Deep Neural Networks

CMSC 422

MARINE CARPUAT
marine@cs.umd.edu

Deep learning slides credit: Vlad Morariu

- What we know so far
  - What are multi-layer perceptrons?
  - How to make predictions in MLPs?
  - How to train MLPs?

- Today
  - Practical issues with (deep) neural network training

Forward Propagation:
given input x, compute network output

Algorithm 24 TwoLayerNetworkPredict($\mathbf{W}, \boldsymbol{v}, \hat{\boldsymbol{x}}$)

for $i = 1$ **to** *number of hidden units* **do**
  $h_i \leftarrow \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}})$ // compute activation of hidden unit i
end for
return $\boldsymbol{v} \cdot \boldsymbol{h}$ // compute output unit

Neural Network Training

Backpropagation algorithm

=

Gradient descent + Chain rule

Backprop in a 2-layer network

Algorithm 25 TwoLayerNetworkTrain($\mathbf{D}$, η, K, $MaxIter$)

$\mathbf{W} \leftarrow D{\times}K$ matrix of small random values // initialize input layer weights
$v \leftarrow K$-vector of small random values // initialize output layer weights
for $iter = 1 \ldots MaxIter$ **do**
  $\mathbf{G} \leftarrow D{\times}K$ matrix of zeros // initialize input layer gradient
  $g \leftarrow K$-vector of zeros // initialize output layer gradient

  Compute Gradient G and g

  $\mathbf{W} \leftarrow \mathbf{W} - \eta\mathbf{G}$ // update input layer weights
  $v \leftarrow v - \eta g$ // update output layer weights
end for
return $\mathbf{W}$, v

What's our Training Objective?

- We'll consider the following objective

$$\min_{\mathbf{W},v} \quad \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)\right)^2$$

  - i.e. our goal is to find parameters **W**, v that minimize squared error

- Other objectives are possible (e.g., other loss functions, add regularizer)

Gradient of objective w.r.t. output layer weights v

$$\nabla_{\boldsymbol{v}} = -\sum_n e_n \boldsymbol{h}_n$$

Error at example n: $y_n - \widehat{y_n}$

Vector of activations of hidden units for example n

Gradient of objective w.r.t. hidden unit weights w_i

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right)^2$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial \boldsymbol{w}_i}$$

Chain rule

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right) v_i = -e v_i$$

$$\frac{\partial f_i}{\partial \boldsymbol{w}_i} = f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$

$$\nabla_{\boldsymbol{w}_i} = -e v_i f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$

(This is on one example only)

Backprop in a 2-layer network

Algorithm 25 TwoLayerNetworkTrain($\mathbf{D}$, η, K, $MaxIter$)

$\mathbf{W} \leftarrow D{\times}K$ matrix of small random values // initialize input layer weights
$\boldsymbol{v} \leftarrow K$-vector of small random values // initialize output layer weights
for $iter = 1 \ldots MaxIter$ **do**
  $\mathbf{G} \leftarrow D{\times}K$ matrix of zeros // initialize input layer gradient
  $\boldsymbol{g} \leftarrow K$-vector of zeros // initialize output layer gradient
  for all $(\boldsymbol{x}, y) \in \mathbf{D}$ **do**
    for $i = 1$ **to** K **do**
      $a_i \leftarrow \boldsymbol{w}_i \cdot \hat{\boldsymbol{x}}$
      $h_i \leftarrow \tanh(a_i)$ // compute activation of hidden unit i
    end for
    $\hat{y} \leftarrow \boldsymbol{v} \cdot \boldsymbol{h}$ // compute output unit
    $e \leftarrow y - \hat{y}$ // compute error
    $\boldsymbol{g} \leftarrow \boldsymbol{g} - e\boldsymbol{h}$ // update gradient for output layer
    for $i = 1$ **to** K **do**
      $\mathbf{G}_i \leftarrow \mathbf{G}_i - e v_i (1 - \tanh^2(a_i)) \boldsymbol{x}$ // update gradient for input layer
    end for
  end for
  $\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{G}$ // update input layer weights
  $\boldsymbol{v} \leftarrow \boldsymbol{v} - \eta \boldsymbol{g}$ // update output layer weights
end for
return $\mathbf{W}$, $\boldsymbol{v}$

