Lecture 6 - Jan 29

Radial Basis Function Expansions
Implementing RBFs with Gradient Descent

<u>Reading</u>

Week 4 Notes in GitHub

<u>Upcoming Deadlines</u>

Homework 2 (Feb 10)

$$\hat{f}(x_i) = \theta_0 + \sum_{j=1}^{m} \theta_j K_{\lambda_j}(\xi_j, x_i)$$ – RBF Network

Can we write it in matrix form?

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{pmatrix} \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

$$X_K = \begin{pmatrix} 1 & K_{\lambda_1}(\xi_1, x_1) & \cdots & K_{\lambda_m}(\xi_m, x_1) \\ 1 & K_{\lambda_1}(\xi_1, x_2) & \cdots & K_{\lambda_m}(\xi_m, x_2) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & K_{\lambda_1}(\xi_1, x_n) & \cdots & K_{\lambda_m}(\xi_m, x_n) \end{pmatrix}$$

$$\hat{y} = X_K \theta \qquad \text{(predicted } y\text{)}$$

A radial function of x_i and ξ_j depends on $\|x_i - \xi_j\|$, but not x_i or ξ_j individually

The most popular is the Gaussian kernel

The most popular is the Gaussian:

$$K_{\lambda_j}(\xi_j, x_i) = \frac{1}{\lambda_j} \exp\left(-\frac{\|x_i - \xi_j\|^2}{2\lambda_j^2}\right)$$

note each ξ_j and x_i is in $\mathbb{R}^d$

Sum of squares loss function

$$L(\theta, \lambda, \xi) = \|X_K \theta - y\|^2$$

unfortunately, X_K depends on $\lambda = (\lambda_1, \dots, \lambda_M)$

$$\xi = \begin{pmatrix} \xi_1^T \\ \xi_2^T \\ \vdots \\ \xi_M^T \end{pmatrix}$$

so, minimizing L with respect to θ and λ and ξ is not nearly as easy as with LBF expansions

Further, L is not convex, so it may have multiple critical values, so, even if we could solve $\nabla L(\theta, \lambda, \xi) = 0$, it would not be assured to be the global minimum.

it would not be assured

Minimizing the Loss Function

To minimize $L(\theta,\lambda,\xi) = \|X_\lambda \theta - y\|^2$, there are some options:

① Use numerical optimization to find minima of L with respect to parameters

$\theta \in \mathbb{R}^{d+1}$, $\lambda \in \mathbb{R}^d_{\geq 0}$, $\xi \in \mathbb{R}^{M \times d}$

Dimension of search space $= d+1+d+Md = (2+M)d+1$

↳ we need to compute ∇L by hand if d or M are large and use this formula instead of approximating partial derivatives to speed up computation (2+M)d+1

fix $\lambda = \lambda_1 = \dots = \lambda_d$ and let λ be a hyperparam.

② Use random training x_i's as ξ_j's

③ Use centroids of nearby x_i's as ξ_j's

④ Use clustering, then find centroids of clusters as ξ_j's

For 2-4, ξ_j's + λ fixed ⟹ unique LBF solution will be the minimum

For 2-4, ξ_j's + λ fixed ... will be the minimum
(+ approx. gradients are fine)

(More on "hyperparameters" later)

An Analytic Expression for the Gradient

Writing the loss function in expanded form

$$L(\theta,\lambda,\xi) = \sum_{i=1}^{n}\left(y_i - \hat{f}(x_i)\right)^2$$

$$= \sum_{i=1}^{n}\left(y_i - \theta_0 - \sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j, x_i)\right)^2$$

$$\frac{\partial L}{\partial \theta_0} = \sum_{i=1}^{n} 2\left(y_i - \theta_0 - \sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j, x_i)\right)(-1)$$

$$-2\sum_{i=1}^{n}\underbrace{\left(y_i - \theta_0 - \sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j, x_i)\right)}_{y_i - \hat{f}(x_i)}$$

$$\frac{\partial L}{\partial \theta_k} = \sum_{i=1}^{n} 2\left(y_i - \theta_0 - \sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j, x_i)\right)\left(-K_{\lambda_k}(\xi_k, x_i)\right)$$

