Homework 1 Solution

Fall 2019

Exercises

1. a) It is not possible to learn the XOR function with a linear function. To see why, suppose toward a contradiction that there does exist a linear function with weight $\boldsymbol{w} \in \mathbb{R}^2$ and bias $b \in \mathbb{R}$ that can learn the XOR function. This implies that the following system of equations in terms of the variables (w_1, w_2, b) must be solvable:

$$\begin{cases} w_1 \times 1 + w_2 \times 0 + b = 1 \\ w_1 \times 0 + w_2 \times 1 + b = 1 \\ w_1 \times 1 + w_2 \times 1 + b = 0 \\ w_1 \times 0 + w_2 \times 0 + b = 0 \end{cases}$$

But this system clearly has no solution (we get $b = 0$ from last equation, then we get $w_1 + w_2 = 1$ from third equation, then we get $w_1 = 1$ and $w_2 = 1$ from first two equations). Hence, we have reached a contradiction.

b) Yes, it is possible to learn the XOR function with a feedforward network with one hidden layer. Consider the network described as follows:

$$f(\boldsymbol{x}) = f^{(2)}(f^{(1)}(\boldsymbol{x})) \tag{1}$$

where

$$\begin{aligned} f^{(1)} &: \mathbb{R}^2 \to \mathbb{R}^2,\ \boldsymbol{x} \mapsto \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c} \\ f^{(2)} &: \mathbb{R}^2 \to \mathbb{R},\ \boldsymbol{z} \mapsto \boldsymbol{w}^T \mathrm{ReLU}(\boldsymbol{z}) + b \end{aligned} \tag{2}$$

with parameters

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},\ \boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},\ \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},\ b = 0 \tag{3}$$

Recall that $\mathrm{ReLU} : \mathbb{R}^d \to \mathbb{R}^d_+,\ \boldsymbol{x} \mapsto (\max\{0, x_1\}, \max\{0, x_2\}, ..., \max\{0, x_d\})$.

To check that f correctly learns the XOR function, simply compute it on all the possible inputs in $\mathcal{X}$. Detailed computations are omitted here as they can be found in the lecture notes and section 6.1 of the textbook.

2. Let $\gamma = 1$ for convenience.

Note that a function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called a kernel if it is of the form

$$K(\boldsymbol{x}, \boldsymbol{x}') = \langle \psi(\boldsymbol{x}), \psi(\boldsymbol{x}') \rangle \tag{4}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in a (possibly infinite-dimensional) vector space, and $\psi : \mathbb{R}^d \to \mathbb{V}$ is the projection function, where $\mathbb{V}$ is some vector space (in fact it should be a Hilbert space).

We wish to show that the radial basis function is a kernel.

Consider the simple $d = 1$ case first. The following sequence of equalities hold:

$$\begin{aligned}
\exp\{-\|x - x'\|_2^2\} &= \exp\{-x^2 - x'^2 + 2xx'\} \\
&= \exp\{-x^2\}\exp\{-x'^2\}\exp\{2xx'\} \quad \text{(Taylor expansion)} \\
&= \exp\{-x^2\}\exp\{-x'^2\}\left(\sum_{i=0}^{\infty}\frac{(2xx')^i}{i!}\right) \\
&= \sum_{i=0}^{\infty}\left(\sqrt{\frac{2^i}{i!}}\exp\{-x^2\}x^i\right)\left(\sqrt{\frac{2^i}{i!}}\exp\{-x'^2\}x'^i\right)
\end{aligned} \tag{5}$$

So the function can be viewed as $\langle\psi(\boldsymbol{x}), \psi(\boldsymbol{x}')\rangle$, where $\psi(x) = (\exp\{-x^2\}, \sqrt{\frac{2^1}{1!}}\exp\{-x^2\}x, \sqrt{\frac{2^2}{2!}}\exp\{-x^2\}x^2, ...)$

In the more general case of $d > 1$, consider the following sequence of equalities:

$$\begin{aligned}
\exp\{-\|\boldsymbol{x} - \boldsymbol{y}\|_2^2\} &= \exp\{-\|\boldsymbol{x}\|_2^2 - \|\boldsymbol{y}\|_2^2 + 2\boldsymbol{x}^T\boldsymbol{y}\} \\
&= \exp\{-\|\boldsymbol{x}\|_2^2\}\exp\{-\|\boldsymbol{y}\|_2^2\}\exp\{2\boldsymbol{x}^T\boldsymbol{y}\} \\
&= \exp\{-\|\boldsymbol{x}\|_2^2\}\exp\{-\|\boldsymbol{y}\|_2^2\}\left(\sum_{i=1}^{\infty}\frac{(2\boldsymbol{x}^T\boldsymbol{y})^i}{i!}\right)
\end{aligned} \tag{6}$$