Forward propagation

Update gradients

Update parameters

Tricky issues with neural network training

- Sensitive to initialization
  - Objective is non-convex, many local optima
  - In practice: start with random values rather than zeros

- Many other hyperparameters
  - Number of hidden units (and potentially hidden layers)
  - Gradient descent learning rate
  - Stopping criterion

Neural networks vs. linear classifiers

Advantages of Neural Networks:
– More expressive
– Less feature engineering

Inconvenients of Neural Networks:
– Harder to train
– Harder to interpret

Try different architectures and training parameters here:

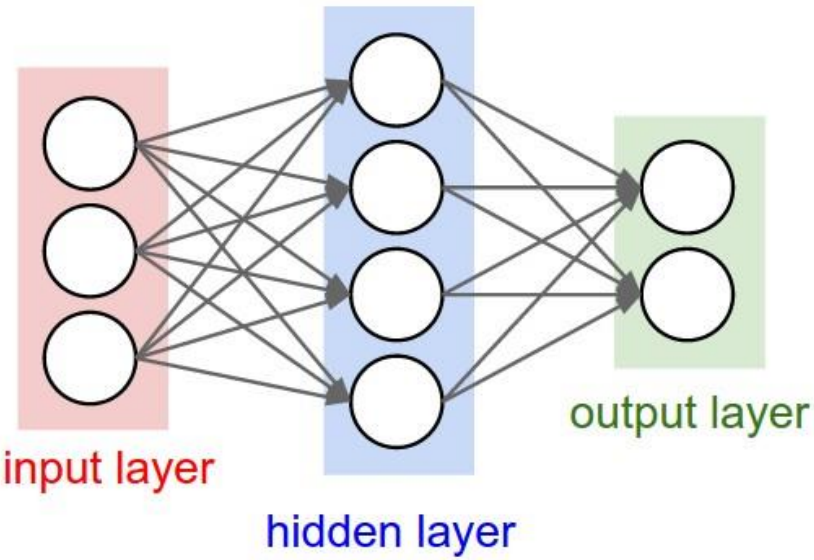
http://playground.tensorflow.org

Neural Network Architectures

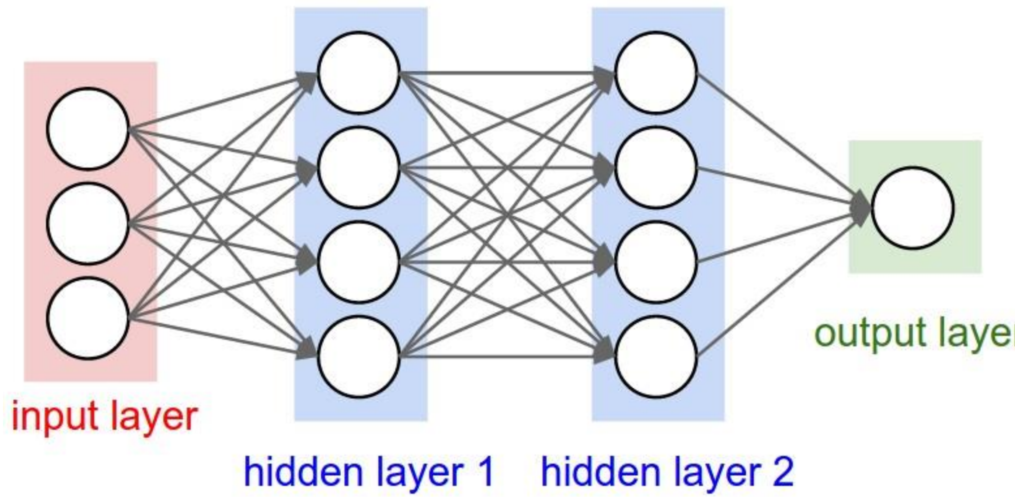
- We focused on a **2-layer feedforward** network

- Many other deeper architectures
  - Feedforward network with more than 2 layers
  - Recurrent network (i.e. network has cycles)
  - Can still be trained with backpropagation
    - But more practical issues arise with deeper networks

Multi-Layer Perceptron (MLP)



