Lecture 17: Optimization

COMP 411, Fall 2021

Victoria Manfredi

Acknowledgements: These slides are based primarily on slides created by Vivek Srikumar (Utah), Dan Roth (Penn), Sergey Levine (UC Berkeley), and content from the book “Machine Learning” by Tom Mitchell

Today's Topics

Training a neural network

- Motivation
- Notation
- Backpropagation

vumanfredi@wesleyan.edu

Training a Neural Network

MOTIVATION

Training a neural network

Given

- **A network architecture**
  - layout of neurons, neuron connectivity, and neuron activations
- **A dataset of labeled examples**
  - $S = \{(\mathbf{x_i}, y_i)\}$

Goal

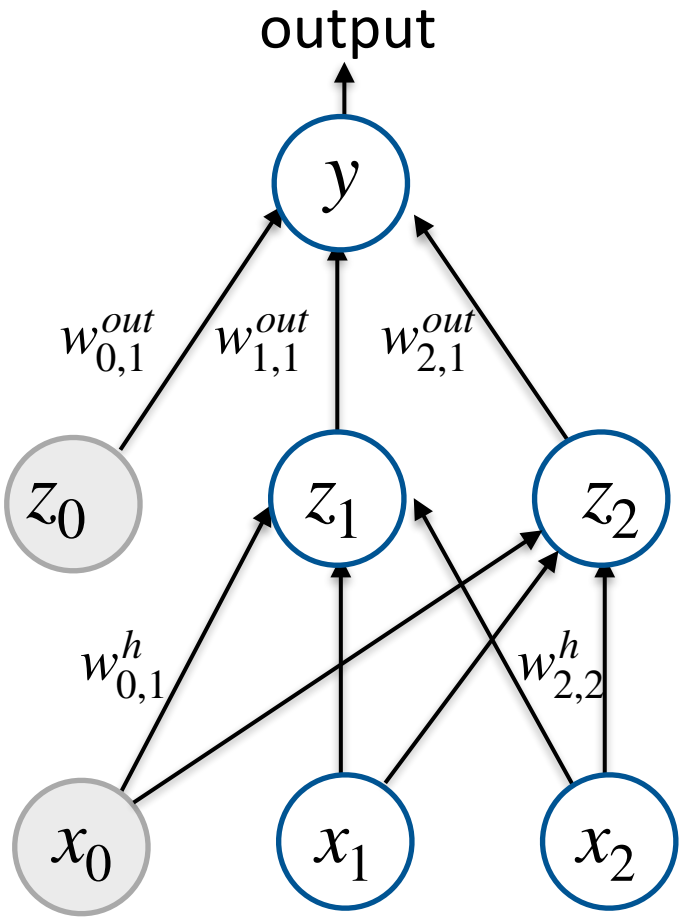
- Learn the weights of the neural network

Remember

- For a fixed architecture, a neural network is a function parameterized by its weights
- Prediction: $y = NN(\mathbf{x}, \mathbf{w})$

Back to our running example

Given an input $\mathbf{x}$, how is the output predicted?



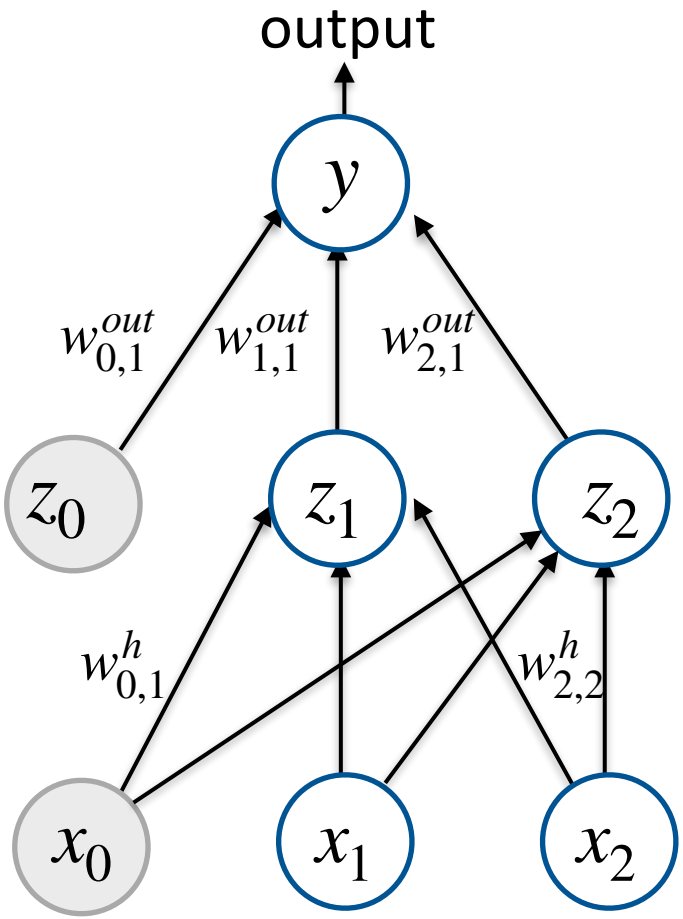
$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

