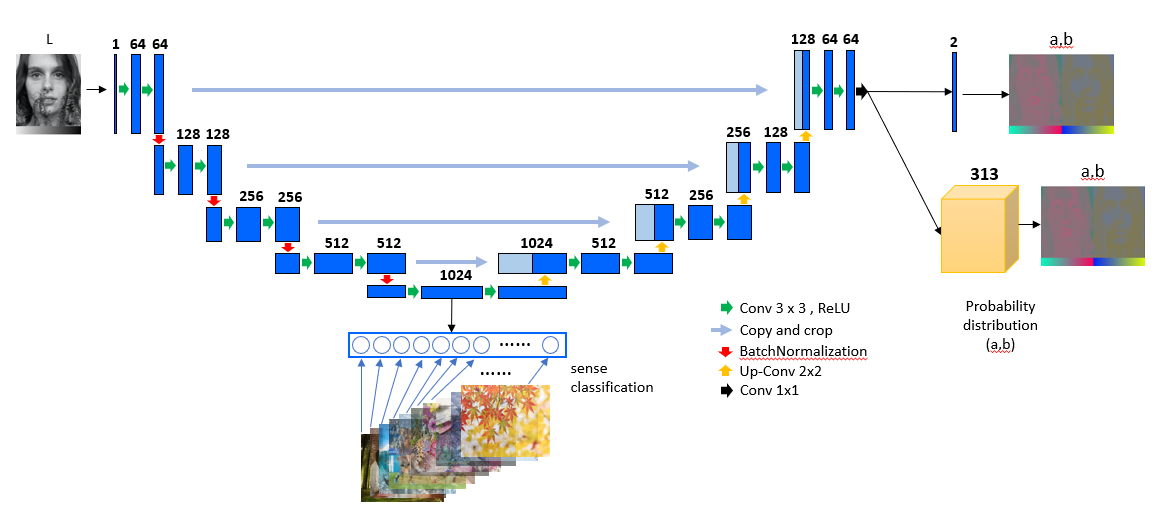
**REPORT PROBLEMS IN OUR MODEL**

**Link GitHub**: <https://github.com/tramtran2/prlab_image_colorization>

# Our Problems:

* **Accuracy is not convergence**
* **Red noise in colorized image**

# Our Models

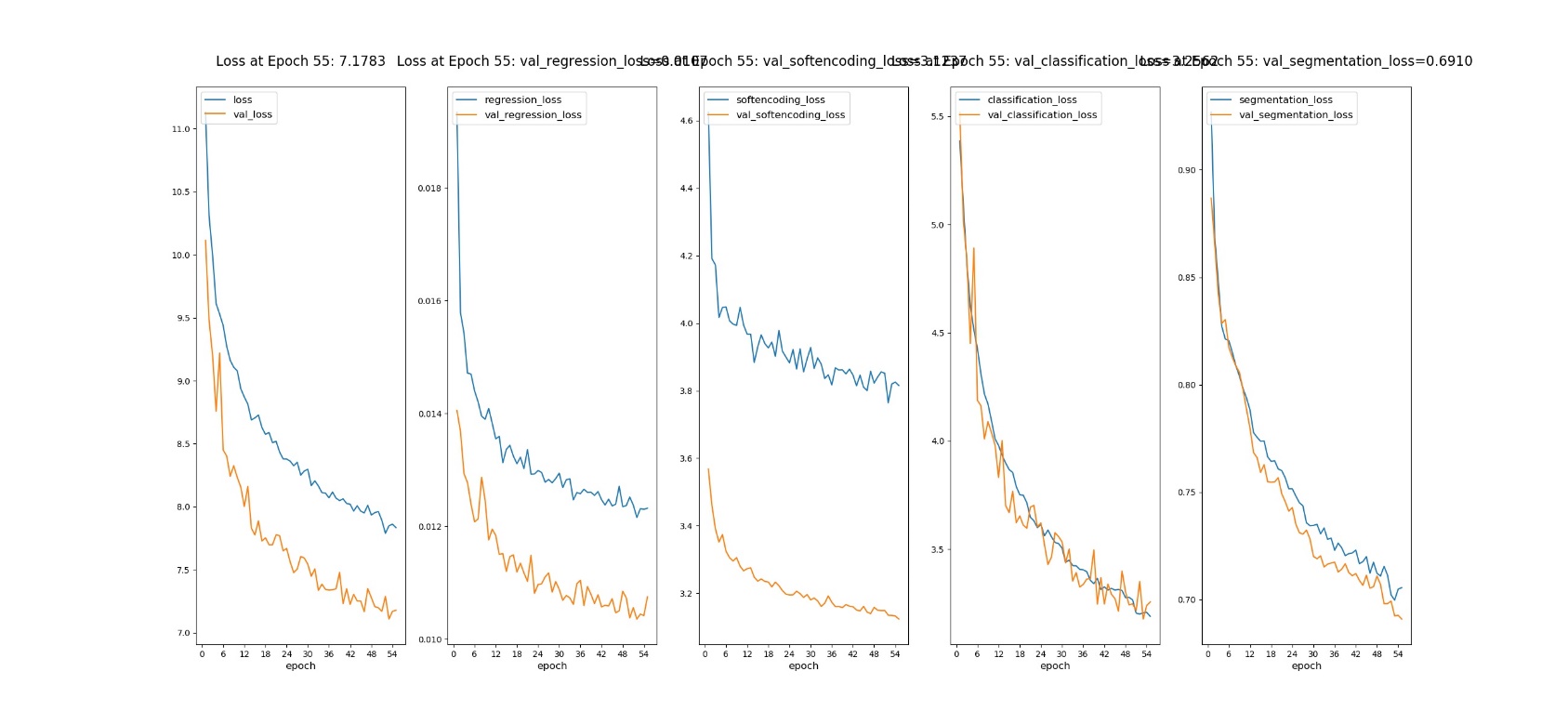


* Input: grayscale image
* Output:
  + Colorization image using regression
  + Colorization image using category cross-entropy with soft encoding with 313 quantization bins
  + Classification branch with scene classification for regularization

