Principles of Machine Learning Systems

4: Training – Under the Hood

Nicholas D. Lane
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies
DL model recap
\[ L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left( -y_i \log(\tilde{y}_i) + (1 - y_i) \log(1 - \tilde{y}_i) \right) + \|\theta\|_2 \]
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies

http://mlsys.cst.cam.ac.uk/teach
Gradient descent – update rule

A common sense tells us to repetitively take a small step in the most downhill direction until a bottom is reached.

That is precisely what we do! The most downhill direction is defined as the negative of the loss function’s gradient.
Gradient descent – update rule

More formally

at the layer $l$ any parameter $i$ is updated at step $s$ like so:

- **Weights:**
  
  \[ w_{l,i}^{s+1} = w_{l,i}^s - r \cdot g_{l,i}^s \]

- **Biases:**
  
  \[ b_{l,i}^{s+1} = b_{l,i}^s - r \cdot g_{l,i}^s \]

Generally, the same holds for any parameter $\theta$:

\[ \theta_{l,i}^{s+1} = \theta_{l,i}^s - r \cdot g_{l,i}^s \]

- **Step size**
- **Gradient**
GD, SGD & mini-batches

1. **Gradient Descent (GD):**  
   *Exact Gradients*
   - Use all training set examples to compute the true gradient.
   - Very slow and very memory intensive.

2. **Stochastic GD (SGD):**  
   *Approximate Gradients*
   - Use only one example to compute the true gradient.
   - Very imprecise, but fast.

3. **Mini-batched SGD (SGD):**  
   *Mini-batch gradients*
   - Middle ground – use N uniformly sampled examples.
   - Can be sped up significantly through parallelization – so not much faster than approximate gradient methods.
   - Much more stable and much better converging than approximate gradient methods.
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies

http://mlsys.cst.cam.ac.uk/teach
```
def SGD(self, training_data, epochs, mini_batch_size, eta, 
        test_data=None):
    """Train the neural network using mini-batch stochastic 
    gradient descent. The `training_data` is a list of tuples 
    `(x, y)` representing the training inputs and the desired 
    outputs. The other non-optional parameters are 
    self-explanatory. If `test_data` is provided then the 
    network will be evaluated against the test data after each 
    epoch, and partial progress printed out. This is useful for 
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size] 
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format( 
                j, self.evaluate(test_data, n_test), n_test)
        else:
            print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying 
        gradient descent using backpropagation to a single mini batch. 
        The `mini_batch` is a list of tuples `(x, y)`, and `eta` 
        is the learning rate.""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw 
            for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb 
            for b, nb in zip(self.biases, nabla_b)]
```

```python
def backprop(self, x, y):
    """Return a tuple `(nabla_b, nabla_w)` representing the
    gradient for the cost function C_x. `nabla_b` and
    `nabla_w` are layer-by-layer lists of numpy arrays, similar
to `self.biases` and `self.weights`."
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x]  # list to store all the activations, layer by layer
    zs = []  # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
```

Source: Nielsen
def backprop(self, x, y):
    """Return a tuple `(nabla_b, nabla_w)` representing the
    gradient for the cost function `C(x)`. `nabla_b` and
    `nabla_w` are layer-by-layer lists of numpy arrays, similar
    to `self.biases` and `self.weights`."
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activations = [x]  # list to store all the activations, layer by layer
    zs = []  # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * 
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable 1 in the loop below is used a little
    # differently to the notation in Chapter 2 of the book. Here,
    # 1 = 1 means the last layer of neurons, 1 = 2 is the
    # second-last layer, and so on. It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    """Train the neural network using mini-batch stochastic gradient descent. The `training_data` is a list of tuples `(x, y)` representing the training inputs and the desired outputs. The other non-optional parameters are self-explanatory. If `test_data` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out. This is useful for tracking progress, but slows things down substantially."""

    if test_data: n_test = len(test_data)
    n = len(training_data)

    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k+mini_batch_size]
                        for k in xrange(0, n, mini_batch_size)]

        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)

        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying gradient descent using backpropagation of a single mini batch. The `mini_batch` is a list of tuples `(x, y)`, and `eta` is the learning rate."""

