BLG 506E COMPUTER VISION ASSIGNMENT 5

Convolutional Nets: The Last Stand

1- We work on CIFAR10-dataset at this assignment. It has 50000 Train Data and 10000 Test Data. Firstly, we load CIFAR10 dataset. We split the data into 49000 train, 1000 validation and 1000 test sets.

2- conv_forward_naive function

We want to establish a convolutional neural network structure. Here we will create the convolutional layer. Here we will set the zeros and how long the filter will shift by using stride and padding.

Here, we want to implementation of the forward for a convolutional layer. The input consists of N data points, each with C channels, height H and width W. We convolve each input with F different filters, where each filter spans all C channels and has height HH and width WW.

def conv_forward_naive(x, w, b, conv_param):

```
    out = None
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape

    stride, pad = conv_param['stride'], conv_param['pad']
    H_ı = 1 + ( H + 2 * pad - HH )// stride
    W_ı = 1 + ( W + 2 * pad - WW )// stride

    padding = [(0, 0), (0, 0), (pad, pad), (pad, pad)]
    x_pad = np.pad(x, padding, 'constant', constant_values=0)

    # create output tensor after convolution layer
    out = np.zeros((N, F, H_ı, W_ı))
    for n in range(N):
        for f in range(F):
            for h_ı in range(H_ı):
                for w_ı in range(W_ı):
out[n, f, h_ı, w_ı] = np.sum(x_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW] * w[f,:]) + b[f]
```

*****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

```
cache = (x, w, b, conv_param)
return out, cache
```

Input:

- x: Input data of shape (N, C, H, W)
- w: Filter weights of shape (F, C, HH, WW)
- b: Biases, of shape (F,)
- conv_param: A dictionary with the following keys:
- 'stride': The number of pixels between adjacent receptive fields in the horizontal and vertical directions.
- 'pad': The number of pixels that will be used to zero-pad the input.

First we take the input dimensions and filter weight dimensions. And then we take stride and pad params.

We can compute the spatial size of the output volume as a function of the input volume size (H and W), filter volume size(HH and WW), the stride with which they are applied (stride).

H' = 1 + (H + 2 * pad - HH) / stride

W' = 1 + (W + 2 * pad - WW) / stride

I use "H_ı" instead of H’ in the program and I use “W_ı” instead of W’.

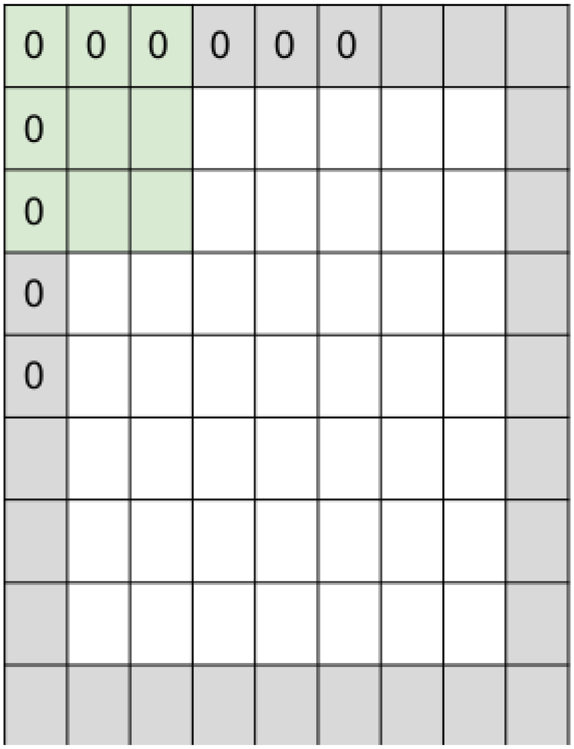
```
padding = [(0, 0), (0, 0), (pad, pad), (pad, pad)]
x_pad = np.pad(x, padding, 'constant', constant_values=0)
```

And then I use np.pad function for padding. We create matrix that padded.

```
out = np.zeros((N, F, H_ı, W_ı))
```

Let's fill out now. First, we create out as a array with the dimensions N, F, H_ı and W_ı.

We walk through the matrix using the weight and height indices for each matrix in each channel in each example.



