For pipeline 3, I want to experiment with wide variety of computer vision techniques, incluing image classification, image generation and image segmentation:

- Image Classification: For pipeline 2 I have experiemented with fine-tuning CNN-based models such as MobileNet, Resnet,... In pipeline 3, I will implement Vision Transformer
- Image Generation: I played around with DiffusionSat, a
- Image Segmentation: I employed multiple variations of UNET, a popular image segmentation framework due to its ability to preserve spatial information in its downsampling path.

CONTINUE IMAGE CLASSIFICATION PIPELINE

VISION TRANSFORMER FROM SCRATCH

Extending on the original Transformer for text embedding that we learned in class where it receives a one-dimensional sequence of word embeddings as input, the Vision Transformer takes two-dimensional images as input as provides an adapted version of positional encoding: Instead of embedding each work in the sentence with its order of appearance, VIT divides the image into small patches and embeds each patches with its position in the image.

This architecture consists of 3 parts:

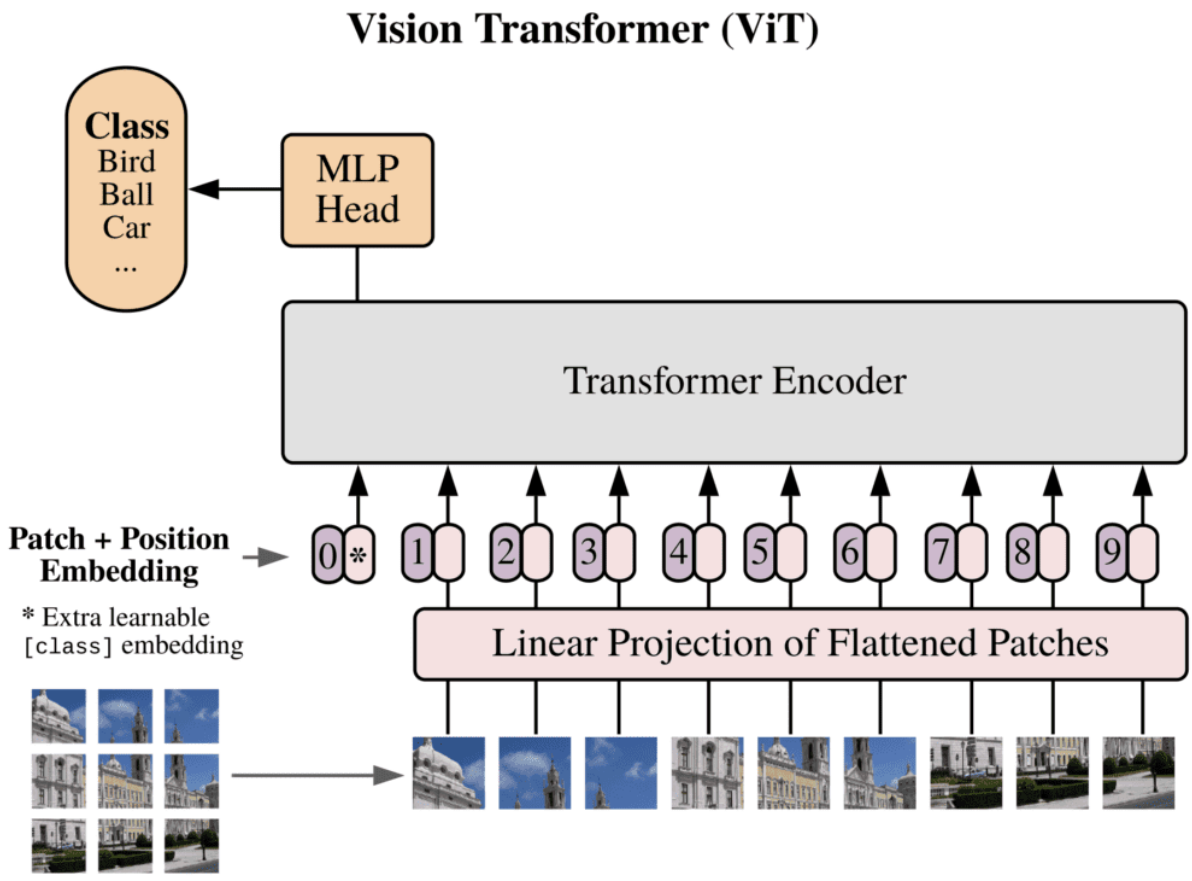
- Positional embedding: patchifies images
- Multilayer Transformer encoder
- A head that transforms the global representation into the output label.

Positional embedding

To structure the input image data in a way that resembles word embedding, the input image, of height H, width W, and C number of channels, is cut up into smaller two-dimensional patches. This results into $N = \frac{HW}{P^2}$ number of patches, where each patch has a resolution of (P, P) pixels.

In the code of "VIT from scratch" below, I define the resolution of pixels per patch to be

```
patch_size_in_vit = 8  # Size of patches for Vision Transformer (ViT)
```



The Architecture of the Vision Transformer (ViT)/ Taken from "An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale"

Before feeding the data into the Transformer, the following operations are applied:

- Each image patch is flattened into a vector, $\mathbf{x}_p^n$, of length $P^2 \times C$, where $n = 1, \ldots N$.
- A sequence of embedded image patches is generated by mapping the flattened patches to D dimensions, with a trainable linear projection, E.
- A learnable class embedding, $\mathbf{x}_{\text{class}}$, is prepended to the sequence of embedded image patches. The value of represents the classification output, .
- The patch embeddings are finally augmented with one-dimensional positional embeddings, $\mathbf{E}_{\text{pos}}$, hence introducing positional information into the input, which is also learned during training.

The sequence of embedding vectors that results from the aforementioned operations is the following:

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \ldots;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{\text{pos}}$$

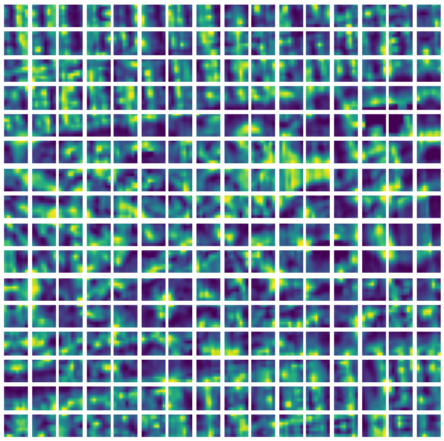
The code and output of the above process is as below:

In [ ]:
```
patch_size_in_vit = 8  # Size of patches for Vision Transformer (ViT)
image_size = 128  # Size of input images

# Define a custom layer for patch extraction
class ExtractPatchesLayer(layers.Layer):
    def __init__(self, patch_size):
        super(ExtractPatchesLayer, self).__init__()
        self.patch_size = patch_size

    def call(self, inputs):
        batch_size = tf.shape(inputs)[0]  # Getting batch size
        patches = tf.image.extract_patches(
            images=inputs,
            sizes=[1, self.patch_size, self.patch_size, 1],  # Patch size
            strides=[1, self.patch_size, self.patch_size, 1],  # Stride for patch extraction
            rates=[1, 1, 1, 1],
            padding="VALID",  # Padding mode
        )
        patch_dims = patches.shape[-1]  # Calculating patch dimensions
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])  # Reshaping patches
        return patches
```

In [ ]:
```
# Initialize the custom patch extraction layer
extract_patches_layer = ExtractPatchesLayer(patch_size_in_vit)

index = 0  # Index of the image in the training dataset
number_of_bands = 1  # Number of bands used in the image data

# Selecting a sample data from the training dataset with specified number of bands
sample_data = X_train[index : index + 1, :, :, number_of_bands : number_of_bands + 1]

# Extracting patches from the sample data using the custom layer
patches = extract_patches_layer(sample_data)

# Printing information about image size, patch size, and patches per image
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size_in_vit} X {patch_size_in_vit}")
print(f"Patches per image: {patches.shape[1]}")
print(f"Elements per patch: {patches.shape[-1]}")
```

```
Image size: 128 X 128
Patch size: 8 X 8
Patches per image: 256
Elements per patch: 64
```

In [ ]:
```
# Visualizing the extracted patches
n = int(np.sqrt(patches.shape[1]))  # Number of patches to display in each dimension
plt.figure(figsize=(4, 4))
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size_in_vit, patch_size_in_vit))
    plt.imshow(patch_img.numpy())  # Displaying the patch image
    plt.axis("off")  # Turning off axis
```



Each patch is embedded into a higher-dimensional space using a dense layer, and positional encodings are added to retain the positional information.

- Patch Embedding: Each patch is projected into a higher-dimensional space to capture more complex relationships(The linear projection dimension I chose is 100, which is a reasonable scale given that the image is small)
- Positional Encoding: Positional embeddings indicate the relative or absolute position of each patch within the image. Without these, Transformers would treat the image as an unordered bag of patches.

In [ ]:
```python
image_size = 128  # Size of the input image
patch_size = 8  # Size of each patch
num_patches = (image_size // patch_size) ** 2  # Number of patches in the image
projection_dim = 100  # Dimensionality of the projected embeddings

def patch_encoder(patch, num_patches, projection_dim):
    """
    Encodes each patch using an embedding layer followed by a dense layer.

    Args:
        patch: Input patch data with shape [batch_size, patch_size, patch_size, channels].
        num_patches: Number of patches in the image.
        projection_dim: Dimensionality of the projected embeddings.

    Returns:
        encoded: Encoded representation of the patch with shape [batch_size, num_patches, projection_dim].
    """
    positions = tf.range(start=0, limit=num_patches, delta=1)  # Generating positional embeddings

    # Embedding layer for positional embeddings
    emd = layers.Embedding(input_dim=num_patches, output_dim=projection_dim)(positions)

    # Dense layer for encoding the patch
    dens = layers.Dense(units=projection_dim)(patch)

    # Adding positional embeddings and patch encoding
    encoded = emd + dens

    return encoded
```

Encoder

The ViT employs the encoder part of the original Transformer architecture. The input to the encoder is a sequence of embedded image patches (including a learnable class embedding prepended to the sequence), which is also augmented with positional information. A classification head attached to the output of the encoder receives the value of the learnable class embedding, to generate a classification output based on its state

Each Transformer layer consists of multi-head attention, skip connections, and a feed-forward MLP.

a. Multi-Head Attention The attention mechanism allows the model to focus on important patches in the image.

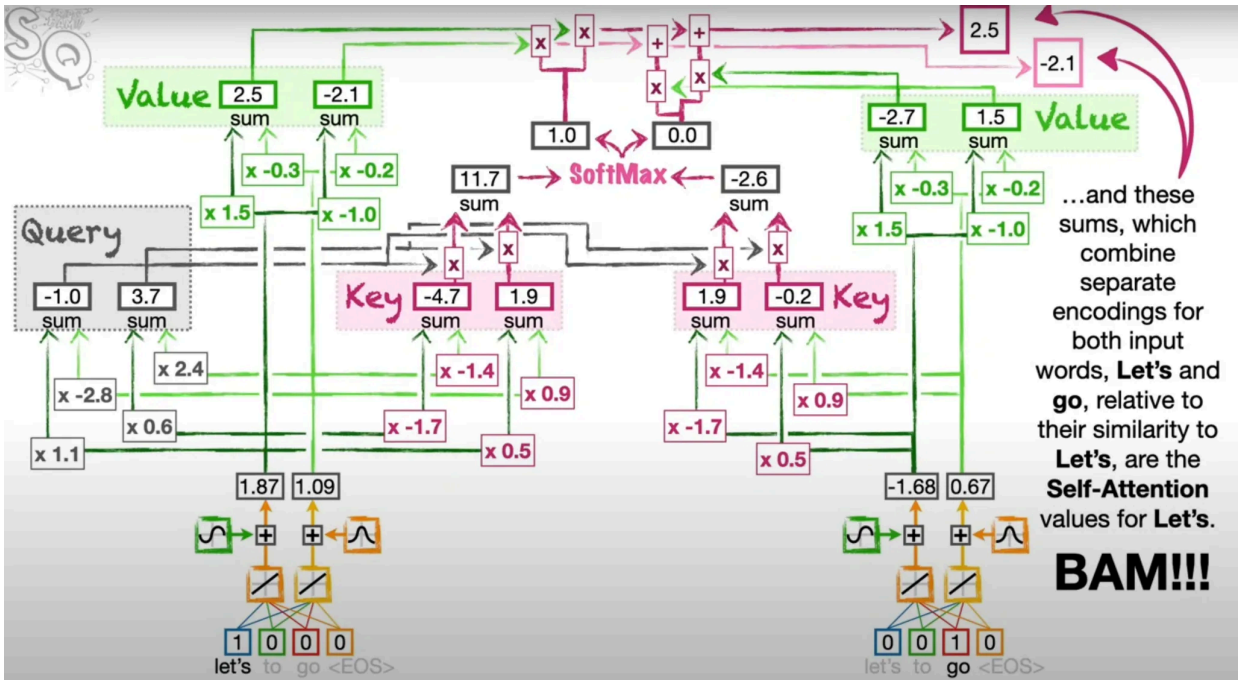
- Self-Attention: Each patch interacts with every other patch.
- Multi-Headed: Multiple attention "heads" allow the model to learn different aspects of the relationships simultaneously.

```
attention_output = layers.MultiHeadAttention(
    num_heads=8,  # Number of attention heads
    key_dim=projection_dim,  # Dimensionality of the embeddings
    dropout=0.2
)(x1, x1)
```

Detailed explanation of self attention mechanism

1. Calculate Similarity: For each patch in the image, the model calculates its similarity to all other patches (including itself). This is done by creating a set of values called query values for each patch, and key values for every other patches.
2. Dot Product: The model computes the dot product between the query values of a patch and the key values of all other patches to measure their similarity. Higher values indicate stronger relevance or connection between those patches.
3. Softmax Function: After calculating the dot products, the results are passed through a softmax function, which converts them into probabilities (ranging between 0 and 1) that add up to 1. This helps the model decide how much focus (weight) to give each patch when processing the current patch. (The scaling by d_k ensures that the Euclidean length of the weight vectors will be approximately in the same magnitude. This helps prevent the attention weights from becoming too small or too large, which could lead to numerical instability or affect the model's ability to converge during training.)
4. Generate a set of values key for every patches in the input/ output sentence
5. Weighting patches: The model multiply each patch value keys by its corresponding soft max output → the model reassigns a weight to each patch, indicating how much each patch should contribute to the final encoding of the current patch. Patches that are more relevant to the current patch (according to their similarity score) are given higher weights.
6. Update the Patch Encoding: The weighted values from all other patches are combined to form the new representation (encoding) of the current patch. This allows the model to consider the context of all other patches in the image while encoding the current patch.

This is a screenshot from StatsQuest that really helps me understand the detailed process of self attention mechanism:



b. Feed-Forward Network (MLP) A feed-forward network processes the output of the attention layer.

In [ ]:
```
transformer_units = [
  projection_dim * 2,
  projection_dim,
]  # Size of the transformer layers

def mlp(x, hidden_units, dropout=0.2):
    """
    Defines a Multi-Layer Perceptron (MLP) block.

    Args:
        x: Input tensor.
        hidden_units: List of integers specifying the number of units in each hidden layer.
        dropout: Dropout rate for the dropout layers.

    Returns:
        x: Output tensor after passing through the MLP block.
    """
    for units in hidden_units:
        x = layers.Dense(units, activation=tf.nn.gelu)(x)  # Fully connected layer with GELU activation
        x = layers.Dropout(dropout)(x)  # Dropout layer
    return x
```

Compile VIT

The ViT architecture stacks multiple Transformer layers, with skip connections to facilitate gradient flow.

```
    for _ in range(8):  # Number of transformer layers
        # Layer Normalization and Attention
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
```

```
        attention_output = layers.MultiHeadAttention(
            num_heads=8, key_dim=projection_dim, dropout=0.2)(x1, x1)
        x2 = layers.Add()([attention_output, encoded_patches])  # Skip connection

        # Layer Normalization and MLP
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        x3 = mlp(x3, hidden_units=[projection_dim * 2, projection_dim], dropout=0.1)
        encoded_patches = layers.Add()([x3, x2])  # Skip connection
```

Classification head

```
    # Representation Layer
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)

    # Classification Head
    features = mlp(representation, hidden_units=[2048, 1024], dropout=0.5)
    logits = layers.Dense(1, activation='sigmoid')(features)
```

In [ ]:

```
# Define parameters for the Vision Transformer
number_of_heads = 8  # Number of attention heads
transformer_layers = 8  # Number of transformer layers

def vit_model(input_shape):
    """
    Creates a Vision Transformer (ViT) model.

    Args:
        input_shape (tuple): Shape of the input images.

    Returns:
        keras.Model: Vision Transformer model.
    """
    # Define model inputs
    inputs = layers.Input(shape=input_shape)

    # Data Augmentation
    augmented_data = data_augmentation(inputs)

    # Extract patches from the augmented input image
    patches = extract_patches_layer(augmented_data)  # Assuming extract_patches_layer is the custom layer defined e

    # Patch Encoder
    encoded_patches = patch_encoder(patches, num_patches, projection_dim)  # Assume patch_encoder is defined elsewh

    # Apply multiple transformer layers
    for _ in range(transformer_layers):
        # Layer Normalization
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)

        # Multi-Head Attention
        attention_output = layers.MultiHeadAttention(
            num_heads=number_of_heads,
            key_dim=projection_dim,
            dropout=0.2
        )(x1, x1)

        # Skip Connection
        x2 = layers.Add()([attention_output, encoded_patches])

        # Normalization Layer
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)

        # MLP Section
        x3 = mlp(x3, hidden_units=transformer_units, dropout=0.1)  # Assume mlp is defined elsewhere

        # Skip Connection
        encoded_patches = layers.Add()([x3, x2])

    # Representation layer
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)  # Flattening the tensor
    representation = layers.Dropout(0.5)(representation)

    # Add MLP for feature extraction
    features = mlp(representation, hidden_units=[2048, 1024], dropout=0.5)  # Apply MLP to the representation

    # Classification layer
    logits = layers.Dense(1, activation='sigmoid')(features)  # Output layer for classification

    # Create the Keras model
    model = keras.Model(inputs=inputs, outputs=logits)

    return model
```

In [ ]:
```
# Create an instance of the ViT model with input shape (64, 64, 13)
model = vit_model((128, 128, 3))

# Print the summary of the model architecture
model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	–
data_augmentation (Sequential)	(None, 128, 128, 3)	7	input_layer[0][0]
extract_patches_layer (ExtractPatchesLayer)	(None, None, 192)	0	data_augmentation[1][…
dense (Dense)	(None, None, 100)	19,300	extract_patches_layer…
add (Add)	(None, 256, 100)	0	dense[0][0]
layer_normalization (LayerNormalization)	(None, 256, 100)	200	add[0][0]
multi_head_attention (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization[0… layer_normalization[0…
add_1 (Add)	(None, 256, 100)	0	multi_head_attention[… add[0][0]
layer_normalization_1 (LayerNormalization)	(None, 256, 100)	200	add_1[0][0]
dense_1 (Dense)	(None, 256, 200)	20,200	layer_normalization_1…
dropout_1 (Dropout)	(None, 256, 200)	0	dense_1[0][0]
dense_2 (Dense)	(None, 256, 100)	20,100	dropout_1[0][0]
dropout_2 (Dropout)	(None, 256, 100)	0	dense_2[0][0]
add_2 (Add)	(None, 256, 100)	0	dropout_2[0][0], add_1[0][0]
layer_normalization_2 (LayerNormalization)	(None, 256, 100)	200	add_2[0][0]
multi_head_attention_1 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_2… layer_normalization_2…
add_3 (Add)	(None, 256, 100)	0	multi_head_attention_… add_2[0][0]
layer_normalization_3 (LayerNormalization)	(None, 256, 100)	200	add_3[0][0]
dense_3 (Dense)	(None, 256, 200)	20,200	layer_normalization_3…
dropout_4 (Dropout)	(None, 256, 200)	0	dense_3[0][0]
dense_4 (Dense)	(None, 256, 100)	20,100	dropout_4[0][0]
dropout_5 (Dropout)	(None, 256, 100)	0	dense_4[0][0]
add_4 (Add)	(None, 256, 100)	0	dropout_5[0][0], add_3[0][0]
layer_normalization_4 (LayerNormalization)	(None, 256, 100)	200	add_4[0][0]
multi_head_attention_2 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_4… layer_normalization_4…
add_5 (Add)	(None, 256, 100)	0	multi_head_attention_… add_4[0][0]
layer_normalization_5 (LayerNormalization)	(None, 256, 100)	200	add_5[0][0]
dense_5 (Dense)	(None, 256, 200)	20,200	layer_normalization_5…
dropout_7 (Dropout)	(None, 256, 200)	0	dense_5[0][0]
dense_6 (Dense)	(None, 256, 100)	20,100	dropout_7[0][0]
dropout_8 (Dropout)	(None, 256, 100)	0	dense_6[0][0]
add_6 (Add)	(None, 256, 100)	0	dropout_8[0][0], add_5[0][0]
layer_normalization_6 (LayerNormalization)	(None, 256, 100)	200	add_6[0][0]
multi_head_attention_3 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_6… layer_normalization_6…
add_7 (Add)	(None, 256, 100)	0	multi_head_attention_… add_6[0][0]

layer_normalization_7 (LayerNormalization)	(None, 256, 100)	200	add_7[0][0]
dense_7 (Dense)	(None, 256, 200)	20,200	layer_normalization_7…
dropout_10 (Dropout)	(None, 256, 200)	0	dense_7[0][0]
dense_8 (Dense)	(None, 256, 100)	20,100	dropout_10[0][0]
dropout_11 (Dropout)	(None, 256, 100)	0	dense_8[0][0]
add_8 (Add)	(None, 256, 100)	0	dropout_11[0][0], add_7[0][0]
layer_normalization_8 (LayerNormalization)	(None, 256, 100)	200	add_8[0][0]
multi_head_attention_4 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_8… layer_normalization_8…
add_9 (Add)	(None, 256, 100)	0	multi_head_attention_… add_8[0][0]
layer_normalization_9 (LayerNormalization)	(None, 256, 100)	200	add_9[0][0]
dense_9 (Dense)	(None, 256, 200)	20,200	layer_normalization_9…
dropout_13 (Dropout)	(None, 256, 200)	0	dense_9[0][0]
dense_10 (Dense)	(None, 256, 100)	20,100	dropout_13[0][0]
dropout_14 (Dropout)	(None, 256, 100)	0	dense_10[0][0]
add_10 (Add)	(None, 256, 100)	0	dropout_14[0][0], add_9[0][0]
layer_normalization_10 (LayerNormalization)	(None, 256, 100)	200	add_10[0][0]
multi_head_attention_5 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_1… layer_normalization_1…
add_11 (Add)	(None, 256, 100)	0	multi_head_attention_… add_10[0][0]
layer_normalization_11 (LayerNormalization)	(None, 256, 100)	200	add_11[0][0]
dense_11 (Dense)	(None, 256, 200)	20,200	layer_normalization_1…
dropout_16 (Dropout)	(None, 256, 200)	0	dense_11[0][0]
dense_12 (Dense)	(None, 256, 100)	20,100	dropout_16[0][0]
dropout_17 (Dropout)	(None, 256, 100)	0	dense_12[0][0]
add_12 (Add)	(None, 256, 100)	0	dropout_17[0][0], add_11[0][0]
layer_normalization_12 (LayerNormalization)	(None, 256, 100)	200	add_12[0][0]
multi_head_attention_6 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_1… layer_normalization_1…
add_13 (Add)	(None, 256, 100)	0	multi_head_attention_… add_12[0][0]
layer_normalization_13 (LayerNormalization)	(None, 256, 100)	200	add_13[0][0]
dense_13 (Dense)	(None, 256, 200)	20,200	layer_normalization_1…
dropout_19 (Dropout)	(None, 256, 200)	0	dense_13[0][0]
dense_14 (Dense)	(None, 256, 100)	20,100	dropout_19[0][0]
dropout_20 (Dropout)	(None, 256, 100)	0	dense_14[0][0]
add_14 (Add)	(None, 256, 100)	0	dropout_20[0][0], add_13[0][0]
layer_normalization_14 (LayerNormalization)	(None, 256, 100)	200	add_14[0][0]
multi_head_attention_7 (MultiHeadAttention)	(None, 256, 100)	322,500	layer_normalization_1… layer_normalization_1…
add_15 (Add)	(None, 256, 100)	0	multi_head_attention_… add_14[0][0]
layer_normalization_15	(None, 256, 100)	200	add_15[0][0]

(LayerNormalization)			
dense_15 (Dense)	(None, 256, 200)	20,200	layer_normalization_1…
dropout_22 (Dropout)	(None, 256, 200)	0	dense_15[0][0]
dense_16 (Dense)	(None, 256, 100)	20,100	dropout_22[0][0]
dropout_23 (Dropout)	(None, 256, 100)	0	dense_16[0][0]
add_16 (Add)	(None, 256, 100)	0	dropout_23[0][0], add_15[0][0]
layer_normalization_16 (LayerNormalization)	(None, 256, 100)	200	add_16[0][0]
flatten (Flatten)	(None, 25600)	0	layer_normalization_1…
dropout_24 (Dropout)	(None, 25600)	0	flatten[0][0]
dense_17 (Dense)	(None, 2048)	52,430,848	dropout_24[0][0]
dropout_25 (Dropout)	(None, 2048)	0	dense_17[0][0]
dense_18 (Dense)	(None, 1024)	2,098,176	dropout_25[0][0]
dropout_26 (Dropout)	(None, 1024)	0	dense_18[0][0]
dense_19 (Dense)	(None, 1)	1,025	dropout_26[0][0]

Total params: 57,455,156 (219.17 MB)

Trainable params: 57,455,149 (219.17 MB)

Non-trainable params: 7 (32.00 B)

In [ ]:
```
# Add WandbMetricsLogger to log metrics and WandbModelCheckpoint to log model checkpoints
wandb_callbacks = [
    WandbMetricsLogger(),
    WandbModelCheckpoint(filepath="my_model_{epoch:02d}.keras"),
]

model.fit(train_dataset,
          epochs=config.epochs,
          batch_size=config.batch_size,
          validation_data=validation_dataset,
          callbacks=wandb_callbacks)
```

```
Epoch 1/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 142s 383ms/step - acc: 0.9220 - loss: 0.3165 - val_acc: 0.4932 - val_loss: 2.1701
Epoch 2/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 125s 343ms/step - acc: 0.9046 - loss: 0.4138 - val_acc: 0.4877 - val_loss: 2.5546
Epoch 3/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 154s 386ms/step - acc: 0.9060 - loss: 0.3788 - val_acc: 0.4913 - val_loss: 2.4256
Epoch 4/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 101s 353ms/step - acc: 0.8952 - loss: 0.4488 - val_acc: 0.4913 - val_loss: 2.2469
Epoch 5/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 176s 472ms/step - acc: 0.8892 - loss: 0.4242 - val_acc: 0.4913 - val_loss: 2.4329
Epoch 6/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 122s 404ms/step - acc: 0.8905 - loss: 0.4066 - val_acc: 0.4913 - val_loss: 1.6674
Epoch 7/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 101s 353ms/step - acc: 0.8762 - loss: 0.3981 - val_acc: 0.4913 - val_loss: 2.3691
Epoch 8/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 109s 382ms/step - acc: 0.8961 - loss: 0.3915 - val_acc: 0.4913 - val_loss: 1.8024
Epoch 9/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 113s 396ms/step - acc: 0.8809 - loss: 0.3984 - val_acc: 0.4913 - val_loss: 1.9964
Epoch 10/10
285/285 ━━━━━━━━━━━━━━━━━━━━ 110s 387ms/step - acc: 0.8931 - loss: 0.3779 - val_acc: 0.4913 - val_loss: 1.9834
```

Out[ ]:
```
<keras.src.callbacks.history.History at 0x7cd7055818d0>
```

PRETRAINED VISION TRANSFORMER

When I trained this VIT from scratch, the model took really long for the validation loss to go down. In the code above, I show the training process after 10 epochs. I continue training for 50 epoches but the performance did not improve and I constantly run out of GPU, so I continue the training of VIT by using pretrained models on the internet: ViTTiny16. ViT-Tiny/16 is a specific variant of the Vision Transformer (ViT) model. It is designed as a smaller and more efficient version of the standard ViT architecture. Each image is divided into non-overlapping patches of 16x16, which are then treated as tokens. It has fewer Transformer encoder layers, smaller embedding dimensions, and fewer attention heads compared to larger variants like ViT-Base, ViT-Large, etc.

One notable aspect of this pretrained VIT model is the use of "token pooling":

- The pooling="token_pooling" argument specifies that token pooling should be applied after the ViT encoder processes the token sequence.
- Instead of returning embeddings for each token (or relying on the [CLS] token), the ViT model applies token pooling to summarize all token embeddings. This aggregation step might leverage mechanisms like attention weights or learned parameters to focus on

the most informative patches.

- Output: The vit model produces a single vector that represents the image as a whole after token pooling. This output serves as the input to the final dense layer for classification.

In [ ]:
```
def get_model():
    inputs = tf.keras.layers.Input(shape=(224, 224, 3))
    normalized = tf.keras.layers.Normalization()(inputs)
    vit = ViTTiny16(
        include_rescaling=False,
        include_top=False,
        name="ViTTiny32",
        weights="imagenet",
        input_tensor=inputs,
        pooling="token_pooling",
        activation=tf.keras.activations.gelu,
    )

    vit.trainable = False

    outputs = tf.keras.layers.Dense(1, activation="sigmoid")(vit.output)
    model = tf.keras.Model(inputs=inputs, outputs=outputs)

    return model
```

In [ ]:
```
VIT_model = get_model()
VIT_model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 224, 224, 3)	0
rescaling_2 (Rescaling)	(None, 224, 224, 3)	0
patching_and_embedding_2 (PatchingAndEmbedding)	(None, 197, 192)	185,664
dropout_170 (Dropout)	(None, 197, 192)	0
transformer_encoder_24 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_25 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_26 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_27 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_28 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_29 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_30 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_31 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_32 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_33 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_34 (TransformerEncoder)	(None, 197, 192)	444,864
transformer_encoder_35 (TransformerEncoder)	(None, 197, 192)	444,864
layer_normalization_74 (LayerNormalization)	(None, 197, 192)	384
get_item_2 (GetItem)	(None, 192)	0
dense_73 (Dense)	(None, 1)	193

Total params: 5,524,609 (21.07 MB)

Trainable params: 193 (772.00 B)

Non-trainable params: 5,524,416 (21.07 MB)

In [ ]:
```
initial_epochs = 20
learning_rate = 0.0001
```

```
num_steps = len(train_dataset) * initial_epochs
decay_steps = num_steps
cosine_decay_scheduler = tf.keras.optimizers.schedules.CosineDecay(
    learning_rate, decay_steps, alpha=0.1
)

VIT_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=cosine_decay_scheduler),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=["accuracy"],
)
```

In [ ]:
```
VIT_history = VIT_model.fit(train_dataset,
                    epochs=initial_epochs,
                    validation_data=validation_dataset,
                    callbacks = callbacks, verbose = 0)
```

```
Restoring model weights from the end of the best epoch: 20.
```

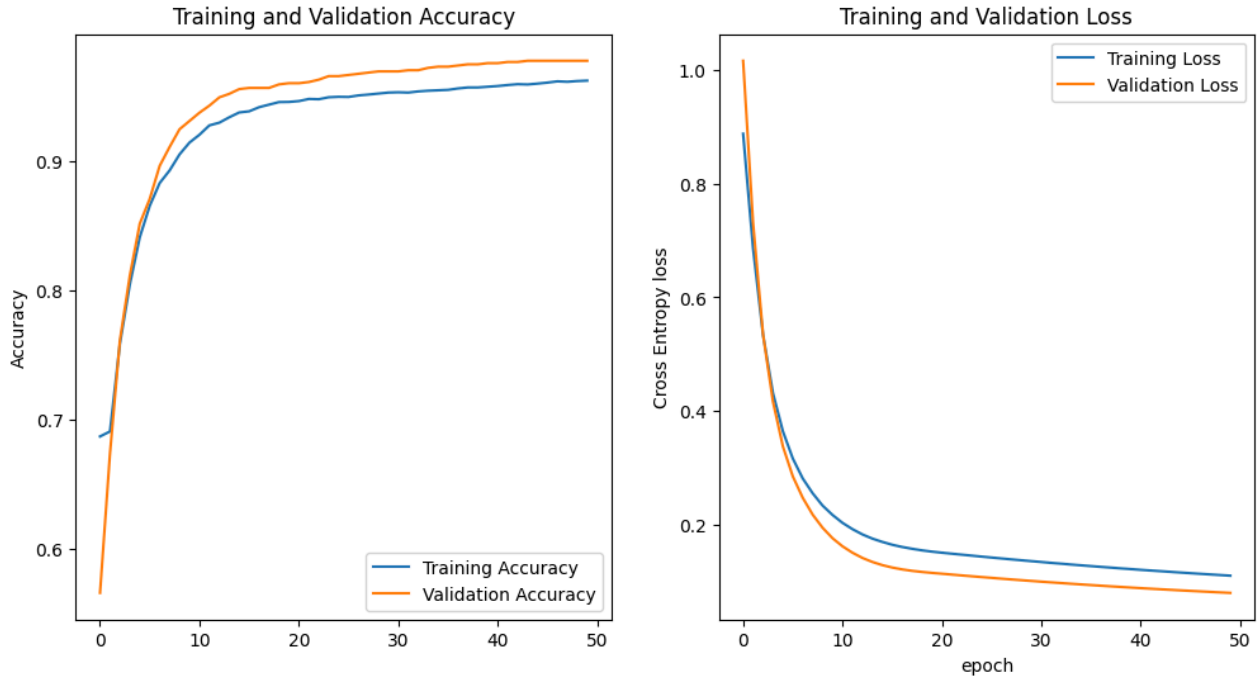
In [ ]:
```
# Record the initial training history
initial_history = VIT_history.history
```

In [ ]:
```
# Second training phase (additional epochs)
additional_epochs = 30  # Add more epochs as required
history_additional = VIT_model.fit(
    train_dataset,
    epochs=initial_epochs + additional_epochs,  # Total epochs, resume from the last epoch
    initial_epoch=initial_epochs,  # Start from where the last phase ended
    validation_data=validation_dataset,
    callbacks=callbacks,
    verbose=0
)
```

```
Restoring model weights from the end of the best epoch: 50.
```

In [ ]:
```
# Record the additional training history
additional_history = history_additional.history
# Combine both histories (initial + additional)
combined_history = {key: initial_history[key] + additional_history[key] for key in initial_history}
acc = combined_history['accuracy']
val_acc = combined_history['val_accuracy']
loss = combined_history['loss']
val_loss = combined_history['val_loss']
```

In [ ]:
```
plot_loss_acc (acc, val_acc, loss, val_loss)
```

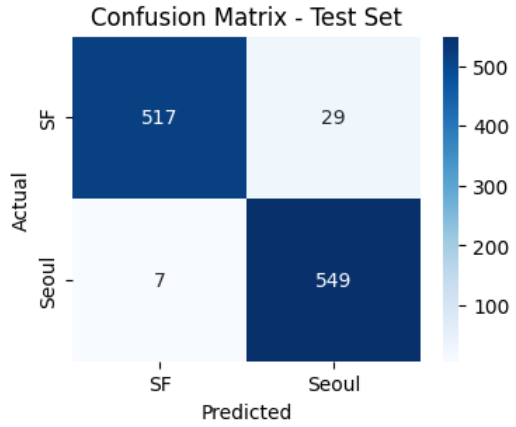


The training proces produces good smooth improvement in both training and validation accuracy/ loss. Notably, validation accuracy is higher than training accuracy and validation loss is lower than training loss. This is possibly due to data augmentation applied to only training set. These techniques intentionally add noise or reduce model capacity during training to prevent overfitting, which can slightly degrade training accuracy. During validation, the model performs better since these regularizations are turned off.

In [ ]:
```
test_conf_report(VIT_model , "VIT model")
```

```
35/35 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step
Classification Report on Test Dataset
              precision    recall  f1-score   support

           0       0.99      0.95      0.97       546
           1       0.95      0.99      0.97       556

    accuracy                           0.97      1102
   macro avg       0.97      0.97      0.97      1102
weighted avg       0.97      0.97      0.97      1102
```

Confusion Matrix on Test Dataset



Out[ ]:
```
{'VIT model': [0.967]}
```

To compare the performance of VIT to models I tried in pipeline 2. MobileNetV2 still yields the best performance, followed by Satlas. Notably, a finetuned Resnet50V2 model still has a slight poorer performance than a non-finetuned VIT model.

Model	Accuracy
MobileNetV2	98%
Resnet50V2	96%
Satlas	97.1%
VIT	96.7%

NEW PIPELINE: IMAGE SEGMENTATION

I want to keep the segmentation task simple so I tried to segment water body from non-water body in my dataset. To create the masks set, I design a new preprocessing pipeline

PREPROCESSING

Re-download satellite images with water masks

In my image download pipeline, I applied function `apply_water_mask` that uses Earth Engine's ee.Image class to apply a water mask to satellite imagery based on the Hansen Global Forest Change dataset.

- A water mask is generated by selecting pixels in the datamask band that have a value of 2, corresponding to water areas.
- Water pixels in the original image are replaced with a uniform value (0 in this example).
- The function uses updateMask to retain only the water pixels, then scales their values to 0 and adds the uniform value (0 in this case).

For every image in the collection, I downloaded an unmasked and a masked version:

In [ ]:
```python
# Function to process Sentinel-2 images for a given area of interest (AOI)
def process_images_with_and_without_masks(aoi):
    # Load the Sentinel-2 image collection and select bands B2, B3, B4
    collection = (ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
                  .filterBounds(aoi)
                  .filterDate('2000-01-01', '2024-01-31')
                  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 1))
                  .select(['B2', 'B3', 'B4']))  # Select bands B2 (Blue), B3 (Green), and B4 (Red)

    # Function to create the water-masked version using the Hansen dataset
    def apply_water_mask(image):
        # Load the Hansen dataset for land/water masking
        hansen_image = ee.Image('UMD/hansen/global_forest_change_2015')
        datamask = hansen_image.select('datamask')

        # Create a mask for water pixels (2 = water in the Hansen dataset)
```

```
        water_mask = datamask.eq(2)  # 1 = land, 2 = water

        # Replace water pixels with a specific value (e.g., 0) and keep land data intact
        uniform_water_value = 0  # Set desired value for water pixels
        water_pixels = image.updateMask(water_mask).multiply(0).add(uniform_water_value)

        # Merge the two images, prioritizing land data
        masked_image = image.blend(water_pixels)  # Replace water with specific value
        return masked_image

    # Function to export both versions of the image
    def export_images(image, index):
        # Original image (no masking)
        original_image_name = f'Original_Image_{index}'
        unmasked_image = image.unmask(value=0, sameFootprint=True)

        task_original = ee.batch.Export.image.toDrive(
            image=unmasked_image.clip(aoi),
            description=original_image_name,
            folder=f'Segmentation_satellites_data',
            region=aoi,
            scale=10,
            maxPixels=1e9,
            fileFormat='GeoTIFF',
            formatOptions={'cloudOptimized': True, 'noData': 0}
        )
        task_original.start()  # Start export task

        # Water-masked version
        masked_image_name = f'Masked_Image_{index}'
        water_masked_image = apply_water_mask(unmasked_image)

        task_masked = ee.batch.Export.image.toDrive(
            image=water_masked_image.clip(aoi),
            description=masked_image_name,
            folder=f'Mask_satellites_data',
            region=aoi,
            scale=10,
            maxPixels=1e9,
            fileFormat='GeoTIFF',
            formatOptions={'cloudOptimized': True, 'noData': 0}
        )
        task_masked.start()  # Start export task

    # Get the list of images from the collection
    image_list = collection.toList(collection.size())

    # Function to export each image with a unique index
    def process_and_export_images(index):
        image = ee.Image(image_list.get(index))  # Get the image at the current index
        export_images(image, index)

    # Iterate through the image collection and export each image
    image_count = collection.size().getInfo()  # Get the total number of images
    for index in range(image_count):
        process_and_export_images(index)
```

I then split the downloaded images into train, validation, set while ensuring that image-mask pair always go together

In [ ]:

```
# List all images in the folders
Original_images = [os.path.join(Image_folder, img) \
                   for img in os.listdir(Image_folder) if img.endswith(".tif")]
Mask_images = [os.path.join(Mask_folder, img) \
                  for img in os.listdir(Mask_folder) if img.endswith(".tif")]

# Sort the lists to maintain consistency
Original_images.sort()
Mask_images.sort()

# Ensure both lists are of the same length
assert len(Original_images) == len(Mask_images), \
"Number of original images does not match number of mask images!"

# Shuffle the images for randomness in the split
combined_images = list(zip(Original_images, Mask_images))
random.shuffle(combined_images)

# Split into training (70%), validation (15%), and test (15%)
total_images = len(combined_images)
train_size = int(0.7 * total_images)
val_size = int(0.15 * total_images)

# Ensure the test set includes the remaining images
test_size = total_images - (train_size + val_size)

# Split the data
train_images = combined_images[:train_size]
val_images = combined_images[train_size:train_size + val_size]
test_images = combined_images[train_size + val_size:]

print(f"Training set: {len(train_images)} images")
```

```
print(f"Validation set: {len(val_images)} images")
print(f"Test set: {len(test_images)} images")
```

```
Training set: 37 images
Validation set: 7 images
Test set: 9 images
```

Preprocessing image-mask pairs

1. Image Scaling (scale_image): Scales pixel values to [0, 1] using MinMaxScaler for numerical stability.
2. Image Resizing (resize_images_together): use bilinear resampling, clips pixel values to [0, 1], and transposes them to (height, width, channels). I make sure to resize both image and mask to the same size.
3. Patch Creation (create_random_patches) Extracts random patches of size (patch_size, patch_size) from images and masks. I first generate random index and apply that random index to extract patches from the original image and the mask at the exact same random location -> ensure that each patch has a corresponding exact mask.
4. Filters patches based on water pixel ratio thresholds. I use water_pixel_threshold_min and water_pixel_threshold_max to filter the patches with the content of water body ranging from 30 to 70%. This helps ensure that the segmentation task does not have to deal with highly unbalanced class (e.g image with only water body or only land body) while still allowing for a level of unbalanceness to test the model's ability to handle segmentation of dominated areas.
5. One-hot encoding mask labels which will later produce `y_train.shape = (128, 128, 2). Each channel corresponds to the boolean value of being a water pixel or not. For example, in the first channel (water channel), if the pixel belongs to the water class its pixel value will be 1, otherwise 0. Similarly ,in the second channel (non-water channel), f the pixel belongs to the water class its pixel value will be 0, otherwise 1.

In [ ]:
```
# Function to scale the image between 0 and 1
def scale_image(image):
    reshaped_image = image.reshape(-1, image.shape[-1])  # Flatten the image for scaling
    reshaped_image = np.nan_to_num(reshaped_image, nan=0)
    scaler = MinMaxScaler()  # Rescale
    scaled_image = scaler.fit_transform(reshaped_image)  # Convert back to original shape
    return scaled_image.reshape(image.shape).astype(np.float32)

# Function to resize both the original image and the mask image to the same target size
def resize_images_together(input_image_path, mask_image_path, patch_size):
    # Open the original image and mask image to get dimensions
    with rasterio.open(input_image_path) as src:
        target_width = (src.width // patch_size) * patch_size
        target_height = (src.height // patch_size) * patch_size
        original_image_data = src.read(
            out_shape=(src.count, target_height, target_width),
            resampling=Resampling.bilinear
        )
        original_image_data = scale_image(original_image_data)
        original_image = np.clip(original_image_data, 0, 1).transpose(1, 2, 0).astype(np.float32)

    with rasterio.open(mask_image_path) as mask_src:
        mask_image_data = mask_src.read(
            out_shape=(mask_src.count, target_height, target_width),
            resampling=Resampling.bilinear
        )
        mask_image_data = scale_image(mask_image_data)
        mask_image = np.clip(mask_image_data, 0, 1).transpose(1, 2, 0).astype(np.float32)

    return original_image, mask_image

# Function to create random patches
def create_random_patches(image, mask_image, patch_size, water_pixel_threshold_min, water_pixel_threshold_max):
    valid_patches = []
    valid_masks = []
    height, width, _ = image.shape
    num_patches = int(((height * width) // (patch_size * patch_size)) * 1.5)
    max_attempts = 100  # Maximum number of attempts to find valid patches

    for _ in range(max_attempts):
        if len(valid_patches) >= num_patches:
            break  # Stop if we've found enough patches

        # Randomly select upper-left corner of the patch
        x = random.randint(0, width - patch_size)
        y = random.randint(0, height - patch_size)

        # Extract the patch from the image
        patch = image[y:y + patch_size, x:x + patch_size, :]  # (patch_size, patch_size, 3)
        mask_patch = mask_image[y:y + patch_size, x:x + patch_size, :]

        # Calculate the proportion of zero pixels in the mask patch (water pixels)
        total_pixels = patch.size
        water_pixels = np.sum(np.abs(mask_patch) == 0)
        water_pixel_ratio = water_pixels / total_pixels

        # If the proportion of water pixels is within the threshold, keep the patch
```

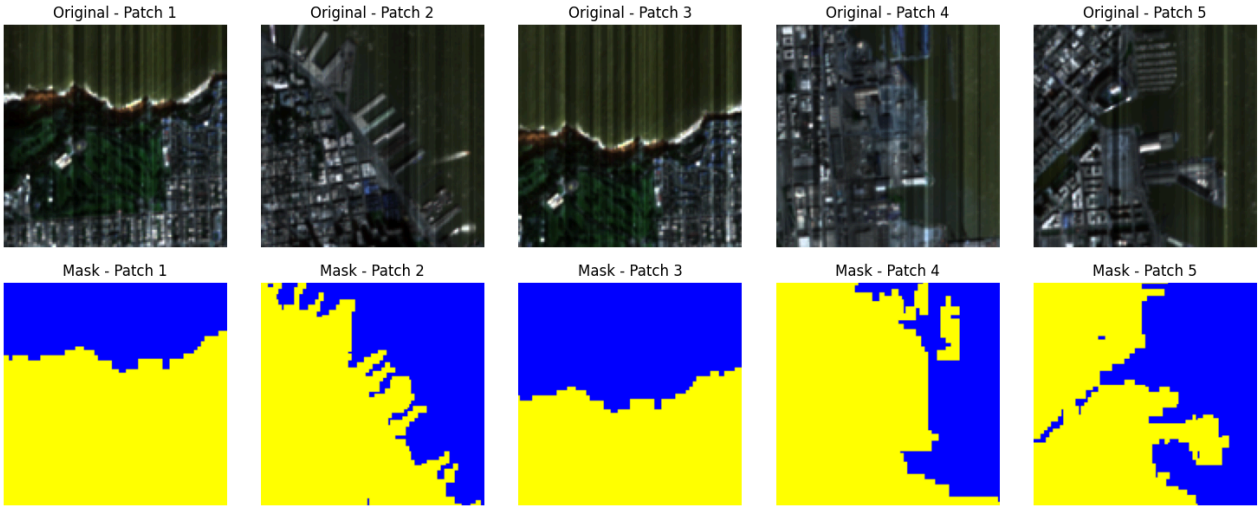
```
        if water_pixel_threshold_min <= water_pixel_ratio <= water_pixel_threshold_max:
            valid_patches.append(patch)

            # Use the first channel of the mask to determine water and non-water
            channel = mask_patch[..., 0]  # Focus on the first channel only

            # Binary masks: 1 for water, 0 for non-water
            water_mask = (channel == 0).astype(int)  # 1 for water
            non_water_mask = (channel != 0).astype(int)  # 1 for non-water
            # Stack masks along the last axis to create a 2-channel mask
            mask = np.stack([water_mask, non_water_mask], axis=-1)  # (patch_size, patch_size, 2)
            valid_masks.append(mask)

    return np.array(valid_patches), np.array(valid_masks)

# Main function to create patches and masks
def create_patches_and_masks(input_image_path, mask_image_path, patch_size, \
                water_pixel_threshold_min=0.30, water_pixel_threshold_max=0.70):
    # Resizing both the image and the mask together
    resized_image, resized_mask_image = resize_images_together(input_image_path, mask_image_path, patch_size)

    # Create random patches and corresponding masks
    patches, masks = create_random_patches\
     (resized_image, resized_mask_image, patch_size, water_pixel_threshold_min, water_pixel_threshold_max)

    return resized_image, patches, masks
```

Visualize image-mask pair

In [ ]:
```
# Visualization function for resized images
def visualize_image(city, image):
    plt.figure(figsize=(5, 4))
    plt.imshow(image)
    plt.title(f"Resized Image of {city}")
    plt.axis('off')  # Turn off axis
    plt.show()
```

In [ ]:
```
def visualize_patches_and_masks(patches, masks, num_patches=5):
    """
    Visualize a subset of image patches and their corresponding water/non-water masks.

    Args:
        patches (numpy.ndarray): Array of image patches
        (shape: [num_patches, patch_size, patch_size, channels]).
        masks (numpy.ndarray): Array of masks (shape: [num_patches, patch_size, patch_size, 2]).
        num_patches (int): Number of patches to visualize.
    """
    num_patches = min(num_patches, len(patches))  # Ensure we don't exceed the number of available patches

    # Create a 2-row subplot layout
    fig, axes = plt.subplots(2, num_patches, figsize=(15, 6))

    # Loop through each patch and corresponding mask
    for i in range(num_patches):
        patch = patches[i]
        mask = masks[i]

        # Create RGB visualization for the mask
        mask_visualization = np.zeros((mask.shape[0], mask.shape[1], 3), dtype=np.uint8)

        # Map water pixels (class 0) to blue
        water_mask = mask[:, :, 0]
        mask_visualization[water_mask == 1] = [0, 0, 255]

        # Map non-water pixels (class 1) to yellow
        non_water_mask = mask[:, :, 1]
        mask_visualization[non_water_mask == 1] = [255, 255, 0]

        # Plot the patch on the first row
        axes[0, i].imshow(patch)
        axes[0, i].set_title(f'Original - Patch {i + 1}')
        axes[0, i].axis('off')

        # Plot the mask on the second row
        axes[1, i].imshow(mask_visualization)
        axes[1, i].set_title(f'Mask - Patch {i + 1}')
        axes[1, i].axis('off')

    # Adjust the layout for better spacing
    plt.tight_layout(pad=1.0, w_pad=1.0, h_pad=2.0)
    plt.show()
```

In [ ]:
```
patch_size = 128
image_0, im0_patches, im0_mask = create_patches_and_masks(train_images[0][0], train_images[0][1], patch_size)
visualize_image("SF", image_0)
visualize_patches_and_masks(im0_patches, im0_mask)
```



Create image-mask pair dataset

In [ ]:
```python
# Preprocess images and create combined datasets
def preprocess_images(image_paths, mask_paths, patch_size):
    """
    Preprocess multiple images and their corresponding masks,
    returning patches along with labels (masks).
    """
    all_patches = []
    all_masks = []

    # Iterate over pairs of image and mask paths
    for image_path, mask_path in zip(image_paths, mask_paths):
        # Get all patches from original image and mask
        _, image_patches, mask_patches = create_patches_and_masks\
         (image_path, mask_path, patch_size, \
          water_pixel_threshold_min=0.30, water_pixel_threshold_max=0.70)

        # Ensure both patches are of the same length
        assert len(image_patches) == len(mask_patches),\
         "Number of patches for image and mask must match."

        # Append the patches to the respective lists
        all_patches.append(image_patches)
        all_masks.append(mask_patches)

    # Stack all patches and masks
    all_patches = np.vstack(all_patches)  # Combine all image patches
    all_masks = np.vstack(all_masks)      # Combine all mask patches

    return all_patches, all_masks
```

Generate preprocessed train, validation and test sets independently

In [ ]:
```python
def create_dataset (dataset):
  # Extract the image and mask paths from train_images
  image_paths = [image[0] for image in dataset]
  mask_paths = [image[1] for image in dataset]

  # Define patch size
  patch_size = 128

  # Preprocess the training images and masks
  patches, masks = preprocess_images(image_paths, mask_paths, patch_size)
```

```
  print(f"Total training patches: {len(patches)}")
  print(f"Total training masks: {len(masks)}")

  return patches, masks
```

In [ ]:
```
X_train, y_train = create_dataset(train_images)
X_val, y_val = create_dataset(val_images)
X_test, y_test = create_dataset(test_images)
```

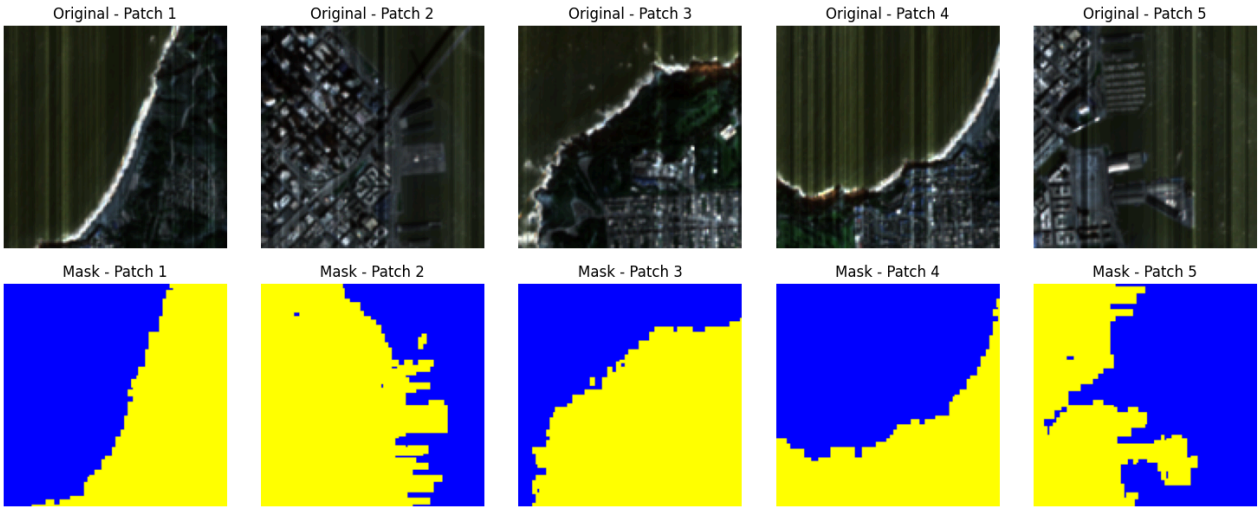
```
Total training patches: 231
Total training masks: 231
Total training patches: 42
Total training masks: 42
Total training patches: 60
Total training masks: 60
```

Data augmentation

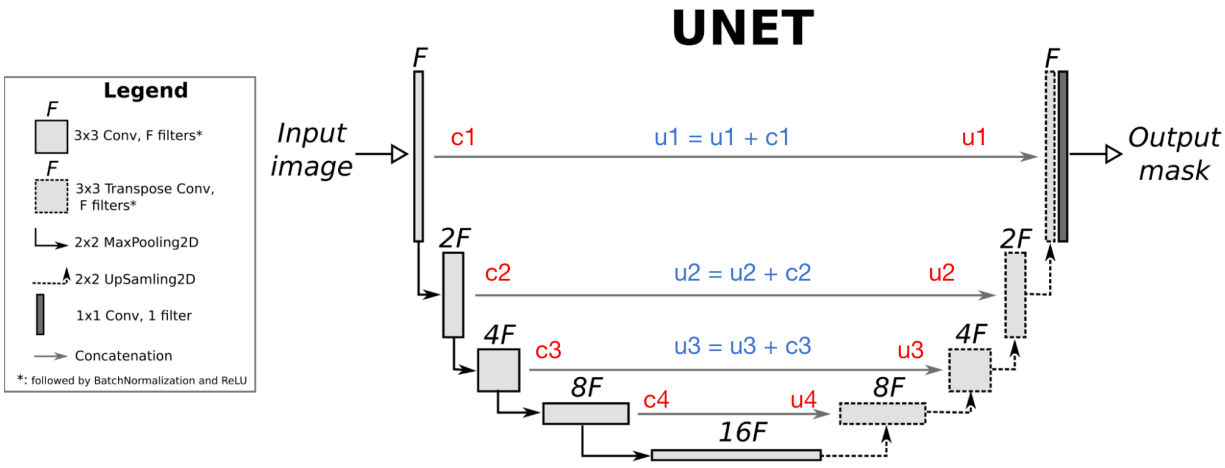
In [ ]:
```
import albumentations as A
import cv2

def augment_images_and_masks(images, masks, augmentations, count):
    """Applies augmentations to images and masks."""
    augmented_images = []
    augmented_masks = []

    for _ in range(count):
        for img, mask in zip(images, masks):
            # Apply augmentations
            transformed = augmentations(image=img, mask=mask)
            augmented_images.append(transformed['image'])
            augmented_masks.append(transformed['mask'])

    return np.array(augmented_images), np.array(augmented_masks)

# Define augmentation pipeline
def get_augmentation_pipeline():
    return A.Compose([
        A.HorizontalFlip(p=0.5),
        A.VerticalFlip(p=0.5),
        A.Rotate(limit=30, interpolation=cv2.INTER_NEAREST, p=0.5),
        A.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2, p=0.5),
        A.OneOf([
            A.CLAHE(clip_limit=2.0, p=0.3),
            A.GridDistortion(p=0.3),
            A.OpticalDistortion(distort_limit=0.3, shift_limit=0.2, interpolation=cv2.INTER_NEAREST, p=0.3),
        ], p=0.5),
        A.RandomCrop(width=128, height=128, p=1.0)  # Match the patch size
    ])

# Apply augmentation to training and validation sets
augmentation_pipeline = get_augmentation_pipeline()

# Example: Augmenting training data with 20x augmentation
X_train_aug, y_train_aug = augment_images_and_masks(X_train_reduced, Y_train_reduced, augmentation_pipeline, count=

# Example: Augmenting validation data with 10x augmentation
X_val_aug, y_val_aug = augment_images_and_masks(X_val, y_val, augmentation_pipeline, count=1)

# Combine original and augmented data for training and validation
X_train_augmented = np.concatenate([X_train, X_train_aug], axis=0)
y_train_augmented = np.concatenate([y_train, y_train_aug], axis=0)
X_val_augmented = np.concatenate([X_val, X_val_aug], axis=0)
y_val_augmented = np.concatenate([y_val, y_val_aug], axis=0)

# Print resulting dataset shapes
print(f"Augmented Training data: {X_train_augmented.shape}, Labels: {y_train_augmented.shape}")
print(f"Augmented Validation data: {X_val_augmented.shape}, Labels: {y_val_augmented.shape}")
```

```
Augmented Training data: (7851, 128, 128, 3), Labels: (7851, 128, 128, 2)
Augmented Validation data: (924, 128, 128, 3), Labels: (924, 128, 128, 2)
```

In [ ]:
```
visualize_patches_and_masks(X_train_augmented, y_train_augmented)
```



EXPLAIN SEGMENTATION MODEL: UNET



UNET is called UNET for its U shape, which consists of a downsampling patch, a bottleneck space that force a compact representation of the original image, and an upsampling path. This is just like an autoencoder. What's make it suitable for the image segmentation task is its ability to produce image reconstruction with high spatial detail, thus make it excels at assigning a class label to each pixel in an image.

Architecture overview

1. **Encoder (Contracting Path)**
   - **Purpose:** Captures spatial context and features.
   - **Components:**
     - Convolutional layers (to extract features).
     - ReLU activations and max-pooling (to downsample and retain essential information).
2. **Bottleneck**
   - **Purpose:** Acts as a bridge, encoding the most abstract representation of features before reconstruction.
   - **Components:**
     - Deeper convolutional layers without pooling.
3. **Decoder (Expanding Path)**
   - **Purpose:** Reconstructs spatial details and segmentation maps.
   - **Components:**
     - Transposed convolutions or upsampling layers (to increase resolution).
     - Skip connections (to merge low-level spatial details from the encoder with high-level decoder features).
4. **Skip Connections**
   - **Purpose:** Merge encoder and decoder outputs at each resolution level, preserving spatial details lost during downsampling.
5. **Output Layer**
   - **Purpose:** Produces the segmentation map.
   - **Components:**

- A convolutional layer with the desired number of channels (e.g., 1 for binary segmentation or `N` for multi-class segmentation).
- Sigmoid or Softmax activation for probabilities.

Skip connection

I want to especially point out the use of skip connection, which is what makes UNET different from an autoencoder (which struggles to reconstruct high-level details)

- In U-Net, the features from the encoder are concatenated with the features from the decoder at each level of the architecture. This means that the output feature maps from the encoder are combined with the feature maps produced by the decoder at the same spatial resolution.
- As images pass through the encoder, they undergo downsampling, which can lead to a loss of fine-grained details. Skip connections help retain this spatial information by providing direct access to the high-resolution feature maps from earlier layers in the encoder.

To demonstrate how the skip connection works, I annotated the image above with the output tensors of each Convolutional Layers in the downsampling path and the output tensors of up-sampling (transposed convolutional) layers in the upsampling path.

- In the Encoder, the size of the image gradually reduces while the depth gradually increases. Starting from 128x128x3 to 8x8x256

-> This basically means the network learns the “WHAT” information in the image, however it has lost the “WHERE” information

- In the decoder, the size of the image gradually increases and the depth gradually decreases. Starting from 8x8x256 to 128x128x1

-> Intuitively, the Decoder recovers the “WHERE” information (precise localization) by gradually applying up-sampling

- To get better precise locations, at every step of the decoder we use skip connections by concatenating the output of the transposed convolution layers with the feature maps from the Encoder at the same level (for example, u1 = u1 + c1, u2 = u2 = c2...) which mean that the output of each expansion feature maps is contributed by both the upsampling features and the original features in the downsampling path where a lot of information has not been loss in the process

Loss function and metrics

In segmentation tasks, the **loss function** measures how well the model predicts the segmentation map compared to the ground truth. **1. Binary Segmentation** For tasks with two classes (e.g., foreground vs. background):

- **Binary Cross-Entropy Loss (BCE):**

$$\text{BCE Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

  - y_i: Ground truth label for pixel i.
  - $\hat{y}_i$: Predicted probability for pixel i.

However, since I have one-hot encoded the masks label, I will use categorical cross-entropy loss with a softmax activation function

2. Specialized Losses for Imbalanced Datasets Segmentation often involves imbalanced data (e.g., small objects vs. large background), so Dice loss is a loss function especially used for imbalanced datasets. The Dice loss function help maximize overlap between prediction and ground truth:

$$\text{Dice Loss} = 1 - \frac{2\sum y\hat{y} + \epsilon}{\sum y + \sum \hat{y} + \epsilon}$$

3. Combination Losses In my implementation, I take the sum of BCE and dice loss to balances pixel-wise accuracy (BCE) and region overlap (Dice).

In [ ]:
```python
# Define Dice Loss Function using Loss subclassing
class DiceLoss(tf.keras.losses.Loss):
    def __init__(self, smooth=1e-6):
        super(DiceLoss, self).__init__()
        self.smooth = smooth

    def call(self, y_true, y_pred):
        # Flatten the input tensors
        y_true_f = tf.reshape(y_true, [-1])
        y_pred_f = tf.reshape(y_pred, [-1])

        intersection = tf.reduce_sum(y_true_f * y_pred_f)
        union = tf.reduce_sum(y_true_f) + tf.reduce_sum(y_pred_f)

        # Dice score (1 - Dice loss)
        dice = (2. * intersection + self.smooth) / (union + self.smooth)
        return 1 - dice

# Define the custom loss function (combined Dice loss and categorical crossentropy)
```

```
class CombinedLoss(tf.keras.losses.Loss):
    def __init__(self):
        super(CombinedLoss, self).__init__()

    def call(self, y_true, y_pred):
        dice_loss = DiceLoss()(y_true, y_pred)  # Use the DiceLoss class
        cce_loss = tf.keras.losses.categorical_crossentropy(y_true, y_pred)  # Use built-in CCE
        return dice_loss + cce_loss
```

4. Metric: IoU Focuses on intersection over union, penalizing false positives and false negatives.

$$\text{IoU Loss} = 1 - \frac{\sum(y \cdot \hat{y})}{\sum(y + \hat{y} - y \cdot \hat{y})}$$

In image segmentation, IoU is more widely used over accuracy since it focuses on the quality of overlap between predictions and ground truth, and it avoids misleading results in class-imbalanced datasets.

Intersection over Union (**IoU**) is often preferred over accuracy for segmentation tasks because it provides a more meaningful measure of how well the predicted segmentation aligns with the ground truth:

Accuracy Can Be Misleading in Imbalanced Datasets

$$\text{Accuracy} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total Pixels}}$$

- In segmentation tasks, background pixels often dominate, especially in tasks like small-object detection.
- A model predicting mostly background (majority class) can achieve high accuracy by correctly classifying background pixels while failing to detect foreground objects. For example, if 95% of the image is background, a model predicting only background achieves 95% accuracy, even though it completely misses the object.

IoU Focuses on Overlap: IoU is unaffected by the abundance of background pixels because it explicitly evaluates how well the predicted segmentation matches the target, focusing on the region of interest instead of overall correctness.

In [ ]:
```
# Accuracy: number of correct predictions / total predictions
def accuracy_metric(y_true, y_pred):
    correct_preds = tf.equal(tf.argmax(y_true, axis=-1), tf.argmax(y_pred, axis=-1))
    accuracy = tf.reduce_mean(tf.cast(correct_preds, tf.float32))
    return accuracy

# Intersection over Union (IoU)
def mean_iou_metric(y_true, y_pred, num_classes=2):
    iou_list = []
    for i in range(num_classes):
        true_class = tf.equal(tf.argmax(y_true, axis=-1), i)
        pred_class = tf.equal(tf.argmax(y_pred, axis=-1), i)
        intersection = tf.reduce_sum(tf.cast(true_class & pred_class, tf.float32))
        union = tf.reduce_sum(tf.cast(true_class | pred_class, tf.float32))
        iou = intersection / (union + tf.keras.backend.epsilon())  # Add epsilon to avoid division by 0
        iou_list.append(iou)
    return tf.reduce_mean(iou_list)  # Average IoU over all classes

# Dice coefficient
def dice_coefficient_metric(y_true, y_pred, num_classes=2):
    dice_list = []
    for i in range(num_classes):
        true_class = tf.equal(tf.argmax(y_true, axis=-1), i)
        pred_class = tf.equal(tf.argmax(y_pred, axis=-1), i)
        intersection = tf.reduce_sum(tf.cast(true_class & pred_class, tf.float32))
        dice = (2 * intersection) / (tf.reduce_sum(tf.cast(true_class, tf.float32)) + tf.reduce_sum(tf.cast(pred_cl
        dice_list.append(dice)
    return tf.reduce_mean(dice_list)  # Average Dice coefficient over all classes

def cross_entropy_loss(y_true, y_pred):
    return np.mean(tf.keras.losses.categorical_crossentropy(y_true, y_pred))
```

Pseudocode

1. The Encoder (Contracting Path) Extract features and reduce spatial dimensions.

```
def encoder_block(input_tensor, filters):
    x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(input_tensor)
    x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(x)
    p = layers.MaxPooling2D((2, 2))(x)
    return x, p
```

- **First Convolution:** Detects simple patterns like edges or corners. The `padding='same'` ensures that the output retains the same spatial dimensions.
- **Second Convolution:** Builds upon features detected earlier, combining them into more complex patterns.
- **Max Pooling:** Reduces spatial resolution to focus on "what" the features represent, rather than "where" they are located.

In the U-Net:

- This operation is applied multiple times (4 times in my actual code implementation below), progressively halving the resolution while increasing the number of filters. This is the part in my code where the number of increasing filters are defined.

`..., filter_num=[64, 128, 256, 512, 1024], ...`

2. **The Bottleneck**: Encodes the abstract, high-level representation of the input.

```
def bottleneck(input_tensor, filters):
    x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(input_tensor)
    x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(x)
    return x
```

- Similar to the encoder block, the bottleneck uses two convolutional layers to process features.
- However, there's no pooling, as the bottleneck works on the already downsampled features from the encoder.

3. **The Decoder (Expanding Path)**: Reconstruct spatial resolution while combining encoder features for precise segmentation.

```
def decoder_block(input_tensor, skip_tensor, filters):
    x = layers.Conv2DTranspose(filters, (2, 2), strides=(2, 2), padding='same')(input_tensor)
    x = layers.Concatenate()([x, skip_tensor])  # Combine decoder and encoder features
    x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(x)
    x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(x)
    return x
```

- **Transpose Convolution:** Doubles the spatial resolution (e.g., $8 \times 8\backslash) \rightarrow \backslash(16 \times 16)$.
- **Skip Connections:** Copy features from the encoder directly into the decoder at the same level of resolution. This helps the decoder recover spatial details that were lost during downsampling.
- **Convolutions:** Combine and refine the upsampled features with the encoder features.

4. **The Output Layer**: Produce pixel-wise predictions.

- A 1x1 convolution is applied to reduce the number of filters to the number of segmentation classes (e.g., 1 for binary segmentation).
- An activation function (e.g., `sigmoid`) converts the output to probabilities for each pixel.

```
outputs = layers.Conv2D(num_classes, (1, 1), activation='sigmoid')(d4)
```

Putting It All Together

The entire U-Net architecture connects these components:

```
def build_unet(input_shape=(128, 128, 3), num_classes=1):
    inputs = layers.Input(input_shape)

    # Encoder
    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 256)
    s4, p4 = encoder_block(p3, 512)

    # Bottleneck
    b = bottleneck(p4, 1024)

    # Decoder
    d1 = decoder_block(b, s4, 512)
    d2 = decoder_block(d1, s3, 256)
    d3 = decoder_block(d2, s2, 128)
    d4 = decoder_block(d3, s1, 64)

    # Output
    outputs = layers.Conv2D(num_classes, (1, 1), activation='sigmoid')(d4)

    return Model(inputs, outputs)
```

UNET

To leverage pretrained backbone for reducing the complexity of training the UNET from scratch, I utilize the library keras-unet-collection which intergrate various variations of UNET models and pretrained backbone

In [ ]:
```
!pip install keras-unet-collection
```

```
Requirement already satisfied: keras-unet-collection in /usr/local/lib/python3.10/dist-packages (0.1.13)
```

In [ ]:
```
from keras_unet_collection import models, losses
```

Encoder and Decoder Parameters:

- stack_num_down=2: This indicates that each downsampling block in the encoder will consist of 2 convolutional layers. Each block extracts features and reduces the spatial resolution.
- stack_num_up=2: Each upsampling block in the decoder will also have 2 convolutional layers. This block upscales the spatial resolution while combining the features from the encoder through skip connections.
- pool='max': Max pooling is used for downsampling in the encoder. It reduces the spatial dimensions of the input by taking the maximum value from the feature map within a certain window (e.g., 2x2).
- unpool='bilinear': For upsampling in the decoder, bilinear interpolation is used to resize the feature maps. This is a method of interpolation that helps to expand the feature maps without introducing artifacts, maintaining smooth transitions.

Pretrained backbone: Resnet50:

- freeze_backbone=True: This parameter freezes the layers of the ResNet50 backbone during training, meaning their weights will not be updated. This is useful when you don't want to retrain the entire ResNet model but only fine-tune the U-Net's decoder and output layers. Freezing the backbone can speed up training and prevent overfitting when using a smaller dataset.
- freeze_batch_norm=True: This freezes the batch normalization layers of the backbone. This is typically done when freezing the backbone to avoid updating the batch normalization statistics during training, which might interfere with the training process.

In [ ]:
```
model_Unet = models.unet_2d(input_shape, filter_num=[64, 128, 256, 512, 1024],
                            n_labels=num_labels,
                            stack_num_down=2, stack_num_up=2,
                            activation='ReLU',
                            output_activation='Softmax',
                            batch_norm=True, pool='max', unpool='bilinear',
                            backbone='ResNet50', weights='imagenet',
                            freeze_backbone=True, freeze_batch_norm=True,
                            name='unet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_
ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

In [ ]:
```
print(model_Unet.summary())
```

```
Model: "unet_model"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	–
ResNet50_backbone (Functional)	[(None, 64, 64, 64), (None, 32, 32, 256), (None, 16, 16, 512), (None, 8, 8, 1024)]	8,589,184	input_layer[0][0]
unet_up0_decode_unpool (UpSampling2D)	(None, 16, 16, 1024)	0	ResNet50_backbone[0][…
unet_up0_conv_before_con… (Conv2D)	(None, 16, 16, 512)	4,718,592	unet_up0_decode_unpoo…
unet_up0_conv_before_con… (BatchNormalization)	(None, 16, 16, 512)	2,048	unet_up0_conv_before_…
unet_up0_conv_before_con… (ReLU)	(None, 16, 16, 512)	0	unet_up0_conv_before_…
unet_up0_concat (Concatenate)	(None, 16, 16, 1024)	0	unet_up0_conv_before_… ResNet50_backbone[0][…
unet_up0_conv_after_conc… (Conv2D)	(None, 16, 16, 512)	4,718,592	unet_up0_concat[0][0]
unet_up0_conv_after_conc… (BatchNormalization)	(None, 16, 16, 512)	2,048	unet_up0_conv_after_c…
unet_up0_conv_after_conc… (ReLU)	(None, 16, 16, 512)	0	unet_up0_conv_after_c…
unet_up0_conv_after_conc… (Conv2D)	(None, 16, 16, 512)	2,359,296	unet_up0_conv_after_c…
unet_up0_conv_after_conc… (BatchNormalization)	(None, 16, 16, 512)	2,048	unet_up0_conv_after_c…
unet_up0_conv_after_conc… (ReLU)	(None, 16, 16, 512)	0	unet_up0_conv_after_c…
unet_up1_decode_unpool (UpSampling2D)	(None, 32, 32, 512)	0	unet_up0_conv_after_c…
unet_up1_conv_before_con… (Conv2D)	(None, 32, 32, 256)	1,179,648	unet_up1_decode_unpoo…
unet_up1_conv_before_con… (BatchNormalization)	(None, 32, 32, 256)	1,024	unet_up1_conv_before_…
unet_up1_conv_before_con… (ReLU)	(None, 32, 32, 256)	0	unet_up1_conv_before_…
unet_up1_concat (Concatenate)	(None, 32, 32, 512)	0	unet_up1_conv_before_… ResNet50_backbone[0][…
unet_up1_conv_after_conc… (Conv2D)	(None, 32, 32, 256)	1,179,648	unet_up1_concat[0][0]
unet_up1_conv_after_conc… (BatchNormalization)	(None, 32, 32, 256)	1,024	unet_up1_conv_after_c…
unet_up1_conv_after_conc… (ReLU)	(None, 32, 32, 256)	0	unet_up1_conv_after_c…
unet_up1_conv_after_conc… (Conv2D)	(None, 32, 32, 256)	589,824	unet_up1_conv_after_c…
unet_up1_conv_after_conc… (BatchNormalization)	(None, 32, 32, 256)	1,024	unet_up1_conv_after_c…
unet_up1_conv_after_conc… (ReLU)	(None, 32, 32, 256)	0	unet_up1_conv_after_c…
unet_up2_decode_unpool (UpSampling2D)	(None, 64, 64, 256)	0	unet_up1_conv_after_c…
unet_up2_conv_before_con… (Conv2D)	(None, 64, 64, 128)	294,912	unet_up2_decode_unpoo…
unet_up2_conv_before_con… (BatchNormalization)	(None, 64, 64, 128)	512	unet_up2_conv_before_…
unet_up2_conv_before_con… (ReLU)	(None, 64, 64, 128)	0	unet_up2_conv_before_…
unet_up2_concat (Concatenate)	(None, 64, 64, 192)	0	unet_up2_conv_before_… ResNet50_backbone[0][…
unet_up2_conv_after_conc… (Conv2D)	(None, 64, 64, 128)	221,184	unet_up2_concat[0][0]

unet_up2_conv_after_conc… (BatchNormalization)	(None, 64, 64, 128)	512	unet_up2_conv_after_c…
unet_up2_conv_after_conc… (ReLU)	(None, 64, 64, 128)	0	unet_up2_conv_after_c…
unet_up2_conv_after_conc… (Conv2D)	(None, 64, 64, 128)	147,456	unet_up2_conv_after_c…
unet_up2_conv_after_conc… (BatchNormalization)	(None, 64, 64, 128)	512	unet_up2_conv_after_c…
unet_up2_conv_after_conc… (ReLU)	(None, 64, 64, 128)	0	unet_up2_conv_after_c…
unet_up3_decode_unpool (UpSampling2D)	(None, 128, 128, 128)	0	unet_up2_conv_after_c…
unet_up3_conv_before_con… (Conv2D)	(None, 128, 128, 64)	73,728	unet_up3_decode_unpoo…
unet_up3_conv_before_con… (BatchNormalization)	(None, 128, 128, 64)	256	unet_up3_conv_before_…
unet_up3_conv_before_con… (ReLU)	(None, 128, 128, 64)	0	unet_up3_conv_before_…
unet_up3_conv_after_conc… (Conv2D)	(None, 128, 128, 64)	36,864	unet_up3_conv_before_…
unet_up3_conv_after_conc… (BatchNormalization)	(None, 128, 128, 64)	256	unet_up3_conv_after_c…
unet_up3_conv_after_conc… (ReLU)	(None, 128, 128, 64)	0	unet_up3_conv_after_c…
unet_up3_conv_after_conc… (Conv2D)	(None, 128, 128, 64)	36,864	unet_up3_conv_after_c…
unet_up3_conv_after_conc… (BatchNormalization)	(None, 128, 128, 64)	256	unet_up3_conv_after_c…
unet_up3_conv_after_conc… (ReLU)	(None, 128, 128, 64)	0	unet_up3_conv_after_c…
unet_output (Conv2D)	(None, 128, 128, 2)	130	unet_up3_conv_after_c…
unet_output_activation (Softmax)	(None, 128, 128, 2)	0	unet_output[0][0]

Total params: 24,157,442 (92.15 MB)
Trainable params: 15,562,498 (59.37 MB)
Non-trainable params: 8,594,944 (32.79 MB)
None

I have many iteration of trying to stabilize the training of this UNET model , since the validation loss keeps oscilating instead of converging near training loss.

Iteration 1 Vanillla configuration

My first try is with a vanilla configuration:

- One loss function: categorical cross-entropy
- Adam optimizer with a fix learning rate = 0.0001
- Freeze the entire Resnet50 backbone
- I also check the original code of the library in their GitHub and I don't find any regularizer being included in the algorithm. My first trial did not fix this but I will fix this in later trials

In [ ]:
```
model_Unet.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate = 0.0001),
              metrics=['accuracy', losses.iou_seg])
```

In [ ]:
```
#define a callback on the validation dataset
es = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                      patience=10,
                                      restore_best_weights=True,
                                      verbose=1)
```

In [ ]:
```
# Define ModelCheckpoint callback to save the best model based on validation loss
model_checkpoint = ModelCheckpoint('/content/drive/MyDrive/Minerva Academic/3rd year/Fall 2024/CS156/Pipeline 3/Che
                                   monitor='val_loss',          # Monitor validation loss
                                   save_best_only=True,         # Save the model only when validation loss improves
                                   verbose=0)
```

In [ ]:
```
start1 = datetime.now()
```

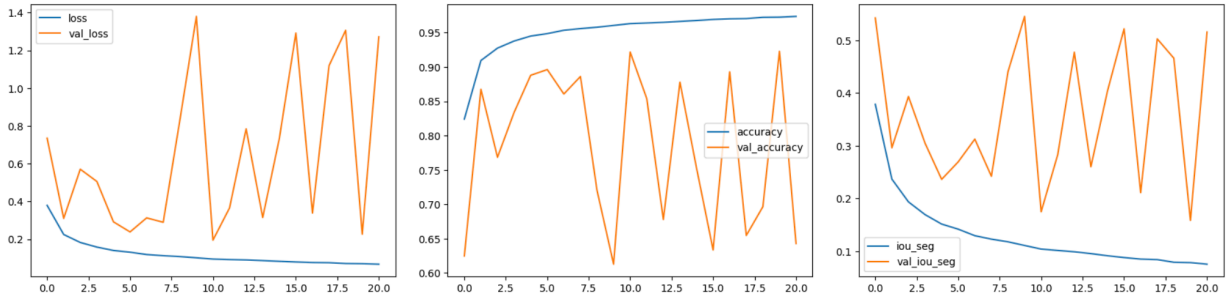
```
Unet_history = model_Unet.fit(train_dataset,
                    validation_data=validation_dataset,
                    verbose=0,
                    batch_size = batch_size,
                    shuffle=False,
                    callbacks=[es, model_checkpoint],
                    epochs=30)

stop1 = datetime.now()
```

```
Epoch 21: early stopping
Restoring model weights from the end of the best epoch: 11.
```

In [ ]:
```
print("UNet execution time is: ", stop1-start1)
```

```
UNet execution time is:  0:24:54.716281
```

In [ ]:
```
Unet_history_df = pd.DataFrame(Unet_history.history)
Unet_history_df[['loss', 'val_loss']].plot()
Unet_history_df[['accuracy', 'val_accuracy']].plot()
Unet_history_df[['iou_seg', 'val_iou_seg']].plot()
```



As you can see, with a vanilla configuration, the validation loss oscilates like crazy while the training loss decreases smoothly.

Iteration 2: Add learning rate scheduler + Combine loss function

CosineDecayRestart learning rate scheduler

- initial_learning_rate = 0.00001: The learning rate is set to a very small value (0.00001), which indicates that the model will start learning slowly. This is useful when working with models that may be prone to overshooting during training, and the smaller learning rate helps with gradual convergence.
- total_steps = steps_per_epoch * epochs: The total number of steps is calculated by multiplying the number of steps per epoch by the total number of epochs (20). This specifies how many total steps the decay will apply to, typically defined by the number of training steps.
- t_mul=2: This parameter increases the period of the cosine decay after each restart. It means that after each restart, the cycle length will be multiplied by 2.
- m_mul=0.9: This reduces the amplitude (maximum value) of the cosine decay after each restart. It makes the learning rate approach zero more gradually after each restart.
- alpha=0.1: This sets the minimum value for the learning rate as a fraction of the initial learning rate. After decay, the learning rate will never fall below 10% of the initial learning rate.

In [ ]:
```
# Hyperparameters
initial_learning_rate = 0.00001  # Small starting learning rate
epochs = 20
batch_size = 32  # Assuming this is defined already

# Calculate the number of steps in the dataset
steps_per_epoch = len(train_dataset) // batch_size  # Number of steps per epoch
total_steps = steps_per_epoch * epochs  # Total number of steps in the training

# Define the learning rate scheduler (Cosine decay with warmup)
lr_warmup_decayed_fn = tf.keras.optimizers.schedules.CosineDecayRestarts(
    initial_learning_rate,
    total_steps,  # Decay over the entire training duration
    t_mul=2,  # Multiply the period of decay by 2 after each restart
    m_mul=0.9,  # Scale the amplitude of the restart
    alpha=0.1,  # Minimum learning rate as a fraction of initial
)

# Create the optimizer using the learning rate scheduler
optimizer = tf.keras.optimizers.Adam(learning_rate=lr_warmup_decayed_fn)
```

Combine loss function

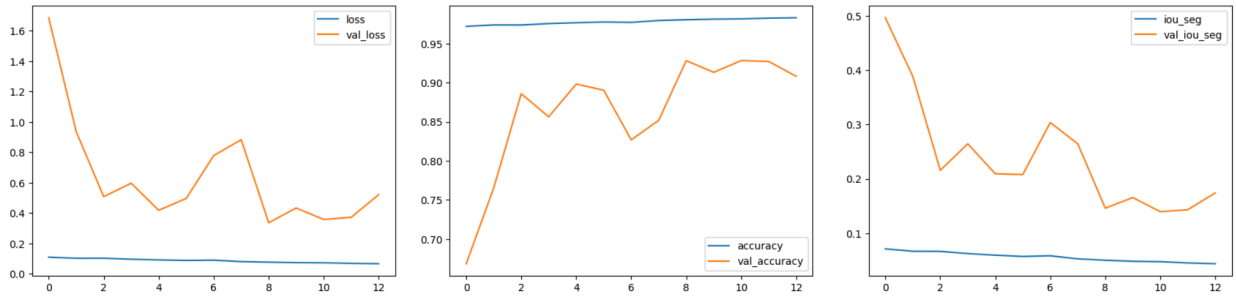
I have defined the loss function: Dice loss + Categorical cross-entropy loss above. I call the combine loss function when compiling model

In [ ]:
```
# Compile the model with the new optimizer and combined loss function
model_Unet.compile(optimizer=optimizer,
                   loss = CombinedLoss(),
                   metrics=['accuracy', losses.iou_seg])
```

In [ ]:
```
# Training the model
start_time = datetime.now()

Unet_history = model_Unet.fit(
    train_dataset,
    validation_data=validation_dataset,
    verbose=1,
    batch_size=32,
    epochs=20,
    callbacks=[es, model_checkpoint],
    shuffle=True
)

end_time = datetime.now()
print(f"Training time: {end_time - start_time}")
```

```
Epoch 1/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 452ms/step - accuracy: 0.9715 - iou_seg: 0.0727 - loss: 0.1121
/usr/local/lib/python3.10/dist-packages/keras/src/saving/serialization_lib.py:390: UserWarning: The object being s
erialized includes a `lambda`. This is unsafe. In order to reload the object, you will have to pass `safe_mode=Fal
se` to the loading function. Please avoid using `lambda` in the future, and use named Python functions instead. Th
is is the `lambda` being serialized:                     loss=lambda y_true, y_pred: dice_loss(y_true, y_pred) + t
f.keras.losses.categorical_crossentropy(y_true, y_pred),

  return {key: serialize_keras_object(value) for key, value in obj.items()}
152/152 ━━━━━━━━━━━━━━━━━━━━ 105s 527ms/step - accuracy: 0.9715 - iou_seg: 0.0727 - loss: 0.1121 - val_accuracy:
0.6684 - val_iou_seg: 0.4967 - val_loss: 1.6861
Epoch 2/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 59s 388ms/step - accuracy: 0.9740 - iou_seg: 0.0665 - loss: 0.1026 - val_accuracy: 0.
7653 - val_iou_seg: 0.3888 - val_loss: 0.9360
Epoch 3/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 87s 419ms/step - accuracy: 0.9753 - iou_seg: 0.0630 - loss: 0.0971 - val_accuracy: 0.
8858 - val_iou_seg: 0.2158 - val_loss: 0.5084
Epoch 4/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 58s 378ms/step - accuracy: 0.9754 - iou_seg: 0.0630 - loss: 0.0971 - val_accuracy: 0.
8562 - val_iou_seg: 0.2646 - val_loss: 0.5971
Epoch 5/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 63s 417ms/step - accuracy: 0.9766 - iou_seg: 0.0596 - loss: 0.0920 - val_accuracy: 0.
8981 - val_iou_seg: 0.2094 - val_loss: 0.4188
Epoch 6/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 76s 377ms/step - accuracy: 0.9787 - iou_seg: 0.0548 - loss: 0.0839 - val_accuracy: 0.
8903 - val_iou_seg: 0.2081 - val_loss: 0.4968
Epoch 7/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 380ms/step - accuracy: 0.9768 - iou_seg: 0.0592 - loss: 0.0915 - val_accuracy: 0.
8267 - val_iou_seg: 0.3037 - val_loss: 0.7790
Epoch 8/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 58s 383ms/step - accuracy: 0.9797 - iou_seg: 0.0523 - loss: 0.0803 - val_accuracy: 0.
8518 - val_iou_seg: 0.2646 - val_loss: 0.8832
Epoch 9/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 89s 428ms/step - accuracy: 0.9801 - iou_seg: 0.0512 - loss: 0.0789 - val_accuracy: 0.
9281 - val_iou_seg: 0.1461 - val_loss: 0.3370
Epoch 10/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 74s 376ms/step - accuracy: 0.9815 - iou_seg: 0.0477 - loss: 0.0730 - val_accuracy: 0.
9132 - val_iou_seg: 0.1656 - val_loss: 0.4337
Epoch 11/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 376ms/step - accuracy: 0.9815 - iou_seg: 0.0477 - loss: 0.0733 - val_accuracy: 0.
9282 - val_iou_seg: 0.1397 - val_loss: 0.3584
Epoch 12/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 58s 382ms/step - accuracy: 0.9830 - iou_seg: 0.0441 - loss: 0.0674 - val_accuracy: 0.
9271 - val_iou_seg: 0.1432 - val_loss: 0.3726
Epoch 13/20
152/152 ━━━━━━━━━━━━━━━━━━━━ 59s 386ms/step - accuracy: 0.9835 - iou_seg: 0.0429 - loss: 0.0655 - val_accuracy: 0.
9080 - val_iou_seg: 0.1743 - val_loss: 0.5217
Epoch 13: early stopping
Restoring model weights from the end of the best epoch: 9.
Training time: 0:16:12.575463
```

I changed the configuration and train for more epoch. The best validation loss is recorded at 0.337. The validation performance does not oscilate as hard as before



Iteration 3: Finetuning Resnet50 backbone

Since Resnet is trained on ImageNet which is a whole different domain than satellite image, I choose to finetune it to see if we can improve model performance. The backbone has 143 layers and I finetune from layer 100.

In [ ]:
```
len(model_Unet.get_layer('ResNet50_backbone').layers)
```

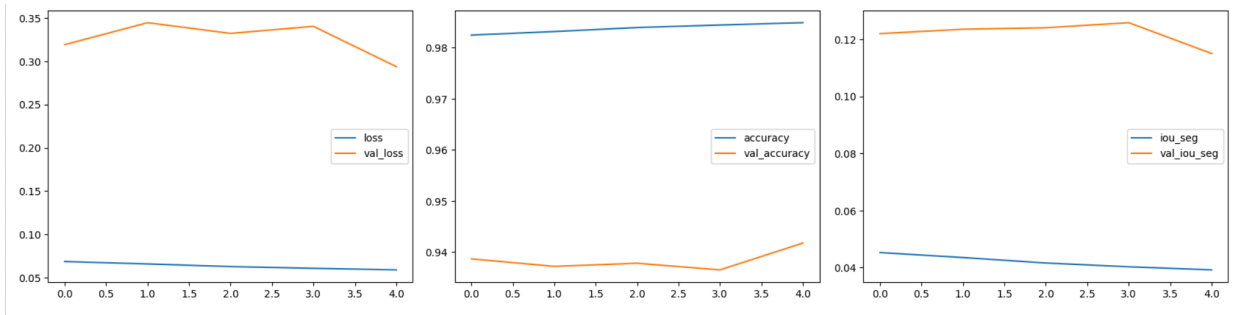
Out[ ]:
```
143
```

In [ ]:
```
# Unfreeze some layers in the backbone (ResNet50)
for layer in model_Unet.get_layer('ResNet50_backbone').layers[100:]:
    layer.trainable = True  # Unfreeze the last 43 layers for fine-tuning
```

Follow the same process, the model is abl to finetune for only 4 more epochs before the validation loss stops improving again

In [ ]:
```
# Training the model
start_time = datetime.now()
initial_epochs = len(model_Unet_history.epoch)
Unet_history_finetune = model_Unet.fit(
    train_dataset,
    validation_data=validation_dataset,
    verbose=1,
    batch_size = 32,
    initial_epoch = initial_epochs,
    epochs= initial_epochs + 10,
    callbacks=[es, model_checkpoint],
    shuffle=True
)

end_time = datetime.now()
print(f"Training time: {end_time - start_time}")
```

```
Epoch 24/33
152/152 ━━━━━━━━━━━━━━━━━━━━ 61s 402ms/step - accuracy: 0.9827 - iou_seg: 0.0449 - loss: 0.0678 - val_accuracy: 0.
9387 - val_iou_seg: 0.1220 - val_loss: 0.3190
Epoch 25/33
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 403ms/step - accuracy: 0.9833 - iou_seg: 0.0434 - loss: 0.0655 - val_accuracy: 0.
9372 - val_iou_seg: 0.1235 - val_loss: 0.3443
Epoch 26/33
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 406ms/step - accuracy: 0.9840 - iou_seg: 0.0415 - loss: 0.0625 - val_accuracy: 0.
9378 - val_iou_seg: 0.1240 - val_loss: 0.3319
Epoch 27/33
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 407ms/step - accuracy: 0.9846 - iou_seg: 0.0400 - loss: 0.0602 - val_accuracy: 0.
9365 - val_iou_seg: 0.1258 - val_loss: 0.3401
Epoch 28/33
152/152 ━━━━━━━━━━━━━━━━━━━━ 61s 398ms/step - accuracy: 0.9851 - iou_seg: 0.0388 - loss: 0.0582 - val_accuracy: 0.
9418 - val_iou_seg: 0.1150 - val_loss: 0.2936
Epoch 28: early stopping
Restoring model weights from the end of the best epoch: 24.
Training time: 0:06:08.384227
```



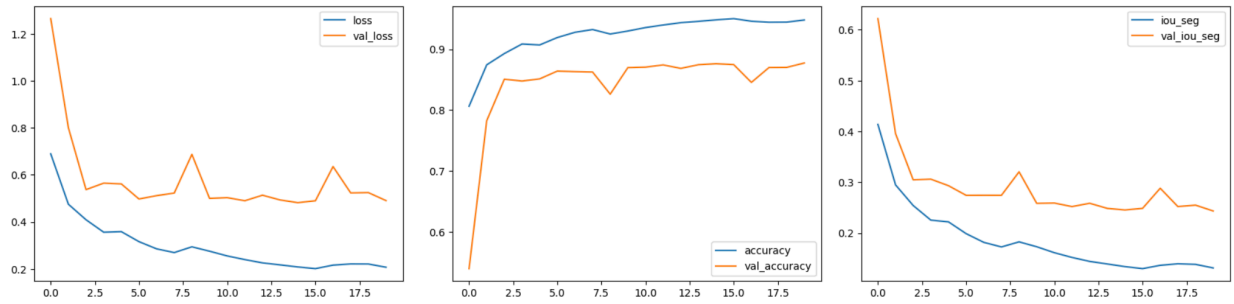
After finetuning, the best validation loss improved from 0.337 to 0.319. The validation improved not significantly and seems to hit a wall so I stop at this stage

Iteration 4: Add regularizer and decrease learning rate from 0.0001 to 0.00001

To avoid overfitting, I add L2 regularize in each Conv2D layer

In [ ]:
```python
from tensorflow.keras.regularizers import l2
def apply_regularization(model):
    for i, layer in enumerate(model.layers):
        # Apply L2 regularization to Conv2D layers
        if isinstance(layer, layers.Conv2D):
            model.layers[i].kernel_regularizer = l2(1e-4)

    return model

# Rebuild model with regularization
model_Unet = apply_regularization(model_Unet)
```

Since the model seems cannot improve further, I decreased the learning rate to see if it can escape the current local minima or not.



The regularizer seems to stabilize the training process since the validation loss does not oscilatting as hard. It could also because with smaller learning rate, the gradient update becomes smoother. However, since I use CosineDecayRestart learning rate, the performance seems to have a periodic spike in validation loss. After the adjustment period, it goes back to the loss trajectory. This show that the model is really sensitive to learning rate

However, since the learning rate is reallly low, the model took so long to converge. The early stopping stops the execuation at 0.4816. I will takes the stable training as a good sign but I do not have resources to continue to traini it further. I assume that if I continue training for more epochs, the performace will eventually converge to the previous best validation loss that the model has happended to reach (0.319). Therefore, I will use the previous best validation loss to evaluate the model performance on test set

In [ ]:
```python
def evaluate_performance (y_test, y_pred):
  # Evaluate each metric on the test set
  accuracy = accuracy_metric(y_test, y_pred)
  mean_iou = mean_iou_metric(y_test, y_pred)
  dice_coefficient = dice_coefficient_metric(y_test, y_pred)
  cross_entropy = cross_entropy_loss(y_test, y_pred)
  # Print the results
  print(f"Accuracy: {accuracy.numpy():.4f}")
  print(f"Mean IoU: {mean_iou.numpy():.4f}")
  print(f"Dice Coefficient: {dice_coefficient.numpy():.4f}")
  print(f"Cross-Entropy Loss: {cross_entropy:.4f}")
```

In [ ]:
```python
evaluate_performance (y_test, y_pred)
```

```
Accuracy: 0.9760
Mean IoU: 0.9530
Dice Coefficient: 0.9759
Cross-Entropy Loss: 0.0644
```

1. **Accuracy: 0.9760**

   - **Interpretation**: The accuracy is very high, meaning that 97.6% of the pixels are correctly classified in the overall image.
   - **Details**: Accuracy in segmentation refers to the percentage of correctly predicted pixels (including both background and foreground). While high accuracy is generally good, it can be misleading in cases where the dataset is **imbalanced**, as it doesn't differentiate between how well the model performs on the foreground (objects of interest) versus the background.
2. **Mean Intersection over Union (Mean IoU): 0.9530**

   - **Interpretation**: The mean IoU is also very high at 95.3%, which indicates a strong overlap between the predicted segmentation mask and the true mask for each class (water/non-water).
   - **Details**: IoU is a measure of how well the predicted segmentation matches the ground truth, considering both false positives and false negatives. It ranges from 0 (no overlap) to 1 (perfect overlap). A value of 0.9530 means the predicted areas for each class overlap with the ground truth areas at a high rate, making it a strong indicator of segmentation quality. The "mean" refers to the average IoU across all classes.
3. **Dice Coefficient: 0.9759**

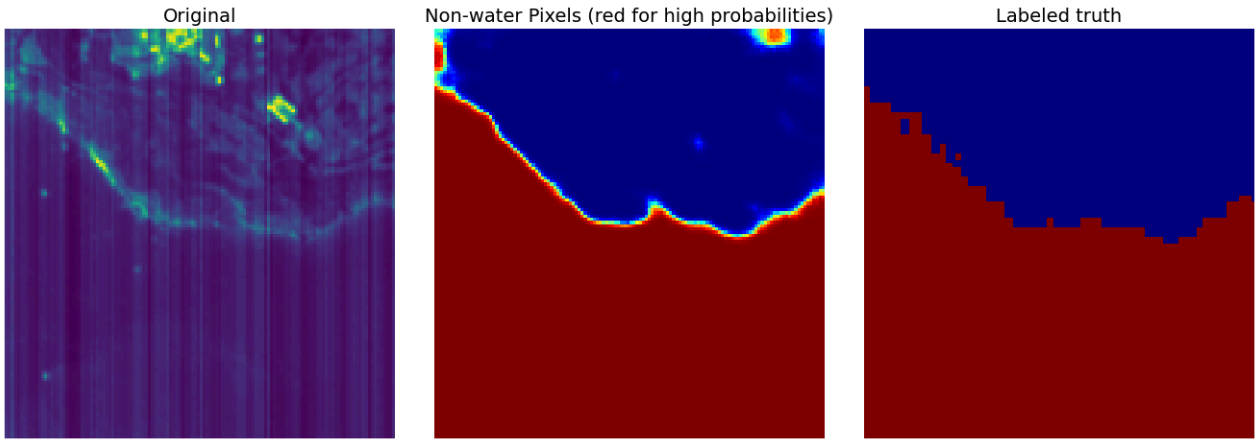
   - **Interpretation**: The Dice Coefficient is 97.59%, which is a very high value, indicating a strong agreement between the predicted and true segmentation masks.
   - **Details**: The Dice Coefficient is another metric for evaluating the overlap between the predicted and actual segmented regions. It is similar to IoU but places slightly more emphasis on false negatives. A value of 0.9759 is very close to perfect, indicating that the segmentation is almost identical to the ground truth.

4. **Cross-Entropy Loss: 0.0644**

   - **Interpretation**: The cross-entropy loss of 0.0644 is relatively low, indicating that the model's predictions are quite close to the true labels.
   - **Details**: Cross-entropy loss is used to measure the error in classification tasks, particularly for pixel-wise classification in segmentation. A lower value means the model's predicted probabilities are close to the actual class labels. A value of 0.0644 is small, indicating good performance, though it could be further optimized depending on the model's specific goals.

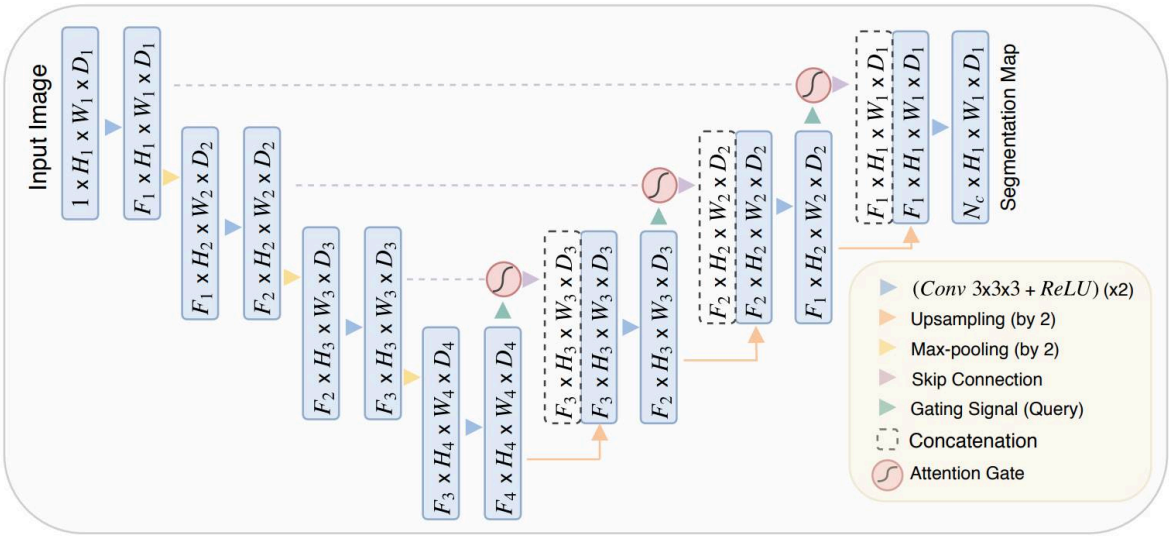
Together, these results indicate a **high-performing segmentation model** with a good balance between accuracy and the quality of the predicted segmentation. The high IoU and Dice Coefficient confirm that the model's predicted regions align closely with the true regions, while the low cross-entropy loss shows it is efficiently learning the class distributions.

In [ ]:
```
i_sample = 12

def ax_decorate_box(ax):
    [j.set_linewidth(0) for j in ax.spines.values()]
    ax.tick_params(axis="both", which="both", bottom=False, top=False, \
                labelbottom=False, left=False, right=False, labelleft=False)
    return ax
def visualize_mask_prediction (prediction, ground_truth):
  # Define the plot layout (3 subplots in 1 row)
  fig, AX = plt.subplots(1, 3, figsize=(13, (13-0.2)/3))
  plt.subplots_adjust(0, 0, 1, 1, hspace=0, wspace=0.1)

  # Decorate each axis for consistent appearance
  for ax in AX:
      ax = ax_decorate_box(ax)

  # Plot the original input (grayscale image)
  AX[0].pcolormesh(np.mean(X_test[i_sample, ...,], axis=-1))
  AX[0].set_title("Original", fontsize=14)

  # Plot the predicted output (predicted probabilities for the first class)
  AX[1].pcolormesh(y_pred[i_sample, ..., 0], cmap=plt.cm.jet)
  AX[1].set_title("Non-water Pixels (red for high probabilities)", fontsize=14)

  # Plot the true labels (ground truth)
  AX[2].pcolormesh(y_test[i_sample, ..., 0], cmap=plt.cm.jet)
  AX[2].set_title("Labeled truth", fontsize=14)

  # Show the plot
  plt.show()

visualize_mask_prediction (y_pred, y_test)
```



From the visualization of the predicted mask, we can see that the model is able to differentiate 2 classes (water and non-water) well. It can also quite correctly predict the boundary between 2 classes. While the ground truth masks have quite sharp pixel delineation along the boundary, the predicted mask is able to smooth out the prediction with a lower probability at the exact decision boundary.

ATTENTION UNET

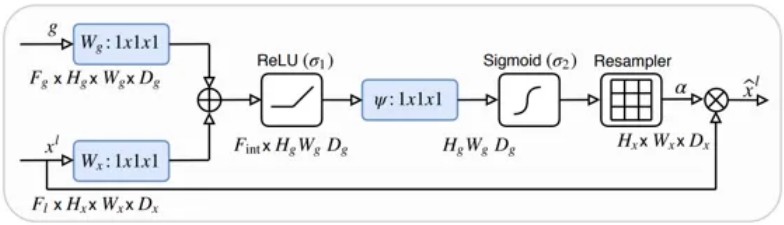


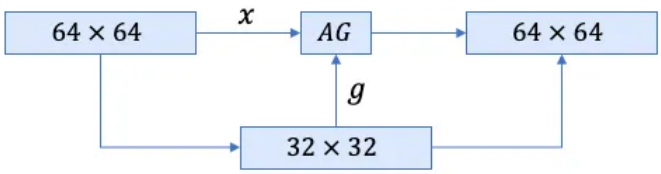
Recalling that the U-Net uses skip connections that combine spatial information from the downsampling path with the upsampling path. However, this brings across many redundant low-level feature extractions, as feature representation is poor in the initial layers.

Attention UNET arises to tackle this redundancy using its attention mechanism:

The Attention UNET introduces a novel Attention Gate that enables the UNET architecture to focus on the important regions or structures with different shapes and sizes. The models trained with Attention Gate implicitly learn to suppress irrelevant regions in an input image while highlighting salient features useful for a specific task. It does so by both using the upsampled feature map and the skip connection feature map and provides a highlighted feature map. In the highlighted feature map, the irrelevant features are suppressed.

This enables us to eliminate the necessity of using any explicit localisation modules.





The attention gate operates by taking two inputs: the vector **x** and the vector **g**.

- **g** is derived from a deeper layer of the network, making it smaller in size but richer in feature representation due to its position in the network hierarchy.
  - For instance, **x** might be a tensor of size 64x64x64 (filters x height x width), whereas **g** could be of size 32x32x32.

To align these two vectors, **x** is processed using a strided convolution that reduces its spatial dimensions to 64x32x32, while **g** is passed through a 1x1 convolution to match this size (64x32x32).

- After this, the two vectors are added element-wise. This addition enhances the weights corresponding to aligned features and diminishes the weights of misaligned features.

The sum is then activated using ReLU, and a 1x1 convolution further compresses the output to 1x32x32 dimensions. A sigmoid activation is applied, scaling the resulting values between 0 and 1, which results in the attention coefficients. Coefficients nearing 1 highlight more important features.

- These attention coefficients are then upsampled to the original dimensions of **x** (64x64) through trilinear interpolation.

Finally, the attention coefficients are multiplied element-wise with the original **x** vector, adjusting the feature map based on the relevance determined by the coefficients. This modified **x** vector is then passed along the skip connection.

Pseudocode

```
def attention_gate(g, s, num_filters):
    Wg = L.Conv2D(num_filters, 1, padding="same")(g)
    Wg = L.BatchNormalization()(Wg)

    Ws = L.Conv2D(num_filters, 1, padding="same")(s)
    Ws = L.BatchNormalization()(Ws)

    out = L.Activation("relu")(Wg + Ws)
    out = L.Conv2D(num_filters, 1, padding="same")(out)
    out = L.Activation("sigmoid")(out)

    return out * s
```

After that, the highlighted feature map is concatenated with the skip connection and then followed by a conv_block.

```
def decoder_block(x, s, num_filters):
    x = L.UpSampling2D(interpolation="bilinear")(x)
    s = attention_gate(x, s, num_filters)
    x = L.Concatenate()([x, s])
    x = conv_block(x, num_filters)
    return x
```

Compiling the Attention Unet

```
def attention_unet(input_shape):
    """ Inputs """
    inputs = L.Input(input_shape)

    """ Encoder """
    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 256)

    b1 = conv_block(p3, 512)

    """ Decoder """
    d1 = decoder_block(b1, s3, 256)
    d2 = decoder_block(d1, s2, 128)
    d3 = decoder_block(d2, s1, 64)

    """ Outputs """
    outputs = L.Conv2D(1, 1, padding="same", activation="sigmoid")(d3)

    """ Model """
    model = Model(inputs, outputs, name="Attention-UNET")
    return
```

Implementation

Using the keras-unet-collection library, I define the Attention Unet model with the same set of increasing filters, the same number of downsampling and upsampling stacks, and the same backbone. In this model, the attention mechanism is specified by the argument attention='add', which indicates that the attention mechanism is used to add weights to specific regions of the feature map. The gates are activated using the atten_activation='ReLU', providing non-linearity.

In [ ]:
```
model_attention_Unet = models.att_unet_2d(input_shape, filter_num=[64, 128, 256, 512, 1024],
                          n_labels=num_labels,
                          stack_num_down=2, stack_num_up=2,
                          activation='ReLU',
                          atten_activation='ReLU', attention='add',
                          output_activation='Softmax',
                          batch_norm=True, pool=False, unpool=False,
                          backbone='ResNet50', weights='imagenet',
                          freeze_backbone=True, freeze_batch_norm=True,
                          name='attention_unet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

In [ ]:
```
print(model_attention_Unet.summary())
```

```
Model: "attention_unet_model"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 128, 128, 3)	0	–
ResNet50_backbone (Functional)	[(None, 64, 64, 64), (None, 32, 32, 256), (None, 16, 16, 512), (None, 8, 8, 1024)]	8,589,184	input_layer_1[0][0]
attention_unet_up0_decod… (Conv2DTranspose)	(None, 16, 16, 512)	4,719,104	ResNet50_backbone[0][…
attention_unet_up0_decod… (BatchNormalization)	(None, 16, 16, 512)	2,048	attention_unet_up0_de…
attention_unet_up0_decod… (ReLU)	(None, 16, 16, 512)	0	attention_unet_up0_de…
attention_unet_up0_att_t… (Conv2D)	(None, 16, 16, 256)	131,328	ResNet50_backbone[0][…
attention_unet_up0_att_p… (Conv2D)	(None, 16, 16, 256)	131,328	attention_unet_up0_de…
attention_unet_up0_att_a… (Add)	(None, 16, 16, 256)	0	attention_unet_up0_at… attention_unet_up0_at…
attention_unet_up0_att_a… (ReLU)	(None, 16, 16, 256)	0	attention_unet_up0_at…
attention_unet_up0_att_p… (Conv2D)	(None, 16, 16, 1)	257	attention_unet_up0_at…
attention_unet_up0_att_s… (Activation)	(None, 16, 16, 1)	0	attention_unet_up0_at…
attention_unet_up0_att_m… (Multiply)	(None, 16, 16, 512)	0	ResNet50_backbone[0][… attention_unet_up0_at…
attention_unet_up0_concat (Concatenate)	(None, 16, 16, 1024)	0	attention_unet_up0_de… attention_unet_up0_at…
attention_unet_up0_conv_… (Conv2D)	(None, 16, 16, 512)	4,718,592	attention_unet_up0_co…
attention_unet_up0_conv_… (BatchNormalization)	(None, 16, 16, 512)	2,048	attention_unet_up0_co…
attention_unet_up0_conv_… (ReLU)	(None, 16, 16, 512)	0	attention_unet_up0_co…
attention_unet_up0_conv_… (Conv2D)	(None, 16, 16, 512)	2,359,296	attention_unet_up0_co…
attention_unet_up0_conv_… (BatchNormalization)	(None, 16, 16, 512)	2,048	attention_unet_up0_co…
attention_unet_up0_conv_… (ReLU)	(None, 16, 16, 512)	0	attention_unet_up0_co…
attention_unet_up1_decod… (Conv2DTranspose)	(None, 32, 32, 256)	1,179,904	attention_unet_up0_co…
attention_unet_up1_decod… (BatchNormalization)	(None, 32, 32, 256)	1,024	attention_unet_up1_de…
attention_unet_up1_decod… (ReLU)	(None, 32, 32, 256)	0	attention_unet_up1_de…
attention_unet_up1_att_t… (Conv2D)	(None, 32, 32, 128)	32,896	ResNet50_backbone[0][…
attention_unet_up1_att_p… (Conv2D)	(None, 32, 32, 128)	32,896	attention_unet_up1_de…
attention_unet_up1_att_a… (Add)	(None, 32, 32, 128)	0	attention_unet_up1_at… attention_unet_up1_at…
attention_unet_up1_att_a… (ReLU)	(None, 32, 32, 128)	0	attention_unet_up1_at…
attention_unet_up1_att_p… (Conv2D)	(None, 32, 32, 1)	129	attention_unet_up1_at…
attention_unet_up1_att_s… (Activation)	(None, 32, 32, 1)	0	attention_unet_up1_at…
attention_unet_up1_att_m… (Multiply)	(None, 32, 32, 256)	0	ResNet50_backbone[0][… attention_unet_up1_at…
attention_unet_up1_concat	(None, 32, 32, 512)	0	attention_unet_up1_de…

(Concatenate)			attention_unet_up1_at…
attention_unet_up1_conv_… (Conv2D)	(None, 32, 32, 256)	1,179,648	attention_unet_up1_co…
attention_unet_up1_conv_… (BatchNormalization)	(None, 32, 32, 256)	1,024	attention_unet_up1_co…
attention_unet_up1_conv_… (ReLU)	(None, 32, 32, 256)	0	attention_unet_up1_co…
attention_unet_up1_conv_… (Conv2D)	(None, 32, 32, 256)	589,824	attention_unet_up1_co…
attention_unet_up1_conv_… (BatchNormalization)	(None, 32, 32, 256)	1,024	attention_unet_up1_co…
attention_unet_up1_conv_… (ReLU)	(None, 32, 32, 256)	0	attention_unet_up1_co…
attention_unet_up2_decod… (Conv2DTranspose)	(None, 64, 64, 128)	295,040	attention_unet_up1_co…
attention_unet_up2_decod… (BatchNormalization)	(None, 64, 64, 128)	512	attention_unet_up2_de…
attention_unet_up2_decod… (ReLU)	(None, 64, 64, 128)	0	attention_unet_up2_de…
attention_unet_up2_att_t… (Conv2D)	(None, 64, 64, 64)	4,160	ResNet50_backbone[0][…
attention_unet_up2_att_p… (Conv2D)	(None, 64, 64, 64)	8,256	attention_unet_up2_de…
attention_unet_up2_att_a… (Add)	(None, 64, 64, 64)	0	attention_unet_up2_at… attention_unet_up2_at…
attention_unet_up2_att_a… (ReLU)	(None, 64, 64, 64)	0	attention_unet_up2_at…
attention_unet_up2_att_p… (Conv2D)	(None, 64, 64, 1)	65	attention_unet_up2_at…
attention_unet_up2_att_s… (Activation)	(None, 64, 64, 1)	0	attention_unet_up2_at…
attention_unet_up2_att_m… (Multiply)	(None, 64, 64, 64)	0	ResNet50_backbone[0][… attention_unet_up2_at…
attention_unet_up2_concat (Concatenate)	(None, 64, 64, 192)	0	attention_unet_up2_de… attention_unet_up2_at…
attention_unet_up2_conv_… (Conv2D)	(None, 64, 64, 128)	221,184	attention_unet_up2_co…
attention_unet_up2_conv_… (BatchNormalization)	(None, 64, 64, 128)	512	attention_unet_up2_co…
attention_unet_up2_conv_… (ReLU)	(None, 64, 64, 128)	0	attention_unet_up2_co…
attention_unet_up2_conv_… (Conv2D)	(None, 64, 64, 128)	147,456	attention_unet_up2_co…
attention_unet_up2_conv_… (BatchNormalization)	(None, 64, 64, 128)	512	attention_unet_up2_co…
attention_unet_up2_conv_… (ReLU)	(None, 64, 64, 128)	0	attention_unet_up2_co…
attention_unet_up3_decod… (Conv2DTranspose)	(None, 128, 128, 64)	73,792	attention_unet_up2_co…
attention_unet_up3_decod… (BatchNormalization)	(None, 128, 128, 64)	256	attention_unet_up3_de…
attention_unet_up3_decod… (ReLU)	(None, 128, 128, 64)	0	attention_unet_up3_de…
attention_unet_up3_conv_… (Conv2D)	(None, 128, 128, 64)	36,864	attention_unet_up3_de…
attention_unet_up3_conv_… (BatchNormalization)	(None, 128, 128, 64)	256	attention_unet_up3_co…
attention_unet_up3_conv_… (ReLU)	(None, 128, 128, 64)	0	attention_unet_up3_co…
attention_unet_up3_conv_… (Conv2D)	(None, 128, 128, 64)	36,864	attention_unet_up3_co…
attention_unet_up3_conv_…	(None, 128, 128, 64)	256	attention_unet_up3_co…

