NATURAL DISASTER INTENSITY ANALYSIS

DETECTING NATURAL DISASTERS WITH KERAS AND DEEP LEARNING

In the first part of this tutorial, we’ll discuss how computer vision and deep .

From there we’ll review our natural disaster dataset which consists of four classes:

* Cyclone/hurricane
* Earthquake
* Flood
* Wildfire

We’ll then design a set of experiments that will:

* Help us fine-tune VGG16 (pre-trained on ImageNet) on our dataset.
* Find optimal learning rates.
* **Train our model and obtain**> 95% accuracy!

Let’s get started!

### How can computer vision and deep learning detect natural disasters?



Natural disasters cannot be prevented — *but they can be detected.*

All around the world we use sensors to monitor for natural disasters:

* **Seismic sensors** (seismometers) and **vibration sensors** (seismoscopes) are used to monitor for earthquakes (and downstream tsunamis).
* **Radar maps** are used to detect the signature “hook echo” of a tornado (i.e., a hook that extends from the radar echo).
* **Flood sensors** are used to measure moisture levels while **water level sensors** monitor the height of water along a river, stream, etc.
* **Wildfire sensors** are still in their infancy but hopefully will be able to detect trace amounts of smoke and fire.

**Each of these sensors is *highly specialized* to the task at hand** — detect a natural disaster early, alert people, and allow them to get to safety.

### Using computer vision we can *augment* existing sensors, thereby increasing the accuracy of natural disaster detectors, and most importantly, allow people to take precautions, stay safe, and prevent/reduce the number of deaths and injuries that happen due to these disasters.

### Our natural disasters image dataset

* **Cyclone/Hurricane:** 928 images
* **Earthquake:** 1,350
* **Flood:** 1,073
* **Wildfire:** 1,077

We then trained a Convolutional Neural Network to recognize each of the natural disaster cases.

### Downloading the natural disasters dataset

$ tree --dirsfirst --filelimit 10 Cyclone\_Wildfire\_Flood\_Earthquake\_Database

Cyclone\_Wildfire\_Flood\_Earthquake\_Database

├── Cyclone [928 entries]

├── Earthquake [1350 entries]

├── Flood [1073 entries]

├── Wildfire [1077 entries]

└── readme.txt

4 directories, 1 file

Here you can see that each of the natural disasters has its own directory with examples of each class residing inside its respective parent directory.

### Project structure

Using the tree command,

$ tree --dirsfirst --filelimit 10

.

├── Cyclone\_Wildfire\_Flood\_Earthquake\_Database

│ ├── Cyclone [928 entries]

│ ├── Earthquake [1350 entries]

│ ├── Flood [1073 entries]

│ ├── Wildfire [1077 entries]

│ └── readme.txt

├── output

│ ├── natural\_disaster.model

│ │ ├── assets

│ │ ├── variables

│ │ │ ├── variables.data-00000-of-00002

│ │ │ ├── variables.data-00001-of-00002

│ │ │ └── variables.index

│ │ └── saved\_model.pb

│ ├── clr\_plot.png

│ ├── lrfind\_plot.png

│ └── training\_plot.png

├── pyimagesearch

│ ├── \_\_init\_\_.py

│ ├── clr\_callback.py

│ ├── config.py

│ └── learningratefinder.py

├── videos

│ ├── floods\_101\_nat\_geo.mp4

│ ├── fort\_mcmurray\_wildfire.mp4

│ ├── hurricane\_lorenzo.mp4

│ ├── san\_andreas.mp4

│ └── terrific\_natural\_disasters\_compilation.mp4

├── Cyclone\_Wildfire\_Flood\_Earthquake\_Database.zip

├── train.py

└── predict.py

11 directories,20 files

Our project contains:

* The natural disaster dataset. Refer to the previous two sections.
* An output/directory where our model and plots will be stored. The results from my experiment are included.
* A selection of videos/for testing the video classification prediction script.
* Our training script, train.py. This script will perform fine-tuning on a VGG16 model pre-trained on the ImageNet dataset.
* Our video classification prediction script, predict.py , which performs a rolling average prediction to classify the video in real-time.

