CS 475 Machine Learning (Spring 2017): Assignment 5

Due on May 5th, 2017 at 3:00PM

Raman Arora

Instructions: Please read these instructions carefully and follow them precisely. Feel free to ask the instructor if anything is unclear!

1. Please submit your solutions electronically via Gradescope.
2. Please submit a PDF file for the written component of your solution including derivations, explanations, etc. You can create this PDF in any way you want: typeset the solution in LATEX (recommended), type it in Word or a similar program and convert/export to PDF, or even hand write the solution (legibly!) and scan it to PDF. We recommend that you restrict your solutions to the space allocated for each problem; you may need to adjust the white space by tweaking the argument to `\vspace{xpt}` command. Please name this document <firstname-lastname>-sol5.pdf.
3. Submit the empirical component of the solution (Python code and the documentation of the experiments you are asked to run, including figures) in a Jupyter notebook file. In addition, we require that you save the Python notebook as a pdf file and append it to the rest of the solutions.
4. In addition, you will need to submit your predictions on digit recognition and sentiment classification tasks to `Kaggle`, as described below, according to the competition rules.
5. **Late submissions:** You have a total of 72 late hours for the entire semester that you may use as you deem fit. After you have used up your quota, there will be a penalty of 100% of your grade on late submission. Note that this is different from previous homework assignments for which we gave you additional 48 hours with 50% penalty. This is to ensure release of solutions for HW5 in time for the final exam.
6. **What is the required level of detail?** When asked to derive something, please clearly state the assumptions, if any, and strive for balance: justify any non-obvious steps, but try to avoid superfluous explanations. When asked to plot something, please include the figure as well as the code used to plot it. If multiple entities appear on a plot, make sure that they are clearly distinguishable (by color or style of lines and markers). When asked to provide a brief explanation or description, try to make your answers concise, but do not omit anything you believe is important. When submitting code, please make sure it's reasonably documented, and describe succinctly in the written component of the solution what is done in each py-file.

Name: Tony Melo

I. Generative Models

Here we will consider the Gaussian generative model for $\mathbf{x} \in \mathbb{R}^d$ which assumes equal, isotropic covariance matrices for each class:

$$\Sigma_c = \begin{bmatrix} \sigma_1^2 & \cdots & 0 \\ & \ddots & \\ 0 & \cdots & \sigma_d^2 \end{bmatrix}. \tag{1}$$

The classes of course have separate means. That is, the conditional density of the j-th coordinate of $\mathbf{x}$ given class c is a Gaussian $\mathcal{N}(\mu_{c,j}, \sigma_j)$, with these densities independent (given class) for different values of j.

For simplicity, let's consider a two-class problem, with $y \in \{0, 1\}$. As we discussed in class, when training the generative model, we simply fit $p(\mathbf{x} \mid y)$ and $p(y)$ to the training data, and use the resulting discriminant analysis to produce a decision rule which predicts

$$\widehat{y}(\mathbf{x}) = \operatorname*{argmax}_c \{p(\mathbf{x} \mid y = c)\, p(y = c)\}. \tag{2}$$

However, this model does also produce an (implicit) estimate for the posterior $p(y = c \mid \mathbf{x})$.

Problem 1 [35 points] Show that the posterior $p(y = c \mid \mathbf{x})$ resulting from the generative model above has the same form as the posterior in logistic regression model,

$$p(y = c \mid \mathbf{x}) = \frac{1}{1 + \exp(w_0 + \mathbf{w} \cdot \mathbf{x})}, \tag{3}$$

for appropriate values of $w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d$. (*Advice: Start with Bayes rule, and use the simplifying assumptions in* (1) *to derive the posterior.*)

By Bayes' Rule

$$p(y=c|x) = \frac{p(x|y=C)p(y=C)}{p(x)} \quad \text{w/ } p(x) = \sum_{k=1}^{C} p(x|y=C_k)p(y=C_k)$$

Here $C=\{0,1\}$, so choose W.L.O.G $p(y=0|x)$ since $p(y=0|x) = 1-p(y=1|x)$

$$p(y=0|x) = \frac{p(x|y=0)p(y=0)}{p(x|y=0)p(y=0) + p(x|y=1)p(y=1)} = \frac{\exp(\log p(x|y=0)p(y=0))}{\exp(\log p(x|y=0)p(y=0)) + \exp(\log p(x|y=1)p(y=1))}$$

$$p(y=0|x) = \frac{1}{1+\exp[\log p(x|y=1) + \log p(y=1) - \log p(x|y=0) - \log p(y=0)]}$$

Now define a per-class discriminant function δ_c s.t

$$\delta_c \triangleq \log p(x|y=c) + \log p(y=c)$$

For each class c:

$$\delta_c(x) = -\log(2\pi)^{d/2} - \tfrac{1}{2}\log(|\Sigma|) - \tfrac{1}{2}(x-\mu_c)^T\Sigma^{-1}(x-\mu_c) + \log p(y=c)$$

