Negative Sampling

- Creating and arranging memory on the is expensive, especially on the GPU
- **Simple solution:** select the same negative samples for each minibatch
- (See Zoph et al. 2015 for details)

Let's Try it Out!

`wordemb-negative-sampling.py`

More Efficient Predictors

Structure-based Approximations

- We can also change the structure of the softmax to be more efficiently calculable
  - **Class-based softmax**
  - **Hierarchical softmax**
  - **Binary codes**
  - **Embedding Prediction**

Class-based Softmax

(Goodman 2001)

- Assign each word to a class
- Predict class first, then word given class

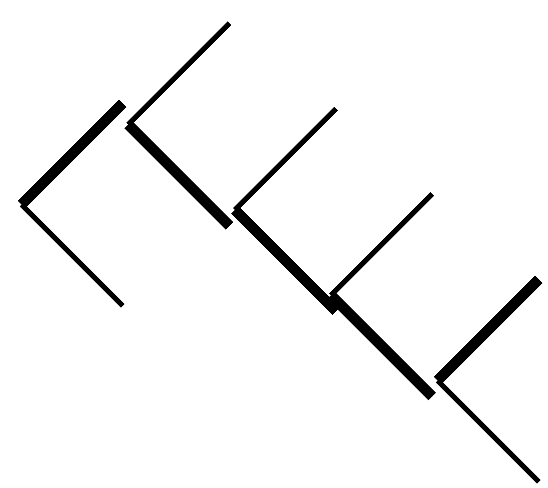
$$P(c|h) = \mathrm{softmax}(W_c h + b_c)$$

$$P(x|c,h) = \mathrm{softmax}(W_x h + b_x)$$

- **Quiz:** What is the computational complexity?

Hierarchical Softmax
(Morin and Bengio 2005)

- Create a tree-structure where we make one decision at every node



0 1 1 1 0 → word 14

- **Quiz:** What is the computational complexity?

Binary Code Prediction
(Dietterich and Bakiri 1995, Oda et al. 2017)

- Choose all bits in a single prediction

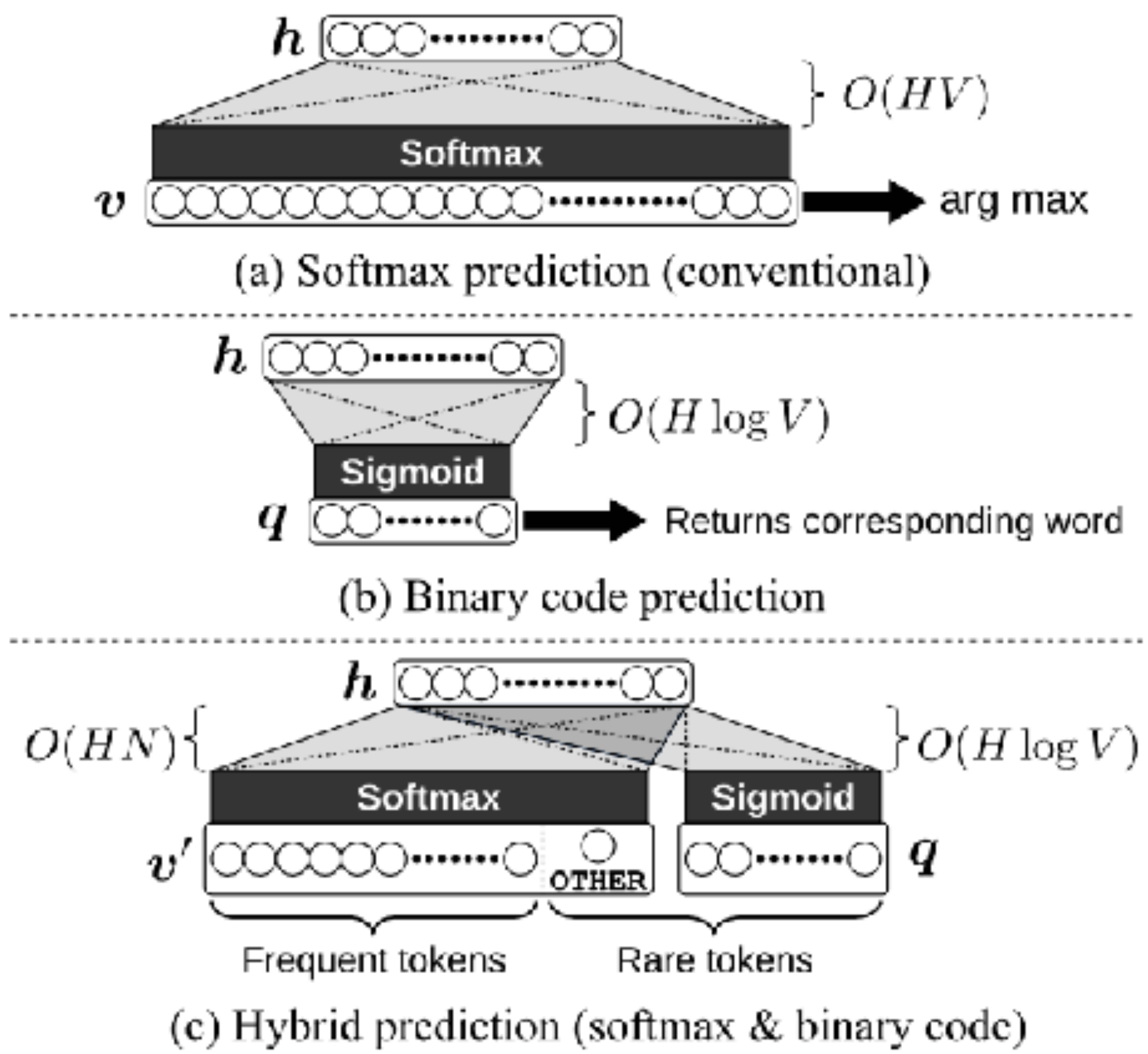
$$\sigma(W_c \, h + b_c) = \begin{matrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{matrix} \Big\downarrow$$

