Neural Networks And Deep Learning Course

AI is the new “The New Electricity”

Gradient descent variants in deep learning: Usually in deep learning we use advanced Gradient Descent optimizers, such as Adam, Momentum Gradient Descent, RMSprop, etc.

End-to-end deep learning: We use a learning algorithm that solves the mapping of $x \rightarrow y$ automatically and holistically. Instead of pre-extracting features manually (hand-crafted features) and use a learning algorithm to create the model, we use a learning algorithm that automatically performs feature learning and fits the model (e.g. Convolutional Neural Networks).

Non-sequence models: MLP, CNN

Sequence models: RNN, LSTM

Recurrent Neural Networks architectures: $\text{void} \rightarrow \text{scalar}$, $\text{void} \rightarrow \text{vec}$, $\text{scalar} \rightarrow \text{vec}$, $\text{scalar} \rightarrow \text{scalar}$, $\text{vec} \rightarrow \text{scalar}$, $\text{vec} \rightarrow \text{vec}$

Structured data: Examples with well-defined features (e.g. house no. bedrooms, sepal length, etc) from datasets such as iris flower or price-house. These examples are usually stored in big tabular databases.

Unstructured data: Examples with not well-defined features (e.g. pixel values, words, audio notes, etc) of raw data for applications such as computer vision, speech recognition and natural language processing.

Human vs Machines as far as structured/unstructured data is concerned: Humans have grown to be good at understanding mostly unstructured data (e.g. vision and auditory systems, human language understanding, etc) instead of extracting knowledge from tabular data. However, machines until the last years were able to understand and predict accurately using structured tabular data. Nowadays, using neural networks and deep learning machines are able also to understand unstructured data (e.g. computer vision, speech recognition, natural language processing, etc).

High Performance Models: To achieve very high performance models we need to use complex deep NNs and large amount of data (as an example we can think of what CNN deep architectures with ImageNet dataset have achieved in computer vision). Scale is the factor that drives the success of deep learning. With a bigger scale of data and NN complexity we can achieve better performances.

Maximum Likelihood Estimation: Minimizing the loss is the same as maximizing the $\log(p(y|x))$.

Vectorization in deep learning: Deep learning algorithms shine when we use a large no. of training examples. A single gradient descent update step needs to loop through all training examples (or a batch of training examples) and run forward/backward pass to calculate the partial derivatives for the weights/parameters. A slow implementation usually uses three loops (a: for the gradient descent iterative algorithm, b: for using the training examples to run forward/backward pass and c) to accumulate gradients for each weight/parameter of the model). A fast implementation usually uses vectorized operations and can actually eliminate the (b) and (c) loops. Vectorization is the art of eliminating “for loops” in calculations. Vectorization is “a must” for deep learning algorithms.

Examples of vectorized operations (forward / backward):

1. Calculating $z^{(i)}$ for a single $x^{(i)}$ example :

   z_i = np.dot(w.T, x_i) + b where w, x_i are vectors and b is a scalar

2. Calculating z values for all examples $x^{(i)}$:

   z = np.dot(w.T, X) + b where w, b are vectors and X is a matrix of $x^{(i)}$ examples

3. Calculate sigmoid activation for all z values :

   s_a = 1.0/(1.0+np.exp(-z)) where z is a vector

4. Calculate the cost for all examples $x^{(i)}$:

   cost = -1/m * np.sum(Y * np.log(A) + (1-Y) * np.log(1-A))

5. Calculate derivatives for w and b :

   dw = 1/m * np.dot(X, (A-Y).T)

   db = 1/m * np.sum(A-Y)

What can be vectorized:

Functions such as log, exp, max, sum can now can be vectorized for vectors, matrixes

```
s = np.exp(-x)
```

Functions such as sigmoid, softmax, derivatives_sigmoid can also be vectorized

```
s = 1.0 / (1.0 + np.exp(-x))
```

Element-wise arithmetic operations, e.g. vector/matrix op scalar/vector/matrix

```
s = M * 3
s = M + v
S = v / v
```

Multiplication of matrix*vector, matrix*matrix, vector*vector

```
np.dot(v, v)
np.dot(M, v)
np.dot(M, N)
```

