Neural Networks

Greg Mori - CMPT 419/726

Bishop PRML Ch. 5

Neural Networks

- Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility
- We will focus on multi-layer perceptrons
  - Mathematical properties rather than plausibility

Applications of Neural Networks

- Many success stories for neural networks, old and new
  - Credit card fraud detection
  - Hand-written digit recognition
  - Face detection
  - Autonomous driving (CMU ALVINN)
  - Object recognition
  - Speech recognition

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

Outline

Feed-forward Networks

- We have looked at generalized linear models of the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right)$$

  for fixed non-linear basis functions $\phi(\cdot)$
  - We now extend this model by allowing adaptive basis functions, and learning their parameters
- In feed-forward networks (a.k.a. multi-layer perceptrons) we let each basis function be another non-linear function of linear combination of the inputs:

$$\phi_j(\boldsymbol{x}) = f\left(\sum_{j=1}^{M} \dots\right)$$

Feed-forward Networks

- We have looked at generalized linear models of the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right)$$

  for fixed non-linear basis functions $\phi(\cdot)$
  - We now extend this model by allowing adaptive basis functions, and learning their parameters
- In feed-forward networks (a.k.a. multi-layer perceptrons) we let each basis function be another non-linear function of linear combination of the inputs:

$$\phi_j(\boldsymbol{x}) = f\left(\sum_{j=1}^{M} \ldots\right)$$

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

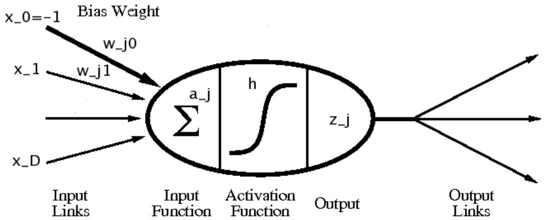
  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

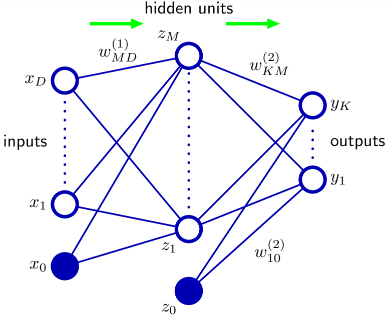


from Russell and Norvig, AIMA2e

Activation Functions

- Can use a variety of activation functions
  - Sigmoidal (S-shaped)
    - Logistic sigmoid $1/(1 + \exp(-a))$ (useful for binary classification)
    - Hyperbolic tangent $\tanh$
  - Radial basis function $z_j = \sum_i (x_i - w_{ji})^2$
  - Softmax
    - Useful for multi-class classification
  - Identity
    - Useful for regression
  - Threshold
  - Max, ReLU, Leaky ReLU, ...
- Needs to be differentiable* for gradient-based learning (later)
- Can use different activation functions in each unit

Feed-forward Networks



- Connect together a number of these units into a feed-forward network (DAG)
- Above shows a network with one layer of hidden units
- Implements function:

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = h\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

Outline

Network Training

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(\boldsymbol{x}_n, t)$, $t_n \in \mathbb{R}$
  - Squared error naturally arises:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} \{y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n\}^2$$

- For binary classification, this is another discriminative model, ML:

$$p(\boldsymbol{t}|\boldsymbol{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

Network Training

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(\boldsymbol{x}_n, t)$, $t_n \in \mathbb{R}$
  - Squared error naturally arises:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} \{y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n\}^2$$

- For binary classification, this is another discriminative model, ML:

$$p(\boldsymbol{t}|\boldsymbol{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

Network Training

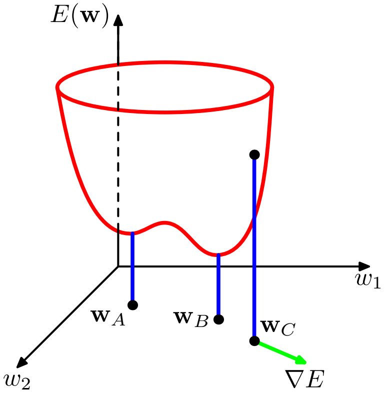
- Given a specified network structure, how do we set its parameters (weights)?
    - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(\boldsymbol{x}_n, t)$, $t_n \in \mathbb{R}$
    - Squared error naturally arises:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} \{y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n\}^2$$

- For binary classification, this is another discriminative model, ML:

$$\begin{aligned} p(\boldsymbol{t}|\boldsymbol{w}) &= \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n} \\ E(\boldsymbol{w}) &= -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \end{aligned}$$

Parameter Optimization



- For either of these problems, the error function $E(\boldsymbol{w})$ is nasty
  - Nasty = non-convex
  - Non-convex = has local minima

