CSC420

ASSIGNMENT 3

Hamza Yousaf

1002199088

0 Preface

To access all the python files, Colab notebooks, and model weight links, please refer to my Github repository:

https://github.com/Invariant/Cats-And-Circles-Image-Segmentation

To test the code, it is recommended that the user loads up the Colab notebooks onto Google Colab and then run the required cells. The user should run all the import, Google Drive, and function definition cells. The MAIN_DIR_PATH constant should be modified to reflect the user's own directory structure (preferable within Google Drive).

1 Image Segmentation

We begin by loading our data using the *load_data* function. The function processes all images and their corresponding masks by first loading them using cv2. Images are loading with 3 channels and masks are loaded with a single channel. Both images and masks are then normalized, and masks are encoded using 1-hot vectors so they are completely binary.

The U-Net model we implemented mostly adhered to the one described by Ronnenberger et. al's paper. There is an encoder and decoder component, both of which follow a similar but opposite paths. In the encoder part, we have two convolution networks, and then a downsampling using maxpool. We do this until we have 1024 channels, and then in the decoder network, there is an upsampling followed by two convolutions along with a concatenation with the corresponding feature map from the encoder network. Since the original paper does not mention any regularizer, we decided to add some dropout after our dense layers to prevent the network from overfitting.

The proposed model is meant to use SGD + Momentum, but we decided to use the Adam optimizer. Adam optimizer has been determined to be one of the best optimizers that uses SGD + Momentum in its implementation along with other tricks to speed up convergence. In our results, we present two loss functions: Dice-Coefficient loss and Cross-Entropy loss. We found that Cross-Entropy allowed our loss function to decrease quickly and our accuracy to increase quickly. The Dice-Coefficient had the network converging slower, but once converged, the loss and dice-coeff had become stagnant at 0.27 and 0.72 respectfully.

MODEL CODE

```
def unet_model(input_size):
    inputs = Input(name="input1", shape=input_size)
    conv1 = conv2d_block(name="conv1", x=inputs, filters=64,
kernel_size=3, padding="same", activation="relu")
    conv2 = conv2d_block(name="conv2", x=conv1, filters=64, kernel_size=3,
padding="same", activation="relu")
```

```
    maxpool1 = MaxPool2D(name="maxpool1", pool_size=(2, 2),
strides=2)(conv2)

    conv3 = conv2d_block(name="conv3", x=maxpool1, filters=128,
kernel_size=3, padding="same", activation="relu")
    conv4 = conv2d_block(name="conv4", x=conv3, filters=128,
kernel_size=3, padding="same", activation="relu")
    maxpool2 = MaxPool2D(name="maxpool2", pool_size=(2, 2),
strides=2)(conv4)

    conv5 = conv2d_block(name="conv5", x=maxpool2, filters=256,
kernel_size=3, padding="same", activation="relu")
    conv6 = conv2d_block(name="conv6", x=conv5, filters=256,
kernel_size=3, padding="same", activation="relu")
    maxpool3 = MaxPool2D(name="maxpool3", pool_size=(2, 2),
strides=2)(conv6)

    conv7 = conv2d_block(name="conv7", x=maxpool3, filters=512,
kernel_size=3, padding="same", activation="relu")
    conv8 = conv2d_block(name="conv8", x=conv7, filters=512,
kernel_size=3, padding="same", activation="relu")
    maxpool4 = MaxPool2D(name="maxpool4", pool_size=(2, 2),
strides=2)(conv8)

    conv9 = conv2d_block(name="conv9", x=maxpool4, filters=1024,
kernel_size=3, padding="same", activation="relu")
    conv10 = conv2d_block(name="conv10", x=conv9, filters=1024,
kernel_size=3, padding="same", activation="relu")
    upconv1 = UpSampling2D(name="upconv1", size=(2, 2))(conv10)

    concat1 = concatenate(name="concat1", inputs=[conv8, upconv1], axis=3)
    conv11 = conv2d_block(name="conv11", x=concat1, filters=512,
kernel_size=3, padding="same", activation="relu")
    conv12 = conv2d_block(name="conv12", x=conv11, filters=512,
kernel_size=3, padding="same", activation="relu")
    upconv2 = UpSampling2D(name="upconv2", size=(2, 2))(conv12)

    concat2 = concatenate(name="concat2", inputs=[conv6, upconv2], axis=3)
    conv13 = conv2d_block(name="conv13", x=concat2, filters=256,
kernel_size=3, padding="same", activation="relu")
    conv14 = conv2d_block(name="conv14", x=conv13, filters=256,
kernel_size=3, padding="same", activation="relu")
    upconv3 = UpSampling2D(name="upconv3", size=(2, 2))(conv14)

    concat3 = concatenate(name="concat3", inputs=[conv4, upconv3], axis=3)
    conv15 = conv2d_block(name="conv15", x=concat3, filters=128,
```

```
kernel_size=3, padding="same", activation="relu")
    conv16 = conv2d_block(name="conv16", x=conv15, filters=128,
kernel_size=3, padding="same", activation="relu")
    upconv4 = UpSampling2D(name="upconv4", size=(2, 2))(conv16)

    concat4 = concatenate(name="concat4", inputs=[conv2, upconv4], axis=3)
    conv17 = conv2d_block(name="conv17", x=concat4, filters=64,
kernel_size=3, padding="same", activation="relu")
    conv18 = conv2d_block(name="conv18", x=conv17, filters=64,
kernel_size=3, padding="same", activation="relu")
    conv19 = Conv2D(name="conv19", filters=1, kernel_size=1,
activation="sigmoid")(conv18)

    u_model = Model(input=inputs, output=conv19)

    u_model.summary()

    return u_model
```

TRAINING CODE

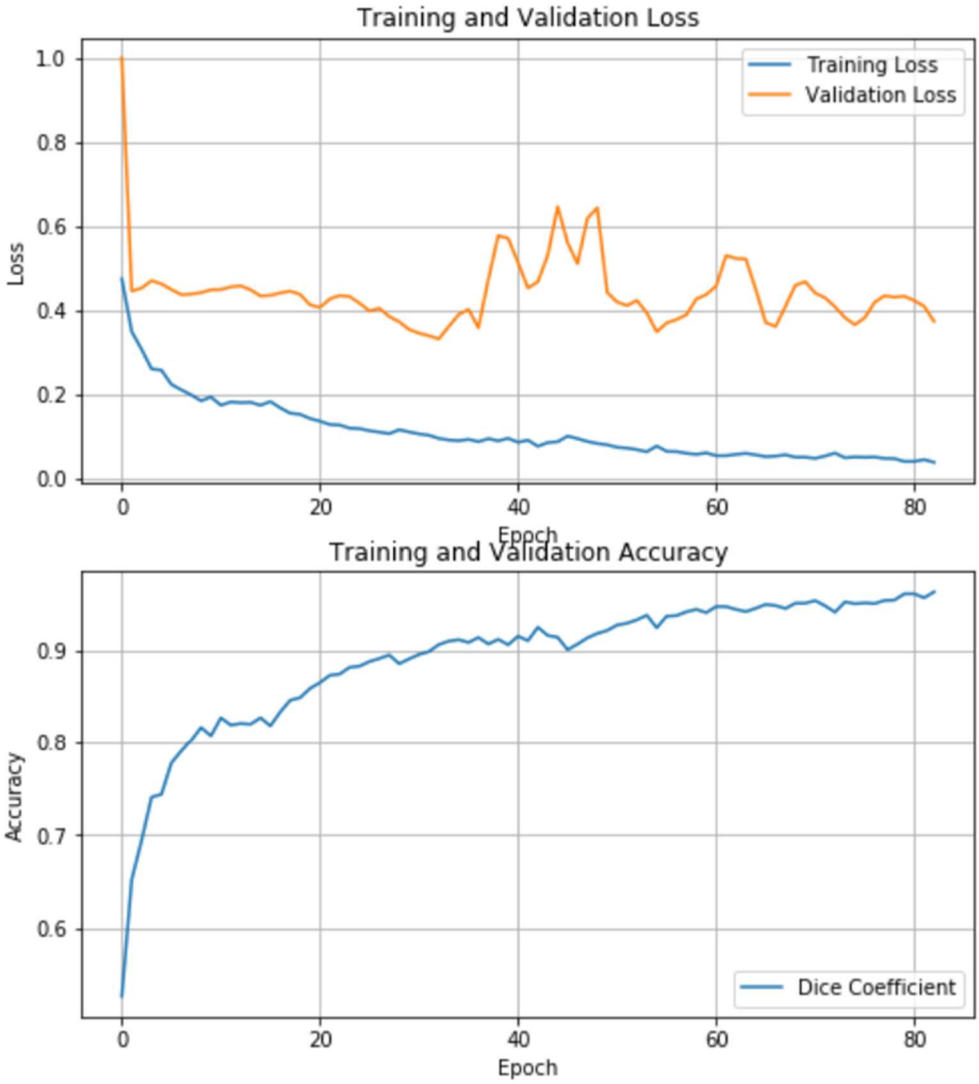
```
def train_model(input_path, output_path, model_name="unet_model",
augment=False, lr=1e-4, epochs=10, valid_split=0,
                early_stop=False, model=None):
    """
    Returns a trained model and the training history.

    :param input_path: path to all the input data (ie. train_X)
    :param output_path: path to all the output data (ie. train_y)
    :param augment: whether or not the augment the data
    :param model_name: name of the new model (weights will be saved under
this name)
    :param lr: learning rate for model
    :param epochs: number of epochs to run for
    :param valid_split: percentage of data that should be used for
validation
    :param early_stop: whether or not to stop early if the validation and
training curves diverge too much
    :param model: an existing model to train
    """
    train_x, train_y = load_data(input_path, output_path, augment)

    if model is None:
        model = unet_model((IMG_WIDTH, IMG_HEIGHT, 3))
        model.compile(optimizer=Adam(lr=lr), loss='binary_crossentropy',
metrics=['accuracy'])
```

```
    # Setup training callbacks
    callbacks = []
    if early_stop:
        callbacks.append(EarlyStopping(monitor='val_loss', verbose=1,
patience=50))
    callbacks.append(ModelCheckpoint(MAIN_DIR_PATH + "/" + model_name +
".hdf5", save_weights_only=True))
    callbacks.append(PlotLossesCallback())

    history = model.fit(train_x, train_y, epochs=epochs,
validation_split=valid_split, callbacks=callbacks)

    plot_model_history(history)

    return model, history
```

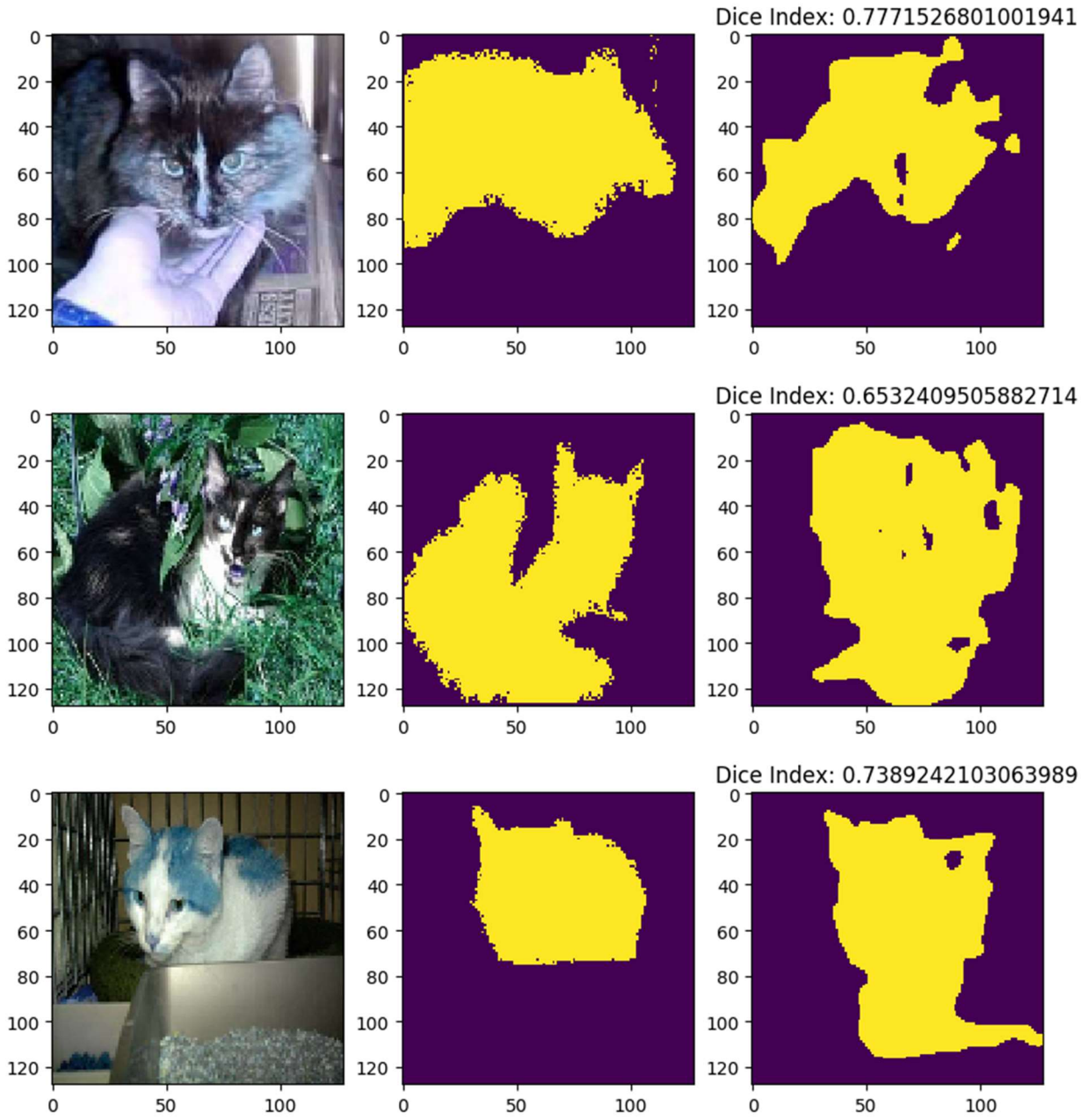
DICE-COEFFICIENT MODEL TRAINING PARAMETERS

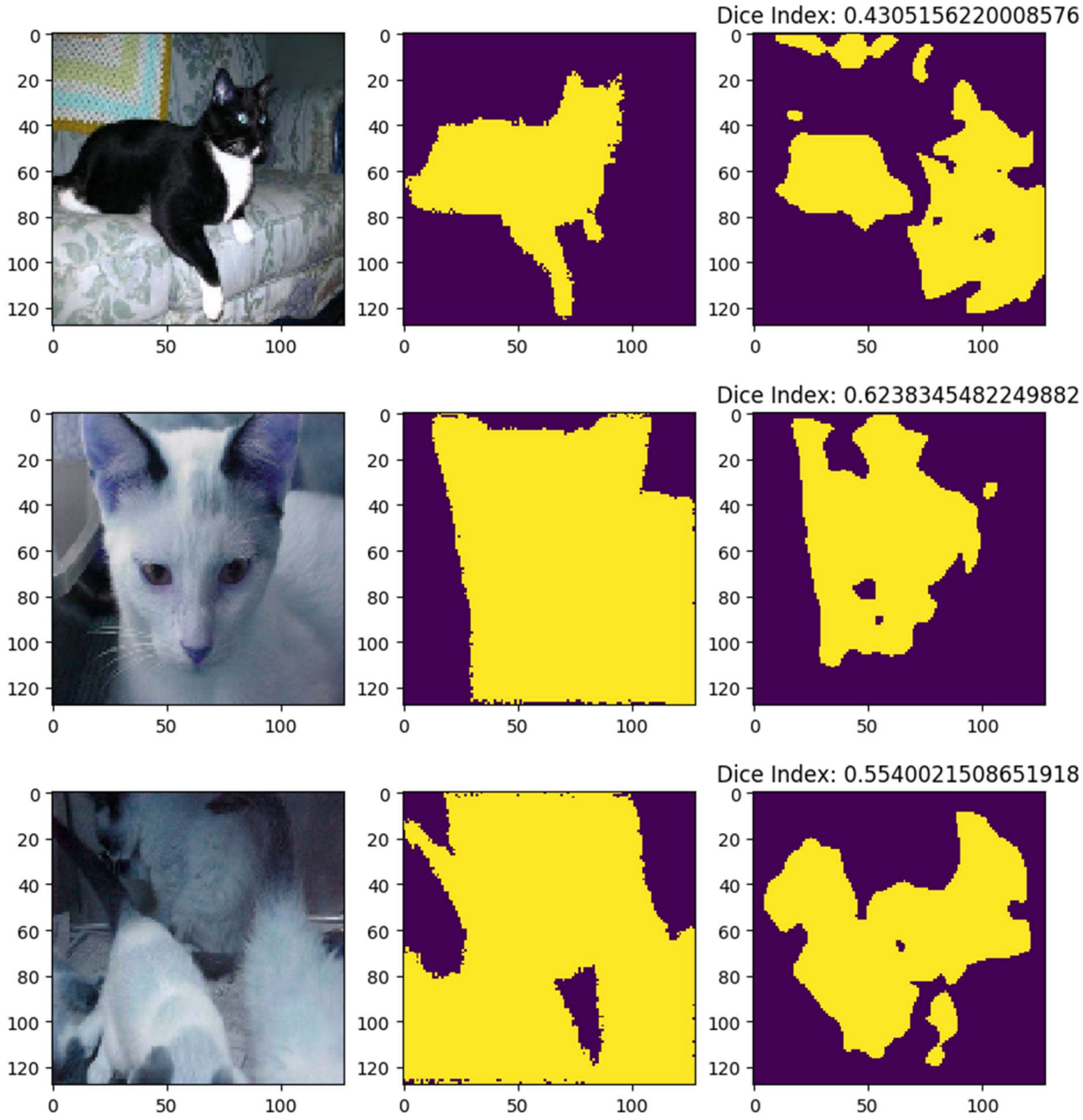
- Optimizer: Adam
- Learning Rate: 1e-3
- Loss: Dice-Coefficient
- Epochs: 200
- Validation Split: 0.1
- Early Stopping: True, Patient=50

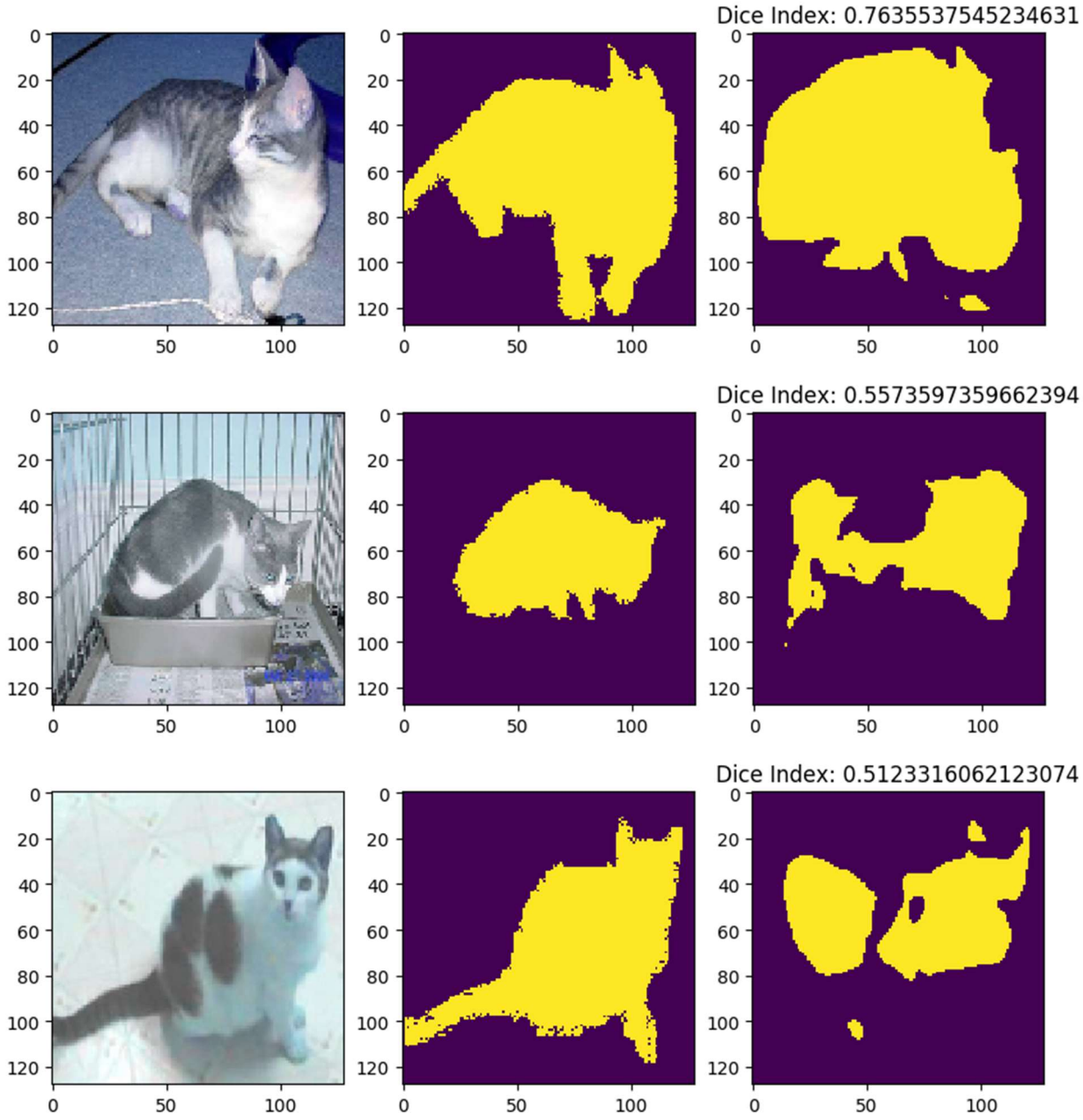
```
dice_coef_loss (cost function):
training   (min:    0.037, max:    0.474, cur:    0.037)
validation (min:    0.331, max:    1.000, cur:    0.373)

dice_coef:
training   (min:    0.526, max:    0.963, cur:    0.963)
validation (min:    0.000, max:    0.669, cur:    0.627)
Epoch 00083: early stopping
```

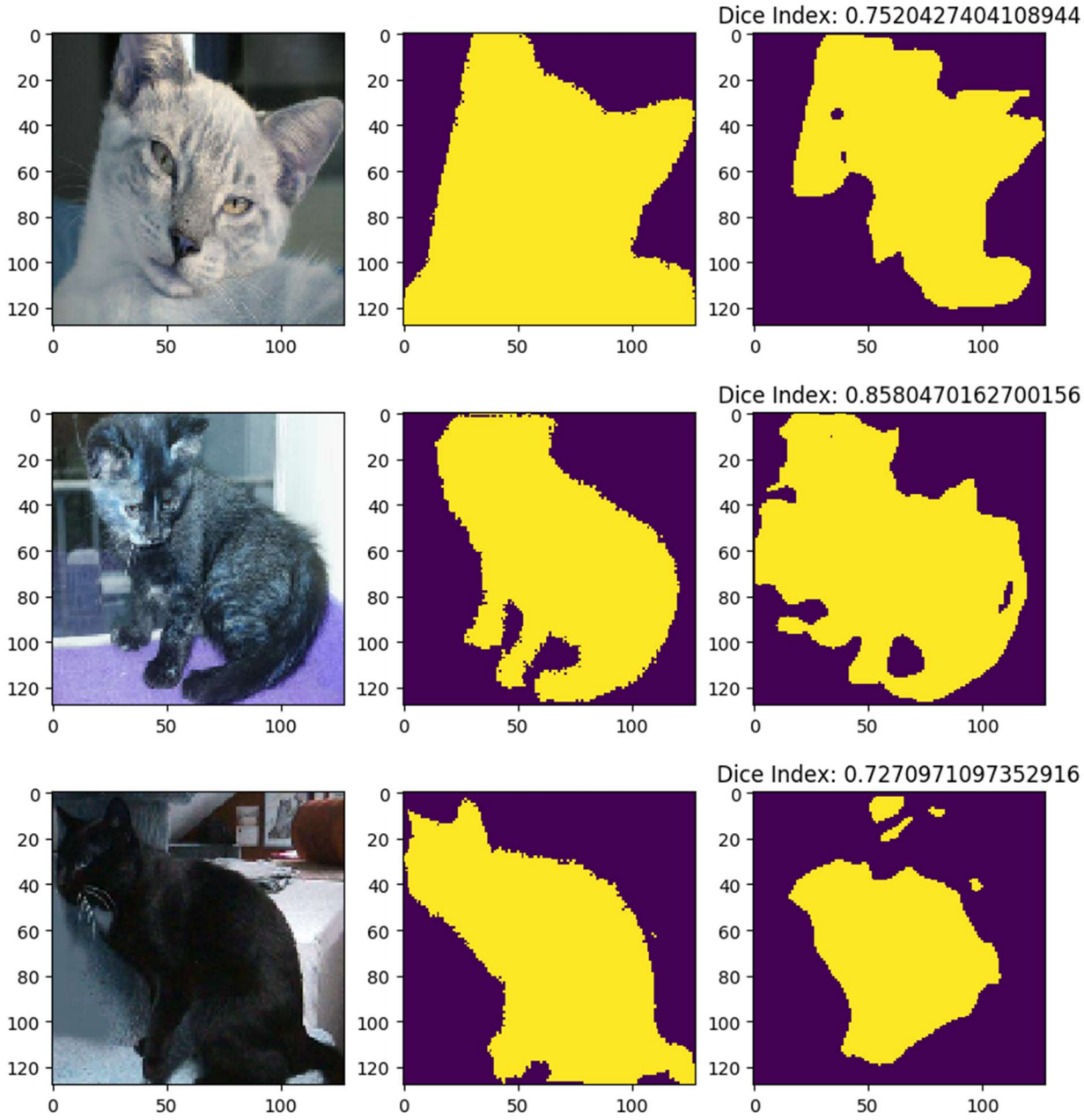


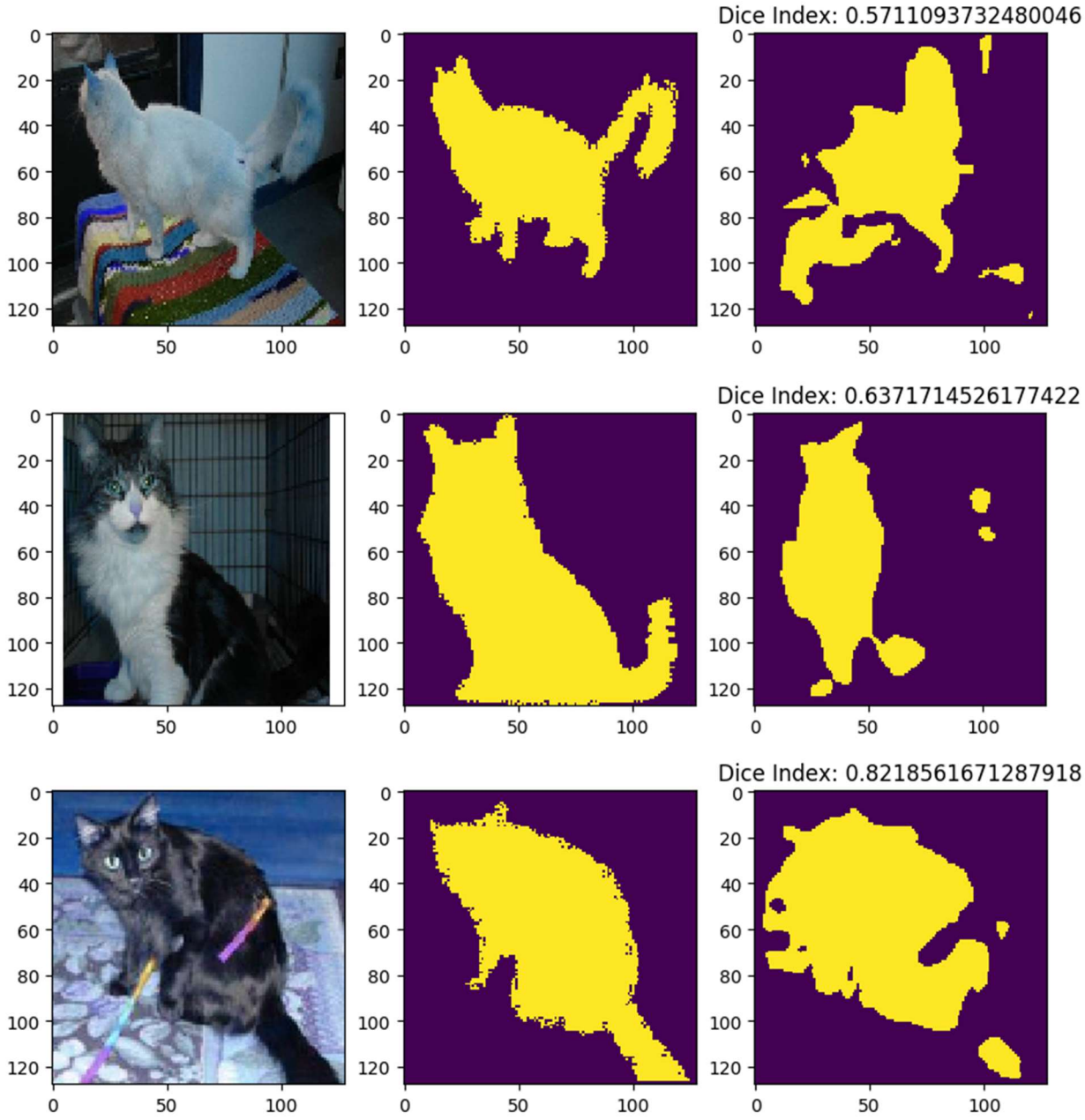
(ORIGINAL IMAGE, GROUND TRUTH MASK, PREDICTED MASK)

