Optimization for Training Deep Models

Xiaogang Wang

xgwang@ee.cuhk.edu.hk

February 12, 2019

Outline

1. Optimization Basics
2. Optimization of training deep neural networks
3. Multi-GPU Training

Training neural networks

- Minimize the cost function on the training set

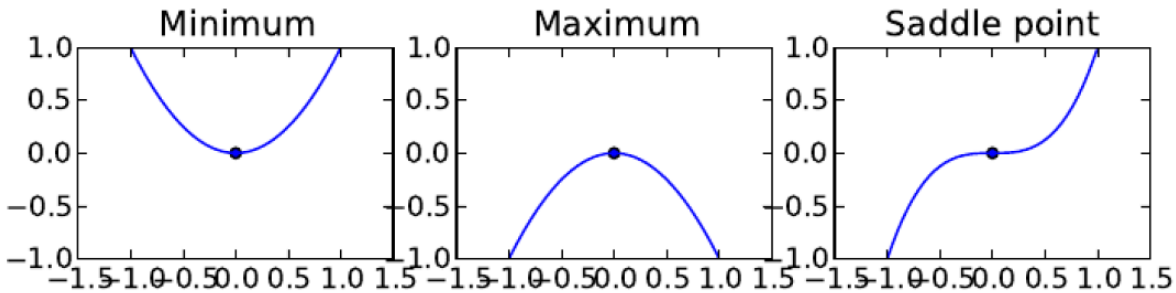
$$\theta^* = \arg\min_{\theta} J(\mathbf{X}^{(\text{train})}, \theta)$$

- Gradient descent

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

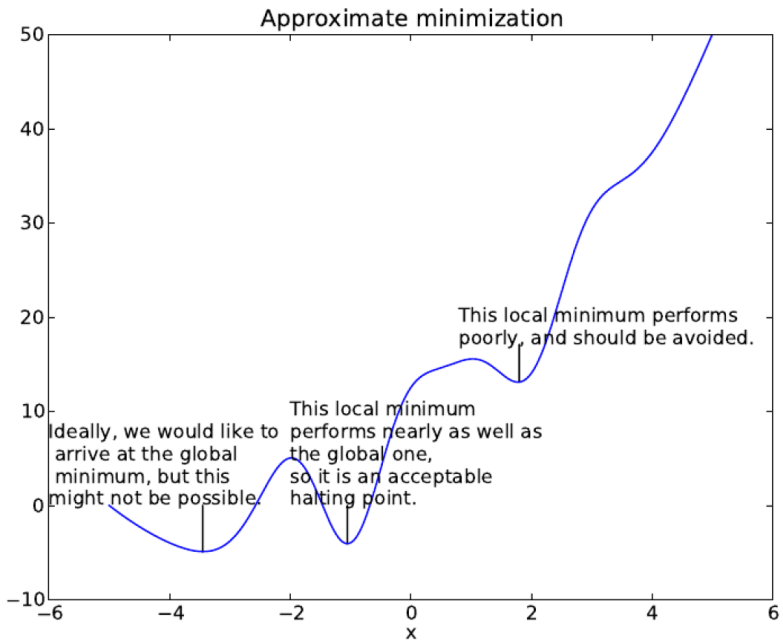
Local minimum, local maximum, and saddle points

- When $\nabla J(\theta) = 0$, the gradient provides no information about which direction to move
- Points at $\nabla J(\theta) = 0$ are known as *critical points* or *stationary points*
- A local minimum is a point where $J(\theta)$ is lower than at all neighboring points, so it is no longer possible to decrease $J(\theta)$ by making infinitesimal steps
- A local maximum is a point where $J(\theta)$ is higher than at all neighboring points, so it is no longer possible to increase $J(\theta)$ by making infinitesimal steps
- Some critical points are neither maxima nor minima. These are known as *saddle points*



Local minimum, local maximum, and saddle points

- In the context of deep learning, we optimize functions that may have many local minima that are not optimal, and many saddle points surrounded by very flat regions. All of this makes optimization very difficult, especially when the input to the function is multidimensional.
- We therefore usually settle for finding a value of J that is very low, but not necessarily minimal in any formal sense.



Jacobian matrix and Hessian matrix

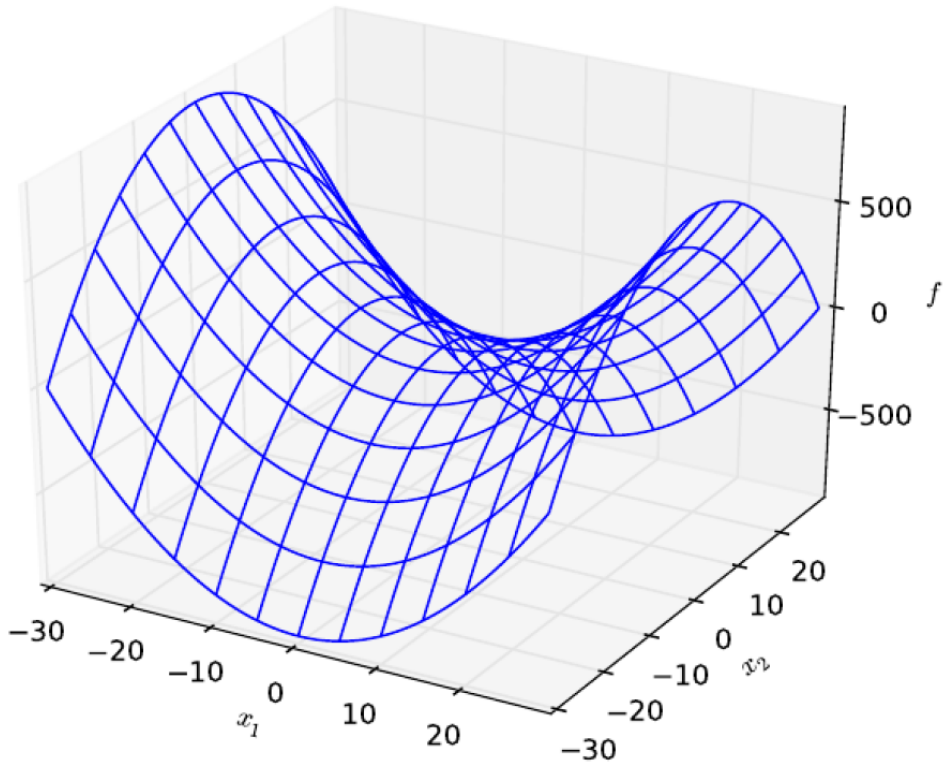
- Jacobian matrix contains all of the partial derivatives of all the elements of a vector-valued function
- Function $\mathbf{f} : \mathcal{R}^m \to \mathcal{R}^n$, then the Jacobian matrix $\mathbf{J} \in \mathcal{R}^{n \times m}$ of $\mathbf{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$
- The second derivative $\frac{\partial^2}{\partial x_i \partial x_j} f$ tells us how the first derivative will change as we vary the input. It is useful for determining whether a critical point is a local maximum, local minimum, or saddle point.
  - $f'(x) = 0$ and $f''(x) > 0$: local minimum
  - $f'(x) = 0$ and $f''(x) < 0$: local maximum
  - $f'(x) = 0$ and $f''(x) = 0$: saddle point or a part of a flat region
- Hessian matrix contains all of the second derivatives of the scalar-valued function

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x})$$

Jacobian matrix and Hessian matrix

- At a critical point, $\nabla f(\mathbf{x}) = 0$, we can examine the eigenvalues of the Hessian to determine whether the critical point is a local maximum. local minimum, or saddle point
  - When the Hessian is positive definite (all its eigenvalues are positive), the point is a local minimum: the directional second derivative in any direction must be positive
  - When the Hessian is negative definite (all its eigenvalues are negative), the point is a local maximum
  - Saddle point: at least one eigenvalue is positive and at least one eigenvalue is negative. $\mathbf{x}$ is a local maximum on one cross section of f but a local maximum on another cross section.

Saddle point



Hessian matrix

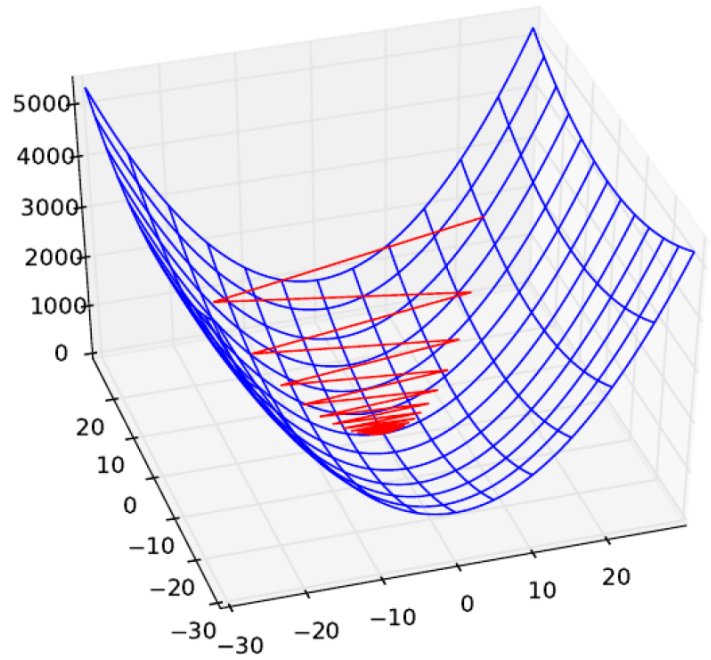
- Condition number: consider the function $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$. When $\mathbf{A} \in \mathcal{R}^{n \times n}$ has an eigenvalue decomposition, its condition number

$$\max_{i,j} |\frac{\lambda_i}{\lambda_j}|$$

  i.e. the ratio of the magnitude of the largest and smallest eigenvalue. When this number is large, matrix inversion is particularly sensitive to error in the input

- The Hessian can also be useful for understanding the performance of gradient descent. When the Hessian has a poor condition number, gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change in the derivative so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer.

