def binary\_crossentropy\_with\_l2(y\_true, y\_pred, l2\_reg=0.01):

cross\_entropy = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

l2\_loss = l2\_reg \* tf.reduce\_sum([tf.nn.l2\_loss(var) for var in tf.trainable\_variables()])

loss = cross\_entropy + l2\_loss

return tf.keras.backend.mean(loss)

l2\_reg=0.01 is a parameter in the function binary\_crossentropy\_with\_l2 that controls the strength of L2 regularization applied to the model weights.

L2 regularization, also known as weight decay, is a commonly used regularization technique that penalizes large weights in a neural network. It works by adding a penalty term to the loss function that is proportional to the square of the L2-norm of the model weights. This penalty term encourages the model to use smaller weights, which can help prevent overfitting and improve generalization performance.

The l2\_reg parameter controls the strength of the L2 regularization by setting the scaling factor for the L2 penalty term. A larger value of l2\_reg corresponds to a stronger regularization penalty, which can help prevent overfitting but may also reduce the model's ability to fit the training data. On the other hand, a smaller value of l2\_reg corresponds to a weaker regularization penalty, which may allow the model to fit the training data more closely but may also increase the risk of overfitting. The optimal value of l2\_reg depends on the specific problem and data set being used, and may need to be tuned through experimentation.

Here's an example of how you could implement learning rate scheduling in Python for a U-Net model using TensorFlow:

python

Copy code

import tensorflow as tf

# Define the U-Net model

model = tf.keras.models.Sequential([

# ... layers of the U-Net model ...

])

# Define the loss function

def binary\_crossentropy\_with\_l2(y\_true, y\_pred, l2\_reg):

cross\_entropy = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

l2\_loss = l2\_reg \* tf.reduce\_sum([tf.nn.l2\_loss(var) for var in tf.trainable\_variables()])

loss = cross\_entropy + l2\_loss

return tf.keras.backend.mean(loss)

# Define the initial value of l2\_reg

initial\_l2\_reg = 0.01

# Define the function that updates the value of l2\_reg during training

def l2\_reg\_scheduler(epoch, lr):

if epoch < 10:

return initial\_l2\_reg

else:

return initial\_l2\_reg / 10.0

# Define the learning rate schedule callback

l2\_reg\_callback = tf.keras.callbacks.LearningRateScheduler(l2\_reg\_scheduler)

# Compile the model with the initial value of l2\_reg

model.compile(optimizer=tf.keras.optimizers.Adam(),

loss=lambda y\_true, y\_pred: binary\_crossentropy\_with\_l2(y\_true, y\_pred, initial\_l2\_reg))

# Train the model with the learning rate schedule callback

model.fit(train\_dataset,

epochs=50,

validation\_data=val\_dataset,

callbacks=[l2\_reg\_callback])

In this example, we define the U-Net model and the loss function binary\_crossentropy\_with\_l2 that includes l2\_reg. We set the initial value of l2\_reg to 0.01, and define a learning rate schedule function l2\_reg\_scheduler that reduces the value of l2\_reg after the first 10 epochs. We then use the LearningRateScheduler callback to apply the learning rate schedule during training, and compile and train the model with the initial value of l2\_reg.

Note that you would need to define your train\_dataset and val\_dataset appropriately based on your specific dataset. Also, you could modify the learning rate schedule function to suit your needs and experiment with different initial and final values of l2\_reg.

**explain loss=lambda y\_true, y\_pred: binary\_crossentropy\_with\_l2(y\_true, y\_pred, initial\_l2\_reg)**

In the line loss=lambda y\_true, y\_pred: binary\_crossentropy\_with\_l2(y\_true, y\_pred, initial\_l2\_reg), we define the loss function that will be used to optimize the U-Net model during training.

The lambda keyword is used to define an anonymous function that takes two arguments y\_true and y\_pred, which represent the true and predicted segmentation masks, respectively. We then pass these arguments, along with the initial value of l2\_reg, to the binary\_crossentropy\_with\_l2 function, which computes the binary cross-entropy loss with L2 regularization.

