# CSE 493G1/599G1: Deep Learning

## Section 4: Backpropagation & Convolutions [solutions]

Thanks for attending section, we hope you found it helpful!

## 0. Reference Material

Intuition for Backprop

Recall some basic facts:

1) The loss function $L$ measures how "bad" our current model is.
2) $L$ is a function of our parameters $W$.
3) We want to minimize $L$.

Thus, we update $W$ to minimize $L$ using $\frac{\partial L}{\partial W}$.

For example, if $\frac{\partial L}{\partial W_1}$ was positive, increasing $W_1$ would increase $L$. Accordingly, we'd choose to decrease $W_1$.

More generally, `weights += (-1 * step_size * gradient)`.
Unfortunately, taking the derivative $\frac{\partial L}{\partial W}$ can get extremely difficult, especially at the scale of state-of-the-art models. For instance, LLaMA 2-70B has 80 transformer layers and 70 billion parameters. Imagine taking 70 billion derivatives, with each derivative having hundreds of applications of chain rule.

Instead, we employ a technique known as **backprop**.

First, we split our function into multiple equations until there is *one operation per equation*. This process is known as **staged computation**. Next, we take the derivatives of each of these smaller equations, before finally linking them together using **chain rule**.

Equations for Convolutions

Assuming the following variables, which imply that the input image has size $(W, H, C)$,

- $W$ is the width of the input image
- $H$ is the height of the input image
- $C$ is the number of channels in the input image
- $F$ is the receptive field size (i.e., the height and width of the conv field)
- $S$ is the stride with which the convolution is applied
- $P$ is the padding
- $K$ is the depth of the conv layer (i.e., the number of filters applied)

The output will have size $\left(\frac{W-F+2P}{S}+1,\ \frac{H-F+2P}{S}+1,\ K\right)$.

The conv layer will have $K\left(F^2C+1\right)$ trainable parameters.

## 1. Compute and Conquer

For each function below, use the staged computation approach to split it into smaller equations.

(a) $f(x, y, z) = (x + y)z$

**Solution:**

Decompose the function as follows:

- $a = x + y$
- $b = z$
- $f = ab$

(b) $h(x, y, z) = (x^2 + 2y)z^3$

**Solution:**

Decompose the function as follows:

- $a = x^2$
- $b = 2y$
- $c = a + b$
- $d = z^3$
- $h = cd$

(c) $g(x, y, z) = \big(\ln(x) + \sin(y)\big)^2 + 4x$

**Solution:**

Decompose the function as follows:

- $a = \ln(x)$
- $b = \sin(y)$
- $c = a + b$
- $d = c^2$
- $f = 4x$
- $g = d + f$

## 2. Oh, node way!

For each function below:

(i) construct a computational graph

(ii) do a forward and backward pass through the graph using the provided input values

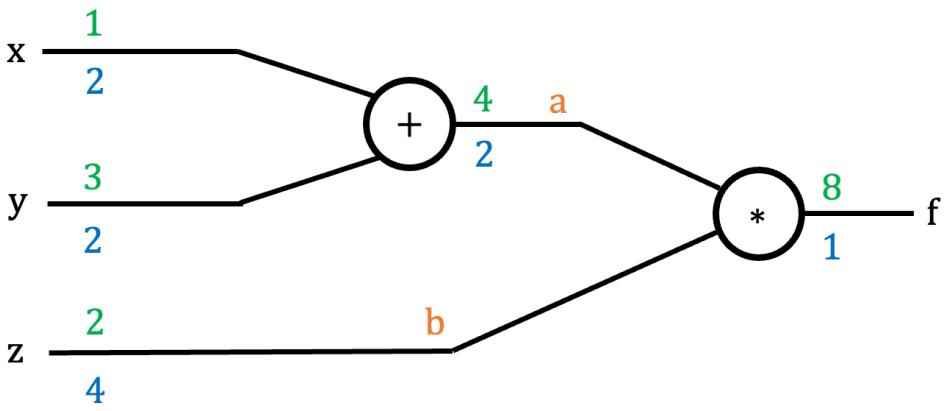
(iii) complete the Python function for a combined forward and backward pass

