Deep Learning: Overview and basics

ATPESC 2020
Friday 2020-03-19

Kyle Gerard Felker
Postdoctoral Appointee, Aurora Early Science Program
Argonne Leadership Computing Facility
felker@anl.gov
Outline

• High-level introduction to deep learning (25 min)
• Hands-on exercises (20 min)
AI for Science

NERSC: Perlmutter
AMD Epyc + NVIDIA A100s (2020)

ALCF: Aurora
Intel Xeon + Xe GPUs (2021)

NERSC: Perlmutter
AMD Epyc + GPUs (2021)
AI, machine learning, and deep learning

“Machine Learning is a subfield of artificial intelligence with the goal of developing algorithms capable of learning from data automatically.”

“A high-bias, low-variance introduction to Machine Learning for physicists”

**AI, machine learning, and deep learning**

“Machine Learning is a subfield of artificial intelligence with the goal of developing algorithms capable of learning from data automatically.”

“A high-bias, low-variance introduction to Machine Learning for physicists”

**Machine learning algorithms:**
- can improve their accuracy given more data
- are capable of “teaching themselves”
- gain knowledge that is not programmed with an explicit set of rules
Motivations for deep learning
Categories of learning problems or paradigms

- **Supervised learning** *(this talk)*
  - Regression: output variable is continuous
  - Classification: output variable is discrete (categorical)

- **Unsupervised learning** *(Corey and Romit’s talk)*
  - Clustering
  - Association

- **Semi-supervised learning**
  - *Mostly* unlabeled data

- **Reinforcement learning**

**Examples of algorithms**

- **Supervised**: decision tree and random forests
  - [https://towardsdatascience.com/from-a-single-decision-tree-to-a-random-forest-b9523be65147](https://towardsdatascience.com/from-a-single-decision-tree-to-a-random-forest-b9523be65147)

- **Unsupervised**: Variational Autoencoder (VAE)
  - [https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5d5](https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5d5)
Linear regression: ordinary least squares (OLS)

Minimizes the Mean Squared Error (MSE), or quadratic loss:

\[ J = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \]

Closed-form solution exists; one-step procedure: **normal equation**

\[ \mathbf{w}_i = \left( \mathbf{X}^T \mathbf{X} \right)^{-1} \mathbf{X}^T \mathbf{y} \]

Alternative solutions may be required or preferred in some cases

Neuron (linear regression)

Input features $\mathbf{x} \in \mathbb{R}^{n \times 1}$

$z_i = \mathbf{w}_i^T \mathbf{x} + b_i$

Prediction (inference)

$i$th neuron

$z_i, b_i \in \mathbb{R}$

$\mathbf{w} \in \mathbb{R}^{n \times 1}$

*notation varies
Forward pass: compute the loss

Given single labeled training example, \( (x^{(m)}, y^{(m)}) \)
Forward pass: compute the loss

Given single labeled training example, \((x^{(m)}, y^{(m)})\)

\[ z_i = w_i^T x + b_i \]
Forward pass: compute the loss

Given single labeled training example, \( (x^{(m)}, y^{(m)}) \)

\[
z_i = \mathbf{w}_i^T \mathbf{x} + b_i \quad \rightarrow \quad \hat{y}
\]
Forward pass: compute the loss

Given single labeled training example, \((x^{(m)}, y^{(m)})\)

\[ z_i = w_i^T x + b_i \rightarrow \hat{y} \]

Quantify error (loss)

\[ \mathcal{L}(w_i, b_i; x^{(m)}, y^{(m)}) = \left( \hat{y} - y^{(m)} \right)^2 \]
Backward pass: compute the derivatives

Given single labeled training example, \( \left( x^{(m)}, y^{(m)} \right) \)

\[
\mathcal{L}(w_i, b_i; x^{(m)}, y^{(m)}) = \left( \hat{y} - y^{(m)} \right)^2
\]
Backward pass: compute the derivatives

Given single labeled training example, \( \left( x^{(m)}, y^{(m)} \right) \)

