Question 1.1

Bayes' Rule: $P(A|B)\cdot P(B) = P(B|A)\cdot P(A)$

$$P(t=1|\vec{x}) = P(\vec{x}|t=1)\cdot P(t=1)/P(\vec{x})$$

$$= P(\vec{x}|t=1)\cdot P(t=1) \;/\; (P(\vec{x}|t=1)\cdot P(t=1) + P(\vec{x}|t=0)\cdot P(t=0))$$

$$P(\vec{x}|t=1) = \prod_{i=1}^{D} P(x_i|t=1) = \left(\frac{1}{\sqrt{2\pi}}\right)^D \cdot \prod_{i=1}^{D}\frac{1}{\sigma_i}\cdot \exp\left(\sum_{i=1}^{D} -\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}\right)$$

$$P(\vec{x}|t=0) = \prod_{i=1}^{D} P(x_i|t=0) = \left(\frac{1}{\sqrt{2\pi}}\right)^D \cdot \prod_{i=1}^{D}\frac{1}{\sigma_i}\cdot \exp\left(\sum_{i=1}^{D} -\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}\right)$$

$$P(t=1|\vec{x}) = \frac{1}{1 + (P(\vec{x}|t=0)P(t=0))/(P(\vec{x}|t=1)\cdot P(t=1))}$$

↑ In this question, we know the underlying true distribution or you are given the learned distribution

$$= \frac{1}{1 + \frac{(1-\alpha)}{\alpha}\exp\left(\sum_{i=1}^{D} -\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2} + \frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}\right)}$$

$$= \frac{1}{1 + \frac{1-\alpha}{\alpha}\exp\left(-\sum_{i=1}^{D}\frac{(x_i-\mu_{i0})^2-(x_i-\mu_{i1})^2}{2\sigma_i^2}\right)}$$

$$(x_i-\mu_{i0})^2 - (x_i-\mu_{i1})^2 = (x_i-\mu_{i0}+x_i-\mu_{i1})(x_i-\mu_{i0}-x_i+\mu_{i1})$$

$$= (2x_i-\mu_{i0}-\mu_{i1})(\mu_{i1}-\mu_{i0})$$

$$= 2x_i\mu_{i1} - \mu_{i0}\mu_{i1} - \mu_{i1}^2 - 2x_i\mu_{i0} + \mu_{i0}^2 + \mu_{i1}\mu_{i0}$$

$$= 2(\mu_{i1}-\mu_{i0})x_i + \mu_{i0}^2 - \mu_{i1}^2$$

$$-\sum_{i=1}^{D}\frac{(x_i-\mu_{i0})^2-(x_i-\mu_{i1})^2}{2\sigma_i^2} = -\sum_{i=1}^{D}\frac{(\mu_{i1}-\mu_{i0})}{\sigma_i^2}x_i - \sum_{i=1}^{D}\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}$$

Thus $$P(t=1|\vec{x}) = 1\Big/\left(1 + \frac{(1-\alpha)}{\alpha}\exp\left(-\sum_{i=1}^{D}\frac{(\mu_{i1}-\mu_{i0})}{\sigma_i^2}x_i - \sum_{i=1}^{D}\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}\right)\right)$$

$$= 1\Big/\left(1+\exp\left(-\sum_{i=1}^{D}\frac{(\mu_{i1}-\mu_{i0})}{\sigma_i^2}x_i - \left(\sum_{i=1}^{D}\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2} - \log\frac{1-\alpha}{\alpha}\right)\right)\right)$$

$$\vec{w} = (w_1,\ldots,w_D)^T \text{ where } w_i = \frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2} \qquad b = \sum_{i=1}^{D}\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2} - \log\frac{1-\alpha}{\alpha}$$

Question 1.2 the sampling process is independent

$$\mathcal{L}(\vec{w},b) = -\log p(t_1,t_2,\dots,t_n \mid x^{(1)}, x^{(2)}, \dots, x^{(n)}, \vec{w}, b)$$

$$= -\log\left[p(t_1|\cdots)\cdot p(t_2|\cdots)\cdot\cdots\cdot p(t_n|\cdots)\right]$$

$$= -\log p(t_1|\cdots) - \log p(t_2|\cdots) - \cdots - \log p(t_n|\cdots)$$

$$= -\sum_{i=1}^{n} \log p(t_i|x^{(i)},\vec{w},b)$$

$$= -\sum_{i=1}^{n} \log\left[t_i\cdot P(t_i=1|x^{(i)},\vec{w},b) + (1-t_i)\cdot P(t_i=0|x^{(i)},\vec{w},b)\right]$$

Since $t_i = 1$ or 0,

$$= -\sum_{i=1}^{n}\left(t_i \log P(t_i=1|x^{(i)},\vec{w},b) + (1-t_i)\log P(t_i=0|x^{(i)},\vec{w},b)\right)$$

$$= -\left(\sum_{i=1}^{n} t_i \log P(t_i=1|x^{(i)},\vec{w},b) + \sum_{i=1}^{n}(1-t_i)\log P(t_i=0|x^{(i)},\vec{w},b)\right)$$

$$\boxed{\mathcal{L}(\vec{w},b) = -\left[\sum_{i=1}^{n}\left(t_i \log\sigma(\vec{w}^T\vec{x}^{(i)}+b)\right) + \sum_{i=1}^{n}\left[(1-t_i)\log(1-\sigma(\vec{w}^T\vec{x}^{(i)}+b))\right]\right]}$$

Let $u_i = \vec{w}^T\vec{x}^{(i)} + b$

$$\frac{\partial \mathcal{L}(\vec{w},b)}{\partial \vec{w}} = -\left[\sum_{i=1}^{n}\frac{\partial(t_i\log\sigma(u^{(i)}))}{\partial\vec{w}} + \sum_{i=1}^{n}\frac{\partial\left[(1-t_i)\log(1-\sigma(u^{(i)}))\right]}{\partial\vec{w}}\right]$$

$$= -\left[\sum_{i=1}^{n} t_i\frac{\partial\log\sigma(u^{(i)})}{\partial\vec{w}} + \sum_{i=1}^{n}(1-t_i)\frac{\partial\log(1-\sigma(u^{(i)}))}{\partial\vec{w}}\right]$$

and $$\frac{\partial\log\sigma(u^{(i)})}{\partial\vec{w}} = \frac{1}{\sigma(u^{(i)})}\cdot\sigma'(u^{(i)})\cdot\frac{\partial u^{(i)}}{\partial\vec{w}}$$

$$\frac{\partial u^{(i)}}{\partial\vec{w}} = \frac{\partial(\vec{w}^T\vec{x}^{(i)}+b)}{\partial\vec{w}} = \vec{x}^{(i)} \qquad (1)$$

$$\sigma'(u^{(i)}) = \left(\frac{1}{1+\exp(-u^{(i)})}\right)' = -\frac{1}{(1+\exp(-u^{(i)}))^2}\cdot(-\exp(-u^{(i)}))$$

$$\sigma'(u^{(i)}) = \frac{\exp(-u^{(i)})}{(1+\exp(-u^{(i)}))^2} \qquad (2)$$

---

according to (1), (2)

$$\frac{\partial\log\sigma(u^{(i)})}{\partial\vec{w}} = \frac{1}{\sigma(u^{(i)})}\cdot\frac{\exp(-u^{(i)})}{(1+\exp(-u^{(i)}))^2}\cdot\vec{x}^{(i)}$$

$$\frac{\partial \log \sigma(u^{(i)})}{\partial \vec{w}} = \frac{(1+\exp(-u^{(i)}))\exp(-u^{(i)})}{(1+\exp(-u^{(i)}))^2} \cdot \vec{x}^{(i)}$$

$$= \frac{\exp(-u^{(i)})}{1+\exp(-u^{(i)})} \cdot \vec{x}^{(i)}$$

$$\frac{\partial \log(1-\sigma(u^{(i)}))}{\partial \vec{w}} = \frac{1}{1-\sigma(u^{(i)})} \cdot (-1) \cdot \sigma'(u^{(i)}) \cdot \frac{\partial u^{(i)}}{\partial \vec{w}}$$

$$= \frac{1}{\frac{1}{1+\exp(-u^{(i)})} - 1} \cdot \frac{\exp(-u^{(i)})}{(1+\exp(-u^{(i)}))^2} \cdot \vec{x}^{(i)}$$

$$= \frac{1+\exp(-u^{(i)})}{-\exp(-u^{(i)})} \cdot \frac{\exp(-u^{(i)})}{(1+\exp(-u^{(i)}))^2} \vec{x}^{(i)}$$

$$= \frac{1}{-(1+\exp(-u^{(i)}))} \cdot \vec{x}^{(i)}$$

$$\frac{\partial L(\vec{w}, b)}{\partial \vec{w}} = -\left[\sum_{i=1}^{n} t_i \frac{\partial \log \sigma(u^{(i)})}{\partial \vec{w}} + \sum_{i=1}^{n} (1-t_i) \frac{\partial \log(1-\sigma(u^{(i)}))}{\partial \vec{w}}\right]$$

$$= -\left[\sum_{i=1}^{n} \frac{t_i \exp(-u^{(i)})}{1+\exp(-u^{(i)})} \cdot \vec{x}^{(i)} + \sum_{i=1}^{n} \frac{(1-t_i)}{-(1+\exp(-u^{(i)}))} \cdot \vec{x}^{(i)}\right]$$

$$= -\left[\sum_{i=1}^{n} \left(\frac{t_i \exp(-u^{(i)}) + (t_i - 1)}{1+\exp(-u^{(i)})} \vec{x}^{(i)}\right)\right]$$

$$= -\sum_{i=1}^{n} \left(\frac{t_i(\exp(-u^{(i)})+1) - 1}{1+\exp(-u^{(i)})} \vec{x}^{(i)}\right)$$

$$= -\sum_{i=1}^{n} \left[\left(t_i - \frac{1}{1+\exp(-u^{(i)})}\right) \vec{x}^{(i)}\right]$$

$$\boxed{\frac{\partial L(\vec{w}, b)}{\partial \vec{w}} = -\sum_{i=1}^{n} \left[\left(t_i - \frac{1}{1+\exp(-w^T \vec{x}^{(i)} - b)}\right) \vec{x}^{(i)}\right]}$$

$$\frac{\partial \log \sigma(u^{(i)})}{\partial b} = \frac{1}{\sigma(u^{(i)})} \cdot \sigma'(u^{(i)}) \cdot \frac{\partial u^{(i)}}{\partial b}$$

$$= \frac{\exp(-u^{(i)})}{1+\exp(-u^{(i)})} \cdot 1 = \frac{\exp(-u^{(i)})}{1+\exp(-u^{(i)})}$$

$$\frac{\partial \log(1-\sigma(u^{(i)}))}{\partial b} = \frac{1}{1-\sigma(u^{(i)})} \cdot (-1) \cdot \sigma'(u^{(i)}) \cdot \frac{\partial u^{(i)}}{\partial b}$$

$$= -\frac{1}{(1+\exp(-u^{(i)}))}$$

Thus $\frac{\partial l(\vec{w}, b)}{\partial b} = -\left[\sum_{i=1}^{n} t_i \frac{\partial \log \sigma(u^{(i)})}{\partial b} + \sum_{i=1}^{n} (1-t_i) \frac{\partial \log(1-\sigma(u^{(i)}))}{\partial b}\right]$

$$= -\left[\sum_{i=1}^{n} \frac{t_i \exp(-u^{(i)})}{1+\exp(-u^{(i)})} + \sum_{i=1}^{n} \frac{(1-t_i)}{-(1+\exp(-u^{(i)}))}\right]$$

$$= -\left[\sum_{i=1}^{n} \left(\frac{t_i \exp(-u^{(i)}) + (t_i - 1)}{1+\exp(-u^{(i)})}\right)\right]$$

$$= -\sum_{i=1}^{n} \left(\frac{t_i(\exp(-u^{(i)})+1) - 1}{1+\exp(-u^{(i)})}\right)$$

$$\frac{\partial l(\vec{w}, b)}{\partial b} = -\sum_{i=1}^{n} \left(t_i - \frac{1}{1+\exp(-u^{(i)})}\right)$$

$$\boxed{\frac{\partial l(\vec{w}, b)}{\partial b} = -\sum_{i=1}^{n} \left(t_i - \frac{1}{1+\exp(-\vec{w}^T \vec{x}^{(i)} - b)}\right)}$$

Q1.3: $P(\vec{w}, b \mid \mathcal{D}) = P(\mathcal{D} \mid \vec{w}, b) \cdot P(\vec{w}, b) / P(\mathcal{D})$

