W4995 Applied Machine Learning

Even More Neural Networks

04/19/17

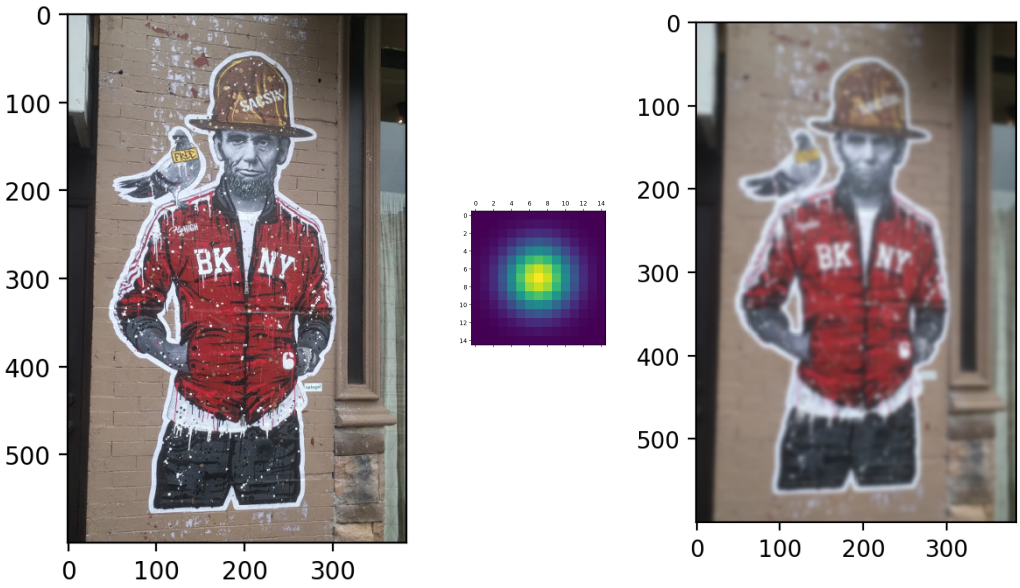
Andreas Müller

General notes for HW4

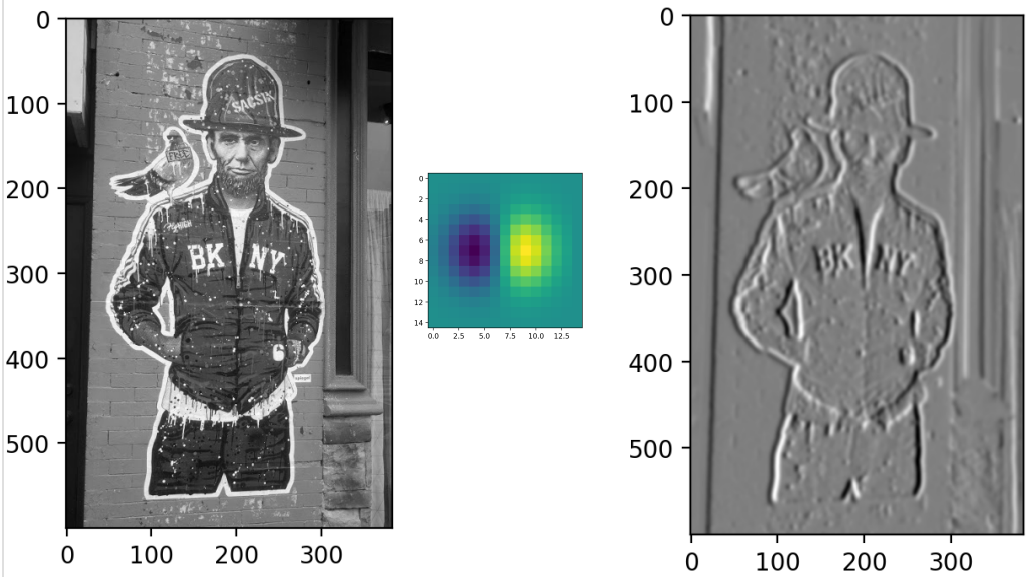
- Never iterate over rows in a dataframe!
- Use series.str for string operations (columns are series)

Convolutional neural networks

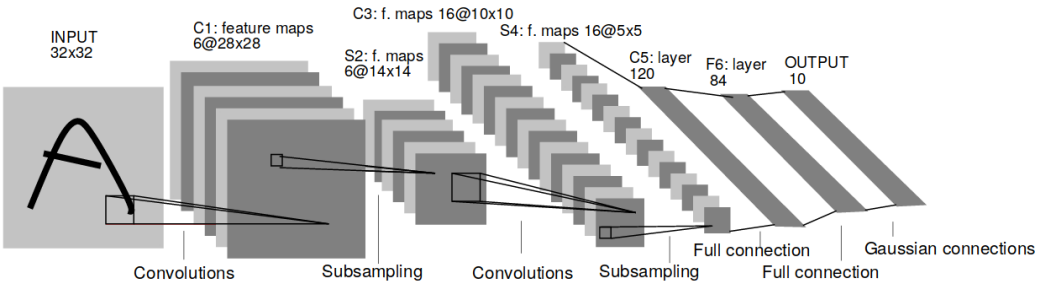
2d smoothing



2d Gradients



Convolutional Neural Networks



Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.