Hessian matrix



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 step size 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.

Second-order optimization methods

- Gradient descent uses only the gradient and is called first-order optimization. Optimization algorithms such as Newton's method that also use the Hessian matrix are called second-order optimization algorithms.
- Newton's method on 1D function $f(x)$. The second-order Taylor expansion $f_T(x)$ of a function f around x_n is

$$f_T(x) = f_T(x_n + \Delta x) \approx f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)\Delta x^2$$

- Ideally, we want to pick a Δx such that $x_n + \Delta x$ is a stationary point of f. Solve for the Δx corresponding to the root of the expansion's derivative:

$$0 = \frac{d}{d\Delta x}\left(f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)\Delta x^2\right) = f'(x_n) + f''(x_n)\Delta x$$

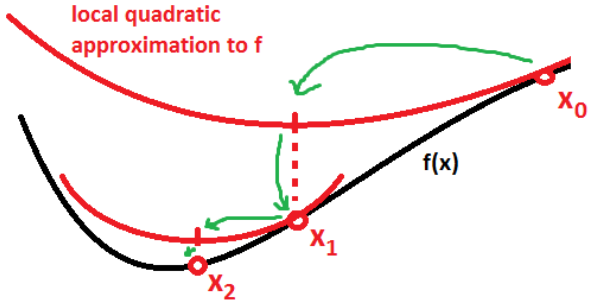
$$\Delta x = -[f''(x_n)]^{-1} f'(x_n)$$

- The update rule therefore is

$$x_{n+1} = x_n + \Delta x$$

Second-order optimization methods

- The 1D function update can be illustrated as follows



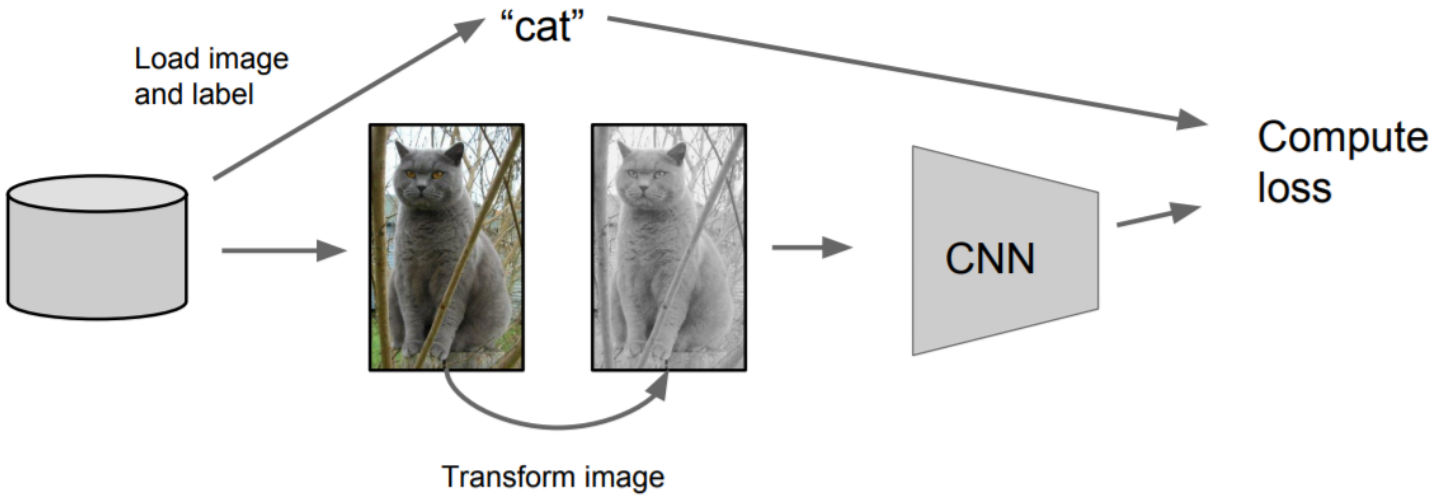
- If we extend the 1D function to a multi-dimension function. The update rule of Newton's method becomes

$$\mathbf{x}_{n+1} = \mathbf{x}_n - H(f)(\mathbf{x}_n)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_n)$$

  When the function can be locally approximated as quadratic, iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would.
- In many other fields, the dominant approach to optimization is to design optimization algorithms for a limited family of functions.
- The family of functions used in deep learning is quite complicated and complex

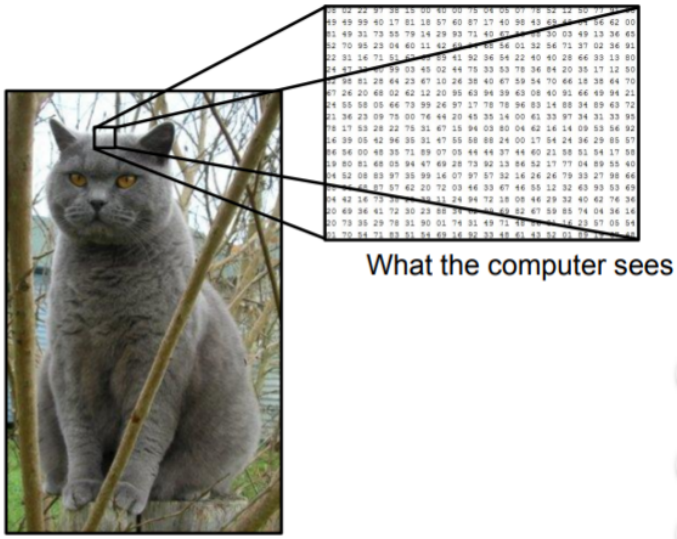
Data augmentation

- 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.



