Neural Networks, Computation Graphs

CMSC 470
Marine Carpuat
Binary Classification with a Multi-layer Perceptron

\[ \begin{align*}
\phi_{"A"} &= 1 \\
\phi_{"site"} &= 1 \\
\phi_{"located"} &= 1 \\
\phi_{"Maizuru"} &= 1 \\
\phi_{"in"} &= 2 \\
\phi_{"Kyoto"} &= 1 \\
\phi_{"priest"} &= 0 \\
\phi_{"black"} &= 0 \\
\end{align*} \]
Example: binary classification with a NN

\[ \phi_0(x_1) = \{-1, 1\}, \quad \phi_0(x_2) = \{1, 1\} \]

\[ \phi_0(x_3) = \{-1, -1\}, \quad \phi_0(x_4) = \{1, -1\} \]

\[ \phi_1(x_1) = \{-1, 1\}, \quad \phi_1(x_2) = \{1, -1\} \]

\[ \phi_1(x_3) = \{-1, 1\}, \quad \phi_1(x_4) = \{-1, -1\} \]
Example: the Final Net

Replace “sign” with smoother non-linear function (e.g. tanh, sigmoid)
Multi-layer Perceptrons are a kind of “Neural Network” (NN)

- Input (aka features)
- Output
- Nodes (aka neuron)
- Layers
- Hidden layers
- Activation function (non-linear)
Neural Networks as Computation Graphs

Example & figures by Philipp Koehn
Computation Graphs Make Prediction Easy: Forward Propagation
Computation Graphs Make Prediction Easy: Forward Propagation
Neural Networks as Computation Graphs

• Decomposes computation into simple operations over matrices and vectors

• Forward propagation algorithm
  • Produces network output given an output
  • By traversing the computation graph in topological order
Neural Networks for Multiclass Classification
Multiclass Classification

- The softmax function

\[
P(y \mid x) = \frac{e^{w \cdot \phi(x, y)}}{\sum_{\tilde{y}} e^{w \cdot \phi(x, \tilde{y})}}
\]

Exact same function as in multiclass logistic regression
Example: A feedforward Neural Network for 3-way Classification

\[ z = \sigma(\Theta^{(x \rightarrow z)} x) \]

\[ p(y \mid x; \Theta^{(z \rightarrow y)}, b) = \text{SoftMax}(\Theta^{(z \rightarrow y)} z + b) \]

From Eisenstein p66
Designing Neural Networks: Activation functions

- Hidden layer can be viewed as set of hidden features
- The output of the hidden layer indicates the extent to which each hidden feature is “activated” by a given input
- The activation function is a non-linear function that determines range of hidden feature values
Designing Neural Networks: Network structure

• 2 key decisions:
  • Width (number of nodes per layer)
  • Depth (number of hidden layers)

• More parameters means that the network can learn more complex functions of the input
Neural Networks so far

• Powerful non-linear models for classification
• Predictions are made as a sequence of simple operations
  • matrix-vector operations
  • non-linear activation functions
• Choices in network structure
  • Width and depth
  • Choice of activation function
• Feedforward networks (no loop)
• Next: how to train?
Training Neural Networks
How do we estimate the parameters (aka “train”) a neural net?

For training, we need:

• Data: (a large number of) examples paired with their correct class (x,y)
• Loss/error function: quantify how bad our prediction \( y \) is compared to the truth \( t \)
  • Let’s use squared error:

\[
\text{error} = \frac{1}{2}(t - y)^2
\]
Stochastic Gradient Descent

• We view the error as a function of the trainable parameters, on a given dataset

• We want to find parameters that minimize the error

\[ w = 0 \]

for / iterations
  for each labeled pair \( x, y \) in the data
    \[ w = w - \mu \frac{d_{\text{error}}(w, x, y)}{dw} \]

Start with some initial parameter values

Go through the training data one example at a time

Take a step down the gradient
Computation Graphs Make Training Easy: Computing Error
Computation Graphs Make Training Easy: Computing Gradients

\[
\frac{dE}{dA} = \frac{dE}{dB} \frac{dB}{dA}
\]
Computation Graphs Make Training Easy: Given forward pass + derivatives for each node
Computation Graphs Make Training Easy: Computing Gradients
Computation Graphs Make Training Easy: Computing Gradients
Computation Graphs Make Training Easy: Updating Parameters

\[
W_1 = \begin{bmatrix} 3.7 & 3.7 \\ 2.9 & 2.9 \end{bmatrix} - \mu \begin{bmatrix} .0171 & 0 \\ -.0308 & 0 \end{bmatrix}
\]

\[
b_1 = \begin{bmatrix} -1.5 \\ -4.6 \end{bmatrix} - \mu \begin{bmatrix} .0382 & .00712 \end{bmatrix}
\]

\[
W_2 = \begin{bmatrix} 4.5 & -5.2 \end{bmatrix} - \mu \begin{bmatrix} .0171 \\ -.0308 \end{bmatrix}
\]

\[
b_2 = \begin{bmatrix} -2.0 \end{bmatrix} - \mu \begin{bmatrix} .0424 \end{bmatrix}
\]

\[
x \quad i_2, i_1 \quad \text{prod} \quad \text{sum} \\ 1, 1 \quad \text{sigmoid} \quad \sigma'(i) \quad \text{prod} \\ i_2, i_1 \quad \text{sum} \\ 1, 1 \quad \text{sigmoid} \quad \sigma'(i) \quad \text{L2} \\ i_2 - i_1 \quad t
\]
Computation Graph: A Powerful Abstraction

• To build a system, we only need to:
  • Define network structure
  • Define loss
  • Provide data
  • (and set a few more hyperparameters to control training)

• Given network structure
  • Prediction is done by forward pass through graph (forward propagation)
  • Training is done by backward pass through graph (back propagation)
  • Based on simple matrix vector operations

• Forms the basis of neural network libraries
  • Tensorflow, Pytorch, mxnet, etc.
Neural Networks

• Powerful non-linear models for classification
• Predictions are made as a sequence of simple operations
  • matrix-vector operations
  • non-linear activation functions
• Choices in network structure
  • Width and depth
  • Choice of activation function
• Feedforward networks (no loop)
• Training with the back-propagation algorithm
  • Requires defining a loss/error function
  • Gradient descent + chain rule
  • Easy to implement on top of computation graphs