The loss function is given by

\[
\mathcal{L}(w_i, b_i; x^{(m)}, y^{(m)}) = \left( \hat{y} - y^{(m)} \right)^2
\]

The derivative is

\[
\frac{\partial \mathcal{L}}{\partial z_i}
\]
Backward pass: compute the derivatives

Given single labeled training example, \((x^{(m)}, y^{(m)})\)

\[
\frac{\partial L}{\partial w_i}, \quad \frac{\partial L}{\partial b_i}, \quad \frac{\partial L}{\partial z_i}
\]

\[
L(w_i, b_i; x^{(m)}, y^{(m)}) = (\hat{y} - y^{(m)})^2
\]
Backward pass: compute the derivatives

Given single labeled training example, \( (x^{(m)}, y^{(m)}) \)

\[
\begin{align*}
\frac{\partial L}{\partial x^{(m)}} &= \frac{\partial L}{\partial w_i} \\
\frac{\partial L}{\partial w_i} &= \frac{\partial L}{\partial b_i} \\
L(w_i, b_i; x^{(m)}, y^{(m)}) &= (\hat{y} - y^{(m)})^2
\end{align*}
\]
Backward pass: compute the derivatives

Given single labeled training example, \((x^{(m)}, y^{(m)})\)

\[
\frac{\partial \mathcal{L}}{\partial x^{(m)}} \quad \frac{\partial \mathcal{L}}{\partial w_i} \quad \frac{\partial \mathcal{L}}{\partial b_i} \quad \frac{\partial \mathcal{L}}{\partial z_i}
\]

\[
\mathcal{L}(w_i, b_i; x^{(m)}, y^{(m)}) = (\hat{y} - y^{(m)})^2
\]

Backpropagation

= chain rule
Backward pass: compute the derivatives

Given single labeled training example, \((x^{(m)}, y^{(m)})\)

\[
\frac{\partial \mathcal{L}}{\partial x^{(m)}} = \frac{\partial \mathcal{L}}{\partial w_i} \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial \mathcal{L}}{\partial b_i}
\]

\[
\mathcal{L}(w_i, b_i; x^{(m)}, y^{(m)}) = (\hat{y} - y^{(m)})^2
\]

Backpropagation = chain rule
Backward pass: compute the derivatives

Given single labeled training example, \((x^{(m)}, y^{(m)})\)

\[
\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial w_i}
\]

\[
\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial b_i}
\]

\[
\frac{\partial z_i}{\partial w_i} = x^{(m)}
\]

\[
\frac{\partial z_i}{\partial b_i} = 1
\]

\[
\frac{\partial z_i}{\partial z_i} = 2(z_i - y)
\]

Backpropagation = chain rule
Training neuron weights with gradient descent

Let $\theta \equiv \{w_i, b_i\}$ trainable parameters

**Cost**: average loss over training examples

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\theta; x^{(m)}, y^{(m)})$$

- Initialize $\theta$ randomly
- For N epochs
  - For each batch of training examples $\{(x_0, y_0), \ldots, (x_b, y_b)\}$:
    * compute loss gradient: $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_{i=1}^{N} \frac{\partial J^i(\theta)}{\partial \theta}$
    * update $\theta$ with update rule:
      $$\theta := \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$
Training neuron weights with gradient descent

Let $\theta \equiv \{w_i, b_i\}$ trainable parameters

**Cost:** average loss over training examples

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\theta; x^{(m)}, y^{(m)})$$

- Initialize $\theta$ randomly
- For N epochs
  - For each batch of training examples $\{(x_0, y_0), \ldots, (x_b, y_b)\}$:
    * compute loss gradient: $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_{i=1}^{N} \frac{\partial J^i(\theta)}{\partial \theta}$
    * update $\theta$ with update rule:
      $$\theta := \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$

$$\eta \equiv \text{learning rate}$$
Global vs. local minima

[Image of a 3D graph showing a function $J(w_1, w_2)$ with global optimum and local optima]