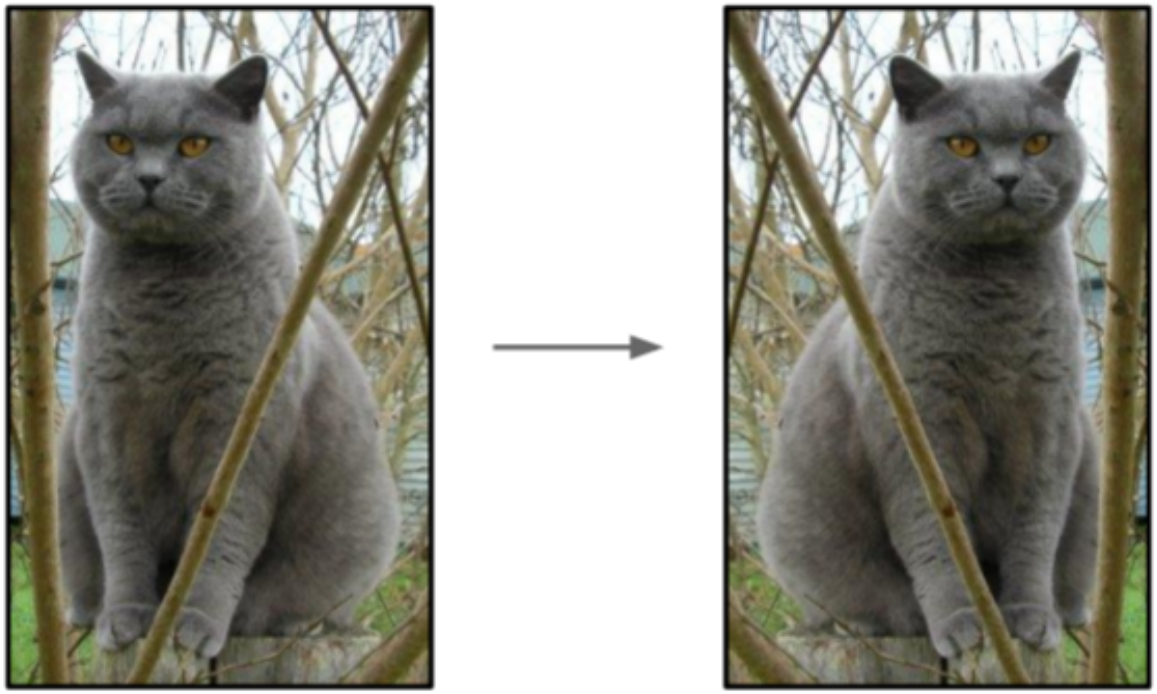
Data augmentation

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



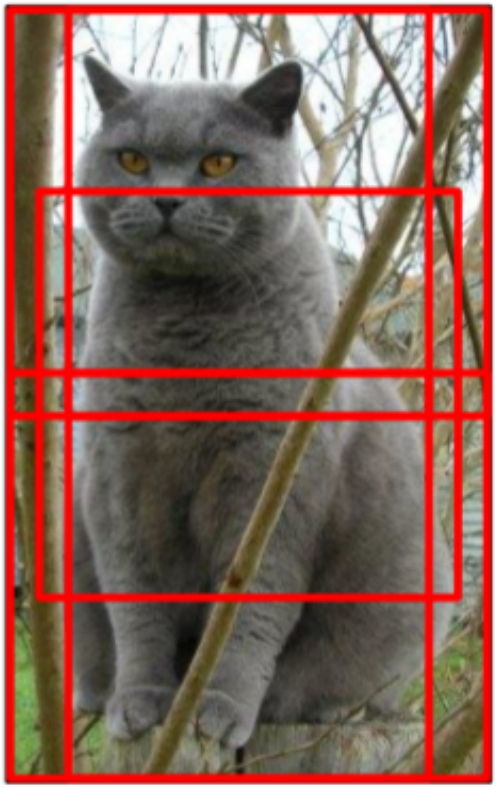
Data augmentation

- Horizontal flipping



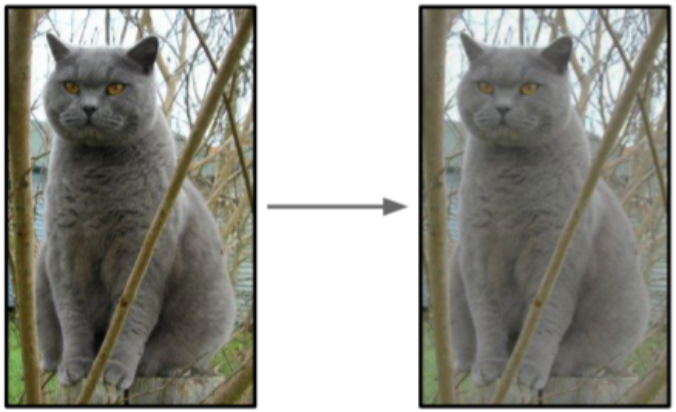
Data augmentation

- 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 10 224×224 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

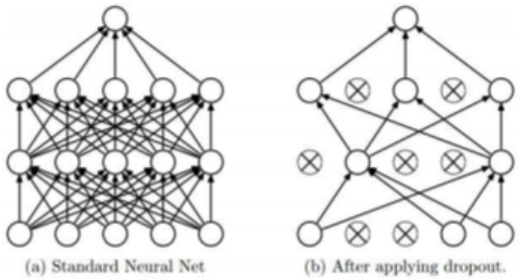
- If the dynamic range of one input feature is much larger than others, during training, the network will mainly adjust weights on this feature while ignore others
- We do not want to prefer one feature over others just because they differ solely measured units
- To avoid such difficulty, the input patterns 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

Shuffling the training samples

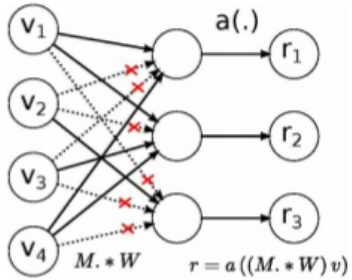
- Networks learn the fatest from the most unexpected sample
- Shuffle the training set so that successive training examples never (rarely) belong to the same class
- Present input examples that produce a large error more frequently than examples that produce a small error
  - This technique applied to data containing outliers can be disastrous because outliers can produce large errors yet should not be presented frequently

Dropout

- Radomly set some input features and the ouputs of hidden units as zero during the training process
- Feature co-adaptation: a feature is only helpful when other specific features are present
  - Because of the existence of noise and data corruption, some features or the responses of hidden nodes can be misdetected
- Dropout prevents feature co-adaptation and can significantly improve the generalization of the trained network
- Can be considered as another approach to regularization
- It can be viewed as averaging over many neural networks
- Slower convergence



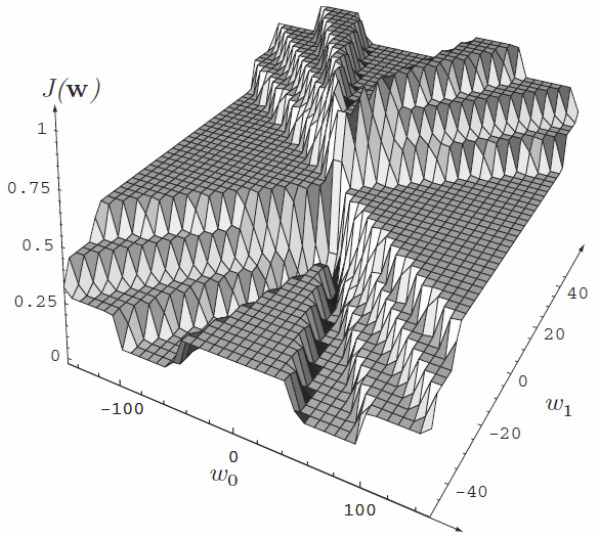
Dropout



DropConnect

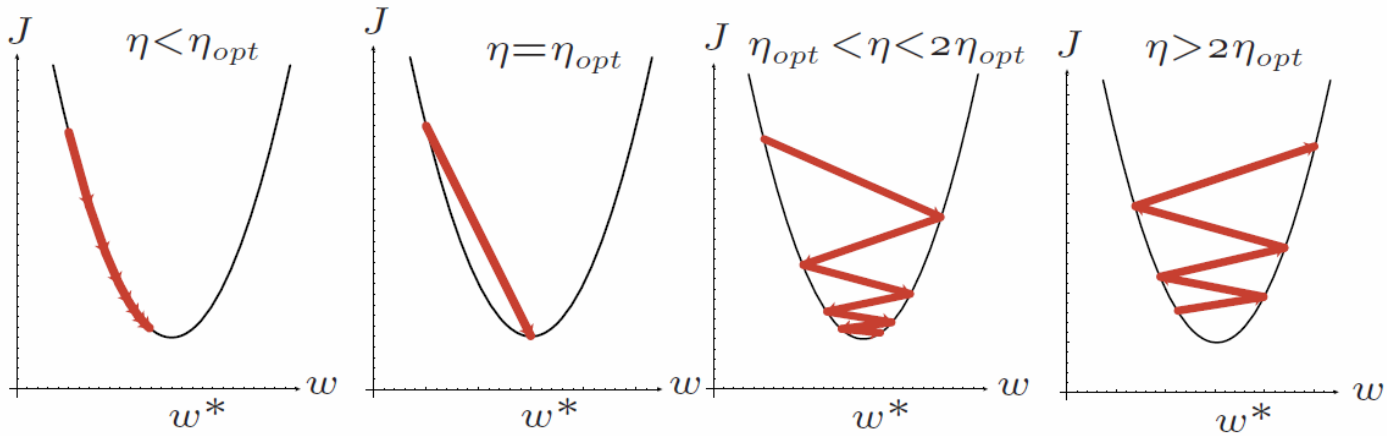
Error surfaces

- Backpropagation is based on gradient descent and tries to find the minimum point of the error surface $\mathbf{J}(\mathbf{w})$
- Generally speaking, it is unlikely to find the global minimum since the error surface is usually very complex
- Backpropagation stops at local minimum and plateaus (regions where error varies only slightly as a function of weights)
- Therefore, it is important to find a good initialization for backpropagation (through pre-training)



Learning rate

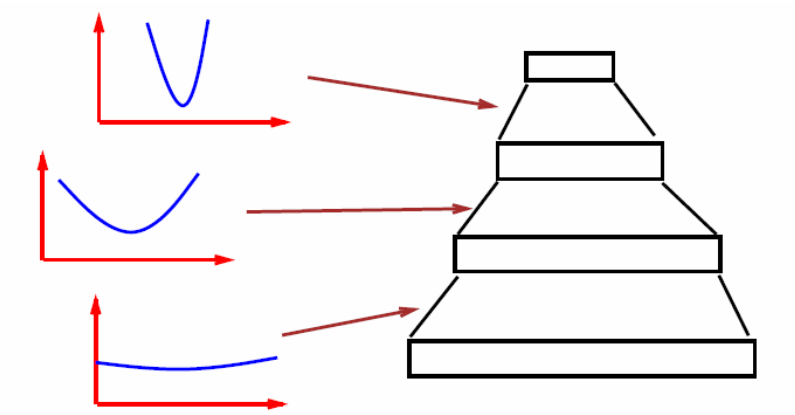
- Decrease the learning rate when the weight vector "oscillates" and increase it when the weight vector follows a steady direction
- One can choose a different learning rate for each weights, so that all the weights in the network converge roughly at the same speed



Gradient descent in a 1D quadratic criterion with different learning rates. The optimal learning rate is found by $\eta_{opt} = \left(\frac{\partial^2 J}{\partial w^2}\right)^{-1}$.

