Model Parameters and Hyper Parameters

```
#initialise parameters and hparameters
hparameters = initialize_hparameters(3, 1, 1)
parameters = initialize_parameters((3,3,3,32), (3,3,32,64), (3,3,64,128), (256, 86528), (10, 256))
```

```
"""## **Initialise Parameters function**"""

def initialize_parameters(w1_s, w2_s, w3_s, w4_s, w5_s):

  """ w1_s/w2_s/w3_s is of the form (f,f,n_C_prev,n_C)
      w4_s is of the form (n_hidden_units, length of output from prev layer)
      w5_s is of the form (n_output, length of output from prev layer )

      b1_s/b2_s/b3_s is of the form (1,1,1,n_C)
      b4_s is of the form (n_hidden_units, 1)
      b5_s is of the form (n_output, 1)
  """
  np.random.seed(42)

  W1 = np.random.randn(w1_s[0], w1_s[1], w1_s[2], w1_s[3]) * 0.01
  b1 = np.zeros(shape=(1, 1, 1, w1_s[3]))

  W2 = np.random.randn(w2_s[0], w2_s[1], w2_s[2], w2_s[3] ) * 0.01
  b2 = np.zeros(shape=(1,1,1, w2_s[3]))

  W3 = np.random.randn(w3_s[0], w3_s[1], w3_s[2], w3_s[3]) * 0.01
  b3 = np.zeros(shape=(1,1,1, w3_s[3]))

  W4 = np.random.randn(w4_s[0], w4_s[1]) * 0.01
  b4 = np.zeros(shape=(w4_s[0], 1))

  W5 = np.random.randn(w5_s[0], w5_s[1]) * 0.01
  b5 = np.zeros(shape=(w5_s[0], 1))

  parameters = {"W1": W1,
                "b1": b1,
                "W2": W2,
                "b2": b2,
                "W3": W3,
                "b3": b3,
                "W4": W4,
                "b4": b4,
                "W5": W5,
                "b5": b5}

  return parameters
```

Location - helper_functions.py

```
"""## **Initialize hparameter**"""

def initialize_hparameters(f, stride, pad):
  hparameters = {"f": f,
                 "stride": stride,
                 "pad": pad
                 }
  return hparameters
```

Location - helper_functions.py

Linear Helper Functions Explained

```
"""## **Unflattening the variable**"""

def unflatten(A_prev, cache):
  A = cache
  a_prev = A_prev.T
  A_unflat = a_prev.reshape(A.shape)
  return A_unflat
```

Location – helper_functions.py

```
"""## **One Hot Encoding Function**"""

def one_hot_encoding(A):

  A = np.squeeze(A.T)
  a_onehot = np.zeros((A.size , A.max() + 1))
  a_onehot[np.arange(A.size), A] = 1
  return a_onehot
```

Location - helper_functions.py

```
"""## **Flatten the input**"""

def flatten(A_prev):
  cache = A_prev
  A_prev_flatten = A_prev.reshape(A_prev.shape[0], -1).T
  return A_prev_flatten, cache
```

Location – Linear_helper_functions.py

Activation Functions defined using equations

```
def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the backward pass efficiently
    """

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache
```

Location – Linear_helper_functions.py

Backward Activation Functions defined for the above activations

```
def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ
```

Location – Linear_helper_functions.py

For ReLU backwards, rather than implementing the actual derivative for ReLU which basically makes all derivative 0 or 1. I only passed back values that were non zero after implementing relu on the input Z vector.

```
def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ
```

Location – Linear_helper_functions.py

Linear forward activation:

```
def linear_forward(A, W, b):
    """
    A -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Z -- the input of the activation function, also called pre-activation parameter
    cache -- a python dictionary containing "A", "W" and "b" ; stored for computing the backward pass efficiently
    """

    Z = W.dot(A) + b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

Location – Linear_helper_functions.py

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    A_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the post-activation value
    cache -- a python dictionary containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

Location – Linear_helper_functions.py

Computing Cost:

```
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (n_class, number of examples)
    Y -- true "label" vector shape (n_class, number of examples)

