CSC 439/539
Statistical Natural Language Processing
Lecture 2b: Best Practices

Mihai Surdeanu

Fall 2017

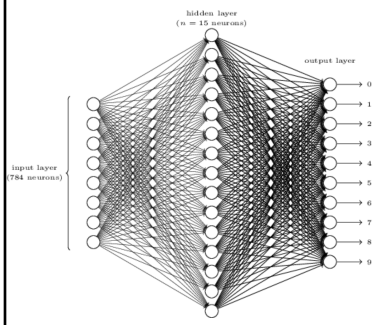
## Take-away

- Multi-class classification
- Regularization
- Other decision functions for NNs
- Other cost functions for NNs
- Optimizers (beyond SGD)
- Weight initialization
- Vanishing and exploding gradients
- Saturated and dead neurons
- Shuffling
- Learning rate
- Dense vs. sparse representations

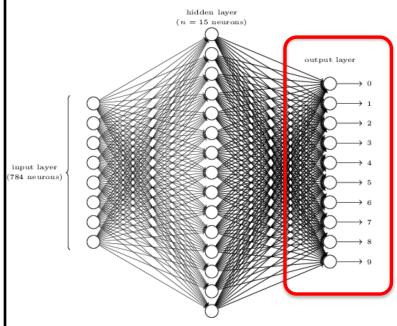
## Multi-class classification

- In general: one-vs-all, or one-vs-rest
- How would you do it for the Perceptron?
  - Example!

## Multi-class classification for LR and NN




## Multi-class classification for LR and NN




**Softmax** layer:

- One output per class. Thus, one distinct vector W per class!
- The scores form a probability distribution

## Softmax

- A generalization of the sigmoid function:

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}}$$

  - Where $a_j$ is the dot product of the weight vector for class *j* and the input vector
- For a discussion of the softmax derivative, see: http://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/
  - But in this course we will rely on auto-differentiation for this

## Regularization

- Regularization = feature selection
- You have already seen 2 examples of regularization. What are they?
- But for LR and NN, we can do this more elegantly, using the same math we've seen already

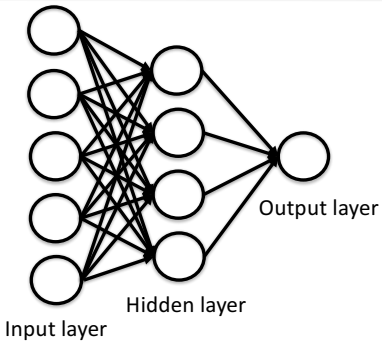
## L2 and L1 Regularization

- We add a new term to the cost function C:
  - C = YOUR_FAVORITE_COST_FUNC_TO_MINIMIZE + $\frac{\lambda}{2}||\theta||_2^2$
  - What does this mean for the partial derivatives in LR?
  - What effect does this have on the individual weights in Theta?
- L1 regularization:
  - C = YOUR_FAVORITE_COST_FUNC_TO_MINIMIZE + $\lambda||\theta||_1$
  - L1 performs a more aggressive feature selection. Why?

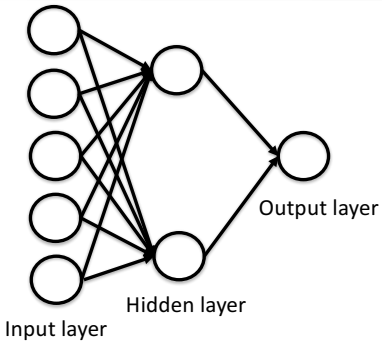
## Dropout

- Randomly disabling (both in the forward and backward steps) a percentage of the neurons in the network, for each training example
  - Designed to prevent the network to rely on specific weights
  - A poor-man's ensemble model (kinda similar to what?)

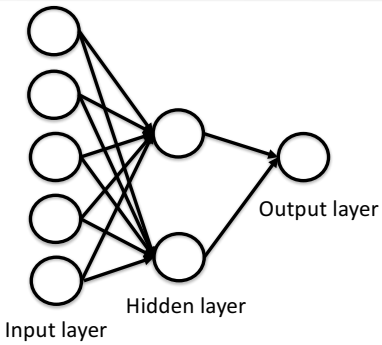
## Dropout




## Dropout




## Dropout



## Decision functions for NNs

- Sigmoid/logistic: $\sigma(x) = 1/(1+e^{-x})$
- Hyperbolic tangent: $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$
- Hard tanh: $hardtanh(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise} \end{cases}$
- Rectified linear unit (ReLU):
$$ReLU(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases}$$
- Rule of thumb: ReLU > tanh > sigmoid