Since each $c \in C$ is a multivariate Gaussian, we can take $-\log(2\pi)^{d/2} - \tfrac{1}{2}\log(|\Sigma|)$ to be a function of x constant across every class c s.t.:

$$\delta_c(x) = \text{const}(x) - \tfrac{1}{2}(x-\mu_c)^T\Sigma^{-1}(x-\mu_c) + \log p(y=c)$$

$$= \text{const}(x) + 2\mu_c\Sigma^{-1}x + \mu_c^T\Sigma^{-1}\mu_c + \log p(y=c)$$

The decision boundary between δ_1 & δ_0 is $\delta_1 - \delta_0$ ➡

$$\delta_1 - \delta_0 = 2\mu_1^T\Sigma^{-1}x + \mu_1^T\Sigma^{-1}\mu_1 + \log p(y=1) - 2\mu_0\Sigma^{-1}x - \mu_0^T\Sigma^{-1}\mu_0 - \log p(y=0)$$

$\delta_1 - \delta_0 = w^Tx + w_0$ where $w_0 = \mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0 + \log p(y=1) - \log p(y=0)$

and $w = \Sigma^{-1}(\mu_1 - \mu_0)$, so by substituting back,

$$p(y=0|x) = \frac{1}{1+\exp(w^Tx + w_0)}$$

Problem 2 [15 points] The previous problem established that the two models – logistic regression and the linear discriminant analysis based on the isotropic Gaussian model (1) – have the same form of the posterior $p(y \mid \mathbf{x})$. Will the two models produce the same classifier when applied to a given training set? Why or why not?

They will not produce the same model because they estimate their parameters differently. LDA imposes a strict Gaussian prior and models the complete data likelihood while logistic regression only models the conditional likelihood with no prior assumptions

II. Multilayer neural networks

In this section we will return to the problem of handwritten digit recognition and attack it with a two-layer neural network. The general architecture of the network is as follows: let $\mathbf{x}$ be the input (pixels of the $d \times d$ digit image). Then, the first layer (with k hidden units) computes

$$\mathbf{f}_1(\mathbf{x}) = h(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \tag{4}$$

where $\mathbf{W}_1$ is a $k \times d^2$ matrix of weights, $\mathbf{b}_1$ is the k-dimensional vector of bias terms, and h is a nonlinearity (activation function for the hidden units); the syntax in (4) means that h is computed on each element of its k-dimensional argument vector (the output of $\mathbf{W}_1\mathbf{x}+\mathbf{b}_1$), yielding k-dimensional activation vector, called in neural network literature the feature map of the layer.

The second layer computes the network activation:

$$\mathbf{f}_2(\mathbf{x}) = \mathbf{W}_2\mathbf{f}_1(\mathbf{x}) + \mathbf{b}_2, \tag{5}$$

where $\mathbf{W}_2$ is a $10 \times k$ matrix and $\mathbf{b}_2$ a 10-dimensional vector. The values of the 10-dimensional vector $\mathbf{f}_2$ are interpreted as the scores (unnormalized log-probabilities) for the 10 classes. So, the posterior for class c computed by the network is

$$p(c|\mathbf{x}) = \frac{\exp(f_{2,c}(\mathbf{x}))}{\sum_j \exp(f_{2,j}(\mathbf{x}))}, \tag{6}$$

and the loss used to train the network is

$$-\frac{1}{n}\sum_{i=1}^{n} \log p(y_i|\mathbf{x}_i) \tag{7}$$

where the summation is over the indices of training examples. You can of course add regularization (e.g., squared norm penalty over elements of $\mathbf{W}_1$, $\mathbf{W}_2$) to the loss.

Again, we will revisit and improve our results on a familiar data set, this time MNIST.

Problem 3 [50 points] Implement a two-layer neural network for MNIST. We have set up a Kaggle competition for the clean version:

- `https://inclass.kaggle.com/c/cs475-neuralnet-mnist`

We have provided skeleton code `PS5_neuralnet.ipynb` that features a complete forward and backward pass. However, you will have to work out what the gradients are.

We provide several different update algorithms in `utils.py`: SGD, SGD with momentum and SGD with Nestorov momentum. Try these methods, and report the differences in their performances.

You should also decide what the activation function h is. The options are ReLU, Logistic and Tanh. For all activations you try, you need to implement their forward and backward functions.

This task will require you to understand the code that we have provided, so read the inline documentation carefully. We encourage you to re-use parts of your code and hyperparameters from

Problem Set 2. e.g. `softmaxloss`, stopping criteria, stepsize reduction, batch_size, etc.

If you used loops in your solution in Problem Set 2, you will quickly find that it is not fast enough. Learn to write things in terms of matrix multiplications.