$= P(\mathcal{D} \mid \vec{w}, b) \cdot P(\vec{w}) \cdot P(b) / P(\mathcal{D})$

$= P(\mathcal{D} \mid \vec{w}, b) \cdot P(\vec{w}) / P(\mathcal{D})$

$$P(\mathcal{D} \mid \vec{w}, b) = \prod_{i=1}^{n} \left[t_i \sigma(\vec{w}^T \vec{x}^{(i)} + b) + (1-t_i)(1 - \sigma(\vec{w}^T \vec{x}^{(i)} + b))\right]$$

$$P(\vec{w}) = \prod_{i=1}^{D} P(w_i) = \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi \frac{1}{\lambda}}} \exp\left(-(w_i - 0)^2 / \frac{2}{\lambda}\right) = \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi/\lambda}} \exp(-\lambda w_i^2/2)$$

$$= \left(\frac{1}{\sqrt{2\pi/\lambda}}\right)^D \exp\left(\sum_{i=1}^{D} (-\lambda w_i^2/2)\right)$$

$$P(\mathcal{D}) = \prod_{i=1}^{n} \left[t_i \alpha + (1-t_i)(1-\alpha)\right]$$

$$\mathcal{L}_{post}(\vec{w}, b) = -\log P(\vec{w}, b \mid \mathcal{D})$$

$$= -\log P(\mathcal{D} \mid \vec{w}, b) - \log P(\vec{w}) + \log P(\mathcal{D})$$

$$= \mathcal{L}(\vec{w}, b) - \left(\log \left(\frac{1}{\sqrt{2\pi/\lambda}}\right)^D + \log \exp\left(\sum_{i=1}^{D} (-\lambda w_i^2/2)\right)\right) + \log P(\mathcal{D})$$

$$= \mathcal{L}(\vec{w}, b) - \sum_{i=1}^{D} (-\lambda w_i^2/2) + \log P(\mathcal{D}) - \log \left(\frac{1}{\sqrt{2\pi/\lambda}}\right)^D$$

$$= \mathcal{L}(\vec{w}, b) + \frac{\lambda}{2} \sum_{i=1}^{D} w_i^2 + \sum_{i=1}^{n} \log(t_i \alpha + (1-t_i)(1-\alpha)) - D \log\left(\frac{1}{\sqrt{2\pi/\lambda}}\right)$$

$$= \mathcal{L}(\vec{w}, b) + \frac{\lambda}{2} \sum_{i=1}^{D} w_i^2 + \sum_{i=1}^{n} \left[t_i \log \alpha + (1-t_i) \log(1-\alpha)\right] - D \log\left(\frac{1}{\sqrt{2\pi/\lambda}}\right)$$

$$\mathcal{L}_{post}(\vec{w}, b) = \mathcal{L}(\vec{w}, b) + \frac{\lambda}{2} \sum_{i=1}^{D} w_i^2 + \underbrace{\log \alpha \cdot \sum_{i=1}^{n} t_i + \log(1-\alpha) \cdot \sum_{i=1}^{n} (1-t_i) - D \log\left(\frac{1}{\sqrt{2\pi/\lambda}}\right)}_{C}$$

$$\frac{\partial \mathcal{L}_{post}(\vec{w}, b)}{\partial \vec{w}} = \frac{\partial \mathcal{L}(\vec{w}, b)}{\partial \vec{w}} + \frac{\lambda}{2} \frac{\partial \sum_{i=1}^{D} w_i^2}{\partial \vec{w}} + 0$$

$$= -\sum_{i=1}^{n} \left[\left(t_i - \frac{1}{1+\exp(-w^T \vec{x}^{(i)} - b)} \right) \vec{x}^{(i)} \right] + \frac{\lambda}{2} \cdot 2\vec{w}$$

$$\boxed{\frac{\partial \mathcal{L}_{post}(\vec{w}, b)}{\partial \vec{w}} = -\sum_{i=1}^{n} \left[\left(t_i - \frac{1}{1+\exp(-w^T \vec{x}^{(i)} - b)} \right) \vec{x}^{(i)} \right] + \lambda \vec{w}}$$

$$\boxed{\frac{\partial \mathcal{L}_{post}(\vec{w}, b)}{\partial b} = \frac{\partial \mathcal{L}(\vec{w}, b)}{\partial b} = -\sum_{i=1}^{n} \left(t_i - \frac{1}{1+\exp(-w^T \vec{x}^{(i)} - b)} \right)}$$

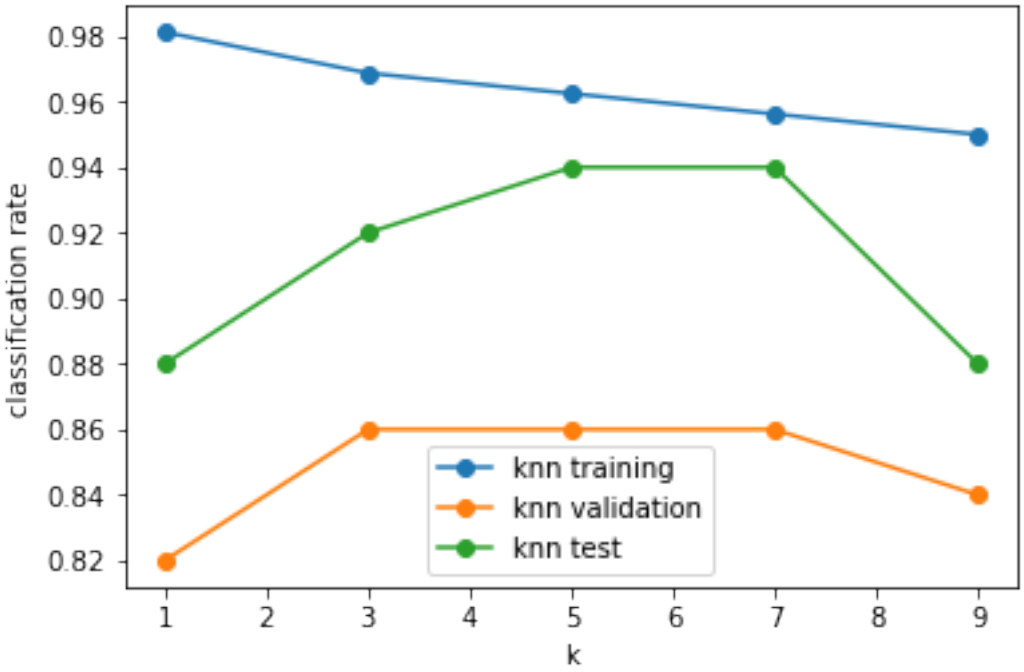
Q2.1

October 17, 2019

0.1 Q2.1

[2]:
```
import matplotlib.pyplot as plt
from run_knn import run_knn
from utils import *

def run_knn_classifcation():
    train_inputs, train_targets = load_train()
    #train_inputs, train_targets = load_train_small()
    valid_inputs, valid_targets = load_valid()

    test_inputs, test_targets = load_test()

    # Initialize the array of k's
    k_arr = [1, 3, 5, 7, 9]
    performance_train = []
    performance = []
    performance_test = []

    # Predict labels of validation set using each k as hyperparameter
    for k in k_arr:
        predicted_labels = run_knn(k, train_inputs, train_targets, train_inputs)
        # Evaluate the performance on the trainning set
        performance_train.append(evaluate(train_targets, predicted_labels))

        predicted_labels = run_knn(k, train_inputs, train_targets, valid_inputs)
        # Evaluate the performance on the validation set
        performance.append(evaluate(valid_targets, predicted_labels))

        # Cheating now...
        predicted_labels = run_knn(k, train_inputs, train_targets, test_inputs)
        # Evaluate the performance on the test set
        performance_test.append(evaluate(test_targets, predicted_labels))

    # Plot the classification rates of training data with respect to the␣
↪hyperparameters.
    plt.plot(k_arr, performance_train, "o-", label='knn training')
```

```
    # Plot the classification rates of validation set with respect to the␣
 ↪hyperparameters.
    plt.plot(k_arr, performance, "o-", label='knn validation')
    # Plot the classification rates of test data with respect to the␣
 ↪hyperparameters.
    plt.plot(k_arr, performance_test, "o-", label='knn test')

    plt.xlabel('k')
    plt.ylabel('classification rate')
    plt.legend()
    plt.show()

def evaluate(labels, predicted_labels):
    """
    Compute evaluation metric.
    Inputs:
        labels           : N x 1 vector of targets.
        predicted_labels : N x 1 vector of predicted labels.
    Outputs:
        frac_correct : (scalar) Fraction of inputs classified correctly.
    """
    correct_pred = labels == predicted_labels

    # Calculate the number of correct predictions
    correct_num = np.sum(correct_pred)

    # Calculate the percentage of correct predictions
    frac_correct = correct_num / float(correct_pred.shape[0])
    return frac_correct

%matplotlib inline
run_knn_classifcation()
```



0.1.1 Comment

The best k I will choose is k = 5, since when k = 5, the classification rate of the validation set is at the maximum. Although at k = 3 and k = 7, the classification rates are at the maximum as well, but the classification rate drops immediately after k = 7 and before k = 3.

The classification rate on the test set is also at the maximum when k = 5, and the classification rate on the test set is higher than the validation set. This is because test set has more generality than the validation set, and validation set only describes parts of the data and thus has more bias

[ ]:

Q2.2

October 17, 2019

0.1 Q2.2

[36]:
```
import numpy as np
from check_grad import check_grad
from utils import *
from logistic import *
import matplotlib.pyplot as plt
```

[3]:
```
def plot_losses(train_losses, valid_losses):
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 2, 1.5])

    ax.set_title("error curve")

    x = range(1, hyperparameters['num_iterations'] + 1)
    # Plot the train ce.
    ax.plot(x, train_losses, label='train loss')
    # Plot the validation set ce.
    ax.plot(x, valid_losses, label='validation loss')

    plt.xlabel('# iteration')
    plt.ylabel('ce')
    plt.legend()
```

[4]:
```
def run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, hyperparameters):

    N, M = train_inputs.shape
    # Logistic regression weights
    # TODO:Initialize to random weights here.
    weights = np.random.rand(M + 1, 1)

    # Store all cross entropy losses of each iteration.
    valid_losses = []
    train_losses = []

    # Begin learning with gradient descent
```

```
    for t in xrange(hyperparameters['num_iterations']):
        # TODO: you may need to modify this loop to create plots, etc.

        # Find the negative log likelihood and its derivatives w.r.t. the␣
↪weights.
        f, df, predictions = logistic_pen(weights, train_inputs, train_targets,␣
↪hyperparameters)

        # Evaluate the prediction.
        cross_entropy_train, frac_correct_train = evaluate(train_targets,␣
↪predictions)
        train_losses.append(cross_entropy_train)

        if np.isnan(f) or np.isinf(f):
            raise ValueError("nan/inf error")

        # update parameters
        weights = weights - hyperparameters['learning_rate'] * df / N

        # Make a prediction on the valid_inputs.
        predictions_valid = logistic_predict(weights, valid_inputs)

        # Evaluate the prediction.
        cross_entropy_valid, frac_correct_valid = evaluate(valid_targets,␣
↪predictions_valid)
        valid_losses.append(cross_entropy_valid)

        # print some stats
#         print ("ITERATION:{:4d}  TRAIN NLOGL:{:4.2f}  TRAIN CE:{:.6f} "
#                "TRAIN FRAC:{:2.2f}  VALID CE:{:.6f}  VALID FRAC:{:2.2f}").
↪format(
#                    t+1, f / N, cross_entropy_train, frac_correct_train*100,
#                    cross_entropy_valid, frac_correct_valid*100)

    return weights, train_losses, valid_losses
```

[5]:
```
def run_check_grad(hyperparameters):
    """Performs gradient check on logistic function.
    """

    # This creates small random data with 20 examples and
    # 10 dimensions and checks the gradient on that data.
    num_examples = 20
    num_dimensions = 10

    weights = np.random.randn(num_dimensions+1, 1)
    data    = np.random.randn(num_examples, num_dimensions)
```

```
    targets = np.random.rand(num_examples, 1)

    diff = check_grad(logistic,      # function to check
                      weights,
                      0.001,         # perturbation
                      data,
                      targets,
                      hyperparameters)

    print "diff =", diff
```

[6]:
```
def report_errors(weights, X_train, y_train, X_valid, y_valid, X_test, y_test):
    train = evaluate(logistic_predict(weights, X_train), y_train)
    valid = evaluate(logistic_predict(weights, X_valid), y_valid)
    test = evaluate(logistic_predict(weights, X_test), y_test)
    print("CE_train: {0}, MISSFRAC_train: {1}".format(train[0], 1 - train[1]))
    print("CE_valid: {0}, MISSFRAC_valid: {1}".format(valid[0], 1 - valid[1]))
    print("CE_test: {0}, MISSFRAC_test: {1}".format(test[0], 1 - test[1]))
```

[7]:
```
import itertools
```

0.1.1 Function for performing grid search on possible hyperparameters

[8]:
```
def grid_params(params):
    expand = []
    comb_num = 1
    template = {}
    for param in params:
        template[param] = None
        expand.append(params[param])
    combs = itertools.product(*expand)
    for comb in combs:
        i = 0
        for param in template:
            template[param] = comb[i]
            i += 1
        yield dict(template)
```

[9]:
```
valid_inputs, valid_targets = load_valid()
# Load test data set.
test_inputs, test_targets = load_train()

paramgrid = {
    'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    'weight_regularization': [0.0],
    'num_iterations': [500]
```

```
}
```

0.1.2 Check gradient of the logistic learning without regularizer

[70]:
```
# Verify that your logistic function produces the right gradient.
# diff should be very close to 0.
run_check_grad({'weight_regularizaiton': 10})
```

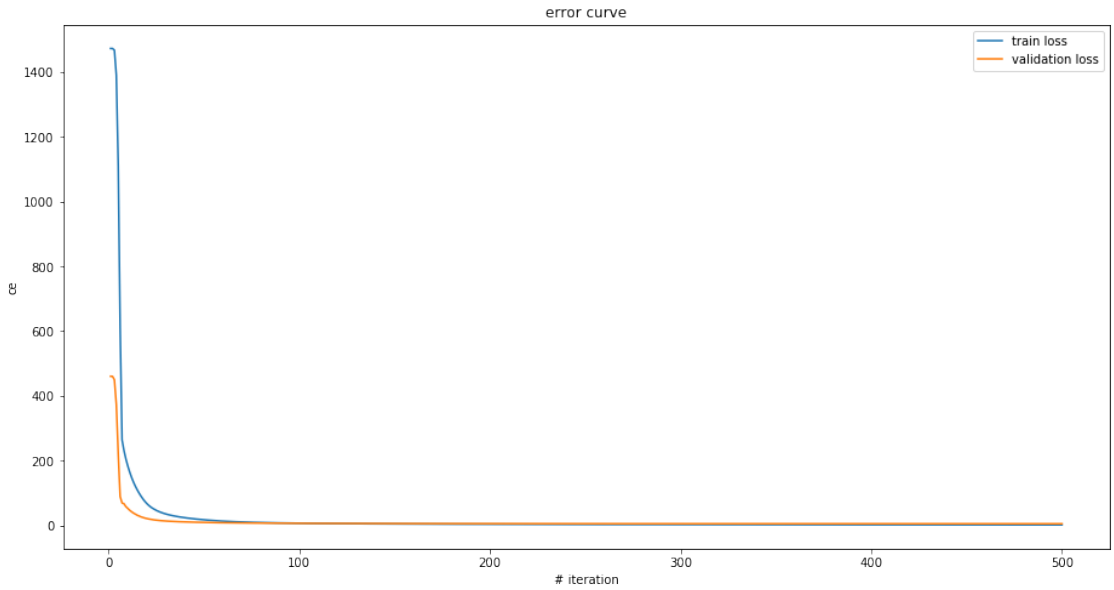
```
[[ 2.69750521  2.69750433]
 [-3.29662739 -3.2966259 ]
 [ 0.64565057  0.64565057]
 [ 3.17626395  3.17626171]
 [-0.09908095 -0.09908052]
 [-2.07892727 -2.07892598]
 [ 2.25025391  2.25025262]
 [ 1.77186854  1.77186682]
 [ 0.76289127  0.76289158]
 [ 1.06685496  1.06685319]
 [-1.94352892 -1.94352726]]
diff = 3.309969600731357e-07
```

0.1.3 Run logistic regression on the large trainning set, perform grid search to find the best hyperparameter set, and report the best hyperparameters setting.