Gradient-based learning applied to document recognition

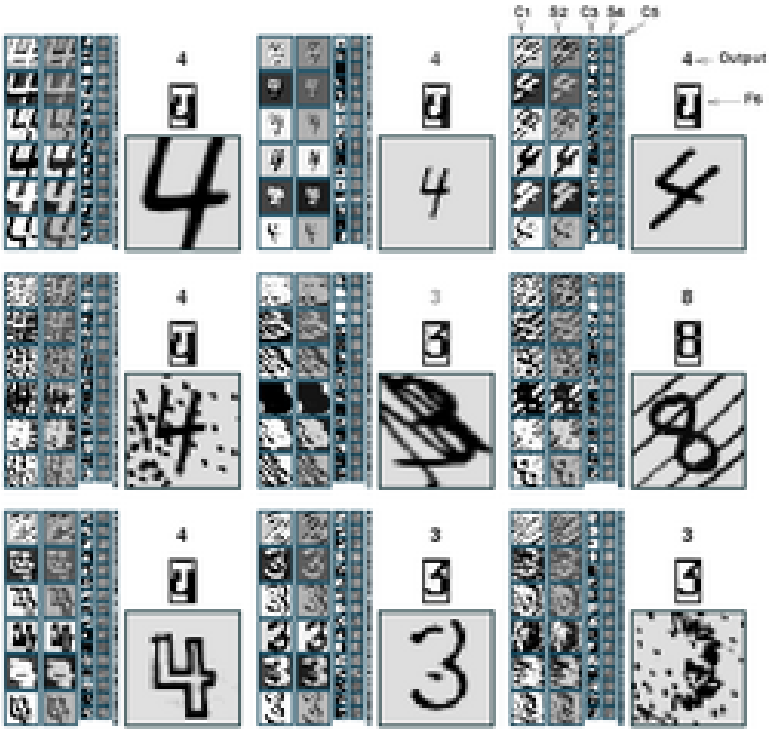
Here is the architecture of an early convolutional net form 1998. The basic architecture in current networks is still the same.

You can multiple layers of convolutions and resampling operations. You start convolving the image, which extracts local features. Each convolutions creates new “feature maps” that serve as input to later convolutions.

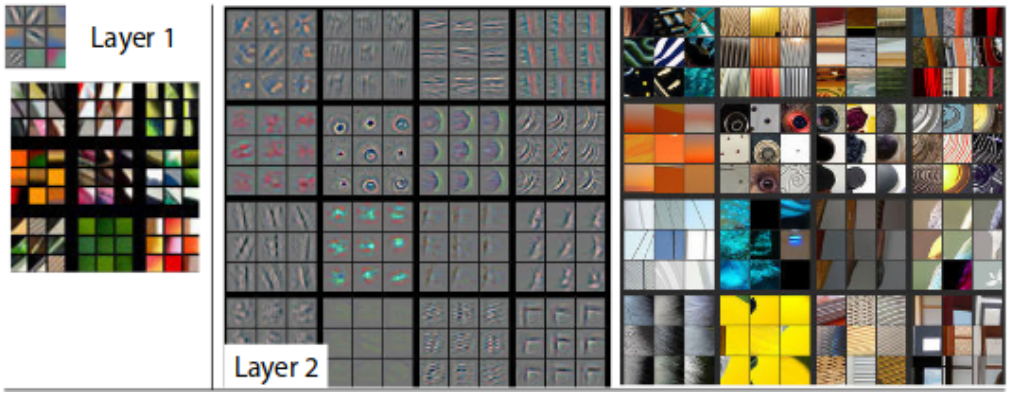
To allow more global operations, after the convolutions the image resolution is changed. Back then it was subsampling, today it is max-pooling.

So you end up with more and more feature maps with lower and lower resolution.