Optional:

More layers You can try more than two-layers, and it should be trivial given the code we provide.

Dropout Dropout is a popular technique in neural network. Instead of using (5), the feed-forward operation during training becomes:

$$r_j \sim Bernoulli(p) \tag{8}$$

$$\mathbf{f}_2(\mathbf{x}) = \mathbf{W}_2(\frac{1}{p}\mathbf{r} * \mathbf{f}_1(\mathbf{x})) + \mathbf{b}_2, \tag{9}$$

$\mathbf{r}$ is a vector of independent Bernoulli random variables each of which has probability p of being 1. This vector is multiplied element-wisely to the output of previous layer. Some of the outputs of previous layer are droped out, and the remained are scaled by $\frac{1}{p}$ to make sure the "scale" of the input to remain the same. During evaluation, we do the same thing as (5).

PS5_neuralnet

May 5, 2017

In [1]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
from utils import *
```

In [2]:
```
# superclass of modules
class Module:
    """
    Module is a super class. It could be a single layer, or a multilayer pe
    """

    def __init__(self):
        self.train = True
        return

    def forward(self, _input):
        """
        h = f(z); z is the input, and h is the output.

        Inputs:
        _input: z

        Returns:
        output h
        """
        pass

    def backward(self, _input, _gradOutput):
        """
        Compute:
        gradient w.r.t. _input
        gradient w.r.t. trainable parameters

        Inputs:
        _input: z
        _gradOutput: dL/dh
```

```
            Returns:
            gradInput: dL/dz
            """
            pass

        def parameters(self):
            """
            Return the value of trainable parameters and its corresponding grad

            Returns:
            params, gradParams
            """
            pass

        def training(self):
            """
            Turn the module into training mode.(Only useful for Dropout layer)
            Ignore it if you are not using Dropout.
            """
            self.train = True

        def evaluate(self):
            """
            Turn the module into evaluate mode.(Only useful for Dropout layer)
            Ignore it if you are not using Dropout.
            """
            self.train = False
```

In [3]:
```
class Sequential(Module):
    """
    Sequential provides a way to plug layers together in a feed-forward mar
    """
    def __init__(self):
        Module.__init__(self)
        self.layers = [] # layers contain all the layers in order

    def add(self, layer):
        self.layers.append(layer) # Add another layer at the end

    def size(self):
        return len(self.layers) # How many layers.

    def forward(self, _input):
        """
        Feed forward through all the layers, and return the output of the l
        """
        # self._inputs saves the input of each layer
```

```
        # self._inputs[i] is the input of i-th layer
        self._inputs = [_input]

        # YOUR CODE HERE
        for layers in self.layers:
            self._inputs.append(layers.forward(self._inputs[-1]))

        # The last element of self._inputs is the output of last layer
        self._output = self._inputs[-1]
        return self._output

    def backward(self, _input, _gradOutput):
        """
        Backpropogate through all the layers using chain rule.
        """
        # self._gradInputs[i] is the gradient of loss w.r.t. the input of i
        self._gradInputs = [None] * (self.size() + 1)
        self._gradInputs[self.size()] = _gradOutput

        # YOUR CODE HERE
        for i in range(self.size()-1, -1, -1):
            self._gradInputs[i] = self.layers[i].backward(self._inputs[i],

        self._gradInput = self._gradInputs[0]
        return self._gradInput

    def parameters(self):
        """
        Return trainable parameters and its corresponding gradient in a nes
        """
        params = []
        gradParams = []
        for m in self.layers:
            _p, _g = m.parameters()
            if _p is not None:
                params.append(_p)
                gradParams.append(_g)
        return params, gradParams

    def training(self):
        """
        Turn all the layers into training mode
        """
        Module.training(self)
        for m in self.layers:
            m.training()

    def evaluate(self):
```

```
        """
        Turn all the layers into evaluate mode
        """
        Module.evaluate(self)
        for m in self.layers:
            m.evaluate()
```

In [4]:
```
class FullyConnected(Module):
    """
    Fully connected layer
    """
    def __init__(self, inputSize, outputSize):
        Module.__init__(self)
        # Initalization
        stdv = 1./np.sqrt(inputSize)

        self.weight = np.random.uniform(-stdv, stdv, (inputSize, outputSize
        self.gradWeight = np.ndarray((inputSize, outputSize))
        self.bias = np.random.uniform(-stdv, stdv, outputSize)
        self.gradBias = np.ndarray(outputSize)

    def forward(self, _input):
        """
        output = W * input + b

        _input:
        N x inputSize matrix

        """
        self._output = np.dot(_input, self.weight) + self.bias

        return self._output

    def backward(self, _input, _gradOutput):
        """
        _input:
        N x inputSize matrix
        _gradOutputSize:
        N x outputSize matrix
        """
        self.gradWeight = np.dot(_input.T, _gradOutput)/_input.shape[0]
        self.gradBias = np.dot(np.ones(_gradOutput.shape[0]), _gradOutput)/

        self._gradInput = np.dot(_gradOutput, self.weight.T)
        return self._gradInput

    def parameters(self):
        """
```

```
        Return weight and bias and their g
        """
        return [self.weight, self.bias], [self.gradWeight, self.gradBias]
```

In [5]:
```
class ReLU(Module):
    """
    ReLU activation, not trainable.
    """
    def __init__(self):
        Module.__init__(self)
        return

    def forward(self, _input):
        """
        output = max(0, input)