Descent Methods

- The typical strategy for optimization problems of this sort is a descent method:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$$

- As we've seen before, these come in many flavours
    - Gradient descent $\nabla E(\mathbf{w}^{(\tau)})$
    - Stochastic gradient descent $\nabla E_n(\mathbf{w}^{(\tau)})$
    - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective
    - Redundancy in training data, escaping local minima

Descent Methods

- The typical strategy for optimization problems of this sort is a descent method:

$$w^{(\tau+1)} = w^{(\tau)} + \Delta w^{(\tau)}$$

- As we've seen before, these come in many flavours
  - Gradient descent $\nabla E(w^{(\tau)})$
  - Stochastic gradient descent $\nabla E_n(w^{(\tau)})$
  - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima

Descent Methods

- The typical strategy for optimization problems of this sort is a descent method:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \Delta \boldsymbol{w}^{(\tau)}$$

- As we've seen before, these come in many flavours
  - Gradient descent $\nabla E(\boldsymbol{w}^{(\tau)})$
  - Stochastic gradient descent $\nabla E_n(\boldsymbol{w}^{(\tau)})$
  - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima

Computing Gradients

- The function $y(\boldsymbol{x}_n, \boldsymbol{w})$ implemented by a network is complicated
  - It isn't obvious how to compute error function derivatives with respect to weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

Computing Gradients

- The function $y(\boldsymbol{x}_n, \boldsymbol{w})$ implemented by a network is complicated
  - It isn't obvious how to compute error function derivatives with respect to weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

Computing Gradients

- The function $y(\boldsymbol{x}_n, \boldsymbol{w})$ implemented by a network is complicated
  - It isn't obvious how to compute error function derivatives with respect to weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

Computing Gradients

- The function $y(\mathbf{x}_n, \mathbf{w})$ implemented by a network is complicated
  - It isn't obvious how to compute error function derivatives with respect to weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

Outline

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$
  - $O(W)$ to compute derivatives wrt all weights
- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all activations a_j
- Calculating derivatives for weights connected to output nodes is easy
  - e.g. For linear output nodes $y_k = \sum_i w_{ki} z_i$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2}(y_{(n),k} - t_{(n),k})^2 = (y_{(n),k} - t_{(n),k}) z_{(n)i}$$

- For hidden layers, propagate error backwards from the output nodes

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$
  - $O(W)$ to compute derivatives wrt all weights
- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all activations a_j
- Calculating derivatives for weights connected to output nodes is easy
  - e.g. For linear output nodes $y_k = \sum_i w_{ki} z_i$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2} (y_{(n),k} - t_{(n),k})^2 = (y_{(n),k} - t_{(n),k}) z_{(n)i}$$

- For hidden layers, propagate error backwards from the output nodes

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$
  - $O(W)$ to compute derivatives wrt all weights
- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all activations a_j
- Calculating derivatives for weights connected to output nodes is easy
  - e.g. For linear output nodes $y_k = \sum_i w_{ki} z_i$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2} (y_{(n),k} - t_{(n),k})^2 = (y_{(n),k} - t_{(n),k}) z_{(n)i}$$

- For hidden layers, propagate error backwards from the output nodes

Chain Rule for Partial Derivatives

- A “reminder”
- For $f(x,y)$, with f differentiable wrt x and y, and x and y differentiable wrt u:

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$$

Error Backpropagation

- We can write

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} E_n(a_{j_1}, a_{j_2}, \dots, a_{j_m})$$

where $\{j_i\}$ are the indices of the nodes in the same layer as node j

- Using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} + \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{ji}}$$

where $\sum_k$ runs over all other nodes k in the same layer as node j.

- Since a_k does not depend on w_{ji}, all terms in the summation go to 0

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

Error Backpropagation

- We can write

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$$

  where $\{j_i\}$ are the indices of the nodes in the same layer as node j
- Using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} + \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{ji}}$$

  where $\sum_k$ runs over all other nodes k in the same layer as node j.
- Since a_k does not depend on w_{ji}, all terms in the summation go to 0

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

Error Backpropagation

- We can write

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$$

where $\{j_i\}$ are the indices of the nodes in the same layer as node j

- Using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} + \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial w_{ji}}$$

where $\sum_k$ runs over all other nodes k in the same layer as node j.

- Since a_k does not depend on w_{ji}, all terms in the summation go to 0

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

Error Backpropagation cont.

- Introduce error $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j \frac{\partial a_j}{\partial w_{ji}}$$

- Other factor is:

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} z_k = z_i$$

Error Backpropagation cont.

