CSE 559A/Fall 2020. Problem Set 5 Solution Key.
It is a violation of the academic integrity policy to scan, copy, or otherwise share these solutions

1. (a)

```
def get_cluster_centers(im,num_clusters):
    h,w,c = im.shape

    spacing = int(np.sqrt((h*w)/num_clusters))
    x_centers,y_centers = np.meshgrid(np.arange(spacing/2,w,spacing),np.arange(spacing/2,h,spacing))
    cluster_centers = np.column_stack((y_centers.flatten(),x_centers.flatten())).astype('int')

    grads = get_gradients(im)
    for cluster in range(num_clusters):
        center = cluster_centers[cluster]
        local_grads = grads[center[0]-1:center[0]+2,center[1]-1:center[1]+2]
        y_min,x_min = np.where(local_grads==np.min(local_grads))
        cluster_centers[cluster,0] = center[0] + np.median(y_min) - 1
        cluster_centers[cluster,1] = center[1] + np.median(x_min) - 1
    return cluster_centers
```

(b)

```
def slic(im,num_clusters,cluster_centers):
    h,w,c = im.shape
    spatial_weight = 15

    augmented_im = np.empty((im.shape[0],im.shape[1],im.shape[2]+2))
    augmented_im[:,:,:3] = im
    xx,yy = np.meshgrid(np.arange(im.shape[1]),np.arange(im.shape[0]))
    augmented_im[:,:,3] = yy
    augmented_im[:,:,4] = xx

    u = augmented_im[cluster_centers[:,0],cluster_centers[:,1],:]
    L = np.zeros((h,w))
    min_dist = np.full((h,w),np.inf)
    window_size = int(np.sqrt(h*w/num_clusters))
    num_changed = h*w
    num_iters = 0
    while num_changed > 0 and num_iters < 100:
        num_iters += 1
        for cluster in range(num_clusters):
            center = cluster_centers[cluster]
            y0 = int(np.maximum(0,center[0] - window_size))
            x0 = int(np.maximum(0,center[1] - window_size))
            y1 = int(np.minimum(h,center[0] + window_size))
            x1 = int(np.minimum(w,center[1] + window_size))

            dist = augmented_im[y0:y1,x0:x1,:] - u[cluster,:]; dist = dist**2
            dist[:,:,3:] = dist[:,:,3:] * spatial_weight
            dist = np.sum(dist,axis=2)

            update_y_inds,update_x_inds = np.where(dist<min_dist[y0:y1,x0:x1])
            adjusted_y = update_y_inds+y0; adjusted_x = update_x_inds+x0
            num_changed = adjusted_y.shape[0]

            min_dist[adjusted_y,adjusted_x] = dist[update_y_inds,update_x_inds]
            L[adjusted_y,adjusted_x] = cluster
        for cluster in range(num_clusters):
            cluster_y,cluster_x = np.where(L==cluster)
            u[cluster,:] = np.mean(augmented_im[cluster_y,cluster_x,:],axis=0)
            cluster_centers[cluster] = u[cluster,3:]
    return L
```

2. (a) The limits of the uniform distribution are chosen that way so that the individual weights have zero mean and variance equal to 1/ number of "inputs" that every output activation will depend on. (Remember that for a random variable distributed uniformly between (a, b), the mean is $(a + b)/2$, and the variance is $(b - a)^2/12$). The bias is initialized to be 0. Assuming that for any layer, all input activations can be modeled as i.i.d with zero mean and unit variance, this tries to make every output activation also be zero mean and unit variance. So if the output $y = \sum_i x_i w_i$ for a set of inputs x_i (which will be all channels, as well as a spatial window for convolutional layers), and weights w_i, the goal is to have $\mathbb{E}y = 0$ and $\mathbb{E}y^2 = 1$.

(b)

```
def init_momentum():
    for p in params:
        p.grad_hist = np.float32(0)
def momentum(lr,mom=0.9):
    for p in params:
        p.grad_hist = p.grad_hist*mom + p.grad
        p.top = p.top - lr*p.grad_hist
```

3.

```
    def forward(self):
        B,H,W,C1 = self.x.top.shape
        KH,KW,_,C2 = self.k.top.shape
        top = np.zeros([B,H-KH+1,W-KW+1,C2])
        for i in range(KH):
            for j in range(KW):
                xcrop = self.x.top[:,i:(H-KH+1+i),j:(W-KW+1+j),:].reshape([-1,C1])
                xcrop = np.matmul(xcrop,self.k.top[i,j,:,:])
                top = top + np.reshape(xcrop,top.shape)

        self.top = top

    def backward(self):
        B,H,W,C1 = self.x.top.shape
        KH,KW,_,C2 = self.k.top.shape
        ygrad = np.reshape(self.grad,[-1,C2])

        if self.x in ops or self.x in params:
            xgrad = np.zeros_like(self.x.top)
            for i in range(KH):
                for j in range(KW):
                    xij = np.matmul(ygrad,self.k.top[i,j,:,:].T)
                    xij = np.reshape(xij,[B,H-KH+1,W-KW+1,C1])
                    xgrad[:,i:(H-KH+1+i),j:(W-KW+1+j),:] =  xgrad[:,i:(H-KH+1+i),j:(W-KW+1+j),:] + xij
            self.x.grad = self.x.grad + xgrad

        if self.k in ops or self.k in params:
            kgrad = np.zeros_like(self.k.top)
            for i in range(KH):
                for j in range(KW):
                    xij = self.x.top[:,i:(H-KH+1+i),j:(W-KW+1+j),:]
                    xij = np.reshape(xij,[-1,C1])
                    kgrad[i,j,:,:] = kgrad[i,j,:,:] + np.matmul(xij.T,ygrad)
            self.k.grad = self.k.grad + kgrad
```