Neural Networks

Andreas Müller

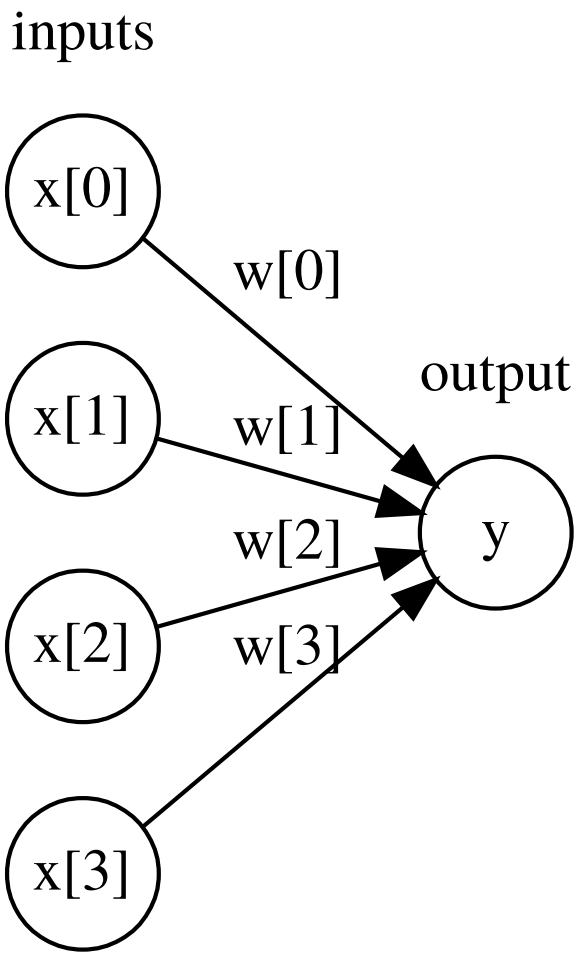
History

- Nearly everything we talk about today existed ~1990
- What changed?
  - More data
  - Faster computers (GPUs)
  - Some improvements:
    - relu
    - Drop-out
    - adam
    - Batch-normalization
    - Residual networks

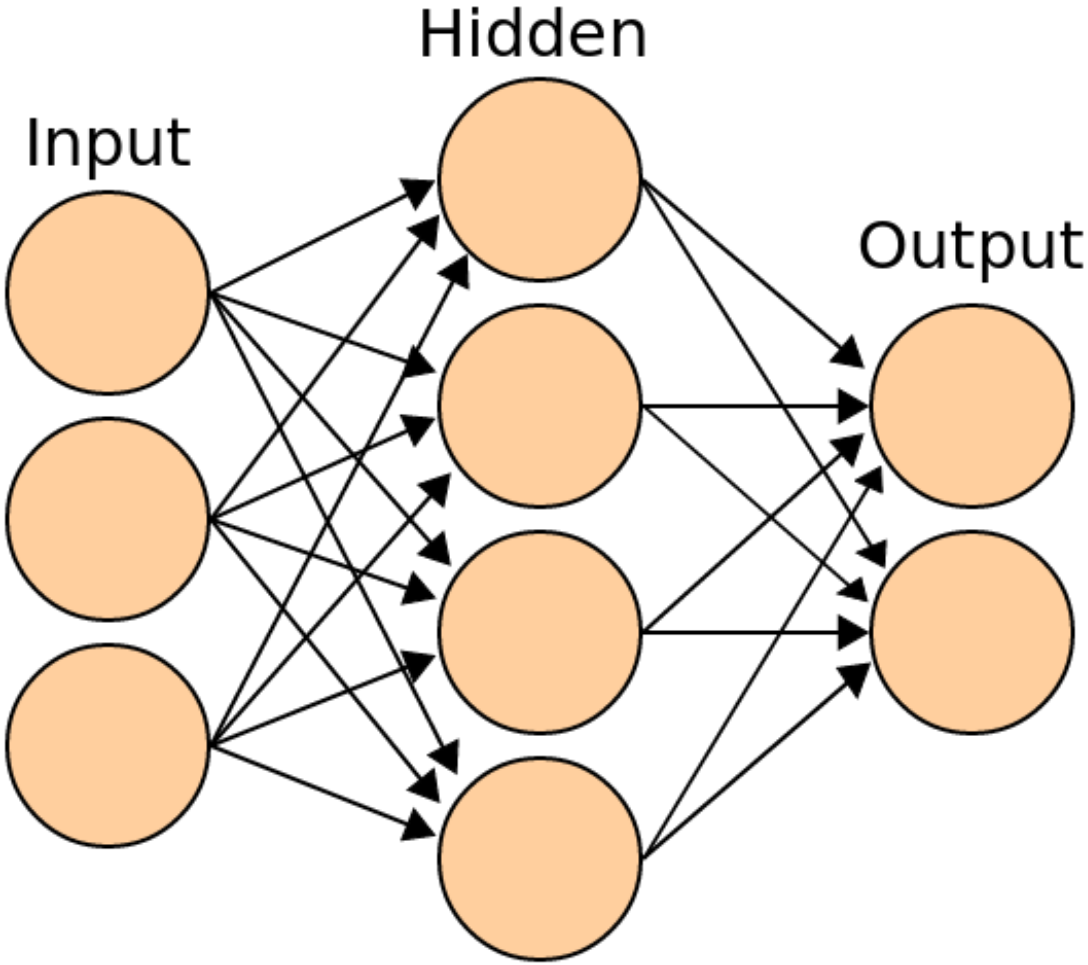
Supervised Neural Networks

- Non-linear models for classification and regression
- Work well for very large datasets
- Non-convex optimization
- Notoriously slow to train – need for GPUs
- Use dot products etc → require preprocessing, similar to SVM or linear models, unlike trees
- MANY variants (Convolutional nets, Gated Recurrent neural networks, Long-Short-Term Memory, recursive neural networks, variational autoencoders, general adverserial networks, neural turing machines...)

Logistic regression drawn as neural net

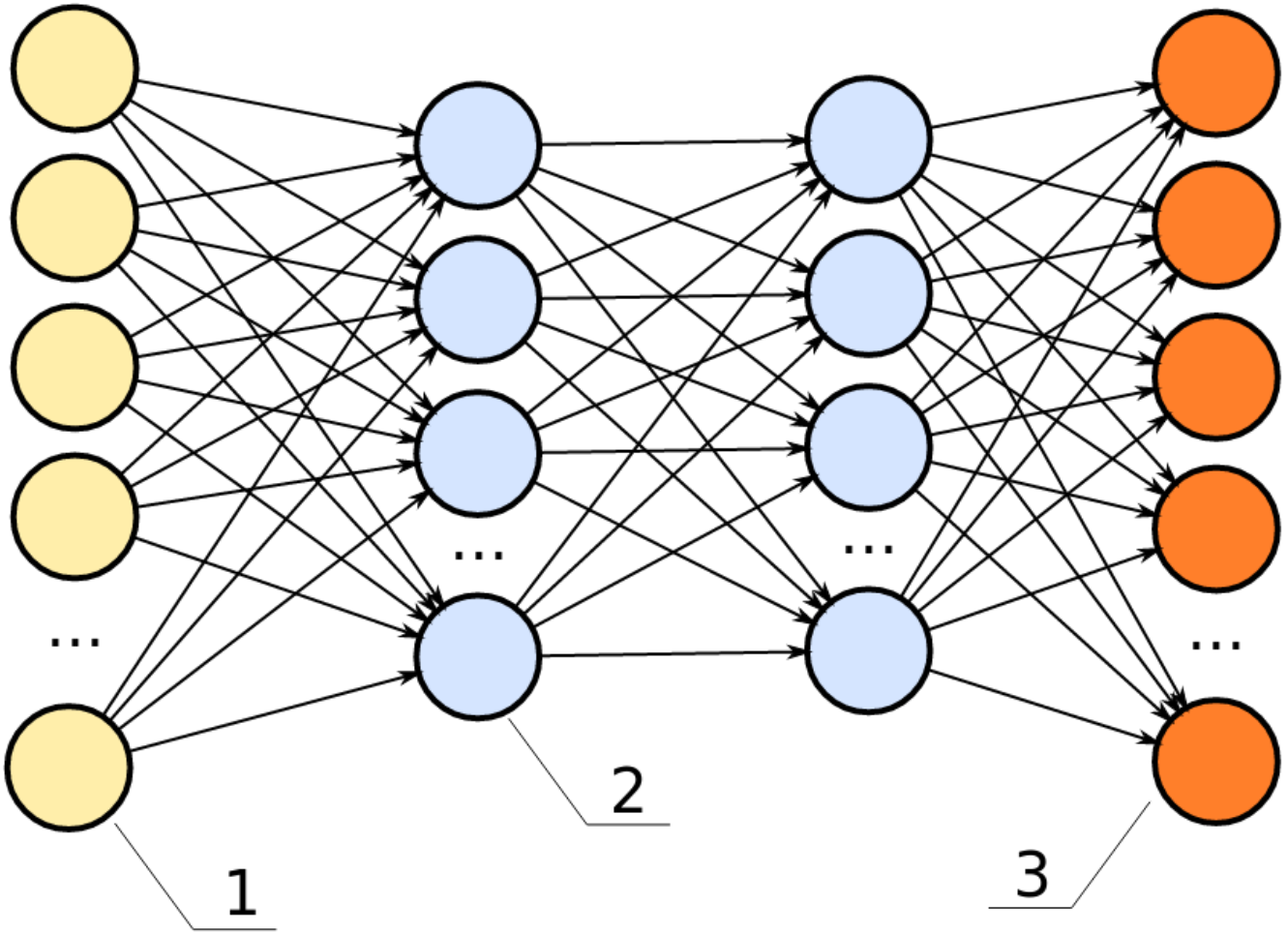


Basic Architecture (for making predictions)

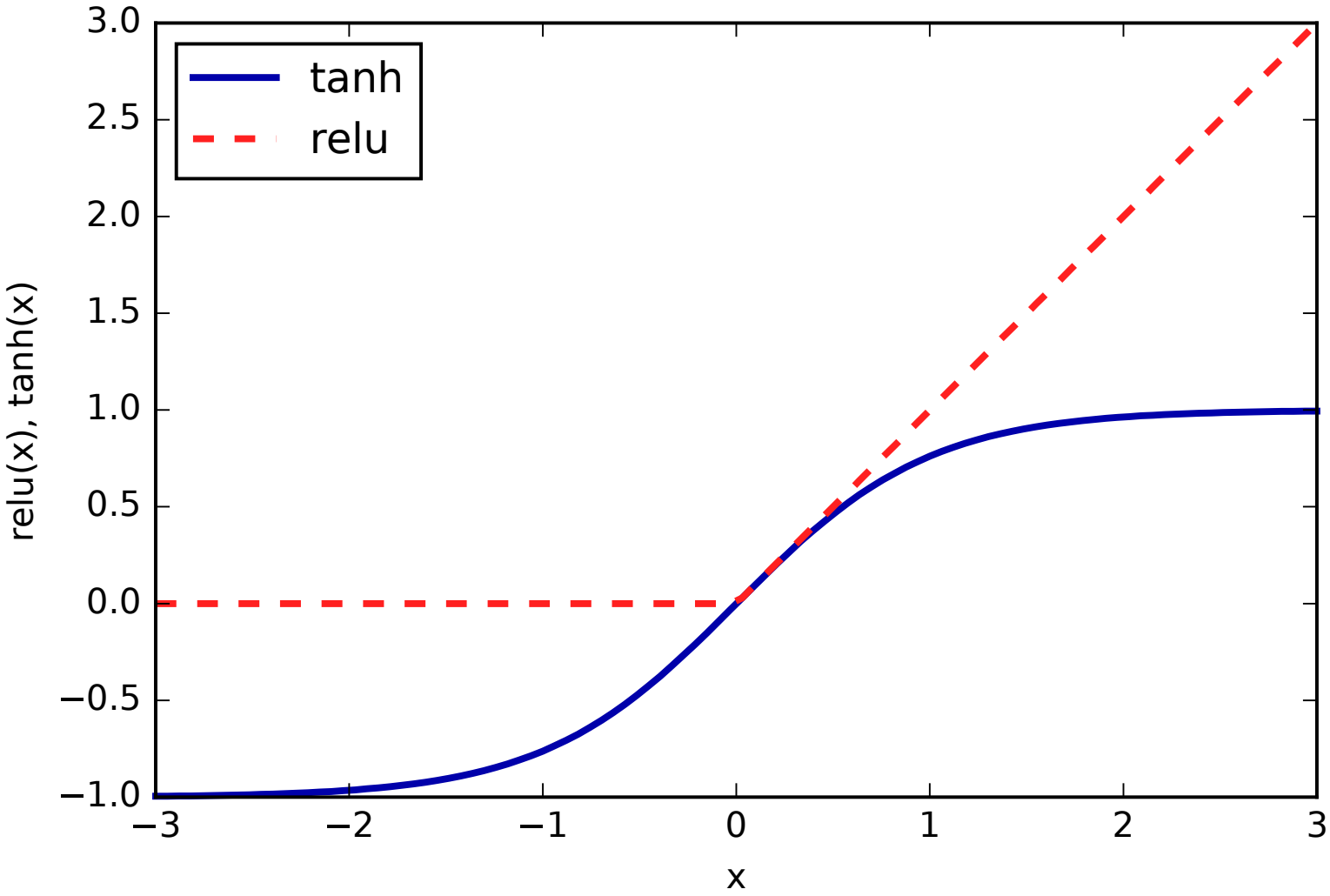


x $\quad$ $h(x) = f(Wx + b)$ $\quad$ $o(x) = g(W'h(x) + b')$

Can have arbitrary many layers



Nonlinear activation function



Training objective

$$h(\mathbf{x}) = f(W_1\mathbf{x} + \mathbf{b}_1)$$

$$o(\mathbf{x}) = g(W_2 h(\mathbf{x}) + \mathbf{b}_2) = g(W_2 f(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

$$\min_{W_1, W_2, \mathbf{b}_1, \mathbf{b}_2} \sum_{i=1}^{N} \ell(y_i, o(\mathbf{x}_i))$$

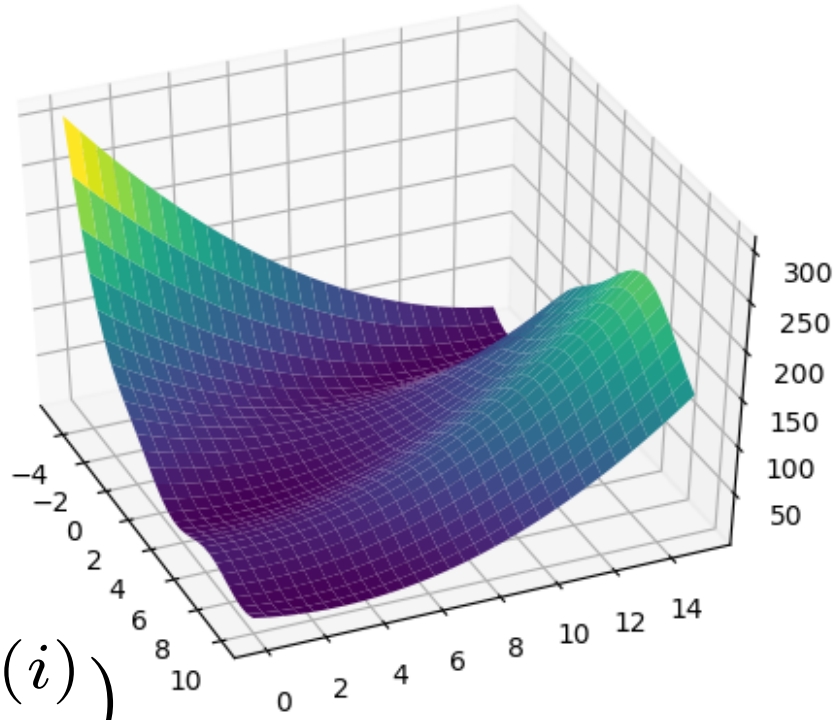
Could add regularization

$$= \min_{W_1, W_2, \mathbf{b}_1, \mathbf{b}_2} \sum_{i=1}^{N} \ell(y_i, g(W_2 f(W_1\mathbf{x}_i + \mathbf{b}_1) + \mathbf{b}_2))$$

