Topic 2: Efficient Learning

Inefficiencies in Learning

- Sampling Error and Overfitting
  - What if, the training data is sampled to have a certain pattern that is not present in the original input-output relationship?
- Variance in Stochastic Gradient Descent
  - Randomness in Gradient estimates can introduce a bias that is directly proportional to the variance in the gradient estimate.
- Complexity of Gradient Computation
  - Gradients for each complex model is very tedious to compute.

Can We Mitigate Overfitting?

- **Best Approach:** Get more data!
  - Train on different *bags* of data
- Use the right model...
  - Hard to accomplish... similar to model-based learning.
- Consider **model ensembles**...
  - Use different function classes (models of different forms)
  - Identify multiple weight parameters for different initializations and take their average
- How about **regularization** to limit the capacity of neural networks?
  - Limit the number of hidden layers and/or units per layer.
  - Early stopping criteria in optim algorithms, before overfitting begins
  - Penalize the objective for large weights using ℓ_1 penalty, or ℓ_2 penalty
  - Introduce hard constraint on weight capacity (*a.k.a.* max-norm, i.e. $||\mathbb{W}||_p \leq c.$).

Bootstrap Aggregation (in short, Bagging)[1]

- Reduce generalized error by aggregating the outcomes of several models.
- Generate a bootstrap sample, say $\mathcal{D}_k$, which follows the same distribution as the training set $\mathcal{D}$, for $k = 1, \cdots, K$.
- Train a new classifier, say $\hat{y}_k = \hat{f}_k(x)$, on each bootstrap sample $\mathcal{D}_k$.
- **Aggregation rule:** Count the number of times a class label appears amongst the K classifiers. The label with highest count is returned as output. Ties are broken by choosing the labels with lowest class label.
- Can also aggregate via averaging the outcomes.
- Let ϵ_k denote the error in the k^{th} classifier, with variance $\mathbb{E}(\epsilon_k^2) = v$, and covariance $\mathbb{E}(\epsilon_k \epsilon_j) = c$.

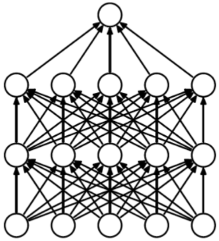
$$\begin{aligned}\mathbb{E}\left[\left(\frac{1}{K}\sum_{k=1}^{K}\epsilon_k\right)^2\right] &= \frac{1}{K^2}\mathbb{E}\left[\sum_{k=1}^{K}\left(\epsilon_k^2 + \sum_{j\neq k}\epsilon_k\epsilon_j\right)\right] \\ &= \frac{1}{K}v + \frac{K-1}{K}c\end{aligned}$$

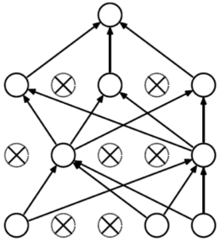
- If $c = 0$, squared error reduces by a factor of K.

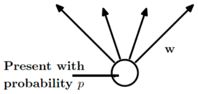
[1] Leo Breiman, "Bagging Predictors," *Machine learning*, vol. 24, no. 2, pp. 123-140, 1996.

Dropout[2]

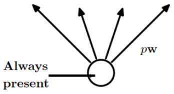
- Bagging and other ensemble methods is hard, especially with large neural network models.
- **Simple Approach:** Drop neurons with a fixed probability p, during training.
- Aggregate by averaging the weight parameters across different models


(a) Standard Neural Net


(b) After applying dropout.



(a) At training time



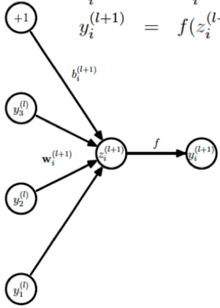
(b) At test time

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929-1958, 2014.

Dropout (cont...)

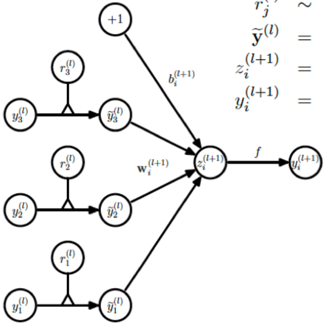
Formally, dropout is modeled as follows...

$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$



(a) Standard network

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\widetilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\widetilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$



(b) Dropout network

- **Note:** Backpropagation on thinned networks over several mini-batches of data
- Dropout + Norm-Regularization + AdaM (with large learning rates and high momentum) $\Rightarrow$ Huge boost in performance!