### Our configuration file

Our project is going to span multiple Python files, so to keep our code tidy and organized (and ensure that we don’t have a multitude of command line arguments), let’s instead create a configuration file to store all important paths and variables.

# import the necessary packages

import os

# initialize the path to the input directory containing our dataset

# of images

DATASET\_PATH = "Cyclone\_Wildfire\_Flood\_Earthquake\_Database"

# initialize the class labels in the dataset

CLASSES = ["Cyclone", "Earthquake", "Flood", "Wildfire"]

The osmodule import allows us to build OS-agnostic paths directly in this config file (**Line 2**).

**Line 6** specifies the root path to our natural disaster dataset.

**Line 7** provides the names of class labels (i.e. the names of the subdirectories in the dataset).

Let’s define our dataset splits:

# define the size of the training, validation (which comes from the

# train split), and testing splits, respectively

TRAIN\_SPLIT = 0.75

VAL\_SPLIT = 0.1

TEST\_SPLIT = 0.25

**Lines 13-15** house our training, testing, and validation split sizes. Take note that the validation split is 10% of the training split (not 10% of all the data).

Next, we’ll define our training parameters:

**Lines 19 and 20** contain the minimum and maximum learning rate for [**Cyclical Learning Rates**](https://pyimagesearch.com/2019/07/29/cyclical-learning-rates-with-keras-and-deep-learning/) (CLR).We’ll learn how to set these learning rate values in the “Finding our initial learning rate” section below.

**Lines 21-24** define the batch size, step size, CLR method, and the number of training epochs.

From there we’ll define the output paths:

# set the path to the serialized model after training

MODEL\_PATH = os.path.sep.join(["output", "natural\_disaste r.model"])

# define the path to the output learning rate finder plot, training

# history plot and cyclical learning rate plot

LRFIND\_PLOT\_PATH = os.path.sep.join(["output", "lrfind\_plot.png"])

TRAINING\_PLOT\_PATH = os.path.sep.join(["output", "training\_plot.png"])

CLR\_PLOT\_PATH = os.path.sep.join(["output", "clr\_plot.png"])

**Lines 27-33** define the following output paths:

* Serialized model after training
* Learning rate finder plot
* Training history plot
* CLR plot

### Implementing our training script with Keras

Our training procedure will consist of two steps:

1. **Step #1:**find optimal learning rates to fine-tune our VGG16 CNN on our dataset.
2. **Step #2:** Use our optimal learning rates in conjunction with Cyclical Learning Rates (CLR) to obtain a high accuracy model.

Our train.pyfile will handle both of these steps.

Go ahead and open up train.py in your favorite code editor and insert the following code:

# set the matplotlib backend so figures can be saved in the background

import matplotlib

matplotlib.use("Agg")

# import the necessary packages

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.applications import VGG16

from tensorflow.keras.layers import Dropout

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Input

from tensorflow.keras.models import Model

from tensorflow.keras.optimizers import SGD

from sklearn.preprocessing import LabelBinarizer

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

from sklearn.metrics import classification\_report

from pyimagesearch.learningratefinder import LearningRateFinder

from pyimagesearch.clr\_callback import CyclicLR

from pyimagesearch import config

from imutils import paths

import matplotlib.pyplot as plt

import numpy as np

import argparse

import pickle

import cv2

import sys

import os

**Lines 2-27** import necessary packages including:

* matplotlib : For plotting (using the "Agg" backend so plot images can be saved to disk).
* tensorflow: Imports including our VGG16 CNN, data augmentation, layer types, and SGD optimizer.
* scikit-learn : Imports including a label binarizer, dataset splitting function, and an evaluation reporting tool.
* LearningRateFinder: Our **[Keras Learning Rate Finder](https://pyimagesearch.com/2019/08/05/keras-learning-rate-finder/" \t "_blank)** class.
* CyclicLR : A Keras callback that oscillates learning rates, known as [**Cyclical Learning Rates**](https://pyimagesearch.com/2019/07/29/cyclical-learning-rates-with-keras-and-deep-learning/). CLRs lead to faster convergence and typically require fewer experiments for hyperparameter updates.
* config : The custom configuration settings we reviewed in the previous section.
* paths : Includes a function for listing the image paths in a directory tree.
* cv2 : OpenCV for preprocessing and display.