**Hint:** it may be useful to consider how you split these functions into smaller equations in the question above.

(a) $f(x, y, z) = (x + y)z$ with input values $x = 1$, $y = 3$, $z = 2$

**Solution:**

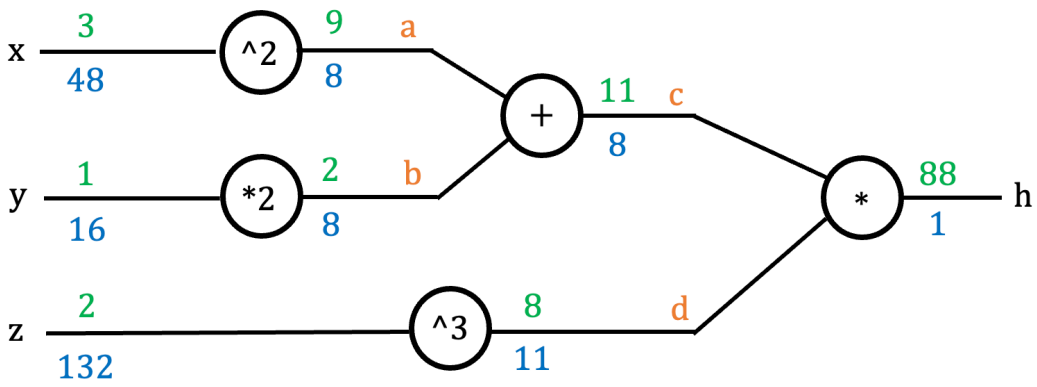
Forward pass values are displayed in green; backward pass values are displayed in blue. The orange letters correspond to the mini-equations from Question 1.




```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `fx`, `fy`, `fz` respectively
def q2a(x, y, z):
    # forward pass
    a = x + y
    b = z
    f = a * b
    out = f

    # backward pass
    ff = 1
    fb = ff * a
    fa = ff * b
    fz = fb * 1
    fx = fa
    fy = fa

    return out, fx, fy, fz
```

(b) $h(x, y, z) = (x^2 + 2y)z^3$ with input values $x = 3$, $y = 1$, $z = 2$

**Solution:**

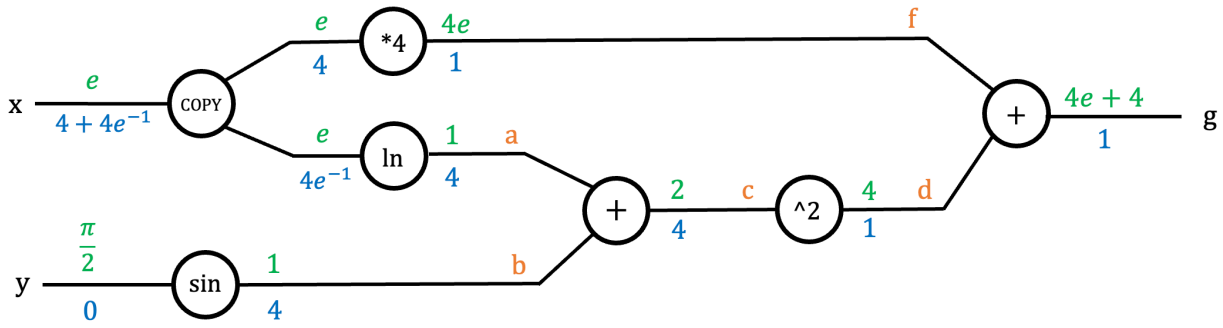



```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `hx`, `hy`, `hz` respectively
def q2b(x, y, z):
    # forward pass
    a = x ** 2
    b = 2 * y
    c = a + b
    d = z ** 3
    h = c * d
    out = h

    # backward pass -- right-most gate
    hh = 1
    hc = hh * d
    hd = hh * c

    # backward pass -- top branches
    ha = hc
    hb = hc
    hx = ha * (2 * x)
    hy = hb * 2

    # backward pass -- bottom branch
    hz = hd * (3 * (z ** 2))

    return out, hx, hy, hz
```

(c) $g(x, y, z) = \big(\ln(x) + \sin(y)\big)^2 + 4x$ with input values $x = e$, $y = \frac{\pi}{2}$, $z = 2$

**Solution:**

We omit $z$ in the computational graph below since it does not appear in the formula for $g$. It is important to realize that the gradient with respect to $z$ is $0$.