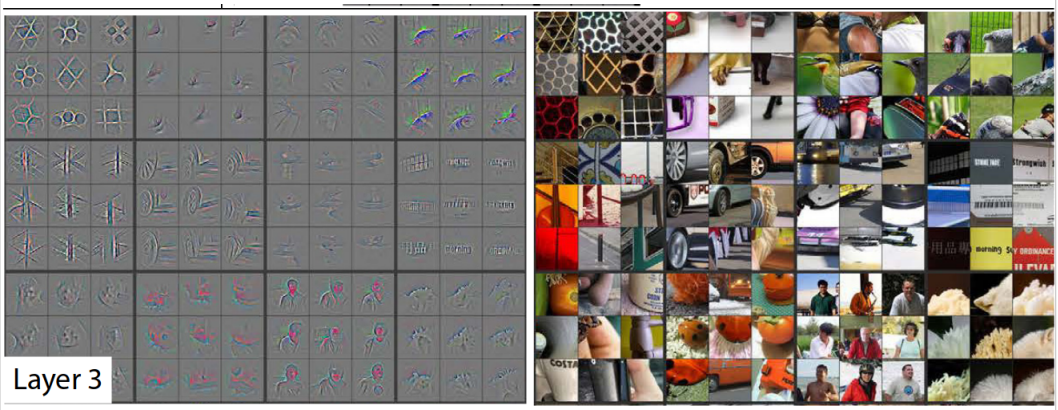
At the end, you have some fully connected layers to do the classificattion.

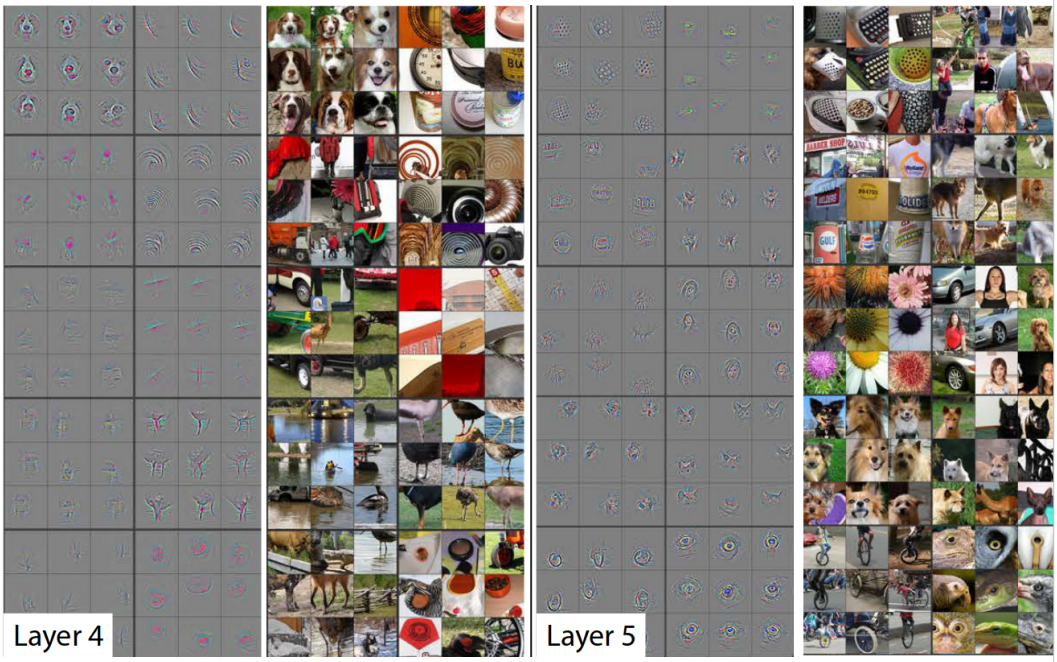

C1
S2
C3
S4
C5
4
Output
Fc
4
4
3
8
4
3
3

Deconvolution



https://www.cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf

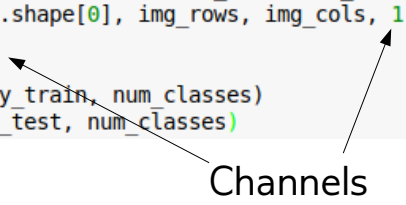

Layer 3


Layer 4
Layer 5

Conv-nets with keras

Preparing data

```
batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()


X_train_images = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
X_test_images = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```



For convolutional nets the data is n_samples, width, height, channels.

MNIST has one channel because it's grayscale. Often you have RGB channels or possibly Lab.

The position of the channels is configurable, using the "channels_first" and "channels_last" options – but you shouldn't have to worry about that.

Create tiny network

```
from keras.layers import Conv2D, MaxPooling2D, Flatten

num_classes = 10
cnn_small = Sequential()
cnn_small.add(Conv2D(8, kernel_size=(3, 3),
              activation='relu',
              input_shape=input_shape))
cnn_small.add(MaxPooling2D(pool_size=(2, 2)))
cnn_small.add(Conv2D(8, (3, 3), activation='relu'))
cnn_small.add(MaxPooling2D(pool_size=(2, 2)))
cnn_small.add(Flatten())
cnn_small.add(Dense(64, activation='relu'))
cnn_small.add(Dense(num_classes, activation='softmax'))
```

For convolutional nets we need 3 new layer types: Conv2d for 2d convolutions, MaxPooling2d for max pooling and Flatten go reshape the input for a dense layer.

There are many other options but these are the most commonly used ones.

Number of Parameters

Convolutional network for MNIST

```
cnn_small.summary()
```

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 26, 26, 8)	80
max_pooling2d_9 (MaxPooling2	(None, 13, 13, 8)	0
conv2d_10 (Conv2D)	(None, 11, 11, 8)	584
max_pooling2d_10 (MaxPooling	(None, 5, 5, 8)	0
flatten_5 (Flatten)	(None, 200)	0
dense_172 (Dense)	(None, 64)	12864
dense_173 (Dense)	(None, 10)	650

Total params: 14,178.0
Trainable params: 14,178.0
Non-trainable params: 0.0

Dense network for MNIST

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 512)	401920
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262656
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 10)	5130

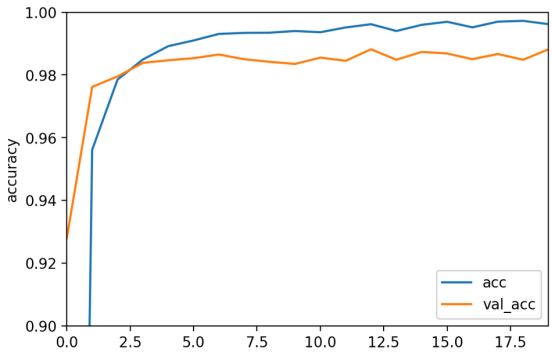
Total params: 669,706.0
Trainable params: 669,706.0
Non-trainable params: 0.0

Train and Evaluate

```
cnn.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_cnn = cnn.fit(X_train_images, y_train,
                      batch_size=128, epochs=20, verbose=1, validation_split=.1)
```

```
cnn.evaluate(X_test_images, y_test)
```

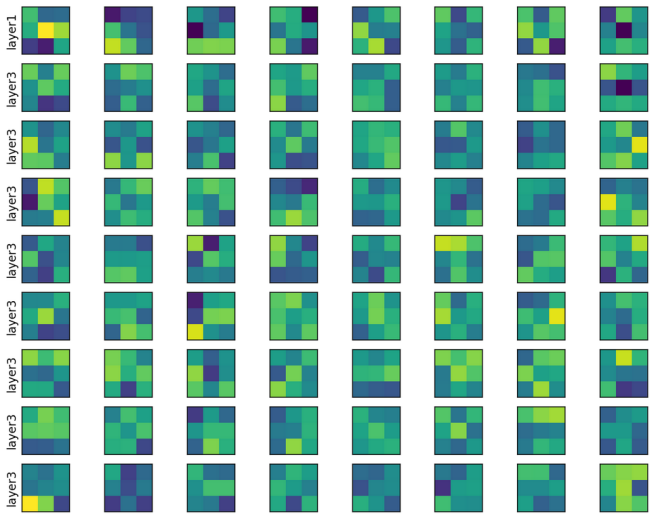
```
 9952/10000 [============================>.] - ETA: 0s
[0.089020583277629253, 0.98429999999999995]
```

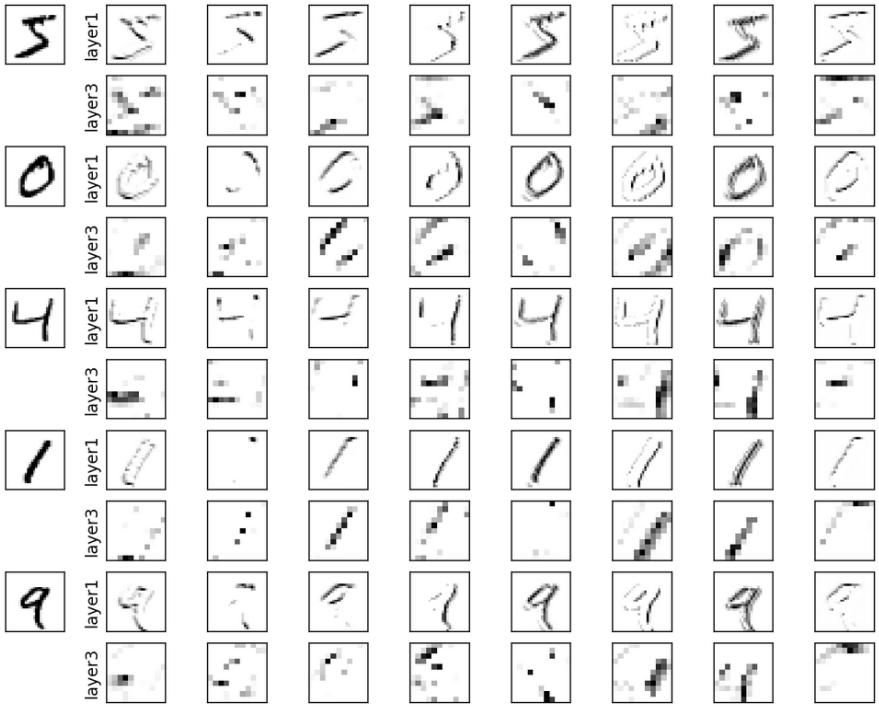


Visualize filters

```
weights, biases = cnn_small.layers[0].get_weights()
weights2, biases2 = cnn_small.layers[2].get_weights()
print(weights.shape)
print(weights2.shape)
```

```
(3, 3, 1, 8)
(3, 3, 8, 8)
```




layer1
layer3
layer1
layer3
layer1
layer3
layer1
layer3
layer1
layer3

Batch normalization

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

https://arxiv.org/abs/1502.03167

Another relatively recent advance in neural networks is batch normalization. The idea is that neural networks learn best when the input is zero mean and unit variance. We can scale the data to get that.

But each layer inside a neural network is itself a neural network with inputs given by the previous layer. And that output might have much larger or smaller scale (depending on the activation function).

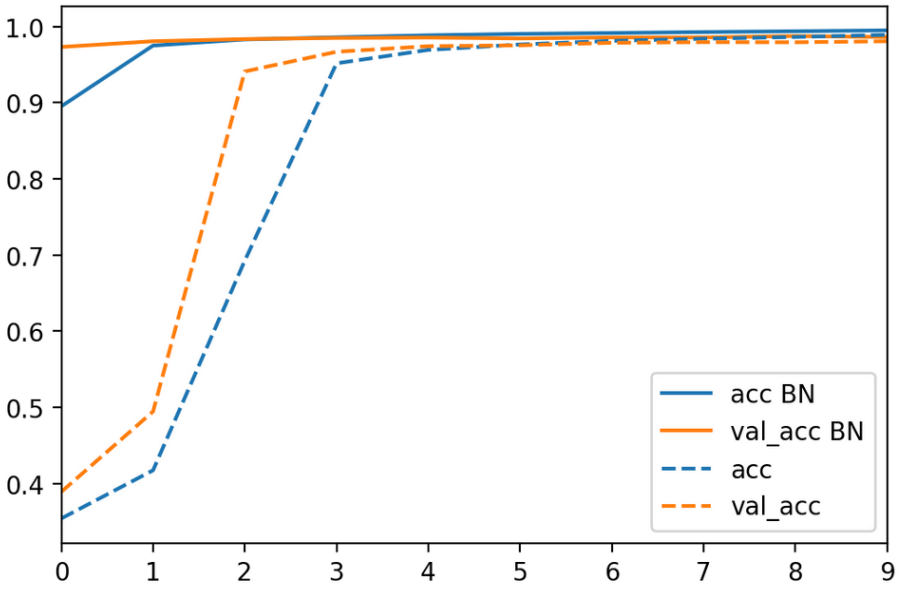
Batch normalization re-normalizes the activations for a layer for each batch during training (as the distribution over activation changes). The avoids saturation when using saturating functions.

To keep the expressive power of the model, additional scale and shift parameters are learned that are applied after the per-batch normalization.

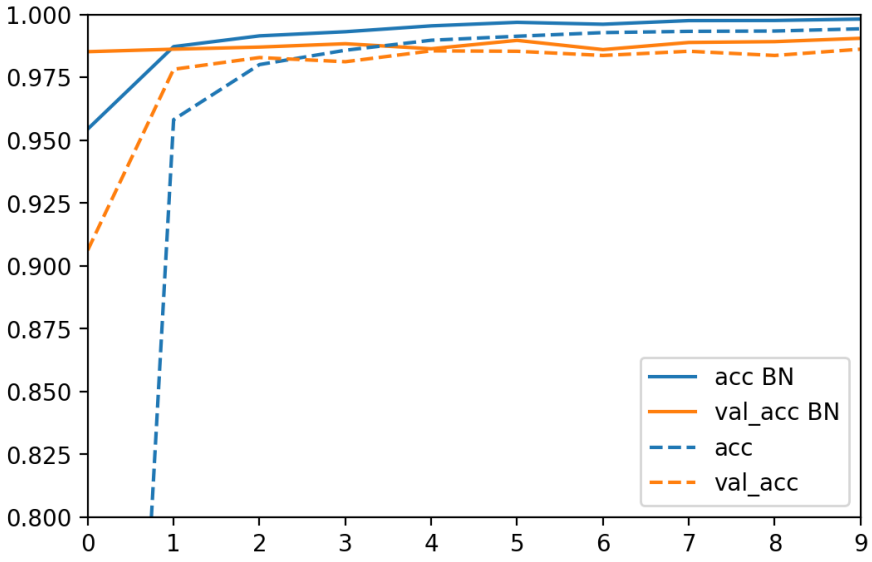
Tiny convnet with Batch Normalization

