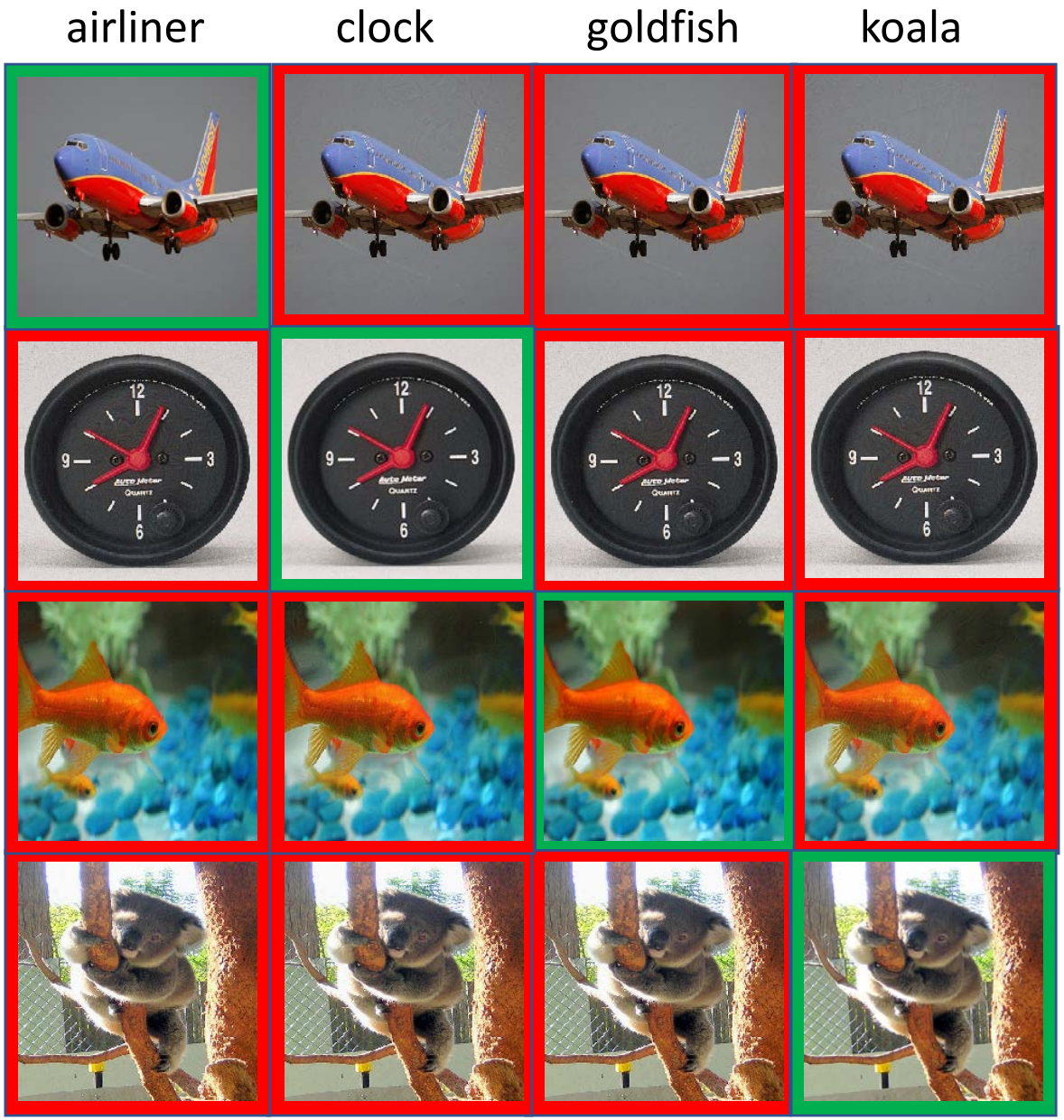
Gradient Ascent: Generating Model Inputs that Maximize Model Outputs

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

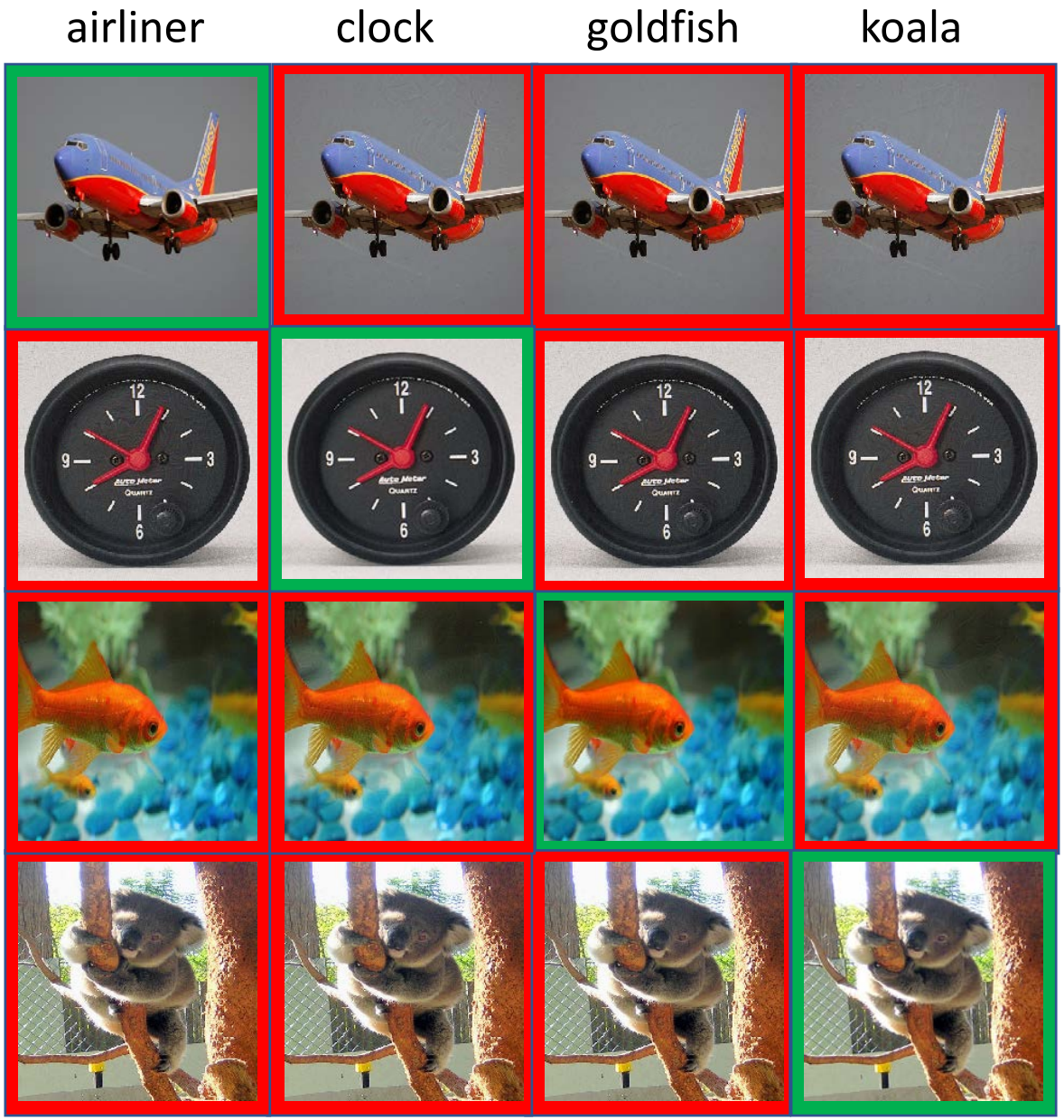
Application 1: Tricking a Model

- The images with green background are real.
  - Part of ImageNet.
  - Classified correctly by pre-trained model ResNet50V2.
- The images with red background are changed versions of the real images.
  - We used the method described in these slides to make changes that would trick ResNet50V2.
  - The heading of each column shows the class label that ResNet50V2 produced for the images in that column.



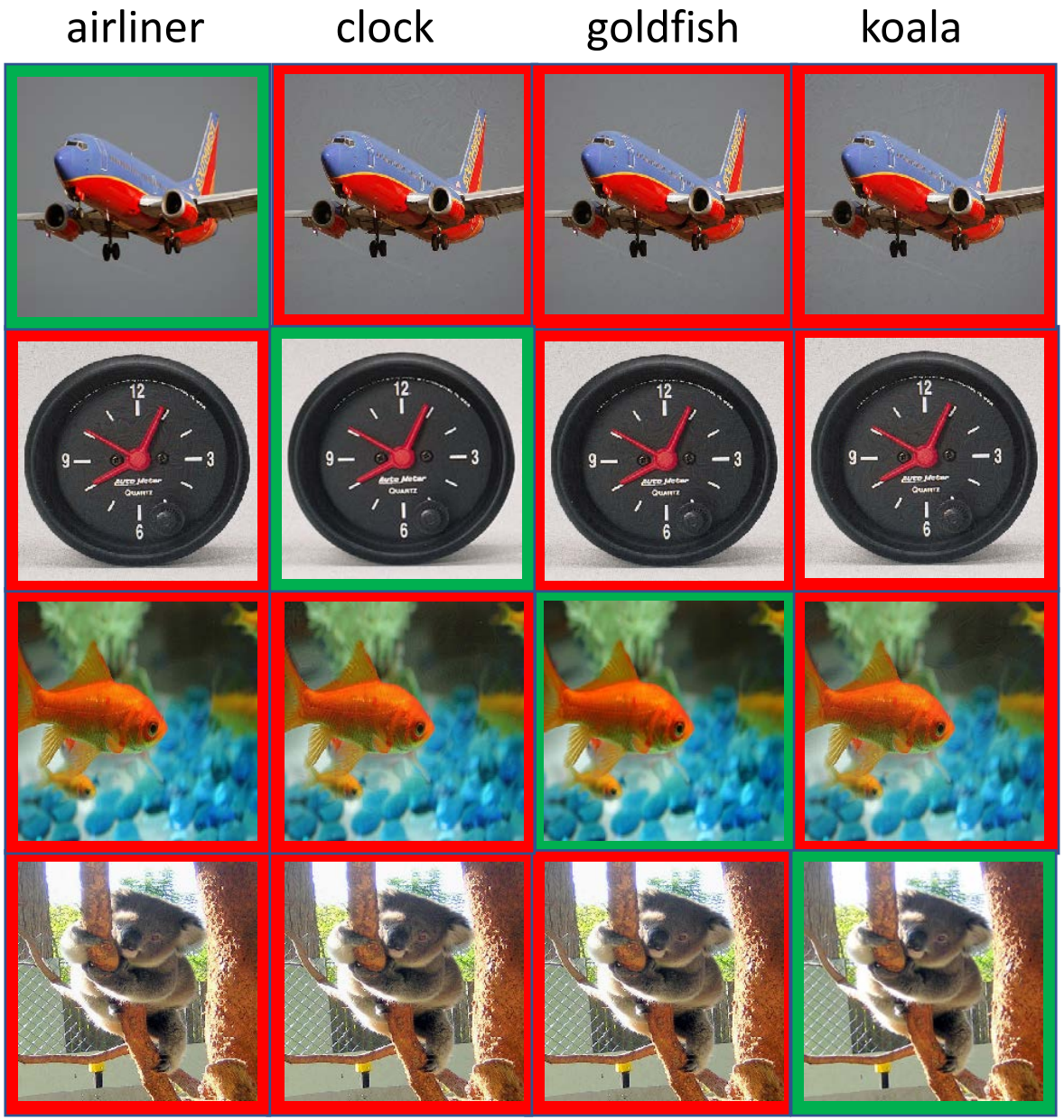
Application 1: Tricking a Model

- Why would it be useful to produce such changed images?