A few observations:

- We have a gradient (4) flowing back to $y$, but it dies on the last gate since $\frac{d}{dy}(\sin(y)) = \cos(x)$ and $\cos(\frac{\pi}{2}) = 0$. This is problematic since it means we don't change $y$ on this gradient descent step despite having feedback suggesting that $y$ should be decremented.
- Since $\ln(x) = \frac{1}{x}$, the local gradient associated with equation $a$ can be undefined if $x = 0$. If you were asked to implement this function and its backwards pass in Python, what are some potential workarounds you might employ?

*Python function printed on the following page.*

```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `gx`, `gy`, `gz` respectively
def q2c(x, y, z):
    # forward pass
    a = np.log(x)
    b = np.sin(y)
    c = a + b
    d = c ** 2
    f = 4 * x
    g = d +f
    out = g

    # backward pass -- right-most gate
    gg = 1
    gf = gg
    gd = gd

    # backward pass -- path via `d`
    gc = gd * (2 *c)
    ga = gc
    gb = gc
    gx_1 = ga * (x ** -1)
    gy = gb * np.cos(y)

    # backward pass -- path via `f`
    gx_2 = gf * 4

    # backward pass -- reconciliation at copy gate
    gx = gx_1 + gx_2

    # z never appears in the function, so it has no gradient
    gz = 0

    return out, gx, gy, gz
```

## 3. Sigmoid Shenanigans

Consider the Sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

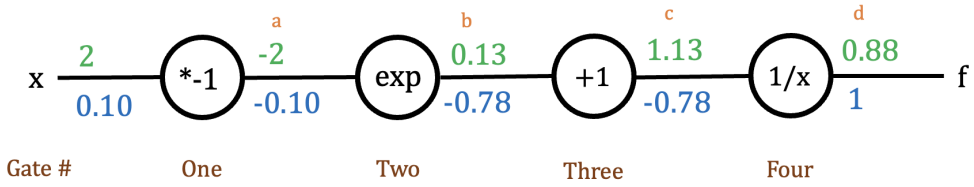
(a) Draw a computational graph and work through the backpropagation. Then, fill in the Python function. If you finish early, work through the analytical derivation for Sigmoid.

As a hint, you could split Sigmoid into the following functions:

$$a(x) = -x \qquad b(x) = e^x \qquad c(x) = 1 + x \qquad d(x) = \frac{1}{x}$$

Observe that chaining these operations gives us Sigmoid: $d(c(b(a(x)))) = \sigma(x)$.

**Solution:**




(b) Suppose $x = 2$. What would the gradient with respect to $x$ be? Feel free to use a calculator on this part.

**Solution:**

Recall that downstream = upstream $\times$ local.

At Gate Four, the upstream gradient is $1$ and the local gradient is $\frac{\partial}{\partial c}\left(\frac{1}{c}\right) = -\frac{1}{c^2} = -\frac{1}{(1.13)^2} = -0.78$. Thus, the downstream gradient is $1 \times -0.78 = -0.78$.

At Gate Three, the upstream is $-0.78$ and the local is $\frac{\partial}{\partial b}\Big(b + 1\Big) = 1$. Thus, the downstream is $-0.78 \times 1 = -0.78$.

At Gate Two, the upstream is $-0.78$ and the local is $\frac{\partial}{\partial a}\Big(e^a\Big) = e^a = e^{-2} = 0.135$. Thus, the downstream is $-0.78 \times 0.135 = -0.10$.

At Gate One, the upstream is $-0.10$ and the local is $\frac{\partial}{\partial x}\Big(-x\Big) = -1$. Thus, the downstream is $-0.10 \times -1 = 0.10$.

Therefore, $\frac{df}{dx} \approx 0.10$. We use $\approx$ here because we rounded decimals throughout our calculations.

(c) You should have gotten around 0.1. If the step size is $0.2$, what would the value of $x$ be after taking one gradient descent step? As a hint, remember that `parameters -= step_size * gradient`.

**Solution:**

Our parameter, $x$, started off at 2. Our step size was 0.2 and our gradient is 0.1. Plugging into the equation for gradient descent, the new value for $x$ is $2 - 0.2(0.1) = 2 - 0.02 = 1.98$.

(d) Implement the function below for a full forward and backward pass through Sigmoid.

