Optimization for Training Deep Models

Xiaogang Wang

xgwang@ee.cuhk.edu.hk

March 22, 2015

Outline

Training neural networks

- Minimize the cost function on the training set

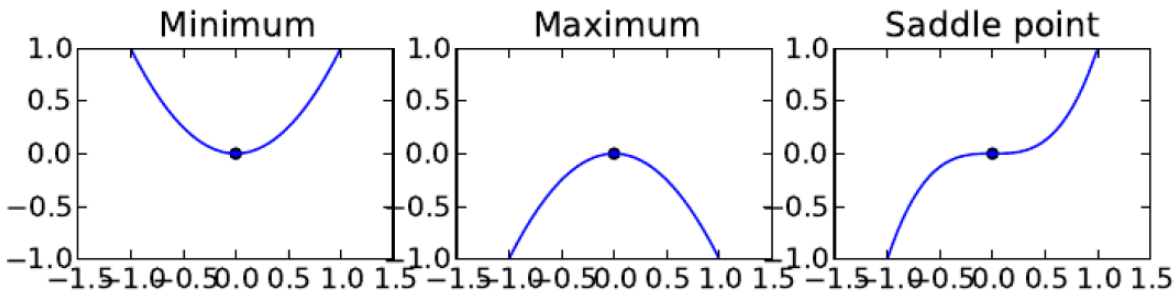
$$\theta^* = \arg\min_{\theta} J(\mathbf{X}^{(\text{train})}, \theta)$$

- Gradient descent

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

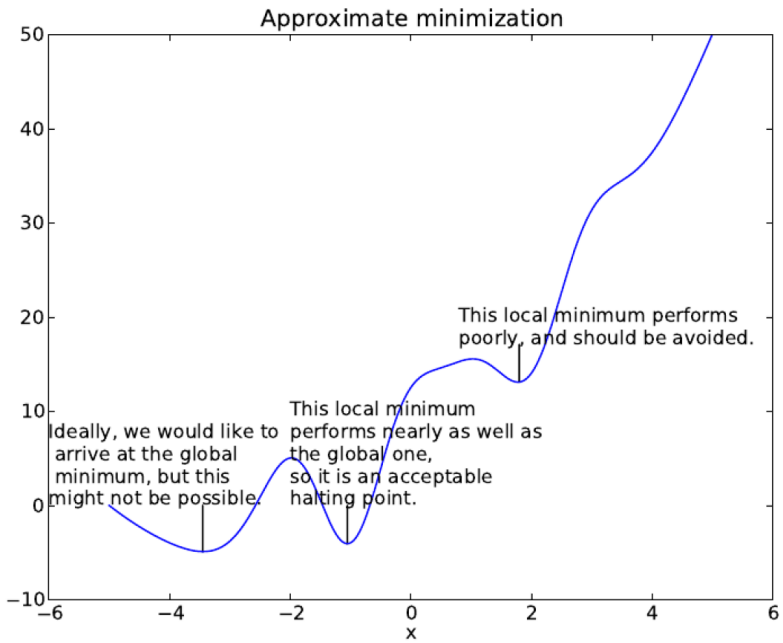
Local minimum, local maximum, and saddle points

- 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

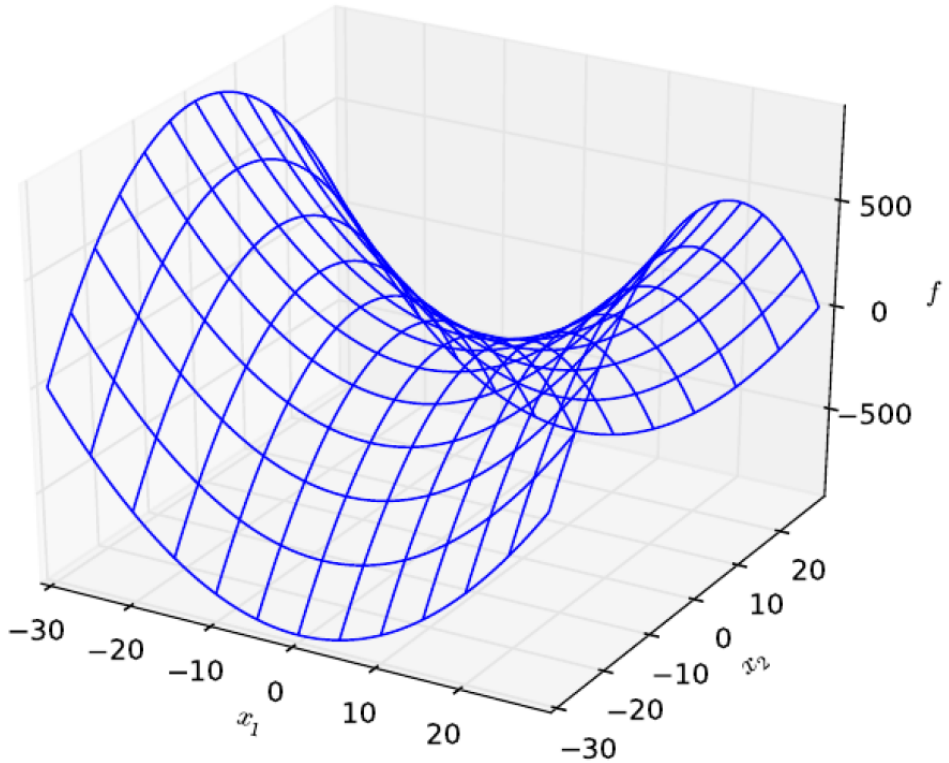
- Jacobian matrix contains all of the partial derivatives of all the elements of a vector-valued function
- Function $\mathbf{f} : \mathcal{R}^m \rightarrow \mathcal{R}^n$, then the Jacobian matrix $\mathbf{J} \in \mathcal{R}^{n \times m}$ of $\mathbf{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$
- 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 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.

