Copy_of_ML_assignment

December 11, 2018

0.1 Member

- Lê Quang Trình - 1513687
- Vn Hu Quc - 1512723

1 Numpy

1.1 Model Implement

```
In [0]: import numpy as np
        from tensorflow import keras
        import matplotlib.pyplot as plt
        from tqdm import tqdm
        import time

        class Layer:
            """ Base class for neural network.
                Two abstract funtion forward and backward to help inference and
                backprogation
            """

            def __init__(self):
                self.parameters = {}
                self.has_weights = False

            def forward(self, x, is_training=True):
                pass

            def backward(self, dy):
                pass


        def get_im2col_indices(x_shape, field_height, field_width,
                               padding=1, stride=(1, 1)):
            """ An implementation of im2col based on some fancy indexing

            Args:
```

```python
        x_shape : [B, Cin, H, W]
        field_height: kH
        field_width: kW

    Return: [B*OH*OW, kH*kW*Cin]
    """

    # First figure out what the size of the output should be
    _, C, H, W = x_shape
    # assert (H + 2 * padding - field_height) % stride[0] == 0
    # assert (W + 2 * padding - field_height) % stride[1] == 0
    out_height = (H + 2 * padding - field_height) // stride[0] + 1
    out_width = (W + 2 * padding - field_width) // stride[1] + 1
    i0 = np.repeat(np.arange(field_height), field_width)
    i0 = np.tile(i0, C)
    i1 = stride[0] * np.repeat(np.arange(out_height), out_width)
    j0 = np.tile(np.arange(field_width), field_height * C)
    j1 = stride[1] * np.tile(np.arange(out_width), out_height)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)

    k = np.repeat(np.arange(C), field_height * field_width).reshape(-1, 1)

    return (k, i, j)


def im2col_indices(x, filter=(2, 2), padding=1, stride=(1, 1)):
    """ An implementation of im2col based on some fancy indexing

    Args:
        x : [B, H, W, Cin]
        field_height: kH
        field_width: kW

    Return: [B*OH*OW, kH*kW*Cin]
    """

    # x => [B, Cin, H, W]
    x = x.transpose(0, 3, 1, 2)

    # Zero-pad the input
    p = padding
    x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

    k, i, j = get_im2col_indices(x.shape, filter[0], filter[1], padding,
                                 stride)
    # cols => [B, kH*kW*Cin, OH*OW]
    cols = x_padded[:, k, i, j]
```

```python
    C = x.shape[1]

    # cols => [B*OH*OW, kH*kW*Cin]
    cols = cols.transpose(0, 2, 1).reshape(-1, filter[0] * filter[1] * C)
    return cols


def col2im_indices(cols, x_shape, filter=(2, 2), padding=0,
                   stride=(1,1) ):
    """ An implementation of col2im based on fancy indexing and np.add.at

    Args:
        cols: [B*OH*OW, kH*kW*Cin]
        x_shape: Shape of initial input (B, H, W, Cin)
        filter: Shape of filter (kH, kW)

    Returns: [B, H, W, Cin]


    """
    N, H, W, C = x_shape
    H_padded, W_padded = H + 2 * padding, W + 2 * padding
    x_padded = np.zeros((N, C, H_padded, W_padded), dtype=cols.dtype)
    k, i, j = get_im2col_indices((N, C, H, W), filter[0], filter[1], padding,
                                 stride)
    # cols => [B*OH*OW, kH*kW*Cin]
    # cols_reshaped => [B, OH*OW, kH*kW*Cin]
    cols_reshaped = cols.reshape(N, -1, C * filter[0] * filter[1])

    # [B, C * kH * kW, OH*OW]
    cols_reshaped = cols_reshaped.transpose(0, 2, 1)
    np.add.at(x_padded, (slice(None), k, i, j), cols_reshaped)
    if padding == 0:
        return x_padded
    return x_padded[:, :, padding:-padding,
                    padding:-padding].tranpose(0, 2, 3, 1)


class Conv2d(Layer):
    """ An implementation of Convolution 2D"""

    def __init__(self, input_shape=(-1, 3, 3, 1),
                 filter=(1, 2, 2, 1), stride=(1, 1),
                 padding='VALID', activation='relu'):
        super(Conv2d, self).__init__()
        self.has_weights = True
        self.input_shape = input_shape
```

```python
        self.filter = filter
        self.stride = stride
        self.cache = {}
        self.w_shape = []
        self.padding = 0
        if activation == 'relu':
            self.activation = Relu()
        else:
            self.activation = None
        self.out_height = (input_shape[1] + 2 * self.padding -
                           filter[1]) // stride[0] + 1
        self.out_width = (input_shape[2] + 2 * self.padding -
                          filter[2]) // stride[1] + 1

        self.linear_in = filter[1] * filter[2] * filter[3]
        self.linear_out = filter[0]
        self._linear = Linear(self.linear_in, self.linear_out)

    def forward(self, x, is_training=True):
        """Implement forward for Conv2d

        x: [B, H, W, Cin]
        """
        B, _, _, _ = x.shape
        self.input_shape = x.shape

        # x2row =>  [B*H*W, Cin]
        x2row = im2col_indices(
            x, filter=(self.filter[1], self.filter[2]),
            padding=self.padding, stride=self.stride)
        assert x2row.shape == (B * self.out_height *
                               self.out_height, self.linear_in)

        # out [B*OH*OW, Cout]
        out = self._linear.forward(x2row)
        if self.activation:
            out = self.activation.forward(out)
        # out [B, OH, OW, Cout]
        return out.reshape((B, self.out_height, self.out_width, self.linear_out))

    def backward(self, dy):
        """Implement forward for Conv2d

        dy: [B, H, W, Cout]
        """
        # dy => [B, OH, OW, Cout]
        dy = dy.reshape(-1, self.linear_out)
        # dx => [B]
```

```
        if self.activation:
            dy = self.activation.backward(dy)
        dx = self._linear.backward(dy)

        # cols => [B* OH*OW, kH*kW*Cin] =>
        #          [kH*kW*Cin ,OH*OW, B] => [kH*kW*Cin ,OH*OW*B]

        dx = col2im_indices(cols=dx,
                            x_shape=self.input_shape,
                            filter=(self.filter[1], self.filter[2]),
                            padding=self.padding,
                            stride=self.stride)

        return dx

    def apply_grads(self, learning_rate=0.01, l2_penalty=1e-4):
        self._linear.apply_grads(learning_rate=0.01, l2_penalty=l2_penalty)


class Flatten(Layer):
    def __init__(self, input_shape):
        super(Flatten, self).__init__()
        self.input_shape = input_shape
        self.out_num = int(np.prod(np.array(self.input_shape[1:])))

    def forward(self, x, is_training=True):
        return x.reshape(-1, self.out_num)


    def backward(self, dy):

        return dy.reshape(self.input_shape)


class Linear(Layer):
    """ Implement fully connected dense layer.

    Recieve input [batch_size, num_in] and produce output [batch_size, num_out]
    folow expression: y = activation(x.w + b)

    Args:
      input_shape: (int) length input
      num_units: (int) length of output
      is_training: (bool) Determine whether is trainging or not
      activation: Default is not using activation
    """

    def __init__(self, num_in, num_out, activation=None):
```

```
        super(Linear, self).__init__()
        self.cache = {}
        self.grads = {}
        self.bias_shape = [num_out]
        self.weight_shape = [num_in, num_out]
        self.has_weights = True
        self.parameters['W'] = self.initiate_vars(self.weight_shape)
        self.parameters['b'] = self.initiate_vars_zero(self.bias_shape)
        self.name = 'Linear'

    def initiate_vars(self, shape, distribution=None):
        if distribution != None:
            raise ValueError(
                'Not implement distribution for initiating variable')

        return np.random.randn(shape[0], shape[1]) * 1e-3

    def initiate_vars_zero(self, shape):
        return np.zeros(shape)

    def forward(self, x, is_training=True):
        # check whether data has valid shape or not
        if is_training:
            self.cache['x'] = x.copy()
            self.batch_size = x.shape[0]
        y = np.dot(x, self.parameters['W']) + self.parameters['b']

        return y

    def backward(self, dy):
        self.grads['db'] = np.sum(dy, axis=0)
        self.grads['dW'] = np.dot(self.cache['x'].T, dy)
        dx = np.dot(dy, self.parameters['W'].T)

        return dx

    def apply_grads(self, learning_rate=0.01, l2_penalty=1e-4):
        self.parameters['W'] -= learning_rate * \
            (self.grads['dW'] + l2_penalty * self.parameters['W'])
        self.parameters['b'] -= learning_rate * \
            (self.grads['db'] + l2_penalty * self.parameters['b'])


class Relu(Layer):
    """ Implement Relu activation function:
    y = x with x >= 0
    y = 0 with x<0
    """
```

```python
    def __init__(self):

        super(Relu, self).__init__()
        self.cache = {}
        self.has_weights = False
        self.name = 'Relu'

    def forward(self, x, is_training=True):
        if(is_training):
            self.cache['x'] = x.copy()
        y = x
        y[y < 0] = 0
        return y

    def backward(self, dy):
        dy[self.cache['x'] <= 0] = 0
        return dy


class Softmax(Layer):
    """ Implement Softmax activation function

    Recieve input with shape [batch_size, num_score] and produce output folowing
    expression:  y = e^score / sum(e^score)

    Args:
    """

    def __init__(self):
        super(Softmax, self).__init__()
        self.cache = {}
        self.has_weights = False
        self.name = 'Softmax'

    def forward(self, data, is_training=True):
        if(len(data.shape) != 2):
            raise ValueError(
                'data have shape is not compatible. Expect [batch_size, nums_score]')
        logits = np.exp(data - np.amax(data, axis=1, keepdims=True))
        logits = logits / np.sum(logits, axis=1, keepdims=True)
        if is_training:
            self.cache['logits'] = np.copy(logits)

        return logits

    def backward(self, dy):
        if(len(dy.shape) != 2):
```

```
            raise ValueError(
                'data have shape is not compatible. Expect [batch_size, nums_score]')
        num_units = dy.shape[-1]

