Optimization for Training Deep Models

October 9-10, 2018

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*
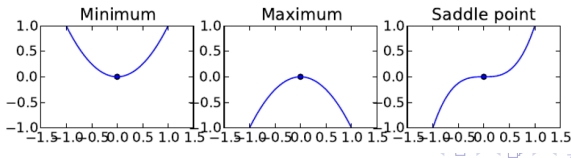
Local minimum, local maximum, and saddle points

- 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.
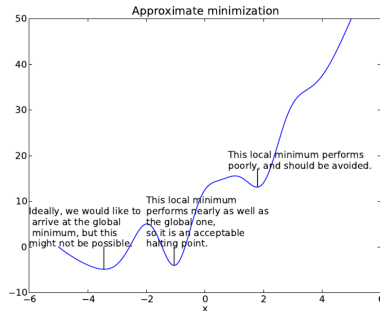
Jacobian matrix and Hessian matrix

- 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 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 minimum on another cross section.

Jacobian matrix and Hessian matrix
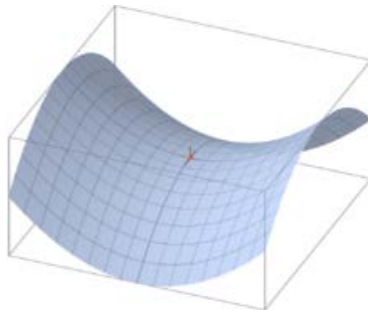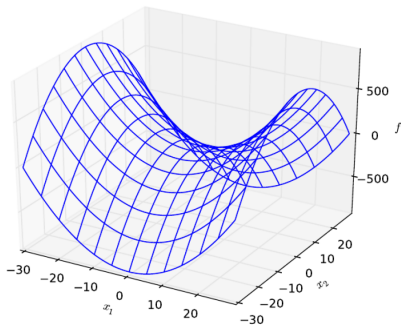

$z = x^2 - y^2$ 的鞍點在 (0,0)

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
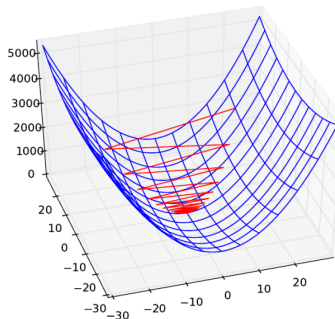
zig-zag

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

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.

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 $J(\mathbf{w})$
- 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)
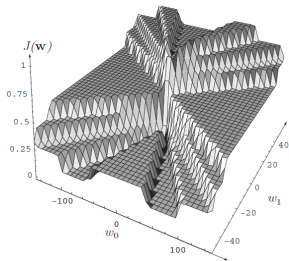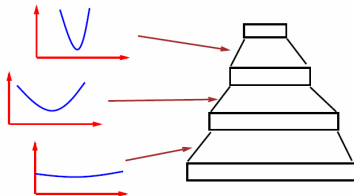
Learning rate

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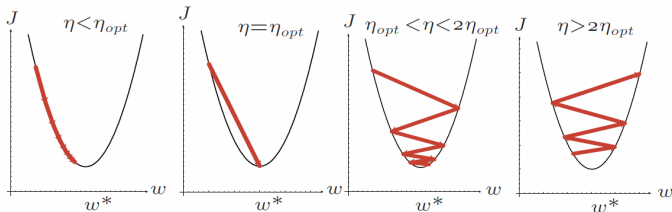
Gradient descent in a 1D quadratic criterion with different learning rates. The optimal learning rate is found by $\eta_{opt} = \left(\frac{\partial^2 J}{\partial^2 w^2} \right)^{-1}$.
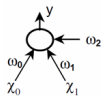
Learning rate

- 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
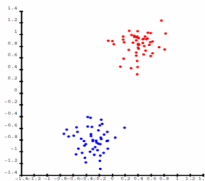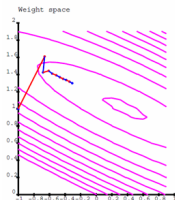
Learning rate

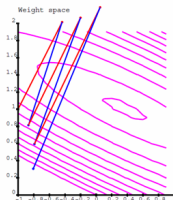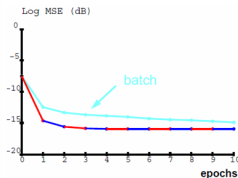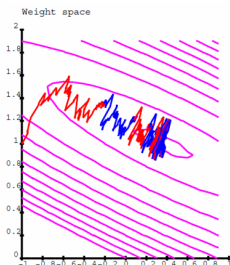- Example of linear network trained in a batch mode.