Jacobian matrix and Hessian matrix



Hessian matrix

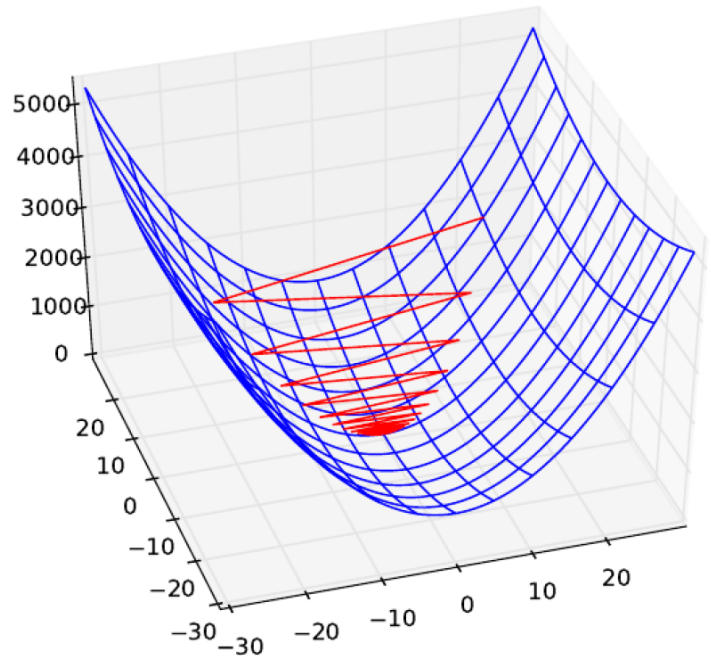
- Condition number: consider the function $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$. When $\mathbf{A} \in \mathcal{R}^{n \times n}$ has an eigenvalue decomposition, its condition number

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

  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.
- Update with Newton's method

$$\mathbf{x}^* = \mathbf{x}_0 - H(f)(\mathbf{x}_0)^{-1}\nabla_{\mathbf{x}} f(\mathbf{x}_0)$$

  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

Stochastic gradient descent

- Given n training samples, our target function can be expressed as

$$J(\mathbf{w}) = \sum_{p=1}^{n} J_p(\mathbf{w})$$

- Batch gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \sum_{p=1}^{n} \nabla J_p(\mathbf{w})$$

- In some cases, evaluating the sum-gradient may be computationally expensive. Stochastic gradient descent samples a subset of summand functions at every step. This is very effective in the case of large-scale machine learning problems. In stochastic gradient descent, the true gradient of $J(\mathbf{w})$ is approximated by a gradient at a single example (or a mini-batch of samples):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J_p(\mathbf{w})$$

Stochastic backpropagation

Algorithm 1 (Stochastic backpropagation)

```
1 begin initialize  network topology (# hidden units), w, criterion θ, η, m ← 0
2   do m ← m + 1
3       x^m ← randomly chosen pattern
4       w_ij ← w_ij + η δ_j x_i;   w_jk ← w_jk + η δ_k y_j
5   until ∇J(w) < θ
6 return w
7 end
```

- In stochastic training, a weight update may reduce the error on the single pattern being presented, yet increase the error on the full training set.

Mini-batch based stochastic gradient descent

- Divide the training set into mini-batches.
- In each epoch, randomly permute mini-batches and take a mini-batch sequentially to approximate the gradient
  - One epoch corresponds to a single presentations of all patterns in the training set
- The estimated gradient at each iteration is more reliable
- Start with a small batch size and increase the size as training proceeds