Regularization in Neural Networks

$$\hat{f}_N = \arg\min_{\mathbb{W}} \; L_N(\mathbb{W}) + \lambda \Phi(\mathbb{W}),$$

where $\Phi(\mathbb{W})$ is the term that penalizes undesirable weights.

- **Weight Decay (ℓ_2-Penalty):** $\Phi(\mathbb{W}) = \frac{1}{2}||\mathbb{W}||_2^2$
  - Also called ridge regression, or Tikhonov regularization
  - Drives weights closer to zero.
  - Let $\Psi_2(\mathbb{W}) = L_N(\mathbb{W}) + \lambda||\mathbb{W}||_2^2 \Rightarrow \nabla\Psi_2(\mathbb{W}) = \nabla L_N(\mathbb{W}) + 2\lambda\mathbb{W}$
  - GD: $\mathbb{W}^{r+1} = (1 - 2\delta\lambda)\mathbb{W}^r - \delta\nabla L_N(\mathbb{W}) \quad \Rightarrow \quad \mathbb{W} \to -\frac{1}{2\lambda}\nabla L_N(\mathbb{W})$
- **ℓ_1-Penalty:** $\Phi_1(\mathbb{W}) = ||\mathbb{W}||_1$
  - Results in sparse $\mathbb{W}$.
  - Let $\Psi_1(\mathbb{W}) = L_N(\mathbb{W}) + \lambda||\mathbb{W}||_1 \Rightarrow \nabla\Psi_1(\mathbb{W}) = \nabla L_N(\mathbb{W}) + 2\lambda \cdot \text{sign}(\mathbb{W})$
  - Weight update: $\mathbb{W}^{r+1} = \mathbb{W}^r - \delta\nabla L_N(\mathbb{W}) - 2\lambda\delta \cdot \text{sign}(\mathbb{W})$ (for each entry in $\mathbb{W}$)
    - $\mathbb{W}^{r+0.5} = \mathbb{W}^r - \delta\nabla L_N(\mathbb{W})$
    - If $\mathbb{W}^{r+0.5} > 0$, then $\mathbb{W}^{r+1} = \max\{0, \mathbb{W}^{r+0.5} - 2\lambda\delta\}$
    - If $\mathbb{W}^{r+0.5} < 0$, then $\mathbb{W}^{r+1} = \min\{0, \mathbb{W}^{r+0.5} + 2\lambda\delta\}$

Batch Normalization[3,4]

