In [1]:
```
import os
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa

from kaggle_datasets import KaggleDatasets
import matplotlib.pyplot as plt
import numpy as np

from PIL import Image

AUTOTUNE = tf.data.experimental.AUTOTUNE
```

Problem Statement

The goal of this exercise is to train a generative adversarial network (GAN) on images of Monet paintings, and then use that network to generate novel images in the style of Money using regular photographs as input. The training data set consists of 300 256 x 256 RGB images of Monet paintings. The required output is 7,000+ images in the style of Monet, using the photos provided as input. I will be using a CycleGAN for this task.

GANs in general are composed of two primary parts: a generator and a discriminator. The generator takes input, possibly noise but also images as in this case, and attempts to create the desired output. The discriminator acts as a binary classifier and compares the generator's output to determine whether it belongs to some set of "real", labeled examples. GANs are so called because each component works against the other, the generator trying to produce better output to fool the discriminator, and the the discriminator trying to get better at distinguishing betwee "real" and generated output.

CycleGANs are a subset of GANs which introduce an element of "cycle consistency loss". This is an additional loss function that quantifies the loss from transforming an image twice, that is, from "real" input, to generated output, and back to "real" again.

EDA

In [2]:
```
GCS_PATH = KaggleDatasets().get_gcs_path()
```

In [3]:
```
MONET_FILENAMES = tf.io.gfile.glob(str(GCS_PATH + '/monet_tfrec/*.tfrec'))
print('Monet TFRecord Files:', len(MONET_FILENAMES))

PHOTO_FILENAMES = tf.io.gfile.glob(str(GCS_PATH + '/photo_tfrec/*.tfrec'))
print('Photo TFRecord Files:', len(PHOTO_FILENAMES))
```

```
Monet TFRecord Files: 5
Photo TFRecord Files: 20
```

In [4]:
```python
IMAGE_SIZE = [256, 256]

def decode_image(image):
    image = tf.image.decode_jpeg(image, channels=3)
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    image = tf.reshape(image, [*IMAGE_SIZE, 3])
    return image

def read_tfrecord(example):
    tfrecord_format = {
        "image_name": tf.io.FixedLenFeature([], tf.string),
        "image": tf.io.FixedLenFeature([], tf.string),
        "target": tf.io.FixedLenFeature([], tf.string)
    }
    example = tf.io.parse_single_example(example, tfrecord_format)
    image = decode_image(example['image'])
    return image
```

In [5]:
```python
def load_dataset(filenames, labeled=True, ordered=False):
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.map(read_tfrecord, num_parallel_calls=AUTOTUNE)
    return dataset
```

In [6]:
```python
monet_ds = load_dataset(MONET_FILENAMES, labeled=True).batch(1)
photo_ds = load_dataset(PHOTO_FILENAMES, labeled=True).batch(1)
```

In [7]:
```python
def print_imgs(directory):
    # Get the list of image file names in the directory
    image_files = os.listdir(directory)[:9]

    # Create a figure and subplot grid
    fig, axes = plt.subplots(3, 3, figsize=(10, 10))

    # Iterate over the image files and plot them in the grid
    for i, image_file in enumerate(image_files):
        # Open the image using PIL
        image_path = os.path.join(directory, image_file)
        image = Image.open(image_path)

        # Plot the image on the corresponding subplot
        row = i // 3
        col = i % 3
        axes[row, col].imshow(image)
        axes[row, col].axis('off')

    # Adjust spacing between subplots
    plt.tight_layout()

    # Display the grid of images
    plt.show()
```

Example Monets

In [8]:
```
print_imgs('/kaggle/input/monet-gan-getting-started/monet_jpg')
```

Example Photos

In [9]:
```
print_imgs('/kaggle/input/monet-gan-getting-started/photo_jpg')
```

Helper Functions

In [10]:

```
OUTPUT_CHANNELS = 3

def downsample(filters, size, apply_instancenorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    result = keras.Sequential()
    result.add(layers.Conv2D(filters, size, strides=2, padding='same',
                             kernel_initializer=initializer, use_bias=False))

    if apply_instancenorm:
        result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))

    result.add(layers.LeakyReLU())
```

```python
    return result
```

In [11]:
```python
def upsample(filters, size, apply_dropout=False):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    result = keras.Sequential()
    result.add(layers.Conv2DTranspose(filters, size, strides=2,
                                      padding='same',
                                      kernel_initializer=initializer,
                                      use_bias=False))

    result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))

    if apply_dropout:
        result.add(layers.Dropout(0.5))

    result.add(layers.ReLU())

    return result
```

Modeling

I will be using a CycleGAN to create an model that can generate images in the style of Monet. It will use both down sampling and up sampling to transform the input image data.

