Mathematical techniques in data science

Lecture 21: Neural networks

April 12th, 2019

Schedule

Week	Chapter
1	Chapter 2: Intro to statistical learning
3	Chapter 4: Classification
4	Chapter 9: Support vector machine and kernels
5, 6	Chapter 3: Linear regression
7	Chapter 8: Tree-based methods + Random forest
8	
9	**Neural networks**
12	PCA $\rightarrow$ Manifold learning
11	Clustering: K-means $\rightarrow$ Spectral Clustering
10	Bayesian methods + UQ
13	Reinforcement learning/Online learning/Active learning
14	Project presentation

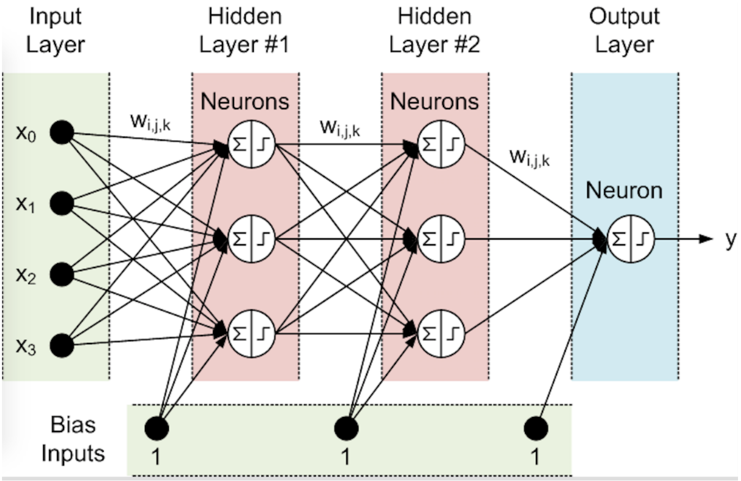
Feed-forward neural networks

- Structure:
  - Graphical representation
  - Activation functions
  - Loss functions
- Training:
  - Stochastic gradient descent
  - Back-propagation

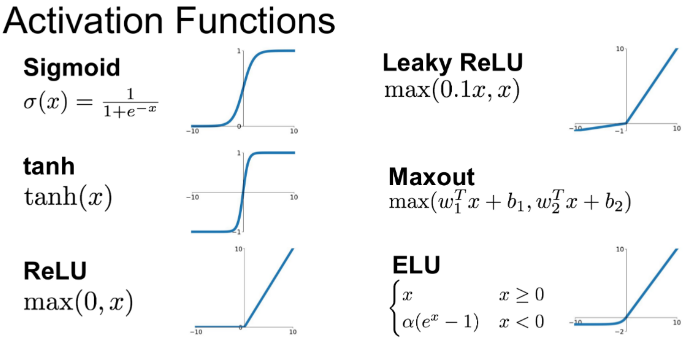
Supervised learning: standard setting

- Hypothesis space: the space of possible functions that the model describes
  For feed-forward neural nets, we need to specify
    - Network's graphical structure: number of layers, number of nodes in each layers, connections between the layers
    - Activation function used in each layers
- In order to measure how well a function fits the training data, a *loss function* needs to be defined

Graphical representation



Activation functions



Module: tf.keras.activations

Functions

`deserialize(...)`

`elu(...)`: Exponential linear unit.

`exponential(...)`

`get(...)`

`hard_sigmoid(...)`: Hard sigmoid activation function.

`linear(...)`

`relu(...)`: Rectified Linear Unit.

`selu(...)`: Scaled Exponential Linear Unit (SELU).

`serialize(...)`

`sigmoid(...)`

`softmax(...)`: Softmax activation function.

`softplus(...)`: Softplus activation function.

Loss functions

- For regression: mean squared error, mean absolute error
- For classification: cross-entropy

Module: tf.keras.losses

Classes

`class BinaryCrossentropy` : Computes the binary cross entropy loss between the labels and predictions.

`class CategoricalCrossentropy` : Computes categorical cross entropy loss between the `y_true` and `y_pred` .

`class MeanAbsoluteError` : Computes the mean of absolute difference between labels and predictions.

`class MeanAbsolutePercentageError` : Computes the mean absolute percentage error between `y_true` and `y_pred` .

`class MeanSquaredError` : Computes the mean of squares of errors between labels and predictions.