- **Internal Covariance Shift:** Change in distribution of activations due to change in model parameters – slows training
- **Solution:** Whitening activation functions
  - Linearly transform features to have zero mean and unit variances, i.e. decorrelated.

```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$       // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                 // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

[3] S. Ioffe, and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *International Conference on Machine Learning*, pp. 448-456, 2015.

[4] Ping Luo, Xinjiang Wang, Wenqi Shao, Zhanglin Peng, "Towards Understanding Regularization in Batch Normalization," *ICLR*, 2019.

Batch Normalization (cont...)

Backpropagation for BatchNorm:

$$
\begin{aligned}
\frac{\partial \ell}{\partial \widehat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\
\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \\
\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
\frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\
\frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i \\
\frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}
\end{aligned}
$$

Mini-Batch Stochastic Gradient Descent

- Random coordinate descent[5] (RCD) provides same convergence rate as that of full gradient descent
- RCD takes a fraction of effort by evaluating gradients only for one (or a small subset of) random coodinate(s).
- But, computer vision problems require a lot of images for training – RCD is *not* sufficient!
- **Solution:** Mini-batch SGD[6]

$$\left[\nabla L_N(\mathbb{W}^{r-1})\right]_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \ell\big(y_i, \hat{y}_i | \mathbb{W}^{r-1}\big)$$

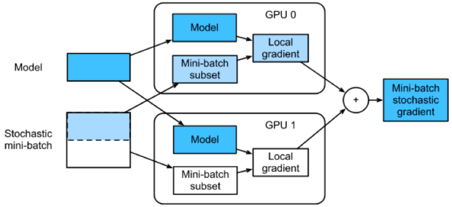
  - Estimate the gradient using a small sample (mini-batch) of training data.
  - Introduces bias which is a function of variance in the gradient estimate.
    - Variance in gradient estimate reduces as the size of mini-batch increases.
    - Alternatively, change the learning rate to $\delta_t = \delta \cdot \gamma^t$, where γ is a discounting factor that forces mini-batch SGD to converge.

[5] Y. Nesterov, "Efficiency of Coordinate Descent Methods on Huge-Scale Optimization Problems," *SIAM Journal on Optimization*, vol. 22, no. 2, pp. 341-362, 2012.

[6] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient Mini-Batch Training for Stochastic Optimization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 661-670. 2014.

Data Parallelism: Accelerating Computation using Hardware

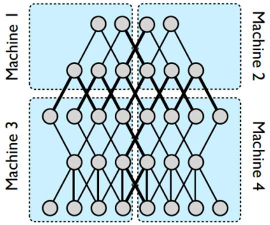
Parallelizing Mini-Batch SGD: Compute each mini-batch on a different GPU.



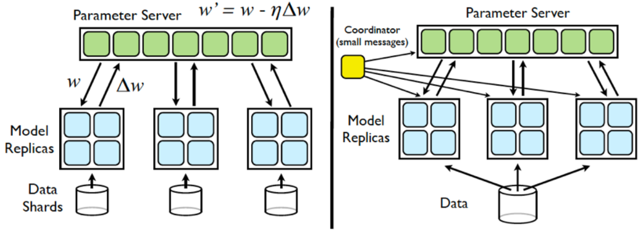
- Minibatch SGD is robust to *asynchronous* gradient computation
- However, some GPUs (workers) may compute gradient for older models due to asynchrony
- **Solution:** Synchronous SGD (Sync-SGD)
  - Wait until all gradients are available
  - Update time dictated by slowest worker.

Model Parallelism - for Large (Wide/Deep) Neural Networks

DistBelief: Each model partition on a different CPU.



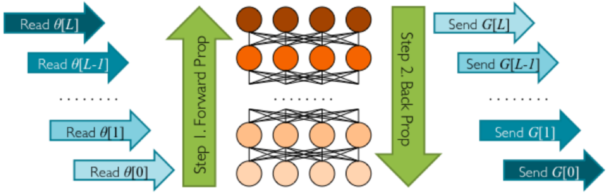
Model Aggregation: Models are trained on each mini-batch on a different GPU.



Source: J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato et al. "Large Scale Distributed Deep Networks," *Advances in neural information processing systems*, vol. 25, 2012.

Synchronous vs. Asynchronous Computation

- Optimization algorithms are robust to asynchronous gradient computation
- However, model parallelism is sensitive to asynchrony
- Synchronous Computation
  - Accurate outcomes
  - Slow due to some *stragglers* in the tail end of queue (gradients for lower layers).

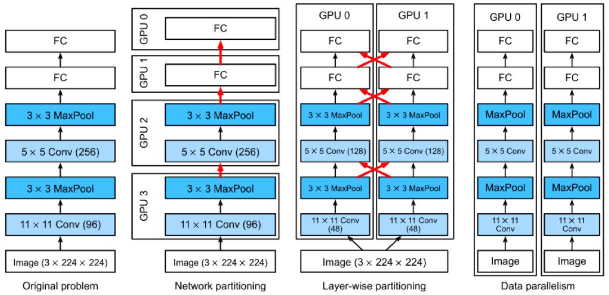


  - **Fix:** Can be accelerated using backup GPUs and discarding straggler computations.
- Mixed Computation
  - Group certain number of workers together, synchronize intra group and perform asynchronous updates across groups[7]
  - Slower than Sync-SGD!

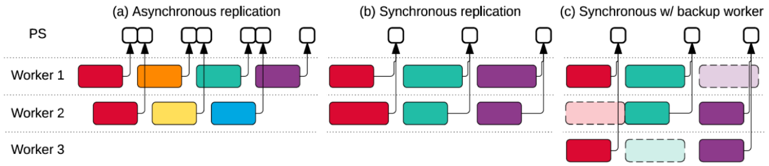
[7] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting Distributed Synchronous SGD," in *Proceedings of ICLR Workshop Track*, 2016. Available: https://arxiv.org/abs/1604.00981.

Combining Model/Data Parallelisms...

In summary, there are several parallel architectures:



Accordingly, modern deep learning libraries use one of the communication strategies:

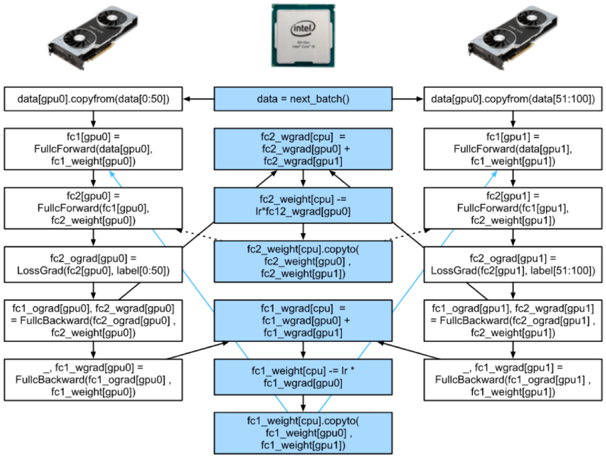


(a) Asynchronous replication
(b) Synchronous replication
(c) Synchronous w/ backup worker
PS
Worker 1
Worker 2
Worker 3

Source: M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265-283. 2016.

Illustrative Example: Data Workflow

Consider a simple two-layer neural network model trained on one CPU and two GPUs:



```
data[gpu0].copyfrom(data[0:50])
fc1[gpu0] = FullcForward(data[gpu0], fc1_weight[gpu0])
fc2[gpu0] = FullcForward(fc1[gpu0], fc2_weight[gpu0])
fc2_ograd[gpu0] = LossGrad(fc2[gpu0], label[0:50])
fc1_ograd[gpu0], fc2_wgrad[gpu0] = FullcBackward(fc2_ograd[gpu0] , fc2_weight[gpu0])
_, fc1_wgrad[gpu0] = FullcBackward(fc1_ograd[gpu0] , fc1_weight[gpu0])

data = next_batch()
fc2_wgrad[cpu] = fc2_wgrad[gpu0] + fc2_wgrad[gpu1]
fc2_weight[cpu] -= lr*fc12_wgrad[gpu0]
fc2_weight[cpu].copyto( fc2_weight[gpu0] , fc2_weight[gpu1])
fc1_wgrad[cpu] = fc1_wgrad[gpu0] + fc1_wgrad[gpu1]
fc1_weight[cpu] -= lr * fc1_wgrad[gpu0]
fc1_weight[cpu].copyto( fc1_weight[gpu0] , fc1_weight[gpu1])

data[gpu0].copyfrom(data[51:100])
fc1[gpu1] = FullcForward(data[gpu1], fc1_weight[gpu1])
fc2[gpu1] = FullcForward(fc1[gpu1], fc2_weight[gpu1])
fc2_ograd[gpu1] = LossGrad(fc2[gpu1], label[51:100])
fc1_ograd[gpu1], fc2_wgrad[gpu1] = FullcBackward(fc2_ograd[gpu1] , fc2_weight[gpu1])
_, fc1_wgrad[gpu1] = FullcBackward(fc1_ograd[gpu1] , fc1_weight[gpu1])
```

The only difficult aspect is to compute gradients using backprop!

Computational Graphs and Automatic Differentiation[8]

- Automatic differentiation is **not** symbolic computation!
  - The goal is not to evaluate a formula – instead compute the gradients efficiently.
  - Identify a sequence of primitive computations, which have specific routines for computing derivatives
  - Perform backprop in a mechanical way.
- Types of Automatic differentiation
  - Forward Mode (slower method)
  - Reverse Mode – Backprop!
- Consider the following simple example of an NN-1 model for scalar input:

Original Loss Function:

$$\ell = \frac{1}{2}\left[\frac{1}{1+\exp(-wx-b)} - y\right]^2$$

Computation Sequence:

$$z = wx + b$$

$$\hat{y} = \frac{1}{1+\exp(-z)}$$

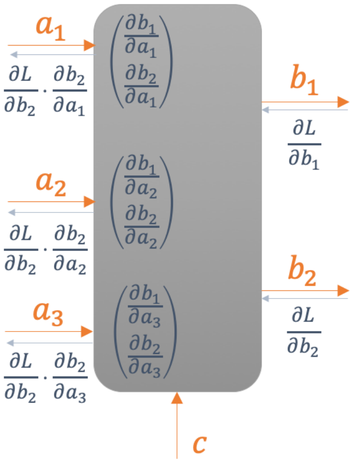
$$\ell = \frac{1}{2}(\hat{y}-y)^2$$

- Automatic Differentiation $\Rightarrow$ Construct a computational graph and evaluate backprop using primitives' gradients

[8] Source: A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. "Automatic Differentiation in Machine Learning: A Survey." *Journal of Marchine Learning Research*, vol. 18, pp. 1-43, 2018.

Message Passing in Computational Graphs

- During the backward pass, each node in the graph receives upstream gradients
- Compute downstream gradients by multiplying them by local gradients


a_1
a_2
a_3
b_1
b_2
c
$\begin{pmatrix} \frac{\partial b_1}{\partial a_1} \\ \frac{\partial b_2}{\partial a_1} \end{pmatrix}$
$\begin{pmatrix} \frac{\partial b_1}{\partial a_2} \\ \frac{\partial b_2}{\partial a_2} \end{pmatrix}$
$\begin{pmatrix} \frac{\partial b_1}{\partial a_3} \\ \frac{\partial b_2}{\partial a_3} \end{pmatrix}$
$\frac{\partial L}{\partial b_1}$
$\frac{\partial L}{\partial b_2}$

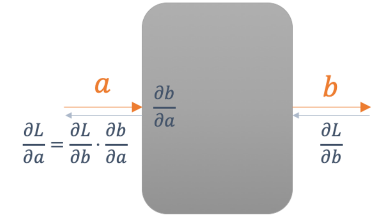
$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial b_1} \cdot \frac{\partial b_1}{\partial a_1} + \frac{\partial L}{\partial b_2} \cdot \frac{\partial b_2}{\partial a_1}$$

$$\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial b_1} \cdot \frac{\partial b_1}{\partial a_2} + \frac{\partial L}{\partial b_2} \cdot \frac{\partial b_2}{\partial a_2}$$