## Other cost/loss functions for NNs

- We have seen the mean squared error cost function in the previous lecture
- Many others are possible. In general, a cost function computes a scalar given two vectors: the true predictions, and the network's output on the same data
- Loss values: always positive, 0 only when the network's output is correct
- Several are commonly used

## Other cost/loss functions for NNs

- Hinge (binary):
  - Outputs must be -1 or +1 (which decision function does this?)
  - $L_{hinge(binary)}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$
- Hinge (multiclass):
  - *t* is the correct class, *k* is the highest scoring incorrect class
  - $L_{hinge(multiclass)}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - (\hat{y}_t - \hat{y}_k))$

## Other cost/loss functions for NNs

- Log loss:
  - A "soft" version of the hinge loss (why?)
  - $L_{log}(\hat{\mathbf{y}}, \mathbf{y}) = \log(1 + exp(-(\hat{y}_t - \hat{y}_k))$
- Categorical cross-entropy loss:
  - Used for multiclass tasks and when the scores are probabilities (most common for us)
  - $L_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$
  - For "hard" classification:

    $L_{cross-entropy(\text{hard classification})}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_t)$

## Cross-entropy vs. MSE

- MSE's derivative depends on the derivative of the activation, which could be very small for sigmoid
  - Learns slowly
- Cross-entropy's derivative depends on the error: the larger the error, the faster it learns
  - Learns faster where it matters
- Let's visualize this: http://neuralnetworksanddeeplearning.com/chap3.html

## Optimizers: SGD

**Algorithm 1** Online Stochastic Gradient Descent Training

**Input:** Function $f(\mathbf{x}; \theta)$ parameterized with parameters $\theta$.
**Input:** Training set of inputs $\mathbf{x_1}, \ldots, \mathbf{x_n}$ and outputs $\mathbf{y_1}, \ldots, \mathbf{y_n}$.
**Input:** Loss function $L$.
**while** stopping criteria not met **do**
  Sample a training example $\mathbf{x_i}, \mathbf{y_i}$
  Compute the loss $L(f(\mathbf{x_i}; \theta), \mathbf{y_i})$
  $\hat{\mathbf{g}} \leftarrow$ gradients of $L(f(\mathbf{x_i}; \theta), \mathbf{y_i})$ w.r.t $\theta$
  $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$
**return** $\theta$

## Optimizers: SGD

**Algorithm 2** Minibatch Stochastic Gradient Descent Training

**Input:** Function $f(\mathbf{x};\theta)$ parameterized with parameters $\theta$.
**Input:** Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
**Input:** Loss function $L$.
**while** stopping criteria not met **do**
  Sample a minibatch of $m$ examples $\{(\mathbf{x_1},\mathbf{y_1}),\ldots,(\mathbf{x_m},\mathbf{y_m})\}$
  $\hat{\mathbf{g}} \leftarrow 0$
  **for** $i = 1$ to $m$ **do**
    Compute the loss $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$
    $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} +$ gradients of $\frac{1}{m}L(f(\mathbf{x_i};\theta),\mathbf{y_i})$ w.r.t $\theta$
  $\theta \leftarrow \theta + \eta_k\hat{\mathbf{g}}$
**return** $\theta$

## Optimizers: SGD

**Algorithm 2** Minibatch Stochastic Gradient Descent Training

**Input:** Function $f(\mathbf{x};\theta)$ parameterized with parameters $\theta$.
**Input:** Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
**Input:** Loss function $L$.
**while** stopping criteria not met **do**
  Sample a minibatch of $m$ examples $\{(\mathbf{x_1},\mathbf{y_1}),\ldots,(\mathbf{x_m},\mathbf{y_m})\}$
  $\hat{\mathbf{g}} \leftarrow 0$
  **for** $i = 1$ to $m$ **do**
    Compute the loss $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$
    $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} +$ gradients of $\frac{1}{m}L(f(\mathbf{x_i};\theta),\mathbf{y_i})$ w.r.t $\theta$
  $\theta \leftarrow \theta + \eta_k\hat{\mathbf{g}}$
**return** $\theta$

Easy to parallelize on a GPU!

## Other optimizers

- SGD+Momentum, Nesterov Momentum
  - Previous gradients are accumulated
  - The current update uses a combination of current gradient and previous ones

## SGD with momentum

- Vanilla SGD has the tendency to “Tony Hawk” it
- Momentum helps accelerate SGD in the right direction (towards the local minimum)

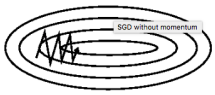



Image 2: SGD without momentum

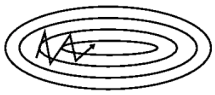


Image 3: SGD with momentum

## Other optimizers

- AdaGrad, AdaDelta, RMSProp, Adam
  - Use an adaptive learning rate for each minibatch, sometimes for each individual coordinate (”feature”)
  - AdaGrad: larger updates for infrequent and smaller updates for frequent parameters (useful for sparse data!)
  - Adam: momentum + individual learning rate per feature

## Weight initialization

- Xavier initialization
  - Small, random weights centered around zero:
  - $\mathbf{W} \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_{in}+d_{out}}}, +\frac{\sqrt{6}}{\sqrt{d_{in}+d_{out}}}\right]$

## Vanishing and exploding gradients

- Explode: gradients become exceedingly close to 1
  - More common in recurrent (additive) networks (we haven't covered these yet)
  - Solution: gradient "clipping", when they exceed a threshold
- Vanish: gradients become exceedingly close to 0
  - Why is this happening?
  - Solution: shallower networks, step-wise training, special architectures (e.g., LSTM – coming later)

## Saturated and dead neurons

- Tanh and sigmoid can become "saturated" – values very close to 1
  - Why is this a problem?
- ReLU activation can "die" – values of 0
  - Why is this a problem?
- Solutions:
  - Change the initialization of weights
  - Change your features, i.e., the range of values assigned to them
  - Reduce learning rate

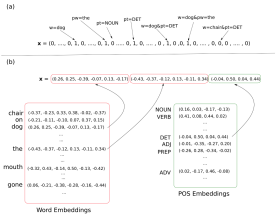
## Shuffling

- Always do it
  - Randomly
  - Curriculum learning: start with easy examples first
- Why?

## Learning rate

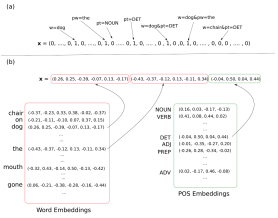
- The learning rate is important to avoid:
  - "Tony Hawking" the data, or
  - Take a very long time to converge
- Solutions:
  - Manual tuning: try at least 0.001, 0.01, 0.1, 1
  - Learning rate scheduling:
    - Decreases the learning rate as a function of the number of observed minibatches:
    - $\eta_t = \eta_0(1 + \eta_0\lambda t)^{-1}$

## A note on dense vs. sparse representations




**Sparse vs. dense feature representations**. Two encodings of the information: *current word is "dog"; previous word is "the"; previous pos-tag is "DET"*. (a) Sparse feature vector. Each dimension represents a feature. Feature combinations receive their own dimensions. Feature values are binary. Dimensionality is very high. (b) Dense, embeddings-based feature vector. Each core feature is represented as a vector. Each feature corresponds to several input vector entries. No explicit encoding of feature combinations. Dimensionality is low. The feature-to-vector mappings come from an embedding table.

## A note on dense vs. sparse representations




**Sparse vs. dense feature representations**. Two encodings of the information: *current word is "dog"; previous word is "the"; previous pos-tag is "DET"*. (a) Sparse feature vector. Each dimension represents a feature. Feature combinations receive their own dimensions. Feature values are binary. Dimensionality is very high. (b) Dense, embeddings-based feature vector. Each core feature is represented as a vector. Each feature corresponds to several input vector entries. No explicit encoding of feature combinations. Dimensionality is low. The feature-to-vector mappings come from an embedding table.

## Readings

- A Primer on Neural Network Models for Natural Language Processing
  - Up to section 6 (including) but skipping section 5 (we will cover that in lecture 3)
  - In D2L, or web: http://u.cs.biu.ac.il/~yogo/nnlp.pdf
- Optional: for a discussion of optimizers: http://ruder.io/optimizing-gradient-descent/
- Optional: for a more advanced discussion (including topics we haven't covered yet), see: http://ruder.io/deep-learning-nlp-best-practices/?utm_campaign=Artificial%2BIntelligence%2BWeekly&utm_medium=email&utm_source=Artificial_Intelligence_Weekly_66

## Take-away

- Multi-class classification
- Regularization
- Other decision functions for NNs
- Other cost functions for NNs
- Optimizers (beyond SGD)
- Weight initialization
- Vanishing and exploding gradients
- Saturated and dead neurons
- Shuffling
- Learning rate
- Dense vs. sparse representations