2 layer network

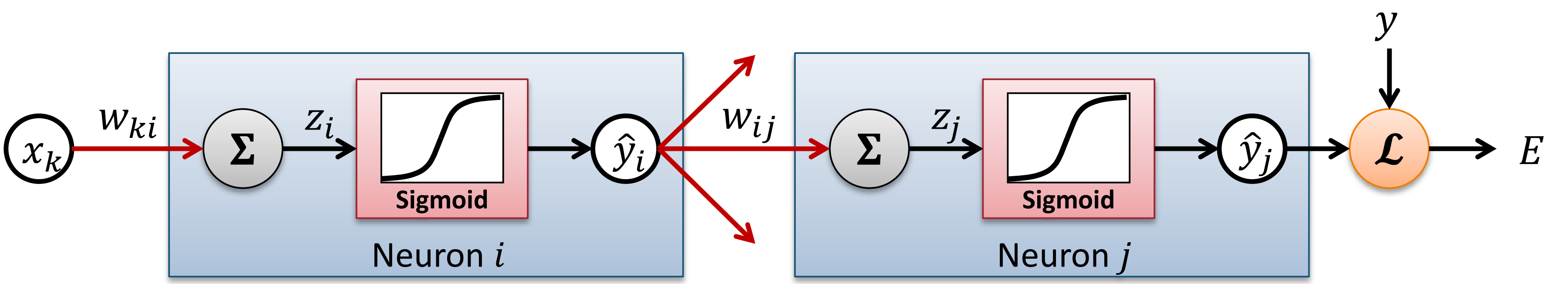


3 layer network

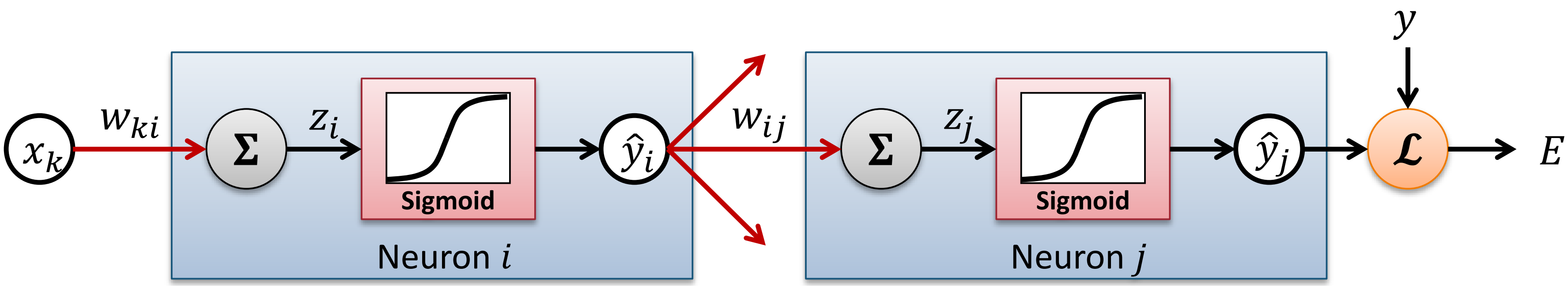
Image source: http://cs231n.github.io/neural-networks-1/

Computational Graphs

- Simple and powerful abstraction to implement forward and back-propagation



Multi-Layer: Backpropagation



$$\frac{\partial E}{\partial z_j} = \frac{d\hat{y}_j}{dz_j}\frac{\partial E}{\partial \hat{y}_j}$$

$$\frac{\partial E}{\partial \hat{y}_i} = \sum_j \frac{dz_j}{d\hat{y}_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{d\hat{y}_j}{dz_j}\frac{\partial E}{\partial \hat{y}_j}$$

$$\frac{\partial E}{\partial w_{ki}} = \sum_n \frac{\partial z_i^n}{\partial w_{ki}}\frac{d\hat{y}_i^n}{dz_i^n}\frac{\partial E}{\partial \hat{y}_i^n} = \sum_n \frac{\partial z_i^n}{\partial w_{ki}}\frac{d\hat{y}_i^n}{dz_i^n}\sum_j w_{ij}\frac{d\hat{y}_j^n}{dz_j^n}\frac{\partial E}{\partial \hat{y}_j^n}$$

Slide credit: Bohyung Han

We can in principle use same gradient descent algorithm as before

Algorithm 25 TwoLayerNetworkTrain($\mathbf{D}$, η, K, $MaxIter$)