        _input:
        N x d matrix
        """
        self._output = np.maximum(0, _input)
        return self._output

    def backward(self, _input, _gradOutput):
        """
        gradInput = gradOutput * mask
        mask = _input > 0

        _input:
        N x d matrix

        _gradOutput:
        N x d matrix
        """


        mask = _input > 0
        self._gradInput = _gradOutput * mask
        return self._gradInput

    def parameters(self):
        """
        No trainable parametersm, return None
        """
        return None, None
```

In [6]:
```
# Optional
class Logistic(Module):
    """
```

```
        Logistic activation, not trainable.
        """
        def __init__(self):
            Module.__init__(self)
            return

        def forward(self, _input):
            self._output = 1.0 / (1.0 + np.exp(-_input))
            return self._output

        def backward(self, _input, _gradOutput):
            self._gradInput = (1.0 - self.output) * self.output * _gradOutput
            return self._gradInput

        def parameters(self):
            return None, None
```

In [7]:
```
# Optional
class Dropout(Module):
    """
    A dropout layer
    """
    def __init__(self, p = 0.5):
        Module.__init__(self)
        self.p = p #self.p is the drop rate, if self.p is 0, then it's a id

    def forward(self, _input):
        self._output = _input
        # YOUR CODE HERE
        # Need to take care of training mode and evaluation mode
        return self._output

    def backward(self, _input, _gradOutput):
        self._gradInput = _gradOutput
        #YOUR CODE HERE
        return self._gradInput

    def parameters(self):
        """
        No trainable parameters.
        """
        return None, None
```

In [8]:
```
class SoftMaxLoss(object):
    def __init__(self):
        return

    def softmax(self, Z):
```

```
            import math
            softmax_eq = lambda x: np.divide(np.array([math.e**(i-max(x)) for i

            S = np.zeros(np.shape(Z))
            for i in range(S.shape[0]):
                S[i,:] = softmax_eq(Z[i,:])
            return S

        def forward(self, _input, _label):
            """
            Softmax and cross entropy loss layer. Should return a scalar, since
            loss. (It's almost identical to what in hw2)

            _input: N x C
            _labels: N x C, one-hot

            Returns: loss (scalar)
            """
            S = self.softmax(_input)
            self._output = -np.sum((_label * np.log(S))) / _label.shape[0]
            return self._output

        def backward(self, _input, _label):
            diff = np.amax(_input, axis=1)
            S = np.exp(np.transpose(np.subtract(np.transpose(_input), diff)))
            S = np.transpose(np.divide(np.transpose(S), np.sum(S, axis=1)))
            self._gradInput = S - _label
            return self._gradInput
```

In [9]:
```
# Test softmaxloss, the relative error should be small enough
def test_sm():
    crit = SoftMaxLoss()
    gt = np.zeros((3, 10))
    gt[np.arange(3), np.array([1,2,3])] = 1
    x = np.random.random((3,10))
    def test_f(x):
        return crit.forward(x, gt)

    crit.forward(x, gt)

    gradInput = crit.backward(x, gt)
    gradInput_num = numeric_gradient(test_f, x, 1, 1e-6)
    #print(gradInput)
    #print(gradInput_num)
    print(relative_error(gradInput, gradInput_num, 1e-8))

test_sm()
```

```
0.500000003159
```

```
In [10]: # Test modules, all the relative errors should be small enough
         def test_module(model):

             model.evaluate()

             crit = TestCriterion()
             gt = np.random.random((3,10))
             x = np.random.random((3,10))
             def test_f(x):
                 return crit.forward(model.forward(x), gt)

             gradInput = model.backward(x, crit.backward(model.forward(x), gt))
             gradInput_num = numeric_gradient(test_f, x, 1, 1e-6)
             print(relative_error(gradInput, gradInput_num, 1e-8))

         # Test fully connected
         model = FullyConnected(10, 10)
         test_module(model)

         # Test ReLU
         model = ReLU()
         test_module(model)

         # Test Dropout
         # model = Dropout()
         # test_module(model)
         # You can only test dropout in evaluation mode.