```
from keras.layers import BatchNormalization

num_classes = 10
cnn_small_bn = Sequential()
cnn_small_bn.add(Conv2D(8, kernel_size=(3, 3),
                 input_shape=input_shape))
cnn_small_bn.add(Activation("relu"))
cnn_small_bn.add(BatchNormalization())
cnn_small_bn.add(MaxPooling2D(pool_size=(2, 2)))
cnn_small_bn.add(Conv2D(8, (3, 3)))
cnn_small_bn.add(Activation("relu"))
cnn_small_bn.add(BatchNormalization())
cnn_small_bn.add(MaxPooling2D(pool_size=(2, 2)))
cnn_small_bn.add(Flatten())
cnn_small_bn.add(Dense(64, activation='relu'))
cnn_small_bn.add(Dense(num_classes, activation='softmax'))
```

Learning speed (and accuracy)



For larger (64 filters) net



VGG and Imagenet Filters

Inspecting VGG

```
from keras import applications

# build the VGG16 network
model = applications.VGG16(include_top=False,
                           weights='imagenet')
model.summary()
```

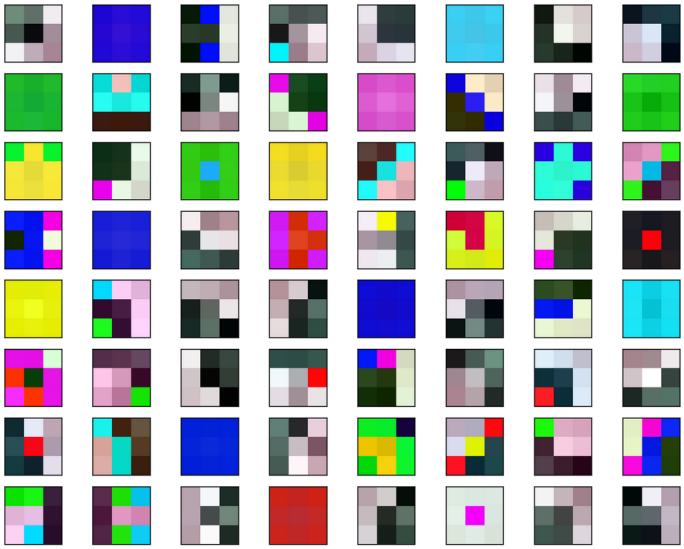
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, None, None, 3)	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1792
block1_conv2 (Conv2D)	(None, None, None, 64)	36928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73856
block2_conv2 (Conv2D)	(None, None, None, 128)	147584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295168
block3_conv2 (Conv2D)	(None, None, None, 256)	590080
block3_conv3 (Conv2D)	(None, None, None, 256)	590080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1 (Conv2D)	(None, None, None, 512)	1180160
block4_conv2 (Conv2D)	(None, None, None, 512)	2359808
block4_conv3 (Conv2D)	(None, None, None, 512)	2359808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2359808
block5_conv2 (Conv2D)	(None, None, None, 512)	2359808
block5_conv3 (Conv2D)	(None, None, None, 512)	2359808
block5_pool (MaxPooling2D)	(None, None, None, 512)	0

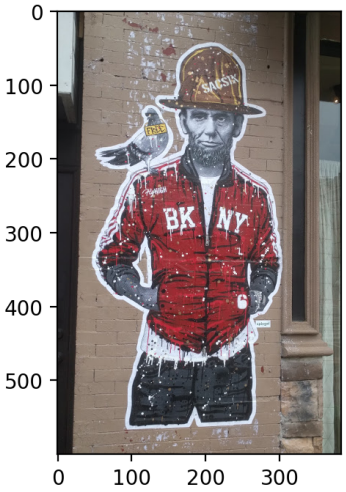
Total params: 14,714,688.0
Trainable params: 14,714,688.0
Non-trainable params: 0.0

VGG filters

```
vgg_weights, vgg_biases = model.layers[1].get_weights()
vgg_weights.shape
```

```
(3, 3, 3, 64)
```





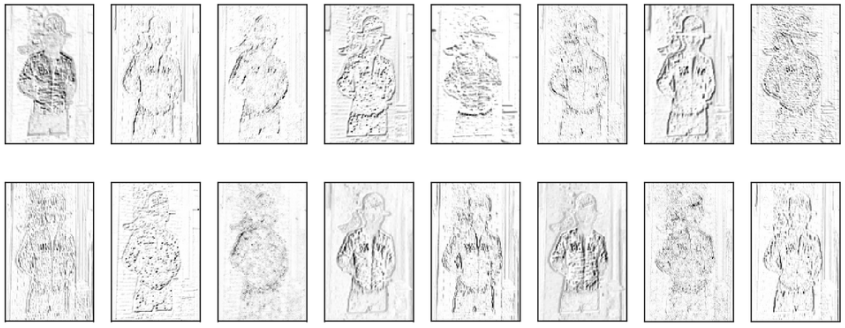

```
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
get_6rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[6].output])

layer3_output = get_3rd_layer_output([[image]])[0]
layer6_output = get_6rd_layer_output([[image]])[0]
```