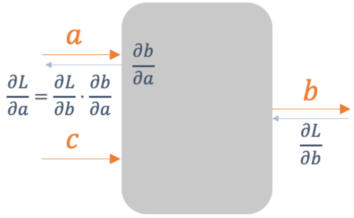
$$\frac{\partial L}{\partial a_3} = \frac{\partial L}{\partial b_1} \cdot \frac{\partial b_1}{\partial a_3} + \frac{\partial L}{\partial b_2} \cdot \frac{\partial b_2}{\partial a_3}$$

Message Passing on Primitives

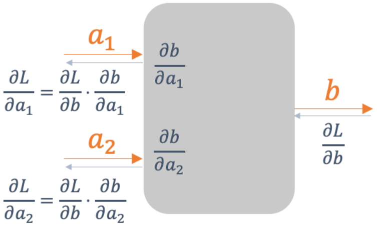
- Identify basic primitives that constitutes a given function
- Construct a computational graph automatically
- Perform message passing on gradients in reverse mode.



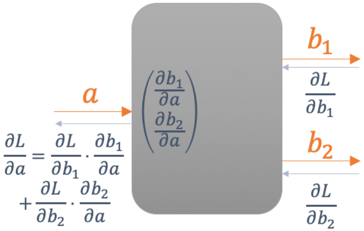
a
$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial a}$
$\frac{\partial b}{\partial a}$
b
$\frac{\partial L}{\partial b}$



a
$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial a}$
c
$\frac{\partial b}{\partial a}$
b
$\frac{\partial L}{\partial b}$



a_1
$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial a_1}$
a_2
$\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial a_2}$
$\frac{\partial b}{\partial a_1}$
$\frac{\partial b}{\partial a_2}$
b
$\frac{\partial L}{\partial b}$



a
$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b_1} \cdot \frac{\partial b_1}{\partial a} + \frac{\partial L}{\partial b_2} \cdot \frac{\partial b_2}{\partial a}$
$\begin{pmatrix} \frac{\partial b_1}{\partial a} \\ \frac{\partial b_2}{\partial a} \end{pmatrix}$
b_1
$\frac{\partial L}{\partial b_1}$
b_2
$\frac{\partial L}{\partial b_2}$

How to Construct a Computational Graph?

Approach:

- Define a node class
- Construct a primitive
- Define primitive on basic numpy functions

```
add_new = primitive(np.add)
mul_new = primitive(np.multiply)
div_new = primitive(np.divide)
sub_new = primitive(np.subtract)
neg_new = primitive(np.negative)
exp_new = primitive(np.exp)
```

Ref. https://tomroth.com.au/compgraph2/

```
class Node:
    """A node in a computation graph."""
    def __init__(self, value, fun, parents):
        self.parents = parents
        self.value = value
        self.fun = fun

    def __repr__(self):
        """A (very) basic string representation"""
        if self.value is None: str_val = 'None'
        else:                  str_val = str(round(self.value,3))
        return   "\n" + "Fun: " + str(self.fun) +\
                " Value: "+ str_val + \
                " Parents: " + str(self.parents)
```

```
from functools import wraps
def primitive(f):
    @wraps(f)
    def inner(*args, **kwargs):
        ## Code to add operation/primitive to computation graph

        # We need to separate out the integer/non node case.
Sometimes you are adding
        # constants to nodes.
        def getval(o):      return o.value if type(o) == Node else
o
        if len(args):       argvals = [getval(o) for o in args]
        else:               argvals = args
        if len(kwargs):     kwargvals = dict([(k,getval(o)) for
k,o in kwargs.items()])
        else:               kwargvals =  kwargs

        # get parents
        l = list(args) + list(kwargs.values())
        parents = [o for o in l if type(o) == Node ]

        value = f(*argvals, **kwargvals)
        print("add", "'" + f.__name__ + "'", "to graph with
value",value)
        return Node(value, f, parents)
    return inner
```

Using these primitives...

