ELEG 5491: Introduction to Deep Learning

Optimization of Deep Neural Networks

Prof. LI Hongsheng

Office: SHB 428
e-mail: hsli@ee.cuhk.edu.hk
web: https://dl.ee.cuhk.edu.hk

Department of Electronic Engineering
The Chinese University of Hong Kong

February 2023

Outline

The objective function and gradients

- Given a general input feature vector x, the function f to be learned generates an output
$$\hat{y} = f(x; \theta)$$
and compares the output with ground-truth y
- The loss function J considering all training samples can be defined as
$$J(\theta; \mathcal{D}) = \sum_{\text{all } (x^{(i)}, y^{(i)}) \in \mathcal{D}} \text{Difference}\left(f(x^{(i)}), y^{(i)}\right)$$
where $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ denotes the training set
- Notation
  - Objective function (loss function) $J : \mathbb{R}^l \to \mathbb{R}$ (assuming the network has l parameters in total)
  - Gradient vector $\nabla J(\theta) = \left[\frac{\partial}{\partial \theta_1} J(\theta), \ldots, \frac{\partial}{\partial \theta_l} J(\theta)\right]^T \in \mathbb{R}^l$

Gradient descent

- We aim at minimizing the cost function on the training set

$$\theta^* = \arg\min_\theta J(\theta; \mathcal{D})$$

where $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ denotes the training set

- The plain gradient descent updates the function parameters θ as

$$\theta = \theta - \eta \nabla J(\theta)$$

where $\theta^{(t)}$ denotes parameters of the function J at iteration t, and α is the manually set learning rate (a hyper-parameter)

- The gradient descent can only find a local optimum of the objective function

Algorithm 1: Plain gradient descent

```
Input: initial $\theta^{(0)}$, gradient vector $\nabla J(\theta)$, learning rate $\alpha$, tolerance $\omega$
$|\Delta\theta| \leftarrow \infty$ ;
while $|\Delta\theta| < \omega$ for more than 10 iterations do
    $\Delta\theta \leftarrow -\eta \nabla J(\theta)$;
    $\theta \leftarrow \theta + \Delta\theta$;
end
```

Mini-batch Stochastic Gradient Descent

- Obtaining $\nabla J(\theta)$ requires evaluating the neural network $f(x)$ for each training sample over the entire train set, which is too time-consuming if the number of training samples is too large
- Mini-batch stochastic gradient descent estimates the gradient vector of $J(\theta)$ by using a mini-batch of training sample at each iteration

$$J(\theta, \mathcal{D}) = \sum_{(x^{(i)}, y^{(i)}) \in \mathcal{B}} \text{Difference}\left(f(x^{(i)}), y^{(i)}\right)$$

where $\mathcal{B}$ is a mini-batch of the training set, which can be sequentially or randomly obtained from the train set $\mathcal{D}$

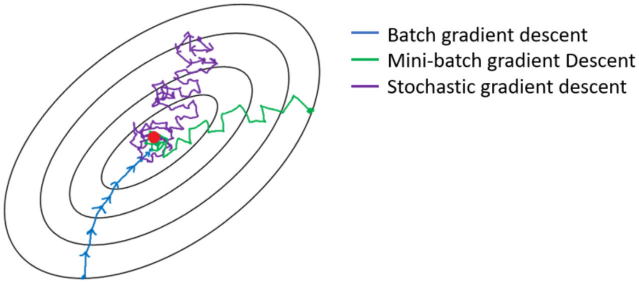
Algorithm 2: Mini-batch stochastic gradient descent

```
Input: initial $\theta^{(0)}$, gradient vector $\nabla J(\theta)$, learning rate $\alpha$, tolerance $\omega$
$|\Delta\theta| \leftarrow \infty$ ;
while The iteration number is below than a threshold do
    Randomly or sequentially sample a mini-batch of samples $\mathcal{B}$ from the train set $\mathbf{D}$;
    Estimate $\nabla J(\theta)$ with the mini-batch of samples $\mathcal{B}$;
    $\theta \leftarrow \theta - \eta \nabla J(\theta)$;
end
```

- Much faster but the estimated gradient vector might be noisy

Mini-batch Stochastic Gradient descent

- General guidelines on forming mini-batches
  - In general, larger mini-batches would results in better results than small mini-batches
  - The use of BN layers requires mini-batches of at least a batch size of 8 or 16
  - The feature and label variations within each mini-batch should be maximized as much as possible
- Comparison of different gradient descent methods

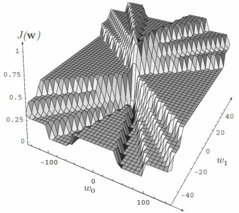


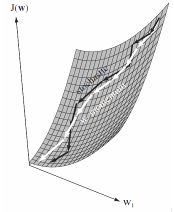
Momentum in Gradient-based Optimization

- One can store the parameter update vector $\Delta\theta^{(t-1)}$ at the previous iteration
- After calculating the negative gradient vector $\nabla J_\theta(\theta^{(t)})$ at the current iteration t
- The parameter update vector at current iteration t is calculated as

$$\Delta\theta^{(t)} = \mathrm{momentum} \cdot \Delta\theta^{(t-1)} + (1 - \mathrm{momentum}) \cdot \nabla J_\theta(\theta^{(t)})$$
$$\Delta\theta^{(0)} = 0$$

  where the hyper-parameter $\mathrm{momentum}$ is generally set as 0.9
- The momentum can lead to faster convergence and prevent sudden change of optimization direction





Weight Decay in Gradient-based Optimization

- In general, the function family that neural networks can represent is huge and NN has power capability of overfitting small-scale dataset
- Given multiple sets of parameters that lead to the same training error (or loss), one would favor the set of parameters that insensitive to input feature vectors' variations
- We can minimize the L_2 norm (magnitude) of the parameter vector $\|\theta\|_2$ to achieve the goal, which is used a regularization term
- The objective (loss) function becomes

$$J(\theta;\mathcal{D}) = \sum_{\text{all } (x^{(i)},y^{(i)})\in\mathcal{B}} \text{Difference}\left(f(x^{(i)}), y^{(i)}\right) + \frac{\lambda}{2}\|\theta\|_2^2$$

  where λ is a hyper-parameter controlling the influence of the regularization term
- With only regularization term, its updating with negative gradient of the above objective function is

$$\theta^{(t+1)} := \theta^{(t)} - \eta\nabla J_\theta(\theta^{(t)})$$

- Some implementations only apply weight decay to linear transformation weights but not bias. PyTorch applies weight decay to both weights and