         # Test Sequential
         model = Sequential()
         model.add(FullyConnected(10, 10))
         model.add(ReLU())
         #model.add(Dropout())
         test_module(model)

9.79148171941e-08
7.36022825478e-10
1.14895878068e-09


In [11]: # Test gradient descent, the loss should be lower and lower
         trainX = np.random.random((10,5))

         model = Sequential()
         model.add(FullyConnected(5, 3))
         model.add(ReLU())
```

```
        #model.add(Dropout())
        model.add(FullyConnected(3, 1))

        crit = TestCriterion()

        it = 0
        state = None
        while True:
            output = model.forward(trainX)
            loss = crit.forward(output, None)
            if it % 100 == 0:
                print(loss)
            doutput = crit.backward(output, None)
            model.backward(trainX, doutput)
            params, gradParams = model.parameters()
            sgd(params, gradParams, 0.01, 0.8)
            if it > 1000:
                break
            it += 1
```

```
0.0449669768688
0.0278891242473
0.0262768875087
0.0247409807511
0.0232851962996
0.0222457335459
0.0212584139104
0.0203039129729
0.0193733688405
0.0184706985933
0.017594509143
```

Now we start to work on real data.

In [12]:
```
        import MNIST_utils
        data_fn = "CLEAN_MNIST_SUBSETS.h5"

        # We only consider large set this time
        print("Load large trainset.")
        Xlarge,Ylarge = MNIST_utils.load_data(data_fn, "large_train")
        print(Xlarge.shape)
        print(Ylarge.shape)

        print("Load valset.")
        Xval,Yval = MNIST_utils.load_data(data_fn, "val")
        print(Xval.shape)
        print(Yval.shape)
```

```
Load large trainset.
(7000L, 576L)
(7000L, 10L)
Load valset.
(2000L, 576L)
(2000L, 10L)
```

```
In [13]: def predict(X, model):
             """
             Evaluate the soft predictions of the model.
             Input:
             X : N x d array (no unit terms)
             model : a multi-layer perceptron
             Output:
             yhat : N x C array
                 yhat[n][:] contains the score over C classes for X[n][:]
             """
             return model.forward(X)

         def error_rate(X, Y, model):
             """
             Compute error rate (between 0 and 1) for the model
             """
             model.evaluate()
             res = 1 - (model.forward(X).argmax(-1) == Y.argmax(-1)).mean()
             model.training()
             return res

         from copy import deepcopy

         def runTrainVal(X,Y,model,Xval,Yval,trainopt):
             """
             Run the train + evaluation on a given train/val partition
             trainopt: various (hyper)parameters of the training procedure
             During training, choose the model with the lowest validation error. (e
             """

             eta = trainopt['eta']

             N = X.shape[0] # number of data points in X

             # Save the model with lowest validation error
             minValError = np.inf
             saved_model = None

             shuffled_idx = np.random.permutation(N)
             start_idx = 0
```

```
        for iteration in range(trainopt['maxiter']):
            if iteration % int(trainopt['eta_frac'] * trainopt['maxiter']) ==
                eta *= trainopt['etadrop']
            # form the next mini-batch
            stop_idx = min(start_idx + trainopt['batch_size'], N)
            batch_idx = range(N)[int(start_idx):int(stop_idx)]
            bX = X[shuffled_idx[batch_idx],:]
            bY = Y[shuffled_idx[batch_idx],:]

            score = model.forward(bX)
            loss = crit.forward(score, bY)
            # print(loss)
            dscore = crit.backward(score, bY)
            model.backward(bX, dscore)

            # Update the data using
            params, gradParams = model.parameters()
            sgd(params, gradParams, eta, weight_decay = trainopt['lambda'])
            start_idx = stop_idx % N

            if (iteration % trainopt['display_iter']) == 0:
                #compute train and val error; multiply by 100 for readability
                trainError = 100 * error_rate(X, Y, model)
                valError = 100 * error_rate(Xval, Yval, model)
                print('{:8} batch loss: {:.3f} train error: {:.3f} val error:

                if valError < minValError:
                    saved_model = deepcopy(model)
                    minValError = valError

        return saved_model, minValError, trainError
```

In [14]:
```
def build_model(input_size, hidden_size, output_size, activation_func = 'R
    """
    Build the model:
    input_size: the dimension of input data
    hidden_size: the dimension of hidden vector
    output_size: the output size of final layer.
    activation_func: ReLU, Logistic, Tanh, etc. (Need to be implemented by
    dropout: the dropout rate: if dropout == 0, this is equivalent to no d
    """
    model = Sequential()
    model.add(FullyConnected(input_size, output_size))
    model.add(ReLU())
    return model
```

In [15]:
```
# -- training options
trainopt = {
```

```
    'eta': .1,    # initial learning rate
    'maxiter': 20000,    # max number of iterations (updates) of SGD
    'display_iter': 500,  # display batch loss every display_iter updates
    'batch_size': 100,
    'etadrop': .5, # when dropping eta, multiply it by this number (e.g.,
    'eta_frac': .25  #
}