$\mathbf{W} \leftarrow D{\times}K$ matrix of small random values // initialize input layer weights
$\boldsymbol{v} \leftarrow K$-vector of small random values // initialize output layer weights
for $iter = 1 \dots MaxIter$ **do**
  $\mathbf{G} \leftarrow D{\times}K$ matrix of zeros // initialize input layer gradient
  $\boldsymbol{g} \leftarrow K$-vector of zeros // initialize output layer gradient
  for all $(\boldsymbol{x}, y) \in \mathbf{D}$ **do**
    for $i = 1$ **to** K **do**
      $a_i \leftarrow \boldsymbol{w}_i \cdot \hat{\boldsymbol{x}}$
      $h_i \leftarrow \tanh(a_i)$ // compute activation of hidden unit i
    end for
    $\hat{y} \leftarrow \boldsymbol{v} \cdot \boldsymbol{h}$ // compute output unit
    $e \leftarrow y - \hat{y}$ // compute error

Forward propagation

    $\boldsymbol{g} \leftarrow \boldsymbol{g} - e\boldsymbol{h}$ // update gradient for output layer
    for $i = 1$ **to** K **do**
      $\mathbf{G}_i \leftarrow \mathbf{G}_i - e v_i (1 - \tanh^2(a_i)) \boldsymbol{x}$ // update gradient for input layer
    end for

Update gradients

  end for
  $\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{G}$ // update input layer weights
  $\boldsymbol{v} \leftarrow \boldsymbol{v} - \eta \boldsymbol{g}$ // update output layer weights

Update parameters

end for
return $\mathbf{W}$, $\boldsymbol{v}$

Issues in Deep Neural Networks

- Long training time
  - There are sometimes a lot of training data
  - Many iterations (epochs) are typically required for optimization
  - Computing gradients in each iteration takes too much time
- Overfitting
  - Learned function fits training data well, but performs poorly on new data (high capacity model, not enough training data)

Slide credit: adapted from Bohyung Han

Improving on Gradient Descent: Stochastic Gradient Descent (SGD)

- Update weights for each example

$$E = \frac{1}{2}(y^n - \hat{y}^n)^2 \qquad \boldsymbol{w}_i(t+1) = \boldsymbol{w}_i(t) - \epsilon \frac{\partial E^n}{\partial \boldsymbol{w}_i}$$

+ **Fast, online**
− **Sensitive to noise**

- Minibatch SGD: Update weights for a small set of examples

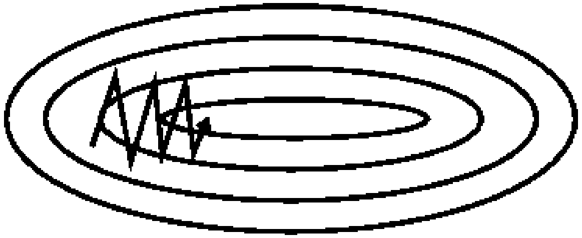
$$E = \frac{1}{2}\sum_{n \in B}(y^n - \hat{y}^n)^2 \qquad \boldsymbol{w}_i(t+1) = \boldsymbol{w}_i(t) - \epsilon \frac{\partial E^B}{\partial \boldsymbol{w}_i}$$

+ **Fast, online**
+ **Robust to noise**

Slide credit: Bohyung Han

Improving on Gradient Descent: SGD with Momentum

SGD w/o momentum



SGD with momentum helps dampen oscillations

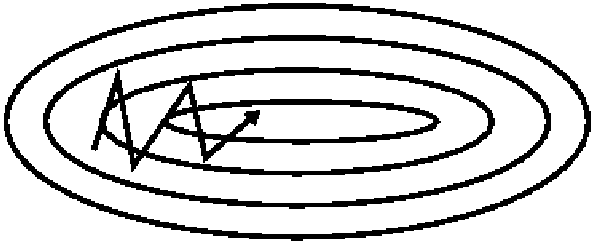


Image: http://ruder.io/optimizing-gradient-descent/index.html#momentum

Improving on Gradient Descent: SGD with Momentum

- Update based on gradients + previous direction

$$v_i(t) = \alpha v_i(t-1) - \epsilon \frac{\partial E}{\partial w_i}(t)$$

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \boldsymbol{v}(t)$$

+ Converge faster
+ Avoid oscillation

Slide credit: Bohyung Han

Improving the Training Objective: Regularization/Weight Decay

- Penalize the size of the weights

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$w_i(t+1) = w_i(t) - \epsilon \frac{\partial C}{\partial w_i} = w_i(t) - \epsilon \frac{\partial E}{\partial w_i} - \lambda w_i$$

+ Improve generalization a lot!