Let’s parse command line arguments and grab our image paths:

# construct the argument parser and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-f", "--lr-find", type=int, default=0,

help="whether or not to find optimal learning rate")

args = vars(ap.parse\_args())

# grab the paths to all images in our dataset directory and initialize

# our lists of images and class labels

print("[INFO] loading images...")

imagePaths = list(paths.list\_images(config.DATASET\_PATH))

data = []

labels = []

Recall that most of our settings are in config.py. There is one exception. The --lr-find command line argument tells our script whether or not to find the optimal learning rate (**Lines 30-33**).

**Line 38** grabs paths to all images in our dataset.

We then initialize two synchronized lists to hold our image data and labels (**Lines 39 and 40**).Let’s populate the data and labels lists now:

# loop over the image paths

for imagePath in imagePaths:

# extract the class label

label = imagePath.split(os.path.sep)[-2]

# load the image, convert it to RGB channel ordering, and resize

# it to be a fixed 224x224 pixels, ignoring aspect ratio

image = cv2.imread(imagePath)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

image = cv2.resize(image, (224, 224))

# update the data and labels lists, respectively

data.append(image)

labels.append(label)

# convert the data and labels to NumPy arrays

print("[INFO] processing data...")

data = np.array(data, dtype="float32")

labels = np.array(labels)

# perform one-hot encoding on the labels

lb = LabelBinarizer()

labels = lb.fit\_transform(labels)

**Lines 43-55** loop over imagePaths, while:Extracting the class label from the path (**Line 45**).

* Loading and preprocessing the image (**Lines 49-51**). Images are converted to RGB channel ordering and resized to 224×224 for VGG16.
* Adding the preprocessed image to the data list (**Line 54**).
* Adding the label to the labels list (**Lines 55**).

**Line 59** performs a final preprocessing step by converting the data to a "float32" datatype NumPy array.Similarly, **Line 60** converts labels to an array so that **Lines 63 and 64** can perform one-hot encoding.

From here, we’ll partition our data and set up data augmentation:

# partition the data into training and testing splits

(trainX, testX, trainY, testY) = train\_test\_split(data, labels,

test\_size=config.TEST\_SPLIT, random\_state=42)

# take the validation split from the training split

(trainX, valX, trainY, valY) = train\_test\_split(trainX, trainY,

test\_size=config.VAL\_SPLIT, random\_state=84)

# initialize the training data augmentation object

aug = ImageDataGenerator(

rotation\_range=30,

zoom\_range=0.15,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

shear\_range=0.15,

horizontal\_flip=True,

fill\_mode="nearest")

**Lines 67-72** construct training, testing, and validation splits.

**Lines 75-82** instantiate our data augmentation object.

At this point we’ll set up our VGG16 model for fine tuning:

# load the VGG16 network, ensuring the head FC layer sets are left

# off

baseModel = VGG16(weights="imagenet", include\_top=False,

input\_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the

# the base model

headModel = baseModel.output

headModel = Flatten(name="flatten")(headModel)

headModel = Dense(512, activation="relu")(headModel)

headModel = Dropout(0.5)(headModel)

headModel = Dense(len(config.CLASSES), activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become

# the actual model we will train)

model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will

# \*not\* be updated during the first training process

for layer in baseModel.layers:

layer.trainable = False

# compile our model (this needs to be done after our setting our

# layers to being non-trainable

print("[INFO] compiling model...")

opt = SGD(lr=config.MIN\_LR, momentum=0.9)

model.compile(loss="categorical\_crossentropy", optimizer=opt,

metrics=["accuracy"])

**Lines 86 and 87** load VGG16  using pre-trained ImageNet weights (but without the fully-connected layer head).

**Lines 91-95** create a new fully-connected layer head followed by **Line 99** which adds the new FC layer to the body of VGG16.

**Lines 103 and 104** mark the body of VGG16 as not trainable — we will be training (i.e. fine-tuning) only the FC layer head.

**Lines 109-111** then compile our model with the Stochastic Gradient Descent (SGD ) optimizer and our specified minimum learning rate.

The first time you run the script, you should set the --lr-findcommand line argument to use the **[Keras Learning Rate Finder](https://pyimagesearch.com/2019/08/05/keras-learning-rate-finder/" \t "_blank)** to determine the optimal learning rate. Let’s see how it works:

# check to see if we are attempting to find an optimal learning rate

# before training for the full number of epochs

if args["lr\_find"] > 0:

# initialize the learning rate finder and then train with learning

# rates ranging from 1e-10 to 1e+1

print("[INFO] finding learning rate...")

lrf = LearningRateFinder(model)

lrf.find(

aug.flow(trainX, trainY, batch\_size=config.BATCH\_SIZE),

1e-10, 1e+1,

stepsPerEpoch=np.ceil((trainX.shape[0] / float(config.BATCH\_SIZE))),

epochs=20,

batchSize=config.BATCH\_SIZE)

# plot the loss for the various learning rates and save the

# resulting plot to disk

lrf.plot\_loss()

plt.savefig(config.LRFIND\_PLOT\_PATH)

# gracefully exit the script so we can adjust our learning rates

# in the config and then train the network for our full set of

# epochs

print("[INFO] learning rate finder complete")

print("[INFO] examine plot and adjust learning rates before training")

sys.exit(0)

**Line 115** checks to see if we should attempt to find optimal learning rates. Assuming so, we:

* Initialize  (**Line 119**).
* Start training with a 1e-10 learning rate and exponentially increase it until we hit 1e+1 (**Lines 120-125**).
* Plot the loss vs. learning rate and save the resulting figure (**Lines 129 and 130**).
* Gracefully exit the script after printing a message instructing the user to inspect the learning rate finder plot (**Lines 135-137**).

After this code executes we now need to:

1. **Step #1:**Review the generated plot.
2. **Step #2:**Update config.py with our MIN\_LR and MAX\_LR, respectively.
3. **Step #3:**Train the network on our full dataset.

Assuming we have completed **Steps #1 and #2**, let’s now handle **Step #3** where our minimum and maximum learning rate have already been found and updated in the config.

In this case, it is time to initialize our [**Cyclical Learning Rate class**](https://pyimagesearch.com/2019/07/29/cyclical-learning-rates-with-keras-and-deep-learning/) and commence training:

# otherwise, we have already defined a learning rate space to train

# over, so compute the step size and initialize the cyclic learning

# rate method

stepSize = config.STEP\_SIZE \* (trainX.shape[0] // config.BATCH\_SIZE)

clr = CyclicLR(

mode=config.CLR\_METHOD,

base\_lr=config.MIN\_LR,

max\_lr=config.MAX\_LR,

step\_size=stepSize)

# train the network

print("[INFO] training network...")

H = model.fit\_generator(

aug.flow(trainX, trainY, batch\_size=config.BATCH\_SIZE),

validation\_data=(valX, valY),

steps\_per\_epoch=trainX.shape[0] // config.BATCH\_SIZE,

epochs=config.NUM\_EPOCHS,

callbacks=[clr],

verbose=1)

**Lines 142-147** initialize our CyclicLR .

**Lines 151-157** then train our model [**using .fit\_generator**](https://pyimagesearch.com/2018/12/24/how-to-use-keras-fit-and-fit_generator-a-hands-on-tutorial/) with our aug data augmentation object and our clrcallback.

Upon training completion, we proceed to evaluate and save our model :

# evaluate the network and show a classification report

print("[INFO] evaluating network...")

predictions = model.predict(testX, batch\_size=config.BATCH\_SIZE)

print(classification\_report(testY.argmax(axis=1),

predictions.argmax(axis=1), target\_names=config.CLASSES))

# serialize the model to disk

print("[INFO] serializing network to '{}'...".format(config.MODEL\_PATH))

model.save(config.MODEL\_PATH)

**Line 161** makes predictions on our test set. Those predictions are passed into **Lines 162 and 163** which print a classification report summary.

**Line 167** serializes and saves the fine-tuned model to disk.

Finally, let’s plot both our training history and CLR history:

# construct a plot that plots and saves the training history

N = np.arange(0, config.NUM\_EPOCHS)

plt.style.use("ggplot")

plt.figure()

plt.plot(N, H.history["loss"], label="train\_loss")

plt.plot(N, H.history["val\_loss"], label="val\_loss")

plt.plot(N, H.history["accuracy"], label="train\_acc")

plt.plot(N, H.history["val\_accuracy"], label="val\_acc")

plt.title("Training Loss and Accuracy")

plt.xlabel("Epoch #")

plt.ylabel("Loss/Accuracy")

plt.legend(loc="lower left")

plt.savefig(config.TRAINING\_PLOT\_PATH)

# plot the learning rate history

N = np.arange(0, len(clr.history["lr"]))

plt.figure()

plt.plot(N, clr.history["lr"])

plt.title("Cyclical Learning Rate (CLR)")

plt.xlabel("Training Iterations")

plt.ylabel("Learning Rate")

plt.savefig(config.CLR\_PLOT\_PATH)

**Lines 170-181** generate a plot of our training history and save the plot to disk.

***Note:*** In TensorFlow 2.0, the history dictionary keys have changed from *acc* to *accuracy* and *val\_acc* to *val\_accuracy* . It is especially confusing since “accuracy” is spelled out now, but “validation” is not. Take special care with this nuance depending on your TensorFlow version.

**Lines 184-190** plot our Cyclical Learning Rate history and save the figure to disk.

### Finding our initial learning rate

Before we attempt to fine-tune our model to recognize natural disasters, let’s first use our [**learning rate finder**](https://pyimagesearch.com/2019/08/05/keras-learning-rate-finder/) to find an optimal set of learning rate ranges. Using this optimal learning rate range we’ll then be able to apply [**Cyclical Learning Rates**](https://pyimagesearch.com/2019/07/29/cyclical-learning-rates-with-keras-and-deep-learning/) to improve our model accuracy.

Make sure you have both:

1. Used the “Downloads” section of this tutorial to download the source code.
2. Downloaded the dataset using the “Downloading the natural disasters dataset” section above.

From there, open up a terminal and execute the following command:

$ python train.py --lr-find 1

[INFO] loading images...

[INFO] processing data...

[INFO] compiling model...

[INFO] finding learning rate...

Epoch 1/20

94/94 [==============================] - 29s 314ms/step - loss: 9.7411 - accuracy: 0.2664

Epoch 2/20

94/94 [==============================] - 28s 295ms/step - loss: 9.5912 - accuracy: 0.2701

Epoch 3/20

94/94 [==============================] - 27s 291ms/step - loss: 9.4601 - accuracy: 0.2731

...

Epoch 12/20

94/94 [==============================] - 27s 290ms/step - loss: 2.7111 - accuracy: 0.7764

Epoch 13/20

94/94 [==============================] - 27s 286ms/step - loss: 5.9785 - accuracy: 0.6084

Epoch 14/20

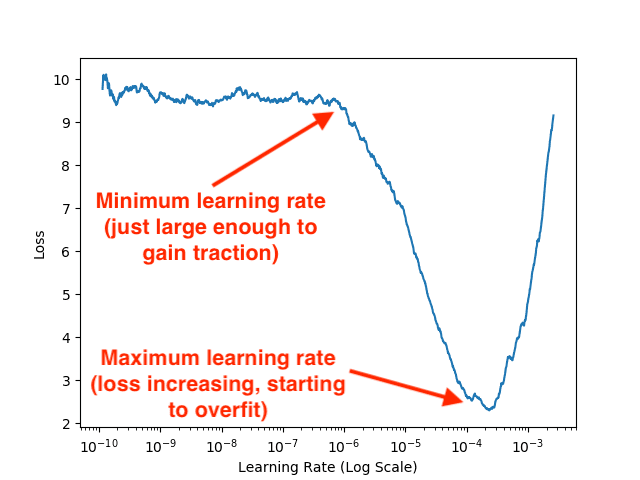
47/94 [==============>...............] - ETA: 13s - loss: 10.8441 - accuracy: 0.3261

[INFO] learning rate finder complete

[INFO] examine plot and adjust learning rates before training

Provided the train.py script exited without error, you should now have a file named lrfind\_plot.png in your output directory.

Take a second now to inspect this image:

**[](https://pyimagesearch.com/wp-content/uploads/2019/11/keras_natural_disaster_lrfind_plot.png)**

**figure 4:** Using a [Keras Learning Rate Finder](https://pyimagesearch.com/2019/08/05/keras-learning-rate-finder/" \t "_blank) to find the optimal learning rates to fine tune our CNN on our natural disaster dataset. We will use the dataset to train a model for detecting natural disasters with the Keras deep learning framework.

Examining the plot you can see that our model initially starts to learn and gain traction around 1e-6 .

Our loss continues to drop until approximately 1e-4 where it starts to rise again, a sure sign of overfitting.

**Our optimal learning rate range is, therefore, 1e-6 to 1e-4 .**

### Update our learning rates

Now that we know our optimal learning rates, let’s go back to our config.py file and update them accordingly:

# define the minimum learning rate, maximum learning rate, batch size,

# step size, CLR method, and number of epochs

MIN\_LR = 1e-6

MAX\_LR = 1e-4

BATCH\_SIZE = 32

STEP\_SIZE = 8

CLR\_METHOD = "triangular"

NUM\_EPOCHS = 48

Notice on **Lines 19 and 20** (highlighted) of our configuration file that the MIN\_LR and MAX\_LRlearning rate values are freshly updated. These values were found by inspecting our **[Keras Learning Rate Finder](https://pyimagesearch.com/2019/08/05/keras-learning-rate-finder/" \t "_blank)** plot in the section above.

### Training the natural disaster detection model with Keras

We can now fine-tune our model to recognize natural disasters!

Execute the following command which will train our network over the full set of epochs:

$ python train.py

[INFO] loading images...

[INFO] processing data...

[INFO] compiling model...

[INFO] training network...

Epoch 1/48

93/93 [==============================] - 32s 343ms/step - loss: 8.5819 - accuracy: 0.3254 - val\_loss: 2.5915 - val\_accuracy: 0.6829

Epoch 2/48

93/93 [==============================] - 30s 320ms/step - loss: 4.2144 - accuracy: 0.6194 - val\_loss: 1.2390 - val\_accuracy: 0.8573

Epoch 3/48

93/93 [==============================] - 29s 316ms/step - loss: 2.5044 - accuracy: 0.7605 - val\_loss: 1.0052 - val\_accuracy: 0.8862

Epoch 4/48

93/93 [==============================] - 30s 322ms/step - loss: 2.0702 - accuracy: 0.8011 - val\_loss: 0.9150 - val\_accuracy: 0.9070

Epoch 5/48

93/93 [==============================] - 29s 313ms/step - loss: 1.5996 - accuracy: 0.8366 - val\_loss: 0.7397 - val\_accuracy: 0.9268

...