(a)

(b)

(c) $\eta = 1.5$

(d) $\eta = 2.5$

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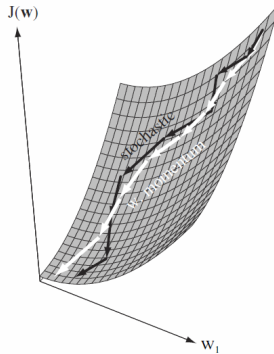
$$\mathbf{w}(m+1) = \mathbf{w}(m) + (1 - \alpha)\Delta\mathbf{w}_{bp}(m) + \alpha\Delta\mathbf{w}(m-1)$$

  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.
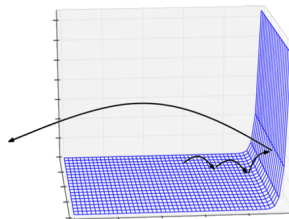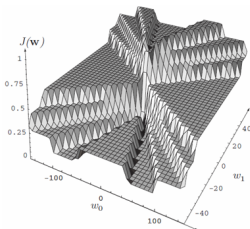
Higher-order nonlinearities

- 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
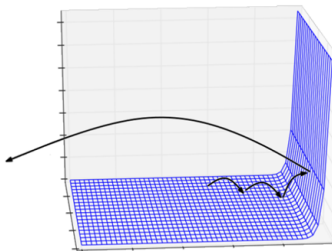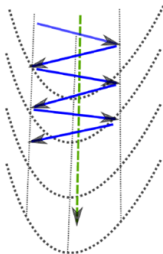
Gradient clipping

- 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.
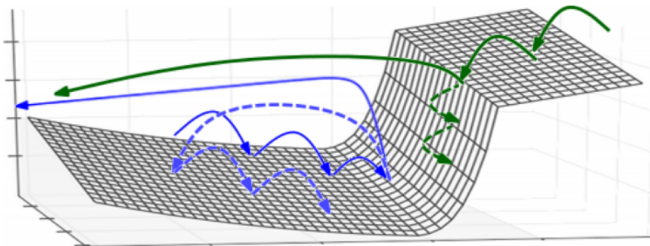
Vanishing and exploding gradients

- 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

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

The Jacobian matrix of derivatives of $f(x)$ with respect to its input vector **x** is

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

where

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

and

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

where $\alpha_t = f_{t-1}(f_{t-2}(\ldots f_2(f_1(\mathbf{x}))))$, 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 **y** should remain fixed over time, so that Θ_2 does not have to readjust to compensate for the change in the distribution of **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 **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 regime 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 regime, 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 **x** over a mini-batch: $\mathcal{B} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$
- Output: $\{\mathbf{net}^{(n)} = \text{BN}_{\mathbf{W},\mathbf{w}_0}(\mathbf{x}^{(n)})\}$
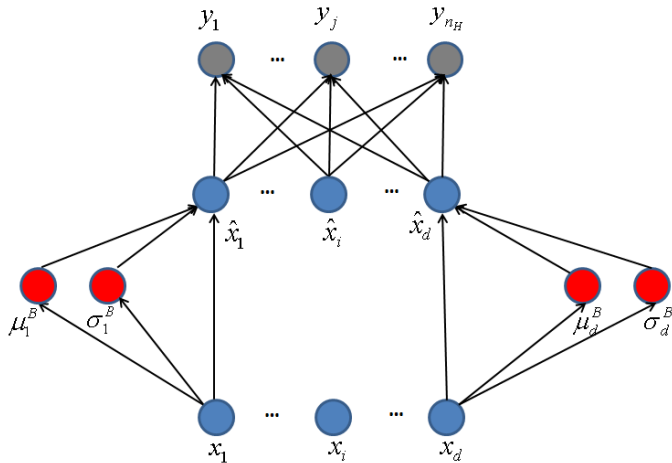
$$\mu_i^{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{n=1}^{m} \mathbf{x}_i^{(n)}$$

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

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

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

Batch Normalization - BP

Batch Normalization - BP

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

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

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

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

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

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

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

Knob 1: Initialization

Non-convex objective: initialization plays a crucial role

- Can we initialize all weights to 0?