- Error δ_j can also be computed using chain rule:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_k}}_{\delta_k} \frac{\partial a_k}{\partial a_j}$$

  where $\sum_k$ runs over all nodes k in the layer **after** node j.

- Eventually:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

  - A weighted sum of the later error "caused" by this weight

Error Backpropagation cont.

- Error δ_j can also be computed using chain rule:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_k}}_{\delta_k} \frac{\partial a_k}{\partial a_j}$$

  where $\sum_k$ runs over all nodes k in the layer **after** node j.
- Eventually:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

  - A weighted sum of the later error “caused” by this weight

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

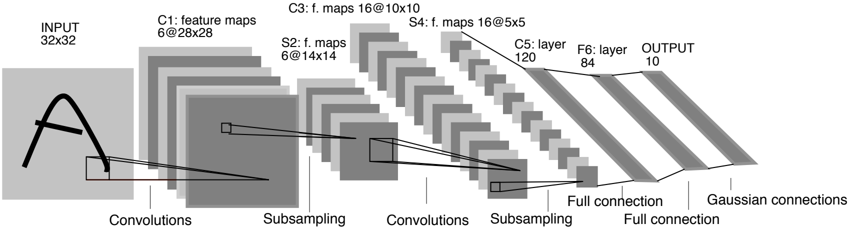
Deep Learning

- Collection of important techniques to improve performance:
  - Multi-layer networks
  - Convolutional networks, parameter tying
  - Hinge activation functions (ReLU) for steeper gradients
  - Momentum
  - Drop-out regularization
  - Sparsity
  - Auto-encoders for unsupervised feature learning
  - ...
- Scalability is key, can use lots of data since stochastic gradient descent is memory-efficient, can be parallelized

Hand-written Digit Recognition

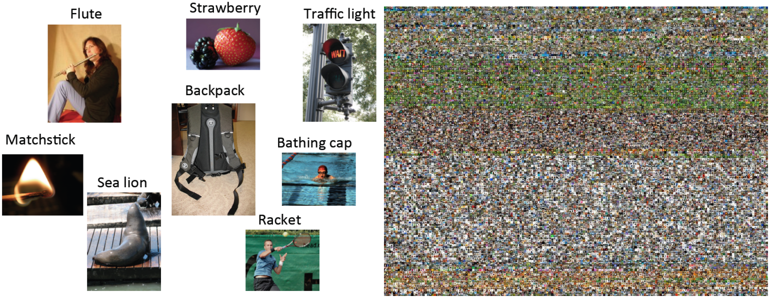
- MNIST - standard dataset for hand-written digit recognition
  - 60000 training, 10000 test images

LeNet-5, circa 1998



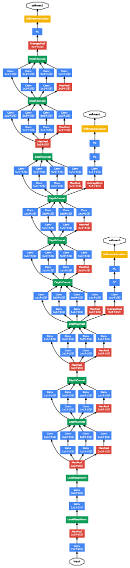
- LeNet developed by Yann LeCun et al.
  - Convolutional neural network
    - Local receptive fields (5x5 connectivity)
    - Subsampling (2x2)
    - Shared weights (reuse same 5x5 "filter")
    - Breaking symmetry

ImageNet



- ImageNet - standard dataset for object recognition in images (Russakovsky et al.)
    - 1000 image categories, $\approx$1.2 million training images (ILSVRC 2013)

GoogLeNet, circa 2014

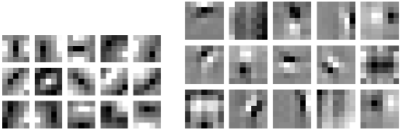


- GoogLeNet developed by Szegedy et al., CVPR 2015
- Modern deep network
- ImageNet top-5 error rate of 6.67% (later versions even better)
- Comparable to human performance (especially for fine-grained categories)

ResNet, circa 2015

- ResNet developed by He et al., ICCV 2015
- 152 layers
- ImageNet top-5 error rate of 3.57%
- Better than human performance (especially for fine-grained categories)

Key Component 1: Convolutional Filters

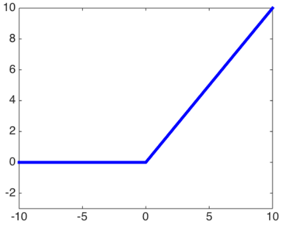


- Share parameters across network
- Reduce total number of parameters
- Provide translation invariance, useful for visual recognition

Key Component 2: Rectified Linear Units (ReLUs)