[114]:
```
# Run logistic regression on the large trainning set.
train_inputs, train_targets = load_train()

best_result = None
for hyperparameters in grid_params(paramgrid):
    weights, train_losses, valid_losses = \
        run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, hyperparameters)
    if best_result is None or best_result[2][-1] > valid_losses[-1]:
        best_result = (weights, train_losses, valid_losses, hyperparameters)

print("Best hyperparameters setting:")
print(best_result[3])
%matplotlib inline
plot_losses(best_result[1], best_result[2])
report_errors(best_result[0], train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, test_inputs, test_targets)
```

```
Best hyperparameters setting:
{'num_iterations': 500, 'learning_rate': 0.4, 'weight_regularization': 0.0}
CE_train: 25.0179785967, MISSFRAC_train: 0.0
```

```
CE_valid: 69.1897168959, MISSFRAC_valid: 0.04
CE_test: 25.0179785967, MISSFRAC_test: 0.0
```

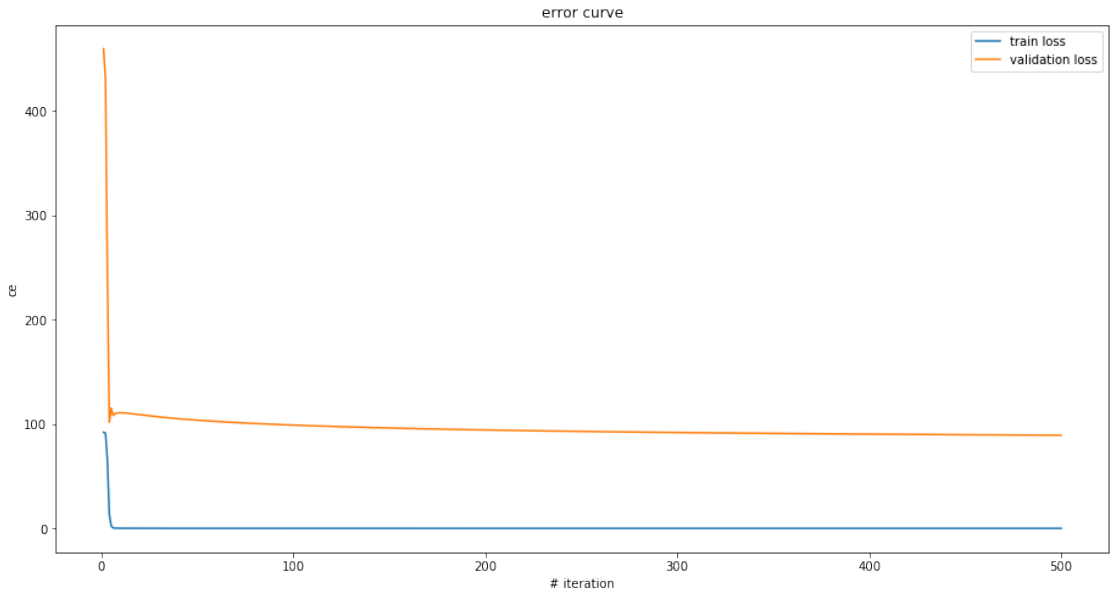


0.1.4 Run logistic regression on the small trainning set, perform grid search to find the best hyperparameter set, and report the best hyperparameters setting.

[115]:
```
# Run logistic regression on the large trainning set.
train_inputs, train_targets = load_train_small()

best_result = None
for hyperparameters in grid_params(paramgrid):
    weights, train_losses, valid_losses = \
        run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, hyperparameters)
    if best_result is None or best_result[2][-1] > valid_losses[-1]:
        best_result = (weights, train_losses, valid_losses, hyperparameters)

print("Best hyperparameters setting:")
print(best_result[3])
%matplotlib inline
plot_losses(best_result[1], best_result[2])
report_errors(best_result[0], train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, test_inputs, test_targets)
```

```
Best hyperparameters setting:
{'num_iterations': 500, 'learning_rate': 0.8, 'weight_regularization': 0.0}
CE_train: 0.066998124944, MISSFRAC_train: 0.0
```

```
CE_valid: 326.732616018, MISSFRAC_valid: 0.38
CE_test: 931.614366121, MISSFRAC_test: 0.3
```



0.1.5 Comment on 2.2

In this experiment, I first perform gird search to compare the perfomance of every possible combinations in the given parameter space. The hyperparameters that give the smallest validation error after trainning is selected as the best configuration.

However, I notice that the best configuration is always changing when repeating the experiment. To select the best hyperparameter setting, I should run each configuration multiple times and calculate the average error. Then choose the configuration which yields the lowest average error. This is what I did in Q2.3.

0.2 Q2.3

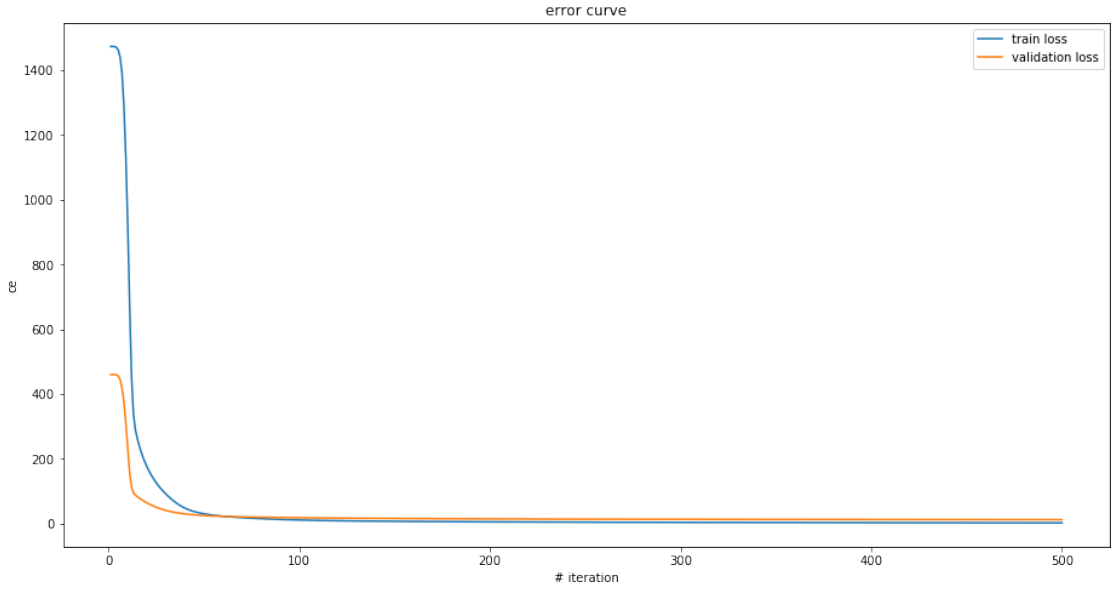
0.2.1 Setup all possible lambda values

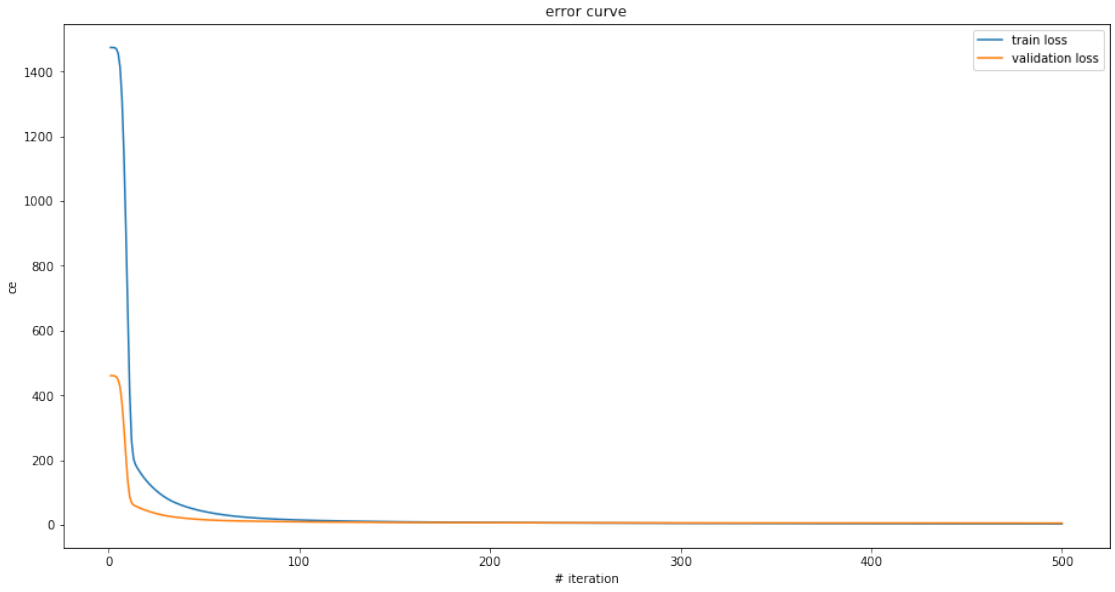
[37]:
```
hyperparameters = {
    'learning_rate': 0.2,
    'weight_regularization': 0,
    'num_iterations': 500
}
lambdas = [0, 0.001, 0.01, 0.1, 1.0]
```

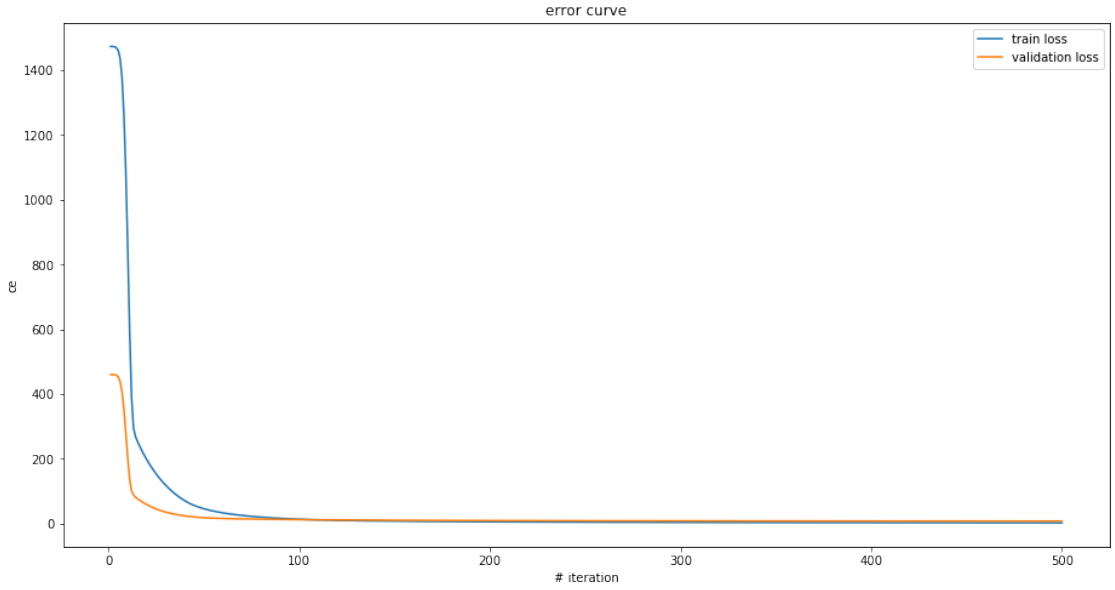
0.2.2 Run penalized logistic regression on large dataset