- Random initialization: Initialize all weights with small random real numbers, e.g., Gaussian with mean zero, $\mathcal{N}(0,0.01)$

  - Consider a node v with n incoming weights w_i

  - Assume parent nodes are also mean zero.
    What is variance of activation $a[v]$ at node v
    with incoming weights $w_i \sim \mathcal{N}(0,0.01)$?
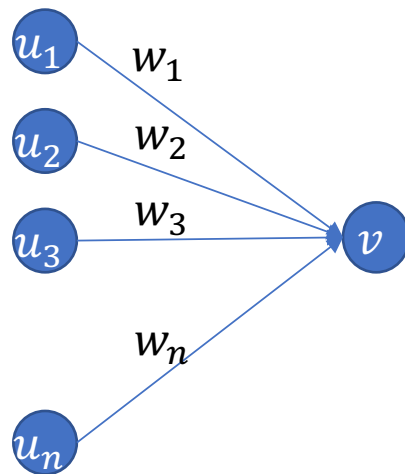
$$var(a[v]) = var(\sum_i w_i u_i) =$$
$$\#parents . var(u_i) var(w_i)$$

Knob 1: Initialization

- Xavier initialization: scale the std-dev to normalize the variance in each node

  o if **node v has n incoming weights**, each incoming weight gets random initialization of $\mathcal{N}(0, \sigma^2/n)$

  o This assumes parent nodes are zero mean

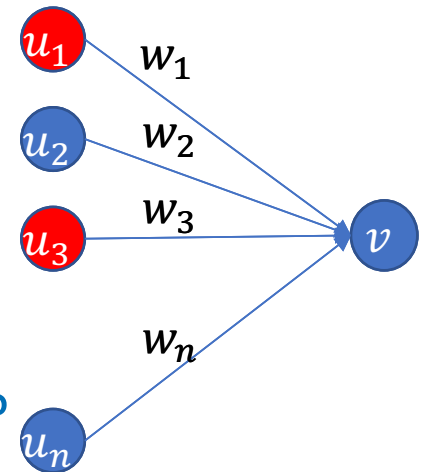 o what values can parent nodes take after activation?
  $$o[u] = \sigma(a[u])$$

    ▪ was proposed for zero mean activations: not satisfied by ReLUs

- Kaiming initialization: specifically for ReLUs.

  - On avg. we will have half of the units active, so initialize incoming weights of node v with $\mathcal{N}(0, 2\sigma^2/n)$
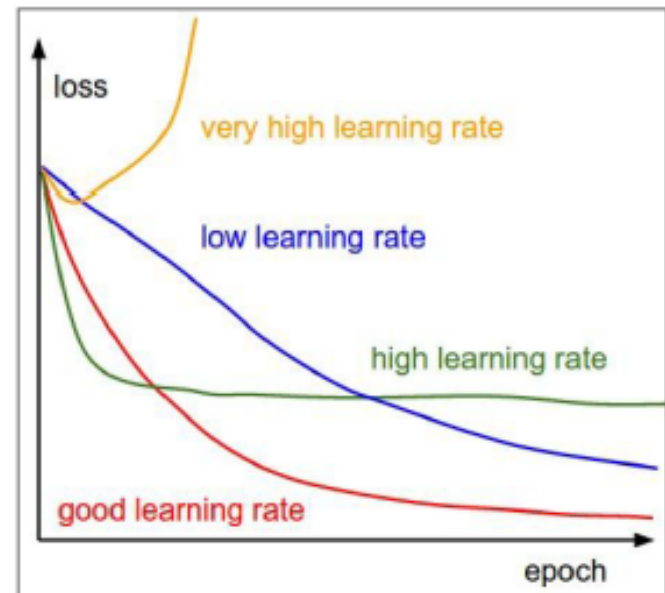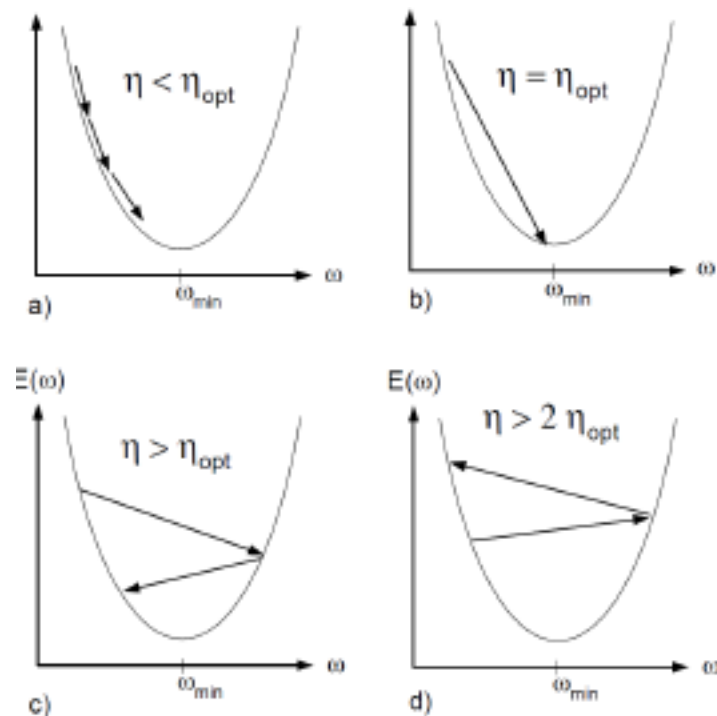
25