```
def start_node(value = None):
    """A function to create an empty node to start off the
graph"""
    fun,parents = lambda x: x, []
    return Node(value, fun, parents)

z = start_node(1.5)
t1 = mul_new(z, -1)
t2 = exp_new(t1)
t3 = add_new(t2, 1)
y = div_new(1,t3)
print("Final answer:", round(y.value,3))  # correct final output
print(y)
```

```
add 'multiply' to graph with value -1.5
add 'exp' to graph with value 0.22313016014842982
add 'add' to graph with value 1.22313016014843
add 'true_divide' to graph with value 0.8175744761936437
Final answer: 0.818

Fun: <ufunc 'true_divide'> Value: 0.818 Parents: [
Fun: <ufunc 'add'> Value: 1.223 Parents: [
Fun: <ufunc 'exp'> Value: 0.223 Parents: [
Fun: <ufunc 'multiply'> Value: -1.5 Parents: [
Fun: <function start_node.<locals>.<lambda> at 0x10fea27b8> Value:
1.5 Parents: []]]]]
```

Computational Graphs in PyTorch: A Simple Example[9]

Consider a simple function: $z = \log(x_1 x_2) * \sin(x_2)$

Primitives:

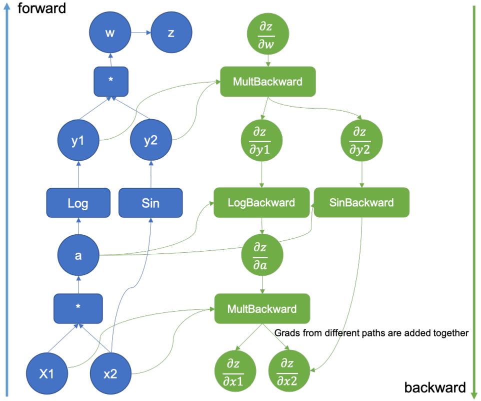
- $a = x_1 x_2$
- $y_1 = \log(a)$
- $y_2 = \sin(x_2)$
- $w = y_1 y_2$
- $z = w$

Derivatives:

- $\frac{\partial z}{\partial w} = 1$
- $\frac{\partial z}{\partial y_1} = \frac{\partial z}{\partial w} \cdot \frac{\partial w}{\partial y_1} = 1 \cdot y_2$
- $\frac{\partial z}{\partial y_2} = \frac{\partial z}{\partial w} \cdot \frac{\partial w}{\partial y_2} = 1 \cdot y_1$
- $\frac{\partial z}{\partial a} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial a} = y_2 \cdot \frac{1}{a}$
- $\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial a} \cdot \frac{\partial a}{\partial x_1} = \frac{y_2}{a} \cdot x_2$
- $\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial a} \cdot \frac{\partial a}{\partial x_2} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_2}$

  $= \frac{y_2}{a} \cdot x_1 + \cos(x_2)$

```
>>> import torch
>>> x = torch.tensor([0.5, 0.75], requires_grad=True)
>>> y = torch.log(x[0] * x[1]) * torch.sin(x[1])
>>> y.backward(1.0)
>>> x.grad
tensor([1.3633, 0.1912])</pre>
```

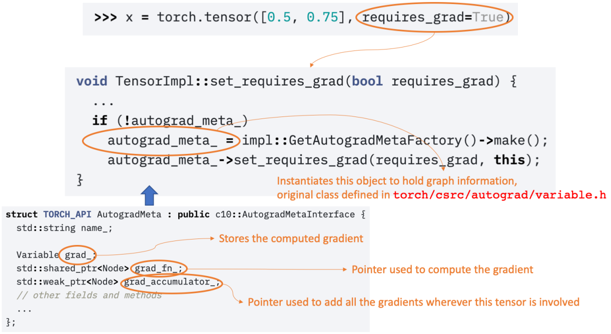

forward
w
z
*
y1
y2
Log
Sin
a
*
X1
x2
$\frac{\partial z}{\partial w}$
MultBackward
$\frac{\partial z}{\partial y1}$
$\frac{\partial z}{\partial y2}$
LogBackward
SinBackward
$\frac{\partial z}{\partial a}$
MultBackward
Grads from different paths are added together
$\frac{\partial z}{\partial x1}$
$\frac{\partial z}{\partial x2}$
backward

[9] Source: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

AutoGrad[10]: Automatic Differentiation in PyTorch

Components of Autograd:

- **tools/autograd:** Contains definitions of derivatives (`derivates.yaml`), and the necessary scripts to compute derivatives. Mostly written in C++.
- **torch/autograd:** Contains Python classes for users to write their own functions (`torch.autograd.Function`) and their gradients (`functional.py`)
- **torch/csrc/autograd:** Contains graph creation files and other execution-related code. Mostly written in C++.