OLS solved with gradient descent

$t = \text{epoch}$

$\text{all} \text{ samples in the training set}$

OLS solved with gradient descent


t = epoch

= neuron sees all samples in the training set
Neuron (generally)

\[ z_i = \mathbf{w}_i^T \mathbf{x} + b_i \]

\[ a_i = \sigma(z_i) \rightarrow \hat{y} \]
Neuron (generally)

\[ z_i = \mathbf{w}_i^T \mathbf{x} + b_i \]

\[ a_i = \sigma(z_i) \rightarrow \hat{y} \]

\[ \sigma(z) = \frac{1}{1 + e^{-z}} \]

= sigmoid = logistic function
Neuron (generally)

\[ z_i = w_i^T x + b_i \]

\[ a_i = \sigma(z_i) \rightarrow \hat{y} \]

\[ \sigma(z) = \frac{1}{1 + e^{-z}} \]

= sigmoid = logistic function

Binary cross-entropy loss function:

\[ \mathcal{L}(\hat{y}, y) = - (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \]
Neuron (generally)

\[
z_i = w_i^T x + b_i
\]

\[
a_i = \sigma(z_i)
\]

\[
\hat{y}
\]

\[
\sigma(z) = \frac{1}{1 + e^{-z}}
\]

= sigmoid = logistic function

**Binary cross-entropy loss function:**

\[
\mathcal{L}(\hat{y}, y) = - (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))
\]
Nonlinear activation functions

FIG. 36  Possible non-linear activation functions for neurons. In modern DNNs, it has become common to use non-linear functions that do not saturate for large inputs (bottom row) rather than saturating functions (top row).
Deep neural network (DNN)

Deep neural network (DNN)

“deep” \( \Rightarrow N > 1 \)

Deep neural network (DNN)

"deep" $\Rightarrow N > 1$

Deep neural network (DNN)

“deep” ⇒ \( N > 1 \)

\( W[l] \in \mathbb{R}^{n[l] \times n[l-1]} \)

Vectorized notation: pass $m$ examples at once

Start with horizontally-stacked input vectors $X \in \mathbb{R}^{n \times m}$
Vectorized notation: pass $m$ examples at once

Start with horizontally-stacked input vectors $X \in \mathbb{R}^{n \times m}$

Vectorized forward propagation
Vectorized notation: pass $m$ examples at once

Start with horizontally-stacked input vectors $X \in \mathbb{R}^{n \times m}$

Vectorized forward propagation

$$Z^{[l]} = W^{[l]} A^{[l-1]} + B^{[l]}$$
Vectorized notation: pass $m$ examples at once

Start with horizontally-stacked input vectors $X \in \mathbb{R}^{n \times m}$

Vectorized forward propagation

$$Z^{[l]} = W^{[l]} A^{[l-1]} + B^{[l]}$$

$$A^{[l]} = \sigma \left( Z^{[l]} \right)$$
Vectorized notation: pass $m$ examples at once

Start with horizontally-stacked input vectors $X \in \mathbb{R}^{n \times m}$

Vectorized forward propagation:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + B^{[l]}$$

$$A^{[l]} = \sigma \left( Z^{[l]} \right)$$

$Z^{[l]}, A^{[l]} \in \mathbb{R}^{n^{[l]} \times m}$
“Layer-by-layer” gradient descent through backpropagation
Convolutional layer
Convolutional layer
Convolutional neural network: VGG-16

https://peltarion.com/
Revolution of Depth

152 layers

ImageNet Classification top-5 error (%)

Kaiming He's ICML 2016 Tutorial
Revolution of size

Revolution of size

175b

* GPT-3
June 2020
Software frameworks

TensorFlow
Keras
PyTorch

Example: PyTorch implementation of 4-layer CNN for MNIST classification

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

https://github.com/pytorch/examples/blob/master/mnist/main.py
Hands-on resources

Google Colaboratory

https://colab.research.google.com/

Tutorial Notebooks

https://github.com/argonne-lcf/ATPESC_MachineLearning/
Thank you!