ℓ squared loss for regression
cross-entropy loss (multi-class log-loss) for classification

Reminder: Gradient Descent

Want: $\arg\min_w F(w)$

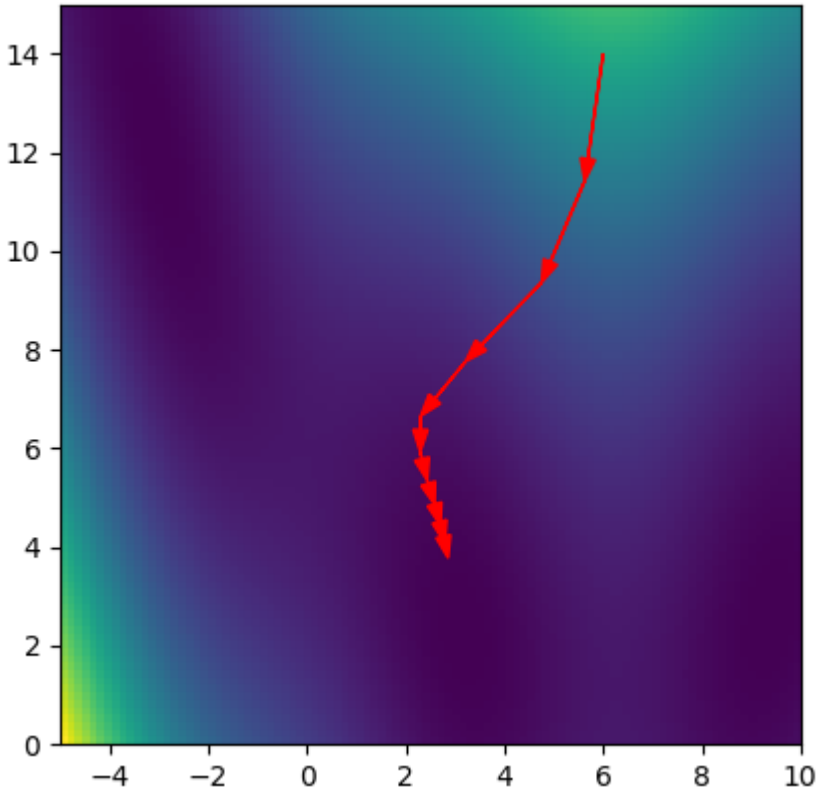


Initialize w_0

$$w^{(i+1)} \leftarrow w^{(i)} - \eta_i \frac{d}{dw} F(w^{(i)})$$

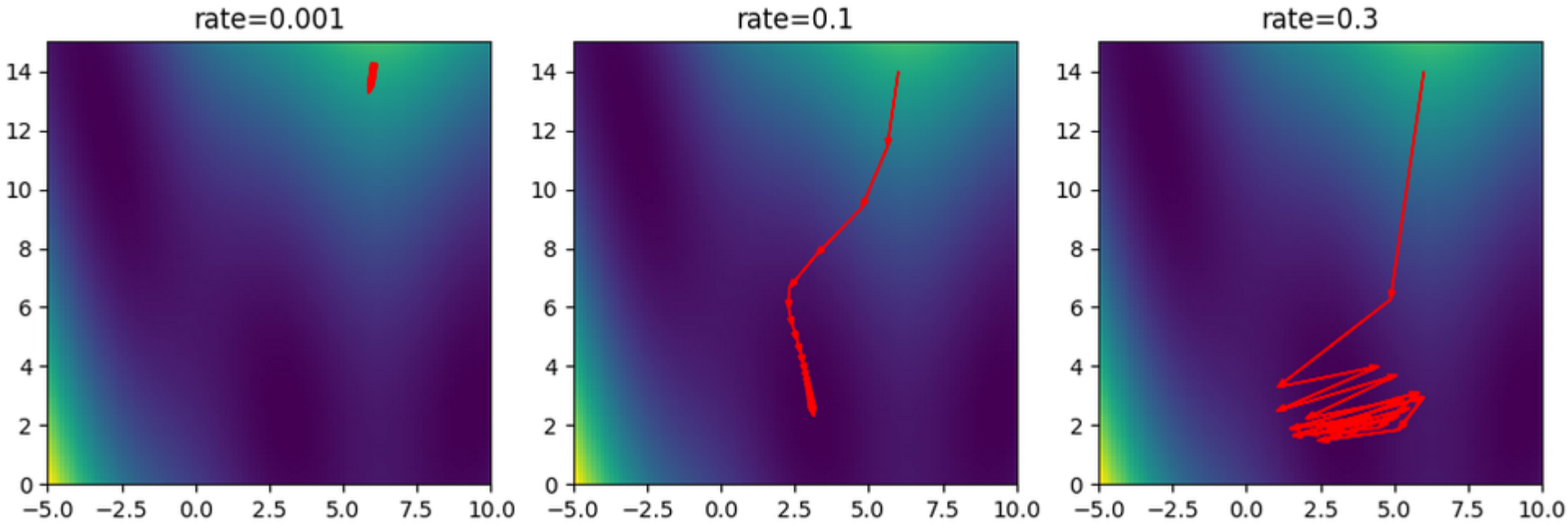
Converges to local minimum

Reminder: Gradient Descent



$$w^{(i+1)} \leftarrow w^{(i)} - \eta_i \frac{d}{dw} F(w^{(i)})$$

Picking a learning rate



$$w^{(i+1)} \leftarrow w^{(i)} - \eta_i \frac{d}{dw} F(w^{(i)})$$

Backpropagation

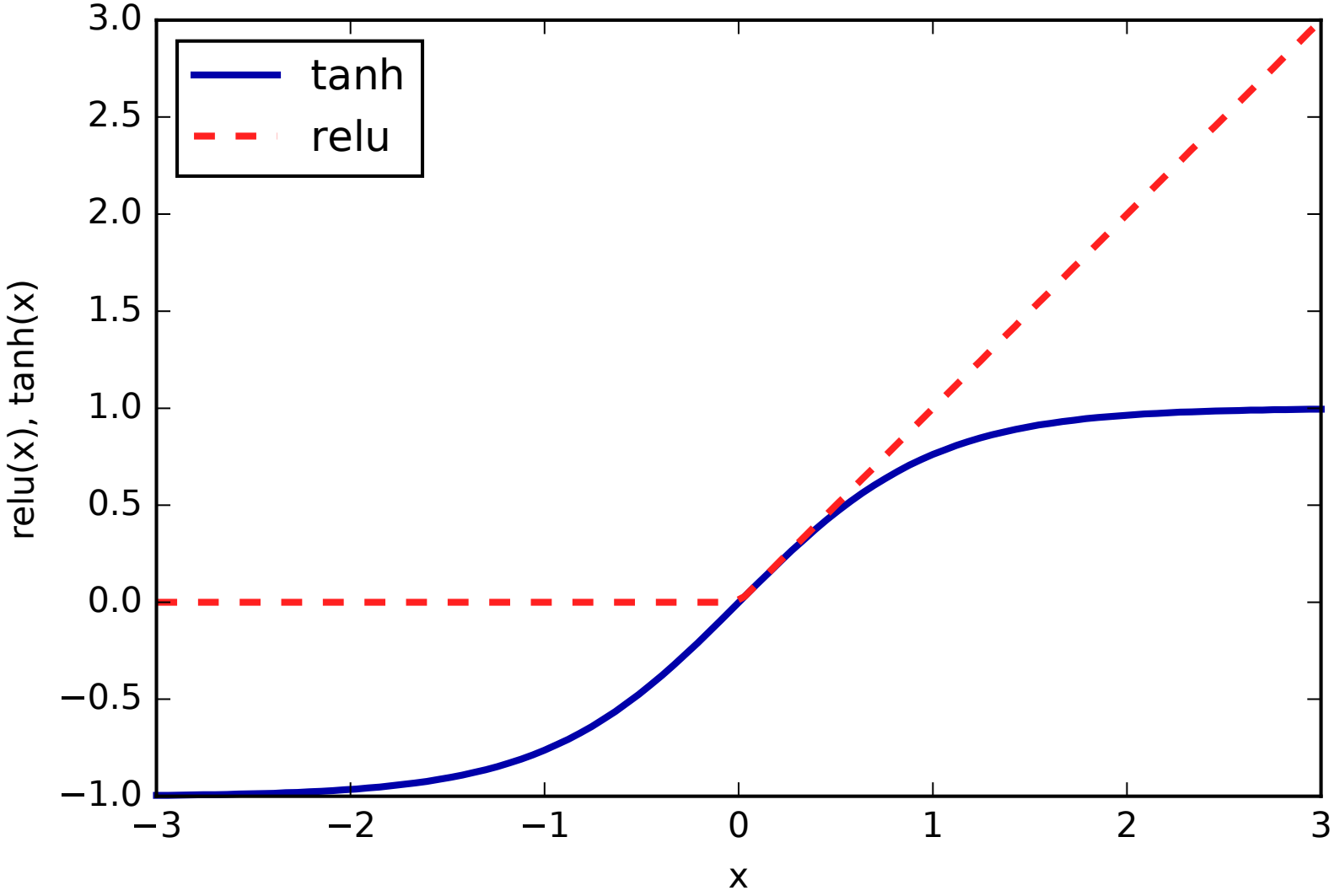
- For gradient based-method need $\frac{\partial o(\mathbf{x})}{\partial W_i}$ $\frac{\partial o(\mathbf{x})}{\partial \mathbf{b}_i}$

$$\text{net}(\mathbf{x}) := W_1\mathbf{x} + b_1$$

$$\frac{\partial o(\mathbf{x})}{\partial W_1} = \underbrace{\frac{\partial o(\mathbf{x})}{\partial h(\mathbf{x})}}_{\text{backpropagation of gradient of layer above.}} \underbrace{\frac{\partial h(\mathbf{x})}{\partial \text{net}(\mathbf{x})}}_{\text{Gradient of Non-linearity f}} \underbrace{\frac{\partial \text{net}(\mathbf{x})}{\partial W_1}}_{\text{Input to 1st layer x}}$$

Backpropagation = Chain Rule + Dynamic Programming

But wait!



Optimizing W, b

$$W_i \leftarrow W_i - \eta \sum_{j=1}^{n} \frac{\ell(\mathbf{x}_j, y_j)}{W_i}$$ batch

$$W_i \leftarrow W_i - \eta \sum_{j=k}^{k+m} \frac{\ell(\mathbf{x}_j, y_j)}{W_i}$$ minibatch

$$W_i \leftarrow W_i - \eta \frac{\ell(\mathbf{x}_j, y_j)}{W_i}$$ Online / stochastic

Learning Heuristics

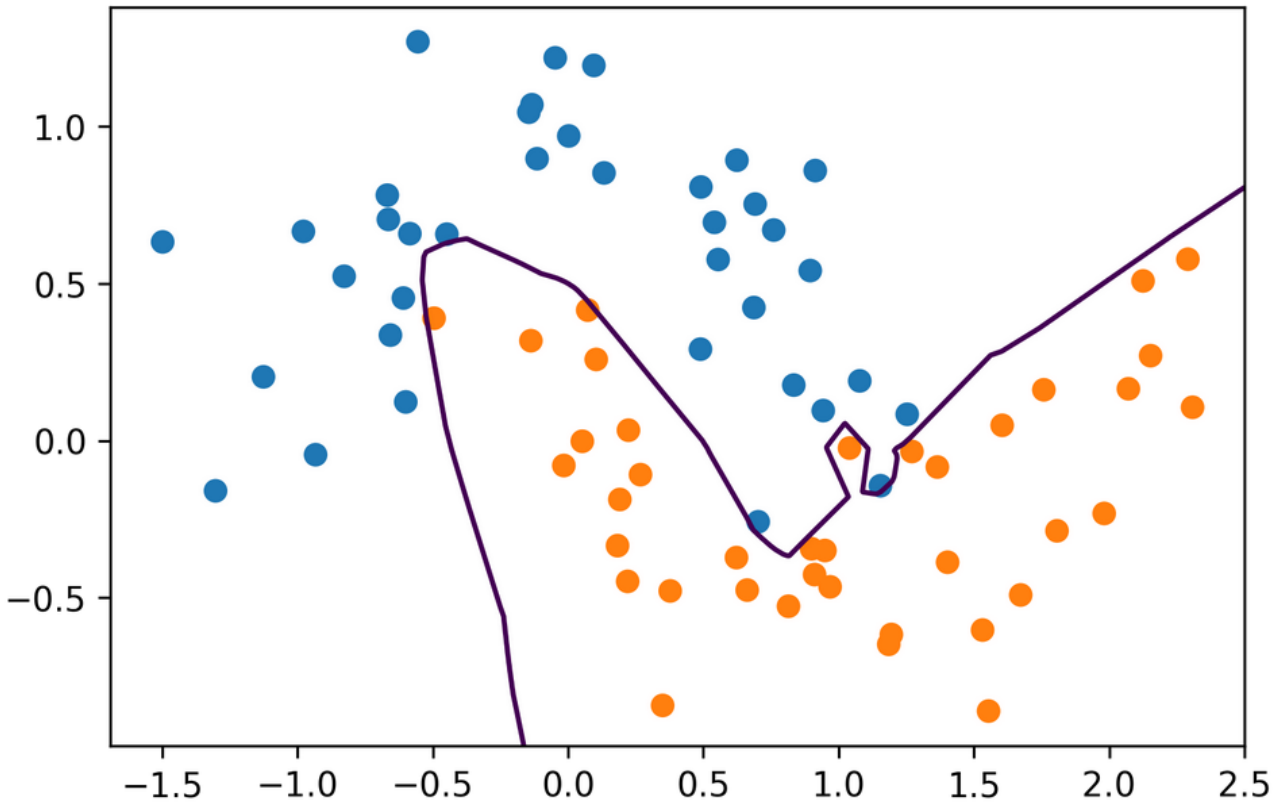
- Constant η not good
- Can decrease η
- Better: adaptive η for each entry if Wi
- State-of-the-art: adam (with magic numbers)

https://arxiv.org/pdf/1412.6980.pd
f
http://sebastianruder.com/optimizing-gradient-descent/

Picking optimization algorithms

- Small dataset: off the shelf like l-bfgs
- Big dataset: adam
- Have time & nerve: tune the schedule