Knob 2: Step size/learning rate

Learning rate/step η_t size is the most important parameter to tune

- Theory from convex optimization: for SGD decay the learning rate with t as $\approx \frac{1}{C+t}$ → Use only as heuristic – does not extent for non-convex function

Figure credit: Lecun et al. (1996), Andrej Karpathy, Greg Shaknarovich

Knob 2: Step size/learning rate

Learning rate/step size η_t is the most important parameter to tune

- In practice: some degree of babysitting
  - start with a reasonable step size, $\eta_t = 0.01$
  - monitor validation/training loss
  - drop η_t (typically 1/10) when learning appears stuck
- Tips
  - wait a bit before dropping;
  - If monitoring training loss,
    - calculating loss on full dataset can be expensive
    - instead use moving average from SGD iterations
 - Crashes due to NaNs etc. often due to η_t

Figure credit: Larson et al.
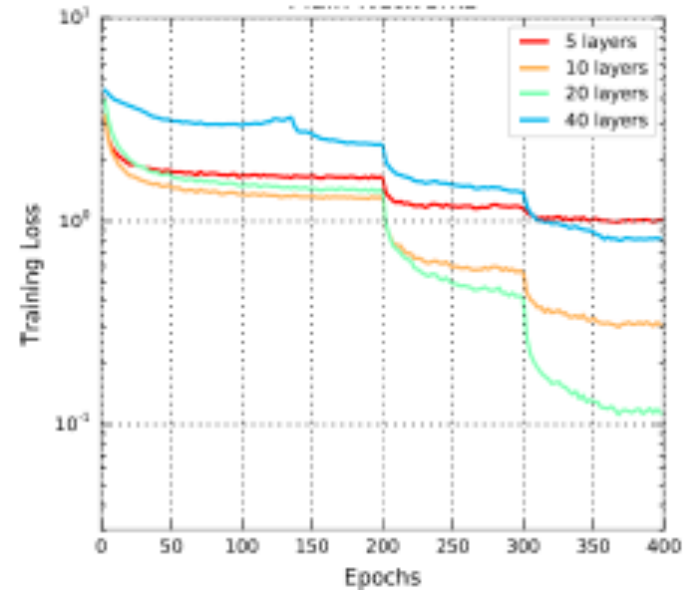
Knob 3: Variants of SGD

$$\widehat{\boldsymbol{W}} = \arg\min_{W} \sum_{i=1}^{N} \ell\left(f_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

- Stochastic gradient descent: for random $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right) \in S$

$$\boldsymbol{W}^{(t+1)} \leftarrow \boldsymbol{W}^{(t)} - \eta^{(t)} \nabla \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

- optim.SGD(model.parameters(), lr = 0.01)

- Two variants of SGD are commonly used:
  - o Momentum
    - optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
  - o Adaptive step sizes
    - Adagrad: optim.Adagrad(model.parameters(), lr = 0.01)
    - Adam: optim.Adam(params, lr=0.001, betas=(0.9, 0.999))