SGD with Decoupled Weight Decay

- J can be defined as a plain loss function without L2 regularization and SGD with weight decay can be defined as

$$\theta^{(t+1)} := \theta^{(t)} - \eta \cdot \nabla J_\theta(\theta^{(t)}) - \eta \cdot \lambda \cdot \theta^{(t)}$$

- Note that the negative gradients above, $-\nabla J_\theta$, are computed from the plain loss function
- If we use the above decouped weight decay formula, we avoid add more computations by modifying the loss (the other benefit will be explained in AdamW)

- SGD with decoupled weight decay and momentum is implemented as

$$\begin{aligned}\Delta\theta^{(t+1)} &= \mathrm{momentum} \cdot \Delta\theta^{(t)} + (1 - \mathrm{momentum})\nabla J_\theta(\theta^{(t)}) \\ \theta^{(t+1)} &= \theta^{(t)} - \eta \cdot \Delta\theta^{(t+1)} - \eta \cdot \lambda \cdot \theta^{(t)}\end{aligned}$$

- This update scheme is actually different from using the loss function with L2 regularization

Brief Introduction to 2nd-order optimization

- Hessian (symmetric matrix) of the objective function J

$$\nabla^2 J(x) = H = \begin{pmatrix} \frac{\partial^2}{\partial\theta_1 \partial_{\theta_1}} J(\theta) & \frac{\partial^2}{\partial_{\theta_1}\partial_{\theta_2}} J(\theta) & \cdots & \frac{\partial^2}{\partial_{\theta_1}\partial_{\theta_n}} J(\theta) \\ \frac{\partial^2}{\partial_{\theta_1}\partial_{\theta_2}} J(\theta) & & & \vdots \\ \vdots & & & \vdots \\ \frac{\partial^2}{\partial_{\theta_n}\partial_{\theta_1}} J(\theta) & \cdots & \cdots & \frac{\partial^2}{\partial_{\theta_n}\partial_{\theta_n}} J(\theta) \end{pmatrix} \in \mathbb{R}^{n\times n}$$

- Newton's method centered around a quadratic approximation of f for points near $x^{(t)}$

$$J(\theta + \Delta\theta) = J(\theta) + \Delta\theta^T \nabla J(\theta) + \frac{1}{2}\Delta\theta^T(\nabla^2 J(\theta))\Delta\theta$$

- Without loss of generality, we write $\theta^{(t+1)} = \theta^{(t)} + \Delta\theta$ and define $h^{(t)}$ as a function of $\Delta\theta$

$$h^{(t)}(\Delta\theta) = J(\theta^{(t)}) + \Delta\theta^T g^{(t)} + \frac{1}{2}\Delta\theta^T H^{(t)}\Delta\theta$$

where $g^{(t)}$ and $H^{(t)}$ denote the gradient and Hessian at $\theta^{(t)}$

Brief Introduction to 2nd-order optimization

- We choose $\Delta\theta$ to minimize the local quadratic approximation of J at $\theta^{(t)}$
- Differentiating $h^{(t)}$ w.r.t. $\Delta\theta$ yields

$$\frac{\partial h^{(t)}(\Delta\theta)}{\partial \Delta\theta} = g^{(t)} + H^{(t)}\Delta\theta$$

- Setting the derivative to zero yields

$$\Delta\theta = (H^{(t)})^{-1} g^{(t)}$$

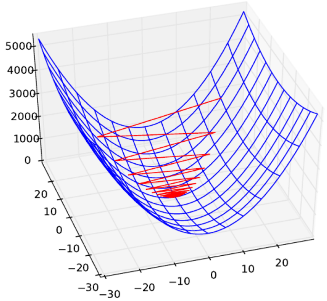
Suggesting that $-(H^{(t)})^{-1} g^{(t)}$ is a good direction to update $\theta^{(t)}$

2nd-order iterative algorithm

- For $t = 1, 2, \dots$
  - Compute $g^{(t)}$ and $(H^{(t)})^{-1}$ for $\theta^{(t)}$
  - $d = (H^{(t)})^{-1} g^{(t)}$
  - $\eta = \min_{\eta \geq 0} J(\theta^{(t)} - \eta d)$
  - $\theta^{(t+1)} \leftarrow \theta^{(t)} - \alpha d$

- The computation of α can be obtained by any line search algorithm. The simplest is backtracking line search – trying smaller and smaller α until the function is small enough

Advantage of 2nd-order optimization



Gradient descent fails to exploit the curvature information contained in Hessian. Here we use gradient descent on a quadratic function whose Hessian matrix has condition number 5. The red lines indicate the path followed by gradient descent. This very elongated quadratic function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature. Because the learning rate is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration. The large positive eigenvalue of the Hessian corresponding to the eigenvector pointed in this direction indicates that this directional derivative is rapidly increasing, so an optimization algorithm based on the Hessian could predict that the steepest direction is not actually a promising search direction in this context.

Resilient Propagation (Rprop)

- RProp is a popular gradient descent algorithm that only uses the signs of gradients to compute updates
- Let $\eta_i^{(t)}$ denote the learning rate for the ith weight at the tth iteration
- Rprop updates parameters as

$$\theta_i^{(t)} = \theta_i^{(t)} - \eta_i^{(t)} \mathrm{sgn}\left(\frac{\partial J}{\partial \theta_i^{(t)}}\right)$$

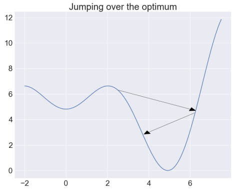
- The learning rate $\eta_i^{(t)}$ is dynamically adapted for each weight θ_i depending on its gradient

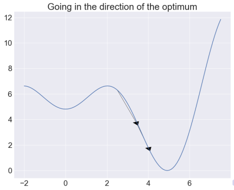
Resilient Propagation (Rprop)

- For each weight, when its gradient sign of the current and previous iterations are them same, we increase the learning rate as this seems to be a good direction
- If the gradient sign changes, it denotes that the parameter just jumps over an optimum. We decrease the learning rate to avoid jumping over the optimum again

$$\eta_i^{(t)} = \begin{cases} \min(\alpha\eta_i^{(t-1)}, \eta_{\max}) & \text{if } \frac{\partial J}{\partial\theta^{(t)}}\frac{\partial J}{\partial\theta^{(t-1)}} > 0, \\ \max(\beta\eta_i^{(t-1)}, \eta_{\min}) & \text{if } \frac{\partial J}{\partial\theta^{(t)}}\frac{\partial J}{\partial\theta^{(t-1)}} < 0, \\ \eta_i^{(t-1)} & \text{otherwise.} \end{cases}$$

- $\alpha > 1 > \beta$ scale the learning rate. Empirically, $\alpha = 1.2$, $\beta = 0.5$
- The learning rate is also clipped to avoid it becoming too large or small