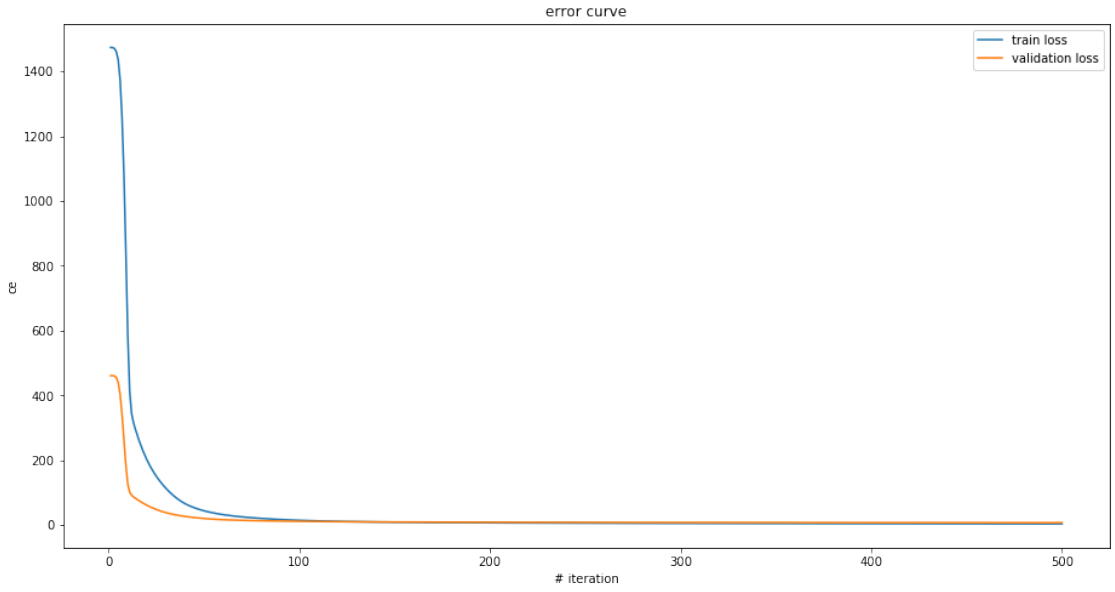
[74]:
```
train_inputs, train_targets = load_train()
```

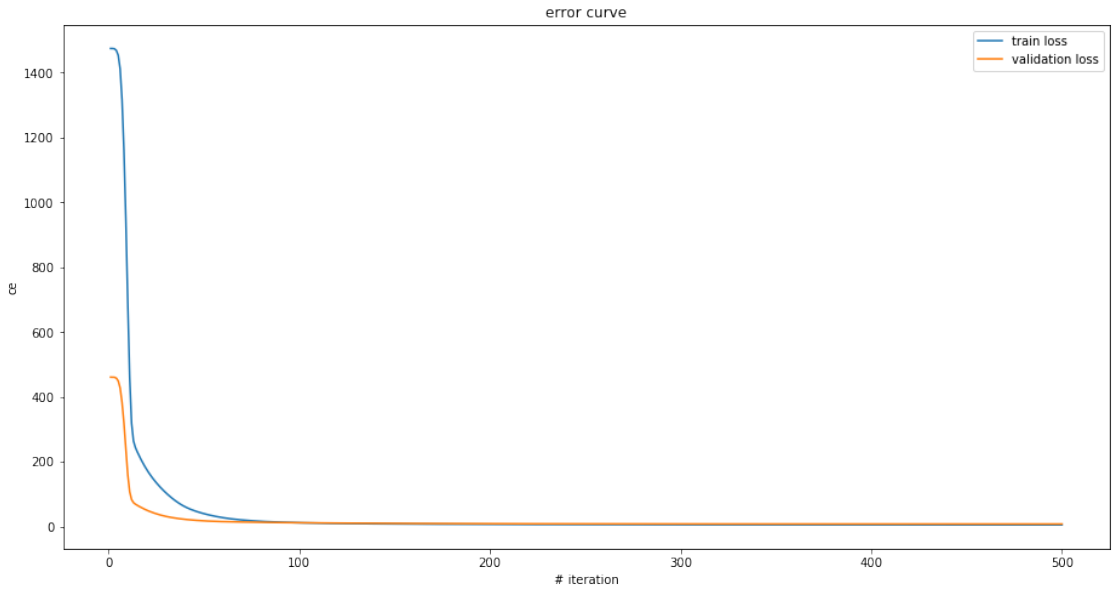
[78]:
```
%matplotlib inline
train_ce_avgs = []
valid_ce_avgs = []
train_fe_avgs = []
valid_fe_avgs = []
weightss = []
for lambd in lambdas:
    hyperparameters['weight_regularization'] = lambd
    train_ce_sum, train_fe_sum = 0, 0
    valid_ce_sum, valid_fe_sum = 0, 0
    weights_arr = []
    for i in range(5):
        weights, train_losses, valid_losses = \
            run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
↪valid_targets, hyperparameters)
        weights_arr.append(weights)
        # Predict with the weights
        y_train = logistic_predict(weights, train_inputs)
        y_valid = logistic_predict(weights, valid_inputs)
        # Compute the cross entrophy error and fraction error
        train_ce, train_fe = evaluate(train_targets, y_train)
        valid_ce, valid_fe = evaluate(valid_targets, y_valid)
        # Compute misclassification rate
        train_fe, valid_fe = 1.0 - train_fe, 1.0 - valid_fe
        # Compute the sum of errors in 5 turns
        train_ce_sum, train_fe_sum = train_ce_sum + train_ce, train_fe_sum +␣
↪train_fe
        valid_ce_sum, valid_fe_sum = valid_ce_sum + valid_ce, valid_fe_sum +␣
↪valid_fe
        # Always print the plot of the last run of each lambda
        if i == 4:
            plot_losses(train_losses, valid_losses)
    # Append the avgs
    train_ce_avgs.append(train_ce_sum / 5.0)
    valid_ce_avgs.append(valid_ce_sum / 5.0)
    train_fe_avgs.append(train_fe_sum / 5.0)
    valid_fe_avgs.append(valid_fe_sum / 5.0)
    weightss.append(weights_arr)
```


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

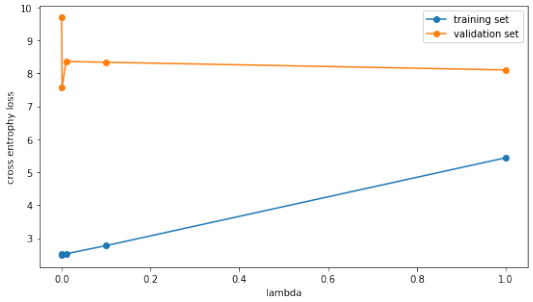

error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

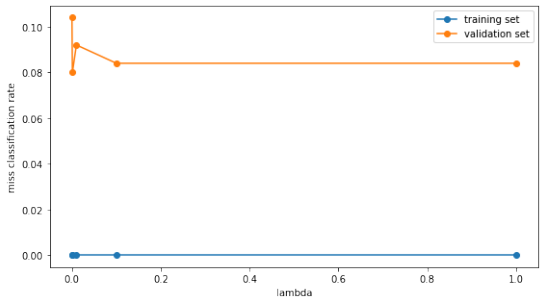


0.2.3 Plot the graphs

[79]:
```
plt.figure(figsize=(20, 5))
plt.subplot(121)
plt.plot(X, train_ce_avgs, '-o', label="training set")
plt.plot(X, valid_ce_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("cross entrophy loss")
plt.legend()

plt.subplot(122)
plt.plot(X, train_fe_avgs, '-o', label="training set")
plt.plot(X, valid_fe_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("miss classification rate")
plt.legend()
```

[79]:
```
<matplotlib.legend.Legend at 0x7fb20af66990>
```





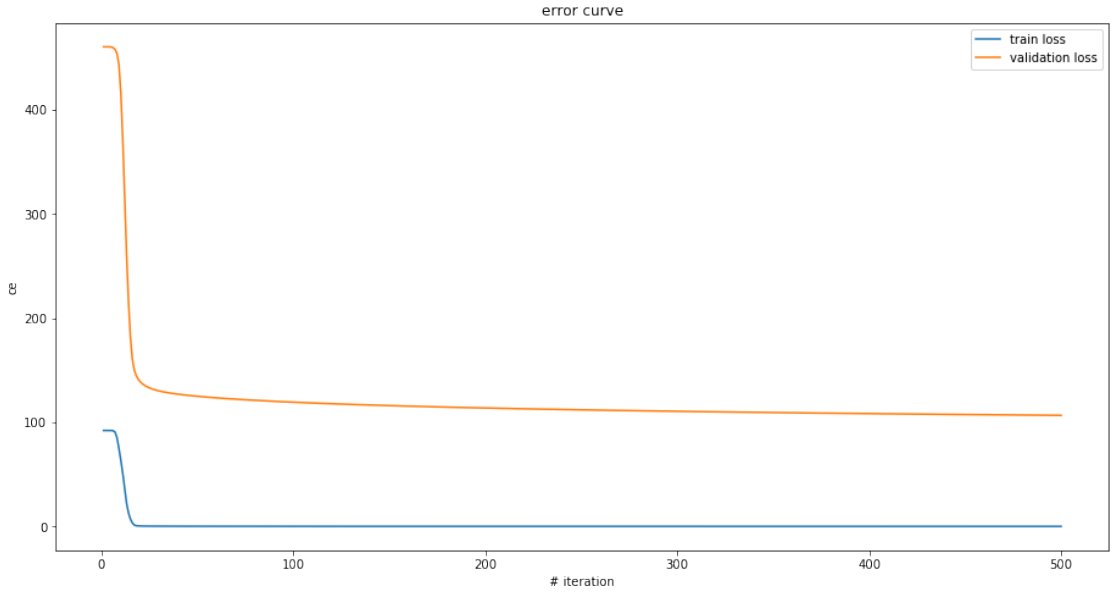
[80]:
```
# Report the best hyperparemeter setting
argmin = np.argmin(valid_ce_avgs)
print("Best lambda setting:")
print(lambdas[argmin])
for i in range(5):
    print("Iteration {0}: ".format(i + 1))
    report_errors(weightss[argmin][i], train_inputs, train_targets,␣
 ↪valid_inputs, valid_targets, test_inputs, test_targets)
```

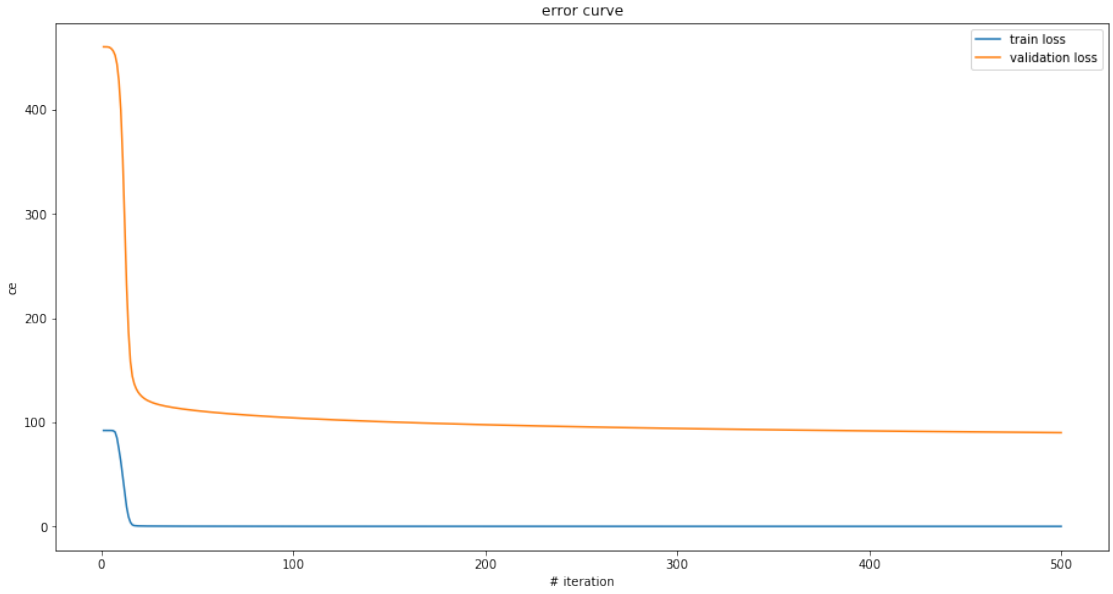
```
Best lambda setting:
0.001
Iteration 1:
CE_train: 44.7075404222, MISSFRAC_train: 0.0
CE_valid: 103.467789306, MISSFRAC_valid: 0.1
CE_test: 44.7075404222, MISSFRAC_test: 0.0
Iteration 2:
CE_train: 45.9161922943, MISSFRAC_train: 0.0
CE_valid: 81.6256268254, MISSFRAC_valid: 0.08
CE_test: 45.9161922943, MISSFRAC_test: 0.0
Iteration 3:
CE_train: 41.0620973085, MISSFRAC_train: 0.0
CE_valid: 96.2279269937, MISSFRAC_valid: 0.12
CE_test: 41.0620973085, MISSFRAC_test: 0.0
Iteration 4:
CE_train: 42.5784271343, MISSFRAC_train: 0.0
CE_valid: 84.9683294723, MISSFRAC_valid: 0.06
CE_test: 42.5784271343, MISSFRAC_test: 0.0
Iteration 5:
CE_train: 49.8034126724, MISSFRAC_train: 0.0
CE_valid: 74.6499798367, MISSFRAC_valid: 0.04
CE_test: 49.8034126724, MISSFRAC_test: 0.0
```

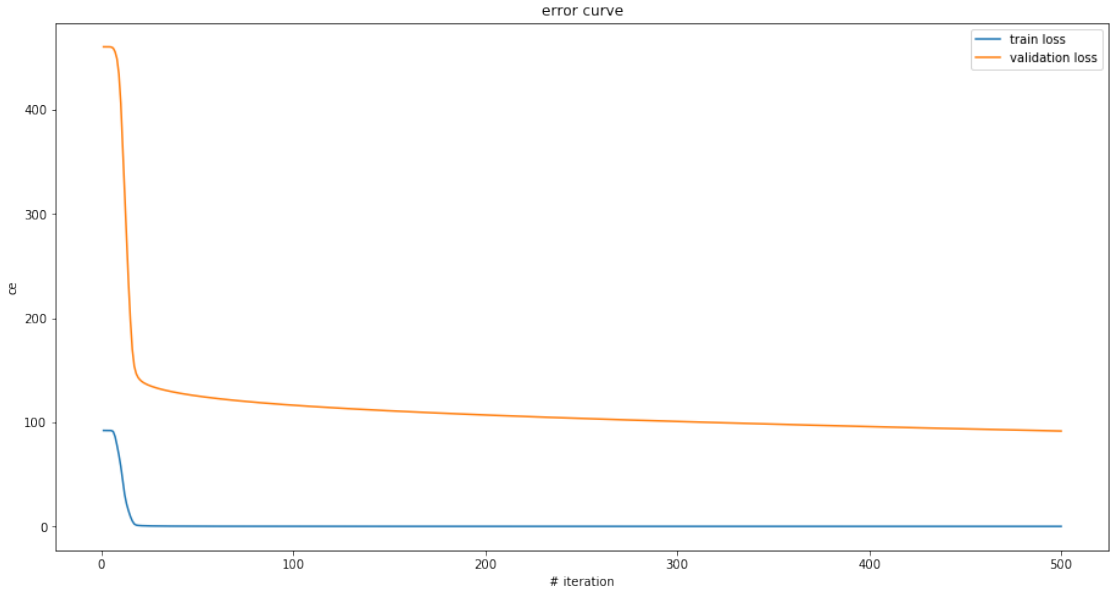
0.2.4 Run penalized logistic regression on small dataset