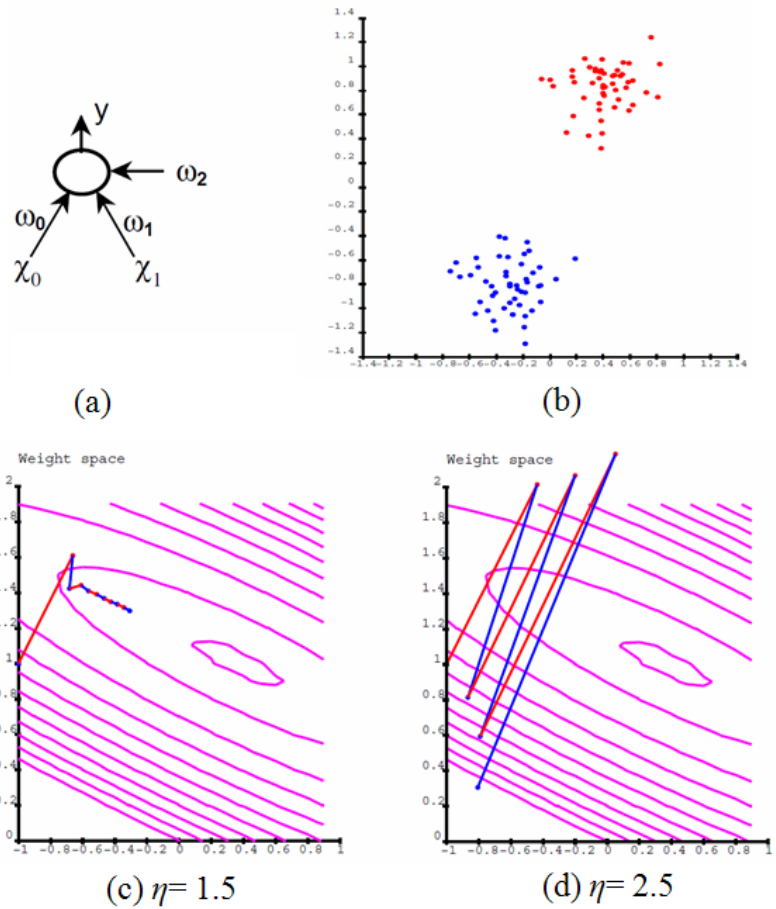
Learning rate

- Learning rates in the lower layers should generally be larger than in the higher layers, since the second derivative is often smaller in the lower layers



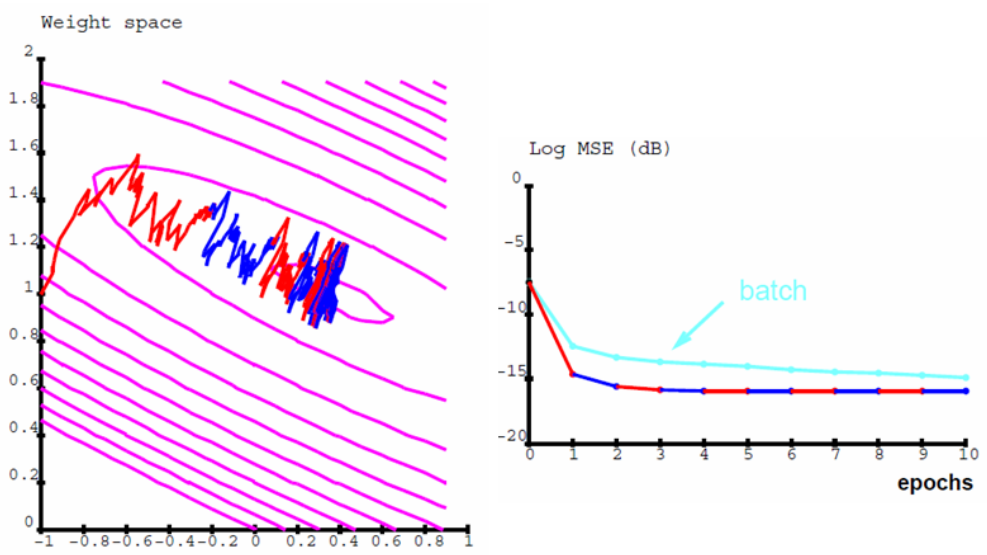
Learning rate

- Example of linear network trained in a batch mode.



Learning rate

- Stochastic learning with $\eta = 0.2$



Incorporation of momentum

- Error surfaces often have plateaus where there are “to many” weights (especially when the number of layers is large) and thus the error depends only weakly upon any one of them.
- Include some fraction α of the previous weight update in stochastic backpropagation

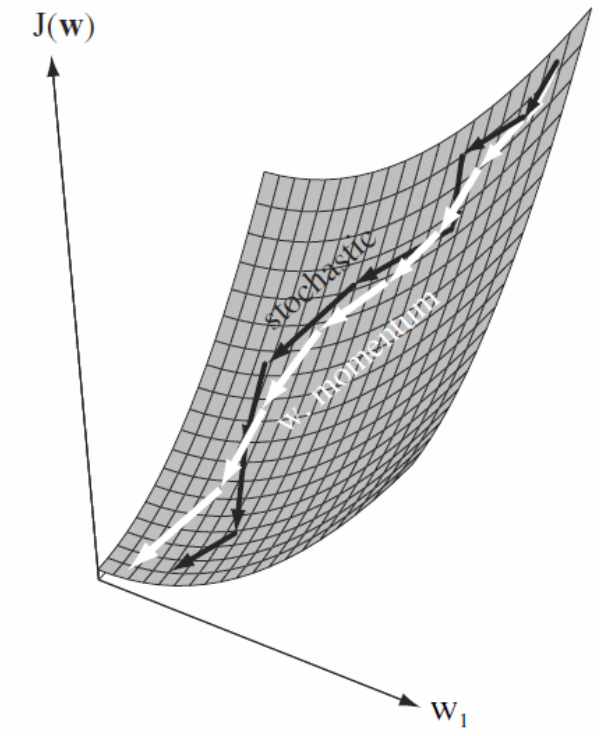
$$\mathbf{w}(m+1) = \mathbf{w}(m) + (1-\alpha)\Delta\mathbf{w}_{bp}(m) + \alpha\Delta\mathbf{w}(m)$$

where $\Delta\mathbf{w}_{bp}(m)$ is the change in $\mathbf{w}(m)$ that would be called for by the backpropagation algorithm

$$\Delta\mathbf{w}(m) = \mathbf{w}(m) - \mathbf{w}(m-1)$$

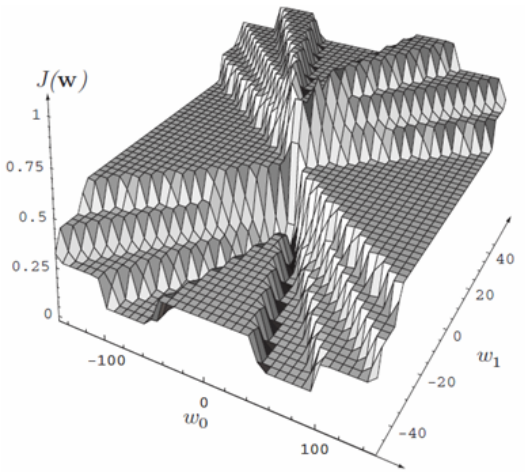
- Allow the network to learn more quickly when plateaus in the error surface exists

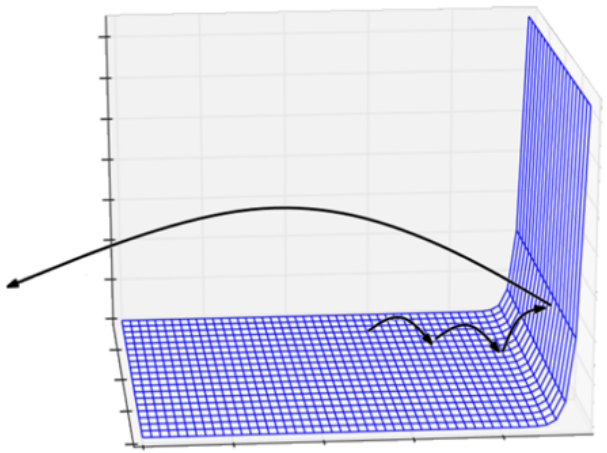
Incorporation of momentum



Plateaus and cliffs

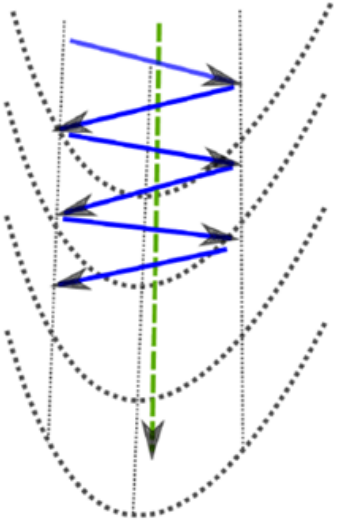
- The error surfaces of training deep neural networks include local minima, plateaus (regions where error varies only slightly as a function of weights), and cliffs (regions where the gradients rise sharply)
- Plateaus and cliffs are more important barriers to training neural networks than local minima
  - It is very difficult (or slow) to effectively update the parameters in plateaus
  - When the parameters approach a cliff region, the gradient update step can move the learner towards a very bad configuration, ruining much progress made during recent training iterations.

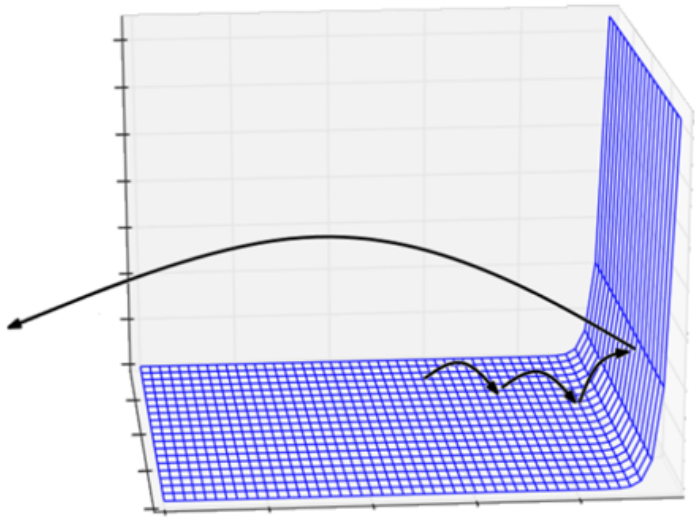




Higher-order nonlinearities

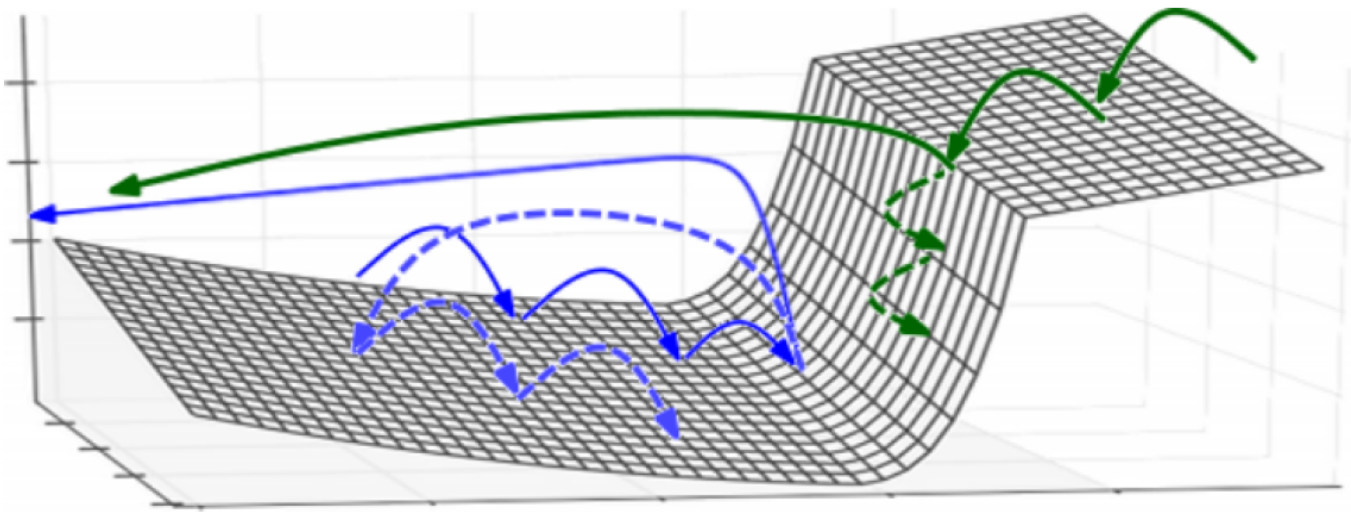
- Second-order methods or momentum assume quadratic shape around the minimum. They increase the size of steps in the low-curvature directions and decrease the sizes of steps in the high-curvature directions (the steep sides of the valley)
- When training deep models, higher order derivatives introduce a lot more non-linearity, which often does not have the nice symmetrical shapes that the second-order “valley” picture builds in our mind





Gradient clipping

- To address the presence of cliffs, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is below a threshold (which is a hyper-parameter). This helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or below.



Vanishing and exploding gradients

- Training a very deep net makes the problem even more serious, since after BP through many layers, the gradients become either very small or very large
- In very deep nets and recurrent nets, the final output is composed of a large number of non-linear transformations
- Even though each of these non-linear stages may be relatively smooth, their composition is going to be much "more non-linear", in the sense that the derivatives through the whole composition will tend to be either very small or very large, with more ups and downs

When composing many non-linearities (like the activation non-linearity in a deep or recurrent neural network), the result is highly non-linear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many ups and downs (not shown here)

Vanishing and exploding gradients

This arises because the Jacobian (matrix of derivatives) of a composition is the product of the Jacobian of each stage, i.e. if

$$f = f_T \circ f_{T-1} \circ \dots f_2 \circ f_1$$

The Jacobian matrix of derivatives of $f(x)$ with respect to its input vector $\mathbf{x}$ is

$$f' = f'_T f'_{T-1} \dots f'_2 f'_1$$

where

$$f' = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$$

and

$$f'_t = \frac{\partial f_t(\alpha_t)}{\partial \alpha_t}$$

where $\alpha_t = f_{t-1}(f_{t-1}(\dots f_2(f_1(\mathbf{x}))))$, i.e. composition has been replaced by matrix multiplication

Vanishing and exploding gradients

- In the scalar case, we can imagine that multiplying many numbers together tends to be either very large or very small
- In the special case where all the numbers in the product have the same value α, this is obvious, since α^T goes to 0 if $\alpha < 1$ and to ∞ if $\alpha > 1$ as T increases
- The more general case of non-identical numbers be understood by taking the logarithm of these numbers, considering them to be random, and computing the variance of the sum of these logarithms. Although some cancellation can happen, the variance grows with T. If those numbers are independent, it grows linearly with T, which means that the product grows roughly as e^T.
- This analysis can be generalized to the case of multiplying square matrices

Internal Covariate Shift

- The inputs to each layer are affected by the parameters of all preceeding layers, and small changes to the network parameters amplify as the network becomes deeper.
- Because of the change in the distributions of layers' inputs (called covariate shift), the layers need to continuously adapt to the new distribution
- Consider an objective function of a network,

$$J = F_2(F_1(\mathbf{u}, \Theta_1), \Theta_2)$$

where F_1 and F_2 are arbitrary transformations at different layers, and Θ_1, Θ_2 are parameters to be learned. Learning Θ_2 can be viewed as if the inputs $\mathbf{y} = F_1(\mathbf{x}, \Theta_1)$ are fed to the sub-network

$$J = F_2(\mathbf{y}, \Theta_2)$$

Internal Covariate Shift

- In order to learn Θ_2 efficiently, the distribution of $\mathbf{y}$ should remain fixed over time, so that Θ_2 does not have to readjust to compensate for the change in the distribution of $\mathbf{y}$
- One should keep $net = \mathbf{W}\mathbf{x} + \mathbf{w}_0$ away from the saturation range, where the gradients of the nonlinear activation function tend to be zero. Since *net* is affected by $\mathbf{W}$, $\mathbf{w}_0$ and the parameters of all the layers below, changes to these parameters during training will likely move many dimensions of *net* into the saturated region of the nonlinearity and slow down depth increases. This problem was once addressed by careful initialization and small learning rates.
- If we could ensure that the distribution of nonlinearity inputs remains more stable as the network trains, the optimizer would be less likely to get stuck in the satruated region, and the training would accelerate.

Batch Normalization

- A normalization step that fixes the means and variances of layer input
- Reduce the dependence of gradients on the scale of the parameters or of their initial values
- It allows to use much higher learning rates without the risk of divergence
- Make it possible to use saturating nonlinearities by preventing the network from getting stuck in the saturated modes

Batch Normalization in every layer

- Input: values of $\mathbf{x}$ over a mini-batch: $\mathcal{B} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$
- Output: $\{\mathbf{net}^{(n)} = \mathrm{BN}_{\mathbf{W},\mathbf{w}_0}(\mathbf{x}^{(n)})\}$

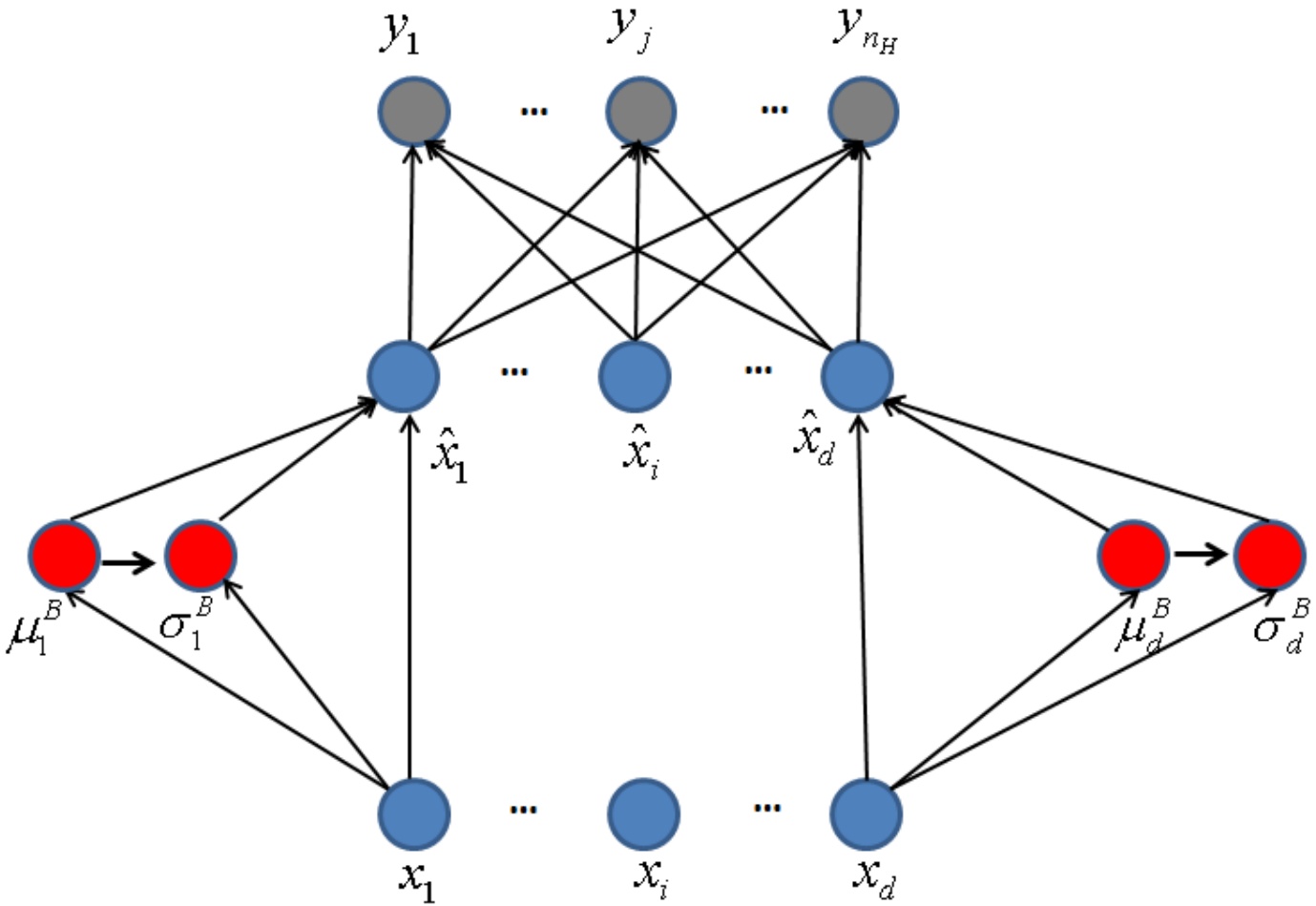
$$\mu_i^{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{n=1}^{m}\mathbf{x}_i^{(n)}$$

