ECE 595 Machine Learning II

Homework 1 : Fundamentals Aspects of Deep Forward Networks

Murali Krishnan Rajasekharan Pillai

September 2019

Question 1 (XOR Functions)

a) It is not possible for a linear function to learn the XOR function in the $\mathbb{R}^2$. The linear model just outputs 0.5 everywhere. This can be explained more intuitively in the following example:

b) Yes, it is possible for a feed-forward network to correctly learn the XOR function. However, this is dependent on whether we use linear or non-linear functions to define the features. Using linear functions to represent the features means that the function will not be able to learn a boundary to separate the two sets of data. This was demonstrated in the previous section.

However, we will be able to learn the XOR function by using the following network using the ReLU activation function:

$$f(x; W, c, w, b) = w^T \max\{0, W^T x + c\} + b$$

Now, we can get the solution for the XOR problem with the following values for the parameters:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

and $b = 0$

Let $\boldsymbol{X}$ be the design matrix containing all 4 points in the binary input space, (one example per row):

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

The first step of the NN is to perform $\boldsymbol{XW}$:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Now we add the bias unit c to obtain:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Then we perform the ReLU activation to obtain:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Finally we multiply with the weight vectr w:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

This is essentially the XOR function.

Question 2 (RBF Kernel)

The radial basis function kernel, also called the RBF Kernel or the Gaussian Kernel assumes the following form:

$$K_{RBF}(x, x') = exp[-\gamma \left\| x - x' \right\|_2^2]$$

We have to prove that the RBF kernel projects vectors into an infinite dimensional space. Since we deal with Euclidean vectors, this space is $\mathbb{R}^\infty$.

Proof. Let $x, x' \in \mathbb{R}^n$ and assuming $\gamma = \frac{1}{2}$. So,

$$\begin{aligned}
K_{RBF}(x, x') &= exp\left[-\frac{1}{2}\left\| x - x' \right\|_2^2\right] \\
&= exp\left[-\frac{1}{2}\langle x - x', x - x' \rangle\right] \\
&= exp\left[-\frac{1}{2}(\langle x, x \rangle + \langle x', x' \rangle - 2\langle x, x' \rangle)\right] \\
&= exp\left[-\frac{1}{2}(\|x\|_2^2 + \|x'\|_2^2)\right] \cdot exp\left[\left(-\frac{1}{2}\right)(-2\langle x, x' \rangle)\right] \\
&= C \cdot exp\left[\left(-\frac{1}{2}\right)(-2\langle x, x' \rangle)\right] && C := exp\left[-\frac{1}{2}(\|x\|_2^2 + \|x'\|_2^2)\right] \text{ is a constant.} \\
&= C \cdot \sum_{n=0}^{\infty} \frac{\langle x, x' \rangle^n}{n!} && \text{Taylor Series expansion of } exp(x)
\end{aligned}$$

□

Hence we can see that the RBF Kernel corresponds to a dot-product in an infinite-dimensional vector space.

Question 3 (Gradient Based Learning)

a) Unlike Linear Models (like Logistic Regression or SVMs), the non-linearity of the Neural Networks (NNs) causes most cost functions to be non-convex. Due to the objective functions being non-convex, there is no guarantee for solving a global optima. However, iterative gradient-based optimizers can drive down the cost function to a very low value by updating the parameters. They use the direction of the steepest descent to iteratively progress towards an optima, giving us an optimal / "trained" set of parameters for the NNs. Hence the parameters of the model are updated according to the first-order Taylor Series expansion:

$$\theta_{new} = \theta_{old} - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta}$$

b) As we can see from the above equation, the model parameters don't get updated if $\frac{\partial J(\theta)}{\partial \theta} = 0$. This is reflected as a "*flat*" region in the parameter space. This is also said to "*saturate*". In this case the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Instead it is better to use a difference approach that ensure that there is always a strong gradient whenever the model has a wrong answer.

Question 4 (Cross Entropy)

a) Suppose we have a set of independent observations $\{(x_1, y_1), \cdots (x_n, y_n)\} \subset \mathcal{X} \times \mathcal{Y}$, sampled from p_{data}. $\mathcal{X}$ is the input space & $\mathcal{Y}$ is the output space.

Proof. We can define the maximum likelihood as:

$$
\begin{aligned}
P_{\text{MLE}}(x,y) &= \underset{\theta}{\text{argmax}}\ P_{\mathcal{X},\mathcal{Y}}(x,y|\theta) \\
&= \underset{\theta}{\text{argmax}}\ \prod_{i=i}^{n} P(y_i|x_i,\theta) \\
\log(P_{\text{MLE}}(x,y)) &= \underset{\theta}{\text{argmax}}\ log\big(\prod_{i=1}^{n} P(y_i|x_i,\theta)\big) \\
&= \underset{\theta}{\text{argmax}}\ \sum_{i=1}^{n} log\big(P(y_i|x_i,\theta)\big) \\
\frac{1}{n}\cdot\log(P_{\text{MLE}}(x,y)) &= \frac{1}{n}\cdot\underset{\theta}{\text{argmax}}\ \sum_{i=1}^{n} log\big(P(y_i|x_i,\theta)\big) \\
&= \underset{\theta}{\text{argmax}}\ \mathbb{E}_{x,y\ p_{\text{data}}}\big[log\big(P(y_i|x_i,\theta)\big)\big] \\
-\frac{1}{n}\cdot\log(P_{\text{MLE}}(x,y)) &= \underset{\theta}{\text{argmin}}\ \mathbb{E}_{x,y\ p_{\text{data}}}\big[-log\big(P(y_i|x_i,\theta)\big)\big]
\end{aligned}
$$

□

b) We expect the cross-entropy cost to work "*better*" with output units that apply an exponential function. Hence softmax or sigmoidal output units should work well with cross-entropy loss.

Question 5 (Sigmoid / Softmax)

a) The Sigmoid and Softmax functions are defined as follows:

$$\text{sigmoid}(z_i) = \sigma(z_i) = \frac{1}{1+\exp(y_i\cdot z_i)}$$

$$\text{softmax}(z_i) = \frac{\exp(y_i\cdot z_i)}{\sum_{j=1}^{k}\exp(y_j\cdot z_j)}$$

The sigmoid can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, which doesn't sum to 1. We can then divide by an appropriate constant to obtain a valid probability distribution. Assuming that the unnormalized log probabilities are linear in y and z. We see that the normalized distribution yields a Bernoulli distribution controlled by a sigmoidal transformation of z:

$$
\begin{aligned}
\log\tilde{P}(y) &= yz, \\
\tilde{P}(y) &= \exp(yz) \\
P(y) &= \frac{\exp(yz)}{\sum_{y'=0}^{1}\exp(y'z)} \\
P(y) &= \sigma((2y-1)z)
\end{aligned}
$$

This approach is natural to use with maximum likelihood learning.

To generalize the above method for the case of discrete variables with n values, a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = \text{P}\,(y=i|\boldsymbol{x})$. This now generalizes to the multi-nomial distribution. This is what the softmax function performs. First, a linear layer predicts unnormalized log probabilities:

$$z = W^T h + b,$$

where $z_i = \log \tilde{P}(y = i|x)$. The softmax function can then exponentiate and normalize z to obtain the desired $\hat{y}$. Formally the softmax function is given by:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

b) Sigmoid and Softmax are extremely beneficial in gradient based learning because they help prevent the occurrence of saturation. When using Sigmoid/Softmax function in conjunction with the cross-entropy loss, the model only saturates when the model already has the right answer:

i) when $y = i$ is the correct label and z is very positive or

ii) when $y = i$ is a wrong label and z is very negative.

Question 6 (Linear Activation Functions)

If every hidden unit in the neural network (NN) are linear, then the model as a whole will be linear. Hence the linear activation function is not a good idea if applied to all hidden layers. However, using linear activation on some layers can be beneficial. This can be demonstrated as follows:

Question 7 (ReLU Activation)

a) ReLU activation function stands for Rectified Linear Units.

$$g(z) = \max\{0, z\}$$

These units are similar to linear input, except they are inactive when $z < 0$. They are typically used on top of an affine transformation.

b) One drawback of the ReLU activation function is that they cannot learn via gradient based methods on examples for which activation is zero. The immediate generalization of the ReLU activation function makes sure that the learning algorithm can receive gradient everywhere. The 2 generalizations to discuss are **absolute value ReLU** and **parametric ReLU**. These generalization of the ReLU's are based on using a non-zero slope α_i, when $z_i < 0, \ni h_i = g(z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$

i) $\alpha_i = -1$: This obtains the **absolute value ReLU**, or $g(z) = |z|$. It is used for object recognition from images, where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

ii) α_i is a learnable parameter: Instead of assigning a value to α_i, the **parametric-ReLU** adds α_i as a variable for the neural network to learn. This fixes the problem of unnecessary sparsity introduced by pure-ReLU.

c) **Maxout Units** generalize ReLUs further. Instead of applying an element-wise function $g(z)$, maxout units divide z into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

$$g(z)_i = \max_{j \in \mathbb{G}^{(i)}} z_j,$$

where $\mathbb{G}^{(i)}$ is the set of indices unto the inputs for group $i, \{(i-1)k+1, \cdots, ik\}$. This provides a way of learning a piecewise linear function that responds to multiple directions in the input $\boldsymbol{x}$ space.

Maxout units are mainly used in dimensionality reduction, however they increase the complexity of the hypothesis space. If the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.

Question 8 (Universal Approximation)

1. The **Universal Approximation Theorem** states that, a feedforward network with a linear output layer and at-least one hidden layer with any "*squashing*" activation function (such as *sigmoid()*, *tanh()*) can approximate any **Borel measurable function** from one finite-dimensional space to another, with any desired non-zero amount of error, provided that the network is given enough hidden units.
   In other words, any continuous function on a closed and bounded subset of $\mathbb{R}^n$, is Borel measurable and therefore maybe approximated by a neural network.

2. The generalization of the Universal Approximation Theorem states that, shallow networks with a broad family of non-polynomial activation functions (including $ReLU$) have universal approximation properties. In other words, a sufficiently wide network can represent any function.
   This generalisation allows us to use deep neural networks that have multiple hidden layers.

3. In many cases the number of hidden units required by the shallow model is exponential in d, where d is the depth of the neural network. Hence the number of parameters in the model would also be exponential in d.

   Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. Deep architectures also express a useful prior over the space of functions that the model learns. This results in better generalization for a wide variety of tasks.

   By increasing the depth of the network we are increasing the complexity of the hypothesis space. This means that the network has more parameters which need training, which would increase the difficulty in training optimization.

Question 9 (Backpropagation)

a) We have $f : \mathbb{R}^n \to \mathbb{R}^m$, and $g : \mathbb{R}^m \to \mathbb{R}$, and $h = g \circ f$

Proof. By definition, this can be written as: h = g(u)
u = f(x)

$$h(x) = g(f(x))$$
$$\nabla_x h(x) = \nabla_x f(x)^T \ \nabla_x g(f(x)) \qquad \{\because (g \circ f)'(x) = g'(f(x)) \cdot f'(x)\}$$

□

(Illustrate the components of each vector/matrix)??

b) Assume $J : \mathbb{R}^d \to \mathbb{R}$ be defined as:

$$J(u_1, \cdots u_d) = f_l \circ f_{l-1} \circ \cdots \circ f_1(u_1, \cdots u_d)$$

i) The pseudo-code for the forward propagation algorithm to compute $\nabla J(u_1, \cdots, u_d)$ is as follows:

Algorithm 1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}, j < i$ with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\mathbf{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \cdots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \cdots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} | j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

The pseudo-code for the backward propagation algorithm to compute $\nabla J(u_1, \cdots, u_d)$ is as follows:

Algorithm 2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ w.r.t. the variables in the graph. This algorithm computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation to obtain activations of the network
Initialize `grad-table`, a data structure that will store the derivatives that have been computed.
The entry `grad-table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$
`grad-table`$[u^{(n)}] \leftarrow 1$
for $j = n - 1$ down to 1 **do**
  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$, using stored valuesL
  `grad-table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad-table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return $\{$`grad-table`$[u^{(i)}] | i = 1, \cdots, n_i\}$

ii) The amount of computation required for performing the back-propagation scales linearly with the number of edges in the computational graph for the forward propagation $\mathcal{G}$. The computation for each edge corresponds to computing a partial derivative (of one node w.r.t one of it's parents) as well as performing one multiplication and one addition. Hence is of the order $\mathcal{O}(ld)$??

c) Considering a multi-layer feedforward network $f : \mathbb{R}^d \to \mathbb{R}^o$, described by:

$$f(x) = h^{(l)}$$
$$h^{(k)}(x) = \sigma(a^{(k)}), a^{(k)} = b^{(k)+W^{(k)}h^{(k-1)}}, k \in \{1, \cdots, l\}$$

σ is the element-wise non-linear activation function, $W^{(k)}$ is the weight matrix of the k-th layer, and $b^(k)$ is the bias vector of the k-th layer. Suppose the loss function is described by:

$$\mathcal{L} : \mathbb{R}^o \times \mathbb{R}^o \to \mathbb{R}, (\hat{\boldsymbol{y}}, \boldsymbol{y}) \mapsto \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

where $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ is the network output and $\boldsymbol{y}$ is the target output.

i) The pseudo-code for the forward propagation algorithm is as follows:

Algorithm 3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$. To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weight & biases). Algorithm 4 shows how to compute gradients of J w.r.t. parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example. Practical applications should use a minibatch.