Epoch 44/48

93/93 [==============================] - 28s 304ms/step - loss: 0.2180 - accuracy: 0.9275 - val\_loss: 0.2608 - val\_accuracy: 0.9476

Epoch 45/48

93/93 [==============================] - 29s 315ms/step - loss: 0.2521 - accuracy: 0.9178 - val\_loss: 0.2693 - val\_accuracy: 0.9449

Epoch 46/48

93/93 [==============================] - 29s 312ms/step - loss: 0.2330 - accuracy: 0.9284 - val\_loss: 0.2687 - val\_accuracy: 0.9467

Epoch 47/48

93/93 [==============================] - 29s 310ms/step - loss: 0.2120 - accuracy: 0.9322 - val\_loss: 0.2646 - val\_accuracy: 0.9476

Epoch 48/48

93/93 [==============================] - 29s 311ms/step - loss: 0.2237 - accuracy: 0.9318 - val\_loss: 0.2664 - val\_accuracy: 0.9485

[INFO] evaluating network...

precision recall f1-score support

Cyclone 0.99 0.97 0.98 205

Earthquake 0.96 0.93 0.95 362

Flood 0.90 0.94 0.92 267

Wildfire 0.96 0.97 0.96 273

accuracy 0.95 1107

macro avg 0.95 0.95 0.95 1107

weighted avg 0.95 0.95 0.95 1107

[INFO] serializing network to 'output/natural\_disaster.model'...

Here you can see that we are obtaining **95% accuracy** when recognizing natural disasters in the testing set!

Examining our training plot we can see that our validation loss follows our training loss, implying there is little overfitting within ou.

Finally, we have our learning rate plot which shows our our CLR callback oscillates the learning rate between our

MIN\_LR

 and

MAX\_LR

, respectively:

### Implementing our natural disaster prediction script

Now that our model has been trained, let’s see how we can use it to make predictions on images/video it has never seen before — **and thereby pave the way for an automatic natural disaster detection system.**

To create this script we’ll take advantage of the temporal nature of videos, specifically the assumption that subsequent frames in a video will have similar semantic contents**.**

By performing rolling prediction accuracy we’ll be able to “smooth out” the predictions and avoid “prediction flickering”.

I have already covered this near-identical script in-depth in my [***Video Classification with Keras and Deep Learning***](https://pyimagesearch.com/2019/07/15/video-classification-with-keras-and-deep-learning/) article. Be sure to refer to that article for the full background and more-detailed code explanations.

To accomplish natural disaster video classification let’s inspect

# import the necessary packages

from tensorflow.keras.models import load\_model

from pyimagesearch import config

from collections import deque

import numpy as np

import argparse

import cv2

# construct the argument parser and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-i", "--input", required=True,

help="path to our input video")

ap.add\_argument("-o", "--output", required=True,

help="path to our output video")

ap.add\_argument("-s", "--size", type=int, default=128,

help="size of queue for averaging")

ap.add\_argument("-d", "--display", type=int, default=-1,

help="whether or not output frame should be displayed to screen")

args = vars(ap.parse\_args())

**Lines 2-7** load necessary packages and modules. In particular, we’ll be using

deque

 from Python’s

collections

 module to assist with our rolling average algorithm.

**Lines 10-19** [**parse command line arguments**](https://pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/) including the path to our input/output videos, size of our rolling average queue, and whether we will display the output frame to our screen while the video is being generated.

# load the trained model from disk

print("[INFO] loading model and label binarizer...")

model = load\_model(config.MODEL\_PATH)

# initialize the predictions queue

Q = deque(maxlen=args["size"])

# initialize the video stream, pointer to output video file, and

# frame dimensions

print("[INFO] processing video...")

vs = cv2.VideoCapture(args["input"])

writer = None

(W, H) = (None, None)

With our

model

 ,

Q

 , and

vs

 ready to go, we’ll begin looping over

# loop over frames from the video file stream

while True:

# read the next frame from the file

(grabbed, frame) = vs.read()

# if the frame was not grabbed, then we have reached the end

# of the stream

if not grabbed:

break

# if the frame dimensions are empty, grab them

if W is None or H is None:

(H, W) = frame.shape[:2]

# clone the output frame, then convert it from BGR to RGB

# ordering and resize the frame to a fixed 224x224

output = frame.copy()

frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

frame = cv2.resize(frame, (224, 224))

frame = frame.astype("float32")

**Lines 38-47** grab a

frame

 and store its dimensions.