        # [batch_size, num_units, 1] . [batch_size, 1, num_units]
        # = [batch_size, num_units, num_units]
        # Represent of matrix ds:
        #   [ds1/dx1 ds1/dx2 ... ds1/dxN]
        #   [ds2/dx1 ds2/dx2 ... ds2/dxN]
        #   [  ...            ...    ... ]
        #   [dsN/dx1 dsN/dx2 ... dsN/dxN]
        # with ds_i/dx_j = S_i(1 - S_j)  , with i==j
        #                = -S_i*S_j      , with i!=j
        ds = -np.matmul(np.expand_dims(self.cache['logits'], axis=-1),
                        np.expand_dims(self.cache['logits'], axis=1))
        ds[:, np.arange(num_units), np.arange(
            num_units)] += self.cache['logits']

        dx = np.matmul(np.expand_dims(dy, axis=1), ds)
        return np.squeeze(dx, axis=1)
        # return dy


class CELoss():
    """ Cross Entropy Loss

    Loss function: L = sum(y.log(s)) / batch_size ,
        y is labels,
        s is predict
    Derivative of Loss: dL/ds_i= y_i/s_i
    """

    def __init__(self):
        self.cache = {}
        self.has_weights = False
        self.eps = 1e-8
        # super(CELoss, self).__init__()

    def compute_loss(self, logits, labels, is_training=True):
        logits = np.clip(logits, self.eps, 1. - self.eps)
        # logits => [batch_size, num_units]
        # labels => [batch_size, num_units]
        if is_training:
            self.cache['labels'] = labels.copy()
            self.cache['logits'] = logits.copy()
        self.batch_size = logits.shape[0]
        loss = - np.sum(labels * np.log(logits)) / self.batch_size
        return loss
```

```python
    def compute_derivation(self, logits, labels):
        # => [batch_size, num_units]

        return - self.cache['labels'] / (self.cache['logits'] * self.batch_size)

class MaxPooling2D(Layer):

    def __init__(self, pool_size=(2, 2), stride=(1, 1), padding=0):
        self.cache = {}
        self.pool_size = pool_size
        self.stride = stride
        self.pading = padding
        self.has_weights = False

    def forward(self, x, is_training=True):
        N, H, W, C = x.shape
        pool_height, pool_width = self.pool_size
        stride_height, stride_width = self.stride

        out_height = (H - pool_height) // stride_height + 1
        out_width = (W - pool_width) // stride_width + 1

        x_split = x.transpose(0, 3, 1, 2).reshape(N * C, H, W, 1)
        # (out_height*out_width*N*C, H*W*1)
        x_cols = im2col_indices(x_split, self.pool_size,
                                padding=0, stride=self.stride)
        x_cols_argmax = np.argmax(x_cols, axis=1)
         # (out_height*out_width*N*C)
        x_cols_max = x_cols[np.arange(x_cols.shape[0]), x_cols_argmax]
         # (N, out_height,out_width, C)
        out = x_cols_max.reshape(out_height,
                                 out_width, N, C).transpose(2, 0, 1, 3)

        if is_training:
            self.cache['x'] = x.copy()
            self.cache['x_cols'] = x_cols
            self.cache['x_cols_argmax'] = x_cols_argmax

        return out # (N, out_height,out_width, C)

    def backward(self, dout):
        x, x_cols,  = self.cache['x'], self.cache['x_cols']
        x_cols_argmax =self.cache['x_cols_argmax']
        N, H, W, C = x.shape
        #dout: # (N, out_height,out_width, C)
        # (out_height*out_width*N*C)
        dout_reshaped = dout.transpose(1, 2, 0, 3).flatten()
```

```python
        # (out_height*out_width*N*C, H*W*1)
        dx_cols = np.zeros_like(x_cols)
        dx_cols[np.arange(dx_cols.shape[0]), x_cols_argmax] = dout_reshaped

        dx = col2im_indices(dx_cols, (N * C, H, W, 1),
                            self.pool_size, padding=0, stride=self.stride)
        #[N, H, W, C]
        dx = dx.reshape((N,C,H,W)).transpose(0,2,3,1)
        return dx

class MaxPooling2DNaive(Layer):

    def __init__(self, pool_size=(2, 2), strides=(1, 1), padding=0):
        self.cache = {}
        self.pool_size = pool_size
        self.strides = strides
        self.pading = padding
        self.has_weights = False

    def forward(self, x, is_training=True):
        N, H, W, C = x.shape
        HH, WW = self.pool_size
        stride_height, stride_width = self.strides

        out_height = (H - HH) // stride_height + 1
        out_width = (W - WW) // stride_width + 1

        out = np.zeros((N, out_height, out_width, C))

        for j in range(out_height):
            h_start = j*stride_height
            for k in range(out_width):
                w_start = k*stride_width
                out[:, j, k, :] = (
                    x[:, h_start:(h_start+HH),
                      w_start:(w_start+WW), :].max(axis=(1, 2)))

        if is_training:
            self.cache['x'] = x.copy()

        return out

    def backward(self, dout):
        dx = None
        x = self.cache['x']
        # Get dimensions
        N, H, W, C = x.shape
        HH, WW = self.pool_size
```

```python
        stride_height, stride_width = self.strides

        # Compute dimension filters
        out_height = (H-HH) // stride_height + 1
        out_width = (W-WW) // stride_width + 1

        # Initialize tensor for dx
        dx = np.zeros_like(x)

        # Backpropagate dout on x
        for i in range(N):
            for z in range(C):
                for j in range(out_height):
                    h_start = j*stride_height
                    for k in range(out_width):
                        w_start = k*stride_width
                        dpatch = np.zeros((HH, WW))
                        input_patch = x[i, h_start:(h_start+HH),
                                        w_start:(w_start+WW), z]
                        idxs_max = np.where(input_patch == input_patch.max())
                        dpatch[idxs_max[0], idxs_max[1]] = dout[i, j, k, z]
                        dx[i, h_start:(h_start+HH),
                           w_start:(w_start+WW), z] += dpatch

        return dx

class DropOut(Layer):
    def __init__(self, prob,  seed = None, distribution='uniform'):
        super(DropOut, self).__init__()
        self.has_weights= False
        self.seed = seed
        self.prob = prob
        self.distribution = distribution
        self.cache = {}

    def forward(self, x, is_training=False):
        if is_training:
            if self.distribution == 'uniform':
                mask = (np.random.rand(*x.shape)<self.prob)/self.prob
            self.cache['mask'] = mask
            out = x*mask
        else:
            out = x
        return out

    def backward(self, dy):
        return self.cache['mask'] * dy
```

```python
class Model:
    def __init__(self, *model, **kwargs):
        self.model = model
        self.num_classes = 0
        self.batch_size = 0
        self.loss = None
        self.optimizer = None
        self.name = kwargs['name'] if 'name' in kwargs else None
        self.history = []

    def add(self, layer):
        self.model.append()

    def set_batch_size(self, batch_size):
        self.batch_size = batch_size

    def set_num_classes(self, num_classes):
        self.num_classes = num_classes

    def set_loss(self, loss):
        self.loss = loss

    def get_batches(self, data, labels, batch_size=256, shuffle=True):
        N = data.shape[0]
        num_batches = N // batch_size
        if(shuffle):
            rand_idx = np.random.permutation(data.shape[0])
            data = data[rand_idx]
            labels = labels[rand_idx]
        for i in np.arange(num_batches):
            yield (data[i*batch_size:(i+1) * batch_size],
                   labels[i*batch_size:(i+1) * batch_size])
        if N % batch_size != 0 and num_batches != 0:
            yield (data[batch_size*num_batches:],
                   labels[batch_size*num_batches:])

    def train(self, train_data, train_labels,
              eval_data, eval_labels,
              batch_size=1024, epochs=50,
              display_after=50, eval_after = 250,
              learning_rate=0.01,
              l2_penalty=1e-4,
              learning_rate_decay=0.95):
        if self.loss is None:
            raise RuntimeError("Set loss first using 'model.set_loss(<loss>)'")
        self.set_batch_size(batch_size)
        self.set_num_classes(train_labels.shape[1])
```

```
        iter = 0
        for epoch in range(epochs):

            print('Running Epoch:', epoch + 1)

            for i, (x_batch, y_batch) in enumerate(
                self.get_batches(train_data, train_labels,
                                 batch_size=batch_size)):
                batch_preds = x_batch.copy()
                for  layer in self.model:
                    batch_preds = layer.forward(batch_preds, is_training=True)

                loss = self.loss.compute_loss(
                    logits=batch_preds, labels=y_batch)
                dA = self.loss.compute_derivation(
                    logits=batch_preds, labels=y_batch)

                for layer in reversed(self.model):
                    dA = layer.backward(dA)

                for layer in self.model:
                    if layer.has_weights:
                        layer.apply_grads(
                            learning_rate=learning_rate, l2_penalty=l2_penalty)
                iter += 1
                if iter % display_after == 0:
                    train_acc = self.evaluate(x_batch, y_batch)
                    eval_acc = self.evaluate(eval_data, eval_labels)
                    print('Step {}, loss: {}, train_acc: {}, eval_acc: {}'
                          .format(
                        iter, loss, train_acc, eval_acc))
                    self.history.append((iter, loss, train_acc, eval_acc))
            learning_rate *= learning_rate_decay


    def predict(self, data):
        batch_preds = data.copy()
        for layer in self.model:
            batch_preds = layer.forward(batch_preds)
        return batch_preds

    def evaluate(self, data, labels):
        predictions = self.predict(data)
        if(predictions.shape != labels.shape):
            raise ValueError('prediction shape does not match labels shape')
        return np.mean(np.argmax(labels, axis=1) == np.argmax(predictions,
                                                              axis=1))
```