Adagrad

- The basic stochastic gradient descent (SGD) optimization used the same learning rate for all parameters θ
- Adagrad uses a different learning rate for every parameter θ_i
- Denote $g^{(t)}$ as the gradient and $g_i^{(t)}$ as the partial derivative of the loss function J w.r.t. the parameter θ_i at iteration t
- The SGD update for parameter θ_i

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{G_{ii}^{(t)} + \epsilon}} g_i^{(t)}$$

  η is the overall learning rate and $G^{(t)} \in \mathbb{R}^{n \times n}$ is a diagonal matrix where each diagonal $G_{ii}^{(t)}$ is the sum of squares of gradients w.r.t. θ_i up to iteration t
- Intuitively, if each parameter θ_i is updated for a too large accumulated amount, its learning rate gradually becomes smaller and smaller
- Weakness: the denominator always increases, causing the learning rate to shrink and eventually become infinitely small

Adadelta

- Instead of accumulating all past gradients, Adadelta restricts the time window of accumulated past gradients to fixed size w
- The running average $E[(g^{(t)})^2]$ at iteration t is calculated as
$$E[g^2]^{(t)} = \gamma E[g^2]^{(t-1)} + (1-\gamma)(g^{(t)})^2, \quad \text{with } E[g^2]^{(0)} = 0$$
γ can be set as 0.9 as a common practice
- The units (magnitudes) of different parameters might not match. The authors observed that, in gradient methods, $\nabla f(x) \propto \frac{1}{\text{units of } \theta}$
- Another exponentially decaying averaging normalization term is defined as
$$E\left[\Delta\theta^2\right]^{(t)} = \gamma E\left[\Delta\theta^2\right]^{(t-1)} + (1-\gamma)(\Delta\theta^2)^{(t)}, \quad \text{with } E[\Delta\theta^2]^{(0)} = 0$$
- The Adadelta update rule:
$$\Delta\theta^{(t)} = \frac{\sqrt{E\left[\Delta\theta^2\right]^{(t)} + \epsilon}}{\sqrt{E[g^2]^{(t)} + \epsilon}} g^{(t)}$$
$$\theta^{(t+1)} = \theta^{(t)} - \Delta\theta^{(t)}$$

RMSprop

- RMSprop is developed independently around the same time as Adadelta
- The update rule is

$$E\left[g^2\right]^{(t)} = 0.9E\left[g^2\right]^{(t-1)} + 0.1(g^{(t)})^2 \quad \text{with } E[g^2]^{(0)} = 0$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{E\left[g^2\right]^{(t)} + \epsilon}} g^{(t)}$$

Adam

- Adaptive Moment Estimation (Adam) also adaptively tunes the learning rate of each parameter
- Adam keeps exponentially decaying average of past squared gradients $v^{(t)}$ and an exponentially decaying average of past gradients $m^{(t)}$

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1)\, g^{(t)},\ \beta_1 = 0.9 \text{ (by default)}$$
$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)\, (g^{(t)})^2\ \ \beta_2 = 0.999 \text{ (by default)}$$

- $m^{(0)}$ and $v^{(0)}$ are initialized as vectors of all 0's and would therefore be biased towards zero, especially during the initial iterations and when β_1 and β_2 are small
- They are further counteracted:

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1},\ \hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2}$$

- The Adam update rule:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{v}^{(t)}} + \epsilon}\hat{m}^{(t)}$$

AdamW: Adam with (Decoupled) Weight Decay

- Adam with decoupled weight decay update (AdamW) rule:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{v}^{(t)}} + \epsilon}\hat{m}^{(t)} - \eta \cdot \lambda \cdot \theta^{(t)}$$

- **Problem of Adam:** Note that if the L2 regularization is used in the objective function, the corresponding update rule (after expansion and ignore ˆ) becomes

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{\beta_1 m^{(t)} + (1-\beta_1)\left(\nabla J_\theta\left(\theta^{(t)}\right) + \lambda\theta^{(t)}\right)}{\sqrt{v^{(t)}} + \epsilon}$$

- Weight decay is influenced by $\sqrt{v^{(t)}}$: if the gradient of a certain weight is large, weight decay is not as effective as the update rule above
- This was the reason why Adam wasn't so successful when it was first released

Quasi-Newton Methods

- θ of the deep neural networks are generally of very high dimension
- Evaluation of the Hessian matrix H might be computationally infeasible in most scenarios
- We can approximate $H = \nabla^2 J(\theta)$ from the data of the previous iterations
- A typical quasi-Newton iteration is

$$\theta^{(t+1)} = \theta^{(t)} + \eta^{(t)} d^{(t)}, \text{ where } d^{(t)} = -B^{(t)} \nabla J(\theta^{(t)})$$

  $\alpha^{(t)}$ is usually chosen by a line search
- $B^{(t)}$ is a positive definite matrix chosen so that the direction $d^{(t)}$ tends to approximate Newton's direction
- Two successive iterates $\theta^{(t)}$ and $\theta^{(t+1)}$ with the gradients $g^{(t)}$ and $g^{(t+1)}$ contain curvature (Hessian) information

$$g^{(t+1)} - g^{(t)} \approx H^{(t+1)}(\theta^{(t+1)} - \theta^{(t)})$$

  This is known as the secant equation or the quasi-Newton condition
- We choose $B^{(t+1)}$ to satisfy

$$B^{(t+1)} q^{(t)} = p^{(t)}, \text{ where } p^{(t)} = \theta^{(t+1)} - \theta^{(t)}, q^{(t)} = g^{(t+1)} - g^{(t)}$$

BFGS and Limited-memory BFGS

- Suppose that at every iteration we update the matrix $B^{(t+1)}$ by taking the matrix $B^{(t)}$ and adding a "correction" matrix $C^{(t)}$

$$(B^{(t)} + C^{(t)})q^{(t)} = p^{(t)} \Rightarrow C^{(t)}q^{(t)} = p^{(t)} - B^{(t)}q^{(t)}$$

- The most popular choice is the Broyden family

$$C^{\mathrm{B}}(\xi) = \frac{pp^T}{p^Tq} - \frac{\mathrm{B}qq^T\ \mathrm{B}}{q^T\ \mathrm{B}q} + \xi\tau vv^T, \text{ where } v = \frac{p}{p^Tq} - \frac{\mathrm{B}q}{\tau}, \tau = q^T\ \mathrm{B}q$$