$$(\sigma_i^{\mathcal{B}})^2 \leftarrow \frac{1}{m}\sum_{n=1}^{m}(\mathbf{x}_i^{(n)} - \mu_i^{\mathcal{B}})^2$$

$$\hat{\mathbf{x}}_i^{(n)} \leftarrow \frac{\mathbf{x}_i^{(n)} - \mu_i^{\mathcal{B}}}{\sqrt{(\sigma_i^{\mathcal{B}})^2 + \epsilon}}$$

$$\mathbf{net}^{(n)} \leftarrow \mathbf{W}\hat{\mathbf{x}}^{(n)} + \mathbf{w}_0 \equiv \mathrm{BN}_{\mathbf{W},\mathbf{w}_0}(\mathbf{x}^{(n)})$$

Batch Normalization - BP



Batch Normalization - BP

$$\frac{\partial J}{\partial \hat{\mathbf{x}}_i^{(n)}} = \sum_j \frac{\partial J}{\partial \mathbf{net}_j^{(n)}} \cdot \mathbf{W}_{ji}$$

$$\frac{\partial J}{\partial (\sigma_i^{\mathcal{B}})^2} = \sum_{n=1}^{m} \frac{\partial J}{\partial \hat{\mathbf{x}}_i^{(n)}} \cdot (\mathbf{x}_i^{(n)} - \mu_i^{\mathcal{B}}) \cdot \frac{-1}{2}((\sigma_i^{\mathcal{B}})^2 + \epsilon)^{-3/2}$$

$$\frac{\partial J}{\partial \mu_i^{\mathcal{B}}} = \left(\sum_{n=1}^{m} \frac{\partial J}{\partial \hat{\mathbf{x}}_i^{(n)}} \cdot \frac{-1}{\sqrt{(\sigma_i^{\mathcal{B}})^2 + \epsilon}} \right) + \frac{\partial J}{\partial (\sigma_i^{\mathcal{B}})^2} \cdot \frac{\sum_{n=1}^{m} -2(\mathbf{x}_i^{(n)} - \mu_i^{\mathcal{B}})}{m}$$

$$\frac{\partial J}{\partial \mathbf{x}_i^{(n)}} = \frac{\partial J}{\partial \hat{\mathbf{x}}_i^{(n)}} \cdot \frac{1}{\sqrt{(\sigma_i^{\mathcal{B}})^2 + \epsilon}} + \frac{\partial J}{\partial (\sigma_i^{\mathcal{B}})^2} \cdot \frac{2(\mathbf{x}_i^{(n)} - \mu_i^{\mathcal{B}})}{m} + \frac{\partial J}{\partial \mu_i^{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial J}{\partial \mathbf{W}_{ji}} = \sum_{n=1}^{m} \frac{\partial J}{\partial net_j^{(n)}} \hat{\mathbf{x}}_i^{(n)}$$

$$\frac{\partial J}{\partial w_{j0}} = \sum_{n=1}^{m} \frac{\partial J}{\partial net_j^{(n)}}$$

Other Normalization Approaches

- Layer normalization (normalize the feature of each sample individually)

  [1] Jimmy Ba, *Layer Normalization*, arXiv, 2016

- Weight normalization (normalize the parameters)

  [2] Tim Salimans, *Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks*, NIPS, 2016

- Normalization propagation (normalize both the input and parameters)

  [3] *Normalization Propagation: A Parametric Technique for Removing Internal Covariate Shift in Deep Networks*, ICML, 2016

Other Gradient-based Optimizers

- The mini-batch Stochastic Gradient Descent (SGD) optimizer is the one of the most frequently used optimizer in practice
- Vanilla mini-batch gradient descent shows a few challenges that need to be addressed
  - Choosing a proper learning rate can be difficult.
  - Learning rate schedules try to adjust the learning rate during training by reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold
  - The same learning rate applies to all parameter updates. If features have different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
  - How to avoid getting trapped in their numerous suboptimal local minima. Some argue that the difficulty arises in fact not from local minima but from saddle points

Adagrad

- It adapts the learning rate to the parameters and use a different learning rate for each parameter
- Performing smaller updates for parameters associated with frequently occurring features
- Performing larger updates for parameters associated with infrequent features
- It is therefore well-suited for dealing with sparse data
- Pennington et al. used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Adagrad

- Adagrad uses a different learning rate for every parameter θ_i at every time step t
- We use g_t to denote the gradient at time step t. $g_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter at time step t

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

- The conventional SGD update for every parameter θ_i at each time step t then becomes

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

- Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each diagnoal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t, and ϵ is generally set to 10^{-8}

Adagrad

- We vectorize by a matrix-vector product $\odot$ between G_t and g_t

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate
- Most implementations use a default learning rate of 0.01 and leave it at that
- Interestingly, without the square root operation, the algorithm performs much worse
- Adagrad's main weakness is its accumulation of the squared gradients in the denominator.
- The accumulated sum keeps growing during training and the learning rate eventually becomes infinitesimally small
- At this point, the algorithm is no longer able to acquire additional knowledge

Adadelta

- Adadelta is proposed to reduce its aggressive, monotonically decreasing learning rate
- Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed time size w
- The running average $E[g^2]_t$ at time step t then depends only on the previous average and the current gradient

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

  We set γ to a similar value as the momentum term, around 0.9
- We now simply repleace the diagonal matrix G_t with past squared gradients $E[g^2]_t$ The parameter update vector is reformulated as

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} = -\frac{\eta}{\text{RMS}[g]_t} g_t, \text{ and } \theta_{t+1} = \theta_t + \Delta\theta_t$$

Adadelta

- The authors note that the units in the previous update do not match the hypothetical units as the parameter
- To resolve the issue, they first define another exponentially decaying average as squared parameter updates

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

- The root mean squared error of parameter updates is thus

$$\text{RMS}[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

- Since $\text{RMS}[\Delta\theta^2]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step $\text{RMS}[\Delta\theta^2]_{t-1}$.
- Replacing learning rate η in the previous update rule with $\text{RMS}[\Delta\theta^2]_{t-1}$ yields the Adadelta update rule

$\Delta\theta_t = -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t$, and the update rule is $\theta_{t+1} = \theta_t + \Delta\theta_t$

RMSprop

- RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates
- RMSprop is developed by Geoff Hinton in a Coursera course
- RMSprop in fact is identical to the first update vector of Adadelta that we derived above

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\text{Update rule: } \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}$$

- Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

Adam

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter
- In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- m_t and v_t are estimates of the first moment (mean) and the second moment (uncentered variance) of the gradients
- m_0 and v_0 are initialized as vectors of 0's. However, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small

Adam

- They counteract these biases by computing bias-corrected first and second moment estimates