**Solution:**

```
import numpy as np

# inputs:
#   - a numpy array `x`
# outputs:
#   - `out`: the result of the forward pass
#   - `fx` : the result of the backward pass
def sigmoid(x):
    # provided: forward pass with cache
    a = -x
    b = np.exp(a)
    c = 1 + b
    f = 1/c
    out = f

    # TODO: backward pass, "fx" represents df / dx
    ff = 1
    fc = ff * -1/(c**2)
    fb = fc * 1
    fa = fb * np.exp(a)
    fx = fa * -1

    return out, fx
```

## 4. A Backprop a Day Keeps the Derivative Away

Consider the following function:

$$f = \frac{\ln x \cdot \sigma\left(\sqrt{y}\right)}{\sigma\left((x+y)^2\right)}$$

Break the function up into smaller parts, then draw a computational graph and finish the Python function.

For reference, the derivative of Sigmoid is $\sigma(x) \cdot \big(1 - \sigma(x)\big)$.

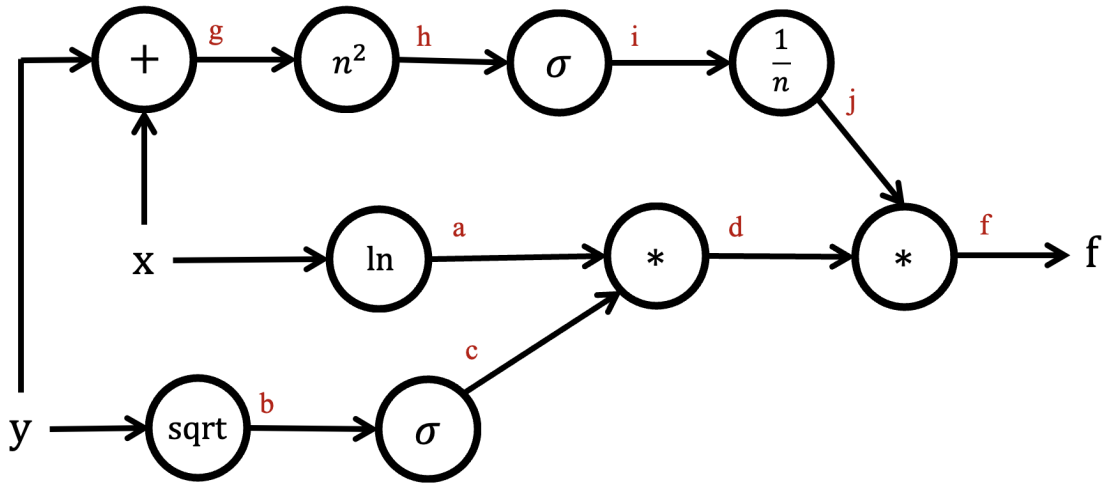
The TA solution breaks the function into 8 additional equations and rewrites $f$ in terms of 2 of those additional equations. Yours doesn't have to match this exactly.

**Solution:**

We begin by breaking the function down:

| | | | | |
|---|---|---|---|---|
| Numerator: | $a = \ln x$ | $b = \sqrt{y}$ | $c = \sigma(b)$ | $d = a \cdot c$ |
| Denominator: | $g = x + y$ | $h = g^2$ | $i = \sigma(h)$ | $j = \frac{1}{i}$ |
| Final: | $f = dj$ | | | |

Although $f = \frac{d}{i}$ is a valid, one-operation gate, we generally try to avoid quotient rule. Therefore, we introduce an extra operation, $i = \frac{1}{j}$, leaving us with $f = di$.




*Python function printed on the following page.*

```python
import numpy as np

# helper function
def sigmoid(x):
    return 1/(1 +np.exp(-x))

# inputs: numpy arrays `x`, `y`
# outputs: forward pass in `out`, gradient for x in `fx`, gradient for y in `fy`
def complex_layer(x, y):
    # forward pass
    a = np.log(x)
    b = np.sqrt(y)
    c = sigmoid(b)
    d = a * c
    g = x + y
    h = g ** 2
    i = sigmoid(h)
    j = 1 / i
    out = d * j

    # backward pass -- output gate
    ff = 1
    fd = ff * j
    fj = ff * d

    # backward pass -- top branch
    fi = fj * -1 / (i ** 2)
    fh = fi * sigmoid(h) * (1 - sigmoid(h))
    fg = fh * 2 * g
    fx_1 = fg
    fy_1 = fg

    # backward pass -- middle branch
    fa = fd * c
    fx_2 = fa / x

    # backward pass -- bottom branch
    fc = fd * a
    fb = fc * sigmoid(b) * (1 - sigmoid(b))
    fy_2 = fb / (2 * np.sqrt(y))

    # backward pass -- reconciliation
    fx = fx_1 + fx_2
    fy = fy_1 + fy_2

    return out, fx, fy
```

## 5. As Convoluting As Possible

(a) What's the formula for determining a conv layer's output size? Assume that the receptive field is a square. Define all the variables you use.

