Lecture 7

Tian Han

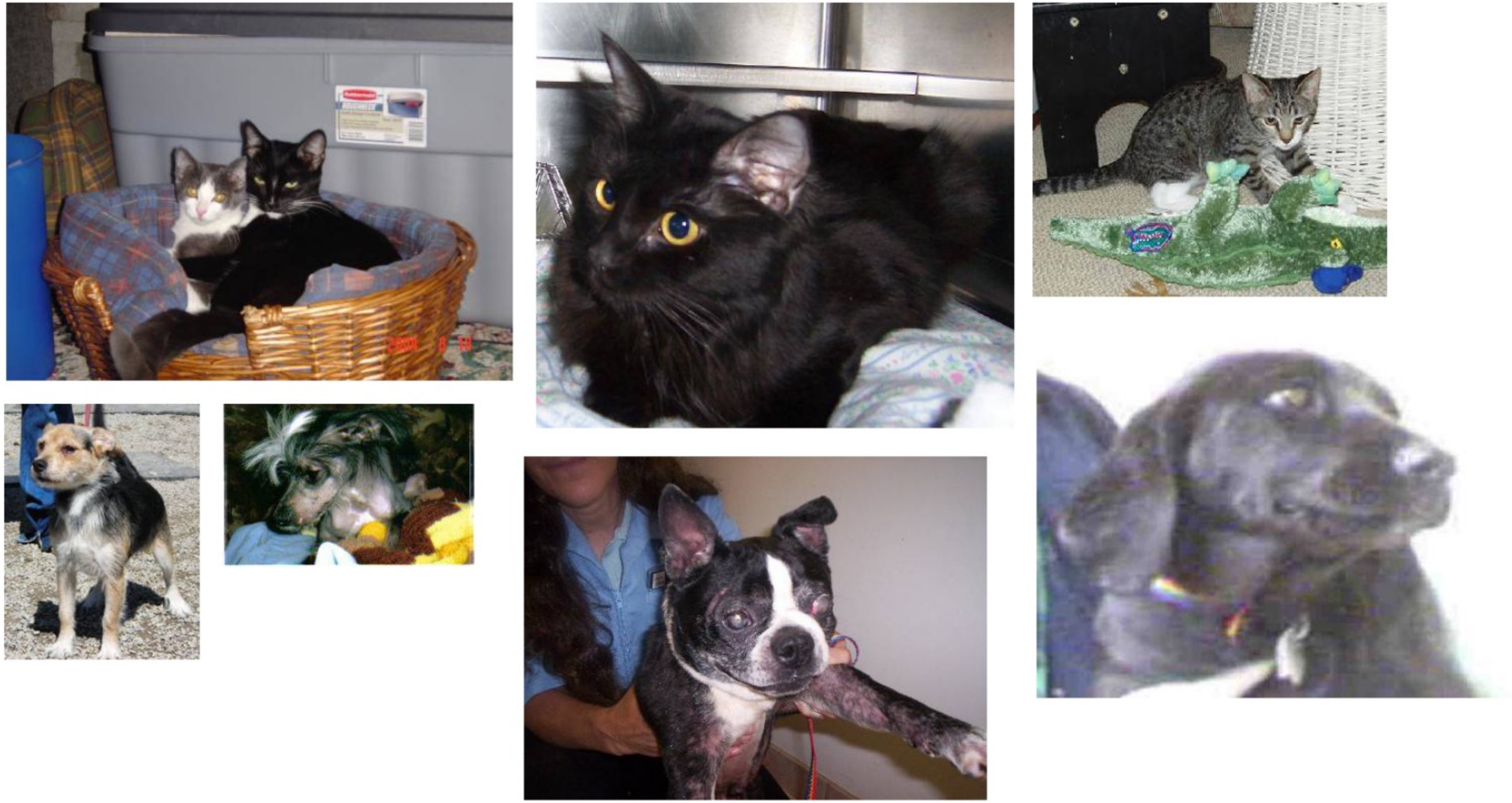
Outline

- Tricks
- Architecture

Some tricks

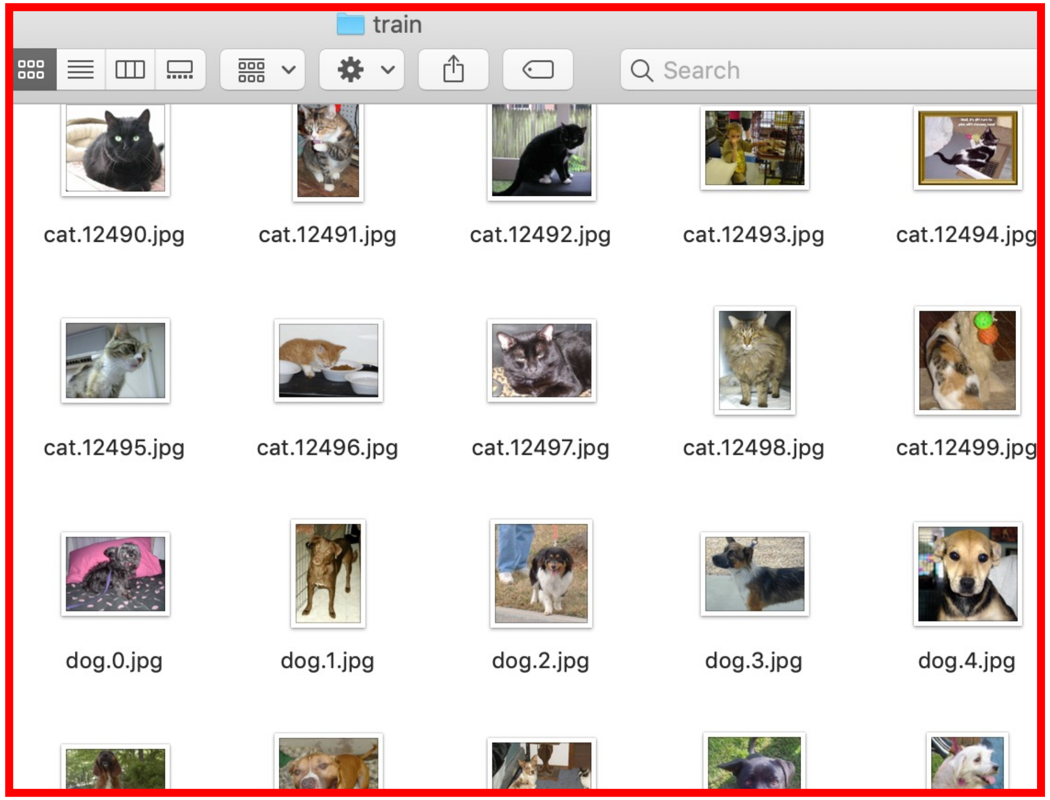
The Dogs vs. Cats Dataset

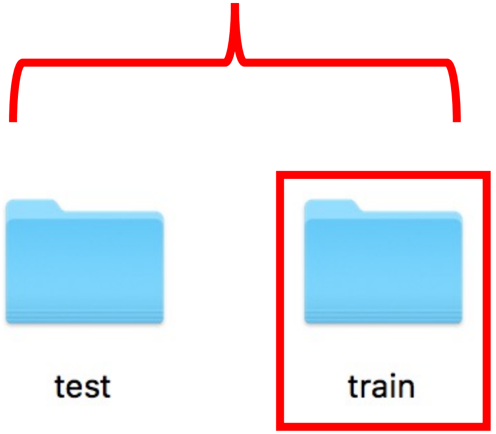
The Dogs vs. Cats Dataset



The Dogs vs. Cats Dataset

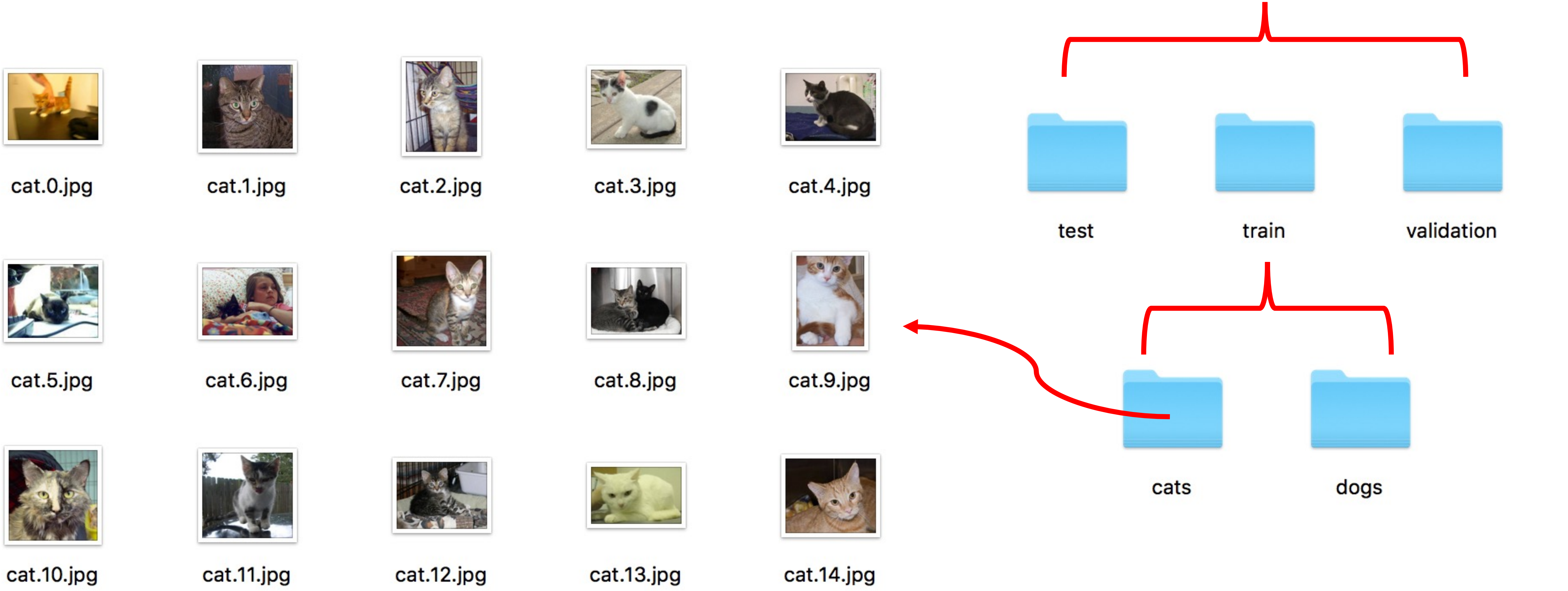
Download: https://www.kaggle.com/c/dogs-vs-cats/data





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 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)
```

1. Load and Process the Dataset

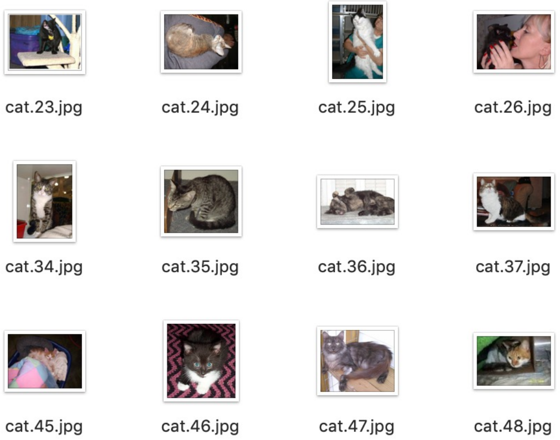
```python
from keras.preprocessing.image import ImageDataGenerator

# 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')
```

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
```

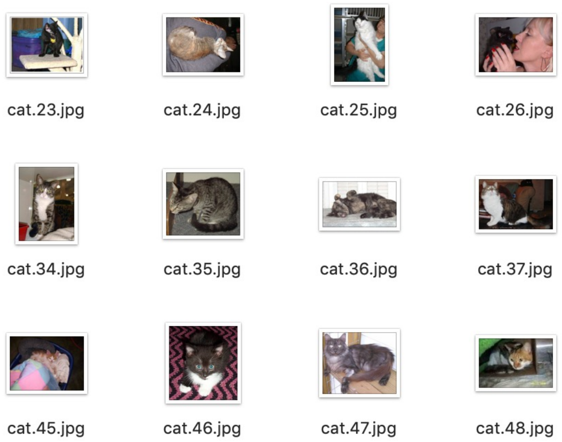
```
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

1. Load and Process the Dataset



Disk → **Memory** → **GPU**

1. Load and Process the Dataset



Disk → **Memory** → **GPU**

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 #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
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)
```

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.