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

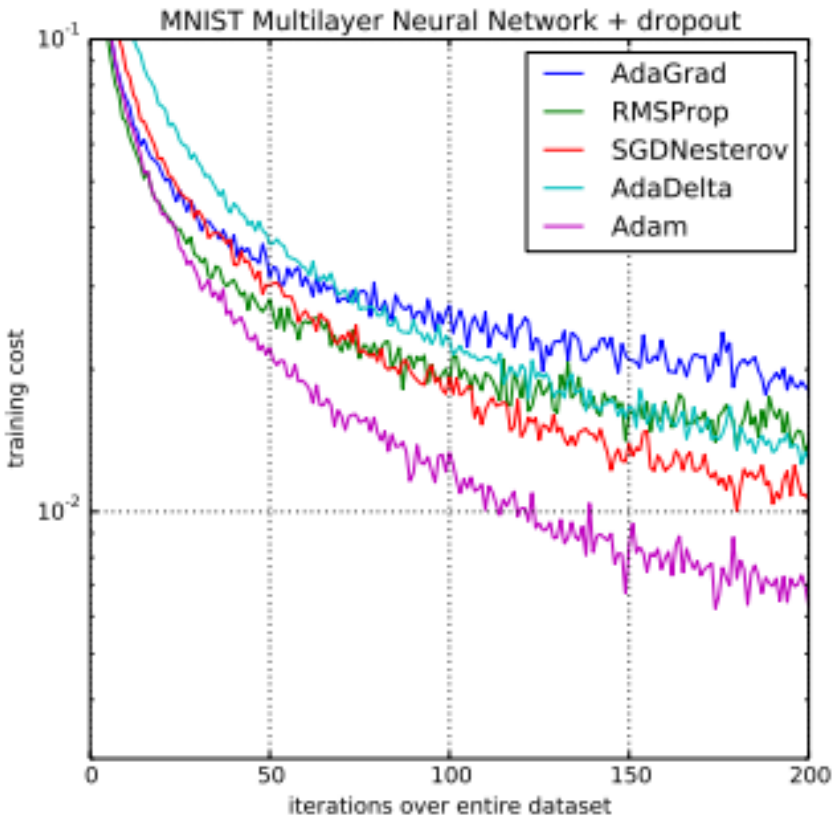
- The Adam update rule is therefore defined as

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ . Almost no one ever changes these values.
- They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

Adam

- The performance comparison on MINST classification with different optimizers



CPU

Spot the CPU!

"central processing unit"

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

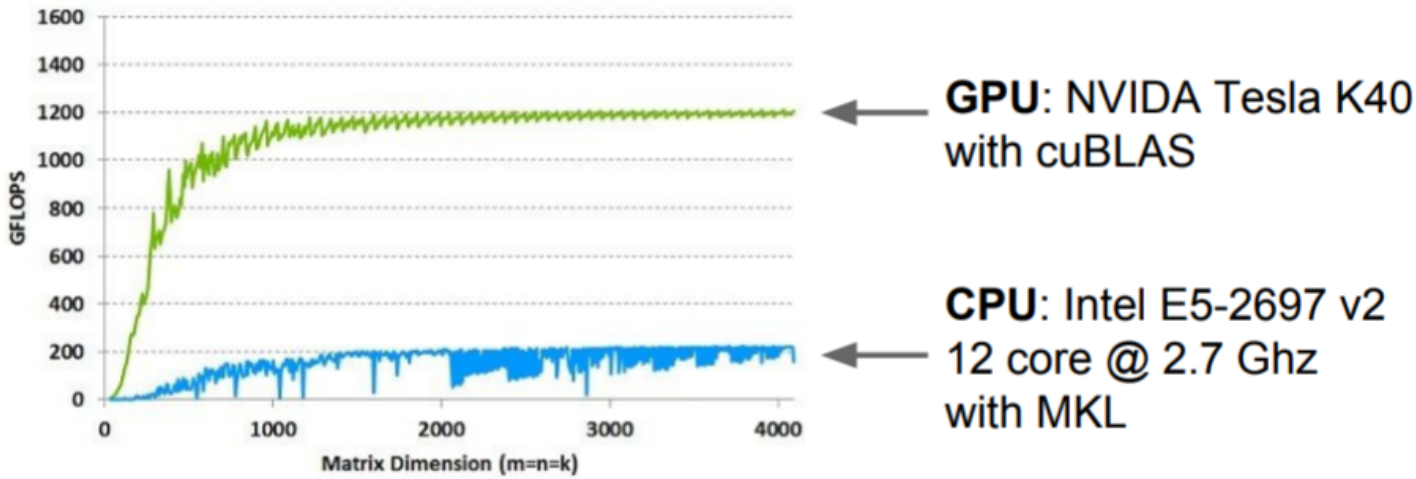
NVIDIA vs AMD

- 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

VS

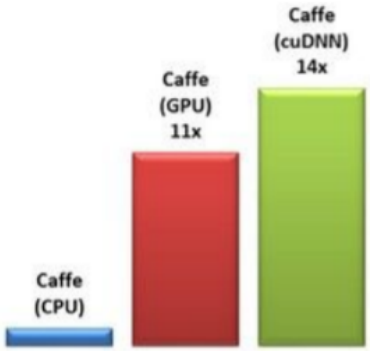
CPU vs GPU

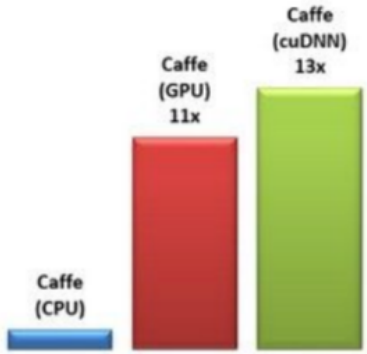
- GPUs are really good at matrix multiplication

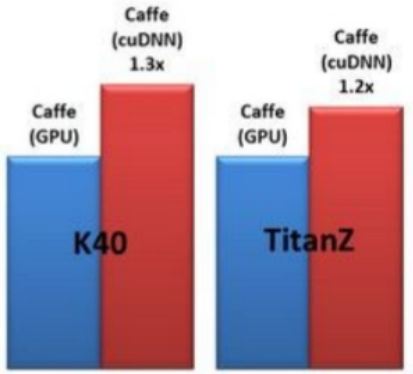


CPU vs GPU

- GPUs are really good at convolution (cuDNN)



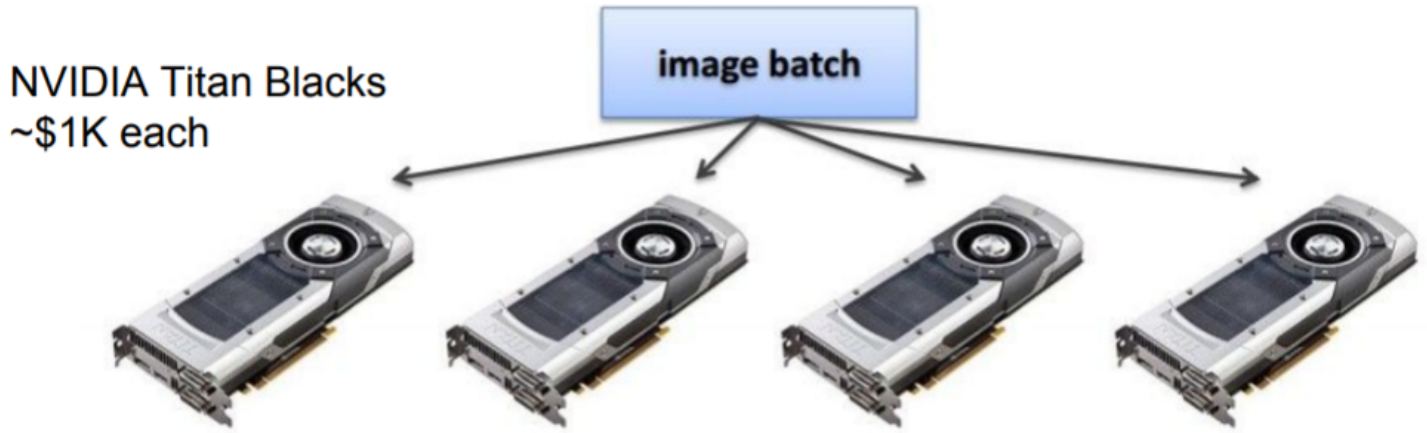




All comparisons are against a 12-core Intel E5-2679v2 CPU @ 2.4GHz running Caffe with Intel MKL 11.1.3.

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?

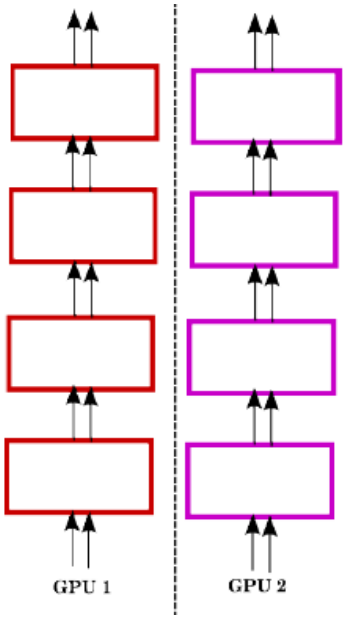
- Further speed-up
- The memory size of a single GPU is limited
  - GeForce GTX 670: 2GB
  - TITAN: 6GB
  - TITAN X: 12GB
  - Tesla K40: 12GB
  - Tesla K80: two K40
  - Tesla P100: 16 GB
  - Tesla V100: 16GB/32GB (USD $10,000)
- Train bigger models
- Data parallelism
- Model parallelism

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

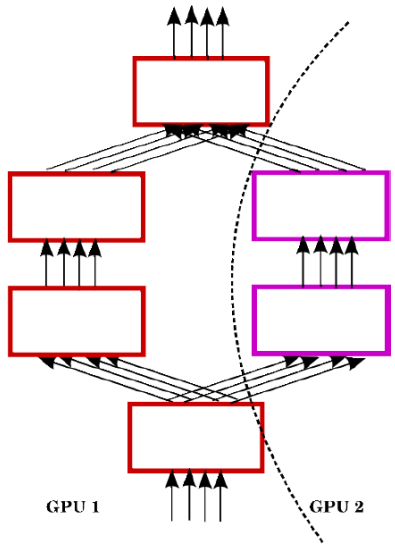


Drawbacks of data parallelism

- Require considerable communication between GPUs, since each GPU 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

