

# Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

In [17]:

```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from asgn1.classifiers.neural_net import TwoLayerNet

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

The autoreload extension is already loaded. To reload it, use:

```
%reload_ext autoreload
```

We will use the class `TwoLayerNet` in the file `asgn1/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

In [18]:

```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

## Forward pass: compute scores

Open the file `asgn1/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

In [19]:

```
scores = net.loss(X)
print 'Your scores:'
print scores
print
print 'correct scores:'
correct_scores = np.asarray([
    [-0.81233741, -1.27654624, -0.70335995],
    [-0.17129677, -1.18803311, -0.47310444],
    [-0.51590475, -1.01354314, -0.8504215 ],
    [-0.15419291, -0.48629638, -0.52901952],
    [-0.00618733, -0.12435261, -0.15226949]])
print correct_scores
print

# The difference should be very small. We get < 1e-7
print 'Difference between your scores and correct scores:'
print np.sum(np.abs(scores - correct_scores))
```

Your scores:

```
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]
```

correct scores:

```
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]
```

Difference between your scores and correct scores:

3.68027207459e-08

## Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization loss.

In [20]:

```
loss, _ = net.loss(X, y, reg=0.1)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print 'Difference between your loss and correct loss:'
print np.sum(np.abs(loss - correct_loss))
```

Difference between your loss and correct loss:

1.79856129989e-13

## Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables  $W1$ ,  $b1$ ,  $W2$ , and  $b2$ . Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

In [23]:

```
from asgn1.gradient_check import eval_numerical_gradient

# Use numericprint lossgradient checking to check your implementation of the backwa
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.
loss, grads = net.loss(X, y, reg=0.1)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.1)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
    print '%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[
```

```
W1 max relative error: 3.561318e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447625e-11
b1 max relative error: 2.738421e-09
```

## Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

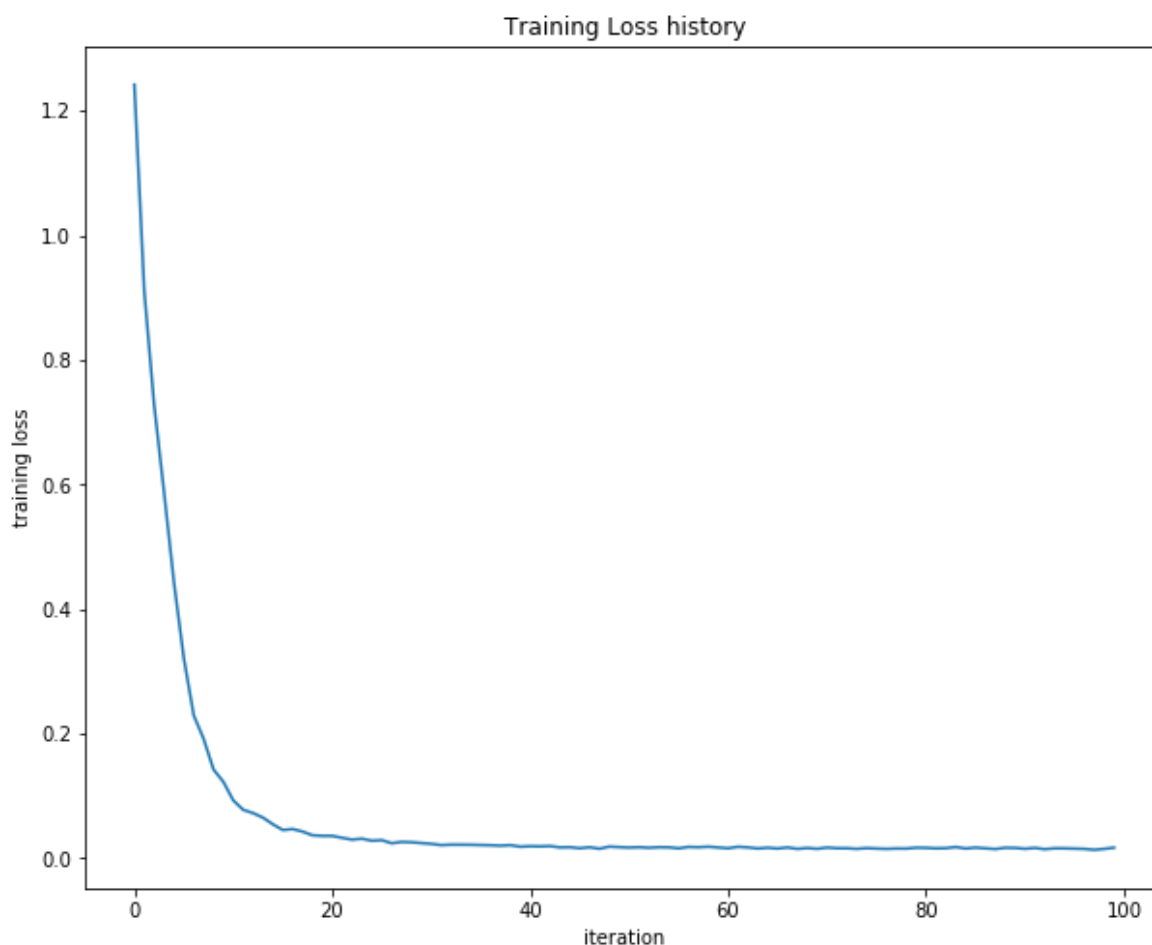
In [24]:

```
net = init_toy_model()
stats = net.train(X, y, X, y,
                  learning_rate=1e-1, reg=1e-5,
                  num_iters=100, verbose=False)

print 'Final training loss: ', stats['loss_history'][-1]

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

Final training loss: 0.0171496079387



## Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

In [25]:

```
from asgn1.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print 'Train data shape: ', X_train.shape
print 'Train labels shape: ', y_train.shape
print 'Validation data shape: ', X_val.shape
print 'Validation labels shape: ', y_val.shape
print 'Test data shape: ', X_test.shape
print 'Test labels shape: ', y_test.shape
```

```
Train data shape: (49000, 3072)
Train labels shape: (49000,)
Validation data shape: (1000, 3072)
Validation labels shape: (1000,)
Test data shape: (1000, 3072)
Test labels shape: (1000,)
```

## Train a network

To train our network we will use SGD with momentum. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [26]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
                  num_iters=1000, batch_size=200,
                  learning_rate=1e-4, learning_rate_decay=0.95,
                  reg=0.5, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print 'Validation accuracy: ', val_acc
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy: 0.287
```

## Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

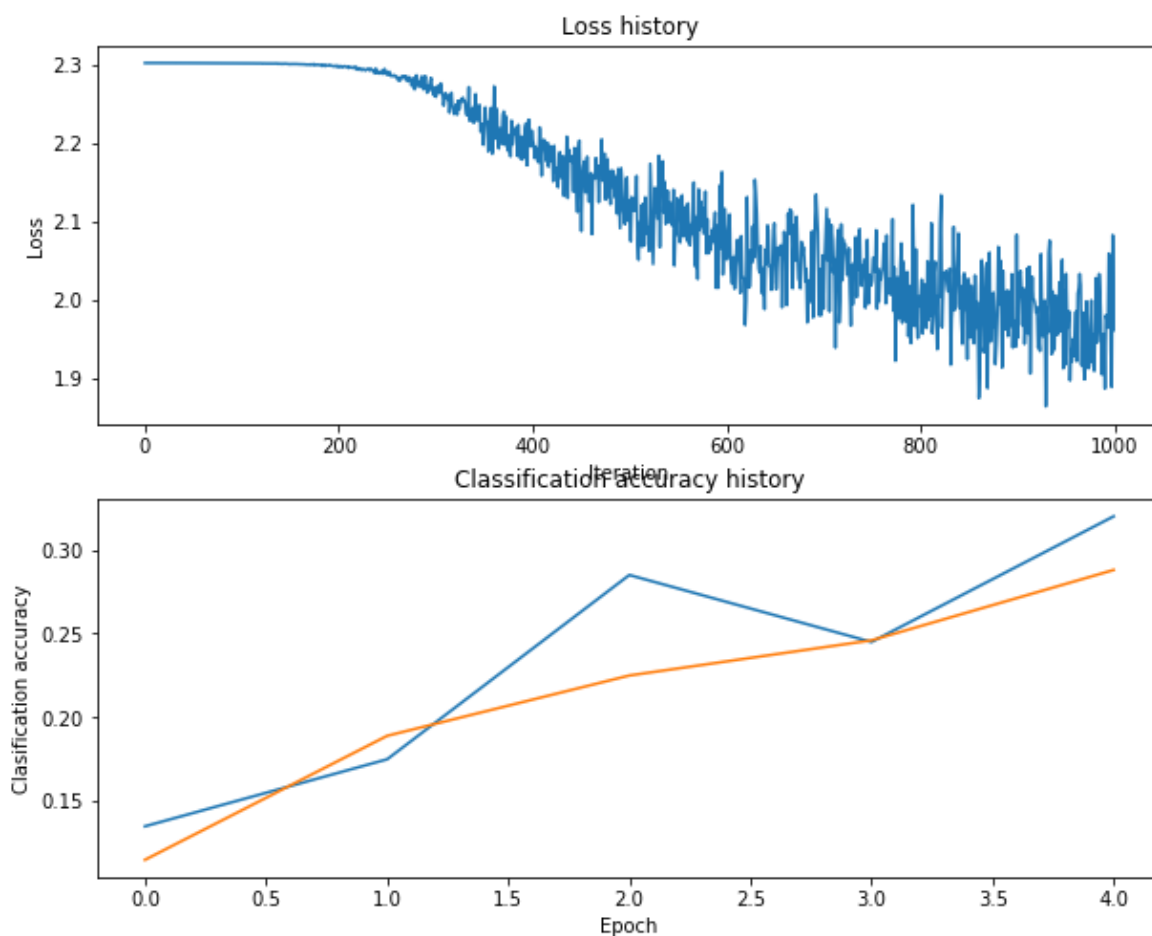
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [27]:

```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.show()
```





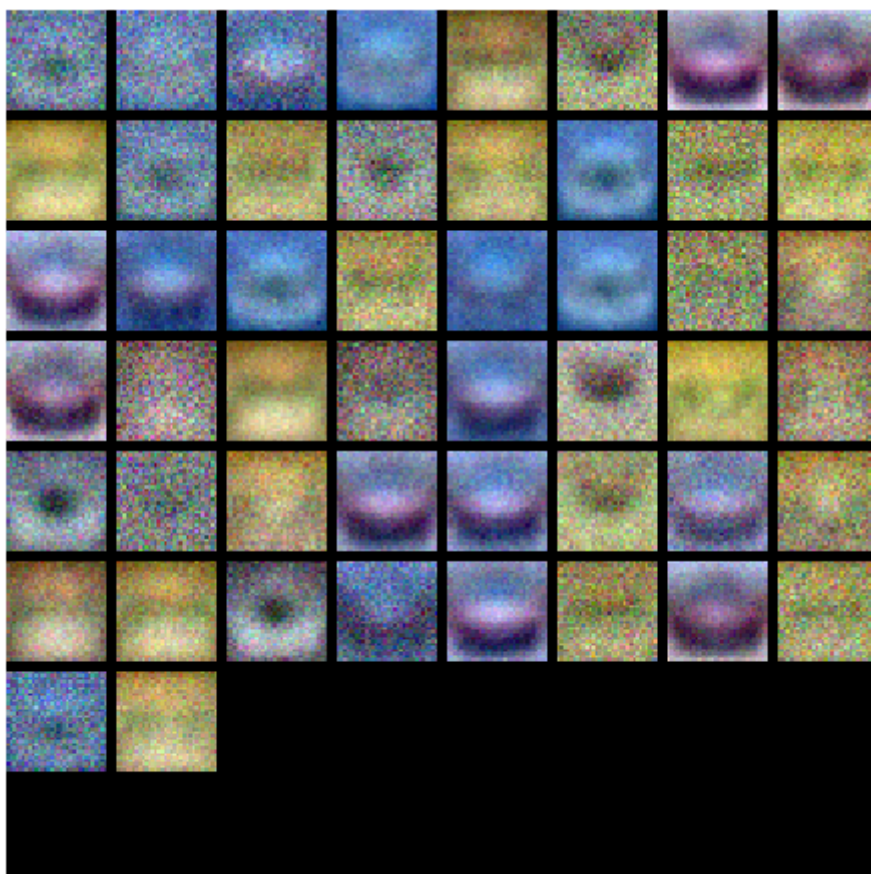
In [28]:

```
from asgn1.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



## Tune your hyperparameters

**What's wrong?.** Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

**Tuning.** Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, number of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

**Approximate results.** You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

**Experiment:** Your goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 52% on the Test set we will award you with one extra bonus point. Feel free to implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [32]:

```
#best_net = None # store the best model into this

#####
# TODO: Tune hyperparameters using the validation set. Store your best trained #
# model in best_net. #
# #
# To help debug your network, it may help to use visualizations similar to the #
# ones we used above; these visualizations will have significant qualitative #
# differences from the ones we saw above for the poorly tuned network. #
# #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to #
# write code to sweep through possible combinations of hyperparameters #
# automatically like we did on the previous exercises. #
#####
results = {}
learning_rates=[1e-3, 3e-3]
regularization_strengths = [0.4, 0.5]
best_val = -1 # The highest validation accuracy that we have seen so far.
best_net = None
input_size = 32 * 32 * 3
hidden_size = 500
num_classes = 10# The LinearSVM object that achieved the highest validation rate.
for i in learning_rates:
    for j in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, num_classes)
        stats = net.train(X_train, y_train, X_val, y_val,
                           num_iters=1600, batch_size=200,
                           learning_rate=i, learning_rate_decay=0.95,
                           reg=j, verbose=True)
        y_train_pred = net.predict(X_train)
        y_val_pred = net.predict(X_val)
        results[i,j]=[np.mean(y_train == y_train_pred), np.mean(y_val == y_val_pred)]
        print np.mean(y_val == y_val_pred)
        if (np.mean(y_val == y_val_pred))>best_val:
            best_val=np.mean(y_val == y_val_pred)
            best_net=net

#####
#                               END OF YOUR CODE                               #
#####
```

```
iteration 0 / 1600: loss 2.305770
iteration 100 / 1600: loss 1.934858
iteration 200 / 1600: loss 1.647140
iteration 300 / 1600: loss 1.504476
iteration 400 / 1600: loss 1.641825
iteration 500 / 1600: loss 1.540090
iteration 600 / 1600: loss 1.481165
iteration 700 / 1600: loss 1.444499
iteration 800 / 1600: loss 1.490395
iteration 900 / 1600: loss 1.461228
iteration 1000 / 1600: loss 1.424270
iteration 1100 / 1600: loss 1.412187
iteration 1200 / 1600: loss 1.420180
iteration 1300 / 1600: loss 1.417426
iteration 1400 / 1600: loss 1.418652
iteration 1500 / 1600: loss 1.320682
0.508
iteration 1600 / 1600: loss 1.320682
```

```
iteration 0 / 1600: loss 2.300410
iteration 100 / 1600: loss 1.851082
iteration 200 / 1600: loss 1.747992
iteration 300 / 1600: loss 1.694262
iteration 400 / 1600: loss 1.646877
iteration 500 / 1600: loss 1.708792
iteration 600 / 1600: loss 1.582569
iteration 700 / 1600: loss 1.560476
iteration 800 / 1600: loss 1.443569
iteration 900 / 1600: loss 1.424598
iteration 1000 / 1600: loss 1.461114
iteration 1100 / 1600: loss 1.475756
iteration 1200 / 1600: loss 1.338838
iteration 1300 / 1600: loss 1.474724
iteration 1400 / 1600: loss 1.459671
iteration 1500 / 1600: loss 1.391509
```

0.498

```
iteration 0 / 1600: loss 2.305703
iteration 100 / 1600: loss 1.813784
iteration 200 / 1600: loss 1.735537
iteration 300 / 1600: loss 1.573730
iteration 400 / 1600: loss 1.800589
iteration 500 / 1600: loss 1.671707
iteration 600 / 1600: loss 1.923902
iteration 700 / 1600: loss 1.575880
iteration 800 / 1600: loss 1.565538
iteration 900 / 1600: loss 1.556297
iteration 1000 / 1600: loss 1.652356
iteration 1100 / 1600: loss 1.668987
iteration 1200 / 1600: loss 1.684082
iteration 1300 / 1600: loss 1.556227
iteration 1400 / 1600: loss 1.688606
iteration 1500 / 1600: loss 1.808763
```

0.448

```
iteration 0 / 1600: loss 2.306405
iteration 100 / 1600: loss 1.779588
iteration 200 / 1600: loss 2.007892
iteration 300 / 1600: loss 1.876091
iteration 400 / 1600: loss 1.578076
iteration 500 / 1600: loss 1.480422
iteration 600 / 1600: loss 2.007673
iteration 700 / 1600: loss 1.701077
iteration 800 / 1600: loss 1.588643
iteration 900 / 1600: loss 1.719767
iteration 1000 / 1600: loss 1.836804
iteration 1100 / 1600: loss 1.794155
iteration 1200 / 1600: loss 1.482373
iteration 1300 / 1600: loss 1.419031
iteration 1400 / 1600: loss 1.703112
iteration 1500 / 1600: loss 1.925961
```

0.459