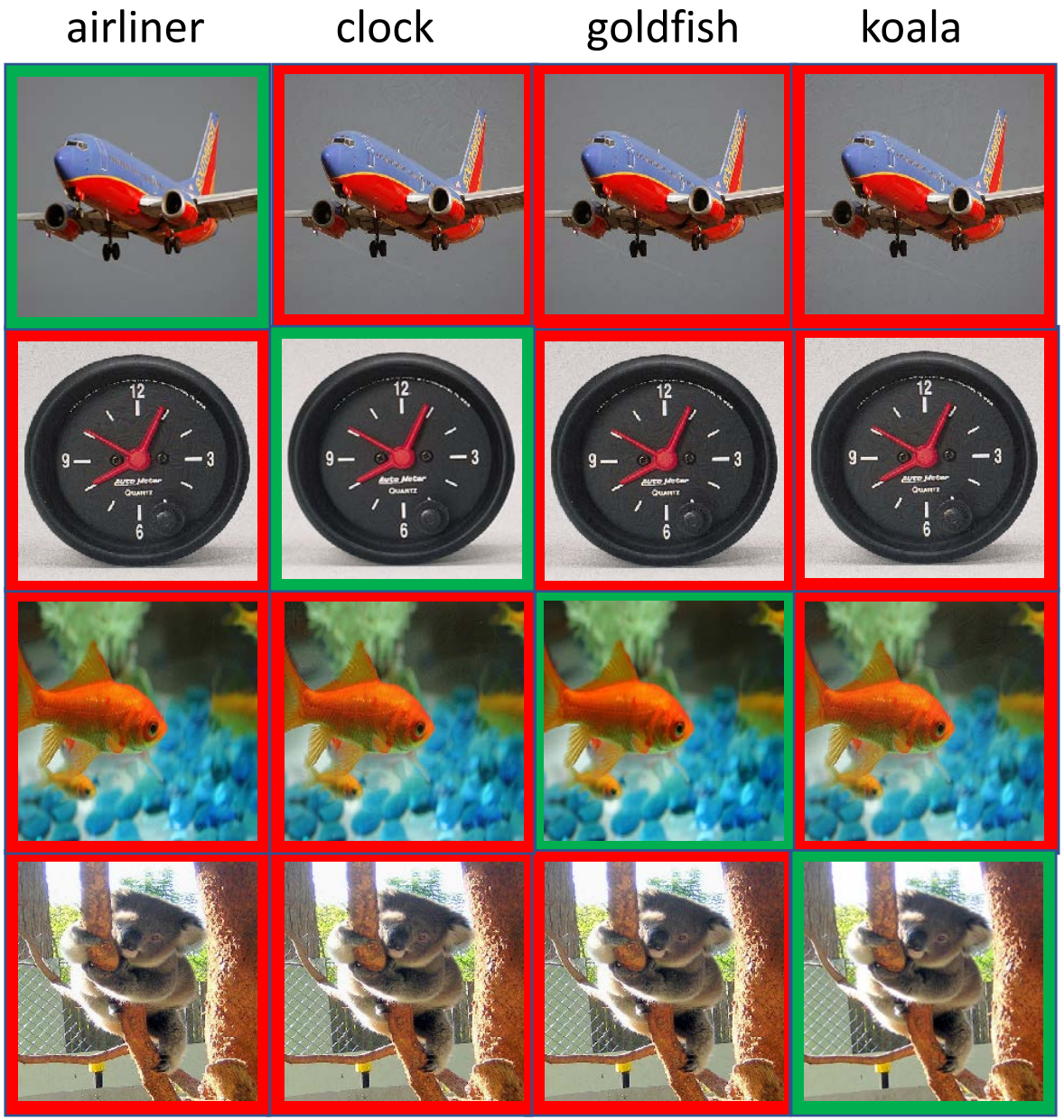
Application 1: Tricking a Model

- Why would it be useful to produce such changed images?
- We may actually want to fool a model, that is operated by adversaries.
- Or, we may want to gain intuition about how well these models match human intelligence.

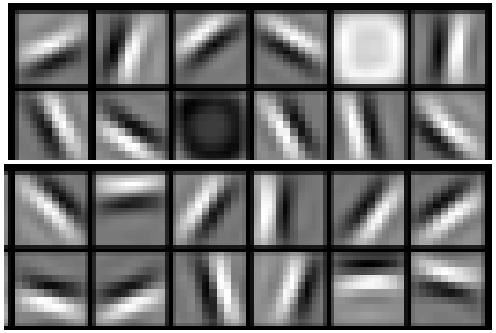


Application 1: Uses Besides Tricking

- Or, imagine that, instead of image classification, our model does mortgage approvals.
- If someone is declined, it can be useful to understand what minimal changes in their profile could have led to approval.

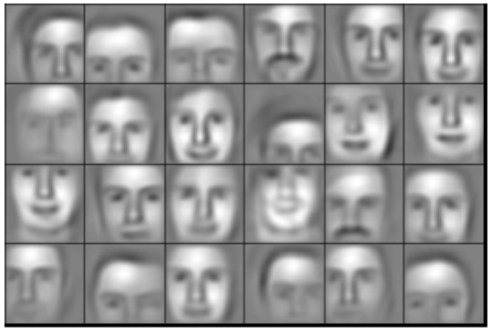


Application 2: CNN Visualization

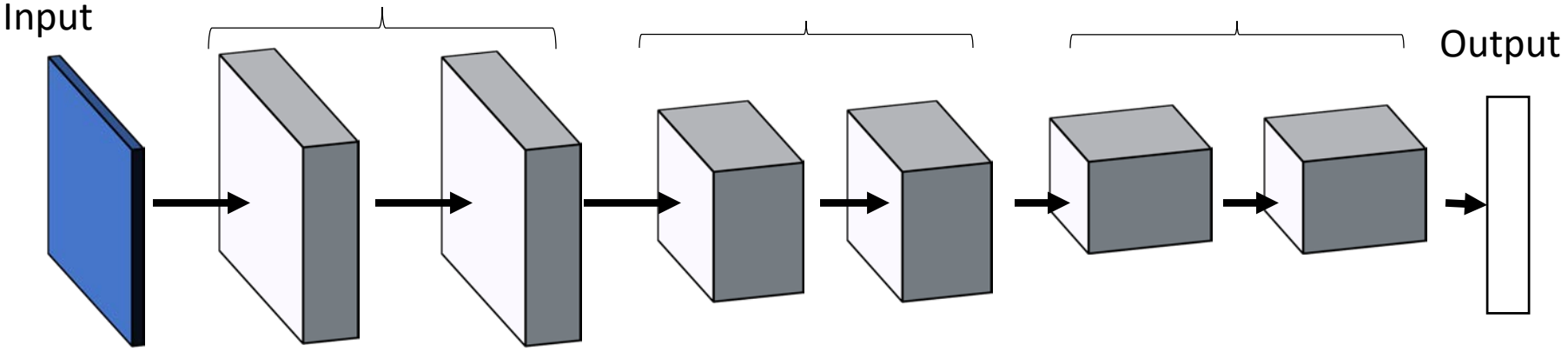


Early layers, extract simple features.

Middle layers, model shape parts.

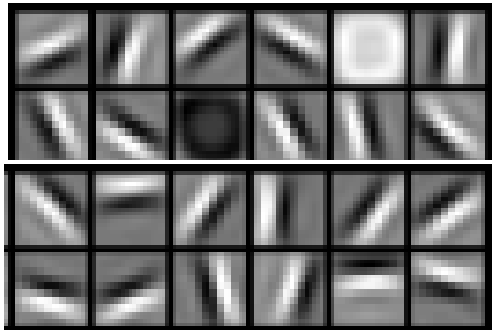


Last layers, model (almost) entire faces.



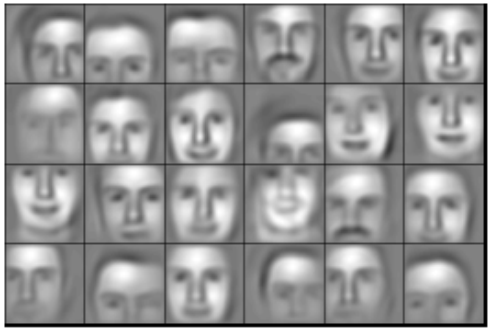
- (We have seen this before): Visualization of a neural network trained with face images.
  - Credit for feature visualizations: [Lee09] Honglak Lee, Roger Grosse, Rajesh Ranganath and Andrew Y. Ng., ICML 2009.

Application 2: CNN Visualization

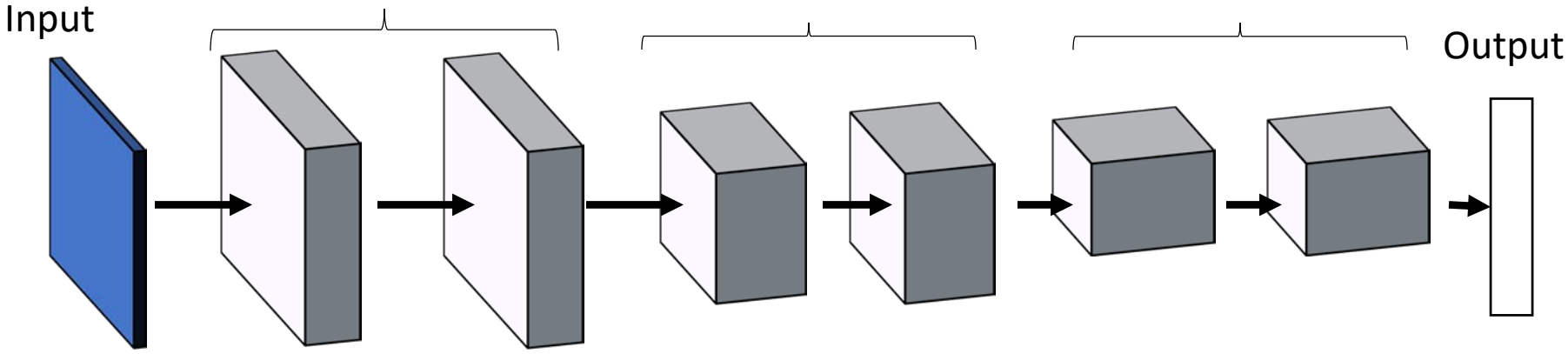


Early layers, extract simple features.

Middle layers, model shape parts.



Last layers, model (almost) entire faces.



- These slides explain how we can get such visualizations.
- Idea: for any layer, or any unit, we find the input image that maximizes the output of that layer or unit.

Generating Inputs that Trick a Model

- In our case study, we use a pre-trained model, available on Keras, called ResNet50V2.
- The model was trained on a subset of ImageNet called "ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012-2017 image classification and localization dataset".
  - This dataset can be downloaded from: https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data
  - However, it takes 167GB of disk space.
  - You do NOT need to download it to run our code.
- This dataset contains 1000 classes.

Example Images from Dataset

- Here are some example images from the dataset, and their associated class labels (out of the 1000 total class labels).

scorpion

robin

goldfish

koala

chimpanzee

airliner

palace

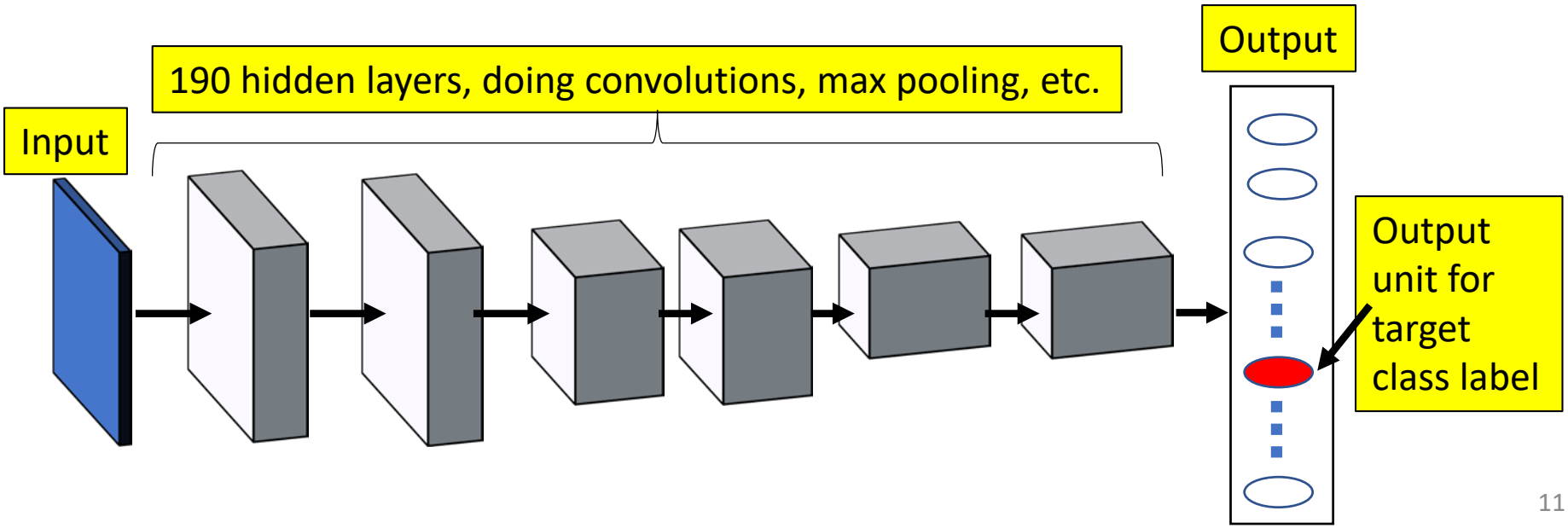
clock

Our Goal: Trick ResNet50V2

- Inputs to our system:
  - The model we want to trick: ResNet50V2 in our case.
  - A real image. For example: 
  - A target class label, that does NOT match the real image. For example: “goldfish”.
- System output: a changed image, with (hopefully) these two properties:
  - It looks similar to the real image.
  - It gets assigned the target class label by ResNet50V2.
  - For the example inputs, we get this output, which indeed gets classified as “goldfish”, and looks very similar to the real image.