Downsampling is useful as it acts as feature reduction, lowering the computational demands of the model. It also allows the model to capture broad details about the image. It can also help the model achieve translational invariance, which means that features can be recognized regardless of their position within an image.

Up sampling is useful because it allows the model to capture small features of the image. It can also help with properly aligning image features in space, and avoiding some kinds of artifacts, like checkerboarding.

In [12]:
```python
def Generator():
    inputs = layers.Input(shape=[256,256,3])

    # bs = batch size
    down_stack = [
        downsample(64, 4, apply_instancenorm=False), # (bs, 128, 128, 64)
        downsample(128, 4), # (bs, 64, 64, 128)
        downsample(256, 4), # (bs, 32, 32, 256)
        downsample(512, 4), # (bs, 16, 16, 512)
        downsample(512, 4), # (bs, 8, 8, 512)
        downsample(512, 4), # (bs, 4, 4, 512)
        downsample(512, 4), # (bs, 2, 2, 512)
        downsample(512, 4), # (bs, 1, 1, 512)
    ]
```

```python
    up_stack = [
        upsample(512, 4, apply_dropout=True), # (bs, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 8, 8, 1024)
        upsample(512, 4), # (bs, 16, 16, 1024)
        upsample(256, 4), # (bs, 32, 32, 512)
        upsample(128, 4), # (bs, 64, 64, 256)
        upsample(64, 4), # (bs, 128, 128, 128)
    ]

    initializer = tf.random_normal_initializer(0., 0.02)
    last = layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                  strides=2,
                                  padding='same',
                                  kernel_initializer=initializer,
                                  activation='tanh') # (bs, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = layers.Concatenate()([x, skip])

    x = last(x)

    return keras.Model(inputs=inputs, outputs=x)
```

In [13]:

```python
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    inp = layers.Input(shape=[256, 256, 3], name='input_image')

    x = inp

    down1 = downsample(64, 4, False)(x) # (bs, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (bs, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (bs, 32, 32, 256)

    zero_pad1 = layers.ZeroPadding2D()(down3) # (bs, 34, 34, 256)
    conv = layers.Conv2D(512, 4, strides=1,
                         kernel_initializer=initializer,
                         use_bias=False)(zero_pad1) # (bs, 31, 31, 512)

    norm1 = tfa.layers.InstanceNormalization(gamma_initializer=gamma_init)(conv)

    leaky_relu = layers.LeakyReLU()(norm1)
```

```python
    zero_pad2 = layers.ZeroPadding2D()(leaky_relu) # (bs, 33, 33, 512)

    last = layers.Conv2D(1, 4, strides=1,
                         kernel_initializer=initializer)(zero_pad2) # (bs, 30, 30,

    return tf.keras.Model(inputs=inp, outputs=last)
```

In [14]:
```python
monet_generator = Generator()
photo_generator = Generator()

monet_discriminator = Discriminator()
photo_discriminator = Discriminator()
```