word 14

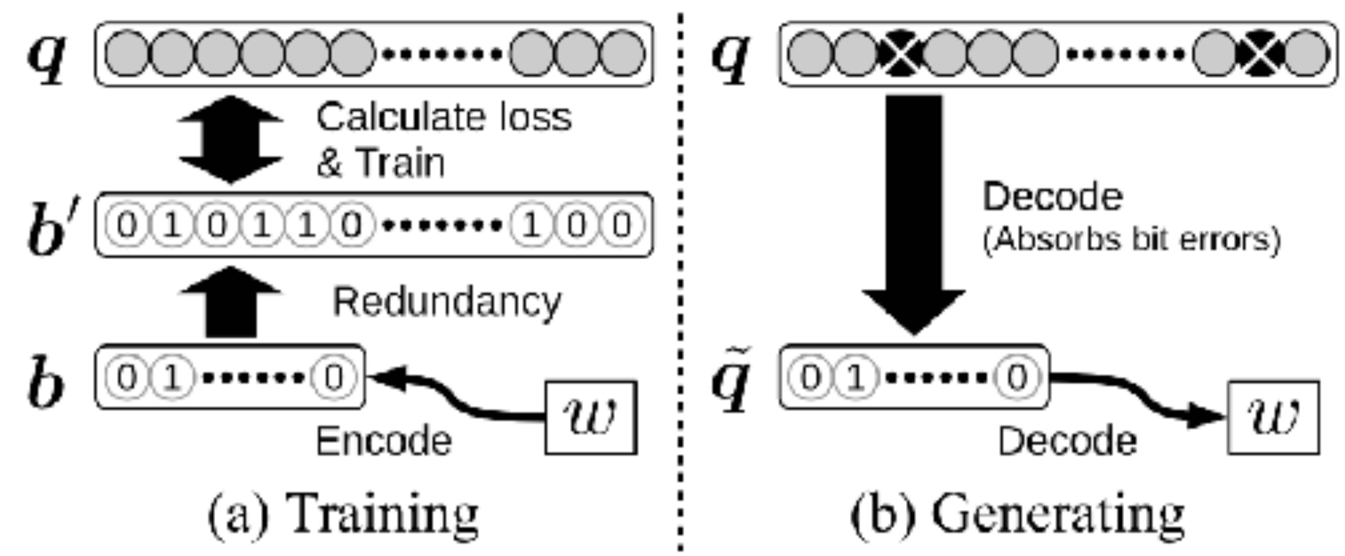
- Simpler to implement and fast on GPU

Two Improvement to Binary Code Prediction

Hybrid Model



Error Correcting Codes

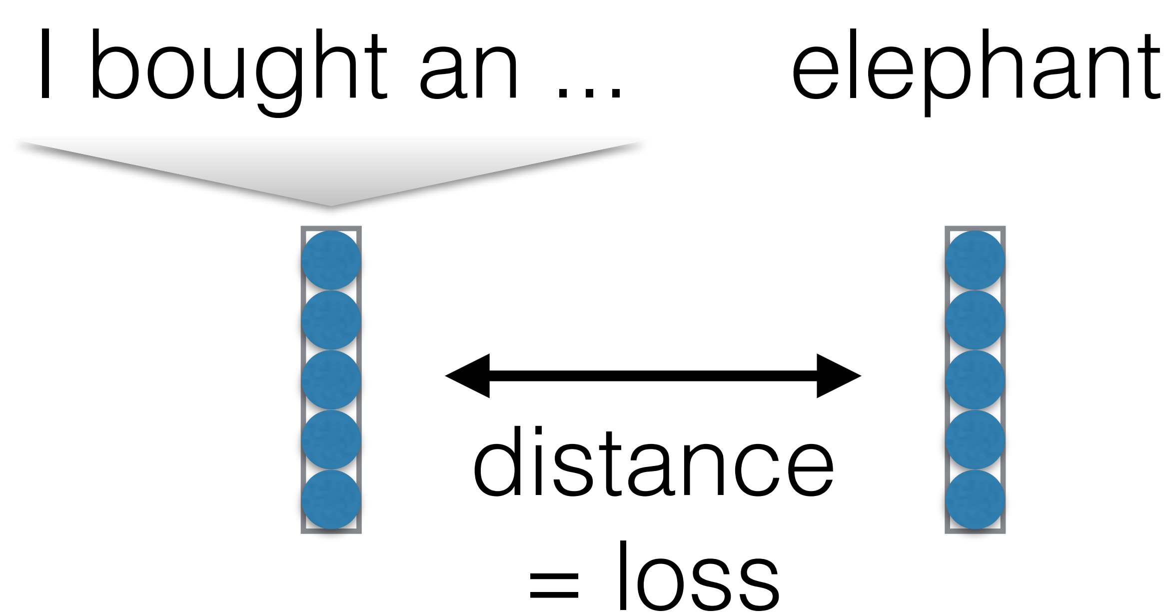


Let’s Try it Out!

`wordemb-binary-code.py`

Embedding Prediction
(Kumar and Tsvetkov 2019)

- Directly predict embeddings of outputs themselves



- **Specifically:** Von-Mises Fisher distribution loss, make embeddings close on the unit ball

Questions?