We have a matrix in which we add padding.

> out[n, f, h_ı, w_ı] = np.sum(x_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW] * w[f,:]) + b[f]

=> x_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW]

==> h_ı * stride : h_ı * stride + HH

==> w_ı * stride : w_ı * stride + WW

In x_pad, we move the HH and WW portion in the matrix and add the sum of each hh x ww matrix to the current output.

Out:

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

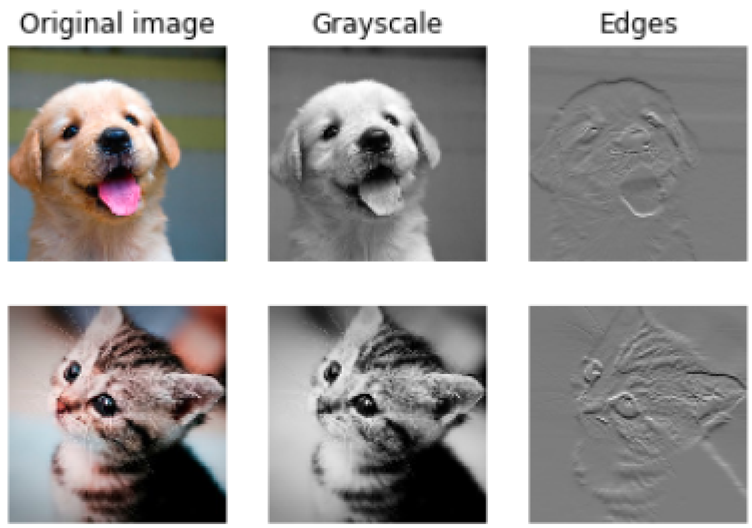
3-

w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]

w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]

w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

Here, we apply these filters for each of the 3 channels to 2 images. we use convolution to apply filter to picture matrix.



4- conv_backward_naive function

Here, we want to implementation of the backward for a convolutional layer.

Inputs:

- dout: Upstream derivatives.

- cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

```
def conv_backward_naive(dout, cache):
    dx, dw, db = None, None, None
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    x, w, b, conv_param = cache
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape
    stride, pad = conv_param['stride'], conv_param['pad']

    N, F, H_ı, W_ı = dout.shape

    padding = [(0, 0), (0, 0), (pad, pad), (pad, pad)]
    x_pad = np.pad(x, padding, 'constant', constant_values=0)

    dx_pad = np.zeros_like(x_pad)
```

```
    dw = np.zeros_like(w)

    db = np.zeros_like(b)


    for n in range(N):

        for f in range(F):

            db[f] += np.sum(dout[n, f])

            for h_ı in range(H_ı):

                for w_ı in range(W_ı):

                    dw[f] += x_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW] * dout[n, f, h_ı, w_ı]

                    dx_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW] += w[f] * dout[n, f, h_ı, w_ı]

    dx = dx_pad[:, :, pad : pad+H, pad : pad+W] #updating according to dimensions


# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

return dx, dw, db
```

First, we get the parameters x, w, b and conv_param from cache.

x, w, b, conv_param = cache

Then as we did in forward; We take the input dimensions and weight dimensions.
N, C, H, W = x.shape
F, C, HH, WW = w.shape

Then we take stride and pad params.

stride, pad = conv_param['stride'], conv_param['pad']

Finally, we take the backward dimensions we found in the forward phase.

N, F, H_ı, W_ı = dout.shape

I use "H_ı" instead of H’ in the program and I use “W_ı” instead of W’.

padding = [(0, 0), (0, 0), (pad, pad), (pad, pad)]
x_pad = np.pad(x, padding, 'constant', constant_values=0)

And then I use np.pad function for padding. We create matrix that padded.

The backward pass for a convolution operation (for both the data and the weights) is also a convolution.

dx_pad = np.zeros_like(x_pad)

dw = np.zeros_like(w)

db = np.zeros_like(b)

Let's fill out now.