Knob 3a: Momentum for SGD

- S(GD) have trouble navigating areas where the curvature is steeper in one dimension than the other
  - ends up oscillating around the slopes and makes slow progress
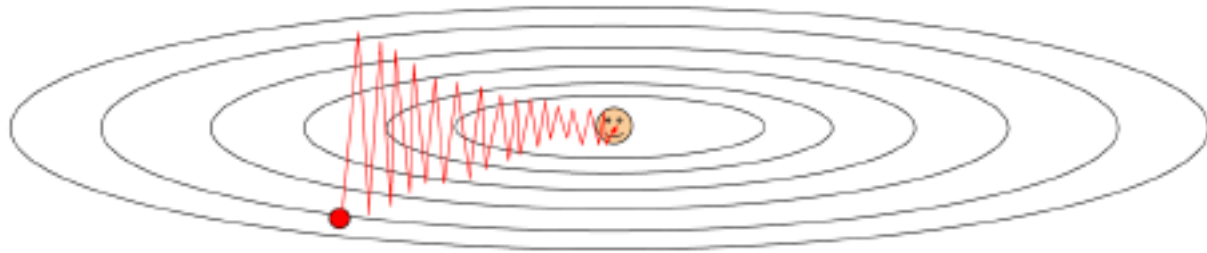
- Fix: Momentum term

$$W^{(t+1)} = W^{(t)} - \eta^{(t)} \nabla \ell(f_W(x), y)$$
$$+ \gamma^{(t)}\left(W^{(t)} - W^{(t-1)}\right)$$

- reduces updates along directions that change gradients frequently
- increases updates along directions where the gradients are consistent
- dampens oscillations
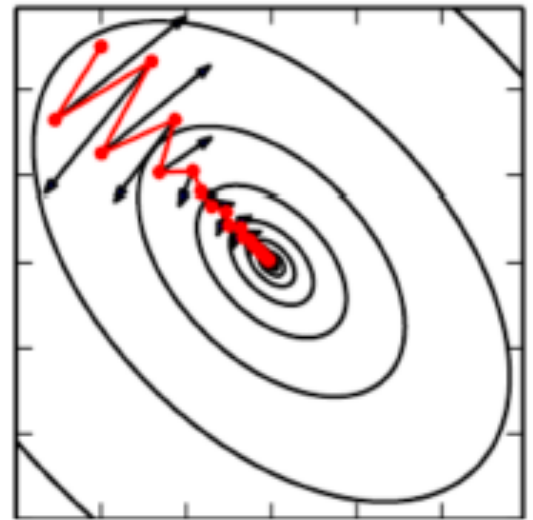
[Goodfellow et al]

Knob 3b: Adaptive step sizes

$$W^{(t+1)}[u \to v] = W^{(t)}[u \to v] - \eta_t g^{(t)}[u \to v]$$

- All weights have same learning rate

AdaGrad: Reduce learning rate proportional to updates.

$$s^{(t)}[u \to v] = s^{(t-1)}[u \to v] + \left(g^{(t)}[u \to v] \right)^2$$

$$W^{(t+1)}[u \to v] = W^{(t)} - \frac{\eta_t}{\sqrt{s^{(t)}[u \to v]} + \epsilon} g^{(t)}[u \to v]$$

- Rarely used, reduces learning rate too aggressively

RMSprop: Adagrad + forgetting

$$s^{(t)}[u \to v] = \delta s^{(t-1)}[u \to v] + (1 - \delta)\left(g^{(t)}[u \to v] \right)^2$$

$$W^{(t+1)}[u \to v] = W^{(t)} - \frac{\eta_t}{\sqrt{s^{(t)}[u \to v]} + \epsilon} g^{(t)}[u \to v]$$

Knob 3b: Adaptive step sizes

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)} - \eta_t \boldsymbol{g}^{(t)}[u \to v]$$

Adam: RMSprop with momentum

$$\boldsymbol{m}^{(t)}[u \to v] = \beta_1 \boldsymbol{m}^{(t-1)}[u \to v] + (1 - \beta_1)\boldsymbol{g}^{(t)}[u \to v]$$

$$\boldsymbol{s}^{(t)}[u \to v] = \beta_2 \boldsymbol{s}^{(t-1)}[u \to v] + (1 - \beta_2)\big(\boldsymbol{g}^{(t)}[u \to v]\big)^2$$