1.2 Pre-processing

```
In [2]: num_classes = 10
        class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                       'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

        fashion_mnist = keras.datasets.fashion_mnist
        (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

        train_images = train_images.reshape((-1, 28, 28, 1))
        test_images = test_images.reshape((-1, 28, 28, 1))
        # train_images = train_images.reshape((-1, 784))
        # test_images = test_images.reshape((-1, 784))


        labels = np.zeros((train_labels.shape[0], 10))
        labels[np.arange(train_labels.shape[0]), train_labels] = 1
        train_labels = labels

        labels = np.zeros((test_labels.shape[0], 10))
        labels[np.arange(test_labels.shape[0]), test_labels] = 1
        test_labels = labels


        batch_size = train_images.shape[0]
        num_train = batch_size*9 // 10

        eval_images = train_images[num_train:]
        eval_labels = train_labels[num_train:]
        train_images = train_images[:num_train]
        train_labels = train_labels[:num_train]
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-
32768/29515 [==================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-i
8192/5148 [==============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-i
4423680/4422102 [==============================] - 0s 0us/step
```

1.3 New model

```
In [0]: model = Model(
                    Conv2d(input_shape=(-1, 28, 28, 1),
                           filter=(10, 2, 2, 1), activation='relu'),
                    DropOut(0.95),
                    Flatten(input_shape= (-1, 27, 27, 10)),
```

```
                    Linear(num_in=27*27*10, num_out=10),
                    Softmax())




model.set_loss(CELoss())
```

1.4 Train

```
In [4]: t1 = time.time()
        model.train(train_images,
                    train_labels, eval_images,
                    eval_labels, learning_rate=0.00001,
                    l2_penalty=0.,
                    epochs=10,
                    batch_size=200,
                    display_after = 50)
        print('Time: ',time.time()-t1, 's')
```

```
Running Epoch: 1
Step 50, loss: 0.9300093215291579, train_acc: 0.66, eval_acc: 0.7046666666666667
Step 100, loss: 0.8970342512331199, train_acc: 0.705, eval_acc: 0.7361666666666666
Step 150, loss: 0.6868037469132008, train_acc: 0.785, eval_acc: 0.7285
Step 200, loss: 0.7520530982652825, train_acc: 0.725, eval_acc: 0.7448333333333333
Step 250, loss: 0.7740931748899686, train_acc: 0.76, eval_acc: 0.723
Running Epoch: 2
Step 300, loss: 0.6566478547456243, train_acc: 0.78, eval_acc: 0.765
Step 350, loss: 0.6398334024404427, train_acc: 0.825, eval_acc: 0.789
Step 400, loss: 0.5494713194785221, train_acc: 0.815, eval_acc: 0.7903333333333333
Step 450, loss: 0.6477460438550806, train_acc: 0.79, eval_acc: 0.803
Step 500, loss: 0.54410309913211, train_acc: 0.815, eval_acc: 0.7965
Running Epoch: 3
Step 550, loss: 0.5241171415574276, train_acc: 0.86, eval_acc: 0.8065
Step 600, loss: 0.5821116185259657, train_acc: 0.805, eval_acc: 0.8143333333333334
Step 650, loss: 0.7323242281477833, train_acc: 0.76, eval_acc: 0.7928333333333333
Step 700, loss: 0.6006108870732819, train_acc: 0.82, eval_acc: 0.8156666666666667
Step 750, loss: 0.6151832836744668, train_acc: 0.825, eval_acc: 0.8121666666666667
Step 800, loss: 0.4918166789282499, train_acc: 0.86, eval_acc: 0.8158333333333333
Running Epoch: 4
Step 850, loss: 0.506375619466815, train_acc: 0.81, eval_acc: 0.8143333333333334
Step 900, loss: 0.5522972907121728, train_acc: 0.845, eval_acc: 0.818
Step 950, loss: 0.48681703271302906, train_acc: 0.825, eval_acc: 0.8143333333333334
Step 1000, loss: 0.5858750078085904, train_acc: 0.855, eval_acc: 0.8235
Step 1050, loss: 0.5395318595114496, train_acc: 0.83, eval_acc: 0.823
Running Epoch: 5
Step 1100, loss: 0.5044151466451926, train_acc: 0.87, eval_acc: 0.8201666666666667
Step 1150, loss: 0.40986890673114323, train_acc: 0.87, eval_acc: 0.8205
```

```
Step 1200, loss: 0.5206328914203676, train_acc: 0.815, eval_acc: 0.8286666666666667
Step 1250, loss: 0.5696881708065145, train_acc: 0.81, eval_acc: 0.816
Step 1300, loss: 0.5796035053492652, train_acc: 0.84, eval_acc: 0.825
Step 1350, loss: 0.6121621540321636, train_acc: 0.8, eval_acc: 0.8218333333333333
Running Epoch: 6
Step 1400, loss: 0.41439834160292605, train_acc: 0.865, eval_acc: 0.8281666666666667
Step 1450, loss: 0.6067887503948669, train_acc: 0.79, eval_acc: 0.8225
Step 1500, loss: 0.4180961628942228, train_acc: 0.87, eval_acc: 0.8258333333333333
Step 1550, loss: 0.630311522798698, train_acc: 0.79, eval_acc: 0.8265
Step 1600, loss: 0.4883724863359235, train_acc: 0.85, eval_acc: 0.8293333333333334
Running Epoch: 7
Step 1650, loss: 0.5545347121110309, train_acc: 0.87, eval_acc: 0.8188333333333333
Step 1700, loss: 0.5311186256872306, train_acc: 0.825, eval_acc: 0.8213333333333334
Step 1750, loss: 0.4086395409636491, train_acc: 0.9, eval_acc: 0.8316666666666667
Step 1800, loss: 0.4522383826119085, train_acc: 0.845, eval_acc: 0.8326666666666667
Step 1850, loss: 0.46670979352041925, train_acc: 0.865, eval_acc: 0.833
Running Epoch: 8
Step 1900, loss: 0.38497560374911316, train_acc: 0.875, eval_acc: 0.8258333333333333
Step 1950, loss: 0.5126364756155509, train_acc: 0.83, eval_acc: 0.8353333333333334
Step 2000, loss: 0.5223076097812728, train_acc: 0.875, eval_acc: 0.8378333333333333
Step 2050, loss: 0.5099602804680518, train_acc: 0.83, eval_acc: 0.8285
Step 2100, loss: 0.48107428961279225, train_acc: 0.855, eval_acc: 0.835
Step 2150, loss: 0.5590983508788504, train_acc: 0.86, eval_acc: 0.837
Running Epoch: 9
Step 2200, loss: 0.5547271538918556, train_acc: 0.83, eval_acc: 0.817
Step 2250, loss: 0.5707837016434391, train_acc: 0.835, eval_acc: 0.834
Step 2300, loss: 0.43484448801376163, train_acc: 0.895, eval_acc: 0.8371666666666666
Step 2350, loss: 0.36155389592297044, train_acc: 0.885, eval_acc: 0.8383333333333334
Step 2400, loss: 0.5016842652265212, train_acc: 0.83, eval_acc: 0.8375
Running Epoch: 10
Step 2450, loss: 0.5538336062338208, train_acc: 0.825, eval_acc: 0.8388333333333333
Step 2500, loss: 0.5000381809754143, train_acc: 0.87, eval_acc: 0.8381666666666666
Step 2550, loss: 0.48955165412820845, train_acc: 0.825, eval_acc: 0.8358333333333333
Step 2600, loss: 0.3665851016583236, train_acc: 0.875, eval_acc: 0.8338333333333333
Step 2650, loss: 0.47759089190708515, train_acc: 0.865, eval_acc: 0.8376666666666667
Step 2700, loss: 0.5403929823716147, train_acc: 0.83, eval_acc: 0.84
Time:  663.4128706455231 s
```

1.5 Evaluate on test set

In [5]:
```
batch = test_images
for layer in model.model:
  batch = layer.forward(batch)

test_lost = model.loss.compute_loss(batch, test_labels)

test_acc = model.evaluate(test_images, test_labels)
```

```
        print('Test loss: {}, acc: {}'.format(test_lost, test_acc))
```

```
Test loss: 0.5011314956349429, acc: 0.8258
```

2 CuPy

2.1 Install environment

```
In [6]: !pip install cupy-cuda92
```

```
Collecting cupy-cuda92
  Downloading https://files.pythonhosted.org/packages/09/fc/5fbb463b996a02c60f901e7dd2ba1d38e1
    100% || 271.3MB 85kB/s
Collecting fastrlock>=0.3 (from cupy-cuda92)
  Downloading https://files.pythonhosted.org/packages/b5/93/a7efbd39eac46c137500b37570c31dedc2
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-packages (from cupy-
Requirement already satisfied: numpy>=1.9.0 in /usr/local/lib/python3.6/dist-packages (from cup
Installing collected packages: fastrlock, cupy-cuda92
Successfully installed cupy-cuda92-5.1.0 fastrlock-0.4
```

```
In [7]: import tensorflow as tf
        device_name = tf.test.gpu_device_name()
        if device_name != '/device:GPU:0':
            raise SystemError('GPU device not found')
        print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

2.2 Model Implement

```
In [0]: import cupy as np
        from tensorflow import keras
        import matplotlib.pyplot as plt
        from tqdm import tqdm
        import time

        class Layer:
            """ Base class for neural network.
                Two abstract funtion forward and backward to help inference and
                backprogation
            """

            def __init__(self):
                self.name = 'layer'
                self.parameters = {}
```