The summands in the infinite sum are in fact what are called polynomial kernels, i.e. they can be written in the form of (4) (check this!). More rigorously speaking, it can be shown (nontrivially) that the sum of two kernels produces a new kernel: for $K_1(\boldsymbol{x}, \boldsymbol{x}') = K_2(\boldsymbol{x}, \boldsymbol{x}') + K_3(\boldsymbol{x}, \boldsymbol{x}')$, the projection function can be written as

$$\psi_1 : \mathbb{R}^d \to \mathbb{V}_2 \times \mathbb{V}_3,\ \boldsymbol{x} \mapsto (\psi_2(\boldsymbol{x}), \psi_3(\boldsymbol{x})) \tag{7}$$

In fact, the resulting feature space of ψ_1 is the orthogonal direct sum of the feature spaces of ψ_2 and ψ_3 (which are Hilbert spaces), therefore the feature space of ψ_3 is also a Hilbert space. It follows that the dimension of the feature space of K_1 is the sum of the dimension of the feature spaces of K_2 and K_3. Now, since the radial basis function kernel is an infinite sum of polynomial kernels, the dimension of its feature space must be infinite dimensional (again a nontrivial result).

3. a) Gradient based learning is essentially utilizing the technique of gradient descent to achieve the purpose of "learning". In this context, "learning" refers to the minimization of a cost function (that penalizes false predictions and possibly other aspects) of a parametric model with respect to the parameters of that model. In general, gradient descent can be applied to optimization problems that are nonlinear, and likely nonconvex. In the case of the cost function being convex, gradient descent is guaranteed to converge to the global minimum with sufficiently small step sizes regardless of the initialized parameters of the model. However, when the cost function is nonconvex, gradient descent does *not* necessarily converge and land on the global minimum of the minimization problem, and the sequence of descent steps it takes (including the final solution it lands on) is dependent on the initialization. For modern neural networks, the weights are often initialized to small random values, and the biases to be either zero or small positive values, as policies of this type had been demonstrated from experience to make training/learning easier to converge to a good local minimum of the optimization landscape.

   b) A significant aspect of gradient based learning is that the gradient with respect to the parameters is best to be nonvanishing during training, as a vanishing gradient essentially means that the gradient descent is coming to a halt, and no more "learning" occurs. What this means is that the pairing of the cost function and the output units of the neural network should produce a function (of the network's parameters) that does not saturate easily, i.e., it should be a function that does not have nearly zero gradient on large subsets of its domain.

   A good pair of output-unit-cost-function pair is the cross-entropy loss and softmax (or sigmoid) output units, as the logarithm in the cross-entropy loss cancels out the exponentials in the softmax, which alleviates the saturation of the softmax loss if the network's output units do not contain exponential terms.

Another important aspect is to obtain a gradient direction that is *consistent*, when evaluated at different points. Hence, linear or close-to-linear behavior is best when using gradient-based learning, as it enables the use of higher learning rates. The other extreme (bad behavior) would be an exponential function, which would have a gradient that is also an exponential, and hence its direction would change rapidly between neighboring points, which would require using very small learning rates, and hence, results in slow training.

4. a) Consider the following derivation:

$$\begin{aligned}
\boldsymbol{\theta}^* &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p_{\text{model}}(\boldsymbol{y}_1, ..., \boldsymbol{y}_n | \boldsymbol{x}_1, ..., \boldsymbol{x}_n, \boldsymbol{\theta}) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \log\left(p_{\text{model}}(\boldsymbol{y}_1, ..., \boldsymbol{y}_n | \boldsymbol{x}_1, ..., \boldsymbol{x}_n, \boldsymbol{\theta})\right) \quad \text{(Monotonicity of the logarithm)} \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \log\left(\Pi_{i=1}^{n} p_{\text{model}}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})\right) \quad \text{(The samples are independent)} \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{n} \log(p_{\text{model}}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \frac{1}{n} \sum_{i=1}^{n} \log(p_{\text{model}}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathbb{E}_{\boldsymbol{x},\boldsymbol{y} \sim p_{\text{data}}}[\log(p_{\text{model}}(\boldsymbol{y} | \boldsymbol{x}, \boldsymbol{\theta}))], \text{ as } n \to \infty \quad \text{(Samples are identically distributed)} \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y} \sim p_{\text{data}}}[\log(p_{\text{model}}(\boldsymbol{y} | \boldsymbol{x}, \boldsymbol{\theta}))]
\end{aligned} \tag{8}$$

