

Convolutional Neural Networks for Image Problems

Shusen Wang

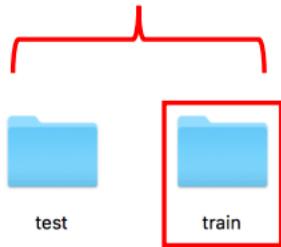
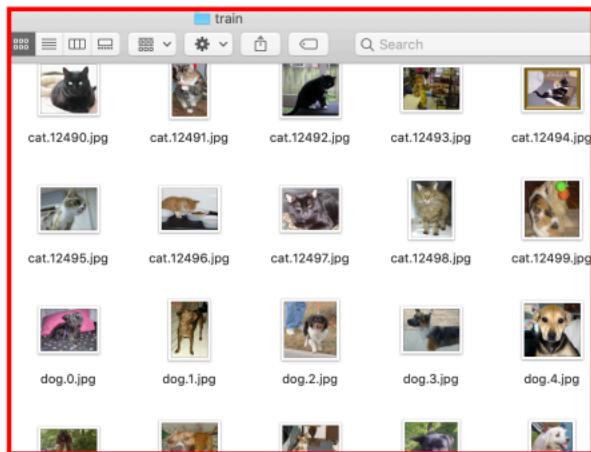
The Dogs vs. Cats Dataset

The Dogs vs. Cats Dataset



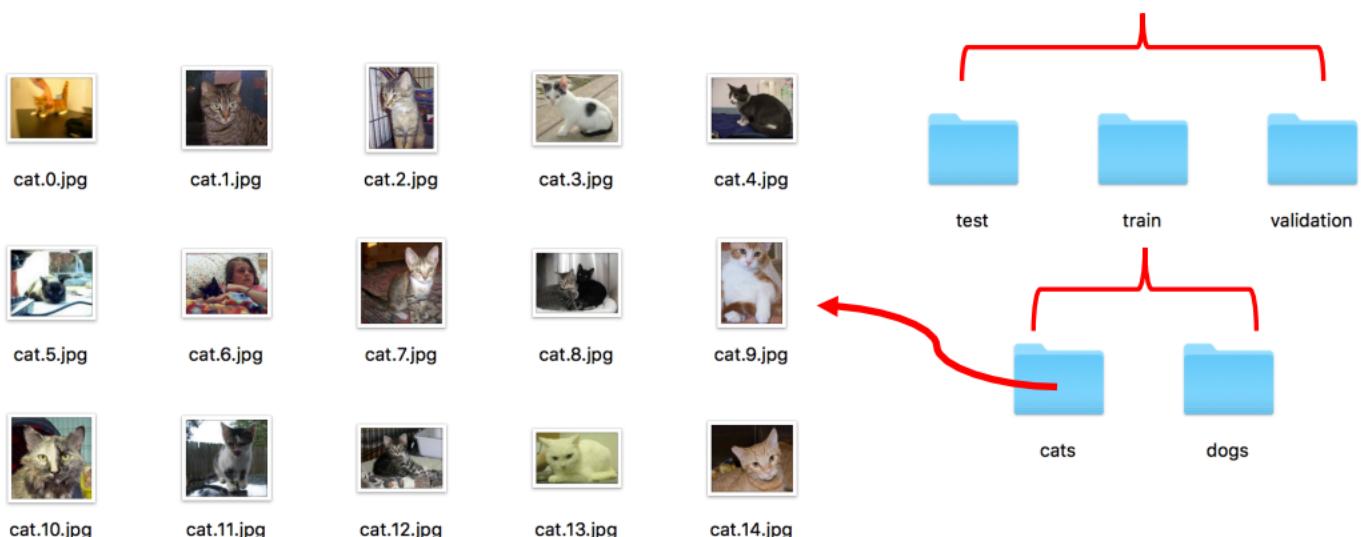
The Dogs vs. Cats Dataset

Download: <https://www.kaggle.com/c/dogs-vs-cats/data>



The Dogs vs. Cats Dataset

Download: <https://www.kaggle.com/c/dogs-vs-cats/data>



The Dogs vs. Cats Dataset

- The dataset has 25,000 training samples.
- I use a subset:
 - 2000 images for training,
 - 1000 images for validation,
 - 1000 images for test.

Implement a CNN Using Keras

1. Load and Process the Dataset

- Currently, the data sit on a drive as JPEG files.
- Data processing:
 1. Read the picture files.
 2. Decode the JPEG content to order-3 tensors.
 3. Resize the images to the same shape, e.g., $150 \times 150 \times 3$.
 4. Rescale the pixel values (between 0 and 255) to the $[0, 1]$ interval.

1. Load and Process the Dataset

```
from keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    # This is the target directory
    train_dir,
    # All images will be resized to 150x150
    target_size=(150, 150),
    batch_size=20,
    # Since we use binary_crossentropy loss, we need binary labels
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')
```

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.

1. Load and Process the Dataset

```
for data_batch, labels_batch in train_generator:  
    print('data batch shape:', data_batch.shape)  
    print('labels batch shape:', labels_batch.shape)  
    break
```

```
data batch shape: (20, 150, 150, 3)  
labels batch shape: (20,)
```

2. Build the CNN

```
from keras import layers
from keras import models

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'))
```

2. Build the CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_1 (MaxPooling2D)	(None, 74, 74, 32)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 36, 36, 64)	0
<hr/>		
conv2d_3 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_3 (MaxPooling2D)	(None, 17, 17, 128)	0
<hr/>		
conv2d_4 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 128)	0
<hr/>		
flatten_1 (Flatten)	(None, 6272)	0
<hr/>		
dense_1 (Dense)	(None, 512)	3211776
<hr/>		
dense_2 (Dense)	(None, 1)	513
<hr/>		
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		

3. Train the CNN

Specify: optimization method, learning rate (LR), loss function, and metric.