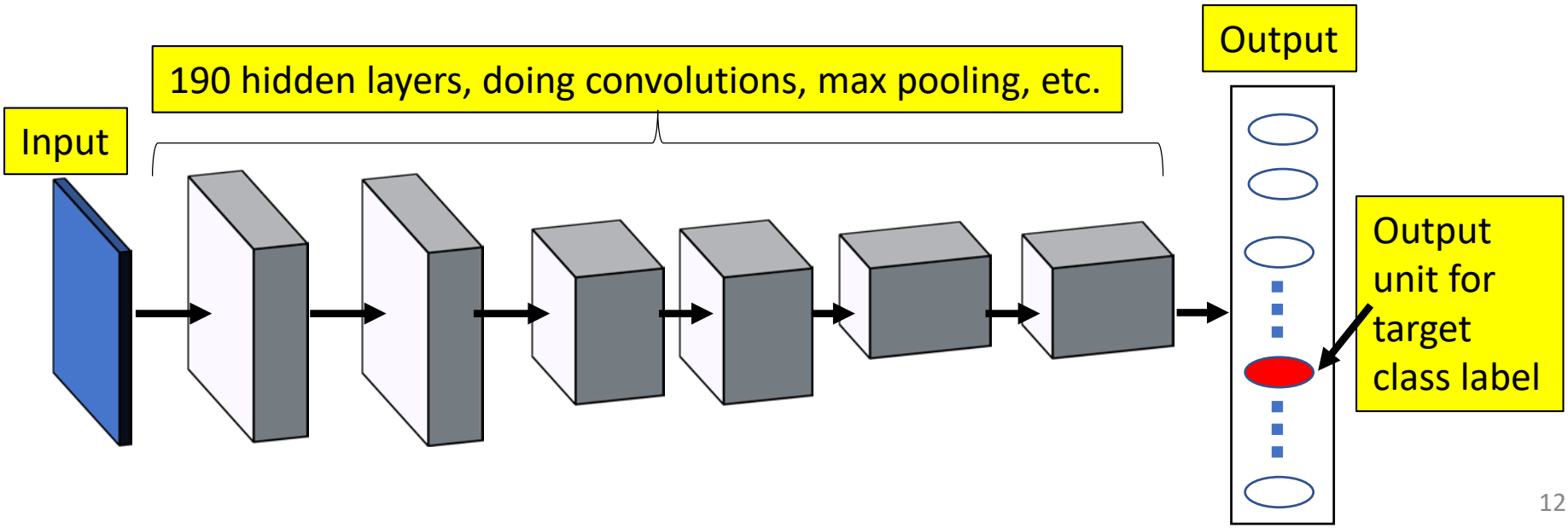
Method: Gradient Ascent

- The actual ResNet50V2 model has 192 layers.
- The drawing below shows a simplified version, keeping the details that are of interest here.



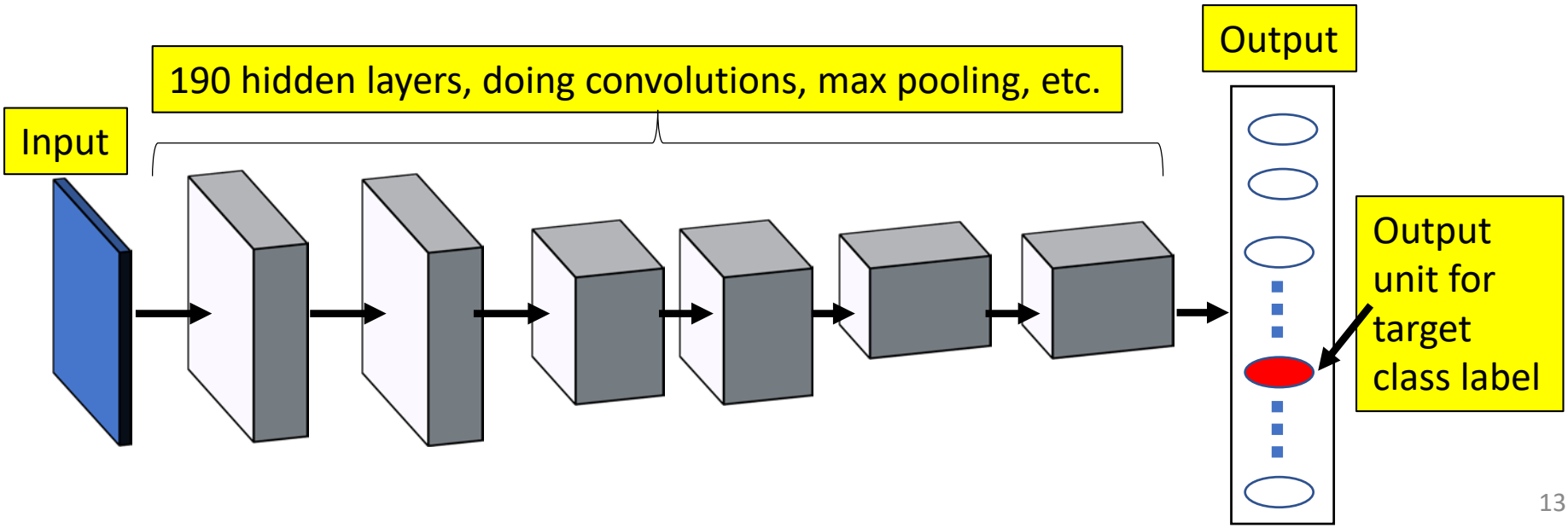
Method: Gradient Ascent

- The output unit shown in red is the one corresponding to the target class label.
  - The output of all output units is between 0 and 1, and sums up to 1.
- We want an image that **maximizes** the red unit's output.



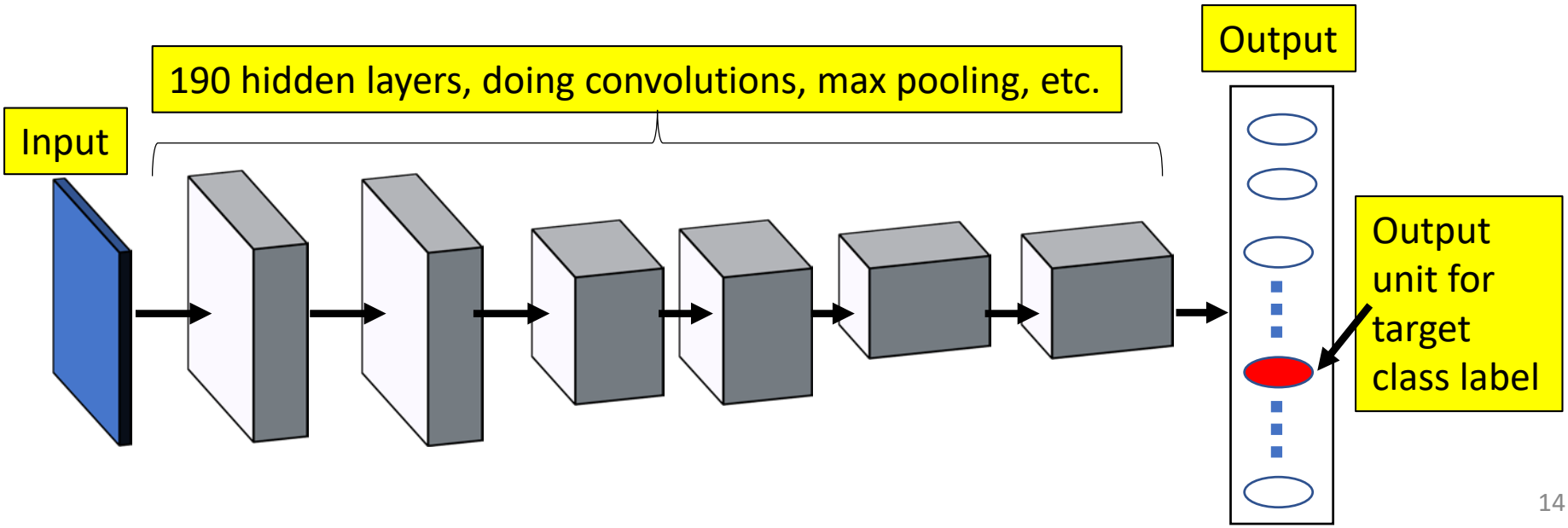
Method: Gradient Ascent

1. X = original image we are given as input.
2. Compute the output of the network for image X.
3. If output of the red unit > threshold, then we are done and we return X.
4. Compute the gradient of the red unit's output with respect to image X.
5. Update image X by **<u>adding</u>** to X a small vector in the direction of the gradient. Then, go back to step 2.



Method: Gradient Ascent

- Note that in step 5 we add to X a vector in the direction of the gradient.
- Why are moving towards the gradient, not in the opposite direction as in backpropagation?
- Because here we want to **maximize** a quantity (the red unit's output), and NOT to minimize a quantity (in backpropagation we minimize the loss function).
- Hence, the method here is called **gradient ascent**, not gradient descent.



Tensorflow Implementation

See files posted under today's lecture:

- adversarial_input.py: code implementing the approach we are discussing.
- adversarial_imagenet.zip contains inputs and outputs:
  - The data folder contains 24 real images from the ImageNet dataset.
  - The **generated_200_1_7** folder contains 288 images, produced using **learning_rate = 1**. Each image file follows format A_B_C.png, where:
    - A is the original filename.
    - B is the target class used in generating the image.
    - C is the result of ResNet50V2 on the image.
  - For 24 of those 288 images, the target class was the true class, and the result was equal to the corresponding real image stored in the data folder.
  - For the remaining 264 images, they succeeded in tricking the model in 90.5% of the cases (239 out of 264).

Tensorflow Implementation

- The **generated_200_10_7** folder contains 288 images, produced using **learning_rate = 10**. Each image file follows the same format A_B_C.png as explained in the previous slide.
- Again, 24 of those images were equal to real images in the data folder.
- For the remaining 264 images, they succeeded in tricking the model in 98.1% of the cases (259 out of 264).
- The success rate was higher than using **learning_rate = 1**.
- However, here the generated images look more noisy and more different than the corresponding real images, compared to using **learning_rate = 1**.

Loading the ResNet50V2 Model

```
model = keras.applications.ResNet50V2(weights="imagenet", include_top=True)
path = 'data/001_n01443537_goldfish_1.jpg'
img = image.load_img(path, target_size=(img_width, img_height))
img = image.img_to_array(img)
x = np.expand_dims(img, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
info = decode_predictions(preds, top=3)
print('Predicted:', info[0])
```

Output:

Predicted: [('n01443537', 'goldfish', 0.99999905), ('n02606052', 'rock_beauty', 3.0257326e-07), ('n01440764', 'tench', 1.5264698e-07)]

- The line in red loads the pre-trained ResNet50V2 model.
- The lines in green load and preprocess an image.
  - **expand_dims** is used to convert the shape of x from (224, 224, 3) to (1, 224, 224, 3), so that it looks like a batch of size 1.
  - **preprocess_input** is imported from the resnet_v2 package, to normalize the input as required by ResNet50V2.

Applying the ResNet50V2 Model

```
model = keras.applications.ResNet50V2(weights="imagenet", include_top=True)
path = 'data/001_n01443537_goldfish_1.jpg'
img = image.load_img(path, target_size=(img_width, img_height))
img = image.img_to_array(img)
x = np.expand_dims(img, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
info = decode_predictions(preds, top=3)
print('Predicted:', info[0])
```

Output:

Predicted: [('n01443537', 'goldfish', 0.99999905), ('n02606052', 'rock_beauty', 3.0257326e-07), ('n01440764', 'tench', 1.5264698e-07)]

- The line in red applies the model to x and gets the output.
- **decode_predictions** is imported from the resnet_v2 package. It converts the model output for each of the top 3 classes to:
  - The imagenet class label.
  - An English version of the class label.
  - The output of the corresponding output unit.

The **make_adversarial** Function

```
def make_adversarial(class_index, img, thr=0.99, learning_rate=1):
    iterations = 200
    for iteration in range(iterations):
        previous = img
        gain, img = gradient_ascent_step(img,
                model, class_index, learning_rate)
        if (gain > thr):
            break

return gain, previous.numpy()[0]
```

- The **make_adversarial** function is the top-level function implementing what we want to do.
- It takes an input image **img**, and it changes it to a result image that (hopefully):
  - looks similar to the input image.
  - gets classified by the model as belonging to the target class label, specified by **class_index**.

The **make_adversarial** Function

```
def make_adversarial(class_index, img, thr=0.99, learning_rate=1):
    iterations = 200
    for iteration in range(iterations):
        previous = img
        gain, img = gradient_ascent_step(img,
                model, class_index, learning_rate)
        if (gain > thr):
            break

return gain, previous.numpy()[0]
```

- Inputs:
  - **class_index** is the index of the target class.
  - **img** is the input image, typically a real image.
  - **thr** specifies the minimum value that we want the output unit of the target class to reach.
  - **learning_rate** specifies how much in the direction of the gradient we should move at each step.

The **make_adversarial** Function

```
def make_adversarial(class_index, img, thr=0.99, learning_rate=1):
    iterations = 200
    for iteration in range(iterations):
        previous = img
        gain, img = gradient_ascent_step(img,
                model, class_index, learning_rate)
        if (gain > thr):
            break

return gain, previous.numpy()[0]
```

- The for loop calls repeatedly **gradient_ascent_step**, to do the gradient ascent. We will explain this shortly.
- **gradient_ascent_step** returns **gain**, which is the current output of the model for the target class, and an updated **img**.
- Note that we break out of the loop if **gain** is high enough.

The **make_adversarial** Function

```
def make_adversarial(class_index, img, thr=0.99, learning_rate=1):
    iterations = 200
    for iteration in range(iterations):
        previous = img
        gain, img = gradient_ascent_step(img,
                model, class_index, learning_rate)
        if (gain > thr):
            break

return gain, previous.numpy()[0]
```

- We return the final **gain**, and the image produced by gradient ascent.
- Note that the result image that we return is **previous**, not **img**.
- The reason is that (as we will see shortly), the **gain** value returned by **gradient_ascent_step** was attained by **previous**.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, model, class_index, learning_rate):
with tf.GradientTape() as tape:
    tape.watch(img)
    outputs = model(img)
    gain = outputs[:, class_index]

  # Compute gradients.
  grads = tape.gradient(gain, img)
  # Normalize gradients.
  grads = tf.math.l2_normalize(grads)
  img += learning_rate * grads
  return gain, img
```

- This function does a single step of gradient ascent.
- Inputs:
  - **img**: the current version of the image that we are making changes to.
  - **model**: the model we want to trick.
  - **class_index**: the target class.
  - **learning_rate**: specifies how much to change the image.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, model, class_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        outputs = model(img)
        gain = outputs[:, class_index]

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

- The lines in red highlight how we compute gradients in Tensorflow.
  - In most of our code this semester we did NOT use this approach, because we relied on Keras to hide these details from us.
- We use **GradientTape** to tell Tensorflow to keep track of gradients when it makes computations.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, model, class_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        outputs = model(img)
        gain = outputs[:, class_index]

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

- **tape.watch(img)** tells Tensorflow to keep track of gradients with respect to the values of **img**.
  - Tensorflow needs to be told which gradients to compute, because if it computed all gradients it would be too inefficient.
  - Some values are watched by default, such as a model's trainable weights.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, model, class_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        outputs = model(img)
        gain = outputs[:, class_index]

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

- The lines in red compute the output of the model given **img** as input.
- Then, we define variable **gain** to be the output of the specific output unit corresponding to the target class.
- We call it **gain** as it is the quantity we want to maximize.
  - Contrast to the term **loss**, used for the quantity we want to minimize in backpropagation.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, model, class_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        outputs = model(img)
        gain = outputs[:, class_index]

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

- Note that the model output is computed within the **GradientTape** scope, so Tensorflow keeps track of gradients of these computations with respect to **img**.
- Once these computations are done, the line in green calls **tape.gradient** to retrieve the gradient vector of **gain** with respect to **img**.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, model, class_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        outputs = model(img)
        gain = outputs[:, class_index]

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

- The lines in red update the image by moving it in the direction of the gradient vector.
  - **tf.math.l2_normalize** normalizes the gradient vector, so that the sum of the squares of its values is equal to 1.
- Notice that the **img** version that we return is NOT the one that achieved the **gain** that we return, because it has been updated.
  - That is why in **make_adversarial** we return the previous image.

Running the Code

```
path = ' data/001_n01443537_goldfish_1.jpg'
img0 = image.load_img(path,
                      target_size=(224, 224))
img = image.img_to_array(img0)
x = np.expand_dims(img, axis=0)
x = preprocess_input(x)
tx = tf.convert_to_tensor(x)
thr = 0.7
learning_rate = 1
(loss, result) = make_adversarial(105, tx, thr=thr,
                      learning_rate=learning_rate)
```

- The lines in red load the real image from a file.
- The lines in green preprocess the image:
  - Convert to numpy array.
  - Make 4D (so that the image becomes a batch of one image).
  - preprocess_input is part of the ResNet50V2 package, it normalizes the input image as needed.
  - The last line calls **make_adversarial** with appropriate parameters.

Running the Code

```
keras.preprocessing.image.save_img("0.png",
                                   result)
img0 = image.load_img("0.png",
                      target_size=(224, 224))
img = image.img_to_array(img0)

rx = np.expand_dims(img, axis=0)
rx = preprocess_input(rx)

preds = model.predict(x)
print('Prediction on original:',
      decode_predictions(preds, top=3)[0])
preds = model.predict(rx)
print('Prediction on adversarial:',
       decode_predictions(preds, top=3)[0])
```

- The first line saves the image to a file.
- The second line loads the image from a file.
- Why are we saving and reloading?

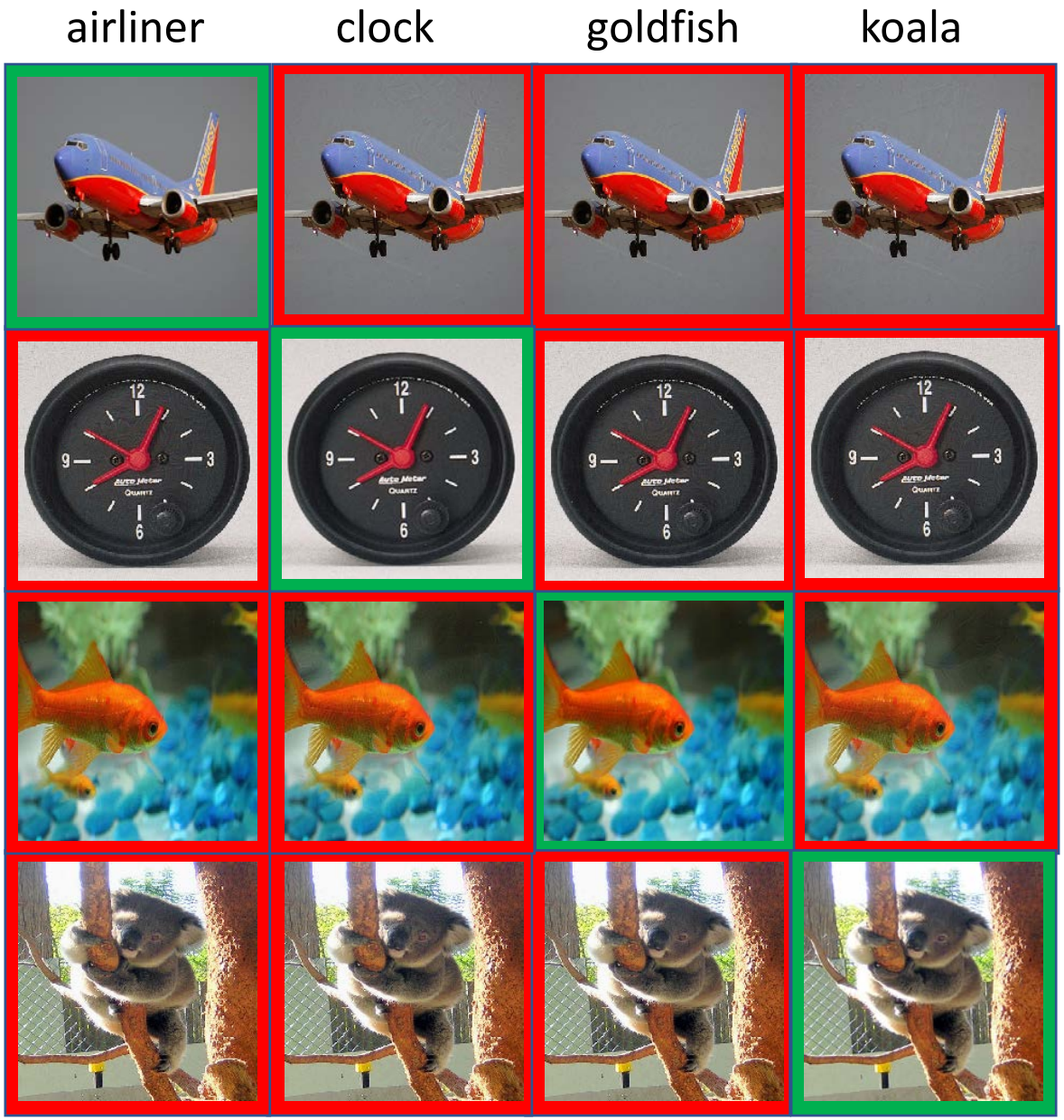
Running the Code

```
keras.preprocessing.image.save_img("0.png",
                                   result)
img0 = image.load_img("0.png",
                      target_size=(224, 224))
img = image.img_to_array(img0)

rx = np.expand_dims(img, axis=0)
rx = preprocess_input(rx)

preds = model.predict(x)
print('Prediction on original:',
      decode_predictions(preds, top=3)[0])
preds = model.predict(rx)
print('Prediction on adversarial:',
       decode_predictions(preds, top=3)[0])
```

- Why are we saving and reloading?
  - The generated image that tricked the model was real-valued, with values in a range we are not quite sure of.
  - We started with values in the [-1, 1] range, but the repeated changes made by gradient ascent may have changed that.
  - When we save and reload, we get values in the [-1, 1] range.
  - There is no guarantee that the reloaded image will fool the model, so we doublecheck.

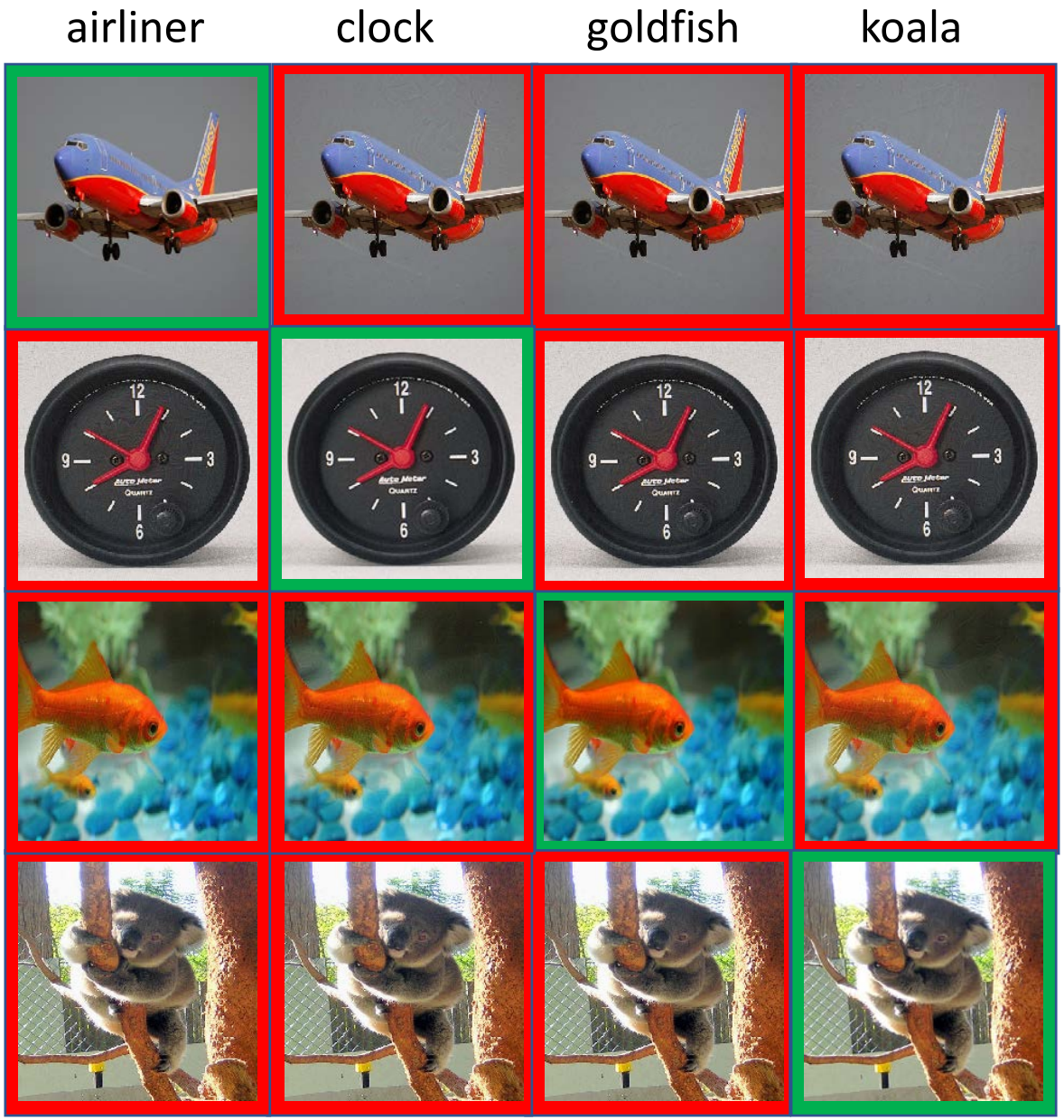
Results

- These are the results we saw at the beginning.
- The four images with green background are real.
  - Part of ImageNet.
  - Classified correctly by pre-trained model ResNet50V2.



Results

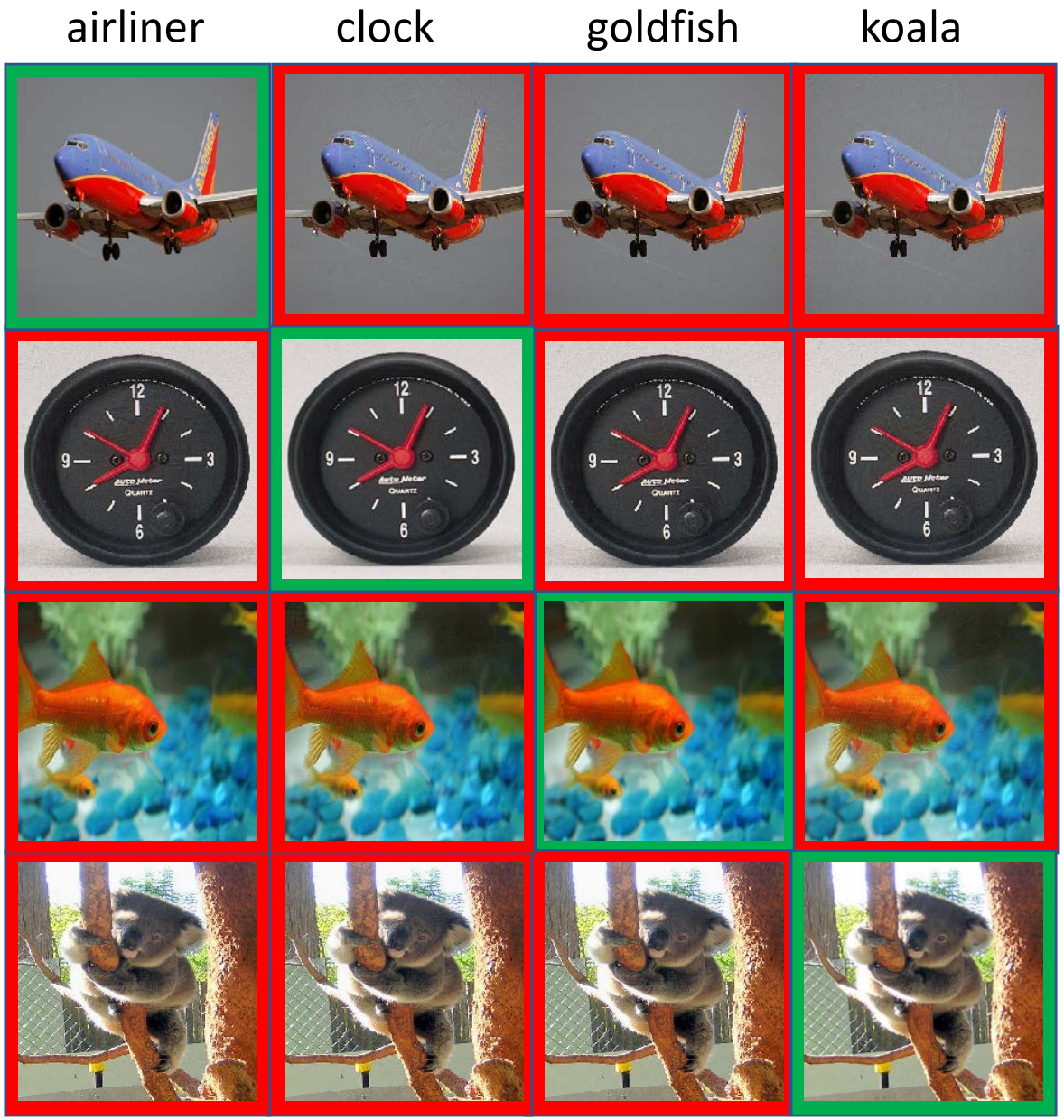
- The 12 images with red background are results of **make_adversarial**, with the target class equal to the column heading.
  - For example, for images in the left column, the target class was “airliner”.
- For each of these 12 results, the model output was indeed the target class, so **make_adversarial** was successful in tricking the model.



Results

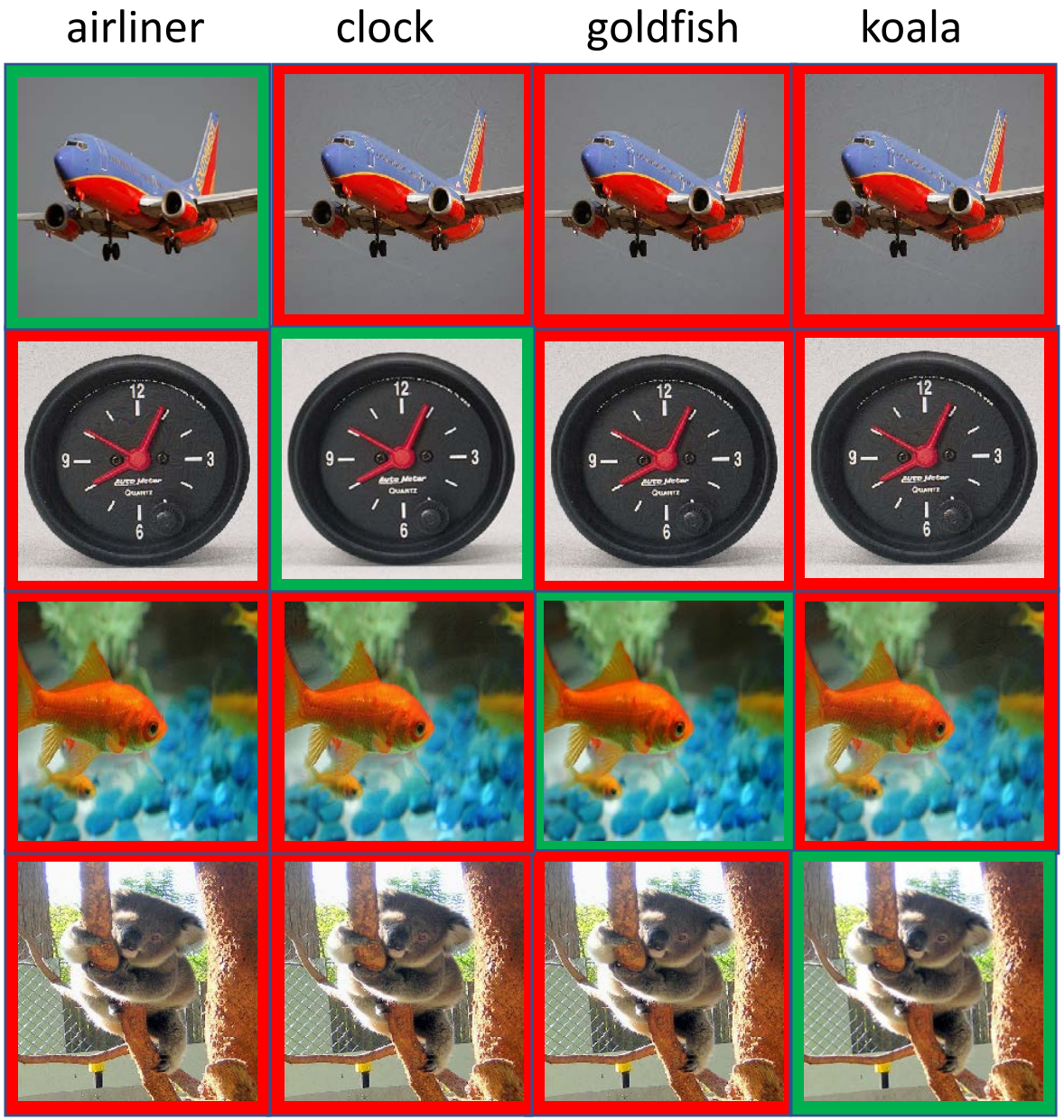
In the posted file adversarial_imagenet.zip:

- The data folder contains 24 real images.
- The generated_200_1_7 folder contains 264 images, generated by giving to **make_adversarial** each combination of a real image and a target class (out of 11 target classes).
- Out of those 264 images, for 239 the model indeed produced the target class as output.
- Success rate: 90.5%.



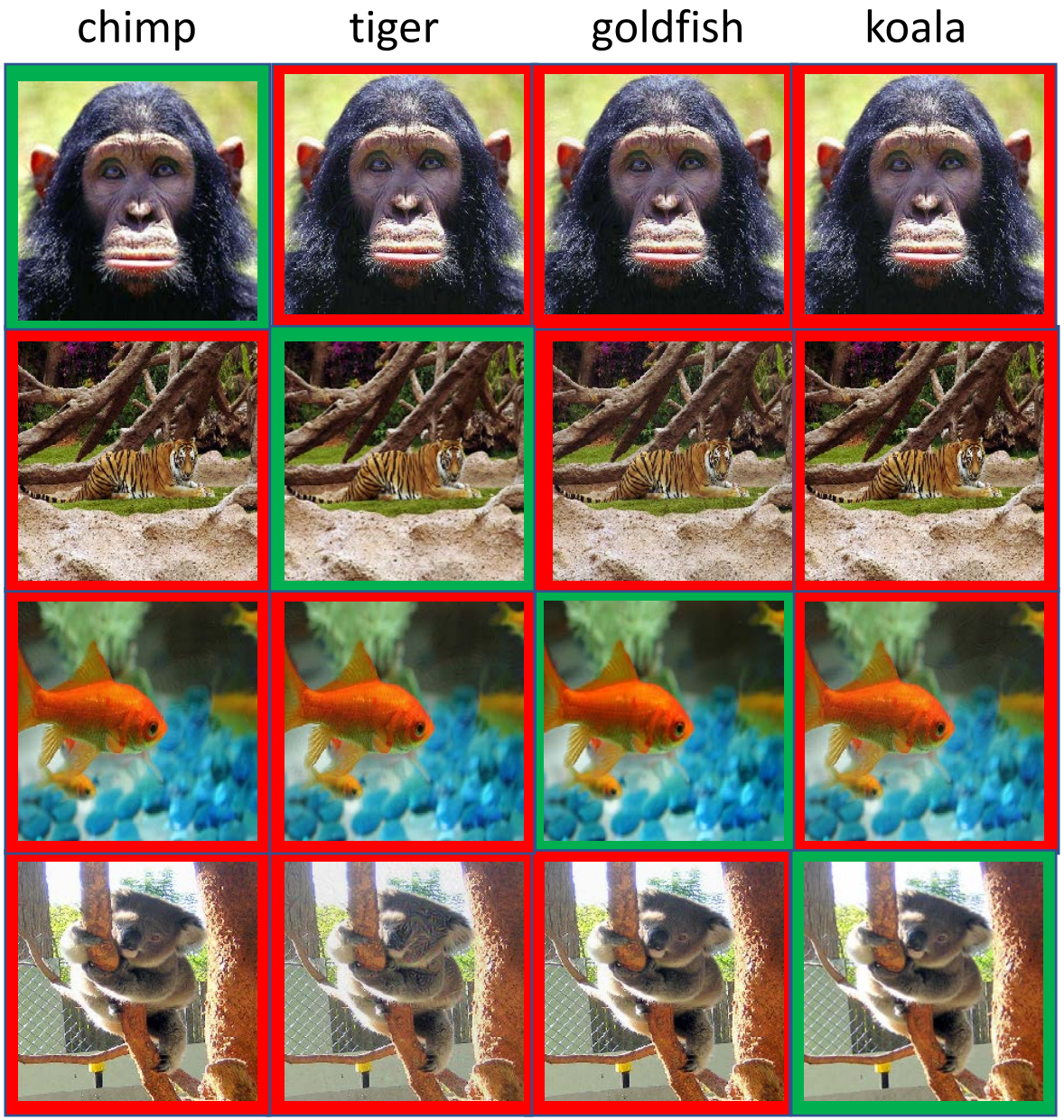
Results

- Minor note: the **generated_200_1_7** folder in adversarial_imagenet.zip actually contains 288 images, but 24 of those are the real images from the **data** folder.
- That is why we only use 264 images in our evaluation.



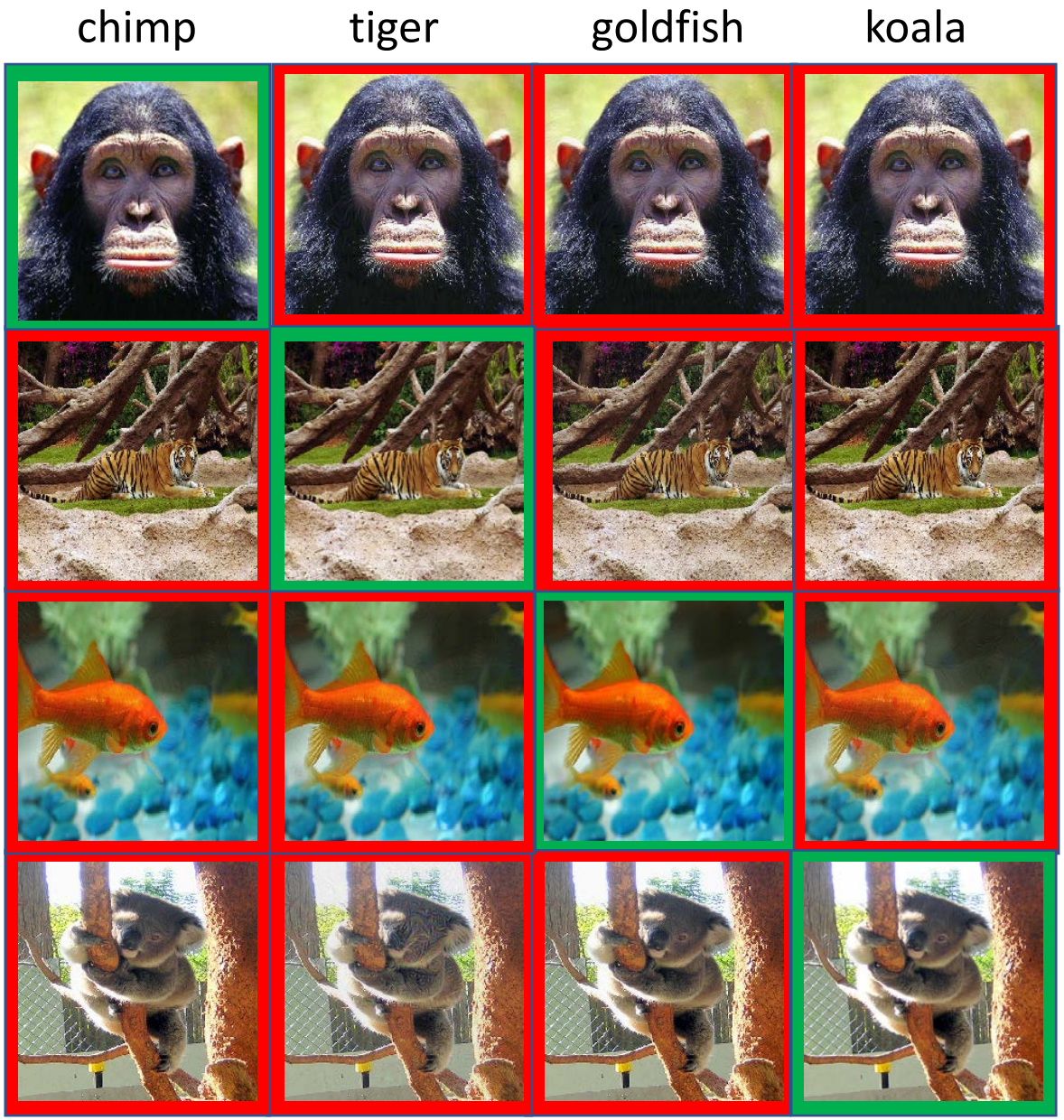
More Results

- Here we see how images of four animals are changed to make the model produce wrong classifications.



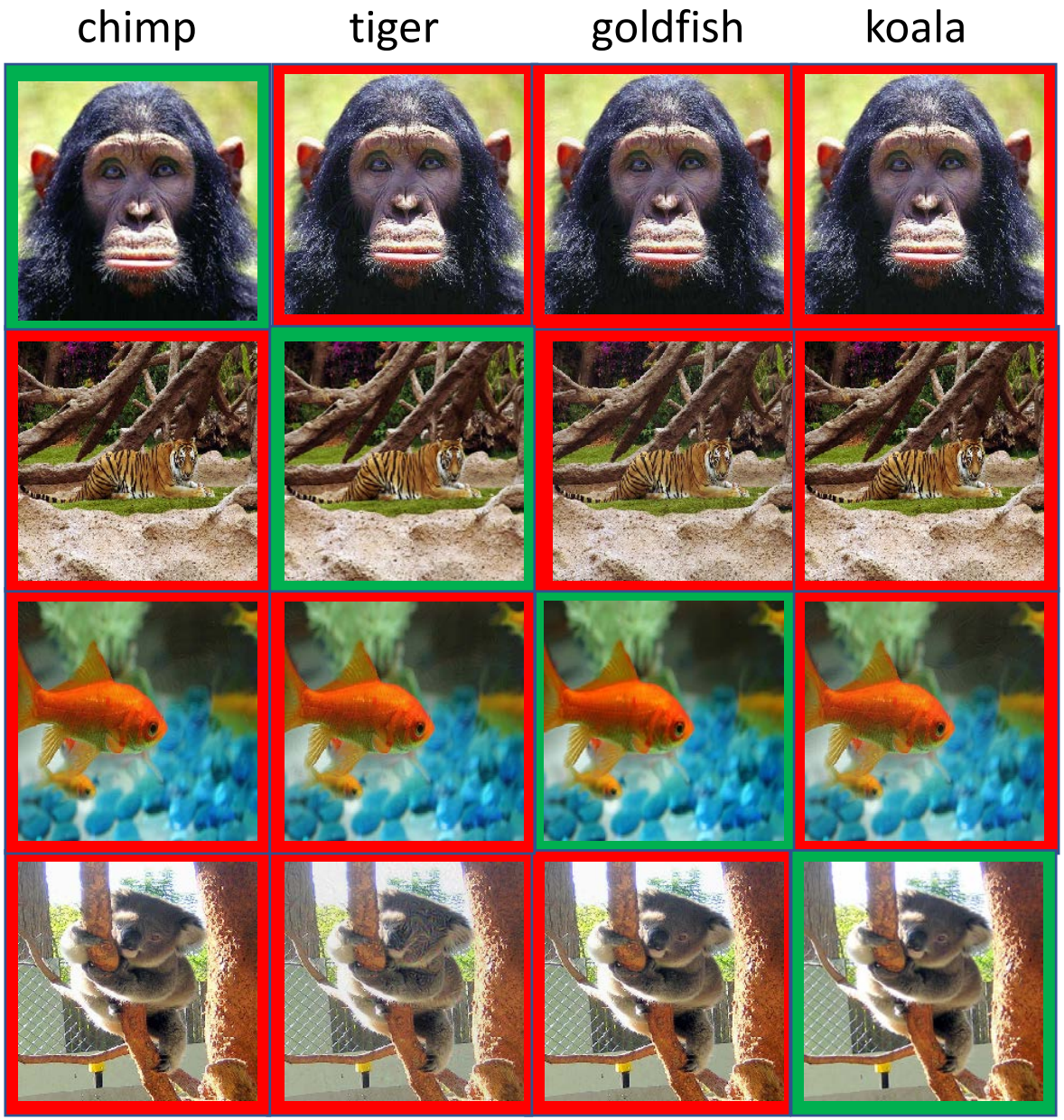
More Results

- Minor note: the koala-to-tiger image (4th row, 2nd column) was generated with **learning_rate = 10**, because the result with **learning_rate = 1** did not trick the model.
- Images generated with **learning_rate = 10** are in the generated_200_10_7 folder of adversarial_imagenet.zip



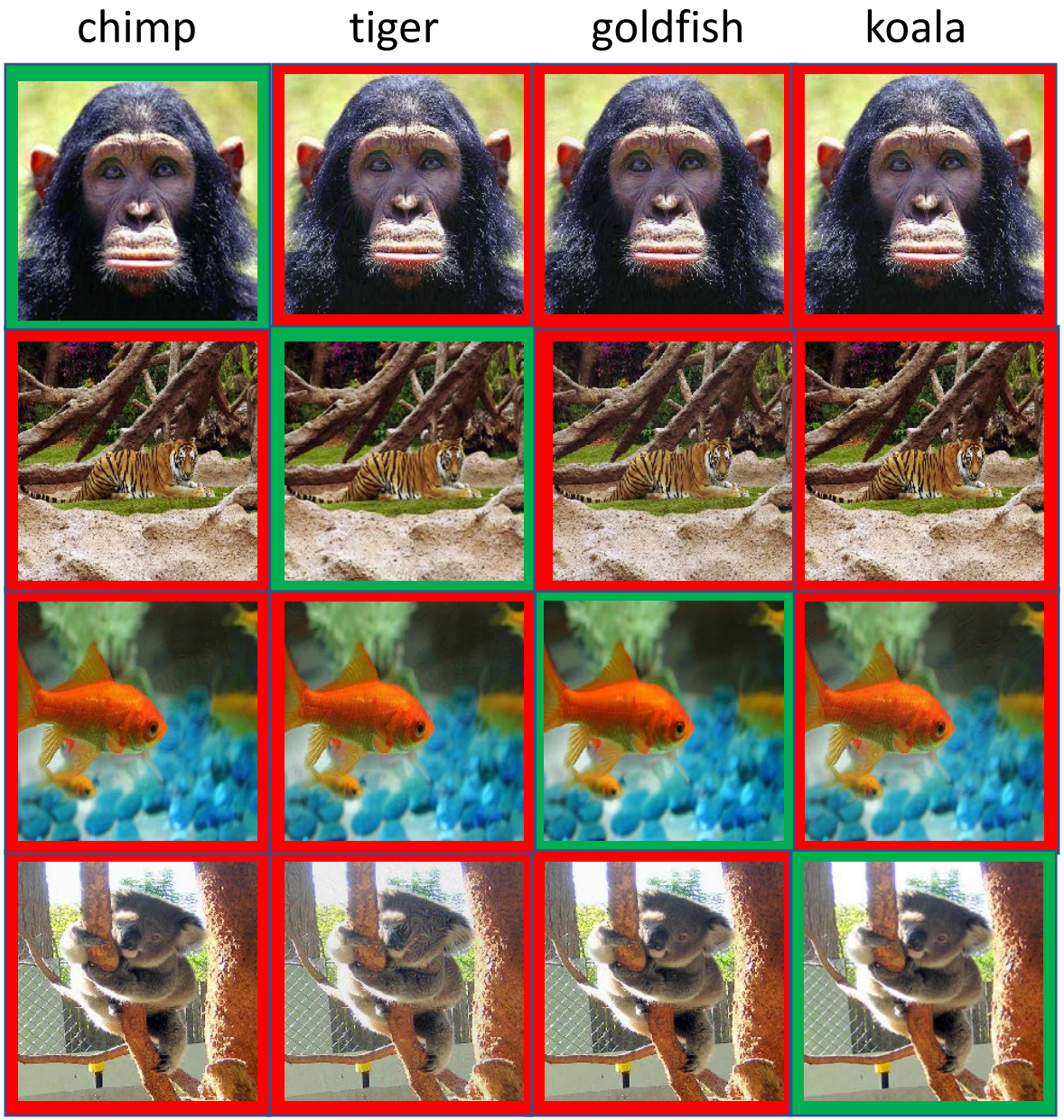
Observations

- The generated images may look noticeably different with respect to the original images.
- At the same time, often the differences are small and hard to notice.
- Still, these are all images where a human would easily recognize the object.



Observations

- These results offer some intuition about how a model "understands" the concepts it has been trained to recognize.
  - ResNetV250 had an overall accuracy of 76%, on 1000 classes.
  - Still, as these results show, it has limited understanding of what a clock or a goldfish really looks like.



Comparing Images Visually

- These two slides contain two images of a goldfish.