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]

    # Compute loss from aL and y.
    cost = (1./m) * np.sum(-np.multiply(Y,np.log(AL).T) - np.multiply(1-Y, np.log(1-AL).T))

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).


    return cost
```

Location – Linear_helper_functions.py

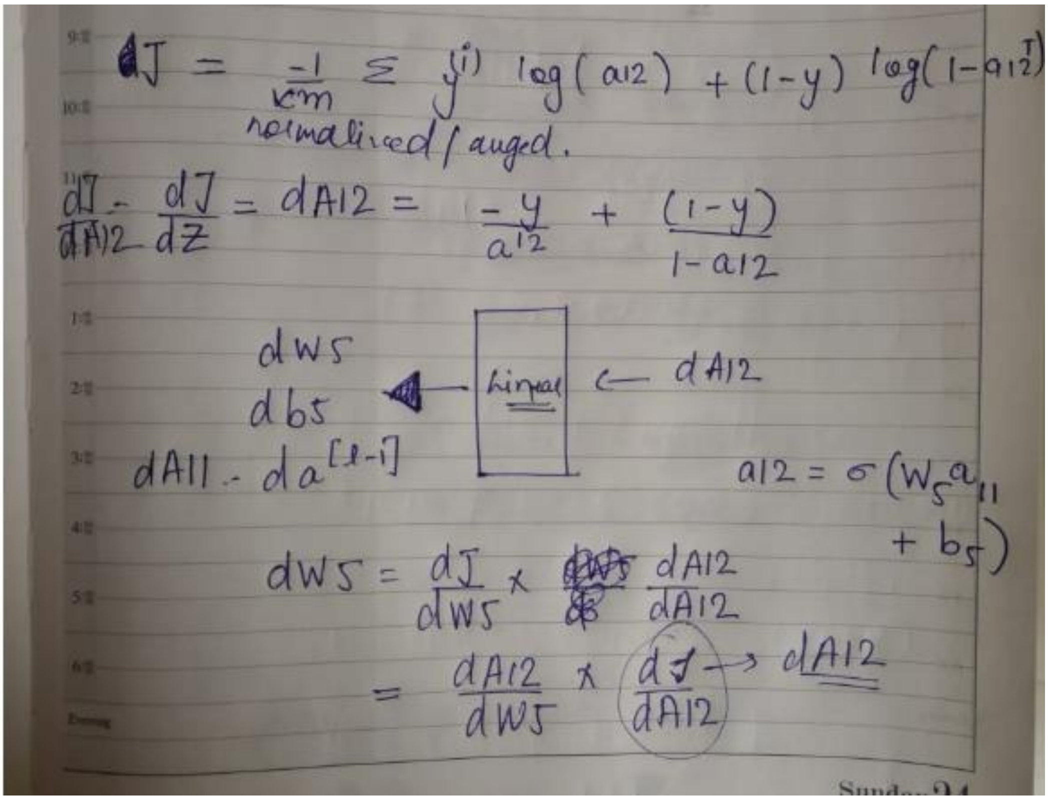
Linear Backward Activation:

```
def linear_backward(dZ, cache):


    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1./m * np.dot(dZ,A_prev.T)
    db = 1./m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T,dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