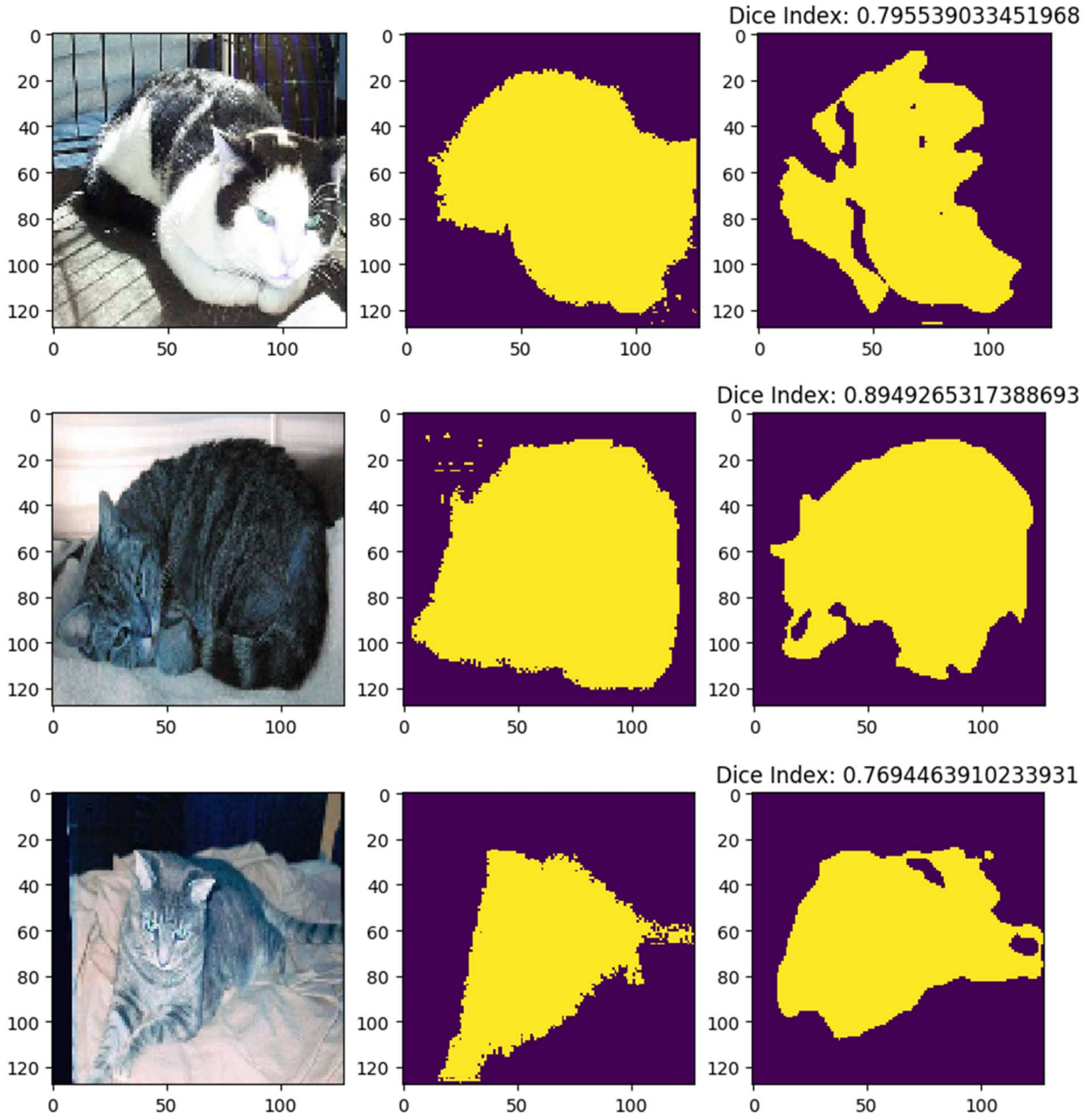


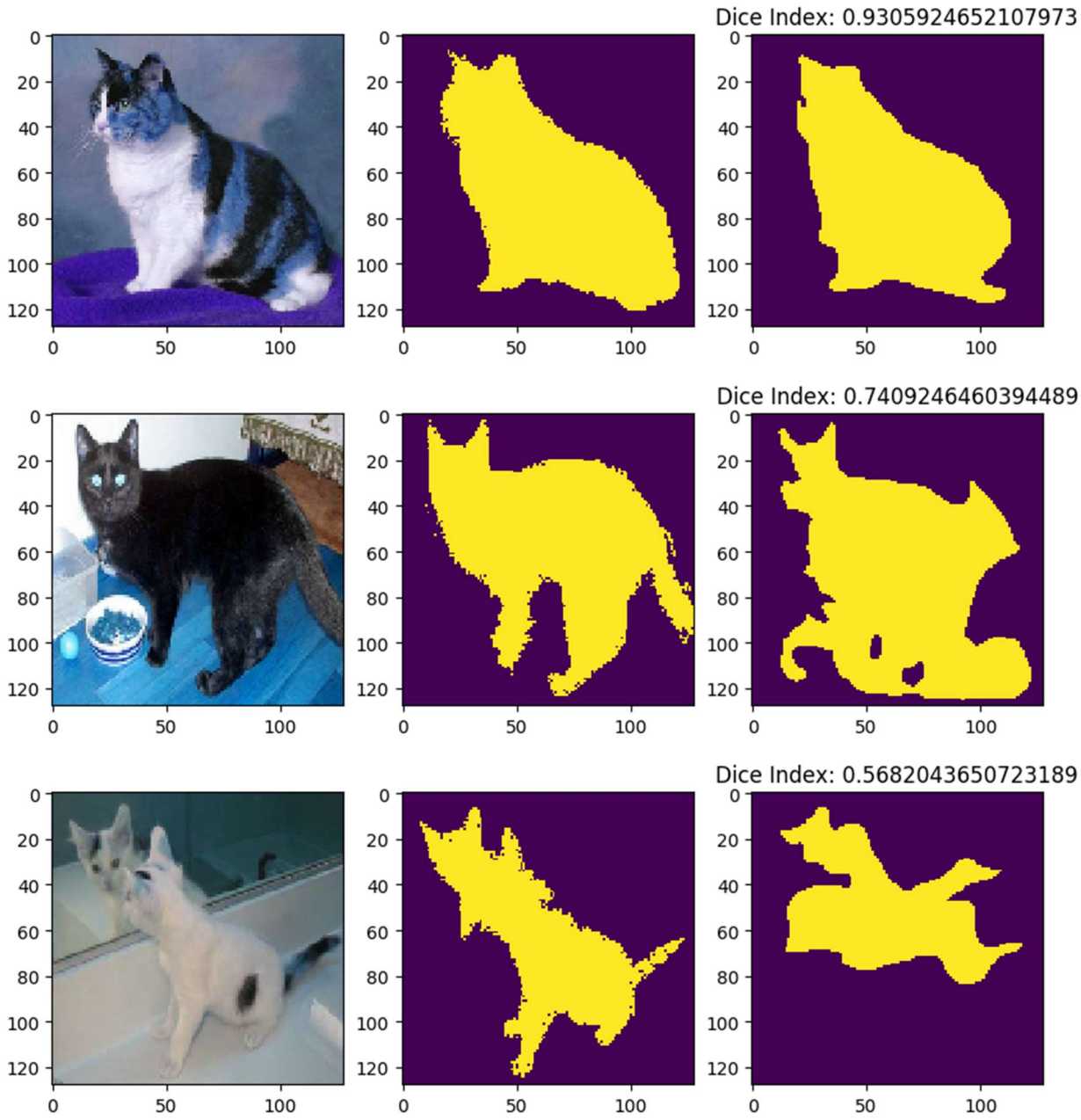

Dice Index: 0.4305156220008576
Dice Index: 0.6238345482249882
Dice Index: 0.5540021508651918


Dice Index: 0.7635537545234631
Dice Index: 0.5573597359662394
Dice Index: 0.5123316062123074


Dice Index: 0.7520427404108944
Dice Index: 0.8580470162700156
Dice Index: 0.7270971097352916


Dice Index: 0.5711093732480046
Dice Index: 0.6371714526177422
Dice Index: 0.8218561671287918


Dice Index: 0.795539033451968
Dice Index: 0.8949265317388693
Dice Index: 0.7694463910233931

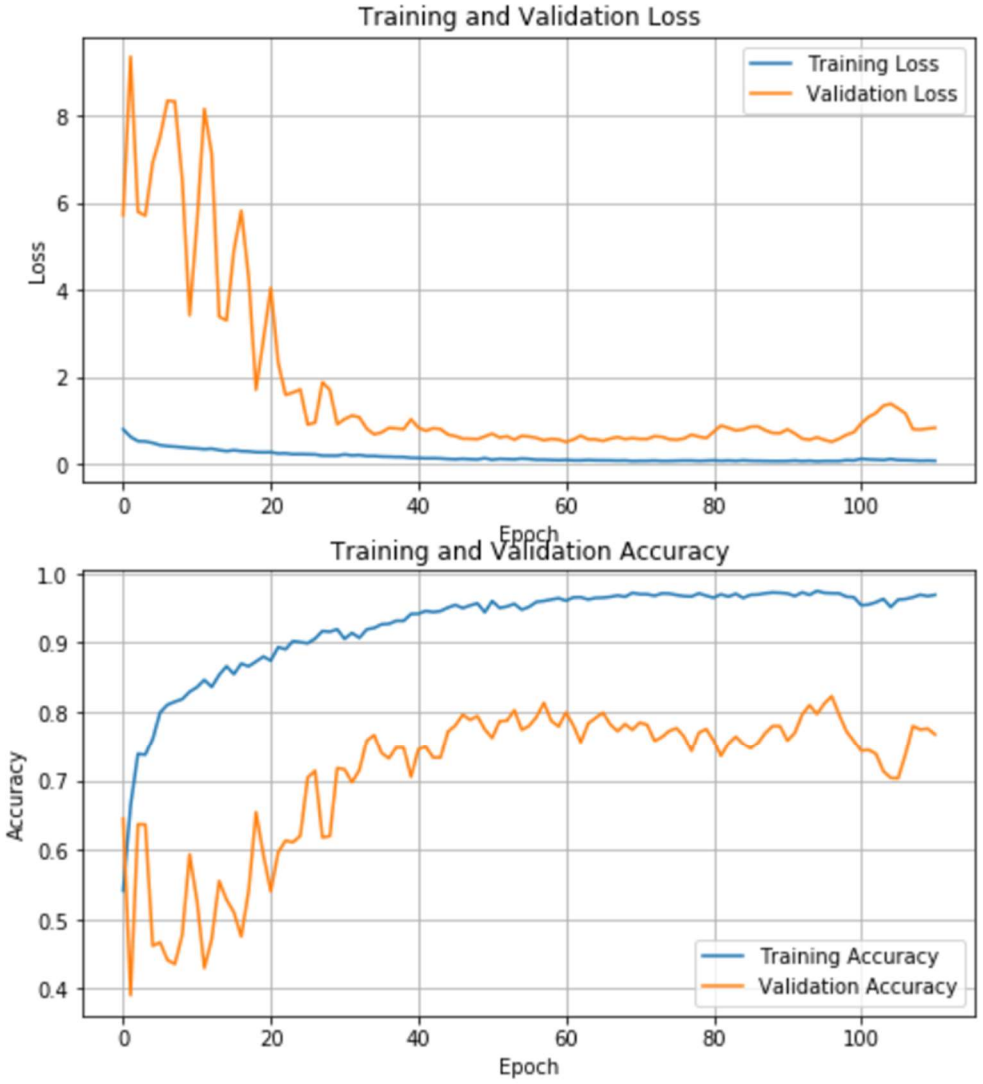


CROSS-ENTROPY MODEL TRAINING PARAMETERS

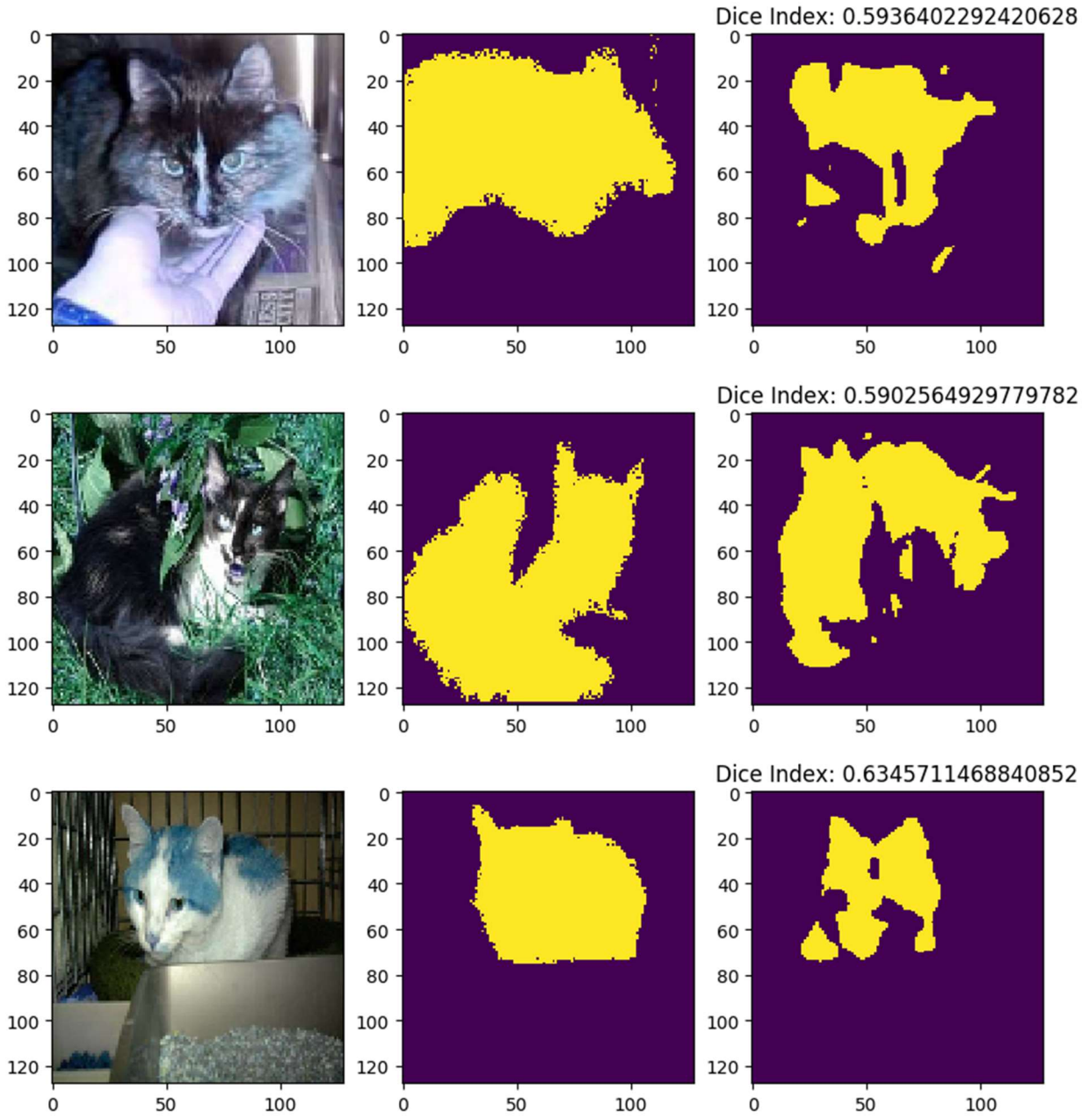
- Optimizer: Adam
- Learning Rate: 1e-3
- Loss: Cross-Entropy
- Epochs: 200

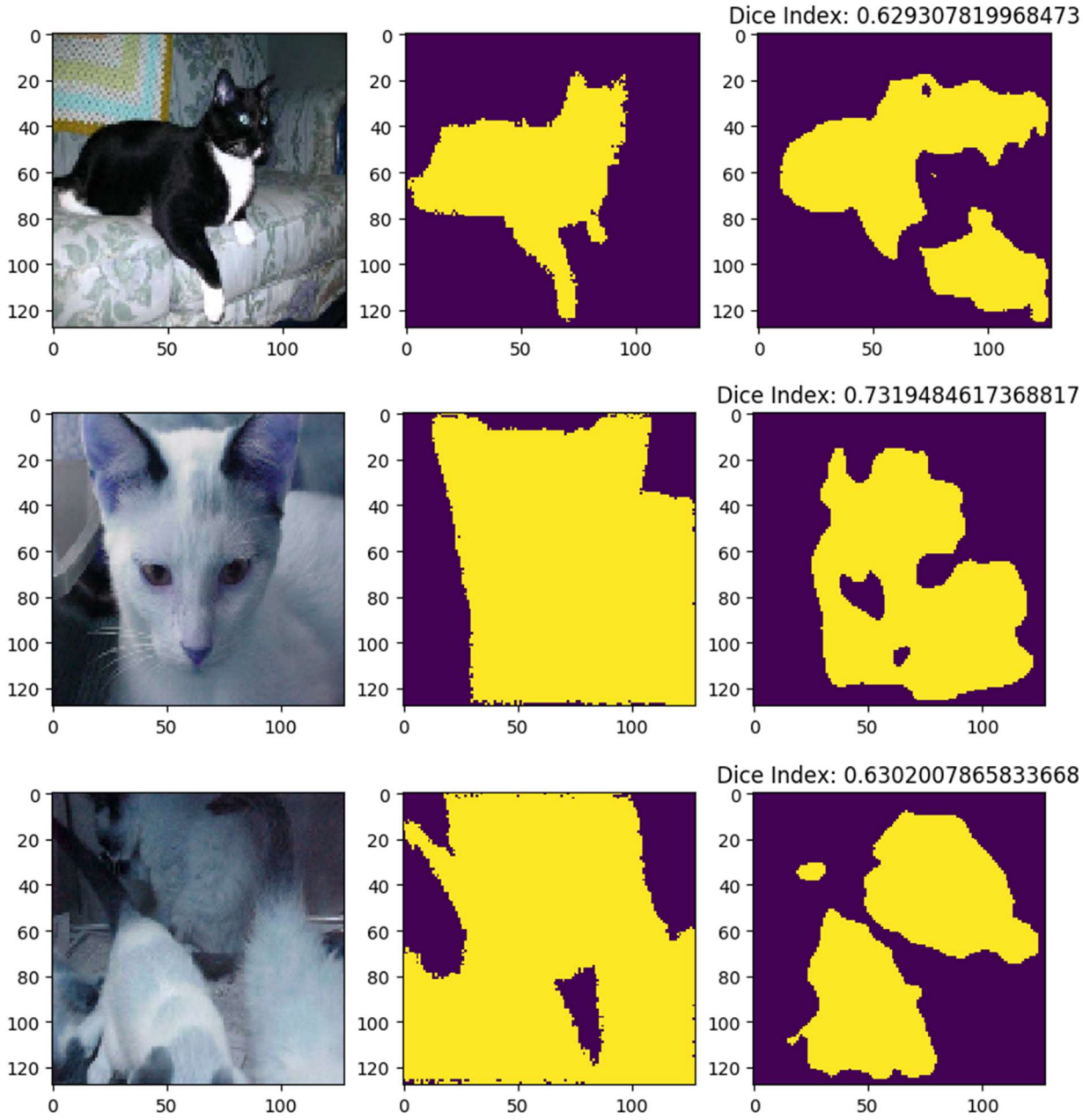
- Validation Split: 0.1
- Early Stopping: True, Patient=50

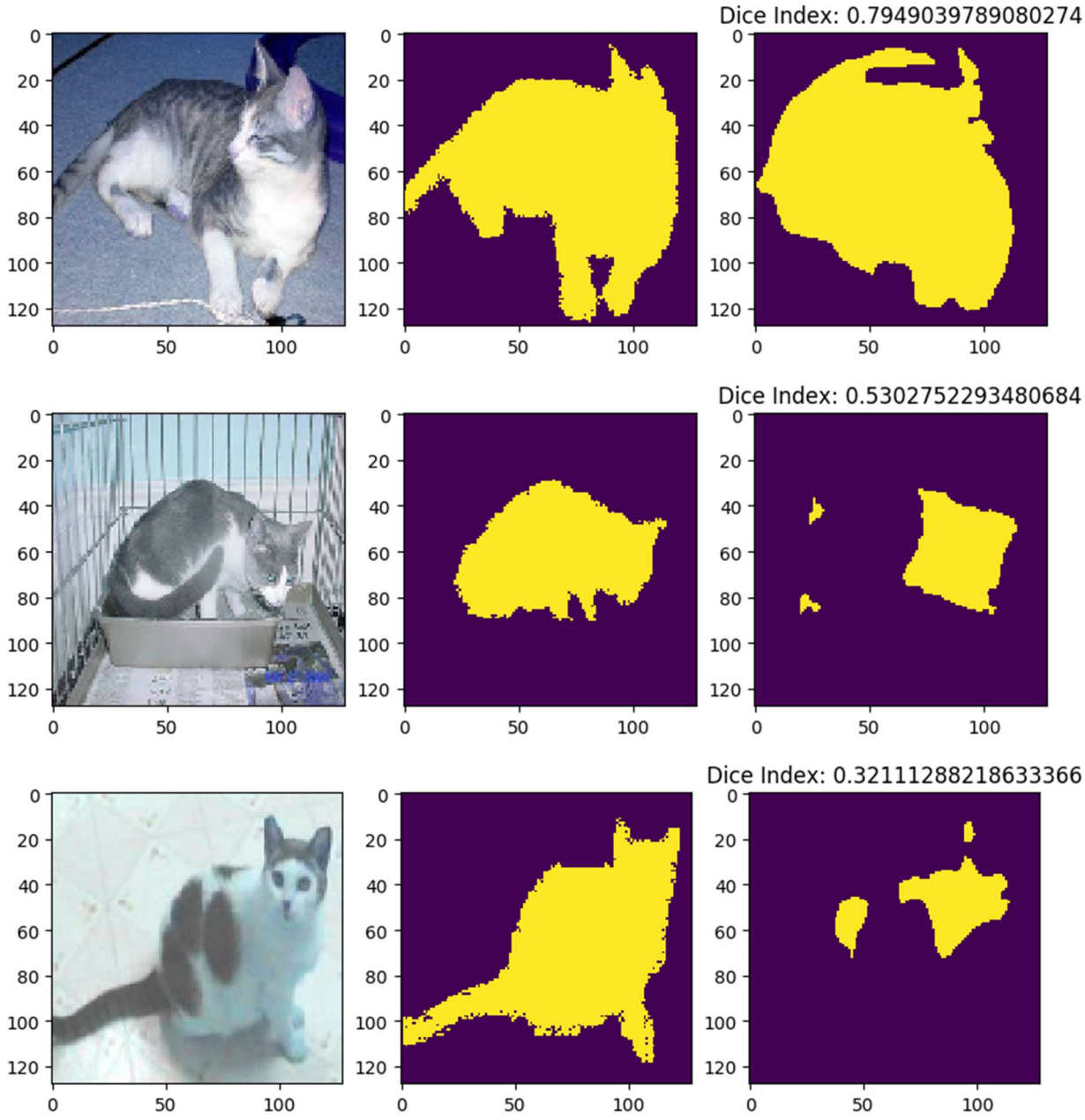
```
Log-loss (cost function):
training   (min:    0.065, max:    0.799, cur:    0.077)
validation (min:    0.510, max:    9.346, cur:    0.834)

Accuracy:
training   (min:    0.541, max:    0.975, cur:    0.969)
validation (min:    0.391, max:    0.823, cur:    0.768)
Epoch 00111: early stopping
```