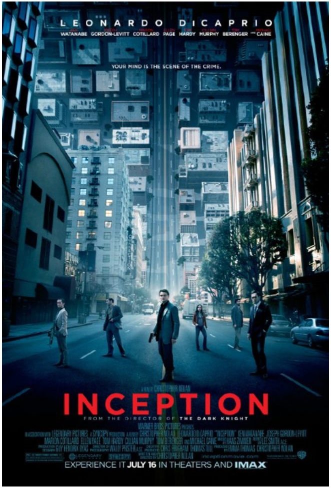
- Vanishing gradient problem
  - If derivatives very small, no/little progress via stochastic gradient descent
  - Occurs with sigmoid function when activation is large in absolute value

Key Component 2: Rectified Linear Units (ReLUs)



- Vanishing gradient problem
  - If derivatives very small, no/little progress via stochastic gradient descent
  - Occurs with sigmoid function when activation is large in absolute value
- ReLU: $h(a_j) = \max(0, a_j)$
- Non-saturating, linear gradients (as long as non-negative activation on some training data)
- Sparsity inducing

Key Component 3: Many, Many Layers



- ResNet: $\approx$152 layers ("shortcut connections")
- GoogLeNet: $\approx$27 layers ("Inception" modules)
- VGG Net: 16-19 layers (Simonyan and Zisserman, 2014)
- Supervision: 8 layers (Krizhevsky et al., 2012)

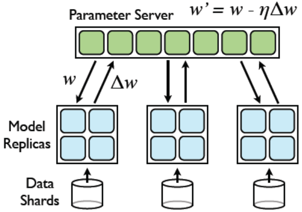
Key Component 4: Momentum

- Trick to escape plateaus / local minima
- Take exponential average of previous gradients

$$\overline{\frac{\partial E_n}{\partial w_{ji}}}^{\tau} = \frac{\partial E_n}{\partial w_{ji}}^{\tau} + \alpha \overline{\frac{\partial E_n}{\partial w_{ji}}}^{\tau-1}$$

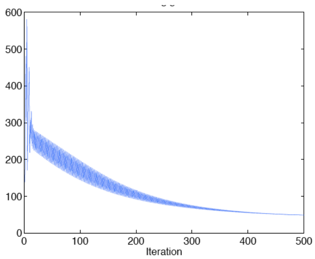
- Maintains progress in previous direction

Key Component 5: Asynchronous Stochastic Gradient Descent



- Big models won't fit in memory
- Want to use compute clusters (e.g. 1000s of machines) to run stochastic gradient descent
- How to parallelize computation?
- Ignore synchronization across machines
  - Just let each machine compute its own gradients and pass to a server storing current parameters
  - Ignore the fact that these updates are inconsistent
  - Seems to just work (e.g. Dean et al. NIPS 2012)

Key Component 6: Learning Rate Schedule



- How to set learning rate η?:

$$\boldsymbol{w}^{\tau} = \boldsymbol{w}^{\tau-1} + \eta \nabla \boldsymbol{w}$$

- Option 1: Run until validation error plateaus. Drop learning rate by x%
- Option 2: Adagrad, adaptive gradient. Per-element learning rate set based on local geometry (Duchi et al. 2010)

Key Component 7: Batch Norm

- Normalize data at each layer by whitening
- Ioffe and Szegedy 2015

Key Component 8: Data Augmentation

- Augment data with additional synthetic variants (10x amount of data)
- Or just use synthetic data, e.g. Sintel animated movie (Butler et al. 2012)

Key Component 9: Data and Compute

- Get lots of data (e.g. ImageNet)
- Get lots of compute (e.g. CPU cluster, GPUs)
- Cross-validate like crazy, train models for 2-3 weeks on a GPU
- Researcher gradient descent (RGD) or Graduate student descent (GSD): get 100s of researchers to each do this, trying different network structures

More information

- `https://sites.google.com/site/deeplearningsummerschool`
- `http://tutorial.caffe.berkeleyvision.org/`
- `ufldl.stanford.edu/eccv10-tutorial`
- `http://www.image-net.org/challenges/LSVRC/2012/supervision.pdf`
- Project ideas
  - Long short-term memory (LSTM) models for temporal data
  - Learning embeddings (word2vec, FaceNet)
  - Structured output (multiple outputs from a network)
  - Zero-shot learning (learning to recognize new concepts without training data)
  - Transfer learning (use data from one domain/task, adapt to another)
  - Network compression / run-time / power optimization
  - Distillation

Conclusion

- Readings: Ch. 5.1, 5.2, 5.3
- Feed-forward networks can be used for regression or classification
  - Similar to linear models, except with adaptive non-linear basis functions
  - These allow us to do more than e.g. linear decision boundaries
- Different error functions
- Learning is more difficult, error function not convex
  - Use stochastic gradient descent, obtain (good?) local minimum
- Backpropagation for efficient gradient computation