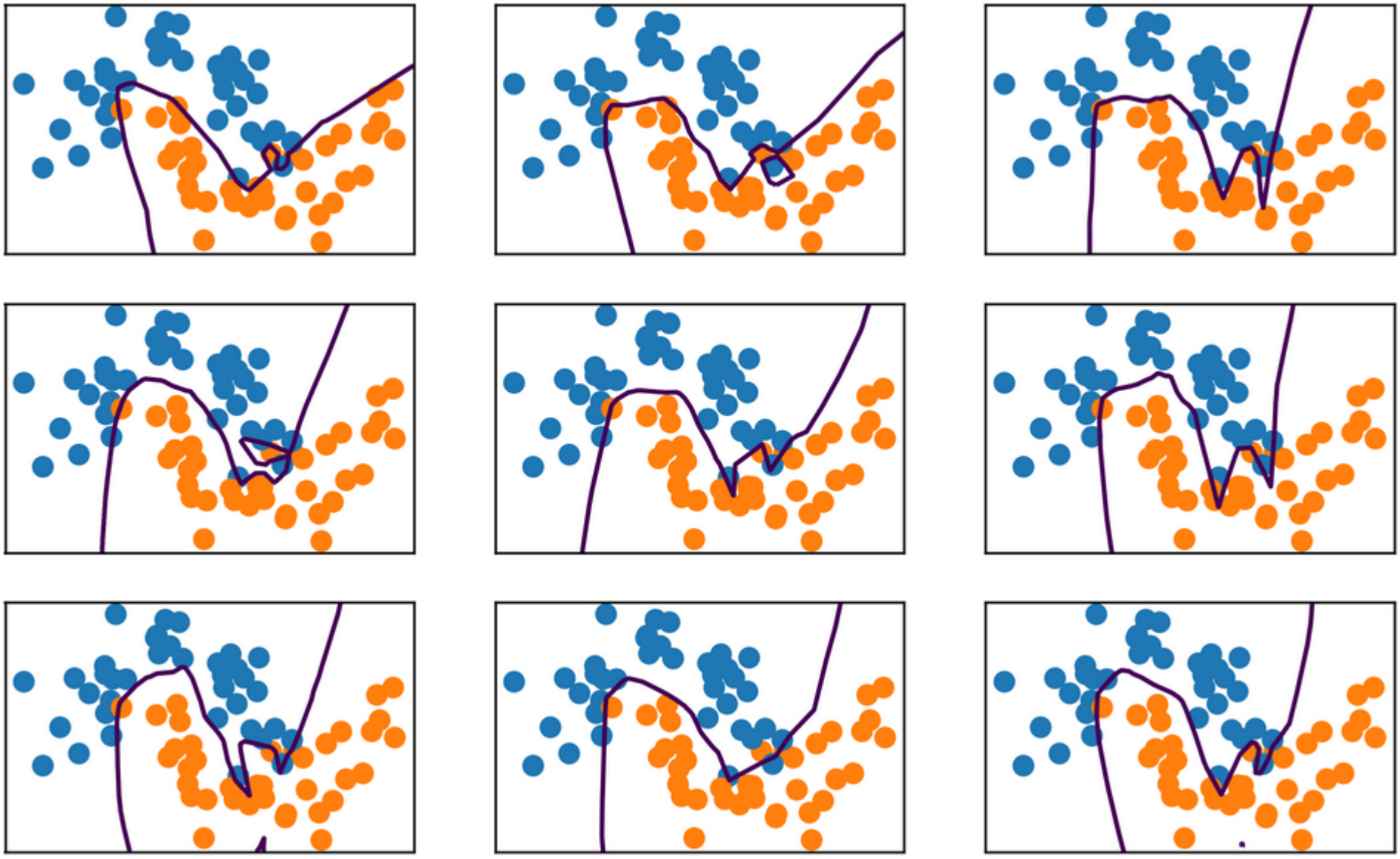
Neural Nets with sklearn




```
mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

```
1.0
0.88
```

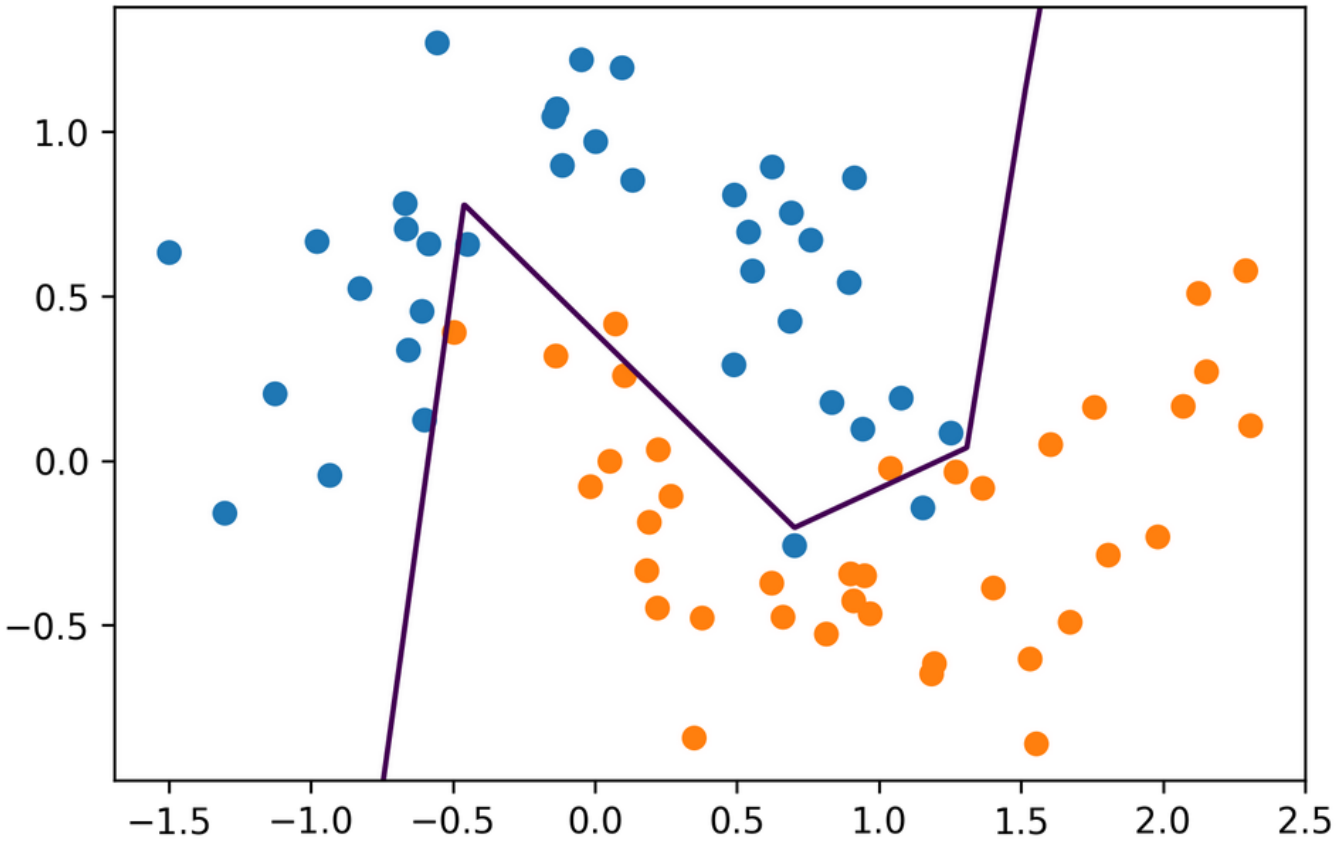
Random State



Hidden Layer Size

```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(5,), random_state=10)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

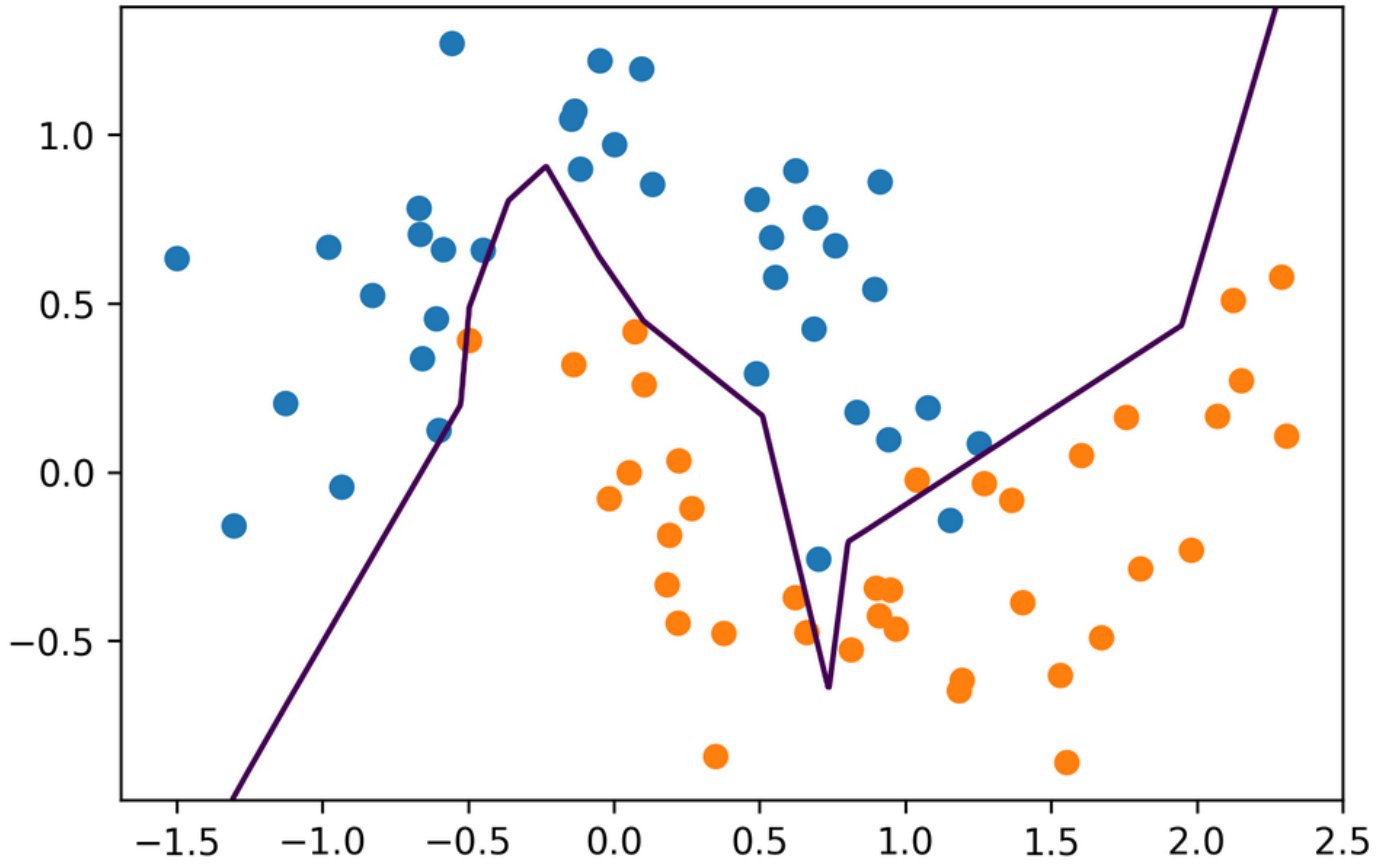
```
0.933333333333
0.92
```



Hidden Layer Size

```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(10, 10, 10), random_state=0)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

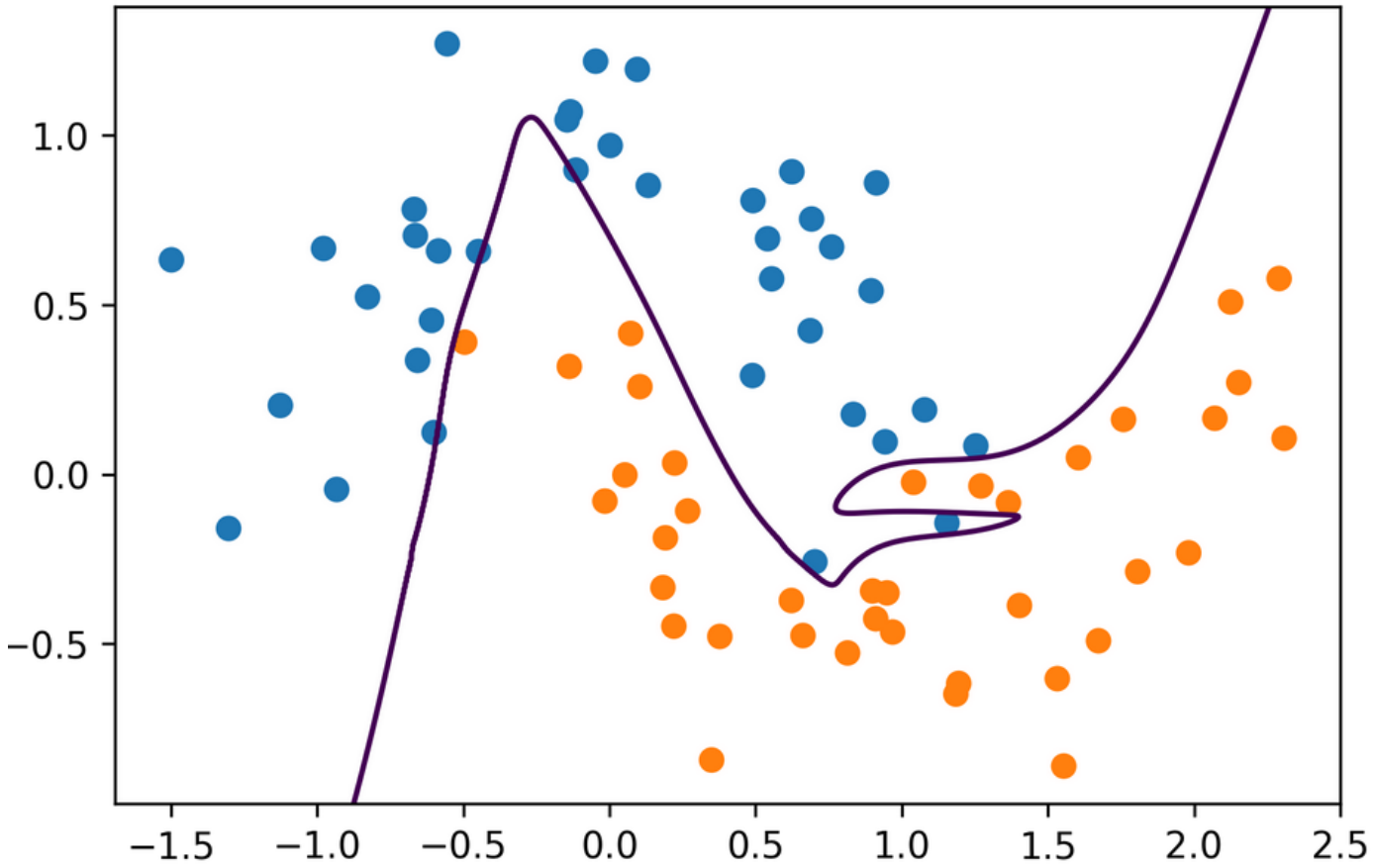
```
0.973333333333
0.84
```



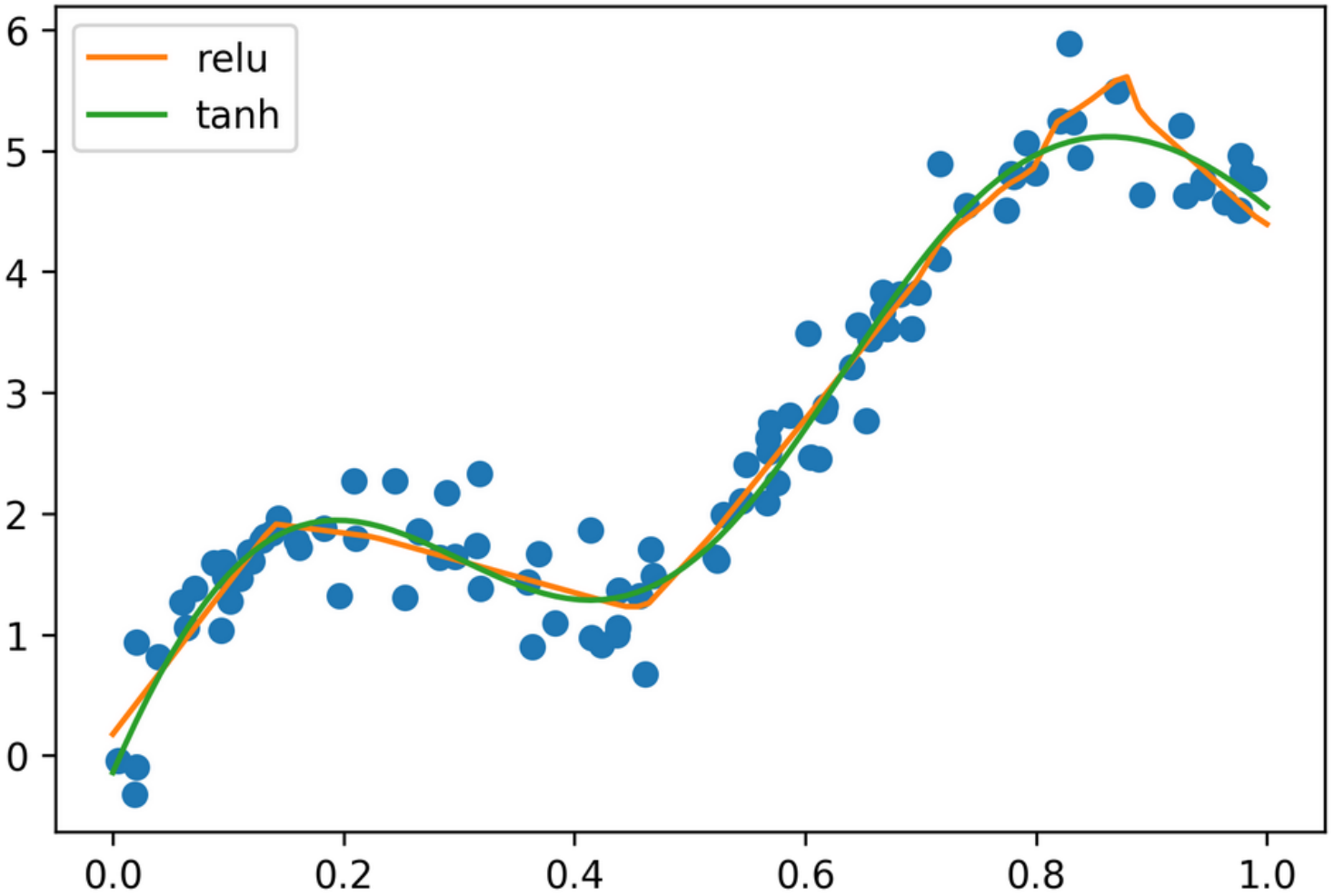
Activation Functions

