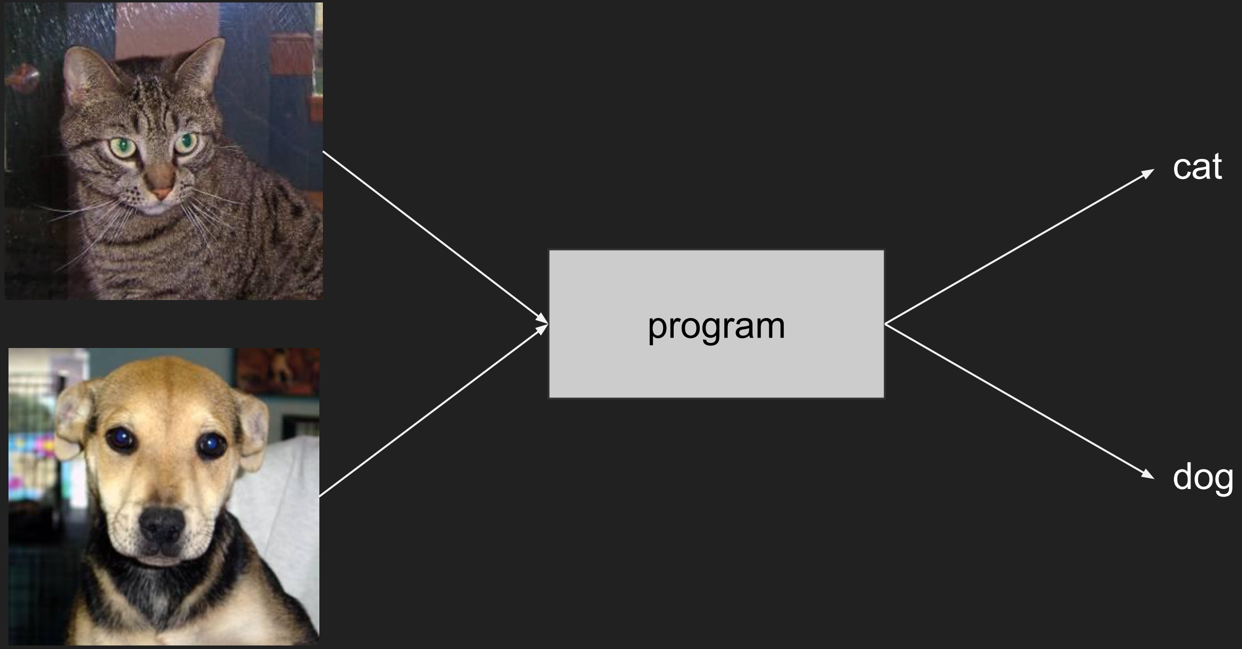
# Ceci n’est pas un chat

Convolutional neural networks from scratch
and
Adversarial examples

# First Goal: create a good cat vs dog classifier



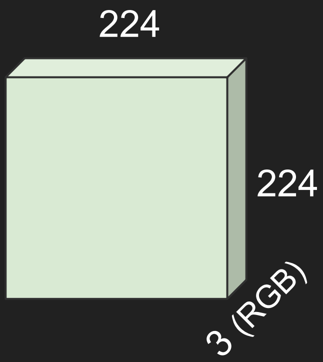
# Second Goal: find adversary images that fool it

| dog | | special noise | | cat |
|---|---|---|---|---|
|  | + ε * |  | = |  |

# It’s all numbers

Input (150 528)

=




Output (2)

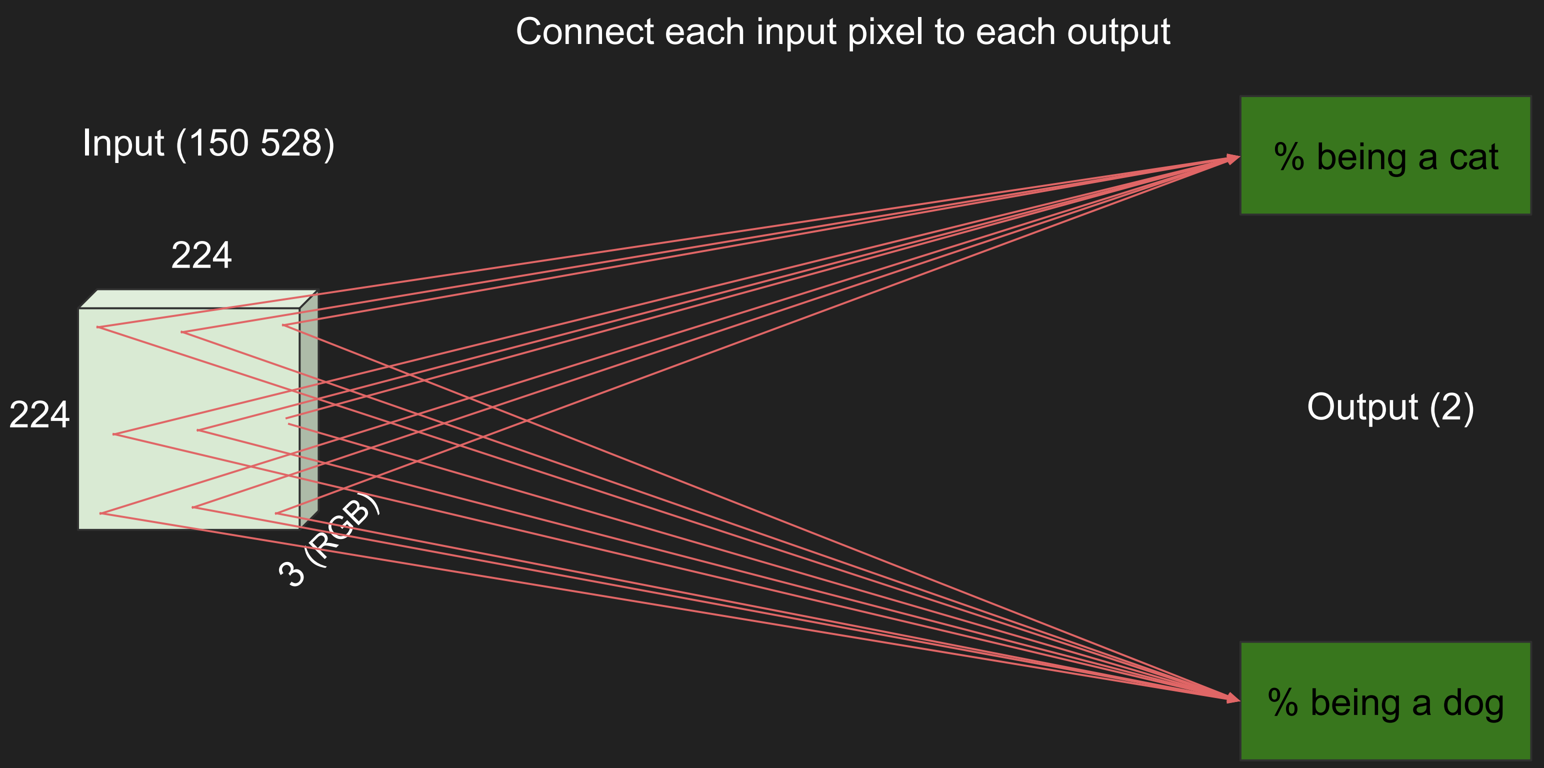
% being a cat

% being a dog

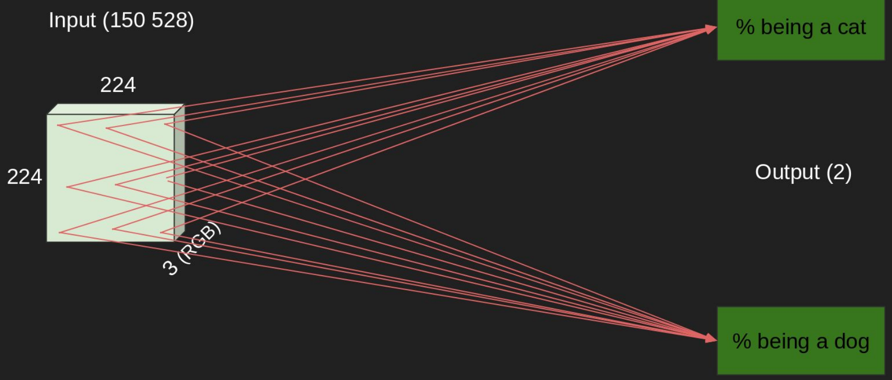
An awesome introduction to neural nets:

[3Blue1Brown: But what is a Neural Network?]

# Neural networks = MANY! simple operations



# Neural networks = MANY! simple operations




$$\text{cat} = i_{1,1,1} * w_{1,1,1,1} + i_{1,1,2} * w_{1,1,2,1} + i_{1,1,3} * w_{1,1,3,1} + i_{1,2,1} * w_{1,2,1,1} + i_{1,2,2} * w_{1,2,2,1} + i_{1,2,3} * w_{1,2,3,1} + \ldots + i_{224,224,1} * w_{224,224,1,1} + i_{224,224,2} * w_{224,224,2,1} + i_{224,224,3} * w_{224,224,3,1} + b_1$$

$$\text{dog} = i_{1,1,1} * w_{1,1,1,2} + i_{1,1,2} * w_{1,1,2,2} + i_{1,1,3} * w_{1,1,3,2} + i_{1,2,1} * w_{1,2,1,2} + i_{1,2,2} * w_{1,2,2,2} + i_{1,2,3} * w_{1,2,3,2} + \ldots + i_{224,224,1} * w_{224,224,1,2} + i_{224,224,2} * w_{224,224,2,2} + i_{224,224,3} * w_{224,224,3,2} + b_2$$

Or simply: OUT = IN, W + B

2 values

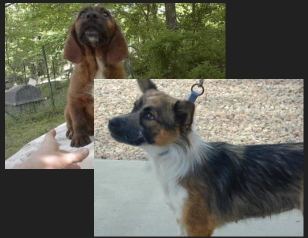
301 056 values

# Train a NN = find those values (W and B)

Random initial guess → Show some examples → Tweak values

$w_{1,1,1,1}$ = 0.12
$w_{1,1,1,2}$ = -0.03
...

cat



dog

$w_{1,1,1,1}$ += 0.04
$w_{1,1,1,2}$ -= 0.007
...

Keywords: stochastic gradient descent (SGD), back-propagation
Nice takes on the subjects:

- [3Blue1Brown: Gradient descent, how neural networks learn]
- [colah's blog: Calculus on Computational Graphs: Backpropagation]

# Softmax: keep the output under control

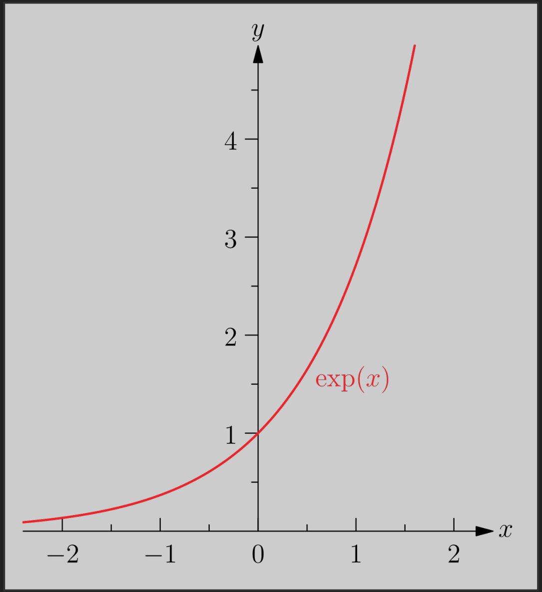
We must guarantee:

- cat + dog = 1
- 0 ≤ cat,dog ≤ 1

a → % being a cat

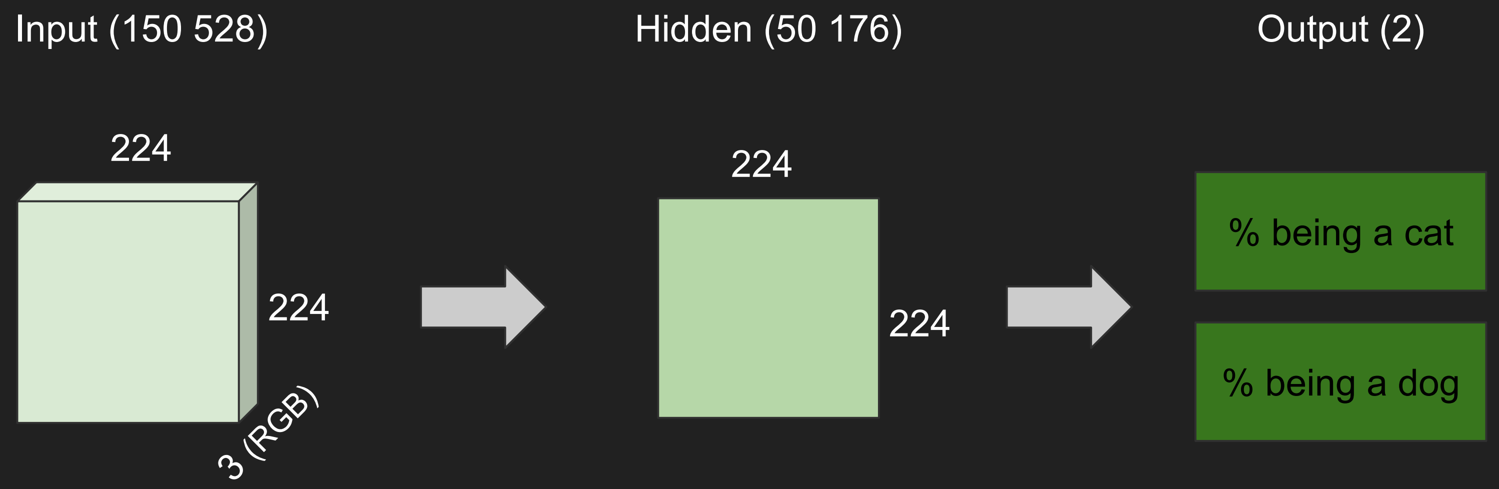
b → % being a dog

cat = $e^a / (e^a + e^b)$
dog = $e^b / (e^a + e^b)$




Or simply: OUT = softmax(IN, W + B)

# Let's go deeper




HIDDEN = IN, $W_1$ + $B_1$

OUT = softmax(HIDDEN, $W_2$ + $B_2$)

# Problem 1: linear collapse

$$\text{HIDDEN} = \text{IN}, W_1 + B_1$$

$$\text{OUT} = \text{softmax}(\text{HIDDEN}, W_2 + B_2)$$

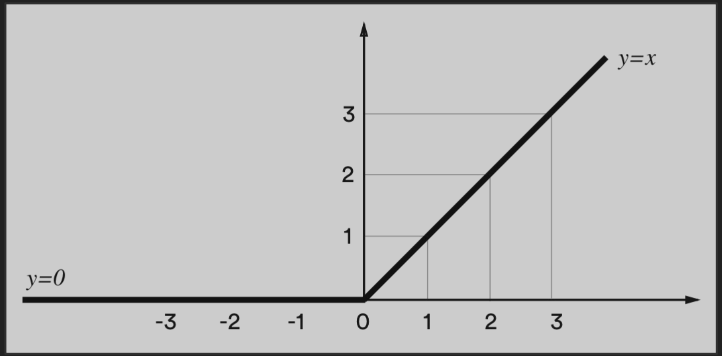
$$\text{OUT} = \text{softmax}(\text{IN}, (W_1, W_2) + (B_1, W_2 + B_2))$$

Exactly the same equation form without the hidden layer

# Solution: toss some non-linearity

Activation function

HIDDEN = relu(IN, $W_1$ + $B_1$)
OUT = softmax(HIDDEN, $W_2$ + $B_2$)




relu(x) = max(0,,)

A mind-blowing dive into the effects of non-linearity:
[colah's blog: Neural Networks, Manifolds, and Topology]

# Problem 2: parameter explosion

Input (150 528) → Output (2)

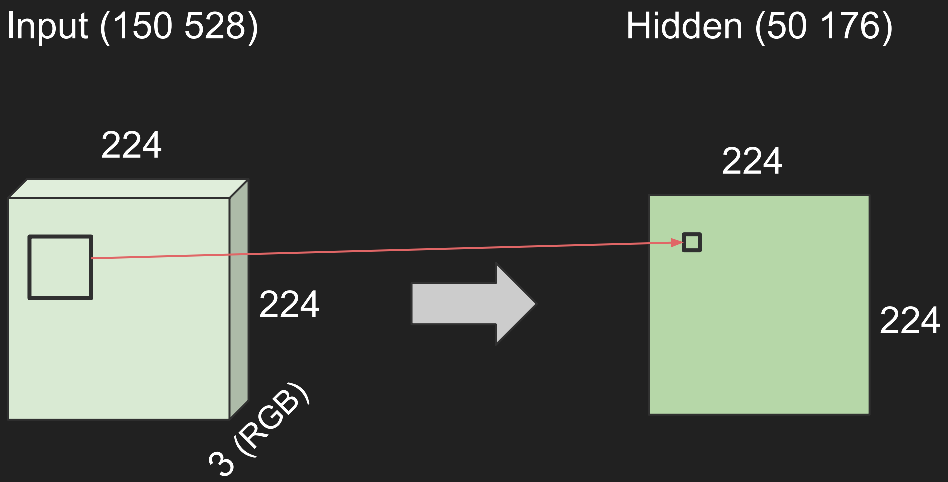
$\| W \| = 150\ 528 * 2 = 301\ 056$
$\| B \| = 2$

Input (150 528) → Hidden (50 176) → Output (2)

$\| W \| = 150\ 528 * 50\ 176 + 50\ 176 * 2 = \underline{7\ 552\ 993\ 280}$
$\| B \| = 50\ 176 + 2 = 50\ 178$

Too complex to solve

# Solution - part 1: local connection

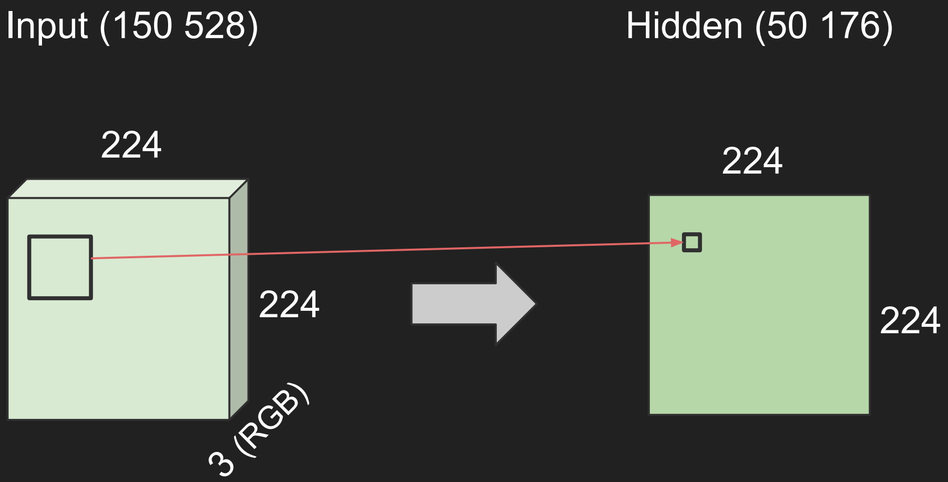



Connect each unit to only a neighboring input region, instead of all of it

Typical size: 3x3

‖ W ‖ = 50 176 * 9 = 451 584
‖ B ‖ = 50 176

# Solution - part 2: shared kernel




Use the same set of values for each unit (invariant to translation)

Typical size: 3x3

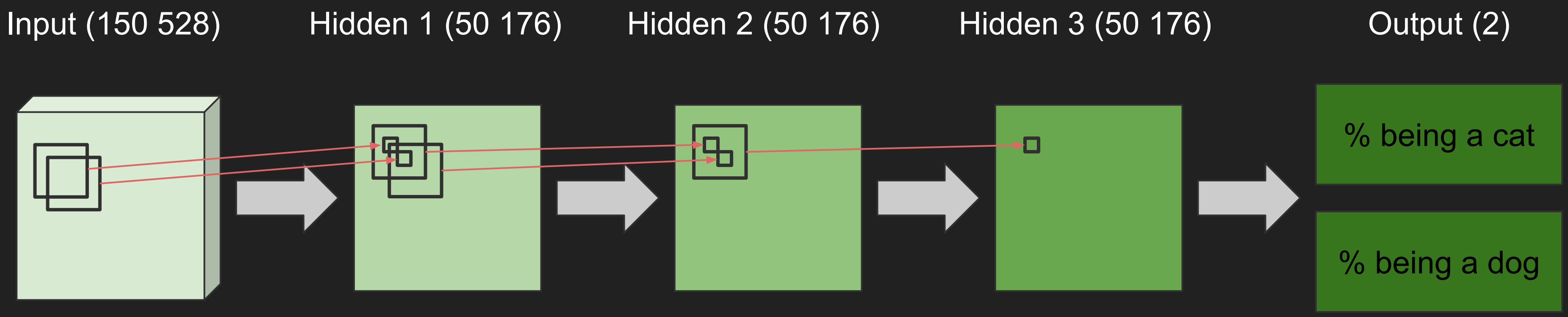
‖ W ‖ = 9
‖ B ‖ = 1

**This is called convolution**

Nice extras:

[Computerphile: How Blurs & Filters Work]
[Computerphile: Separable Filters and a Bauble]

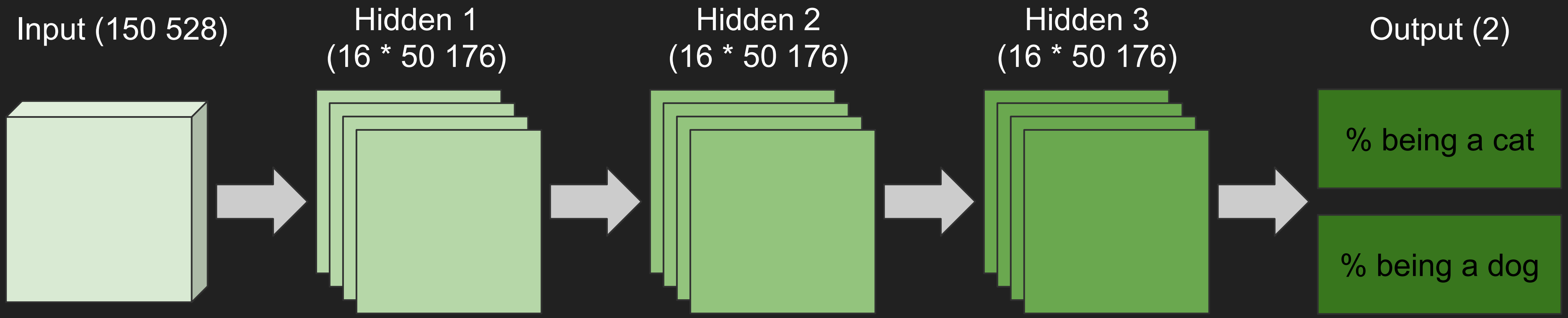
# Let’s go deeper




Each new layer has a more abstract representation of the input:

pixels edge corner eye

# Let’s go wider

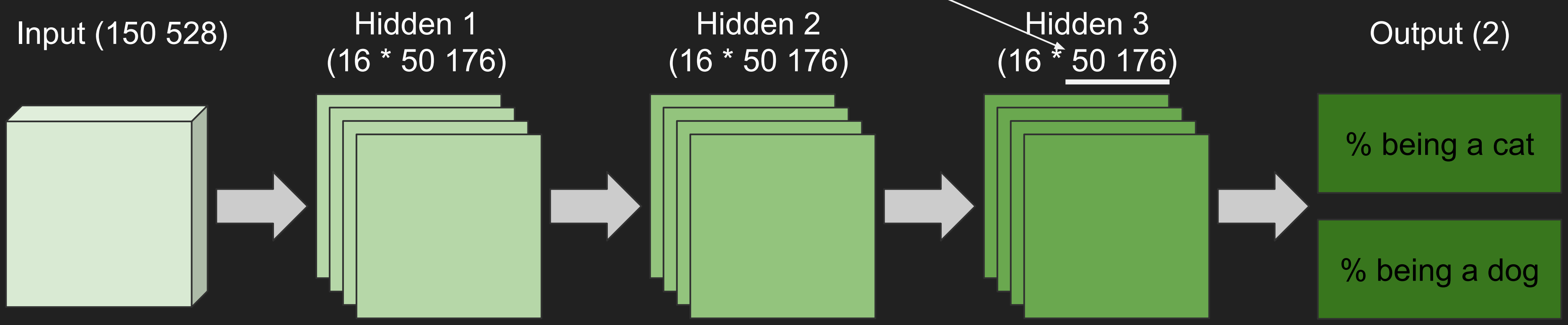



Each new layer has **multiple** abstract representations of the input:

| pixels | vertical edge<br>horizontal edge<br>brightness<br>blueness | square<br>circle<br>triangle | eye<br>mouth<br>ear |
|---|---|---|---|

# Two problems

1: Too much detail at higher layers:
no need to know that many times that an eye was detected



# Two problems

2: Too slow composition: it doesn't see the "big picture"

Input pixels per unit: 3x3 5x5 7x7

Input (150 528)

Hidden 1 (16 * 50 176)

Hidden 2 (16 * 50 176)

Hidden 3 (16 * 50 176)

Output (2)

# Solution: group pixels (pooling)

Typical: that the max of each group of 2x2 units

| Layer: | conv | **pool** | conv | **pool** | conv | **pool** | out |
|---|---|---|---|---|---|---|---|
| Size: | 224x224x16 | **112x112x16** | 112x112x16 | **56x56x16** | 56x56x16 | **28x28x16** | 2 |
| Input pixels per unit | 3x3 | **4x4** | 8x8 | **10x10** | 18x18 | **22x22** | 224x224 |

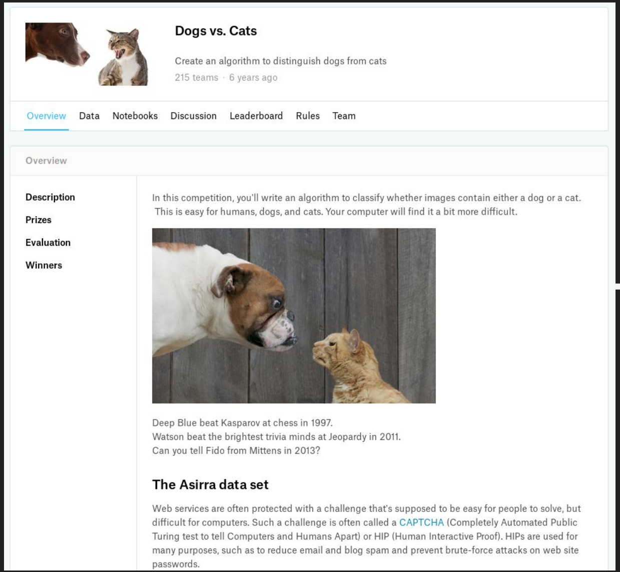
# Let’s code it

```python
model = Sequential([
    Lambda(lambda x: x / 255 - 0.5, input_shape=(224, 224, 3)),
    Conv2D(16, (3, 3), padding='same', activation='relu'),
    MaxPool2D(),
    Conv2D(16, (3, 3), padding='same', activation='relu'),
    MaxPool2D(),
    Conv2D(16, (3, 3), padding='same', activation='relu'),
    MaxPool2D(),
    Flatten(),
    Dense(2, activation='softmax')
])
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 224, 224, 3)       0
_________________________________________________________________
conv2d (Conv2D)              (None, 224, 224, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 112, 112, 16)      2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 56, 56, 16)        2320
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 16)        0
_________________________________________________________________
flatten (Flatten)            (None, 12544)             0
_________________________________________________________________
dense (Dense)                (None, 2)                 25090
=================================================================
Total params: 30,178
Trainable params: 30,178
Non-trainable params: 0
_________________________________________________________________
```

Documentation: tf.keras.Sequential
Repository: ceci-nest-pas-un-chat

# Let’s prepare our data


Dogs vs. Cats

Create an algorithm to distinguish dogs from cats

215 teams · 6 years ago

Overview Data Notebooks Discussion Leaderboard Rules Team

Overview

Description
Prizes
Evaluation
Winners

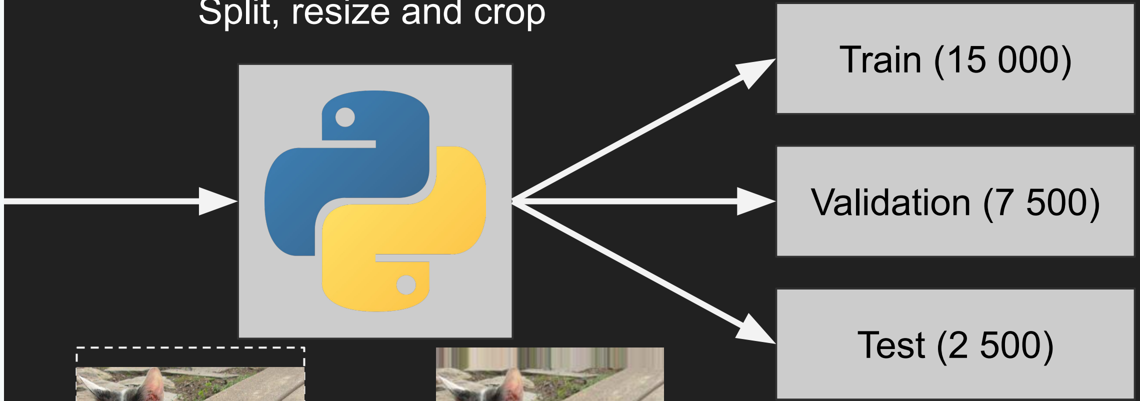
In this competition, you'll write an algorithm to classify whether images contain either a dog or a cat. This is easy for humans, dogs, and cats. Your computer will find it a bit more difficult.

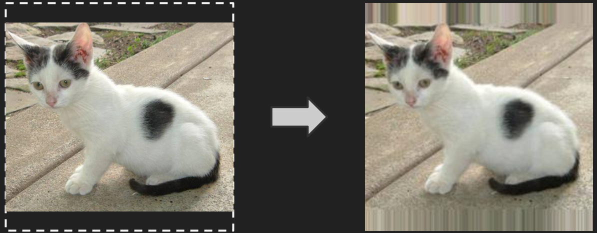
Deep Blue beat Kasparov at chess in 1997.
Watson beat the brightest trivia minds at Jeopardy in 2011.
Can you tell Fido from Mittens in 2013?

The Asirra data set

Web services are often protected with a challenge that's supposed to be easy for people to solve, but difficult for computers. Such a challenge is often called a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) or HIP (Human Interactive Proof). HIPs are used for many purposes, such as to reduce email and blog spam and prevent brute-force attacks on web site passwords.

Split, resize and crop



Kaggle: [Dogs vs. Cats]
Python: [prepare_data.py]

# Let’s train it

```
base_gen = ImageDataGenerator()

train_generator = base_gen.flow_from_directory('data/training', (224, 224))
validation_generator = base_gen.flow_from_directory('data/validation', (224, 224))
test_generator = base_gen.flow_from_directory('data/test', (224, 224))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model.fit_generator(train_generator, epochs=20, validation_data=validation_generator)
```

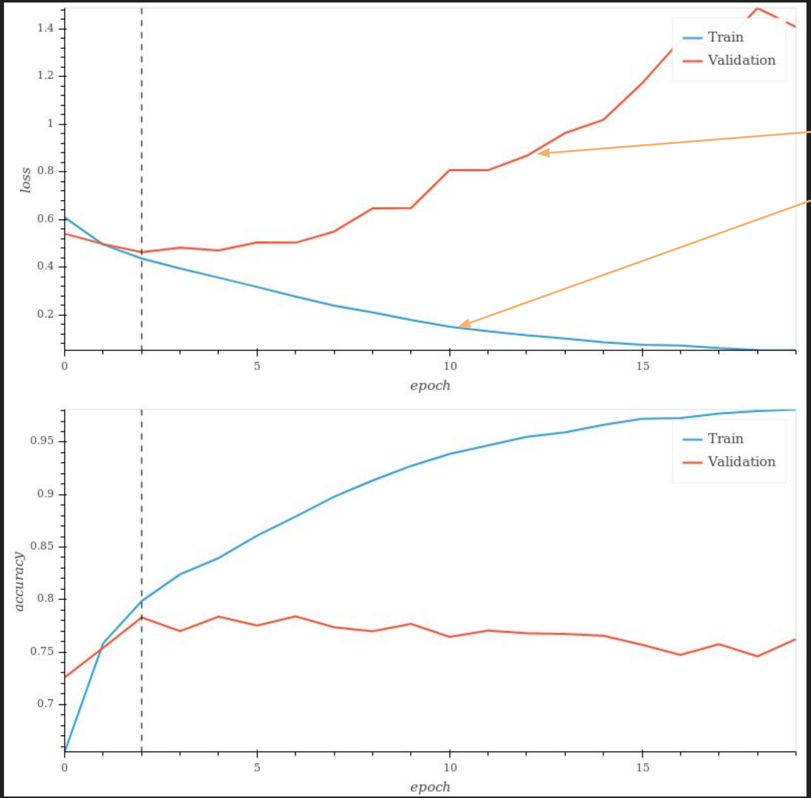
= -log(right_class)

= right_answers / total

pass through the
training set 20x

Amazing explanation of cross-entropy: [colah's blog: Visual Information Theory]()

# Training stats




Overfit: model “memorizes” answers for test set, and generalization is compromised

Use validation to known when to stop:

```
model.fit_generator(train_generator, epochs=1000,
validation_data=validation_generator, callbacks=[
    EarlyStopping(patience=1, restore_best_weights=True)
])
```

More robust techniques: dropout, batch normalization, residual convolution

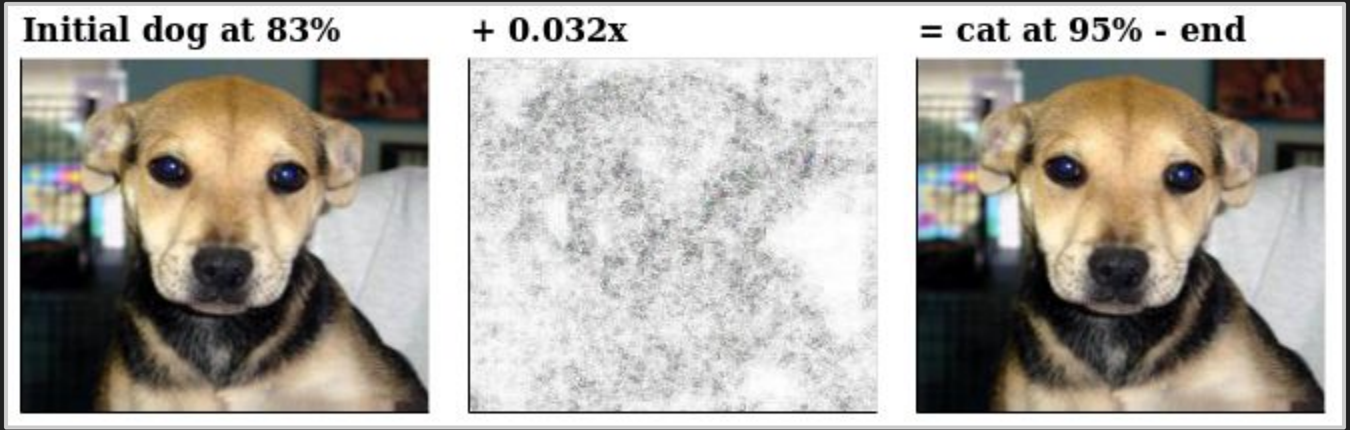
# Results over test set



| | | predicted | |
|---|---|---|---|
| | | cat | dog |
| real | cat | 1013 | 237 |
| | dog | 270 | 980 |

# Cool! Let’s break it

Goal: modify an image to manipulate the result



# In the real world


Mistaken by 45 mi/h speed limit
https://arxiv.org/abs/1707.08945

Active research area

Open-source collection of attacks and defences:
https://github.com/tensorflow/cleverhans

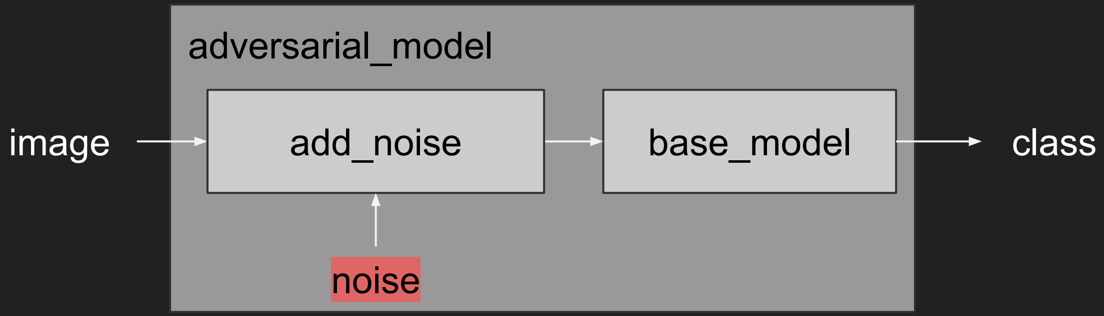
# How?

cat

image + 255 * noise → base_model → target wrong classification

loss = 0.1 * l2(noise) - log(target_wrong_category)

# How?

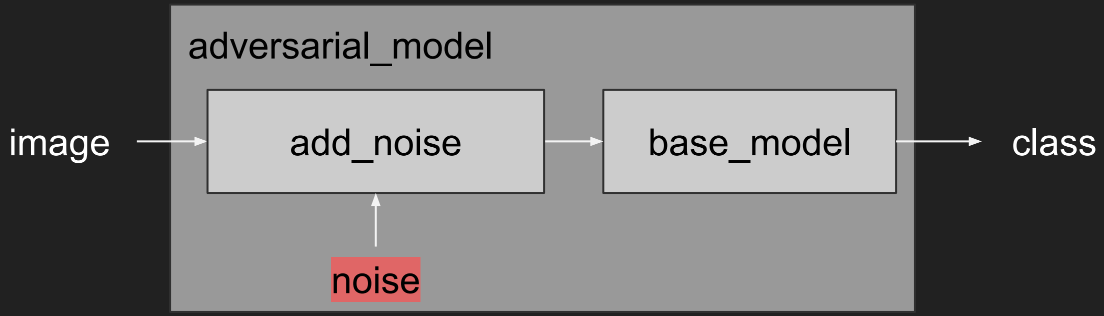



```
base_model = load_model('models/simple')
base_model.trainable = False

base_img_input = Input((224, 224, 3))
adversarial_img_layer = AddNoise(l2(0.1), 255)
adversarial_img = adversarial_img_layer(base_img_input)
model_output = base_model(adversarial_img)
adversarial_model = Model(inputs=base_img_input, outputs=model_output)
```

control the tradeoff between image similarity and fooling power

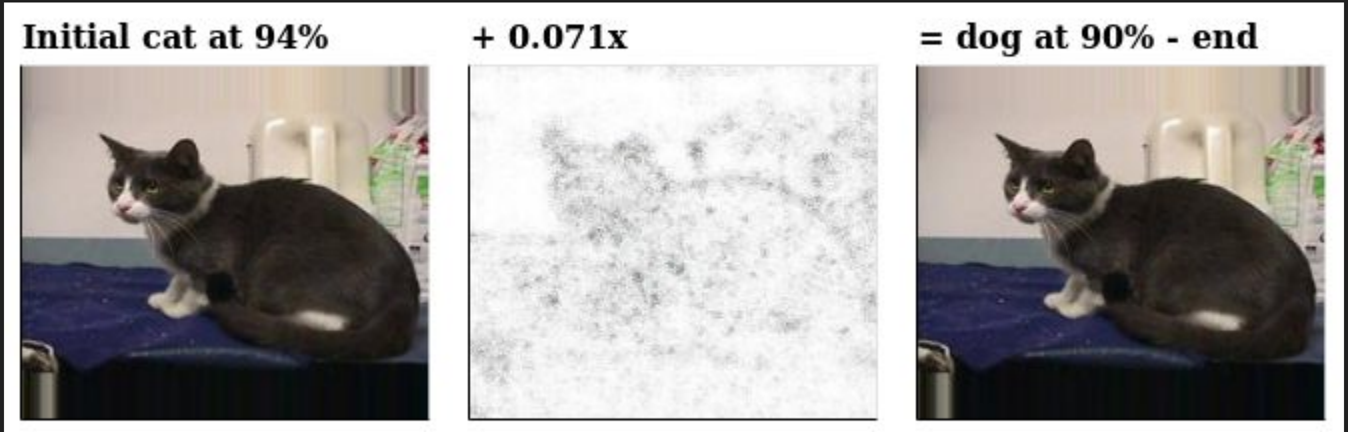
# How?




```
class AddNoise(Layer):
    def __init__(self, regularizer=None, scale=1, **kwargs):
        self.scale = scale
        self.regularizer = regularizer
        super().__init__(**kwargs)
    def build(self, input_shape):
        self.noise = self.add_weight('noise', input_shape[1:], regularizer=self.regularizer)
        super().build(input_shape)
    def call(self, inputs):
        return tf.clip_by_value(inputs + self.scale * self.noise, 0, 255)
```

# How?

```
adversarial_model.compile(optimizer='adam', loss='categorical_crossentropy')
adversarial_model.fit(x=base_img, y=target_wrong_class, epochs=1000, callbacks=[
    EarlyStopping(monitor='loss', patience=10, restore_best_weights=True)
])
```



# Extra: transfer learning with VGG16

```python
model_input = Input((224, 224, 3))
base_model_input = Lambda(vgg16.preprocess_input)(model_input)

base_model = VGG16(include_top=False,
    input_tensor=base_model_input,
    pooling='avg')
base_model.trainable = False

x = base_model.output
x = Dense(128, activation='relu', name='fc1')(x)
x = Dense(16, activation='relu', name='fc2')(x)
x = Dense(2, activation='softmax', name='predictions')(x)

model = Model(model_input, x)
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
lambda (Lambda)              (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
fc1 (Dense)                  (None, 128)               65664
_________________________________________________________________
fc2 (Dense)                  (None, 16)                2064
_________________________________________________________________
predictions (Dense)          (None, 2)                 34
=================================================================
Total params: 14,782,450
Trainable params: 67,762
Non-trainable params: 14,714,688
_________________________________________________________________
```

# Quite of an improvement!

previous model
accuracy = 79.7%

| | predicted | |
|---|---|---|
| real | | cat | dog |
| | cat | 1013 | 237 |
| | dog | 270 | 980 |

VGG16
accuracy = 97.8%

| | predicted | |
|---|---|---|
| real | | cat | dog |
| | cat | 1223 | 27 |
| | dog | 28 | 1222 |

# Let’s break it