**Solution:**

For a $(W, H, C)$ input, the output of a conv layer with a $(F, F, C)$ receptive field applied with padding $P$, stride $S$, and depth $K$ is given by:

$$\left(\frac{W - F + 2P}{S} + 1, \frac{H - F + 2P}{S} + 1, K\right)$$

(b) Consider a conv layer that takes a $32 \times 32 \times 3$ input and applies a $5 \times 5 \times 3$ filter with no padding. Compute the output sizes if we use a stride of 1, 2, and 3. Then, compute the output size if we use a stride of 2 and 3 with a padding of 3.

**Hint:** certain strides may result in an invalid configuration for this conv layer.

**Solution:**

First, identify that $W = 32$, $H = 32$, $C = 3$, $F = 5$. Since $W = H$, we will only perform one of these computations.

Now, let's compute the output sizes for each of the configurations.

- $P = 0$ and $S = 1$: output size $(28, 28, 1)$ since

$$\frac{W - F + 2P}{S} + 1 = \frac{32 - 5 + 2 \cdot 0}{1} + 1 = 27 + 1 = 28$$

- $P = 0$ and $S = 2$: invalid configuration because

$$\frac{W - F + 2P}{S} + 1 = \frac{32 - 5 + 2 \cdot 0}{2} + 1 = 13.5 + 1 = 14.5 \notin \mathbb{Z}^+$$

- $P = 0$ and $S = 3$: output size $(10, 10, 1)$ since

$$\frac{W - F + 2P}{S} + 1 = \frac{32 - 5 + 2 \cdot 0}{3} + 1 = 9 + 1 = 10$$

- $P = 3$ and $S = 2$: invalid configuration because

$$\frac{W - F + 2P}{S} + 1 = \frac{32 - 5 + 2 \cdot 3}{2} + 1 = 16.5 + 1 = 17.5 \notin \mathbb{Z}^+$$

- $P = 3$ and $S = 3$: output size $(12, 12, 1)$ since

$$\frac{W - F + 2P}{S} + 1 = \frac{32 - 5 + 2 \cdot 3}{3} = 11 + 1 = 12$$

Note that, no matter what padding we apply, we cannot make a stride of 2 work.

**Note:** People will often leave out the channels dimension when writing out a filter (i.e., they might refer to a $5 \times 5 \times 3$ filter as a $5 \times 5$ filter). We will now adopt this shorthand as well.

(c) Consider the first conv layer of AlexNet, which takes an input of size $227 \times 227 \times 3$ and applies 96 separate $11 \times 11$ convolutional filters with stride of $4$ and no padding. People will sometimes write this as applying one $11 \times 11$ filter with depth $96$; it means the same thing. What are our output dimensions?

**Solution:**

Our outut will have size $(55, 55, 96)$ since we applied $96$ filters and

$$\frac{H - F}{S} + 1 = \frac{W - F}{S} + 1 = \frac{227 - 11}{4} + 1 = \frac{216}{4} + 1 = 55$$

Notice that we didn't bother including the $2P$ term in our calculation above since the convolution is applied with zero padding.

(d) Develop a formula for the number of trainable parameters in a conv layer. Assume that the receptive field is a square and that the conv layer has biases. Define all the variables you use.

**Solution:**

There are $K$ filters. Each filter can be thought of as its own linear classifier of sorts with weights and biases. The filter has size $(F, F, C)$, so it has $F \cdot F \cdot C = F^2C$ weights. By convention, the filter has 1 bias term. Combining these two, we see that each filter has $F^2C + 1$ trainable parameters. Thus, across $K$ filters, a conv layer will have $K(F^2C + 1)$ trainable parameters.

(e) Consider the first conv layer of AlexNet mentioned above. How many trainable parameters are there?