Batch backpropagation

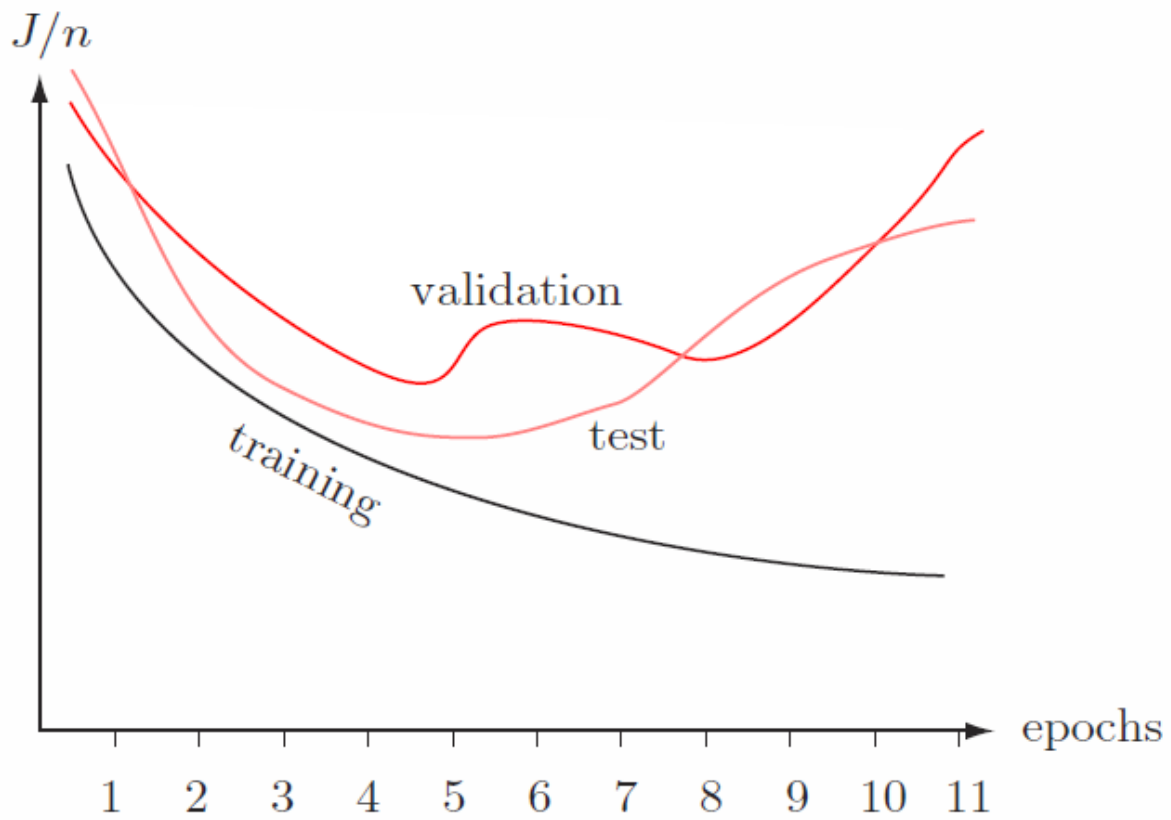
Algorithm 2 (Batch backpropagation)

1 **begin initialize** network topology (# hidden units), $\mathbf{w}$, criterion θ, η, $r \leftarrow 0$
2 **do** $r \leftarrow r + 1$ (increment epoch)
3 $m \leftarrow 0$; $\Delta w_{ij} \leftarrow 0$; $\Delta w_{jk} \leftarrow 0$
4 **do** $m \leftarrow m + 1$
5 $\mathbf{x}^m \leftarrow$ select pattern
6 $\Delta w_{ij} \leftarrow \Delta w_{ij} + \eta \delta_j x_i$; $\Delta w_{jk} \leftarrow \Delta w_{jk} + \eta \delta_k y_j$
7 **until** $m = n$
8 $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$; $w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$
9 **until** $\nabla J(\mathbf{w}) < \theta$
10 **return** w
11 **end**

Summary

- Stochastic learning
  - Estimate of the gradient is noisy, and the weights may not move precisely down the gradient at each iteration
  - Faster than batch learning, especially when training data has redundance
  - Noise often results in better solutions
  - The weights fluctuate and it may not fully converge to a local minimum
- Batch learning
  - Conditions of convergence are well understood
  - Some acceleration techniques only operate in batch learning
  - Theoretical analysis of the weight dynamics and convergence rates are simpler

Plot learning curves on the training and validation sets



Plot the average error per pattern (i.e. $1/n \sum_p J_p$) versus the number of epochs.

Learning curve on the training set

- The average training error typically decreases with the number of epochs and reaches an asymptotic value
- This asymptotic value could be high if **underfitting** happens. The reasons could be
  - The classification problem is difficult (Bayes error is high) and there are a large number of training samples
  - The expressive power of the network is not enough (the numbers of weights, layers and nodes in each layer)
  - Bad initialization and get stuck at local minimum (pre-training for better initialization)
- If the learning rate is low, the training error tends to decrease monotonically, but converges slowly. If the learning rate is high, the training error may oscillate.

Learning curve on the test and validation set

- The average error on the validation or test set is virtually always higher than on the training set. It could increase or oscillate when **overfitting** happen. The reasons could be
  - Training samples are not enough
  - The expressive power of the network is too high
  - Bad initialization and get stuck at local minimum (pre-training for better initialization)
- Stop training at a minimum of the error on the validation set

Data augmentation

- If the training set is small, one call 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.

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

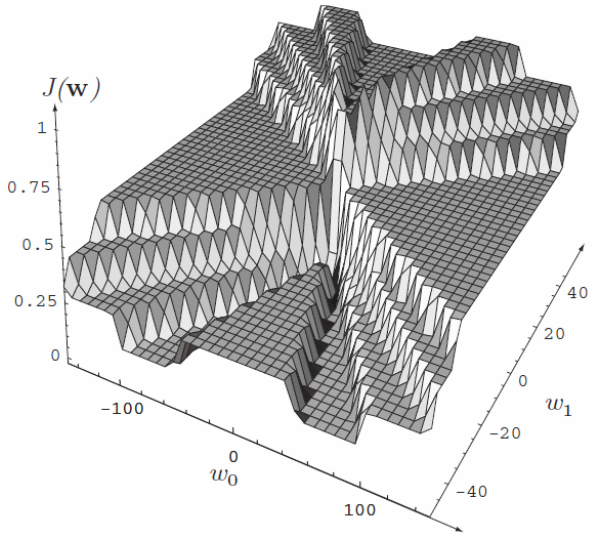
- Networks learn the fatest from the most unexpected sample
- Shuffle the training set so that successive training examples never (rarely) belong to the same class
- Present input examples that produce a large error more frequently than examples that produce a small error
  - This technique applied to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