Back to our running example

Given an input $\mathbf{x}$, how is the output predicted?



$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

Suppose the true label for this example is a number y_i

We can write the square loss for this example as:

$$L = \frac{1}{2}(y - y_i)^2$$

Recall: Learning as loss minimization

We have a classifier NN that is completely defined by its weights. Learn the weights by minimizing a loss L

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

How do we solve the optimization problem?

Recall: Learning as loss minimization

We have a classifier NN that is completely defined by its weights. Learn the weights by minimizing a loss L

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

Saw that this strategy worked for perceptron and LMS regression: each minimizes a different loss function.

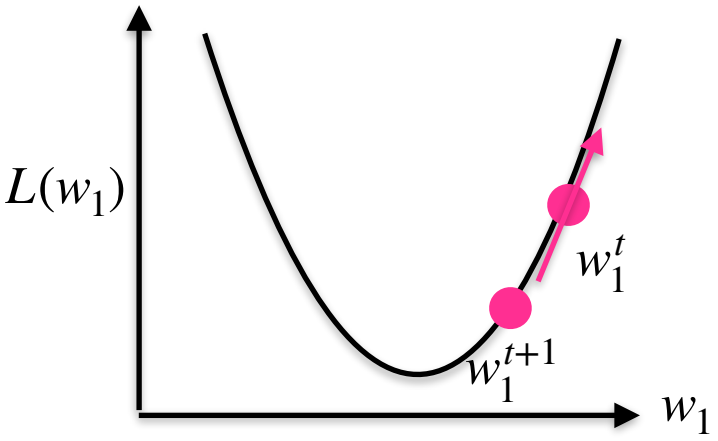
Same idea for non-linear models too!

Gradient descent

An algorithm:

1. Find a direction v where $L(\mathbf{w})$ decreases
2. $\mathbf{w} \leftarrow \mathbf{w} + \alpha v$

Goal is to minimize $L(\mathbf{w})$. Which way does $L(\mathbf{w})$ decrease?



Negative slope / decreasing = want to go to right

Positive slope / increasing = want to go to left

In general: for each dimension, go in the direction opposite the slope along that dimension

$$v_1 = -\frac{dL(\mathbf{w})}{dw_1} \qquad v_1 = -\frac{dL(\mathbf{w})}{dw_2} \qquad \ldots \qquad \nabla_{\mathbf{w}} L(\mathbf{w}) = \begin{bmatrix} \frac{dL(\mathbf{w})}{dw_1} \\ \vdots \\ \frac{dL(\mathbf{w})}{dw_n} \end{bmatrix}$$

Gradient descent

- Measures the local gradient of the error (cost) function with respect to the parameter vector w and it goes in the direction of decreasing gradient. Once the gradient is zero, you have reached a minimum

- When using gradient descent, you should ensure that all features have a similar scale, or else it will take much longer to converge.