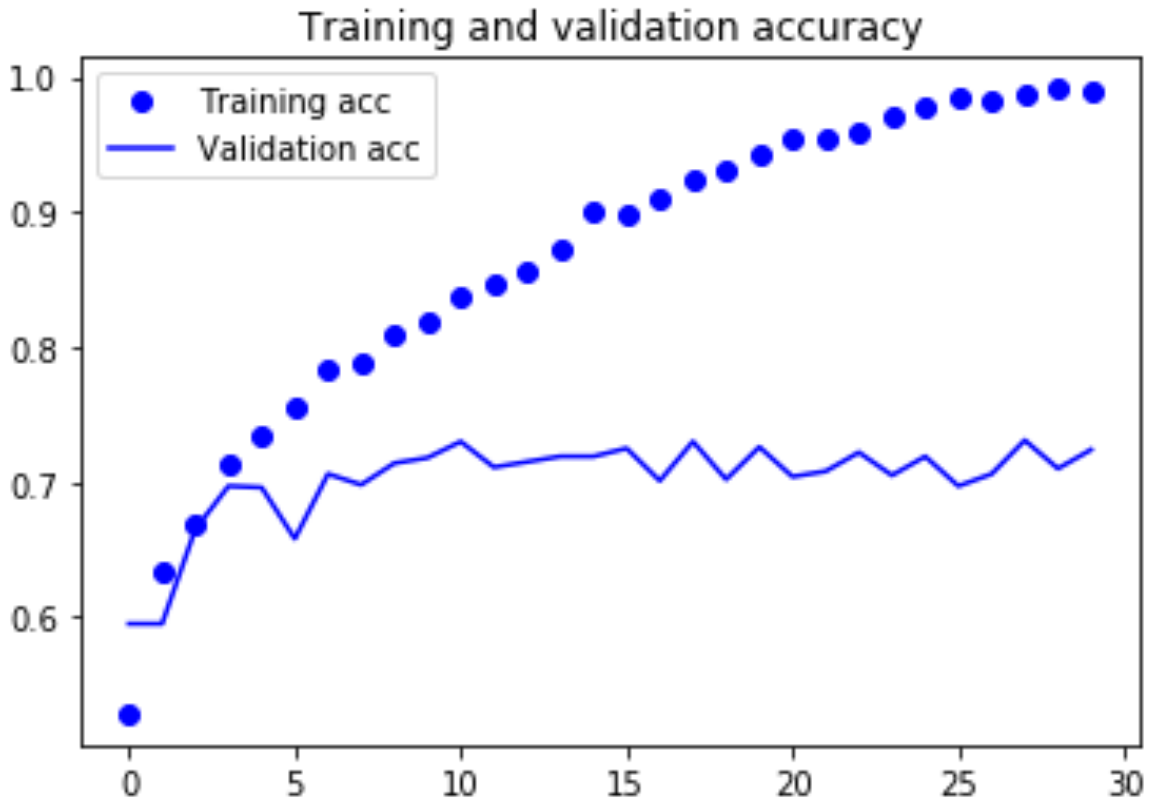
```
Epoch 1/30
100/100 [==============================] - 9s - loss: 0.6898 - acc: 0.5285 - val_loss: 0.6724 - val_acc: 0.5950
Epoch 2/30
100/100 [==============================] - 8s - loss: 0.6543 - acc: 0.6340 - val_loss: 0.6565 - val_acc: 0.5950
Epoch 3/30
100/100 [==============================] - 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

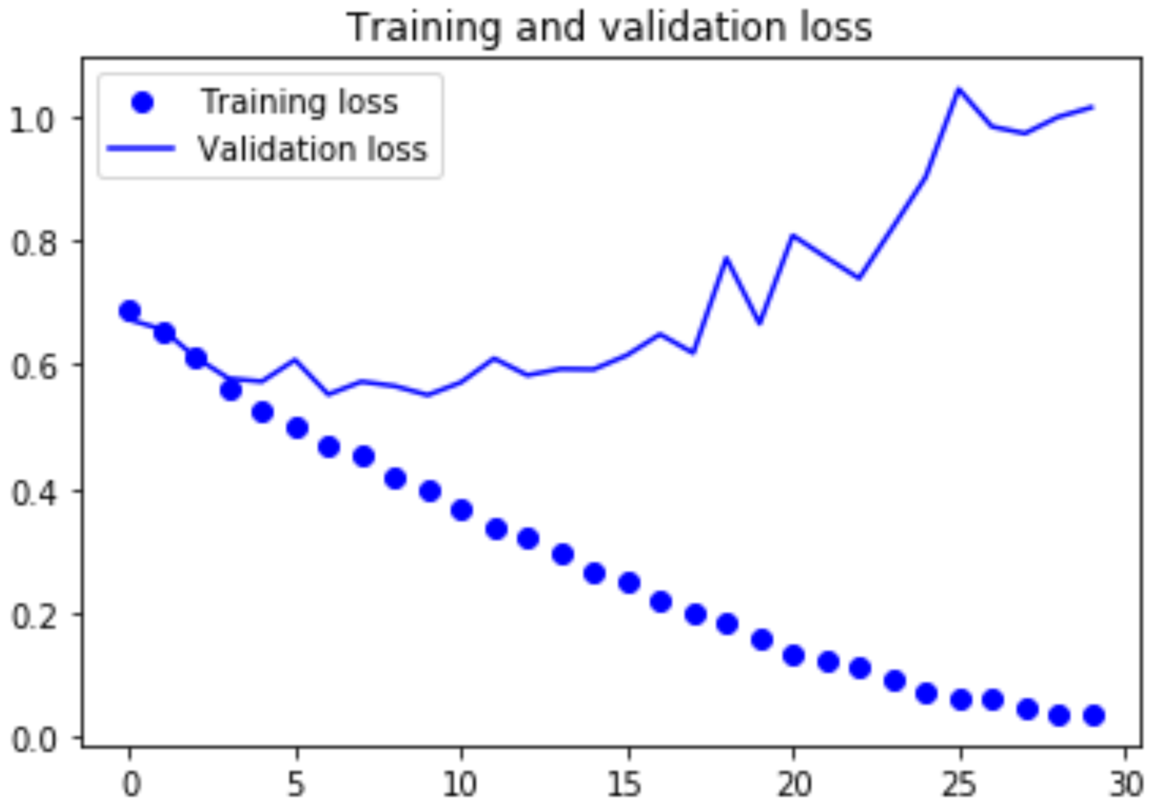
Plot the ***accuracy*** against ***epochs*** (1 epoch = 1 pass over the data).



- Training acc: 99.0%
- Validation acc: 72.4%
- Likely 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?

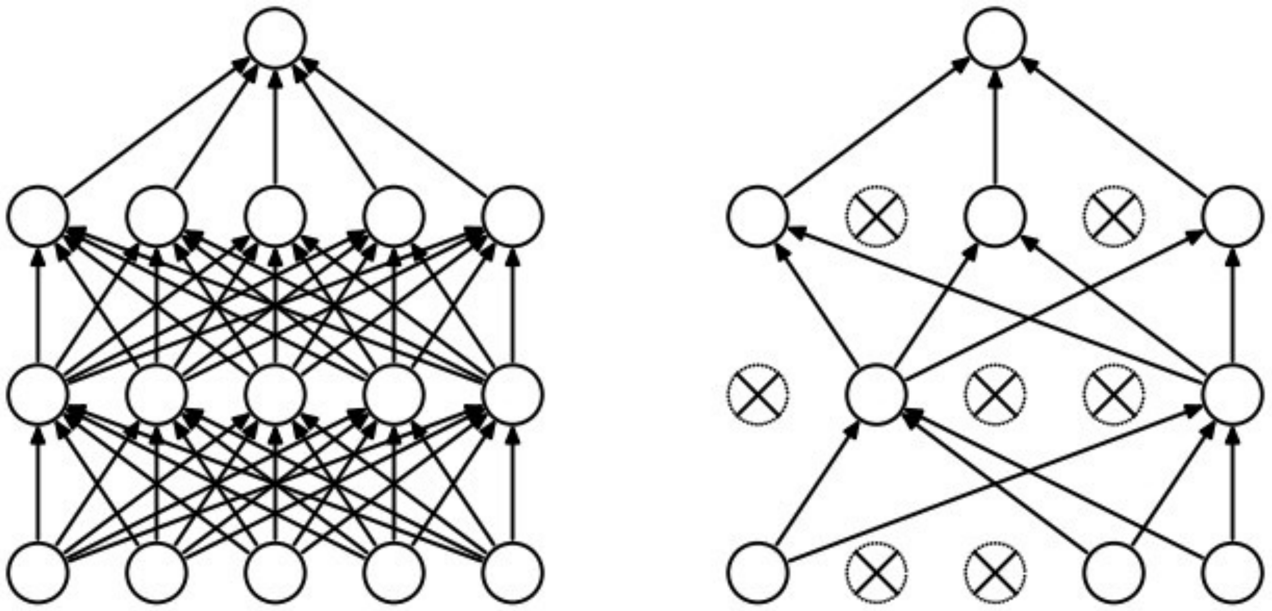
```
================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
________________________________
```

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

Trick 1: Dropout

Dropout: Basic Idea

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



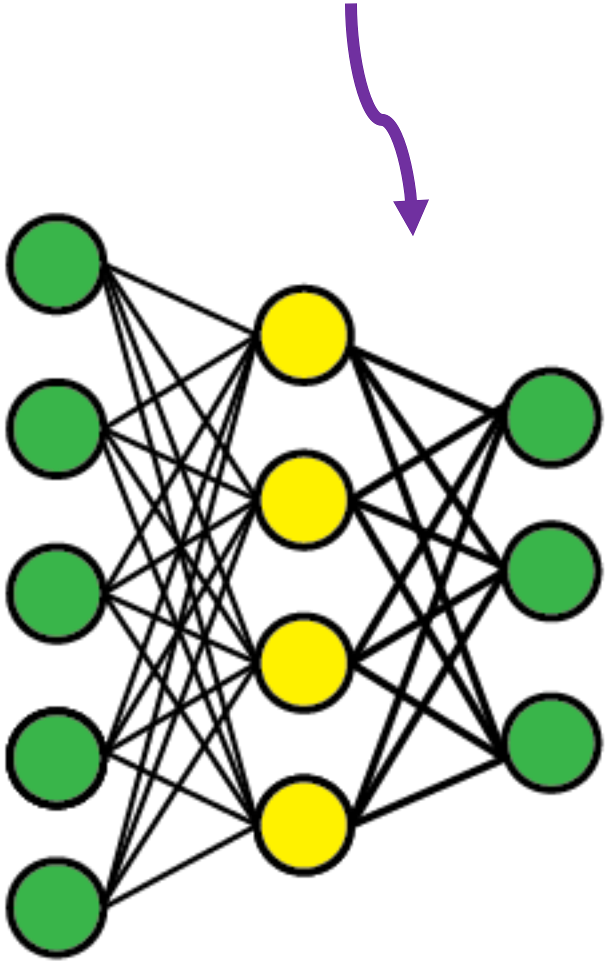
(a) Standard Neural Net (b) After applying dropout.

Dropout: Basic Idea

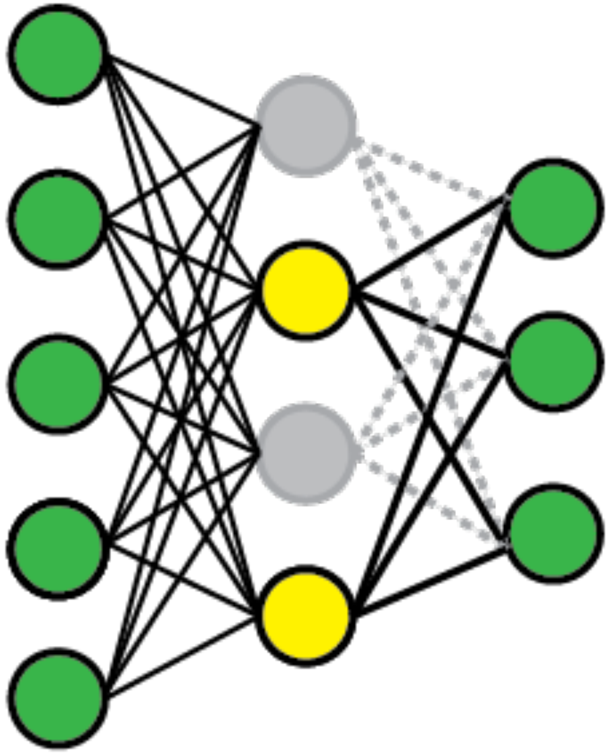
- Train
  - In each iteration of training (1 forward + 1 backward), randomly mask 50% (or an arbitrary percentage) of the neurons.
- Prediction
  - No dropout.
  - Use all the parameters.

Dropout: Implementation

Do dropout for this layer

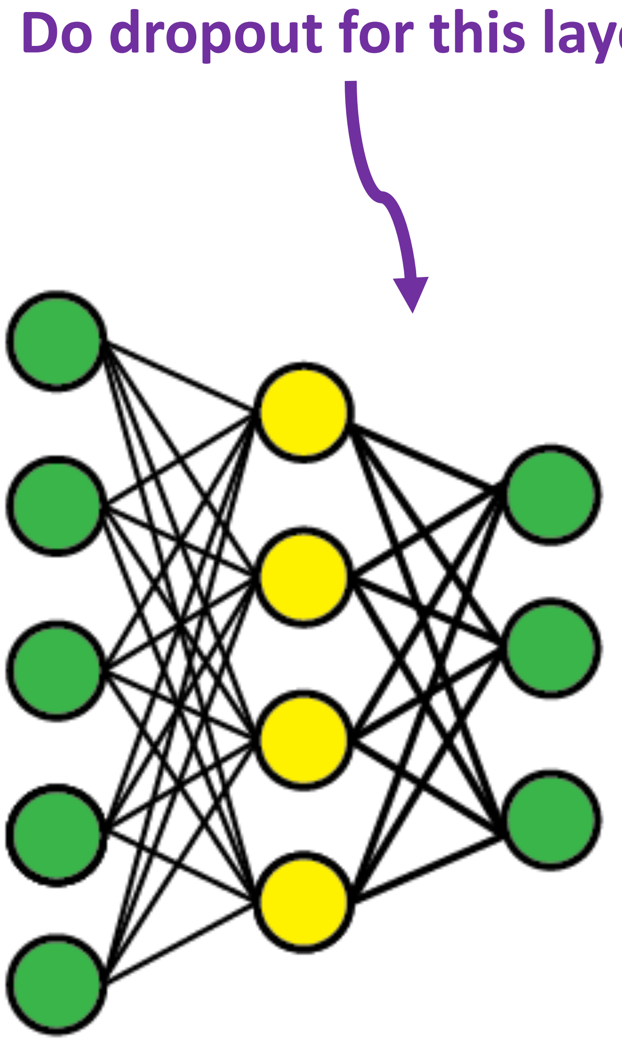


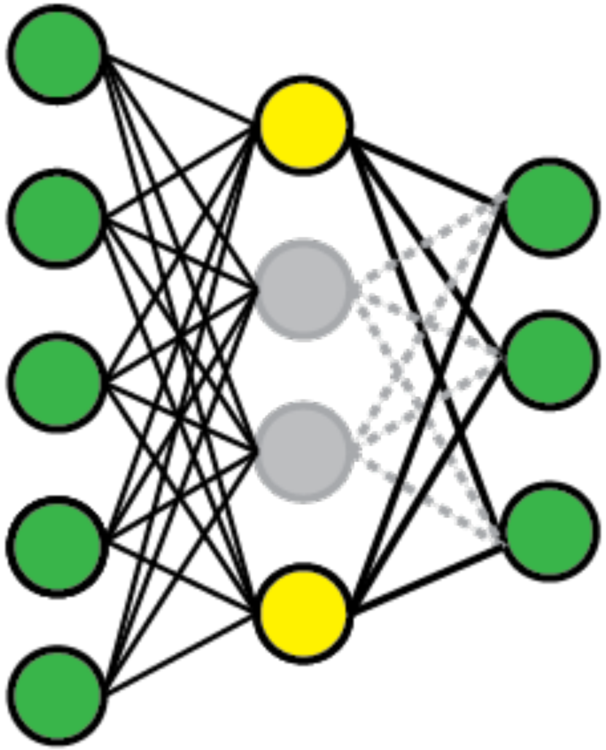
- **For a batch of training samples**
  - Randomly choose 50% of the neurons.
  - Set the selected neurons to zeros.
  - Multiply the unselected neurons by $\frac{1}{0.5} = 2$.



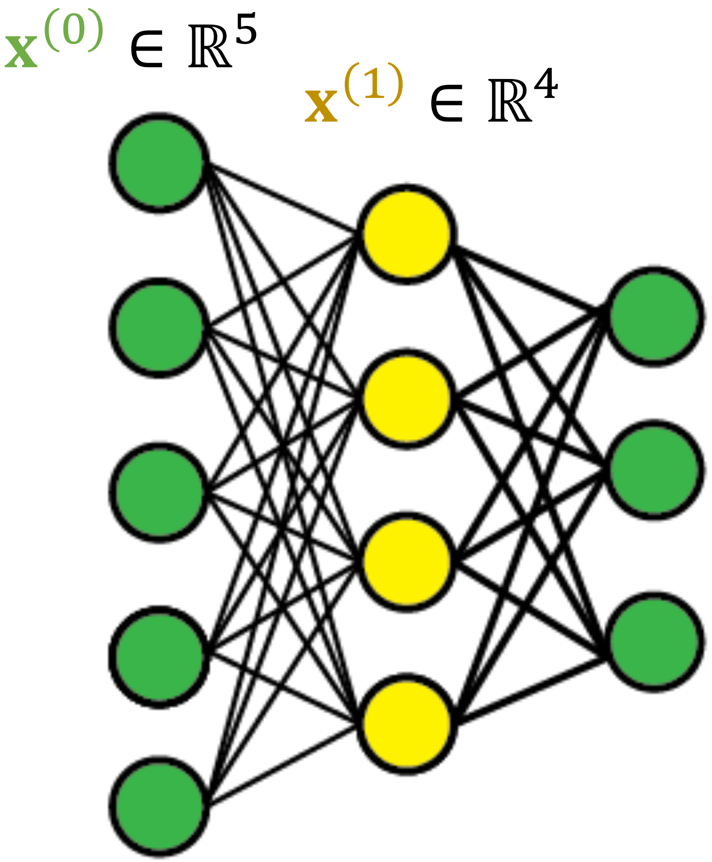
Dropout: Implementation

- **For a batch of training samples…**
- **For another batch**
  - Do an independent random sampling.





Dropout: Implementation

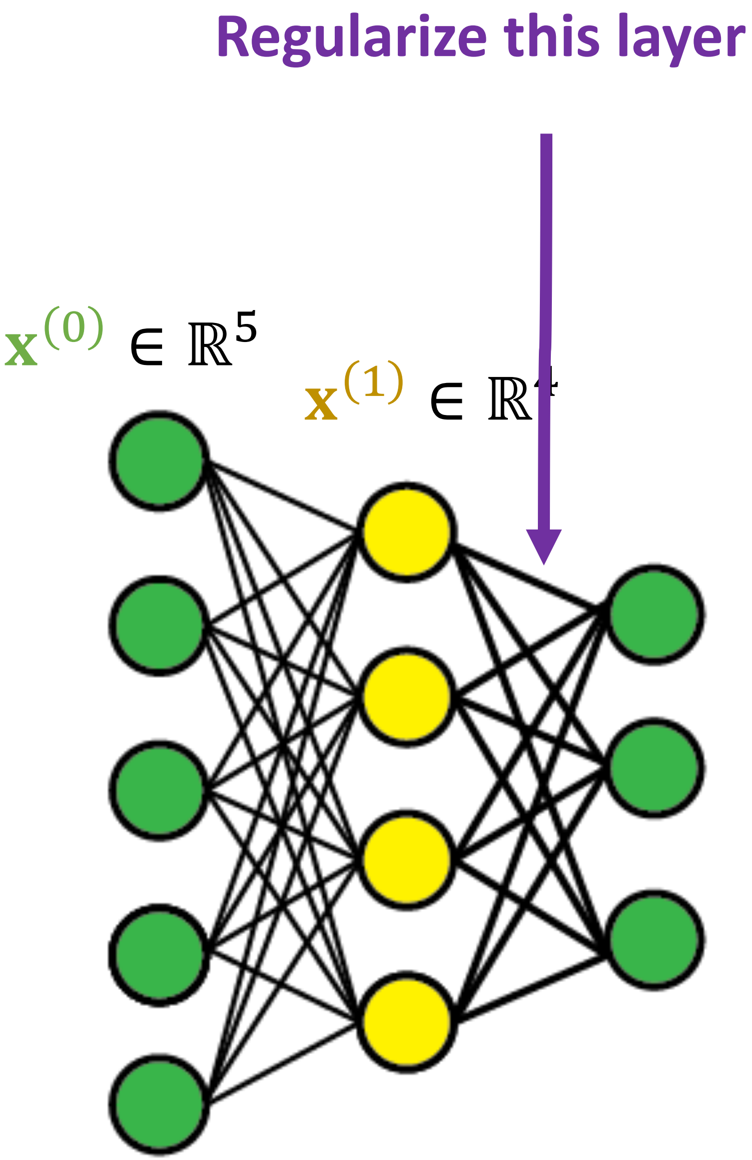


- **Input**: vector $\mathbf{x}^{(0)} \in \mathbb{R}^5$.

- $\mathbf{z}^{(1)} = \mathbf{W}^{(\mathbf{0})} \mathbf{x}^{(0)} \in \mathbb{R}^4$.
- $\mathbf{x}^{(1)} = \max\{\mathbf{0}, \mathbf{z}^{(1)}\} \in \mathbb{R}^4$.

- $\mathbf{z}^{(2)} = \mathbf{W}^{(1)} \mathbf{x}^{(1)} \in \mathbb{R}^3$.
- **Output**: $\mathbf{z}^{(2)} \in \mathbb{R}^3$.

Dropout: Implementation



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

- $\mathbf{m} \in \mathbb{R}^4$ is a random vector. (Each entry is 0 or 2, w.p. 50%.)
- Apply $\mathbf{m}$ to $\mathbf{x}^{(1)}$:

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

"$\circ$" is entrywise multiplication.

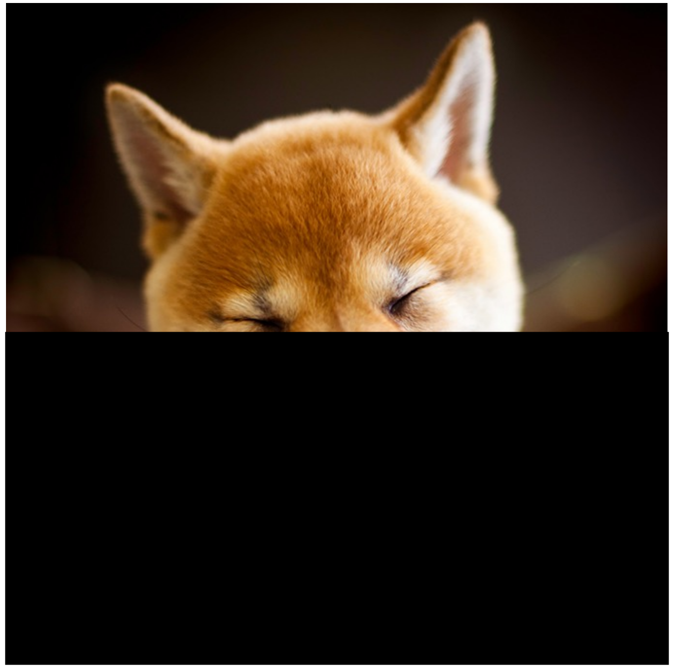
Keras's Dropout Layer

- Dropout only before the 1st dense layer to regularize the 1st dense layer.
- Because the 1st dense layer has too 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.



Why Does Dropout Work?

- In training, dropout forces the network to make decision based on part of the features.

- Dropout is a regularization [1].
  - Alleviate overfitting.
  - Like the L1 and L2 norm regularizations.
  - But dropout is empirically better.

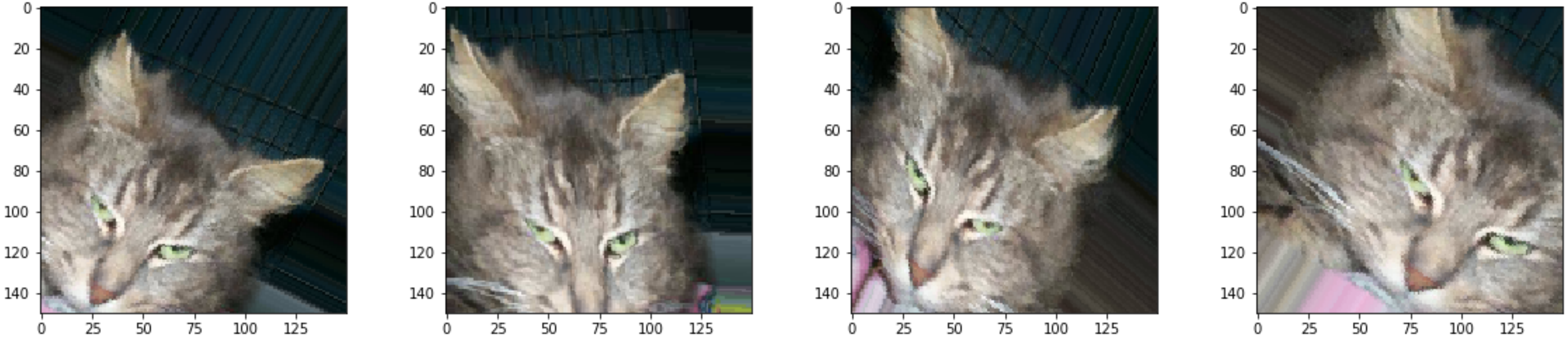
Reference:

1. Wager, Wang, & Liang. Dropout Training as Adaptive Regularization. In *NIPS*, 2013.

Trick 2: Data Augmentation

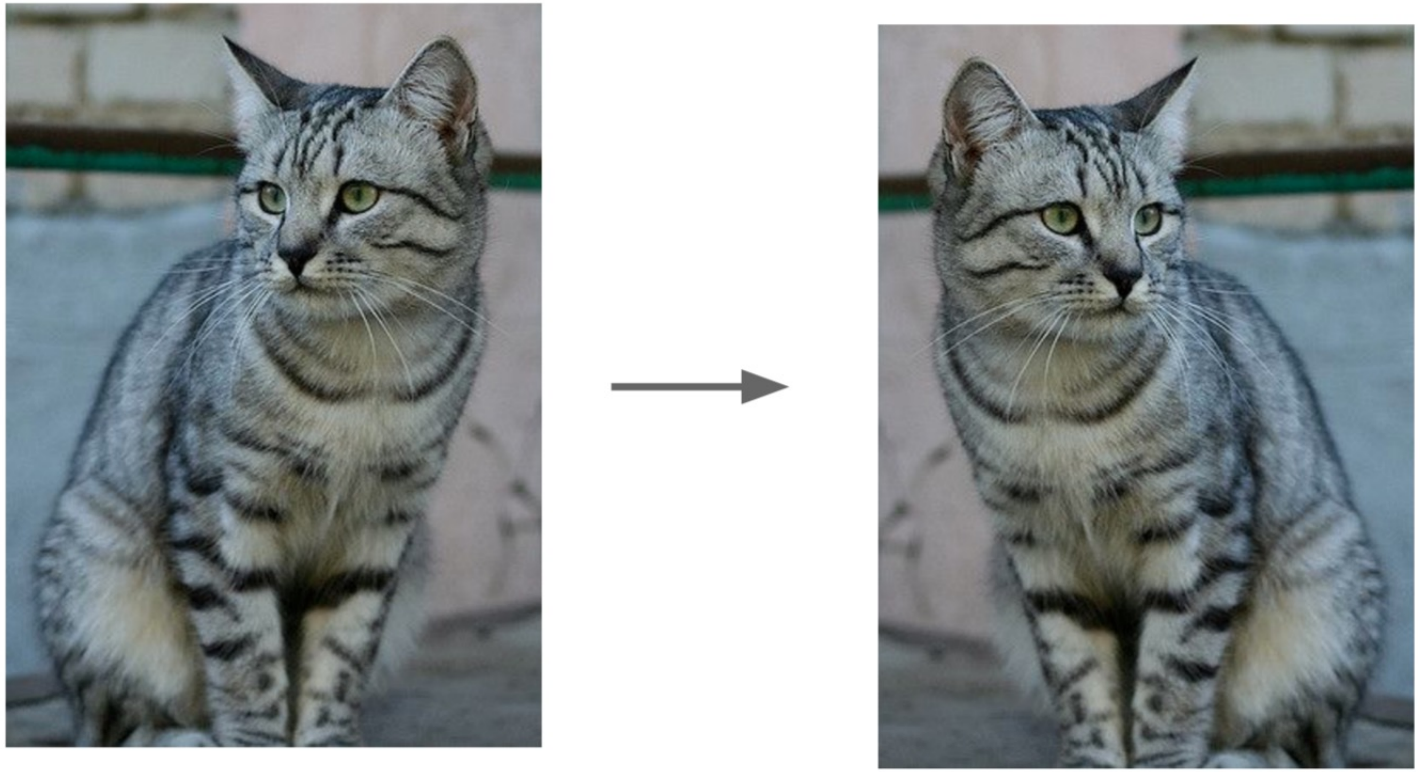
Data Augmentation

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



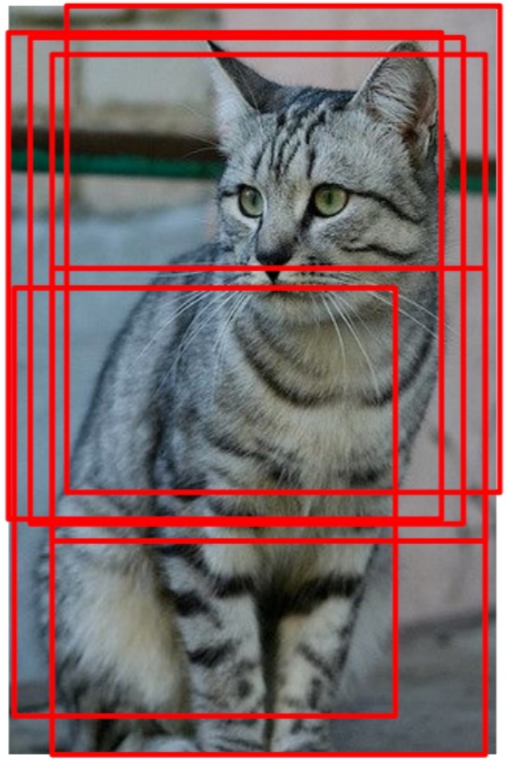
Data Augmentation: Examples

Horizontal Flips



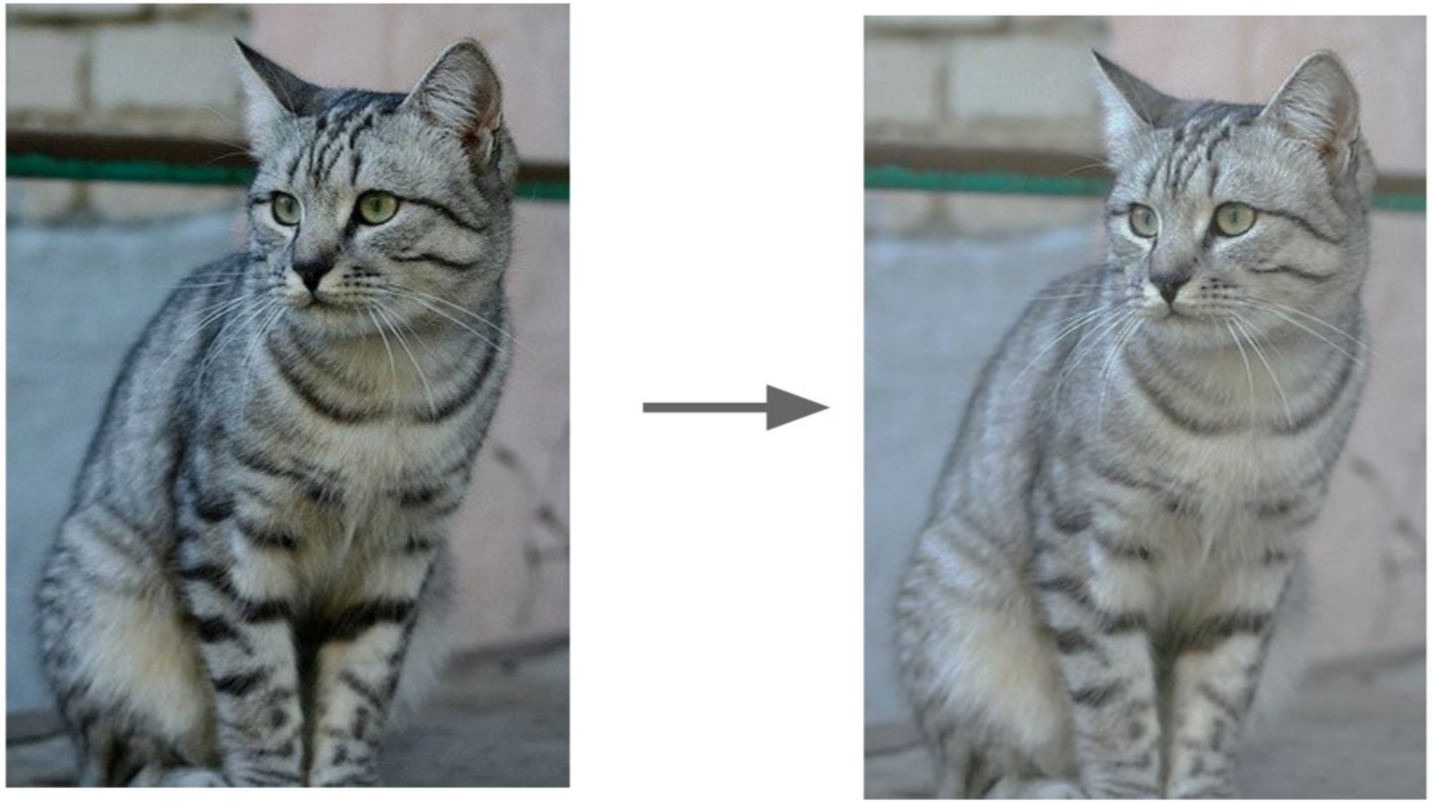
Data Augmentation: Examples

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')
```

Setup Data Augmentation Using Keras

During training, Keras performs the followings:

1. Load a batch of $b = 20$ samples to memory.
2. Process the samples:
   - Reshaping (aligning the images as $150 \times 150 \times 3$ tensors).
   - Rescaling (scaling the pixels from [0, 255] to [0, 1].)
   - Data augmentation.
   - ➔ The result is a $20 \times 150 \times 150 \times 3$ tensor.
3. Perform backpropagation using the tensor as input; update the model parameters.
4. Free the tensor (i.e., the batch of samples) from memory.