`class MeanSquaredLogarithmicError` : Computes the mean squared logarithmic error between `y_true` and `y_pred` .

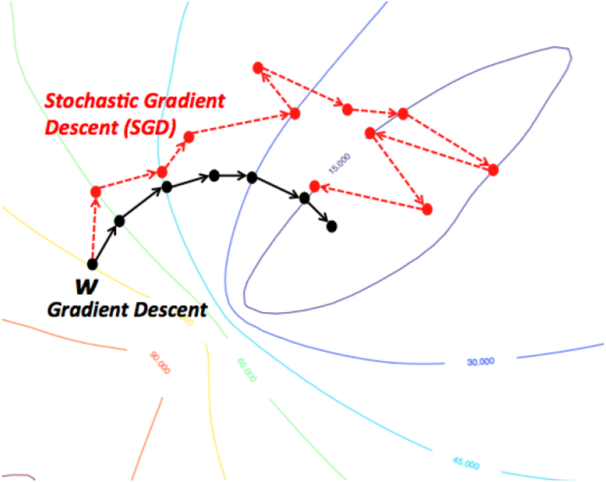
Training of feed-forward neural nets

- Stochastic gradient descent
- Back-propagation

Stochastic gradient descent

- Mini-batch stochastic gradient descent
  - randomly shuffle examples in the training set, divide them into k mini-batches of data of size m
  - for each batch I_i (i=1, . . . , k), approximate the empirical risk by
    $$\hat{\mathcal{L}}(w) = \frac{1}{m} \sum_{j \in I_i} L(w, x_j, y_j)$$
    and update w by gradient descent
  - Repeat these steps M times
- Terminology:
  - m: batch-size
  - k: iteration
  - M: number of epochs

Stochastic gradient descent (SGD)



- The most computationally heavy part in the training of a neural net is to compute

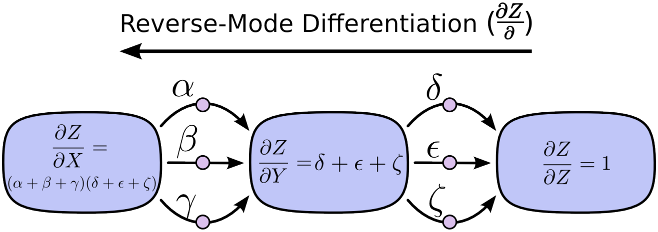
$$\frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

- Numerical differentiation is not realistic, and symbolic differentiation is impossible

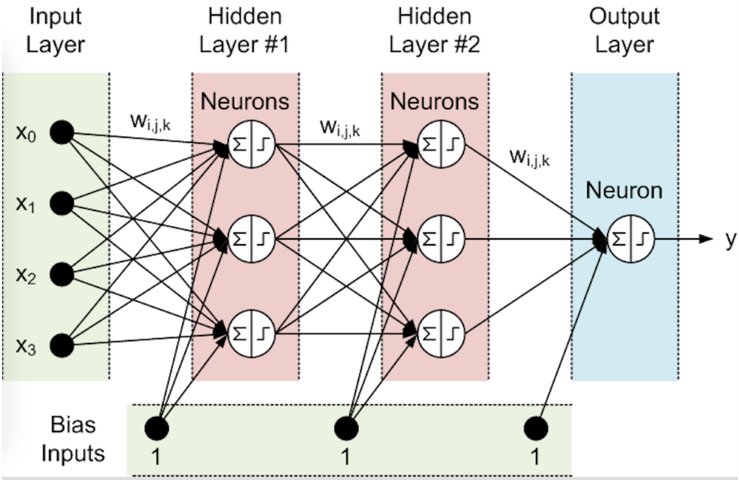
Automatic differentiation

- Assume that
$$y = f(g(h(x)))$$
- Denote $x = u_0$, $h(u_0) = u_1$, $g(u_1) = u_2$, $f(u_2) = u_3 = y$, then
$$\frac{dy}{du_i} = \frac{dy}{du_{i+1}} \frac{du_{i+1}}{du_i}$$
- To compute the derivatives of y with respect to all the u_i's, we need
  - a forward run: compute the values of the u_i's, starting from $u_0 = x$
  - a backward run: compute the values of $\frac{dy}{du_i}$, starting from $\frac{dy}{du_3} = 1$

