

# **Advanced Neural Networks**



# Overview

- Convolutional neural networks
- Data augmentation
- Using pre-trained networks
- Batch normalization



# Convolutional neural nets

- When processing image data, we want to discover 'local' patterns (between nearby pixels)
  - edges, lines, structures
- Consider *windows* (or *patches*) of pixels (e.g 5x5)

□  
ml



# Convolution

- Slide an  $n \times n$  *filter* (or *kernel*) over  $n \times n$  *patches* of the input feature map
- Replace pixel values with the convolution of the kernel with the underlying image patch

ml

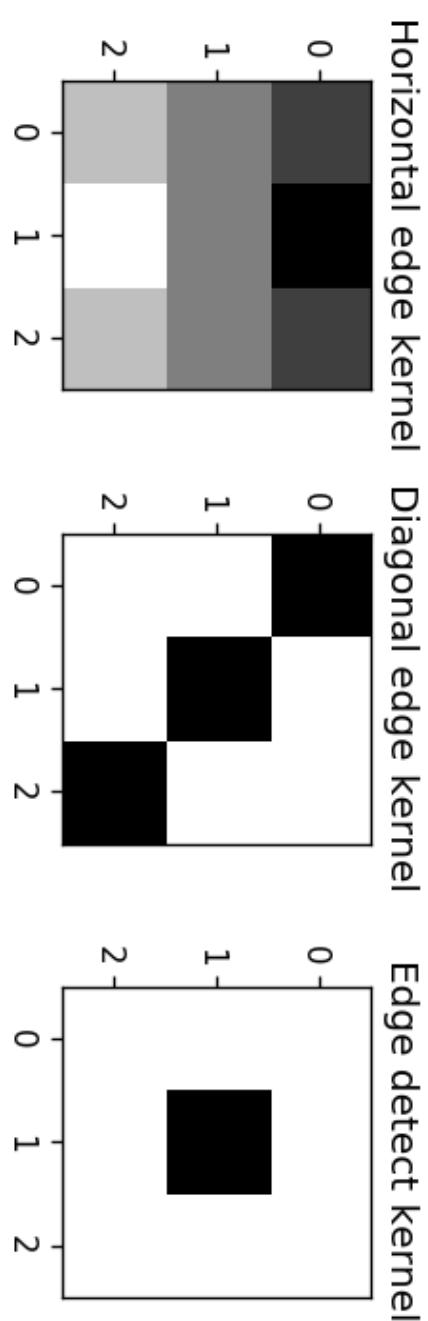


- The convolution operation itself takes the sum of the values of the element-wise product of the image patch with the kernel

```
def apply_kernel(center, kernel, original_image):  
    image_patch = original_image>window_slice(center, kernel)]  
    # An element-wise multiplication followed by the sum  
    return np.sum(kernel * image_patch)
```



- Different kernels can detect different types of patterns in the image





## Demonstration: horizontal edge filter

- Responds only to horizontal edges, sensitive to the 'direction' of the edge



MNIST Demonstration: horizontal edge filter



## MNIST Demonstration: diagonal edge filter



MNIST Demonstration: edge detect filter



## Image convolution in practice

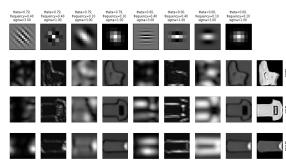
- Even before deep learning, convolutions were used a lot to preprocess image data
- *Families* of kernels were run on every image (e.g. Gabor filters)



## Demonstration: Fashion MNIST



Fashion MNIST with multiple filters (filter bank)





## Convolutional layers: Feature maps

- Input images are 3D tensor (height,width,channels)
- We slide  $d$  filters across the input image in parallel, producing a  $(1 \times 1 \times d)$  output per patch. These are reassembled into the final *feature map* with  $d$  channels.
- The filters are randomly initialized, we want to *learn* the optimal values





## Border effects

- Consider a  $5 \times 5$  image and a  $3 \times 3$  filter: there are only 9 possible locations, hence the output is a  $3 \times 3$  feature map
- If we want to maintain the image size, we use *zero-padding*, adding 0's all around the input tensor.

□ ml



## Undersampling

- Sometimes, we want to *downsample* a high-resolution image
  - Faster processing, less noisy (hence less overfitting)
- One approach is to *skip* values during the convolution
  - Distance between 2 windows: *stride length*
- Example with stride length 2 (without padding):

□  
ml



# Max-pooling

- Another approach to shrink the input tensors is *max-pooling*:
  - Run a filter with a fixed stride length over the image
    - Usually 2x2 filters and stride length 2
  - The filter returns the *max* (or *avg*) of all values
- Aggressively reduces the number of weights (less overfitting)
- Information from every input node spreads more quickly to every output node
  - In pure convnets, one input value spreads to 3x3 nodes of the first layer, 5x5 nodes of the second, etc.
  - You'd need much deeper networks, which are much harder to train



# Convolutional nets in practice

- Let's model MNIST again, this time using convnets
  - Conv2D for 2D convolutional layers
    - Default: 32 filters, randomly initialized (from uniform distribution)
  - MaxPooling2D for max-pooling
    - 2x2 pooling reduces the number of inputs by a factor 4

```
model = models.Sequential()  
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,  
28, 1))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(64, (3, 3), activation='relu'))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```



Observe how the input image is reduced to a  $3 \times 3 \times 64$  feature map

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_3 (Conv2D)	(None, 3, 3, 64)	36928

=====  
Total params: 55,744  
Trainable params: 55,744  
Non-trainable params: 0

=====

Compare to the architecture without max-pooling:

- Output layer is a 22x22x64 feature map!

Layer	(type)	Output Shape	Param #
conv2d_4	(Conv2D)	(None, 26, 26, 32)	320
conv2d_5	(Conv2D)	(None, 24, 24, 64)	18496
conv2d_6	(Conv2D)	(None, 22, 22, 64)	36928
====			
Total params:	55,744		
Trainable params:	55,744		
Non-trainable params:	0		
====			



- To classify the images, we still need a Dense and Softmax layer.
- We need to flatten the  $3 \times 3 \times 36$  feature map to a vector of size 576

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```



Final architecture

Layer	(type)	Output Shape	Param #
conv2d_1	(Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1	(MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2	(Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_2	(MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_3	(Conv2D)	(None, 3, 3, 64)	36928
flatten_1	(Flatten)	(None, 576)	0
dense_1	(Dense)	(None, 64)	36928

- Train and test as usual (takes about 5 minutes):
- Compare to the 97.8% accuracy of the earlier dense architecture

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64, verbose=0)
)
test_loss, test_acc = model.evaluate(test_images, test_labels)

10000/10000 [=====] - 3s 290us/
step
Accuracy: 0.9899
```



# Convnets on small datasets

- Let's move to a more realistic dataset: Cats vs Dogs
  - We take a balanced subsample of 4000 real colored images
  - 2000 for training, 1000 validation, 1000 testing
- Convnets learn local patterns, which is highly efficient
- *Translation invariant*: a pattern can be recognized even if it is shifted to another part of the image
  - More robust, efficient to train (with fewer examples)
- We can use tricks such as *data augmentation*
- We can re-use *pre-trained* networks



# Data preprocessing

- We use Keras' ImageDataGenerator to:
  - Decode JPEG images to floating-point tensors
  - Rescale pixel values to [0,1]
  - Resize images to 150x150 pixels
- Returns a Python *generator* we can endlessly query for images
  - Batches of 20 images per query
- Separately for training, validation, and test set



```
train_generator = train_datagen.flow_from_directory(  
    train_dir, # Directory with images  
    target_size=(150, 150), # Resize images  
    batch_size=20, # Return 20 images at a time  
    class_mode='binary') # Binary labels
```



# Build a model from scratch

- Like MNIST, but one more layer and more filters

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



Layer	(type)	Output Shape	Param #
conv2d_4	(Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_3	(MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_5	(Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_4	(MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_6	(Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_5	(MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_7	(Conv2D)	(None, 15, 15, 128)	147584

max\_pooling2d\_6 (MaxPooling2 (None, 7, 7, 128) 0

flatten\_2 (Flatten) (None, 6272) 0

dense\_3 (Dense) (None, 512) 321177  
6

dense\_4 (Dense) (None, 1) 513

=====

Total params: 3,453,121  
Trainable params: 3,453,121  
Non-trainable params: 0

=====

# Training

- Since the data comes from a generator, we use `fit_generator`
  - 100 steps per epoch (of 20 images each), for 30 epochs
  - Also provide sa generator for the validation data

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])

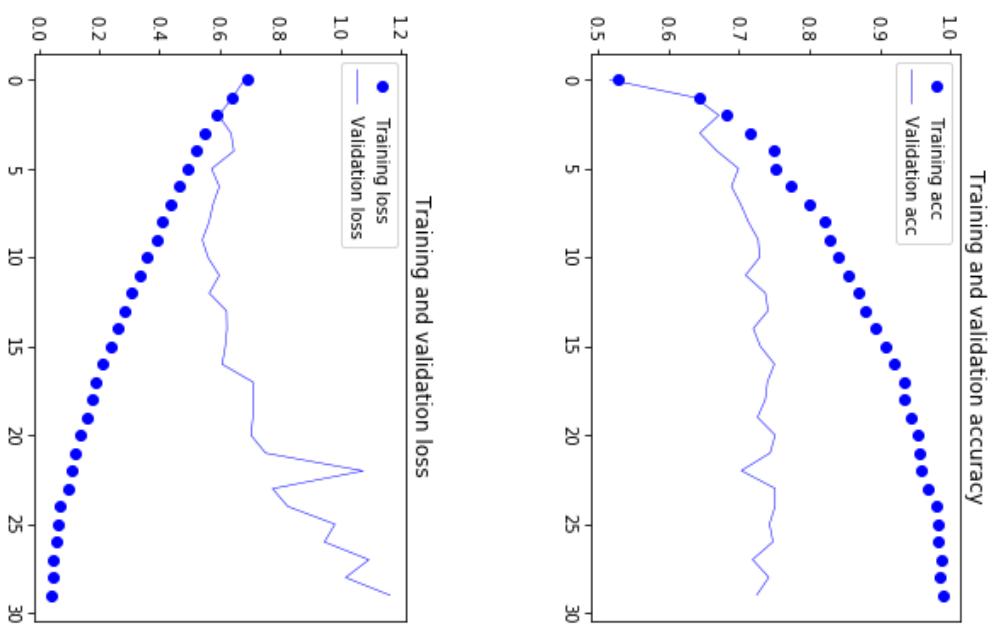
history = model.fit_generator(
    train_generator, steps_per_epoch=100,
    epochs=30, verbose=0,
    validation_data=validation_generator,
    validation_steps=50)
```



- Training takes more than an hour (on CPU)
- We save the trained model to disk so that we can reload it later  
`model.save(os.path.join(model_dir, 'cats_and_dogs_small_1.h5'))`



Our model is overfitting: we need more training examples, more regularization



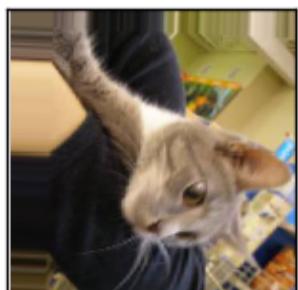
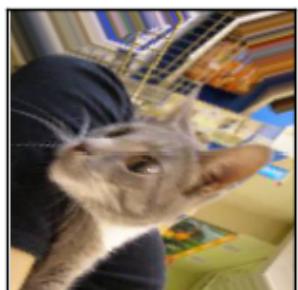
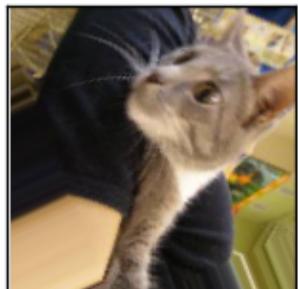
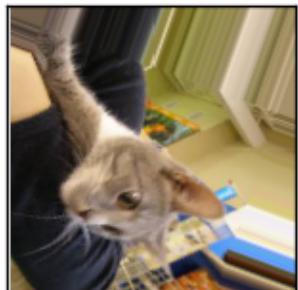
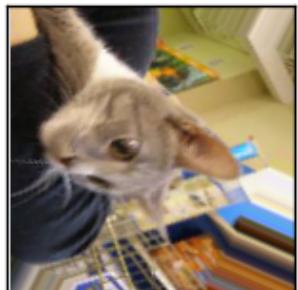
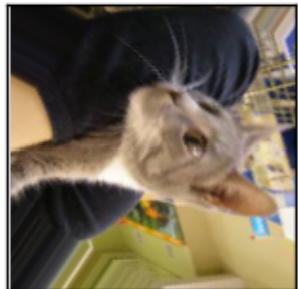


# Data augmentation

- Generate new images via image transformations
  - Rotation, translation, shear, zoom, horizontal flip,...
- Keras has a tool for this:

```
datagen = ImageDataGenerator(  
    rotation_range=40, width_shift_range=0.2,  
    height_shift_range=0.2, shear_range=0.2,  
    zoom_range=0.2, horizontal_flip=True,  
    fill_mode='nearest')
```





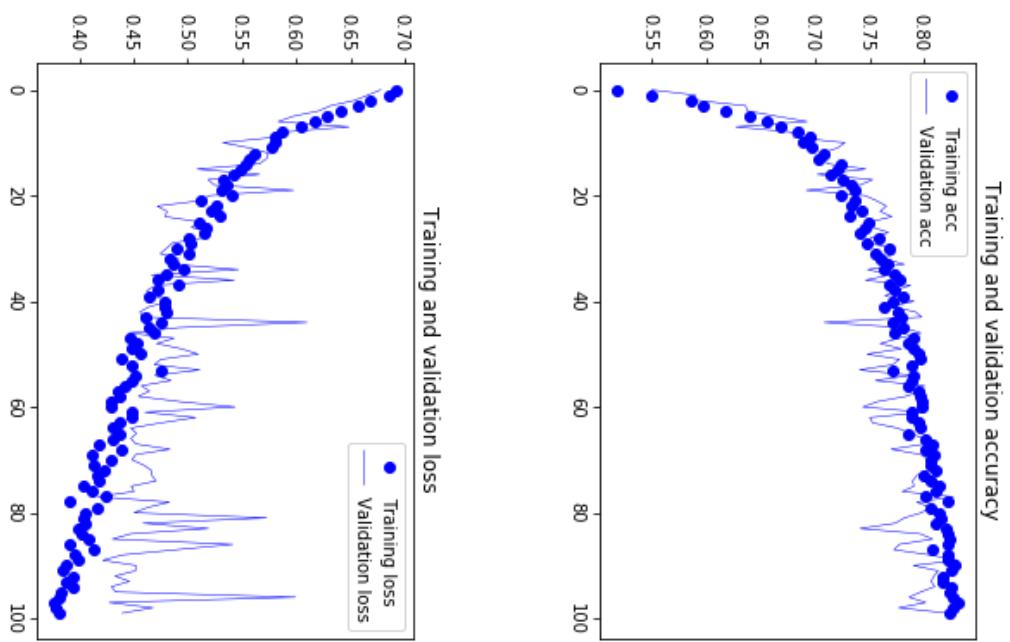


We also add Dropout before the Dense layer

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



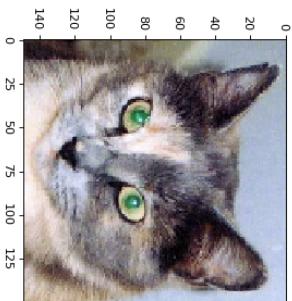
(Almost) no more overfitting!





# Visualizing the intermediate outputs

- Let's see what the convnet is learning exactly by observing the intermediate feature maps
  - A layer's output is also called its *activation*
- Since our feature maps have depth 32/64/128, we need to visualize *all* of them
- We choose a specific input image, and observe the outputs





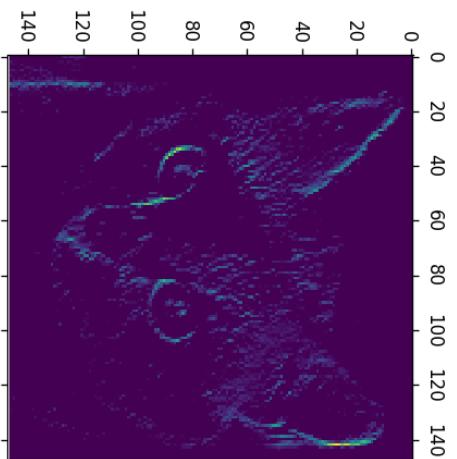
- We create a new model that is composed of the first 8 layers (the convolutional part)
- We input our example image and read the output

```
layer_outputs = [layer.output for layer in model.layers[:8]]  
activation_model = models.Model(inputs=model.input, outputs=layer_output  
s)  
activations = activation_model.predict(img_tensor)
```



Output of the first Conv2D layer, 4th channel (filter):

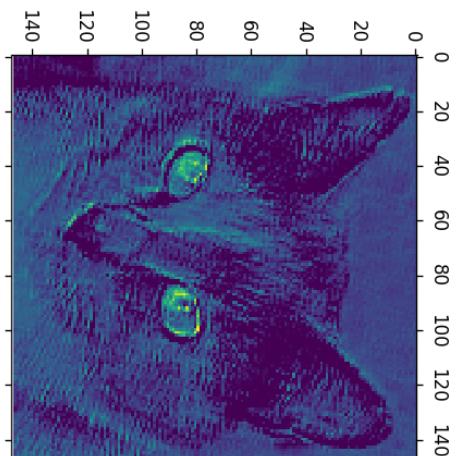
- Similar to a diagonal edge detector
- Your own channels may look different





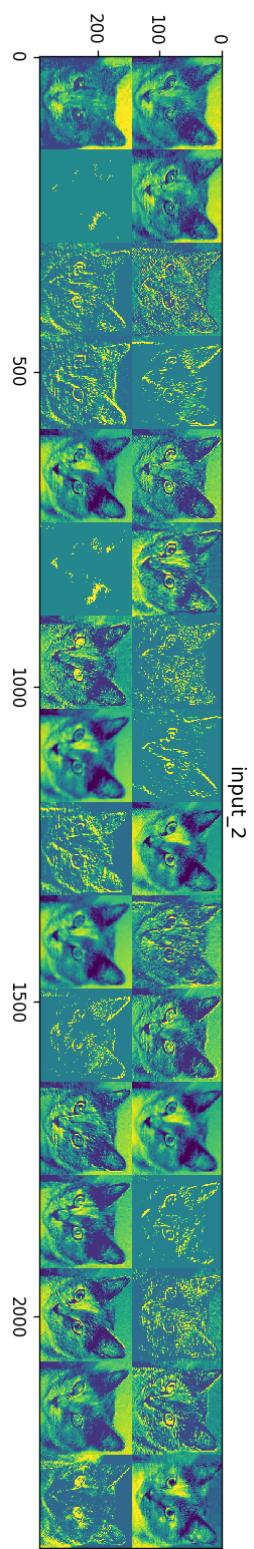
Output of the 22th channel (filter):

- Cat eye detector?

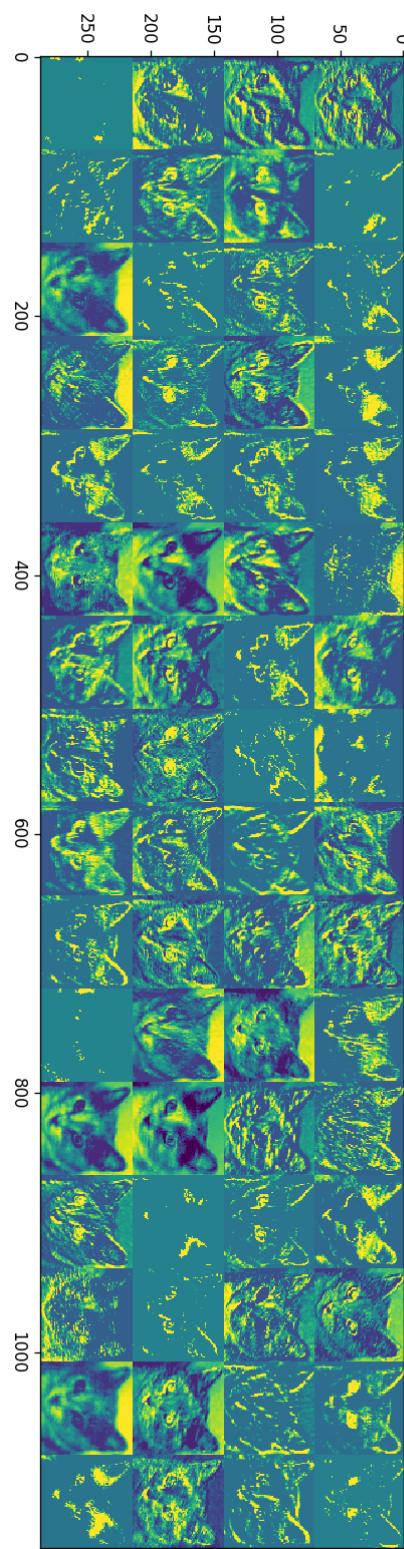




- First 2 convolutional layers: various edge detectors



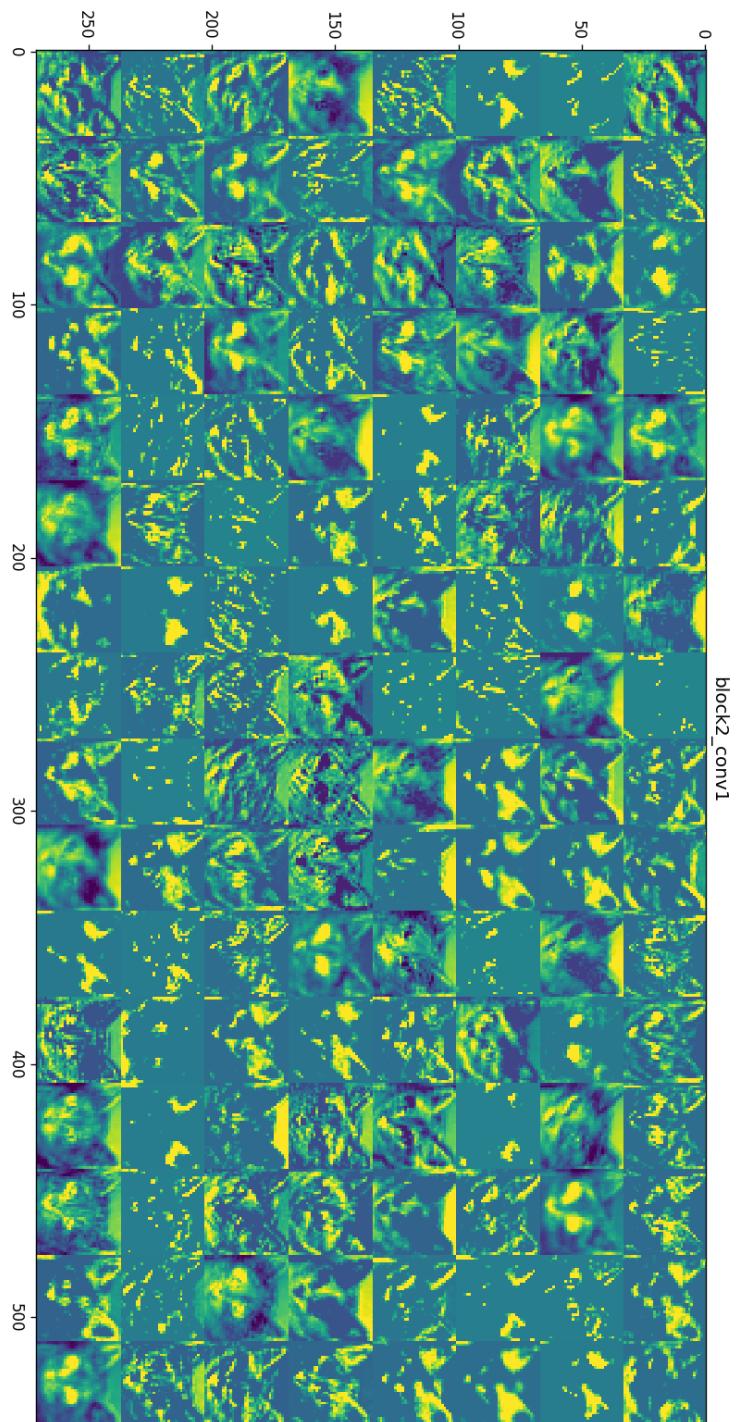
block1\_conv1



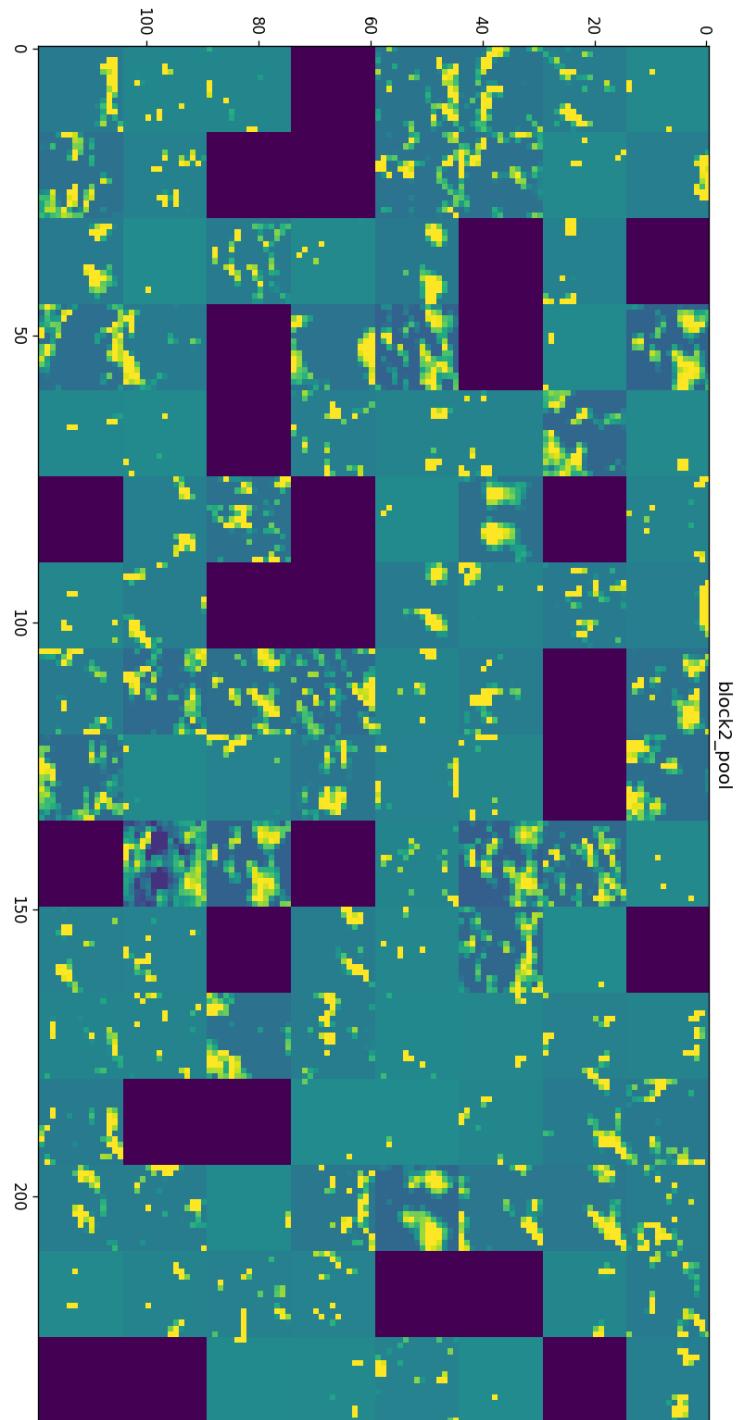
block1\_conv2



- 3rd convolutional layer: increasingly abstract: ears, eyes

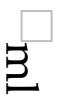


- Last convolutional layer: increasing sparsity. The learned patterns don't exist in the training data



## Spacial hierarchies

- Deep convnets can learn *spacial hierarchies* of patterns
  - First layer can learn very local patterns (e.g. edges)
  - Second layer can learn specific combinations of patterns
  - Every layer can learn increasingly complex *abstractions*





# Visualizing the learned filters

- The filters themselves can be visualized by finding the input image that they are maximally responsive to
- *gradient ascent in input space:*
  - start from a blank image
  - use loss to update the pixel values to values that the filter responds to more strongly

```
from keras import backend as K
input_img = np.random.random((1, size, size, 3)) * 20 + 128.
loss = K.mean(layer_output[:, :, :, filter_index]) # Max response
grads = K.gradients(loss, model.input)[0] # Compute gradient
for i in range(40): # Run gradient ascent for 40 steps
    loss_value, grads_value = K.function([input_img], [loss, grads])
    input_img_data += grads_value * step
```

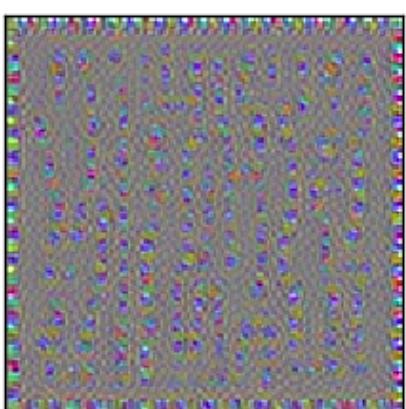
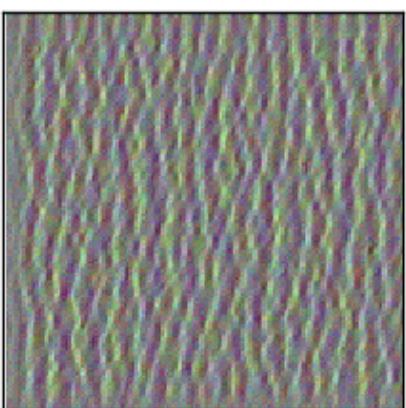
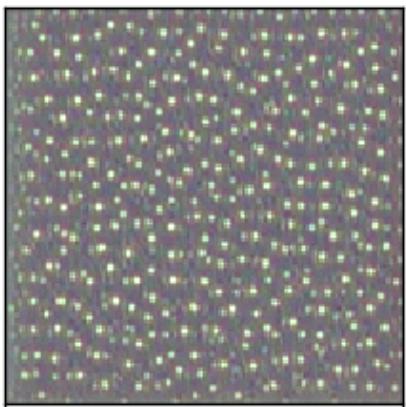


Let's do this for the VGG16 network pretrained on ImageNet

```
model = VGG16(weights='imagenet', include_top=False)
```

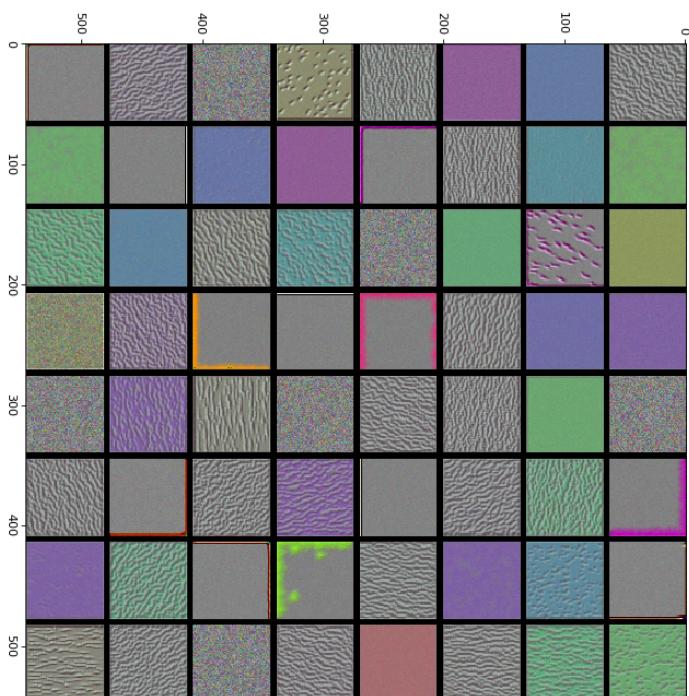
Layer (type)	Output Shape	Param #
input_4 (InputLayer)	(None, None, None, 3)	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1792
block1_conv2 (Conv2D)	(None, None, None, 64)	36928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73856
block2_conv2 (Conv2D)	(None, None, None, 128)	147584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0

- Visualize convolution filters 0-2 from layer 5 of the VGG network trained on ImageNet



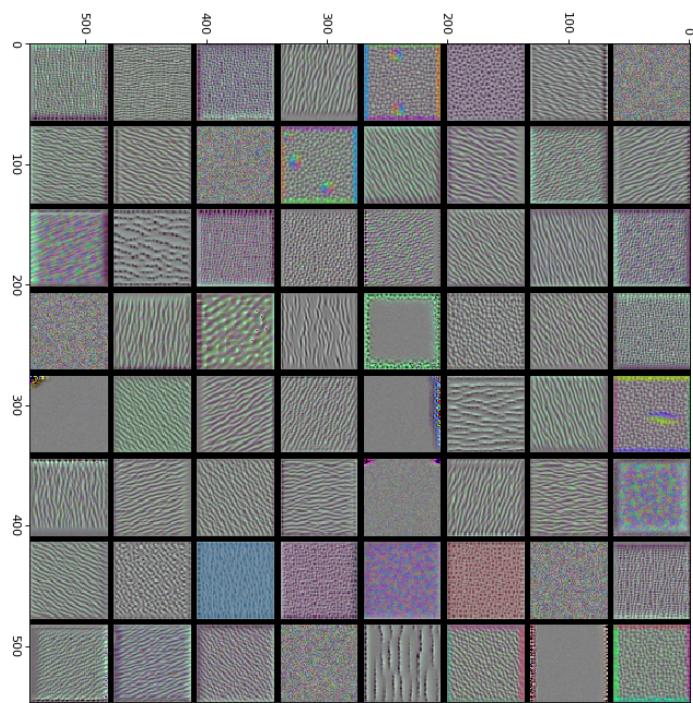


First 64 filters for 1st convolutional layer in block 1: simple edges and colors



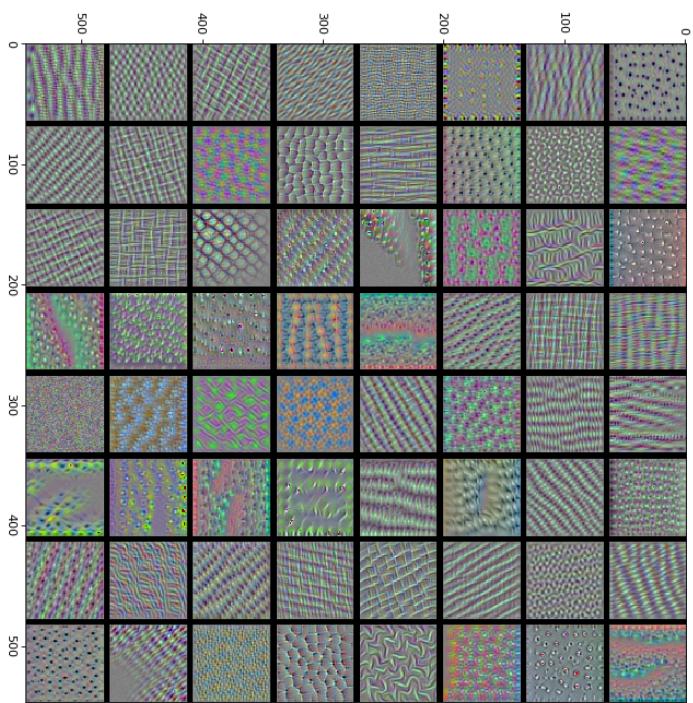


Filters in 2nd block of convolution layers: simple textures (combined edges and colors)



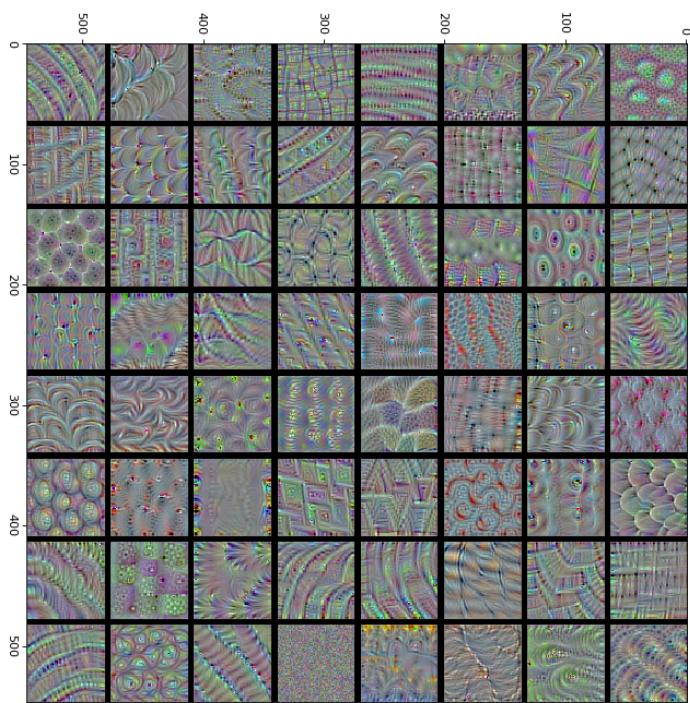


## Filters in 3rd block of convolution layers: more natural textures





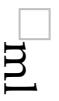
Filters in 4<sup>th</sup> block of convolution layers: feathers, eyes, leaves,...





# Using pretrained networks

- We can re-use pretrained networks instead of training from scratch
- Learned features can be a generic model of the visual world
- Use *convolutional base* to construct features, then train any classifier on new data





- Let's instantiate the VGG16 model (without the dense layers)
- Final feature map has shape (4, 4, 512)

```
conv_base = VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))
```

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0

# Using pre-trained networks: 3 ways

- Fast feature extraction without data augmentation
  - Call predict from the convolutional base
  - Use results to train a dense neural net
- Feature extraction with data augmentation
  - Extend the convolutional base model with a Dense layer
  - Run it end to end on the new data
  - Very expensive (only do this on GPU)
- Fine-tuning
  - Do any of the above two to train a classifier
  - Unfreeze a few of the top convolutional layers
    - Updates only the more abstract representations
  - Jointly train all layers on the new data



# Fast feature extraction without data augmentation

- Extract filtered images and their labels
  - You can use a data generator again

```
generator = datagen.flow_from_directory(dir, target_size=(150, 150),
                                         batch_size=batch_size, class_mode='binary')
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
```

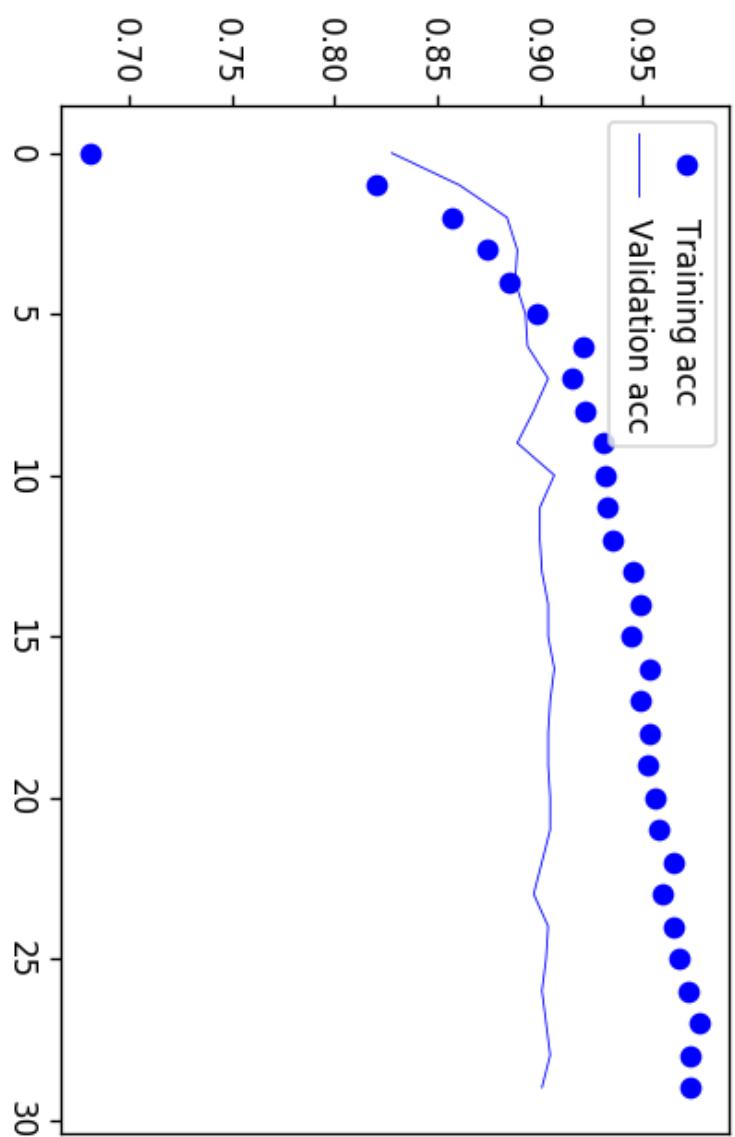
- Build Dense neural net (with Dropout)
- Train and evaluate with the transformed examples

```
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```



- Validation accuracy around 90%, much better than training from scratch
- Still overfitting, despite the Dropout: not enough training data

## Training and validation accuracy



## Feature extraction with data augmentation

- Use data augmentation to get more training data
- Simply add the Dense layers to the convolutional base
- *Freeze* the convolutional base (before you compile)

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
conv_base.trainable = False
```



Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	147146
flatten_4 (Flatten)	(None, 8192)	0
dense_9 (Dense)	(None, 256)	209740
dense_10 (Dense)	(None, 1)	257

Total params: 16,812,353  
Trainable params: 2,097,665  
Non-trainable params: 14,714,688

---



## Data augmentation and training (takes a LONG time)

```
train_datagen = ImageDataGenerator(  
    rescale=1./255, rotation_range=40, width_shift_range=0.2,  
    height_shift_range=0.2, shear_range=0.2, zoom_range=0.2,  
    horizontal_flip=True, fill_mode='nearest')  
train_generator = train_datagen.flow_from_directory(dir,  
    target_size=(150, 150), batch_size=20, class_mode='binary')  
history = model.fit_generator(  
    train_generator, steps_per_epoch=100, epochs=30,  
    validation_data=validation_generator, validation_steps=50)
```



We now get about 96% accuracy, and very little overfitting

□  
ml

□  
ml



## Fine-tuning

- Add your custom network on top of an already trained base network.
- Freeze the base network.
- Train the part you added.
- Unfreeze some layers in the base network.
- Jointly train both these layers and the part you added.

```
for layer in conv_base.layers:  
    if layer.name == 'block5_conv1':  
        set_trainable = True  
  
    else:  
        layer.trainable = False
```



Visualized

□  
ml

□  
ml



- Load trained network, finetune

- Use a small learning rate, large number of epochs
- You don't want to unlearn too much

```
model = load_model(os.path.join(model_dir, 'cats_and_dogs_small_3.h5'))  
model.compile(loss='binary_crossentropy',  
              optimizer=optimizers.RMSprop(lr=1e-5),  
              metrics=['acc'])  
  
history = model.fit_generator(  
    train_generator, steps_per_epoch=100, epochs=100,  
    validation_data=validation_generator,  
    validation_steps=50)
```



- Learning curves are a bit noisy, we can smooth them using a running average

```
def smooth_curve(points, factor=0.8):  
    smoothed_points = []  
    for point in points:  
        if smoothed_points:  
            previous = smoothed_points[-1]  
            smoothed_points.append(previous * factor + point * (1 - factor))  
        else:  
            smoothed_points.append(point)  
  
    return smoothed_points
```



- Results: 97% accuracy (1% better)
- Better validation accuracy, worse validation loss

□  
ml

□  
ml



# Visualizing class activation

- We can also visualize which part of the input image had the greatest influence on the final classification
  - Helpful for interpreting what is learned (or misclassified)
- *Class activation maps*: produce heatmap over the input image
  - Take the output feature map of a convolution layer
  - Weigh every channel (filter) by the gradient of the class with respect to the channel
- Find important channels, see what activates those



- Let's do this with VGG (including the dense layers) and an image from ImageNet

```
model = VGG16(weights='imagenet')
```

ml



- Load image

- Resize to 224 x 224 (what VGG was trained on)
  - Do the same preprocessing (Keras VGG utility)

```
from keras.applications.vgg16 import preprocess_input
img_path = './images/10_elephants.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0) # Transform to batch of size (1, 224, 224,
3)
x = preprocess_input(x)
```



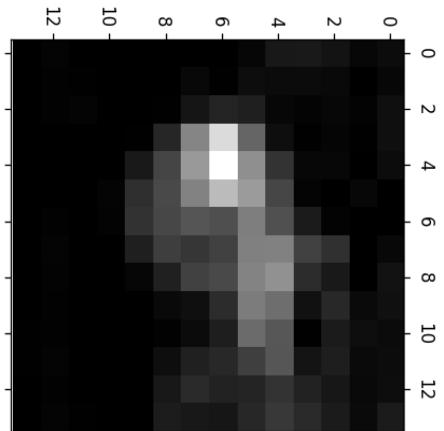
- Sanity test: do we get the right prediction?

```
preds = model.predict(x)
```

```
Predicted: [('n02504458', 'African_elephant', 0.9094207), ('n01871265', 'tusker', 0.08618318), ('n02504013', 'Indian_elephant', 0.004354581)]
```



Visualize the class activation map





- Superimpose on our image

□  
ml



# One more thing: Batch Normalization

- Normalization (in general) aims to make different examples more similar to each other
  - Easier to learn and generalize
- Standardization (centering the data to 0 and scaling to 1 stddev) is very common
  - This assumes that the data is normally distributed
- Batch normalization layer adaptively normalize data, even as the mean and variance change over time during training.
  - It works by internally maintaining an exponential moving average of the batch-wise mean and variance of the data seen during training.
  - Helps with gradient propagation, allows for deeper networks.



BatchNormalization layer is typically used after a convolutional or densely connected layer:

```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))
conv_model.add(layers.BatchNormalization())
dense_model.add(layers.Dense(32, activation='relu'))
dense_model.add(layers.BatchNormalization())
```



# Take-aways

- Convnets are ideal for attacking visual-classification problems.
- They learn a hierarchy of modular patterns and concepts to represent the visual world.
- Representations are easy to inspect
- Data augmentation helps fight overfitting
- Batch normalization helps train deeper networks
- You can use a pretrained convnet to do feature extraction and fine-tuning