We walk through the matrix using the weight and height indices for each matrix in each channel in each example.

dw = x * dout

While doing this, we sum the values in each window (height * weight) and assign them to dw in that window.

dw[f] += x_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW] * dout[n, f, h_ı, w_ı]

dx = w * dout

While doing this, we sum the values in each window (height * weight) and assign them to dw in that window.

dx_pad[n, :, h_ı * stride : h_ı * stride + HH, w_ı * stride : w_ı * stride + WW] += w[f] * dout[n, f, h_ı, w_ı]

And then we update dx according to dimensions

dx = dx_pad[:, :, pad : pad+H, pad : pad+W]

Out:

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

5- max_pool_forward_naive

Here, we want to implementation of the forward for a max pooling layer.

def max_pool_forward_naive(x, pool_param):

out = None

<u># *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****</u>

```
    N, C, H, W = x.shape
    stride = pool_param['stride']
    pool_height = pool_param['pool_height']
    pool_width = pool_param['pool_width']

    H_ı = 1 + (H - pool_height) // stride
    W_ı = 1 + (W - pool_width) // stride

    out = np.zeros((N, C, H_ı, W_ı))

    for n in range(N):
        for h_ı in range(H_ı):
                for w_ı in range(W_ı):
                hh, ww = h_ı * stride, w_ı * stride
                out[n, :, h_ı, w_ı] = np.max(x[n, :, hh : hh + pool_height, ww : ww + pool_width],
axis=(-1, -2))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    cache = (x, pool_param)
    return out, cache
```

Inputs:

- x: Input data, of shape (N, C, H, W)
- pool_param: dictionary with the following keys:
- 'pool_height': The height of each pooling region
- 'pool_width': The width of each pooling region
- 'stride': The distance between adjacent pooling regions

First we take the input dimensions and filter weight dimensions. And then we take stride param.

N, C, H, W = x.shape

stride = pool_param['stride']

pool_height = pool_param['pool_height']

pool_width = pool_param['pool_width']

We can compute the spatial size of the output volume as a function of the input volume size (H and W), filter volume size(pool_height and pool_width), the stride with which they are applied (stride), and the amount of zero padding used (padding) on the border.

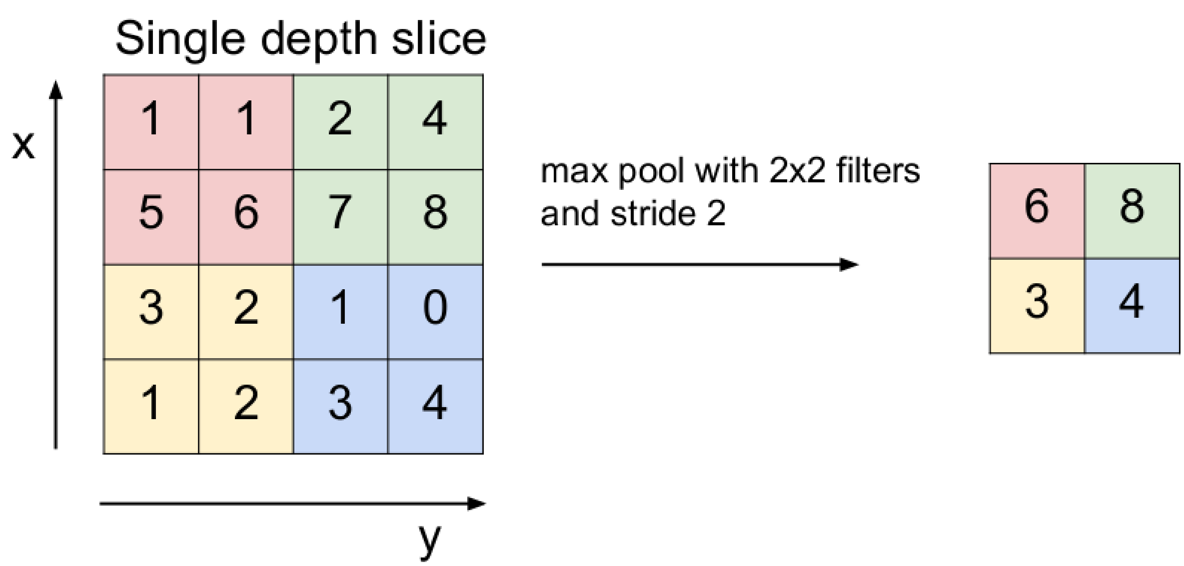
H' = 1 + (H - pool_height) / stride

W' = 1 + (W - pool_width) / stride

I use "H_ı" instead of H' in the program and I use “W_ı” instead of W’.

out = np.zeros((N, C, H_ı, W_ı))

Let's fill out now. First, we create out as a array with the dimensions N, H_ı and W_ı.