NFEATURES = Xlarge.shape[1]

# we will maintain a record of models trained for different values of lamb
# these will be indexed directly by lambda value itself
trained_models = dict()

# set the (initial?) set of lambda values to explore
lambdas = np.array([0, 0.001, 0.01, 0.1])
hidden_sizes = np.array([10])

for lambda_ in lambdas:
    for hidden_size_ in hidden_sizes:
        trainopt['lambda'] = lambda_
        model = build_model(NFEATURES, hidden_size_, 10, dropout = 0)
        crit = SoftMaxLoss()
        # -- model trained on large train set
        trained_model,valErr,trainErr = runTrainVal(Xlarge, Ylarge, model,
        trained_models[(lambda_, hidden_size_)] = {'model': trained_model,
        print('train set model: -> lambda= %.4f, train error: %.2f, val er

best_trained_lambda = 0.
best_trained_model = None
best_trained_val_err = 100.
for (lambda_, hidden_size_), results in trained_models.items():
    print('lambda= %.4f, hidden size: %5d, val error: %.2f' %(lambda_, hid
    if results['val_err'] < best_trained_val_err:
        best_trained_val_err = results['val_err']
        best_trained_model = results['model']
        best_trained_lambda = lambda_

print("Best train model val err:", best_trained_val_err)
print("Best train model lambda:", best_trained_lambda)
```

```
   0 batch loss: 2.325 train error: 86.000 val error: 84.600
 500 batch loss: 0.373 train error: 11.114 val error: 10.250
1000 batch loss: 0.322 train error: 9.471 val error: 9.000
1500 batch loss: 0.287 train error: 8.914 val error: 8.550
2000 batch loss: 0.381 train error: 8.457 val error: 7.900
2500 batch loss: 0.262 train error: 7.957 val error: 7.650
3000 batch loss: 0.331 train error: 7.714 val error: 7.750
```

```
  3500 batch loss: 0.297 train error: 7.543 val error: 7.600
  4000 batch loss: 0.224 train error: 7.214 val error: 7.750
  4500 batch loss: 0.230 train error: 7.243 val error: 7.700
  5000 batch loss: 0.214 train error: 7.157 val error: 7.750
  5500 batch loss: 0.314 train error: 7.000 val error: 7.750
  6000 batch loss: 0.197 train error: 6.871 val error: 7.800
  6500 batch loss: 0.304 train error: 6.800 val error: 7.800
  7000 batch loss: 0.270 train error: 6.729 val error: 7.900
  7500 batch loss: 0.202 train error: 6.600 val error: 7.850
  8000 batch loss: 0.210 train error: 6.714 val error: 7.750
  8500 batch loss: 0.200 train error: 6.643 val error: 7.900
  9000 batch loss: 0.296 train error: 6.586 val error: 7.850
  9500 batch loss: 0.179 train error: 6.457 val error: 7.800
 10000 batch loss: 0.295 train error: 6.457 val error: 7.700
 10500 batch loss: 0.259 train error: 6.414 val error: 7.700
 11000 batch loss: 0.192 train error: 6.400 val error: 7.750
 11500 batch loss: 0.200 train error: 6.457 val error: 7.750
 12000 batch loss: 0.192 train error: 6.443 val error: 7.750
 12500 batch loss: 0.284 train error: 6.400 val error: 7.750
 13000 batch loss: 0.171 train error: 6.343 val error: 7.750
 13500 batch loss: 0.292 train error: 6.329 val error: 7.700
 14000 batch loss: 0.254 train error: 6.314 val error: 7.750
 14500 batch loss: 0.187 train error: 6.300 val error: 7.750
 15000 batch loss: 0.195 train error: 6.329 val error: 7.900
 15500 batch loss: 0.188 train error: 6.300 val error: 7.750
 16000 batch loss: 0.278 train error: 6.286 val error: 7.800
 16500 batch loss: 0.166 train error: 6.286 val error: 7.850
 17000 batch loss: 0.289 train error: 6.257 val error: 7.700
 17500 batch loss: 0.250 train error: 6.257 val error: 7.750
 18000 batch loss: 0.184 train error: 6.229 val error: 7.850
 18500 batch loss: 0.192 train error: 6.214 val error: 7.800
 19000 batch loss: 0.186 train error: 6.229 val error: 7.800
 19500 batch loss: 0.275 train error: 6.229 val error: 7.750
train set model: -> lambda= 0.0000, train error: 6.23, val error: 7.60
     0 batch loss: 2.266 train error: 79.629 val error: 79.100
   500 batch loss: 0.239 train error: 11.400 val error: 10.500
  1000 batch loss: 0.412 train error: 9.757 val error: 8.950
  1500 batch loss: 0.244 train error: 9.000 val error: 7.950
  2000 batch loss: 0.308 train error: 8.529 val error: 7.900
  2500 batch loss: 0.226 train error: 8.157 val error: 7.750
  3000 batch loss: 0.442 train error: 7.757 val error: 7.550
  3500 batch loss: 0.290 train error: 7.457 val error: 7.500
  4000 batch loss: 0.109 train error: 7.157 val error: 7.600
  4500 batch loss: 0.336 train error: 7.000 val error: 7.500
  5000 batch loss: 0.174 train error: 6.929 val error: 7.650
  5500 batch loss: 0.266 train error: 6.886 val error: 7.700
  6000 batch loss: 0.186 train error: 6.829 val error: 7.800
  6500 batch loss: 0.390 train error: 6.800 val error: 7.800
```