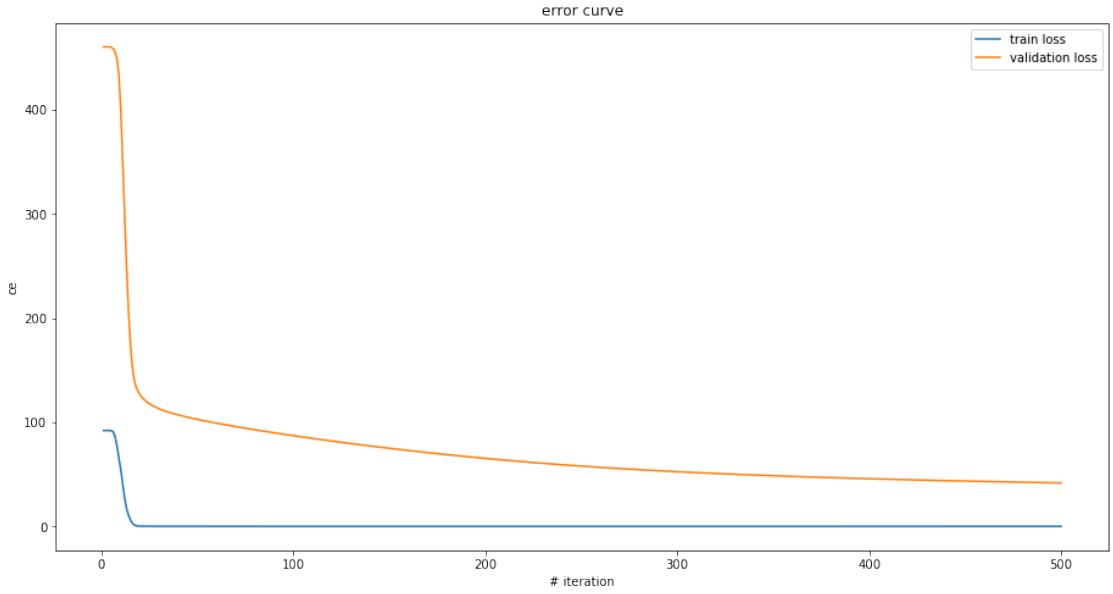
[88]:
```
train_inputs, train_targets = load_train_small()
```

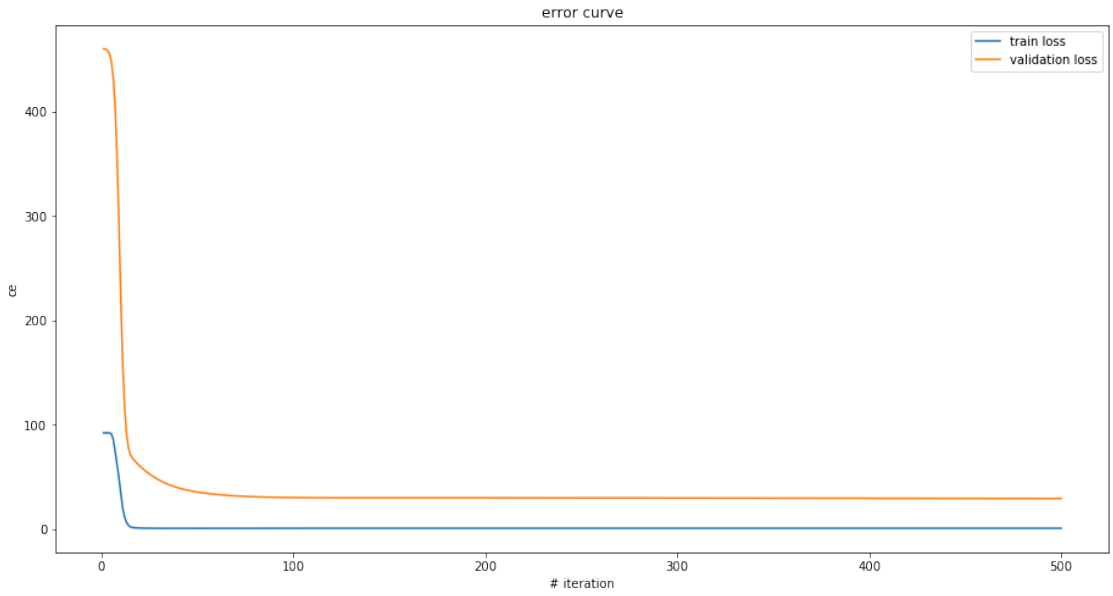
[89]:
```
%matplotlib inline
train_ce_avgs = []
valid_ce_avgs = []
train_fe_avgs = []
valid_fe_avgs = []
lambdas = [0, 0.001, 0.01, 0.1, 1.0]
weightss = []
for lambd in lambdas:
    hyperparameters['weight_regularization'] = lambd
    train_ce_sum, train_fe_sum = 0, 0
    valid_ce_sum, valid_fe_sum = 0, 0
    for i in range(5):
        weights, train_losses, valid_losses = \
            run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
↪valid_targets, hyperparameters)
        # Predict with the weights
        y_train = logistic_predict(weights, train_inputs)
        y_valid = logistic_predict(weights, valid_inputs)
        # Compute the cross entrophy error and fraction error
        train_ce, train_fe = evaluate(train_targets, y_train)
        valid_ce, valid_fe = evaluate(valid_targets, y_valid)
        # Compute misclassification rate
        train_fe, valid_fe = 1.0 - train_fe, 1.0 - valid_fe
        # Compute the sum of errors in 5 turns
        train_ce_sum, train_fe_sum = train_ce_sum + train_ce, train_fe_sum +␣
↪train_fe
        valid_ce_sum, valid_fe_sum = valid_ce_sum + valid_ce, valid_fe_sum +␣
↪valid_fe
        if i == 4:
            plot_losses(train_losses, valid_losses)
    # Append the avgs
    train_ce_avgs.append(train_ce_sum / 5.0)
    valid_ce_avgs.append(valid_ce_sum / 5.0)
    train_fe_avgs.append(train_fe_sum / 5.0)
    valid_fe_avgs.append(valid_fe_sum / 5.0)
    weightss.append(weights)
```


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500

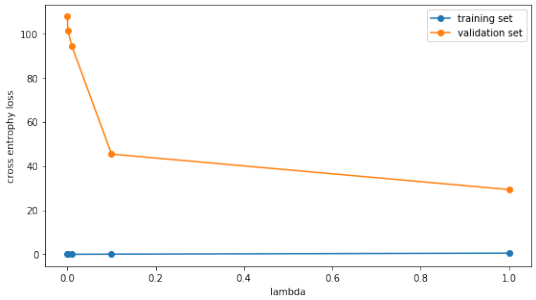

error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500

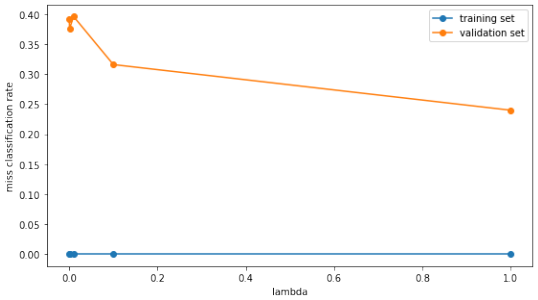


0.2.5 Plot the graphs

[90]:
```
plt.figure(figsize=(20, 5))
plt.subplot(121)
plt.plot(lambdas, train_ce_avgs, '-o', label="training set")
plt.plot(lambdas, valid_ce_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("cross entrophy loss")
plt.legend()

plt.subplot(122)
plt.plot(lambdas, train_fe_avgs, '-o', label="training set")
plt.plot(lambdas, valid_fe_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("miss classification rate")
plt.legend()
```

[90]:
```
<matplotlib.legend.Legend at 0x7fb20b0def10>
```





[91]:
```
# Report the best hyperparemeter setting
argmin = np.argmin(valid_ce_avgs)
print("Best lambda setting:")
print(lambdas[argmin])
for i in range(5):
    print("Iteration {0}: ".format(i + 1))
    report_errors(weightss[i], train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, test_inputs, test_targets)
```

```
Best lambda setting:
1.0
Iteration 1:
CE_train: 0.238065732443, MISSFRAC_train: 0.0
CE_valid: 351.811191892, MISSFRAC_valid: 0.42
CE_test: 1053.15105128, MISSFRAC_test: 0.3625
Iteration 2:
CE_train: 0.277737797047, MISSFRAC_train: 0.0
CE_valid: 378.571513844, MISSFRAC_valid: 0.38
CE_test: 1125.71133289, MISSFRAC_test: 0.3875
Iteration 3:
CE_train: 0.300291217389, MISSFRAC_train: 0.0
CE_valid: 376.784600517, MISSFRAC_valid: 0.44
CE_test: 968.124116102, MISSFRAC_test: 0.325
Iteration 4:
CE_train: 1.65767820029, MISSFRAC_train: 0.0
CE_valid: 317.607770185, MISSFRAC_valid: 0.32
CE_test: 891.038081042, MISSFRAC_test: 0.29375
Iteration 5:
CE_train: 9.74684898525, MISSFRAC_train: 0.0
CE_valid: 342.076474772, MISSFRAC_valid: 0.24
CE_test: 1001.9836519, MISSFRAC_test: 0.25625
```

0.2.6 Comment:

The cross entropy of the validation set is fluctuating in a small range when training on the large dataset, but keep reducing when training on the small dataset as lambda increases.

The reason for this is, large datasize has more sample. Therefore the trainning model is more generalized and less likely to overfit. However, when the datasize is small, it is more likely to overfit when training. Therefore, the regularizer penalized the score when the model is overfitting. That's why the change on lambda has more impact on the small dataset than the large dataset.

In conclusion, for large dataset, no penalization is better. For small dataset, lambda = 1 performs better.

[ ]:

Q2.2 logistic.py

October 17, 2019

0.1 Q2.2 logistic.py

[1]:
```
""" Methods for doing logistic regression."""

import numpy as np
from utils import sigmoid

def logistic_predict(weights, data):
    """
    Compute the probabilities predicted by the logistic classifier.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to the bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
    Outputs:
        y:          :N x 1 vector of probabilities. This is the output of the␣
 ↪classifier.
    """
    # TODO: Finish this function
    N = data.shape[0]
    # append vector contains only 1s
    data = np.concatenate((data, np.full((N, 1), 1)), axis=1)

    # predict the output given weights
    y = sigmoid(np.matmul(data, weights))

    return y

def evaluate(targets, y):
    """
    Compute evaluation metrics.
    Inputs:
```

```
        targets : N x 1 vector of targets.
        y       : N x 1 vector of probabilities.
    Outputs:
        ce          : (scalar) Cross entropy. CE(p, q) = E_p[-log q]. Here we want to compute CE(targets, y)
        frac_correct : (scalar) Fraction of inputs classified correctly.
    """
    # TODO: Finish this function
    # Squeeze 2d column or row vector into 1d array
    targets = np.squeeze(targets)
    y = np.squeeze(y)
    # Calculate cross entropy loss
    ce = -(np.dot(targets, np.log(y + 1e-8)) + np.dot(1 - targets, np.log(1 - y + 1e-8)))

    # Calculate the percentage of correct prediction with 0.5 as threshold
    correct_num = np.sum(np.absolute(y - targets) <= .5)
    frac_correct = correct_num / float(targets.shape[0])

    return ce, frac_correct

def logistic(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
        targets:    N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:       The sum of the loss over all data points. This is the objective that we want to minimize.
        df:      (M+1) x 1 vector of derivative of f w.r.t. weights.
        y:       N x 1 vector of probabilities.
    """
    # TODO: Finish this function
    no_penalize = hyperparameters.copy()
    no_penalize["weight_regularization"] = 0
```

```
    return logistic_pen(weights, data, targets, no_penalize)


def logistic_pen(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to␣
 ↪weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
        targets:    N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:              The sum of the loss over all data points. This is the␣
 ↪objective that we want to minimize.
        df:             (M+1) x 1 vector of derivative of f w.r.t. weights.
    """

    # TODO: Finish this function
    y = logistic_predict(weights, data)

    # Calculate the difference between the prediction and true value
    diff = targets - y

    # Compute gradient of dL/dw, and change to column vector
    df = (np.matmul(diff.T, data)).T

    # Add derivative of bias to the last column
    df = np.append(df, [[np.sum(diff)]], axis=0)
    df = -df

    # Derived the regulizer and add to the derivative
    df[:-1] += hyperparameters["weight_regularization"] * weights[:-1] * target.
 ↪shape[0]
    # Compute cross entropy loss
    f, _ = evaluate(targets, y)

    # Include the regulizer
```

```
    f += (hyperparameters["weight_regularization"] / 2) * np.sum(weights**2) *␣
↪target.shape[0]
    return f, df, y
```

[ ]:

Question3

October 17, 2019

0.1 Question 3

[9]:
```
"""
Instruction:

In this section, you are asked to train a NN with different hyperparameters.
To start with training, you need to fill in the incomplete code. There are 3
places that you need to complete:
a) Backward pass equations for an affine layer (linear transformation + bias).
b) Backward pass equations for ReLU activation function.
c) Weight update equations with momentum.

After correctly fill in the code, modify the hyperparameters in "main()".
You can then run this file with the command: "python nn.py" in your terminal.
The program will automatically check your gradient implementation before start.
The program will print out the training progress, and it will display the
training curve by the end. You can optionally save the model by uncommenting
the lines in "main()".
"""

from __future__ import division
from __future__ import print_function

from util import LoadData, Load, Save, DisplayPlot
import sys
import numpy as np
```

[10]:
```
def InitNN(num_inputs, num_hiddens, num_outputs):
    """Initializes NN parameters.

    Args:
        num_inputs:    Number of input units.
        num_hiddens:   List of two elements, hidden size for each layer.
        num_outputs:   Number of output units.

    Returns:
        model:         Randomly initialized network weights.
```

```
    """
    W1 = 0.1 * np.random.randn(num_inputs, num_hiddens[0])
    W2 = 0.1 * np.random.randn(num_hiddens[0], num_hiddens[1])
    W3 = 0.01 * np.random.randn(num_hiddens[1], num_outputs)
    b1 = np.zeros((num_hiddens[0]))
    b2 = np.zeros((num_hiddens[1]))
    b3 = np.zeros((num_outputs))
    model = {
        'V_w1': np.zeros((num_inputs, num_hiddens[0])),
        'V_w2': np.zeros(num_hiddens),
        'V_w3': np.zeros((num_hiddens[1], num_outputs)),
        'W1': W1,
        'W2': W2,
        'W3': W3,
        'b1': b1,
        'b2': b2,
        'b3': b3
    }
    return model
```

[11]:
```
def Affine(x, w, b):
    """Computes the affine transformation.

    Args:
        x: Inputs (or hidden layers)
        w: Weights
        b: Bias

    Returns:
        y: Outputs
    """
    # y = np.dot(w.T, x) + b
    y = x.dot(w) + b
    return y
```

[12]:
```
def AffineBackward(grad_y, h, w):
    """Computes gradients of affine transformation.
    hint: you may need the matrix transpose np.dot(A,B).T = np.dot(B,A) and (A.
 ↪T).T = A

    Args:
        grad_y: gradient from last layer
        h: inputs from the hidden layer
        w: weights from last layer

    Returns:
        grad_h: Gradients wrt. the inputs/hidden layer.
```

```
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    # Insert your code here.
    grad_h = np.dot(grad_y, w.T)
    grad_w = np.dot(h.T, grad_y)
    grad_b = np.sum(grad_y, axis=0)
    return grad_h, grad_w, grad_b
```

[13]:
```
def ReLU(z):
    """Computes the ReLU activation function.

    Args:
        z: Inputs

    Returns:
        h: Activation of z
    """
    return np.maximum(z, 0.0)
```

[14]:
```
def ReLUBackward(grad_h, z):
    """Computes gradients of the ReLU activation function wrt. the unactivated␣
 ↪inputs.

    Args:
        z: Inputs

    Returns:
        grad_z: Gradients wrt. the hidden state prior to activation.
    """
    grad_z = grad_h * (z > 0)
    return grad_z
```

[90]:
```
def Softmax(x):
    """Computes the softmax activation function.

    Args:
        x: Inputs

    Returns:
        y: Activation
    """
    return np.exp(x) / np.exp(x).sum(axis=1, keepdims=True)
```

[16]:
```
def NNForward(model, x):
    """Runs the forward pass.
var = {
```

```
    Args:
        model: Dictionary of all the weights.
        x:     Input to the network.

    Returns:
        var:   Dictionary of all intermediate variables.
    """
    z1 = Affine(x, model['W1'], model['b1'])
    h1 = ReLU(z1)
    z2 = Affine(h1, model['W2'], model['b2'])
    h2 = ReLU(z2)
    y = Affine(h2, model['W3'], model['b3'])
    var = {
        'x': x,
        'z1': z1,
        'h1': h1,
        'z2': z2,
        'h2': h2,
        'y': y
    }
    return var
```

[17]:
```
def NNBackward(model, err, var):
    """Runs the backward pass.

    Args:
        model:    Dictionary of all the weights.
        err:      Gradients to the output of the network.
        var:      Intermediate variables from the forward pass.
    """
    dE_dh2, dE_dW3, dE_db3 = AffineBackward(err, var['h2'], model['W3'])
    dE_dz2 = ReLUBackward(dE_dh2, var['z2'])
    dE_dh1, dE_dW2, dE_db2 = AffineBackward(dE_dz2, var['h1'], model['W2'])
    dE_dz1 = ReLUBackward(dE_dh1, var['z1'])
    _, dE_dW1, dE_db1 = AffineBackward(dE_dz1, var['x'], model['W1'])
    model['dE_dW1'] = dE_dW1
    model['dE_dW2'] = dE_dW2
    model['dE_dW3'] = dE_dW3
    model['dE_db1'] = dE_db1
    model['dE_db2'] = dE_db2
    model['dE_db3'] = dE_db3
    pass
```

[18]:
```
def NNUpdate(model, eps, momentum):
    """Update NN weights.

    Args:
```

```
        model:    Dictionary of all the weights.
        eps:      Learning rate.
        momentum: Momentum.
    """
    # Insert your code here.
    # Update velocities
    model['V_w1'] = momentum * model['V_w1'] + (1 - momentum) * model['dE_dW1']
    model['V_w2'] = momentum * model['V_w2'] + (1 - momentum) * model['dE_dW2']
    model['V_w3'] = momentum * model['V_w3'] + (1 - momentum) * model['dE_dW3']
    # Update the weights and biases.
    model['W1'] -= eps * model['V_w1']
    model['W2'] -= eps * model['V_w2']
    model['W3'] -= eps * model['V_w3']
    model['b1'] -= eps * model['dE_db1']
    model['b2'] -= eps * model['dE_db2']
    model['b3'] -= eps * model['dE_db3']
```

[53]:
```
def Train(model, forward, backward, update, hypers, verbose=True, diagram=True):
    %matplotlib tk
    """Trains a simple MLP.

    Args:
        model:           Dictionary of model weights.
        forward:         Forward prop function.
        backward:        Backward prop function.
        update:          Update weights function.
        eps:             Learning rate.
        momentum:        Momentum.
        num_epochs:      Number of epochs to run training for.
        batch_size:      Mini-batch size, -1 for full batch.

    Returns:
        stats:           Dictionary of training statistics.
            - train_ce:       Training cross entropy.
            - valid_ce:       Validation cross entropy.
            - train_acc:      Training accuracy.
            - valid_acc:      Validation accuracy.
    """
    eps, momentum, num_epochs, batch_size = \
        hypers["eps"], hypers["momentum"], hypers["num_epochs"],␣
↪hypers["batch_size"]
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
        target_test = LoadData('./toronto_face.npz')
    rnd_idx = np.arange(inputs_train.shape[0])
    train_ce_list = []
    valid_ce_list = []
    train_acc_list = []
```

```
    valid_acc_list = []
    num_train_cases = inputs_train.shape[0]
    if batch_size == -1:
        batch_size = num_train_cases
    num_steps = int(np.ceil(num_train_cases / batch_size))
    for epoch in range(num_epochs):
        np.random.shuffle(rnd_idx)
        inputs_train = inputs_train[rnd_idx]
        target_train = target_train[rnd_idx]
        for step in range(num_steps):
            # Forward prop.
            start = step * batch_size
            end = min(num_train_cases, (step + 1) * batch_size)
            x = inputs_train[start: end]
            t = target_train[start: end]

            var = forward(model, x)
            prediction = Softmax(var['y'])

            train_ce = -np.sum(t * np.log(prediction)) / x.shape[0]
            train_acc = (np.argmax(prediction, axis=1) ==
                         np.argmax(t, axis=1)).astype('float').mean()
            if verbose:
                print(('Epoch {:3d} Step {:2d} Train CE {:.5f} '
                       'Train Acc {:.5f}').format(
                    epoch, step, train_ce, train_acc))

            # Compute error.
            error = (prediction - t) / x.shape[0]

            # Backward prop.
            backward(model, error, var)

            # Update weights.
            update(model, eps, momentum)

        valid_ce, valid_acc = Evaluate(
            inputs_valid, target_valid, model, forward, batch_size=batch_size)
        if verbose:
            print(('Epoch {:3d} '
                   'Validation CE {:.5f} '
                   'Validation Acc {:.5f}\n').format(
                epoch, valid_ce, valid_acc))
        train_ce_list.append((epoch, train_ce))
        train_acc_list.append((epoch, train_acc))
        valid_ce_list.append((epoch, valid_ce))
        valid_acc_list.append((epoch, valid_acc))
```

```
        if diagram:
            DisplayPlot(train_ce_list, valid_ce_list, 'Cross Entropy', number=0)
            DisplayPlot(train_acc_list, valid_acc_list, 'Accuracy', number=1)

    if verbose:
        print()
    train_ce, train_acc = Evaluate(
        inputs_train, target_train, model, forward, batch_size=batch_size)
    valid_ce, valid_acc = Evaluate(
        inputs_valid, target_valid, model, forward, batch_size=batch_size)
    test_ce, test_acc = Evaluate(
        inputs_test, target_test, model, forward, batch_size=batch_size)
    print('CE: Train %.5f Validation %.5f Test %.5f' %
          (train_ce, valid_ce, test_ce))
    print('Acc: Train {:.5f} Validation {:.5f} Test {:.5f}'.format(
        train_acc, valid_acc, test_acc))

    stats = {
        'train_ce': train_ce_list,
        'valid_ce': valid_ce_list,
        'train_acc': train_acc_list,
        'valid_acc': valid_acc_list
    }

    return model, stats
```

[20]:
```
def Evaluate(inputs, target, model, forward, batch_size=-1):
    """Evaluates the model on inputs and target.

    Args:
        inputs: Inputs to the network.
        target: Target of the inputs.
        model:  Dictionary of network weights.
    """
    num_cases = inputs.shape[0]
    if batch_size == -1:
        batch_size = num_cases
    num_steps = int(np.ceil(num_cases / batch_size))
    ce = 0.0
    acc = 0.0
    for step in range(num_steps):
        start = step * batch_size
        end = min(num_cases, (step + 1) * batch_size)
        x = inputs[start: end]
        t = target[start: end]
        prediction = Softmax(forward(model, x)['y'])
        ce += -np.sum(t * np.log(prediction))
```

```
        acc += (np.argmax(prediction, axis=1) == np.argmax(
            t, axis=1)).astype('float').sum()
    ce /= num_cases
    acc /= num_cases
    return ce, acc
```

[21]:
```
def CheckGrad(model, forward, backward, name, x):
    """Check the gradients

    Args:
        model: Dictionary of network weights.
        name: Weights name to check.
        x: Fake input.
    """
    np.random.seed(0)
    var = forward(model, x)
    loss = lambda y: 0.5 * (y ** 2).sum()
    grad_y = var['y']
    backward(model, grad_y, var)
    grad_w = model['dE_d' + name].ravel()
    w_ = model[name].ravel()
    eps = 1e-7
    grad_w_2 = np.zeros(w_.shape)
    check_elem = np.arange(w_.size)
    np.random.shuffle(check_elem)
    # Randomly check 20 elements.
    check_elem = check_elem[:20]
    for ii in check_elem:
        w_[ii] += eps
        err_plus = loss(forward(model, x)['y'])
        w_[ii] -= 2 * eps
        err_minus = loss(forward(model, x)['y'])
        w_[ii] += eps
        grad_w_2[ii] = (err_plus - err_minus) / 2 / eps
    np.testing.assert_almost_equal(grad_w[check_elem], grad_w_2[check_elem],
                                   decimal=3)
```

[23]:
```
# Export name
model_fname = 'nn_model.npz'
stats_fname = 'nn_stats.npz'
```

[141]:
```
# Setup hyperparameters
hyperparameters = {
    "num_hiddens": [16, 32],
    "eps": 0.1,
    "momentum": 0.0,
    "num_epochs": 1000,
```

```
    "batch_size": 100
}
```

[142]:
```
# Input-output dimensions.
num_inputs = 2304
num_outputs = 7

# Initialize model.
model = InitNN(num_inputs, hyperparameters["num_hiddens"], num_outputs)

# Uncomment to reload trained model here.
# model = Load(model_fname)

# Check gradient implementation.
print('Checking gradients...')
x = np.random.rand(10, 48 * 48) * 0.1
CheckGrad(model, NNForward, NNBackward, 'W3', x)
CheckGrad(model, NNForward, NNBackward, 'b3', x)
CheckGrad(model, NNForward, NNBackward, 'W2', x)
CheckGrad(model, NNForward, NNBackward, 'b2', x)
CheckGrad(model, NNForward, NNBackward, 'W1', x)
CheckGrad(model, NNForward, NNBackward, 'b1', x)
```