Train the CNN

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

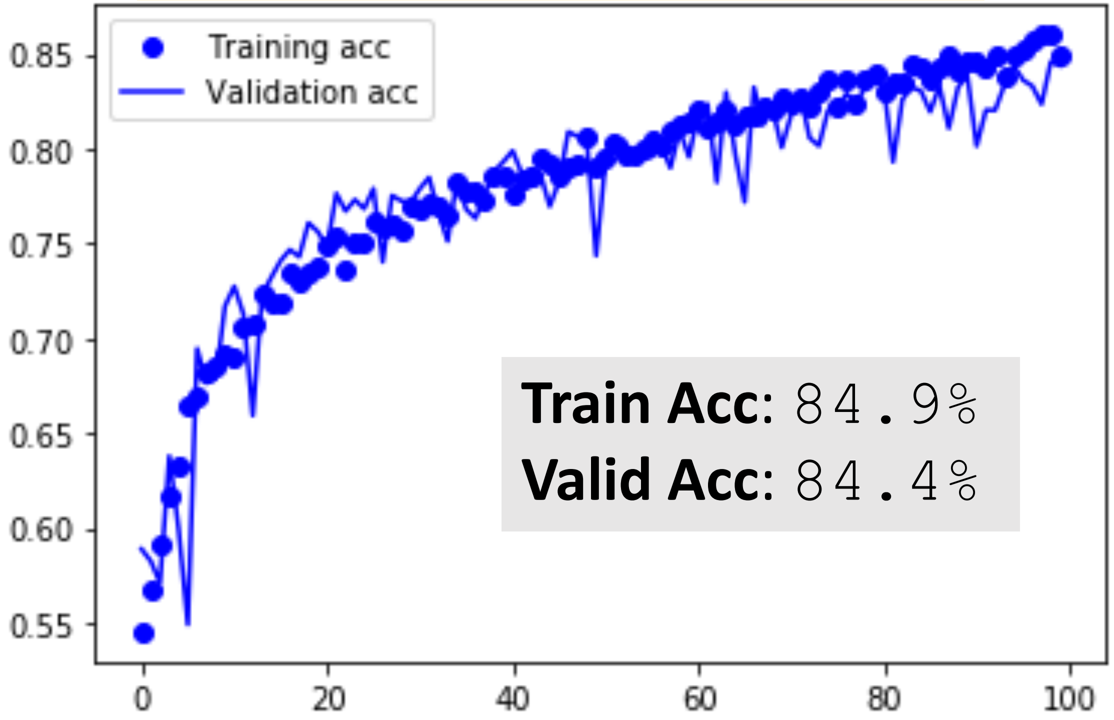
```
Epoch 1/100
100/100 [==============================] - 24s - loss: 0.6857 - acc: 0.5447 - val_loss: 0.6620 - val_acc: 0.5888
Epoch 2/100
100/100 [==============================] - 23s - loss: 0.6710 - acc: 0.5675 - val_loss: 0.6606 - val_acc: 0.5825
Epoch 3/100
100/100 [==============================] - 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
```

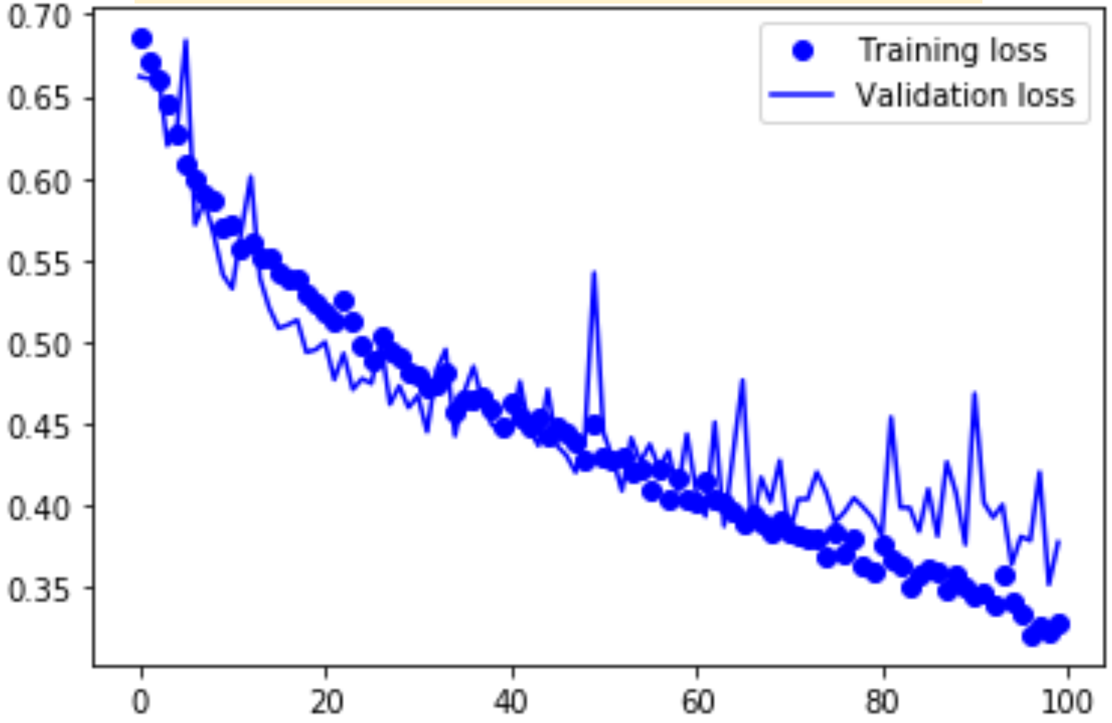
Examine the Results

accuracy against ***epochs***



Train Acc: `84.9%`
Valid Acc: `84.4%`

loss against ***epochs***



Take-Home Message

- To train CNN for images, always use data augmentation.
  - It gives you more data for free!
  - Use data augmentation for only training samples.
  - Don’t use it for validation and test samples!!!

Take-Home Message

- To train CNN for images, always use data augmentation.
  - It gives you more data for free!
  - Use data augmentation for only training samples.
  - Don’t use it for validation and test samples!!!

- If a layer has too many parameters, place a dropout layer before it.
  - Dropout randomly masks the inputs.
  - Dropout is regularization.
  - Regularization alleviates overfitting.
  - The training becomes a little slower.

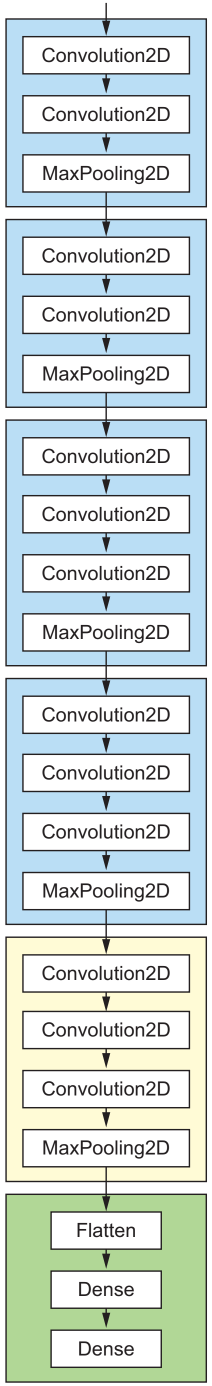
Trick 3: Pretrain

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.

Train a Deep Neural Network?



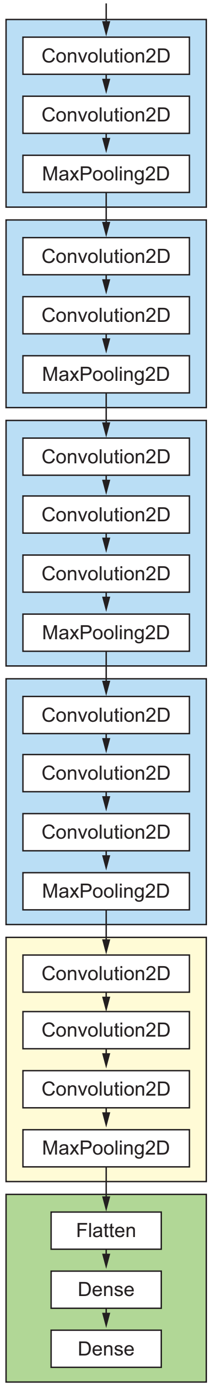
the VGG16 network

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

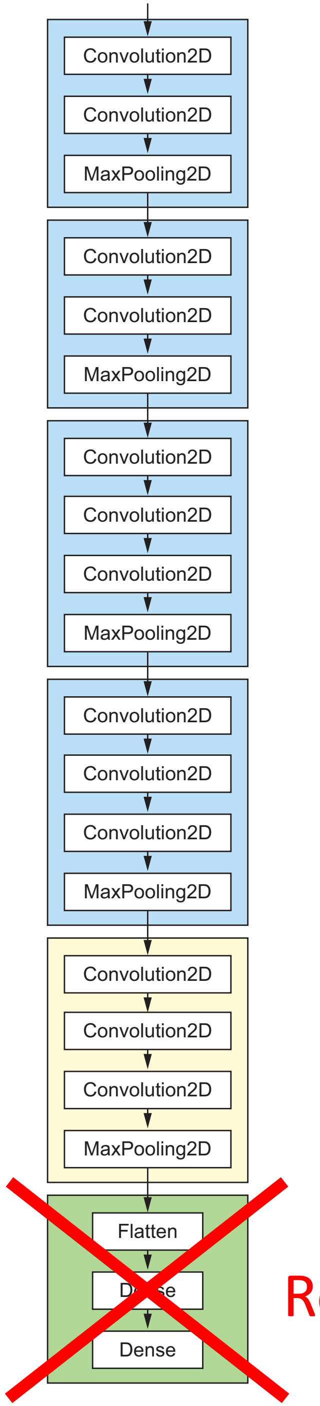
Pretrain



the VGG16 network

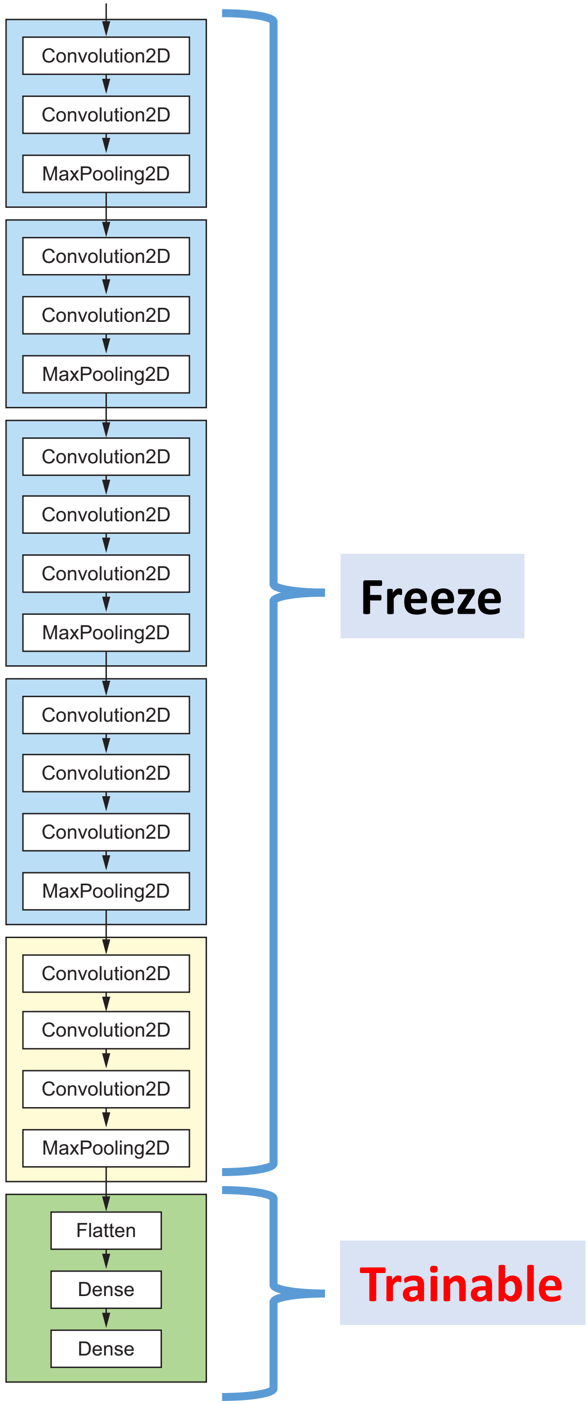
1. Pre-train a deep net on large-scale dataset, e.g., ImageNet ($14M$ images with labels).

Pretrain



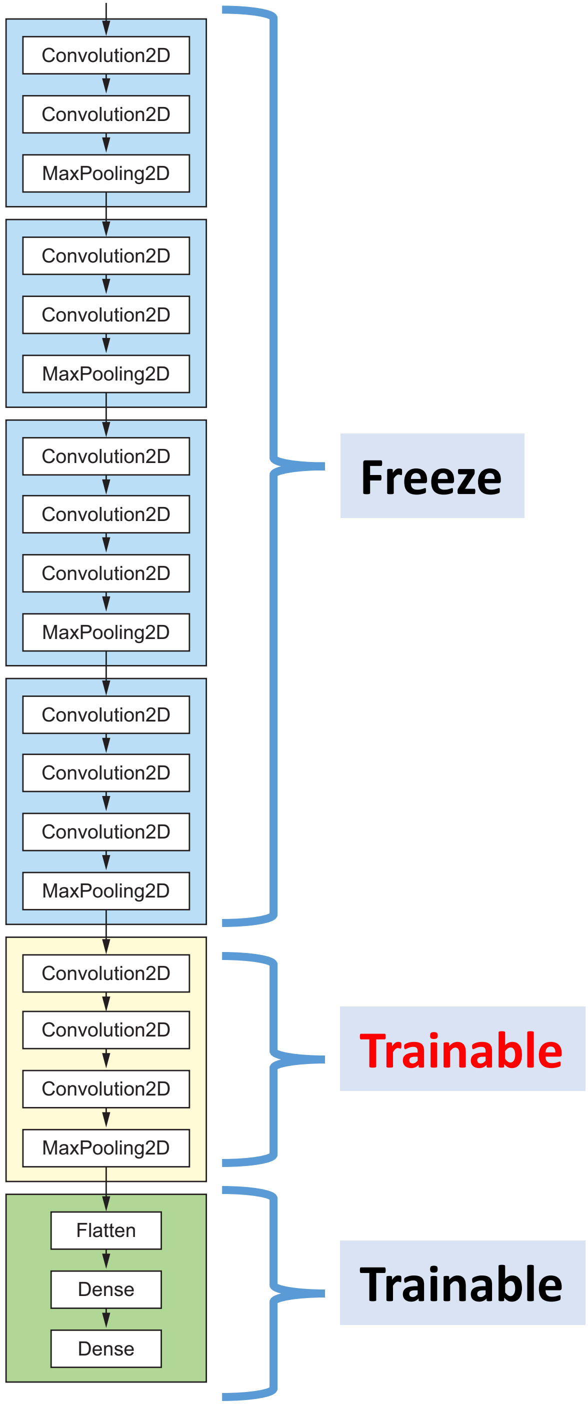
Remove the top layers

1. Pre-train a deep net on large-scale dataset, e.g., ImageNet ($14M$ images with labels).
2. Remove the top layers. *Why?*
   - Different output shapes and activation functions.
   - New classifier specialized for the new task.



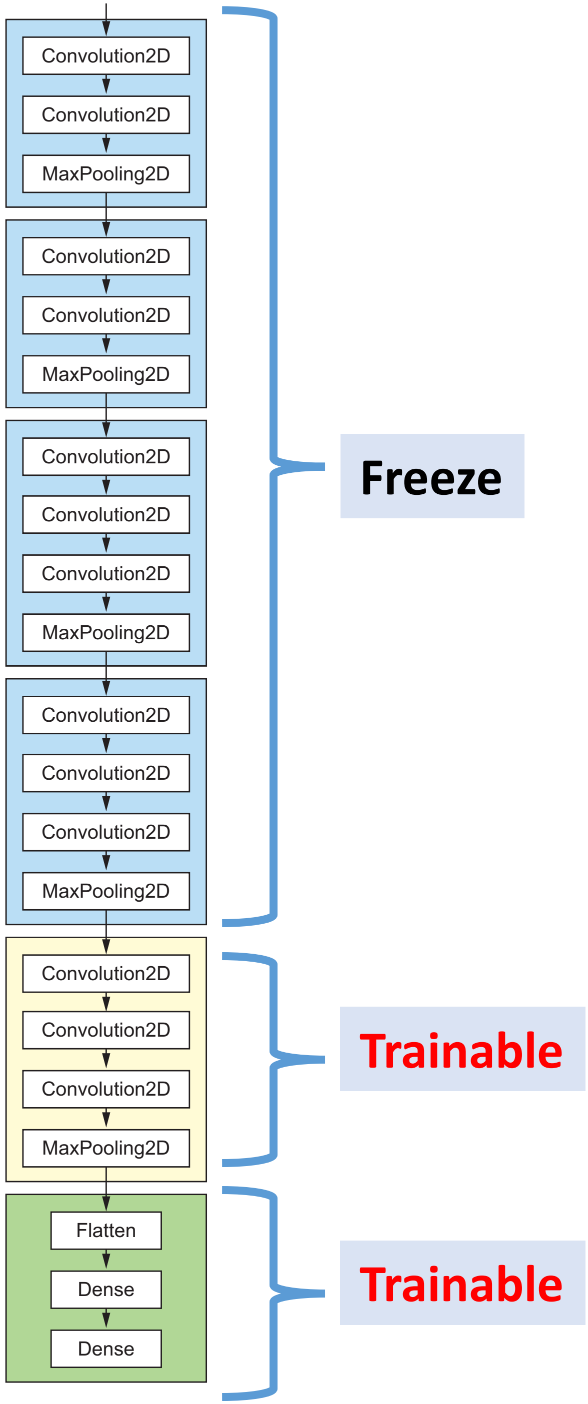
Pretrain

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



Pretrain

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

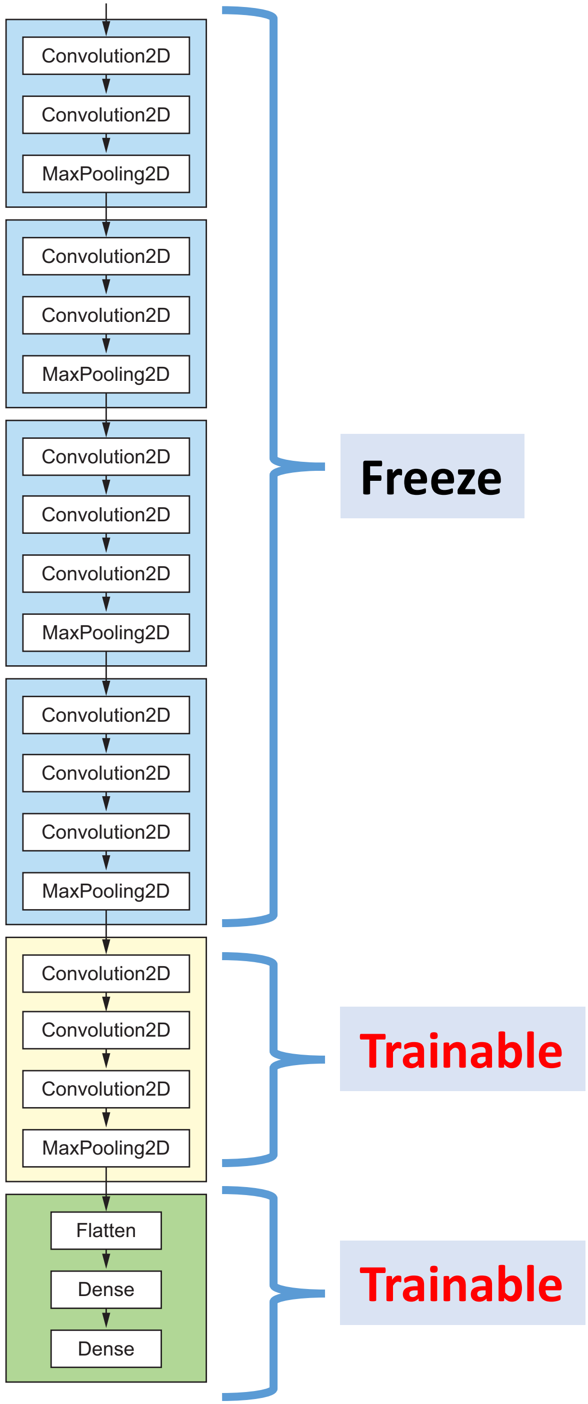


Pretrain

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

Question: Can Steps 4 & 5 be merged?

Pretrain



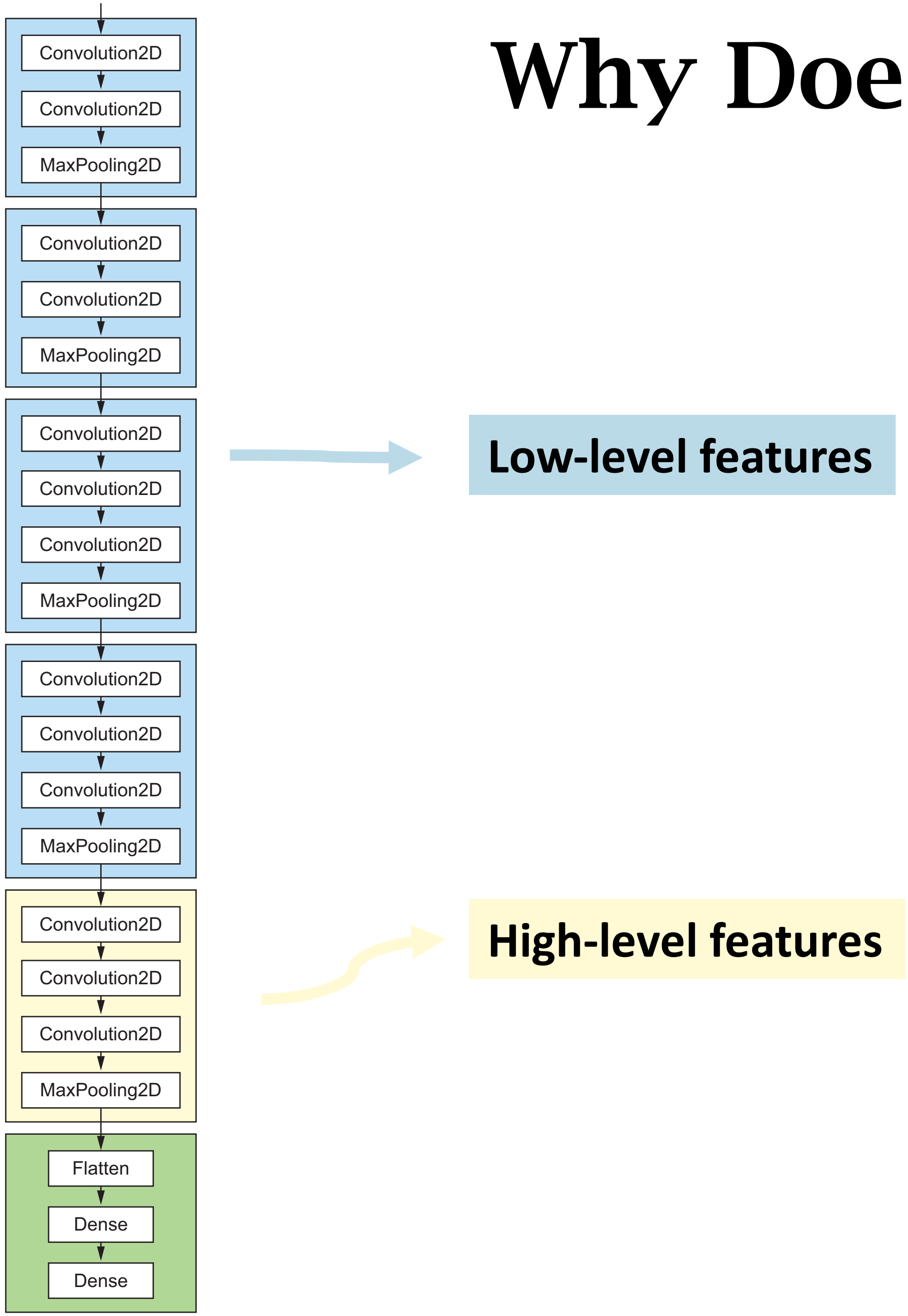
No. Merging Steps 4 & 5 is a bad idea.

- 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.

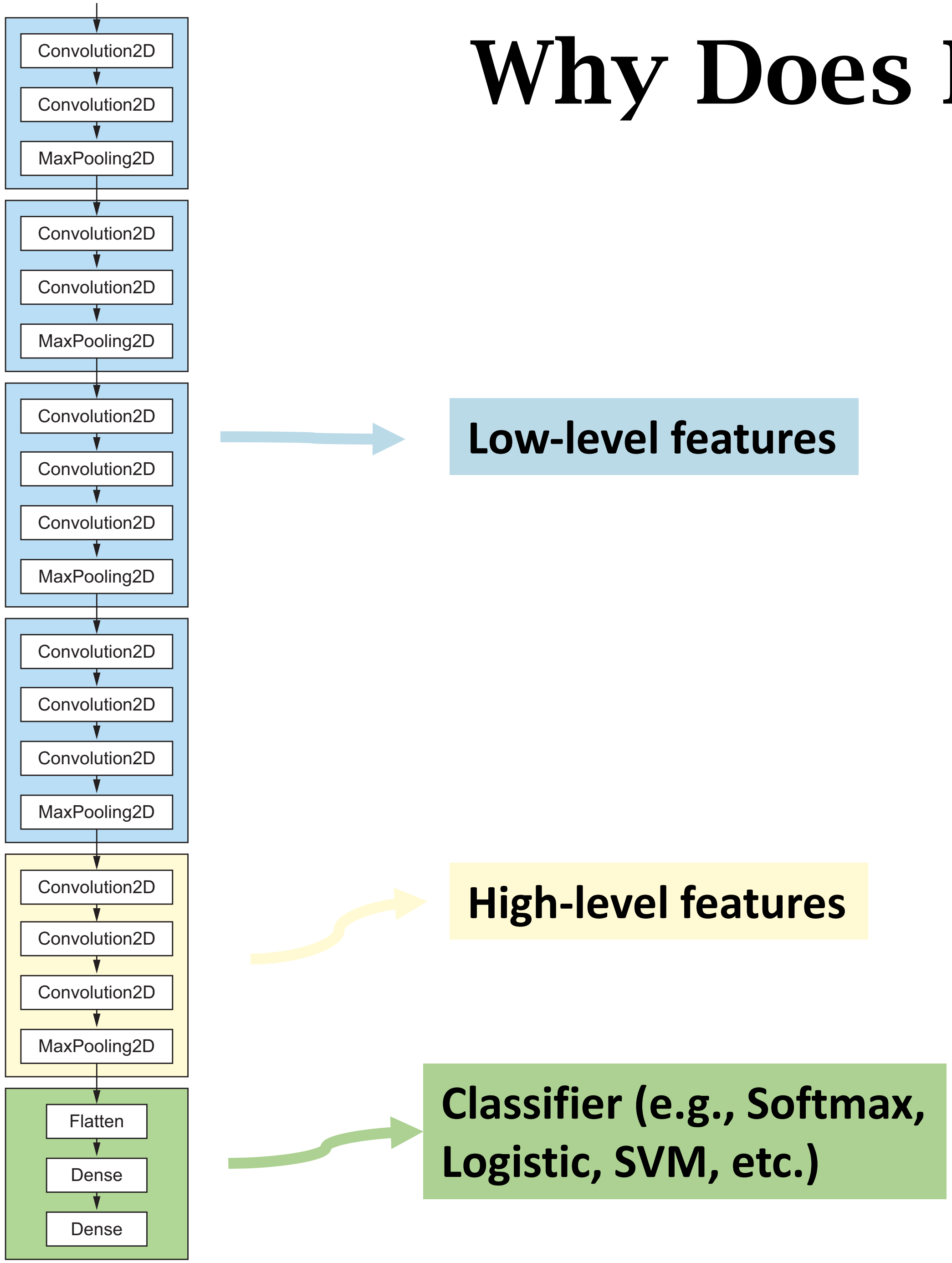
Question: Can Steps 4 & 5 be merged?

Why Does Pretraining Work?



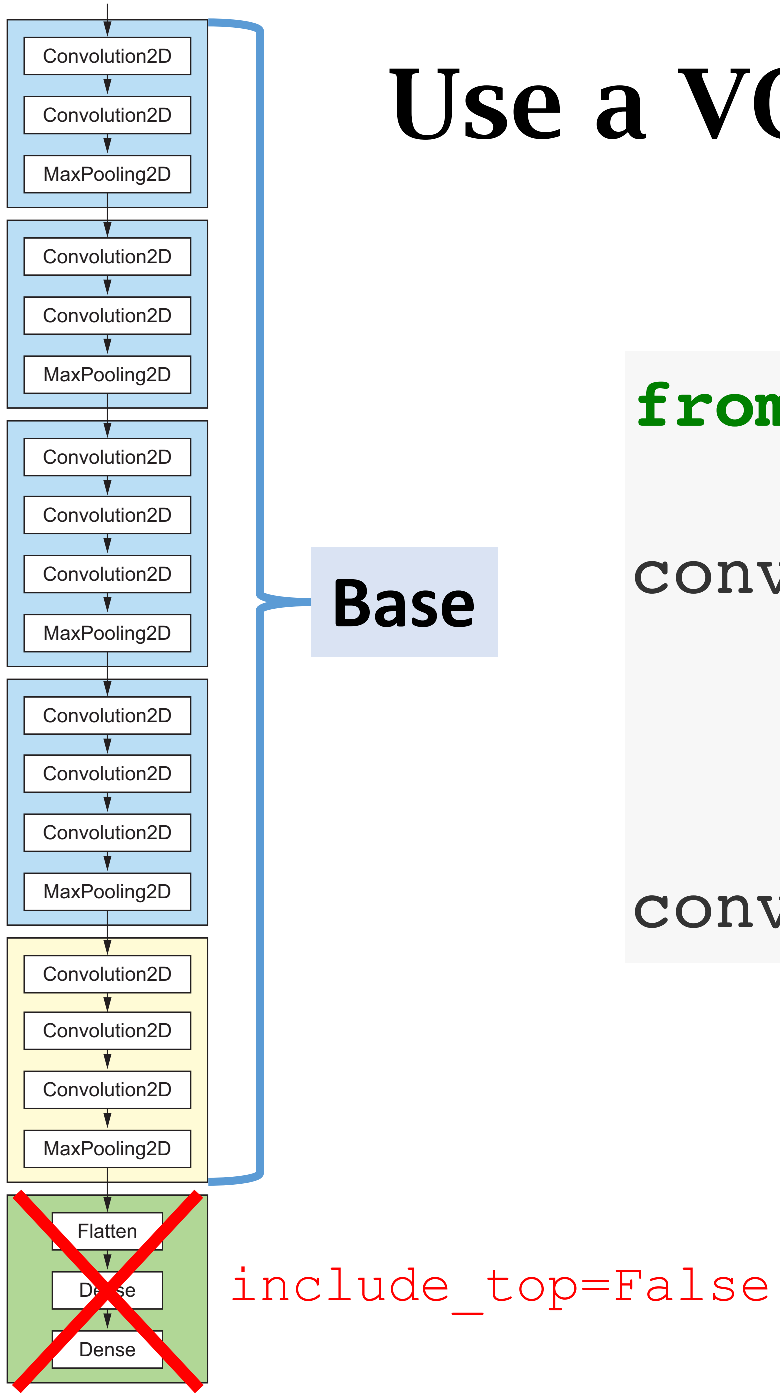
- The ***Conv Layers*** are for *feature extraction*.
  - The **low-level feature** (edges, shapes, patterns, etc.) learned from ImageNet are effective to other image problems.
  - The **high-level features** learned from ImageNet are useful, but less effective.

Why Does Pretraining Work?



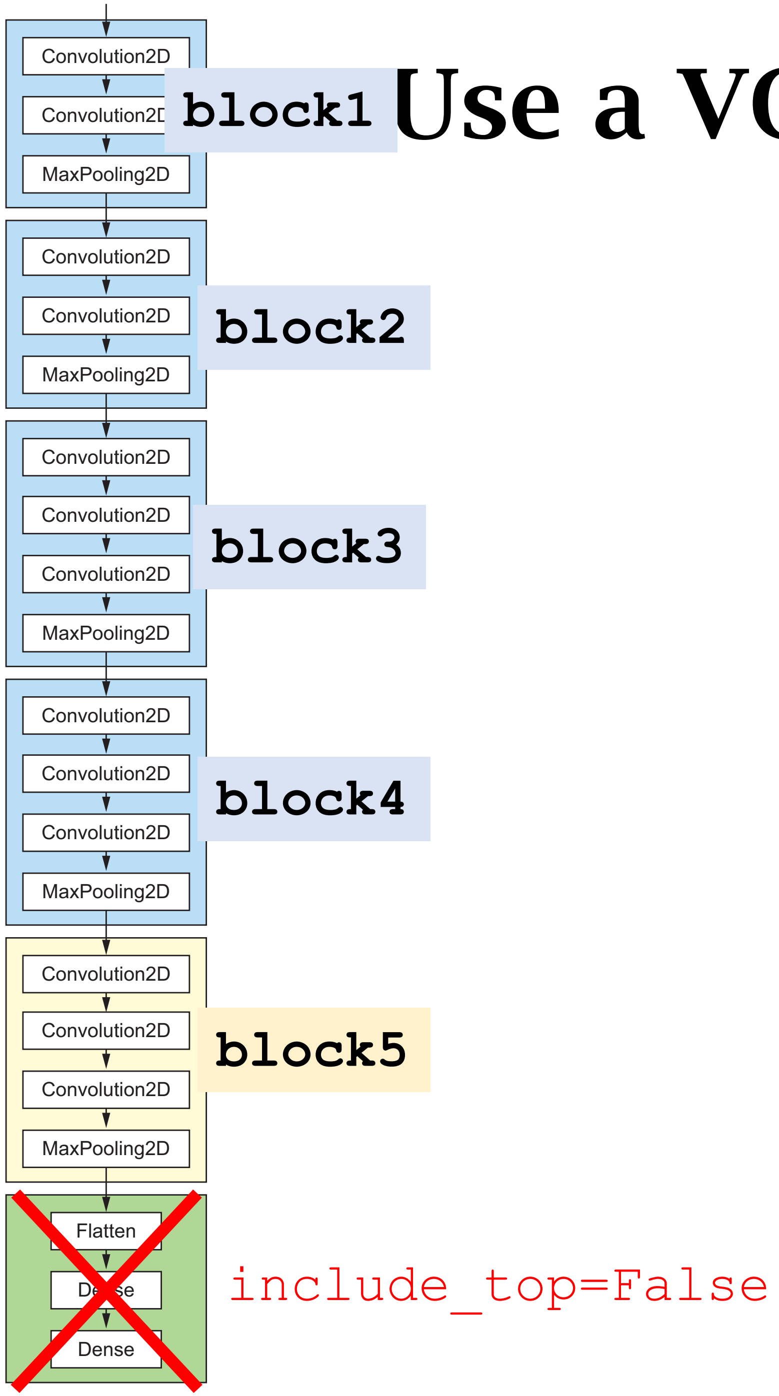
- The ***Conv Layers*** are for *feature extraction*.
  - The **low-level feature** (edges, shapes, patterns, etc.) learned from ImageNet are effective to other image problems.
  - The **high-level features** learned from ImageNet are useful, but less effective.
- Think of the ***FC layers*** as a classifier 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()
```

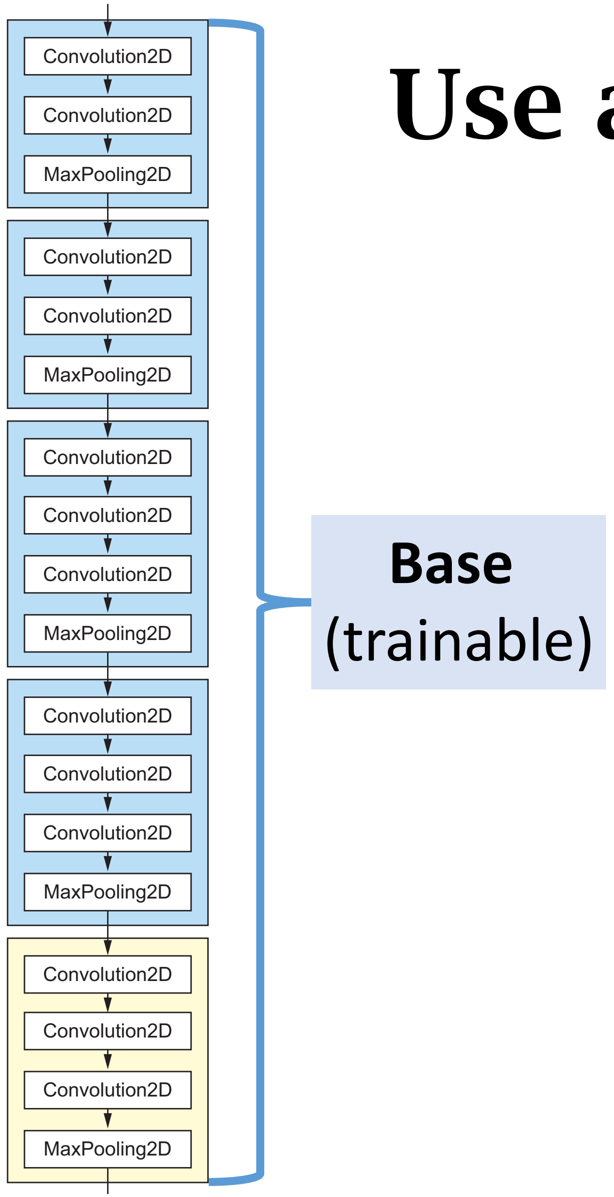
Use a VGG16 Net Pretrained on ImageNet



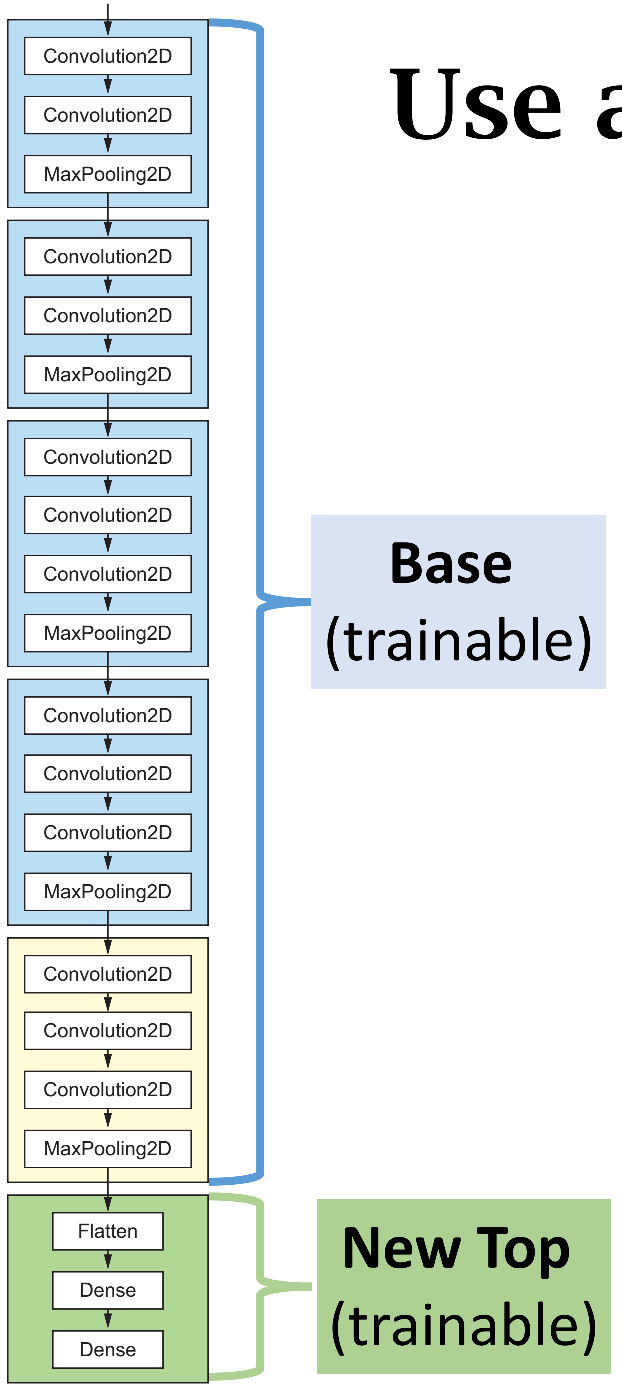
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 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



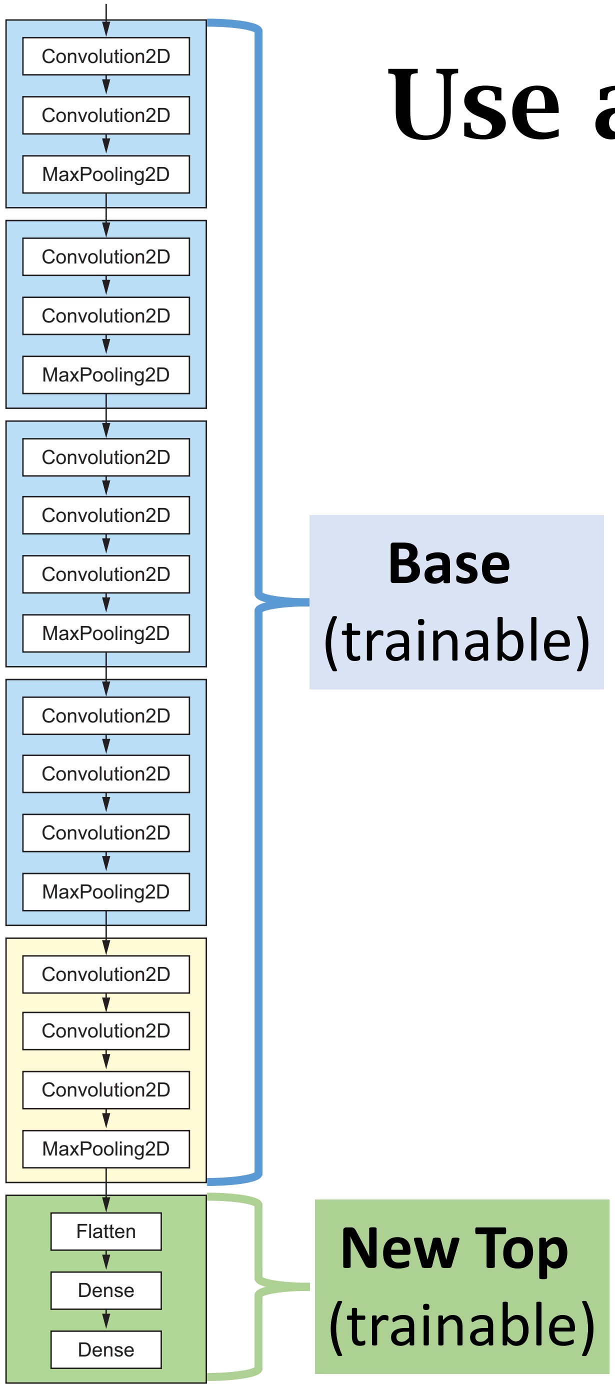

```
from keras import models
from keras import layers

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



Use a VGG16 Net Pretrained on ImageNet

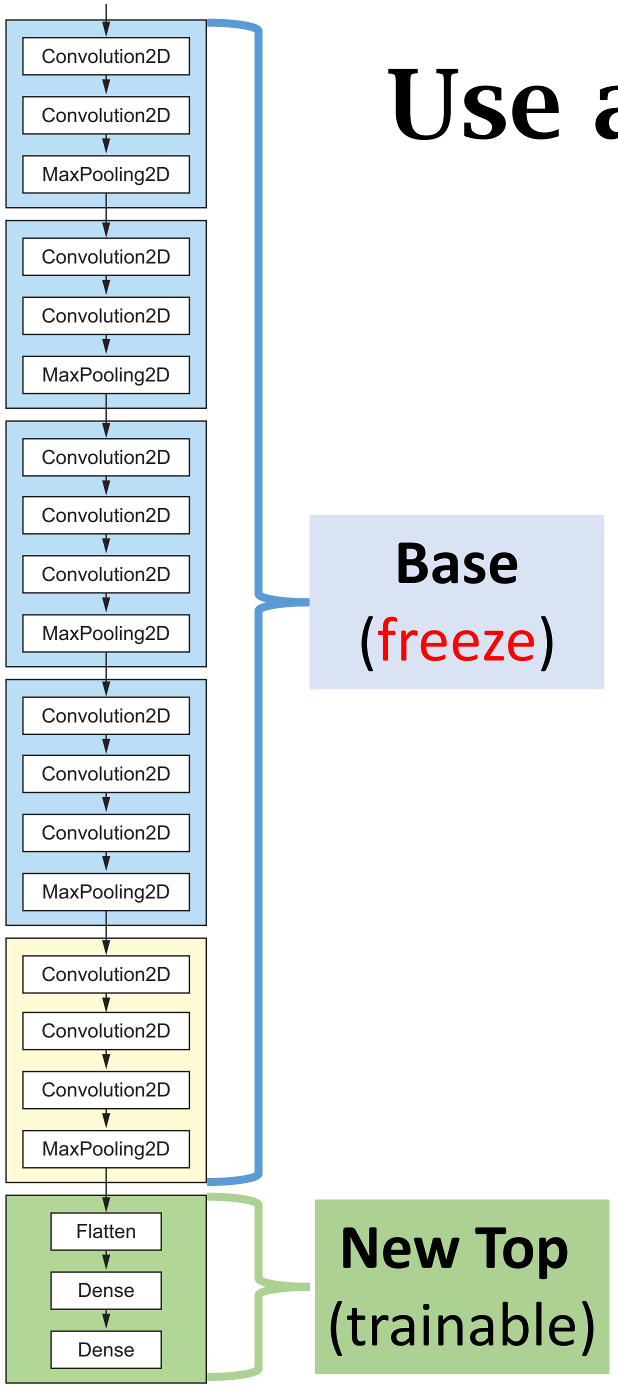
```
from keras import models
from keras import layers

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




```
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

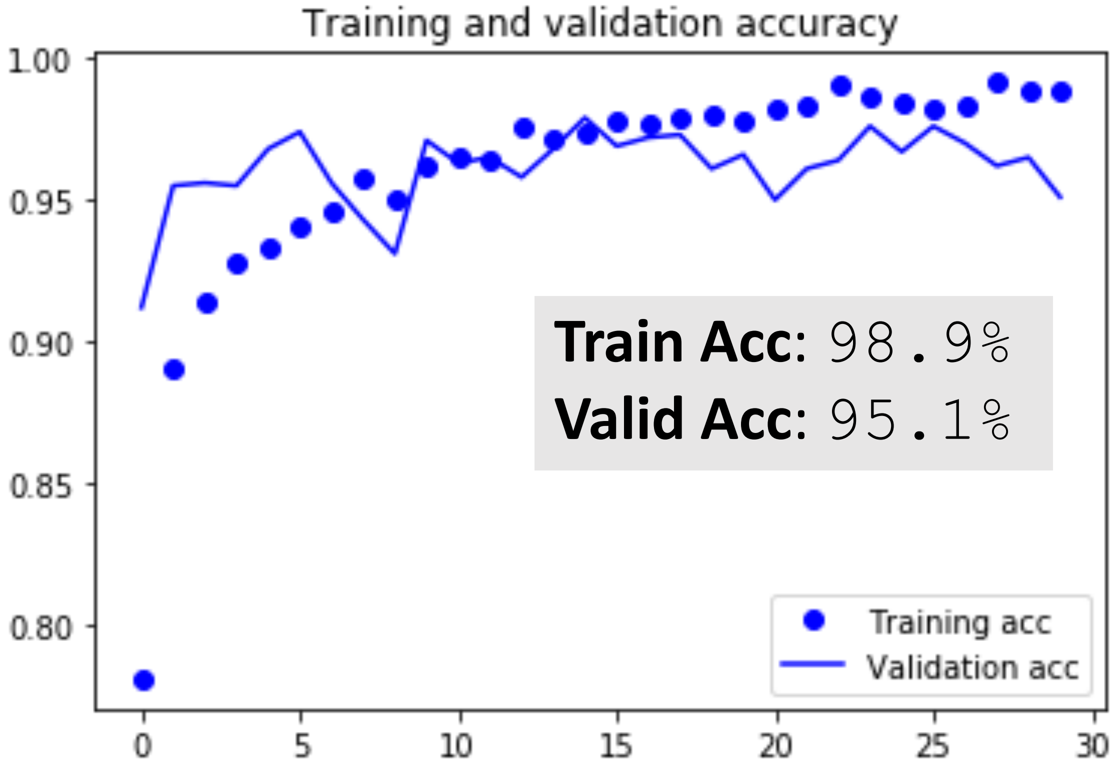



```
conv_base.trainable = False
model.summary()
```