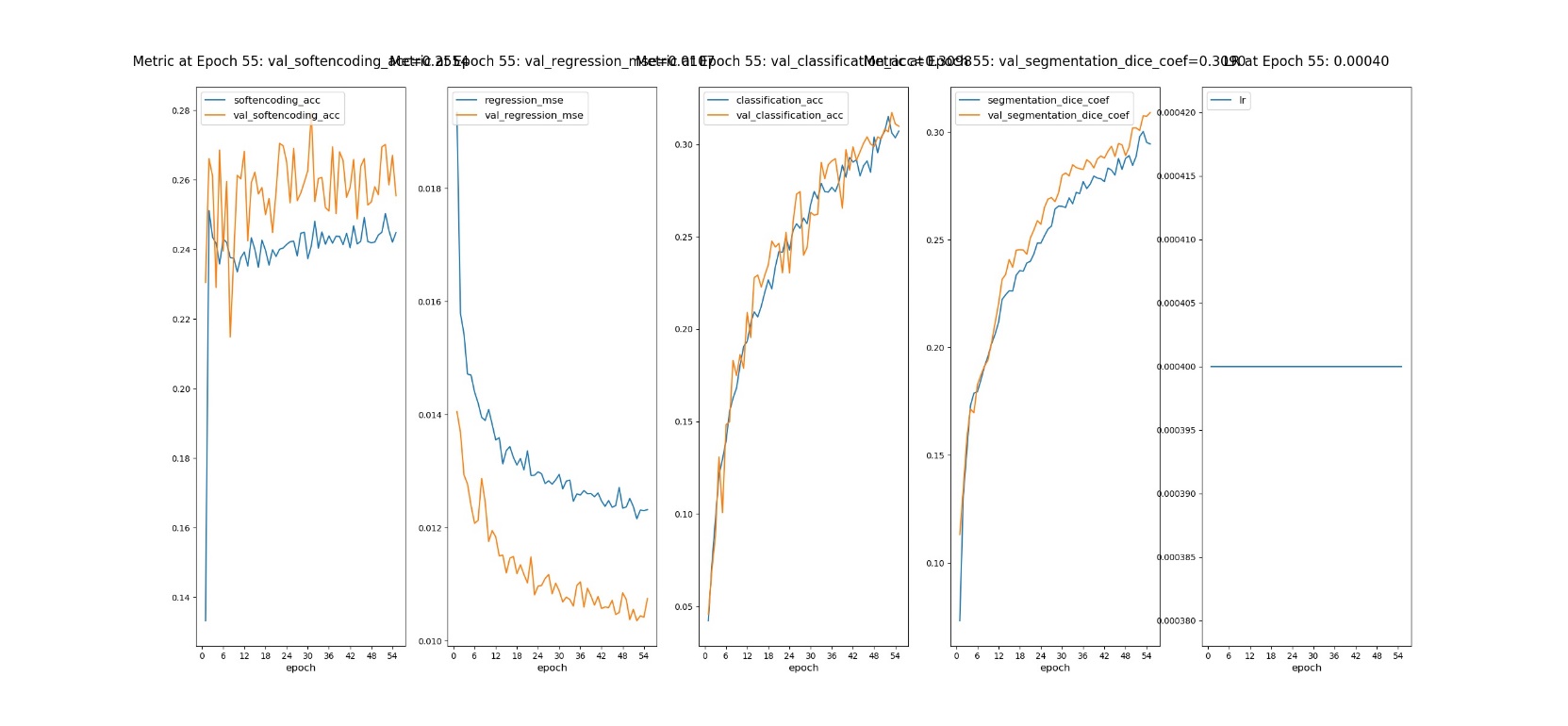
# Training History

* Using RMSProp, Adam with Step Decay, or Constant or Cycle Learning Rate
* Training and validating on Coco-Stuff

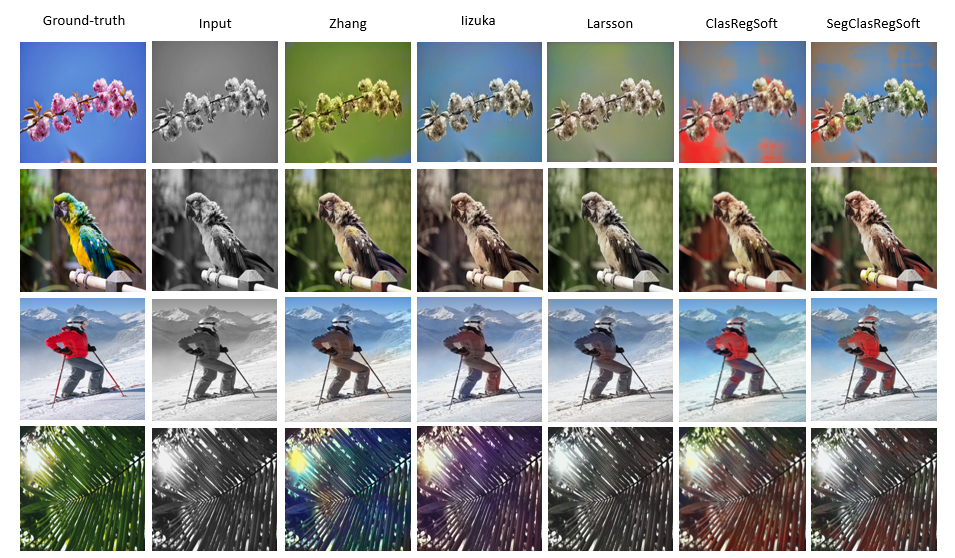
|  |  |
| --- | --- |
|  |  |
| **Ground-truth** | ***Results (with Red Noise and some regions are not colorized)*** |



* ***Loss of soft-encoding is good for convergence (image above – column 3) but the accuracy is noise with around 0.32.***