```
 7000 batch loss: 0.258 train error: 6.729 val error: 7.750
 7500 batch loss: 0.097 train error: 6.643 val error: 7.800
 8000 batch loss: 0.319 train error: 6.586 val error: 7.850
 8500 batch loss: 0.161 train error: 6.557 val error: 7.800
 9000 batch loss: 0.253 train error: 6.471 val error: 7.650
 9500 batch loss: 0.172 train error: 6.514 val error: 7.700
10000 batch loss: 0.370 train error: 6.557 val error: 7.800
10500 batch loss: 0.245 train error: 6.443 val error: 7.750
11000 batch loss: 0.092 train error: 6.371 val error: 7.700
11500 batch loss: 0.311 train error: 6.357 val error: 7.650
12000 batch loss: 0.153 train error: 6.343 val error: 7.700
12500 batch loss: 0.246 train error: 6.343 val error: 7.600
13000 batch loss: 0.166 train error: 6.371 val error: 7.700
13500 batch loss: 0.360 train error: 6.357 val error: 7.650
14000 batch loss: 0.239 train error: 6.314 val error: 7.650
14500 batch loss: 0.090 train error: 6.257 val error: 7.600
15000 batch loss: 0.306 train error: 6.300 val error: 7.550
15500 batch loss: 0.149 train error: 6.300 val error: 7.650
16000 batch loss: 0.242 train error: 6.300 val error: 7.600
16500 batch loss: 0.162 train error: 6.300 val error: 7.550
17000 batch loss: 0.353 train error: 6.286 val error: 7.550
17500 batch loss: 0.235 train error: 6.271 val error: 7.600
18000 batch loss: 0.089 train error: 6.214 val error: 7.550
18500 batch loss: 0.304 train error: 6.257 val error: 7.550
19000 batch loss: 0.147 train error: 6.243 val error: 7.500
19500 batch loss: 0.240 train error: 6.271 val error: 7.600
train set model: -> lambda= 0.0010, train error: 6.27, val error: 7.50
    0 batch loss: 2.330 train error: 89.029 val error: 88.600
  500 batch loss: 0.362 train error: 11.443 val error: 10.300
 1000 batch loss: 0.442 train error: 9.800 val error: 8.800
 1500 batch loss: 0.289 train error: 9.029 val error: 8.200
 2000 batch loss: 0.319 train error: 8.471 val error: 8.100
 2500 batch loss: 0.346 train error: 8.000 val error: 7.800
 3000 batch loss: 0.266 train error: 7.743 val error: 7.900
 3500 batch loss: 0.404 train error: 7.471 val error: 7.800
 4000 batch loss: 0.126 train error: 7.329 val error: 7.600
 4500 batch loss: 0.312 train error: 7.329 val error: 7.850
 5000 batch loss: 0.202 train error: 7.086 val error: 7.750
 5500 batch loss: 0.262 train error: 6.943 val error: 7.950
 6000 batch loss: 0.288 train error: 6.843 val error: 7.900
 6500 batch loss: 0.226 train error: 6.729 val error: 7.800
 7000 batch loss: 0.368 train error: 6.671 val error: 7.850
 7500 batch loss: 0.107 train error: 6.686 val error: 7.800
 8000 batch loss: 0.291 train error: 6.671 val error: 7.750
 8500 batch loss: 0.186 train error: 6.643 val error: 7.700
 9000 batch loss: 0.246 train error: 6.571 val error: 7.800
 9500 batch loss: 0.267 train error: 6.429 val error: 7.800
10000 batch loss: 0.210 train error: 6.414 val error: 7.800
```