- Partial derivative with respect to weight: how much will the cost function change if you change weight just a bit. Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting partial derivative from the weight

Stochastic gradient descent

- The problem with batch gradient descent is that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, stochastic gradient descent picks a random instance in the training set at every step and computes the gradient based on that instance.
  - each training step is much faster but much more stochastic then when using batch gradient descent
  - solution: gradually reduce learning rate

- Mini-batch gradient descent: compute gradients on small random sets of instances

Stochastic gradient descent

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

Given a training set $S = \{(\mathbf{x_i}, y_i)\}, \mathbf{x} \in \mathfrak{R}^{\mathbf{d}}$

1. Initialize parameters $\mathbf{w}$
2. For epoch $= 1 \ldots T$:
   - Shuffle the training set
   - For each training example $(\mathbf{x_i}, y_i) \in S$:
     - ➡ Treat this example as the entire dataset

       Compute the gradient of the loss $\nabla L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$

     - ➡ Update $\mathbf{w} \leftarrow \mathbf{w} - \gamma_t \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$

γ_t: learning rate, many tweaks possible

Return $\mathbf{w}$

Stochastic gradient descent

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

The objective is not convex
Initialization can be important

Given a training set $S = \{(\mathbf{x_i}, y_i)\}, \mathbf{x} \in \mathfrak{R}^{\mathbf{d}}$

1. Initialize parameters $\mathbf{w}$
2. For epoch $= 1 \ldots T$:
   - Shuffle the training set
   - For each training example $(\mathbf{x_i}, y_i) \in S$:
     - ➡ Treat this example as the entire dataset

       Compute the gradient of the loss $\nabla L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$

     - ➡ Update $\mathbf{w} \leftarrow \mathbf{w} - \gamma_t \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$

γ_t: learning rate, many tweaks possible

Return $\mathbf{w}$

Stochastic gradient descent

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

The objective is not convex
Initialization can be important

Given a training set $S = \{(\mathbf{x_i}, y_i)\}, \mathbf{x} \in \mathfrak{R}^{\mathbf{d}}$

1. Initialize parameters $\mathbf{w}$
2. For epoch $= 1 \ldots T$:
   - Shuffle the training set
   - For each training example $(\mathbf{x_i}, y_i) \in S$:
     - ➡ Treat this example as the entire dataset

       Compute the gradient of the loss $\nabla L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$

     - ➡ Update $\mathbf{w} \leftarrow \mathbf{w} - \gamma_t \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$

γ_t: learning rate, many tweaks possible

Return $\mathbf{w}$

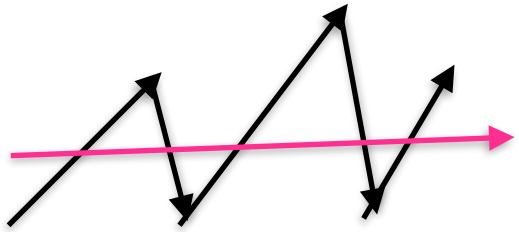
Training a Neural Network

MOMENTUM

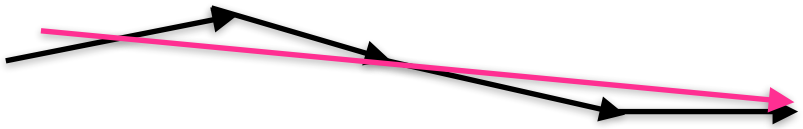
Momentum

Averaging together successive gradients seem to yield a much better direction

Intuition: if successive gradients steps point in different directions we should cancel off the directions that disagree



If successive gradient steps point in similar directions, we should go faster in that direction



Momentum

Update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \gamma_t g_k$$

before: $g_k = \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$

now: $g_k = \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i) + \mu g_{k-1}$ ("blend in" previous direction)

Algorithms

RMSProp: Estimate per-dimension magnitude (running average), then divide each dimension by its magnitude

AdaGrad: Estimate per-dimension cumulative magnitude, then divide each dimension by its magnitude

Adam: Combine momentum and RMSProp

Training a Neural Network

BACKPROPAGATION

The derivative of the loss function?

$$\nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$$

If the neural network is a differentiable function, we can find the gradient

- Or maybe its sub-gradient
- This is decided by the activation functions and the loss function

Easy if only one layer. But how do find the sub-gradient of a more complex function?

- E.g., 150 layer neural network for image classification!

We need an efficient algorithm:
Backpropagation