**Lines 51-54** duplicate our

frame

 for

output

 purposes and then preprocess it for classification. The preprocessing steps are, and must be, the same as those that we performed for training.

Now let’s make a natural disaster prediction on the frame:

[**→ Launch Jupyter Notebook on Google Colab → Launch Jupyter Notebook on Google Colab**](https://pyimagesearch.com/2019/11/11/detecting-natural-disasters-with-keras-and-deep-learning/#pyis-cta-modal)

Detecting Natural Disasters with Keras and Deep Learning

# make predictions on the frame and then update the predictions

# queue

preds = model.predict(np.expand\_dims(frame, axis=0))[0]

Q.append(preds)

# perform prediction averaging over the current history of

# previous predictions

results = np.array(Q).mean(axis=0)

i = np.argmax(results)

label = config.CLASSES[i]

**Lines 58 and 59** perform inference and add the predictions to our queue.

**Line 63** performs a rolling average prediction of the predictions available in the

Q

 .

**Lines 64 and 65** then extract the highest probability class label so that we can annotate our frame:

[**→ Launch Jupyter Notebook on Google Colab → Launch Jupyter Notebook on Google Colab**](https://pyimagesearch.com/2019/11/11/detecting-natural-disasters-with-keras-and-deep-learning/#pyis-cta-modal)

Detecting Natural Disasters with Keras and Deep Learning

# draw the activity on the output frame

text = "activity: {}".format(label)

cv2.putText(output, text, (35, 50), cv2.FONT\_HERSHEY\_SIMPLEX,

1.25, (0, 255, 0), 5)

# check if the video writer is None

if writer is None:

# initialize our video writer

fourcc = cv2.VideoWriter\_fourcc(\*"MJPG")

writer = cv2.VideoWriter(args["output"], fourcc, 30,

(W, H), True)

# write the output frame to disk

writer.write(output)

# check to see if we should display the output frame to our

# screen

if args["display"] > 0:

# show the output image

cv2.imshow("Output", output)

key = cv2.waitKey(1) & 0xFF

# if the `q` key was pressed, break from the loop

if key == ord("q"):

break

# release the file pointers

print("[INFO] cleaning up...")

writer.release()

vs.release()

**Lines 68-70** annotate the natural disaster activity in the corner of the

output

 frame.

**Lines 73-80** handle writing the

output

 frame to a video file.

If the

--display

 flag is set, **Lines 84-91** display the frame to the screen and capture keypresses.

Otherwise, processing continues until completion at which point the loop is finished and we perform cleanup (**Lines 95 and 96**).

### Predicting natural disasters with Keras

For the purposes of this tutorial, I downloaded example natural disaster videos via YouTube — the exact videos are listed in the “Credits” section below. You can either use your own example videos or download the videos via the credits list.

Either way, make sure you have used the “Downloads” section of this tutorial to download the source code and pre-trained natural disaster prediction model.

Once downloaded you can use the following command to launch the

predict.py

 script:

$ python predict.py --input videos/terrific\_natural\_disasters\_compilation.mp4 \

--output output/natural\_disasters\_output.avi

[INFO] processing video...

[INFO] cleaning up...

Here you can see a sample result of our model correctly classifying this video clip as “flood”:

**Figure 7:** Natural disaster “flood” classification with Keras and Deep Learning.

The following example comes from the 2016 Fort McMurray wildfire:

**Figure 8:** Detecting “wildfires” and other natural disasters with Keras, deep learning, and computer vision.

For fun, I then tried applying the natural disaster detector to the movie San Andreas

3:52

2:32

v

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | . | |
|  | | |
|  | | |
|  | |  |