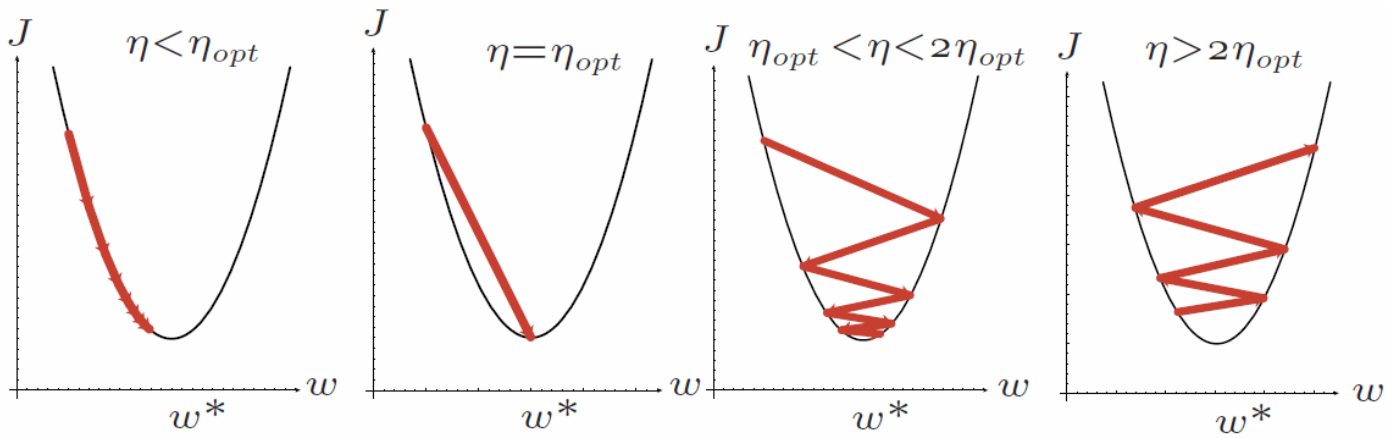
Error surfaces

- 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

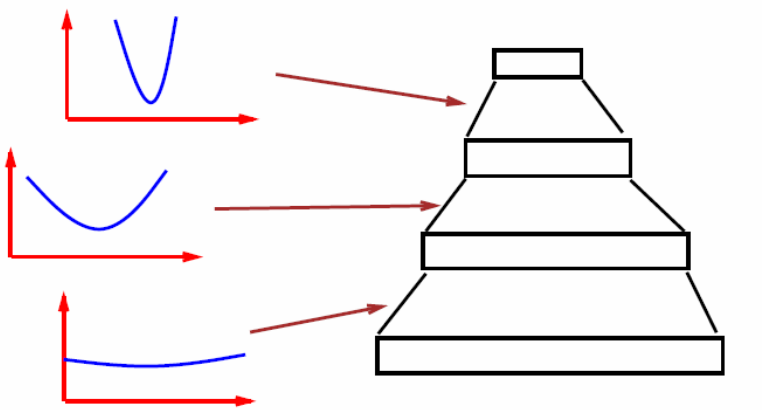
- Decrease the learning rate when the weight vector "oscillates" and increase it when the weight vector follows a steady direction
- One can choose a different learning rate for each weights, so that all the weights in the network converge roughly at the same speed