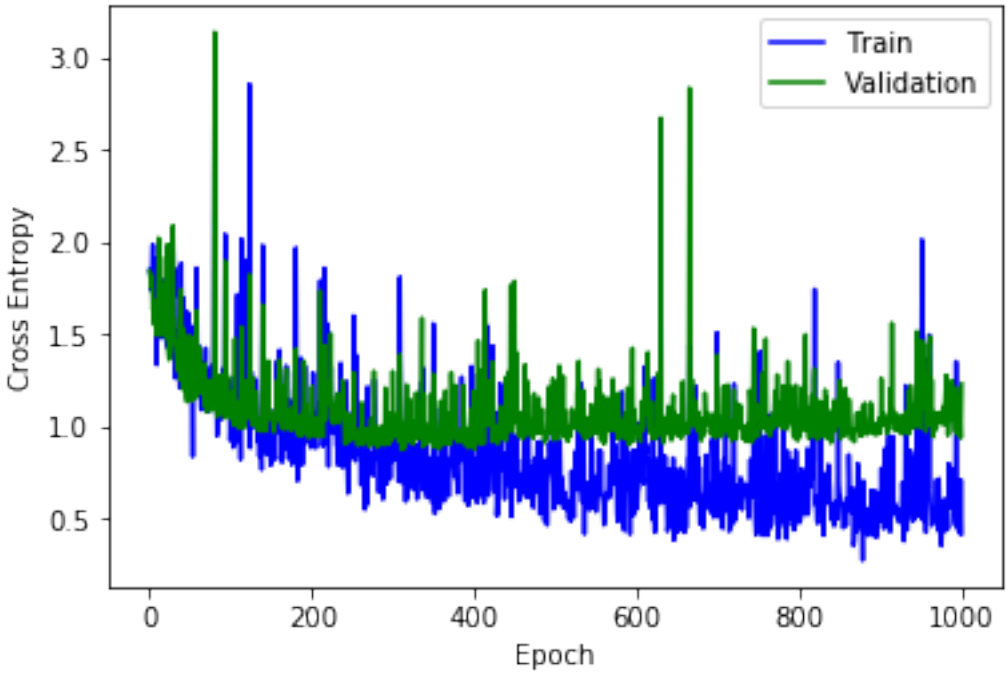
```
Checking gradients…
```

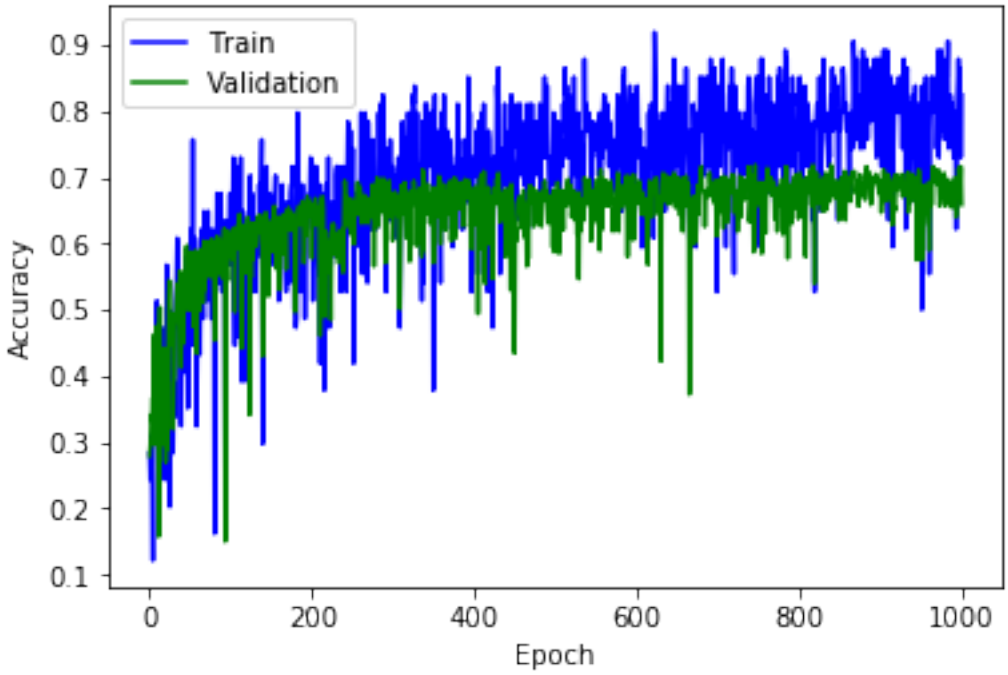
0.2 Q 3.1

Train with default parameters

[143]:
```
#Train model.
stats = Train(model, NNForward, NNBackward, NNUpdate, hyperparameters,␣
 ↪verbose=False, diagram=False)
DisplayPlot(stats[1]['train_ce'], stats[1]['valid_ce'], 'Cross Entropy',␣
 ↪number=0)
DisplayPlot(stats[1]['train_acc'], stats[1]['valid_acc'], 'Accuracy', number=1)
```

```
CE: Train 0.72919 Validation 1.23030 Test 1.18131
Acc: Train 0.73681 Validation 0.65871 Test 0.63377
```





Comment on the stats: The perfomance in terms of accuarcy of validation set is less than the trainning set.

[ ]:
```
#### Plot of cross entropy
![title](Figure_0.png)

#### Plot of accuracy
![title](Figure_1.png)
```

0.3 Q3.2

[26]:
```
import matplotlib.pyplot as plt
```

[54]:
```
def train_with_hyper_list(hypers, parameters, name):
    # Input-output dimensions.
    num_inputs = 2304
    num_outputs = 7

    hyper_clone = dict(hypers)

    train_ce = []
    train_acc = []
    valid_ce = []
    valid_acc = []

    for param in parameters:

        # Reinitialize model.
        model = InitNN(num_inputs, hypers["num_hiddens"], num_outputs)
        # Check gradient implementation.
        print('Checking gradients...')
        x = np.random.rand(10, 48 * 48) * 0.1
        CheckGrad(model, NNForward, NNBackward, 'W3', x)
        CheckGrad(model, NNForward, NNBackward, 'b3', x)
        CheckGrad(model, NNForward, NNBackward, 'W2', x)
        CheckGrad(model, NNForward, NNBackward, 'b2', x)
        CheckGrad(model, NNForward, NNBackward, 'W1', x)
        CheckGrad(model, NNForward, NNBackward, 'b1', x)
        print('Gradient check passed...')
        print('Starting trainning...')
        print()
        print(name + " = " + str(param) + ":")
        print()
        hyper_clone[name] = param
        # Train model.
        stats = Train(model, NNForward, NNBackward, NNUpdate, hyper_clone,␣
↪verbose=False, diagram=False)
        # Only concern the ce of the last iteration
```

```
        train_ce.append(stats[1]["train_ce"][-1][1])
        train_acc.append(stats[1]["train_acc"][-1][1])
        valid_ce.append(stats[1]["valid_ce"][-1][1])
        valid_acc.append(stats[1]["valid_acc"][-1][1])
        print()
    %matplotlib inline
    plt.figure(figsize=(20, 5))
    plt.subplot(121)
    plt.plot(parameters, train_ce, '-o', label="training set")
    plt.plot(parameters, valid_ce, '-o', label="validation set")
    plt.xlabel(name)
    plt.ylabel("cross entrophy loss")
    plt.legend()

    plt.subplot(122)
    plt.plot(parameters, train_acc, '-o', label="training set")
    plt.plot(parameters, valid_acc, '-o', label="validation set")
    plt.xlabel(name)
    plt.ylabel("accuracy")
    plt.legend()

    # Report the best hyperparameter setting
    argmin = np.argmin(valid_ce)
    best = parameters[argmin]
    print(name + " choosen: " + str(best))

    return best
```

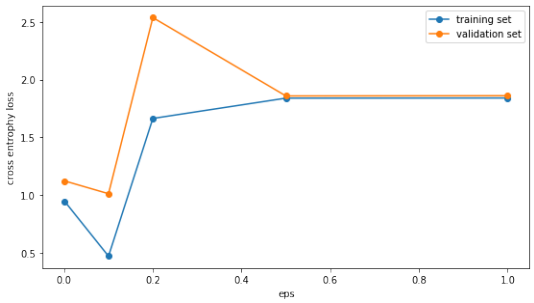
Train with different learning rates

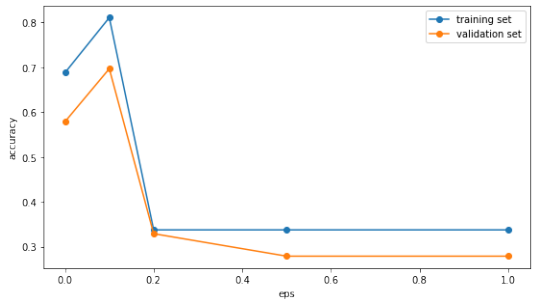
[44]:
```
learning_rates = [0.001, 0.1, 0.2, 0.5, 1.0]
```

[45]:
```
best_eps = train_with_hyper_list(hyperparameters, learning_rates, "eps")
```

```
Checking gradients…
Gradient check passed…
Starting trainning…

eps = 0.001:

CE: Train 1.10412 Validation 1.12326 Test 1.15264
Acc: Train 0.60729 Validation 0.57995 Test 0.57662

Checking gradients…
Gradient check passed…
Starting trainning…
```

```
eps = 0.1:

CE: Train 0.53448 Validation 1.01282 Test 0.96629
Acc: Train 0.80794 Validation 0.69690 Test 0.68312

Checking gradients…
Gradient check passed…
Starting trainning…

eps = 0.2:

CE: Train 2.45219 Validation 2.53740 Test 2.33438
Acc: Train 0.33817 Validation 0.32936 Test 0.37143

Checking gradients…
Gradient check passed…
Starting trainning…

eps = 0.5:

CE: Train 1.86108 Validation 1.85905 Test 1.83904
Acc: Train 0.28542 Validation 0.27924 Test 0.31688

Checking gradients…
Gradient check passed…
Starting trainning…

eps = 1.0:

CE: Train 1.86245 Validation 1.86104 Test 1.84036
Acc: Train 0.28542 Validation 0.27924 Test 0.31688

eps choosen: 0.1
```





0.3.1 Comment

As the learning rate increase, within 0.1, the larger the learning rate, the cross entropy was more convergent. However, as the learning rate get bigger 0.1, the cross entropy didn't converge as the learning rate increase. It is due to the fact that, the cross entrophy keep escaping from the global minimum.

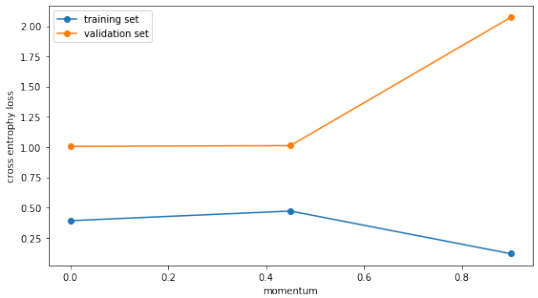
[84]:
```
hyperparameters["eps"] = best_eps
```

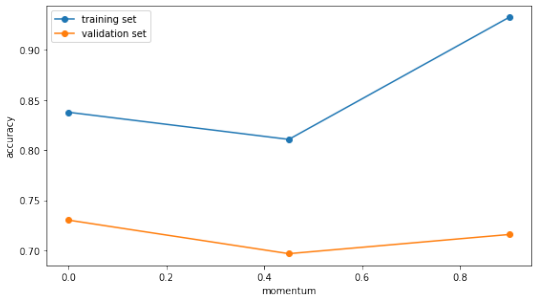
Given the choosen best learning rate, try different momentum.

[47]:
```
momentums = [0.0, 0.45, 0.9]
```

[48]:
```
best_momentum = train_with_hyper_list(hyperparameters, momentums, "momentum")
```

```
Checking gradients…
Gradient check passed…
Starting trainning…

momentum = 0.0:

CE: Train 0.38254 Validation 1.00641 Test 0.91246
Acc: Train 0.86663 Validation 0.73031 Test 0.71169

Checking gradients…
Gradient check passed…
Starting trainning…

momentum = 0.45:

CE: Train 0.53448 Validation 1.01282 Test 0.96629
Acc: Train 0.80794 Validation 0.69690 Test 0.68312

Checking gradients…
Gradient check passed…
Starting trainning…

momentum = 0.9:

CE: Train 0.30314 Validation 2.07246 Test 2.10522
Acc: Train 0.89923 Validation 0.71599 Test 0.66234

momentum choosen: 0.0
```





0.3.2 Comment

As the momentum increases, the convergence of the trainning set doesn't change a lot, but the convergence of the validation set increases.

[83]:
```
hyperparameters["momentum"] = best_momentum
```

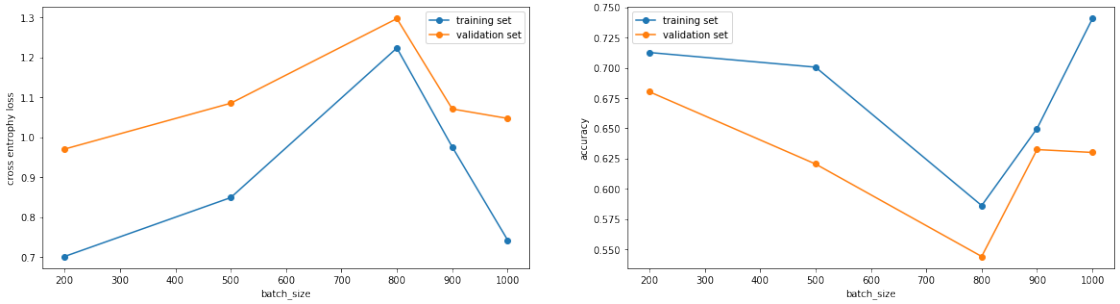
Given the choosen best momentum and learning rate, try different mini-batch sizes.

[59]:
```
batches = [200, 500, 800, 900, 1000]
```

[60]:
```
best_batch_size = train_with_hyper_list(hyperparameters, batches, "batch_size")
```

```
Checking gradients…
Gradient check passed…
Starting trainning…

batch_size = 200:

CE: Train 0.66596 Validation 0.97049 Test 0.88406
Acc: Train 0.76378 Validation 0.68019 Test 0.68831

Checking gradients…
Gradient check passed…
Starting trainning…

batch_size = 500:

CE: Train 1.00303 Validation 1.08501 Test 1.08839
Acc: Train 0.63367 Validation 0.62053 Test 0.59221

Checking gradients…
Gradient check passed…
Starting trainning…
```

```
batch_size = 800:

CE: Train 1.29053 Validation 1.29646 Test 1.32629
Acc: Train 0.52786 Validation 0.54415 Test 0.52727

Checking gradients…
Gradient check passed…
Starting trainning…

batch_size = 900:

CE: Train 1.01786 Validation 1.07067 Test 1.03939
Acc: Train 0.62596 Validation 0.63246 Test 0.60260

Checking gradients…
Gradient check passed…
Starting trainning…

batch_size = 1000:

CE: Train 0.98664 Validation 1.04708 Test 1.07973
Acc: Train 0.63041 Validation 0.63007 Test 0.62857

batch_size choosen: 200
```



0.3.3 Comment

The cross entrophy losses of both training and test set first increase until the batch size reach 800. After 800, the cross entrophy losses start to decrease. However, the accuracy of predicting the label on the validation set is never as high as the when the batch size is 200. It is obvious that, when the batch is 200, the difference of the cross entrophy losses between the validation set and the training set is the greatest. It indicates that the smaller batch size helps to reduce overfit of the model.

[85]:
```
hyperparameters["batch_size"] = best_batch_size
```

When tuning the eps, momentum, and batch size of the network, I first tried different values of learning rate and select the one which yeilded the lowest cross validation error, then fixed the momentum and batch size one by one by fixing the others and trying different values and select the one yielded the lowest cross validation error.

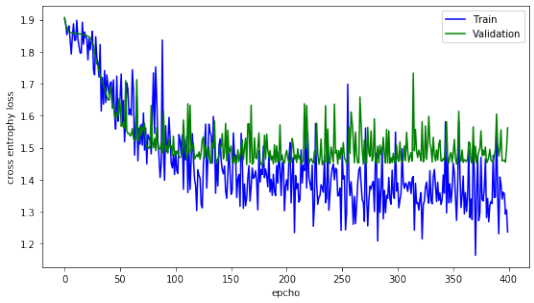
0.4 Q3.3

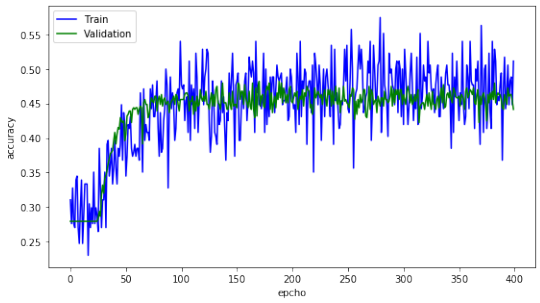
Try 3 different values fo the number of hiddent units

[82]:
```
# Fix momentum
hyperparameters["momentum"] = 0.9
# Change learning rate and number of epochs
hyperparameters["eps"] = 0.1
hyperparameters["num_epochs"] = 400
index = 0

hidden_units = [[1, 2], [30, 60], [50, 100]]
for hidden_unit in hidden_units:

    # Reinitialize model.
    model = InitNN(num_inputs, hidden_unit, num_outputs)

    # Uncomment to reload trained model here.
    # model = Load(model_fname)

    # Check gradient implementation.
    print('Checking gradients...')
    x = np.random.rand(10, 48 * 48) * 0.1
    CheckGrad(model, NNForward, NNBackward, 'W3', x)
    CheckGrad(model, NNForward, NNBackward, 'b3', x)
    CheckGrad(model, NNForward, NNBackward, 'W2', x)
    CheckGrad(model, NNForward, NNBackward, 'b2', x)
    CheckGrad(model, NNForward, NNBackward, 'W1', x)
    CheckGrad(model, NNForward, NNBackward, 'b1', x)
    print('Gradient check passed...')
    print()
    print('Starting trainning...')
    print()
    print("hiddent_unit" + " = " + str(hidden_unit) + ":")
    # Train model.
    stats = Train(model, NNForward, NNBackward, NNUpdate, hyperparameters,␣
↪verbose=False, diagram=False)
    %matplotlib inline
    plt.figure(figsize=(20, 5))
    plt.subplot(121)
    train = np.array(stats[1]["train_ce"])
    valid = np.array(stats[1]["valid_ce"])
```

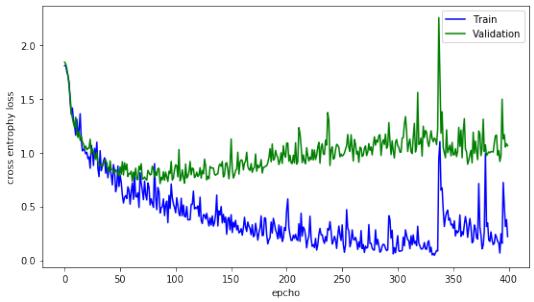
```
    plt.plot(train[:, 0], train[:, 1], 'b', label='Train')
    plt.plot(valid[:, 0], valid[:, 1], 'g', label='Validation')
    plt.xlabel("epcho")
    plt.ylabel("cross entrophy loss")
    plt.legend()

    plt.subplot(122)
    train = np.array(stats[1]["train_acc"])
    valid = np.array(stats[1]["valid_acc"])
    plt.plot(train[:, 0], train[:, 1], 'b', label='Train')
    plt.plot(valid[:, 0], valid[:, 1], 'g', label='Validation')
    plt.xlabel("epcho")
    plt.ylabel("accuracy")
    plt.legend()
    plt.draw()
    plt.pause(0.0001)
    print()
```

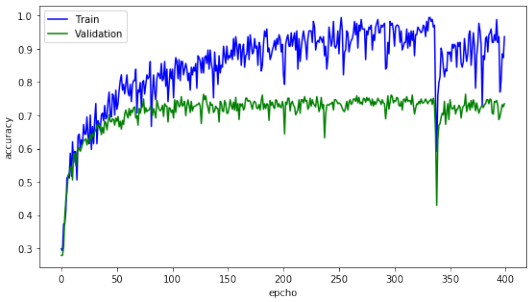
```
Checking gradients...
Gradient check passed...

Starting trainning...

hiddent_unit = [1, 2]:
CE: Train 1.39660 Validation 1.56225 Test 1.50709
Acc: Train 0.46680 Validation 0.44153 Test 0.44935
```



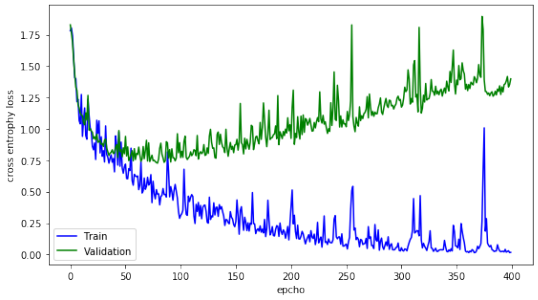


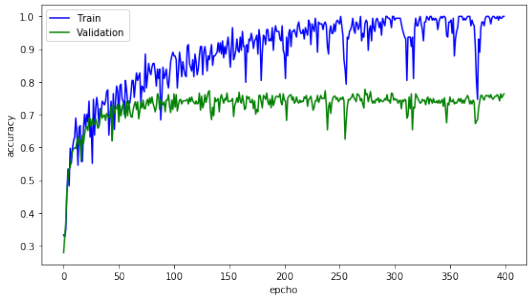

```
Checking gradients...
Gradient check passed...

Starting trainning...

hiddent_unit = [30, 60]:
CE: Train 0.25548 Validation 1.06624 Test 0.96192
Acc: Train 0.90427 Validation 0.73508 Test 0.70130
```






```
Checking gradients…
Gradient check passed…

Starting trainning…

hiddent_unit = [50, 100]:
CE: Train 0.01521 Validation 1.39853 Test 1.02426
Acc: Train 0.99911 Validation 0.76372 Test 0.77403
```





0.4.1 Comment:

The networks with more hidden units have more generality. It can be seen from the fact that the accuarcy on the validation and test set is higher than the network with less hidden units. However, The networks with more hidden unit are more likely to overfit, since the accuracy on the training set is much higher than the accuarcy on the validation set on those networks. The networks with only 3 hidden units is underfittng, and the accuracy difference is very small between the trainning set and validation set.

0.5 Q3.4

[88]:
```
# Setup hyperparameters
hyperparameters = {
    "num_hiddens": [16, 32],
    "eps": 0.1,
    "momentum": 0.0,
    "num_epochs": 1000,
    "batch_size": 100
}

# Set everything to the best configuration
hyperparameters["eps"] = best_eps
hyperparameters["momentum"] = best_momentum
hyperparameters["batch_size"] = best_batch_size
```

[91]:
```
# Train the final model
model = InitNN(num_inputs, hyperparameters["num_hiddens"], num_outputs)
stats = Train(model, NNForward, NNBackward, NNUpdate, hyperparameters,␣
 ↪verbose=False, diagram=False)

# Load the test data
inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
    target_test = LoadData('./toronto_face.npz')
```

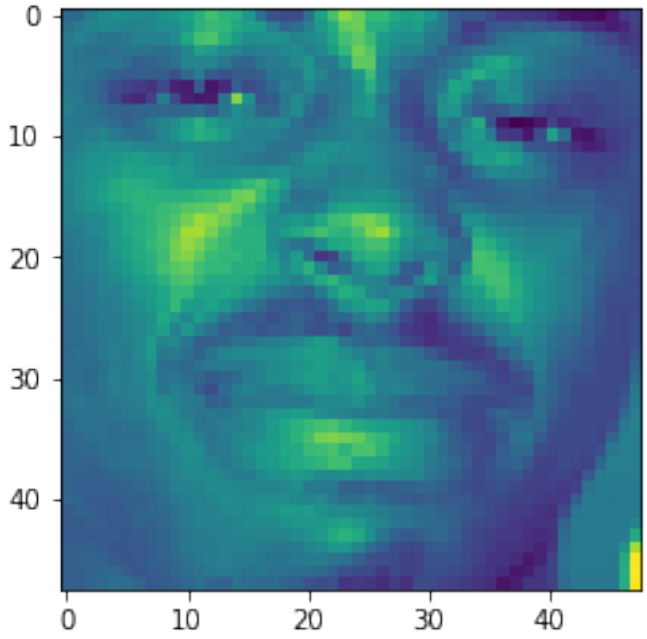
```
CE: Train 0.63639 Validation 0.96965 Test 1.16427
Acc: Train 0.76615 Validation 0.71360 Test 0.70130
```

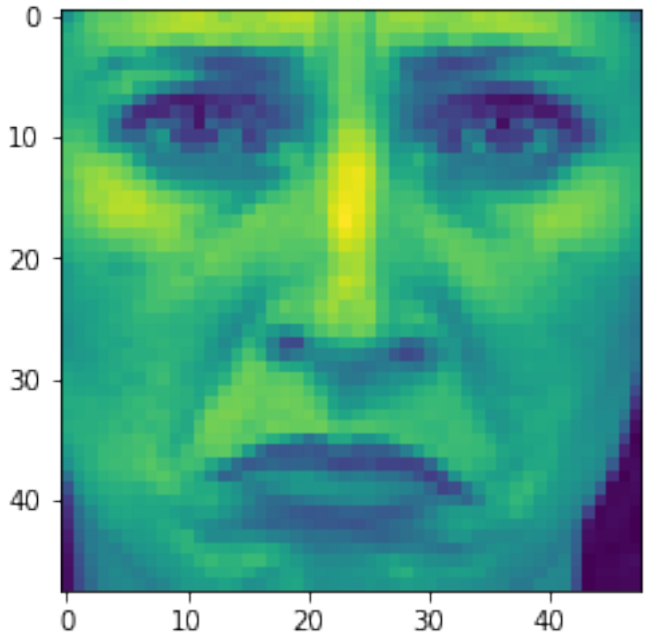
[136]:
```
# Predict label with the given model
y = NNForward(model, inputs_test)['y']

# Transfer the prediction to probabilities
prob = Softmax(y)

# Calculate the score (confidence level) for each sample
scores = np.amax(prob, axis=1)

# Find the unconfident prediction
unconfident = []
for i in range(len(scores)):
    if scores[i] < 0.3:
        unconfident.append(i)
```

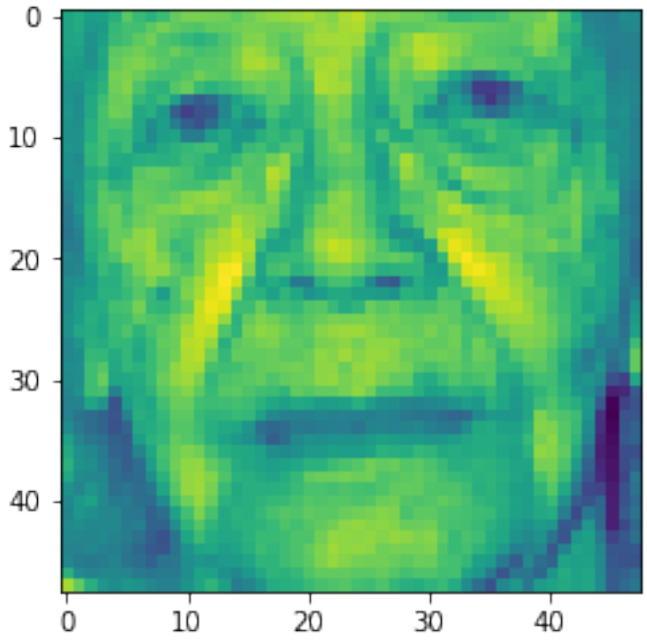
[137]:
```
unconfident
```

[137]:
```
[87, 276, 339]
```

[138]:
```
%matplotlib inline
for image_index in unconfident:
    plt.imshow(np.reshape(inputs_test[image_index], newshape=(48,48)))
    plt.pause(0.0001)
```







0.5.1 Comment

It is hard even human being to define what emotion the person is in the image. For example, in the first image, It is hard to tell if the person is happy or sad. And for the second image, it hard to tell if the person is angry or sad. Therefore, it is not likely that the network will predict it correctly.

[ ]: