Reliable and Interpretable Artificial Intelligence

Dr. Dana Drachsler-Cohen
ETH Zurich

Fall 2017
So far: Synthesis

First, studied algorithms that accurately solve a problem
  e.g., given specification or by learning an intent from examples
  accuracy is guaranteed, obtaining a specification is not trivial
Then, studied synthesis algorithms that use statistical information
  e.g., Slang, Neural Turing Machine
  models with high accuracy, require labeled data
  Does not impose burden on the user
We will learn that these models are easily fooled, i.e., they are not robust
Part II: Robustness of Deep Learning
Examples of (lack of) Robustness

\[ x + \epsilon \text{sign}(\nabla_x J(\theta, x, y)) \]

\[ x \]

"panda"  
57.7% confidence

\[ \text{sign}(\nabla_x J(\theta, x, y)) \]

"nematode"  
8.2% confidence

\[ x + \epsilon \text{sign}(\nabla_x J(\theta, x, y)) \]

"gibbon"  
99.3% confidence

taken from: Explaining and Harnessing Adversarial Examples. Goodfellow, Shlens, Szegedy, ICLR '15
Today: Background on Deep Learning
(the background needed to understand robustness)
Technical Material (Today)

Perceptron

(Empirical) Loss functions

Training with gradient descent

Deep learning models

Back propagation
Motivation: Classify Handwritten Digits

Dataset: MNIST
Images consist of 28 x 28 pixels (hence, 784 pixels)

The space of possible images is: $[0,1]^{784}$ (not countable)

A black pixel has value 0, a white pixel has value 1

Goal: a model that classifies images to digits: $f: [0,1]^{784} \rightarrow \{0, \ldots, 9\}$
Classification through Machine Learning

Take a **data-driven approach** and learn a **model** (function) \( f \) from data

\[ f: I \rightarrow C \text{ approximates} \text{ the optimal function } f^*: I \rightarrow C \]

A model is an **architecture** with real-valued **weights** and **biases**

The architecture defines the space of expressible models
Example of a Model $f$: A Perceptron

A classifier parametrized by $w_0, ..., w_{n-1}$ and $b$

$$\tilde{x} = (x_0, ..., x_{n-1})$$

$$f(x) = \begin{cases} 1 & \Sigma w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Linearly separates the space
Evaluating Models: Loss Function

Goal: model and optimal classifier are equal:

$$\forall i. f^*(i) = f(i)$$

Induces a loss function which measures how good a classifier is:

$$\sum_{i \in I} [f^*(i) \neq f(i)]$$

where \([\cdot]\) is the Iverson brackets (1 if condition is true, 0 otherwise)

The smaller the loss, the better the classifier

Goal: find weights and biases of the model \(f\) which minimize loss
Empirical Loss

Practical challenge: not all labels of input data $I$ are given

Approach: estimate the loss function on some of the labelled inputs $D$

Given the labelled data $D$, compute the empirical loss: $\Sigma_{i \in D}[f^*(i) \neq f(i)]$

Operationally:

To avoid overfitting to $D$, split $D$ into training $D_{Tr}$ and test $D_{Te}$ sets
Learn model by minimizing the loss on $D_{Tr}$, estimate loss on $D_{Te}$
Tuning Model’s Parameters

The optimal solution of $\Sigma_{i \in D_{Tr}} [f^*(i) \neq f(i)]$ is a global minima

If $\Sigma_{i \in D_{Tr}} [f^*(i) \neq f(i)]$ were differentiable, we could compute the minima by checking when the derivative is zero

Define (a different) loss function that is differentiable and find a point that nullifies its derivative
Example of Differentiable Loss Function

The mean squared error (MSE):

\[ \text{MSE} = \sum_{i \in D_{Tr}} (f^*(i) - f(i))^2 \]

If the model consists of a single weight: \( f(i) = w \cdot i \), then:

\[ \text{MSE} = \sum_{i \in D_{Tr}} (f^*(i) - w \cdot i)^2 \]

The minimum nullifies the derivative:

\[ \sum_{i \in D_{Tr}} 2(f^*(i) - w \cdot i) \cdot (-i) = 0 \]

Can compute the best model (i.e., \( w \)) analytically.
Gradients

For $f(w_0, \ldots, w_{n-1}, b)$ with (>1 parameter), derivative of the

$\text{MSE} = \sum_{i \in D_{Tr}} (f^*(i) - f(i))^2$ is generalized to gradient

A gradient is a vector defined by partial derivatives of the variables

$$\nabla \text{MSE} = \left( \frac{\partial \text{MSE}}{\partial w_0}, \ldots, \frac{\partial \text{MSE}}{\partial w_{n-1}}, \frac{\partial \text{MSE}}{\partial b} \right)$$

Minimum nullifies $\nabla \text{MSE}$ in all dimensions $\rightarrow$ Hard to compute analytically
Example: Mean Squared Error Gradients

Example: \( f(i_1, i_2) = w_1 \cdot i_1 + w_2 \cdot i_2 + b \)

\[ \nabla \text{MSE} = \left( \sum_{(i_1, i_2) \in D_{Tr}} 2(f^*(i_1, i_2) - (w_1 \cdot i_1 + w_2 \cdot i_2 + b)) \cdot (-i_1), \right. \]

\[ \frac{\partial \text{MSE}}{\partial w_1} \]

\[ \left. \sum_{(i_1, i_2) \in D_{Tr}} 2(f^*(i_1, i_2) - (w_1 \cdot i_1 + w_2 \cdot i_2 + b)) \cdot (-i_2), \right. \]

\[ \frac{\partial \text{MSE}}{\partial w_2} \]

\[ \sum_{(i_1, i_2) \in D_{Tr}} 2(f^*(i_1, i_2) - (w_1 \cdot i_1 + w_2 \cdot i_2 + b)) \cdot (-1) \]

\[ \frac{\partial \text{MSE}}{\partial b} \]

\[ \text{MSE} = \sum_{(i_1, i_2) \in D_{Tr}} (f^*(i_1, i_2) - f(i_1, i_2))^2 \]
Finding Minima via Gradient Descent

1. Initialize randomly with certain values for weights/bias $a_0$, set $n = 0$

2. Compute the gradient of the MSE at $a_n$

3. The next point is the one maximizing the decrease in MSE

   $$a_{n+1} = a_n - \gamma \nabla MSE(a_n)$$  \hspace{1em} (\gamma \text{ is called the learning rate})

   $$n = n + 1$$

4. If loss is small enough, complete; otherwise, repeat 2
Recap

Models approximate a function (that is hard to find algorithmically)

A model is an architecture (e.g., perceptron) and a set of parameters

Parameters are tuned to have lowest loss via gradient descent

(A very simplified introduction, omits many important details: regularization, weight initialization, validation set, …)
Deep Learning

Perceptrons are too simplistic models

Deep models **combine** multiple simple models (e.g., perceptron)

A deep model is a directed graph of **neurons**

Neurons are organized in **layers**

A neuron is a simple model followed by an **activation function**
Activation Functions

Determine whether to propagate the output of the neuron

Examples: tangent, sigmoid, ReLU

\[ ReLU(a) = \max(0, a) \]
Neurons and Layers: Notation

Layer 0: Input Layer

Layer 1

Layer 2

\[ p_1^1 = w_{1,1}^1 \cdot l_1^0 + w_{1,2}^1 \cdot l_2^0 + b_1^1 \]

\[ l_1^1 = \text{ReLU}(p_1^1) \]

\[ p_2^1 = w_{1,1}^2 \cdot l_1^1 + w_{1,2}^2 \cdot l_2^1 + b_1^2 \]

\[ l_2^1 = \text{ReLU}(p_2^1) \]

\[ p_1^2 = w_{2,1}^1 \cdot l_1^2 + w_{2,2}^1 \cdot l_2^2 + b_2^1 \]

\[ l_1^2 = \text{ReLU}(p_1^2) \]

\[ p_2^2 = w_{1,1}^2 \cdot l_1^1 + w_{1,2}^2 \cdot l_2^1 + b_1^2 \]
Feed Forward Neural Network (FF NN)

Neurons are connected to all neurons in the **next layer**

In theory:

Any continuous function over the real numbers can be **approximated** with an FF NN “as much as we want”
Computing Outputs of Deep Models

To compute output, **feed forward** the input through the network

\[
f(l^0_1) = ReLU(w^2_{1,1} \cdot l^1_1 + w^2_{1,2} \cdot l^1_2 + b^2_1) = \\
ReLU(w^2_{1,1} \cdot ReLU(w^1_{1,1} \cdot l^0_1 + b^1_1) + w^2_{1,2} \cdot ReLU(w^1_{2,1} \cdot l^0_1 + b^1_1) + b^2_1
\]
Gradient Descent in Deep Models