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

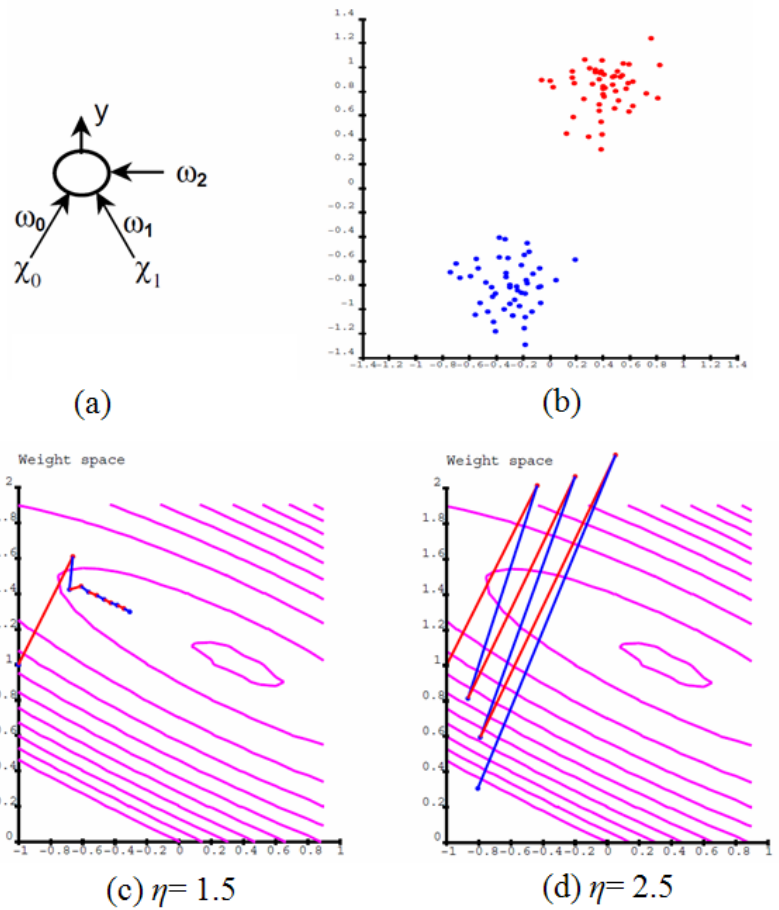
Learning rate

- 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



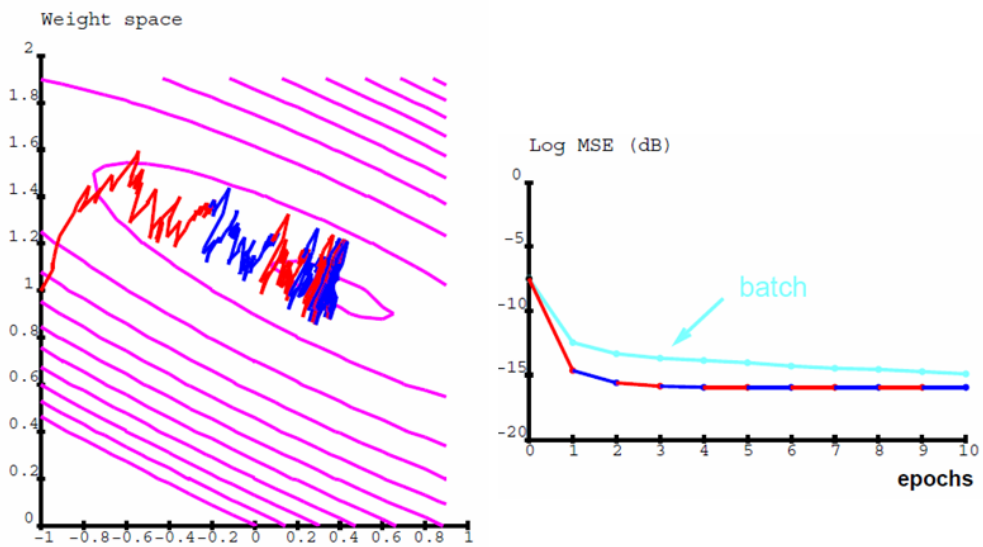
Learning rate

- Example of linear network trained in a batch mode.



Learning rate

- Stochastic learning with $\eta = 0.2$



Incorpotation 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

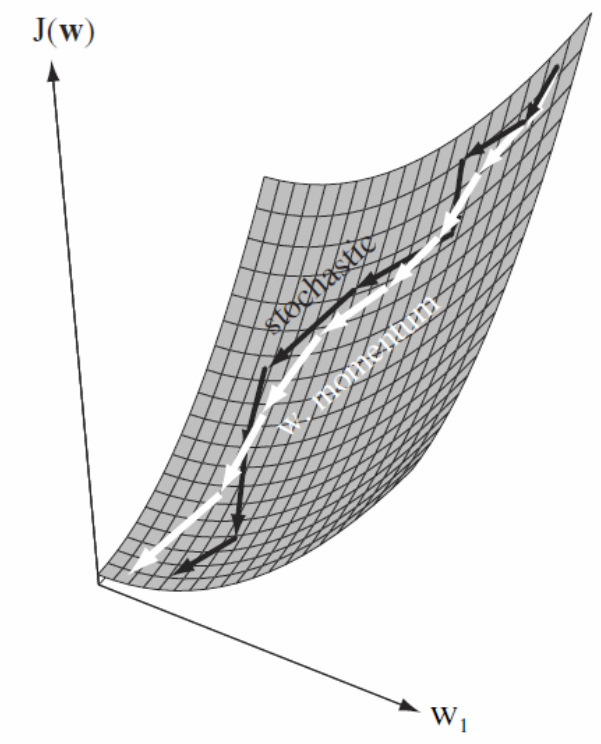
$$\mathbf{w}(m+1) = \mathbf{w}(m) + (1-\alpha)\Delta\mathbf{w}_{bp}(m) + \alpha\Delta\mathbf{w}(m-1)$$

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

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

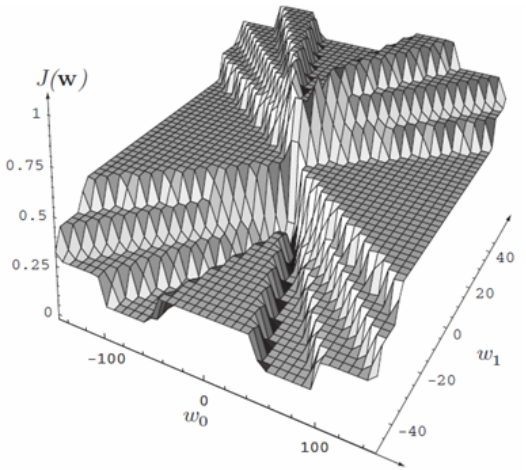
- Allow the network to learn more quickly when plateaus in the error surface exists