```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(10, 10, 10), activation="tanh", random_state=0)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

```
1.0
0.92
```



Regression




```
from sklearn.neural_network import MLPRegressor
mlp_relu = MLPRegressor(solver="lbfgs").fit(X, y)
mlp_tanh = MLPRegressor(solver="lbfgs", activation='tanh').fit(X, y)
```

Complexity Control

- Number of parameters
- Regularization
- Early stopping
- (drop-out)

Getting flexible & Scaling up

Deep Learning Libraries

- tf.learn (Tensorflow)
- Keras (Tensorflow, Theano)
- Lasagna (Theano)
- Torch.nn / PyTorch (torch)
- Chainer (chainer)
- MXNet (MXNet)

- Also see: http://mxnet.io/architecture/program_model.html

Quick look at TensorFlow

- “down to the metal” - don’t use for everyday tasks
- Three steps for learning:
  - Build the computation graph (using array operations and functions etc)
  - Create an Optimizer (gradient descent, adam, …) attached to the graph.
  - Run the actual computation.

```python
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# create graph: model
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# create graph: loss
loss = tf.reduce_mean(tf.square(y - y_data))

# bind optimizer
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# run graph
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))
```

No computation

Allocate variables

All the work / computation

https://www.tensorflow.org/versions/r0.10/get_started/

Introduction to Keras

Keras Sequential

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

```
Using TensorFlow backend.
```

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

```
model = Sequential([
    Dense(32, input_shape=(784,), activation='relu'),
    Dense(10, activation='softmax'),
])
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_165 (Dense)            (None, 32)                25120
_________________________________________________________________
activation_113 (Activation)  (None, 32)                0
_________________________________________________________________
dense_166 (Dense)            (None, 10)                330
_________________________________________________________________
activation_114 (Activation)  (None, 10)                0
=================================================================
Total params: 25,450.0
Trainable params: 25,450.0
Non-trainable params: 0.0
_________________________________________________________________
```

Setting optimizer

compile

```
compile(self, optimizer, loss, metrics=None, sample_weight_mode=None)
```

Configures the learning process.

Arguments

- **optimizer**: str (name of optimizer) or optimizer object. See optimizers.
- **loss**: str (name of objective function) or objective function. See objectives.
- **metrics**: list of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. See metrics.

```
model.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
```

Training the model

```
fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None, validation_split=0.0, validatio
```

Trains the model for a fixed number of epochs.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **batch_size**: integer. Number of samples per gradient update.
- **epochs**: integer, the number of epochs to train the model.
- **verbose**: 0 for no logging to stdout, 1 for progress bar logging, 2 for one log line per epoch.
- **callbacks**: list of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_split**: float (0. < x < 1). Fraction of the data to use as held-out validation data.
- **validation_data**: tuple (x_val, y_val) or tuple (x_val, y_val, val_sample_weights) to be used as held-out validation data. Will override validation_split.
- **shuffle**: boolean or str (for 'batch'). Whether to shuffle the samples at each epoch. 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks.

Preparing MNIST data

```
from keras.datasets import mnist
import keras

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

num_classes = 10
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
60000 train samples
10000 test samples
```

Fit model

```
model.fit(X_train, y_train, batch_size=128, epochs=10, verbose=1)
```

```
Epoch 1/10
60000/60000 [==============================] - 2s - loss: 0.4996 - acc: 0.8642
Epoch 2/10
60000/60000 [==============================] - 2s - loss: 0.2432 - acc: 0.9318
Epoch 3/10
60000/60000 [==============================] - 1s - loss: 0.2004 - acc: 0.9429
Epoch 4/10
60000/60000 [==============================] - 1s - loss: 0.1734 - acc: 0.9500
Epoch 5/10
60000/60000 [==============================] - 1s - loss: 0.1540 - acc: 0.9554
Epoch 6/10
60000/60000 [==============================] - 1s - loss: 0.1393 - acc: 0.9597
Epoch 7/10
60000/60000 [==============================] - 1s - loss: 0.1273 - acc: 0.9627
Epoch 8/10
60000/60000 [==============================] - 1s - loss: 0.1172 - acc: 0.9656
Epoch 9/10
60000/60000 [==============================] - 1s - loss: 0.1088 - acc: 0.9683
Epoch 10/10
60000/60000 [==============================] - 1s - loss: 0.1020 - acc: 0.9703
```

Fit with Validation

```
model.fit(X_train, y_train, batch_size=128, epochs=10, verbose=1, validation_split=.1)
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 2s - loss: 0.5146 - acc: 0.8616 - val_loss: 0.2425 - val_acc: 0.9322
Epoch 2/10
54000/54000 [==============================] - 1s - loss: 0.2618 - acc: 0.9266 - val_loss: 0.1934 - val_acc: 0.9442
Epoch 3/10
54000/54000 [==============================] - 1s - loss: 0.2161 - acc: 0.9397 - val_loss: 0.1717 - val_acc: 0.9537
Epoch 4/10
54000/54000 [==============================] - 1s - loss: 0.1879 - acc: 0.9470 - val_loss: 0.1519 - val_acc: 0.9570
Epoch 5/10
54000/54000 [==============================] - 1s - loss: 0.1676 - acc: 0.9528 - val_loss: 0.1440 - val_acc: 0.9603
Epoch 6/10
54000/54000 [==============================] - 1s - loss: 0.1506 - acc: 0.9566 - val_loss: 0.1296 - val_acc: 0.9638
Epoch 7/10
54000/54000 [==============================] - 1s - loss: 0.1378 - acc: 0.9603 - val_loss: 0.1281 - val_acc: 0.9627
Epoch 8/10
54000/54000 [==============================] - 1s - loss: 0.1268 - acc: 0.9626 - val_loss: 0.1177 - val_acc: 0.9660
Epoch 9/10
54000/54000 [==============================] - 1s - loss: 0.1175 - acc: 0.9659 - val_loss: 0.1159 - val_acc: 0.9657
Epoch 10/10
54000/54000 [==============================] - 1s - loss: 0.1096 - acc: 0.9677 - val loss: 0.1131 - val acc: 0.9662
```

Evaluating on Test Set

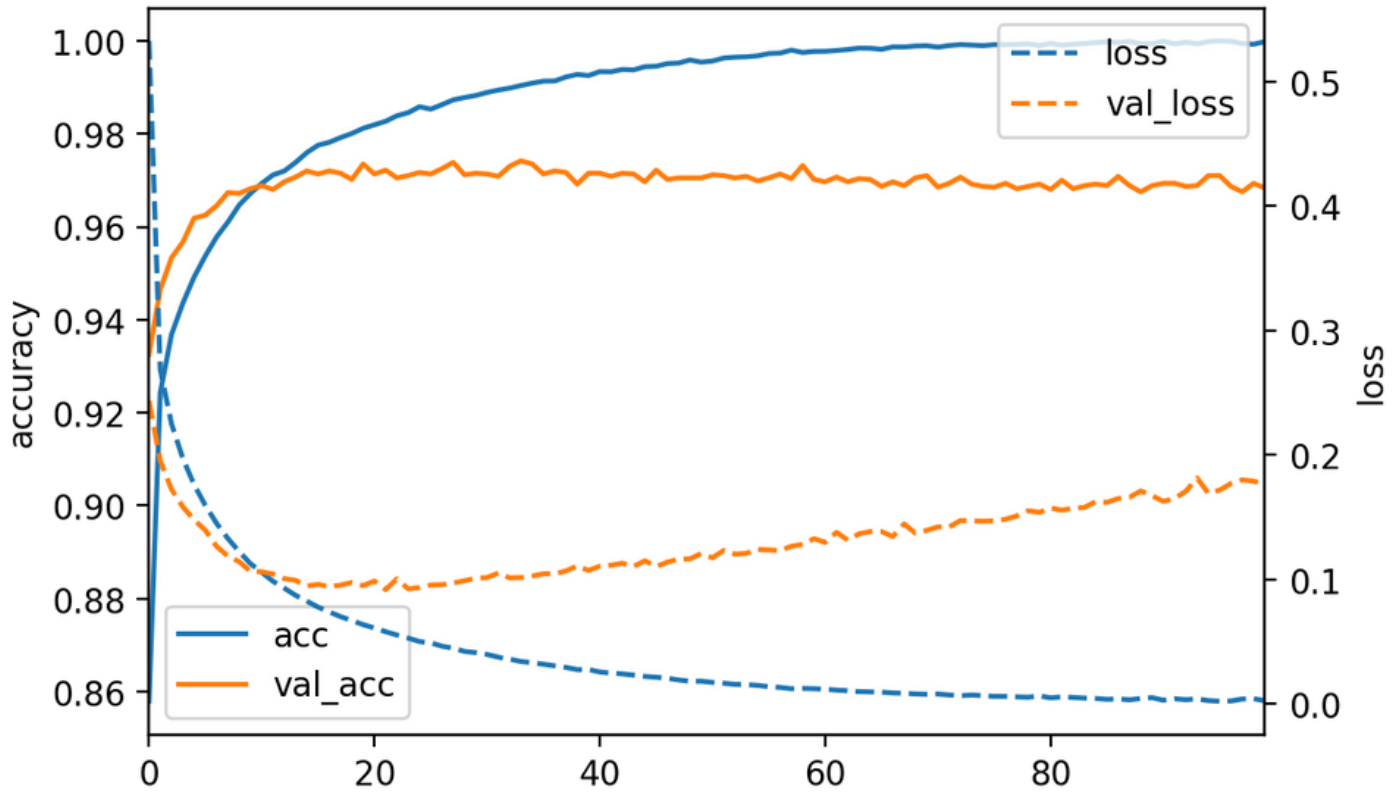
```
score = model.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 0.120
Test Accuracy: 0.966
```

Loggers and Callbacks

```
model = Sequential([
    Dense(32, input_shape=(784,), activation='relu'),
    Dense(10, activation='softmax'),
])
model.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_callback = model.fit(X_train, y_train, batch_size=128,
                             epochs=100, verbose=1, validation_split=.1)
```

```
pd.DataFrame(history_callback.history).plot()
```



Convolutional neural networks

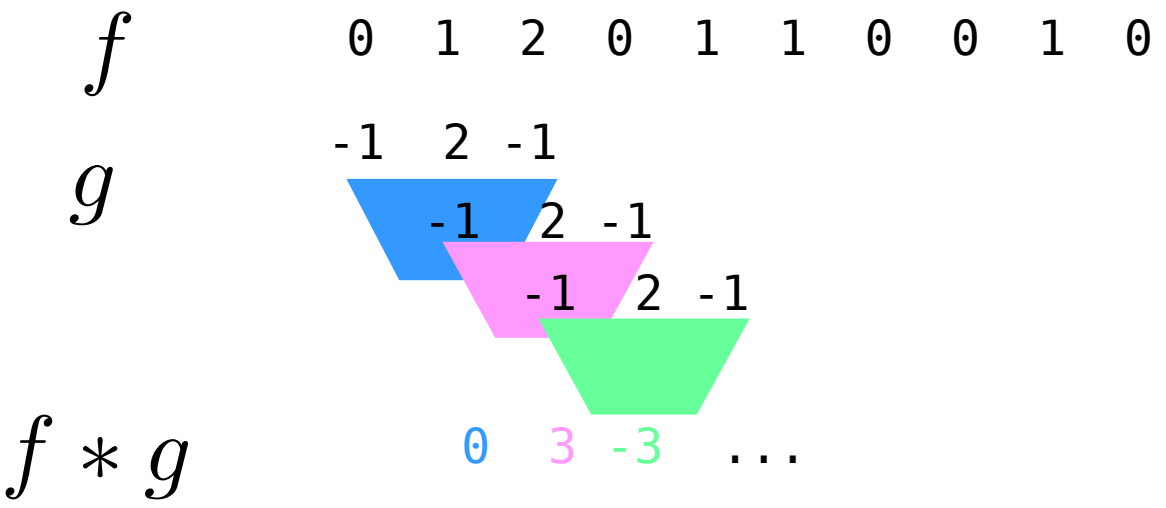
Idea

- Translation invariance
- Weight sharing

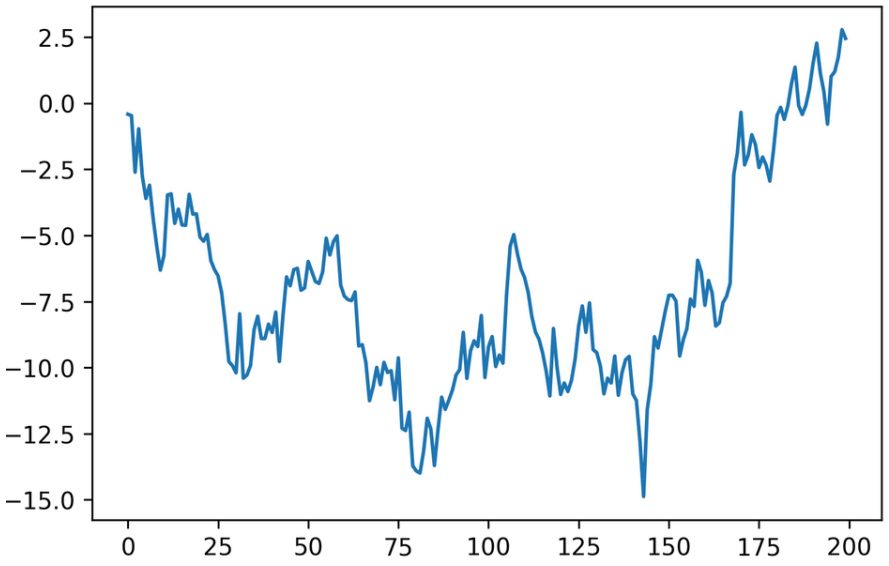
Definition of Convolution

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[n-m]$$

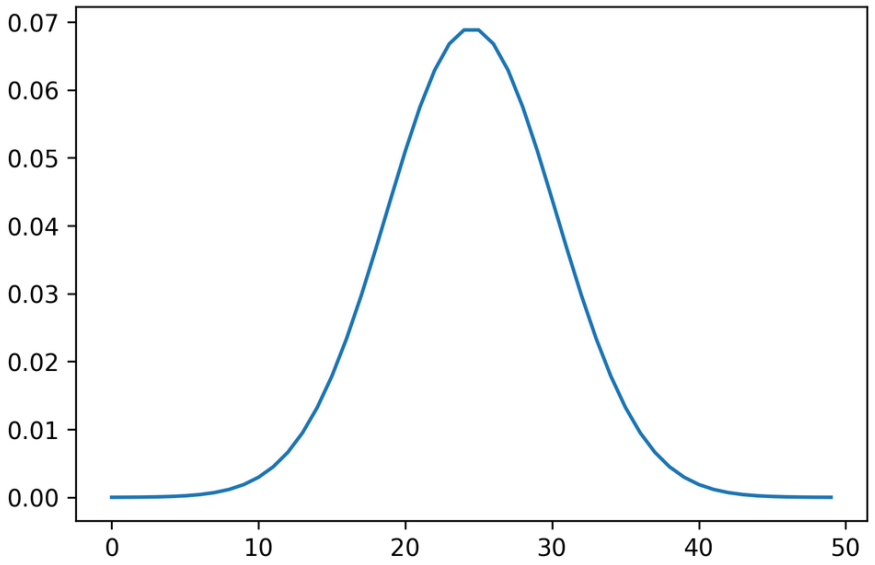
$$= \sum_{m=-\infty}^{\infty} f[n-m]\, g[m].$$