In [15]:
```python
class CycleGan(keras.Model):
    def __init__(
        self,
        monet_generator,
        photo_generator,
        monet_discriminator,
        photo_discriminator,
        lambda_cycle=10,
    ):
        super(CycleGan, self).__init__()
        self.m_gen = monet_generator
        self.p_gen = photo_generator
        self.m_disc = monet_discriminator
        self.p_disc = photo_discriminator
        self.lambda_cycle = lambda_cycle

    def compile(
        self,
        m_gen_optimizer,
        p_gen_optimizer,
        m_disc_optimizer,
        p_disc_optimizer,
        gen_loss_fn,
        disc_loss_fn,
        cycle_loss_fn,
        identity_loss_fn
    ):
        super(CycleGan, self).compile()
        self.m_gen_optimizer = m_gen_optimizer
        self.p_gen_optimizer = p_gen_optimizer
        self.m_disc_optimizer = m_disc_optimizer
        self.p_disc_optimizer = p_disc_optimizer
        self.gen_loss_fn = gen_loss_fn
        self.disc_loss_fn = disc_loss_fn
        self.cycle_loss_fn = cycle_loss_fn
        self.identity_loss_fn = identity_loss_fn

    def train_step(self, batch_data):
        real_monet, real_photo = batch_data

        with tf.GradientTape(persistent=True) as tape:
```

```
            # photo to monet back to photo
            fake_monet = self.m_gen(real_photo, training=True)
            cycled_photo = self.p_gen(fake_monet, training=True)

            # monet to photo back to monet
            fake_photo = self.p_gen(real_monet, training=True)
            cycled_monet = self.m_gen(fake_photo, training=True)

            # generating itself
            same_monet = self.m_gen(real_monet, training=True)
            same_photo = self.p_gen(real_photo, training=True)

            # discriminator used to check, inputing real images
            disc_real_monet = self.m_disc(real_monet, training=True)
            disc_real_photo = self.p_disc(real_photo, training=True)

            # discriminator used to check, inputing fake images
            disc_fake_monet = self.m_disc(fake_monet, training=True)
            disc_fake_photo = self.p_disc(fake_photo, training=True)

            # evaluates generator loss
            monet_gen_loss = self.gen_loss_fn(disc_fake_monet)
            photo_gen_loss = self.gen_loss_fn(disc_fake_photo)

            # evaluates total cycle consistency loss
            total_cycle_loss = self.cycle_loss_fn(real_monet, cycled_monet, self.la

            # evaluates total generator loss
            total_monet_gen_loss = monet_gen_loss + total_cycle_loss + self.identit
            total_photo_gen_loss = photo_gen_loss + total_cycle_loss + self.identit

            # evaluates discriminator loss
            monet_disc_loss = self.disc_loss_fn(disc_real_monet, disc_fake_monet)
            photo_disc_loss = self.disc_loss_fn(disc_real_photo, disc_fake_photo)

        # Calculate the gradients for generator and discriminator
        monet_generator_gradients = tape.gradient(total_monet_gen_loss,
                                                  self.m_gen.trainable_variables)
        photo_generator_gradients = tape.gradient(total_photo_gen_loss,
                                                  self.p_gen.trainable_variables)

        monet_discriminator_gradients = tape.gradient(monet_disc_loss,
                                                      self.m_disc.trainable_variabl
        photo_discriminator_gradients = tape.gradient(photo_disc_loss,
                                                      self.p_disc.trainable_variabl

        # Apply the gradients to the optimizer
        self.m_gen_optimizer.apply_gradients(zip(monet_generator_gradients,
                                                 self.m_gen.trainable_variables))

        self.p_gen_optimizer.apply_gradients(zip(photo_generator_gradients,
                                                 self.p_gen.trainable_variables))

        self.m_disc_optimizer.apply_gradients(zip(monet_discriminator_gradients,
                                                  self.m_disc.trainable_variables))
```

```
        self.p_disc_optimizer.apply_gradients(zip(photo_discriminator_gradients,
                                                  self.p_disc.trainable_variables))

        return {
            "monet_gen_loss": total_monet_gen_loss,
            "photo_gen_loss": total_photo_gen_loss,
            "monet_disc_loss": monet_disc_loss,
            "photo_disc_loss": photo_disc_loss
        }
```

Define loss functions

The loss functions below are how the model determines which weights need to be updated, and with what strength and direction. The cycle loss function is the distinguishing feature of CycleGANs, and it encourages bi-directional stability of the transformation. That is, an image that is converted should be able to be converted back and appear close to the original.

In [16]:
```
def discriminator_loss(real, generated):
    real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.k

    generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss * 0.5
```

In [17]:
```
def generator_loss(generated):
    return tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.
```

In [18]:
```
def calc_cycle_loss(real_image, cycled_image, LAMBDA):
    loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

    return LAMBDA * loss1
```

In [19]:
```
def identity_loss(real_image, same_image, LAMBDA):
    loss = tf.reduce_mean(tf.abs(real_image - same_image))
    return LAMBDA * 0.5 * loss
```

Train Model

In [20]:
```
monet_generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
photo_generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

monet_discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
photo_discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```

In [21]:
```
cycle_gan_model = CycleGan(
    monet_generator, photo_generator, monet_discriminator, photo_discriminator
)
```