Deep model can be tuned by gradient descent

Need to compute the partial derivatives of all weights and biases

Hard to compute
Gradients in Deep Models: Difficulty

\[ p_1^1 = w_{1,1}^1 l_1^0 + b_1^1 \]
\[ l_1^1 = \text{ReLU}(p_1^1) \]

\[ p_2^1 = w_{2,1}^1 l_1^0 + b_2^1 \]
\[ l_2^1 = \text{ReLU}(p_2^1) \]

\[ p_2^2 = w_{1,2}^2 l_1^1 + b_2^1 \]
\[ l_2^2 = \text{ReLU}(p_2^2) \]

\[ f(l_1^0) \]

\[ MSE(l_1^0) = \sum_{l_1^0 \in D_{tr}} (f^*(l_1^0) - l_1^2)^2 \]

\[
\frac{\partial MSE}{\partial w_{1,1}^2} = \frac{\partial MSE}{\partial l_1^2} \cdot \frac{\partial l_1^2}{\partial p_1^2} \cdot \frac{\partial p_1^2}{\partial w_{1,1}^2}
\]

\[
\frac{\partial MSE}{\partial w_{1,1}^1} = \frac{\partial MSE}{\partial l_1^2} \cdot \frac{\partial l_1^2}{\partial p_1^2} \cdot \frac{\partial p_1^2}{\partial l_1^1} \cdot \frac{\partial l_1^1}{\partial p_1^1} \cdot \frac{\partial p_1^1}{\partial w_{1,1}^1}
\]
Gradients in Deep Models: Difficulty

\[ p^1_1 = w^1_{1,1} l^0_1 + b^1_1 \]
\[ l^1_1 = ReLU(p^1_1) \]

\[ p^1_2 = w^1_{2,1} l^0_1 + b^1_2 \]
\[ l^1_2 = ReLU(p^1_2) \]

\[ p^2_1 = w^2_{1,1} \cdot l^1_1 + w^2_{1,2} \cdot l^1_2 + b^2_1 \]
\[ l^2_1 = ReLU(p^2_1) \]

\[ p^2_2 = w^2_{2,1} \cdot l^1_2 + b^2_2 \]
\[ l^2_2 = ReLU(p^2_2) \]

\[ f(l^0_1) \]

\[ f^*(l^0_1) \]

\[ \text{MSE}(l^0_1) = \Sigma_{l^0_1 \in D_{Tr}} (f^*(l^0_1) - l^2_1)^2 \]

\[ \frac{\partial \text{MSE}}{\partial w^2_{1,1}} = \frac{\partial \text{MSE}}{\partial l^2_2} \cdot \frac{\partial l^2_2}{\partial p^2_1} \cdot \frac{\partial p^2_1}{\partial w^2_{1,1}} \]

\[ \frac{\partial \text{MSE}}{\partial w^1_{1,1}} = \frac{\partial \text{MSE}}{\partial l^2_2} \cdot \frac{\partial l^2_2}{\partial p^2_1} \cdot \frac{\partial p^2_1}{\partial l^1_2} \cdot \frac{\partial l^1_2}{\partial p^1_1} \cdot \frac{\partial p^1_1}{\partial w^1_{1,1}} \]
(Back)propagates the common parts of the derivatives

\[
\frac{\partial \text{MSE}}{\partial l_1^2} \cdot \frac{\partial l_1^2}{\partial p_1^2} \cdot \frac{\partial p_1^2}{\partial l_1^1} \cdot \frac{\partial l_1^1}{\partial p_1^1} \cdot \frac{\partial p_1^1}{\partial w_{1,1}^1}
\]
Overall

Defined learning models and deep models

Learned to train models through loss function and gradient descent

Learned to use backpropagation to train deep networks

Next Time: robustness of deep learning
Multiclass Classification

Output layer has a neuron for each class

Outputs normalized to a probability distribution with softmax:

\[ P(c_i) = \frac{e^{f_i(x)}}{\sum_j e^{f_j(x)}} \]

For example, a model classifying to digits: \( f: [0,1]^{784} \rightarrow [0,1]^{10} \)

<table>
<thead>
<tr>
<th>Digit</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.7</td>
</tr>
<tr>
<td>1</td>
<td>0.6</td>
</tr>
<tr>
<td>2</td>
<td>0.7</td>
</tr>
<tr>
<td>3</td>
<td>0.8</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Matrix Representation of FF NN

A perceptron computes: 
\[ l_i^{n+1} = \bar{w}_i^{n+1T} \cdot \bar{l}^n + b_i^{n+1} \]

A layer computes 
\[ \bar{l}^{n+1} = W^{n+1} \cdot \bar{l}^n + \bar{b}^{n+1} \] for:
\[ W^{n+1} = \begin{pmatrix} w_{1,1}^{n+1} & w_{1,2}^{n+1} \\ w_{2,1}^{n+1} & w_{2,2}^{n+1} \end{pmatrix}, \quad \bar{b}^{n+1} = \begin{pmatrix} b_1^{n+1} \\ b_2^{n+1} \end{pmatrix} \]