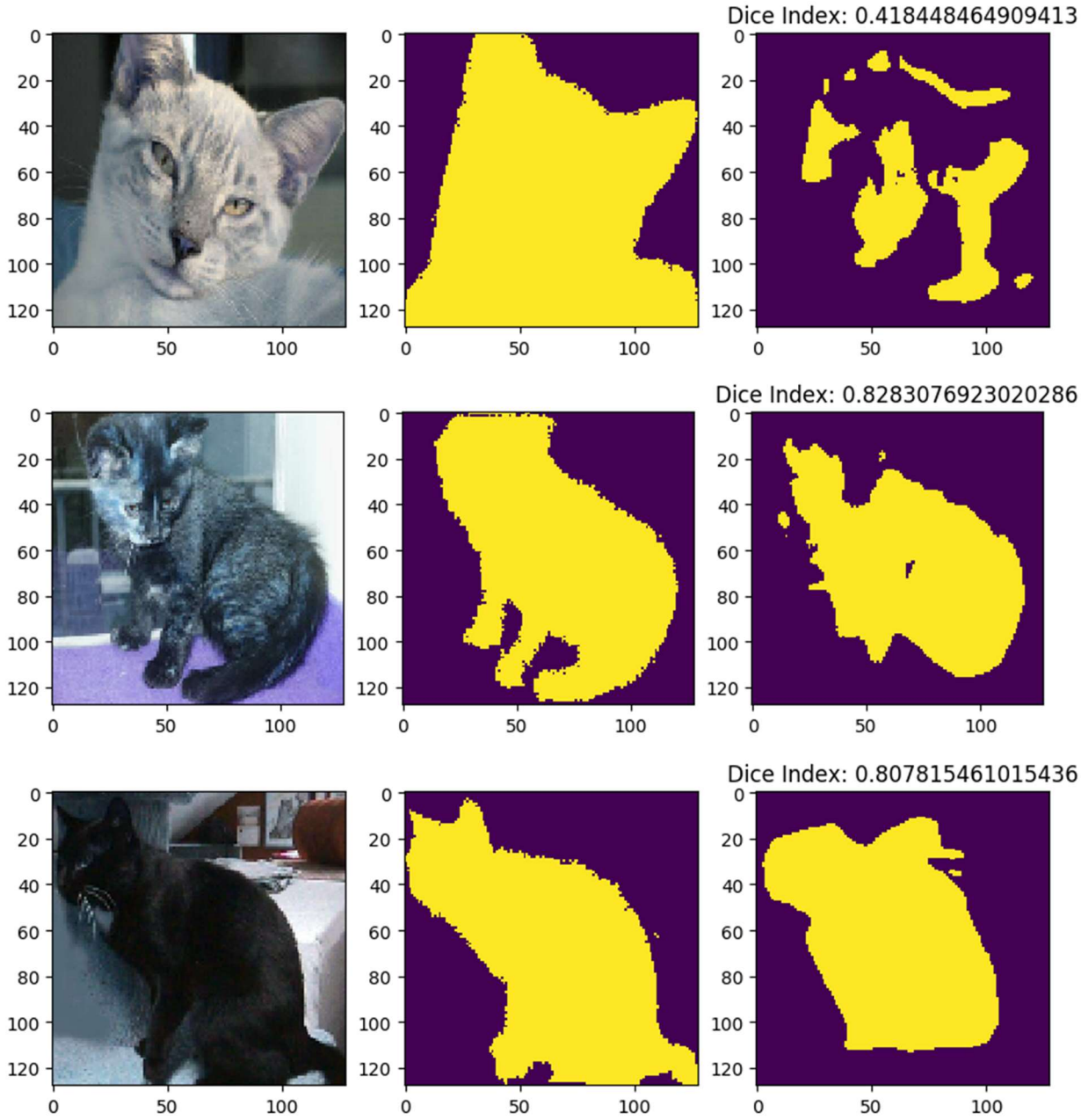


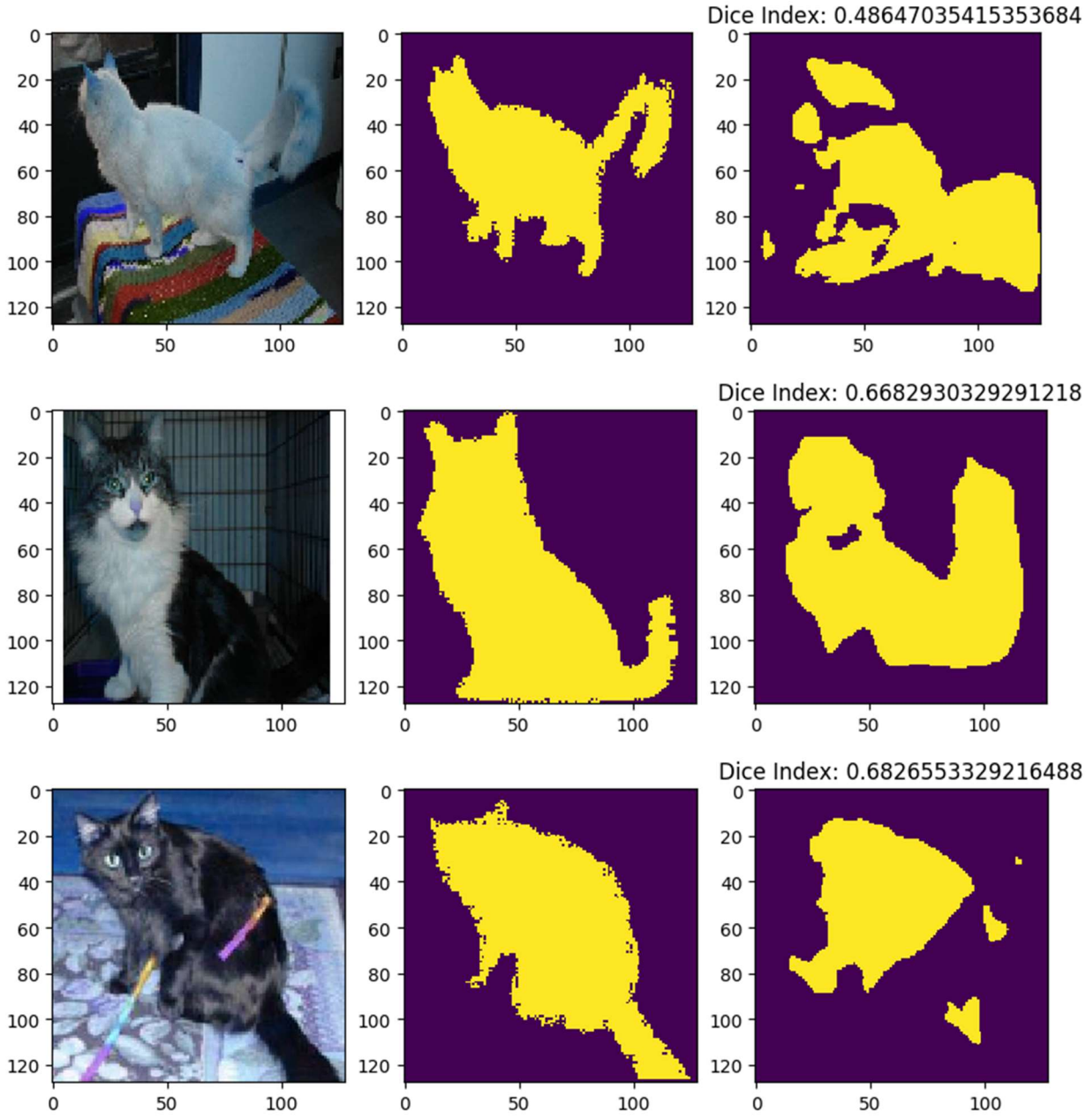
(ORIGINAL IMAGE, GROUND TRUTH MASK, PREDICTED MASK)

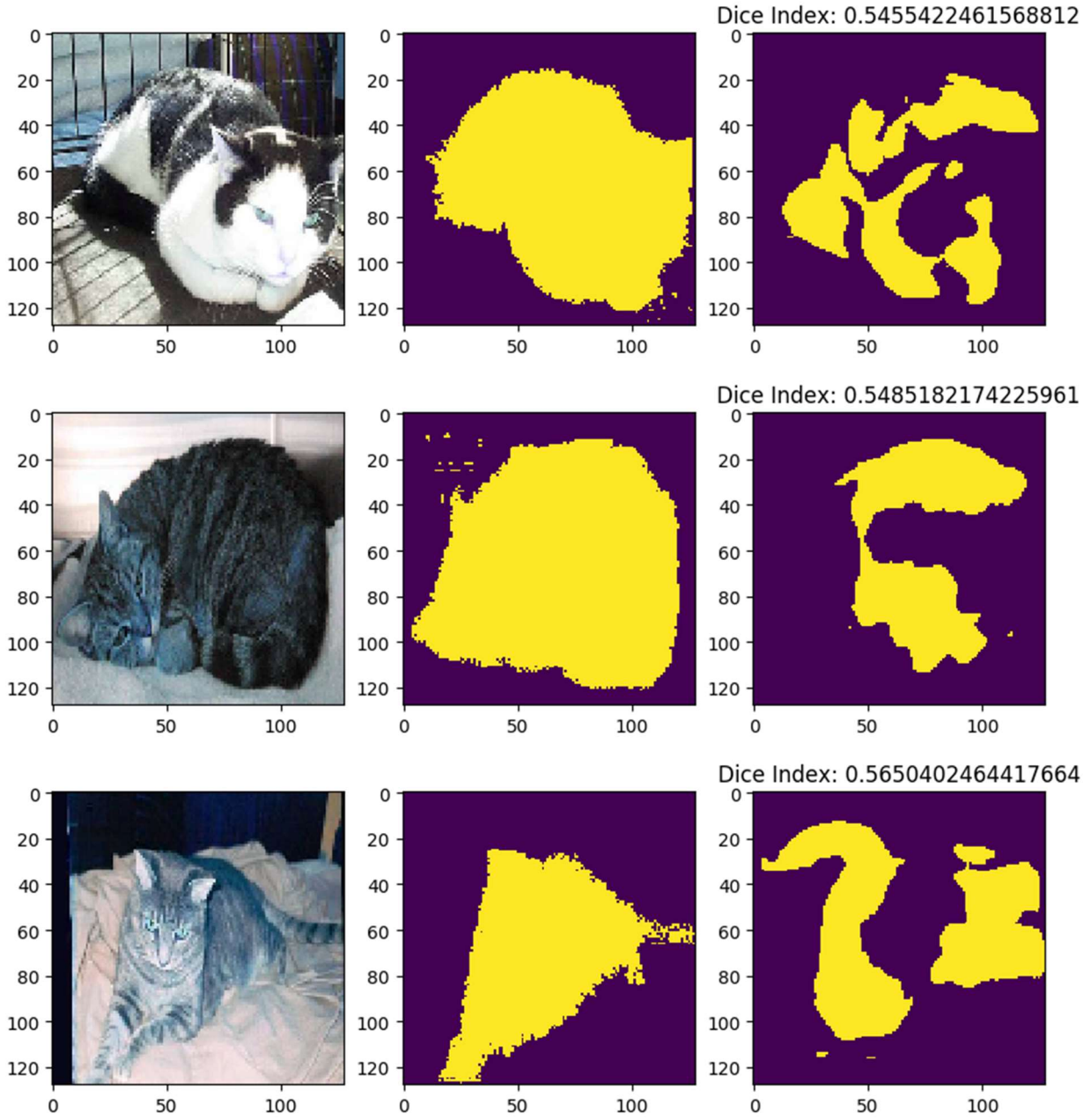


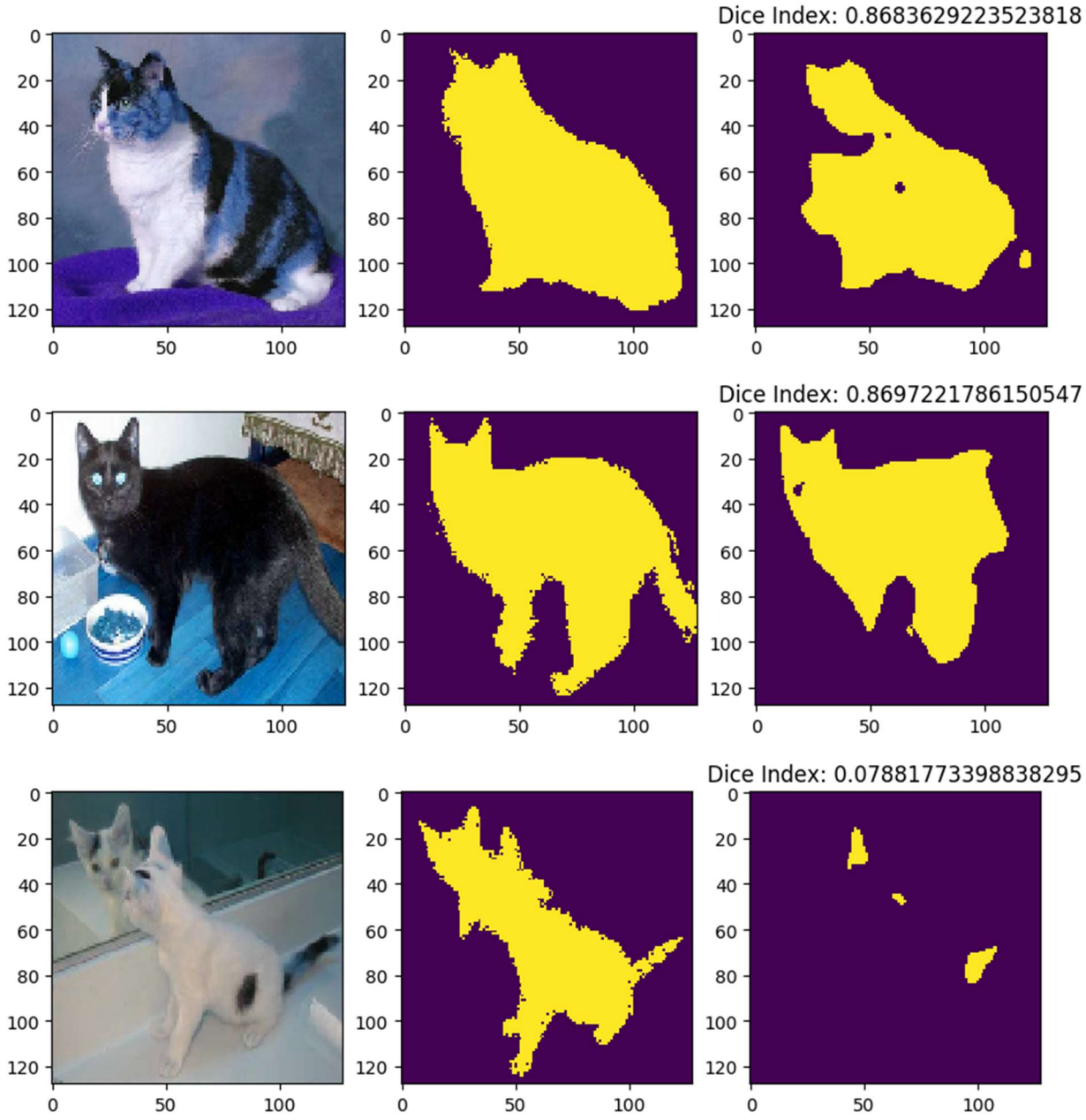

Dice Index: 0.629307819968473
Dice Index: 0.7319484617368817
Dice Index: 0.6302007865833668


Dice Index: 0.7949039789080274
Dice Index: 0.5302752293480684
Dice Index: 0.32111288218633366


Dice Index: 0.418448464909413
Dice Index: 0.8283076923020286
Dice Index: 0.807815461015436


Dice Index: 0.48647035415353684
Dice Index: 0.6682930329291218
Dice Index: 0.6826553329216488


Dice Index: 0.5455422461568812
Dice Index: 0.5485182174225961
Dice Index: 0.5650402464417664



1.2 Data Augmentation

Data augmentation was implemented by taking every image & mask combo in the training set a randomly performing a vertical flip, horizontal flip, rotation, shear, and translation. These augmented images & masks were then added to the training set along with the originals.

AUGMENT CODE

```
def load_data(input_path, label_path, augment=False):
    """
    Returns processed data from the given paths. Data is returned in two
    arrays: feature array, expected output array

    If augment==True, then the returned arrays would additionally contain
    augmented data.

    :param input_path: a relative path to all the input data
    :param label_path: a relative path to all the labels
    :param augment: whether to augment data
    """

    img_paths = glob.glob(input_path + "/*")
    mask_paths = glob.glob(label_path + "/*")
    img_paths.sort()
    mask_paths.sort()

    img_list = []
    mask_list = []

    for f in range(len(img_paths)):
        img = load_image(img_paths[f])

        # Convert mask to grayscale and reshape
        mask = load_image(mask_paths[f])
        mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
        mask = np.reshape(mask, (mask.shape[0], mask.shape[1], 1))

        # Normalize data
        img = img.astype(np.float32)
        mask = mask.astype(np.float32)
        img, mask = normalize_data(img, mask)

        img_list.append(img)
        mask_list.append(mask)

        # If augment is required, then augment the image and mask and
        # also add to the list
        if augment:
            img2, mask2 = augment_data(img, mask.squeeze())
            img_list.append(img2)
            mask_list.append(np.reshape(mask2, (mask2.shape[0],
mask2.shape[1], 1)))
```

```
    return np.array(img_list), np.array(mask_list)
```

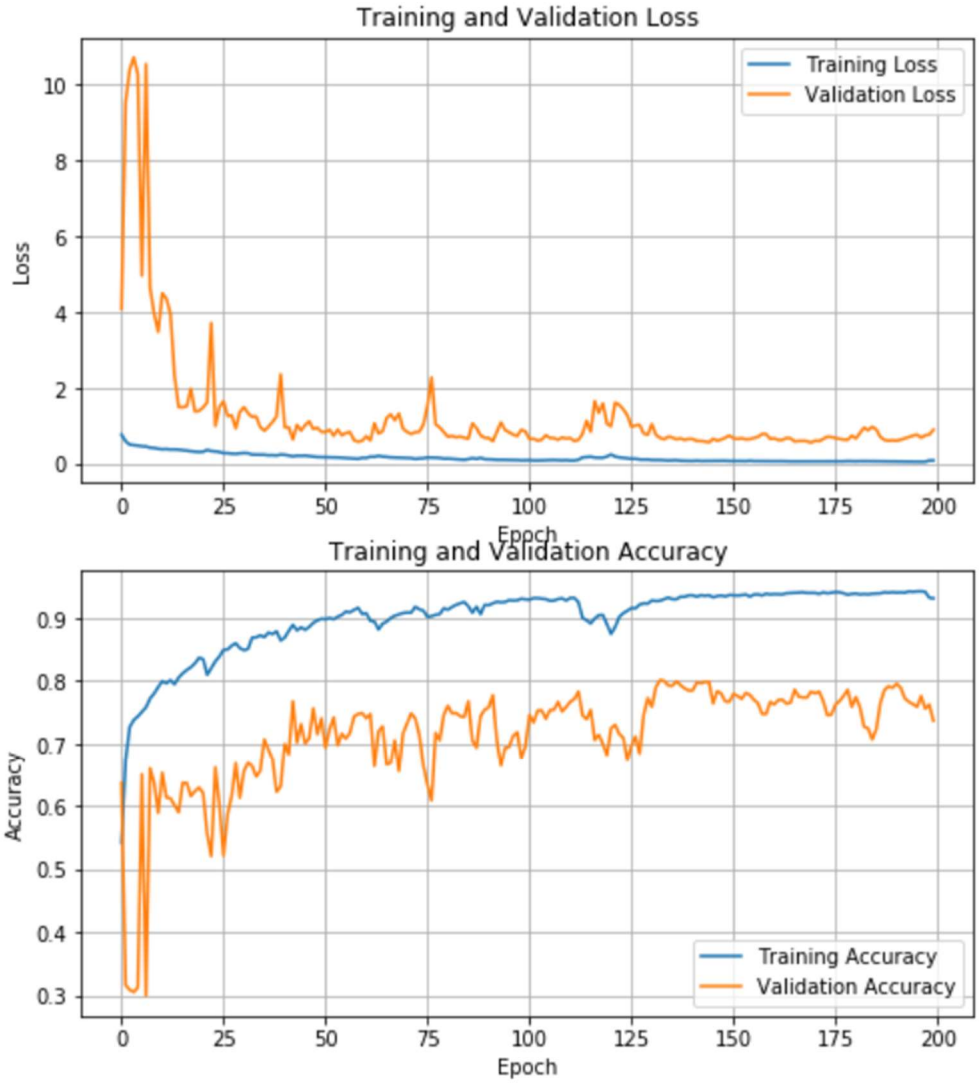
```
def augment_data(img, mask):
    """
    Performs multiple randomly generated augmentations to the image and
its mask.

    Augmentations performed:
        - Flips (x-axis and y-axis)
        - Rotations
        - Zooms
        - Sheers

    :param img: the image
    :param mask: the mask
    """
    # 50% change of flipping on x-axis
    if random.random() >= 0.5:
        img = cv2.flip(img, 0)
        mask = cv2.flip(mask, 0)

    # 50% change of flipping on y-axis
    if random.random() >= 0.5:
        img = cv2.flip(img, 1)
        mask = cv2.flip(mask, 1)

    # Rotate randomly
    rot_m = cv2.getRotationMatrix2D((img.shape[0] / 2, img.shape[1] / 2),
random.uniform(0, 360), 1)
    img = cv2.warpAffine(img, rot_m, (img.shape[0], img.shape[1]))
    mask = cv2.warpAffine(mask, rot_m, (img.shape[0], img.shape[1]))

    # Warp Affine (shear)
    pt1 = 5 + 20 * np.random.uniform() - 20 / 2
    pt2 = 50 + 20 * np.random.uniform() - 20 / 2
    pts1 = np.float32([[5, 5], [50, 5], [5, 50]])
    pts2 = np.float32([[pt1, 5], [pt2, pt1], [5, pt2]])

    shear_m = cv2.getAffineTransform(pts1, pts2)
    img = cv2.warpAffine(img, shear_m, (img.shape[0], img.shape[1]))
    mask = cv2.warpAffine(mask, shear_m, (img.shape[0], img.shape[1]))

    # Translation
    x = 10 * np.random.uniform() - 5 / 2
    y = 10 * np.random.uniform() - 5 / 2
```

```
    trans_m = np.float32([[1, 0, x], [0, 1, y]])
    img = cv2.warpAffine(img, trans_m, (img.shape[0], img.shape[1]))
    mask = cv2.warpAffine(mask, trans_m, (img.shape[0], img.shape[1]))

    return img, mask
```

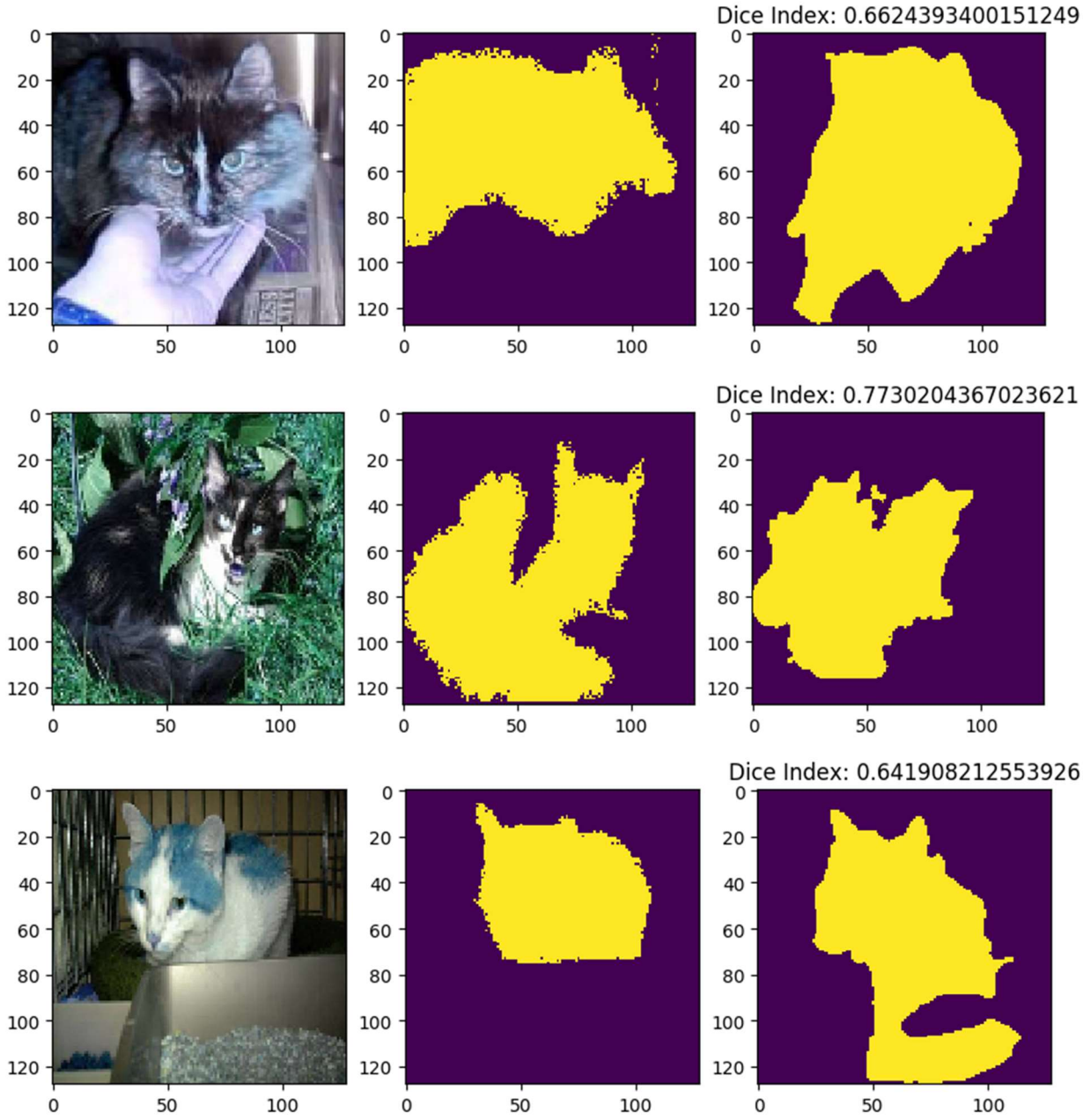
DATA AUGMENTATION MODEL TRAINING PARAMETERS

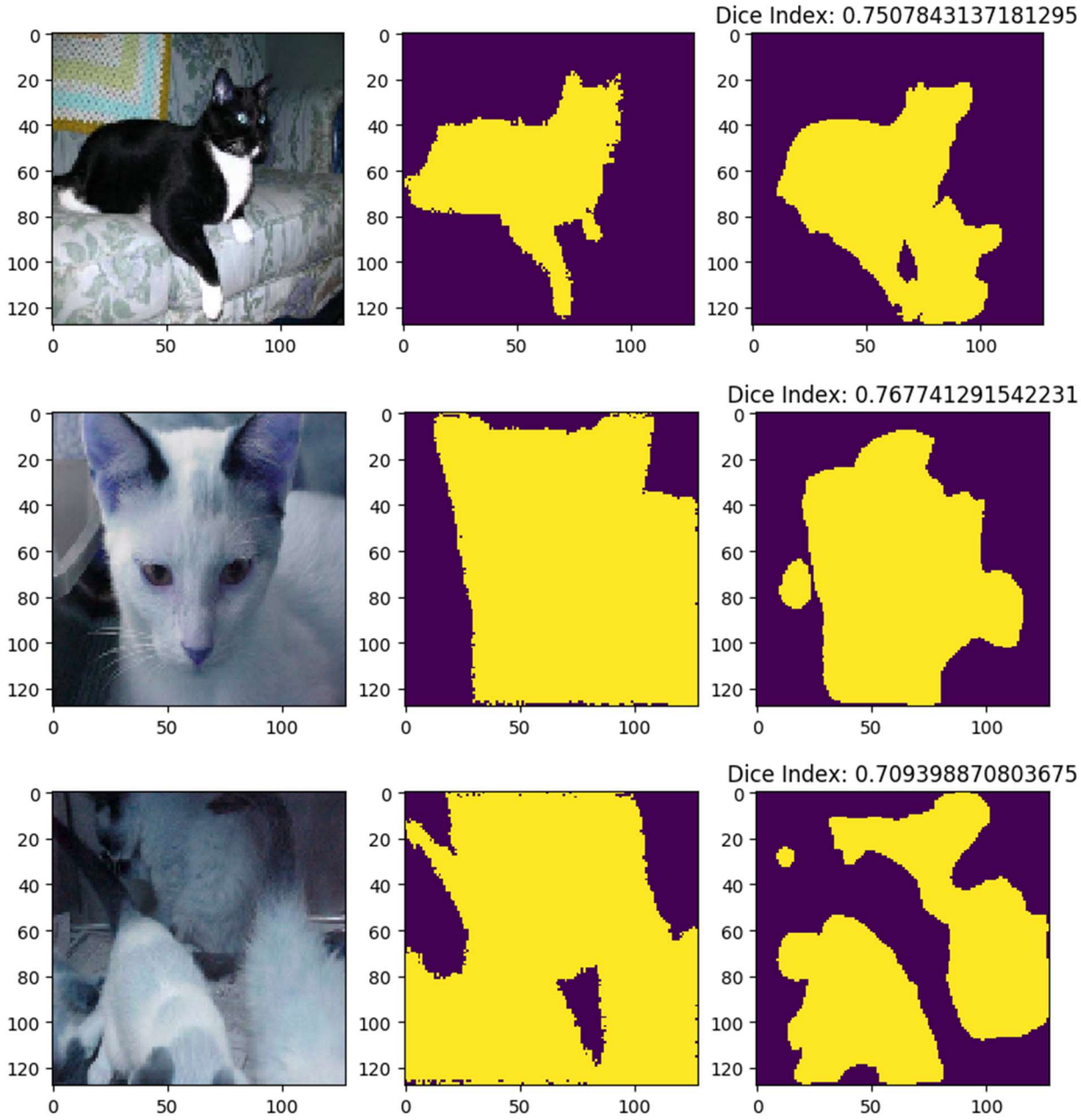
- Optimizer: Adam
- Learning Rate: 1e-3
- Loss: Cross-Entropy
- Epochs: 200
- Validation Split: 0.1
- Early Stopping: True, Patient=50