```
cycle_gan_model.compile(
    m_gen_optimizer = monet_generator_optimizer,
    p_gen_optimizer = photo_generator_optimizer,
    m_disc_optimizer = monet_discriminator_optimizer,
    p_disc_optimizer = photo_discriminator_optimizer,
    gen_loss_fn = generator_loss,
    disc_loss_fn = discriminator_loss,
    cycle_loss_fn = calc_cycle_loss,
    identity_loss_fn = identity_loss
)
```

In [22]:
```
cycle_gan_model.fit(
    tf.data.Dataset.zip((monet_ds, photo_ds)),
    epochs=10
)
```

```
Epoch 1/10
300/300 [==============================] - 188s 443ms/step - monet_gen_loss: 5.1248
- photo_gen_loss: 5.3362 - monet_disc_loss: 0.6581 - photo_disc_loss: 0.6323
Epoch 2/10
300/300 [==============================] - 136s 452ms/step - monet_gen_loss: 3.6282
- photo_gen_loss: 3.7505 - monet_disc_loss: 0.6608 - photo_disc_loss: 0.6269
Epoch 3/10
300/300 [==============================] - 136s 450ms/step - monet_gen_loss: 3.5729
- photo_gen_loss: 3.7091 - monet_disc_loss: 0.6514 - photo_disc_loss: 0.6208
Epoch 4/10
300/300 [==============================] - 135s 450ms/step - monet_gen_loss: 3.4259
- photo_gen_loss: 3.5794 - monet_disc_loss: 0.6454 - photo_disc_loss: 0.6067
Epoch 5/10
300/300 [==============================] - 135s 450ms/step - monet_gen_loss: 3.2899
- photo_gen_loss: 3.4240 - monet_disc_loss: 0.6340 - photo_disc_loss: 0.6087
Epoch 6/10
300/300 [==============================] - 135s 450ms/step - monet_gen_loss: 3.2282
- photo_gen_loss: 3.3212 - monet_disc_loss: 0.6202 - photo_disc_loss: 0.6074
Epoch 7/10
300/300 [==============================] - 135s 450ms/step - monet_gen_loss: 3.2189
- photo_gen_loss: 3.2947 - monet_disc_loss: 0.6179 - photo_disc_loss: 0.6124
Epoch 8/10
300/300 [==============================] - 135s 449ms/step - monet_gen_loss: 3.2154
- photo_gen_loss: 3.2573 - monet_disc_loss: 0.6098 - photo_disc_loss: 0.6141
Epoch 9/10
300/300 [==============================] - 135s 449ms/step - monet_gen_loss: 3.1956
- photo_gen_loss: 3.2087 - monet_disc_loss: 0.6068 - photo_disc_loss: 0.6199
Epoch 10/10
300/300 [==============================] - 135s 449ms/step - monet_gen_loss: 3.1709
- photo_gen_loss: 3.1810 - monet_disc_loss: 0.6047 - photo_disc_loss: 0.6138
```

Out[22]:
```
<keras.callbacks.History at 0x78a554094290>
```

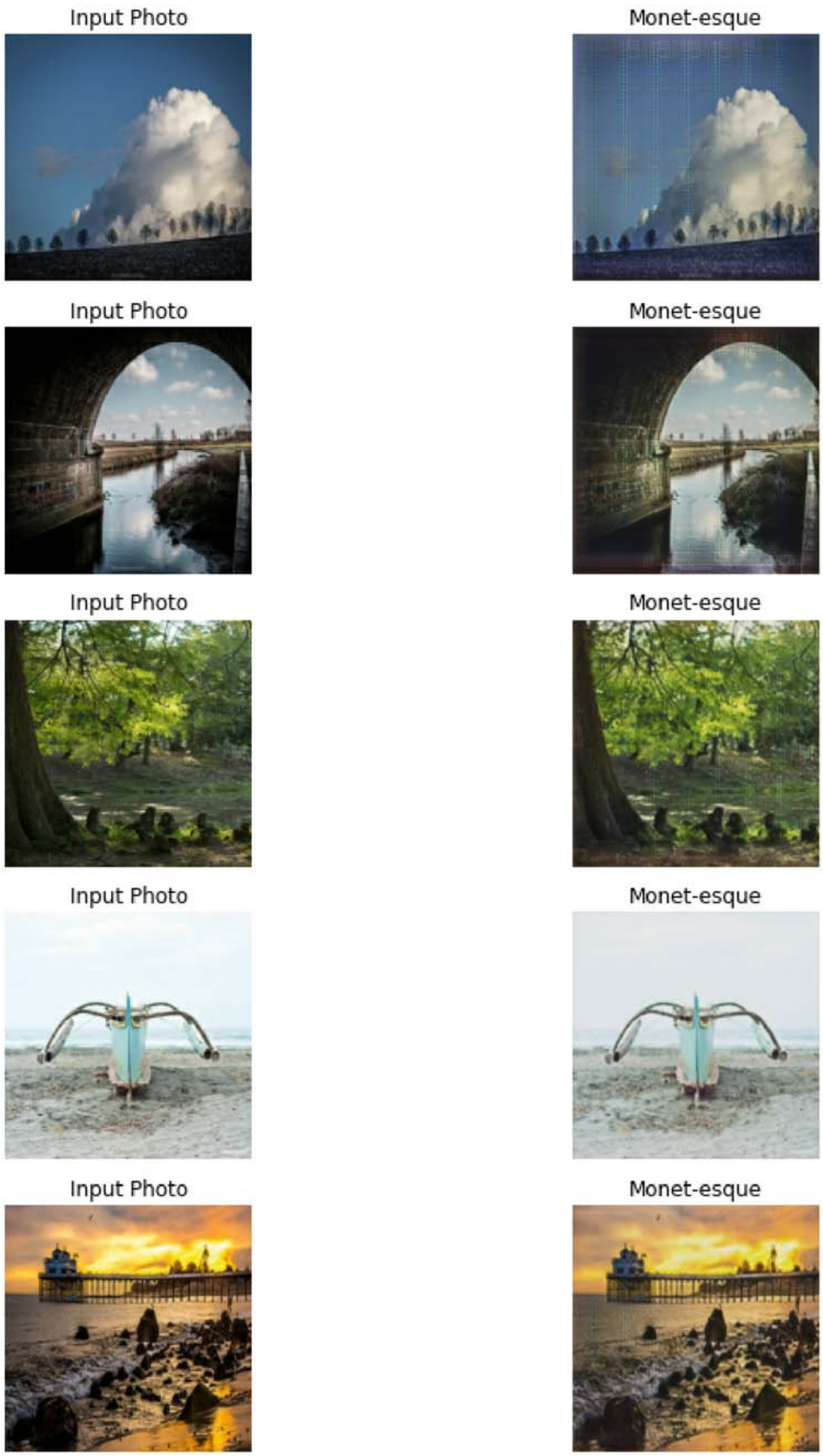
Example Generated Images

In [23]:
```
_, ax = plt.subplots(5, 2, figsize=(12, 12))
for i, img in enumerate(photo_ds.take(5)):
```