```
print(layer3_output.shape)
print(layer6_output.shape)
```

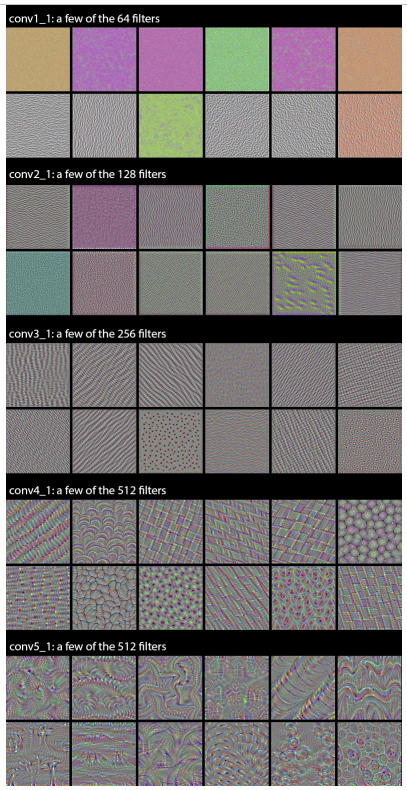
```
(1, 300, 192, 64)
(1, 150, 96, 128)
```

after first pooling layer

after second pooling layer

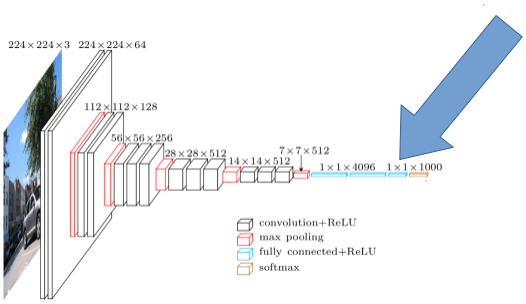


https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html



Using pre-trained networks

- Train on “large enough” data.
- Apply to new “small” dataset.
- Take activations of last or second to last fully connected layer.



See http://cs231n.github.io/transfer-learning/

Often we have a small but specific image dataset for a particular application. Training a neural net is not feasible unless we have tens of thousands or hundreds of thousands of images.

However, if we have a convolutional neural net that was already trained on a large dataset that is similar enough, we can hope that the features it learned are also helpful for our task.

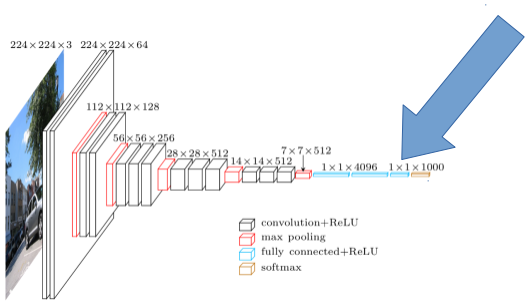
The easiest way to adapt a trained network to a new task is to just apply it to our dataset and take the activations of the second to last or last layer.

If the original task was rich enough – say 10000 different classes as in imagenet – these layers contain a lot of information about the image.

We can then use these activations as features for another classifier like a linear model or smaller dense neural network.

Using pre-trained networks

- Train on “large enough” data.
- Apply to new “small” dataset.
- Take activations of last or second to last fully connected layer.



The main point is that we don’t need to retrain all the weights in the network. You can think of it as retraining only the last layer – the classification layer – of the network, while holding all the convolutional filters fixed. If they learned generic patterns like edges and patterns, these will still be useful for your task.

You can download pre-trained neural networks for many architectures online.

Using a pre-trained network is sometimes also known as transfer learning.

This potentially doesn’t work with images from a very different domain, like medical images.

Ball snake vs Carpet Python

```
import flickrapi
flickr = flickrapi.FlickrAPI(api_key, api_secret, format='json')
```

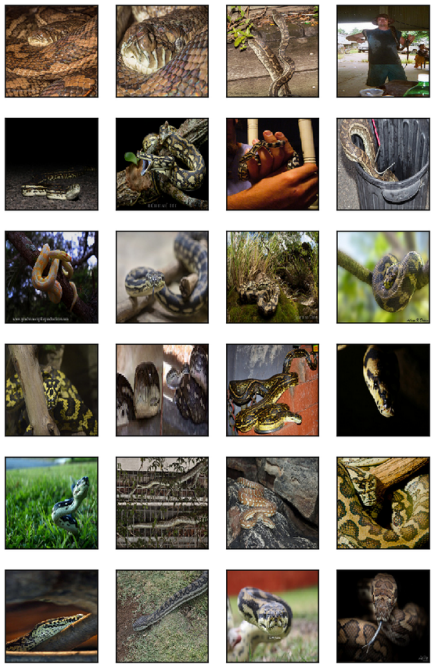
```
def search_ids(search_string="python", per_page=10):
    photos_response = flickr.photos.search(text=search_string, per_page=per_page, sort='relevance')
    photos = json.loads(photos_response.decode('utf-8'))['photos']['photo']
    ids = [photo['id'] for photo in photos]
    return ids
```