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

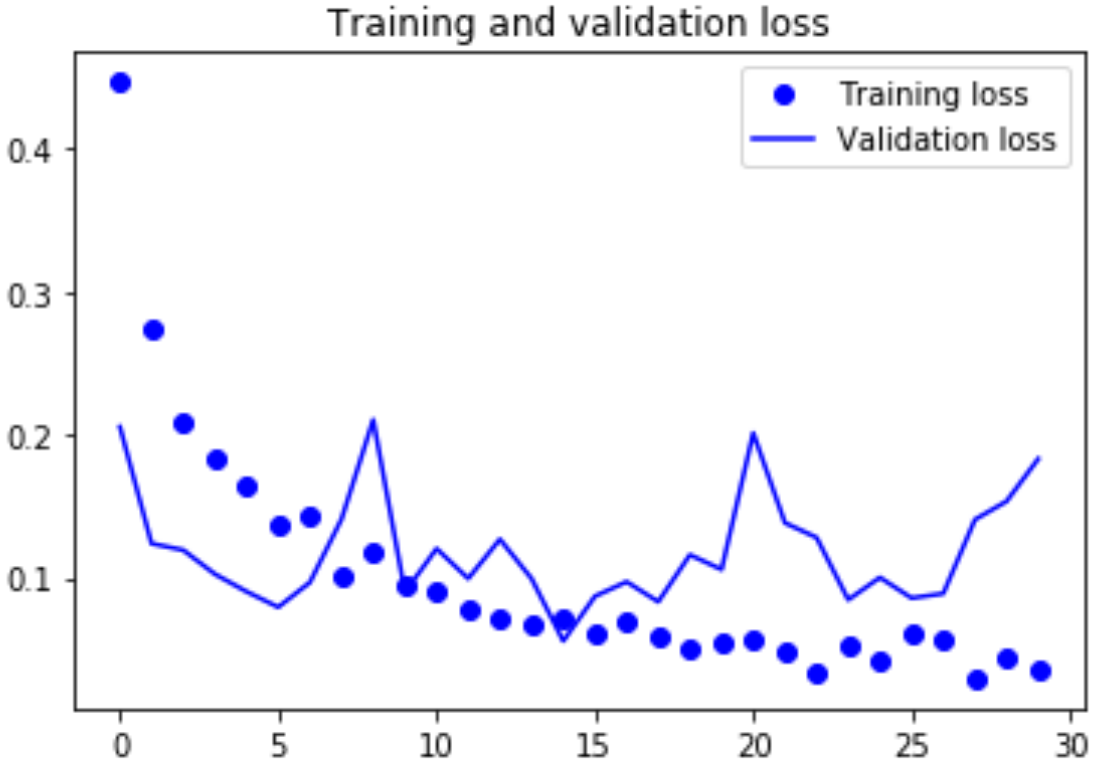
After Training the New Top

accuracy against ***epochs***

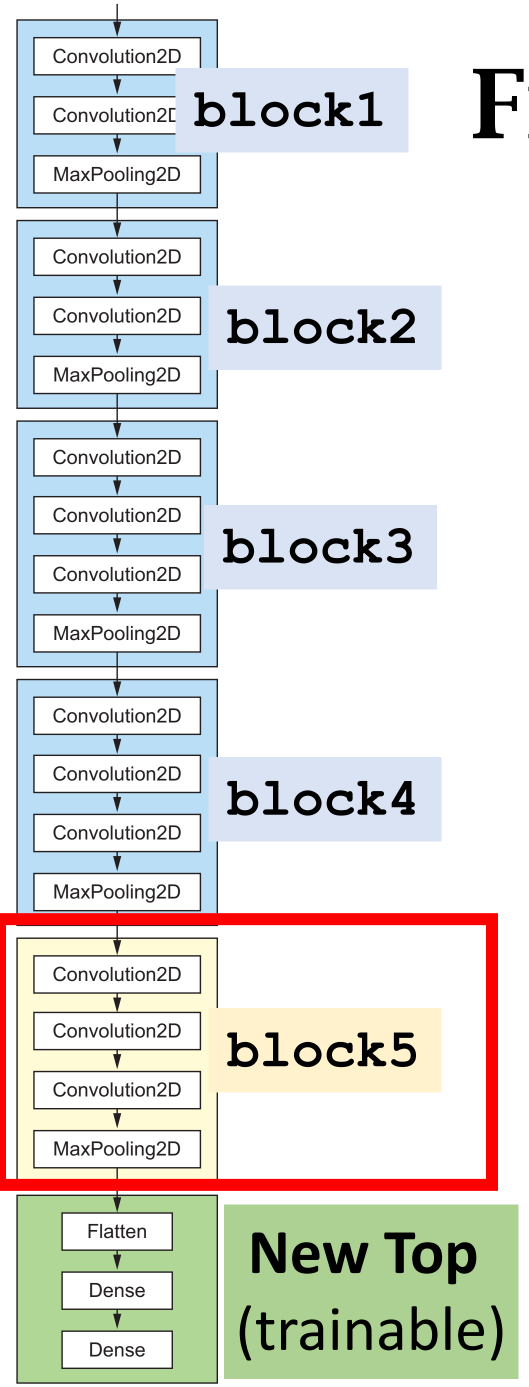


Train Acc: `98.9%`
Valid Acc: `95.1%`

loss against ***epochs***

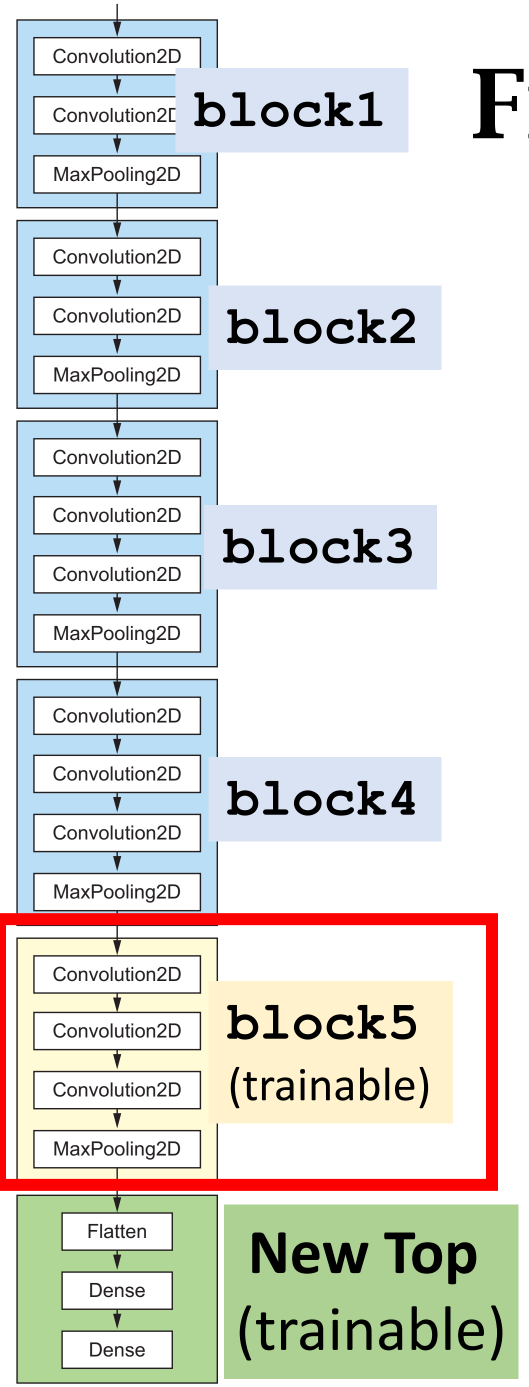


Fine Tuning the Top Conv Layers



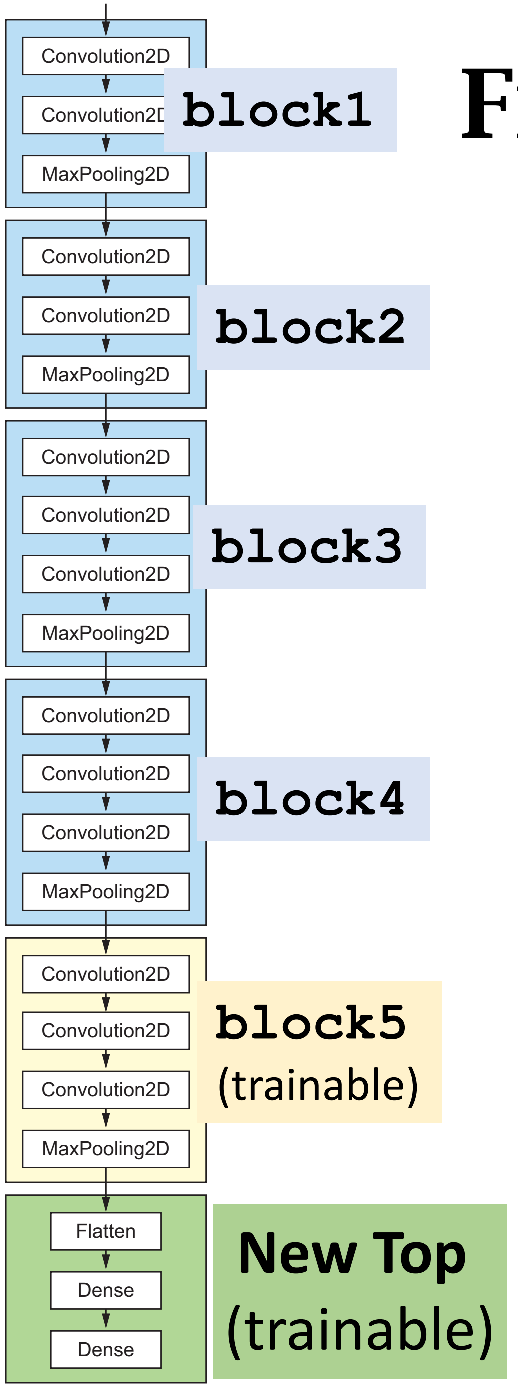

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 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
```

Fine Tuning the Top Conv Layers




```
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)
```

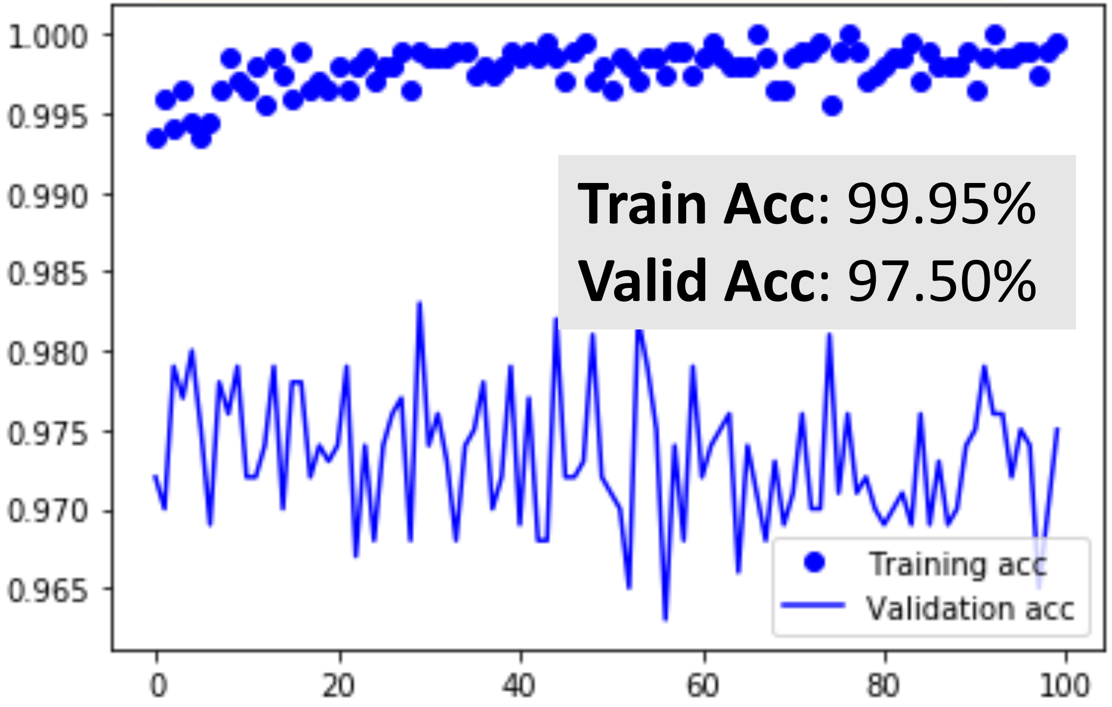
```
Epoch 1/100
100/100 [==============================] - 32s - loss: 0.0215 - acc: 0.9935 - val_loss: 0.0980 - val_acc: 0.9720
Epoch 2/100
100/100 [==============================] - 32s - loss: 0.0131 - acc: 0.9960 - val_loss: 0.1247 - val_acc: 0.9700
Epoch 3/100
100/100 [==============================]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

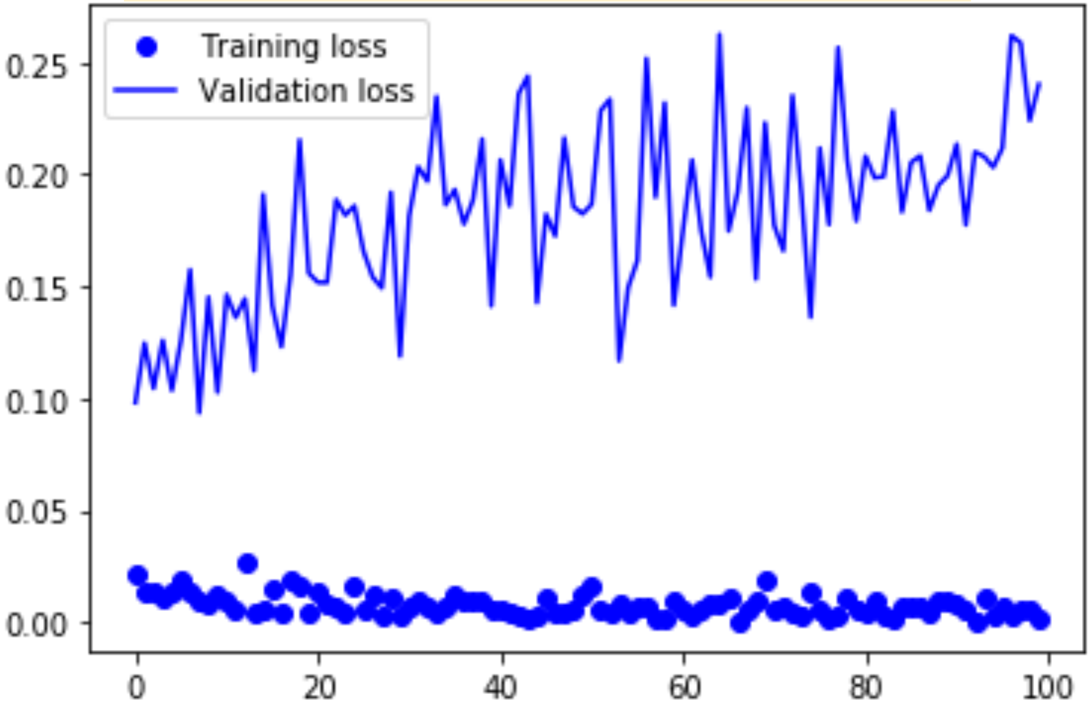
Fine Tuning the Top Conv Layers

accuracy against ***epochs***



Train Acc: 99.95%
Valid Acc: 97.50%

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 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%`

Ensemble Methods

Ensemble Methods

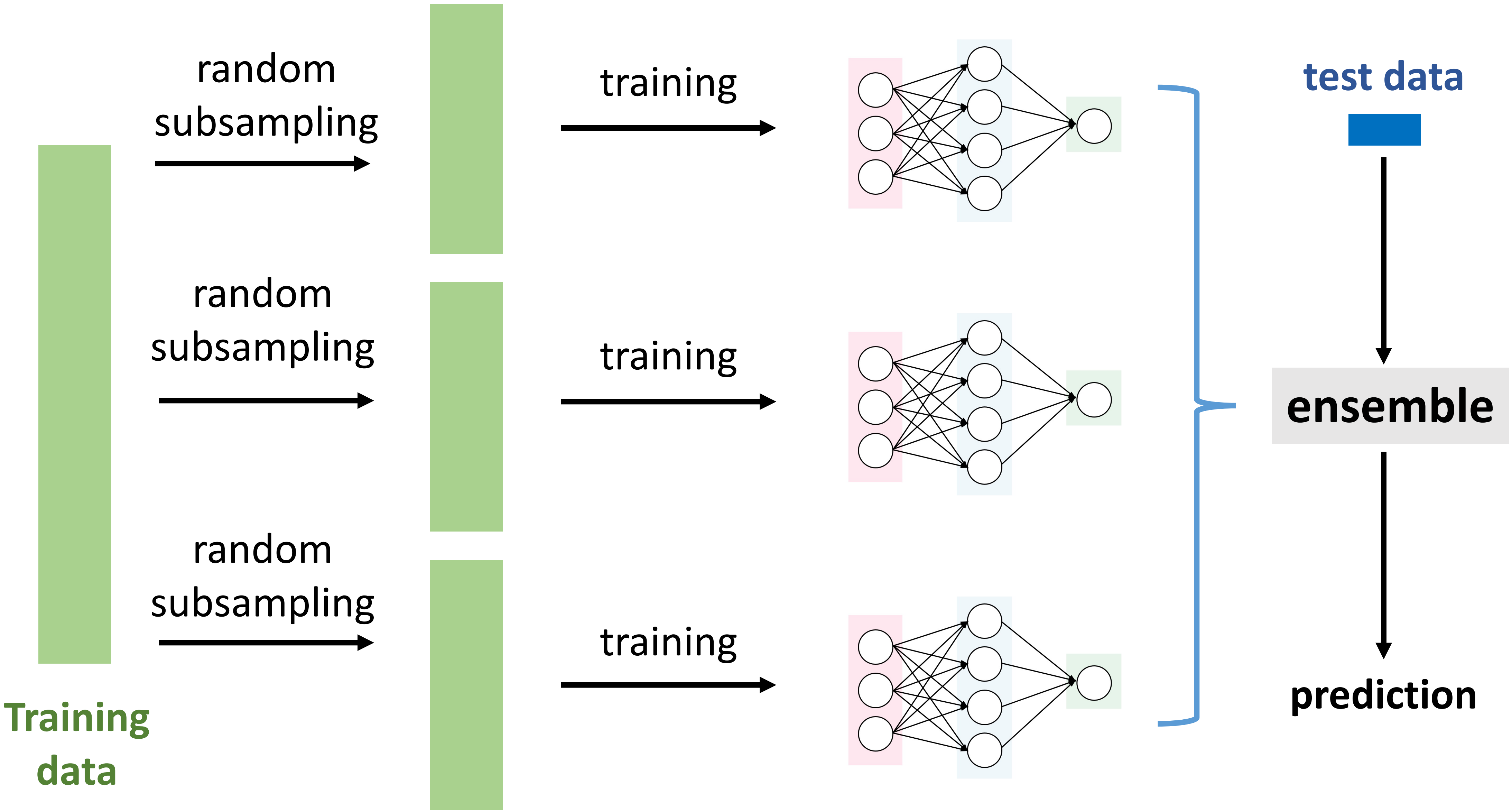
- Varying data (bagging).
  - Fit a VGG16 network on a subset of data ➔ Model 1
  - Fit a VGG16 network on a subset of data ➔ Model 2
  - Fit a VGG16 network on a subset of data ➔ Model 3

Ensemble Methods

- Varying data (bagging).
  - Fit a VGG16 network on a subset of data ➔ Model 1 ➔ pred 1
  - Fit a VGG16 network on a subset of data ➔ Model 2 ➔ pred 2
  - Fit a VGG16 network on a subset of data ➔ Model 3 ➔ pred 3

Vote

Bagging (aka Bootstrap Aggregating)



Ensemble Methods

- Varying data (bagging).
  - Fit a VGG16 network on a subset of data → Model 1 → pred 1
  - Fit a VGG16 network on a subset of data → Model 2 → pred 2
  - Fit a VGG16 network on a subset of data → Model 3 → pred 3

Vote

- Varying models.
  - Different network structures.
  - Different random initializations.
  - Different optimization algorithms.

Why Ensemble?

- Deep neural networks are unstable

Sensitive to hyper-parameters

Why Ensemble?

- Deep neural networks are unstable and random.

Sensitive to hyper-parameters

Random initialization

Stochastic gradient

different local optima

Why Ensemble?

- Deep neural networks are unstable and random.

Sensitive to hyper-parameters

Random initialization

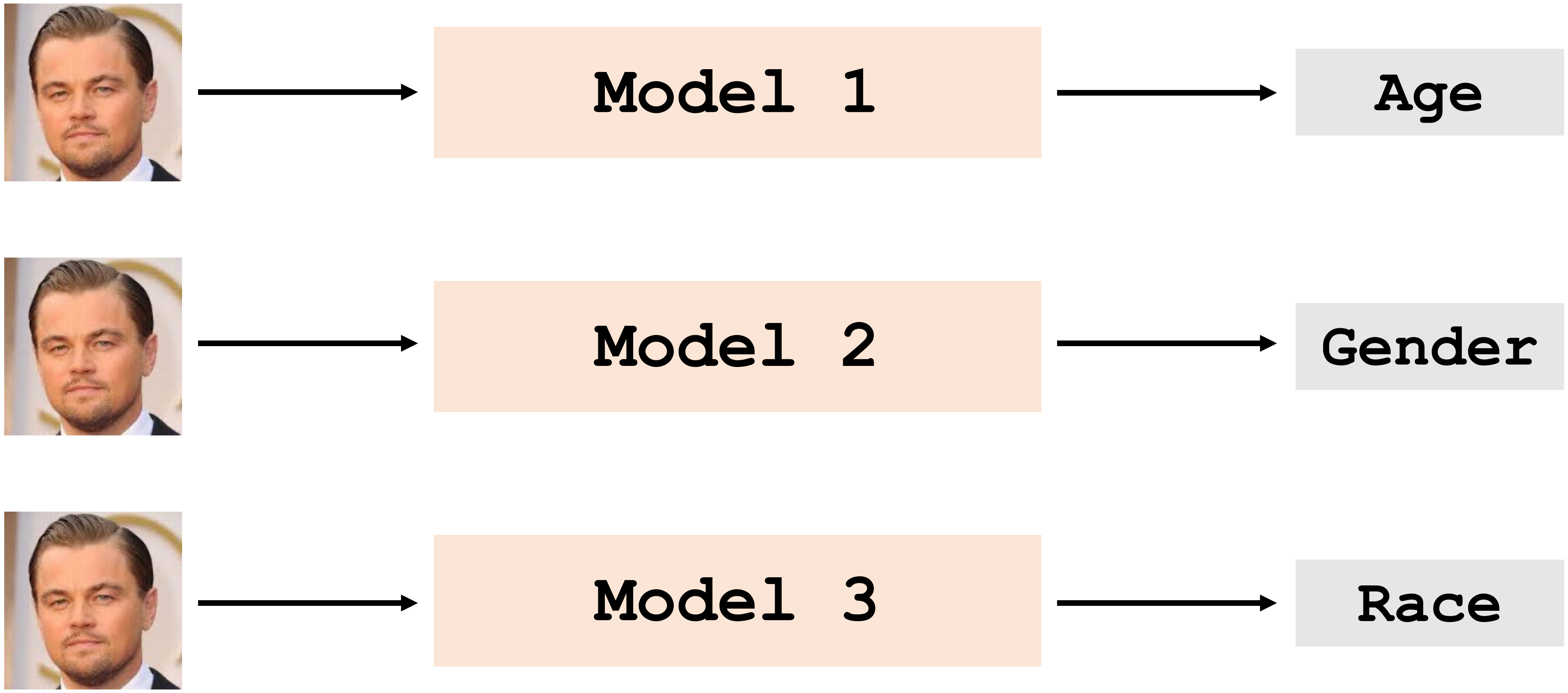
Stochastic gradient

→ different local optima

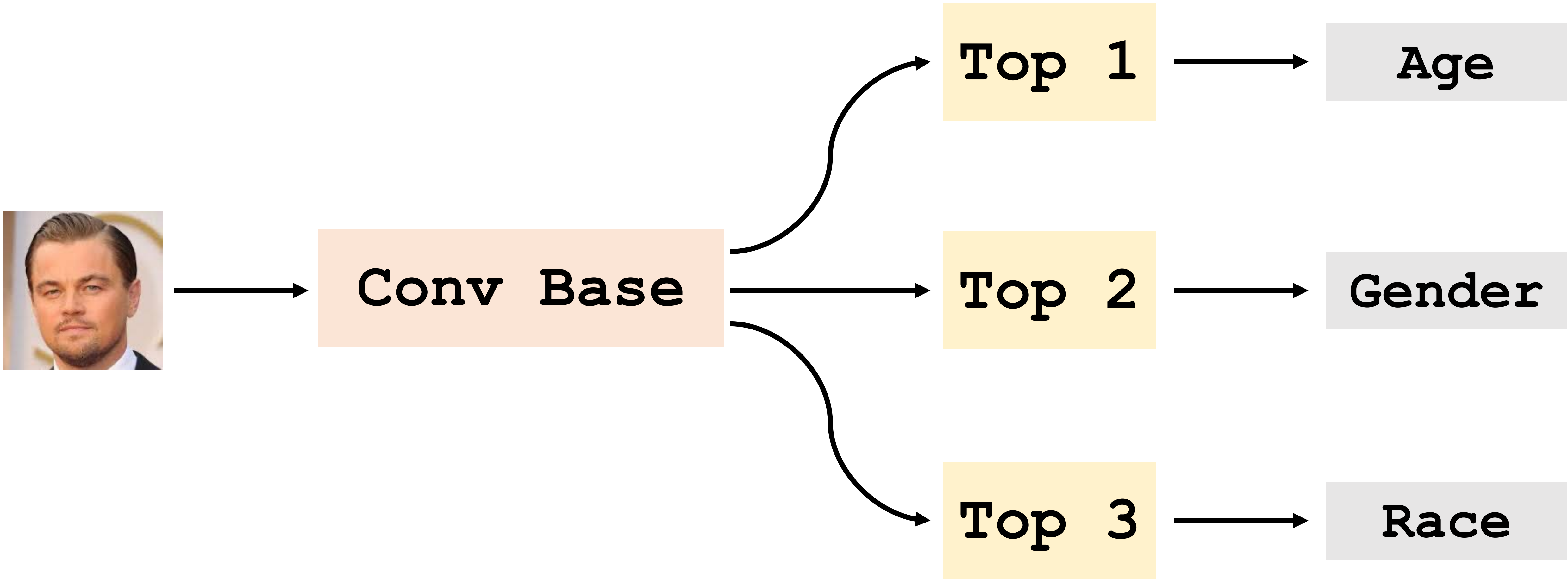
- Ensemble methods reduce variance.

Multi-Task Learning

Without Parameter Sharing



Multi-Task Learning



Multi-Task Learning

$\mathrm{Loss1} = (\mathrm{Age\_Label} - \mathrm{Age\_Pred})^2$ **Regression**

$\mathrm{Loss2} = \mathrm{dist}(\mathrm{Gender\_Label}, \mathrm{Gender\_Pred})$ **Binary classification**

$\mathrm{Loss3} = \mathrm{dist}(\mathrm{Race\_Label}, \mathrm{Race\_Pred})$ **Multi-class classification**

- Objective function: $\mathrm{Loss1} + \lambda \cdot \mathrm{Loss2} + \gamma \cdot \mathrm{Loss3}.$

Multi-Task Learning

$$\text{Loss1} = (\text{Age\_Label} - \text{Age\_Pred})^2$$

Regression

$$\text{Loss2} = \text{dist}(\text{Gender\_Label}, \text{Gender\_Pred})$$

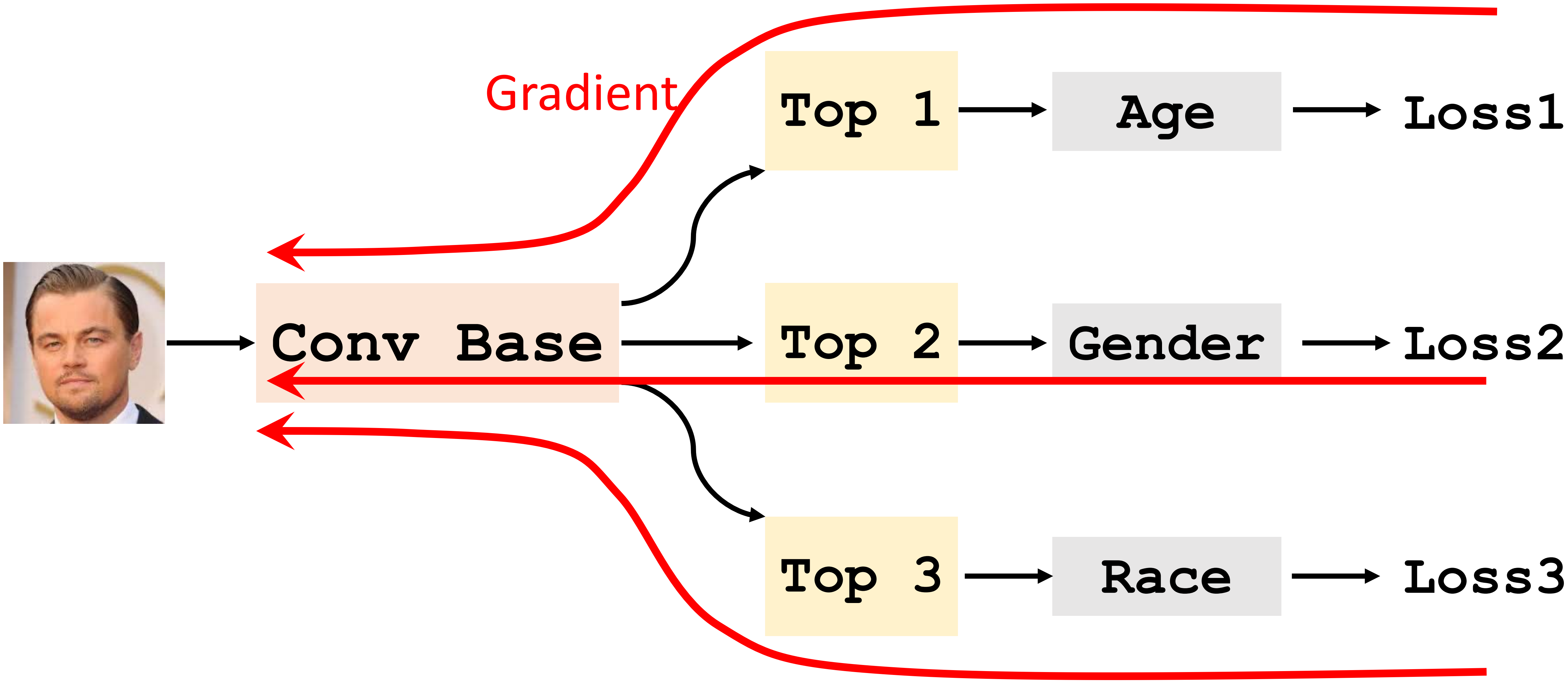
Binary classification

$$\text{Loss3} = \text{dist}(\text{Race\_Label}, \text{Race\_Pred})$$

Multi-class classification

- Objective function: $\text{Loss1} + \lambda \cdot \text{Loss2} + \gamma \cdot \text{Loss3}.$
- Why hyper-parameters λ and γ?
  - Loss1 is ~10.
  - Loss2 and Loss3 are ~0.1.
  - Without the scaling, the Conv Base will be determined by the age task.

Multi-Task Learning



Summary

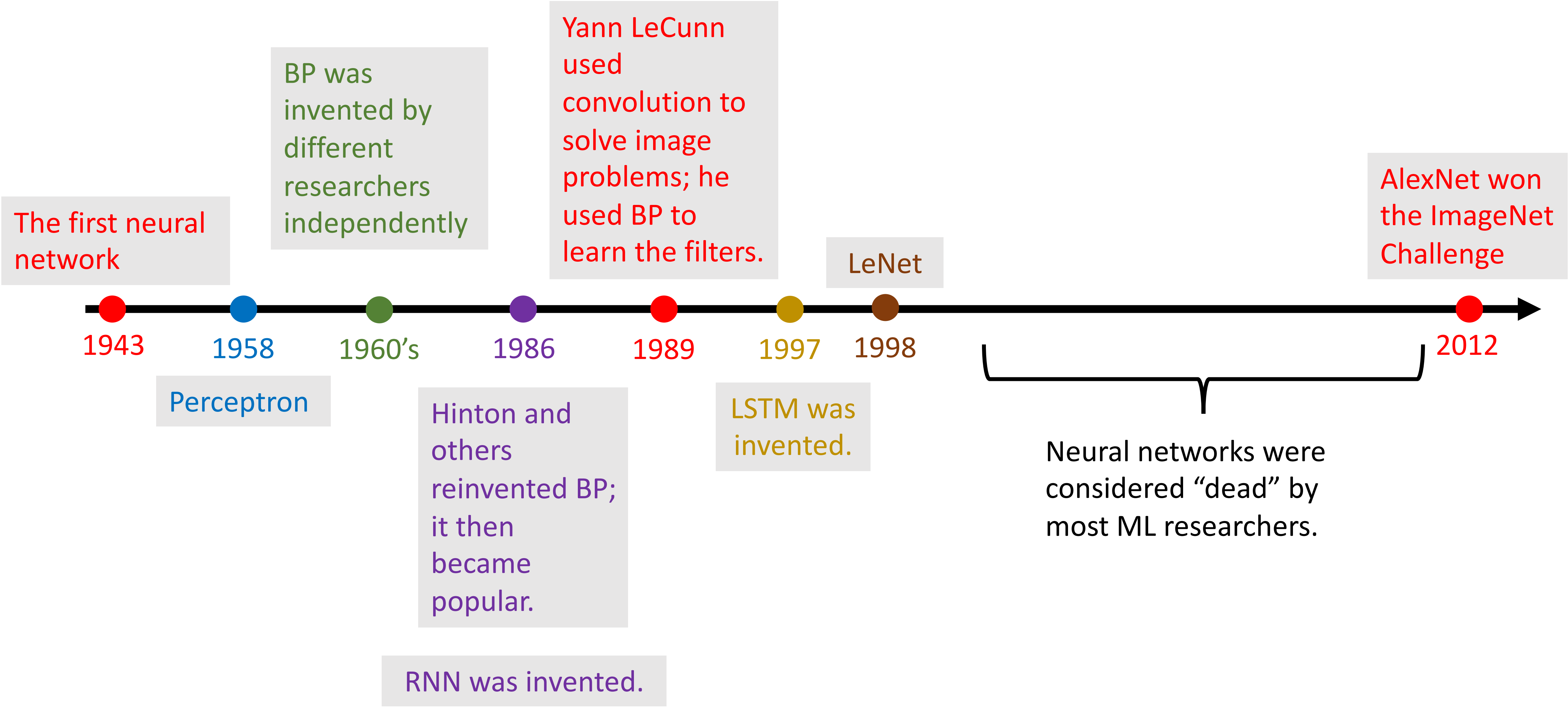
Tricks for Better Generalization

- Trick 1: Dropout regularization.
- Trick 2: Data augmentation.
- Trick 3: Pretrain.
- Trick 4: Ensemble methods.
- Trick 5: Multi-task learning.

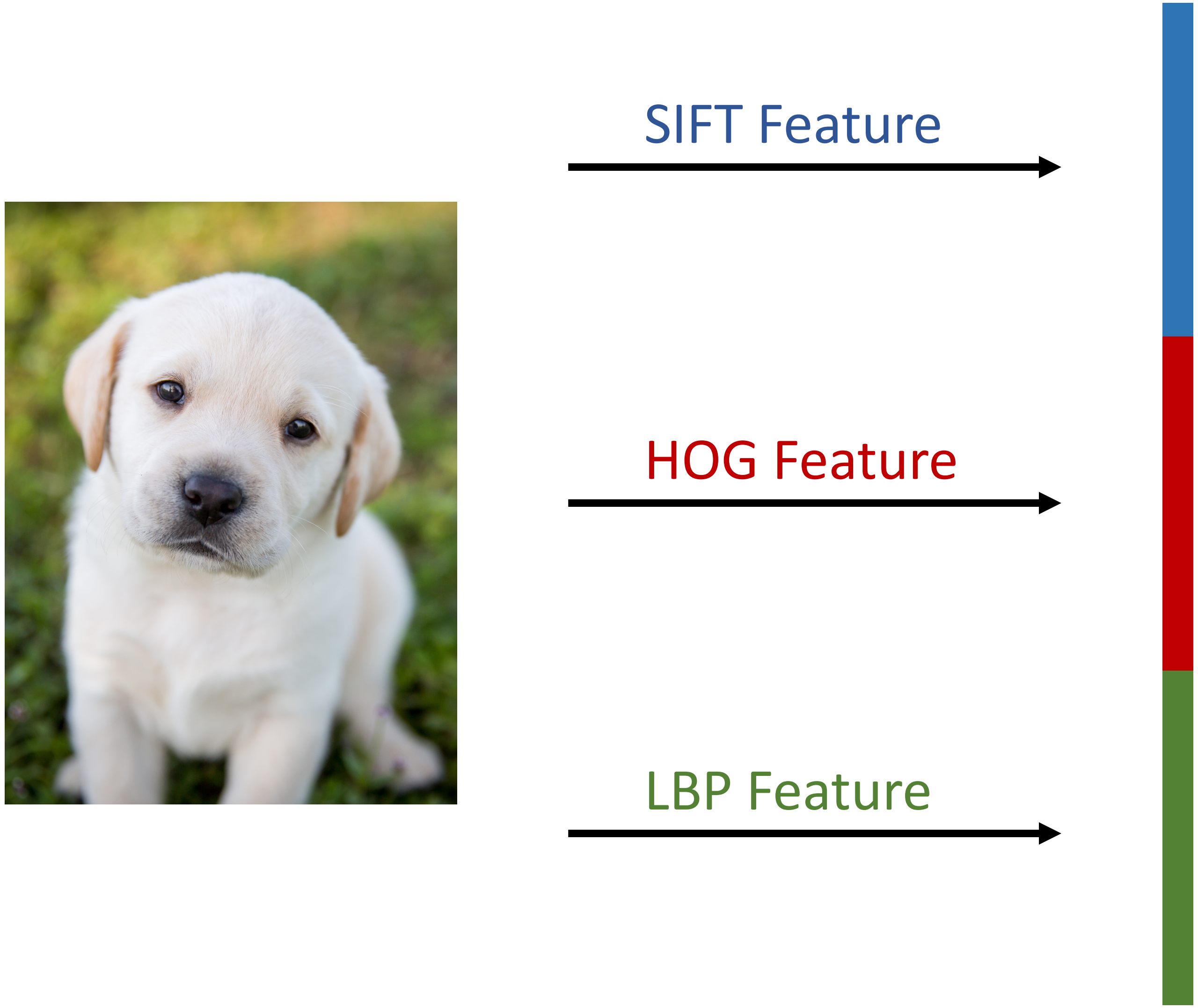
Common CNN Architectures

Background

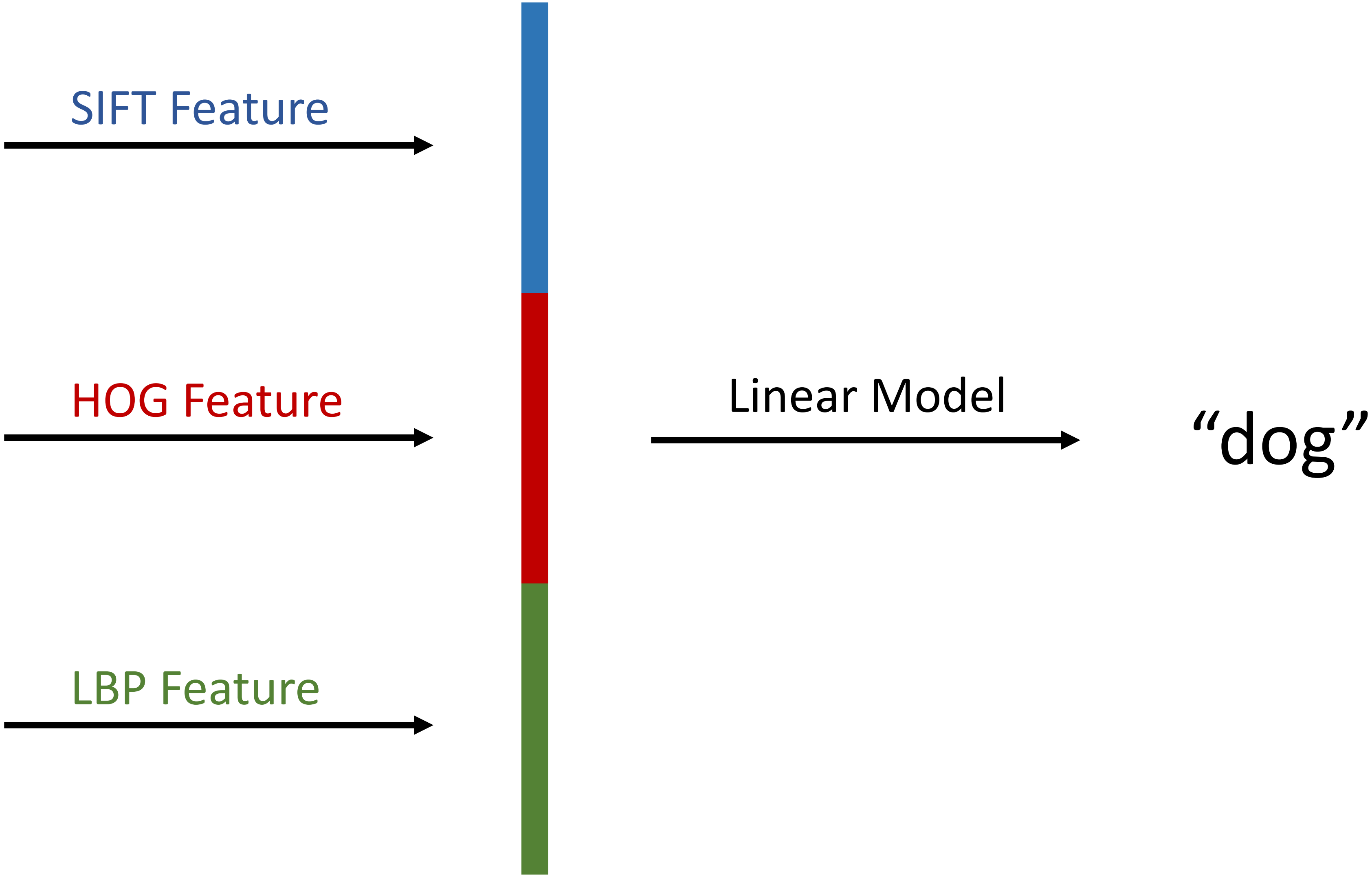
History of Neural Networks



Traditional Approaches

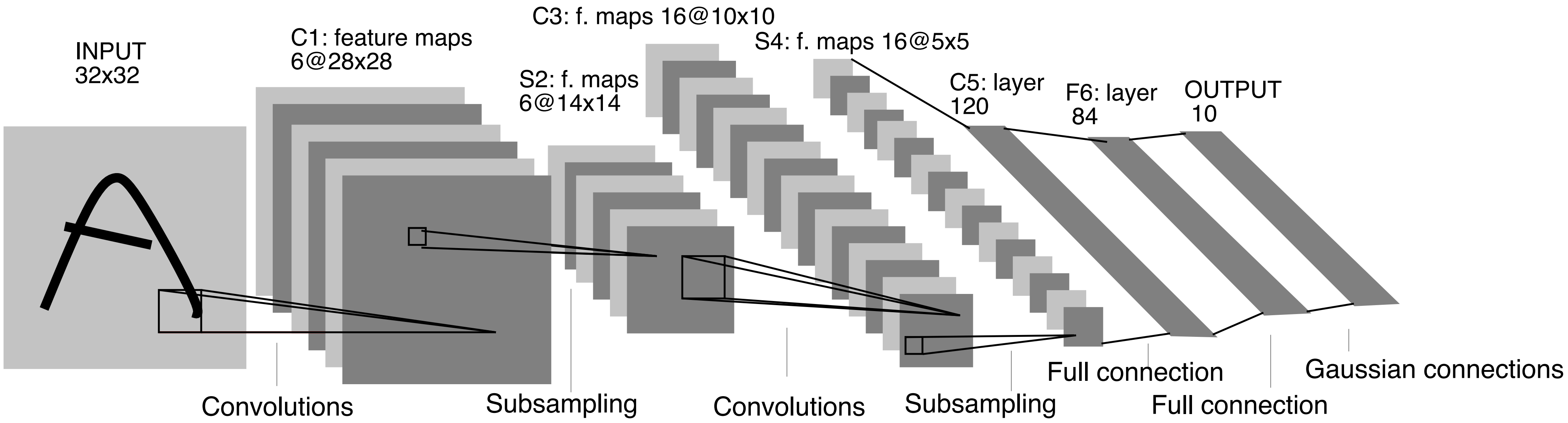


Traditional Approaches



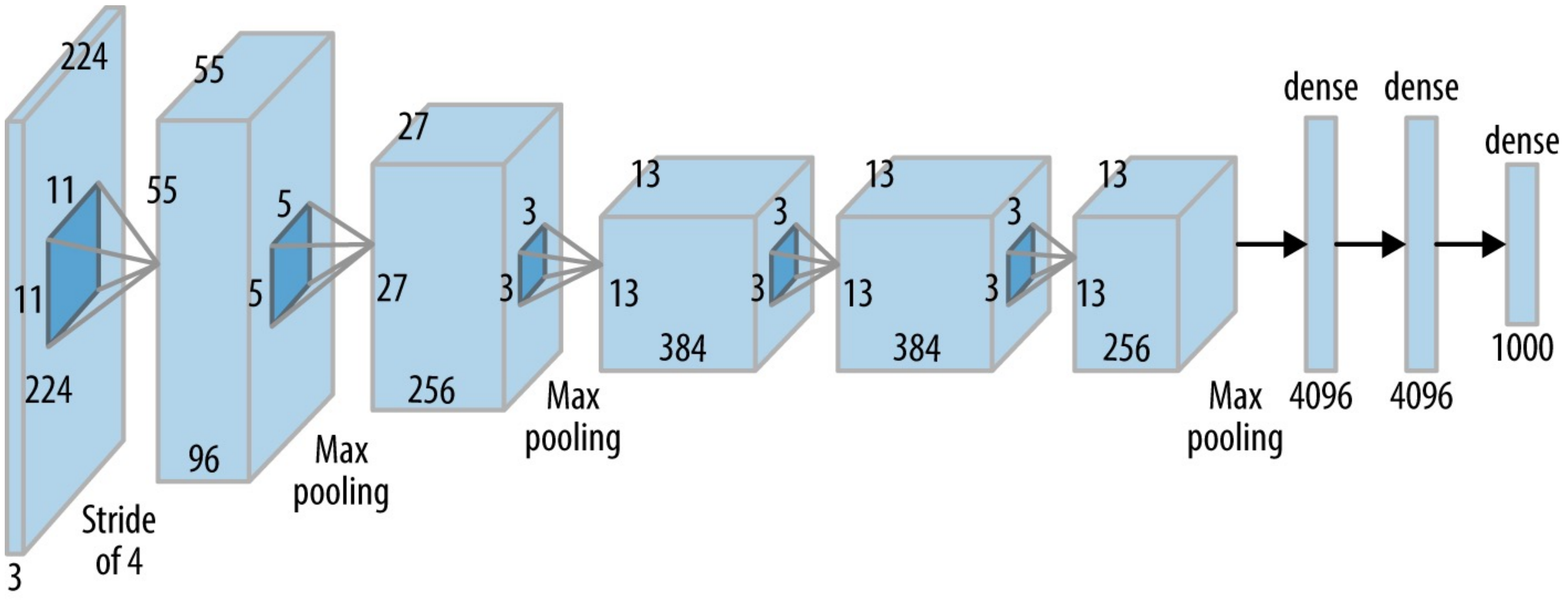
Classic CNN Architectures

LeNet-5 [Yan LeCun *et al.* 1998]



- Layers: 2 Conv + 2 FC layers.
- Paper: [Gradient-based learning applied to document recognition]

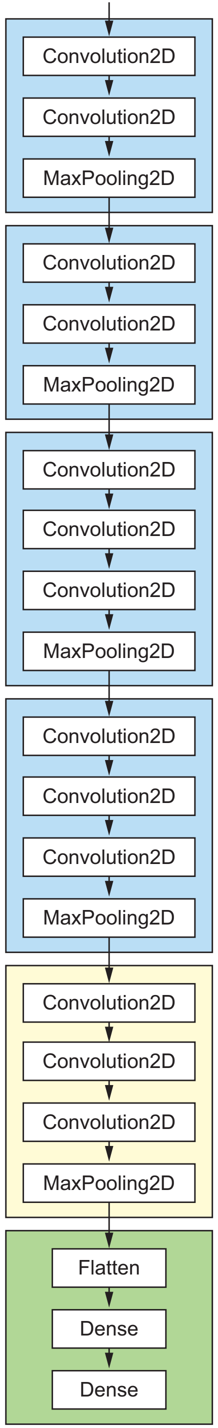
AlexNet [Alex Krizhevsky *et al*. 2012]



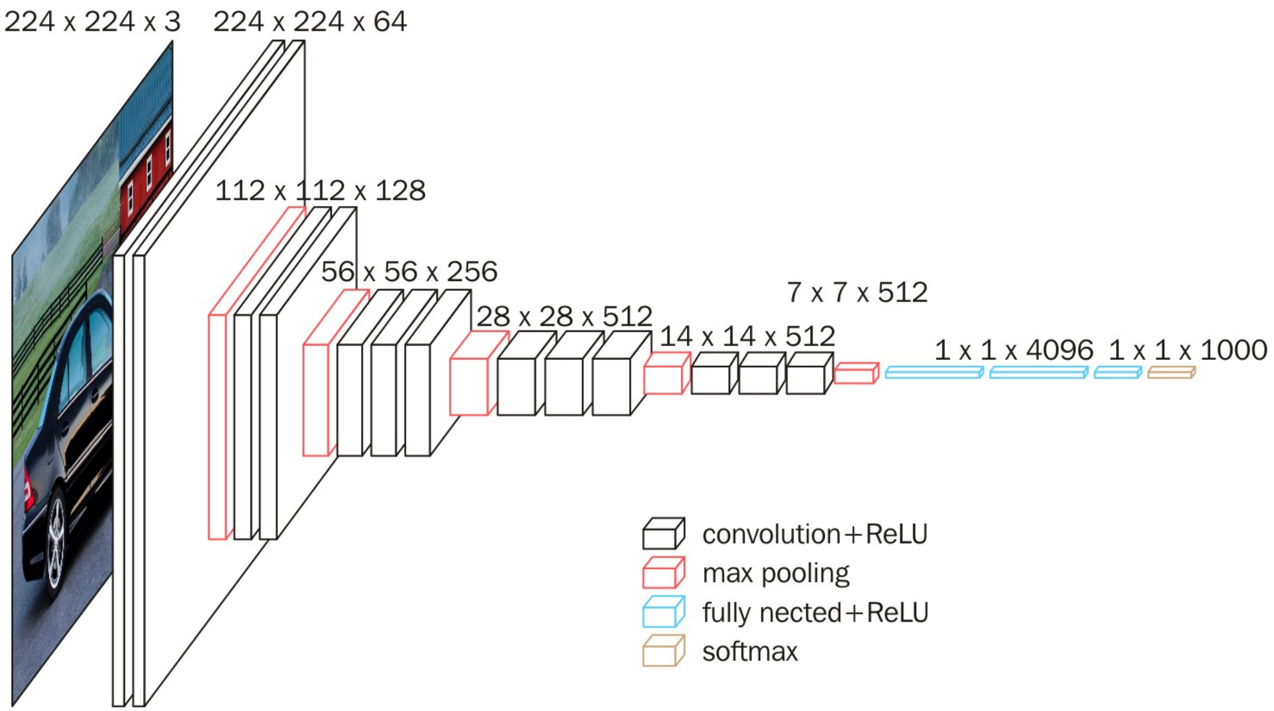
- Layers: 5 Conv + 3 FC layers.
- Number of trainable parameters: $60M$.
- Paper: [ImageNet Classification with Deep Convolutional Neural Networks]

```python
DROPOUT = 0.5
model_input = Input(shape = (227, 227, 3))

# First convolutional Layer (96x11x11)
z = Conv2D(filters = 96, kernel_size = (11,11), strides = (4,4), activation = "relu")(model_input)
z = MaxPooling2D(pool_size = (3,3), strides=(2,2))(z)
z = BatchNormalization()(z)

# Second convolutional Layer (256x5x5)
z = ZeroPadding2D(padding = (2,2))(z)
z = Convolution2D(filters = 256, kernel_size = (5,5), strides = (1,1), activation = "relu")(z)
z = MaxPooling2D(pool_size = (3,3), strides=(2,2))(z)
z = BatchNormalization()(z)

# Rest 3 convolutional layers
z = ZeroPadding2D(padding = (1,1))(z)
z = Convolution2D(filters = 384, kernel_size = (3,3), strides = (1,1), activation = "relu")(z)

z = ZeroPadding2D(padding = (1,1))(z)
z = Convolution2D(filters = 384, kernel_size = (3,3), strides = (1,1), activation = "relu")(z)

z = ZeroPadding2D(padding = (1,1))(z)
z = Convolution2D(filters = 256, kernel_size = (3,3), strides = (1,1), activation = "relu")(z)

z = MaxPooling2D(pool_size = (3,3), strides=(2,2))(z)
z = Flatten()(z)

z = Dense(4096, activation="relu")(z)
z = Dropout(DROPOUT)(z)

z = Dense(4096, activation="relu")(z)
z = Dropout(DROPOUT)(z)

final_dim = 1 if N_CATEGORY == 2 else N_CATEGORY
final_act = "sigmoid" if N_CATEGORY == 2 else "softmax"
model_output = Dense(final_dim, activation=final_act)(z)

model = Model(model_input, model_output)
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 227, 227, 3)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 55, 55, 96)        34944
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 27, 27, 96)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 27, 27, 96)        384
_________________________________________________________________
zero_padding2d_5 (ZeroPaddin (None, 31, 31, 96)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 27, 27, 256)       614656
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 13, 13, 256)       0
_________________________________________________________________
batch_normalization_4 (Batch (None, 13, 13, 256)       1024
_________________________________________________________________
zero_padding2d_6 (ZeroPaddin (None, 15, 15, 256)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 13, 13, 384)       885120
_________________________________________________________________
zero_padding2d_7 (ZeroPaddin (None, 15, 15, 384)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 13, 13, 384)       1327488
_________________________________________________________________
zero_padding2d_8 (ZeroPaddin (None, 15, 15, 384)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 13, 13, 256)       884992
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_3 (Dense)              (None, 4096)              37752832
_________________________________________________________________
dropout_3 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_4 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_4 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               409700
=================================================================
Total params: 58,692,452.0
Trainable params: 58,691,748.0
Non-trainable params: 704.0
_________________________________________________________________
```



VGG16 [Simonyan & Zisserman 2014]

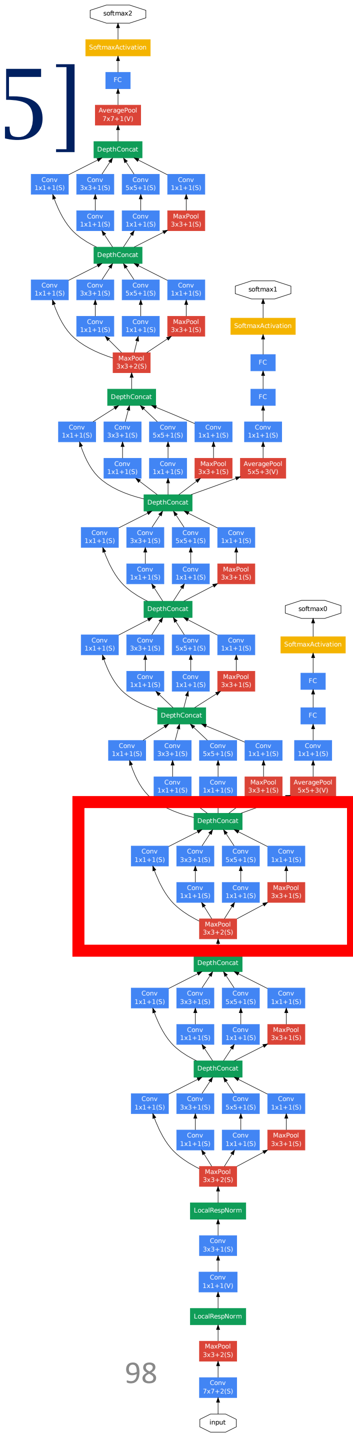


- Layers: 13 Conv + 2 FC layers.
- Number of trainable parameters: $138M$.
- Paper: [Very Deep Convolutional Networks for Large-Scale Image Recognition]

“Modern” CNN Architectures

GoogLeNet/Inception [Szegedy *et al.* 2015]

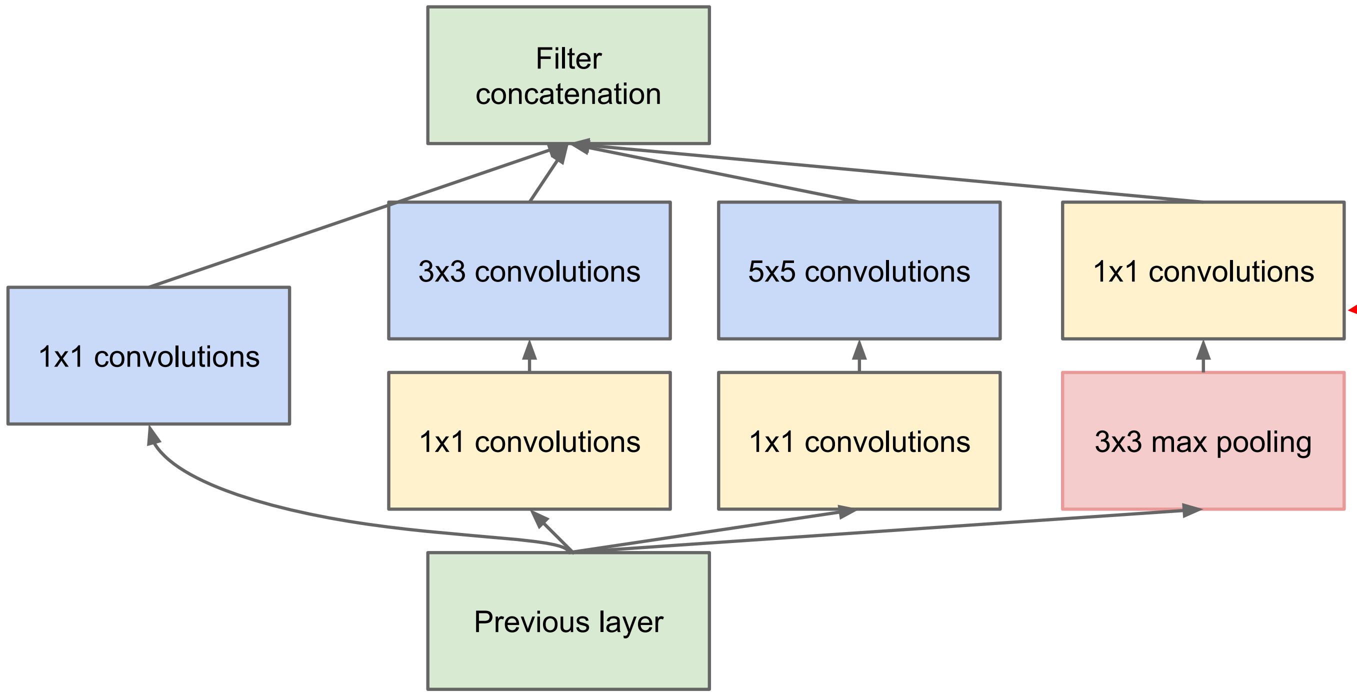
- Stack of the "Inception" modules.
- Only $5M$ parameters. (VGG16 has $138M$ parameters!)

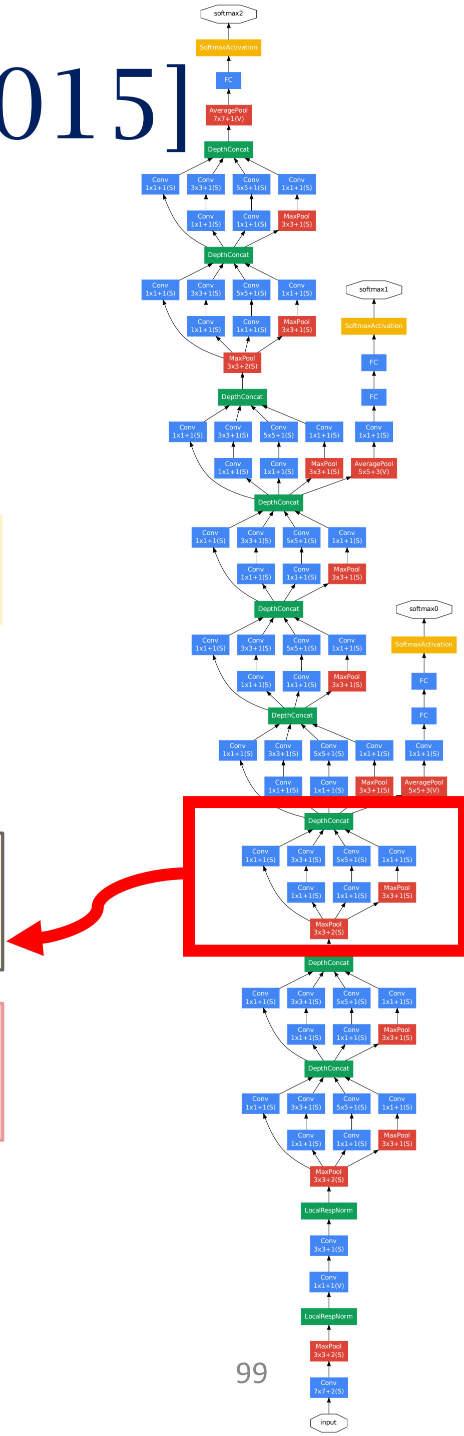


GoogLeNet/Inception [Szegedy *et al.* 2015]

- Stack of the “Inception” modules.
- Only $5M$ parameters. (VGG16 has $138M$ parameters!)

An inception module

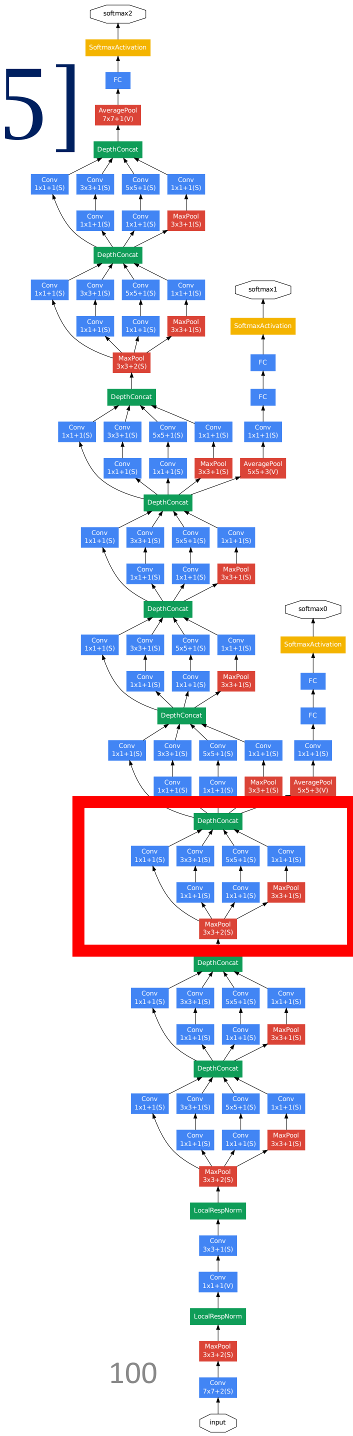




GoogLeNet/Inception [Szegedy *et al.* 2015]

```python
from keras.layers import Input, Conv2D, MaxPooling2D, concatenate

x_input = Input(shape=(d1, d2, 64*4))

tower1 = Conv2D(64, (1,1), padding='same', activation='relu')(x_input)

tower2 = Conv2D(64, (1,1), padding='same', activation='relu')(x_input)
tower2 = Conv2D(64, (3,3), padding='same', activation='relu')(tower2)

tower3 = Conv2D(64, (1,1), padding='same', activation='relu')(x_input)
tower3 = Conv2D(64, (5,5), padding='same', activation='relu')(tower3)

tower4 = MaxPooling2D((3,3), strides=(1,1), padding='same')(x_input)
tower4 = Conv2D(64, (1,1), padding='same', activation='relu')(tower4)

x_output = concatenate([tower1, tower2, tower3, tower4], axis = 3)
```

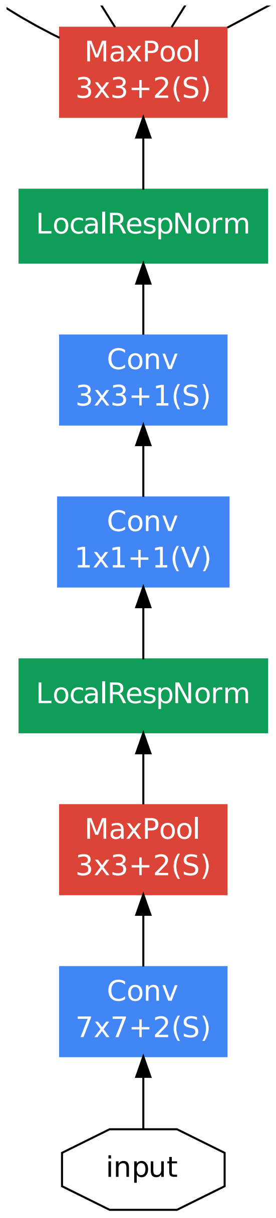
Output: $d_1 \times d_2 \times 256$ (the same as input)

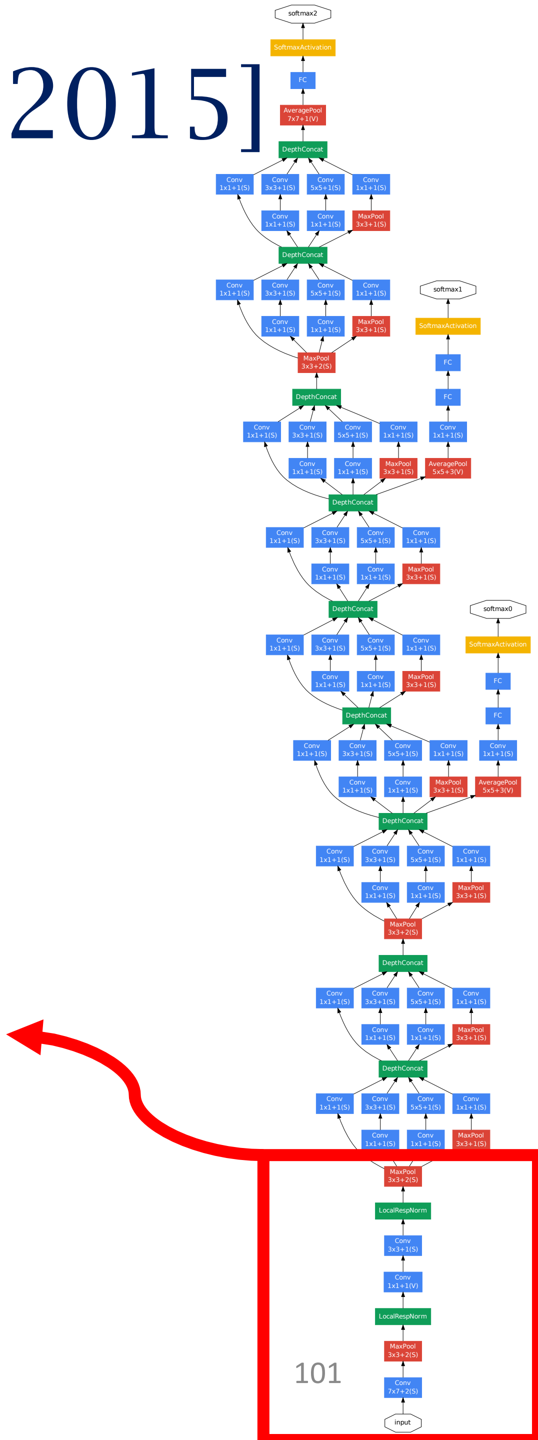


GoogLeNet/Inception [Szegedy *et al.* 2015]

Bottom layers

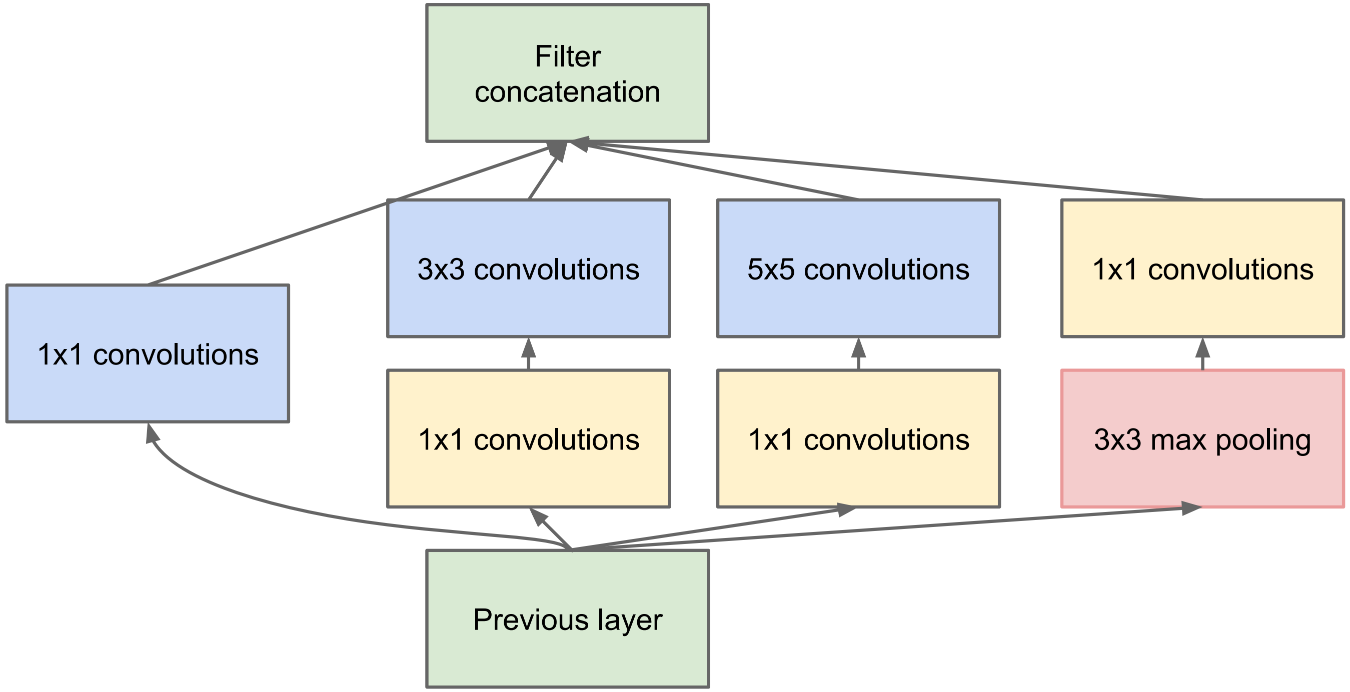
- A simple ConvNet.
- 3 Conv + 2 Pooling + 2 Normalization Layers.

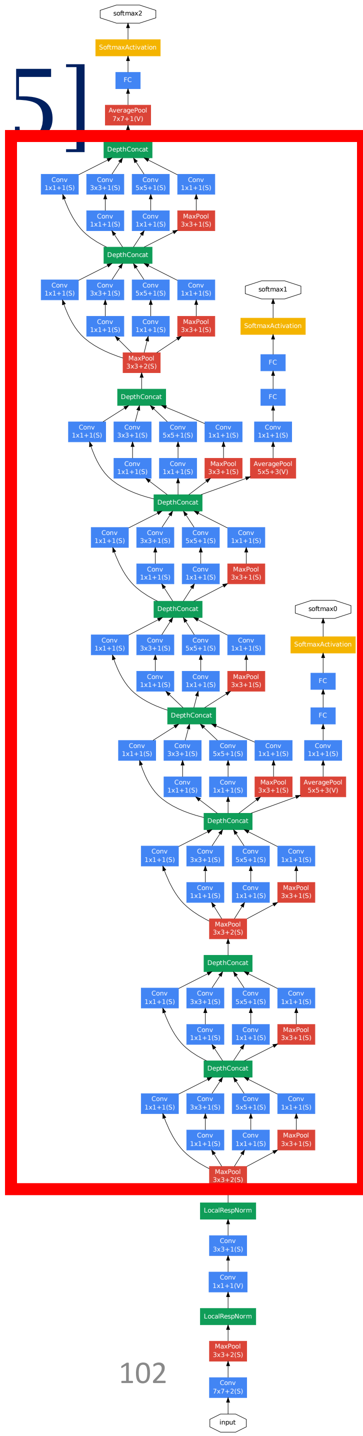




GoogLeNet/Inception [Szegedy *et al.* 2015]

Stack of 9 Inception modules



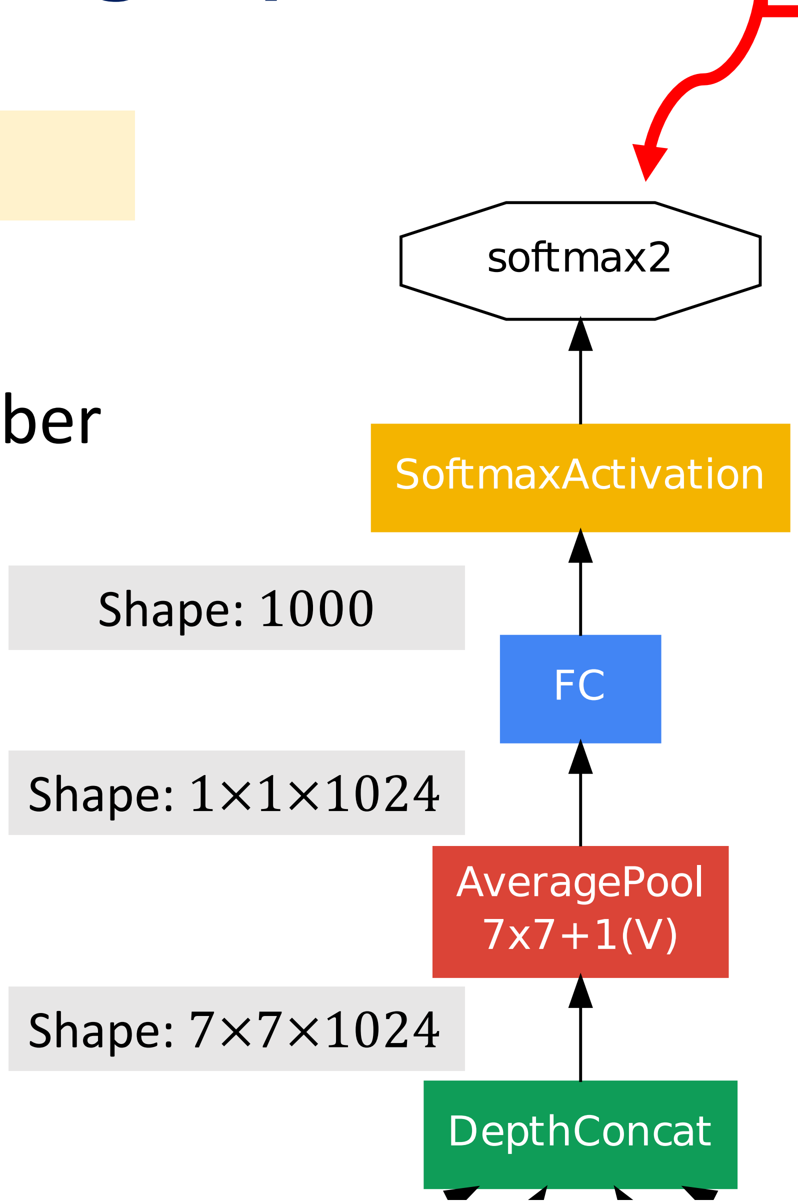


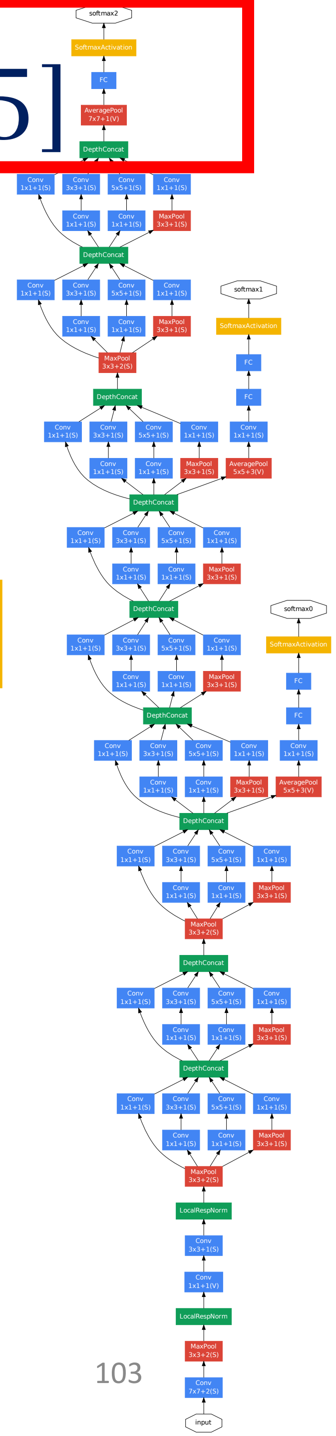
GoogLeNet/Inception [Szegedy *et al.* 2015]

Output layers

- Using AveragePool instead of Flatten.
- AveragePool Layer for reducing the number of parameters.

Question: If AveragePool is replaced by Flatten, then what will be #params in the FC layer?





GoogLeNet/Inception [Szegedy *et al.* 2015]

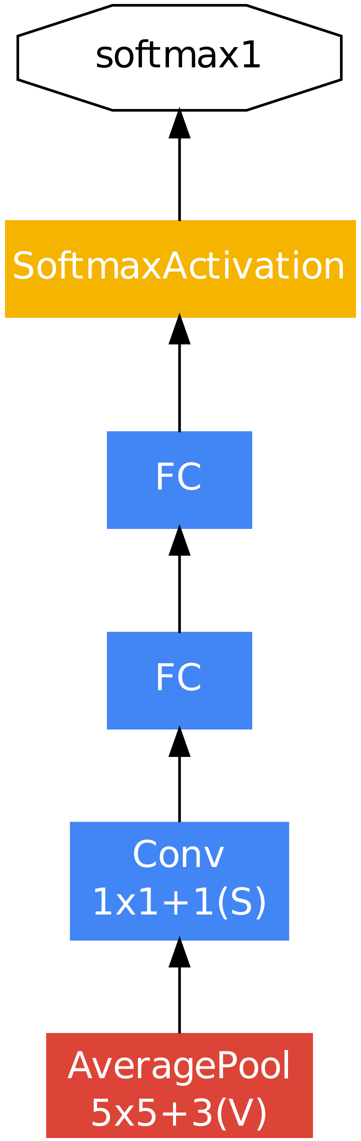
Auxiliary outputs

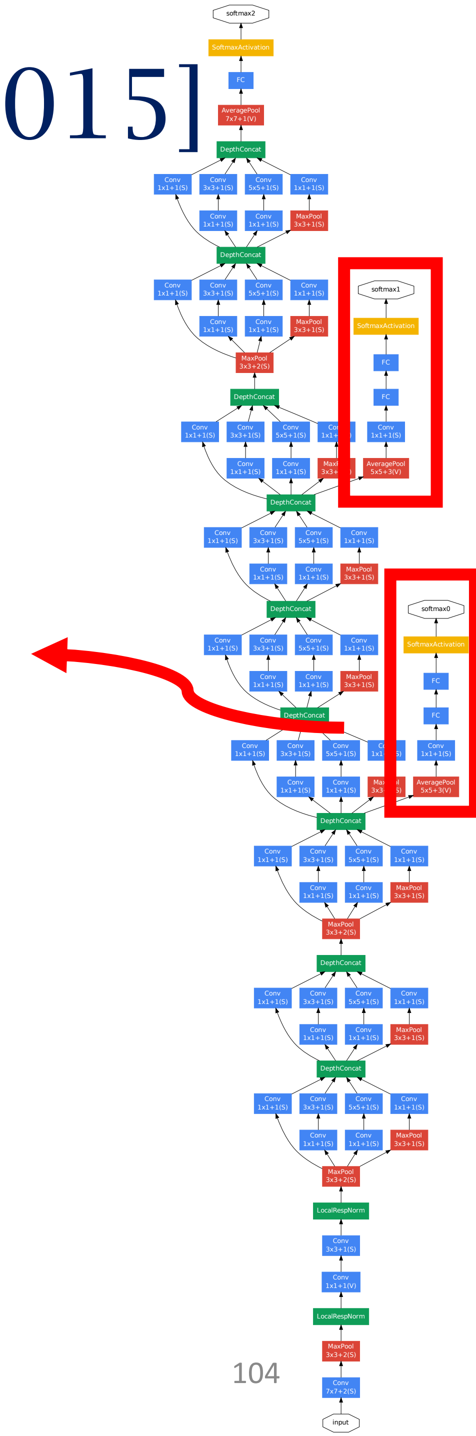
During Training:

- Inject additional gradient at lower layers.
- Make the optimization easier.

During Test:

- Disable the auxiliary outputs.





How Deep Can We Go?

- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.

classic architectures (sequential)

Question: Why VGG16 and VGG19? Why not deeper classic architectures?

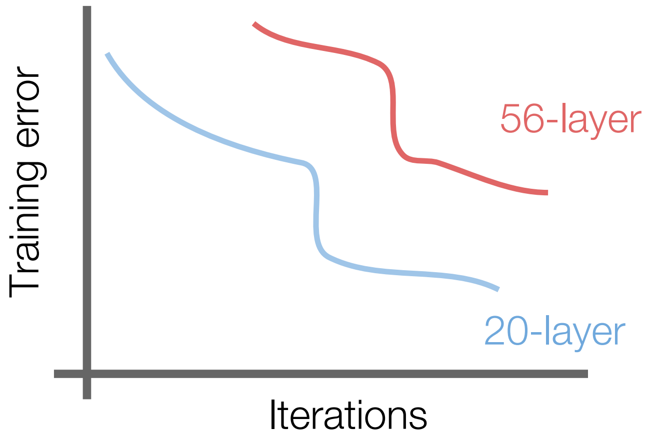
How Deep Can We Go?

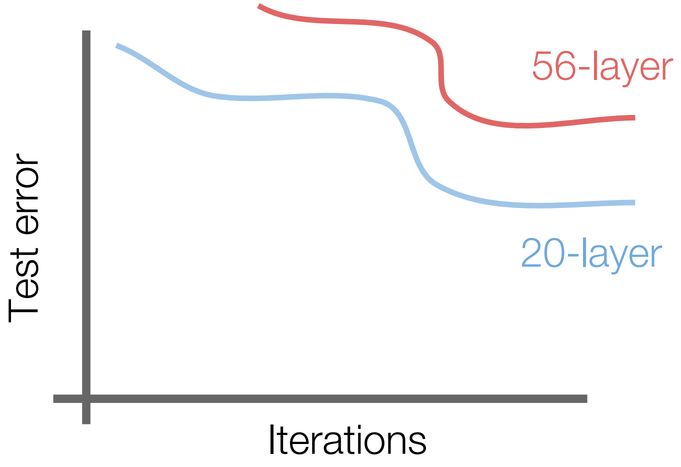
- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.

classic architectures (sequential)

Question: Why VGG16 and VGG19? Why not deeper classic architectures?

Answer: Deeper nets have worse training and test errors.





How Deep Can We Go?

- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.

classic architectures (sequential)

Question: What makes deeper VGG nets worse?

Answer: It is bad optimization, not overfitting.

How Deep Can We Go?

- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.

classic architectures (sequential)

Question: What makes deeper VGG nets worse?

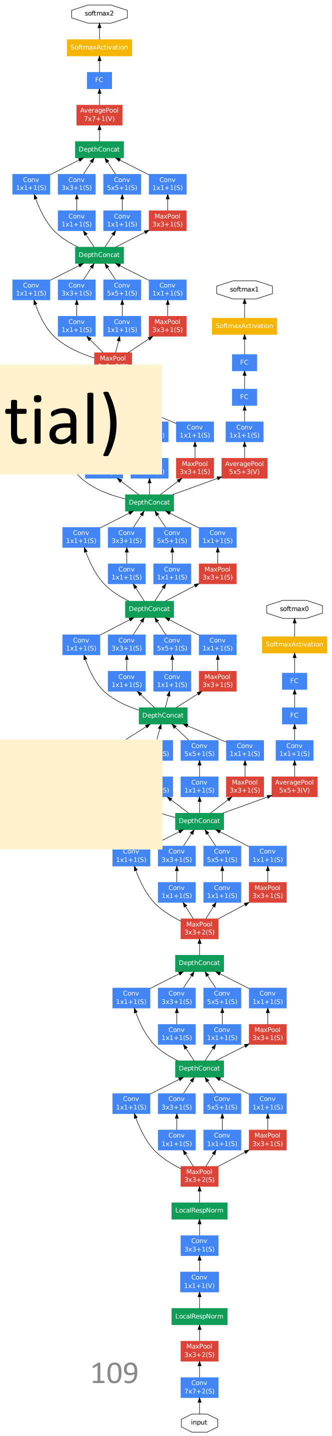
- The vanishing gradient problem.
  - Derivative of the loss function w.r.t. a bottom layer can be vanishingly small.
  - It makes deep nets difficult to optimize. The bottom layers are not well trained.
  - The model is good; but you cannot find a good local minimum.

How Deep Can We Go?

- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.
- Inception: 21 Conv + 1 FC layers.

classic architectures (sequential)

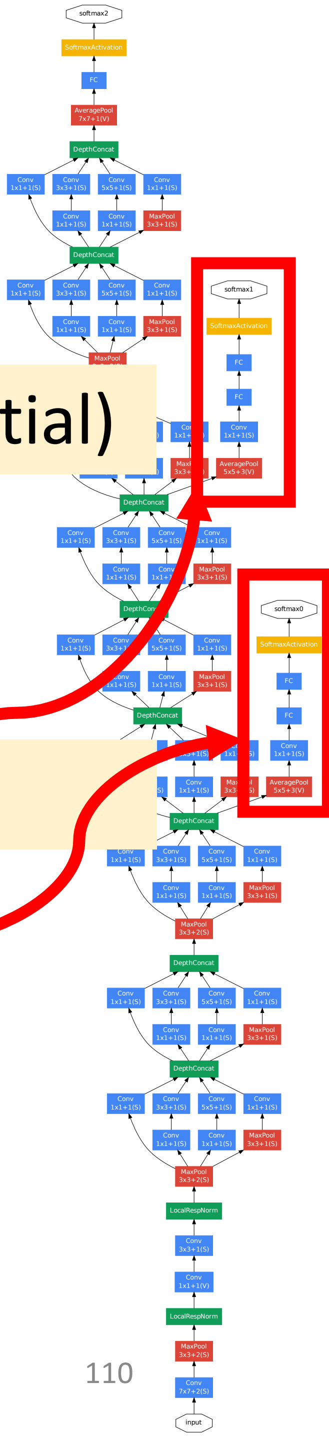
Question: Why can Inception go deeper than VGG16?



How Deep Can We Go?

- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.
- Inception: 21 Conv + 1 FC layers.

classic architectures (sequential)



Question: Why can Inception go deeper than VGG16?

- Auxiliary outputs inject additional gradient at lower layers.

How Deep Can We Go?

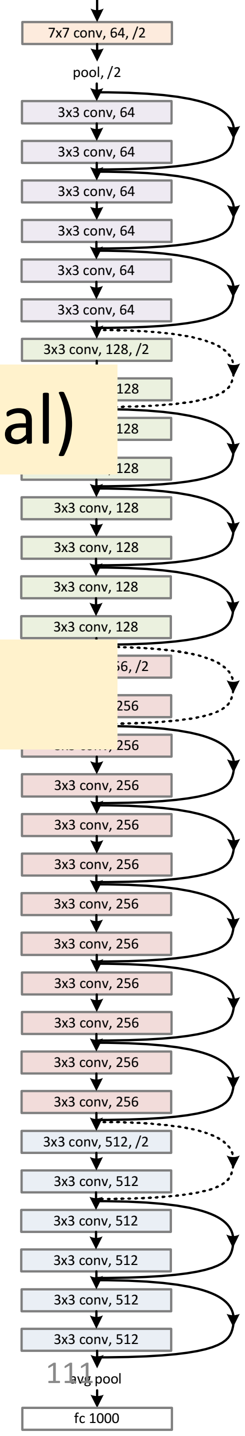
- LeNet-5: 2 Conv + 2 FC layers.
- AlexNet: 5 Conv + 3 FC layers.
- VGG16: 13 Conv + 2 FC layers.

classic architectures (sequential)

- Inception: 21 Conv + 1 FC layers.
- ResNet: Up to 151 Conv + 1 FC layers.

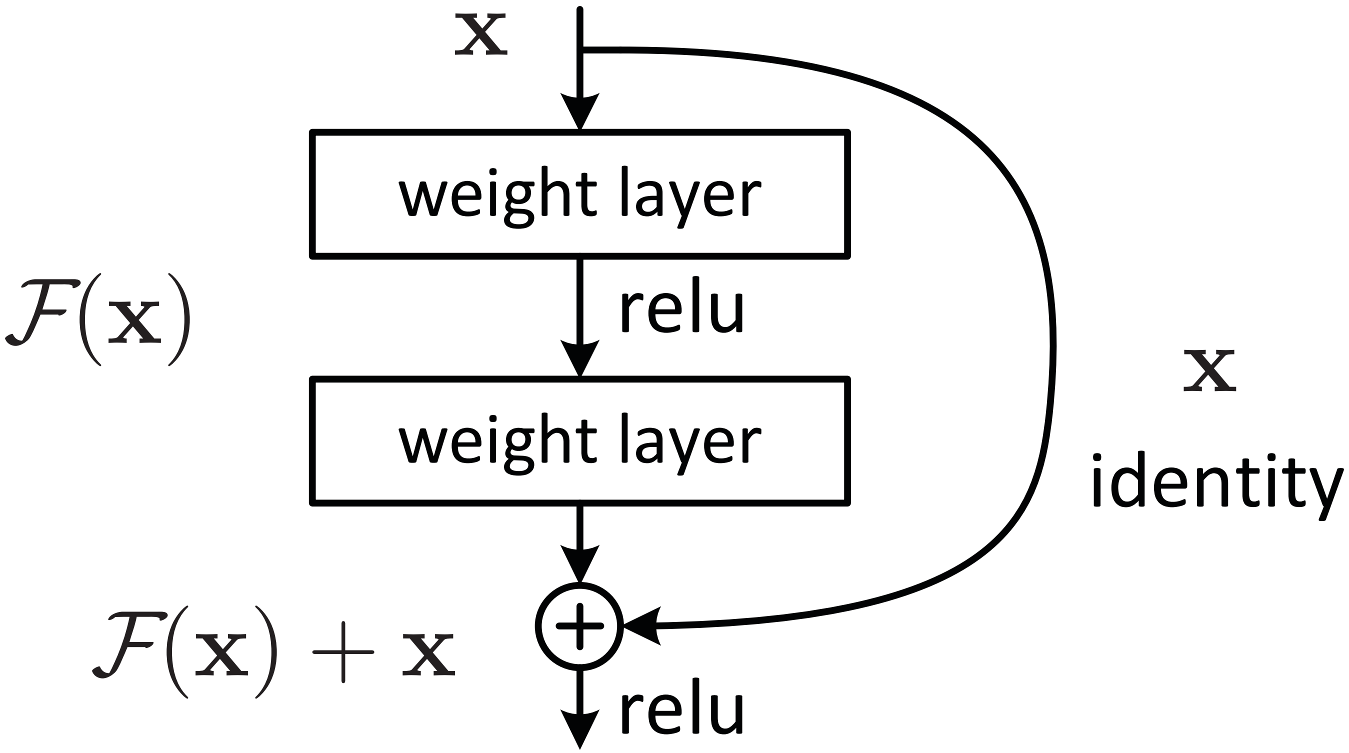
modern architectures

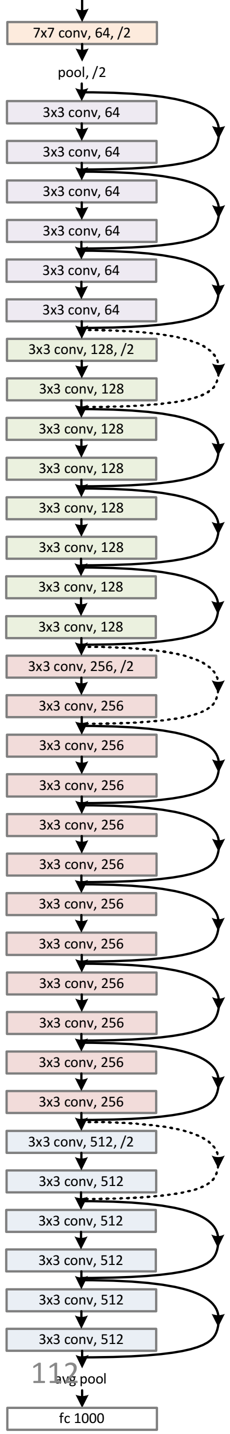
ResNet’s key idea: skip connections for curing vanishing gradient.



ResNet [He *et al.* 2015]

A block of ResNet

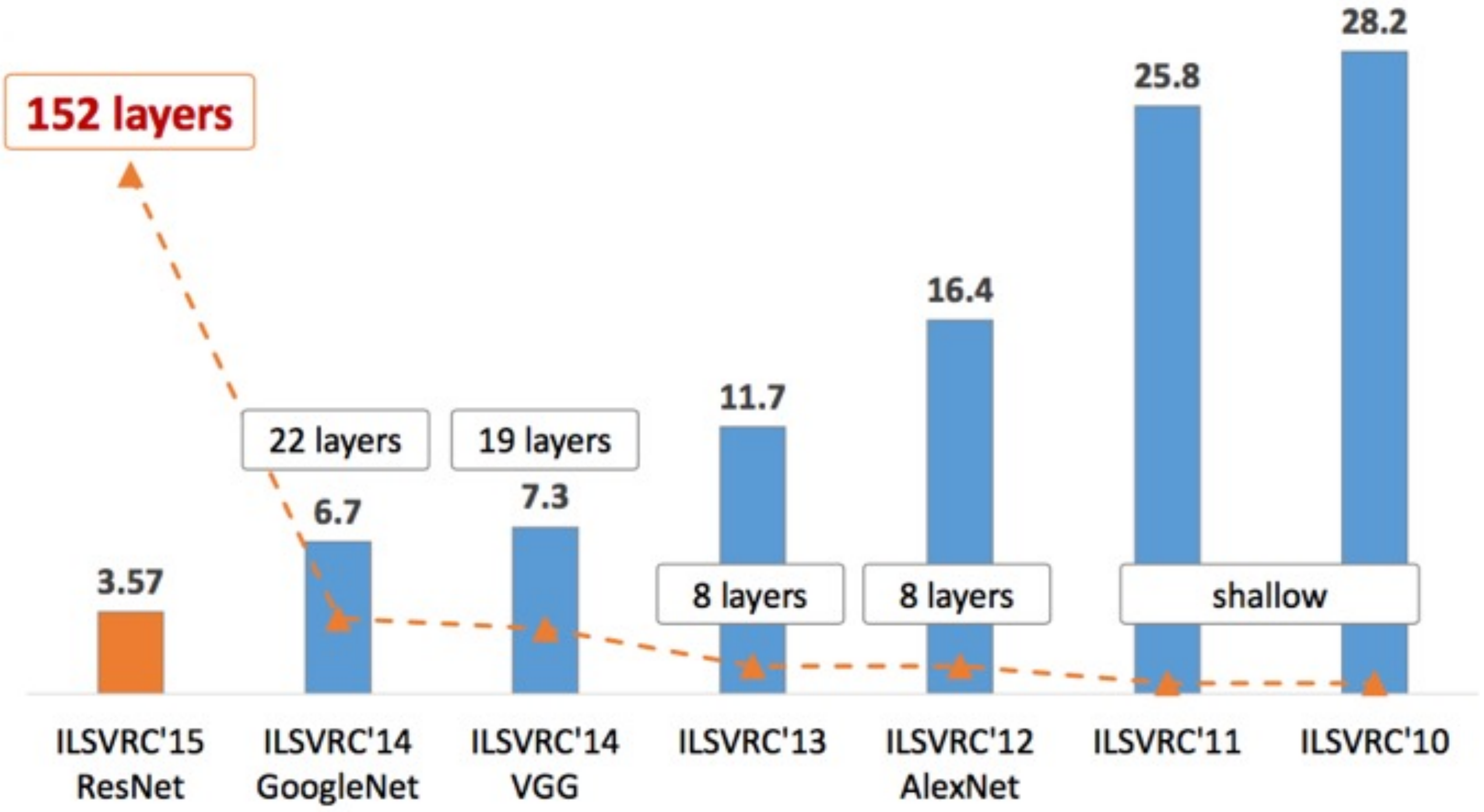




Winners of the ImageNet Challenge

number of layers

top-5 error rate



Classification Error: Top-1 vs. Top-5

Label: "**wolf**"

ConvNet →

class	pred. probability
husky	0.50
wolf	0.20
dog	0.18
fox	0.08
snow	0.01
fur	0.01
forest	0.01
⋮	

Classification Error: Top-1 vs. Top-5

Label: "**wolf**"

ConvNet

class	pred. probability
husky	0.50
wolf	0.20
dog	0.18
fox	0.08
snow	0.01
fur	0.01
forest	0.01

Evaluated by the Top-1 error,
this prediction is **wrong**!

Classification Error: Top-1 vs. Top-5

Label: "**wolf**"

ConvNet

class	pred. probability
husky	0.50
wolf	0.20
dog	0.18
fox	0.08
snow	0.01
fur	0.01
forest	0.01

Evaluated by the Top-5 error, the prediction is **correct**!