By defining the loss function in this way, we can specify the value of l2\_reg during model compilation, and TensorFlow will use this loss function to optimize the model parameters during training. This allows us to include L2 regularization in the loss function, which can help to prevent overfitting and improve the generalization performance of the model.

Here's the modified code to use binary cross-entropy loss with L2 regularization:

css

Copy code

import tensorflow as tf

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import ModelCheckpoint, CSVLogger, ReduceLROnPlateau

from sklearn.model\_selection import train\_test\_split

import pandas as pd

import os

import numpy as np

def binary\_crossentropy\_with\_l2(y\_true, y\_pred, l2\_reg=0.01):

cross\_entropy = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

l2\_loss = l2\_reg \* tf.reduce\_sum([tf.nn.l2\_loss(var) for var in tf.trainable\_variables()])

loss = cross\_entropy + l2\_loss

return tf.keras.backend.mean(loss)

def train():

model = cloud\_net\_model.model\_arch(input\_rows=in\_rows,

input\_cols=in\_cols,

num\_of\_channels=num\_of\_channels,

num\_of\_classes=num\_of\_classes)

model.compile(optimizer=Adam(lr=starting\_learning\_rate), loss=binary\_crossentropy\_with\_l2, metrics=['accuracy'])

model\_checkpoint = ModelCheckpoint(weights\_path, monitor='val\_loss', save\_best\_only=True)

lr\_reducer = ReduceLROnPlateau(factor=decay\_factor, cooldown=0, patience=patience, min\_lr=end\_learning\_rate, verbose=1)

csv\_logger = CSVLogger(experiment\_name + '\_log\_1.log')

train\_img\_split, val\_img\_split, train\_msk\_split, val\_msk\_split = train\_test\_split(train\_img, train\_msk,

test\_size=val\_ratio,

random\_state=42, shuffle=True)

if train\_resume:

model.load\_weights(weights\_path)

print("\nTraining resumed...")

else:

print("\nTraining started from scratch... ")

print("Experiment name: ", experiment\_name)

print("Input image size: ", (in\_rows, in\_cols))

print("Number of input spectral bands: ", num\_of\_channels)

print("Learning rate: ", starting\_learning\_rate)

print("Batch size: ", batch\_sz, "\n")

model.fit\_generator(

generator=mybatch\_generator\_train(list(zip(train\_img\_split, train\_msk\_split)), in\_rows, in\_cols, batch\_sz, max\_bit),

steps\_per\_epoch=np.ceil(len(train\_img\_split) / batch\_sz), epochs=max\_num\_epochs, verbose=1,

validation\_data=mybatch\_generator\_validation(list(zip(val\_img\_split, val\_msk\_split)), in\_rows, in\_cols, batch\_sz, max\_bit),

validation\_steps=np.ceil(len(val\_img\_split) / batch\_sz),

callbacks=[model\_checkpoint, lr\_reducer, ADAMLearningRateTracker(end\_learning\_rate), csv\_logger])

return model

GLOBAL\_PATH = 'G:\cloud dataset\dataset'

TRAIN\_FOLDER = os.path.join(GLOBAL\_PATH, '38-Cloud\_training')

TEST\_FOLDER = os.path.join(GLOBAL\_PATH, '38-Cloud\_test')

in\_rows = 192

in\_cols = 192

num\_of\_channels = 4

num\_of\_classes = 1

starting\_learning\_rate = 1e-4

end\_learning\_rate = 1e-8

max\_num\_epochs = 2000

val\_ratio = 0.2

patience = 15

decay\_factor = 0.7

batch\_sz = 12

max\_bit = 65535

experiment\_name = "Cloud-Net"

weights\_path = os.path.join(GLOBAL\_PATH, experiment\_name + '.h5')

train\_resume = False

# getting input images names

train\_patches\_csv\_name = 'training\_patches\_38-cloud.csv'

df\_train\_img = pd.read\_csv(os.path.join(TRAIN\_FOLDER, train\_patches\_csv\_name))

train\_img, train\_msk = get\_input\_image