Slide credit: Bohyung Han

Training (Deep) Neural Networks

- Computational graphs
- Improvements to gradient descent
  - Stochastic gradient descent
  - Momentum
  - Weight decay

Neural Network history

Perceptron

- Proposed by Frank Rosenblatt in 1957
- Real inputs/outputs, threshold activation function

Revival in the 1980's

Backpropagation discovered in 1970's but popularized in 1986

- David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams. "Learning representations by back-propagating errors." In Nature, 1986.

MLP is a universal approximator

- Can approximate any non-linear function in theory, given enough neurons, data
- Kurt Hornik, Maxwell Stinchcombe, Halbert White. "Multilayer feedforward networks are universal approximators." Neural Networks, 1989

Generated lots of excitement and applications

http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/

Neural Networks Applied to Vision

LeNet – vision application

– LeCun, Y; Boser, B; Denker, J; Henderson, D; Howard, R; Hubbard, W; Jackel, L, "Backpropagation Applied to Handwritten Zip Code Recognition," in Neural Computation, 1989

– USPS digit recognition, later check reading

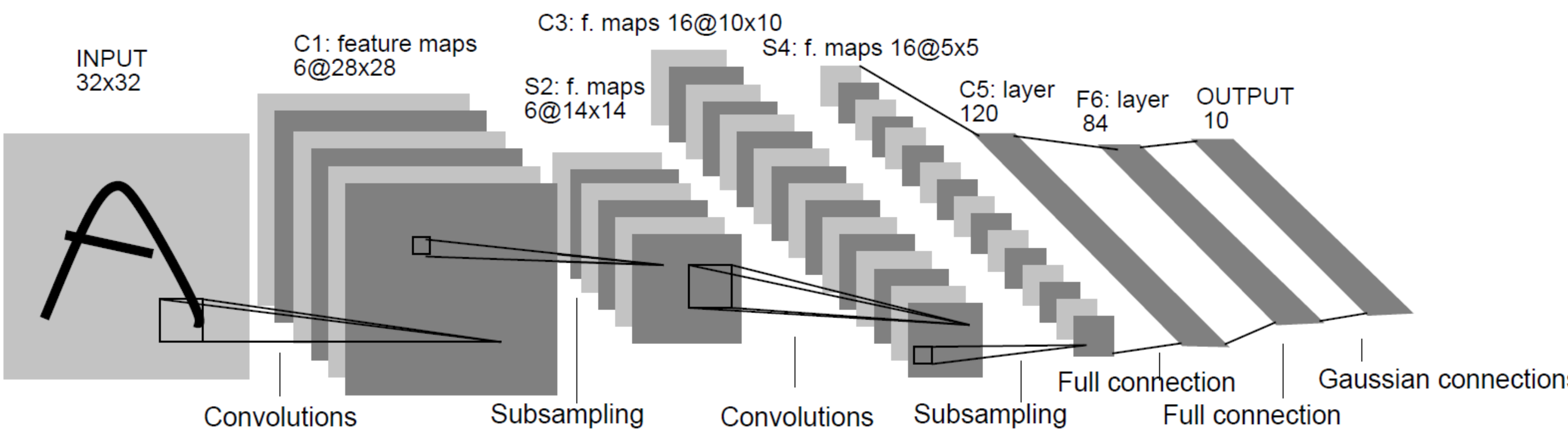


Image credit: LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 1998.

New “winter” and revival in early 2000’s

New “winter” in the early 2000’s due to

- problems with training NNs
- Support Vector Machines (SVMs), Random Forests (RF) – easy to train, nice theory

Revival again by 2011-2012

- Name change (“neural networks” -> “deep learning”)
- + Algorithmic developments that made training somewhat easier
- + Big data + GPU computing
- = performance gains on many tasks (esp Computer Vision)

http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-4/

Training (Deep) Neural Networks

- Computational graphs
- Improvements to gradient descent
  - Stochastic gradient descent
  - Momentum
  - Weight decay
- Next:
  - The Vanishing Gradient Problem
  - Examples of current deep learning architectures