$$= -2\sum_{i=1}^{n} K_{\lambda_k}(\xi_k, x_i)\underbrace{\left(y_i - \theta_0 - \sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j, x_i)\right)}_{y_i - \hat{f}(x_i)}$$

for $k = 1, 2, \ldots, d$

For $\frac{\partial L}{\partial \lambda_k}$, we need to use $K_{\lambda_j}(\xi_j, x_i) = \frac{1}{\lambda_j}\exp\left(-\frac{\|\xi_j - x_i\|^2}{2\lambda_j^2}\right)$

$$\frac{\partial L}{\partial \lambda_k} = -2\sum_{i=1}^{n}\left(y_i - \theta_0 - \sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j, x_i)\right)\cdot\underbrace{\frac{\partial}{\partial \lambda_k}\left(\frac{\theta_k}{\lambda_k}\exp\left(-\frac{\|\xi_k - x_i\|^2}{2\lambda_k^2}\right)\right)}$$

$$\frac{-\theta_k}{\lambda_k^2}\exp\left(-\frac{\|\xi_k-x_i\|^2}{2\lambda_k^2}\right)+\frac{1}{\lambda_k}\exp\left(-\frac{\|\xi_k-x_i\|^2}{2\lambda_k^2}\right)\cdot\left(\cancel{-}\;\cancel{+2}\,\frac{\|\xi_k-x_i\|^2}{\cancel{2}\lambda_k^3}\right)$$

1 more common

$$\frac{\theta_k}{\lambda_k}\underbrace{\frac{1}{\lambda_k}\exp\left(-\frac{\|\xi_k-x_i\|^2}{2\lambda_k^2}\right)}_{K_{\lambda_k}(\xi_k,x_i)}\left[\frac{\|\xi_k-x_i\|^2}{\lambda_k^2}-1\right]$$

$$\frac{\partial L}{\partial \lambda_k}=\frac{-2\theta_k}{\lambda_k}\sum_{i=1}^{n}\left(\frac{\|\xi_k-x_i\|^2}{\lambda_k^2}-1\right)K_{\lambda_k}(\xi_k,x_i)\underbrace{\left(y_i-\theta_0-\sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j,x_i)\right)}_{y_i-f(x_i)}$$

$$\frac{\partial L}{\partial \xi_{kl}}=-2\sum_{i=1}^{n}\left(y_i-\theta_0-\sum_{j=1}^{m}\theta_j K_{\lambda_j}(\xi_j,x_i)\right)\frac{\partial}{\partial \xi_{kl}}\left(\frac{\theta_k}{\lambda_k}\exp\left(-\frac{\|\xi_k-x_i\|^2}{2\lambda_k^2}\right)\right)$$

$$=\frac{-\cancel{2}\theta_k}{\lambda_k}\sum_{i=1}^{n}\left(y_i-\hat{f}(x_i)\right)\frac{\partial}{\partial \xi_{kl}}\underbrace{\left(\exp\left(-\frac{(\xi_{k1}-x_{i1})^2+\dots+(\xi_{kl}-x_{il})^2+\dots+(\xi_{kd}-x_{id})^2}{\cancel{2}\lambda_k^2}\right)\right)}$$

$$\exp\left(-\frac{\|\xi_k-x_i\|^2}{2\lambda_k^2}\right)\left(\frac{-\cancel{2}(\xi_{kl}-x_{il})}{\cancel{2}\lambda_k^2}\right)=-K_{\lambda_k}(\xi_k,x_i)\cdot\frac{\xi_{kl}-x_{il}}{\lambda_k}$$

$$\frac{\partial L}{\partial \xi_{kl}} = \frac{-2\theta_k}{\lambda_k^2} \sum_{i=1}^{n} \left(y_i - \hat{f}(x_i)\right) K_{\lambda_k}(\xi_k, x_i)(x_{il} - \xi_{kl})$$

$$= \frac{-2\theta_k}{\lambda_k^2} \sum_{i=1}^{n} (x_{il} - \xi_{kl}) K_{\lambda_k}(\xi_k, x_i)\left(y_i - \hat{f}(x_i)\right)$$