(BatchNormalization)			
attention_unet_up3_conv_… (ReLU)	(None, 128, 128, 64)	0	attention_unet_up3_co…
attention_unet_up3_conv_… (Conv2D)	(None, 128, 128, 64)	36,864	attention_unet_up3_co…
attention_unet_up3_conv_… (BatchNormalization)	(None, 128, 128, 64)	256	attention_unet_up3_co…
attention_unet_up3_conv_… (ReLU)	(None, 128, 128, 64)	0	attention_unet_up3_co…
attention_unet_output (Conv2D)	(None, 128, 128, 2)	130	attention_unet_up3_co…
attention_unet_output_ac… (Softmax)	(None, 128, 128, 2)	0	attention_unet_output…

Total params: 24,536,837 (93.60 MB)

Trainable params: 15,941,765 (60.81 MB)

Non-trainable params: 8,595,072 (32.79 MB)

None

Following the same configuration I define above:

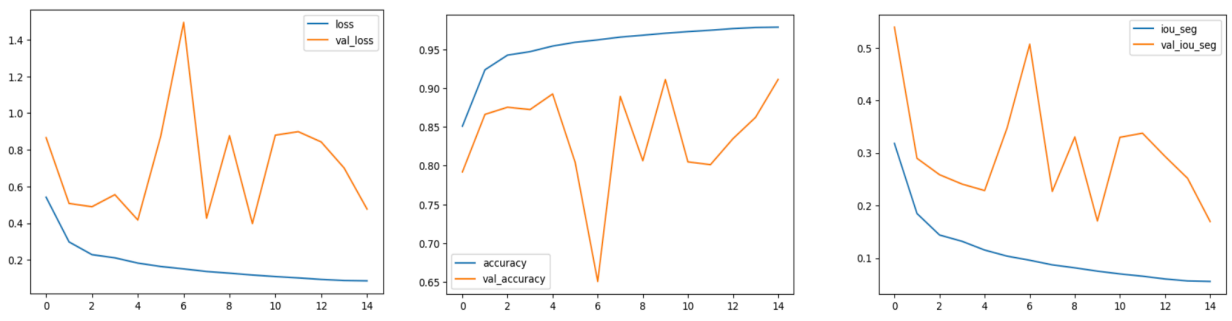
- Combine loss function: Dice loss + categorical cross-entropy
- Adam optimizer with cosine decay restarts learning rate scheduer
- Finetune the Resnet50 backbone from the 100th layers
- Add L2 regularizer

In [ ]:
```
# Callbacks
es = EarlyStopping(monitor='val_loss',
                   patience=8,
                   restore_best_weights=True,
                   verbose=1)
# Training the model
start_time = datetime.now()
initial_epochs = len(Attention_Unet_history.epoch)
Attention_Unet_history = model_attention_Unet.fit(
    train_dataset,
    validation_data=validation_dataset,
    verbose=1,
    batch_size = 32,
    initial_epoch = initial_epochs,
    epochs= initial_epochs+20,
    callbacks=[es, model_checkpoint],
    shuffle=True
)

end_time = datetime.now()
print(f"Training time: {end_time - start_time}")
```

```
Epoch 16/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 53s 349ms/step - accuracy: 0.9737 - iou_seg: 0.0681 - loss: 0.1055 - val_accuracy: 0.
8570 - val_iou_seg: 0.2792 - val_loss: 0.7027
Epoch 17/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 53s 351ms/step - accuracy: 0.9739 - iou_seg: 0.0670 - loss: 0.1045 - val_accuracy: 0.
7929 - val_iou_seg: 0.3440 - val_loss: 1.0615
Epoch 18/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 346ms/step - accuracy: 0.9766 - iou_seg: 0.0610 - loss: 0.0940
/usr/local/lib/python3.10/dist-packages/keras/src/saving/serialization_lib.py:390: UserWarning: The object being s
erialized includes a `lambda`. This is unsafe. In order to reload the object, you will have to pass `safe_mode=Fal
se` to the loading function. Please avoid using `lambda` in the future, and use named Python functions instead. Th
is is the `lambda` being serialized:                    loss=lambda y_true, y_pred: dice_loss(y_true, y_pred) + t
f.keras.losses.categorical_crossentropy(y_true, y_pred),

  return {key: serialize_keras_object(value) for key, value in obj.items()}
```

```
152/152 ━━━━━━━━━━━━━━━━━━━━ 56s 371ms/step - accuracy: 0.9766 - iou_seg: 0.0610 - loss: 0.0940 - val_accuracy: 0.
9189 - val_iou_seg: 0.1558 - val_loss: 0.3636
Epoch 19/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 80s 357ms/step - accuracy: 0.9786 - iou_seg: 0.0560 - loss: 0.0862 - val_accuracy: 0.
8606 - val_iou_seg: 0.2687 - val_loss: 0.6102
Epoch 20/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 55s 365ms/step - accuracy: 0.9797 - iou_seg: 0.0531 - loss: 0.0817 - val_accuracy: 0.
9199 - val_iou_seg: 0.1579 - val_loss: 0.3641
Epoch 21/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 81s 357ms/step - accuracy: 0.9810 - iou_seg: 0.0499 - loss: 0.0764 - val_accuracy: 0.
8716 - val_iou_seg: 0.2353 - val_loss: 0.6667
Epoch 22/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 55s 361ms/step - accuracy: 0.9818 - iou_seg: 0.0479 - loss: 0.0732 - val_accuracy: 0.
8915 - val_iou_seg: 0.2056 - val_loss: 0.5817
Epoch 23/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 55s 359ms/step - accuracy: 0.9828 - iou_seg: 0.0453 - loss: 0.0693 - val_accuracy: 0.
8723 - val_iou_seg: 0.2313 - val_loss: 0.6878
Epoch 24/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 359ms/step - accuracy: 0.9837 - iou_seg: 0.0432 - loss: 0.0659 - val_accuracy: 0.
8937 - val_iou_seg: 0.1974 - val_loss: 0.5890
Epoch 25/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 55s 360ms/step - accuracy: 0.9843 - iou_seg: 0.0417 - loss: 0.0636 - val_accuracy: 0.
9109 - val_iou_seg: 0.1661 - val_loss: 0.5155
Epoch 26/35
152/152 ━━━━━━━━━━━━━━━━━━━━ 82s 357ms/step - accuracy: 0.9849 - iou_seg: 0.0401 - loss: 0.0611 - val_accuracy: 0.
9075 - val_iou_seg: 0.1738 - val_loss: 0.5273
Epoch 26: early stopping
Restoring model weights from the end of the best epoch: 18.
Training time: 0:11:46.907003
```

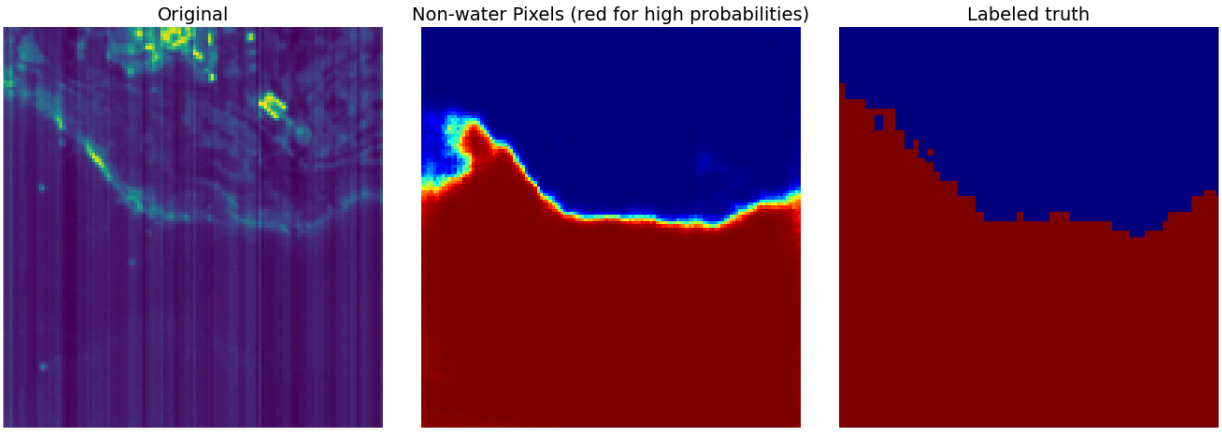
In [ ]:
```
Attention_Unet_history_df = pd.DataFrame(Attention_Unet_history.history)
Attention_Unet_history_df[['loss', 'val_loss']].plot()
Attention_Unet_history_df[['accuracy', 'val_accuracy']].plot()
Attention_Unet_history_df[['iou_seg', 'val_iou_seg']].plot()
```



The training process also meets difficulty to converge (I run the model 3 times but it does not converge). The best validation loss is recorded at 0.3636 which is slightly higher than that of UNET model

In [ ]:
```
evaluate_performance (y_test, y_pred)
```

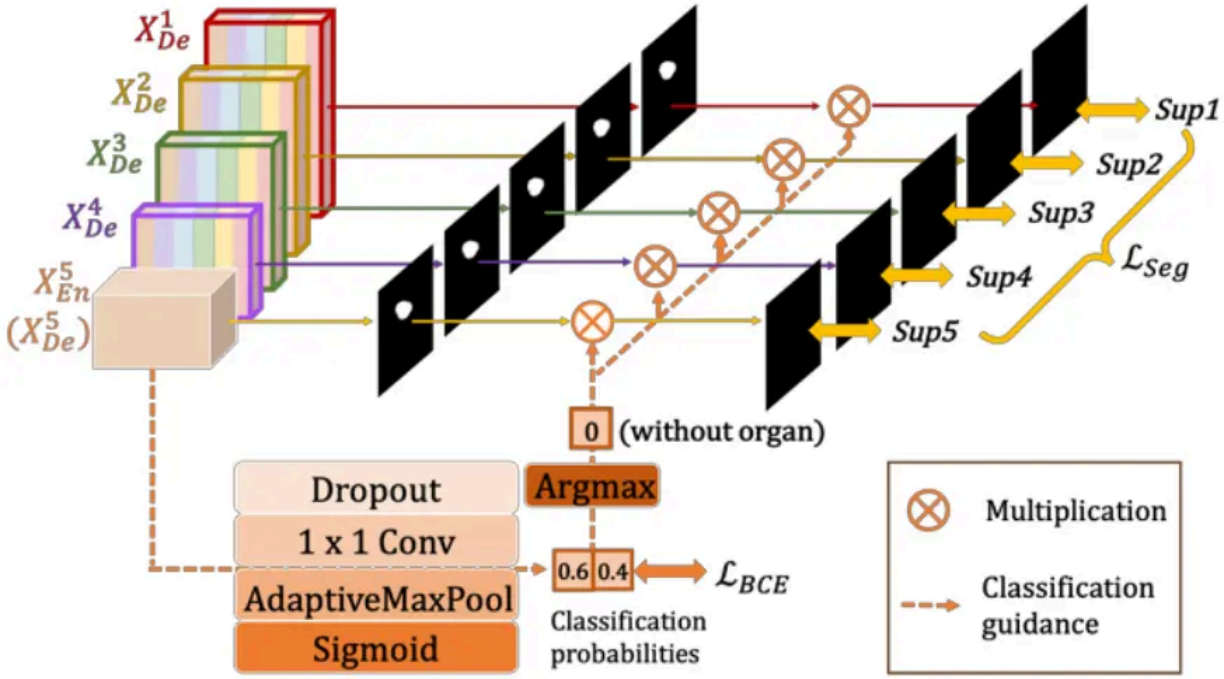
```
Accuracy: 0.9659
Mean IoU: 0.9339
Dice Coefficient: 0.9658
Cross-Entropy Loss: 0.1035
```



Looking at the mask prediction, the reconstruction quality is high with most of the water region are predicted with confidence (high probability) while most of the non-water region are not mispredicted (not even assigned low probability of being water pixels). This showcases the effects of attention mechanism of Attenion UNET since the model seems to have higher spatial awareness than the UNET.

The mask prediction shows that Attention UNET **knows exactly where to look for** when asking for water pixels and non-water pixels.

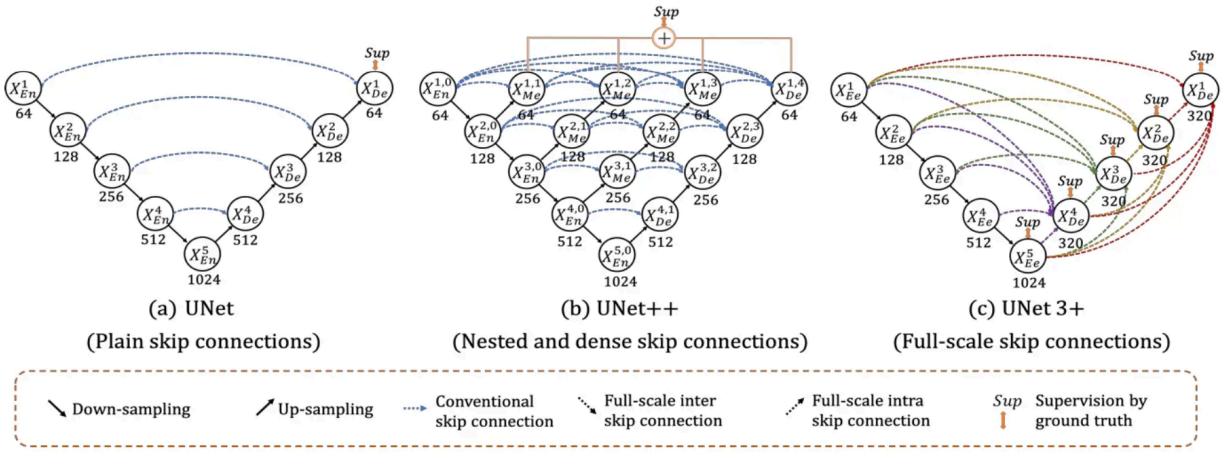
UNET3+



UNet 3+ is an advanced variation of the classic U-Net model, designed to improve segmentation performance by introducing full-scale skip connections, deep supervision, and a Classification-Guided Module (CGM).

Key Features of UNet 3+ and Differences from U-Net and Attention U-Net:

1. **Full-Scale Skip Connections**:
   - **Classic U-Net**: In U-Net, skip connections are used to transfer features from the encoder part (downsampling) to the decoder part (upsampling) of the network. These connections help preserve spatial information for finer segmentation.
   - **UNet 3+**: In contrast, UNet 3+ enhances this idea with **full-scale skip connections**. It aggregates feature maps from multiple levels (across different stages of downsampling and upsampling) instead of just from corresponding levels of the encoder and decoder. This means each upsampling block receives input from not only the current encoder level but also from previous and next stages, enriching the feature maps with information from different resolutions.



1. **Deep Supervision**:

   - **Classic U-Net**: The original U-Net model only produces a final output after passing through the entire network, relying on the last layer's feature representation to make the final segmentation decision.
   - **UNet 3+**: UNet 3+ incorporates **deep supervision**, which involves using intermediate feature maps during the upsampling process to guide learning. This means the model is supervised not only at the final output but also at multiple intermediate points, improving the ability to capture fine-grained details in the segmentation task. This method leads to faster convergence and better performance, especially when dealing with complex segmentation tasks.
2. **Classification-Guided Module (CGM)**:

   - **Classic U-Net and Attention U-Net**: These models focus primarily on feature extraction, segmentation, and localization of relevant regions in an image. Attention U-Net enhances this with attention gates to focus on relevant regions of the feature

maps.

- **UNet 3+**: UNet 3+ introduces the **Classification-Guided Module (CGM)**, which is designed to improve the model's focus on discriminative features for better classification. The CGM helps the model pay attention to areas that are more important for segmentation by leveraging the classification output during training.

In UNet 3+, the CGM operates by adding a supervised classification branch to the network. This means that alongside the segmentation loss (typically computed with cross-entropy or Dice loss), there is an additional loss that measures how well the model can classify the image into its respective class.The classification loss is used to guide the feature maps within the encoder and decoder blocks of the network.

The idea is that the features needed for accurate classification (which focus on identifying the most relevant regions or objects in an image) can also help improve segmentation. Basically, CGM takes the output tensor of the deepest downsampling level, and converts it into a single value that indicates the probability of the existence of the segmentation target.

Implementation

In this architecture, five downsampling blocks are used with increasing filter numbers (from 32 to 512). Each downsampling block consists of two convolutional layers. The upsampling block, however, aggregates feature maps from multiple levels of the network. Each tensor received by the upsampling block is processed with 32 filters, concatenated, and passed through another convolutional layer with 160 filters. This design improves the model's ability to capture multi-scale information and create a more robust segmentation mask.

In [ ]:
```
name = 'unet3plus'
activation = 'ReLU'
filter_num_down = [32, 64, 128, 256, 512]
filter_num_skip = [32, 32, 32, 32]
filter_num_aggregate = 160

stack_num_down = 2
stack_num_up = 1
n_labels = 2

# `unet_3plus_2d_base` accepts an input tensor
# and produces output tensors from different upsampling levels
# ---------------------------------------- #
input_tensor = tf.keras.layers.Input((128, 128, 3))
# base architecture
X_decoder = base.unet_3plus_2d_base(
    input_tensor, filter_num_down, filter_num_skip, filter_num_aggregate,
    stack_num_down=stack_num_down, stack_num_up=stack_num_up, activation=activation,
    batch_norm=True,pool=True, unpool=True,
    backbone='ResNet50', weights='imagenet',
    freeze_backbone=True, freeze_batch_norm=True,
    name=name)
```

In [ ]:
```
X_decoder
```

Out[ ]:
```
[<KerasTensor shape=(None, 8, 8, 1024), dtype=float16, sparse=False, name=keras_tensor_179>,
 <KerasTensor shape=(None, 16, 16, 160), dtype=float16, sparse=False, name=keras_tensor_194>,
 <KerasTensor shape=(None, 32, 32, 160), dtype=float16, sparse=False, name=keras_tensor_209>,
 <KerasTensor shape=(None, 64, 64, 160), dtype=float16, sparse=False, name=keras_tensor_225>,
 <KerasTensor shape=(None, 128, 128, 160), dtype=float16, sparse=False, name=keras_tensor_232>]
```

Deep Supervision:

During the upsampling phase, the model receives intermediate outputs (from various levels of the decoder) and applies convolution layers to generate predictions. Each upsampling level involves:

- A convolution layer (Conv2D) with 3x3 kernels to process features.
- Upsampling using bilinear interpolation.
- Activation via sigmoid to output pixel-wise segmentation probabilities.

These intermediate outputs are stored in OUT_stack for deep supervision.

In [ ]:
```
# allocating deep supervision tensors
OUT_stack = []
# reverse indexing `X_decoder`, so smaller tensors have larger list indices
X_decoder = X_decoder[::-1]

# deep supervision outputs
for i in range(1, len(X_decoder)):
    # 3-by-3 conv2d --> upsampling --> softmax output activation
    pool_size = 2**(i)
    X = Conv2D(n_labels, 3, padding='same', name='{}_output_conv1_{}'.format(name, i-1))(X_decoder[i])

    X = UpSampling2D((pool_size, pool_size), interpolation='bilinear',
                     name='{}_output_sup{}'.format(name, i-1))(X)

    X = Activation('sigmoid', name='{}_output_sup{}_activation'.format(name, i-1))(X)
    # collecting deep supervision tensors
    OUT_stack.append(X)
```

```
# the final output (without extra upsampling)
# 3-by-3 conv2d --> sigmoid output activation
X = Conv2D(n_labels, 3, padding='same', name='{}_output_final'.format(name))(X_decoder[0])
X = Activation('sigmoid', name='{}_output_final_activation'.format(name))(X)
# collecting final output tensors
OUT_stack.append(X)
```

The deep-supervision and final output tensors are shown as follows:

In [ ]:
```
OUT_stack
```

Out[ ]:
```
[<KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_235>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_238>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_241>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_244>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_246>]
```

The steps of the CGM are:

- A Dropout layer is applied for regularization (drop out rate = 0.3).
- A 1x1 convolution reduces the feature map's depth to match the number of filters in filter_num_skip.
- Global Max Pooling reduces the spatial dimensions, summarizing the most important features.
- Sigmoid activation generates a scalar mask that highlights important regions of the image.
- The output from the CGM is a mask that is used to modify the segmentation outputs by scaling them with this global mask, using the multiply operation.

In [ ]:
```
# Classification-guided Module (CGM)
# ---------------------------------------- #
# dropout --> 1-by-1 conv2d --> global-maxpooling --> sigmoid
X_CGM = X_decoder[-1]
X_CGM = Dropout(rate=0.3)(X_CGM)
X_CGM = Conv2D(filter_num_skip[-1], 1, padding='same')(X_CGM)
X_CGM = GlobalMaxPooling2D()(X_CGM)
X_CGM = Activation('sigmoid')(X_CGM)

# Wrap tf.reduce_max in a Lambda layer
CGM_mask = Lambda(lambda x: tf.reduce_max(x, axis=-1))(X_CGM)

for i in range(len(OUT_stack)):
    if i < len(OUT_stack)-1:
        # deep-supervision
        OUT_stack[i] = multiply([OUT_stack[i], CGM_mask], name='{}_output_sup{}_CGM'.format(name, i))
    else:
        # final output
        OUT_stack[i] = multiply([OUT_stack[i], CGM_mask], name='{}_output_final_CGM'.format(name))
```

Applying CGM to Deep Supervision Outputs:

- The deep supervision outputs (OUT_stack) are multiplied by the CGM mask to give a classification-guided focus to the segmentations.
- For each deep supervision output (except the final one), the output is adjusted using the CGM mask. This helps the model focus more on regions that are classified as relevant by the CGM.
- For the final output, the CGM mask is applied similarly.