**Solution:**

$$K(CF^2 + 1) = 96(3 \cdot 11^2 + 1) = 96(3 \cdot 121 + 1) = 96(364) = 34944$$

(f) Consider a conv layer which takes a $31 \times 31 \times 5$ input and applies a $3 \times 3$ filter with depth 25, stride 2, and padding 1. How many trainable parameters does it have?

**Solution:**

$$K(CF^2 + 1) = 25(5 \cdot 3^2 + 1) = 25(5 \cdot 9 + 1) = 25(46) = 1150$$

**Takeaway:** The values you set $K$ and $F$ to (these are hyperparameters!) will have a significant impact on the number of parameters your model has. You must be careful not to add too many parameters to your model.

(g) Recall your answer for the output of AlexNet's first conv layer from above. This output is fed directly into a max pool layer which applies a 3x3 pool filter at stride 2 with no padding. What will the output size be? How many trainable parameters does this layer introduce?

**Solution:**

The pool layer takes an input of $(55, 55, 96)$. The pool layer slides a pool filter across the image, similar to how the conv layer slid a convulational filter across the image. Thus, we can use the same formula as before:

$$\frac{H-F}{S}+1=\frac{W-F}{S}+1=\frac{55-3}{2}+1=26+1=27$$

This gives us an output of size $(27, 27, 96)$. Note that the channel dimension stays constant since the pool is applied separately to each of the $96$ input channels.

This layer introduces 0 trainable parameters since the pool filter uses a fixed function ($\max$).

(h) Did the pool layer change the number of channels? Does this pattern generalize to pool layers of all sizes?

**Solution:**

Pool layers generally do NOT change the number of channels.

(i) AlexNet precipitated the deep learning revolution. Explain one of the paper's key contributions.

**Solution:**

Some of AlexNet's major contributions include:

- Although previous papers had implemented neural networks on GPUs, Alex wrote a highly-optimized GPU implementation of convolutions code and successfully parallelized training across 2 GPUs.
- Popularized the use of ReLU as an activation function.
- Popularized the use of dropout to prevent overfitting.
- Proved that, when designed correctly, deeper neural networks could perform better than the shallower neural networks that were popular at the time. (Note that AlexNet was not the first paper to demonstrate the plausibility of deep neural nets; for example, see this).

This is not a comprehensive list.

## 6. Kernel of Truth

Consider the following input matrix $I$ and filter $F$.

As an aside, you may also hear a convolution filter referred to as a kernel, mask, or convolutional matrix.

$$I = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ -1 & 0 & 1 & 2 \\ 0 & -2 & 4 & 0 \end{bmatrix} \qquad F = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

(a) Apply $F$ on $I$ with padding 0 and stride 1.

**Solution:**

Applying the convolution,

$$\begin{bmatrix} 1(1) - 1(2) + 0(4) + 2(3) & 1(2) - 1(3) + 0(3) + 2(2) & 1(3) - 1(4) + 0(2) + 2(1) \\ 1(4) - 1(3) + 0(-1) + 2(0) & 1(3) - 1(2) + 0(0) + 2(1) & 1(2) - 1(1) + 0(1) + 2(2) \\ 1(-1) + -1(0) + 0(0) + 2(-2) & 1(0) - 1(1) + 0(-2) + 2(4) & 1(1) - 1(2) + 0(4) + 2(0) \end{bmatrix}$$

Simplifying,

$$\begin{bmatrix} 5 & 3 & 1 \\ 1 & 3 & 5 \\ -5 & 7 & -1 \end{bmatrix}$$

(b) Apply $F$ on $I$ with padding 1 and stride 2.

**Solution:**

Note that after padding $I$ looks like this:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 0 \\ 0 & 4 & 3 & 2 & 1 & 0 \\ 0 & -1 & 0 & 1 & 2 & 0 \\ 0 & 0 & -2 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Applying the convolution yields,

$$\begin{bmatrix} 2 & 6 & 0 \\ -6 & 3 & 1 \\ 0 & -6 & 0 \end{bmatrix}$$

(c) Apply a max pool on $I$ with a $3 \times 3$ filter using a padding of 1 and a stride of 3.[1]

**Solution:**

$$\begin{bmatrix} 4 & 4 \\ 0 & 4 \end{bmatrix}$$

[1] In practice, pool layers are usually applied after conv layers; they are typically *not* applied directly on the input.