```python
        self.has_weights = False

    def forward(self, x, is_training=True):
        pass

    def backward(self, dy):
        pass


def get_im2col_indices(x_shape, field_height, field_width,
                       padding=1, stride=(1, 1)):
    """ An implementation of im2col based on some fancy indexing

    Args:
        x_shape : [B, Cin, H, W]
        field_height: kH
        field_width: kW

    Return: [B*OH*OW, kH*kW*Cin]
    """

    # First figure out what the size of the output should be
    _, C, H, W = x_shape
    # assert (H + 2 * padding - field_height) % stride[0] == 0
    # assert (W + 2 * padding - field_height) % stride[1] == 0
    out_height = (H + 2 * padding - field_height) // stride[0] + 1
    out_width = (W + 2 * padding - field_width) // stride[1] + 1
    i0 = np.repeat(np.arange(field_height), field_width)
    i0 = np.tile(i0, C)
    i1 = stride[0] * np.repeat(np.arange(out_height), out_width)
    j0 = np.tile(np.arange(field_width), field_height * C)
    j1 = stride[1] * np.tile(np.arange(out_width), out_height)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)

    k = np.repeat(np.arange(C), field_height * field_width).reshape(-1, 1)

    return (k, i, j)


def im2col_indices(x, filter=(2, 2), padding=1, stride=(1, 1)):
    """ An implementation of im2col based on some fancy indexing

    Args:
        x : [B, H, W, Cin]
        field_height: kH
        field_width: kW
```

```python
    Return: [B*OH*OW, kH*kW*Cin]
    """

    # x => [B, Cin, H, W]
    x = x.transpose(0, 3, 1, 2)

    # Zero-pad the input
    p = padding
    x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

    k, i, j = get_im2col_indices(x.shape, filter[0], filter[1], padding,
                                 stride)
    # cols => [B, kH*kW*Cin, OH*OW]
    cols = x_padded[:, k, i, j]
    C = x.shape[1]


    # cols => [B*OH*OW, kH*kW*Cin]
    cols = cols.transpose(0, 2, 1).reshape(-1, filter[0] * filter[1] * C)
    return cols


def col2im_indices(cols, x_shape, filter=(2, 2), padding=0,
                   stride=(1,1) ):
    """ An implementation of col2im based on fancy indexing and np.add.at

    Args:
        cols: [B*OH*OW, kH*kW*Cin]
        x_shape: Shape of initial input (B, H, W, Cin)
        filter: Shape of filter (kH, kW)

    Returns: [B, H, W, Cin]


    """
    N, H, W, C = x_shape
    H_padded, W_padded = H + 2 * padding, W + 2 * padding
    x_padded = np.zeros((N, C, H_padded, W_padded), dtype=np.float32)
    k, i, j = get_im2col_indices((N, C, H, W), filter[0], filter[1], padding,
                                 stride)
    # cols => [B*OH*OW, kH*kW*Cin]
    # cols_reshaped => [B, OH*OW, kH*kW*Cin]
    cols_reshaped = cols.reshape(N, -1, C * filter[0] * filter[1])

    # [B, C * kH * kW, OH*OW] @@@
    cols_reshaped = cols_reshaped.transpose(0, 2, 1)
    idx = np.array(range(x_padded.shape[0])).reshape((x_padded.shape[0],1,1))
    np.scatter_add(x_padded,
```

```
                     (idx.astype(np.int32), k.astype(np.int32),
                      i.astype(np.int32), j.astype(np.uint64)),
                     cols_reshaped.astype(np.float32))
#     np.add.at(x_padded, (slice(None), k, i, j), cols_reshaped)
    if padding == 0:
#         print(np.max(x_padded), '---')
        return x_padded.transpose(0, 2, 3, 1)
    return x_padded[:, :, padding:-padding,
                    padding:-padding].transpose(0, 2, 3, 1)


class Conv2d(Layer):
    """ An implementation of Convolution 2D"""

    def __init__(self, input_shape=(-1, 3, 3, 1),
                 filter=(1, 2, 2, 1), stride=(1, 1),
                 padding='VALID', activation='relu'):
        super(Conv2d, self).__init__()
        self.name = 'conv2d'
        self.has_weights = True
        self.input_shape = input_shape
        self.filter = filter
        self.stride = stride
        self.cache = {}
        self.w_shape = []
        self.padding = 0
        if activation == 'relu':
            self.activation = Relu()
        else:
            self.activation = None
        self.out_height = (input_shape[1] + 2 * self.padding -
                           filter[1]) // stride[0] + 1
        self.out_width = (input_shape[2] + 2 * self.padding -
                          filter[2]) // stride[1] + 1

        self.linear_in = filter[1] * filter[2] * filter[3]
        self.linear_out = filter[0]
        self._linear = Linear(self.linear_in, self.linear_out)

    def forward(self, x, is_training=True):
        """Implement forward for Conv2d

        x: [B, H, W, Cin]
        """
        B, _, _, _ = x.shape
        self.input_shape = x.shape

        # x2row =>  [B*H*W, Cin]
```

```
        x2row = im2col_indices(
            x, filter=(self.filter[1], self.filter[2]),
            padding=self.padding, stride=self.stride)
        assert x2row.shape == (B * self.out_height * self.out_height,
                               self.linear_in)

        # out [B*OH*OW, Cout]
        out = self._linear.forward(x2row)
        if self.activation:
            out = self.activation.forward(out)
        # out [B, OH, OW, Cout]
        return out.reshape((B, self.out_height, self.out_width, self.linear_out))

    def backward(self, dy):
        """Implement forward for Conv2d

        dy: [B, H, W, Cout]
        """
        # dy => [B, OH, OW, Cout]
        dy = dy.reshape(-1, self.linear_out)
        # dx => [B]
        if self.activation:
            dy = self.activation.backward(dy)
        dx = self._linear.backward(dy)

        # cols => [B* OH*OW, kH*kW*Cin] => [kH*kW*Cin ,OH*OW, B] => [kH*kW*Cin ,OH*OW*
        # dx = dx.reshape(-1, self.out_height*self.out_width,
        #                  self.linear_in).tranpose(2, 1, 0).reshape(self.linear_in, -1
        #
        dx = col2im_indices(cols=dx, x_shape=self.input_shape, filter=(
            self.filter[1], self.filter[2]),
                            padding=self.padding, stride=self.stride)

        return dx

    def apply_grads(self, learning_rate=0.01, l2_penalty=1e-4):
        self._linear.apply_grads(learning_rate=0.01, l2_penalty=l2_penalty)


class Flatten(Layer):
    def __init__(self, input_shape):
        super(Flatten, self).__init__()
        self.input_shape = input_shape
        self.out_num = int(np.prod(np.array(self.input_shape[1:])))

    def forward(self, x, is_training=True):
        return x.reshape(-1, self.out_num)
```

```
    def backward(self, dy):

        return dy.reshape(self.input_shape)


class Linear(Layer):
    """ Implement fully connected dense layer.

    Recieve input [batch_size, num_in] and produce output [batch_size, num_out]
    folow expression: y = activation(x.w + b)

    Args:
      input_shape: (int) length input
      num_units: (int) length of output
      is_training: (bool) Determine whether is trainging or not
      activation: Default is not using activation
    """

    def __init__(self, num_in, num_out, activation=None):
        super(Linear, self).__init__()
        self.name = 'linear'
        self.cache = {}
        self.grads = {}
        self.bias_shape = [num_out]
        self.weight_shape = [num_in, num_out]
        self.has_weights = True
        self.parameters['W'] = self.initiate_vars(self.weight_shape)
        self.parameters['b'] = self.initiate_vars_zero(self.bias_shape)
        self.name = 'linear'

    def initiate_vars(self, shape, distribution=None):
        if distribution != None:
            raise ValueError(
                'Not implement distribution for initiating variable')

        return np.random.randn(shape[0], shape[1]) * 1e-3

    def initiate_vars_zero(self, shape):
        return np.zeros(shape)

    def forward(self, x, is_training=True):
        # check whether data has valid shape or not
        if is_training:
            self.cache['x'] = x.copy()
            self.batch_size = x.shape[0]
        y = np.dot(x, self.parameters['W']) + self.parameters['b']
```

```
        return y

    def backward(self, dy):
        self.grads['db'] = np.sum(dy, axis=0)
        self.grads['dW'] = np.dot(self.cache['x'].T, dy)
        dx = np.dot(dy, self.parameters['W'].T)

        return dx

    def apply_grads(self, learning_rate=0.01, l2_penalty=1e-4):
        self.parameters['W'] -= learning_rate * \
            (self.grads['dW'] + l2_penalty * self.parameters['W'])
        self.parameters['b'] -= learning_rate * \
            (self.grads['db'] + l2_penalty * self.parameters['b'])


class Relu(Layer):
    """ Implement Relu activation function:
    y = x with x >= 0
    y = 0 with x<0
    """

    def __init__(self):

        super(Relu, self).__init__()
        self.cache = {}
        self.has_weights = False
        self.name = 'Relu'

    def forward(self, x, is_training=True):
        if(is_training):
            self.cache['x'] = x.copy()
        y = x
        y[y < 0] = 0
        return y

    def backward(self, dy):
        dy[self.cache['x'] <= 0] = 0
        return dy


class Softmax(Layer):
    """ Implement Softmax activation function

    Recieve input with shape [batch_size, num_score] and produce output folowing
    expression:  y = e^score / sum(e^score)