We will get the largest values in the (pool_height x pool_weight) matrix, as we apply max pooling.

We walk through the matrix using the weight and height indices for each matrix in each channel in each example.

We short the h_ı * stride and w_ı * stride statements because the operations are a bit long.

hh, ww = h_ı * stride, w_ı * stride

out[n, :, h_ı, w_ı] = np.max(x[n, :, hh : hh + pool_height, ww : ww + pool_width])

Out:

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

6- max_pool_backward_naive function

Here, we want to implementation of the backward for a max-pooling layer.

Inputs:

- dout: Upstream derivatives
- cache: A tuple of (x, pool_param) as in the forward pass.

```
def max_pool_backward_naive(dout, cache):
    dx = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    x, pool_param = cache
    N, C, H, W = x.shape

    stride = pool_param['stride']

    pool_height = pool_param['pool_height']
    pool_width = pool_param['pool_width']

    H_ı = 1 + (H - pool_height) // stride
    W_ı = 1 + (W - pool_width) // stride

    dx = np.zeros_like(x)

    for n in range(N):
        for c in range(C):
            for h_ı in range(H_ı):
                for w_ı in range(W_ı):
                    hh, ww = h_ı * stride, w_ı * stride
```

index = np.unravel_index(np.argmax(x[n, c, hh : hh + pool_height, ww :ww + pool_width], axis=None),(pool_height, pool_width))

dx[n, c, hh : hh + pool_height, ww :ww + pool_width][index] = dout[n, c, h_ı, w_ı]

*****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

return dx

First, we get the parameters x, pool_param from cache.

x, pool_param = cache

Then as we did in forward; We take the input dimensions.
N, C, H, W = x.shape

Then we take stride params.

stride = pool_param['stride']

for spatial dimesions at max-pooling layer, we use ;

H' = 1 + (H - pool_height) / stride

W' = 1 + (W - pool_width) / stride

I use "H_ı" instead of H’ in the program and I use “W_ı” instead of W’.

We create dx like x dimensions.

dx = np.zeros_like(x)

Let's fill out now.

We walk through the matrix using the weight and height indices for each matrix in each channel or depth in each example.

We short the h_ı * stride and w_ı * stride statements because the operations are a bit long.

hh, ww = h_ı * stride, w_ı * stride

Now we would normally take the largest value in the window in x and write it out. We can take the index of this value and find the dimensions of x with the function np.unravel_index. Then we throw out dout the dx matrix of the dimensions we found.

ind = np.unravel_index(np.argmax(x[n, c, hh : hh + pool_height, ww :ww + pool_width], axis=None),(pool_height, pool_width))

dx[n, c, hh : hh + pool_height, ww :ww + pool_width][ind] = dout[n, c, h_ı, w_ı]

Out:

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

7-

```
Testing conv_forward_fast:
Naive: 6.199742s
Fast: 0.024439s
Speedup: 253.681313x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 13.676021s
Fast: 0.015082s
Speedup: 906.756054x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  0.0
```

```
Testing pool_forward_fast:
Naive: 0.155627s
fast: 0.002014s
speedup: 77.257072x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.361706s
fast: 0.011060s
speedup: 32.704730x
dx difference:  0.0
```

We see acceleration in both convolutional forward and backward and max-pooling forward and backward. Of course, it is remarkable that the acceleration in max-pooling operations is less than the other.

