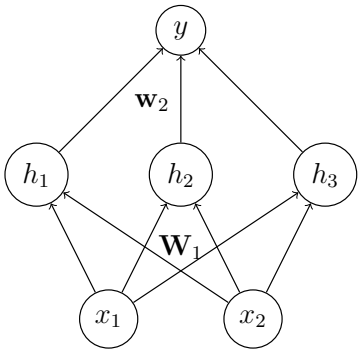
ENM531 Midterm exam (Spring 2019)

Instructor: Paris Perdikaris

Thursday, February 28, 2019

1. **[5 pts]** Recall how linear regression could be made more powerful using a basis function expansion, i.e. a function ϕ which maps each data point $\boldsymbol{x}$ to a feature vector $\phi(\boldsymbol{x})$. This is analogous to fitting a feed-forward neural net with one hidden layer, where one set of weights is held fixed. (Such a network is shown in the following figure.) Which set of weights is held fixed? Briefly explain what the hidden activations and both sets of weights correspond to.



2. **[5 pts]** Compute the convolution of the following two arrays:

$$(4, 1, -1, 3) \star (-2, -1)$$

Your answer should be an array of length 5. (You do not need to show your work, but it may help you get partial credit.)

3. **[5 pts]** Mike and Michelle have implemented two neural networks for recognizing handwritten digits from 16×16 grayscale images. Each network has a single hidden layer, and makes predictions using a soft-max output layer with 10 units, one for each digit class.

   - Michelle's network is a convolutional net. The hidden layer consists of three 16×16 convolutional feature maps, each with filters of size 5×5, and uses the logistic nonlinearity. All of the hidden units are connected to all of the output units.
   - Mike's network is a fully connected network with no weight sharing. The hidden layer consists of 768 logistic units (the same number of units as in Michelle's convolutional layer).

   Briefly explain one advantage of Michelle's approach and one advantage of Mike's approach.

4. **[15 pts]** Inspect the PyTorch code provided in the last page and spot possible mistakes. Report the lines of code where you found a mistake and write down their correct implementation.

5. **[30 pts]** Provide a short answer to the following questions:

   (a) Briefly describe something that can go wrong if we choose too high of a learning rate for full batch gradient descent. **[5 pts]**

(b) Briefly describe something that can go wrong if we choose too low of a learning rate for full batch gradient descent. **[5 pts]**

(c) Briefly describe the different techniques you know for regularizing deep neural networks. **[10 pts]**

(d) Briefly explain what is meant by over-fitting. Is it true that if you choose the hyper-parameters (e.g. number of hidden units in a neural network) well, then there will be no over-fitting? Why or why not? (Either YES or NO is acceptable, as long as you justify your answer.) **[10 pts]**

6. **[40 pts]** Consider the following Bayesian binary classification model with a Bernoulli likelihood over i.i.d. input/output observations $\mathcal{D} = (\boldsymbol{X}, y)$:

$$p(y|\boldsymbol{X}, W, \gamma) = \prod_{i=1}^{N} a_i^{y_i} (1 - a_i)^{1-y_i}, \tag{1}$$

where $a_i = \sigma(w^T x_i)$, with $\sigma()$ denoting the logistic sigmoid function. The weights W and their precision λ are assigned prior distributions:

$$p(W|\lambda) = \mathcal{N}(w|0, \lambda^{-1}),$$
$$p(\lambda) = \text{Gam}(\lambda|\alpha_0^\lambda, \beta_0^\lambda)$$

where $(\alpha_0^\lambda, \beta_0^\lambda)$ are assumed to be known parameters, and Gam denotes the Gamma distribution.

(a) **[5 pts]** Use Bayes rule to express the posterior over the model parameters $p(W, \lambda|\boldsymbol{X}, y)$ as a function of the likelihood and the priors.

(b) **[15 pts]** Derive the optimization objective for training the model parameters (W, λ) using maximum likelihood estimation (MLE). In other words, derive the expression for $-\log p(y|\boldsymbol{X}, W, \lambda)$ using the definition of the Bernoulli probability mass function defined in equation 1.

(c) **[20 pts]** Derive the optimization objective for training the model parameters (W, λ) using maximum a-posteriori estimation (MAP). To do so, recall that the posterior over the model parameters is proportional to $p(W, \lambda|\boldsymbol{X}, y) \propto p(y|\boldsymbol{X}, W, \lambda)p(W|\lambda)p(\lambda)$, and use the following expression for computing the log of the Gamma distribution:

$$\log \text{Gam}(c|\alpha, \beta) = (\alpha - 1) \log c - \beta c + \log c.$$

```
class NeuralNetRegression:
    # Initialize the class
    def __init__(self, X, Y, layers):

        # Check if there is a GPU available
        if torch.cuda.is_available() == True:
            self.dtype = torch.cuda.FloatTensor
        else:
            self.dtype = torch.FloatTensor

        # Normalize the data
        self.Xmean, self.Xstd = X.mean(1), X.std(1)
        self.Ymean, self.Ystd = Y.mean(1), Y.std(1)
        X = (X - self.Xmean) / self.Xstd
        Y = (Y - self.Ymean) / self.Ystd

        # Define PyTorch variables
        X = torch.from_numpy(X).type(self.dtype)
        Y = torch.from_numpy(Y).type(self.dtype)
        self.X = Variable(X, requires_grad=True)
        self.Y = Variable(Y, requires_grad=True)
        self.layers = layers

        # Initialize network weights and biases
        self.weights, self.biases = self.initialize_NN(layers)

        # Store loss values
        self.training_loss = []

        # Define optimizer
        self.optimizer = torch.optim.Adam(self.params, lr=1e-3)


    # Initialize network weights and biases using Xavier initialization
    def initialize_NN(self, layers):
        # Xavier initialization
        def xavier_init(size):
            in_dim = size[0]
            out_dim = size[1]
            xavier_stddev = 17.0 / np.sqrt(in_dim + out_dim)
            return Variable(xavier_stddev*torch.randn(in_dim, out_dim).type(self.dtype), requires_grad=False)

        weights = []
        biases = []
        num_layers = len(layers)
        for l in range(0,num_layers-1):
            W = xavier_init(size=[layers[l+1], layers[l+1]])
            b = Variable(torch.zeros(layers[0],layers[l+1]).type(self.dtype), requires_grad=False)
            weights.append(W)
            biases.append(b)
        return weights, biases


    # Evaluates the forward pass
    def forward_pass(self, H):
        num_layers = len(self.layers)
        for l in range(0,num_layers-2):
            W = self.weights[l]
            b = self.biases[l]
            H = torch.add(F.tanh(torch.matmul(H, W)), b)
        H = torch.add(torch.matmul(H, W), b)
        return H


    # Computes the mean square error loss
    def compute_loss(self, X, Y):
        loss = torch.abs((Y - self.forward_pass(X))**2)
        return loss
```

Figure 1: Example PyTorch class for neural network regression.