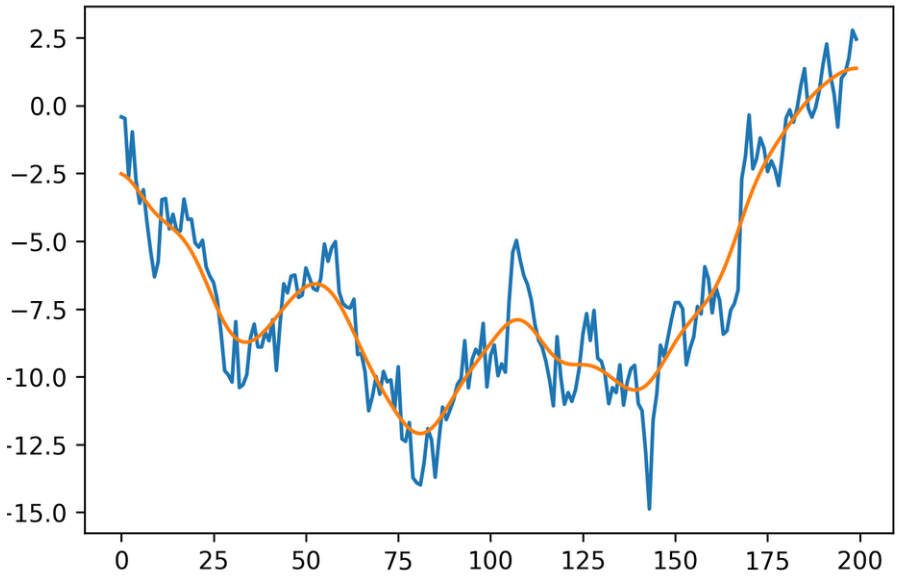
1d example: Gaussian smoothing



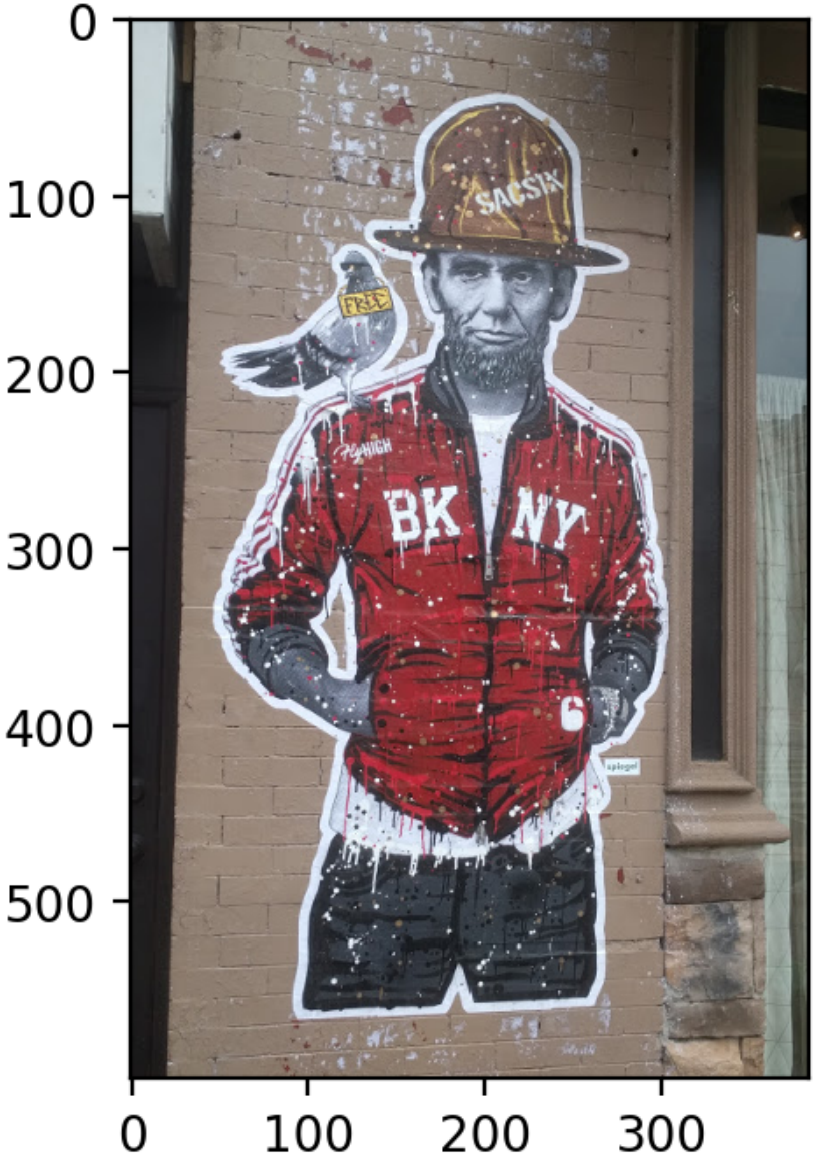
$*$

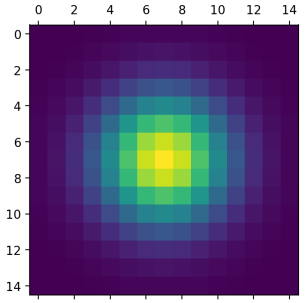


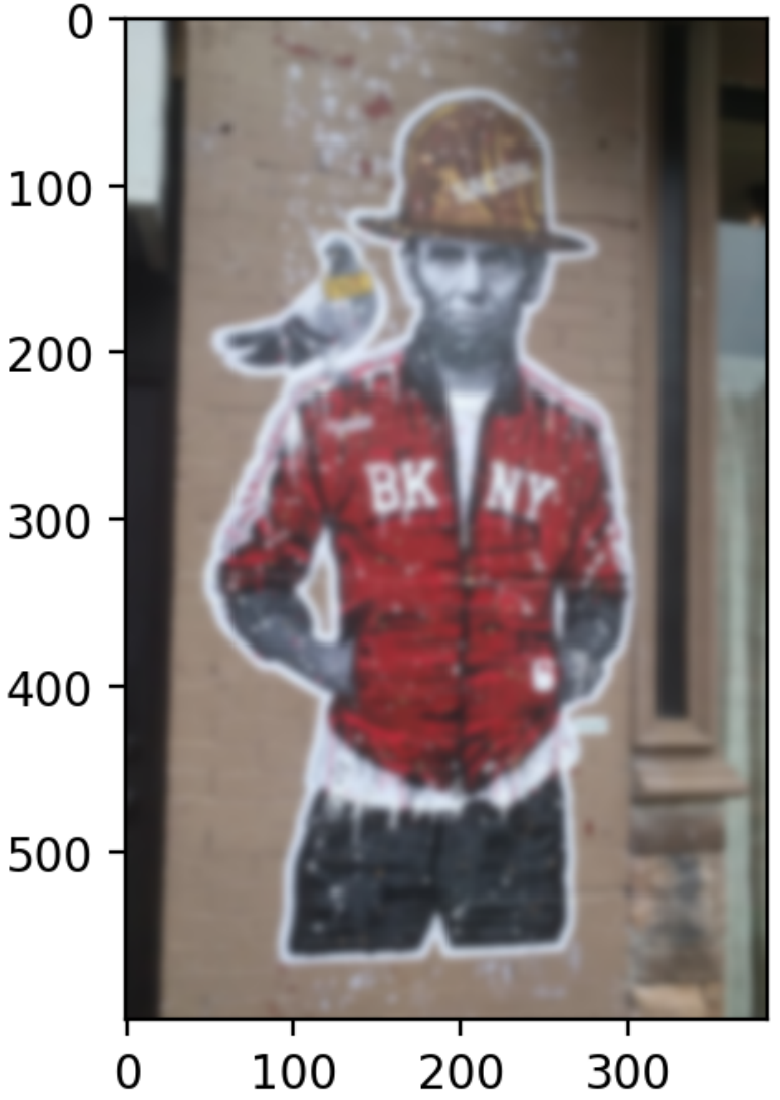
$=$



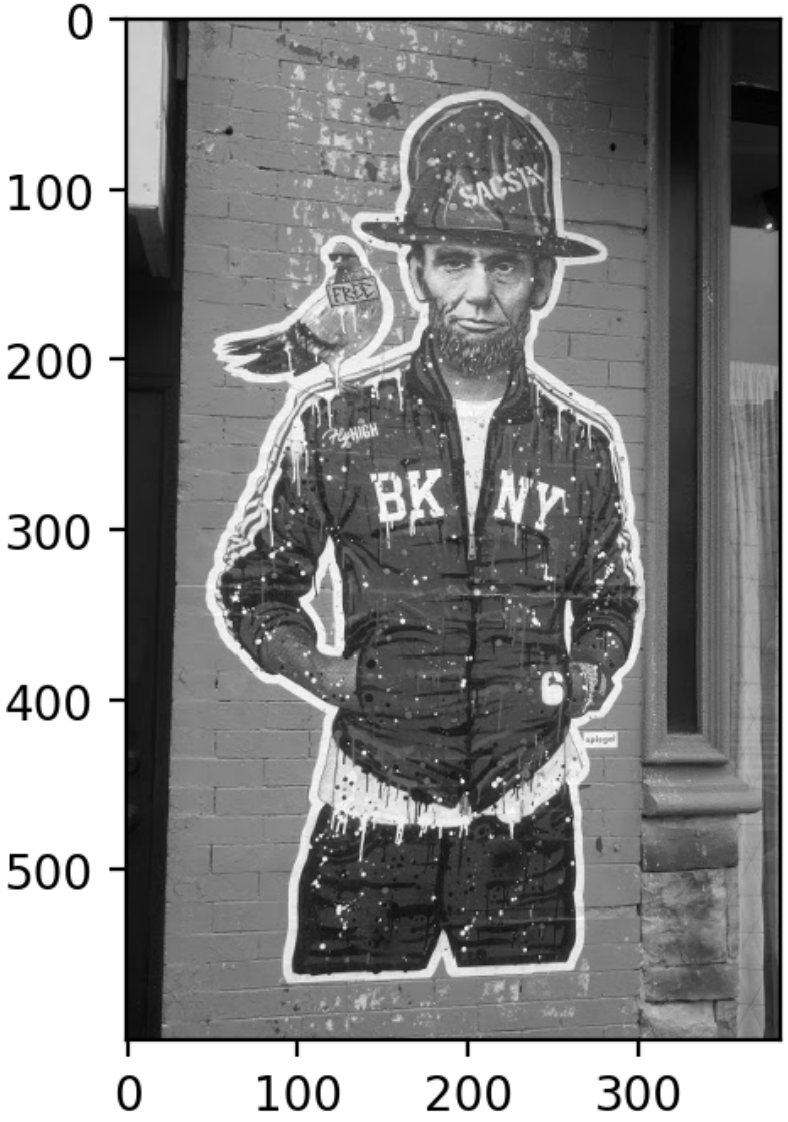
2d smoothing

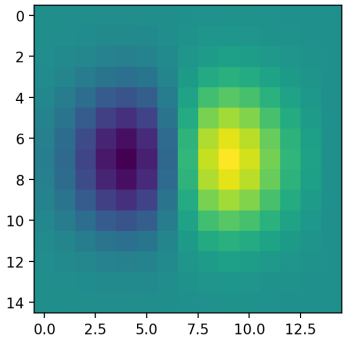


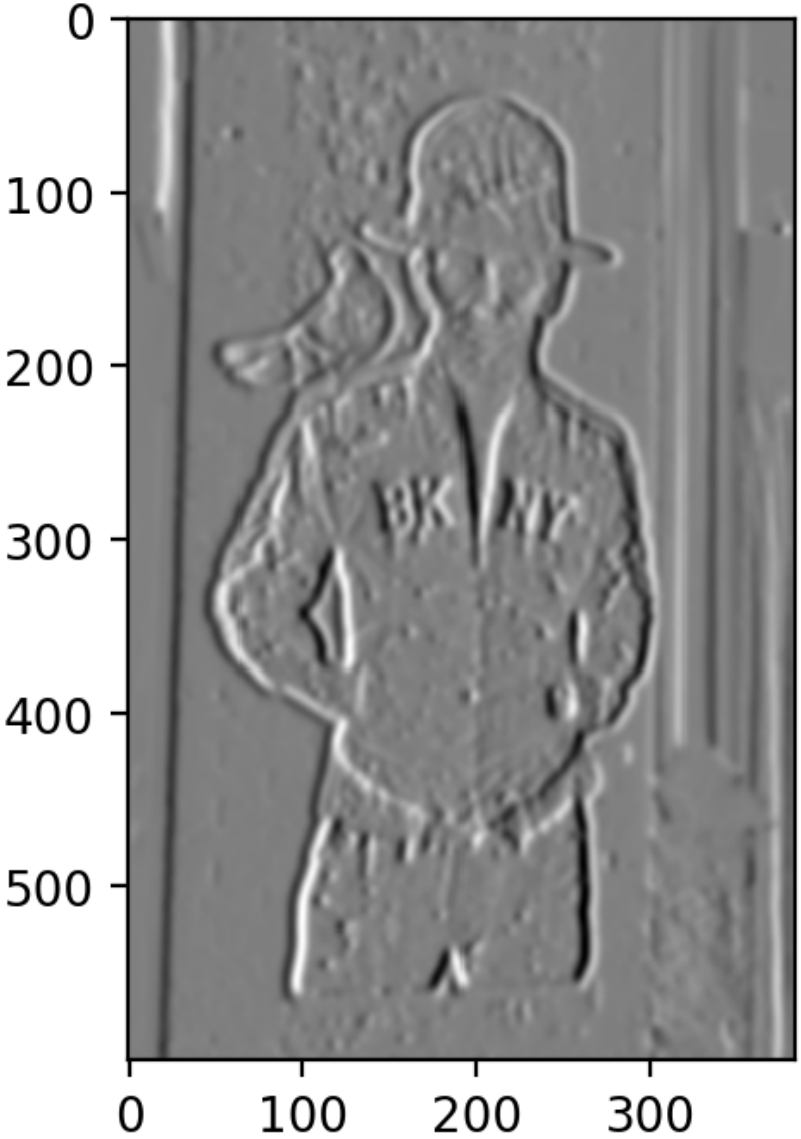




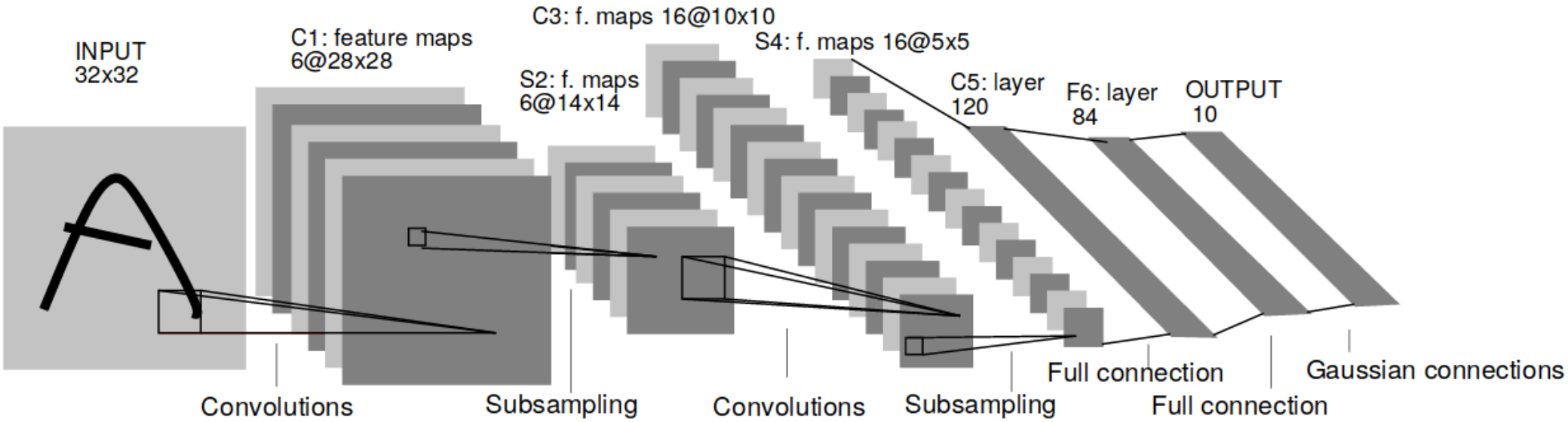
2d Gradients







Convolutional Neural Networks

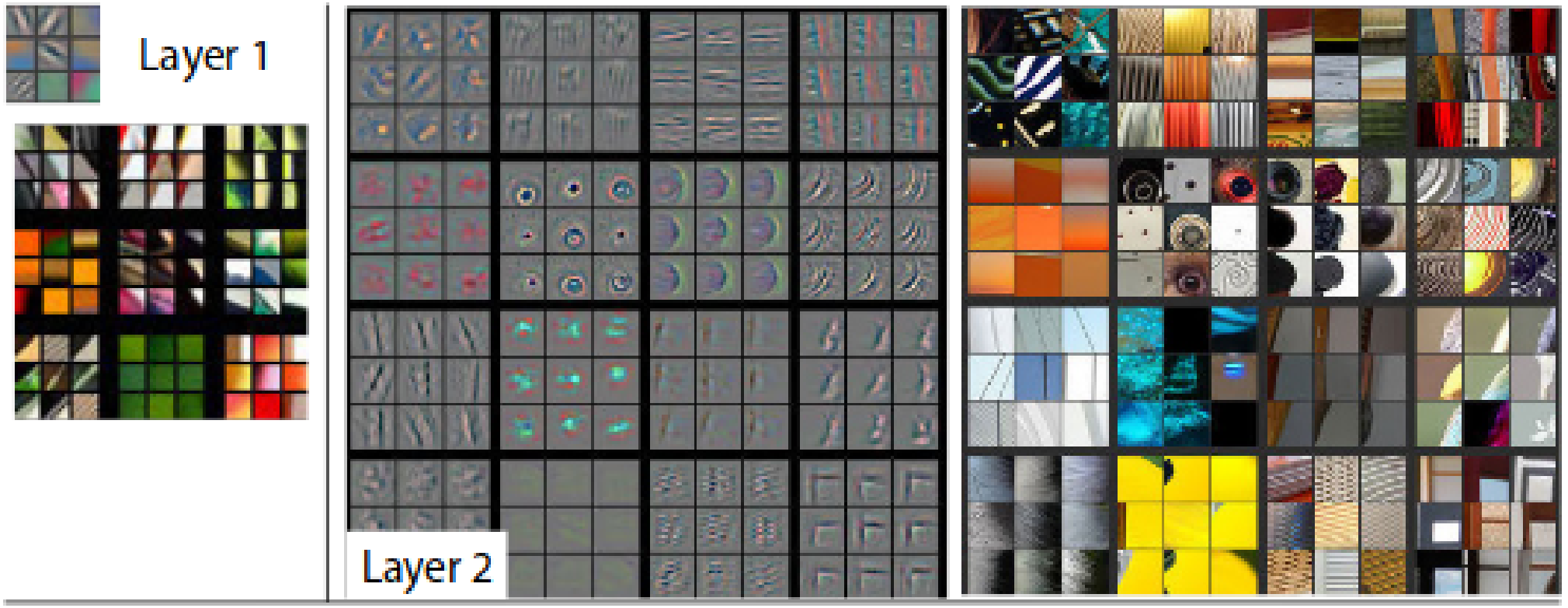


Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.

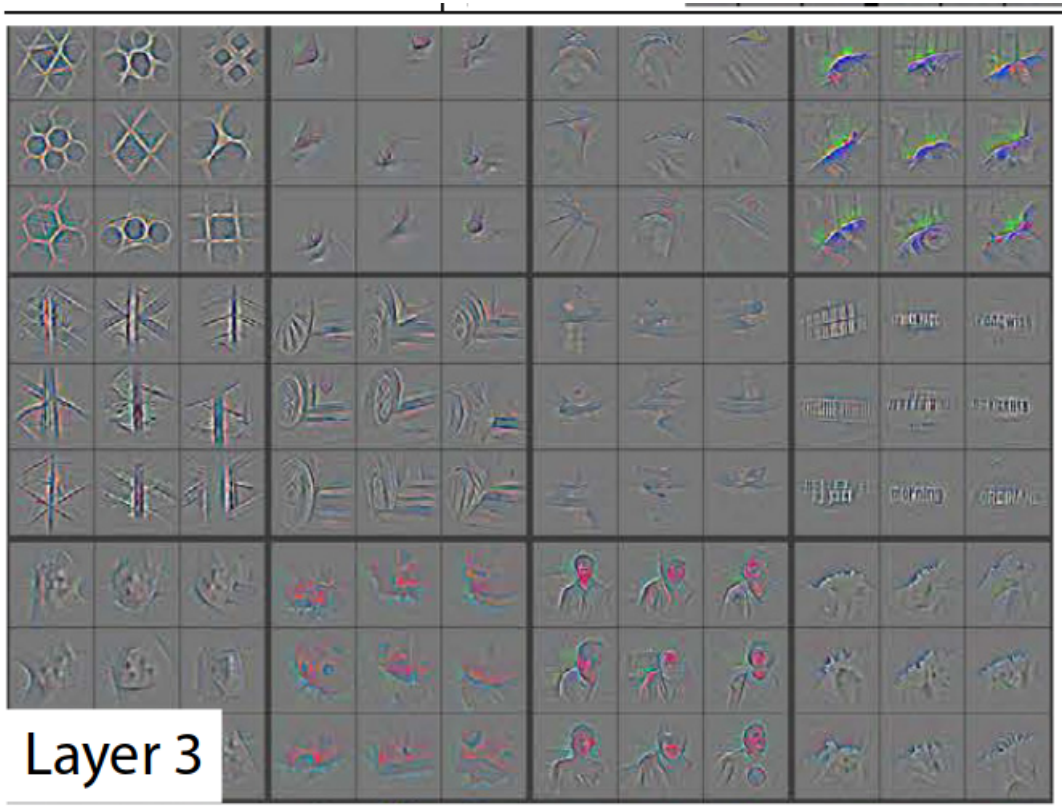
Gradient-based learning applied to document recognition

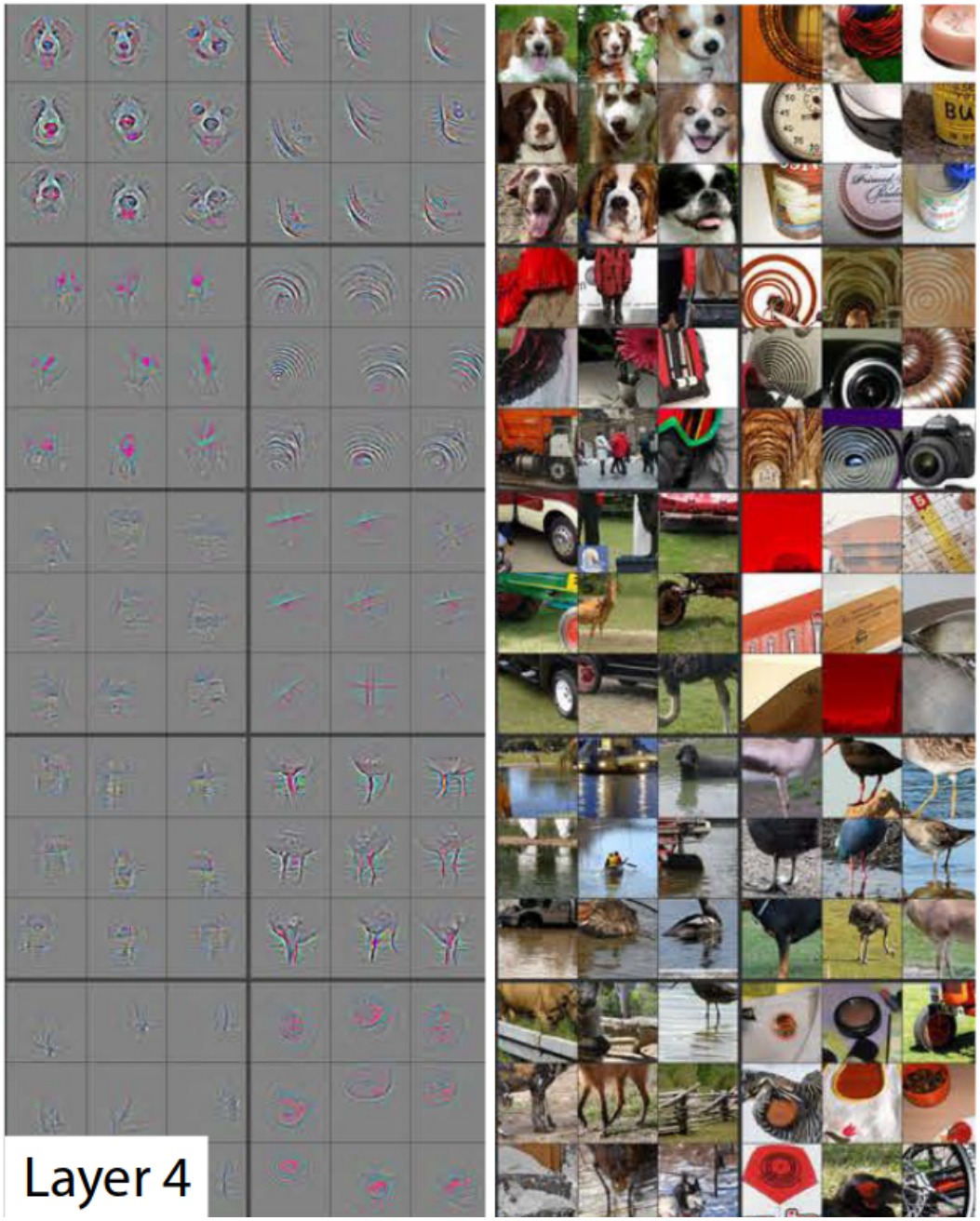
C1
S2
C3
S4
C5
4
Output
F6
4
4
4
3
8
4
3
3

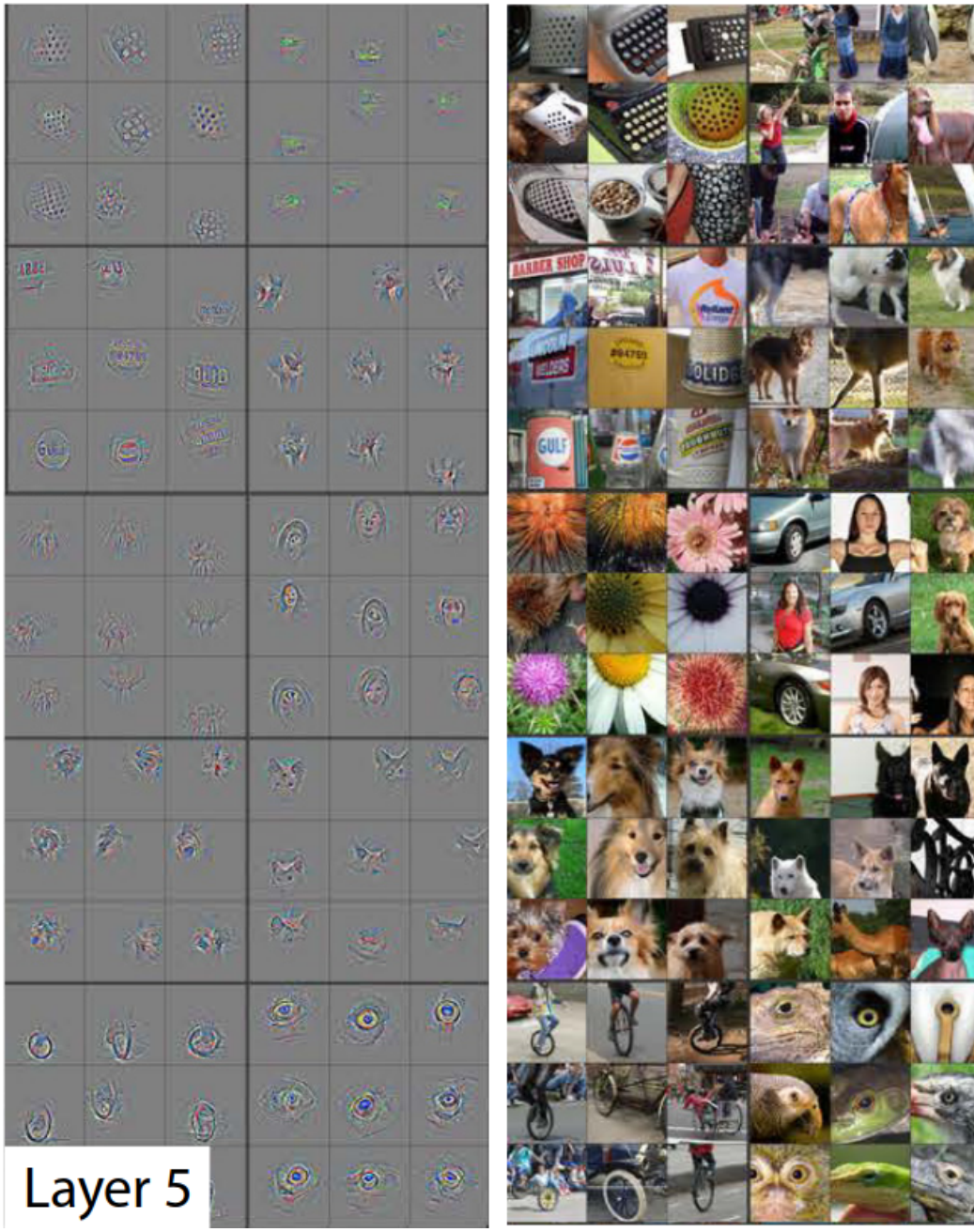
Deconvolution



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


Layer 3

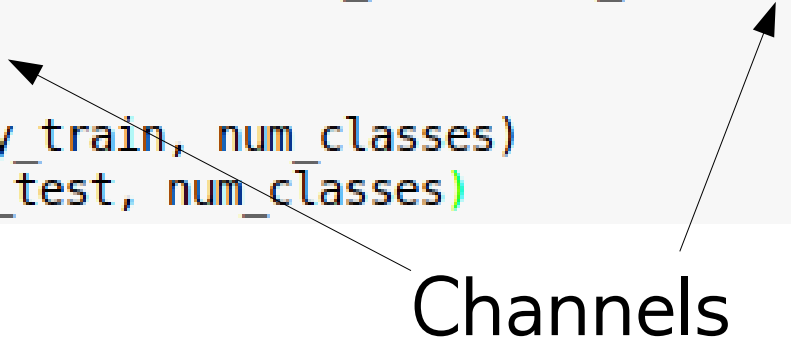

Layer 4


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)
```



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'))
```

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

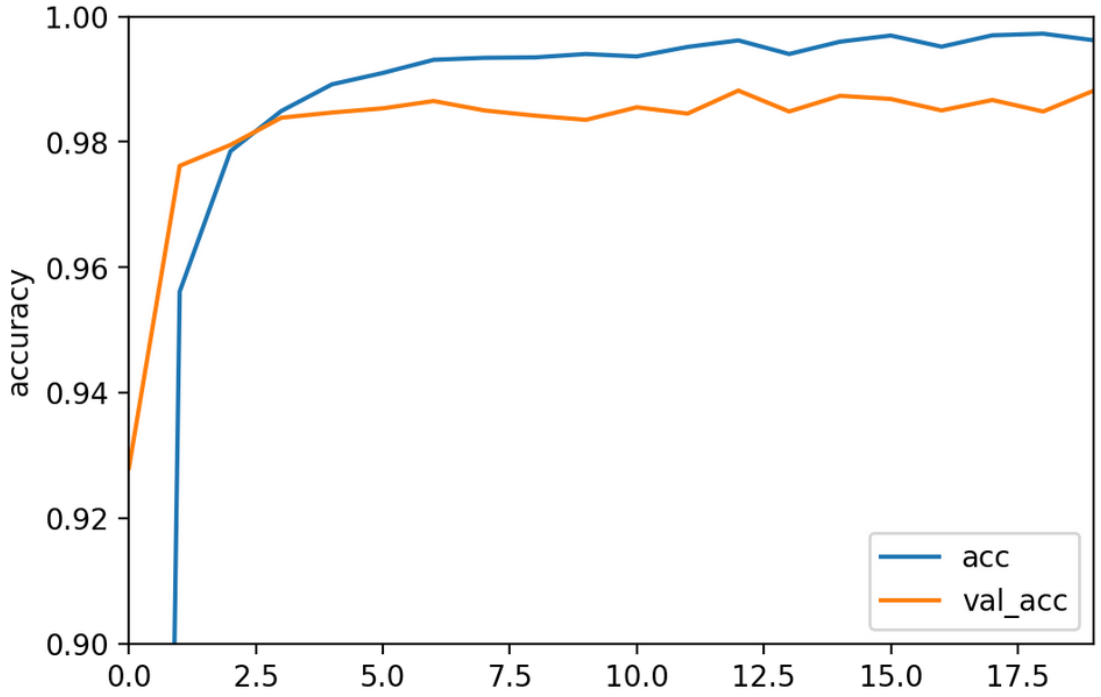
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
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)
```

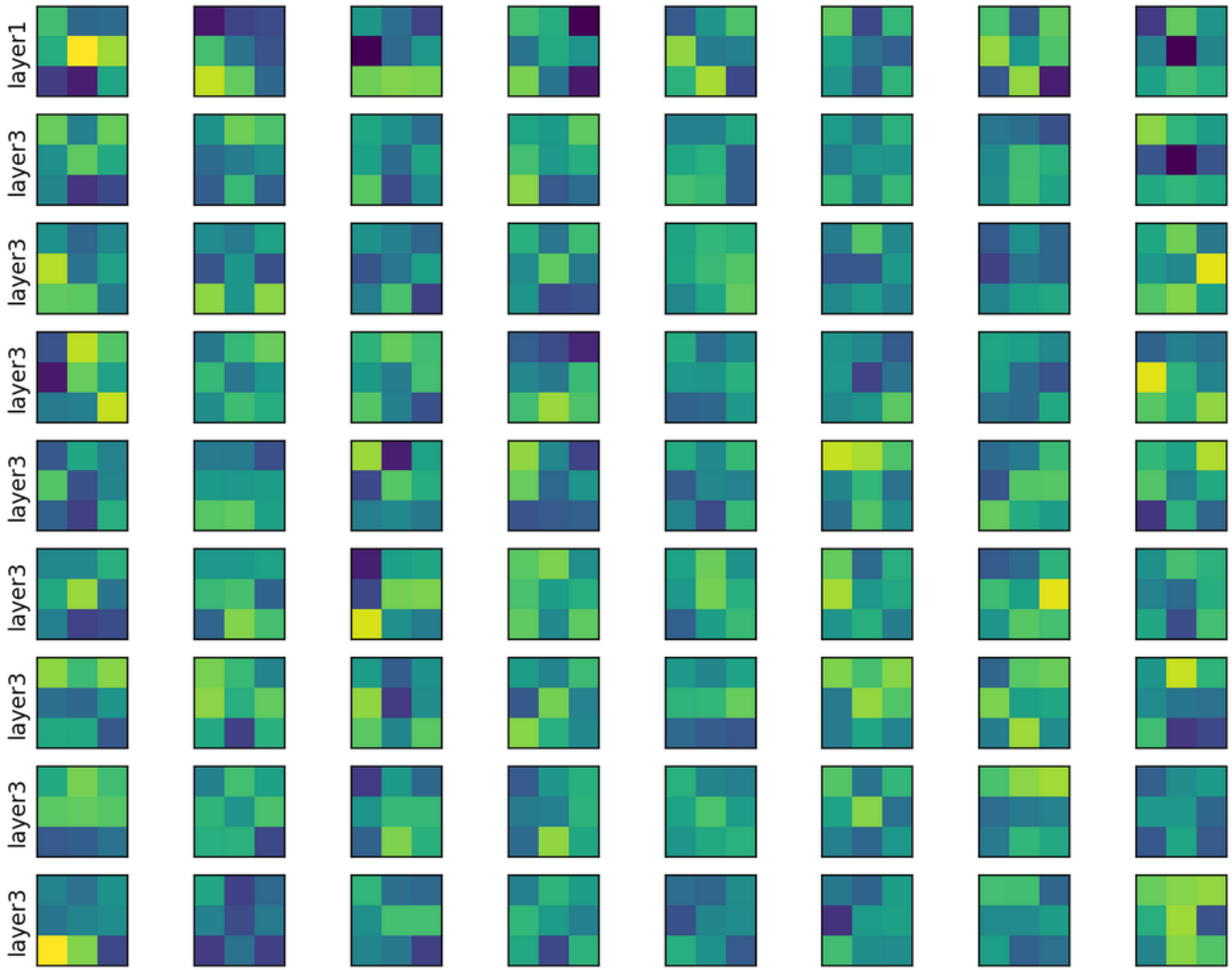
```
 9952/10000 [============================>.] - ETA: 0s
[0.089020583277629253, 0.98429999999999995]
```



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)
```




layer3
layer1
layer3
layer1
layer3
layer1
layer3
layer1
layer3
layer1

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

VGG and Imagenet Filters

Inspecting VGG

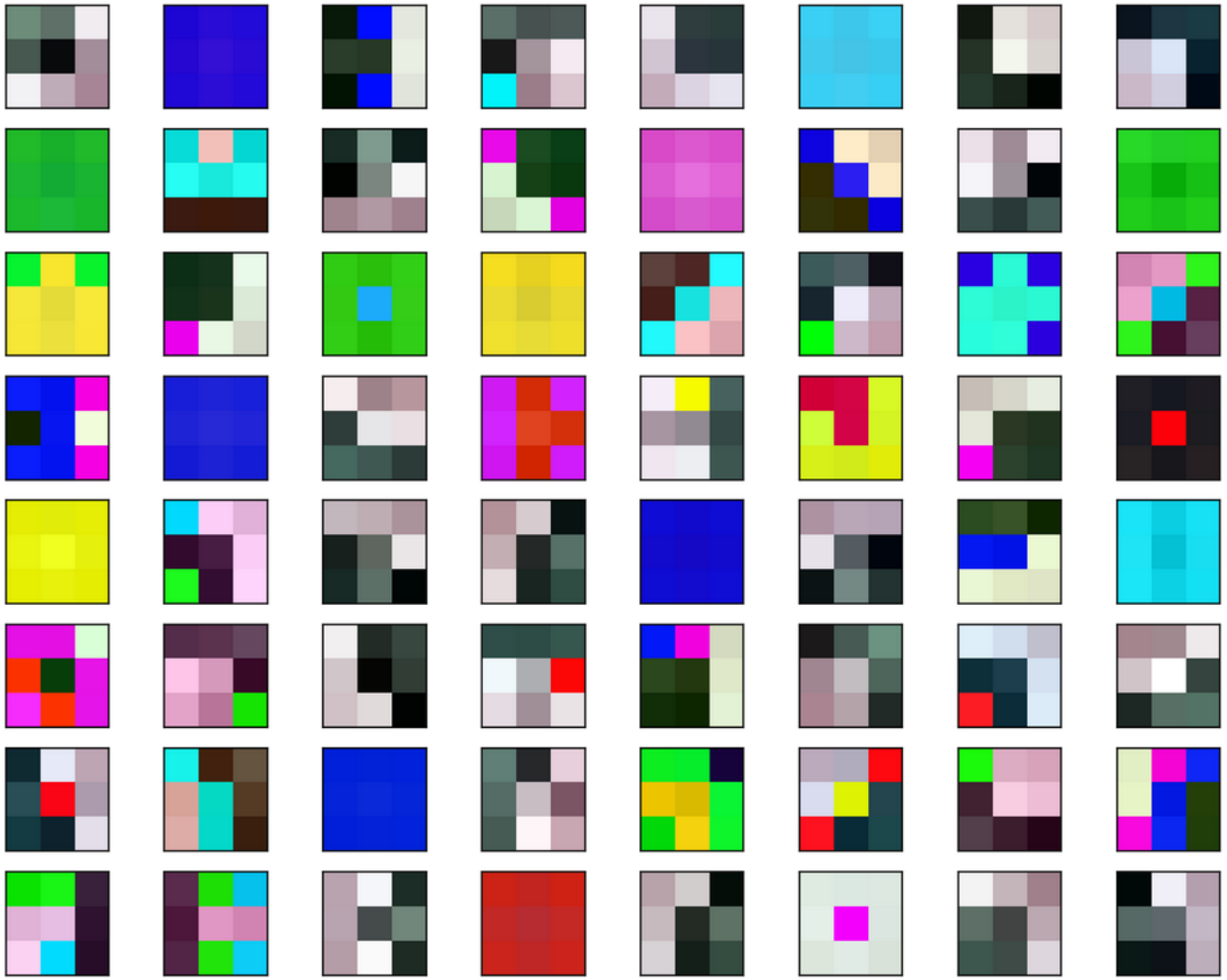
```
from keras import applications

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

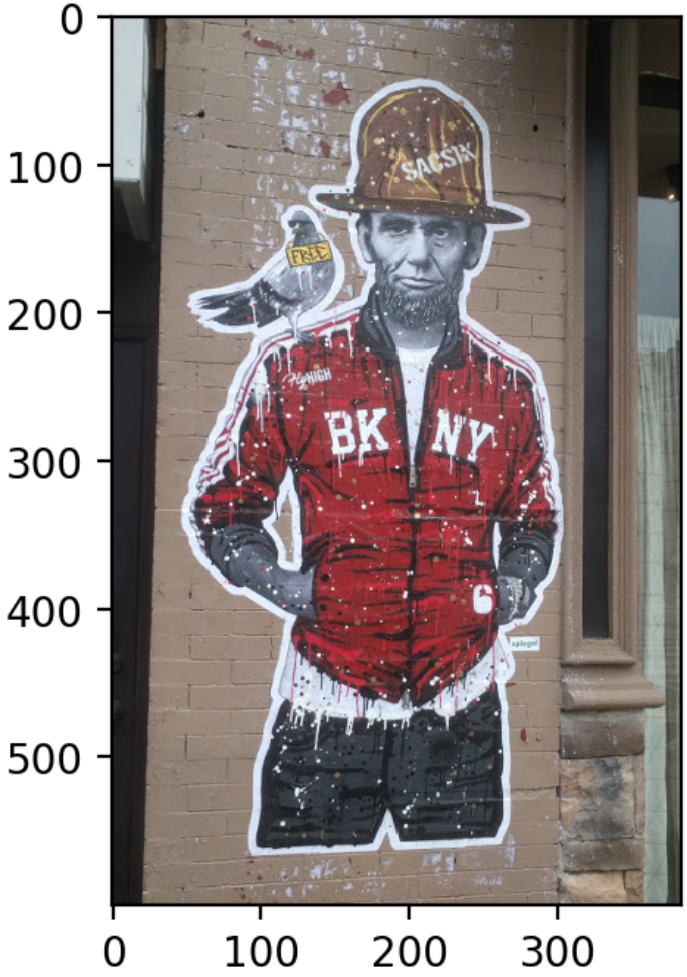
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None, None, 3)     0
_________________________________________________________________
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)
```