- One is real, and classified as a goldfish by ResNet50V2.
- One was generated by our model, and is classified as a chimpanzee by ResNet50V2.
- Can you tell which one is real?

Comparing Images Visually

- These two slides contain two images of a goldfish.
- One is real, and classified as a goldfish by ResNet50V2.
- One was generated by our model, and is classified as a chimpanzee by ResNet50V2.
- Can you tell which one is real?

Comparing Images Visually

- Answer: the first image (shown on the left here) is the real one.
- On the right, you can see a faint “ghost” of a chimp’s upper face.

Comparing Images Visually

- These two slides contain two images of a chimpanzee.
- One is real, and classified as a chimpanzee by ResNet50V2.
- One was generated by our model, and is classified as a goldfish by ResNet50V2.
- Can you tell which one is real?

Comparing Images Visually

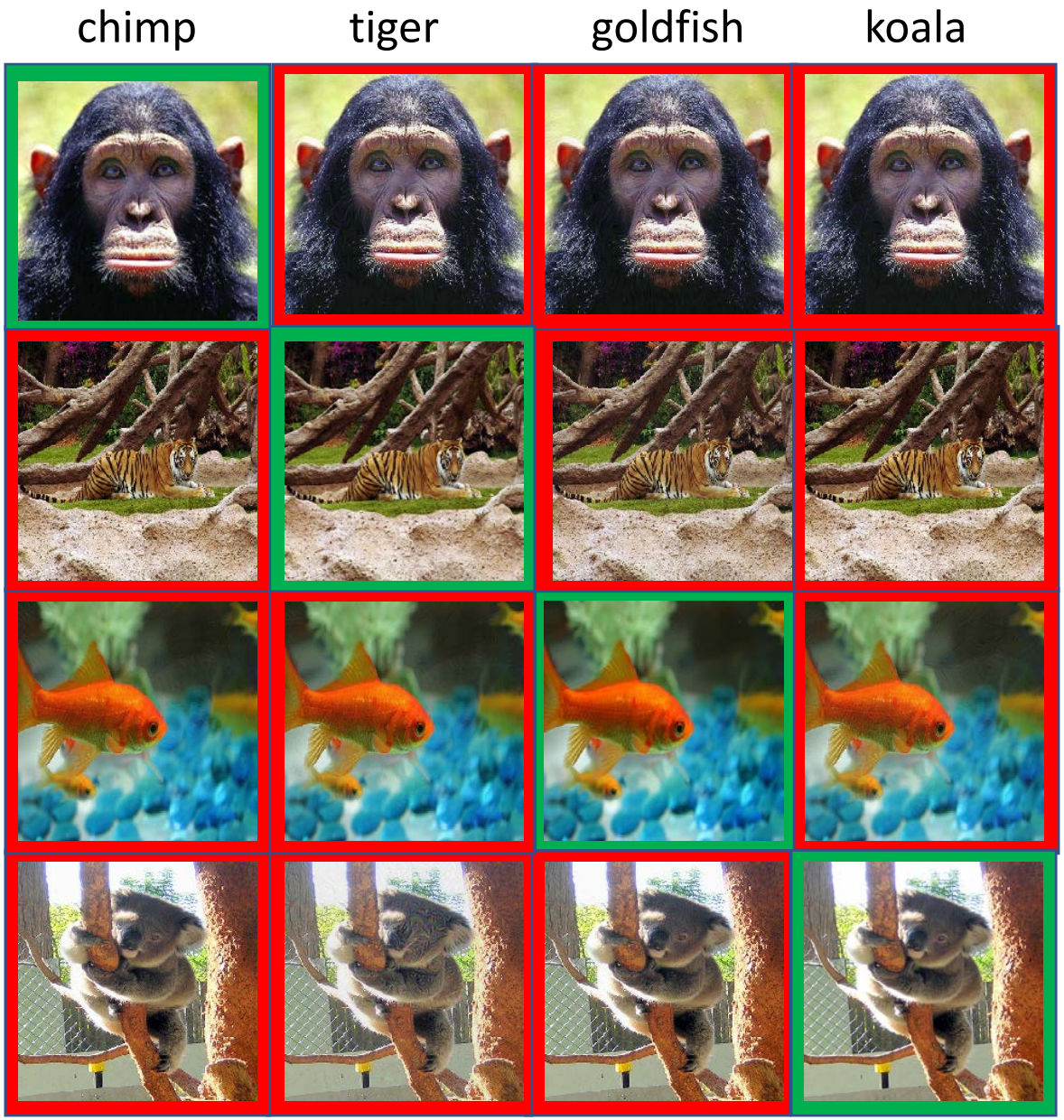
- These two slides contain two images of a chimpanzee.
- One is real, and classified as a chimpanzee by ResNet50V2.
- One was generated by our model, and is classified as a goldfish by ResNet50V2.
- Can you tell which one is real?

Comparing Images Visually

- Answer: the first image (shown on the left here) is the real one.

Measuring Quality of Results

- We can define two measures of performance:
  - Percentage of source images that we change successfully so that they get misclassified.
  - Average squared (or absolute) difference between original images and changed images.



Further Reading

- Here are some good starting points if you want to learn more about generating adversarial inputs, and more sophisticated methods for doing it.
- **"Intriguing properties of neural networks."**

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob Fergus. ICLR 2014
https://arxiv.org/pdf/1312.6199.pdf

- **"Explaining and Harnessing Adversarial Examples."**

Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy. ICLR 2015.

https://arxiv.org/pdf/1412.6572.pdf

- **"A survey on adversarial attacks and defences."**

Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay,

Debdeep Mukhopadhyay. CAAI Transactions on Intelligence Technology, 2021.

https://ietresearch.onlinelibrary.wiley.com/doi/epdf/10.1049/cit2.12028

Visualizing Behavior of Hidden Layers

- So far, we have seen how we can change images to maximize the response of a specific output unit.
- The same idea can be applied to hidden units as well.
- Why would it be useful to find images that maximize the response of a specific hidden unit?
  - Such images can provide intuition about the type of patterns that that specific input unit has learned to identify.

Visualizing a Convolutional Layer

- In Keras, a Conv2D layer applies a certain number of 2D filters to an image.
- We can write code that:
  - Chooses one of the Conv2D layers in a model.
  - Chooses one of the filters in the chosen Conv2D layer.
  - Generates an image that maximizes the output of that filter.
- We will apply this to a simple convolutional neural network trained on the MNIST dataset.
- The code for this is posted on the website, as file gradient_ascent_mnist.py.

Our CNN Model