  $\xi \in [0,1]$ and the above formula indeed satisfies the secant condition

- Setting $\xi = 1$, we obtain the BFGS update

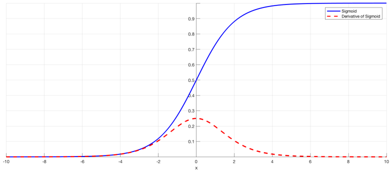
$$C^{\mathrm{BFGS}} = C^{\mathrm{B}}(1) = \frac{pp^T}{p^Tq}\left[1 + \frac{q^TBq}{p^Tq}\right] - \frac{Bqp^T + pq^TB}{p^Tq}$$

- L-BFGS (Limited-memory BFGS) further approximates BFGS using a limited amount of memory

- Gradient vanishing refers to the problem that the gradients of lower layers normally get smaller and smaller, gradually approaching zero, causing gradient-based optimization method to never converge to the optimum
- Gradient vanishing is most apparent (but can also observed for networks with other activation functions) for networks with sigmoid (tanh) activation functions
- The gradient can be calculated as (below is not a strict formula)

$$\frac{\partial J}{\partial y^1} = \sigma(y^2)'(W^1)^T\sigma(y^2)'(W^2)^T\sigma(y^3)'(W^3)^T\sigma(y^4)'\frac{\partial J}{\partial y^4}$$

$$\begin{tikzcd} \bigcirc \arrow[r] & y^1 \arrow[r, "W^1"] & y^2 \arrow[r, "W^2"] & y^3 \arrow[r, "W^3"] & y^4 \arrow[r] & J \end{tikzcd}$$



- The maximum of $\sigma'(x) \approx 0.25$. Multiplying the multiple < 1 values might makes the gradients gradually smaller

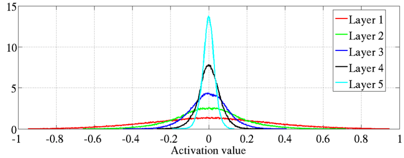
- Gradient exploding refers to the gradients of lower layers become extremely large. It is usually caused by too large values of $W^1, W^2, W^3, \dots$
- Some solutions:
  - Batch normalization is a standard method for solving both the exploding and the vanishing gradient problems
  - Gradient clipping clip the norm of $\nabla_\theta J$ by ϵ

$$g = \begin{cases} \nabla_\theta J, & \text{if } \|\nabla_\theta J\| < \epsilon, \\ \epsilon \cdot \frac{\nabla_\theta J}{\|\nabla_\theta J\|}, & \text{otherwise.} \end{cases}$$

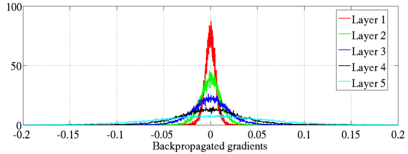
  - Long-short Term Memory for Recurrent Neural Networks
  - Residual connections to make the gradient back-propagated easier through the network
  - Other activation functions: e.g., ReLU allows back-propagating gradients easier
  - Weight initialization to reduce vanishing or exploding gradients

Weight initialization

- Gradient-based optimization methods require initial parameters/weights
- The simplest initialization method is to initilize weights of all layers following the same standard normal distribution or uniform distribution
- However, if the weights are initialized improperly, it can lead to exploding or vanishing weights and gradients: either the outputs of the network explode to infinity, or they vanish to 0



Feature values of a 5-layer MLP with tanh function and with Gaussian random initialization [Xavier et al.]



Back-propagated gradients of a 5-layer MLP with tanh function and with Gaussian random initialization

Analysis of forward response distribution

- The forward computation of a fully-connected layer is formulated as

$$y = Wx + b$$

where $x \times \mathbb{R}^u$, $y, b \in \mathbb{R}^d$, $W \in \mathbb{R}^{u \times d}$

- The overall objective of weight initialization: maintaining the same signal magnitude across different layers

$$\mathrm{Var}(y_i) = \mathrm{Var}(x_j)$$

- Assumptions:
  - W, x, b are independent of each other
  - The elements of $W \in \mathbb{R}^{u \times d}$, i.e., W_{ij} are independent and identically distributed (i.i.d.) and $E[W_{ij}] = 0$
  - The elements of $b \in \mathbb{R}^d$ (i.e. b_i) are initialized as all zeros so $\mathrm{Var}[b_i] = 0$
  - The elements of $x \in \mathbb{R}^u$ (i.e. x_j) are i.i.d. and $E[x_j] = 0$
- Review: if X and Y are independent, we have

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$$

$$\mathrm{Var}(XY) = \mathrm{Var}(X)\,\mathrm{Var}(Y) + (E[X])^2\,\mathrm{Var}(Y) + \mathrm{Var}(X)(E[Y])^2$$

Analysis of forward response distribution

- The variance of $\{y_i\}_{i=1}^{d}$ is calculated as

$$\begin{aligned}
\mathrm{Var}\left(y_i\right) &= \mathrm{Var}\left(W_i x + b_i\right) \\
&= \mathrm{Var}\left(\sum_{j=1}^{d} W_{ij} x_j + b_i\right) \\
&= d\,\mathrm{Var}\left(W_{ij} x_j\right) \\
&= d\left(\mathrm{Var}\left(W_{i,j}\right)\mathrm{Var}\left(x_j\right) + \left(E\left[W_{i,j}\right]\right)^2 \mathrm{Var}\left(x_j\right) + \mathrm{Var}\left(W_{ij}\right)\left(E\left[x_j\right]\right)^2\right) \\
&= d\left(\mathrm{Var}\left(W_{ij}\right)\mathrm{Var}\left(x_j\right) + (0)^2 \mathrm{Var}\left(x_j\right) + \mathrm{Var}\left(W_{ij}\right)\left(E\left[x_j\right]\right)^2\right) \\
&= d\,\mathrm{Var}\left(W_{ij}\right)\mathrm{Var}(x_j) \\
&= d\,\mathrm{Var}\left(W_{i,j}\right)\left(\mathrm{Var}\left(x_j\right) + \left(E\left[x_j\right]\right)^2\right) = d\,\mathrm{Var}\left(W_{ij}\right)\left(E\left[x_j^2\right] - E[x_j]^2 + I\right. \\
&= d\,\mathrm{Var}\left(W_{ij}\right) E\left[x_j^2\right]
\end{aligned}$$

- To achieve $\mathrm{Var}\left(y_i\right) = \mathrm{Var}\left(x_j\right)$, we have $d\mathrm{Var}(W_{ij}) = 1$ and $\mathrm{Var}(W_{ij}) = \frac{1}{d}$
- For normal random numbers, $W_{ij} \sim \mathcal{N}(0, 1/d)$
- For uniform random numbers, $W_{ij} \sim \mathcal{U}(-\sqrt{3/d}, \sqrt{3/d})$