8- Convolutinal Sandwich Layers

```
Testing conv_relu_pool
dx error:  9.591132621921372e-09
dw error:  5.802391137330214e-09
db error:  1.0146343411762047e-09

Testing conv_relu:
dx error:  1.5218619980349303e-09
dw error:  2.702022646099404e-10
db error:  1.451272393591721e-10
```

9- ThreeLayerConvNet class

A three-layer convolutional network with the following architecture:

conv - relu - 2x2 max pool - affine - relu - affine – softmax

```
def __init__(self, input_dim=(3, 32, 32), num_filters=32, filter_size=7,
             hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=0.0,
             dtype=np.float32):
    self.params = {}
    self.reg = reg
    self.dtype = dtype
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    C, H, W = input_dim

    # Convolutional layer
    W1 = np.random.normal(0, weight_scale, size=(num_filters, C, filter_size, filter_size))
    b1 = np.zeros(shape=(num_filters,))



    affine_dim = int(num_filters* (1 + (H - 2)/2) * ( 1 + (W - 2)/2) ) # max pooling, stride is taken as 1
    #print(affine_dim)
    W2 = np.random.normal(0, weight_scale, size=(affine_dim, hidden_dim))
    b2 = np.zeros(shape=(hidden_dim,))
```

```
# Output affine layer
W3 = np.random.normal(0, weight_scale, size=(hidden_dim, num_classes))
b3 = np.zeros(shape=(num_classes,))

self.params['W1'], self.params['b1'] = W1, b1
self.params['W2'], self.params['b2'] = W2, b2
self.params['W3'], self.params['b3'] = W3, b3



# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for k, v in self.params.items():
self.params[k] = v.astype(dtype)
```

First we take input _dim.

C, H, W = input_dim

Then we create Convolutional layer. For this, we initialize weights Gaussian with standard deviation equal to weight_scale and we initialize biases zero. np.random.normal draw random samples from a Gaussian distribution.

W1 = np.random.normal(0, weight_scale, size=(num_filters, C, filter_size, filter_size))

b1 = np.zeros(shape=(num_filters,))

Here we look at the loss function as described in the instruction.

==>pass pool_param to the forward pass for the max-pooling layer

==>pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

H' = 1 + (H - pool_height) / stride => 1 + (H - 2) / 2

W' = 1 + (W - pool_width) / stride => 1 + (W - 2) / 2

affine_dim = int(num_filters* (1 + (H - 2)/2) * (1 + (W - 2)/2))

After applying convolution and max pooling, we calculate the dimensions of the result and send it to the affine layer.

```
W2 = np.random.normal(0, weight_scale, size=(affine_dim, hidden_dim))
b2 = np.zeros(shape=(hidden_dim,))
```

Ardından, boyutu hidden_dim olan ve 10 sınıf için olan 3. katmanı başlatıyoruz.

```
# Output affine layer
 W3 = np.random.normal(0, weight_scale, size=(hidden_dim, num_classes))
  b3 = np.zeros(shape=(num_classes,))
```

```
def loss(self, X, y=None):
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    out_conv_relu_pool, cache_conv_relu_pool = conv_relu_pool_forward(X, W1, b1,
conv_param, pool_param)
    out_affine_relu, cache_affine_relu = affine_relu_forward(out_conv_relu_pool, W2, b2)

    scores, cache_scores = affine_forward(out_affine_relu, W3, b3)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Here we implement the forward pass for the three-layer convolutional net, computing the class scores for X and storing them in the scores variable.

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        loss, dscores = softmax_loss(scores, y)
        loss += 0.5 * self.reg * np.sum(W1 * W1)
        loss += 0.5 * self.reg * np.sum(W2 * W2)
        loss += 0.5 * self.reg * np.sum(W3 * W3)

        dout, dW3, db3 = affine_backward(dscores, cache_scores)
        dW3 += self.reg * W3

        dout, dW2, db2 = affine_relu_backward(dout, cache_affine_relu)
```

```
dW2 += self.reg * W2

dout, dW1, db1 = conv_relu_pool_backward(dout, cache_conv_relu_pool)
dW1 += self.reg * W1

grads['W3'], grads['b3'] = dW3, db3
grads['W2'], grads['b2'] = dW2, db2
grads['W1'], grads['b1'] = dW1, db1

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

First we send our scores and real values (y) to softmax_loss. Then we apply L2 regulazition to each weight.

Then we apply the backward functions for all 3 layers.

Finally, we add the parameters we find to a dictionary.

Out:

```
Initial loss (no regularization):  2.302586071243987
Initial loss (with regularization):  2.508255635671795
```

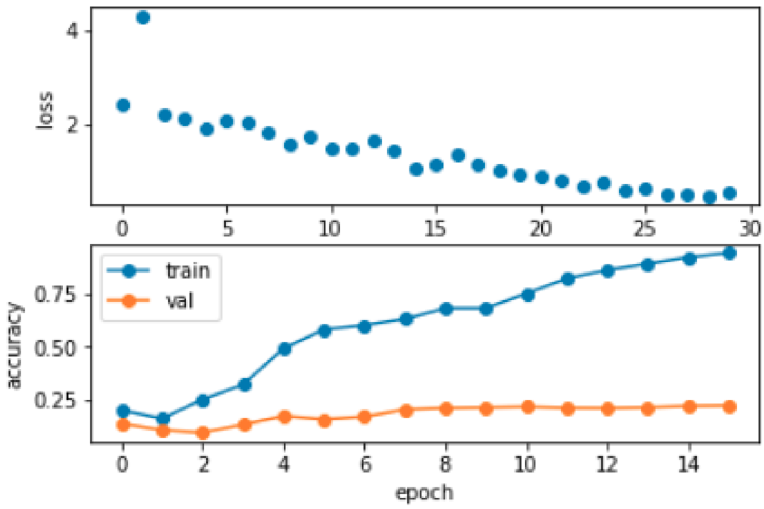
Gradient Check Result:

```
W1 max relative error: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10
```

10. Overfit small data, see it on the training-validation plot

```
(Epoch 10 / 15) train acc: 0.750000; val_acc: 0.216000
(Iteration 21 / 30) loss: 0.866430
(Iteration 22 / 30) loss: 0.785418
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.211000
(Iteration 23 / 30) loss: 0.657135
(Iteration 24 / 30) loss: 0.733124
(Epoch 12 / 15) train acc: 0.860000; val_acc: 0.210000
(Iteration 25 / 30) loss: 0.589628
(Iteration 26 / 30) loss: 0.603623
(Epoch 13 / 15) train acc: 0.890000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.515211
(Iteration 28 / 30) loss: 0.497358
(Epoch 14 / 15) train acc: 0.920000; val_acc: 0.219000
(Iteration 29 / 30) loss: 0.466170
(Iteration 30 / 30) loss: 0.548365
(Epoch 15 / 15) train acc: 0.940000; val_acc: 0.221000
```

When we override small data, we saw validation accuracy of 0.221.

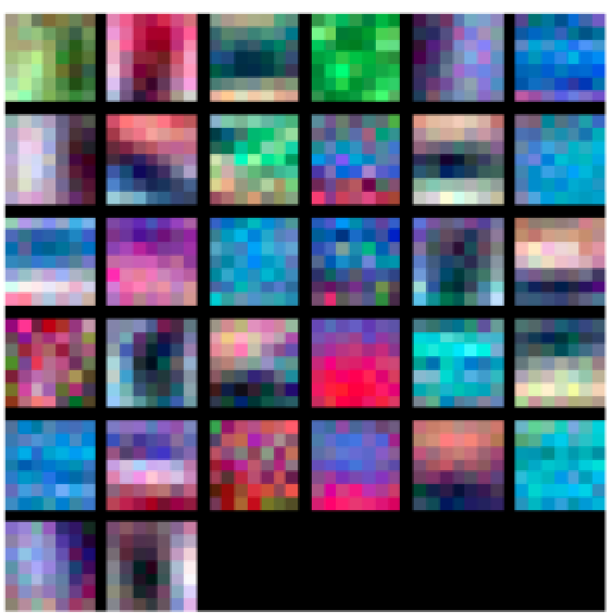


11. Now train your convolutional network

After train our convolution network, as a result, we saw validation accuracy of 0.547

```
(Iteration 901 / 980) loss: 1.285789
(Iteration 921 / 980) loss: 1.353234
(Iteration 941 / 980) loss: 1.348549
(Iteration 961 / 980) loss: 1.443794
(Epoch 1 / 1) train acc: 0.560000; val_acc: 0.547000
```

We can see first layer convolutional filters from the trained network.



Thank you so much for everything for these assignments, for your words after the questions. It was you who remained undecided about whether to do the 12th question. I watched the lord of the rings and it's a movie series that I love, but I couldn't understand the post there. Thank you again for everything.