```
>>> x = torch.tensor([0.5, 0.75], requires_grad=True)
```

```
void TensorImpl::set_requires_grad(bool requires_grad) {
  ...
  if (!autograd_meta_)
    autograd_meta_ = impl::GetAutogradMetaFactory()->make();
    autograd_meta_->set_requires_grad(requires_grad, this);
}
```

Instantiates this object to hold graph information, original class defined in `torch/csrc/autograd/variable.h`

```
struct TORCH_API AutogradMeta : public c10::AutogradMetaInterface {
  std::string name_;

  Variable grad_;
  std::shared_ptr<Node> grad_fn_;
  std::weak_ptr<Node> grad_accumulator_;
  // other fields and methods
  ...
};
```

Stores the computed gradient

Pointer used to compute the gradient

Pointer used to add all the gradients wherever this tensor is involved

[10] Source: https://github.com/pytorch/pytorch

Autograd: Computing the Gradient

```
>>> x = torch.tensor([0.5, 0.75], requires_grad=True)
>>> v = x[0] * x[1]
>>> v
tensor(0.3750, grad_fn=<MulBackward0>)
```

```
variable_list MulBackward0::apply(variable_list&& grads) {
  std::lock_guard<std::mutex> lock(mutex_);

  IndexRangeGenerator gen;
  auto self_ix = gen.range(1);
  auto other_ix = gen.range(1);
  variable_list grad_inputs(gen.size());
  auto& grad = grads[0];
  auto self = self_.unpack();
  auto other = other_.unpack();
  bool any_grad_defined = any_variable_defined(grads);
  if (should_compute_output({ other_ix })) {
    auto grad_result = any_grad_defined ? (mul_tensor_backward(grad, self, other_scalar_type)) : Tensor();
    copy_range(grad_inputs, other_ix, grad_result);
  }
  if (should_compute_output({ self_ix })) {
    auto grad_result = any_grad_defined ? (mul_tensor_backward(grad, other, self_scalar_type)) : Tensor();
    copy_range(grad_inputs, self_ix, grad_result);
  }
  return grad_inputs;
}
```

Generates this function automatically using **tools/autograd** based on the script present in **derivatives.yaml**

Inherited from **TraceableFunction** class, a descendant of **Node** with just a property to enable tracing for debugging/optim purposes.

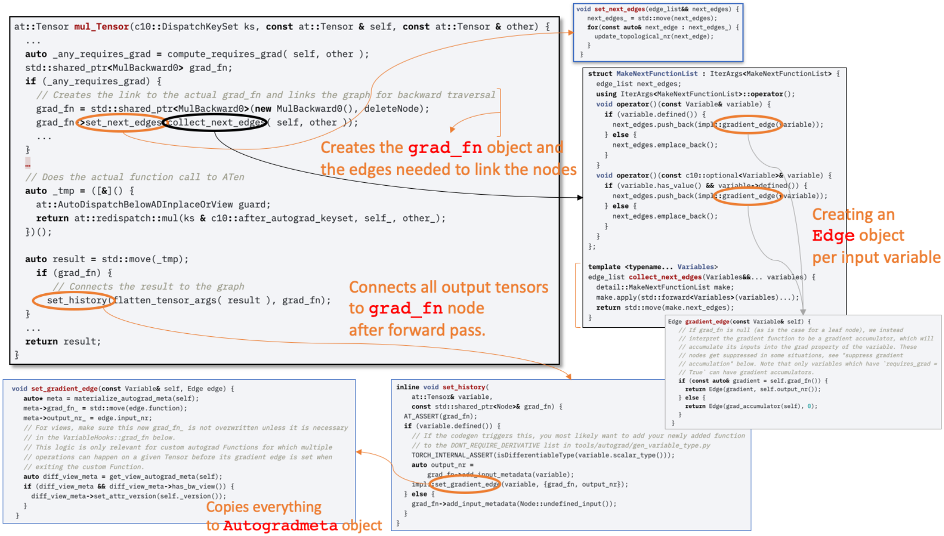
```
struct Edge {
  ...
  /// The function this `Edge` points to.
  std::shared_ptr<Node> function;
  /// The identifier of a particular input to the function.
  uint32_t input_nr;
};
```