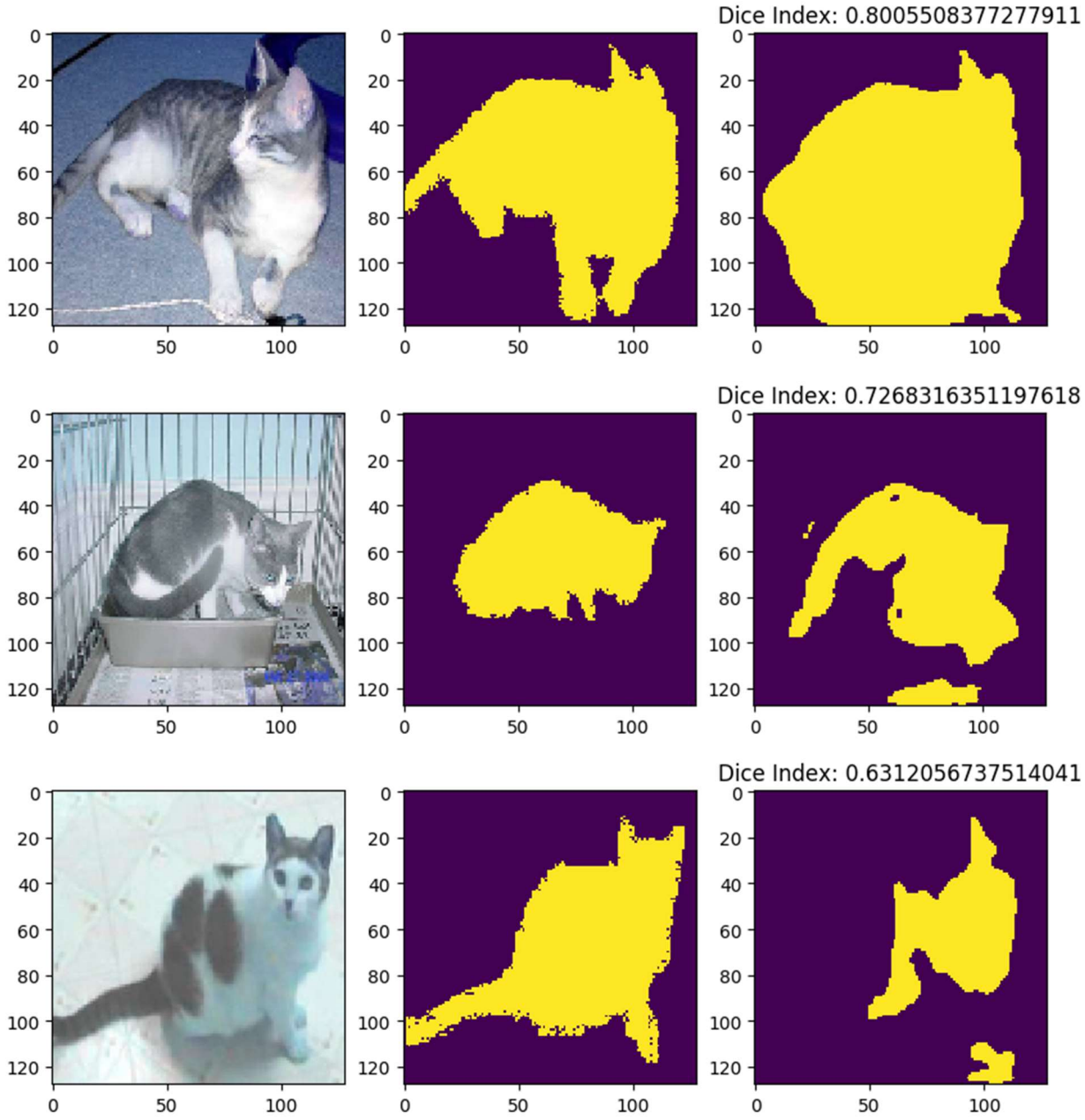
```
Log-loss (cost function):
training   (min:    0.053, max:    0.771, cur:    0.088)
validation (min:    0.559, max:   10.719, cur:    0.898)

Accuracy:
training   (min:    0.542, max:    0.943, cur:    0.932)
validation (min:    0.300, max:    0.802, cur:    0.737)
```

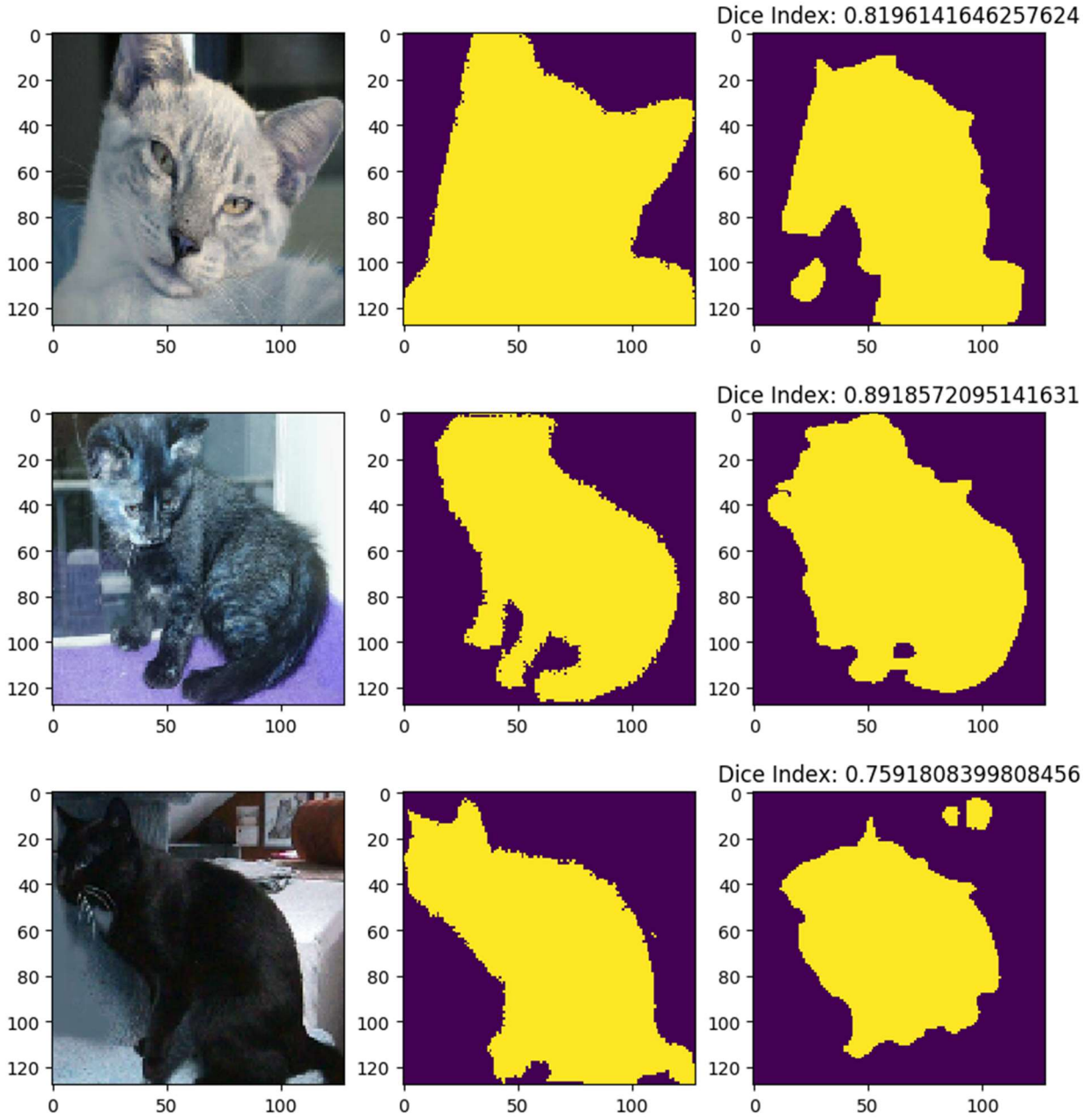


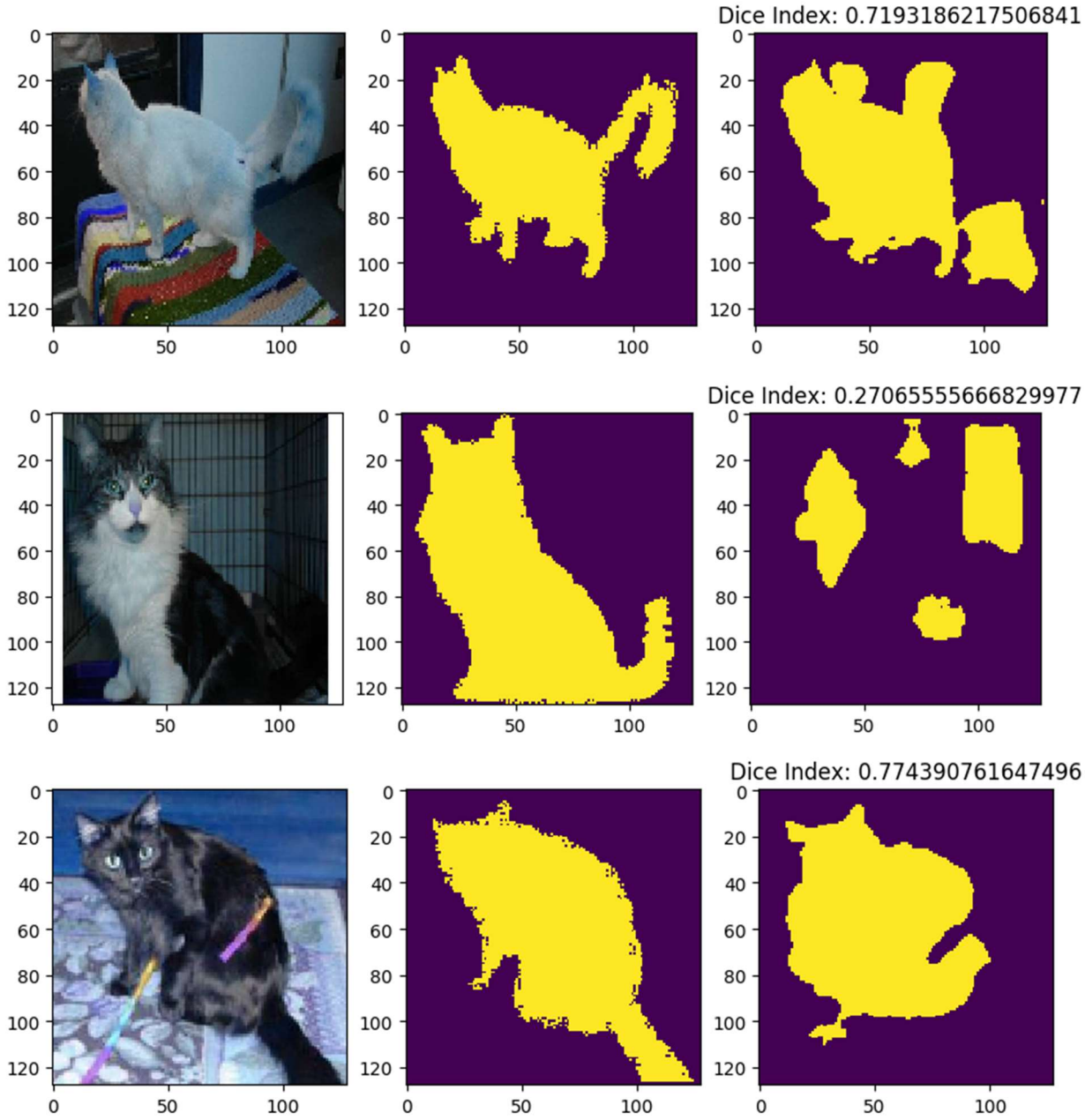
(ORIGINAL IMAGE, GROUND TRUTH MASK, PREDICTED MASK)

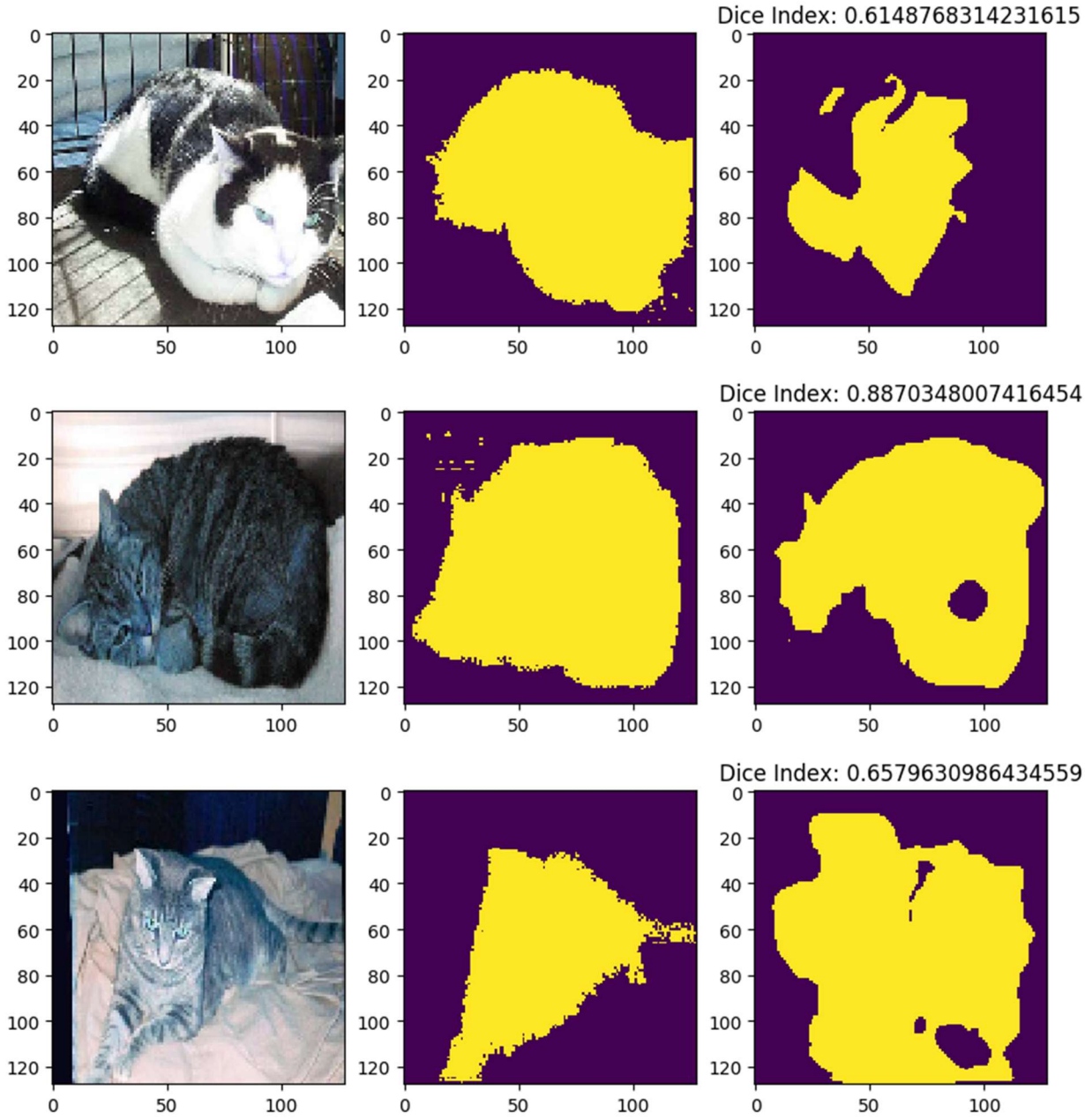

Dice Index: 0.6624393400151249
Dice Index: 0.7730204367023621
Dice Index: 0.641908212553926

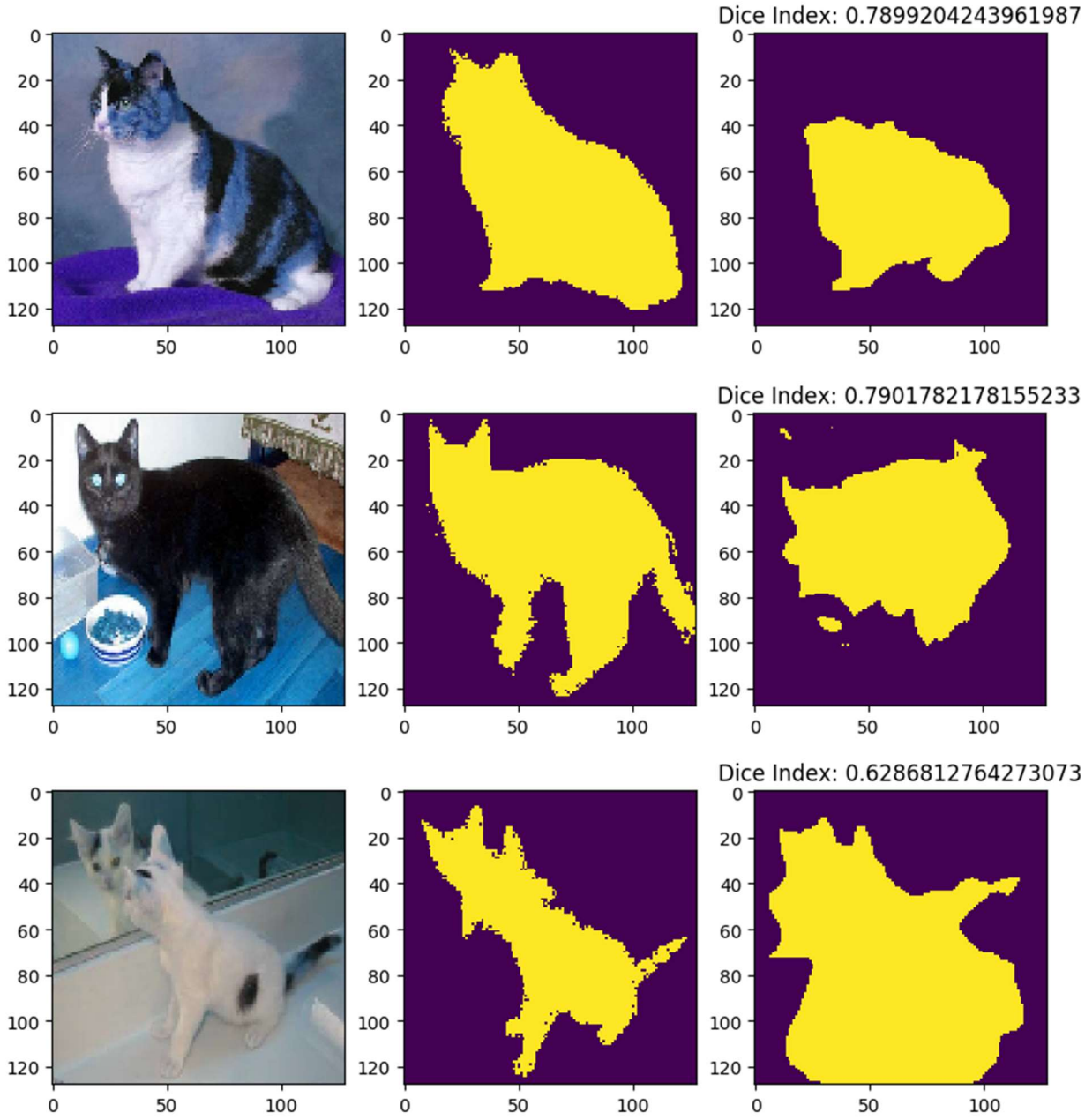

Dice Index: 0.7507843137181295
Dice Index: 0.767741291542231
Dice Index: 0.709398870803675


Dice Index: 0.8005508377277911
Dice Index: 0.7268316351197618
Dice Index: 0.6312056737514041


Dice Index: 0.8196141646257624
Dice Index: 0.8918572095141631
Dice Index: 0.7591808399808456


Dice Index: 0.7193186217506841
Dice Index: 0.27065555666829977
Dice Index: 0.774390761647496


Dice Index: 0.6148768314231615
Dice Index: 0.8870348007416454
Dice Index: 0.6579630986434559



1.3 Transfer Learning

We first started by training our model on horses dataset found online containing 308 training images and their corresponding binary segmentation masks. It should be noted at this point, we had improved our methods of training by adding a validation split and early stopping. We also improved the visualizations of our training statistics.

TRAINING CODE

```
def train_for_transfer_learning(input_path, output_path, model_name,
trained_model_name, fine_tune_at, augment=False,
                                epochs=10, valid_split=0,
early_stop=False):
    """
    Returns a trained model and the training history. Uses the weights of
an already trained model
    to fine-tune the new model.

    :param input_path: path to all the input data (ie. train_X)
    :param output_path:  path to all the output data (ie. train_y)
    :param model_name: name of the new model (weights will be saved under
this name)
    :param trained_model_name: name of an already trained model to get the
weights from (must exist in main directory)
    :param fine_tune_at: from which layer to start fine tuning at (all
layers before this will be frozen)
    :param augment: whether or not the augment the data
    :param epochs: number of epochs to run for
    :param valid_split: percentage of data that should be used for
validation
    :param early_stop: whether or not to stop early if the validation and
training curves diverge too much
    """
    model = unet_model((IMG_WIDTH, IMG_HEIGHT, 3))
    model.load_weights(MAIN_DIR_PATH + "/" + trained_model_name + ".hdf5")

    # Fine tune up to a specific layer
    for layer in model.layers[:fine_tune_at]:
        layer.trainable = False

    model.compile(optimizer=Adam(lr=1e-3), loss='binary_crossentropy',
metrics=['accuracy'])

    # Train like normal
    new_model, history = train_model(input_path, output_path, model_name,
augment, epochs, valid_split, early_stop, model)

    return new_model, history
```

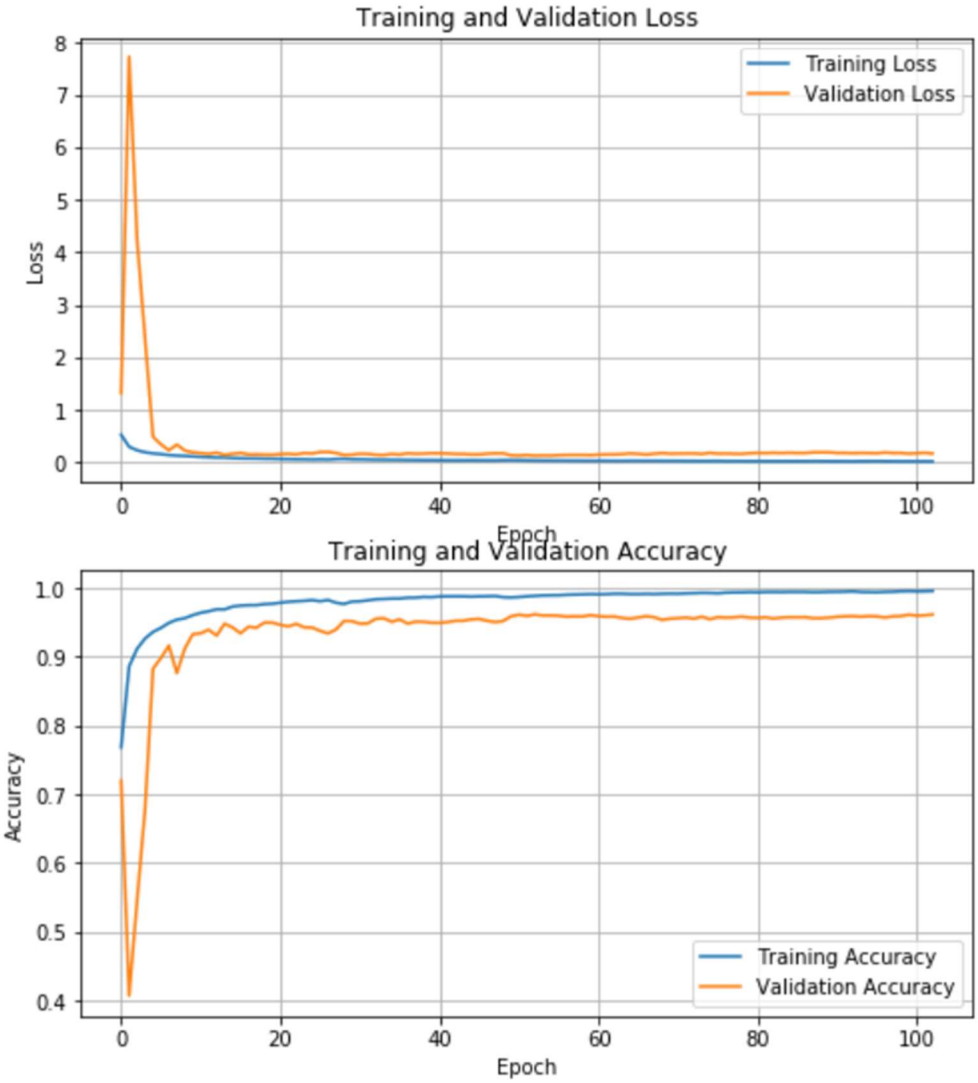
HORSE MODEL TRANSFER LEARNING TRAINING PARAMETERS

- Optimizer: Adam
- Learning Rate: 1e-3

- Loss: Cross-Entropy
- Validation Split: 0.1
- Early Stopping: True, Patient=50
- Epochs: 200

FINAL OUTPUT

```
Log-loss (cost function):
training   (min:    0.012, max:    0.520, cur:    0.012)
validation (min:    0.122, max:    7.724, cur:    0.163)

Accuracy:
training   (min:    0.768, max:    0.995, cur:    0.995)
validation (min:    0.407, max:    0.961, cur:    0.961)
Epoch 00103: early stopping
```



Once we finished training the horse model, we load the weights onto our cat model. We set the first 15 layers of the network to be non-trainable. We chose 15 layers because the first few layers learn very simple and generic features which generalize to almost all types of images. As you go higher up, the features are increasingly more specific to the dataset on which the model was trained. We fine-tuned this number until we found the one which gave us the best result.

CAT MODEL SUMMARY

```
Model: "model_4"
_________________________________________________________________________________
_________________________
Layer (type)                 Output Shape              Param #     Connected
to
=================================================================================
=========================
input1 (InputLayer)          (None, 128, 128, 3)       0
_________________________________________________________________________________
_________________________
conv1 (Conv2D)               (None, 128, 128, 64) 1792
input1[0][0]
_________________________________________________________________________________
_________________________
batch_normalization_55 (BatchNo (None, 128, 128, 64) 256
conv1[0][0]
_________________________________________________________________________________
_________________________
activation_55 (Activation)   (None, 128, 128, 64) 0
batch_normalization_55[0][0]
_________________________________________________________________________________
_________________________
conv2 (Conv2D)               (None, 128, 128, 64) 36928
activation_55[0][0]
_________________________________________________________________________________
_________________________
batch_normalization_56 (BatchNo (None, 128, 128, 64) 256
conv2[0][0]
_________________________________________________________________________________
_________________________
activation_56 (Activation)   (None, 128, 128, 64) 0
batch_normalization_56[0][0]
_________________________________________________________________________________
_________________________
maxpool1 (MaxPooling2D)      (None, 64, 64, 64)    0
activation_56[0][0]
_________________________________________________________________________________
_________________________
conv3 (Conv2D)               (None, 64, 64, 128)  73856
maxpool1[0][0]
```

```
_____________________________
batch_normalization_57 (BatchNo (None, 64, 64, 128)  512
conv3[0][0]
_____________________________
activation_57 (Activation)      (None, 64, 64, 128)  0
batch_normalization_57[0][0]
_____________________________
conv4 (Conv2D)                  (None, 64, 64, 128)  147584
activation_57[0][0]
_____________________________
batch_normalization_58 (BatchNo (None, 64, 64, 128)  512
conv4[0][0]
_____________________________
activation_58 (Activation)      (None, 64, 64, 128)  0
batch_normalization_58[0][0]
_____________________________
maxpool2 (MaxPooling2D)         (None, 32, 32, 128)  0
activation_58[0][0]
_____________________________
conv5 (Conv2D)                  (None, 32, 32, 256)  295168
maxpool2[0][0]
_____________________________
batch_normalization_59 (BatchNo (None, 32, 32, 256)  1024
conv5[0][0]
_____________________________
activation_59 (Activation)      (None, 32, 32, 256)  0
batch_normalization_59[0][0]
_____________________________
conv6 (Conv2D)                  (None, 32, 32, 256)  590080
activation_59[0][0]
_____________________________
batch_normalization_60 (BatchNo (None, 32, 32, 256)  1024
conv6[0][0]
_____________________________
activation_60 (Activation)      (None, 32, 32, 256)  0
batch_normalization_60[0][0]
_____________________________
maxpool3 (MaxPooling2D)         (None, 16, 16, 256)  0
activation_60[0][0]
```

```
__________________________________________________________________________________________________
____________________________
conv7 (Conv2D)                  (None, 16, 16, 512)  1180160
maxpool3[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_61 (BatchNo (None, 16, 16, 512)  2048
conv7[0][0]
__________________________________________________________________________________________________
____________________________
activation_61 (Activation)      (None, 16, 16, 512)  0
batch_normalization_61[0][0]
__________________________________________________________________________________________________
____________________________
conv8 (Conv2D)                  (None, 16, 16, 512)  2359808
activation_61[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_62 (BatchNo (None, 16, 16, 512)  2048
conv8[0][0]
__________________________________________________________________________________________________
____________________________
activation_62 (Activation)      (None, 16, 16, 512)  0
batch_normalization_62[0][0]
__________________________________________________________________________________________________
____________________________
maxpool4 (MaxPooling2D)         (None, 8, 8, 512)    0
activation_62[0][0]
__________________________________________________________________________________________________
____________________________
conv9 (Conv2D)                  (None, 8, 8, 1024)   4719616
maxpool4[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_63 (BatchNo (None, 8, 8, 1024)   4096
conv9[0][0]
__________________________________________________________________________________________________
____________________________
activation_63 (Activation)      (None, 8, 8, 1024)   0
batch_normalization_63[0][0]
__________________________________________________________________________________________________
____________________________
conv10 (Conv2D)                 (None, 8, 8, 1024)   9438208
activation_63[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_64 (BatchNo (None, 8, 8, 1024)   4096
conv10[0][0]
__________________________________________________________________________________________________
____________________________
activation_64 (Activation)      (None, 8, 8, 1024)   0
batch_normalization_64[0][0]
```

```
__________________________________________________________________________________________________
____________________
upconv1 (UpSampling2D)          (None, 16, 16, 1024) 0
activation_64[0][0]
__________________________________________________________________________________________________
____________________
concat1 (Concatenate)           (None, 16, 16, 1536) 0
activation_62[0][0]

upconv1[0][0]
__________________________________________________________________________________________________
____________________
conv11 (Conv2D)                 (None, 16, 16, 512)  7078400
concat1[0][0]
__________________________________________________________________________________________________
____________________
batch_normalization_65 (BatchNo (None, 16, 16, 512)  2048
conv11[0][0]
__________________________________________________________________________________________________
____________________
activation_65 (Activation)      (None, 16, 16, 512)  0
batch_normalization_65[0][0]
__________________________________________________________________________________________________
____________________
conv12 (Conv2D)                 (None, 16, 16, 512)  2359808
activation_65[0][0]
__________________________________________________________________________________________________
____________________
batch_normalization_66 (BatchNo (None, 16, 16, 512)  2048
conv12[0][0]
__________________________________________________________________________________________________
____________________
activation_66 (Activation)      (None, 16, 16, 512)  0
batch_normalization_66[0][0]
__________________________________________________________________________________________________
____________________
upconv2 (UpSampling2D)          (None, 32, 32, 512)  0
activation_66[0][0]
__________________________________________________________________________________________________
____________________
concat2 (Concatenate)           (None, 32, 32, 768)  0
activation_60[0][0]

upconv2[0][0]
__________________________________________________________________________________________________
____________________
conv13 (Conv2D)                 (None, 32, 32, 256)  1769728
concat2[0][0]
__________________________________________________________________________________________________
____________________
batch_normalization_67 (BatchNo (None, 32, 32, 256)  1024
conv13[0][0]
```

```
__________________________________________________________________________________________________
____________________________
activation_67 (Activation)      (None, 32, 32, 256)  0
batch_normalization_67[0][0]
__________________________________________________________________________________________________
____________________________
conv14 (Conv2D)                 (None, 32, 32, 256)  590080
activation_67[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_68 (BatchNo (None, 32, 32, 256)  1024
conv14[0][0]
__________________________________________________________________________________________________
____________________________
activation_68 (Activation)      (None, 32, 32, 256)  0
batch_normalization_68[0][0]
__________________________________________________________________________________________________
____________________________
upconv3 (UpSampling2D)          (None, 64, 64, 256)  0
activation_68[0][0]
__________________________________________________________________________________________________
____________________________
concat3 (Concatenate)           (None, 64, 64, 384)  0
activation_58[0][0]

upconv3[0][0]
__________________________________________________________________________________________________
____________________________
conv15 (Conv2D)                 (None, 64, 64, 128)  442496
concat3[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_69 (BatchNo (None, 64, 64, 128)  512
conv15[0][0]
__________________________________________________________________________________________________
____________________________
activation_69 (Activation)      (None, 64, 64, 128)  0
batch_normalization_69[0][0]
__________________________________________________________________________________________________
____________________________
conv16 (Conv2D)                 (None, 64, 64, 128)  147584
activation_69[0][0]
__________________________________________________________________________________________________
____________________________
batch_normalization_70 (BatchNo (None, 64, 64, 128)  512
conv16[0][0]
__________________________________________________________________________________________________
____________________________
activation_70 (Activation)      (None, 64, 64, 128)  0
batch_normalization_70[0][0]
__________________________________________________________________________________________________
__________________________
```

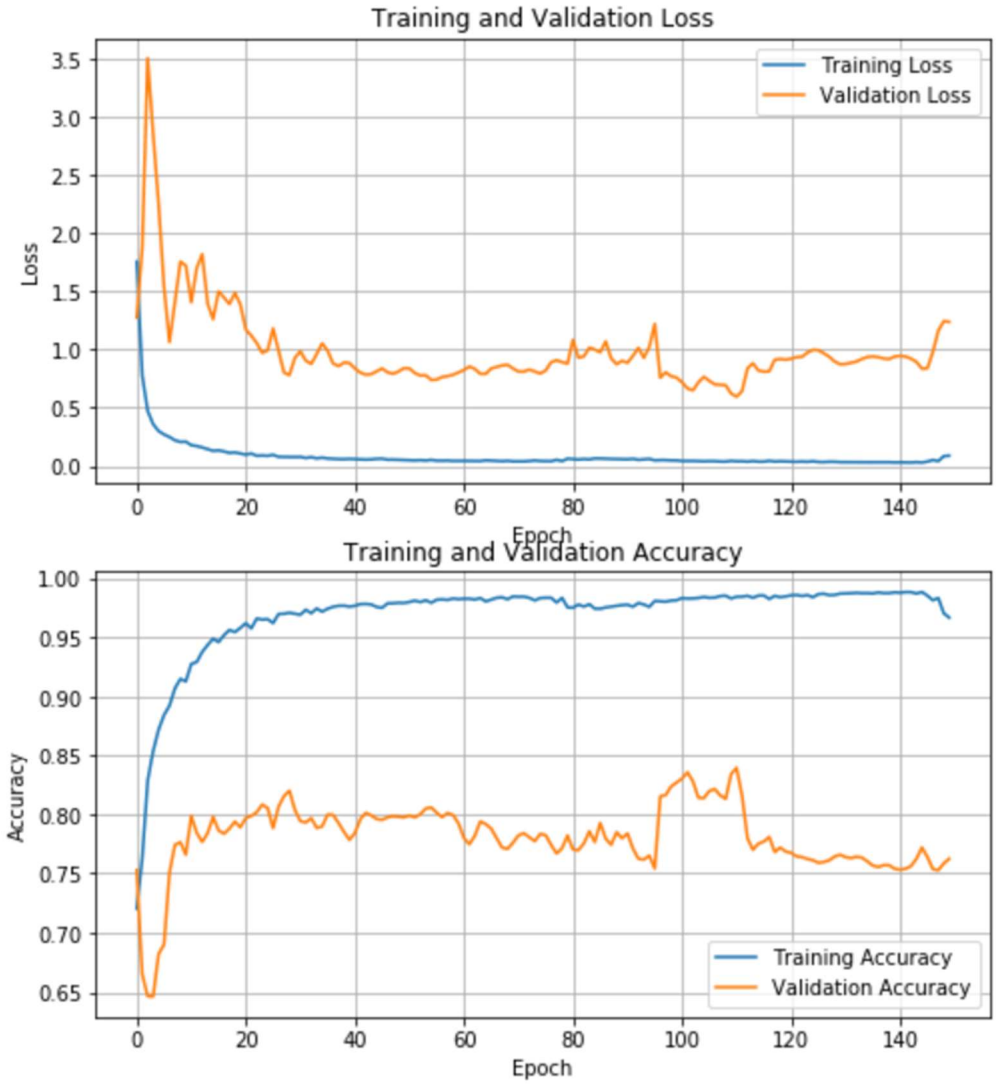
```
upconv4 (UpSampling2D)          (None, 128, 128, 128 0
activation_70[0][0]
__________________________________________________________________________________
_____________________
concat4 (Concatenate)           (None, 128, 128, 192 0
activation_56[0][0]

upconv4[0][0]
__________________________________________________________________________________
_____________________
conv17 (Conv2D)                 (None, 128, 128, 64) 110656
concat4[0][0]
__________________________________________________________________________________
_____________________
batch_normalization_71 (BatchNo (None, 128, 128, 64) 256
conv17[0][0]
__________________________________________________________________________________
_____________________
activation_71 (Activation)      (None, 128, 128, 64) 0
batch_normalization_71[0][0]
__________________________________________________________________________________
_____________________
conv18 (Conv2D)                 (None, 128, 128, 64) 36928
activation_71[0][0]
__________________________________________________________________________________
_____________________
batch_normalization_72 (BatchNo (None, 128, 128, 64) 256
conv18[0][0]
__________________________________________________________________________________
_____________________
activation_72 (Activation)      (None, 128, 128, 64) 0
batch_normalization_72[0][0]
__________________________________________________________________________________
_____________________
conv19 (Conv2D)                 (None, 128, 128, 1)  65
activation_72[0][0]
==================================================================================
```

CAT MODEL POST TRANSFER LEARNING TRAINING PARAMETERS

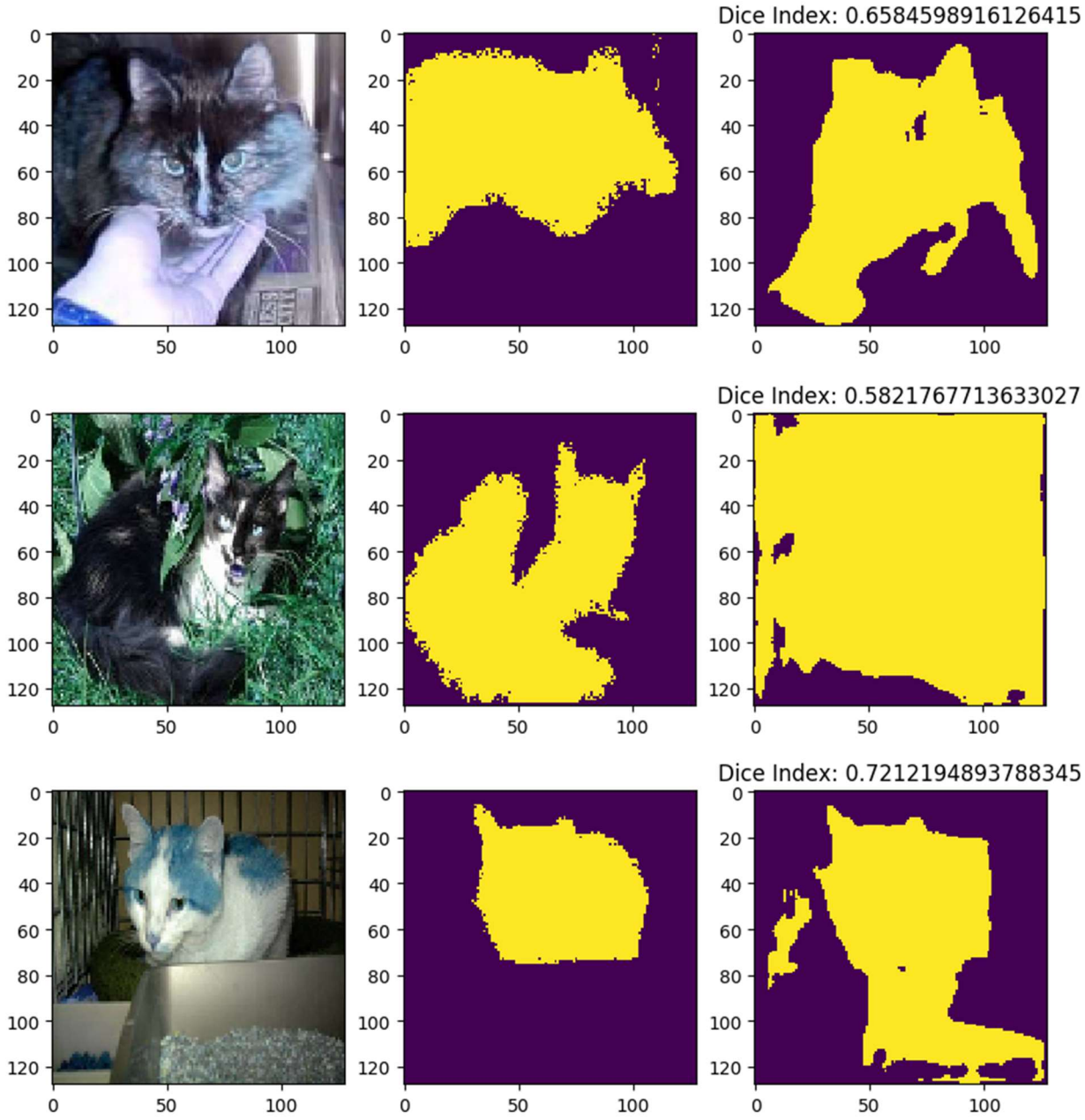
- Optimizer: Adam
- Learning Rate: 1e-3
- Loss: Cross-Entropy
- Epochs: 150
- Validation Split: 0.1
- Early Stopping: True, Patient=50

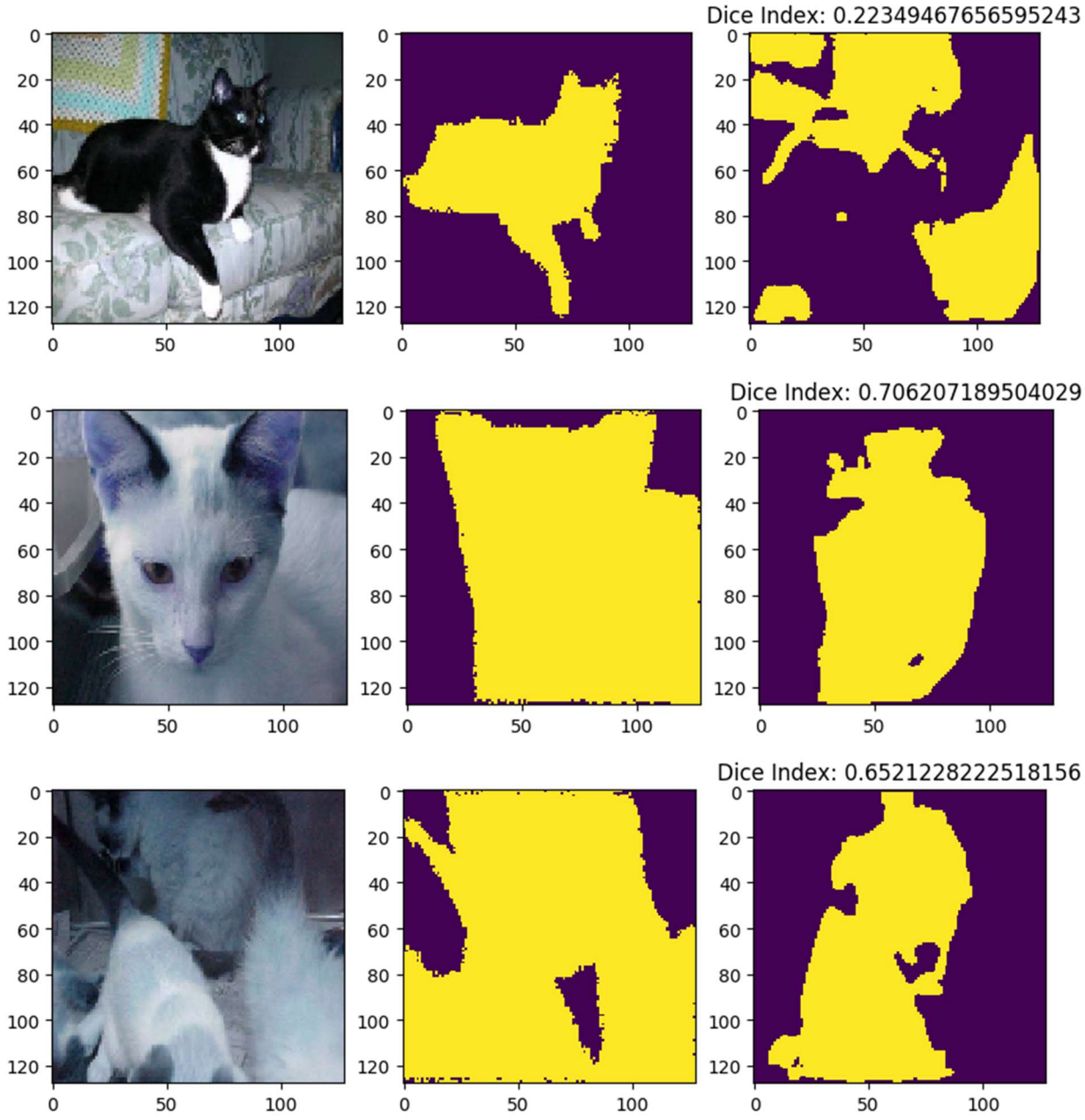
FINAL OUTPUT

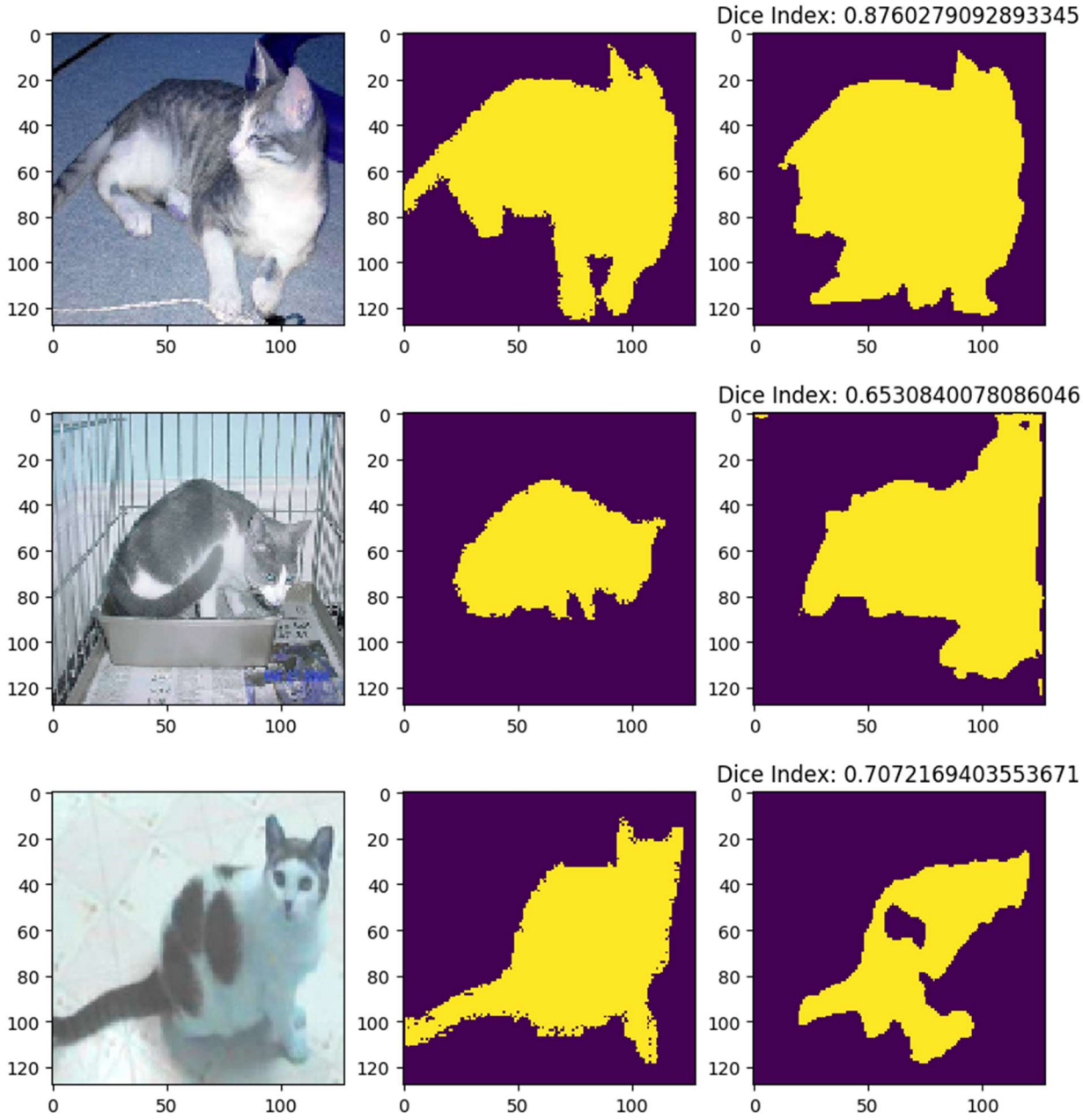
```
Log-loss (cost function):
training   (min:    0.029, max:    1.759, cur:    0.088)
validation (min:    0.596, max:    3.512, cur:    1.240)

Accuracy:
training   (min:    0.720, max:    0.989, cur:    0.967)
validation (min:    0.646, max:    0.839, cur:    0.762)
```