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)} - \frac{\eta_t}{\sqrt{\boldsymbol{s}^{(t)}[u \to v]} + \epsilon}\boldsymbol{m}^{(t)}[u \to v]$$

- Most commonly used adaptive method.
  - optim.Adam(params, lr=0.001, betas=(0.9, 0.999))

- Good first step:
  - Pick one of (SGD+momentum) or (Adam)

Knob 4: Mini-batches

- Instead of using a single example to obtain gradient estimate, use multiple examples:

- Pick m examples $B^{(t)} = \left\{ i_1^{(t)}, i_2^{(t)}, \ldots, i_m^{(t)} \right\}$ randomly

$$g^{(t)} = \frac{1}{m} \sum_{i \in B^{(t)}} \nabla_W \ell\left(f_{W^{(t)}}(\boldsymbol{x}^{(i)}), y^{(i)} \right)$$

☺ At each iteration: better gradient estimate, better (more accurate) update step
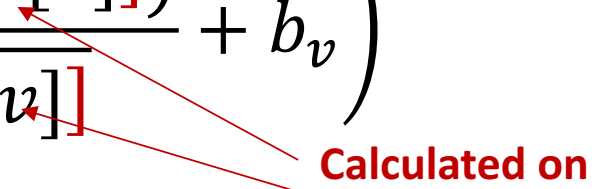
☹ But at the cost of m backprops per update

Allows parallelization, pipelining, efficient memory access

Knob 5: (Mini)Batch Normalization

$$o[v] = \sigma\left(c_v \frac{(a[v] - \widehat{\mathbb{E}}[a[v]])}{\sqrt{\widehat{\text{Var}}[a[v]]}} + b_v\right)$$
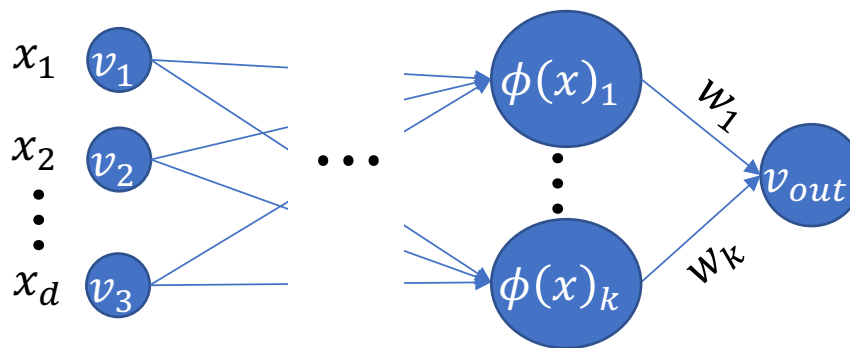
Calculated on minibatch

- Different parametrization of same function class
- SGD (or AdaGrad or ADAM) on $\{\boldsymbol{W}, \{c_v\}, \{b_v\}\}$
- Greatly helps with optimization in practice

Bonus knob: warm start/pre-training

- Suppose we want to continue training for more epochs
  - save snapshots of weights and resume again
  - need to carefully initialize learning rate now
- Also, can use weights pre-trained from another task as initialization for fine tuning a new task
  - e.g, take features from network trained for imagenet image classification and just change the last layer for new task

Bonus knob: warm start/pre-training

- Suppose we want to continue training for more epochs
  - save snapshots of weights and resume again
  - need to carefully initialize learning rate now
- Also, can use weights pre-trained from another task as initialization for fine tuning a new task
  - e.g, take features from network trained for imagenet image classification and just change the last layer for new task
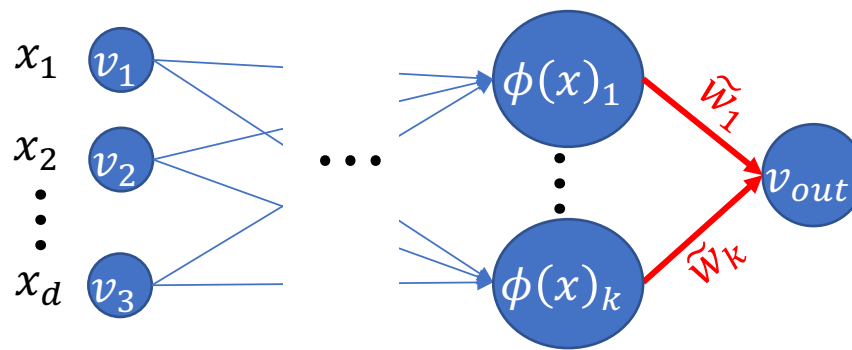
Be careful with step size/learning rate -