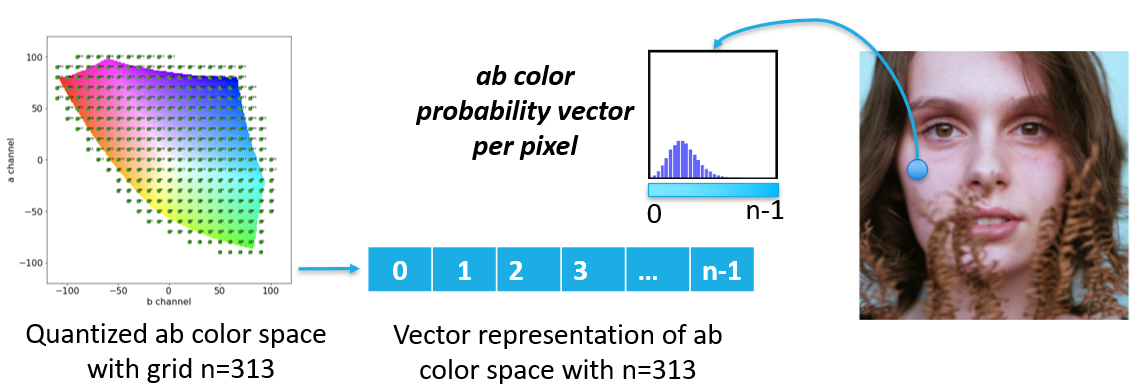


**Testing with collecting images:**



***Red Noise in our method (last two columns)***

# Soft-Encoding



**Implementations:**

For **pts\_in\_hull**: getting from Richard Zhang works (<https://richzhang.github.io/colorization/>)

**Soft-Encoding:**

def load\_nn\_finder(pts\_in\_hull\_path, nb\_neighbors = 5):

    # Load the array of quantized ab value

    q\_ab = np.load(pts\_in\_hull\_path)

    nn\_finder = nn.NearestNeighbors(n\_neighbors=nb\_neighbors, algorithm='ball\_tree').fit(q\_ab)

    return q\_ab, nn\_finder

# load\_nn\_finder

def get\_soft\_encoding(image\_Lab, nn\_finder, nb\_q, sigma\_neighbor = 5):

    """

    image\_Lab = read\_image("...")["res\_image\_Lab"]

    q\_ab, nn\_finder = load\_nn\_finder("pts\_in\_hull.npy", nb\_neighbors = 5)

    y = get\_soft\_encoding(image\_Lab, nn\_finder, nb\_q = q\_ab.shape[0], sigma\_neighbor = 5)

    """

    # get and normalize image\_ab

    # due to preprocessing weighted with minus 128

    image\_ab = image\_Lab[:, :, 1:].astype(np.int32) - 128

    h, w = image\_ab.shape[:2]

    a = np.ravel(image\_ab[:, :, 0])

    b = np.ravel(image\_ab[:, :, 1])

    ab = np.vstack((a, b)).T

    # Get the distance to and the idx of the nearest neighbors

    dist\_neighb, idx\_neigh = nn\_finder.kneighbors(ab)

    # Smooth the weights with a gaussian kernel

    wts = np.exp(-dist\_neighb \*\* 2 / (2 \* sigma\_neighbor \*\* 2))

    wts = wts / np.sum(wts, axis=1)[:, np.newaxis]

    # format the target

    y = np.zeros((ab.shape[0], nb\_q))

    idx\_pts = np.arange(ab.shape[0])[:, np.newaxis]

    y[idx\_pts, idx\_neigh] = wts

    y = y.reshape(h, w, nb\_q)

    return y

# get\_soft\_encoding

**Decode with annealing:**

def decode\_soft\_encoding\_image\_v1(x\_image, y\_image, q\_ab, epsilon = 1e-8, T = 0.38, x\_post\_fn = None, usegpu = False, \*\*kwargs):

    """

    q\_ab = np.load(os.path.join(module\_dir, "data", "pts\_in\_hull.npy").replace("\\", "/"))

    x\_image: L channel (batchsize, height, width, 1) --> gray

    y\_image: softencoding of ab channel (height, width, nb\_q)

    """

    if len(x\_image.shape) == 2:

        x\_batch\_image = x\_image.reshape((1,) + x\_image.shape[0:2] + (1,))

    elif len(x\_image.shape) == 3 and x\_image.shape[2]==1:

        x\_batch\_image = x\_image.reshape((1,) + x\_image.shape[0:3])

    else:

        assert("Invalid Gray Image with shape (w, h) or (w, h, 1)")

    # if

    y\_batch\_image = y\_image.reshape((1,) + y\_image.shape[0:3])

    y\_batch\_RGB = decode\_soft\_encoding\_batch\_image\_v1(x\_batch\_image, y\_batch\_image, q\_ab, epsilon, T, x\_post\_fn, usegpu = usegpu)

    return y\_batch\_RGB.reshape(y\_batch\_RGB.shape[1:])

# decode\_image\_v1

**Decode without annealing:**

def decode\_soft\_encoding\_image\_v0(x\_image, y\_image, q\_ab, x\_post\_fn = None, \*\*kwargs):

    """

    q\_ab = np.load(os.path.join(module\_dir, "data", "pts\_in\_hull.npy").replace("\\", "/"))

    x\_image: L channel (batchsize, height, width, 1) --> gray

    y\_image: softencoding of ab channel (height, width, nb\_q)

    """

    if len(x\_image.shape) == 2:

        x\_batch\_image = x\_image.reshape((1,) + x\_image.shape[0:2] + (1,))

    elif len(x\_image.shape) == 3 and x\_image.shape[2]==1:

        x\_batch\_image = x\_image.reshape((1,) + x\_image.shape[0:3])

    else:

        assert("Invalid Gray Image with shape (w, h) or (w, h, 1)")

    # if

    y\_batch\_image = y\_image.reshape((1,) + y\_image.shape[0:3])

    y\_batch\_RGB = decode\_soft\_encoding\_batch\_image\_v0(x\_batch\_image, y\_batch\_image, q\_ab, x\_post\_fn)

    return y\_batch\_RGB.reshape(y\_batch\_RGB.shape[1:])

# decode\_image\_v0

# Loss

* **Scene-context classification:** Category Cross-Entropy (CCE) loss:

Where C is the number of scene, yi/ is the ground-truth/predicted scene probability.

* **Pixel Classification of ab color distribution**: Weighted Category Cross-Entropy Loss:

Where h, w is the height and width of image, N is the number of quantized colors of ab color distribution,  **is the weighted of color-class at pixel (h,w) to encourage the rare-color**, / is the ground-truth/prediction probability of the soft-encoding color i at pixel (h,w).

* **Regression ab channel**: Using Mean Square Error (MSE) Loss:

Where / is the ground-truth/prediction of ab values at pixel (h,w)

**Implementations**:

def build\_categorical\_crossentropy\_color\_loss(prior\_factor\_path = "prior\_factor.npy", nb\_q = 313, train\_session = None):

    # Load the color prior factor that encourages rare colors

    prior\_factor = None

    if prior\_factor\_path is None:

        prior\_factor = np.ones(nb\_q).astype(np.float32)

    elif os.path.exists(prior\_factor\_path) == False:

        assert(f'prior\_factor\_path: {prior\_factor\_path} is not exists!')

    else:

        prior\_factor = np.load(prior\_factor\_path).astype(np.float32)

    # if

    q = len(prior\_factor) # number of bins

    def categorical\_crossentropy\_color(y\_true, y\_pred):

        y\_true = K.reshape(y\_true, (-1, q))

        y\_pred = K.reshape(y\_pred, (-1, q))

        idx\_max = K.argmax(y\_true, axis=1)

        weights = K.gather(prior\_factor, idx\_max)

        weights = K.reshape(weights, (-1, 1))

        # multiply y\_true by weights

        y\_true = y\_true \* weights

        # cross\_ent = K.categorical\_crossentropy(y\_pred, y\_true)

        cross\_ent = K.categorical\_crossentropy(y\_true, y\_pred)

        cross\_ent = K.mean(cross\_ent, axis=-1)

        return cross\_ent

    # wrapper

    return categorical\_crossentropy\_color

# build\_categorical\_crossentropy\_color\_loss