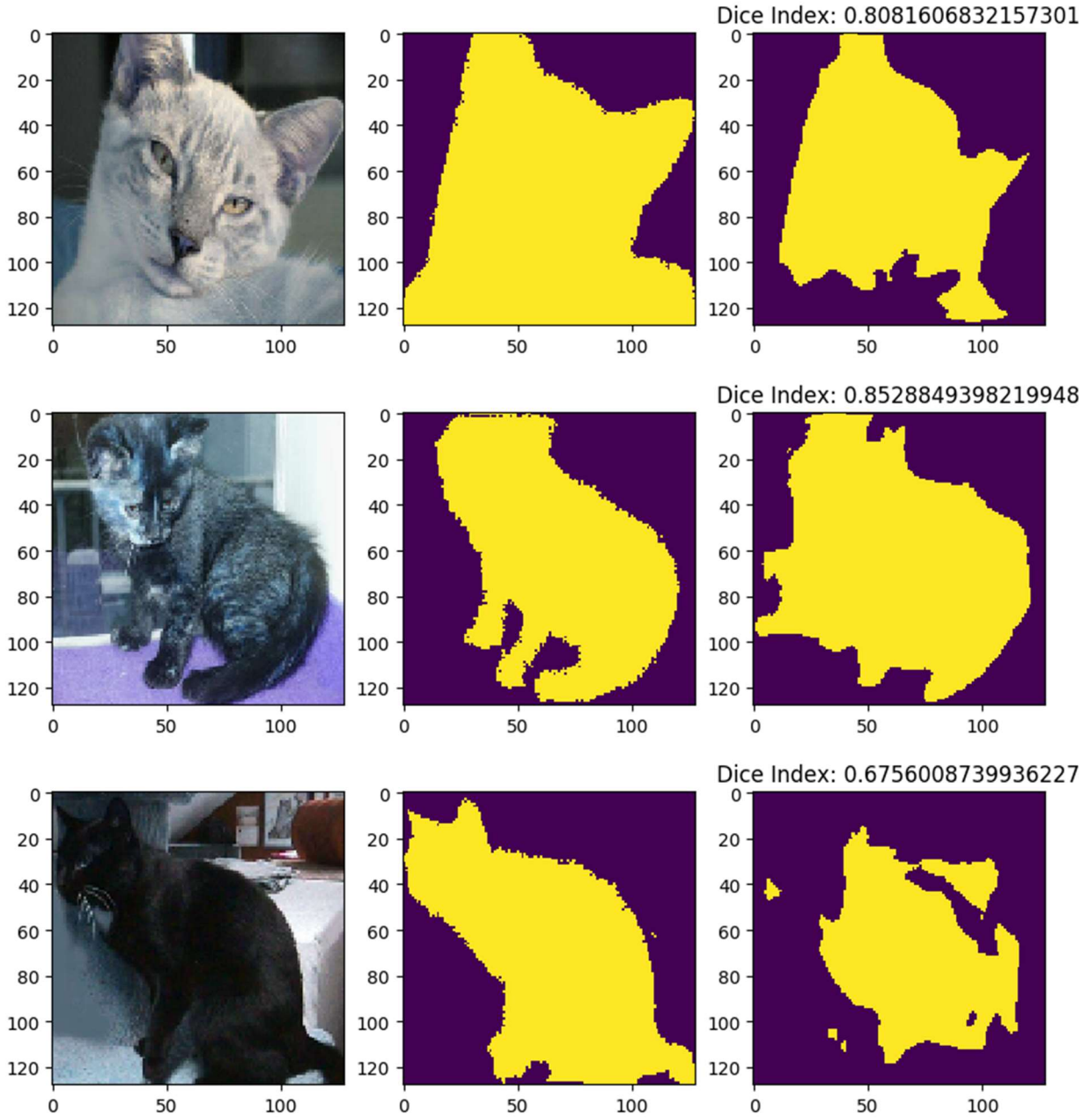


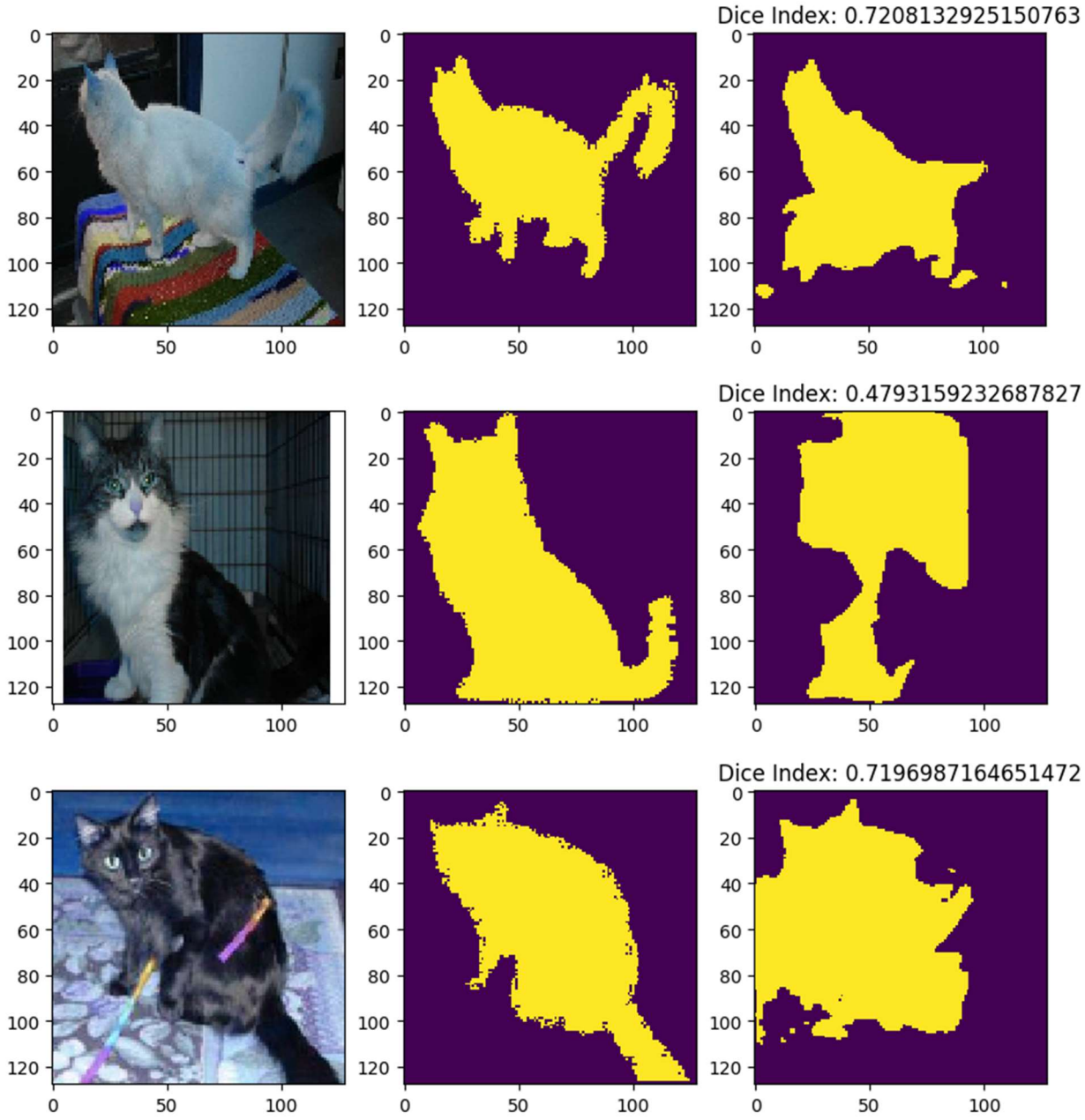
(ORIGINAL IMAGE, GROUND TRUTH MASK, PREDICTED MASK)

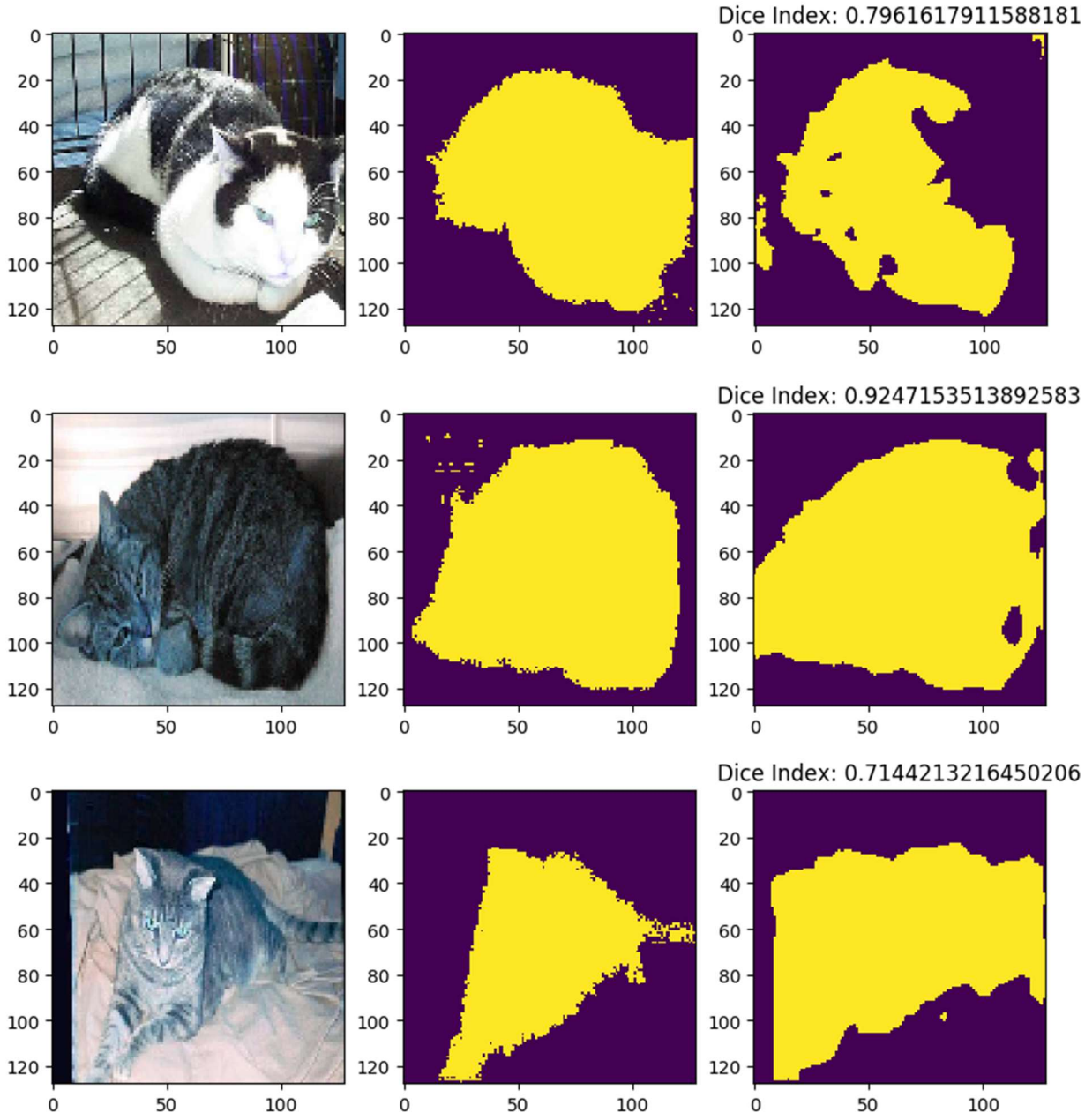

Dice Index: 0.6584598916126415
Dice Index: 0.5821767713633027
Dice Index: 0.7212194893788345

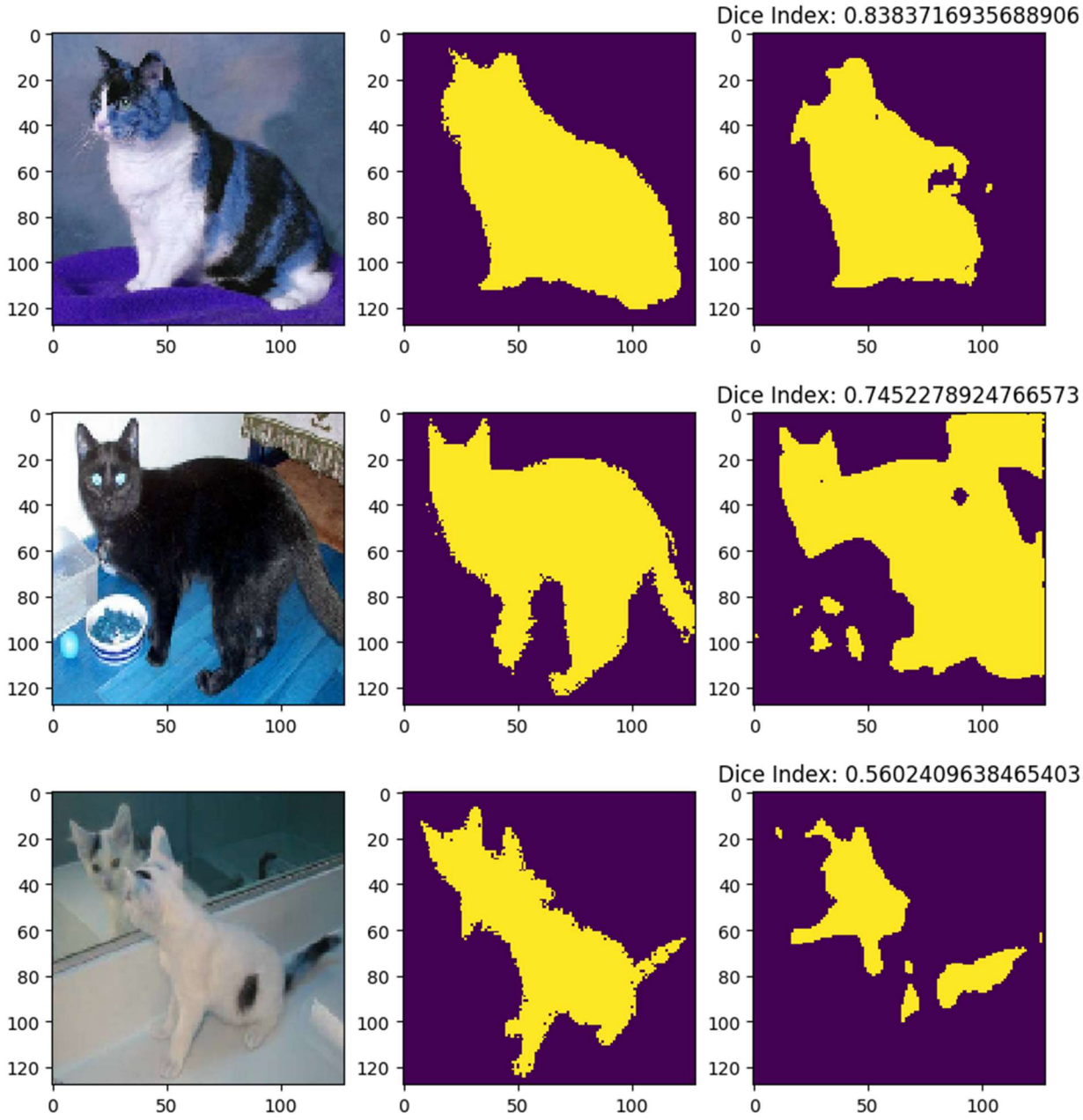

Dice Index: 0.22349467656595243
Dice Index: 0.706207189504029
Dice Index: 0.6521228222518156


Dice Index: 0.8760279092893345
Dice Index: 0.6530840078086046
Dice Index: 0.7072169403553671


Dice Index: 0.8081606832157301
Dice Index: 0.8528849398219948
Dice Index: 0.6756008739936227


Dice Index: 0.7208132925150763
Dice Index: 0.4793159232687827
Dice Index: 0.7196987164651472


Dice Index: 0.7961617911588181
Dice Index: 0.9247153513892583
Dice Index: 0.7144213216450206



1.4 Visualizing segmentation predictions

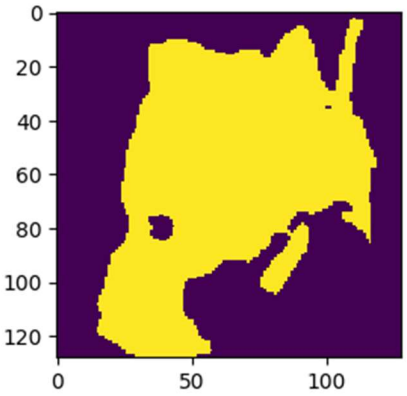
We utilized opencv's *FindContours* and *DrawContours* methods for visualizing our predicted masks.

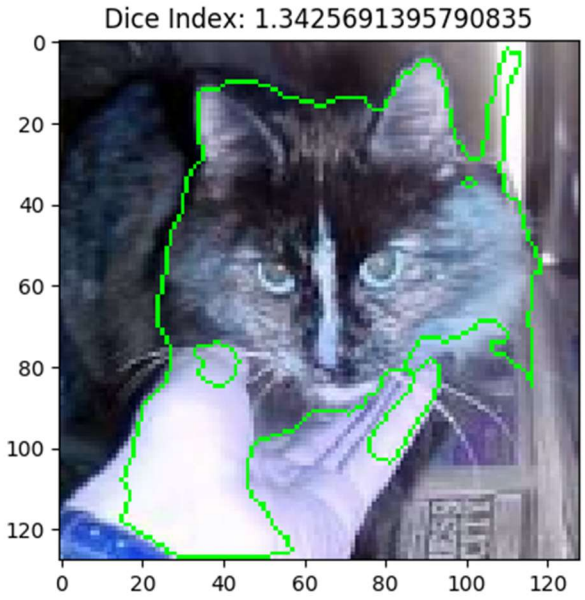
VISUALIZATION WITH CONTOURS CODE

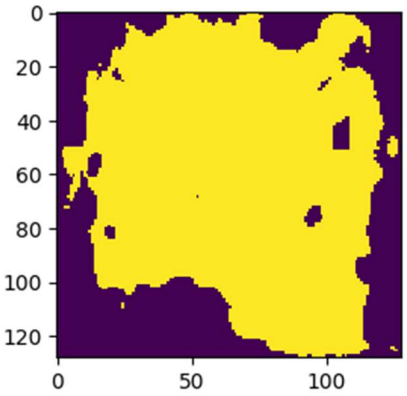
```
def visualize_segmentations_with_contours(images, ground_masks,
pred_masks, save_path=""):
    """
    Visualizes the predicted masks by drawing them as contours on their
corresponding images.

    :param images: the list of images
    :param ground_masks: the list of ground truth masks for the image
    :param pred_masks: the list of predicted masks for the image
    :param save_path: path to save all the images (if empty, then images
won't be saved)
    """
    for i in range(len(pred_masks)):
        p_mask = pred_masks[i].squeeze()
        p_mask[p_mask >= 0.5] = 255
        p_mask[p_mask < 0.5] = 0
        p_mask = p_mask.astype(np.uint8)
        dice_index = dice_coef_np(ground_masks[i].squeeze(), p_mask)

        # Apply cv2.threshold() to get a binary image
        _, thresh = cv2.threshold(p_mask, 150, 255, cv2.THRESH_BINARY)
        _, contours, _ = cv2.findContours(thresh, cv2.RETR_LIST,
cv2.CHAIN_APPROX_NONE)

        img = images[i].copy()
        for contour in contours:
            cv2.drawContours(img, contour, -1, (0, 255, 0), 1)

        figure = plt.figure(figsize=(10, 10))
        plt.subplot(231), plt.imshow(p_mask)
        plt.subplot(222), plt.imshow(cv2.blur(img, (3, 3)))

        plt.title("Dice Index: " + str(dice_index))

        plt.show()
        if save_path is not None or save_path != "":
            figure.savefig(save_path + "/result_" + str(i) + ".png",
dpi=100, bbox_inches='tight')
```

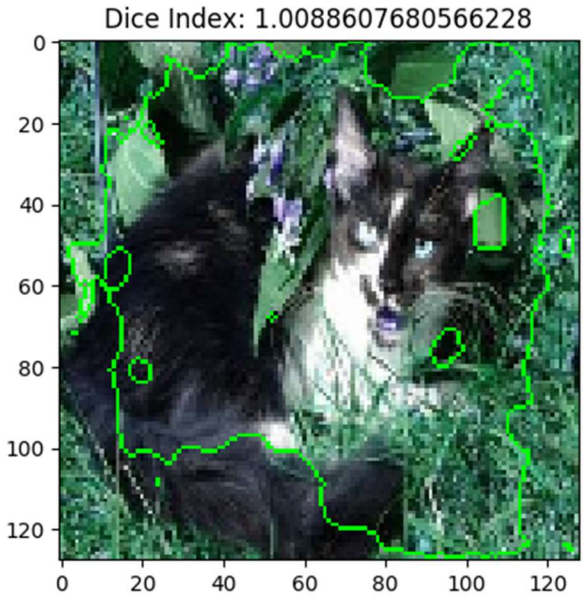
RESULTS

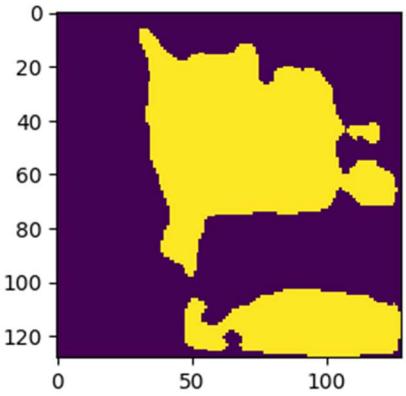
We generated these results using the cat model from **1.3 Transfer Learning.**