Analysis of backward response distribution

- Forward: $y = Wx + b$; Backward: $\frac{\partial J}{\partial x_j} = W^T \frac{\partial J}{\partial y_i}$
- Objective: $\mathrm{Var}\left(\partial J/\partial x\right) = \mathrm{Var}\left(\partial J/\partial y\right)$
- Assumptions:
  - $\partial J/\partial y$ and W are independent of each other
  - $\partial J/\partial y_i$ are i.i.d. and $E[\partial J/\partial y_i] = 0$
  - W_{ij} are i.i.d. and $E[W_{ij}] = 0$
- The analysis is similar as before and we have

$$\mathrm{Var}(\partial J/\partial x_j) = u\mathrm{Var}(W_{ij})\mathrm{Var}(\partial J/\partial y_i)$$

- To ensure $\mathrm{Var}(\partial J/\partial x) = \mathrm{Var}(\partial J/\partial y)$, we have $u\mathrm{Var}(W_{ij}) = 1$ and $\mathrm{Var}(W_{ij}) = 1/u$
- For normal random numbers, $W_{ij} \sim \mathcal{N}(0, 1/u)$
- For uniform random numbers, $W_{ij} \sim \mathcal{U}(-\sqrt{3/u}, \sqrt{3/u})$

Xavier initialization

- In general, $u \neq d$. The harmonic mean is used for $\mathrm{Var}(W_{ij})$:

$$\mathrm{Var}(W_{ij}) = \frac{2}{d+u}$$

- For normal random numbers, $W_{ij} \sim \mathcal{N}(0, 1/(d+u))$
- For uniform random numbers, $W_{ij} \sim \mathcal{U}(-\sqrt{6/(d+u)}, \sqrt{6/(d+u)})$
- However, Xavier initialization doesn't consider any activation/non-linearity function at all

Kaiming initialization

- The Kaiming initialization is similar but it considers ReLU activation function
$$\mathrm{ReLU}(y) = \mathrm{ReLU}(Wx + b)$$
- Follow the previous derivation, we have
$$\mathrm{Var}(y_i) = u\mathrm{Var}(W_{ij})E[x_j^2]$$
But we no longer have $E[x_j^2] = \mathrm{Var}(x_j)$ unless $E[x_j] = 0$, because ReLU outputs are non-negative
- We can simplify the $E[x_j^2]$ term (we drop the subscript j but add a layer-index superscript below)
$$\begin{aligned} E[(x^l)^2] &= \int_{-\infty}^{\infty} (x^l)^2 P(x^l) dx^l \\ &= \int_{-\infty}^{\infty} \max(0, y^{l-1})^2 P(y^{l-1}) dy^{l-1} \\ &= \int_{0}^{\infty} (y^{l-1})^2 P(y^{l-1}) dy^{l-1} \\ &= 0.5 \int_{-\infty}^{\infty} (y^{l-1})^2 P(y^{l-1}) dy^{l-1} \\ &= 0.5\,\mathrm{Var}(y^{l-1}) \end{aligned}$$

Kaiming initialization

- Let n^l denotes the number of output units at layer l
- The variance of units can be obtained as

$$\mathrm{Var}(y^l) = 0.5 \cdot n^l \cdot \mathrm{Var}(W^l) \cdot \mathrm{Var}(y^{l-1})$$

- Combining layer 1 to L

$$\mathrm{Var}(y^L) = \mathrm{Var}\left(y^1\right) \left(\prod_{l=2}^{L} \frac{n^l}{2} \mathrm{Var}\left(W^l\right) \right)$$

  and we will make the later term to remain a constant 1 to prevent vanishing or exploding gradients

$$\frac{n^l}{2} \mathrm{Var}(W^l) = 1, \quad \forall l$$

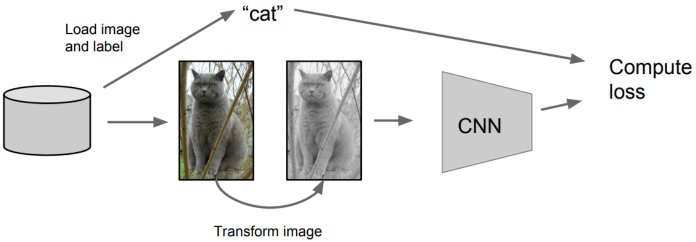
- The weights at layer l should be initialized to keep the forward variance constant 1

$$W_{ij}^l \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

- The formula for maintaining gradient variance constant 1 can be derived similarly

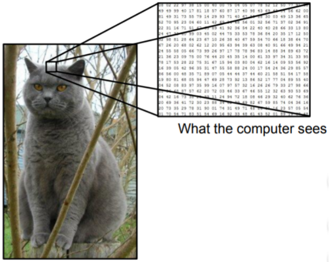
Data augmentation

- If the training set is small, one can synthesize some training samples by adding Gaussian noise to real training samples
- Domain knowledge can be used to synthesize training samples. For example, in image classification, more training images can be synthesized by translation, scaling, and rotation.



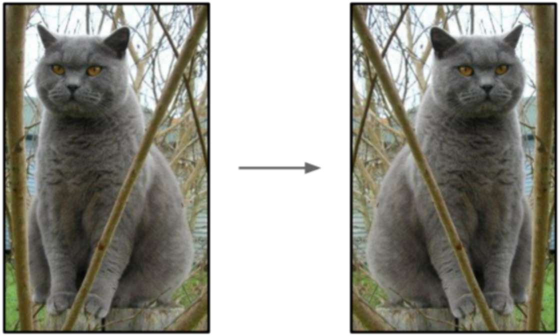
Data augmentation

- Change the pixels without changing the label
- Train on transformed data
- Very widely used in practice



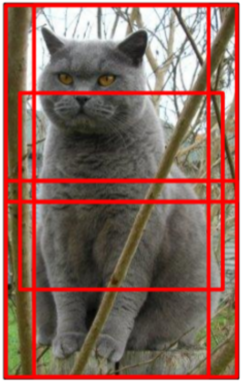
Data augmentation

- Horizontal flipping



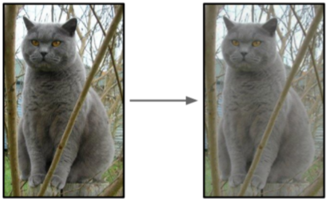
Data augmentation

- Random crops/scales
- Training for image classification networks (AlexNet/VGG/ResNet)
  - Pick random L in range $[256, 480]$
  - Resize training image, short side $= L$
  - Sample random 224×224 patch
- Testing: average a fixed set of crops
  - Resize image at 5 scales: $\{224, 256, 384, 480, 640\}$
  - For each size, use ten 224×224 crops: 4 corners $+$ center $+$ flips



Data augmentation

- Color jitter
- Simple: randomly jitter contrast
- Complex:
  - Apply PCA to all [R, G, B] pixels in training set
  - Sample a “color offset” along principal component directions
  - Add offset to all pixels of a training image



Data augmentation

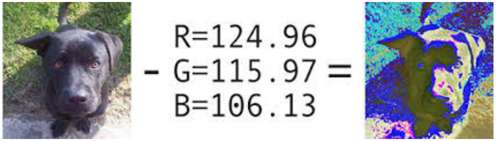
- Get creative!
- Random mix/combinations of :
  - Translation
  - Rotation
  - Stretching
  - shearing
  - lens distortions
  - etc.