Neural Network Optimization

- Main technique: Stochastic Gradient Descent
- Back propagation: allows calculating gradients efficiently
- No guarantees: not convex, can take a long time, but:
  - Often still works fine, finds a good local minimum
- Over parameterization: it *seems* that using LARGE network (sometimes with #weights≫#sameples) helps optimization
  - Remember lecture 2, where doing this was a bad idea!!
  - Not well understood

Optimization

- Check
  - Add gradCheck()
  - Randomly permute data for SGD sequence

- Choose activations to avoid
  - Gradient clipping
  - Gradient explosion

- SGD "knobs" in NN training
  - Initialization → Kaiming/Xavier, or warm start initialization
  - Step size/learning rate → very important to tune based on training/validation loss
  - SGD variants
    - Momentum for SGD → usually added with SGD (default parameter momentum=0.9 often works well)
    - Adaptive variants of SGD → common alternative to SGD+momentum is Adam with $\beta_2 \gg \beta_1, e.g., \beta_2 = 0.999, \beta_1 = 0.9$
  - Mini-batch SGD → ~ 128 common
  - Batch normalization → use batch normalization

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
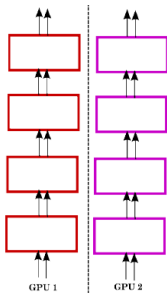
Drawbacks of data parallelism

- 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
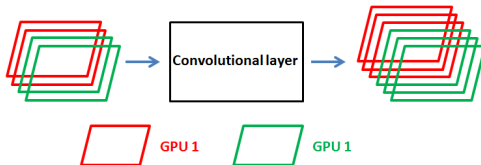
Model parallelism

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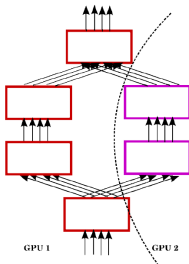
The architecture is split into two columns which make easier to split computation across the two GPUs
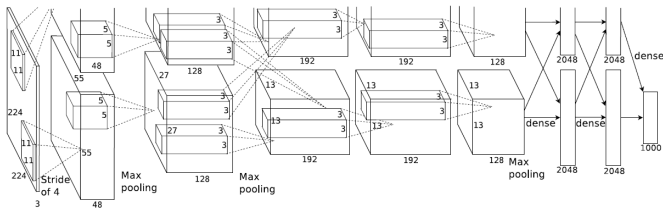
Model parallelism

- 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
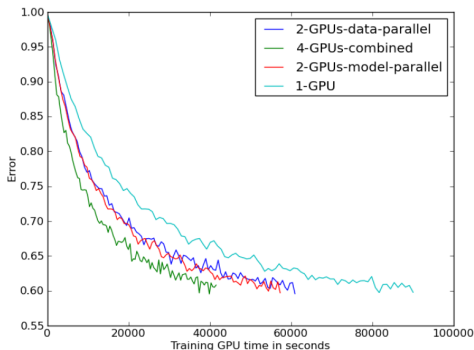
Hybrid data and model parallelism

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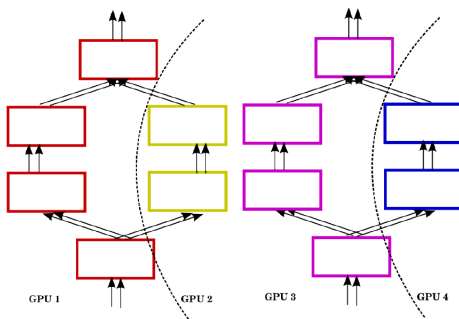
Examples of how model and data parallelism can be combined in order to make effective use of 4 GPUs

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
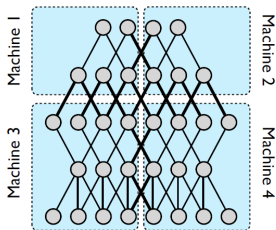
Distributed computation with CPU cores

- 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