Note that the second-to-last line only holds rigorously if the requirements of the Weak Law of Large Numbers are satisfied.

b) We examine the last line of the the derivation above. As the output of the neural network is the $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$, we expect it to behave in a way that is compatible with the logarithm in the context of gradient-based learning; that is, we wish to avoid having large regions in the parameter space that have small/saturated gradient. The softmax/sigmoid output units are the most popular choice, as they can be interpreted probabilistically, and the exponential terms in them help avoid vanishing gradients.

5. The Sigmoid function:

$$\sigma : \mathbb{R} \to (0, 1),\ x \mapsto \frac{1}{1 + e^{-x}} \tag{9}$$

The Softmax function: fix $N \in \mathbb{N}$,

$$S : \mathbb{R}^N \to (0, 1)^N,\ \boldsymbol{x} \mapsto \left(\frac{e^{x_1}}{\sum_{i=1}^{N} e^{x_i}}, ..., \frac{e^{x_1}}{\sum_{i=1}^{N} e^{x_i}} \right) \tag{10}$$

First of all, the sigmoid is just the binary special case of the softmax. Moreover, we can interpret these two functions probabilistically.

In the specific context of binary classification problem, if the sigmoid is used as the output unit of the neural network, then its output essentially represents the probability of the input sample to belong to either class 1 or class 0, i.e. it is representing $p_{\text{model}}(y = 1|x, \boldsymbol{\theta})$. To elaborate, we usually adopt the following inference policy: if the output is above 0.5, we classify the input as belonging to class 1, and class 0 otherwise. From the statistics perspective, we are carrying out maximum likelihood estimation here.

When we have more than two classes, the softmax can be used to represent $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$. The underlying principle for inference is very similar to how we utilize the sigmoid: we choose the index of the softmax output that has the largest value as the class label of the input sample.

The advantages of choosing the softmax/sigmoid with cross-entropy in gradient-based learning had been studied in part b) of the last question.

6. Linear activations are sometimes used in the fully-connected/dense layers of neural networks to reduce the number of parameters in the layer and facilitate faster training. Consider a dense layer specified by $\boldsymbol{h}_1 = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{W} \in \mathbb{R}^{n\times p}$; note that $\boldsymbol{W}$ contains np parameters. Now consider replacing $\boldsymbol{W}$ with $\boldsymbol{UV}$, where $\boldsymbol{U} \in \mathbb{R}^{n\times q}$ and $\boldsymbol{V} \in \mathbb{R}^{q\times p}$, so this different dense layer has $(n+p)q$ parameters. For small values of q, the latter dense layer could have significantly smaller number of parameters, leading to potentially faster training. Do note that it is not true that a general matrix $\boldsymbol{W} \in \mathbb{R}^{n\times p}$ can be written in the factored form $\boldsymbol{W} = \boldsymbol{UV}$ like above, this factorization is only possible if $\boldsymbol{W}$ is low-rank, in fact, $\text{rank}(\boldsymbol{W}) \leq \text{rank}(\boldsymbol{U}) \leq \min\{n, q\}$.

   Empirically, it has been found that nonlinear activations for hidden units (other than the dense layers) often serve better for neural networks than the linear activations in popular tasks such as object recognition and those belonging to natural language processing; piecewise linear activation functions like ReLU and its variants, sigmoid, tanh are good examples.

7. a) $\text{ReLU} : \mathbb{R}^d \to \mathbb{R}^d_+, \boldsymbol{x} \mapsto (\max\{x_1, 0\}, ..., \max\{x_d, 0\})$

   b) The absolute value activation function works as follows:

$$\boldsymbol{x} \mapsto (|x_1|, ..., |x_d|) \tag{11}$$

   It is useful in object recognition from images. Roughly speaking, it is invariant under polarity reversal of input illumination. To elaborate, consider an edge-detecting kernel (in an early layer of a convolutional neural network for example). For a patch in the image, if the regions in the patch corresponding to the positive parts of the kernel are brighter and larger in size than those corresponding to the negative parts of the kernel, then the output of the convolution produces a positive output; other situations can be considered with similar reasoning. Now we focus on negative convolution output. In such a case, the output is as important as the positive outputs, but a ReLU would not activate on them, while an absolute value function does activate.