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

3. Train the CNN

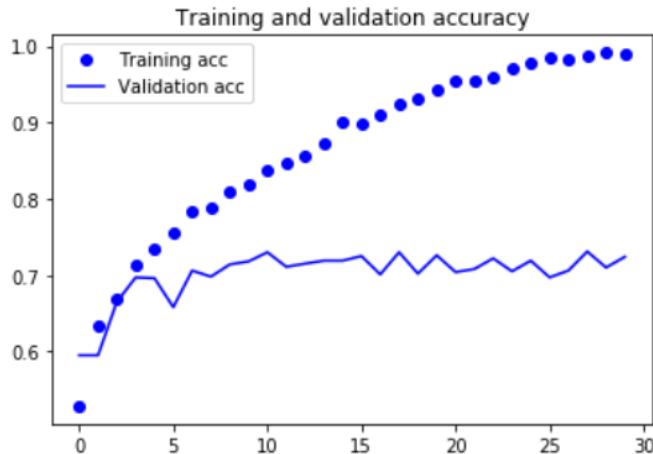
```
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100,   
    epochs=30,  
    validation_data=validation_generator,  
    validation_steps=50)
```

- Totally $n = 2000$ training samples.
- Batch size is $b = 20$.
- Thus $\frac{n}{b} = 100$ batches per epoch.

```
Epoch 1/30  
100/100 [=====] - 9s - loss: 0.6898 - acc: 0.5285 - val_loss: 0.6724 - val_acc: 0.5950  
Epoch 2/30  
100/100 [=====] - 8s - loss: 0.6543 - acc: 0.6340 - val_loss: 0.6565 - val_acc: 0.5950  
Epoch 3/30  
100/100 [=====] - 8s - loss: 0.6143 - acc: 0.6690 - val_loss: 0.6116 - val_acc: 0.6650  
Epoch 4/30  
100/100 [=====] - 8s - loss: 0.5626 - acc: 0.7125 - val_loss: 0.5774 - val_acc: 0.6970  
●  
●  
●  
Epoch 29/30  
100/100 [=====] - 8s - loss: 0.0375 - acc: 0.9915 - val_loss: 0.9987 - val_acc: 0.7100  
Epoch 30/30  
100/100 [=====] - 8s - loss: 0.0387 - acc: 0.9895 - val_loss: 1.0139 - val_acc: 0.7240
```

4. Examine the Results

Plot the *accuracy* against *epochs* (1 epoch = 1 pass over the data).



- Training acc: 99.0%
- Validation acc: 72.4%
- Overfitting!

4. Examine the Results

Plot the *loss* against *epochs* (1 epoch = 1 pass over the data).



- Training loss is decreasing.
- Validation loss decreases and then increase.

Why Overfitting?

```
=====
```

Total params: 3,453,121

Trainable params: 3,453,121

Non-trainable params: 0

- 2K images for training,
- 3.4M parameters.

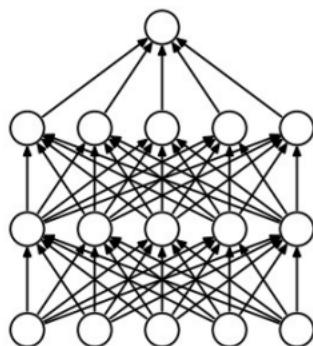
- Over $3M$ parameters; Just $2K$ training samples.
- Overfitting is not surprising.

Trick: Dropout

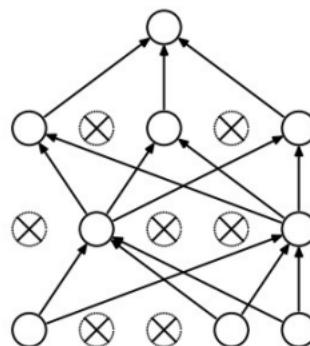
Dropout: Basic Idea

- Train

- In each iteration of training (1 forward + 1 backward), randomly mask 50% (or an arbitrary percentage) of the neurons.



(a) Standard Neural Net



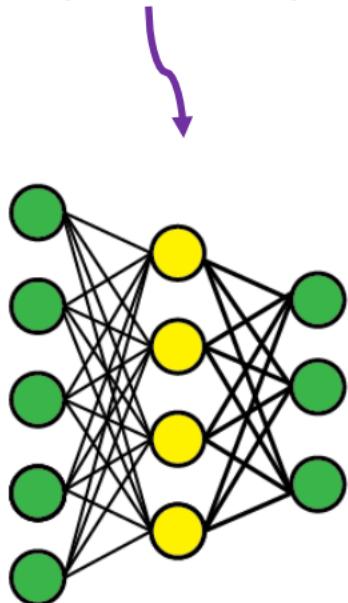
(b) After applying dropout.

Dropout: Basic Idea

- Train
 - In each iteration of training (1 forward + 1 backward), randomly mask 50% (or an arbitrary percentage) of the neurons.
- Prediction
 - No dropout.
 - Use all the parameters.