```
def get_url(photo_id="33510015330"):
    response = flickr.photos.getsizes(photo_id=photo_id)
    sizes = json.loads(response.decode('utf-8'))['sizes']['size']
    for size in sizes:
        if size['label'] == "Small":
            return size['source']
```

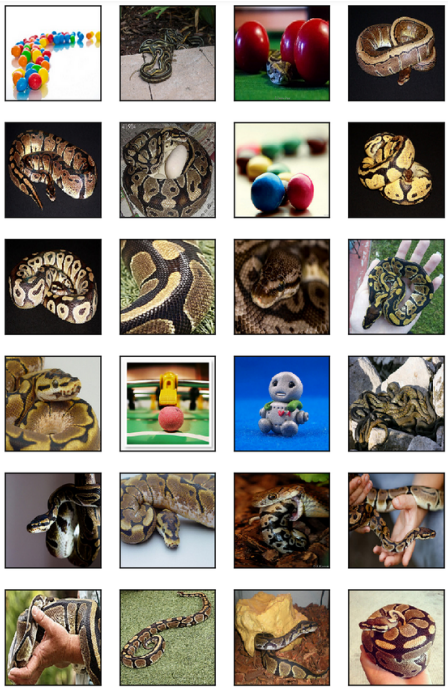
```
ids = search_ids("ball snake", per_page=100)
urls_ball = [get_url(photo_id=i) for i in ids]
```

```
from urllib.request import urlretrieve
import os
for url in urls_carpet:
    urlretrieve(url, os.path.join("snakes", "carpet", os.path.basename(url)))
```

Carpet Python (100 total)



Ball Snake (100 total)



Extracting Features using VGG

```
from keras.preprocessing import image

images_carpet = [image.load_img(os.path.join("snakes", "carpet", os.path.basename(url)), target_size=(224, 224))
                 for url in urls_carpet]
images_ball = [image.load_img(os.path.join("snakes", "ball", os.path.basename(url)), target_size=(224, 224))
                 for url in urls_ball]
X = np.array([image.img_to_array(img) for img in images_carpet + images_ball])
```

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

```
X.shape
```

```
(200, 224, 224, 3)
```

```
from keras.applications.vgg16 import preprocess_input
X_pre = preprocess_input(X)
features = model.predict(X_pre)
```

```
features.shape
```

```
(200, 7, 7, 512)
```

```
features_ = features.reshape(200, -1)
```

Classification with Logreg

```
from sklearn.model_selection import train_test_split
y = np.zeros(200, dtype='int')
y[100:] = 1
X_train, X_test, y_train, y_test = train_test_split(features_, y, stratify=y)
```

```
from sklearn.linear_model import LogisticRegressionCV
lr = LogisticRegressionCV().fit(X_train, y_train)
```

```
print(lr.score(X_train, y_train))
```

```
1.0
```

```
print(lr.score(X_test, y_test))
```

```
0.82
```

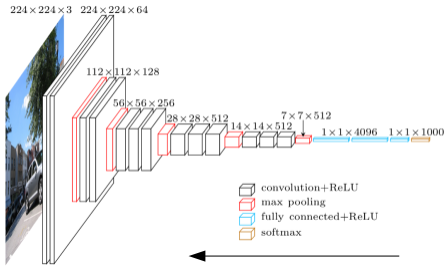
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, lr.predict(X_test))
```

```
array([[24,  1],
       [ 8, 17]])
```

Caveat: I haven't really checked if these make sense.

Finetuning

- Start with pre-trained net
- Back-propagate error through all layers
- “tune” filters to new data.



See https://keras.io/applications/

A more complicated variant of this is to load a network trained on some other dataset, and replace the last layer with your classification task.

Instead of training only the last layer, we can also keep training all the previous layers, backpropagating the gradient through the network and adjusting the previously learned filters for our task.

You can think of this as warm-starting a neural network from one that was trained on another dataset.

If you do that, we often want to train the last layer a little bit before we backpropagate through the network. Otherwise the random initialization of the last layer might destroy the filters that we used for initialization.

Adverserial Samples

(a) (b)

Right column: correctly classified image, left column classified as Ostrich.
Center: difference.

Intriguing properties of neural networks https://arxiv.org/abs/1312.6199

Since convolutional neural nets are so good at image recognition, some people think they are pretty infallible. But they are not. There is this interesting paper about intriguing properties of neural networks, that introduces adverserial samples.

Adverserial samples are samples that were created by an adversary or attacker to fool your model. Here, they changed images to be classified as Ostrich by AlexNet trained on imagenet.

The picture on the left is change just slightly, and went from correctly classified to classified as Ostrich.

This technique uses gradient descent on the input and requires access to all the weights in the network to create the samples.

Given how high-dimensional the input space is, this is not very surprising from a mathematical perspective, but it might be somewhat unexpected.