        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                        for b, nb in zip(self.biases, nabla_b)]
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies

http://mlsys.cst.cam.ac.uk/teach
Types of parallelism

model parallelism

1. Master
2. Worker
3. Model
4. Node values

data parallelism

1. Master
2. Worker
3. Data
4. Gradients
5. Parameter updates
SGD sources of data parallelism

Minibatch model update:

\[ \theta_{l,i}^{s+1} = \theta_{l,i}^{s} - \frac{r}{B} \sum_{b=1}^{B} g_{l,i,b}^{s} \]

where \( \theta_{l,i}^{s} \) is a parameter \( i \) of a layer \( l \) at step \( s \) in the training process, \( r \) is the learning rate, \( B \) is the batch size, and \( g_{l,i,b}^{s} \) is the gradient of a parameter \( i \) of a layer \( l \) at step \( s \) coming from example \( b \) of the mini-batch of size \( B \).
SGD sources of data parallelism

Parallelize gradient samples:

\[ \theta_{l,i}^{s+1} = \theta_{l,i}^{s} - \frac{r}{B} \sum_{b=1}^{B} g_{l,i,b}^{s} \]

Matrix multiplications in the computation of the example-level gradient samples can be parallelized.
SGD sources of data parallelism

Parallelize minibatch sum:

\[ \theta_{l,i}^{s+1} = \theta_{l,i}^s - \frac{r}{B} \sum_{b=1}^{B} g_{l,i,b}^s \]

The individual example-level gradient estimates can be parallelized before they are averaged and applied in the model update.
SGD sources of data parallelism

Parallelize multiple iterations of the update:

\[
\theta_{l,i}^{s+1} = \theta_{l,i}^s - \frac{r}{B} \sum_{b=1}^{B} g_{l,i,b}^s
\]

Separate model updates can be initialized and computed in parallel.
SGD sources of data parallelism

Parallelize hyperparameter optimization (model):

The search of hyperparameter search can, and should, be distributed between separate workers whenever possible.

http://mlsys.cst.cam.ac.uk/teach
Principles of Machine Learning Systems – v1.5
SGD sources of data parallelism

Parallelize ensembles (model):

Ensembles are collections of independently trained DL models that vote on a joint output. Both the training and the inference can be significantly sped up by parallelizing.
AlexNet on GPU

5 hidden convolutional layers

3 channel RGB input

Max pooling

Stride of 4

2-way split between the two GPUs

2 hidden fully connected

Source: Krizhevsky

http://mlsys.cst.cam.ac.uk/teach

Principles of Machine Learning Systems – v1.5
AlexNet on GPU so why the split?

• The **4096 neuron long layers** alone account for almost **1GB** of parameters needed to be stored in memory. This is before the storage of activations is accounted for.

• The authors chose to split the network between two **GTX 580 GPUs which have 3GB** device memory each.

Source: Krizhevsky
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies
Types of memory

1. **Model Memory** stores the model parameters, i.e. the weights and biases of each layer in the network.

2. **Optimizer Memory** stores the gradients and any momentum buffers during training. For instance, standard SGD with momentum saves one momentum value corresponding to each weight in the model.

3. **Activation Memory** stores the activations of each layer in the network – that is the outputs of each layer in the net.
Profiling

Figure 1: Pie charts of training memory consumption for (a) WideResNet on CIFAR-10 [left] and (b) DC-Transformer on IWSLT’14 German to English [right].

Source: Sohoni
Roadmap for Today

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies
Overview

- Sparse training
- Quantization
- Microbatching
- Gradient checkpointing

- Model & **Activations memory**
- Model & **Activations memory**
- **Activations memory**
- **Activations memory**

Source: Sohoni

http://mlsys.cst.cam.ac.uk/teach

Principles of Machine Learning Systems – v1.5
Sparse training

Dynamic Sparse Reparameterization

1. initializes the network with a fixed sparsity pattern,
2. during training the smallest-magnitude weights are pruned
3. same number of nonzeros are introduced back in the net

\[ \text{rewiring} \]

Source: Mostafa
Quantization (recap)

Quantized training

1. keep weights (and activations) in low precision numerical formats
2. perform updates in full precision (usually single precision float32)
3. quantize before next forward pass: stochastic rounding

Source: Han
Microbatching

**Minibatch size** directly translates one-to-one to activations size.

![Graph showing test accuracy vs. microbatch size](image)

Figure 4: Test accuracy vs. microbatch size, for WRN-28-2 on CIFAR-10. Vertical lines denote 95% confidence intervals. Dotted line is the baseline with a single microbatch, i.e. same minibatch and microbatch size of 100.