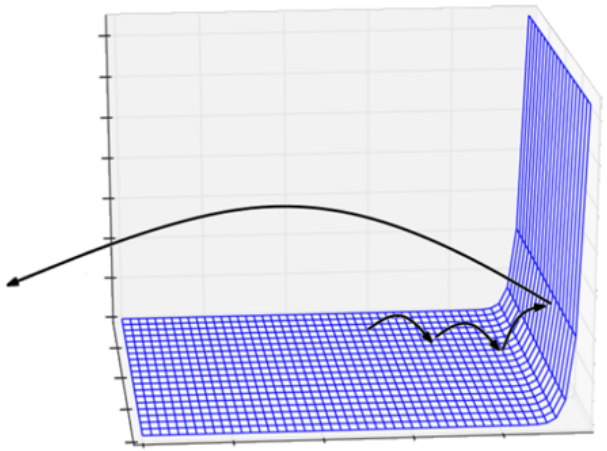
Incorpotation of momentum



Plateaus and cliffs

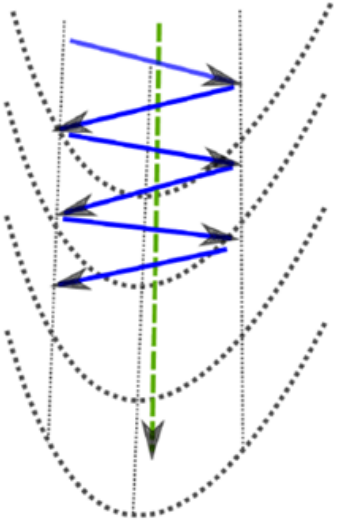
- The error surfaces of training deep neural networks include local minima, plateaus (regions where error varies only slightly as a function of weights), and cliffs (regions where the gradients rise sharply)
- Plateaus and cliffs are more important barriers to training neural networks than local minima
  - 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.

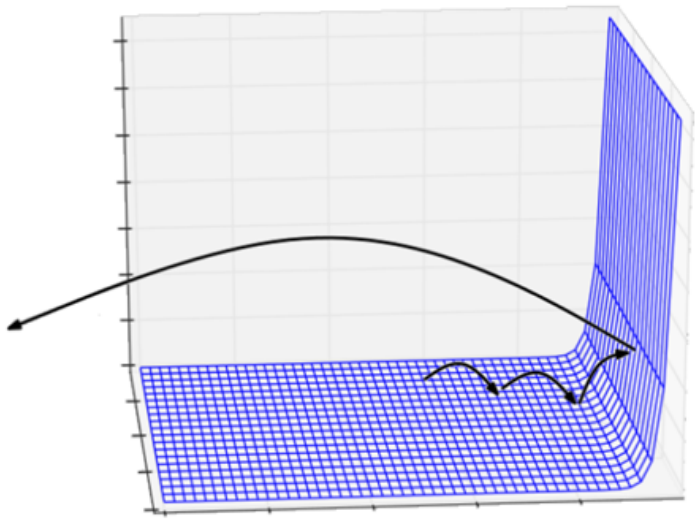




Higher-order nonlinearities

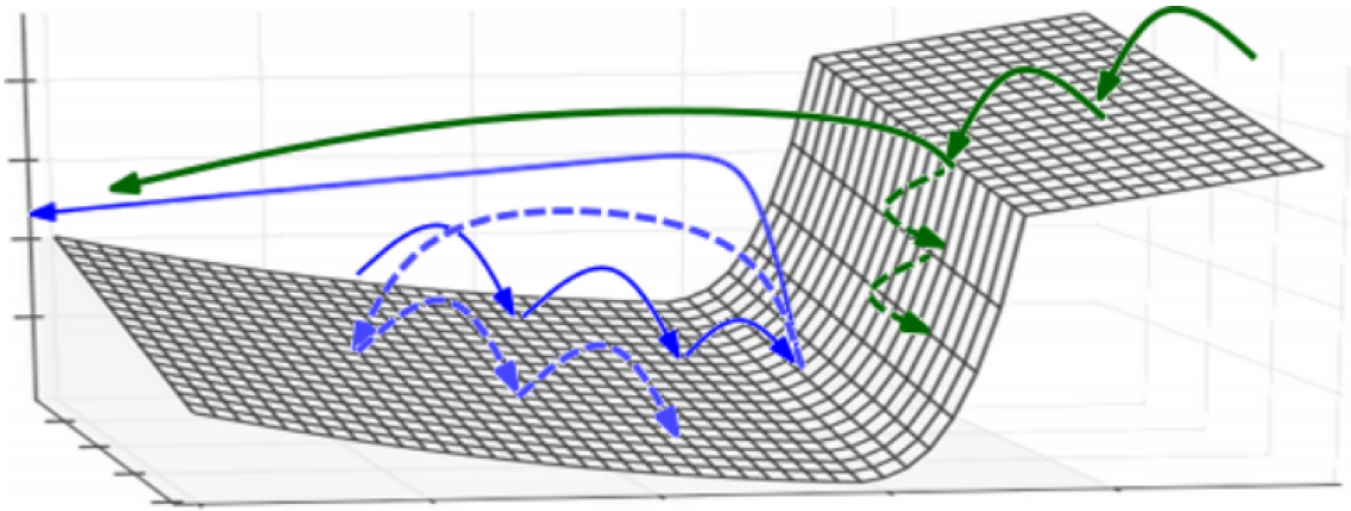
- Second-order methods or momentum assume quadratic shape around the minimum. They increase the size of steps in the low-curvature directions and decrease the sizes of steps in the high-curvature directions (the steep sides of the valley)
- When training deep models, higher order derivatives introduce 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

Why need multi-GPU?