Location – Linear_helper_functions.py

```
def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for computing backward propagation efficiently
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

Calculating derivative w.r.t linear block



$dA12 \rightarrow \sigma$ - relu

$A12 = \sigma(W_5 a_{11} + b_5)$

$\Rightarrow \frac{dA12}{\partial W5} = A_{11}$

$= \frac{1}{m}(dA12 \times (A_{11})^T) \rightarrow$ normalize.

$$\boxed{dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}}$$

similarity:

$$db = \frac{1}{m} \Sigma dZ^{[l]}$$

$$dA^{[l-1]} = W^{[l]T} dA12^{[l]}$$

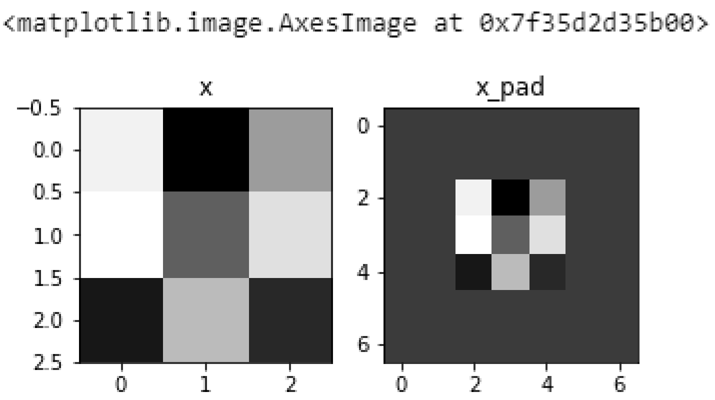
sending back the derivative

Convolutional Helper Functions Explained

Padding and single step convolution for conv_forward:

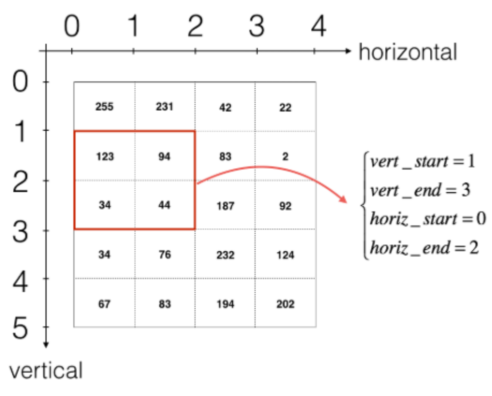
```
"""## **Zero Padding Function**"""

def zero_pad(X, pad):
  X_pad = np.pad(X, ((0,0),(pad,pad),(pad,pad),(0,0)), mode = 'constant', constant_values = (0,0))
  return X_pad

"""## **Single Step Convolution**"""

def conv_single_step(a_slice_prev, W, b):
  s = np.multiply(a_slice_prev, W)
  Z = np.sum(s)
  Z = Z + float(b)
  return Z
```

Location – conv_helper_functions.py



Convolution Forward:

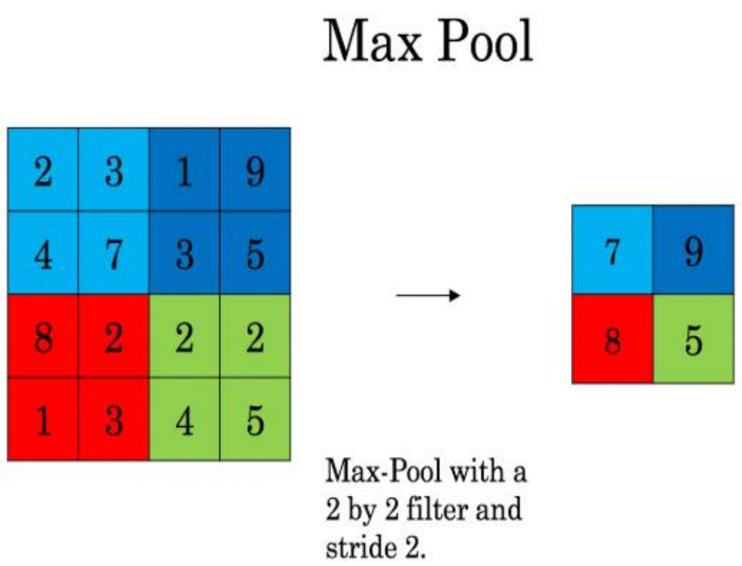
Logic behind applying mask (image in the right).