Require: Network Depth, l
Require: $\boldsymbol{W^{(i)}}, i \in \{1, \cdots, l\}$ the weight matrices of the model
Require: $\boldsymbol{b^{(i)}}, i \in \{1, \cdots, l\}$ the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \cdots, l$ **do**
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$

The pseudo-code for the backward propagation is as follows:

Algorithm 4 Backward computation for the DNN of Algorithm 3, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$. This computation yields gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients which can be interpreted as an indication of how each layer's output should change to reduce error, once can obtain the gradients on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update or used with other gradient-based optimization methods.

After forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, y)$
for $k = l, l-1, \cdots, 1$ **do**
    Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise if f is element-wise)
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
    Compute gradients on weights and biases (including the regularization term, when needed)
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda\nabla_{\boldsymbol{b}^{(k)}}\Omega(\theta)$
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}[\boldsymbol{h}^{(k-1)}]^T + \lambda\nabla_{\boldsymbol{W}^{(k)}}\Omega(\theta)$
    Propagate the gradients w.r.t the next lower-level hidden layer's activations:
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = [\boldsymbol{W}^k]^T\boldsymbol{g}$
end for

ii) The computational graph for the back-prop algorithms is as follows:

d) The algorithm performs of the order of one Jacobian product per node in the graph. By storing the values in the nodes, back-propagation avoids exponential explosion in repeated sub-expression. We might be able to avoid more subexpressions by simplifying the computational graph. We can also choose to conserve memory by recomputing rather than storing some subexpressions.

Question 10 (Backpropagation Implementation)

a) Libraries such as Torch & Caffe use the **symbol-to-number differentiation** approach to back-propagation. In this approach, the libraries take a computational graph and a set of numerical values for the inputs to the graphs, then return a set of numerical values describing the gradients at those input values.

Libraries such as Theano & Tensorflow take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives. The advantage of this approach is that the derivatives are described in the same language as the original expression. In this approach, the derivatives are just another computational graph and hence back-propagation can be run to obtain higher derivatives. This approach is called the **symbol-to-symbol differentiation** approach

b) Each operation `op` is also associated with a `bprop` operation. This operatoin can compute a Jacobian-vector product as follows:

$$\nabla_{\boldsymbol{X}} z = \sum_j (\nabla_{\boldsymbol{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

This is how the back-propagation algorithm is able to achieve generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that participates in. For example,

We might use a matrix multiplication operation to create a variable $\boldsymbol{C} = \boldsymbol{AB}$. Suppose that the gradient of a scalar z w.r.t $\boldsymbol{C}$ is given by $\boldsymbol{G}$. The matrix multiplication operation is responsible for defining two back-propagation rules, one for each of its input argument. If we call the `bprop` method to request the gradient w.r.t $\boldsymbol{A}$ given that the gradient on the output is $\boldsymbol{G}$, then the `bprop` method of the matrix multiplication operation must state that the gradient w.r.t $\boldsymbol{A}$ is given by $\boldsymbol{GB}^T$. Likewise, it we call the `bprop` method to request the gradient w.r.t $\boldsymbol{B}$, then the matrix operation is responsible for implementing the `bprop` method and specifying the desired gradient is given by $\boldsymbol{A}^T\boldsymbol{G}$. The back-prop algorithm itself does not need to know any

differentiation rules. It only needs to call each operation's `bprop` rules with the right arguments. Formally, `op.bprop(inputs,` $\boldsymbol{X}, \boldsymbol{G}$) must return:

$$\sum_i (\nabla_X \texttt{op.f(inputs)}_i G_i,$$

Here, `inputs` is a list of inputs that are supplied to the operation,
`op.f` is the mathematical function that the operation implements,
X is the input whose gradient we wish to compute,
G is the gradient on the output of the operation.