```
model = keras.Sequential(    [
      keras.Input(shape=input_shape),
      keras.layers.Conv2D(32, kernel_size=(3, 3),
                                 activation="relu"),
      keras.layers.MaxPooling2D(pool_size=(2, 2)),
      keras.layers.Conv2D(64, kernel_size=(3, 3),
                                 activation="relu"),
      keras.layers.MaxPooling2D(pool_size=(2, 2)),
      keras.layers.Flatten(),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(number_of_classes,
                               activation="softmax"),
   ])
```

- We used this model early in the semester.

Our CNN Model

```
filename = "mnist_cnn_1.keras"
model = keras.models.load_model(filename)
model.summary()
```

- Here we load a model that has already been trained on MNIST, and achieves 99.27% accuracy on the test set.

```
Output:

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320
 max_pooling2d_6 (MaxPooling2D) (None, 13, 13, 32)     0
 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496
 max_pooling2d_7 (MaxPooling2D) (None, 5, 5, 64)       0
 flatten (Flatten)           (None, 1600)              0
 dropout (Dropout)           (None, 1600)              0
 dense (Dense)               (None, 10)                16010
=================================================================
```

Selecting the Layer to Visualize

```
layer_name = "conv2d"
layer = model.get_layer(name=layer_name)
truncated_model = keras.Model(inputs=model.inputs, outputs=layer.output)
```

- From the output of model.summary() in the previous slide, we see that the first convolutional layer is named “conv2d”.
- We want to visualize the behavior of that layer.
- The line in red gets that layer from the model.
- The line in green uses the Functional API to create a truncated model, that:
  - Takes the same input as the original model.
  - Produces the output of the selected layer as final output.

The **visualize_filter** Function

```
def visualize_filter(filter_index):
    iterations = 30
    learning_rate = 10.0
    img = initialize_image()
    for iteration in range(iterations):
        gain, img = gradient_ascent_step(img,
                                         filter_index, learning_rate)

    # Decode the resulting input image
    img = deprocess_image(img[0].numpy())
    return gain, img

def initialize_image():
    img = tf.random.uniform((1, 28, 28, 1))
    return img * 0.25
```

- The **visualize_filter** function is the top-level function.
  - It follows the same ideas as the **make_adversarial** function we saw earlier.
- Key difference from **make_adversarial:**
  - In **make_adversarial**, the initial image is given as an input argument.
  - Here, the initial image is created within the function (see lines in red), as a random image.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, filter_index,
                                    learning_rate):
  with tf.GradientTape() as tape:
    tape.watch(img)
    activation = truncated_model(img)
    filter_activation = activation[:, 2:-2, 2:-2,
                                   filter_index]
    gain = tf.reduce_mean(filter_activation)

  # Compute gradients.
  grads = tape.gradient(gain, img)
  # Normalize gradients.
  grads = tf.math.l2_normalize(grads)
  img += learning_rate * grads
  return gain, img
```

- Again, there is only a minor difference (shown in red) from the **gradient_ascent_step** function we used earlier, for generating adversarial inputs.
  - The difference is how we compute the gain, i.e., the quantity that we want to maximize.
  - In generating adversarial inputs, the gain was based on the output of the output unit corresponding to the target class.
  - Here, the filter produces a 2D array as output. We use the mean value of that output as the gain.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, filter_index,
                                    learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        activation = truncated_model(img)
        filter_activation = activation[:, 2:-2, 2:-2,
                                       filter_index]
        gain = tf.reduce_mean(filter_activation)

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

- Again, the gain is computing within a GradientTape scope, to keep track of the gradient of the gain with respect to **img**.
- **activation** is the output of the Conv2D layer. It is a 3D array.
- **filter_activation** is the 2D slice of **activation** corresponding to the output of the filter specified by **filter_index**.
  - We discard the top and bottom two rows, and the left and right two columns.

The **gradient_ascent_step** Function

```
def gradient_ascent_step(img, filter_index,
                                    learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        activation = truncated_model(img)
        filter_activation = activation[:, 2:-2, 2:-2,
                                       filter_index]
        gain = tf.reduce_mean(filter_activation)

    # Compute gradients.
    grads = tape.gradient(gain, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return gain, img
```

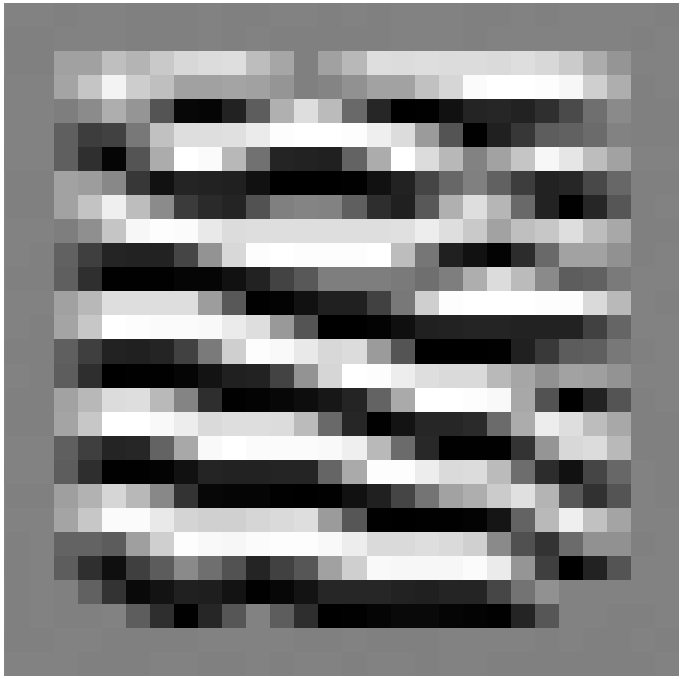
- **gain** is the average of all the values in **filter_activation**.
- The rest of the code is the same as in the version used for generating adversarial inputs.

Running the Code

```
filter_index = 0
gain, img = visualize_filter(filter_index)
filename = "0_%d.png" & (filter_index)
keras.preprocessing.image.save_img(filename, img)
display(Image(filename))
```

- It looks like filter 0 has been trained to identify horizontal patterns.

Visualization of filter 0

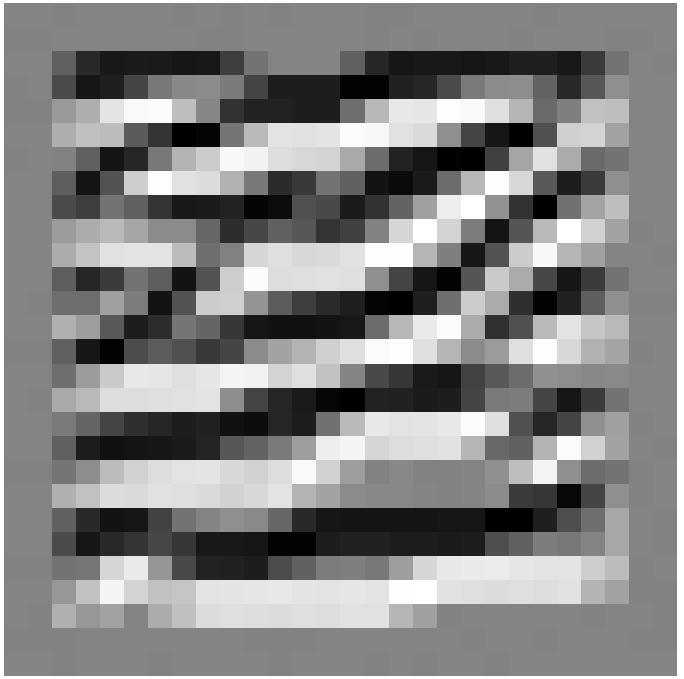


Running the Code

```
filter_index = 1
gain, img = visualize_filter(filter_index)
filename = "0_%d.png" & (filter_index)
keras.preprocessing.image.save_img(filename, img)
display(Image(filename))
```

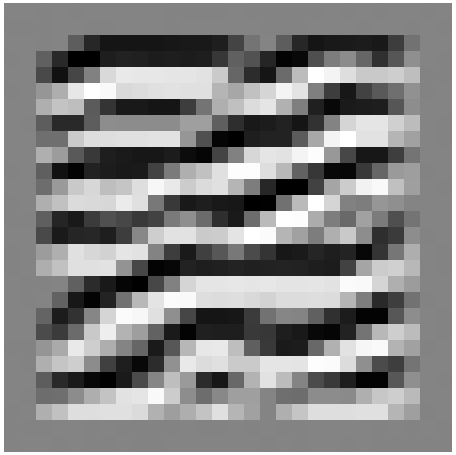
- Filter 1 has also been trained to identify horizontal patterns, but:
  - These patterns slant upwards as we move to the right.
  - For filter 0, the patterns slant downwards as we move to the right.

Visualization of filter 1

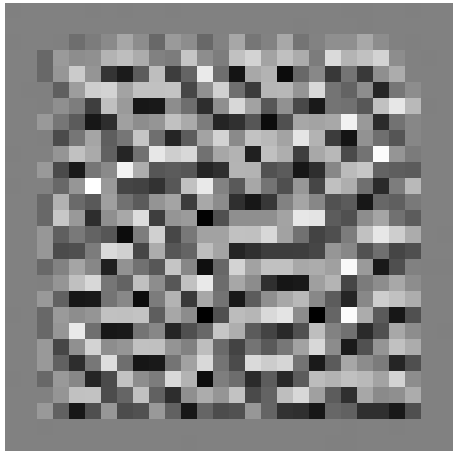


More Visualizations

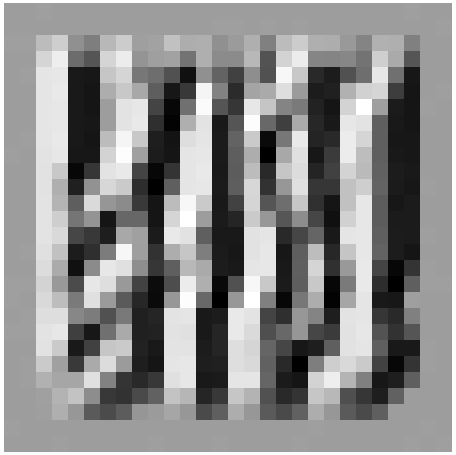
filter 2



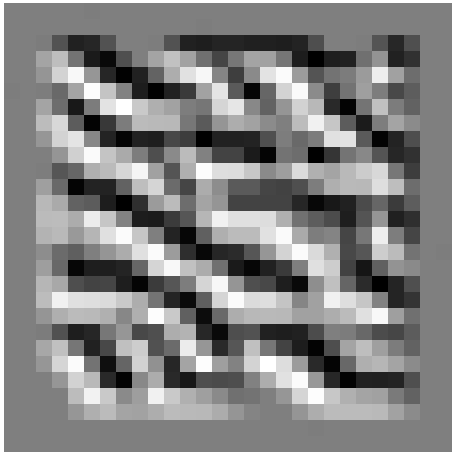
filter 3



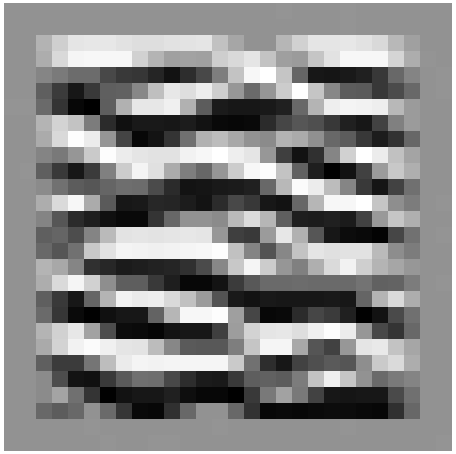
filter 4



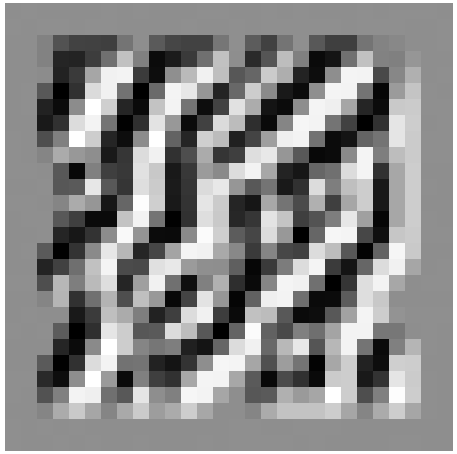
filter 5



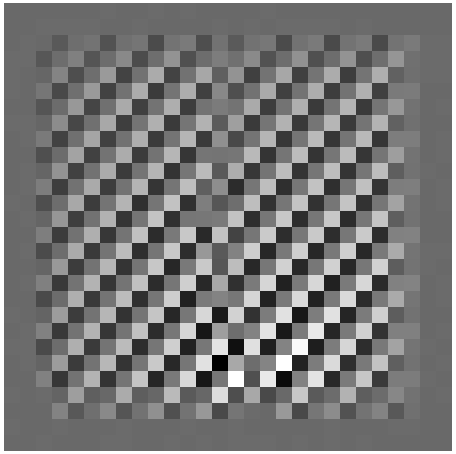
filter 6



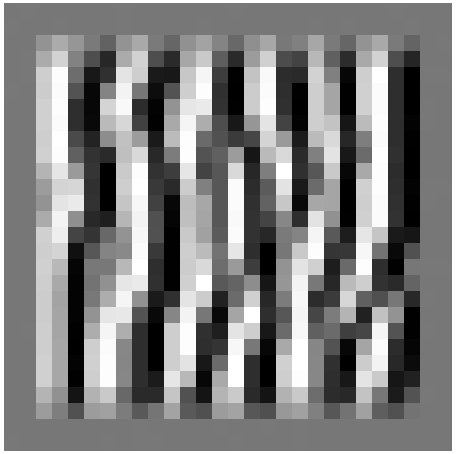
filter 7



filter 8

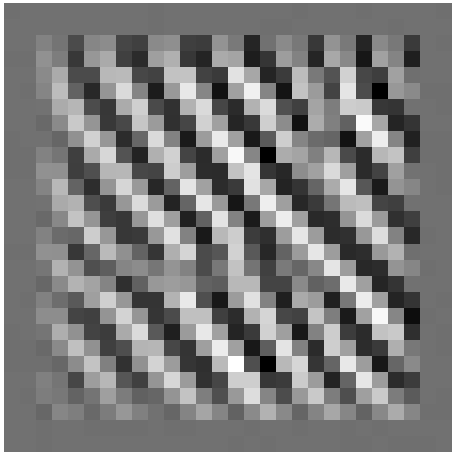


filter 9

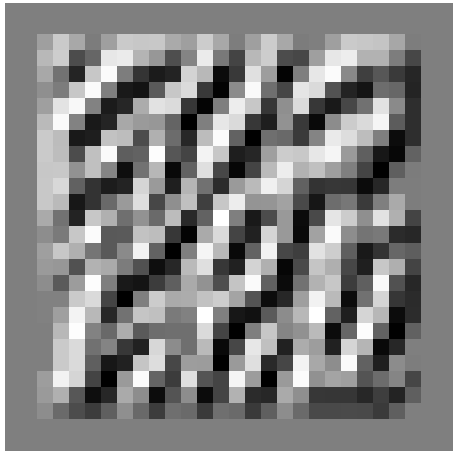


More Visualizations

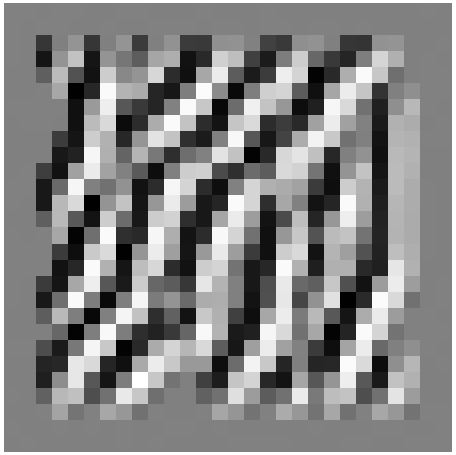
filter 10



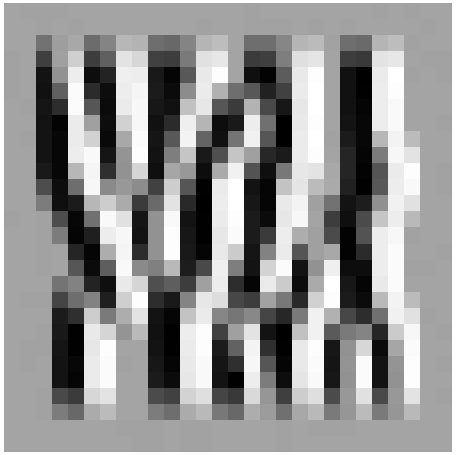
filter 11



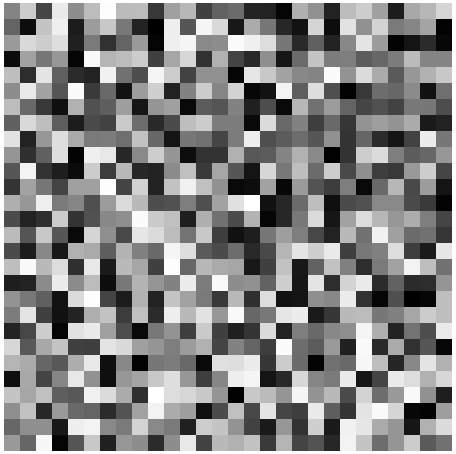
filter 12



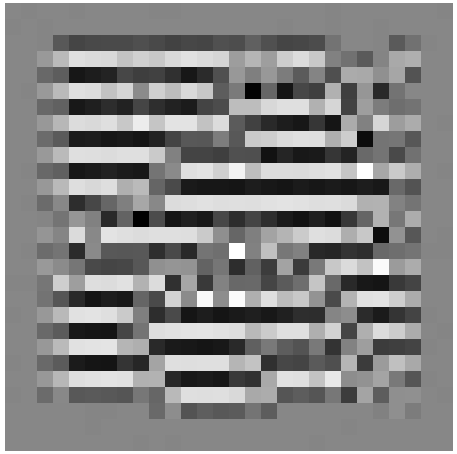
filter 13



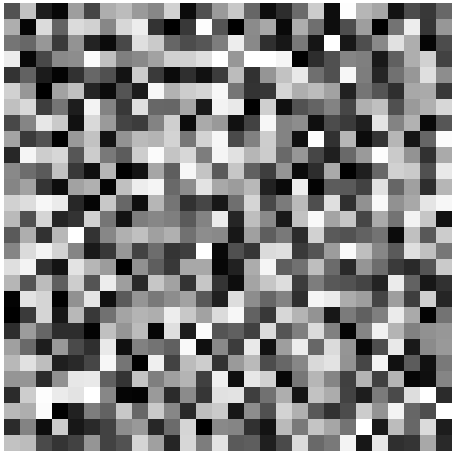
filter 14



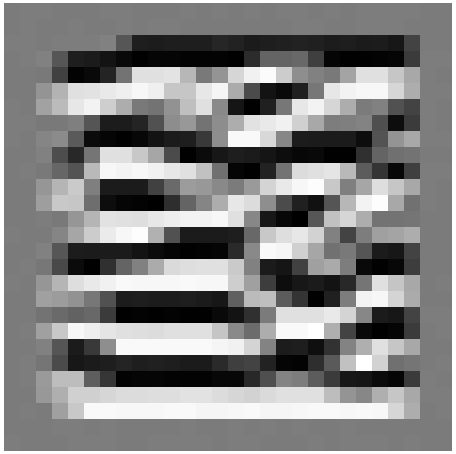
filter 15



filter 16



filter 17



Recap

- We saw how to use gradient ascent to construct inputs that maximize outputs of specific units and layers of a model.
- We applied this in two different ways:
  - To generate adversarial inputs, i.e., to tweak existing images so that the tweaked versions get misclassified by the model.
  - To visualize the types of patterns that hidden units in the network are trained to provide maximal responses to.
- Both these applications can help us gain some intuition about the behavior of our models.