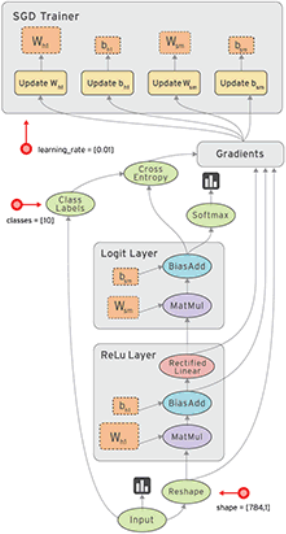
Automatic differentiation



Feed-forward neural networks



Computational graph



Back-propagation

- Advantage: The cost to compute the partial derivatives with respect to all parameters are just twice the cost of a forward evaluations
- Drawback: The functions used to describe the network (activation functions and loss functions) needs to belong to the class of functions supported by the computational platform

Setting up with tensorflow.keras

tf.keras.models.Sequential

- Using the function Sequential to (sequentially) add the layers

$$model = tf.keras.models.Sequential(\ldots)$$

- For each layer, we specify
  - the shape of input (for the first hidden layer)
  - the shape of output
  - the activation function
- For simple feed forward neural net, the only type of layer to use is 'Dense'

Training

- Use model.compile() to specify
  - the optimizer
  - the loss function
  - a metric of accuracy
- Use model.fit() to specify
  - x_{train} and y_{train}
  - number of epochs (default:1)
  - batch size (default: 32)
  - learning rate

Module: tf.keras.optimizers

Contents

Defined in `tensorflow/_api/v1/keras/optimizers/__init__.py`.

Built-in optimizer classes.

Classes

`class Adadelta`: Adadelta optimizer.

`class Adagrad`: Adagrad optimizer.

`class Adam`: Adam optimizer.

`class Adamax`: Adamax optimizer from Adam paper's Section 7.

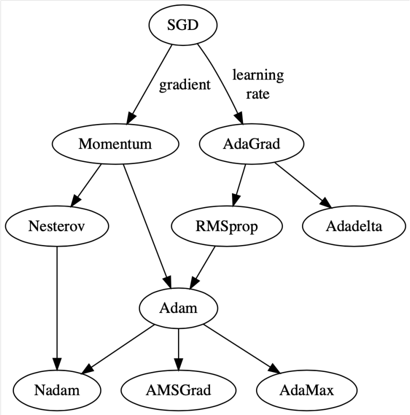
`class Nadam`: Nesterov Adam optimizer.

`class Optimizer`: Abstract optimizer base class.

`class RMSprop`: RMSProp optimizer.

`class SGD`: Stochastic gradient descent optimizer.

Optimizers



Reading: an overview of gradient descent algorithms
http://ruder.io/optimizing-gradient-descent/

General ideas

- Vanilla gradient descent updates parameters by

$$\theta = \theta - \eta \nabla \hat{J}(\theta)$$

  where θ is the learning rate
- Problems
  - choosing η is difficult
  - same learning rate applies to all parameter updates $\rightarrow$ inefficiency
  - on high dimension, not only SGD might be stuck at local minima, it may also be stuck at saddle points
- Ideas
  - use momentum
  - adjust the learning rate

Momentum

- SGD-momentum

$$\begin{aligned} v_t &= \gamma v_{t-1} + (1-\gamma)\nabla_\theta J(\theta) \\ \theta &= \theta - \eta v_t \end{aligned}$$

- Nesterov Accelerated Gradient (NAG)

$$\begin{aligned} \theta^* &= \theta - \eta v_{t-1} \\ v_t &= \gamma v_{t-1} + (1-\gamma)\nabla_\theta J(\theta^*) \\ \theta &= \theta - \eta v_t \end{aligned}$$

Dimension-specific learning rate

- Adagrad

  $$\theta = \theta - \frac{\eta}{\sqrt{S_t + \epsilon}} \nabla \hat{J}(\theta)$$

  where

  $$S_{t,i} = S_{t-1,i} + (\nabla_i \hat{J}(\theta))^2$$

- RMSProp:
  Same idea, but S_t is a decaying average of the gradients

  $$S_{t,i} = \gamma S_{t-1,i} + (1-\gamma)(\nabla_i \hat{J}(\theta))^2$$

  $\rightarrow$ similar to the idea of momentum
  $\rightarrow$ referred to as second order moment