Source: Sohoni
Sparse, Quant., Microbatching eval.

Figure 6: Plot of test accuracy vs. sparsity for WRN-28-2 on CIFAR-10, for FP16 and FP32. The lines align extremely closely with one another, meaning that half precision does not significantly affect the accuracy.

Microbatch Size 10

Microbatch Size 4

Figure 7: Plot of test accuracy vs. training memory usage for WRN-28-2 on CIFAR-10. Each curve represents a different sparsity level, while different points on a given curve represent different microbatch sizes (100, 10, 4, and 2). All settings use FP16 and CHECKPOINT-RESIDUAL-2*.

Figure 8: Plot of test accuracy vs. training memory usage for WideResNet on CIFAR-10, when using a microbatch size of 10 (left) or 4 (right). The three curves in each plot are generated by varying the number of layers (“shallow”), the width of the layers (“narrow”), or the sparsity level of the layers (“sparse”). All settings use FP16 and CHECKPOINT-RESIDUAL-2*.

Table 3: Pareto-optimal settings for CIFAR-10 WideResNet training. The baseline, which is dense (100% nonzero) and uses 32-bit precision, a minibatch size of 100, no microbatching, and no checkpointing, is the top row. Memory reduction is computed relative to this baseline. All other settings use 16-bit precision, and CHECKPOINT-RESIDUAL-2*. Accuracies are reported as percentages; accuracies exceeding the standard-length training baseline are bolded. (Note: % refers to the percentage of nonzeros in convolutions.)

Source: Sohoni
Figure 6: Plot of test accuracy vs. sparsity for WRN-28-2 on CIFAR-10, for FP16 and FP32. The lines align extremely closely with one another, meaning that half precision does not significantly affect the accuracy.

Source: Sohoni
Figure 8: Plot of test accuracy vs. training memory usage for WideResNet on CIFAR-10, when using a microbatch size of 10 (left) or 4 (right). The three curves in each plot are generated by varying the number of layers (“shallow”), the width of the layers (“narrow”), or the sparsity level of the layers (“sparse”). All settings use FP16 and CHECKPOINT-RESIDUAL-2*.

Source: Sohoni
Summary & evaluation

Figure 7: Plot of test accuracy vs. training memory usage for WRN-28-2 on CIFAR-10. Each curve represents a different sparsity level, while different points on a given curve represent different microbatch sizes (100, 10, 4, and 2). All settings use FP16 and CHECKPOINT-RESIDUAL-2*.

Source: Sohoni
Table 3: Pareto-optimal settings for CIFAR-10 WideResNet training. The baseline, which is dense (100% nonzero) and uses 32-bit precision, a minibatch size of 100, no microbatching, and no checkpointing, is the top row. Memory reduction is computed relative to this baseline. All other settings use 16-bit precision, and CHECKPOINT-RESIDUAL-2*. Accuracies are reported as percentages; accuracies exceeding the standard-length training baseline are bolded. (Note: Nonzero % refers to the percentage of nonzeros in convolutions.)

Source: Sohoni
Gradient checkpointing

**Main idea:** save memory at the cost of re-doing some of the forward passes by only storing a subset of the network activations and recomputing the rest when they are needed.

- No accuracy trade-off: re-computed and original activations are mathematically and numerically the same!
- Pure trade-off between FLOPs and memory.
- Fig. FLOPs/original FLOPs vs. MB.

Source: Sohoni
## Putting it all together

![Graph showing memory usage comparison between ResNet (CIFAR-10) and Transformer (IWSLT’14)](source: Sohoni)

<table>
<thead>
<tr>
<th>Memory</th>
<th>Sparsity</th>
<th>Low Precision</th>
<th>Microbatching</th>
<th>Checkpointing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimizer</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activations</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td>↑</td>
</tr>
</tbody>
</table>

Source: Sohoni
Summary of the Day

1. The fundamentals of DL training
   1. DL model as a compute graph
   2. Training algorithm specifics
   3. NumPy implementation
   4. Exploitable parallelism

2. Resource management
   1. Characterising resource requirements
   2. Speed-up strategies