To summarize...

$$\frac{\partial L}{\partial \theta_0} = 2 \sum_{i=1}^{n} \underbrace{\left(\hat{f}(x_i) - y_i\right)}_{1}$$

$$\frac{\partial L}{\partial \theta_k} = 2 \sum_{i=1}^{n} \underbrace{K_{\lambda_k}(\xi_k, x_i)}_{2} \overbrace{\left(\hat{f}(x_i) - y_i\right)}^{1} \quad \text{for } k = 1, \dots, M$$

$$\frac{\partial L}{\partial \lambda_k} = \frac{2\theta_k}{\lambda_k} \sum_{i=1}^{n} \underbrace{\left(\frac{\|\xi_k - x_i\|^2}{\lambda_k^2} - 1\right)}_{3} \underbrace{K_{\lambda_k}(\xi_k, x_i)}_{2} \overbrace{\left(\hat{f}(x_i) - y_i\right)}^{1} \quad \text{for } k = 1, \dots, M$$

$$\frac{\partial L}{\partial \xi_{kl}} = \frac{2\theta_k}{\lambda_k^2} \sum_{i=1}^{n} \underbrace{(x_{il} - \xi_{kl})}_{4} \underbrace{K_{\lambda_k}(\xi_k, x_i)}_{2} \overbrace{\left(\hat{f}(x_i) - y_i\right)}^{1} \quad \text{for } \begin{array}{l} k = 1, \dots, M \\ l = 1, \dots, d \end{array}$$

$$\nabla L = \left(\frac{\partial L}{\partial \theta_0}, \dots, \frac{\partial L}{\partial \theta_M}, \frac{\partial L}{\partial \lambda_1}, \dots, \frac{\partial L}{\partial \lambda_M}, \frac{\partial L}{\partial \xi_{11}}, \dots, \frac{\partial L}{\partial \xi_{Md}}\right)$$

↳ this looks long, but each iteration of gradient descent with approximate gradients would require evaluating L $(M+2)d+2$ times, so this descen[illegible]

evaluating L $(M+2)d+2$ times, so this

single computation of ∇L reusing 1-4

efficiently

$$\|x-y\|_2^2 = \sum_{i=1}^{d} (x_i - y_i)^2$$

RBF Code

```python
# create a RBF network class
class GaussianRBFnetwork:
    # initialize the model
    def __init__(self, d, M, alpha = 0.001, initialization = 'uniform'):
        # the dimension of the datapoints
        self.d = d

        # the number of radial basis functions
        self.M = M

        # the learning rate
        self.alpha = alpha

        # initialize the parameters
        if initialization == 'uniform':
            theta = 10 * np.random.uniform(-1, 1, size = (M + 1))
            lam = 10 * np.random.uniform(0, 1, size = (M))
            xi = 10 * np.random.uniform(-1, 1, size = (M, d))
            print(theta)
            print(lam)
            print(xi)

        # save the initial parameters
        self.theta = theta
        self.lam = lam
        self.xi = xi

    # fit the model to some data X with labels y
    def fit(self, X, y, epochs = 1000, update = 10):
        self.n = X.shape[0]

        # save the training data
        self.input = np.hstack((np.ones([self.n, 1]), X))

        # save the training labels
        self.output = y

        # initialize the kernel-weighted inputs
        XK = np.zeros([self.n, self.M])
        XK = np.hstack((np.ones([self.n, 1]), XK))
```

```python
        # save the learning rate locally
        alpha = self.alpha

        # run gradient descent with exact gradient
        # pre-compute terms
        for j in range(epochs):
            # compute the kernel-weighted inputs
            for i in range(self.n):
                for k in range(self.M):
                    XK[i, k + 1] = self.GaussianKernel(self.input[i, 1:], self.xi[k], self.lam[k])

            # training predictions
            predictions = XK @ self.theta

            # compute the error
            error = predictions - self.output

            # compute theweighted error
            weightederror = np.atleast_2d(error).T * XK

            # compute the theta partial derivatives
            thetagrad = np.sum(weightedError, axis = 0)

            # compute the lambda partial derivatives
            term3 = np.zeros([self.n, self.M])
            for k in range(self.M):
                term3[:, k] = ((np.linalg.norm(self.xi[k] - self.input[:,1:], axis = 1)/self.lam[k]) ** 2 - 1) * self.theta[k]/self.lam[k]

            lamerror = term3 * weightederror[:, 1:]
            lamgrad = np.sum(lamerror, axis = 0)

            # compute the xi partial derivatives
            term4 = np.zeros([self.n, self.M, self.d])
            for k in range(self.M):
                for l in range(self.d):
                    term4[:, k, l] = (self.input[:, l] - self.xi[k, l]) * self.theta[k] / self.lam[k] ** 2

            xierror = term4 * np.atleast_3d(weightederror[:, 1:])
            xigrad = np.sum(xierror, axis = 0)

            # weight update
            self.theta -= self.alpha * thetagrad
            self.lam -= self.alpha * lamgrad
            self.xi -= self.alpha * xigrad

            if j % update == 0:
                print('Epoch', j, '\tLoss =', np.sum(error ** 2)/self.M)
                self.alpha = (1 - j / epochs) * alpha
    # fit the model to some data X
    def predict(self, X):
        # compute predictions
        n = X.shape[0]

        # initialize the kernel-weighted inputs
        XK = np.zeros([n, self.M])
        XK = np.hstack((np.ones([n, 1]), XK))

        # compute the kernel-weighted inputs
        for i in range(n):
            for k in range(self.M):
                XK[i, k + 1] = self.GaussianKernel(X[i, 1:], self.xi[k], self.lam[k])

        # training predictions
        predictions = XK @ self.theta

        return predictions

    def GaussianKernel(self, x, xi, lam):
        return (1/lam) * np.exp(-np.linalg.norm(x - xi) ** 2 / (2 * lam ** 2))
```