```
"""## **Convolution Forward**"""

def conv_forward(A_prev, W, b, hparameters):

  (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

  (f, f, n_C_prev, n_C) = W.shape

  stride = hparameters['stride']
  pad = hparameters['pad']

  n_H = int((n_H_prev - f + 2*pad)/stride) + 1
  n_W = int((n_W_prev - f + 2*pad)/stride) + 1

  Z = np.zeros((m, n_H, n_W, n_C))

  A_prev_pad = zero_pad(A_prev, pad)

  for i in range(m):
    a_prev_pad = A_prev_pad[i,:,:,:]
    for h in range(n_H):
      vert_start = h*stride
      vert_end = h*stride + f

      for w in range(n_W):
        horiz_start = w*stride
        horiz_end = w*stride + f

        for c in range(n_C):
          a_slice_prev = a_prev_pad[ vert_start:vert_end, horiz_start:horiz_end, :]

          weights = W[:,:,:,c]
          biases = b[:,:,:,c]
          Z[i,h,w,c] = conv_single_step(a_slice_prev, weights, biases)
  assert(Z.shape == (m,n_H, n_W, n_C))

  cache = (A_prev, W, b, hparameters)

  return Z, cache
```

Location – conv_helper_functions.py

Pool forward



```
"""## **Pool Forward Layer**"""

def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implements the forward pass of the pooling layer

    Arguments:
    A_prev -- Input data, numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    hparameters -- python dictionary containing "f" and "stride"
    mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

    Returns:
    A -- output of the pool layer, a numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache used in the backward pass of the pooling layer, contains the input and hparameters
    """

    # Retrieve dimensions from the input shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve hyperparameters from "hparameters"
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Define the dimensions of the output
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Initialize output matrix A
    A = np.zeros((m, n_H, n_W, n_C))

    ### START CODE HERE ###
    for i in range(m):                         # loop over the training examples
        for h in range(n_H):                     # loop on the vertical axis of the output volume
            # Find the vertical start and end of the current "slice" (â‰ˆ2 lines)
            vert_start = h*stride
            vert_end = h*stride + f

            for w in range(n_W):                 # loop on the horizontal axis of the output volume
                # Find the vertical start and end of the current "slice" (â‰ˆ2 lines)
                horiz_start = w*stride
                horiz_end = w*stride +f

                for c in range (n_C):            # loop over the channels of the output volume

                    # Use the corners to define the current slice on the ith training example of A_prev, channel c. (â‰ˆ1 line)
                    a_prev_slice = A_prev[i, vert_start:vert_end , horiz_start:horiz_end, c]

                    # Compute the pooling operation on the slice.
                    # Use an if statement to differentiate the modes.
                    # Use np.max and np.mean.
                    if mode == "max":
                        A[i, h, w, c] = np.max(a_prev_slice)
                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_prev_slice)

    ### END CODE HERE ###

    # Store the input and hparameters in "cache" for pool_backward()
    cache = (A_prev, hparameters)

    # Making sure your output shape is correct
    assert(A.shape == (m, n_H, n_W, n_C))
```

Location – conv_helper_functions.py

Convolution Backwards:

```
"""## **Convolution Backward Pass**"""

def conv_backward(dZ, cache):

  (A_prev, W, b, hparameters) = cache
  (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
  (f,f,n_C_prev, n_C) = W.shape

  stride = hparameters['stride']
  pad = hparameters['pad']

  (m,n_H,n_W,n_C) = dZ.shape

  dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
  dW = np.zeros((f,f,n_C_prev, n_C))
  db = np.zeros((1,1,1,n_C))

  A_prev_pad = zero_pad(A_prev,pad)
  dA_prev_pad = zero_pad(dA_prev, pad)

  for i in range(m):
    a_prev_pad = A_prev_pad[i,:,:,:]
    da_prev_pad = dA_prev_pad[i,:,:,:]

    for h in range(n_H):
      for w in range(n_W):
        for c in range(n_C):

          vert_start = h*stride
          vert_end = h*stride + f
          horiz_start = w*stride
          horiz_end = w*stride + f

          a_slice = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end,:]

          da_prev_pad[vert_start:vert_end, horiz_start:horiz_end,:] += W[:,:,:,c]*dZ[i,h,w,c]
          dW[:,:,:,c] += a_slice*dZ[i,h,w,c]
          db[:,:,:,c] += dZ[i,h,w,c]