Model parallelism

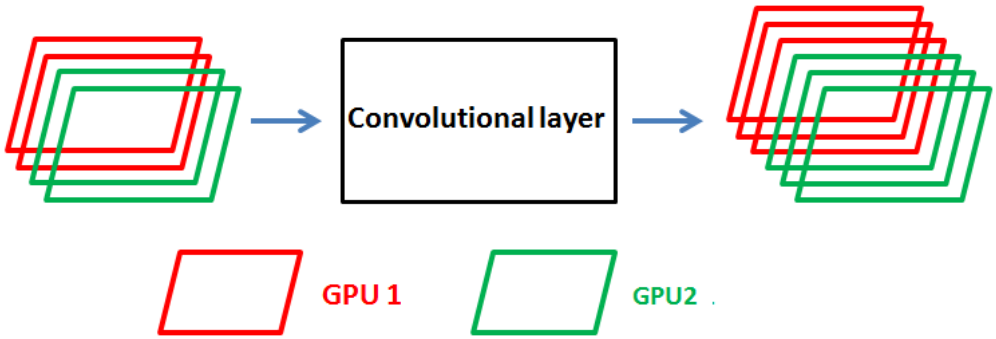
- Consist of splitting an individual network's computation across multiple GPUs
- For instance, convolutional layer with N filters can be run on two GPUs, each of which convolves its input with $N/2$ filters



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

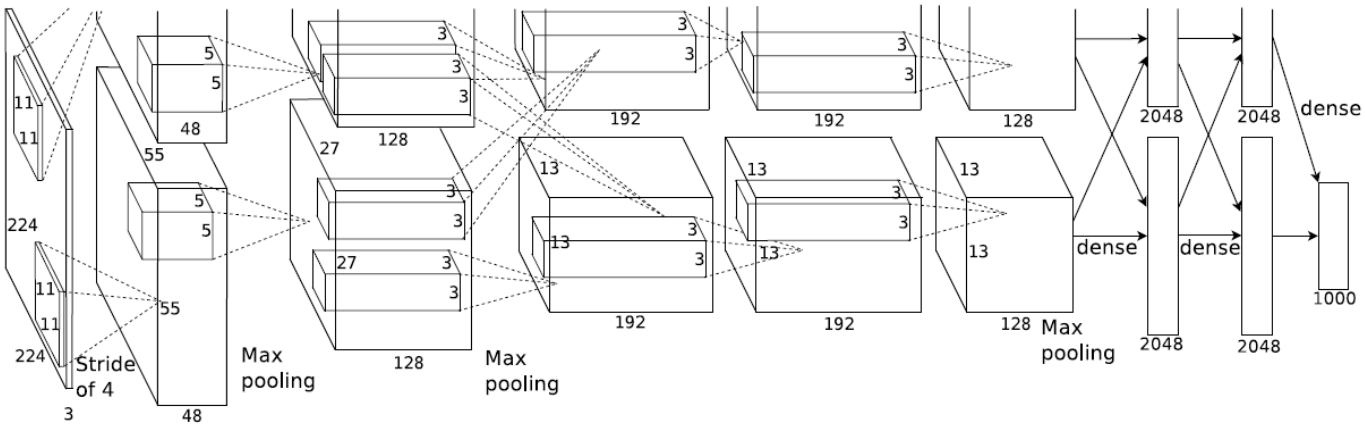
Model parallelism

- 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



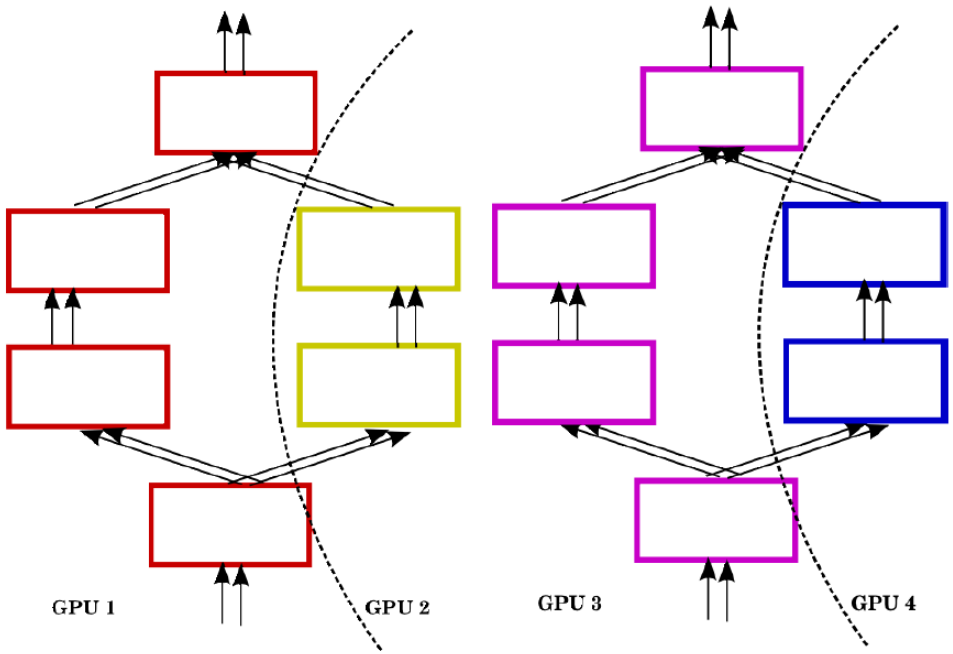
A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet classification with deep convolutional neural networks,” in NIPS, 2012.

Model parallelism

- 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

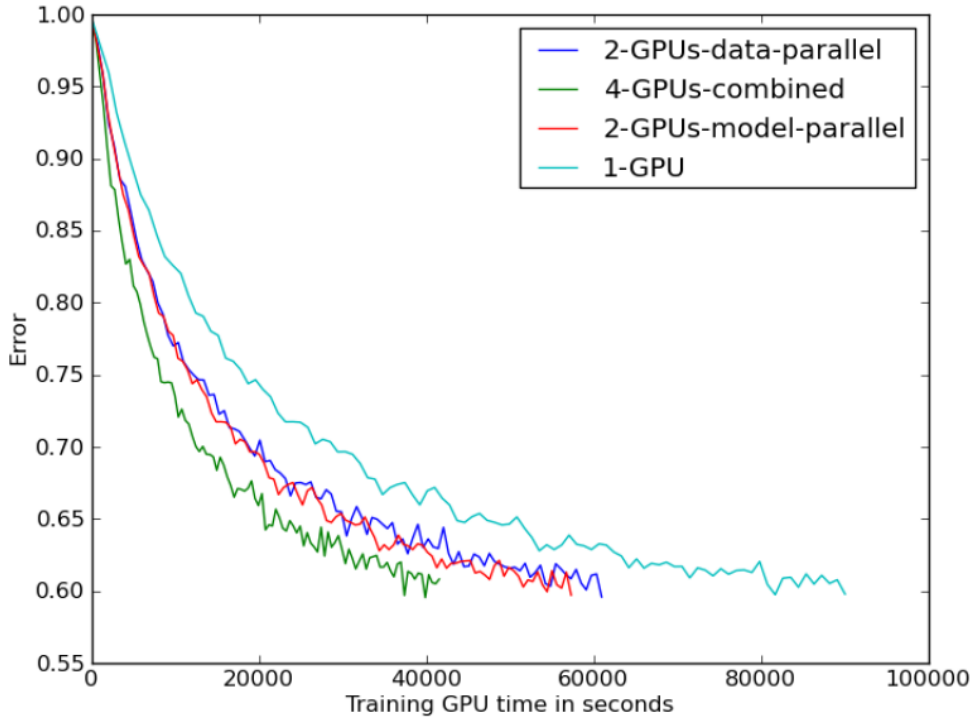
Hybrid data and model parallelism

- 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



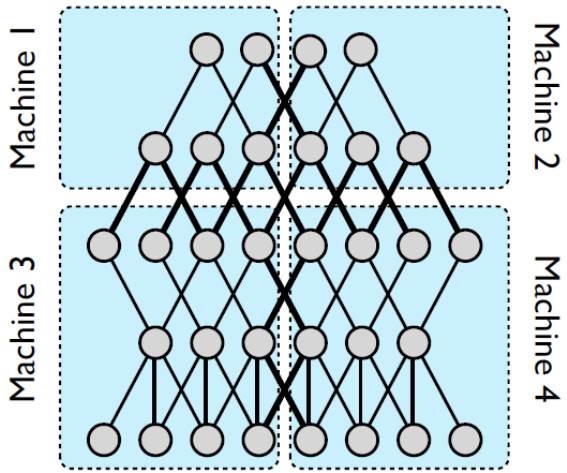
Test error on ImageNet a function of time using different forms of parallelism. All experiments used the same mini-batch size (256) and ran for 100 epochs (here showing only the first 10 for clarity of visualization) with the same architecture and the same hyper-parameter setting as in Alex net. If plotted 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
- Y. Bengio, I. J. GoodFellow and A. Courville, “Numerical Optimization” in “Deep Learning”, Book in preparation for MIT Press
- 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