Final Output (With CGM Applied): The final outputs in OUT_stack are adjusted using the CGM mask, which helps in improving the accuracy and focus of the segmentation in areas relevant to the classification.

In [ ]:
```
OUT_stack
```

Out[ ]:
```
[<KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_254>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_255>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_256>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_257>,
 <KerasTensor shape=(None, 128, 128, 2), dtype=float16, sparse=False, name=keras_tensor_258>]
```

Using the same configuration with regularizer, learning rate scheduler above, I run the model. The segmentation model is trained with cross-entropy loss. Each deep-supervision branch contributes 12.5% of the total loss value, whereas the final output loss contribution is 50%. This ensures that the final segmentation output has a stronger influence on the model's optimization than the intermediate outputs.

In [ ]:
```
# executing all the above cells in one time to avoid dupilcated tensor names.
unet3plus = tf.keras.models.Model([input_tensor,], OUT_stack)

# Compile the model
unet3plus.compile(loss=[CombinedLoss(), CombinedLoss(), CombinedLoss(), CombinedLoss(), CombinedLoss()],
                  loss_weights=[0.25, 0.25, 0.25, 0.25, 1.0],
                  optimizer=optimizer)

# Assign the compiled model back to model_unet3plus for training
model_unet3plus = unet3plus
```

In [ ]:
```
model_unet3plus.summary()
```

Model: "functional_5"

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 128, 128, 3)	0	–
ResNet50_backbone (Functional)	[(None, 64, 64, 64), (None, 32, 32, 256), (None, 16, 16, 512), (None, 8, 8, 1024)]	8,589,184	input_layer_2[0][0]
unet3plus_up_0_en0_unpool (UpSampling2D)	(None, 16, 16, 1024)	0	ResNet50_backbone[0][…
unet3plus_down_0_en2_max… (MaxPooling2D)	(None, 16, 16, 256)	0	ResNet50_backbone[0][…
unet3plus_down_from0_to0… (Conv2D)	(None, 16, 16, 32)	294,912	unet3plus_up_0_en0_un…
unet3plus_down_from0_to1… (Conv2D)	(None, 16, 16, 32)	147,456	ResNet50_backbone[0][…
unet3plus_down_from0_to2… (Conv2D)	(None, 16, 16, 32)	73,728	unet3plus_down_0_en2_…
unet3plus_down_from0_to0… (BatchNormalization)	(None, 16, 16, 32)	128	unet3plus_down_from0_…
unet3plus_down_from0_to1… (BatchNormalization)	(None, 16, 16, 32)	128	unet3plus_down_from0_…
unet3plus_down_from0_to2… (BatchNormalization)	(None, 16, 16, 32)	128	unet3plus_down_from0_…
unet3plus_down_from0_to0… (ReLU)	(None, 16, 16, 32)	0	unet3plus_down_from0_…
unet3plus_down_from0_to1… (ReLU)	(None, 16, 16, 32)	0	unet3plus_down_from0_…
unet3plus_down_from0_to2… (ReLU)	(None, 16, 16, 32)	0	unet3plus_down_from0_…
unet3plus_concat_0 (Concatenate)	(None, 16, 16, 96)	0	unet3plus_down_from0_… unet3plus_down_from0_… unet3plus_down_from0_…
unet3plus_fusion_conv_0_0 (Conv2D)	(None, 16, 16, 160)	138,240	unet3plus_concat_0[0]…
unet3plus_fusion_conv_0_… (BatchNormalization)	(None, 16, 16, 160)	640	unet3plus_fusion_conv…
unet3plus_fusion_conv_0_… (ReLU)	(None, 16, 16, 160)	0	unet3plus_fusion_conv…
unet3plus_up_1_en0_unpool (UpSampling2D)	(None, 32, 32, 1024)	0	ResNet50_backbone[0][…
unet3plus_up_1_en1_unpool (UpSampling2D)	(None, 32, 32, 160)	0	unet3plus_fusion_conv…
unet3plus_down_from1_to0… (Conv2D)	(None, 32, 32, 32)	294,912	unet3plus_up_1_en0_un…
unet3plus_down_from1_to1… (Conv2D)	(None, 32, 32, 32)	46,080	unet3plus_up_1_en1_un…
unet3plus_down_from1_to2… (Conv2D)	(None, 32, 32, 32)	73,728	ResNet50_backbone[0][…
unet3plus_down_from1_to0… (BatchNormalization)	(None, 32, 32, 32)	128	unet3plus_down_from1_…
unet3plus_down_from1_to1… (BatchNormalization)	(None, 32, 32, 32)	128	unet3plus_down_from1_…
unet3plus_down_from1_to2… (BatchNormalization)	(None, 32, 32, 32)	128	unet3plus_down_from1_…
unet3plus_down_from1_to0… (ReLU)	(None, 32, 32, 32)	0	unet3plus_down_from1_…
unet3plus_down_from1_to1… (ReLU)	(None, 32, 32, 32)	0	unet3plus_down_from1_…
unet3plus_down_from1_to2… (ReLU)	(None, 32, 32, 32)	0	unet3plus_down_from1_…
unet3plus_concat_1 (Concatenate)	(None, 32, 32, 96)	0	unet3plus_down_from1_… unet3plus_down_from1_… unet3plus_down_from1_…

unet3plus_fusion_conv_1_0 (Conv2D)	(None, 32, 32, 160)	138,240	unet3plus_concat_1[0]…
unet3plus_fusion_conv_1_… (BatchNormalization)	(None, 32, 32, 160)	640	unet3plus_fusion_conv…
unet3plus_fusion_conv_1_… (ReLU)	(None, 32, 32, 160)	0	unet3plus_fusion_conv…
unet3plus_up_2_en0_unpool (UpSampling2D)	(None, 64, 64, 1024)	0	ResNet50_backbone[0][…
unet3plus_up_2_en1_unpool (UpSampling2D)	(None, 64, 64, 160)	0	unet3plus_fusion_conv…
unet3plus_up_2_en2_unpool (UpSampling2D)	(None, 64, 64, 160)	0	unet3plus_fusion_conv…
unet3plus_down_from2_to0… (Conv2D)	(None, 64, 64, 32)	294,912	unet3plus_up_2_en0_un…
unet3plus_down_from2_to1… (Conv2D)	(None, 64, 64, 32)	46,080	unet3plus_up_2_en1_un…
unet3plus_down_from2_to2… (Conv2D)	(None, 64, 64, 32)	46,080	unet3plus_up_2_en2_un…
unet3plus_down_from2_to0… (BatchNormalization)	(None, 64, 64, 32)	128	unet3plus_down_from2_…
unet3plus_down_from2_to1… (BatchNormalization)	(None, 64, 64, 32)	128	unet3plus_down_from2_…
unet3plus_down_from2_to2… (BatchNormalization)	(None, 64, 64, 32)	128	unet3plus_down_from2_…
unet3plus_down_from2_to0… (ReLU)	(None, 64, 64, 32)	0	unet3plus_down_from2_…
unet3plus_down_from2_to1… (ReLU)	(None, 64, 64, 32)	0	unet3plus_down_from2_…
unet3plus_down_from2_to2… (ReLU)	(None, 64, 64, 32)	0	unet3plus_down_from2_…
unet3plus_concat_2 (Concatenate)	(None, 64, 64, 96)	0	unet3plus_down_from2_… unet3plus_down_from2_… unet3plus_down_from2_…
unet3plus_fusion_conv_2_0 (Conv2D)	(None, 64, 64, 160)	138,240	unet3plus_concat_2[0]…
unet3plus_fusion_conv_2_… (BatchNormalization)	(None, 64, 64, 160)	640	unet3plus_fusion_conv…
unet3plus_fusion_conv_2_… (ReLU)	(None, 64, 64, 160)	0	unet3plus_fusion_conv…
unet3plus_plain_up3_deco… (UpSampling2D)	(None, 128, 128, 160)	0	unet3plus_fusion_conv…
unet3plus_plain_up3_conv… (Conv2D)	(None, 128, 128, 160)	230,400	unet3plus_plain_up3_d…
unet3plus_plain_up3_conv… (BatchNormalization)	(None, 128, 128, 160)	640	unet3plus_plain_up3_c…
unet3plus_plain_up3_conv… (ReLU)	(None, 128, 128, 160)	0	unet3plus_plain_up3_c…
dropout_2 (Dropout)	(None, 8, 8, 1024)	0	ResNet50_backbone[0][…
unet3plus_plain_up3_conv… (Conv2D)	(None, 128, 128, 160)	230,400	unet3plus_plain_up3_c…
conv2d_2 (Conv2D)	(None, 8, 8, 32)	32,800	dropout_2[0][0]
unet3plus_plain_up3_conv… (BatchNormalization)	(None, 128, 128, 160)	640	unet3plus_plain_up3_c…
unet3plus_output_conv1_0 (Conv2D)	(None, 64, 64, 2)	2,882	unet3plus_fusion_conv…
global_max_pooling2d_2 (GlobalMaxPooling2D)	(None, 32)	0	conv2d_2[0][0]
unet3plus_output_conv1_1 (Conv2D)	(None, 32, 32, 2)	2,882	unet3plus_fusion_conv…
unet3plus_output_conv1_2 (Conv2D)	(None, 16, 16, 2)	2,882	unet3plus_fusion_conv…

unet3plus_output_conv1_3 (Conv2D)	(None, 8, 8, 2)	18,434	ResNet50_backbone[0][…
unet3plus_plain_up3_conv… (ReLU)	(None, 128, 128, 160)	0	unet3plus_plain_up3_c…
unet3plus_output_sup0 (UpSampling2D)	(None, 128, 128, 2)	0	unet3plus_output_conv…
activation_2 (Activation)	(None, 32)	0	global_max_pooling2d_…
unet3plus_output_sup1 (UpSampling2D)	(None, 128, 128, 2)	0	unet3plus_output_conv…
unet3plus_output_sup2 (UpSampling2D)	(None, 128, 128, 2)	0	unet3plus_output_conv…
unet3plus_output_sup3 (UpSampling2D)	(None, 128, 128, 2)	0	unet3plus_output_conv…
unet3plus_output_final (Conv2D)	(None, 128, 128, 2)	2,882	unet3plus_plain_up3_c…
unet3plus_output_sup0_ac… (Activation)	(None, 128, 128, 2)	0	unet3plus_output_sup0…
lambda_1 (Lambda)	(None)	0	activation_2[0][0]
unet3plus_output_sup1_ac… (Activation)	(None, 128, 128, 2)	0	unet3plus_output_sup1…
unet3plus_output_sup2_ac… (Activation)	(None, 128, 128, 2)	0	unet3plus_output_sup2…
unet3plus_output_sup3_ac… (Activation)	(None, 128, 128, 2)	0	unet3plus_output_sup3…
unet3plus_output_final_a… (Activation)	(None, 128, 128, 2)	0	unet3plus_output_fina…
unet3plus_output_sup0_CGM (Multiply)	(None, 128, 128, 2)	0	unet3plus_output_sup0… lambda_1[0][0]
unet3plus_output_sup1_CGM (Multiply)	(None, 128, 128, 2)	0	unet3plus_output_sup1… lambda_1[0][0]
unet3plus_output_sup2_CGM (Multiply)	(None, 128, 128, 2)	0	unet3plus_output_sup2… lambda_1[0][0]
unet3plus_output_sup3_CGM (Multiply)	(None, 128, 128, 2)	0	unet3plus_output_sup3… lambda_1[0][0]
unet3plus_output_final_C… (Multiply)	(None, 128, 128, 2)	0	unet3plus_output_fina… lambda_1[0][0]

Total params: 10,849,706 (41.39 MB)

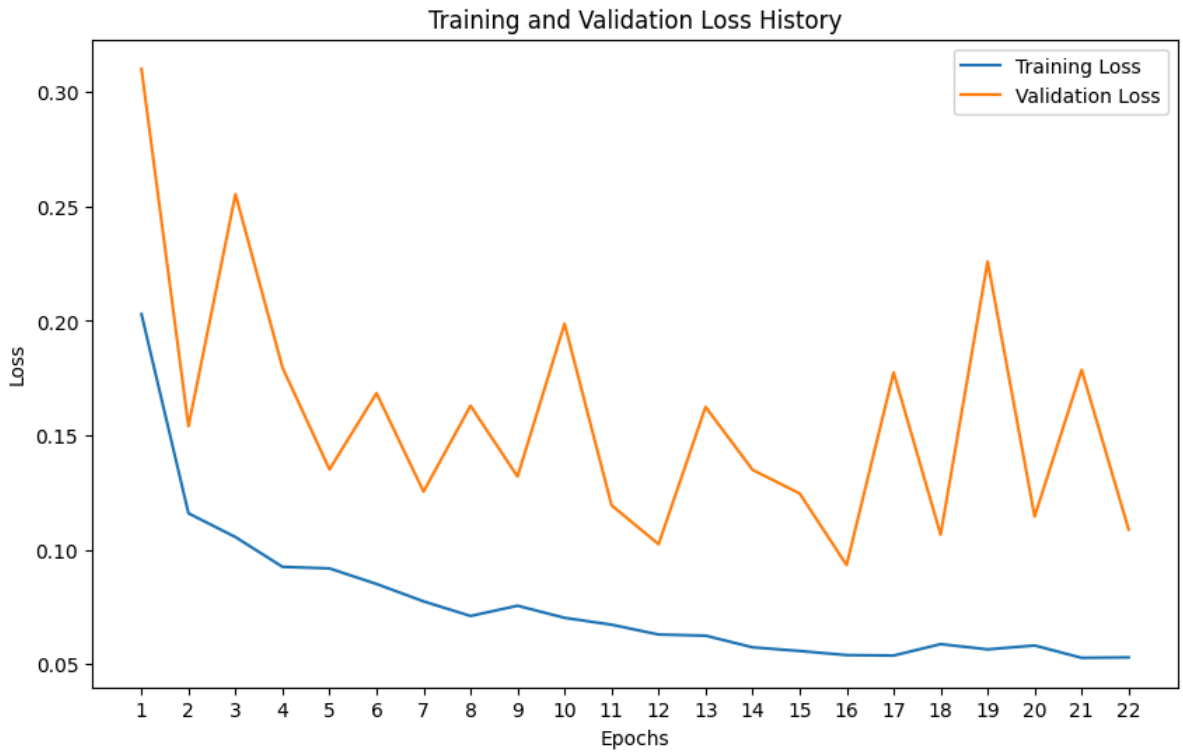
Trainable params: 2,258,346 (8.61 MB)

Non-trainable params: 8,591,360 (32.77 MB)

In [ ]:
```
# Training the model
start_time = datetime.now()
unet3plus_history = model_unet3plus.fit(
    train_dataset,
    validation_data=validation_dataset,
    verbose=1,
    batch_size = 32,
    epochs= 30,
    callbacks=[es, model_checkpoint],
    shuffle=True
)

end_time = datetime.now()
print(f"Training time: {end_time - start_time}")
```

```
Epoch 1/30
/usr/local/lib/python3.10/dist-packages/keras/src/optimizers/base_optimizer.py:678: UserWarning: Gradients do not exist for variables ['kernel', 'gamma', 'beta', 'kernel', 'gamma', 'beta', 'kernel', 'bias', 'kernel', 'bias', 'kernel', 'bias', 'kernel', 'bias'] when minimizing the loss. If using `model.compile()`, did you forget to provide a `loss` argument?
  warnings.warn(
```

```
188/188 ━━━━━━━━━━━━━━━━━━━━ 155s 382ms/step - loss: 0.2030 - val_loss: 0.3102
Epoch 2/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 111ms/step - loss: 0.1159 - val_loss: 0.1540
Epoch 3/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 106ms/step - loss: 0.1055 - val_loss: 0.2554
Epoch 4/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0925 - val_loss: 0.1796
Epoch 5/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 110ms/step - loss: 0.0918 - val_loss: 0.1351
Epoch 6/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0850 - val_loss: 0.1684
Epoch 7/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 112ms/step - loss: 0.0774 - val_loss: 0.1254
Epoch 8/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0710 - val_loss: 0.1629
Epoch 9/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 107ms/step - loss: 0.0755 - val_loss: 0.1320
Epoch 10/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0702 - val_loss: 0.1988
Epoch 11/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 114ms/step - loss: 0.0672 - val_loss: 0.1194
Epoch 12/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 110ms/step - loss: 0.0629 - val_loss: 0.1024
Epoch 13/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 40s 105ms/step - loss: 0.0624 - val_loss: 0.1624
Epoch 14/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 107ms/step - loss: 0.0573 - val_loss: 0.1349
Epoch 15/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 106ms/step - loss: 0.0557 - val_loss: 0.1246
Epoch 16/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 110ms/step - loss: 0.0539 - val_loss: 0.0933
Epoch 17/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0537 - val_loss: 0.1775
Epoch 18/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 105ms/step - loss: 0.0587 - val_loss: 0.1066
Epoch 19/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 105ms/step - loss: 0.0564 - val_loss: 0.2260
Epoch 20/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 21s 107ms/step - loss: 0.0581 - val_loss: 0.1145
Epoch 21/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0527 - val_loss: 0.1786
Epoch 22/30
188/188 ━━━━━━━━━━━━━━━━━━━━ 20s 105ms/step - loss: 0.0529 - val_loss: 0.1088
Epoch 22: early stopping
Restoring model weights from the end of the best epoch: 16.
Training time: 0:10:02.978352
```



In [ ]:
```
y_pred = model_unet3plus.predict(test_dataset)
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 17s 11s/step
```

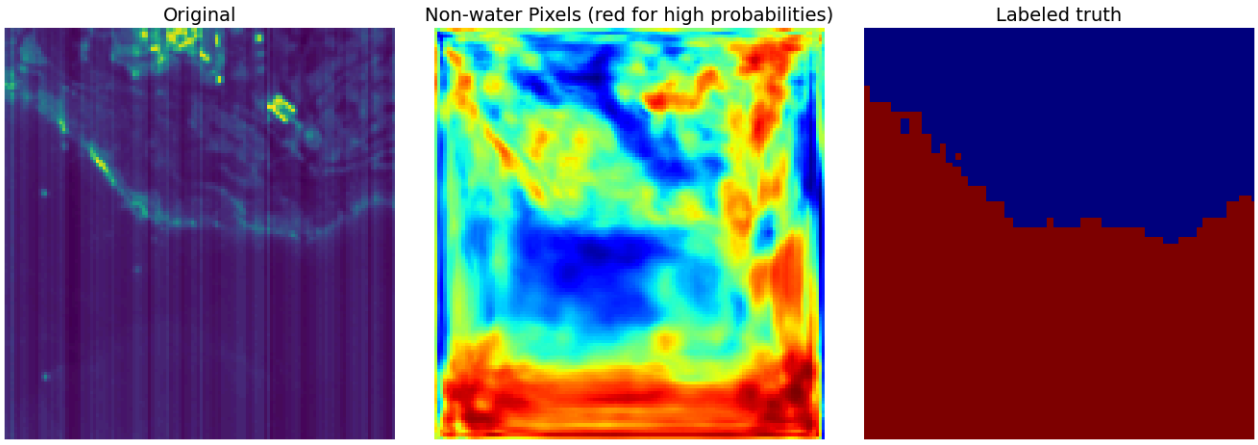
Since the OUT stack produce 5 features map for each of the upsampling leve, the final prediction only takes the highest upsampling level, thus `y_pred[-1]`

In [ ]:
```
y_pred = y_pred[-1]
```

In [ ]:
```
evaluate_performance (y_test, y_pred)
```

```
Accuracy: 0.6909
Mean IoU: 0.5204
Dice Coefficient: 0.6825
Cross-Entropy Loss: 0.5776
```

In [ ]:
```
i_sample = 12

def ax_decorate_box(ax):
    [j.set_linewidth(0) for j in ax.spines.values()]
    ax.tick_params(axis="both", which="both", bottom=False, top=False, \
                   labelbottom=False, left=False, right=False, labelleft=False)
    return ax
def visualize_mask_prediction (prediction, ground_truth):
  # Define the plot layout (3 subplots in 1 row)
  fig, AX = plt.subplots(1, 3, figsize=(13, (13-0.2)/3))
  plt.subplots_adjust(0, 0, 1, 1, hspace=0, wspace=0.1)

  # Decorate each axis for consistent appearance
  for ax in AX:
      ax = ax_decorate_box(ax)

  # Plot the original input (grayscale image)
  AX[0].pcolormesh(np.mean(X_test[i_sample, ...,], axis=-1))
  AX[0].set_title("Original", fontsize=14)

  # Plot the predicted output (predicted probabilities for the first class)
  AX[1].pcolormesh(y_pred[i_sample, ..., 0], cmap=plt.cm.jet)
  AX[1].set_title("Non-water Pixels (red for high probabilities)", fontsize=14)

  # Plot the true labels (ground truth)
  AX[2].pcolormesh(y_test[i_sample, ..., 0], cmap=plt.cm.jet)
  AX[2].set_title("Labeled truth", fontsize=14)

  # Show the plot
  plt.show()

visualize_mask_prediction (y_pred, y_test)
```



UNET3PLUS has poor performance, indicated by metrics such as Accuracy (0.6909), Mean IoU (0.5204), Dice Coefficient (0.6825), and Cross-Entropy Loss (0.5776), coupled with the observation that predicted masks are as close as random. This might be due to various reasons:

- The model did not converged fully, resulting in underfitting
- U-Net 3+ is a sophisticated architecture. If the dataset is small or not sufficiently complex, the model may overfit or fail to leverage its full potential.

SUMMARY OF IMAGE SEGMENTATION MODELS PERFORMANCE

Model	Accuracy	IoU	Dice Coef	Cross Entropy Loss
UNET	97.6%	95.3%	97.59%	0.0644
Attention UNET	96.59%	93.39%	96.58%	0.1035
UNET3+	69.09%	52.04%	68.25%	0.5776

PIPELINE 3: IMAGE GENERATION

I experiment with Diffusion, a variation of Stable Diffusion but trained on satellite images and allow users to include text prompt and a metadata of the location where the images should be generated (for example, generate the image of ResHall by including the geolocation (lat, lon) of the exact address of Reshall).

DiffusionSat is an extension of the Stable Diffusion model, a framework for generating images from text descriptions using diffusion-based models. At its core, the model leverages denoising diffusion probabilistic models (DDPM), where noise is progressively added to an image over several steps, and then a neural network is trained to reverse this process, effectively "denoising" it into a meaningful image.

DiffusionSat modifies this general architecture by integrating satellite data as the central input, focusing on creating high-resolution geospatial images, or other specific image types derived from satellite observations. These models often employ a U-Net architecture that can handle conditional image generation based on text, and it is well-suited for various applications, including the generation of satellite imagery from text prompts.

I adapt the code by pulling from the GitHUb of the creator of DiffusionSat and finetune it using the lastest checkpoints that they included in the Github page. Even though I do not run model training, I wil try my best to explain their code base.