```
    prediction = monet_generator(img, training=False)[0].numpy()
    prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
    img = (img[0] * 127.5 + 127.5).numpy().astype(np.uint8)

    ax[i, 0].imshow(img)
    ax[i, 1].imshow(prediction)
    ax[i, 0].set_title("Input Photo")
    ax[i, 1].set_title("Monet-esque")
    ax[i, 0].axis("off")
    ax[i, 1].axis("off")

plt.tight_layout()
plt.show()
```


Input Photo
Monet-esque
Input Photo
Monet-esque
Input Photo
Monet-esque
Input Photo
Monet-esque
Input Photo
Monet-esque

Conclusion

The generated images are not radically different from the input images. There is some indication that they are trending towards a more impressionistic and less photorealistic appearance, but it is clear they are not quite there. Unfortunately, as this is an extremely resource-intensive model, my ability to experiment with different parameters is limited. I suspect that increasing training time would significantly improve the results, as would more elaborate up and downsampling functions.

Submission

In [24]:
```
import PIL
! mkdir '/kaggle/working/images'
```

In [ ]:
```
i = 1
for img in photo_ds:
    prediction = monet_generator(img, training=False)[0].numpy()
    prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
    im = PIL.Image.fromarray(prediction)
    im.save("/kaggle/working/images/" + str(i) + ".jpg")
    i += 1
```

In [ ]:
```
import shutil
shutil.make_archive("/kaggle/working/images", 'zip', "/kaggle/working")
```

References

https://keras.io/examples/generative/dcgan_overriding_train_step/

https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial

https://www.kaggle.com/c/gan-getting-started/data

Github

https://github.com/obbrown1/Deep-Learning-Week-5