Normalizing input

- If the dynamic range of one input feature is much larger than others, during training, the network will mainly adjust weights on this feature while ignore others
- We do not want to prefer one feature over others just because they differ solely measured units
- For **general feature vectors**, to avoid such difficulty, the input patterns should be shifted so that the average over the training set of each feature is zero, and then be scaled to have the same variance as 1 in each feature
- Input variables should be uncorrelated if possible
    - If inputs are uncorrelated then it is possible to solve for the value of one weight without any concern for other weights
    - With correlated inputs, one must solve for multiple weights simultaneously, which is a much harder problem
    - PCA can be used to remove linear correlations in inputs

Normalizing input

- For image data, as the three RGB channels have roughly the same magnitude, there is generally no need to normalize their magnitude to have unit variance
- Two choices:
  - Subtracting the mean pixel



  - Subtracting the mean image

Shuffling the training samples

- Networks learn the fastest from the most unexpected sample
- Shuffle the training set so that successive training examples never (rarely) belong to the same class
- Present input examples that produce a large error more frequently than examples that produce a small error

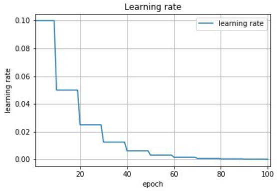
Learning rate schedules

- In general, the learning rate of a training process would gradually become smaller as the iteration number increases
- A constant learning rate schedule is feasible but highly unlikely to be used in practice
- **Time-based decay:** decreases the learning rate with the following equation

$$\mathrm{lr} = \mathrm{lr} \times \frac{\mathrm{lr}^{(0)}}{1 + \mathrm{decay} \cdot \#\mathrm{iterations}}$$

  $\mathrm{lr}^{(0)}$ is the initial learning rate and decay is a manually-set decaying rate
- **Step decay:** A typical learning rate schedule (used in AlexNet, VGG, etc.) is to drop the learning rate to the $1/10$ of the previous value

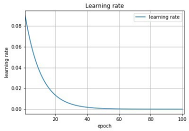
$$\mathrm{lr} = \mathrm{lr} \times 1/10 \quad \text{if } \mathrm{mod}\,(\#\mathrm{iteration}, \#\mathrm{step}) == 0$$



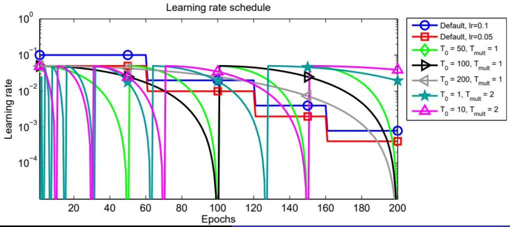
Learning rate schedules

- **Exponential decay:** Another common schedule is exponential decay (used in GoogLeNet) with hyper-parameter k (e.g., $k = 0.1$)

$$\mathrm{lr} = \mathrm{lr}^{(0)} \cdot \exp(-k \cdot \#\mathrm{iteration})$$



- **Cosine annealing with warm restart:** (1) the cosine function is used as the learning rate annealing function; (2) after every several epochs, the learning rate is restated to the initial learning rate

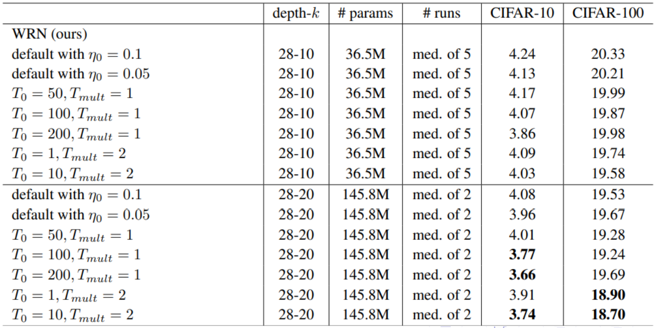


Cosine annealing with warm restart

- Learning rate schedule

$$\mathrm{lr}^{(t)} = \mathrm{lr}^i_{\min} + \frac{1}{2}\left(\mathrm{lr}^i_{\max} - \mathrm{lr}^i_{\min}\right)\left(1 + \cos\left(\frac{\#\text{iterations}}{T_i}\pi\right)\right)$$

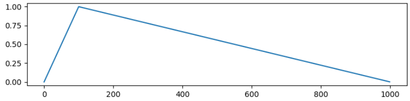
- $[\mathrm{lr}^i_{\min}, \mathrm{lr}^i_{\max}]$ is the minimal and maximal learning rates of the ith run. The learning rate restarts once T_i iterations are run
- To start with an initially small period T_i and increase it by a factor of T_{multi} at every restart



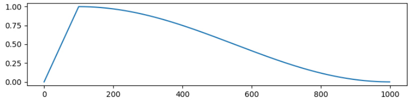
	depth-k	# params	# runs	CIFAR-10	CIFAR-100
WRN (ours)					
default with $\eta_0 = 0.1$	28-10	36.5M	med. of 5	4.24	20.33
default with $\eta_0 = 0.05$	28-10	36.5M	med. of 5	4.13	20.21
$T_0 = 50, T_{mult} = 1$	28-10	36.5M	med. of 5	4.17	19.99
$T_0 = 100, T_{mult} = 1$	28-10	36.5M	med. of 5	4.07	19.87
$T_0 = 200, T_{mult} = 1$	28-10	36.5M	med. of 5	3.86	19.98
$T_0 = 1, T_{mult} = 2$	28-10	36.5M	med. of 5	4.09	19.74
$T_0 = 10, T_{mult} = 2$	28-10	36.5M	med. of 5	4.03	19.58
default with $\eta_0 = 0.1$	28-20	145.8M	med. of 2	4.08	19.53
default with $\eta_0 = 0.05$	28-20	145.8M	med. of 2	3.96	19.67
$T_0 = 50, T_{mult} = 1$	28-20	145.8M	med. of 2	4.01	19.28
$T_0 = 100, T_{mult} = 1$	28-20	145.8M	med. of 2	**3.77**	19.24
$T_0 = 200, T_{mult} = 1$	28-20	145.8M	med. of 2	**3.66**	19.69
$T_0 = 1, T_{mult} = 2$	28-20	145.8M	med. of 2	3.91	**18.90**
$T_0 = 10, T_{mult} = 2$	28-20	145.8M	med. of 2	**3.74**	**18.70**