Reshaping tensors to different shapes

```
v = image.reshape((image.shape[0]*image.shape[1]*image.shape[2]), 1)
```

Fast normalization of data (e.g. x / ||x||)

```
x_norm = np.linalg.norm(x, axis=1, keepdims=True)
```

```
x = x / x_norm
```

Loss functions given grunt-truth and predicted values (supervised learning)

```
loss = sum(abs(yhat – y))

loss = np.dot((yhat - y), (yhat - y))
```

NumPy Broadcasting: In numpy the */-+ and np.multiply() operations between matrixes and vectors, scalars are element-wise. In these operations auto-expanding (broadcasting) is used implicitly. To perform matrix multiplication (inner or cross product) we should use np.dot(a, b) operation.

Normalization / Standardization: The sigmoid activation function gives a useful gradient for values around and close to 0. This is the reason why we should zero-center our data by subtracting the mean (mean standardization) and also normalize them by subtracting either the standard deviation or the range (min-max) of data. By normalizing and standardizing our data we help the learning algorithm to take advantage of the gradient of sigmoid activation functions in order to start learning.

Logistic Regression and MLP: A shallow multilayer perceptron (MLP) is actually a stack of logistic regression nodes.

Why tanh is almost always superior than than sigmoid: We know that performing **mean standardization and normalization** in the input data helps a lot when using the sigmoid activation function because it has a useful gradient **around** and **close** to zero. In other words the standardized and normalized features have mean 0 and are close to it then this **helps the training** of the model. However, the activations of the sigmoid functions from the first layer of deep MLP are not zero-centered and sigmoid activation functions in next hidden layer get non-zero-centered data. Alternatively, using the tanh activation function we can ensure that the activations will be zero-centered in all layers and this makes the **learning faster**. So, in MLPs we can replace all activation functions except the last one (for the case of binary classification, use sigmoid in last node) with tanh activation function.

Also, both tanh and sigmoid are problematic in deep NNs because with large-scale activations the activation functions have no useful gradient to use in backpropagation (close to zero gradient/slope) and the network slows down learning (known as **problem of vanishing gradients**). One solution of that is to use the ReLU activation function. **Rule of thump:** Use ReLU/LreLU everywhere except the last node (e.g. sigmoid in case of binary classification). Also, deep NNs usually use **batch normalization** between hidden layers which mean-standardize and normalize the activations from previous layer for the next layer. This is a golden technique that helps NNs to learn very fast and avoid vanishing gradients problem.

Sigmoid vs ReLU:

1. The ReLU activation function and its derivative (for forward and backward pass) are computationally cheaper allowing to build bigger NNs that need more computations.

2. A NN with ReLU activations functions converges faster than using the sigmoid. So, it is very good for prototyping and testing various models and ideas (e.g. when we do model selection).

3. With ReLU is more unlikely a neuron to vanish (gradient vanishing problem). NNs with sigmoid activation function lead to the **problem of vanishing gradients** in which learning slows down or even stops. ReLU is faster cause is more unlikely this problem to happen.

MLP classification / regression

For the output node(s):

- **Binary classification:** We use the sigmoid in the output for probability (0-1 range)
- **Multi-label classification:** We use the softmax in the output to generate probabilities vector
- **Regression:** No activation function ("linear activation") or ReLU to discard negative values

MLP as a linear regression or logistic regression model:

- In case no activation functions are used in an MLP then the output y is just a linear function of input x and the model operates like **linear regression**
- In case only the output node of an MLP has an activation function and it is a sigmoid then the whole model behaves **like logistic regression**

Why do we prefer deep vs shallow (wide) NNs? In general, NNs (either deep or shallow) tend to approximate / separate non-linear data effectively. However, we prefer to use deep instead of shallow (wide) NNs. **Reason 1:** Learning a mapping between $x \rightarrow y$ is easier for a deep than a vanilla NN cause vanilla needs exponentially more hidden units. **Reason 2:** A deep NN learns different kind of features in each layer. For example, it can learn simple features (e.g. edges, phonemes) in the first layers and while we go deeper we learn more complex features (e.g. faces, words) in next layers. So, we have a hierarchical representation / decomposition of the information.