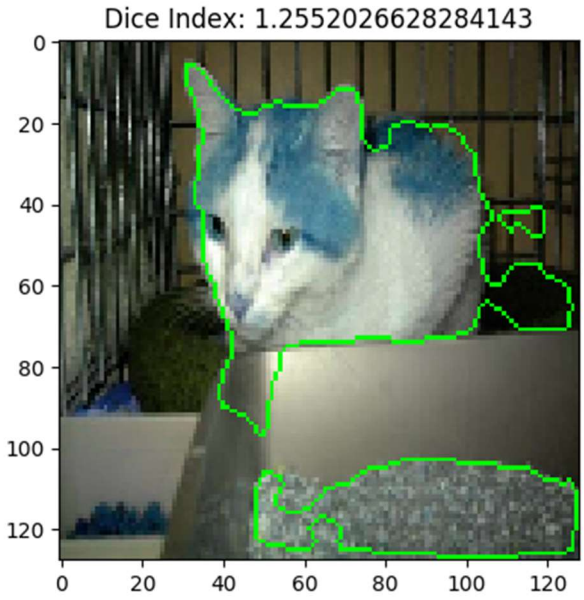

0
20
40
60
80
100
120
0
50
100

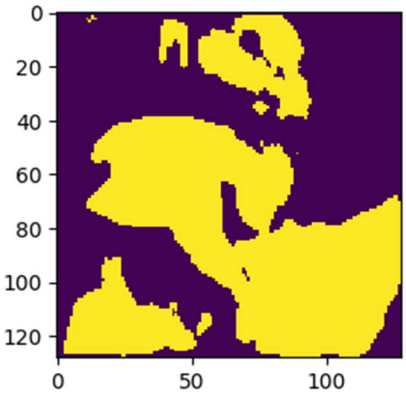

Dice Index: 1.3425691395790835
0
20
40
60
80
100
120
0
20
40
60
80
100
120

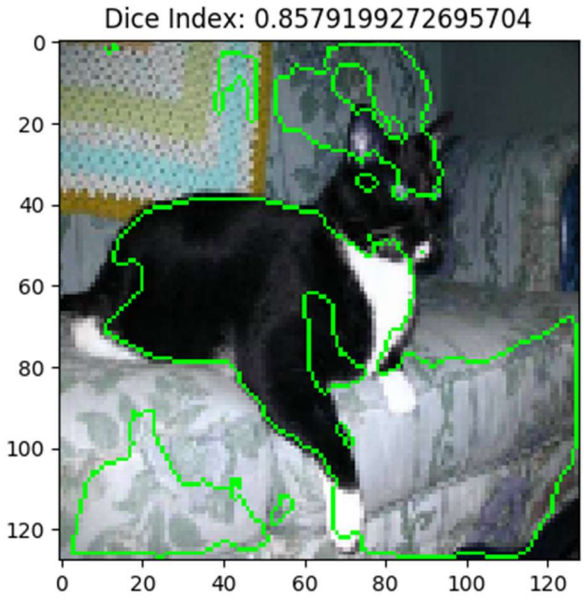

0
20
40
60
80
100
120
0
50
100

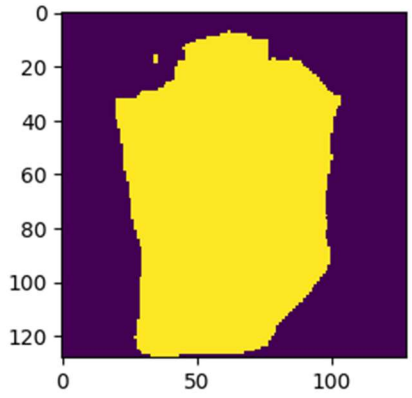

Dice Index: 1.0088607680566228
0
20
40
60
80
100
120
0
20
40
60
80
100
120

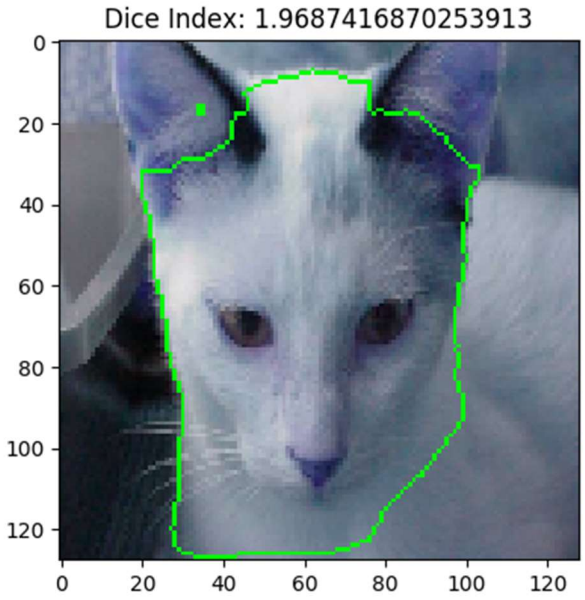

0
20
40
60
80
100
120
0
50
100

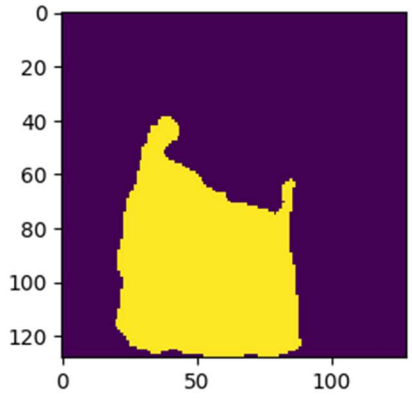

Dice Index: 1.2552026628284143
0
20
40
60
80
100
120
0
20
40
60
80
100
120

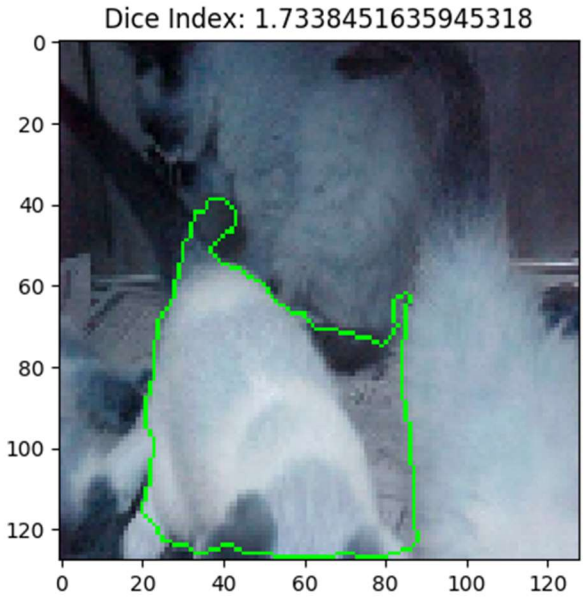

0
20
40
60
80
100
120
0
50
100

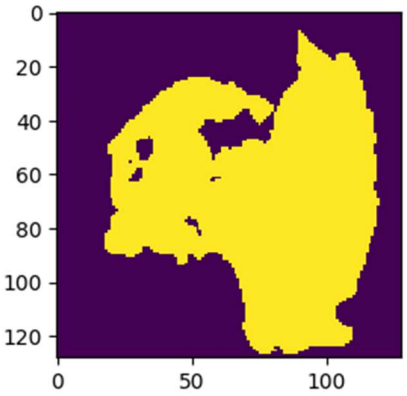

Dice Index: 0.8579199272695704
0
20
40
60
80
100
120
0
20
40
60
80
100
120

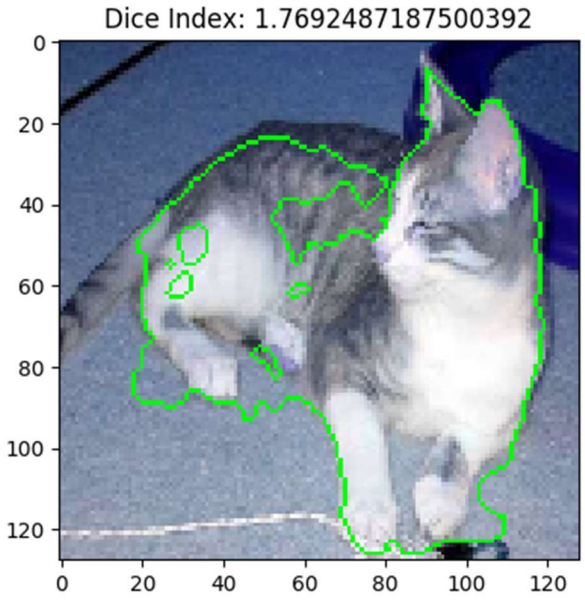

0
20
40
60
80
100
120
0
50
100

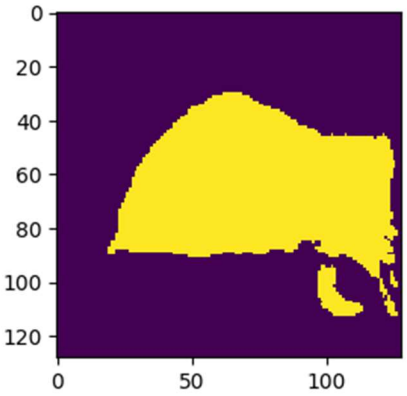

Dice Index: 1.9687416870253913
0
20
40
60
80
100
120
0
20
40
60
80
100
120

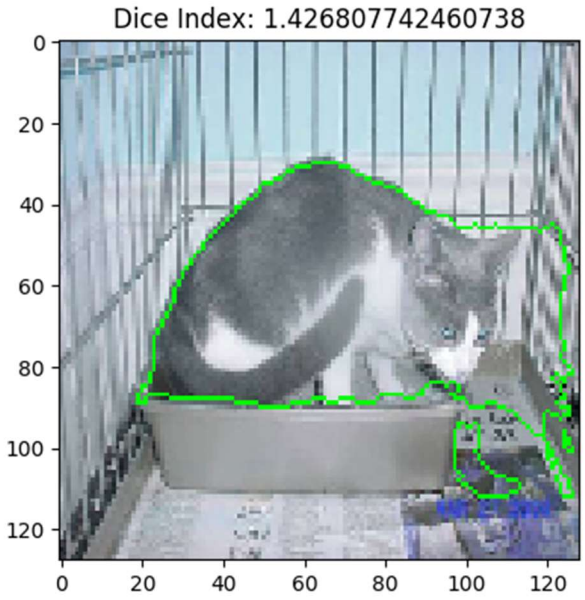

0
20
40
60
80
100
120
0
50
100

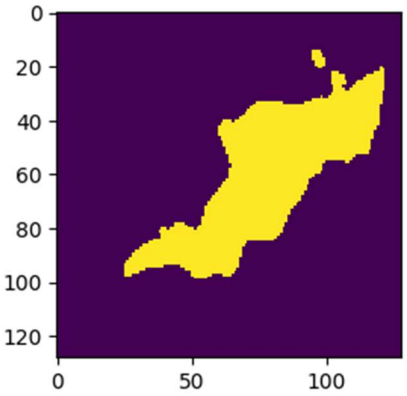

Dice Index: 1.7338451635945318
0
20
40
60
80
100
120
0
20
40
60
80
100
120

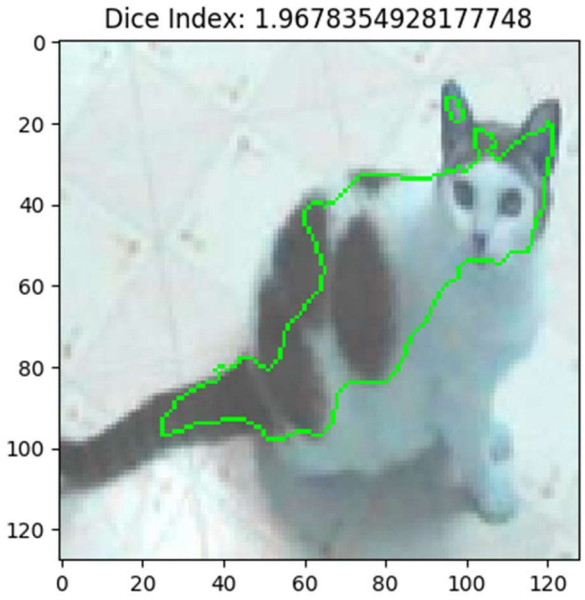

0
20
40
60
80
100
120
0
50
100

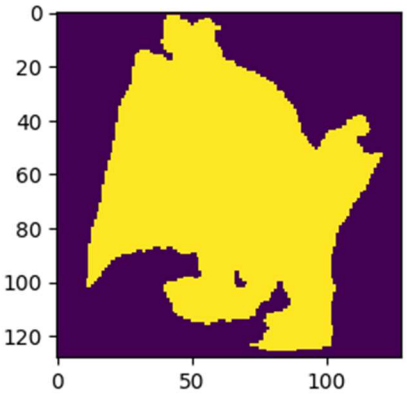

Dice Index: 1.7692487187500392
0
20
40
60
80
100
120
0
20
40
60
80
100
120

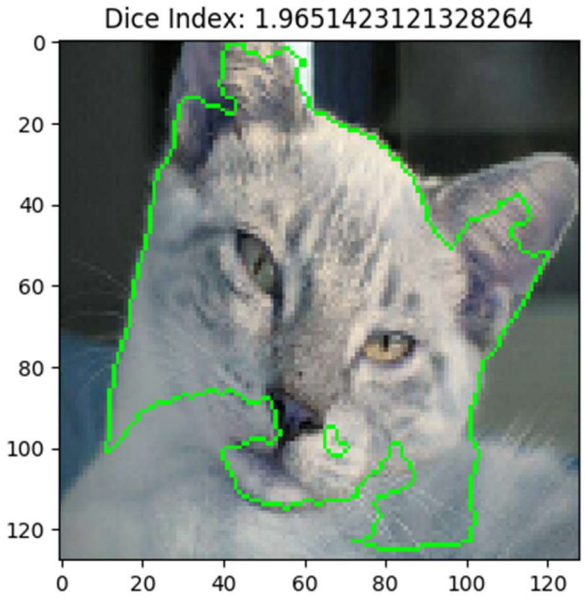

0
20
40
60
80
100
120
0
50
100

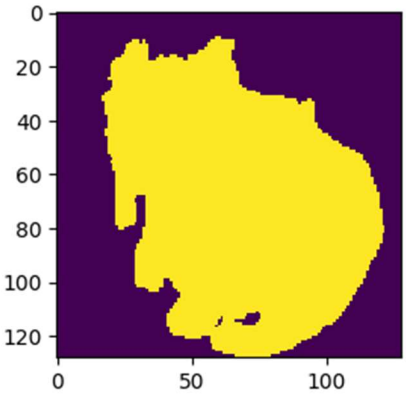

Dice Index: 1.426807742460738
0
20
40
60
80
100
120
0
20
40
60
80
100
120

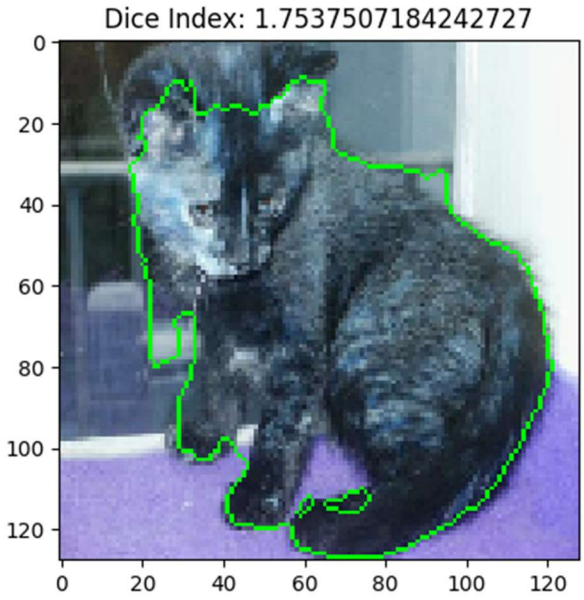

0
20
40
60
80
100
120
0
50
100

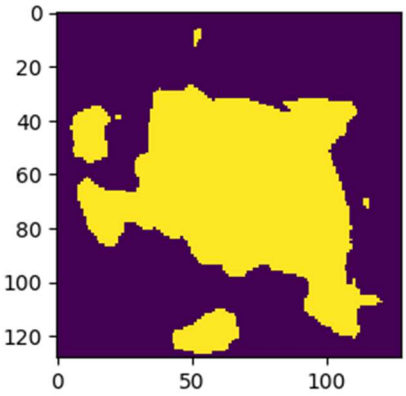

Dice Index: 1.9678354928177748
0
20
40
60
80
100
120
0
20
40
60
80
100
120

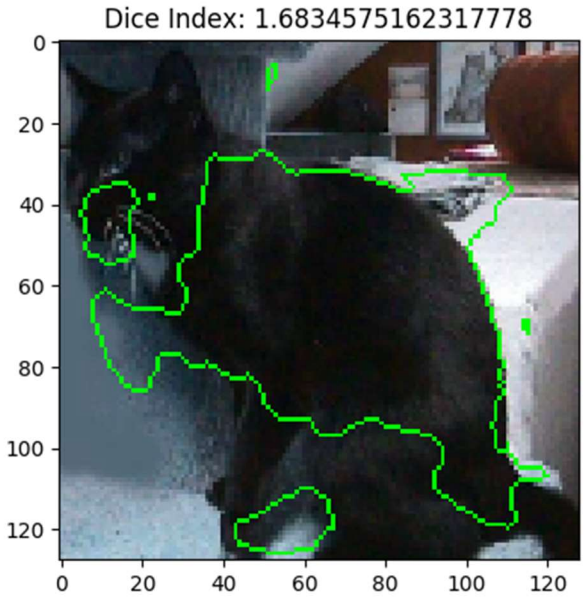

0
20
40
60
80
100
120
0
50
100

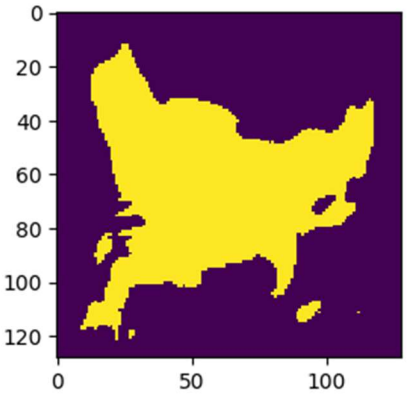

Dice Index: 1.9651423121328264
0
20
40
60
80
100
120
0
20
40
60
80
100
120

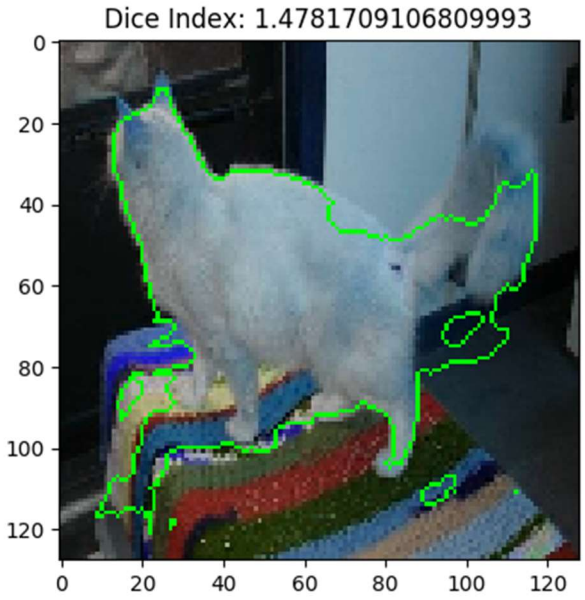

0
20
40
60
80
100
120
0
50
100

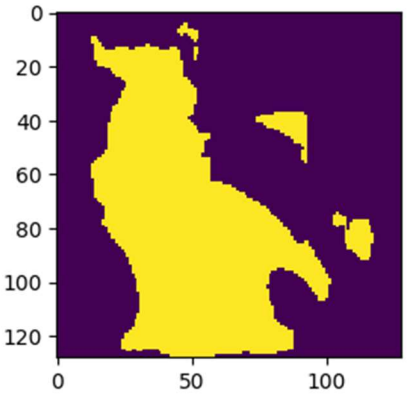

Dice Index: 1.7537507184242727
0
20
40
60
80
100
120
0
20
40
60
80
100
120

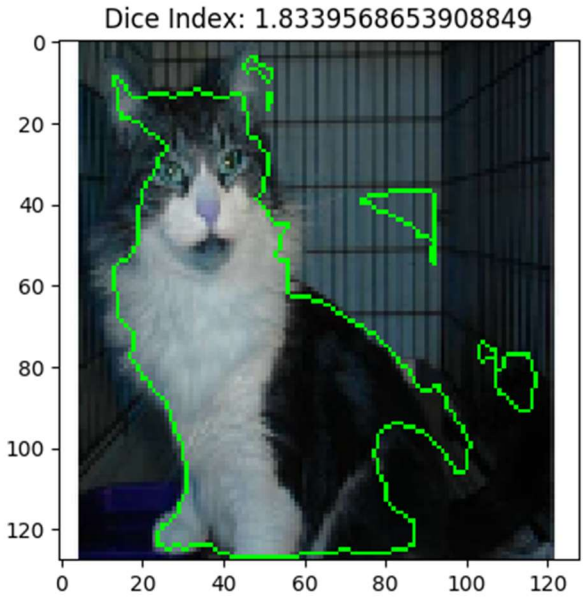

0
20
40
60
80
100
120
0
50
100

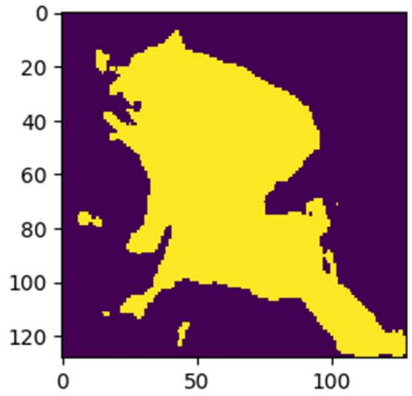

Dice Index: 1.6834575162317778
0
20
40
60
80
100
120
0
20
40
60
80
100
120

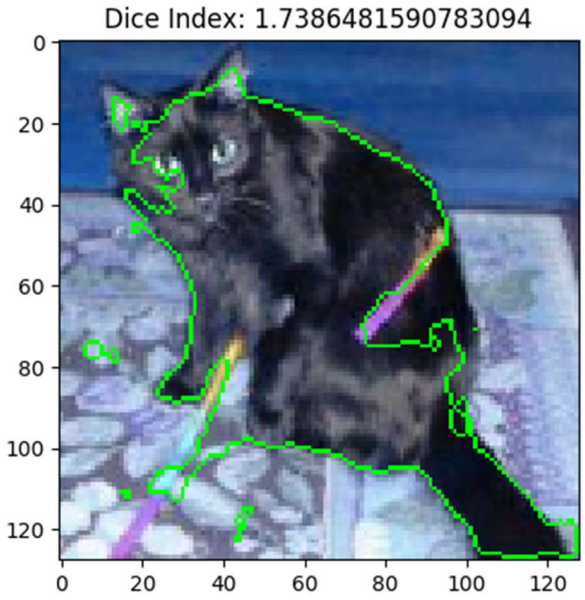

0
20
40
60
80
100
120
0
50
100

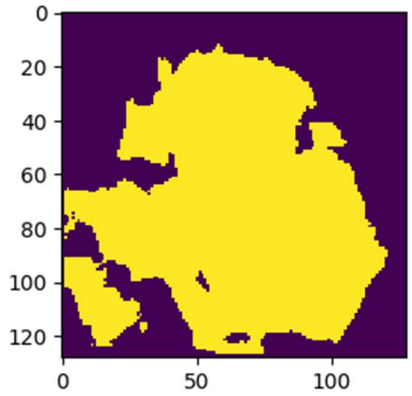

Dice Index: 1.4781709106809993
0
20
40
60
80
100
120
0
20
40
60
80
100
120

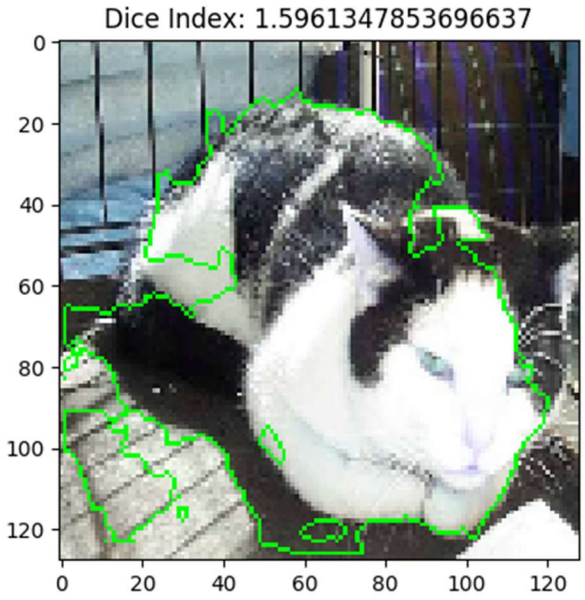

0
20
40
60
80
100
120
0
50
100

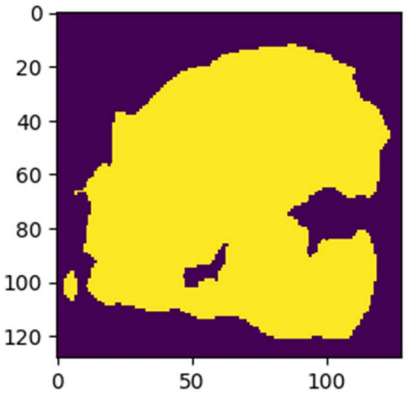

Dice Index: 1.8339568653908849
0
20
40
60
80
100
120
0
20
40
60
80
100
120

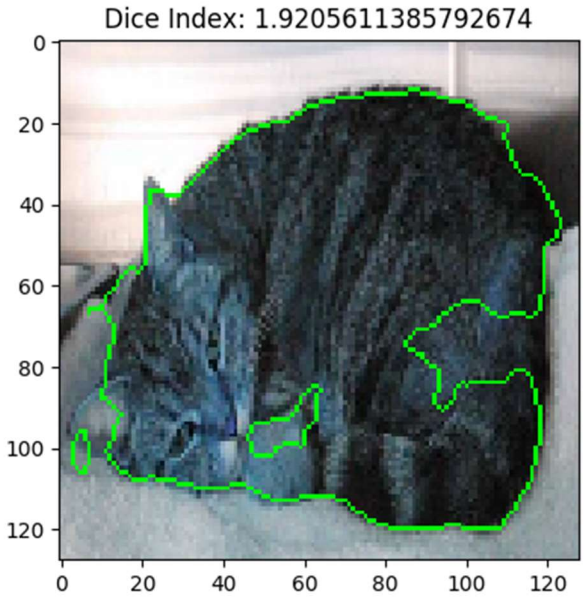

0
20
40
60
80
100
120
0
50
100

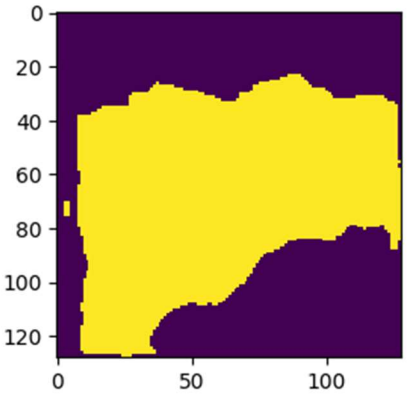

Dice Index: 1.7386481590783094
0
20
40
60
80
100
120
0
20
40
60
80
100
120

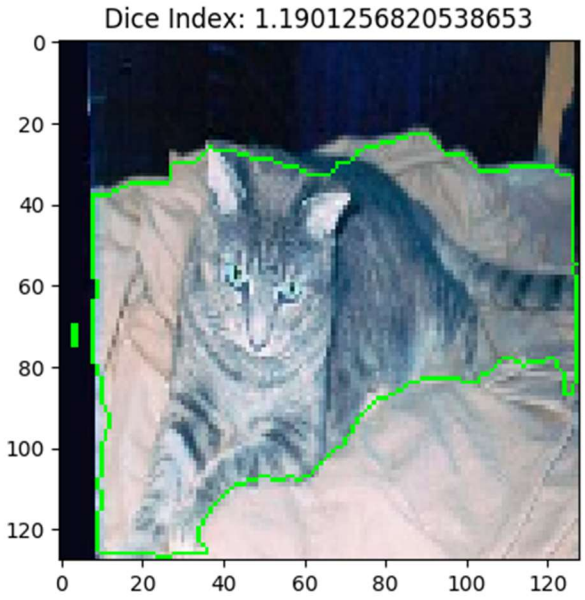

0
20
40
60
80
100
120
0
50
100

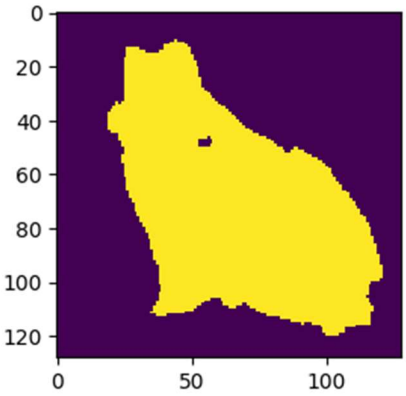

Dice Index: 1.5961347853696637
0
20
40
60
80
100
120
0
20
40
60
80
100
120

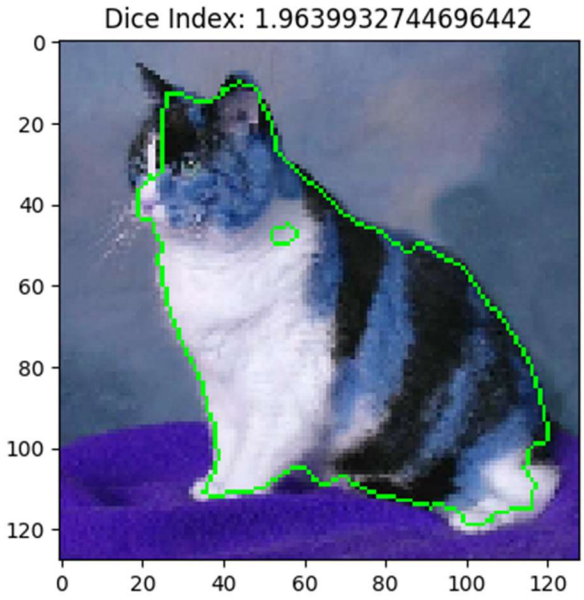

0
20
40
60
80
100
120
0
50
100

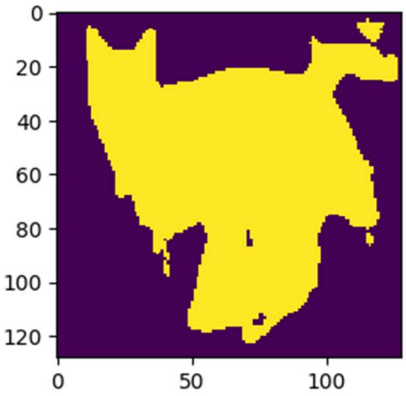

Dice Index: 1.9205611385792674
0
20
40
60
80
100
120
0
20
40
60
80
100
120

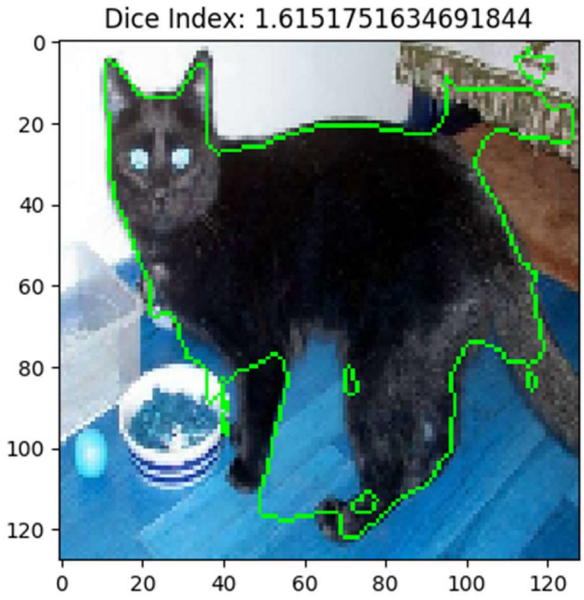

0
20
40
60
80
100
120
0
50
100

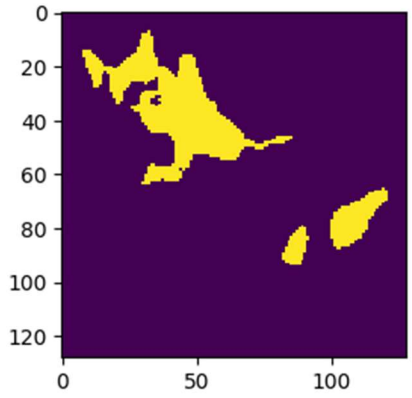

Dice Index: 1.1901256820538653
0
20
40
60
80
100
120
0
20
40
60
80
100
120

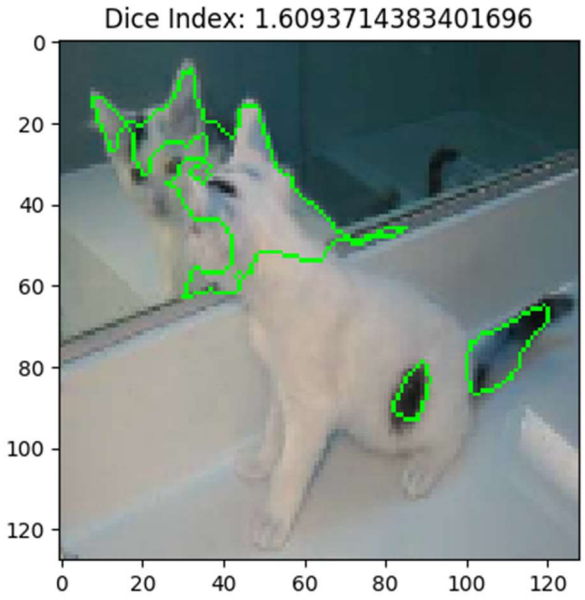

0
20
40
60
80
100
120
0
50
100


Dice Index: 1.9639932744696442
0
20
40
60
80
100
120
0
20
40
60
80
100
120


0
20
40
60
80
100
120
0
50
100


Dice Index: 1.6151751634691844
0
20
40
60
80
100
120
0
20
40
60
80
100
120





2.1 Bounding Box Design – Problem Definition

We decided to create a neural net that mimicked the idea of U-Net, which is essentially an encoder-decoder network. The input to our network would be the noisy images with circle, and a corresponding binary mask of the circle (no noise). We would feed this into our network and use a Cross-Entropy loss function for optimization. We decided this was the best way to represent the problem because the network is trying to find image segmentations for each pixel, where a pixel can be segmented to being part of the circle or part of the background. But, since a circle is a very simple object, we were able reduce the number of layers in the standard U-Net architecture and end up with a much simpler architecture that can find the necessary features of a circle.

Once our network generates a predicted mask for the given image, we run it through **Hough Circle Transform** to get the rows, cols, and radius of the circle. Opencv provides a very useful implementation of this method that generates the required parameters given a grayscale image (our predicted binary mask).