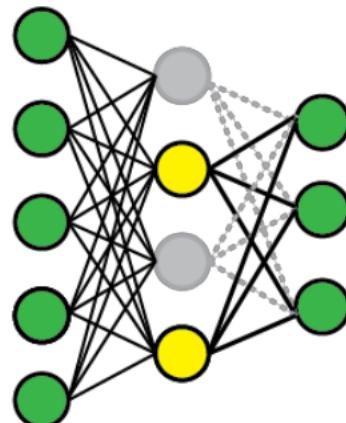
Dropout: Implementation

Do dropout for this layer



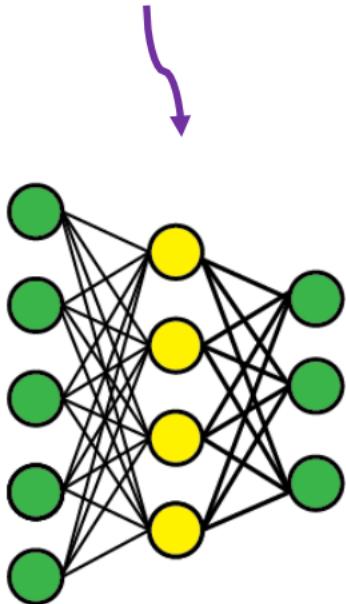
- **For a batch of training samples**

- Randomly choose 50% of the neurons.
- Set the selected neurons to zeros.
- Multiply the unselected neurons by $\frac{1}{0.5} = 2$.

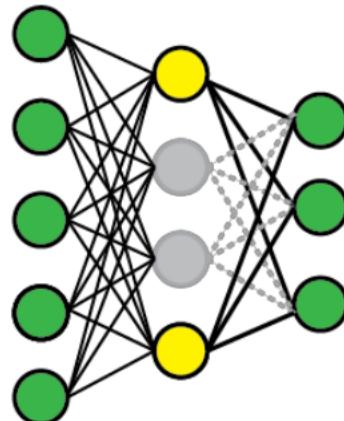


Dropout: Implementation

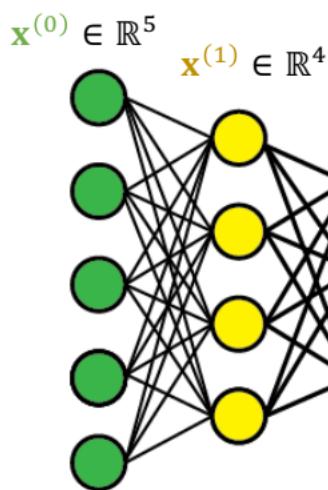
Do dropout for this layer



- For a batch of training samples...
- For another batch, do an independent random sampling.



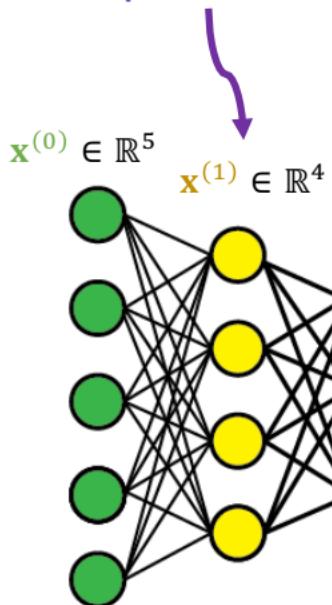
Dropout: Implementation



- **Input:** vector $x^{(0)} \in \mathbb{R}^5$.
- $z^{(1)} = W^{(0)} x^{(0)} \in \mathbb{R}^4$.
- $x^{(1)} = \max\{0, z^{(1)}\} \in \mathbb{R}^4$.
- $z^{(2)} = W^{(1)} x^{(1)} \in \mathbb{R}^3$.
- **Output:** $\text{SoftMax}(z^{(2)}) \in \mathbb{R}^3$.

Dropout: Implementation

Do dropout for this layer



- **Input:** vector $x^{(0)} \in \mathbb{R}^5$.
- $z^{(1)} = W^{(0)} x^{(0)} \in \mathbb{R}^4$.
- $x^{(1)} = \max\{0, z^{(1)}\} \in \mathbb{R}^4$.
- Add a dropout layer →
- $z^{(2)} = W^{(1)} \tilde{x}^{(1)} \in \mathbb{R}^3$.
- **Output:** SoftMax($z^{(2)}$) $\in \mathbb{R}^3$.

• $m \in \mathbb{R}^4$ is a random vector.
(Each entry is 0 or 2, w.p. 50%).

• Apply m to $x^{(1)}$:

$$\tilde{x}^{(1)} = m \circ x^{(1)}$$

“ \circ ” is elementwise multiplication.

Keras's Dropout Layer

- Dropout only before the 1st FC layer to regularize the 1st FC layer.
- Because the 1st FC layer has many trainable parameters.

```
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'))

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

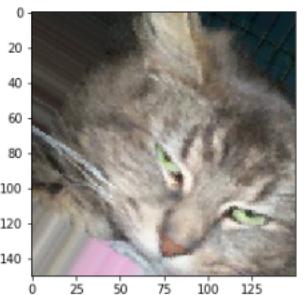
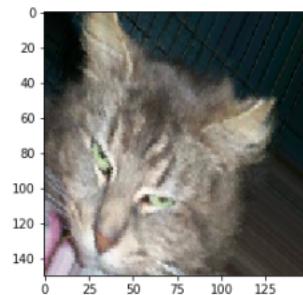
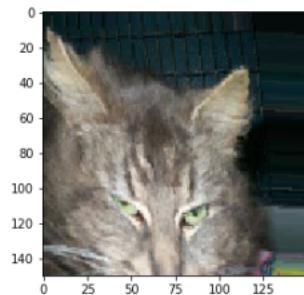
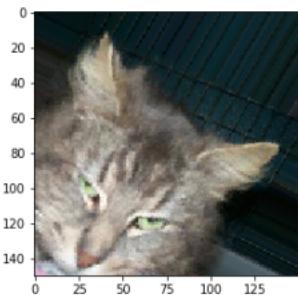
Why Does Dropout Work?

- In training, dropout forces the network to make decision based on part of the features.
- Dropout is a regularization.
 - Alleviate overfitting.
 - Like the L1 and L2 norm regularizations.
 - But dropout is empirically better.

Trick: Data Augmentation

Data Augmentation

- Data augmentation: generating more training data from existing training samples.
- E.g., flip, rotation, crop, shift, add random noise.



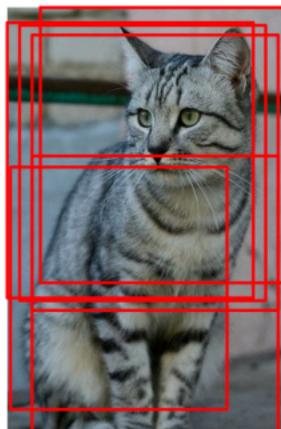
Data Augmentation: Examples

Horizontal Flips



Data Augmentation: Examples

Random crops and scales



Data Augmentation: Examples

Color Jitter (randomize contrast and brightness)



Setup Data Augmentation Using Keras

```
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, )
```

```
train_generator = train_datagen.flow_from_directory(  
    # This is the target directory  
    train_dir,  
    # All images will be resized to 150x150  
    target_size=(150, 150),  
    batch_size=32,  
    # Since we use binary_crossentropy loss, we need binary labels  
    class_mode='binary' )
```

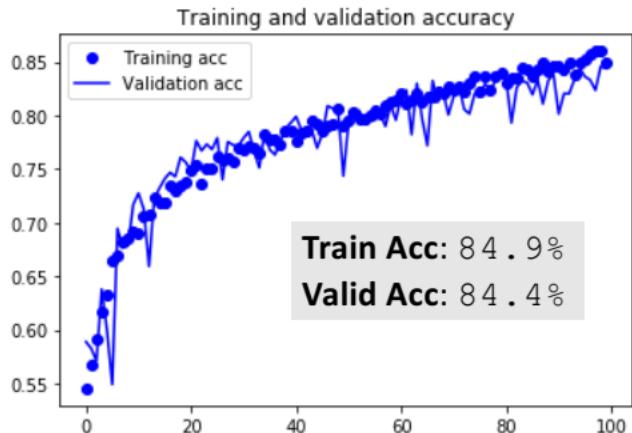
Train the CNN

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

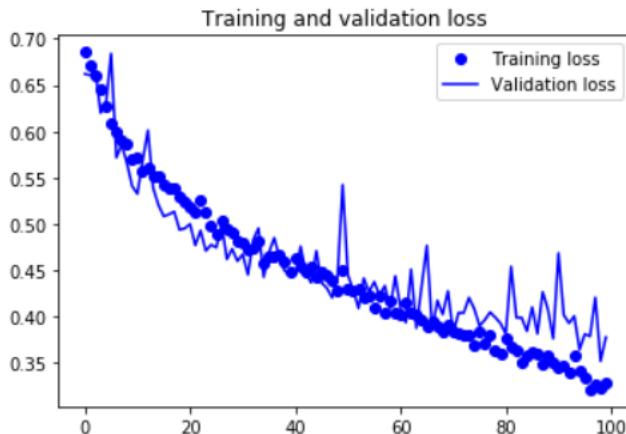
```
Epoch 1/100  
100/100 [=====] - 24s - loss: 0.6857 - acc: 0.5447 - val_loss: 0.6620 - val_acc: 0.5888  
Epoch 2/100  
100/100 [=====] - 23s - loss: 0.6710 - acc: 0.5675 - val_loss: 0.6606 - val_acc: 0.5825  
Epoch 3/100  
100/100 [=====] - 22s - loss: 0.6609 - acc: 0.5913 - val_loss: 0.6663 - val_acc: 0.5711  
●  
●  
●  
Epoch 99/100  
100/100 [=====] - 22s - loss: 0.3255 - acc: 0.8581 - val_loss: 0.3518 - val_acc: 0.8460  
Epoch 100/100  
100/100 [=====] - 22s - loss: 0.3280 - acc: 0.8491 - val_loss: 0.3776 - val_acc: 0.8439
```

Examine the Results

accuracy against *epochs*



loss against *epochs*



Take-Home Message

- To train a ConvNet for images, **always** use data augmentation.
 - It gives you more data for free!
- If a fully-connected layer has too many parameters, put a dropout layer before it.
 - Regularization prevents overfitting.
 - The training becomes a little slower.

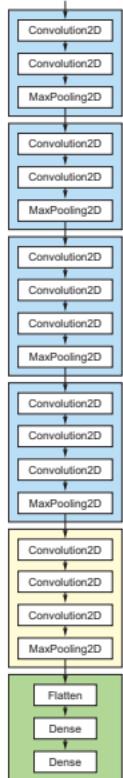
Trick: Pretrain

Train a Deep Neural Network?

```
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'))

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

- We have trained a neural net with 4 Conv Layers and 2 FC Layers.
- Relatively shallow.



the VGG16 network

Train a Deep Neural Network?

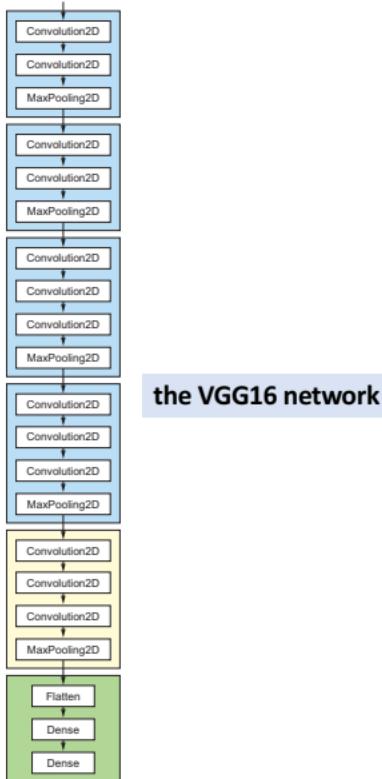
Can we train a deep neural network?

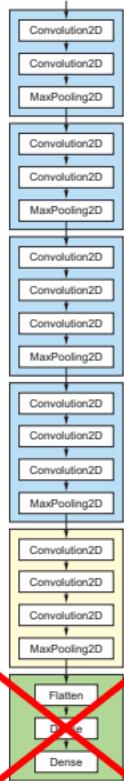
- It is hard!
 - The number of parameters is huge.
 - The deep network has a huge capacity.
 - We have merely $2K$ training samples.
- Naively training a deep network will results in overfitting, surely.

Solution: pretrain

Pretrain

1. Pre-train a deep net on large-scale dataset,
e.g., ImageNet (14M images with labels).





Pretrain

1. Pre-train a deep net on large-scale dataset, e.g., ImageNet (14M images with labels).
2. Remove the high-level layers. *Why?*

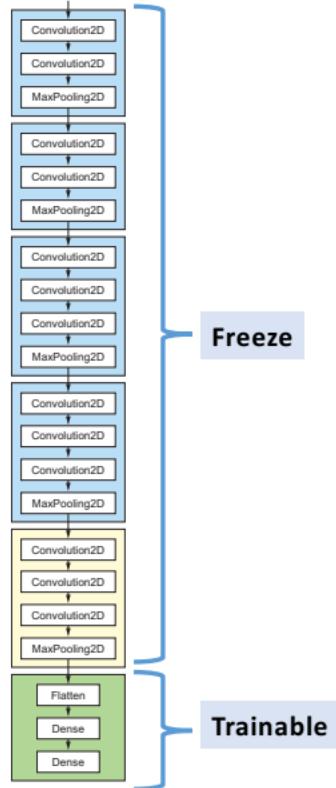
The output of VGG16 is 1000-dim, because ImageNet has 1000 classes

We need 1-dim output (binary classification --- cats v.s. dogs).

The output-layer of VGG16 has Softmax activation function, because ImageNet is multiclass.

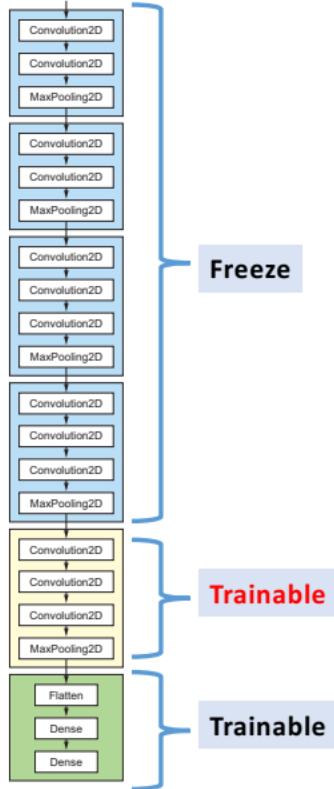
We use sigmoid activation function for *binary* classification.

Remove the high-level layers



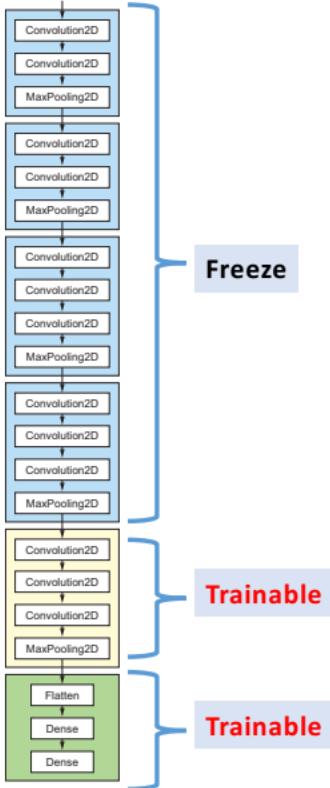
Pretrain

1. Pre-train a deep net on large-scale dataset, e.g., ImageNet (14M images with labels).
2. Remove the high-level layers.
3. Build new high-level layers (randomly initialized).
4. Freeze the low-level layers; Train the high-level layers.



Pretrain

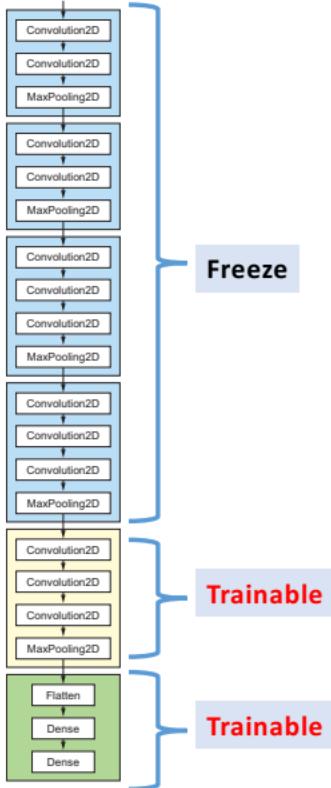
1. Pre-train a deep net on large-scale dataset, e.g., ImageNet (14M images with labels).
2. Remove the high-level layers.
3. Build new high-level layers (randomly initialized).
4. Freeze the low-level layers; Train the high-level layers.
5. Optional: **Fine-tune** the top Conv Layers.



Pretrain

1. Pre-train a deep net on large-scale dataset, e.g., ImageNet (14M images with labels).
2. Remove the high-level layers.
3. Build new high-level layers (randomly initialized).
4. Freeze the low-level layers; **Train the high-level layers.**
5. Optional: **Fine-tune the top Conv Layers.**

Question: Can Steps 4 & 5 be merged?



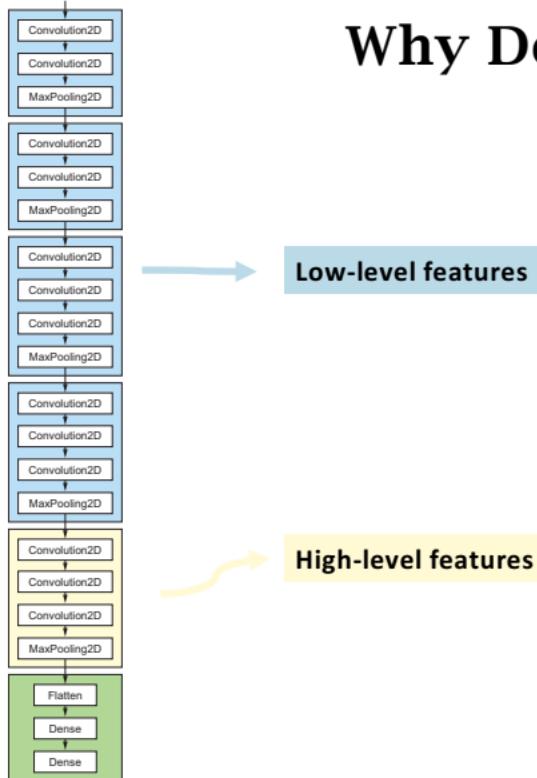
Pretrain

1. Pre-train a deep net on large-scale dataset, such as ImageNet (14 million images).

NO. Merging Steps 4 & 5 is a bad idea.

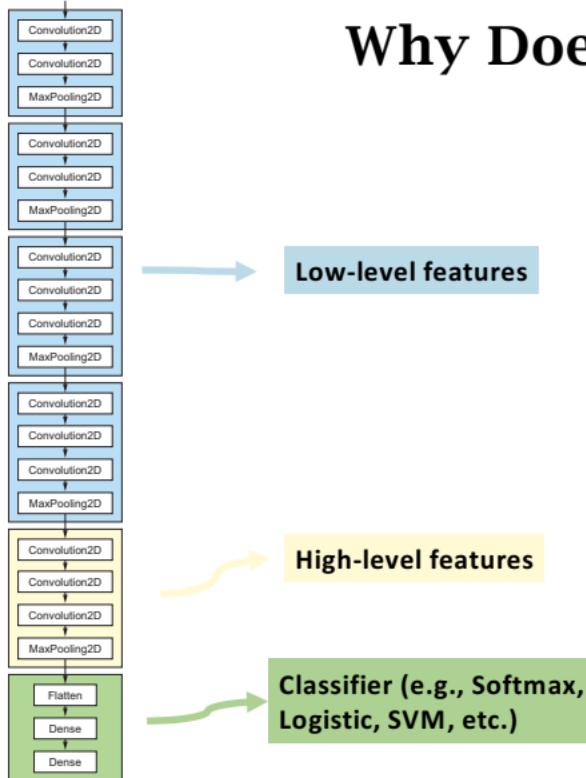
- If the top layers are random, the initial gradients are large.
- The large gradient will destroy the Conv layers.
- Thus, train the Conv layer after training the top layers.

Question: Can Steps 4 & 5 be merged?



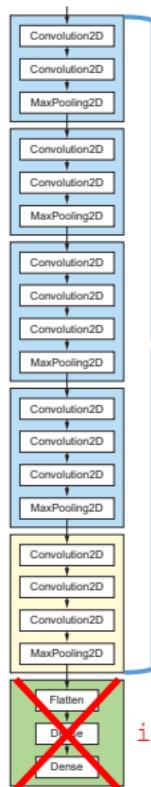
Why Does Pretrain Work?

- The ***Conv Layers*** are for *feature extraction*.
 - The **low-level feature** (edges, shapes, patterns, etc.) trained on the ImageNet are effective to other image problems.
 - The **high-level features** are useful, but less effective.



Why Does Pretrain Work?

- The **Conv Layers** are for *feature extraction*.
 - The **low-level feature** (edges, shapes, patterns, etc.) trained on the ImageNet are effective to other image problems.
 - The **high-level features** are useful, but less effective.
- Think of the **high-level FC layers** as a classifier which takes the extracted features as input.
- *Less trainable parameters, less prone to overfitting.*



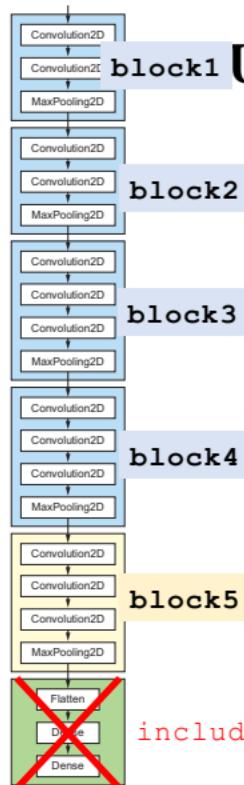
Use a VGG16 Net Pretrained on ImageNet

```
from keras.applications import VGG16

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

conv_base.summary()

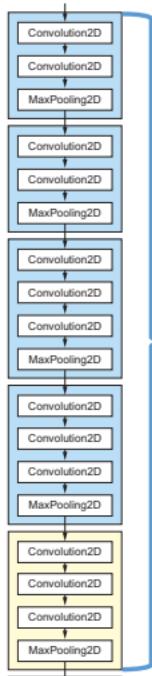
include_top=False
```



Use a VGG16 Net Pretrained on ImageNet

Layer (type)	Output Shape	Param #
input_2 (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
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

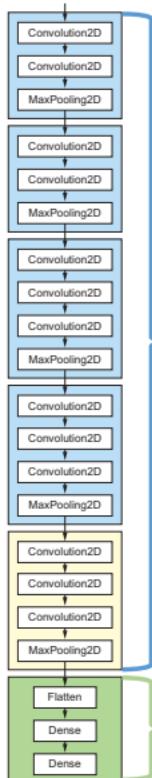
Total params: 14,714,688
 Trainable params: 14,714,688
 Non-trainable params: 0



Use a VGG16 Net Pretrained on ImageNet

```
from keras import models
from keras import layers

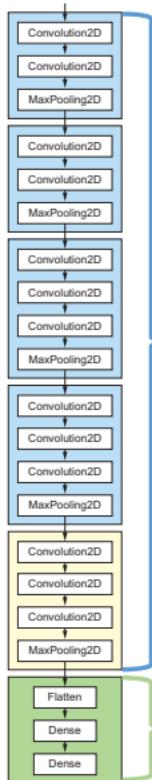
model = models.Sequential()
model.add(conv_base)
```



Use a VGG16 Net Pretrained on ImageNet

```
from keras import models
from keras import layers

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'))
```



Use a VGG16 Net Pretrained on ImageNet

```
from keras import models
from keras import layers

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'))

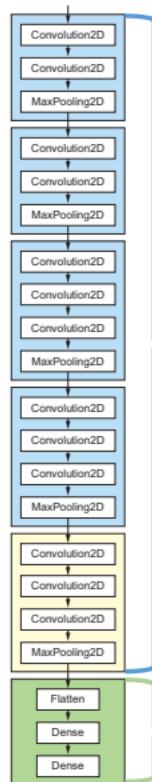
model.summary()
```

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
dense_2 (Dense)	(None, 1)	257

Total params: 16,812,353

Trainable params: 16,812,353

Non-trainable params: 0



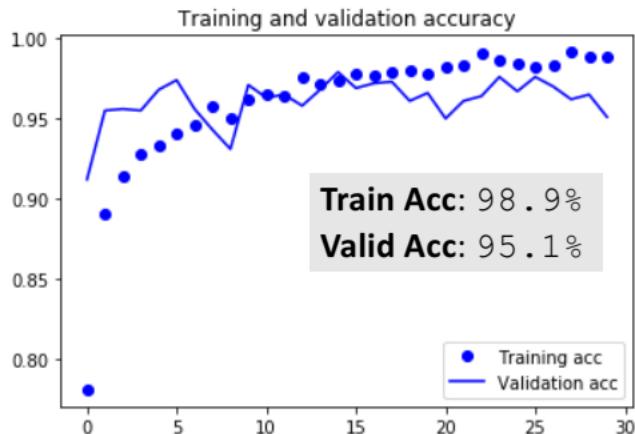
Use a VGG16 Net Pretrained on ImageNet

```
conv_base.trainable = False  
model.summary()
```

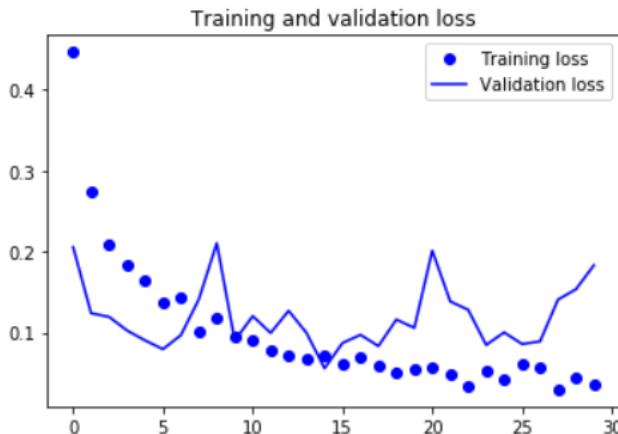
Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
dense_2 (Dense)	(None, 1)	257
<hr/>		
Total params: 16,812,353		
Trainable params: 2,097,665		
Non-trainable params: 14,714,688		

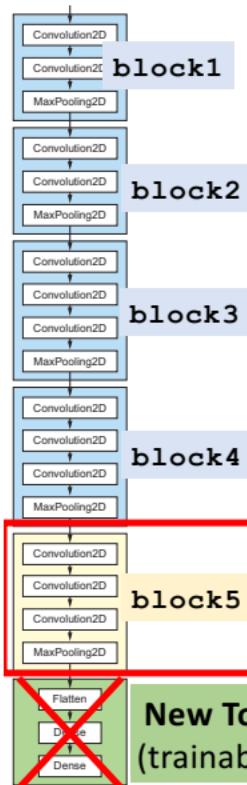
After Training the New Top

accuracy against *epochs*



loss against *epochs*

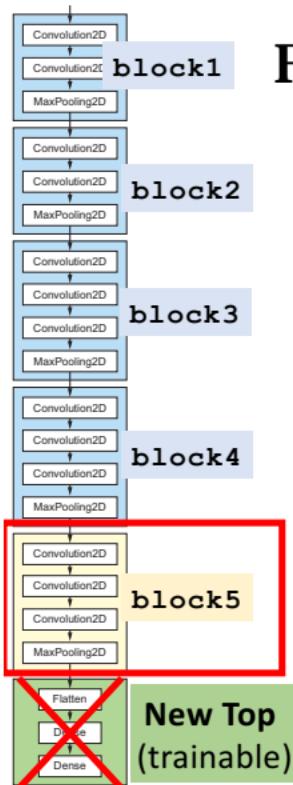




Fine Tuning the Top Conv Layers

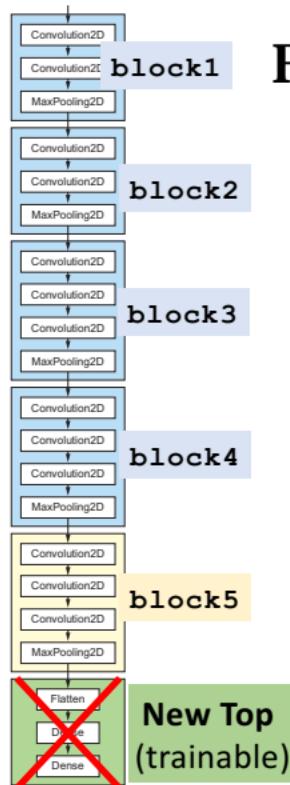
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

Total params: 14,714,688
 Trainable params: 14,714,688
 Non-trainable params: 0



```
trainable_layer_names = ['block5_conv1', 'block5_conv2',
                        'block5_conv3', 'block5_pool']
conv_base.trainable = True

for layer in conv_base.layers:
    if layer.name in trainable_layer_names:
        layer.trainable = True
    else:
        layer.trainable = False
```



Fine Tuning the Top Conv Layers

```
model.summary()
```

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
dense_2 (Dense)	(None, 1)	257
Total params:	16,812,353	
Trainable params:	9,177.089	
Non-trainable params:	7,635,264	

Fine Tuning the Top Conv Layers

Re-compile before training

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=[ 'acc' ])
```

Fine Tuning the Top Conv Layers

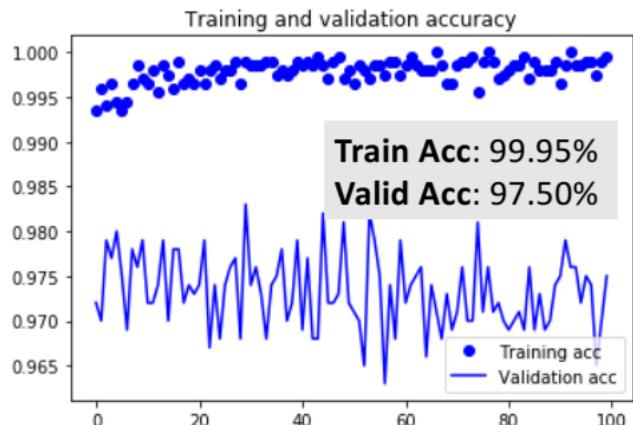
```
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100,  
    epochs=100,  
    validation_data=validation_generator,  
    validation_steps=50)
```

```
Epoch 1/100  
100/100 [=====] - 32s - loss: 0.0215 - acc: 0.9935 - val_loss: 0.0980 - val_acc: 0.9720  
Epoch 2/100  
100/100 [=====] - 32s - loss: 0.0131 - acc: 0.9960 - val_loss: 0.1247 - val_acc: 0.9700  
Epoch 3/100  
100/100 [=====] - 32s - loss: 0.0140 - acc: 0.9940 - val_loss: 0.1044 - val_acc: 0.9790  
•  
•  
•
```

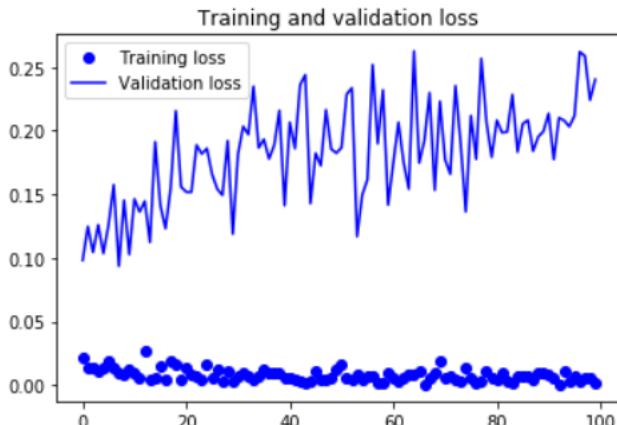
```
Epoch 99/100  
100/100 [=====] - 33s - loss: 0.0060 - acc: 0.9990 - val_loss: 0.2242 - val_acc: 0.9700  
Epoch 100/100  
100/100 [=====] - 33s - loss: 0.0010 - acc: 0.9995 - val_loss: 0.2403 - val_acc: 0.9750
```

Fine Tuning the Top Conv Layers

accuracy against *epochs*



loss against *epochs*



Fine Tuning the Top Conv Layers

Evaluate the model on the test set

```
test_generator = test_datagen.flow_from_directory(  
    test_dir,  
    target_size=(150, 150),  
    batch_size=20,  
    class_mode='binary')  
  
test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)  
print('test acc:', test_acc)
```

Found 1000 images belonging to 2 classes.
test acc: 0.967999992371

Summary

Summary of the Results

- A small ConvNet (4 Conv Layers + 2 FC Layers), with $3.5M$ parameters.
 - Training accuracy: 99.0%
 - Validation accuracy: 72.4%
- The small ConvNet + 1 Dropout Layer + Data Augmentation.
 - Training accuracy: 84.9%
 - Validation accuracy: 84.4%
- VGG16 Net pre-trained on the ImageNet. (Train the new top layers.)
 - Training accuracy: 98.9%
 - Validation accuracy: 95.1%
- VGG16 Net pre-trained on the ImageNet. (Fine-tune the top Conv Layers.)
 - Training accuracy: 99.95%
 - Validation accuracy: 97.5%