```
struct TORCH_API Node : std::enable_shared_from_this<Node> {
 ...
 /// Evaluates the function on the given inputs and returns the result of the
  /// function call.
  variable_list operator()(variable_list&& inputs) {
  ...
  }

protected:
  /// Performs the `Node`'s actual operation.
  virtual variable_list apply(variable_list&& inputs) = 0;
  …
  edge_list next_edges_;
```

But, how is graph constructed, and how are the nodes linked automatically?

Autograd: Linking Nodes (Tensor Multiplication)



```
at::Tensor mul_Tensor(c10::DispatchKeySet ks, const at::Tensor & self, const at::Tensor & other) {
  ...
  auto _any_requires_grad = compute_requires_grad( self, other );
  std::shared_ptr<MulBackward0> grad_fn;
  if (_any_requires_grad) {
    // Creates the link to the actual grad_fn and links the graph for backward traversal
    grad_fn = std::shared_ptr<MulBackward0>(new MulBackward0(), deleteNode);
    grad_fn->set_next_edges(collect_next_edges( self, other ));
    ...
  }
  …
  // Does the actual function call to ATen
  auto _tmp = ([&]() {
    at::AutoDispatchBelowADInplaceOrView guard;
    return at::redispatch::mul(ks & c10::after_autograd_keyset, self_, other_);
  })();

  auto result = std::move(_tmp);
    if (grad_fn) {
      // Connects the result to the graph
      set_history(flatten_tensor_args( result ), grad_fn);
  }
  ...
  return result;
}
```

Creates the **grad_fn** object and the edges needed to link the nodes

Connects all output tensors to **grad_fn** node after forward pass.

```
void set_next_edges(edge_list&& next_edges) {
   next_edges_ = std::move(next_edges);
   for(const auto& next_edge : next_edges_) {
     update_topological_nr(next_edge);
   }
 }
```

```
struct MakeNextFunctionList : IterArgs<MakeNextFunctionList> {
  edge_list next_edges;
  using IterArgs<MakeNextFunctionList>::operator();
  void operator()(const Variable& variable) {
    if (variable.defined()) {
      next_edges.push_back(impl::gradient_edge(variable));
    } else {
      next_edges.emplace_back();
    }
  }
  void operator()(const c10::optional<Variable>& variable) {
    if (variable.has_value() && variable->defined()) {
      next_edges.push_back(impl::gradient_edge(variable));
    } else {
      next_edges.emplace_back();
    }
  }
};

template <typename... Variables>
edge_list collect_next_edges(Variables&&... variables) {
  detail::MakeNextFunctionList make;
  make.apply(std::forward<Variables>(variables)...);
  return std::move(make.next_edges);
}
```

Creating an **Edge** object per input variable

```
Edge gradient_edge(const Variable& self) {
   // If grad_fn is null (as is the case for a leaf node), we instead
   // interpret the gradient function to be a gradient accumulator, which will
   // accumulate its inputs into the grad property of the variable. These
   // nodes get suppressed in some situations, see "suppress gradient
   // accumulation" below. Note that only variables which have `requires_grad =
   // True` can have gradient accumulators.
   if (const auto& gradient = self.grad_fn()) {
     return Edge(gradient, self.output_nr());
   } else {
     return Edge(grad_accumulator(self), 0);
   }
 }
```

```
void set_gradient_edge(const Variable& self, Edge edge) {
   auto* meta = materialize_autograd_meta(self);
   meta->grad_fn_ = std::move(edge.function);
   meta->output_nr_ = edge.input_nr;
   // For views, make sure this new grad_fn_ is not overwritten unless it is necessary
   // in the VariableHooks::grad_fn below.
   // This logic is only relevant for custom autograd Functions for which multiple
   // operations can happen on a given Tensor before its gradient edge is set when
   // exiting the custom Function.
   auto diff_view_meta = get_view_autograd_meta(self);
   if (diff_view_meta && diff_view_meta->has_bw_view()) {
     diff_view_meta->set_attr_version(self._version());
   }
 }
```

Copies everything to **Autogradmeta** object

```
inline void set_history(
    at::Tensor& variable,
    const std::shared_ptr<Node>& grad_fn) {
  AT_ASSERT(grad_fn);
  if (variable.defined()) {
    // If the codegen triggers this, you most likely want to add your newly added function
    // to the DONT_REQUIRE_DERIVATIVE list in tools/autograd/gen_variable_type.py
    TORCH_INTERNAL_ASSERT(isDifferentiableType(variable.scalar_type()));
    auto output_nr =
        grad_fn->add_input_metadata(variable);
    impl::set_gradient_edge(variable, {grad_fn, output_nr});
  } else {
    grad_fn->add_input_metadata(Node::undefined_input());
  }
}
```