    dA_prev[i,:,:,:] = da_prev_pad[pad:-pad,pad:-pad,:]

  assert(dA_prev.shape == (m,n_H_prev, n_W_prev, n_C_prev))

  return dA_prev, dW, db
```

Location – conv_helper_functions.py

d_Aprev ⇐ conv layer ⇐ dA/dZ

dW

db

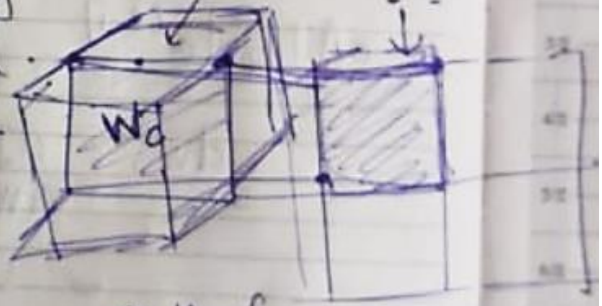
(f, f, n_C_prev, n_C)

(m, n_H, n_W, n_C)

$$dA = dA + \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw}$$

dA: (m, n_H_prev, n_W_prev, n_C_prev)

dA: accumulating w.r.t all Wc used.



n_H = f

$$dW_c \mathrel{+}= \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw}$$

accumulating grad w.r.t all slices

a slice used to generate Z_{ij}

$$db = \sum_{h} \sum_{w} dZ_{hw}$$

similar to Linear back prop.

Pool Backwards:

1) Create a mask
2) Apply mask to the incoming gradient

```
def create_mask_from_window(x):
  mask = (x ==np.max(x))

  return mask

def distribute_value(dZ, shape):
  (n_H,n_W) = shape
  average = np.sum(dZ)/(n_H*n_W)
  a = np.ones((n_H,n_W))*average

  return a
```

```
def pool_backward(dA, cache, mode = "max"):
    """
    Implements the backward pass of the pooling layer

    Arguments:
    dA -- gradient of cost with respect to the output of the pooling layer, same shape as A
    cache -- cache output from the forward pass of the pooling layer, contains the layer's input and hparameters
    mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

    Returns:
    dA_prev -- gradient of cost with respect to the input of the pooling layer, same shape as A_prev
    """

    ### START CODE HERE ###

    # Retrieve information from cache (≈1 line)
    (A_prev, hparameters) = cache

    # Retrieve hyperparameters from "hparameters" (≈2 lines)
    stride = hparameters["stride"]
    f = hparameters["f"]

    # Retrieve dimensions from A_prev's shape and dA's shape (≈2 lines)
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape

    # Initialize dA_prev with zeros (≈1 line)
    dA_prev = np.zeros(A_prev.shape)

    for i in range(m):                       # loop over the training examples
        # select training example from A_prev (≈1 line)
        a_prev = A_prev[i]
        for h in range(n_H):                   # loop on the vertical axis
            for w in range(n_W):               # loop on the horizontal axis
                for c in range(n_C):           # loop over the channels (depth)
                    # Find the corners of the current "slice" (≈4 lines)
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Compute the backward propagation in both modes.
                    if mode == "max":
                        # Use the corners and "c" to define the current slice from a_prev (≈1 line)
                        a_prev_slice = a_prev[vert_start:vert_end, horiz_start:horiz_end, c]
                        # Create the mask from a_prev_slice (≈1 line)
                        mask = create_mask_from_window(a_prev_slice)
                        # Set dA_prev to be dA_prev + (the mask multiplied by the correct entry of dA) (≈1 line)
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] += np.multiply(mask, dA[i, h, w, c])

                    elif mode == "average":
                        # Get the value a from dA (≈1 line)
                        da = dA[i, h, w, c]
                        # Define the shape of the filter as fxf (≈1 line)
                        shape = (f, f)
                        # Distribute it to get the correct slice of dA_prev. i.e. Add the distributed value of da. (≈1 l:
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] += distribute_value(da, shape)

    ### END CODE ###

    # Making sure your output shape is correct
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev
```

Location – conv_helper_functions.py