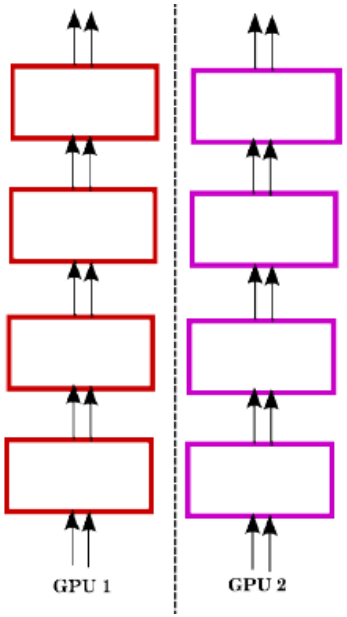
- Further speed-up
- The memory size of a single GPU is limited
  - GeForce GTX 670: 2GB
  - TITAN: 6GB
  - TITAN X: 12GB
  - Tesla K40: 12GB
  - Tesla K80: two K40
- Train bigger models
- Data parallelism
- Model parallelism

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

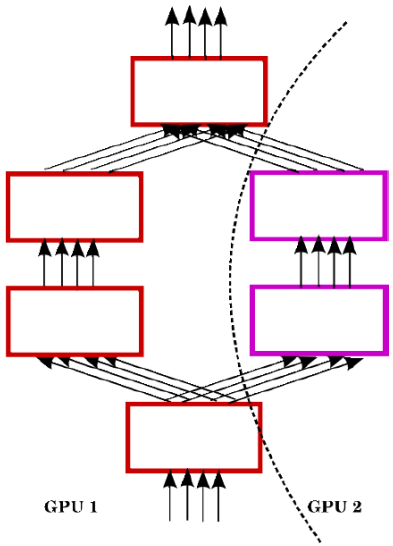


Drawbacks of data parallelism

- 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, hampering convergence

Model parallelism

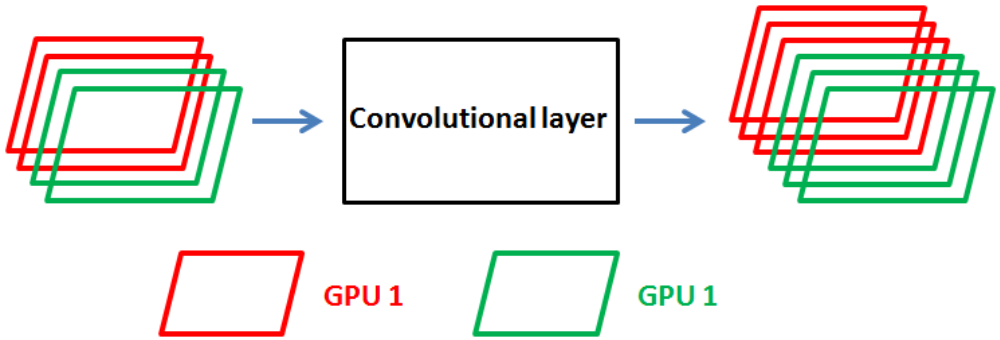
- Consist of splitting an individual network's computation across multiple GPUs
- For instance, convolutional layer with N filters can be run on two GPUs, each of which convolves its input with $N/2$ filters