   The parametric ReLU activation function works as follows:

$$\boldsymbol{x} \mapsto (\max\{x_1, 0\} + \alpha_1 \min\{0, x_1\}, ..., \max\{x_d, 0\} + \alpha_d \min\{0, x_d\}) \tag{12}$$

   The α_i's are trainable parameters of the network. Note that the leaky ReLU is a special case of the parametric ReLU, as it can be obtained by fixing the α_i's to a small number (e.g. 0.01) and made uniform across the coordinates. Also, absolute value ReLU can be obtained by setting $\alpha_i = -1, \forall i$. The advantage of using parametric ReLUs is the added flexibility, and avoiding the "dead" neuron problem caused by ReLU, because the derivative of parametric ReLU on the negative half of the real number line is no longer fixed to be 0.

   c) Given $\boldsymbol{z} \in \mathbb{R}^d$ the output of a hidden layer, the maxout unit $g : \mathbb{R}^k \to \mathbb{R}$ operates as follows:

$$g(\boldsymbol{z})_i = \max_{j\in\mathbb{G}^{(i)}} z_j \tag{13}$$

   where $\mathbb{G}^{(i)} = \{(i-1)k + 1, ..., ik\}$ is the i-th group of indices of the input vector $\boldsymbol{z}$.

   In the original paper of the maxout network by Goodfellow et al., the maxout unit was used in a more specific way, as the authors were pooling across *channels*. In other words, supposing that $z_{ij} = \boldsymbol{x}^T\boldsymbol{W}_{\cdot ij} + b_{ij}$, where $\boldsymbol{x} \in \mathbb{R}^d$ is the input to the current hidden layer, $\boldsymbol{W} \in \mathbb{R}^{d\times m\times k}$ is the layer's tensor and $\boldsymbol{b} \in \mathbb{R}^{m\times k}$ is the bias vector, the maxout unit $g : \mathbb{R}^{m\times k} \to \mathbb{R}^m$ is defined by

$$g(\boldsymbol{z})_i = \max_{j\in\{1,...,k\}} z_{ij} \tag{14}$$

   The maxout unit in general is capable of learning a piecewise linear convex function with up to k pieces, and it is known that as k grows toward infinity, a piecewise linear convex function can approximate any convex function. For instance, when $k = 2$, the maxout unit can approximate the ReLU and its 2-piece variants discussed in part b of this problem. Moreover, a feedforward neural network with maxout activation is actually a universal function approximator: it can approximate any continuous function defined on a compact set in $\mathbb{R}^d$. Another benefit of maxout is that due

to each maxout unit being dependent on multiple filters, the redundancy helps alleviate the issue of "catastrophic fogetting", in which neural networks forget how to perform tasks that they were trained on.

A potential problem with this activation function is that as each maxout unit is requiring k inputs in comparison to activation functions such as ReLU which requires only one input, possibly more parameters are required in the current hidden layer, so regularization might be needed. On the other hand, it could be the case that less parameters are required in the next layer due to the aggressive reduction in dimensionality (in comparison to, for example, a usual 2×2 maxpool) in the current layer.

8. a) A feedfoward neural network with a linear output layer and at least one hidden layer with an activation function that saturate for large positive and negative arguments can approximate - with enough hidden units - any Borel measurable function from a finite-dimensional space to another within any non-zero error rate. Moreover, the derivatives of the network can also approximate the derivatives of the function arbitrarily well.

   Note that the function to be approximated does not have to have a continuous domain and/or continuous target space, as discrete spaces are allowed.

   b) The universal approximation theorem had been extended to neural networks with more general activation functions, including the ReLU activation. However, exponentially many hidden units (possibly with one hidden unit corresponding to each input conguration that needs to be distinguished) might be needed. A simple example is to consider the space of functions $\mathcal{F} = \{f : \{0,1\}^n \to \{0,1\}\}$. As there are 2^n possible inputs to a function in $\mathcal{F}$, and 2 possible outputs of a function, there are $\Pi_{i=1}^{2^n} 2 = 2^{2^n}$ possible functions in $\mathcal{F}$. In general, as $\log_2(2^{2^n}) = 2^n$, $O(2^n)$ bits can be required to represent a function from $\mathcal{F}$.

   c) A wide shallow network is a universal approximator, however, such a network could require exponentially many hidden units to fit a function, and can be extremely hard to train due to potential overfitting problems springing from the high model complexity. On the other hand, it is known that the number of linear regions carved out by a deep rectifier-based network with d inputs, depth l, and n units per hidden layer is $O(\binom{n}{d}^{d(l-1)} n^d)$, hence much fewer parameters is needed, and we usually obtain better generalization of the model. However, depending on the optimization landscape, a deeper network can be more likely to only reach a local minimum - which might be sufficient for many application - and might also require careful initialization, as it is effectively search in a smaller part of the available vector space, than a shallower network that has the same number of units.

9. a) $\nabla h(\boldsymbol{x}) = Df(\boldsymbol{x})^T \nabla g(f(\boldsymbol{x}))$, where $Df(.)$ is the Jacobian matrix of f. More explicitly, if we denote $\boldsymbol{y} = f(\boldsymbol{x})$ and $z = h(\boldsymbol{x}) = g(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{15}$$

   b) i) Fix $\boldsymbol{u} \in \mathbb{R}^d$. First compute the forward pass values, store the values in the following form in memory:

$$\boldsymbol{u}^{(i)} = f_i(\boldsymbol{u}^{(i-1)}), i \in \{1, ..., l\} \tag{16}$$

   where we adopt the convention that the input vector $\boldsymbol{u}^{(0)} = \boldsymbol{u}$.
   Then we carry out the following algorithm (next page):

   ii) The number of computation depends linearly on the following sum: $\sum_{j=1}^{l+1} n_j n_{j-1}$.

   c) i) The procedure is quite similar to that of the last part. Denote $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y})$, and denote $\odot$ as the elementwise multiplication.

   ii) Set l in the previous part to 1, and the rest of the algorithm can be directly carried over to this part.

Algorithm 1 Question 9 b) Algorithm

procedure
  Initialize `grad_table`, a data structure that will store the derivatives that have been computed;
  `grad_table`$[\boldsymbol{u}^{(l)}] \leftarrow 1$;
  for $j = l - 1$ `down to` 0 **do**
    for $k = 1$ `to` n_j **do**
      `grad_table`$[u_k^{(j)}] \leftarrow \sum_m$ `grad_table`$[u_m^{(j+1)}] \frac{\partial u_m^{(j+1)}}{\partial u_k^{(j)}}$;
    end for
  end for
  return `grad_table`$[\boldsymbol{u}^{(0)}]$;
end procedure

Algorithm 2 Question 9 c) Algorithm

procedure
  After the forward-pass computation, initialize: $\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J$;
  for $k = l$ `down to` 1 **do**
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot \sigma'(\boldsymbol{a}^{(k)})$;
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$;
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$;
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T}\boldsymbol{g}$;
  end for
  return the gradients;
end procedure

d) The memory storage cost of the above algorithm (part c)) is $O(n_h)$, where n_h is the number of hidden units in the network, i.e. the storage complexity roughly scales linearly with the size of the network. The computational complexity is dominated by matrix multiplications, and scales quadratically with the number of hidden units in the neural network.

For the naive approach to backpropagation where we do not store any forward pass values and partial derivatives in memory, the computational complexity scales exponentially with the n_h in the network.

10. a) Torch and Caffe take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values; this approach is called Symbol-to-Number differentiation. On the other hand, Theano and Tensorflow take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives.

b) Formally, the `op.backprop(inputs, X, G)` is defined by

$$\sum_i (\nabla_X \texttt{op.f(inputs)}_i) G_i \tag{17}$$

where `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, and `G` is the gradient on the output of the operation.

For more concrete examples, consider variable $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$ created from matrix multiplication, and let the gradient of some scalar z w.r.t. $\boldsymbol{C}$ be given by $\boldsymbol{G}$. Then `op.backprop(inputs, A, G)` returns $\boldsymbol{G}\boldsymbol{B}^T$, and `op.backprop(inputs, B, G)` returns $\boldsymbol{A}^T\boldsymbol{G}$.