NETWORK VISUALIZATION

```
Layer (type)                 Output Shape              Param #     Connected
to
==================================================================================
==========================
input1 (InputLayer)          (None, 200, 200, 1)   0
_____________________________________________________________________________
_____________________
```

```
conv1 (Conv2D)                  (None, 200, 200, 64) 640
input1[0][0]
__________________________________________________________________________________________________
batch_normalization_51 (BatchNo (None, 200, 200, 64) 256
conv1[0][0]
__________________________________________________________________________________________________
activation_51 (Activation)      (None, 200, 200, 64) 0
batch_normalization_51[0][0]
__________________________________________________________________________________________________
maxpool1 (MaxPooling2D)         (None, 100, 100, 64) 0
activation_51[0][0]
__________________________________________________________________________________________________
conv2 (Conv2D)                  (None, 100, 100, 128 73856
maxpool1[0][0]
__________________________________________________________________________________________________
batch_normalization_52 (BatchNo (None, 100, 100, 128 512
conv2[0][0]
__________________________________________________________________________________________________
activation_52 (Activation)      (None, 100, 100, 128 0
batch_normalization_52[0][0]
__________________________________________________________________________________________________
maxpool2 (MaxPooling2D)         (None, 50, 50, 128)  0
activation_52[0][0]
__________________________________________________________________________________________________
conv3 (Conv2D)                  (None, 50, 50, 256)  295168
maxpool2[0][0]
__________________________________________________________________________________________________
batch_normalization_53 (BatchNo (None, 50, 50, 256)  1024
conv3[0][0]
__________________________________________________________________________________________________
activation_53 (Activation)      (None, 50, 50, 256)  0
batch_normalization_53[0][0]
__________________________________________________________________________________________________
upconv1 (UpSampling2D)          (None, 100, 100, 256 0
activation_53[0][0]
__________________________________________________________________________________________________
concat1 (Concatenate)           (None, 100, 100, 384 0
activation_52[0][0]

upconv1[0][0]
```

```
__________________________________________________________________________________
_____________________________
conv4 (Conv2D)                  (None, 100, 100, 128 442496
concat1[0][0]
__________________________________________________________________________________
_____________________________
batch_normalization_54 (BatchNo (None, 100, 100, 128 512
conv4[0][0]
__________________________________________________________________________________
_____________________________
activation_54 (Activation)      (None, 100, 100, 128 0
batch_normalization_54[0][0]
__________________________________________________________________________________
_____________________________
upconv2 (UpSampling2D)          (None, 200, 200, 128 0
activation_54[0][0]
__________________________________________________________________________________
_____________________________
concat2 (Concatenate)           (None, 200, 200, 192 0
activation_51[0][0]

upconv2[0][0]
__________________________________________________________________________________
_____________________________
conv5 (Conv2D)                  (None, 200, 200, 64) 110656
concat2[0][0]
__________________________________________________________________________________
_____________________________
batch_normalization_55 (BatchNo (None, 200, 200, 64) 256
conv5[0][0]
__________________________________________________________________________________
_____________________________
activation_55 (Activation)      (None, 200, 200, 64) 0
batch_normalization_55[0][0]
__________________________________________________________________________________
_____________________________
conv6 (Conv2D)                  (None, 200, 200, 1)  65
activation_55[0][0]
==================================================================================
======================
Total params: 925,441
Trainable params: 924,161
Non-trainable params: 1,280
```

2.2 Implementation

Our CNN uses convolutional layers in the encoder part, after which, we perform maxpooling (downsampling). There is a single convolution layer in the middle of our network containing 256 filters. Then in the decoder part, we have 2 more convolution layers, after which, we perform upsampling, and concatenation with their corresponding

feature maps in the encoder part of the network. Each convolution utilizes batch-normalization and uses a **relu** activation function.

MODEL CODE

```
def conv2d_block(name, x, filters, kernel_size, padding, activation):
    conv = Conv2D(name=name, filters=filters, kernel_size=kernel_size,
padding=padding)(x)
    conv = BatchNormalization()(conv)
    conv = Activation(activation=activation)(conv)
    return conv
```

```
def circle_model(input_size):
    inputs = Input(name="input1", shape=input_size)
    conv1 = conv2d_block(name="conv1", x=inputs, filters=64,
kernel_size=3, padding="same", activation="relu")
    maxpool1 = MaxPool2D(name="maxpool1", pool_size=(2, 2),
strides=2)(conv1)

    conv2 = conv2d_block(name="conv2", x=maxpool1, filters=128,
kernel_size=3, padding="same", activation="relu")
    maxpool2 = MaxPool2D(name="maxpool2", pool_size=(2, 2),
strides=2)(conv2)

    conv3 = conv2d_block(name="conv3", x=maxpool2, filters=256,
kernel_size=3, padding="same", activation="relu")
    upconv1 = UpSampling2D(name="upconv1", size=(2, 2))(conv3)

    concat1 = concatenate(name="concat1", inputs=[conv2, upconv1], axis=3)
    conv4 = conv2d_block(name="conv4", x=concat1, filters=128,
kernel_size=3, padding="same", activation="relu")
    upconv2 = UpSampling2D(name="upconv2", size=(2, 2))(conv4)

    concat2 = concatenate(name="concat2", inputs=[conv1, upconv2], axis=3)
    conv5 = conv2d_block(name="conv5", x=concat2, filters=64,
kernel_size=3, padding="same", activation="relu")

    conv6 = Conv2D(name="conv6", filters=1, kernel_size=1,
activation="sigmoid")(conv5)
    u_model = Model(input=inputs, output=conv6)

    u_model.summary()

    return u_model
```

FIND CIRCLE CODE

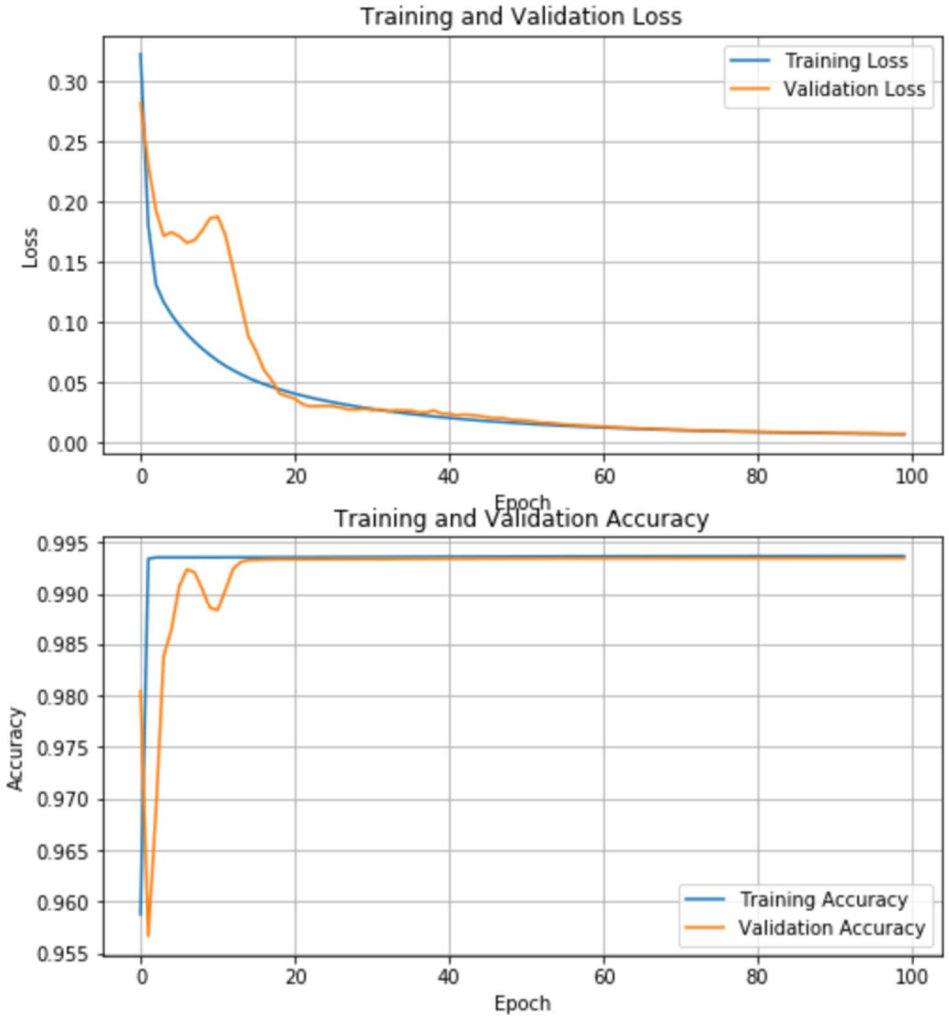
```
def find_circle(model, img):
    row, col, rad = 0, 0, 0

    # Get prediction
    y_pred = model.predict(np.expand_dims(img.reshape(img.shape[0],
img.shape[1], 1), axis=0))

    p_mask = y_pred.squeeze()
    p_mask[p_mask >= 0.5] = 255
    p_mask[p_mask < 0.5] = 0
    p_mask = p_mask.astype(np.uint8)

    # Apply cv2.threshold() to get a binary image
    _, thresh = cv2.threshold(p_mask, 50, 255, cv2.THRESH_BINARY)

    # Detect circles in the image
    circles = cv2.HoughCircles(thresh, method=cv2.HOUGH_GRADIENT, dp=1,
minDist=thresh.shape[0] / 2, param1=10,
                               param2=8)

    n_img = img.copy()
    # Ensure at least some circles were found
    if circles is not None:
        # Convert the (x, y) coordinates and radius of the circles to
integers
        circles = np.round(circles[0, :]).astype("int")
        # Loop over the (x, y) coordinates and radius of the circles
        for (x, y, r) in circles:
            row, col, rad = y, x, r
            cv2.circle(n_img, (x, y), r, (0, 255, 0), 2)

    return (row, col, rad), n_img
```

CIRCLES MODEL TRAINING PARAMETERS

- Optimizer: Adam
- Learning Rate: 1e-4
- Loss: Cross-Entropy
- Epochs: 150
- Validation Split: 0.1
- Early Stopping: True, Patient=50

FINAL OUTPUT

```
Log-loss (cost function):
training   (min:    0.007, max:    0.323, cur:    0.007)
validation (min:    0.007, max:    0.282, cur:    0.007)

Accuracy:
training   (min:    0.959, max:    0.994, cur:    0.994)
validation (min:    0.957, max:    0.993, cur:    0.993)
```



2.3 IOU Optimization

When a neural network learns, it uses loss function to determine in which direction (and magnitude) the weights of the networks should be nudged. This requires computing gradients; therefore the loss function should be easily differentiable. The gradient of say, L2 loss, can help our network determine how to nudge our weights.

The problem with using IOU as a loss function is that it is not directly differentiable with respect to the parameters of our network, therefore our network would have a very difficult time determining how to nudge the weights. IOU does not cleanly map to an

error surface that can allow us to compute gradients to give us directions for nudging weights.

It is, however, worth mentioning that there are existing papers experiment with use of an approximated IOU loss function. One such paper is mentioned below:

Yu, Jiahui, et al. "Unitbox: An advanced object detection network." *Proceedings of the 24th ACM international conference on Multimedia*. ACM, 2016.

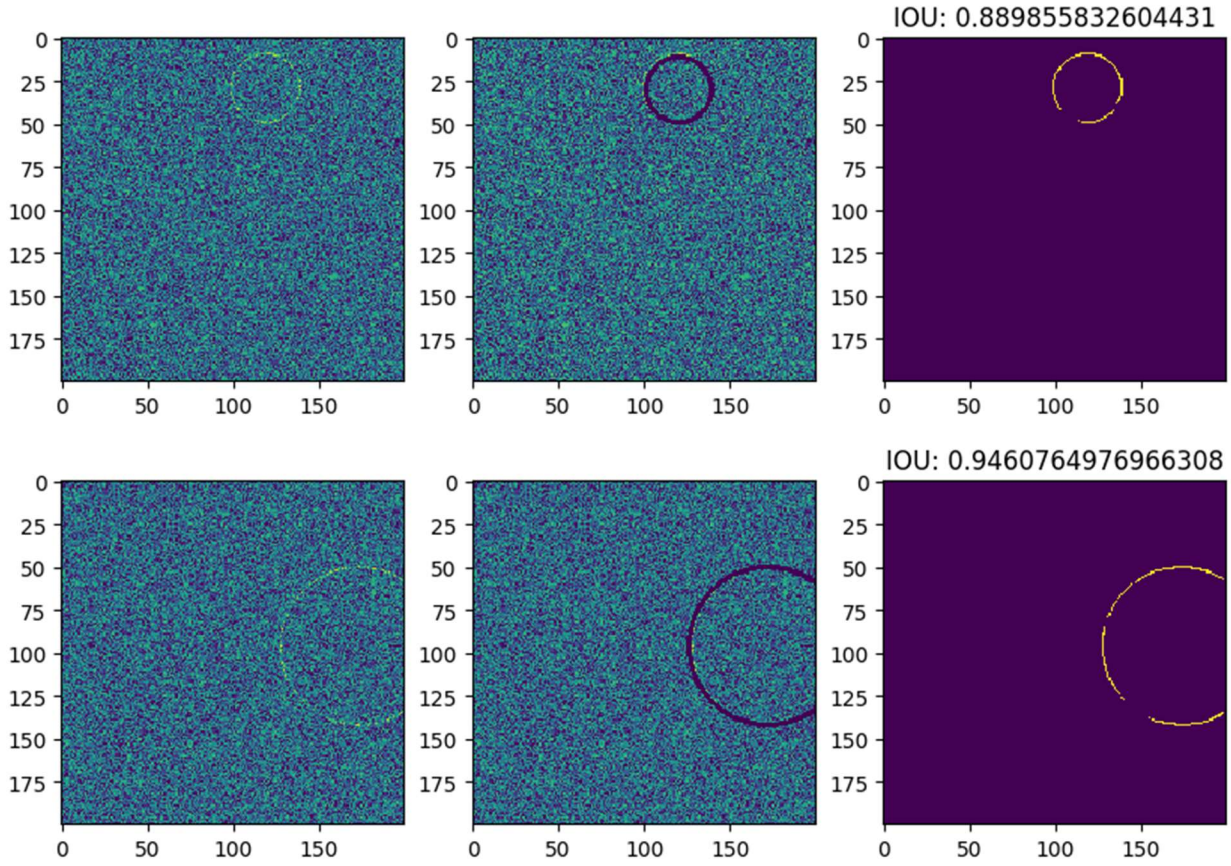
2.4 Visualization and error analysis

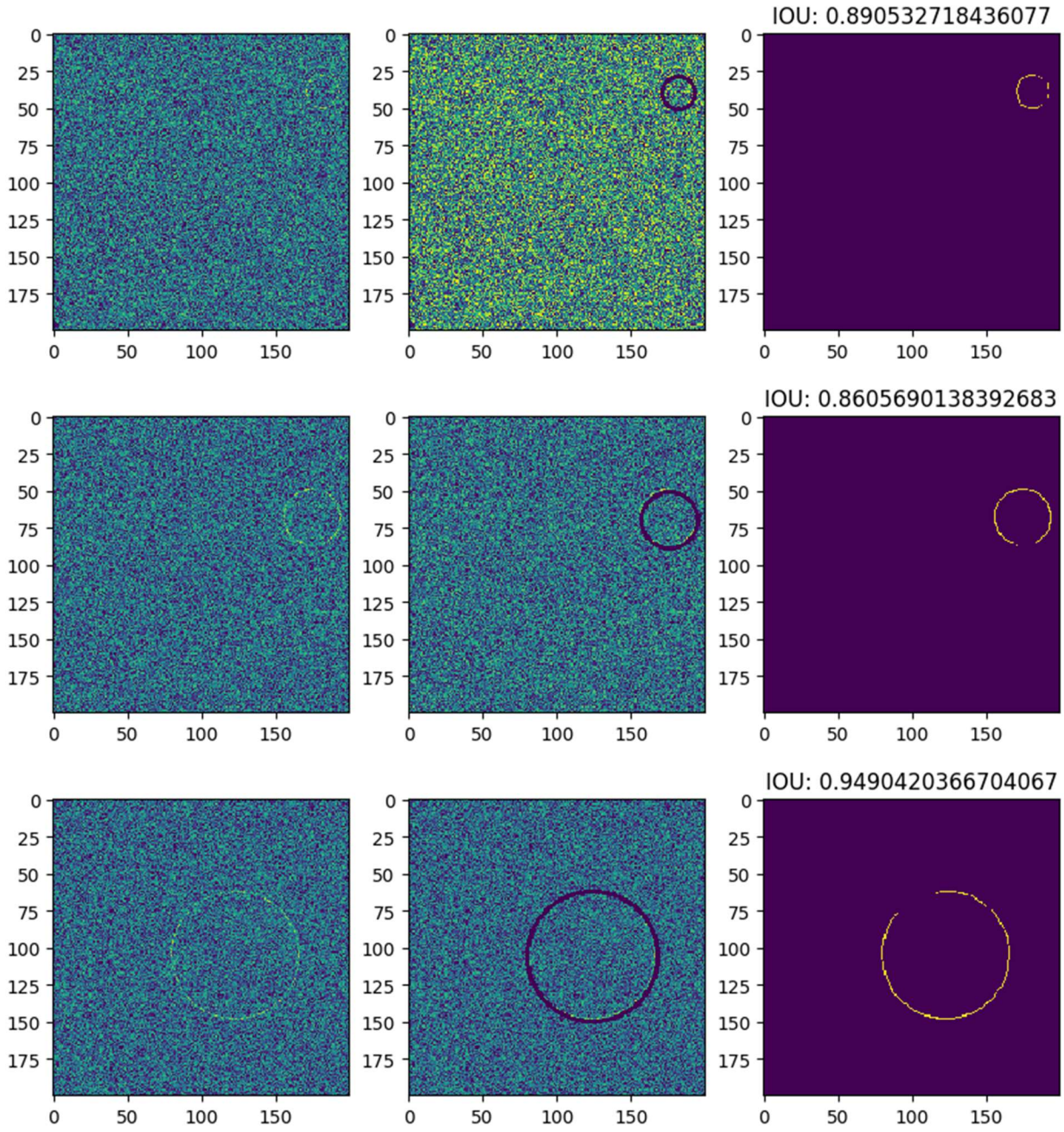
VISUALIZATION CODE

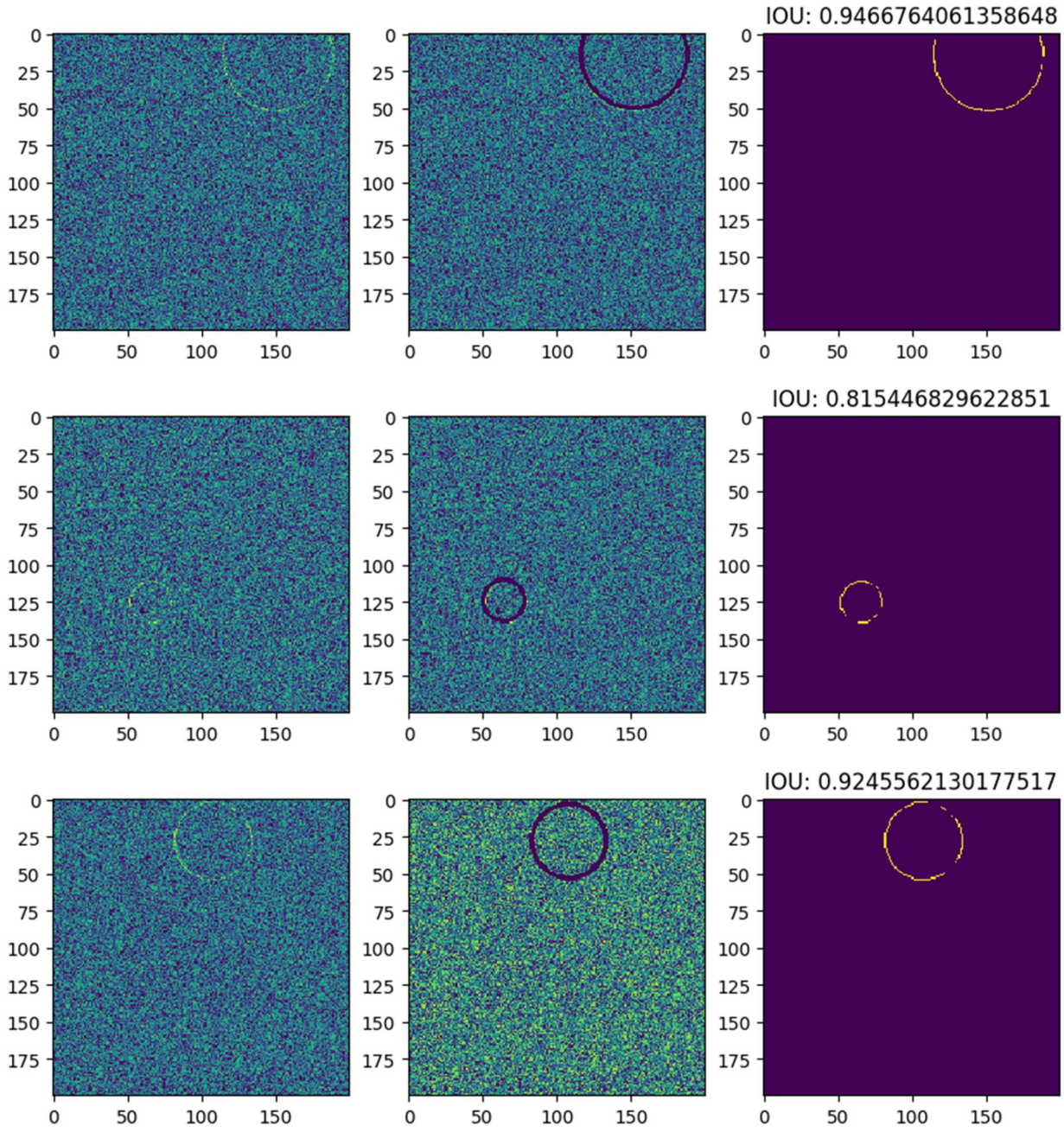
```
def visualize_segmentations_with_contours(images, ground_masks,
pred_masks, save_path=""):
    """
    Visualizes the predicted masks by drawing them as contours on their
corresponding images.

    :param images: the list of images
    :param ground_masks: the list of ground truth masks for the image
    :param pred_masks: the list of predicted masks for the image
    :param save_path: path to save all the images (if empty, then images
won't be saved)
    """
    for i in range(len(pred_masks)):
        p_mask = pred_masks[i].squeeze()
        p_mask[p_mask >= 0.5] = 255
        p_mask[p_mask < 0.5] = 0
        p_mask = p_mask.astype(np.uint8)

        # Apply cv2.threshold() to get a binary image
        _, thresh = cv2.threshold(p_mask, 50, 255, cv2.THRESH_BINARY)
        img = images[i].copy()

        # detect circles in the image
        circles = cv2.HoughCircles(thresh, method=cv2.HOUGH_GRADIENT,
dp=1, minDist=thresh.shape[0]/2, param1=10, param2=8)

        # Ensure at least some circles were found
        if circles is not None:
            # Convert the (x, y) coordinates and radius of the circles to
integers
            circles = np.round(circles[0, :]).astype("int")

            # Loop over the (x, y) coordinates and radius of the circles
            for (x, y, r) in circles:
```

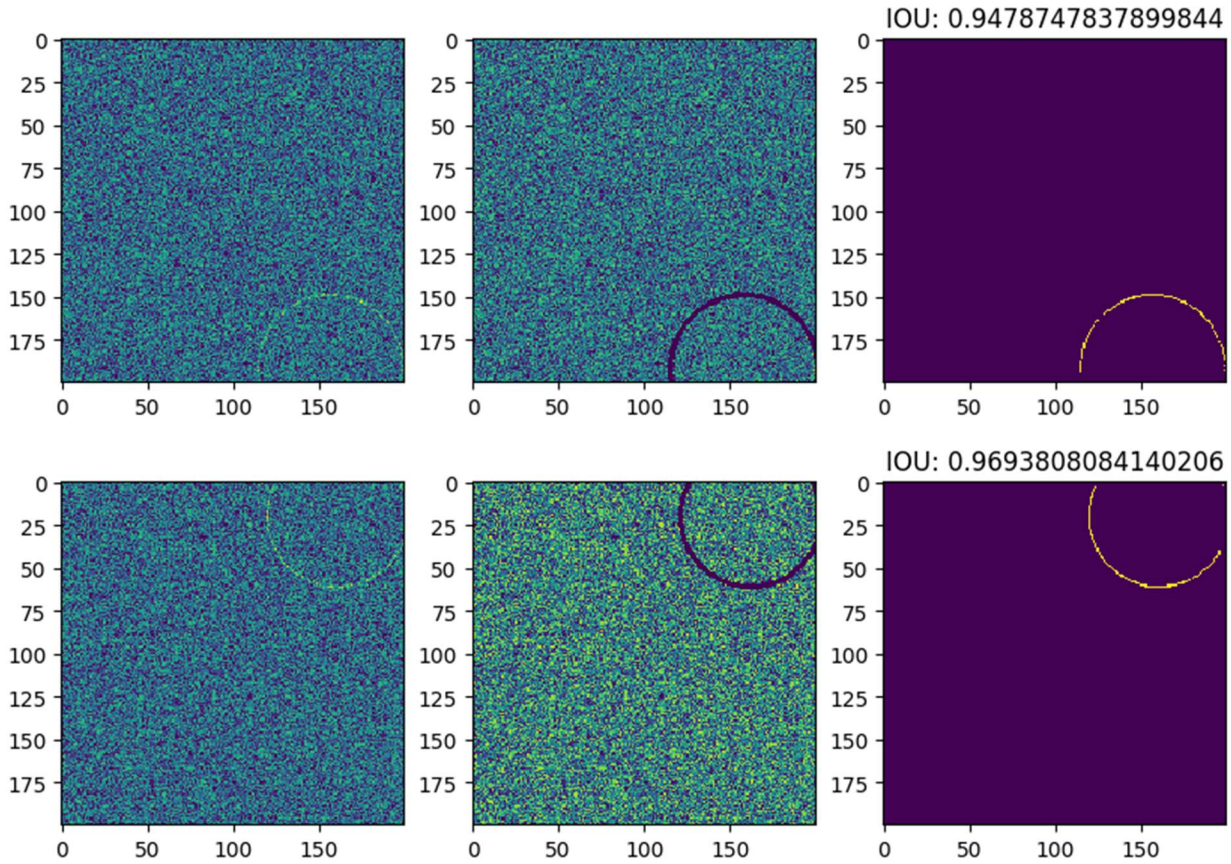
```
                # Draw the circle in the output image
                cv2.circle(img, (x, y), r, (0, 255, 0), 2)

        figure = plt.figure(figsize=(10, 10))
        plt.subplot(231), plt.imshow(images[i].squeeze())
        plt.subplot(222), plt.imshow(img.squeeze())

        plt.show()
        if save_path is not None or save_path != "":
            figure.savefig(save_path + "/result_" + str(i) + ".png",
dpi=100, bbox_inches='tight')
```

VISUAL RESULTS




IOU: 0.890532718436077
IOU: 0.8605690138392683
IOU: 0.9490420366704067


IOU: 0.9466764061358648
IOU: 0.815446829622851
IOU: 0.9245562130177517



3 Hot Dog or Not Hot Dog

We assume the network is fairly confident in its output scalar c (c > 0.99).

We are not given any information on the dataset the network was trained on. If the network was trained on a dataset of hot dogs, then if we gave the network a picture of a hot dog, it could correctly tell us that it is a hot dog; if we gave it a picture of something that is not a hot dog, then the network would be certain it is not a hot dog.

But imagine the network was trained on cat images, and the label for the image was, as before, hot dog or not hot dog. The data would be clearly mislabeled, so when the network would be asked to predict a picture, if we were to feed it a picture of a cat, it would say with certainty that is a hot dog; if we were to feed it a picture of a hot dog, it would say that it isn't a hot dog.

In conclusion, the neural networks are as smart as we can make them. They can easily be fooled using mislabeled data, or even **adversarial** data (look up adversarial training online).