@inproceedings{ khanna2024diffusionsat, title={DiffusionSat: A Generative Foundation Model for Satellite Imagery}, author={Samar Khanna and Patrick Liu and Linqi Zhou and Chenlin Meng and Robin Rombach and Marshall Burke and David B. Lobell and Stefano Ermon}, booktitle={The Twelfth International Conference on Learning Representations}, year={2024}, url={https://openreview.net/forum?id=I5webNFDgQ} }

In the provided code snippet of the `StableDiffusionPipeline` class in their GitHub page:

1. `StableDiffusionPipeline` Initialization

```
class StableDiffusionPipeline(DiffusionPipeline):
    def __init__(self, vae: AutoencoderKL, text_encoder: CLIPTextModel, tokenizer: CLIPTokenizer,
                 unet: SatUNet, scheduler: KarrasDiffusionSchedulers, safety_checker:
StableDiffusionSafetyChecker,
                 feature_extractor: CLIPFeatureExtractor, requires_safety_checker: bool = True):
```

- The `StableDiffusionPipeline` inherits from `DiffusionPipeline`, which encapsulates the common functionality for diffusion-based models.
- **Components initialized**:
  - `vae`: **Variational Autoencoder** (VAE) for encoding and decoding images in latent space.
  - `text_encoder`: A pre-trained **CLIP text encoder**, which converts text into embeddings that guide the image generation.
  - `tokenizer`: Used to convert text inputs into tokens that can be processed by the `text_encoder`.
  - `unet`: The model responsible for denoising the encoded image latents (here, it is a `SatUNet` model, which can be specialized for satellite data).
  - `scheduler`: A scheduling mechanism for controlling how noise is removed during image generation, ensuring smooth transitions from noise to the final image.
  - `safety_checker` and `feature_extractor`: For evaluating and extracting features from the generated images, respectively, to ensure they meet safety guidelines.

2. Encoder and Decoder Setup

```
self.register_modules(
    vae=vae,
    text_encoder=text_encoder,
    tokenizer=tokenizer,
    unet=unet,
    scheduler=scheduler,
    safety_checker=safety_checker,
    feature_extractor=feature_extractor,
)
```

- Here, all the components (VAE, text encoder, tokenizer, U-Net, etc.) are registered as part of the pipeline.
- `unet=SatUNet`: This highlights the specialization of the U-Net model for satellite data (instead of generic imagery like in the original Stable Diffusion model). The **SatUNet** would likely have custom layers or adjustments designed to better handle the features of satellite imagery, which might include geographic features, terrain, and other attributes unique to satellite observations.

3. Prompt Encoding Function

```
def _encode_prompt(self, prompt, device, num_images_per_prompt, do_classifier_free_guidance,
                   negative_prompt=None, prompt_embeds: Optional[torch.FloatTensor] = None,
                   negative_prompt_embeds: Optional[torch.FloatTensor] = None):
```

- **Text-to-Image Conversion**: This function encodes the provided textual input (i.e., the user's prompt) into embeddings that the model will use to generate the image.
- **Classifier Free Guidance**: This technique allows for more controlled generation by manipulating the generated image in response to both positive and negative prompts.
- **Integration with Satellite Data**: The satellite image generation would leverage the text encoder to extract meaningful embeddings from the prompts, guiding the generation of spatially-aware imagery.

```
text_inputs = self.tokenizer(prompt, padding="max_length", max_length=self.tokenizer.model_max_length,
                             truncation=True, return_tensors="pt")
text_input_ids = text_inputs.input_ids
```

- The input prompt is tokenized into IDs, and the length is managed to ensure that the tokens fit within the model's limits.
- This process is crucial for handling text-based prompts related to satellite imagery (e.g., "satellite image of forest cover in Brazil").

How DiffusionSat Works in Detail

1. **Input Text Processing**:
   - A text prompt such as "satellite image of forest loss in tropical regions" is tokenized and embedded by the **CLIPTextModel**.
2. **Latent Space Encoding/Decoding**:
   - The text encoding is used to guide the image generation process. The **AutoencoderKL** (VAE) converts the image into a latent representation.
3. **Noise Addition and Denoising**:
   - The model works by adding noise to a latent space representation and progressively denoising it using a **UNet-based model** (in this case, **SatUNet** for handling satellite-specific features).
   - The **KarrasDiffusionSchedulers** control how the noise is removed over time, guiding the generation process.

By customizing the UNet model (via **SatUNet**) DiffusionSat allows the generation of highly relevant satellite imagery from text prompts, making it an effective tool for geospatial applications such as environmental monitoring, urban planning, and more.

In [ ]:
```
!git clone https://github.com/samar-khanna/DiffusionSat.git
%cd DiffusionSat
!pip install torch torchvision --index-url https://download.pytorch.org/whl/cu118
!pip install -e ".[torch]"  # Install editable DiffusionSat
!pip install -r requirements_remaining.txt
!pip show diffusionsat
```

In [ ]:
```
import sys
sys.path.append('/content/DiffusionSat')

import argparse
import logging
import math
import os
import random
import webdataset as wds
from pathlib import Path
from copy import deepcopy
import pandas as pd
import numpy as np
from PIL import Image

from pyproj import Geod
from shapely.geometry import shape as shapey
from shapely.wkt import loads as shape_loads

import torch
from torchvision import transforms
```

```
The cache for model files in Transformers v4.22.0 has been updated. Migrating your old cache. This is a one-time o
nly operation. You can interrupt this and resume the migration later on by calling `transformers.utils.move_cache
()`.
0it [00:00, ?it/s]
```

In [ ]:
```
rm -rf ~/.cache/huggingface
```

In [ ]:
```
from diffusionsat import (
    SatUNet, DiffusionSatPipeline,
    SampleEqually,
    fmow_tokenize_caption, fmow_numerical_metadata,
    spacenet_tokenize_caption, spacenet_numerical_metadata,
    satlas_tokenize_caption, satlas_numerical_metadata,
    combine_text_and_metadata, metadata_normalize,
)
```

```
2024-12-18 16:30:25.422920: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:485] Unable to register cuFF
T factory: Attempting to register factory for plugin cuFFT when one has already been registered
2024-12-18 16:30:25.443777: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:8454] Unable to register cuD
NN factory: Attempting to register factory for plugin cuDNN when one has already been registered
2024-12-18 16:30:25.449967: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1452] Unable to register cu
BLAS factory: Attempting to register factory for plugin cuBLAS when one has already been registered
2024-12-18 16:30:26.955519: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find
TensorRT
/content/DiffusionSat/src/diffusers/models/cross_attention.py:30: FutureWarning: Importing from cross_attention is
deprecated. Please import from diffusers.models.attention_processor instead.
  deprecate(
```

Before finetuning

I initially set use_metadata = False. Looking at the images generated below, I can clearly see that the image was generated using only image caliberation from text prompt (the normal text-to-image output from models like DALL-E,...) without incorporating spatial information from the specific location that the users want to generate.

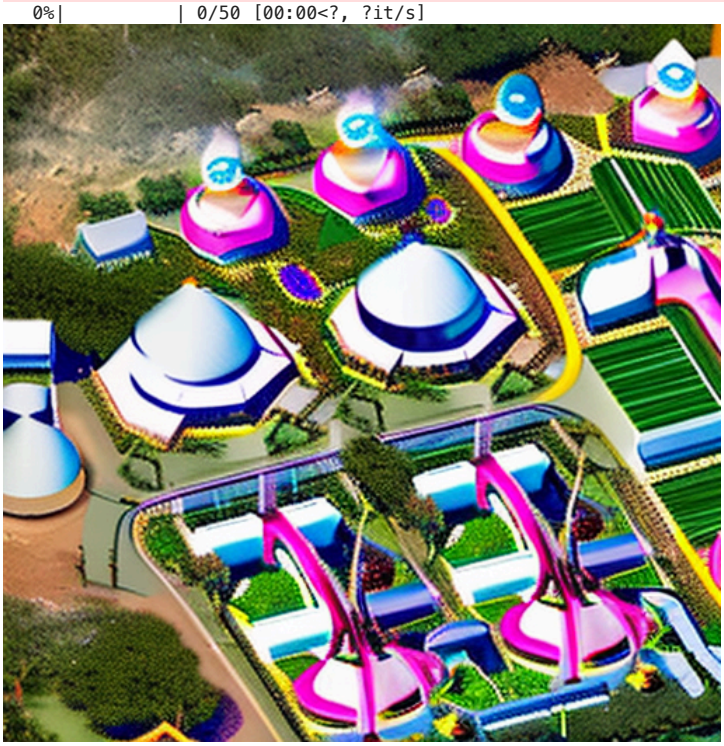
In [ ]:
```
unet1 = SatUNet.from_pretrained("stabilityai/stable-diffusion-2-1", subfolder="unet", use_metadata=False, torch_dty
pipe1 = DiffusionSatPipeline.from_pretrained("stabilityai/stable-diffusion-2-1", unet=unet1, torch_dtype=torch.floa
pipe1 = pipe1.to("cuda")
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_download.py:797: FutureWarning: `resume_download` is
deprecated and will be removed in version 1.0.0. Downloads always resume when possible. If you want to force a new
download, use `force_download=True`.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/to
kens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
unet/config.json:   0%|          | 0.00/939 [00:00<?, ?B/s]
diffusion_pytorch_model.safetensors:   0%|          | 0.00/3.46G [00:00<?, ?B/s]
model_index.json:   0%|          | 0.00/537 [00:00<?, ?B/s]
Fetching 11 files:   0%|          | 0/11 [00:00<?, ?it/s]
tokenizer/merges.txt:   0%|          | 0.00/525k [00:00<?, ?B/s]
text_encoder/config.json:   0%|          | 0.00/633 [00:00<?, ?B/s]
(…)ature_extractor/preprocessor_config.json:   0%|          | 0.00/342 [00:00<?, ?B/s]
scheduler/scheduler_config.json:   0%|          | 0.00/345 [00:00<?, ?B/s]
tokenizer/tokenizer_config.json:   0%|          | 0.00/824 [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/1.36G [00:00<?, ?B/s]
tokenizer/vocab.json:   0%|          | 0.00/1.06M [00:00<?, ?B/s]
tokenizer/special_tokens_map.json:   0%|          | 0.00/460 [00:00<?, ?B/s]
diffusion_pytorch_model.safetensors:   0%|          | 0.00/335M [00:00<?, ?B/s]
vae/config.json:   0%|          | 0.00/611 [00:00<?, ?B/s]
```

In [ ]:
```
caption = "a fmow satellite image of a electric substation in India"
metadata = metadata_normalize([76.5712666476, 28.6965307997, 0.929417550564, 0.0765712666476, 2015, 2, 27]).tolist(
image = pipe1(caption, num_inference_steps=50, guidance_scale=7.5, height=512, width=512).images[0]
image
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/50 [00:00<?, ?it/s]
```

Out[ ]:

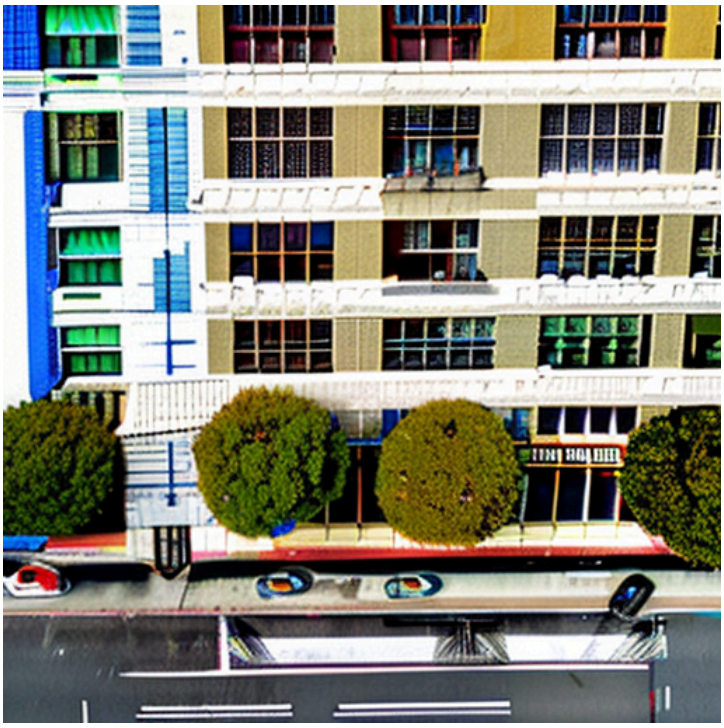


In [ ]:
```
caption = "a fmow satellite image of a building in San Francisco"
metadata = metadata_normalize([-122.40876543143693,37.78354599626813, 0.929417550564, 0.0765712666476, 2020, 2, 27]
```

In [ ]:
```
building_SF = pipe1(caption, num_inference_steps=50, guidance_scale=7.5, height=512, width=512).images[0]
building_SF
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/50 [00:00<?, ?it/s]
```

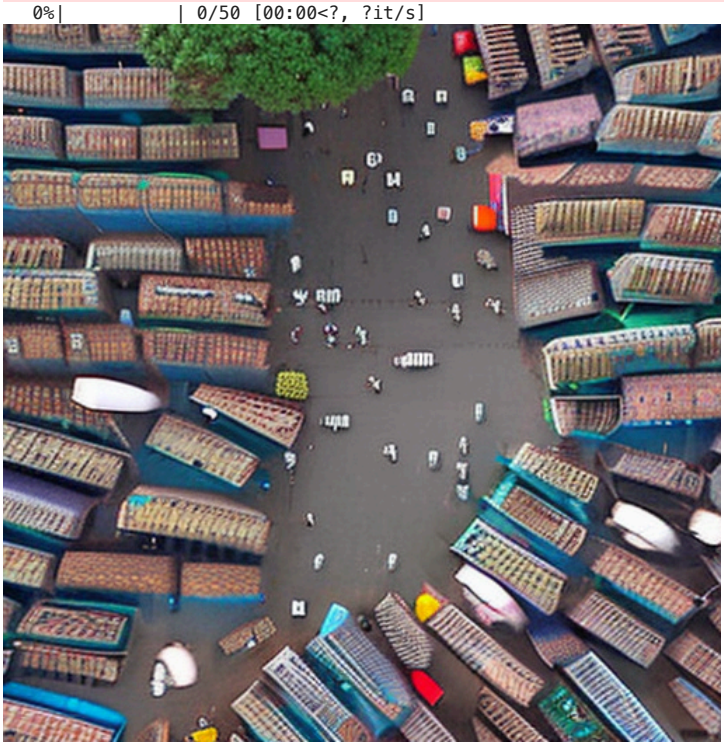
Out[ ]:



In [ ]:
```
street_HN = pipe1("a fmow satellite image of a crowded street in Ha Noi", num_inference_steps=50, guidance_scale=7.
street_HN
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/50 [00:00<?, ?it/s]
```

Out[ ]:



In [ ]:
```
species_Africa = pipe1("a fmow satellite image of endangered species habitat in Africa", num_inference_steps=50, gu
species_Africa
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/50 [00:00<?, ?it/s]
```

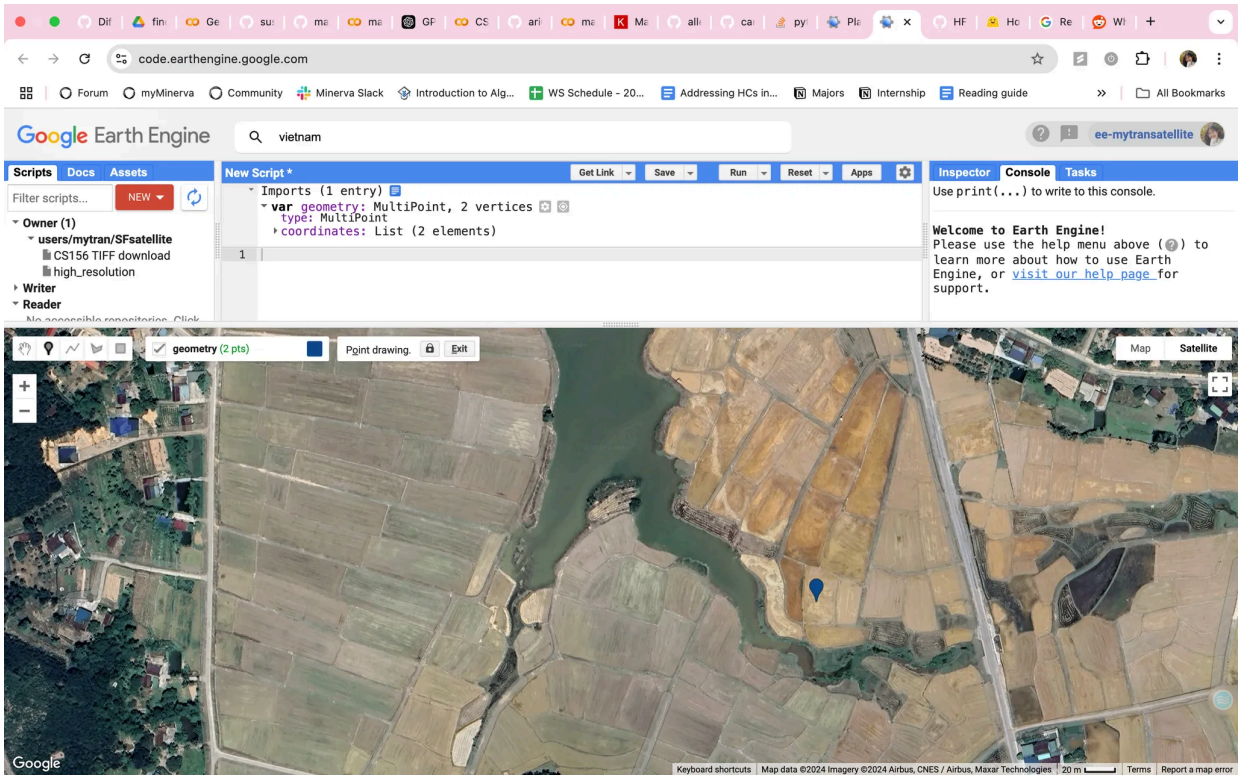
Out[ ]:

After finetuning

After finetuning, I include a metadata, number_of_inference_step determines how many steps it takes to finetune the model to the specific location that the metadata is pointing at, and the guidance_scale controls the extent of "creativity" that the models generate images (how closely the users want the model to use specific location information in the reality to make its image generation).

In [ ]:
```
path = '/content/drive/MyDrive/Minerva Academic/3rd year/Fall 2024/CS156/finetune_sd21_sn-satlas-fmow_snr5_md7norm_
unet = SatUNet.from_pretrained(path + 'checkpoint-150000', subfolder="unet", torch_dtype=torch.float16)
pipe = DiffusionSatPipeline.from_pretrained(path, unet=unet, torch_dtype=torch.float16)
pipe = pipe.to("cuda")
```

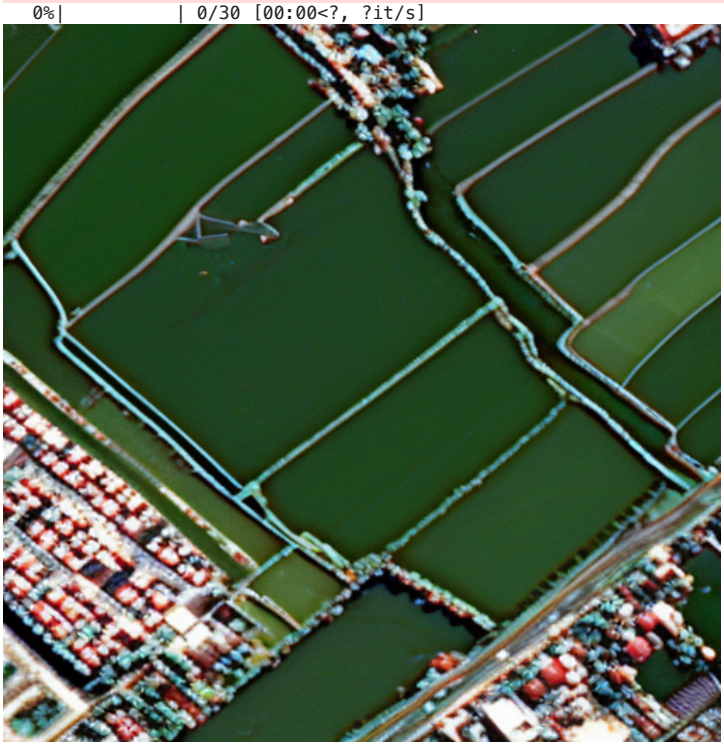
I first search for metadata of the location I want to explore using Google Earth Engine. As you can see I pick a point in the middle of the rice paddle field. The image generated incorporate this spatial information quite well since it also centers the picture around the paddle field.



In [ ]:
```
rice_metadata = metadata_normalize([105.42519898381683,18.897997783830185, 0.929417550564, 0.0765712666476, 2024, 5
finetune_rice = pipe("a fmow satellite image of a rice paddle field in Vietnam", metadata=rice_metadata, num_infere
finetune_rice
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/30 [00:00<?, ?it/s]
```

Out[ ]:



The image generated using the location of the ResHall addresss also incorporate specific spatial information. insptead of a general image of a buildinbg, it now generates an image that is close to the observation that we can see from a satellite point of view, and it also shows close building distribution in the Rreshall area.

In [ ]:
```
caption = "a fmow satellite image of a building in a Tenderloin homeless neighborhood in Turk Street in San Francis
metadata = metadata_normalize([-122.40876543143693,37.78354599626813, 0.929417550564, 0.0765712666476, 2020, 2, 27]
```

In [ ]:
```
finetune_reshall = pipe(caption, metadata=metadata, num_inference_steps=20, guidance_scale=7.5, height=512, width=5
finetune_reshall
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/20 [00:00<?, ?it/s]
```

Out[ ]:

IN the 3 images below, I gradually increase the num_inference_step and guidance_scale of a finetuned picture of Ha Noi, Vietnam. I observe that as we increase those 2 parameters, the spatial scale of the picture is larger (a more zoom image output) and the image is

indeed less generic. The first image just shows a general satellite image of a crowded area while the third image includes specific spatial information, especially how the roofs are all in red color which is very similar to the color of the roofs in Ha Noi.

In [ ]:
```
HN_caption = "a fmow satellite image of a crowded street in Ha Noi"
HN_metadata = metadata_normalize([105.81097881273732,21.01100233446188, 0.929417550564, 0.0765712666476, 2020, 2, 2
```

In [ ]:
```
finetune_HN = pipe(HN_caption, metadata=HN_metadata, num_inference_steps=20, guidance_scale=7.5, height=512, width=
finetune_HN
```

```
/content/DiffusionSat/diffusionsat/pipeline.py:594: FutureWarning: Accessing config attribute `in_channels` direct
ly via 'SatUNet' object attribute is deprecated. Please access 'in_channels' over 'SatUNet's config object instea
d, e.g. 'unet.config.in_channels'.
  num_channels_latents = self.unet.in_channels
  0%|          | 0/20 [00:00<?, ?it/s]
```

Out[ ]:

In [ ]:
```
finetune_HN = pipe(HN_caption, metadata=HN_metadata, num_inference_steps=50, guidance_scale=14, height=512, width=5
finetune_HN
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

Out[ ]:

In [ ]:
```

```

In [ ]:
```
finetune_HN = pipe(HN_caption, metadata=HN_metadata, num_inference_steps=100, guidance_scale=20, height=512, width=
finetune_HN
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
```

Out[ ]:

AI statement: I mainly use AI to explain the sources I found on the internet and to debug.