Small Example

```
model = GaussianRBFnetwork(d = 1, M = 2, alpha = 0.00001)

Xt = np.array([[2], [4], [5]])
yt = np.array([7, 8, 10])

model.fit(Xt, yt, epochs = 100000, update = 10000)

# predict the outputsk
predictions = model.predict(Xt)

# plot the training points
plt.scatter(Xt, yt, label = 'Training Data')

# compute the training and test mean absolute error
trainError = mean_absolute_error(yt, predictions)

# return quality metrics
print('The r^2 score is', r2_score(yt, predictions))
print('The mean absolute error on the training set is', trainError)

# plot the fitted model with the training data
xModel = np.atleast_2d(np.linspace(Xt[0][0],Xt[-1][0],100)).T

# compute the predicted curve
yModel = model.predict(xModel)

print(yModel)

plt.plot(xModel, yModel, 'r')
```

Real Dataset Example

```
# read the shampoo sales dataset
data = pd.read_csv('data/shampoo.csv')

# save the targets
y = data['Sales'].to_numpy()

# make a column vector of 0s with n elements
X = np.zeros([y.shape[0], 1])

# convert the vector to (0, 1, 2, ..., n)
X[:,0] = [i for i in range(y.shape[0])]

#X = scale(X)

# split the data into train and test sets
trainX, testX, trainY, testY = train_test_split(X, y, test_size = 0.25, random_state = 1)
```

```python
model = GaussianRBFnetwork(d = 1, M = 2, alpha = 0.001)

model.fit(trainX, trainY, epochs = 10000, update = 1000)

# predict the outputs
trainPredictions = model.predict(trainX)

# plot the training points
plt.scatter(trainX, trainY, label = 'Training Data')

# plot the fitted model with the training data
xModel = np.atleast_2d(np.linspace(X[0][0],X[-1][0],100)).T

# compute the predicted curve
yModel = model.predict(xModel)
print(yModel)

plt.plot(xModel, yModel, 'r')

testPredictions = model.predict(testX)

# compute the training and test mean absolute error
trainError = mean_absolute_error(trainY, trainPredictions)
testError = mean_absolute_error(testY, testPredictions)

# return quality metrics
print('The r^2 score is', r2_score(trainY, trainPredictions))
print('The mean absolute error on the training set is', trainError)
print('The mean absolute error on the testing set is', testError)
```