```
 10500 batch loss: 0.351 train error: 6.400 val error: 7.850
 11000 batch loss: 0.100 train error: 6.371 val error: 7.800
 11500 batch loss: 0.281 train error: 6.400 val error: 7.850
 12000 batch loss: 0.177 train error: 6.400 val error: 7.850
 12500 batch loss: 0.237 train error: 6.357 val error: 7.850
 13000 batch loss: 0.258 train error: 6.286 val error: 7.850
 13500 batch loss: 0.203 train error: 6.271 val error: 7.800
 14000 batch loss: 0.344 train error: 6.229 val error: 7.850
 14500 batch loss: 0.096 train error: 6.257 val error: 7.800
 15000 batch loss: 0.276 train error: 6.243 val error: 7.800
 15500 batch loss: 0.173 train error: 6.200 val error: 7.800
 16000 batch loss: 0.232 train error: 6.186 val error: 7.850
 16500 batch loss: 0.252 train error: 6.171 val error: 7.800
 17000 batch loss: 0.199 train error: 6.186 val error: 7.800
 17500 batch loss: 0.339 train error: 6.186 val error: 7.800
 18000 batch loss: 0.095 train error: 6.186 val error: 7.800
 18500 batch loss: 0.273 train error: 6.186 val error: 7.750
 19000 batch loss: 0.170 train error: 6.171 val error: 7.800
 19500 batch loss: 0.230 train error: 6.157 val error: 7.850
train set model: -> lambda= 0.0100, train error: 6.16, val error: 7.60
     0 batch loss: 2.312 train error: 86.443 val error: 87.100
   500 batch loss: 0.626 train error: 18.557 val error: 17.750
  1000 batch loss: 0.506 train error: 17.286 val error: 16.750
  1500 batch loss: 0.604 train error: 16.614 val error: 16.300
  2000 batch loss: 0.631 train error: 16.243 val error: 16.050
  2500 batch loss: 0.444 train error: 15.843 val error: 16.000
  3000 batch loss: 0.370 train error: 15.657 val error: 15.850
  3500 batch loss: 0.473 train error: 15.386 val error: 15.800
  4000 batch loss: 0.457 train error: 15.286 val error: 15.800
  4500 batch loss: 0.394 train error: 15.171 val error: 15.850
  5000 batch loss: 0.523 train error: 15.057 val error: 15.800
  5500 batch loss: 0.570 train error: 15.014 val error: 15.700
  6000 batch loss: 0.396 train error: 15.000 val error: 15.750
  6500 batch loss: 0.332 train error: 14.957 val error: 15.800
  7000 batch loss: 0.431 train error: 14.957 val error: 15.900
  7500 batch loss: 0.435 train error: 14.914 val error: 15.800
  8000 batch loss: 0.370 train error: 14.857 val error: 15.900
  8500 batch loss: 0.504 train error: 14.786 val error: 15.900
  9000 batch loss: 0.555 train error: 14.771 val error: 16.050
  9500 batch loss: 0.382 train error: 14.714 val error: 15.900
 10000 batch loss: 0.318 train error: 14.671 val error: 16.100
 10500 batch loss: 0.416 train error: 14.657 val error: 15.850
 11000 batch loss: 0.425 train error: 14.629 val error: 15.800
 11500 batch loss: 0.357 train error: 14.614 val error: 15.800
 12000 batch loss: 0.490 train error: 14.629 val error: 15.800
 12500 batch loss: 0.547 train error: 14.586 val error: 15.900
 13000 batch loss: 0.375 train error: 14.600 val error: 15.800
 13500 batch loss: 0.312 train error: 14.571 val error: 15.950
```

```
   14000 batch loss: 0.409 train error: 14.529 val error: 15.900
   14500 batch loss: 0.421 train error: 14.557 val error: 15.900
   15000 batch loss: 0.351 train error: 14.543 val error: 16.000
   15500 batch loss: 0.482 train error: 14.514 val error: 15.900
   16000 batch loss: 0.542 train error: 14.514 val error: 16.000
   16500 batch loss: 0.371 train error: 14.500 val error: 15.950
   17000 batch loss: 0.309 train error: 14.500 val error: 15.900
   17500 batch loss: 0.405 train error: 14.471 val error: 15.950
   18000 batch loss: 0.419 train error: 14.500 val error: 15.900
   18500 batch loss: 0.347 train error: 14.486 val error: 15.900
   19000 batch loss: 0.478 train error: 14.457 val error: 15.950
   19500 batch loss: 0.540 train error: 14.471 val error: 16.000
train set model: -> lambda= 0.1000, train error: 14.47, val error: 15.70
lambda= 0.0010, hidden size:    10, val error: 7.50
lambda= 0.0100, hidden size:    10, val error: 7.60
lambda= 0.1000, hidden size:    10, val error: 15.70
lambda= 0.0000, hidden size:    10, val error: 7.60
Best train model val err: 7.5
Best train model lambda: 0.001
```

In [16]:
```
#Generate a Kaggle submission file using `model`
kaggleX = MNIST_utils.load_data(data_fn, 'kaggle')
kaggleYhat = predict(kaggleX, best_trained_model).argmax(-1)
save_submission('submission-mnist.csv', kaggleYhat)
```

```
('Saved:', 'submission-mnist.csv')
```

In [ ]:

In [ ]: