exercise_1

November 13, 2018

```
In [1]: import numpy as np
        import _pickle as cPickle
        import os
        import gzip
        import matplotlib.pyplot as plt
```

1 First exercise: Classifying MNIST with MLPs

In this exercise you will implement a Neural Network (or MLP) and classify the MNIST digits with it. MNIST is a "well hung" dataset that has been used a lot over the years to benchmark different classification algorithms. To learn more about it have a look here: http://yann.lecun.com/exdb/mnist/ .

2 Data Loading

We first define a function for downloading and loading MNIST. **WARNING**: Executing it will obviously use up some space on your machine ;).

```
In [2]: def mnist(datasets_dir='./data'):
            if not os.path.exists(datasets_dir):
                os.mkdir(datasets_dir)
            data_file = os.path.join(datasets_dir, 'mnist.pkl.gz')
            if not os.path.exists(data_file):
                print('... downloading MNIST from the web')
                try:
                    import urllib
                    urllib.urlretrieve('http://google.com')
                except AttributeError:
                    import urllib.request as urllib
                url = 'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'
                urllib.urlretrieve(url, data_file)

            print('... loading data')
            # Load the dataset
            f = gzip.open(data_file, 'rb')
            try:
                train_set, valid_set, test_set = cPickle.load(f, encoding="latin1")
```

```
    except TypeError:
        train_set, valid_set, test_set = cPickle.load(f)
    f.close()

    test_x, test_y = test_set
    test_x = test_x.astype('float32')
    test_x = test_x.astype('float32').reshape(test_x.shape[0], 1, 28, 28)
    test_y = test_y.astype('int32')
    valid_x, valid_y = valid_set
    valid_x = valid_x.astype('float32')
    valid_x = valid_x.astype('float32').reshape(valid_x.shape[0], 1, 28, 28)
    valid_y = valid_y.astype('int32')
    train_x, train_y = train_set
    train_x = train_x.astype('float32').reshape(train_x.shape[0], 1, 28, 28)
    train_y = train_y.astype('int32')
    rval = [(train_x, train_y), (valid_x, valid_y), (test_x, test_y)]
    print('... done loading data')
    return rval
```

3 Neural Network Layers

We now define "bare bone" neural network layers. The parts marked with **TODO** are where you should finish the implementation! Conceptually we will implement the layers as follows:

Each layer has a constructor that takes an input layer plus some additional arguments such as layer size and the activation function name. The layer then uses the provided input layer to compute the layer dimensions, weight shapes, etc. and setup all auxilliary variables.

Each layer then has to provide three functions (as defined in the Layer class below): *output_shape()*, *fprop()* and *brop()*. The output_shape function is used to figure out the shape for the next layer and the *fprop()/bprop()* functions are used to compute forward and backward passes through the network.

```
In [3]: # start by defining simple helpers
        def sigmoid(x):
            return 1.0/(1.0+np.exp(-x))

        def sigmoid_d(x):
            # <MJ>
            x = sigmoid(x) * (1 - sigmoid(x))
            # </MJ>
            return x

        def tanh(x):
            return np.tanh(x)

        def tanh_d(x):
            # <MJ>
            x = 1.0 - tanh(x)**2
```

```
    # </MJ>
    return x

def relu(x):
    return np.maximum(0.0, x)

def relu_d(x):
    # <MJ>
    dx = np.zeros_like(x)
    dx[x>0] = 1
    # </MJ>
    return dx

# <MJ>
def linear(x):
    return x

def linear_d(x):
    return 1
# </MJ>

def softmax(x, axis=1):
    # to make the softmax a "safe" operation we will
    # first subtract the maximum along the specified axis
    # so that np.exp(x) does not blow up!
    # Note that this does not change the output.
    x_max = np.max(x, axis=axis, keepdims=True)
    x_safe = x - x_max
    e_x = np.exp(x_safe)
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

def one_hot(labels):
    """this creates a one hot encoding from a flat vector:
    i.e. given y = [0,2,1]
     it creates y_one_hot = [[1,0,0], [0,0,1], [0,1,0]]
    """
    classes = np.unique(labels)
    n_classes = classes.size
    one_hot_labels = np.zeros(labels.shape + (n_classes,))
    for c in classes:
        one_hot_labels[labels == c, c] = 1
    return one_hot_labels

def unhot(one_hot_labels):
    """ Invert a one hot encoding, creating a flat vector """
    return np.argmax(one_hot_labels, axis=-1)

# then define an activation function class
```

```
class Activation(object):

    def __init__(self, tname):
        if tname == 'sigmoid':
            self.act = sigmoid
            self.act_d = sigmoid_d
        elif tname == 'tanh':
            self.act = tanh
            self.act_d = tanh_d
        elif tname == 'relu':
            self.act = relu
            self.act_d = relu_d
        else:
            # <MJ>
            self.act = linear
            self.act_d = linear_d
            # </MJ>

    def fprop(self, input):
        # we need to remember the last input
        # so that we can calculate the derivative with respect
        # to it later on
        self.last_input = input
        return self.act(input)

    def bprop(self, output_grad):
        return output_grad * self.act_d(self.last_input)

# define a base class for layers
class Layer(object):

    def fprop(self, input):
        """ Calculate layer output for given input
            (forward propagation).
        """
        raise NotImplementedError('This is an interface class, please use a derived inst

    def bprop(self, output_grad):
        """ Calculate input gradient and gradient
            with respect to weights and bias (backpropagation).
        """
        raise NotImplementedError('This is an interface class, please use a derived inst

    def output_size(self):
        """ Calculate size of this layer's output.
        input_shape[0] is the number of samples in the input.
        input_shape[1:] is the shape of the feature.
        """
```

```
            raise NotImplementedError('This is an interface class, please use a derived inst

    # define a base class for loss outputs
    # an output layer can then simply be derived
    # from both Layer and Loss
    class Loss(object):

        def loss(self, output, output_net):
            """ Calculate mean loss given real output and network output. """
            raise NotImplementedError('This is an interface class, please use a derived inst

        def input_grad(self, output, output_net):
            """ Calculate input gradient real output and network output. """
            raise NotImplementedError('This is an interface class, please use a derived inst

    # define a base class for parameterized things
    class Parameterized(object):

        def params(self):
            """ Return parameters (by reference) """
            raise NotImplementedError('This is an interface class, please use a derived inst

        def grad_params(self):
            """ Return accumulated gradient with respect to params. """
            raise NotImplementedError('This is an interface class, please use a derived inst

    # define a container for providing input to the network
    class InputLayer(Layer):

        def __init__(self, input_shape):
            if not isinstance(input_shape, tuple):
                raise ValueError("InputLayer requires input_shape as a tuple")
            self.input_shape = input_shape

        def output_size(self):
            return self.input_shape

        def fprop(self, input):
            return input

        def bprop(self, output_grad):
            return output_grad

    class FullyConnectedLayer(Layer, Parameterized):
        """ A standard fully connected hidden layer, as discussed in the lecture.
        """

        def __init__(self, input_layer, num_units,
                 init_stddev, activation_fun=Activation('relu')):
        self.num_units = num_units
        self.activation_fun = activation_fun
        # the input shape will be of size (batch_size, num_units_prev)
        # where num_units_prev is the number of units in the input
        # (previous) layer
        self.input_shape = input_layer.output_size()

        # <MJ>
        # this is the weight matrix it should have shape: (num_units_prev, num_units)
        self.W = np.random.normal(0, init_stddev, (self.input_shape[1], num_units))
        # and this is the bias vector of shape: (num_units)

        self.b = np.zeros(shape = (1, num_units))
        # </MJ>

        # create dummy variables for parameter gradients
        # no need to change these here!
        self.dW = None
        self.db = None

    def output_size(self):
        return (self.input_shape[0], self.num_units)

    def fprop(self, input):
        # <MJ>
        self.last_input = input
        W, b = self.params()
        if (self.activation_fun == None):
            return np.dot(self.last_input, W) + b
        last_input = self.activation_fun.fprop(np.dot(self.last_input, W) + b)
        return last_input
        # </MJ>

    def bprop(self, output_grad):
        """ Calculate input gradient (backpropagation). """

        # <MJ>
        n = output_grad.shape[0]
        if (self.activation_fun == None):
            gradient = output_grad
        else:
            gradient = self.activation_fun.bprop(output_grad)
        self.dW = (1/n) * np.dot(self.last_input.T, gradient)
        self.db = (1/n) * np.sum(gradient, axis = 0, keepdims = True)

        grad_input = np.dot(gradient, self.W.T)
        return grad_input
        # </MJ>

    def params(self):
        return self.W, self.b

    def grad_params(self):
        return self.dW, self.db

# finally we specify the interface for output layers
# which are layers that also have a loss function
# we will implement two output layers:
#  a Linear, and Softmax (Logistic Regression) layer
# The difference between output layers and and normal
# layers is that they will be called to compute the gradient
# of the loss through input_grad(). bprop will never
# be called on them!
class LinearOutput(Layer, Loss):
    """ A simple linear output layer that
        uses a squared loss (e.g. should be used for regression)
    """
    def __init__(self, input_layer):
        self.input_size = input_layer.output_size()

    def output_size(self):
        return (1,)

    def fprop(self, input):
        return input

    def bprop(self, output_grad):
        raise NotImplementedError(
            'LinearOutput should only be used as the last layer of a Network'
            + ' bprop() should thus never be called on it!'
        )

    def input_grad(self, Y, Y_pred):
        # <MJ>
        return Y_pred - Y
        # </MJ>

    def loss(self, Y, Y_pred):
        loss = 0.5 * np.square(Y_pred - Y)
        return np.mean(np.sum(loss, axis=1))

class SoftmaxOutput(Layer, Loss):
    """ A softmax output layer that calculates
        the negative log likelihood as loss
        and should be used for classification.
        """

        def __init__(self, input_layer):
            self.input_size = input_layer.output_size()

        def output_size(self):
            return (1,)

        def fprop(self, input):
            return softmax(input)

        def bprop(self, output_grad):
            raise NotImplementedError(
                'SoftmaxOutput should only be used as the last layer of a Network'
                + ' bprop() should thus never be called on it!'
            )

        def input_grad(self, Y, Y_pred):
            # <MJ>
            return Y_pred - Y
            # </MJ>

        def loss(self, Y, Y_pred):
            # Assume one-hot encoding of Y
            out = Y_pred
            # to make the loss numerically stable
            # you should add an epsilon in the log ;)
            eps = 1e-10

            # <MJ>
            loss = np.sum(-np.log(out + eps) * Y, axis = 1)
            # </MJ>
            return np.mean(loss)
```

4 Neural Network class

With all layers in place (and properly implemented by you) we can finally define a neural network. For our purposes a neural network is simply a collection of layers which we will cycle through and on which we will call fprop and bprop to compute partial derivatives with respect to the input and the parameters.

Pay special attention to the *check_gradients()* function in which you should implement automatic differentiation. This function will become your best friend when checking the correctness of your implementation.

```
In [4]: class NeuralNetwork:
            """ Our Neural Network container class.
```

```
"""
def __init__(self, layers):
    self.layers = layers

def _loss(self, X, Y):
    Y_pred = self.predict(X)
    return self.layers[-1].loss(Y, Y_pred)

def predict(self, X):
    """ Calculate an output Y for the given input X. """
    Y_pred = X
    # <MJ>
    for layer in range(len(self.layers)):
        Y_pred = self.layers[layer].fprop(Y_pred)
    # </MJ>

    return Y_pred

def backpropagate(self, Y, Y_pred, upto=0):
    """ Backpropagation of partial derivatives through
        the complete network up to layer 'upto'
    """
    next_grad = self.layers[-1].input_grad(Y, Y_pred)
    # <MJ>
    for i in reversed(range(upto, len(self.layers) - 1)):
        next_grad = self.layers[i].bprop(next_grad)
    # </MJ>

    return next_grad

def classification_error(self, X, Y):
    """ Calculate error on the given data
        assuming they are classes that should be predicted.
    """
    Y_pred = unhot(self.predict(X))
    error = Y_pred != Y
    return np.mean(error)

def sgd_epoch(self, X, Y, learning_rate, batch_size):
    n_samples = X.shape[0]
    n_batches = n_samples // batch_size
    for b in range(n_batches):
        # <MJ>
        x_batch = X[b * batch_size: (b + 1) * batch_size]
        y_batch = Y[b * batch_size: (b + 1) * batch_size]
        y_pred = self.predict(x_batch)
        gradient = self.backpropagate(y_batch, y_pred)
```

```
            for layer in range(1, len(self.layers) - 1):
                W, b = self.layers[layer].params()
                dW, db = self.layers[layer].grad_params()
                W -= learning_rate * dW
                b -= learning_rate * db
            # </MJ>
            pass

    def gd_epoch(self, X, Y):
        # (you can assume the inputs are already shuffled)
        # <MJ>
        y_pred = self.predict(X)
        gradient = self.backpropagate(y_batch, y_pred, upto = 1)

        for layer in range(1, len(self.layers) - 1):
            W, b = self.layers[layer].params()
            dW, db = self.layers[layer].grad_params()
            W -= learning_rate * dW
            b -= learning_rate * db
        # </MJ>
        pass

    def train(self, X, Y, X_valid, Y_valid, learning_rate=0.1, max_epochs=100, batch_siz
              descent_type="sgd", y_one_hot=True):

        # <MJ>
        """ Train network on the given data. """
        n_samples = X.shape[0]
        n_batches = n_samples // batch_size
        if y_one_hot:
            Y_train = one_hot(Y)
            Y_valid_hot = one_hot(Y_valid)
        else:
            Y_train = Y
        print("... starting training")
        train_loss = np.zeros(max_epochs + 1)
        valid_loss = np.zeros(max_epochs + 1)
        for e in range(max_epochs + 1):
            if descent_type == "sgd":
                self.sgd_epoch(X, Y_train, learning_rate, batch_size)
            elif descent_type == "gd":
                self.gd_epoch(X, Y_train, learning_rate)
            else:
                raise NotImplementedError("Unknown gradient descent type {}".format(desc

            # Output error on the training data
            train_loss[e] = self._loss(X, Y_train)
            train_error = self.classification_error(X, Y)
            print('epoch {:.4f}, loss {:.4f}, train error {:.4f}'.format(e, train_loss[e

            # Output error on the validation data
            valid_loss[e] = self._loss(X_valid, Y_valid_hot)
            valid_error = self.classification_error(X_valid, Y_valid)
            print('epoch {:.4f}, loss {:.4f}, valid error {:.4f}'.format(e, valid_loss[e

        return train_loss, valid_loss
        # </MJ>

    def check_gradients(self, X, Y):
        """ Helper function to test the parameter gradients for
        correctness. """
        for l, layer in enumerate(self.layers):
            if isinstance(layer, Parameterized):
                print('checking gradient for layer {}'.format(l))
                for p, param in enumerate(layer.params()):
                    # we iterate through all parameters
                    param_shape = param.shape
                    # define functions for conveniently swapping
                    # out parameters of this specific layer and
                    # computing loss and gradient with these
                    # changed parametrs
                    def output_given_params(param_new):
                        """ A function that will compute the output
                            of the network given a set of parameters
                        """
                        # copy provided parameters
                        param[:] = np.reshape(param_new, param_shape)
                        # return computed loss
                        return self._loss(X, Y)

                    def grad_given_params(param_new):
                        """A function that will compute the gradient
                           of the network given a set of parameters
                        """
                        # copy provided parameters
                        param[:] = np.reshape(param_new, param_shape)
                        # Forward propagation through the net
                        Y_pred = self.predict(X)
                        # Backpropagation of partial derivatives
                        self.backpropagate(Y, Y_pred, upto=l)
                        # return the computed gradient
                        return np.ravel(self.layers[l].grad_params()[p])

                    # let the initial parameters be the ones that
                    # are currently placed in the network and flatten them
                    # to a vector for convenient comparisons, printing etc.
                param_init = np.ravel(np.copy(param))

                # To debug you network's gradients use scipys
                # gradient checking!
                epsilon = 1e-5
                import scipy.optimize
                err = scipy.optimize.check_grad(output_given_params,
                      grad_given_params, param_init)
                print('diff scipy {:.2e}'.format(err))
                assert(err < epsilon)

                # reset the parameters to their initial values
                param[:] = np.reshape(param_init, param_shape)
```

5 Gradient Checking

After implementing everything it is always a good idea to setup some layers and perform gradient checking on random data. **Note** that this is only an example! It is not a useful network architecture ;). We also expect you to play around with this to test all your implemented components.

```
In [5]: input_shape = (5, 10)
        n_labels = 6
        layers = [InputLayer(input_shape)]

        layers.append(FullyConnectedLayer(
                        layers[-1],
                        num_units=15,
                        init_stddev=0.1,
                        activation_fun=Activation('relu')
        ))
        layers.append(FullyConnectedLayer(
                        layers[-1],
                        num_units=6,
                        init_stddev=0.1,
                        activation_fun=Activation('tanh')
        ))
        layers.append(FullyConnectedLayer(
                        layers[-1],
                        num_units=n_labels,
                        init_stddev=0.1,
                        activation_fun=Activation('relu')
        ))
        layers.append(SoftmaxOutput(layers[-1]))
        nn = NeuralNetwork(layers)
```

```
In [6]: # create random data
        X = np.random.normal(size=input_shape)
        # and random labels
        Y = np.zeros((input_shape[0], n_labels))
        for i in range(Y.shape[0]):
            idx = np.random.randint(n_labels)
            Y[i, idx] = 1.
```

```
In [7]: nn.check_gradients(X, Y)
```

```
checking gradient for layer 1
diff scipy 2.01e-07
diff scipy 6.85e-08
checking gradient for layer 2
diff scipy 1.54e-07
diff scipy 2.61e-08
checking gradient for layer 3
diff scipy 8.40e-08
diff scipy 4.22e-08
```

6 Training on MNIST

Finally we can let our network run on the MNIST dataset!

First load the data and reshape it.

```
In [8]: # load
        Dtrain, Dval, Dtest = mnist()
        X_train, y_train = Dtrain
        X_valid, y_valid = Dval
        X_test, y_test = Dtest
```

```
... loading data
... done loading data
```

Dtrain contains 50k images which are of size 28 x 28 pixels. Hence:

```
In [9]: print("X_train shape: {}".format(np.shape(X_train)))
        print("y_train shape: {}".format(np.shape(y_train)))
```

```
X_train shape: (50000, 1, 28, 28)
y_train shape: (50000,)
```

y_train will automatically be converted in the *train()* function to one_hot encoding.

But we need to reshape X_train, as our Network expects flat vectors of size 28*28 as input!

```
In [10]: X_train = X_train.reshape(X_train.shape[0], -1)
         print("Reshaped X_train size: {}".format(X_train.shape))
         X_valid = X_valid.reshape((X_valid.shape[0], -1))
         print("Reshaped X_valid size: {}".format(X_valid.shape))
         X_test = X_test.reshape((X_test.shape[0], -1))
         print("Reshaped X_test size: {}".format(X_test.shape))
```

```
Reshaped X_train size: (50000, 784)
Reshaped X_valid size: (10000, 784)
Reshaped X_test size: (10000, 784)
```

Ah, much better ;-)!

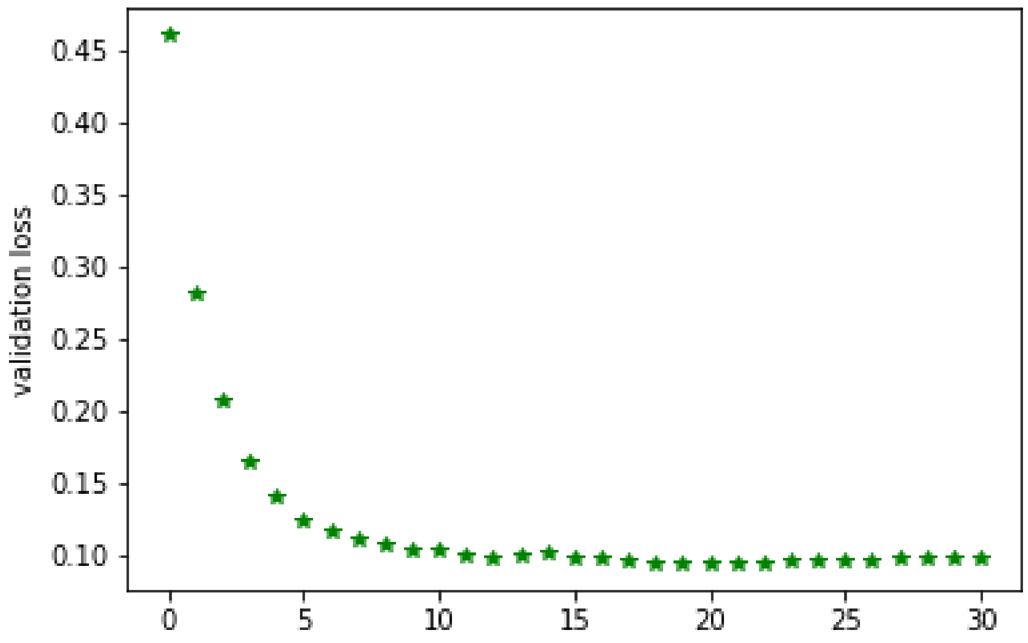
Now we can finally really start training a Network!

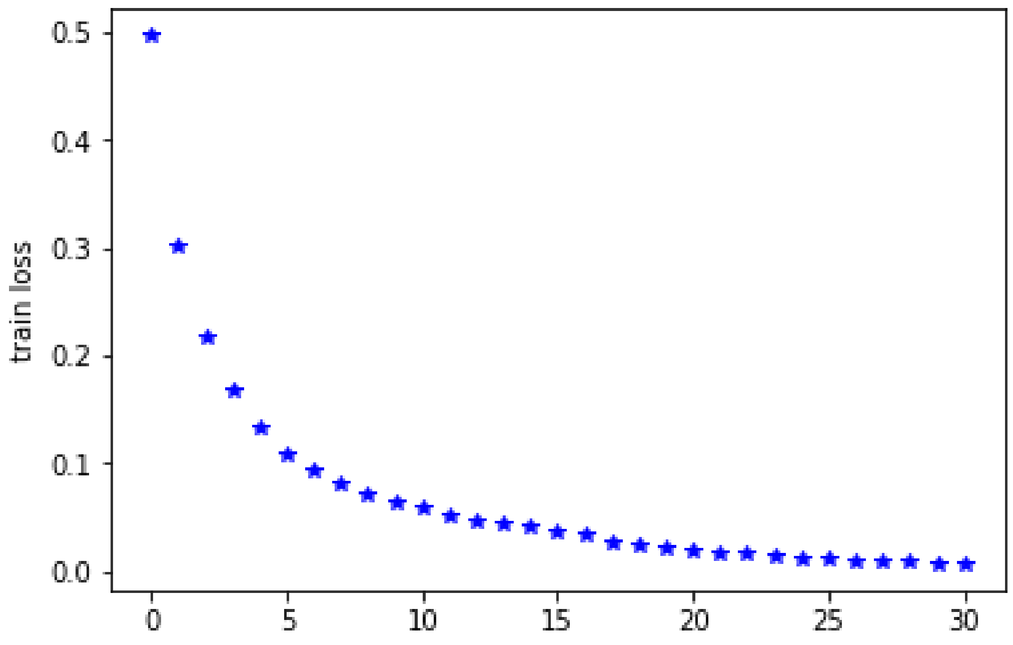
I pre-defined a small Network for you below. Again This is not really a good default and will not produce state of the art results. Please play around with this a bit. See how different activation functions and training procedures (gd / sgd) affect the result.

```
In [11]: import time

         # Setup a small MLP / Neural Network
         # we can set the first shape to None here to indicate that
         # we will input a variable number inputs to the network
         input_shape = (None, 28*28)
         layers = [InputLayer(input_shape)]
         layers.append(FullyConnectedLayer(
                         layers[-1],
                         num_units=80,
                         init_stddev=0.01,
                         activation_fun=Activation('relu')
         ))
         # layers.append(FullyConnectedLayer(
         #                 layers[-1],
         #                 num_units=70,
         #                 init_stddev=0.01,
         #                 activation_fun=Activation('relu')
         # ))
         layers.append(FullyConnectedLayer(
                         layers[-1],
                         num_units=70,
                         init_stddev=0.01,
                         # last layer has no nonlinearity
                         # (softmax will be applied in the output layer)
                         activation_fun=Activation('tanh')
         ))
         layers.append(FullyConnectedLayer(
                         layers[-1],
                         num_units=10,
                         init_stddev=0.01,
                         # last layer has no nonlinearity
                         # (softmax will be applied in the output layer)
                         activation_fun=None
         ))
         layers.append(SoftmaxOutput(layers[-1]))
```

```
nn = NeuralNetwork(layers)
# Train neural network
t0 = time.time()
max_epochs = 30
train_loss, valid_loss = nn.train(X_train, y_train, X_valid, y_valid, learning_rate=0.1
                                  max_epochs=max_epochs, batch_size=64, y_one_hot=True
t1 = time.time()
print('Duration: {:.1f}s'.format(t1-t0))

error = nn.classification_error(X_test, y_test) * 100
print('Error: {:.3f}' .format(error))

plt.plot(range(max_epochs + 1), valid_loss, 'g*')
plt.ylabel('validation loss')
plt.show()
plt.plot(range(max_epochs + 1), train_loss, 'b*')
plt.ylabel('train loss')
plt.show()
```

```
... starting training
epoch 0.0000, loss 0.4962, train error 0.1484
epoch 0.0000, loss 0.4601, valid error 0.1394
epoch 1.0000, loss 0.3025, train error 0.0921
epoch 1.0000, loss 0.2803, valid error 0.0838
epoch 2.0000, loss 0.2166, train error 0.0663
epoch 2.0000, loss 0.2063, valid error 0.0625
epoch 3.0000, loss 0.1671, train error 0.0520
epoch 3.0000, loss 0.1649, valid error 0.0501
epoch 4.0000, loss 0.1337, train error 0.0418
epoch 4.0000, loss 0.1406, valid error 0.0434
epoch 5.0000, loss 0.1093, train error 0.0338
epoch 5.0000, loss 0.1243, valid error 0.0372
epoch 6.0000, loss 0.0938, train error 0.0290
epoch 6.0000, loss 0.1161, valid error 0.0360
epoch 7.0000, loss 0.0807, train error 0.0249
epoch 7.0000, loss 0.1097, valid error 0.0338
epoch 8.0000, loss 0.0719, train error 0.0225
epoch 8.0000, loss 0.1067, valid error 0.0326
epoch 9.0000, loss 0.0636, train error 0.0202
epoch 9.0000, loss 0.1032, valid error 0.0315
epoch 10.0000, loss 0.0589, train error 0.0189
epoch 10.0000, loss 0.1026, valid error 0.0307
epoch 11.0000, loss 0.0528, train error 0.0170
epoch 11.0000, loss 0.0999, valid error 0.0301
epoch 12.0000, loss 0.0468, train error 0.0151
epoch 12.0000, loss 0.0979, valid error 0.0290
epoch 13.0000, loss 0.0444, train error 0.0140
epoch 13.0000, loss 0.0986, valid error 0.0285
epoch 14.0000, loss 0.0429, train error 0.0134
epoch 14.0000, loss 0.1004, valid error 0.0289
epoch 15.0000, loss 0.0378, train error 0.0121
epoch 15.0000, loss 0.0983, valid error 0.0286
epoch 16.0000, loss 0.0332, train error 0.0107
epoch 16.0000, loss 0.0970, valid error 0.0277
epoch 17.0000, loss 0.0280, train error 0.0085
epoch 17.0000, loss 0.0951, valid error 0.0265
epoch 18.0000, loss 0.0246, train error 0.0074
epoch 18.0000, loss 0.0940, valid error 0.0259
epoch 19.0000, loss 0.0221, train error 0.0066
epoch 19.0000, loss 0.0939, valid error 0.0256
epoch 20.0000, loss 0.0205, train error 0.0060
epoch 20.0000, loss 0.0944, valid error 0.0253
epoch 21.0000, loss 0.0183, train error 0.0053
epoch 21.0000, loss 0.0946, valid error 0.0256
epoch 22.0000, loss 0.0160, train error 0.0044
epoch 22.0000, loss 0.0943, valid error 0.0249
epoch 23.0000, loss 0.0146, train error 0.0039
epoch 23.0000, loss 0.0948, valid error 0.0243
epoch 24.0000, loss 0.0130, train error 0.0033
epoch 24.0000, loss 0.0953, valid error 0.0245
epoch 25.0000, loss 0.0120, train error 0.0028
epoch 25.0000, loss 0.0962, valid error 0.0250
epoch 26.0000, loss 0.0106, train error 0.0023
epoch 26.0000, loss 0.0965, valid error 0.0249
epoch 27.0000, loss 0.0097, train error 0.0020
epoch 27.0000, loss 0.0968, valid error 0.0248
epoch 28.0000, loss 0.0089, train error 0.0017
epoch 28.0000, loss 0.0966, valid error 0.0246
epoch 29.0000, loss 0.0080, train error 0.0016
epoch 29.0000, loss 0.0972, valid error 0.0248
epoch 30.0000, loss 0.0075, train error 0.0014
epoch 30.0000, loss 0.0975, valid error 0.0247
Duration: 263.8s
Error: 2.390
```


validation loss
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
5
10
15
20
25
30


train loss
0.5
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20
25
30

7 Figure out a reasonable Network that achieves good performance

As the last part of this task, setup a network that works well and gets reasonable accuracy, say ~ 1-3 percent error on the **validation set**. Train this network on the complete data and compute the **test error**.

Visualize the validation loss and training loss for each iteration in a plot, e.g. using matplotlib lab = np.array([1, 2, 0]) one_hot(lab)