    Args:
```

```
    """

    def __init__(self):
        super(Softmax, self).__init__()
        self.cache = {}
        self.has_weights = False
        self.name = 'Softmax'

    def forward(self, data, is_training=True):
        if(len(data.shape) != 2):
            raise ValueError(
                'data have shape is not compatible. Expect [batch_size, nums_score]')
        logits = np.exp(data - np.amax(data, axis=1, keepdims=True))
        logits = logits / np.sum(logits, axis=1, keepdims=True)
        if is_training:
            self.cache['logits'] = np.copy(logits)

        return logits

    def backward(self, dy):
        if(len(dy.shape) != 2):
            raise ValueError(
                'data have shape is not compatible. Expect [batch_size, nums_score]')
        num_units = dy.shape[-1]

        # [batch_size, num_units, 1] . [batch_size, 1, num_units]
        # = [batch_size, num_units, num_units]
        # Represent of matrix ds:
        #   [ds1/dx1 ds1/dx2 ... ds1/dxN]
        #   [ds2/dx1 ds2/dx2 ... ds2/dxN]
        #   [  ...           ...     ... ]
        #   [dsN/dx1 dsN/dx2 ... dsN/dxN]
        # with ds_i/dx_j = S_i(1 - S_j)  , with i==j
        #                = -S_i*S_j      , with i!=j
        ds = -np.matmul(np.expand_dims(self.cache['logits'], axis=-1),
                        np.expand_dims(self.cache['logits'], axis=1))
        ds[:, np.arange(num_units), np.arange(
            num_units)] += self.cache['logits']

        dx = np.matmul(np.expand_dims(dy, axis=1), ds)
        return np.squeeze(dx, axis=1)
        # return dy


class MaxPooling2D(Layer):

    def __init__(self, pool_size=(2, 2), stride=(1, 1), padding=0):
        self.name = 'maxpool2d'
```

```python
        self.cache = {}
        self.pool_size = pool_size
        self.stride = stride
        self.pading = padding
        self.has_weights = False

    def forward(self, x, is_training=True):
        N, H, W, C = x.shape
        pool_height, pool_width = self.pool_size
        stride_height, stride_width = self.stride

        out_height = (H - pool_height) // stride_height + 1
        out_width = (W - pool_width) // stride_width + 1
        x_split = x.transpose(0, 3, 1, 2).reshape(N * C, H, W, 1)
        x_cols = im2col_indices(x_split, self.pool_size,
                                padding=0, stride=self.stride) # (out_height*out_width
        x_cols_argmax = np.argmax(x_cols, axis=1)
        x_cols_max = x_cols[np.arange(x_cols.shape[0]), x_cols_argmax]              #
        out = x_cols_max.reshape(out_height,
                                 out_width, N, C).transpose(2, 0, 1, 3) # (N, out_heig

        if is_training:
            self.cache['x'] = x.copy()
            self.cache['x_cols'] = x_cols
            self.cache['x_cols_argmax'] = x_cols_argmax

        return out # (N, out_height,out_width, C)

    def backward(self, dout):
        x, x_cols,  = self.cache['x'], self.cache['x_cols']
        x_cols_argmax =self.cache['x_cols_argmax']
        N, H, W, C = x.shape
        #dout: # (N, out_height,out_width, C)
        dout_reshaped = dout.transpose(1, 2, 0, 3).flatten() # (out_height*out_width*N
        dx_cols = np.zeros(x_cols.shape)                       # (out_height*out_width*
        dx_cols[np.arange(dx_cols.shape[0]), x_cols_argmax] = dout_reshaped

        dx = col2im_indices(dx_cols, (N * C, H, W, 1),
                            self.pool_size, padding=0, stride=self.stride)  #[N, H, W,
        dx = dx.reshape((N,C,H,W)).transpose(0,2,3,1)
        return dx

class MaxPooling2DNaive(Layer):

    def __init__(self, pool_size=(2, 2), strides=(1, 1), padding=0):
        self.cache = {}
        self.pool_size = pool_size
        self.strides = strides
```

```python
        self.pading = padding
        self.has_weights = False

    def forward(self, x, is_training=True):
        N, H, W, C = x.shape
        HH, WW = self.pool_size
        stride_height, stride_width = self.strides

        out_height = (H - HH) // stride_height + 1
        out_width = (W - WW) // stride_width + 1

        out = np.zeros((N, out_height, out_width, C))

        for j in range(out_height):
            h_start = j*stride_height
            for k in range(out_width):
                w_start = k*stride_width
                out[:, j, k, :] = (
                    x[:, h_start:(h_start+HH),
                      w_start:(w_start+WW), :].max(axis=(1, 2)))

        if is_training:
            self.cache['x'] = x.copy()

        return out

    def backward(self, dout):
        dx = None
        x = self.cache['x']
        # Get dimensions
        N, H, W, C = x.shape
        HH, WW = self.pool_size
        stride_height, stride_width = self.strides

        # Compute dimension filters
        out_height = (H-HH) // stride_height + 1
        out_width = (W-WW) // stride_width + 1

        # Initialize tensor for dx
        dx = np.zeros_like(x)

        # Backpropagate dout on x
        for i in range(N):
            for z in range(C):
                for j in range(out_height):
                    h_start = j*stride_height
                    for k in range(out_width):
                        w_start = k*stride_width
```

```
                        dpatch = np.zeros((HH, WW))
                        input_patch = x[i, h_start:(h_start+HH),
                                        w_start:(w_start+WW), z]
                        idxs_max = np.where(input_patch == input_patch.max())
                        dpatch[idxs_max[0], idxs_max[1]] = dout[i, j, k, z]
                        dx[i, h_start:(h_start+HH),
                           w_start:(w_start+WW), z] += dpatch

        return dx




class CELoss():
    """ Cross Entropy Loss

    Loss function: L = sum(y.log(s)) / batch_size ,
        y is labels,
        s is predict
    Derivative of Loss: dL/ds_i= y_i/s_i
    """

    def __init__(self):
        self.cache = {}
        self.has_weights = False
        self.eps = 1e-8
        # super(CELoss, self).__init__()

    def compute_loss(self, logits, labels, is_training=True):
        logits = np.clip(logits, self.eps, 1. - self.eps)
        # logits => [batch_size, num_units]
        # labels => [batch_size, num_units]
        if is_training:
            self.cache['labels'] = labels.copy()
            self.cache['logits'] = logits.copy()
        self.batch_size = logits.shape[0]
        loss = - np.sum(labels * np.log(logits)) / self.batch_size
        return loss

    def compute_derivation(self, logits, labels):
        # => [batch_size, num_units]

        # return (logits - labels)/self.batch_size
        # return - self.cache['labels'] / (self.cache['logits'] * self.cache['logits']
        return - self.cache['labels'] / (self.cache['logits'] * self.batch_size)
```

```python
class DropOut(Layer):
    def __init__(self, prob,  seed = None, distribution='uniform'):
        super(DropOut, self).__init__()
        self.has_weights= False
        self.seed = seed
        self.prob = prob
        self.distribution = distribution
        self.cache = {}

    def forward(self, x, is_training=False):
        if is_training:
            if self.distribution == 'uniform':
                mask = (np.random.rand(*x.shape)<self.prob)/self.prob
            self.cache['mask'] = mask
            out = x*mask
        else:
            out = x
        return out

    def backward(self, dy):
        return self.cache['mask'] * dy


class Model:
    def __init__(self, *model, **kwargs):
        self.model = model
        self.num_classes = 0
        self.batch_size = 0
        self.loss = None
        self.optimizer = None
        self.name = kwargs['name'] if 'name' in kwargs else None
        self.loss_history = []
        self.train_acc_history = []
        self.eval_acc_history = []
        self.iters = []

    def add(self, layer):
        self.model.append()

    def set_batch_size(self, batch_size):
        self.batch_size = batch_size

    def set_num_classes(self, num_classes):
        self.num_classes = num_classes

    def set_loss(self, loss):
        self.loss = loss
```

```python
def get_batches(self, data, labels, batch_size=256, shuffle=True):
    N = data.shape[0]
    num_batches = N // batch_size
    if(shuffle):
        rand_idx = np.random.permutation(data.shape[0])
        data = data[rand_idx]
        labels = labels[rand_idx]
    for i in np.arange(num_batches):
        yield (data[i*batch_size:(i+1) * batch_size],
               labels[i*batch_size:(i+1) * batch_size])
    if N % batch_size != 0 and num_batches != 0:
        yield (data[batch_size*num_batches:],
               labels[batch_size*num_batches:])

def train(self, train_data, train_labels,
          eval_data, eval_labels,
          batch_size=1024, epochs=50,
          display_after=50, eval_after = 250,
          learning_rate=0.01,
          l2_penalty=1e-4,
          learning_rate_decay=0.95):
    if self.loss is None:
        raise RuntimeError("Set loss first using 'model.set_loss(<loss>)'")
    self.set_batch_size(batch_size)
    self.set_num_classes(train_labels.shape[1])

    iter = 0
    for epoch in range(epochs):

        print('Running Epoch:', epoch + 1)

        for i, (x_batch, y_batch) in enumerate(
            self.get_batches(train_data, train_labels,
                             batch_size=batch_size)):
            batch_preds = x_batch.copy()
            for  layer in self.model:
                batch_preds = layer.forward(batch_preds, is_training=True)

            loss = self.loss.compute_loss(
                logits=batch_preds, labels=y_batch)
            dA = self.loss.compute_derivation(
                logits=batch_preds, labels=y_batch)

            temp1 = []
            temp2 = []
            temp3 = []
            for i, layer in enumerate(reversed(self.model)):
                dA = layer.backward(dA)
```

```
                    temp1.append(np.max(dA))
                    if layer.name == 'linear':
                        temp2.append((np.max(layer.grads['dW']),
                                      np.max(layer.grads['db'] ) ))
                    if layer.name == 'conv2d':
                        temp3.append((np.max(layer._linear.grads['dW']),
                                      np.max(layer._linear.grads['db']) ))

                for layer in self.model:
                    if layer.has_weights:
                        layer.apply_grads(
                            learning_rate=learning_rate, l2_penalty=l2_penalty)
                iter += 1
                if iter % display_after == 0:
                    train_acc = self.evaluate(x_batch, y_batch)
                    eval_acc = self.evaluate(eval_data, eval_labels)
                    print('Step {}, loss: {}, train_acc: {}, eval_acc: {}'.format(
                        iter, loss, train_acc, eval_acc))
                    self.iters.append(iter)
                    self.loss_history.append(loss)
                    self.train_acc_history.append(train_acc)
                    self.eval_acc_history.append(eval_acc)

            learning_rate *= learning_rate_decay


    def predict(self, data):
        batch_preds = data.copy()
        for layer in self.model:
            batch_preds = layer.forward(batch_preds)
        return batch_preds

    def evaluate(self, data, labels):
        predictions = self.predict(data)
        if(predictions.shape != labels.shape):
            raise ValueError('prediction shape does not match labels shape')
        return np.mean(np.argmax(labels,axis=1)==np.argmax(predictions, axis=1))
```

2.3 Pre-processing

In [0]:
```
num_classes = 10
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = np.array(train_images, dtype=np.float32)
test_images = np.array(test_images, dtype=np.float32)
```

```
train_labels = np.array(train_labels, dtype=np.int32)
test_labels = np.array(test_labels, dtype=np.int32)

# train_images = train_images / 255.0
# test_images = test_images / 255.0
train_images = train_images.reshape((-1, 28, 28, 1))
test_images = test_images.reshape((-1, 28, 28, 1))
# train_images = train_images.reshape((-1, 784))
# test_images = test_images.reshape((-1, 784))


labels = np.zeros((train_labels.shape[0], 10))
labels[np.arange(train_labels.shape[0]), train_labels] = 1
train_labels = labels

labels = np.zeros((test_labels.shape[0], 10))
labels[np.arange(test_labels.shape[0]), test_labels] = 1
test_labels = labels

batch_size = train_images.shape[0]
num_train = batch_size*9 // 10

eval_images = train_images[num_train:]
eval_labels = train_labels[num_train:]
train_images = train_images[:num_train]
train_labels = train_labels[:num_train]
```

2.4 Dropout model

```
In [0]: model1 = Model(
                     Conv2d(input_shape=(-1, 28, 28, 1),
                            filter=(10, 2, 2, 1), activation='relu'),
                     DropOut(0.7),
                     Flatten(input_shape= (-1, 27, 27, 10)),
                     Linear(num_in=27*27*10, num_out=10),
                     Softmax())

        model1.set_loss(CELoss())
```

2.4.1 Train

```
In [11]: t1 = time.time()
         model1.train(train_images,
                      train_labels, eval_images,
                      eval_labels, learning_rate=0.00001,
                      l2_penalty=0.,
                      epochs=10,
                      batch_size=200,
```

```
                    display_after = 50)
        print('Time: ',time.time()-t1, 's')
```

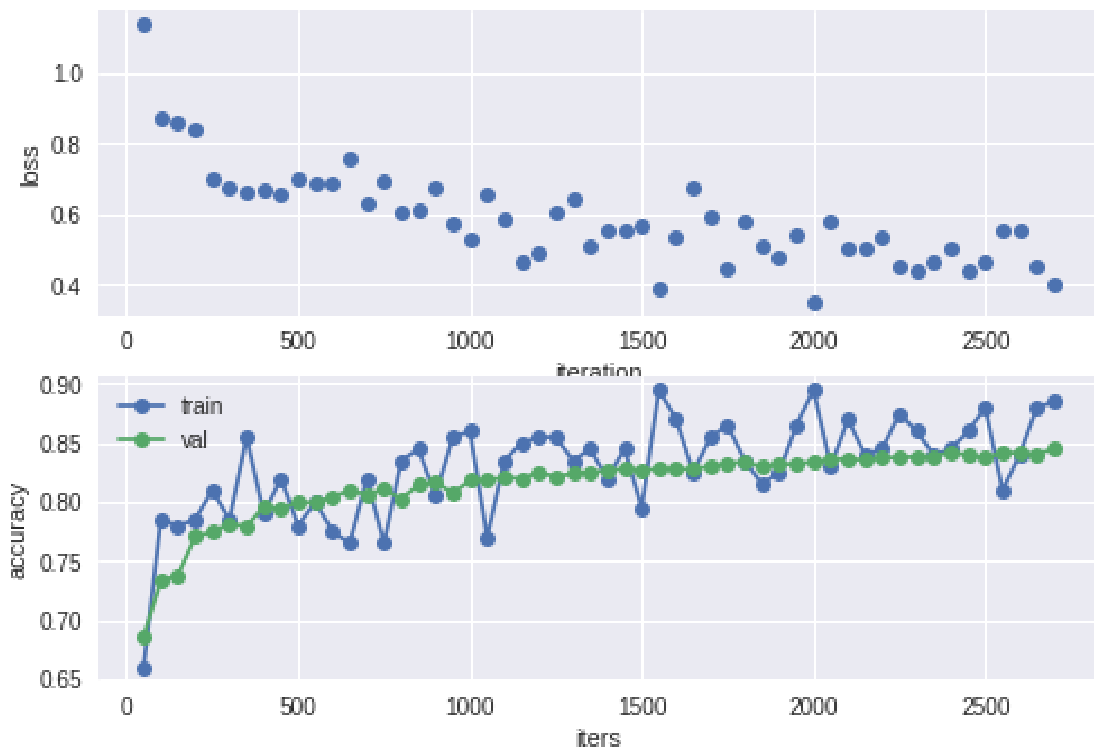
```
Running Epoch: 1
Step 50, loss: 1.1378407274352689, train_acc: 0.66, eval_acc: 0.686
Step 100, loss: 0.8694343215098129, train_acc: 0.785, eval_acc: 0.7331666666666666
Step 150, loss: 0.861820227466097, train_acc: 0.78, eval_acc: 0.737666666666667
Step 200, loss: 0.8382911474651726, train_acc: 0.785, eval_acc: 0.7706666666666667
Step 250, loss: 0.6972122242795435, train_acc: 0.81, eval_acc: 0.7745
Running Epoch: 2
Step 300, loss: 0.6753853681613334, train_acc: 0.785, eval_acc: 0.7818333333333334
Step 350, loss: 0.6633016510497063, train_acc: 0.855, eval_acc: 0.7791666666666667
Step 400, loss: 0.6674593611520112, train_acc: 0.79, eval_acc: 0.7961666666666667
Step 450, loss: 0.6529086732348012, train_acc: 0.82, eval_acc: 0.794
Step 500, loss: 0.6981333452287334, train_acc: 0.78, eval_acc: 0.8
Running Epoch: 3
Step 550, loss: 0.6880382476759612, train_acc: 0.8, eval_acc: 0.7991666666666667
Step 600, loss: 0.6889265865664055, train_acc: 0.775, eval_acc: 0.8043333333333333
Step 650, loss: 0.7541314131975053, train_acc: 0.765, eval_acc: 0.8096666666666666
Step 700, loss: 0.6328606576603067, train_acc: 0.82, eval_acc: 0.8063333333333333
Step 750, loss: 0.6933119248373341, train_acc: 0.765, eval_acc: 0.8106666666666666
Step 800, loss: 0.6040472561786026, train_acc: 0.835, eval_acc: 0.8026666666666666
Running Epoch: 4
Step 850, loss: 0.6110051555766144, train_acc: 0.845, eval_acc: 0.8143333333333334
Step 900, loss: 0.672930141496069, train_acc: 0.805, eval_acc: 0.817
Step 950, loss: 0.5706717599832749, train_acc: 0.855, eval_acc: 0.8071666666666667
Step 1000, loss: 0.5313616399067231, train_acc: 0.86, eval_acc: 0.8183333333333334
Step 1050, loss: 0.6585805917732885, train_acc: 0.77, eval_acc: 0.8183333333333334
Running Epoch: 5
Step 1100, loss: 0.5872062288344165, train_acc: 0.835, eval_acc: 0.8208333333333333
Step 1150, loss: 0.46456018510217123, train_acc: 0.85, eval_acc: 0.8198333333333333
Step 1200, loss: 0.4905504284322342, train_acc: 0.855, eval_acc: 0.8251666666666667
Step 1250, loss: 0.6075132135259371, train_acc: 0.855, eval_acc: 0.8215
Step 1300, loss: 0.6426138507661681, train_acc: 0.835, eval_acc: 0.824
Step 1350, loss: 0.5069314524418553, train_acc: 0.845, eval_acc: 0.8255
Running Epoch: 6
Step 1400, loss: 0.5547588134141174, train_acc: 0.82, eval_acc: 0.8263333333333334
Step 1450, loss: 0.5546071568499635, train_acc: 0.845, eval_acc: 0.8285
Step 1500, loss: 0.5680553415184154, train_acc: 0.795, eval_acc: 0.827
Step 1550, loss: 0.3905150664265657, train_acc: 0.895, eval_acc: 0.8281666666666667
Step 1600, loss: 0.5337820626016188, train_acc: 0.87, eval_acc: 0.828
Running Epoch: 7
Step 1650, loss: 0.6748892278254977, train_acc: 0.825, eval_acc: 0.8278333333333333
Step 1700, loss: 0.5930970012031243, train_acc: 0.855, eval_acc: 0.8303333333333334
Step 1750, loss: 0.448333660822492, train_acc: 0.865, eval_acc: 0.8326666666666667
Step 1800, loss: 0.5804999812195976, train_acc: 0.835, eval_acc: 0.834
Step 1850, loss: 0.5081234814749124, train_acc: 0.815, eval_acc: 0.8298333333333333
Running Epoch: 8
```

```
Step 1900, loss: 0.47943782926736345, train_acc: 0.825, eval_acc: 0.832
Step 1950, loss: 0.5419878446369428, train_acc: 0.865, eval_acc: 0.8321666666666667
Step 2000, loss: 0.35408119235925284, train_acc: 0.895, eval_acc: 0.8341666666666666
Step 2050, loss: 0.5789340780489045, train_acc: 0.83, eval_acc: 0.8358333333333333
Step 2100, loss: 0.5037018338935421, train_acc: 0.87, eval_acc: 0.8368333333333333
Step 2150, loss: 0.5033871638596844, train_acc: 0.84, eval_acc: 0.8355
Running Epoch: 9
Step 2200, loss: 0.5363438188156348, train_acc: 0.845, eval_acc: 0.8378333333333333
Step 2250, loss: 0.4509217048488066, train_acc: 0.875, eval_acc: 0.8375
Step 2300, loss: 0.4383085756887124, train_acc: 0.86, eval_acc: 0.8386666666666667
Step 2350, loss: 0.4681024764419871, train_acc: 0.84, eval_acc: 0.839
Step 2400, loss: 0.5038422247339331, train_acc: 0.845, eval_acc: 0.842
Running Epoch: 10
Step 2450, loss: 0.43683211084871426, train_acc: 0.86, eval_acc: 0.8398333333333333
Step 2500, loss: 0.4666983885304228, train_acc: 0.88, eval_acc: 0.8373333333333334
Step 2550, loss: 0.5553651270832676, train_acc: 0.81, eval_acc: 0.842
Step 2600, loss: 0.5538391020056785, train_acc: 0.84, eval_acc: 0.8415
Step 2650, loss: 0.4497331510382021, train_acc: 0.88, eval_acc: 0.8408333333333333
Step 2700, loss: 0.40241282418889873, train_acc: 0.885, eval_acc: 0.845
Time:  110.48339653015137 s
```

2.4.2 Sumary

In [12]:
```
# Dropout
plt.subplot(2, 1, 1)
plt.plot(model1.iters, model1.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(model1.iters, model1.train_acc_history, '-o')
plt.plot(model1.iters, model1.eval_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('iters')
plt.ylabel('accuracy')
plt.show()
```



In [0]:

2.4.3 Evaluate on test set

```
In [13]: batch = test_images
         for layer in model1.model:
           batch = layer.forward(batch)

         test_lost = model1.loss.compute_loss(batch, test_labels)

         test_acc = model1.evaluate(test_images, test_labels)

         print('Test loss: {}, acc: {}'.format(test_lost, test_acc))
```

```
Test loss: 0.4656645951374645, acc: 0.8378
```

2.5 No-Dropout model

```
In [0]: no_dropout_model = Model(
                      Conv2d(input_shape=(-1, 28, 28, 1),
                             filter=(10, 2, 2, 1), activation='relu'),
                      Flatten(input_shape= (-1, 27, 27, 10)),
```

```
                 Linear(num_in=27*27*10, num_out=10),
                 Softmax())
```

```
    no_dropout_model.set_loss(CELoss())
```

2.5.1 Train

```
In [15]: t1 = time.time()
         no_dropout_model.train(train_images,
                      train_labels, eval_images,
                      eval_labels, learning_rate=0.00001,
                      l2_penalty=0.,
                      epochs=10,
                      batch_size=200,
                      display_after = 50)
         print('Time: ',time.time()-t1, 's')
```

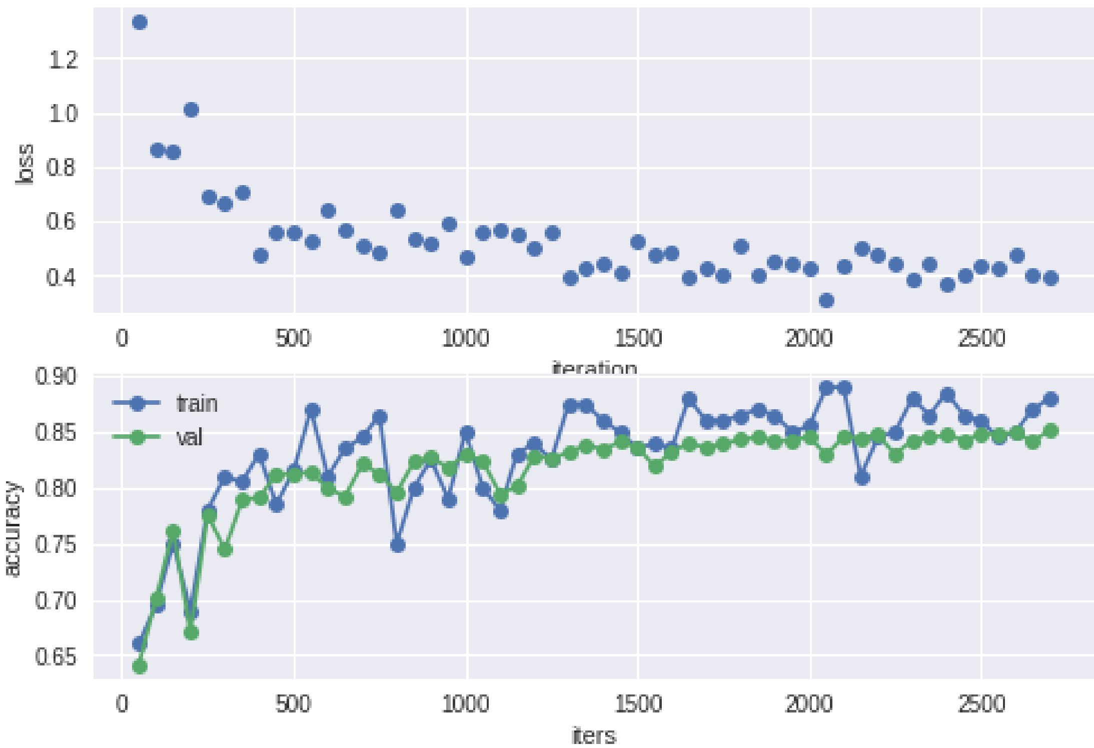
```
Running Epoch: 1
Step 50, loss: 1.3348772986676678, train_acc: 0.66, eval_acc: 0.6415
Step 100, loss: 0.8683842634542654, train_acc: 0.695, eval_acc: 0.701
Step 150, loss: 0.8586395879663303, train_acc: 0.75, eval_acc: 0.7618333333333334
Step 200, loss: 1.0109595135135712, train_acc: 0.69, eval_acc: 0.6701666666666667
Step 250, loss: 0.692774184369158, train_acc: 0.78, eval_acc: 0.7756666666666666
Running Epoch: 2
Step 300, loss: 0.6630058120753994, train_acc: 0.81, eval_acc: 0.7443333333333333
Step 350, loss: 0.707199712655177, train_acc: 0.805, eval_acc: 0.7891666666666667
Step 400, loss: 0.48071539018416537, train_acc: 0.83, eval_acc: 0.7908333333333334
Step 450, loss: 0.5596217021396255, train_acc: 0.785, eval_acc: 0.812
Step 500, loss: 0.5620245737765824, train_acc: 0.815, eval_acc: 0.8123333333333334
Running Epoch: 3
Step 550, loss: 0.5236331430730682, train_acc: 0.87, eval_acc: 0.8136666666666666
Step 600, loss: 0.6417094329961706, train_acc: 0.81, eval_acc: 0.7993333333333333
Step 650, loss: 0.5695296919962707, train_acc: 0.835, eval_acc: 0.7921666666666667
Step 700, loss: 0.5107207241858857, train_acc: 0.845, eval_acc: 0.821
Step 750, loss: 0.4826091647353054, train_acc: 0.865, eval_acc: 0.8123333333333334
Step 800, loss: 0.6459076759860163, train_acc: 0.75, eval_acc: 0.7958333333333333
Running Epoch: 4
Step 850, loss: 0.5326805796204256, train_acc: 0.8, eval_acc: 0.823
Step 900, loss: 0.5182315682861798, train_acc: 0.825, eval_acc: 0.8273333333333334
Step 950, loss: 0.5917983443784267, train_acc: 0.79, eval_acc: 0.8175
Step 1000, loss: 0.4715173045812041, train_acc: 0.85, eval_acc: 0.8295
Step 1050, loss: 0.5584880714297054, train_acc: 0.8, eval_acc: 0.8238333333333333
Running Epoch: 5
Step 1100, loss: 0.5641485050924409, train_acc: 0.78, eval_acc: 0.7945
```

```
Step 1150, loss: 0.5520295139287635, train_acc: 0.83, eval_acc: 0.8006666666666666
Step 1200, loss: 0.5046534480591558, train_acc: 0.84, eval_acc: 0.8281666666666667
Step 1250, loss: 0.5605885123998275, train_acc: 0.825, eval_acc: 0.8263333333333334
Step 1300, loss: 0.39622633126153234, train_acc: 0.875, eval_acc: 0.8315
Step 1350, loss: 0.4257398998720645, train_acc: 0.875, eval_acc: 0.8385
Running Epoch: 6
Step 1400, loss: 0.4402230310832479, train_acc: 0.86, eval_acc: 0.8343333333333334
Step 1450, loss: 0.41527123840019586, train_acc: 0.85, eval_acc: 0.841
Step 1500, loss: 0.5302166400711252, train_acc: 0.835, eval_acc: 0.8363333333333334
Step 1550, loss: 0.4808349347619062, train_acc: 0.84, eval_acc: 0.819
Step 1600, loss: 0.48554607852892145, train_acc: 0.835, eval_acc: 0.8316666666666667
Running Epoch: 7
Step 1650, loss: 0.39105693111559653, train_acc: 0.88, eval_acc: 0.8401666666666666
Step 1700, loss: 0.42993188554977835, train_acc: 0.86, eval_acc: 0.8358333333333333
Step 1750, loss: 0.40176346592436185, train_acc: 0.86, eval_acc: 0.8395
Step 1800, loss: 0.5101159555401285, train_acc: 0.865, eval_acc: 0.8431666666666666
Step 1850, loss: 0.4041705529618589, train_acc: 0.87, eval_acc: 0.8451666666666666
Running Epoch: 8
Step 1900, loss: 0.45123631021246574, train_acc: 0.865, eval_acc: 0.8418333333333333
Step 1950, loss: 0.4446152852183718, train_acc: 0.85, eval_acc: 0.8426666666666667
Step 2000, loss: 0.42743721219354186, train_acc: 0.855, eval_acc: 0.8455
Step 2050, loss: 0.31607136667282215, train_acc: 0.89, eval_acc: 0.8293333333333334
Step 2100, loss: 0.4363787266150468, train_acc: 0.89, eval_acc: 0.8456666666666667
Step 2150, loss: 0.5001722558417668, train_acc: 0.81, eval_acc: 0.8448333333333333
Running Epoch: 9
Step 2200, loss: 0.48052263543216456, train_acc: 0.845, eval_acc: 0.8475
Step 2250, loss: 0.4460287702675301, train_acc: 0.85, eval_acc: 0.8298333333333333
Step 2300, loss: 0.3827204290067789, train_acc: 0.88, eval_acc: 0.8426666666666667
Step 2350, loss: 0.4467645352663075, train_acc: 0.865, eval_acc: 0.845
Step 2400, loss: 0.36638621137102517, train_acc: 0.885, eval_acc: 0.8471666666666666
Running Epoch: 10
Step 2450, loss: 0.40016899845224685, train_acc: 0.865, eval_acc: 0.8426666666666667
Step 2500, loss: 0.4372406576062932, train_acc: 0.86, eval_acc: 0.8475
Step 2550, loss: 0.4275682332326707, train_acc: 0.845, eval_acc: 0.8483333333333334
Step 2600, loss: 0.4801387263517162, train_acc: 0.85, eval_acc: 0.849
Step 2650, loss: 0.40327112429433726, train_acc: 0.87, eval_acc: 0.8418333333333333
Step 2700, loss: 0.39676697076421624, train_acc: 0.88, eval_acc: 0.8511666666666666
Time:  87.56203365325928 s
```

2.5.2 Sumary

In [16]:
```
# No Dropout
plt.subplot(2, 1, 1)
plt.plot(no_dropout_model.iters, no_dropout_model.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')
```

```
plt.subplot(2, 1, 2)
plt.plot(no_dropout_model.iters, no_dropout_model.train_acc_history, '-o')
plt.plot(no_dropout_model.iters, no_dropout_model.eval_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('iters')
plt.ylabel('accuracy')
plt.show()
```



In [0]:

2.5.3 Evaluate on test set

In [17]:
```
batch = test_images
for layer in model1.model:
  batch = layer.forward(batch)

test_lost = no_dropout_model.loss.compute_loss(batch, test_labels)

test_acc = no_dropout_model.evaluate(test_images, test_labels)

print('Test loss: {}, acc: {}'.format(test_lost, test_acc))
```

```
Test loss: 0.4656645951374645, acc: 0.8465
```

3 Keras

3.1 Pre-processing

```
In [0]: import numpy as np

        num_classes = 10
        class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                       'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

        fashion_mnist = keras.datasets.fashion_mnist
        (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()


        train_images = train_images / 255.0
        test_images = test_images / 255.0
        train_images = train_images.reshape((-1, 28, 28, 1))
        test_images = test_images.reshape((-1, 28, 28, 1))
        # train_images = train_images.reshape((-1, 784))
        # test_images = test_images.reshape((-1, 784))


        labels = np.zeros((train_labels.shape[0], 10))
        labels[np.arange(train_labels.shape[0]), train_labels] = 1
        train_labels = labels

        labels = np.zeros((test_labels.shape[0], 10))
        labels[np.arange(test_labels.shape[0]), test_labels] = 1
        test_labels = labels


        batch_size = train_images.shape[0]
        num_train = batch_size*9 // 10

        eval_images = train_images[num_train:]
        eval_labels = train_labels[num_train:]
        train_images = train_images[:num_train]
        train_labels = train_labels[:num_train]
```

3.2 Model

```
In [19]: import keras
         from keras.models import Sequential
         from keras.layers import Dense, Dropout, Flatten
         from keras.layers import Conv2D, MaxPooling2D
         model = Sequential()
         model.add(Conv2D(10, kernel_size=(2, 2),
                          activation='relu',
```

```
                          input_shape=(28,28,1)))
         model.add(Dropout(0.3))
         model.add(Flatten())
         model.add(Dense(10, activation='softmax'))
         model.compile(loss=keras.losses.categorical_crossentropy,
                       optimizer=keras.optimizers.Adam(), metrics=['accuracy'])
```

```
Using TensorFlow backend.
```

```
In [20]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 27, 27, 10)        50
_________________________________________________________________
dropout_1 (Dropout)          (None, 27, 27, 10)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 7290)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                72910
=================================================================
Total params: 72,960
Trainable params: 72,960
Non-trainable params: 0
_________________________________________________________________
```

3.3 Train

```
In [21]: t1 = time.time()
         history = model.fit(train_images, train_labels,
                   batch_size=200,
                   epochs=10,
                   verbose=1,
                   validation_data=(eval_images, eval_labels))

         print('Time: ',time.time()-t1, 's')
```

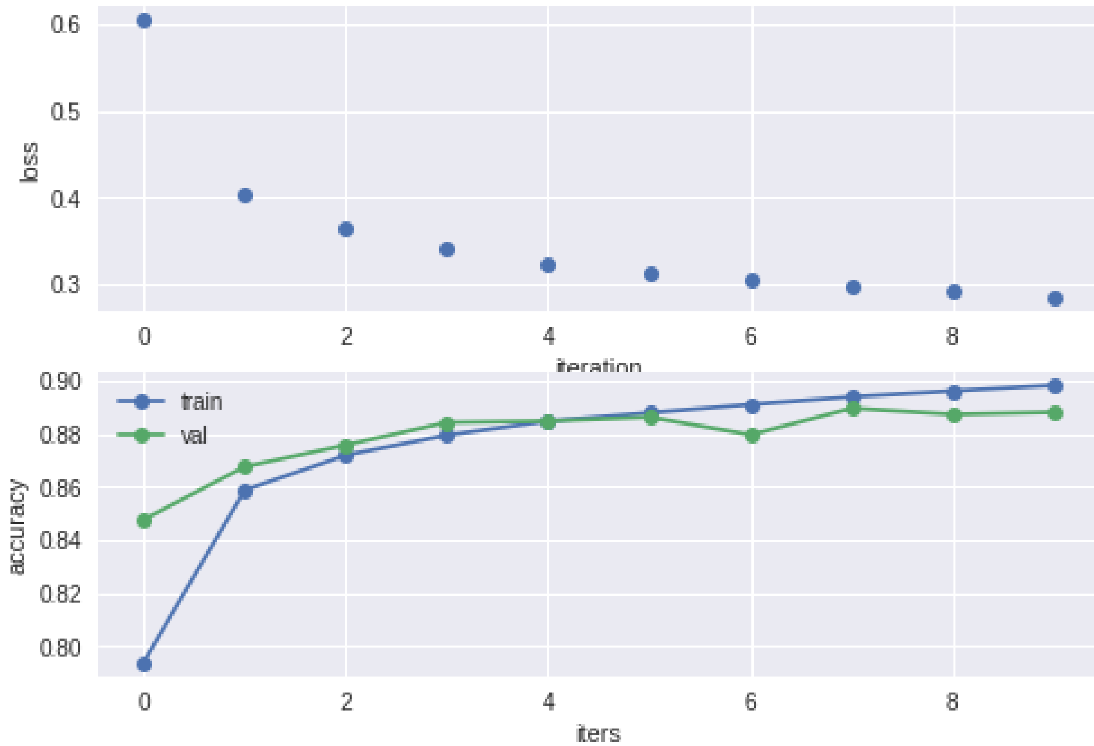
```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 5s 91us/step - loss: 0.6053 - acc: 0.7938 - val_
Epoch 2/10
54000/54000 [==============================] - 3s 53us/step - loss: 0.4027 - acc: 0.8586 - val_
Epoch 3/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.3641 - acc: 0.8719 - val_
Epoch 4/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.3404 - acc: 0.8794 - val_
```

```
Epoch 5/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.3231 - acc: 0.8846 - val_
Epoch 6/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.3129 - acc: 0.8878 - val_
Epoch 7/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.3050 - acc: 0.8909 - val_
Epoch 8/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.2964 - acc: 0.8938 - val_
Epoch 9/10
54000/54000 [==============================] - 3s 54us/step - loss: 0.2910 - acc: 0.8960 - val_
Epoch 10/10
54000/54000 [==============================] - 3s 51us/step - loss: 0.2849 - acc: 0.8981 - val_
Time:  31.31539034843445 s
```

3.4 Sumary

```
In [22]: plt.subplot(2, 1, 1)
         plt.plot(history.history['loss'], 'o')
         plt.xlabel('iteration')
         plt.ylabel('loss')

         plt.subplot(2, 1, 2)
         plt.plot(history.history['acc'], '-o')
         plt.plot(history.history['val_acc'], '-o')
         plt.legend(['train', 'val'], loc='upper left')
         plt.xlabel('iters')
         plt.ylabel('accuracy')
         plt.show()
```



3.5 Evaluate

```
In [23]: score = model.evaluate(test_images, test_labels, verbose=0)
         print('Test loss:', score[0])
         print('Test accuracy:', score[1])
```

```
Test loss: 0.3220656955957413
Test accuracy: 0.8837
```