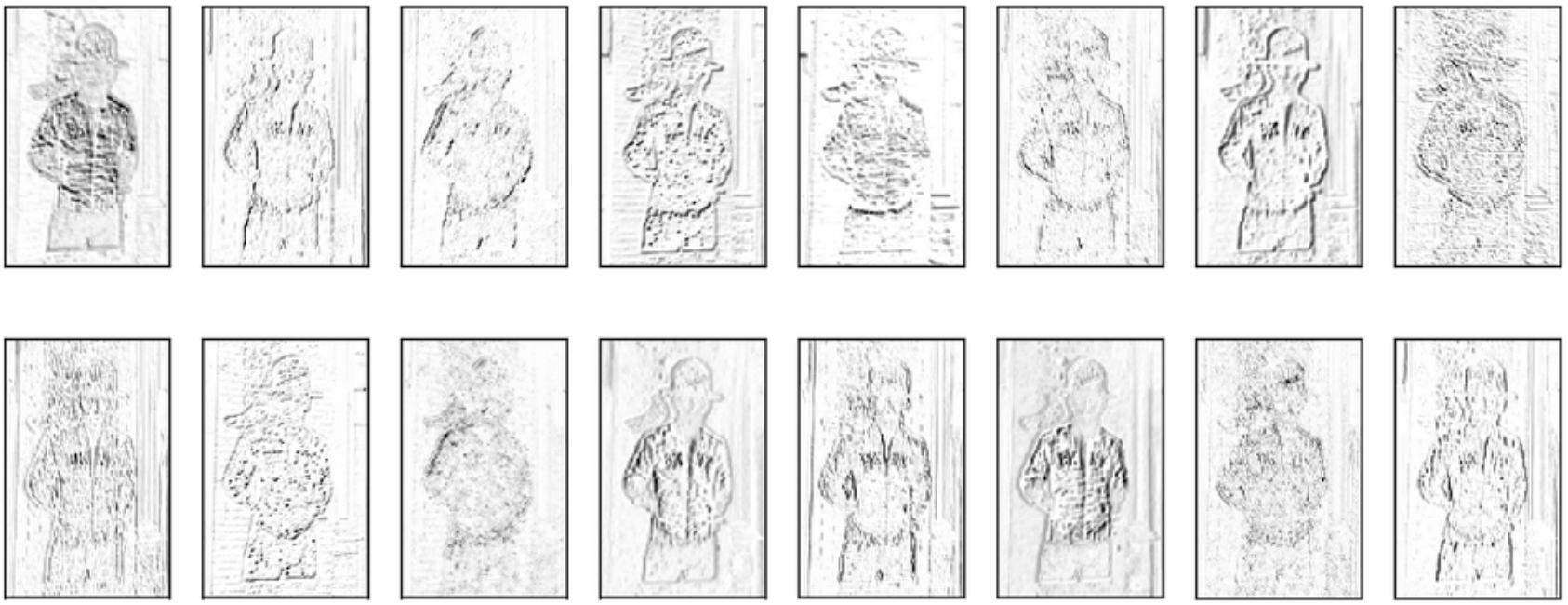

```
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
get_6rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[6].output])

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

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

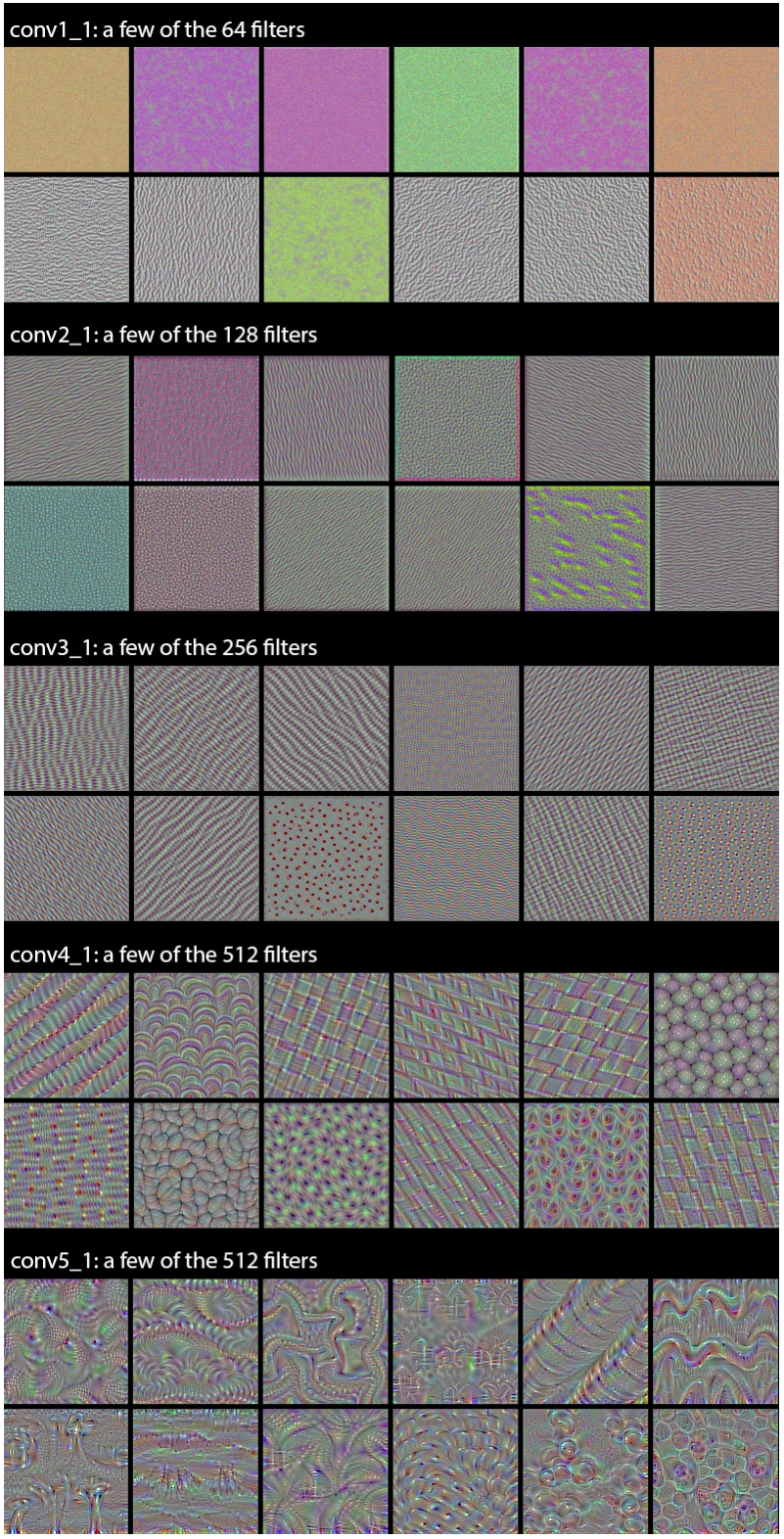
```
(1, 300, 192, 64)
(1, 150, 96, 128)
```

after first pooling layer

after second pooling layer

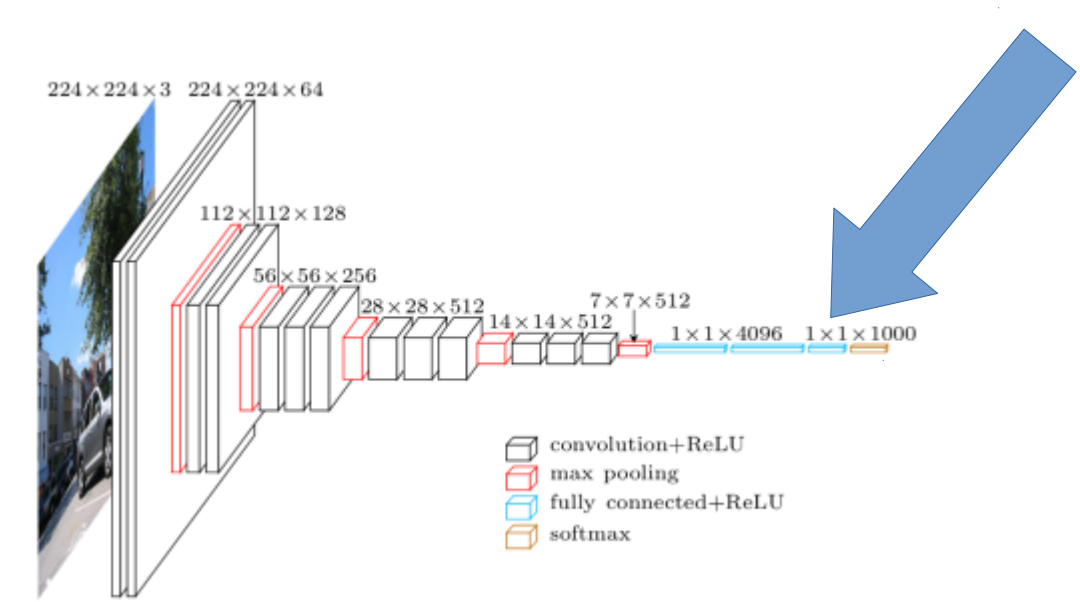


https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html



Using pre-trained networks

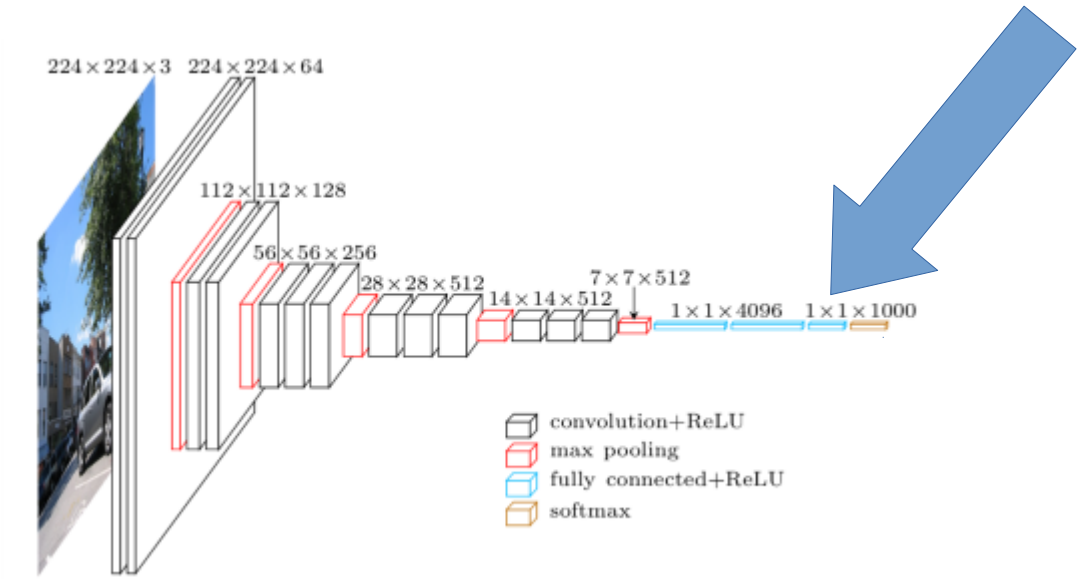
- Train on “large enough” data.
- Apply to new “small” dataset.
- Take activations of last or second to last fully connected layer.



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

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.



Ball snake vs Carpet Python

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

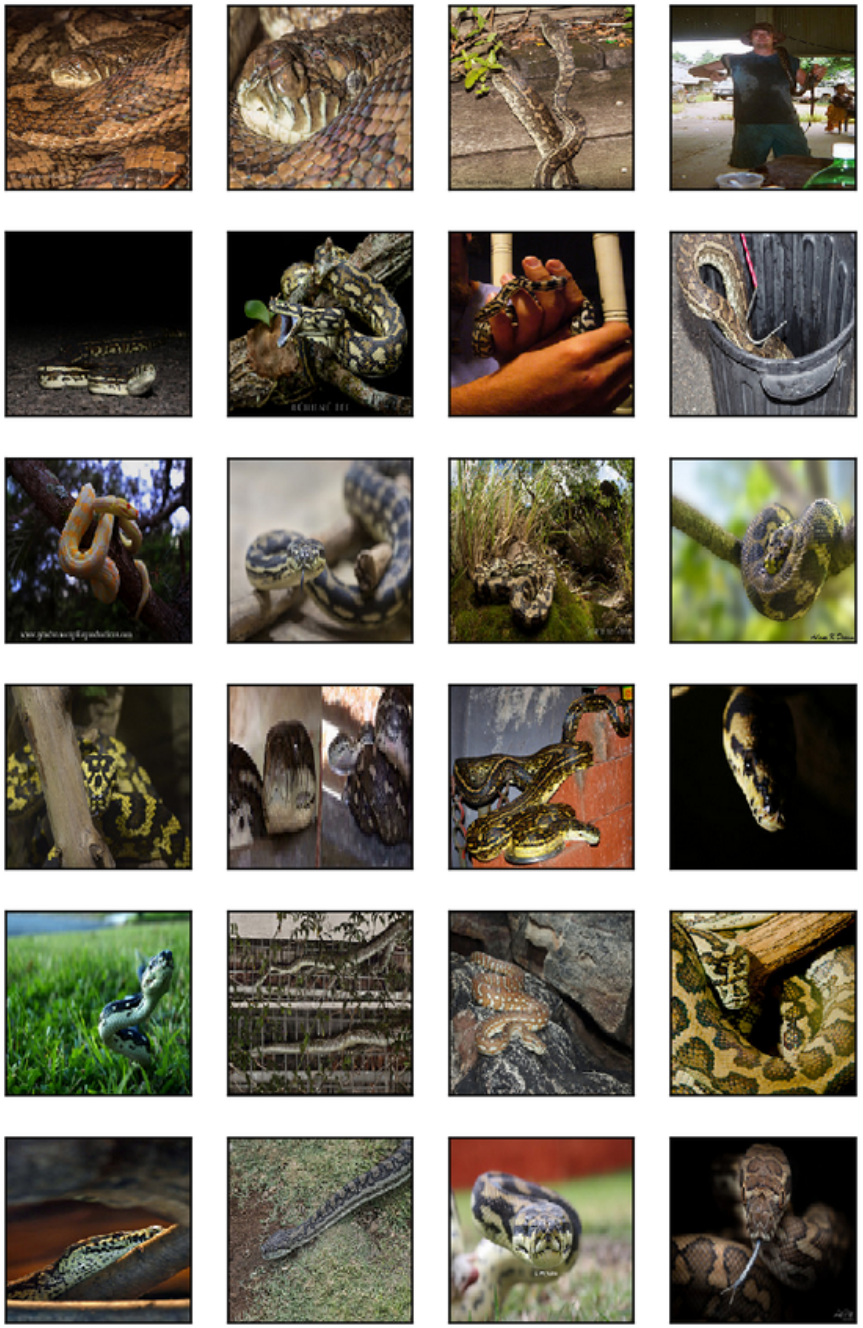
```python
def search_ids(search_string="python", per_page=10):
    photos_response = flickr.photos.search(text=search_string, per_page=per_page, sort='relevance')
    photos = json.loads(photos_response.decode('utf-8'))['photos']['photo']
    ids = [photo['id'] for photo in photos]
    return ids
```

```python
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']
```

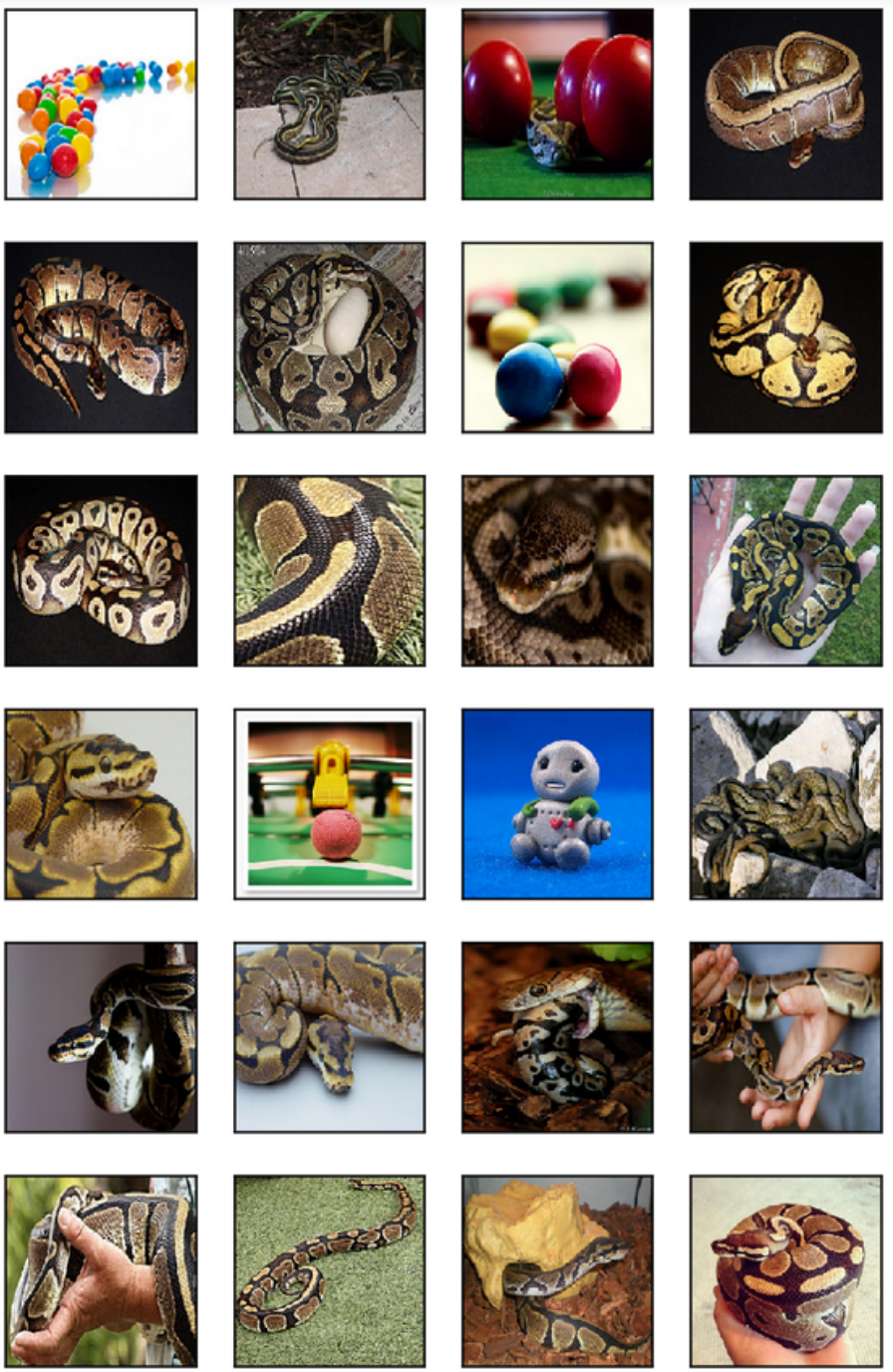
```python
ids = search_ids("ball snake", per_page=100)
urls_ball = [get_url(photo_id=i) for i in ids]
```

```python
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
```

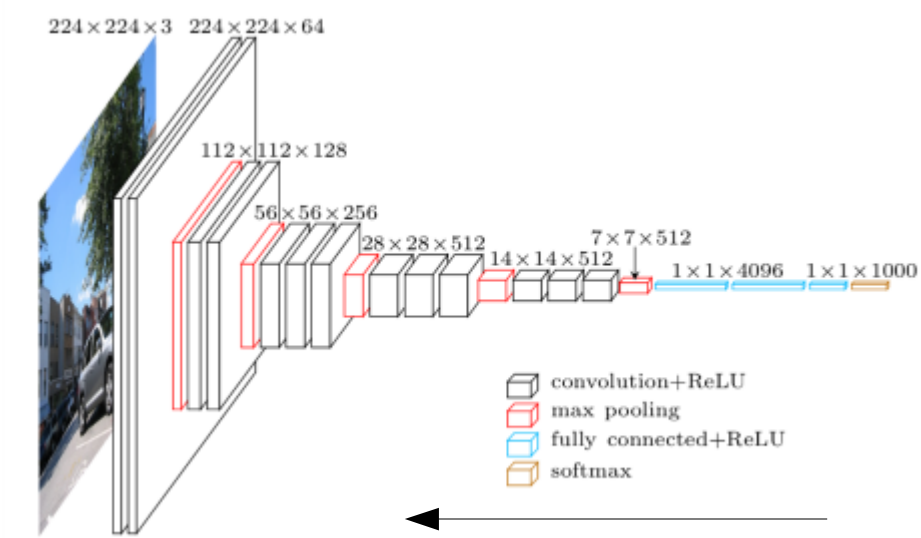
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, lr.predict(X_test))
```

```
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/