Learning Rate Warmup

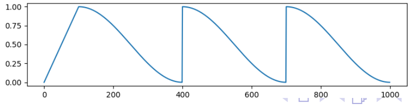
- For training certain network architectures (e.g., Transformer), a warmup stage (learning rate gradually increases before decreasing) is found helpful sometimes
- Linear schedule with a warmup phase



- Cosine schedule with a warmup phase



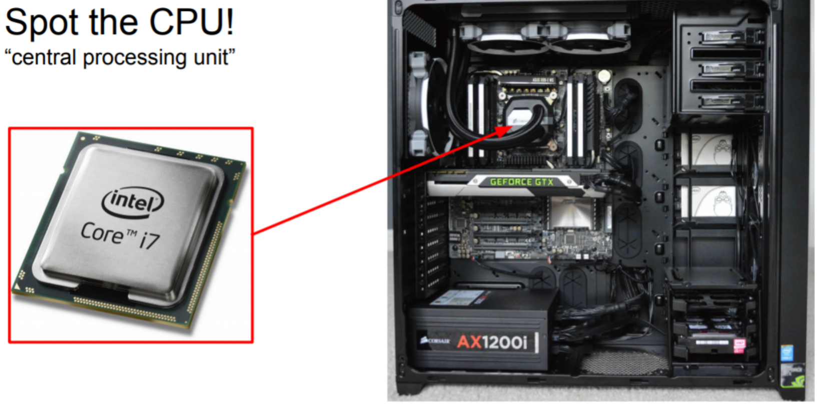
- Cosine schedule with warmup and restart



Adaptive learning rates v.s. manually designed schedules

- Note that although we have introduced algorithms that can adaptively update the learning rate, such as Adadelta, Adagrad, ADAM, engineers and researchers still manually change the learning rates with the previous mentioned learning-rate schedules

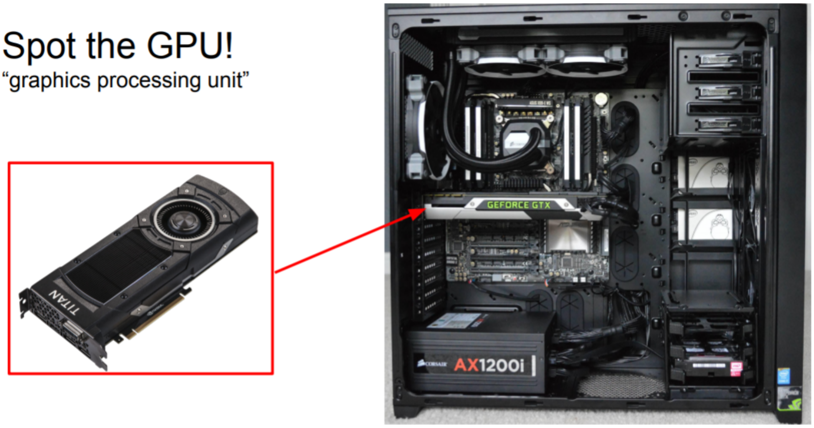
CPU


Spot the CPU!
"central processing unit"
intel
Core™ i7
GEFORCE GTX
CORSAIR
AX1200i

GPU

Spot the GPU!

"graphics processing unit"



CPU vs GPU

- CPU
  - Few, fast cores (1 - 16)
  - Good at sequential processing
- GPU
  - Many, slower cores (thousands)
  - Originally for graphics
  - Good at parallel computation

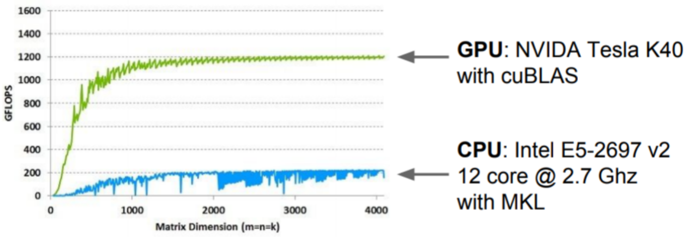
NVIDIA vs AMD

- NVIDIA is more commonly used in the research community
- cuDNN drivers by NVIDIA is the basis for all deep learning libraries
- You can implement your own layers using CUDA, the NVIDIA's programming language for parallel computing on GPU

NVIDIA VS AMD

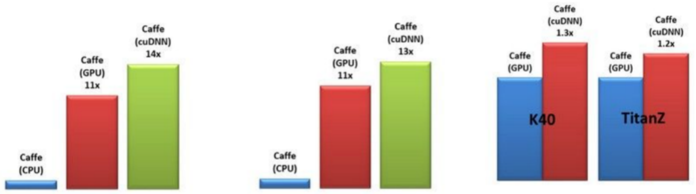
CPU vs GPU

- GPUs are really good at matrix multiplication



CPU vs GPU

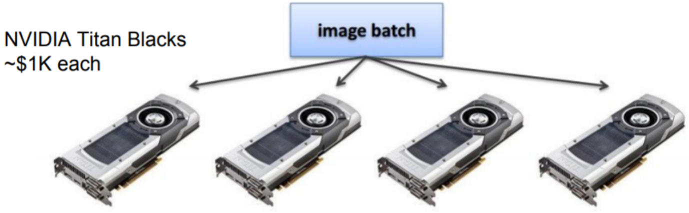
- GPUs are really good at convolution (cuDNN)



All comparisons are against a 12-core Intel E5-2679v2 CPU @ 2.4GHz running Caffe with Intel MKL 11.1.3.

GPU Training

- Even with GPUs, training can be slow
- ResNet-101: 1 week using 4 TITAN GPUs on ImageNet dataset



All comparisons are against a 12-core Intel E5-2679v2 CPU @ 2.4GHz running Caffe with Intel MKL 11.1.3.

Why need multi-GPU?

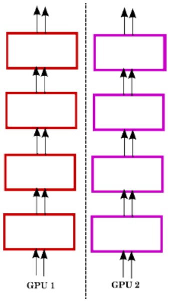
- Further speed-up
- The memory size of a single GPU is limited
  - GeForce GTX 670: 2GB
  - TITAN: 6GB
  - TITAN X: 12GB
  - Tesla K40: 12GB
  - Tesla K80: two K40
  - Tesla P100: 16 GB
  - Tesla V100: 16GB/32GB (USD $5,000)
- Train bigger models
- Data parallelism
- Model parallelism

Cost of using multi-GPU

- Synchronization
- Communication overhead
  - Communication between GPUs in the same server
  - Communication between GPU servers

Data parallelism

- The mini-batch is split across several GPUs. Each GPU is responsible computing gradients with respect to all model parameters, but does so using a subset of the samples in the mini-batch
- The model (parameters) has a complete (same) copy in each GPU
- The gradients computed from multiple GPUs are averaged to update parameters in both GPUs

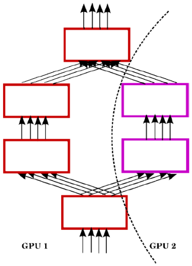


Drawbacks of data parallelism

- Limitations
  - Require considerable communication between GPUs, since must communicate both gradients and parameter values on every update step
  - Each GPU must use a large number of samples to effectively utilize the highly parallel device; thus, the mini-batch size effectively gets multiplied by the number of GPUs
- Synchronized batch normalization
  - Typical implementation of BatchNorm working on multiple devices (GPUs) is fast (with no communication overhead), it inevitably reduces the size of batch size, which potentially degenerates the performance
  - This is not a significant issue in some standard vision tasks such as ImageNet classification (as the batch size per device is usually large enough to obtain good statistics)
  - However, it will hurt the performance in some tasks that the batch size is usually very small (e.g., 1 per GPU)
  - Batch normalization across multiple GPUs is therefore needed. It requires extra communication overhead but can stabilize the training

Model parallelism

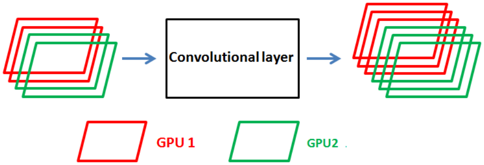
- Consist of splitting an individual network's computation across multiple GPUs
- For instance, convolutional layer with N filters can be run on two GPUs, each of which convolves its input with $N/2$ filters



The architecture is split into two columns which make easier to split computation across the two GPUs

Model parallelism

- A mini batch has the same copy in each GPU
- GPUs have to be synchronized and communicate at every layer if computing gradients in a GPU requires outputs of all the feature maps at the lower layer

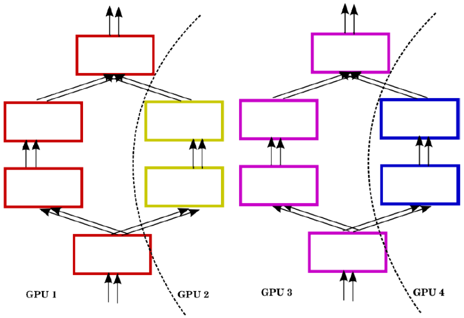


Model parallelism

- Krizhevsky et al. customized the architecture of the network to better leverage model parallelism: the architecture consists of two “columns” each allocated on one GPU
- Columns have cross connections only at one intermediate layer and at the very top fully connected layers
- While model parallelism is more difficult to implement, it has two potential advantages relative to data parallelism
    - It may require less communication bandwidth when the cross connnections involve small intermediate feature maps
    - It allows the instantiation of models that are too big for a single GPU’s memory

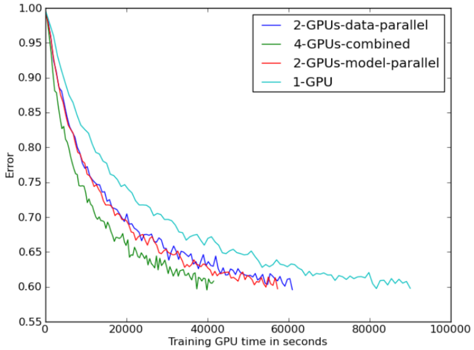
Hybrid data and model parallelism

- Data and model parallelism can be hybridized.



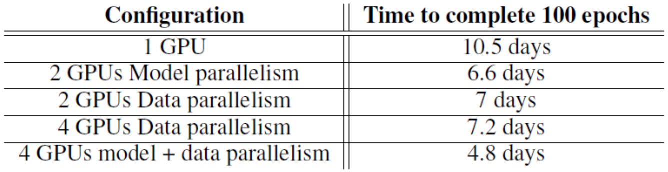
Examples of how model and data parallelism can be combined in order to make effective use of 4 GPUs

Hybrid data and model parallelism



Test error on ImageNet a function of time using different forms of parallelism. All experiments used the same mini-batch size (256) and ran for 100 epochs (here showing only the first 10 for clarity of visualization) with the same architecture and the same hyper-parameter setting as in Alex net. If plotted against number of weight updates, all these curves would almost perfectly coincide.

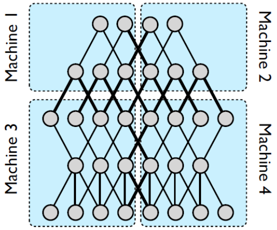
Hybrid data and model parallelism



Configuration	Time to complete 100 epochs
1 GPU	10.5 days
2 GPUs Model parallelism	6.6 days
2 GPUs Data parallelism	7 days
4 GPUs Data parallelism	7.2 days
4 GPUs model + data parallelism	4.8 days

Distributed computation with CPU cores

- Model parallelism: Only those nodes with edges that cross partition boundaries will need to have their state transmitted between machines. Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once.
- Within each partition, computation for individual nodes will the be parallelized across all available CPU cores
- It requires data synchronization and data transfer between machines during both training and inference



Distributed computation with CPU cores

- Models with local connectivity structures tend to be more amendable to extensive distribution than fully-connected structures, given their lower communication requirements
- Models with a large number of parameters or high computational demands typically benefit from access to more CPUs and memory, up to the point where communication costs dominate
- It means that the speedup cannot keep increasing with infinite number of machines
- The typical cause of less-than-ideal speedup is variance in processing times across the different machines, leading to many machines waiting for the single slowest machine to finish a given phase of computation

Reading Materials

- R. O. Duda, P. E. Hart, and D. G. Stork, "Pattern Classification," Chapter 6, 2000.
- Y. LeCun, L. Bottou, G. B. Orr, and K. Muller, "Efficient BackProp," Technical Report, 1998.
- Y. Bengio, I. J. GoodFellow and A. Courville, "Numerical Computation" in "Deep Learning", Book in preparation for MIT Press
- Glorot, X. and Bengio, Y., "Understanding the difficulty of training deep feedforward neural networks", AISTATS 2010
- K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification ", ICCV 2015
- I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
- D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015
- O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-GPU Training of ConvNets", arXiv:1312.583, 2014
- J. Dean, G. S. Corrado, R. Monga, and K. Chen, "Large Scale Distributed Deep Networks," NIPS 2012