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

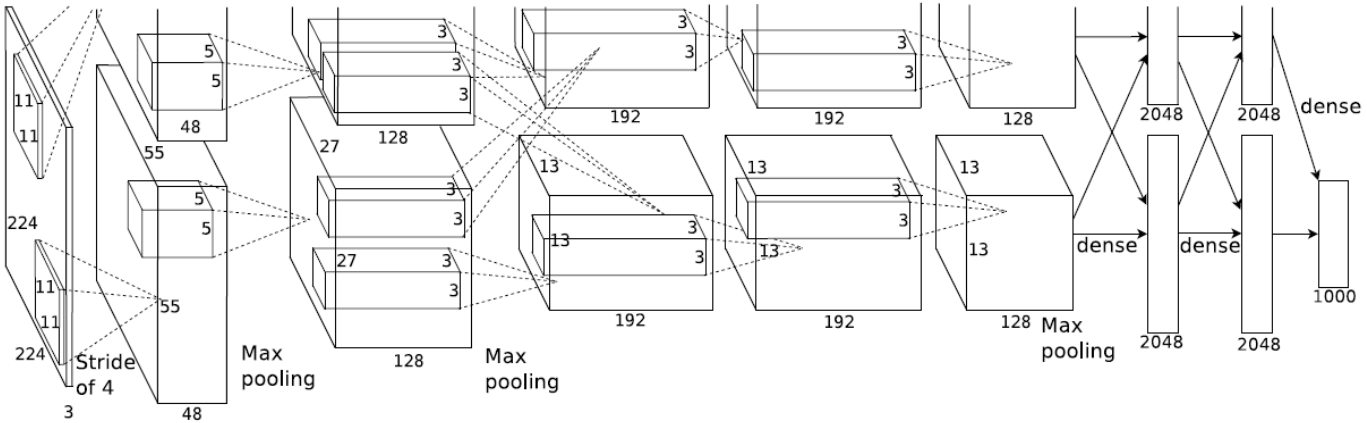
Model parallelism

- 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



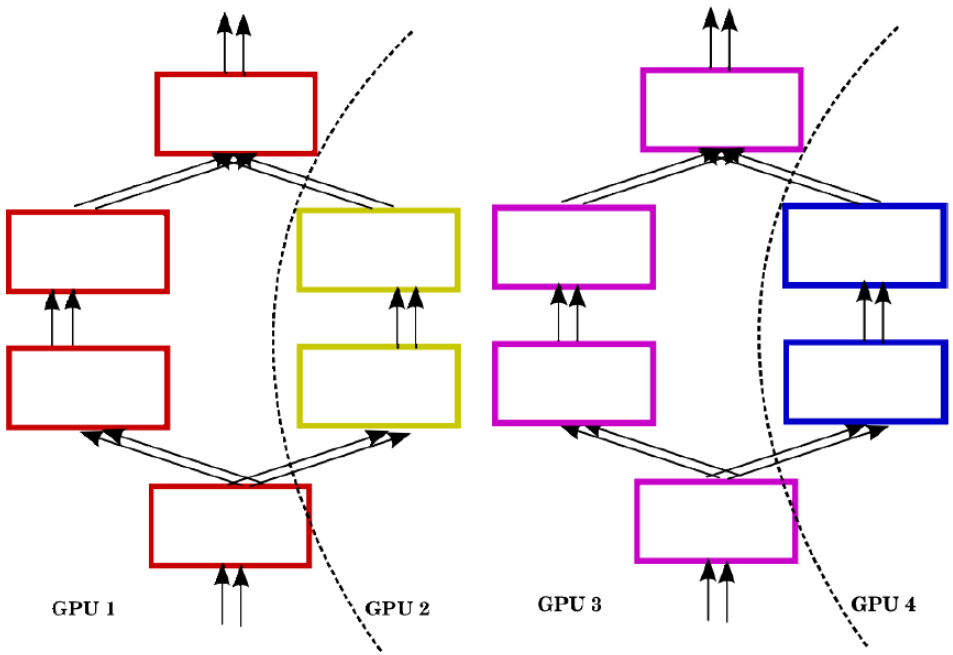
A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet classification with deep convolutional neural networks,” in NIPS, 2012.

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

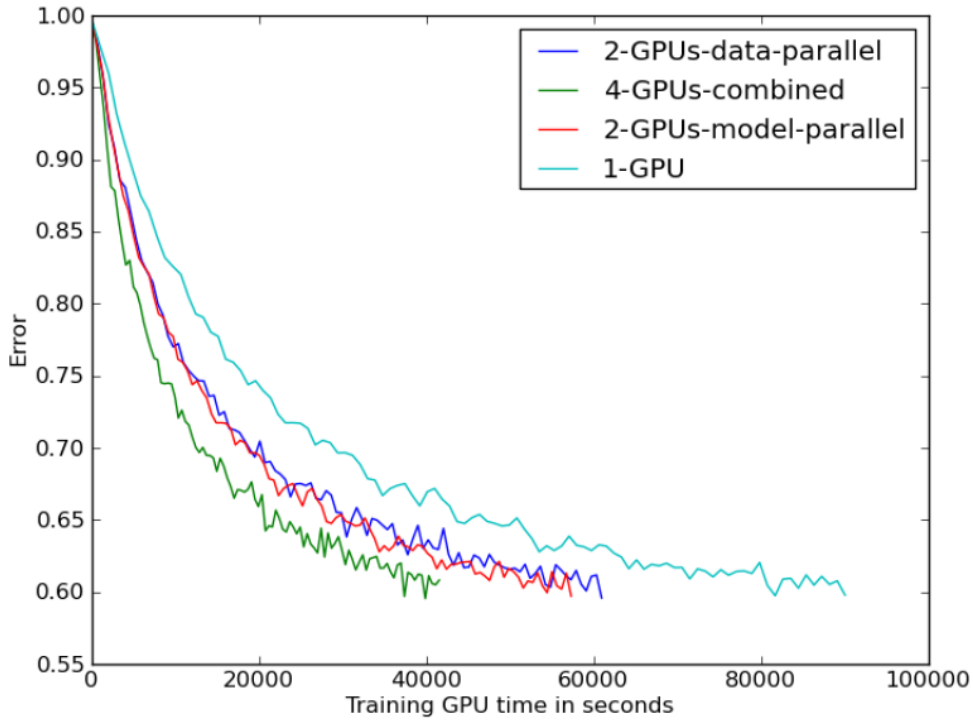
Hybrid data and model parallelism

- 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



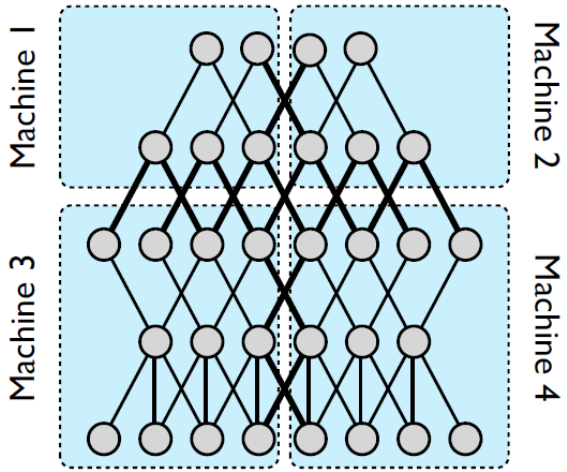
Test error on ImageNet a function of time using different forms of parallelism. All experiments used the same mini-batch size (256) and ran for 100 epochs (here showing only the first 10 for clarity of visualization) with the same architecture and the same hyper-parameter setting as in Alex net. If plotted 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 sate 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