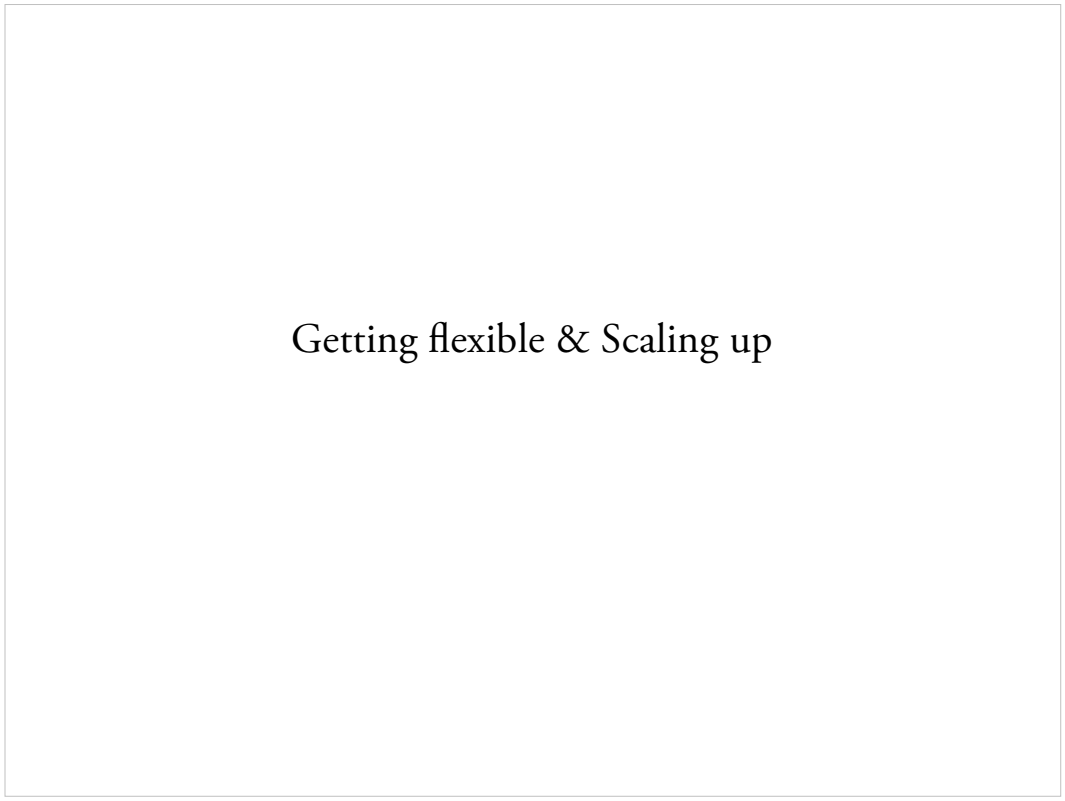
W4995 Applied Machine Learning

More Neural Networks

04/12/17

Andreas Müller



So today we'll talk about using neural nets beyond scikit-learn.

There are two main motivations for this: scaling to larger datasets and larger networks using GPUs, and having more flexible ways to create the neural network architectures.

If you want to get serious about neural nets, you are likely to include some modifications to the standard network. So how could you go about doing that?

Write your own neural networks

```
class NeuralNetwork(object):
    def __init__(self):
        # initialize coefficients and biases
        pass
    def forward(self, x):
        activation = x
        for coef, bias in zip(self.coef_, self.bias_):
            activation = self.nonlinearity(np.dot(activation, coef) + bias)
        return activation
    def backward(self, x):
        # compute gradient of stuff in forward pass
        pass
```

It's actually pretty straight-forward to write the prediction functions, or forward pass of a neural network. There is matrix multiplications and nonlinearities. It quickly becomes more complicated when you use some of the tricks we'll discuss today, but even for this writing down the gradient can be a bit tricky and something people used to get wrong all the time.

So they came up with a trick that avoided having to write down the gradients yourself, also known as the backward pass.

That trick is autodiff.

Autodiff

```
# http://mxnet.io/architecture/program_model.html
class array(object) :
    """Simple Array object that support autodiff."""
    def __init__(self, value, name=None):
        self.value = value
        if name:
            self.grad = lambda g : {name : g}

    def __add__(self, other):
        assert isinstance(other, int)
        ret = array(self.value + other)
        ret.grad = lambda g : self.grad(g)
        return ret

    def __mul__(self, other):
        assert isinstance(other, array)
        ret = array(self.value * other.value)
        def grad(g):
            x = self.grad(g * other.value)
            x.update(other.grad(g * self.value))
            return x
        ret.grad = grad
        return ret
```

```
# some examples
a = array(np.array([1, 2]), 'a')
b = array(np.array([3, 4]), 'b')
c = b * a
d = c + 1
print(d.value)
print(d.grad(1))
```

```
[4 9]
{'b': array([1, 2]), 'a': array([3, 4])}
```

Here is a toy implementation of the idea behind autodiff. It's a class called array with some operations such as addition of integer and multiplication with other arrays.

It also has a method that returns the gradient called grad. The gradient of the array is just the identity function.

The trick is what happens with the addition and multiplication. If you add something to an array or you multiply two arrays, the result again is an array, and has again a gradient. The product actually has two gradients, one for each array involved.

The magic here is that while we are doing some computation, we are keep track of that computation and building up a graph of how to compute the gradient of it.

Autodiff

```python
# http://mxnet.io/architecture/program_model.html
class array(object) :
    """Simple Array object that support autodiff."""
    def __init__(self, value, name=None):
        self.value = value
        if name:
            self.grad = lambda g : {name : g}

    def __add__(self, other):
        assert isinstance(other, int)
        ret = array(self.value + other)
        ret.grad = lambda g : self.grad(g)
        return ret

    def __mul__(self, other):
        assert isinstance(other, array)
        ret = array(self.value * other.value)
        def grad(g):
            x = self.grad(g * other.value)
            x.update(other.grad(g * self.value))
            return x
        ret.grad = grad
        return ret
```

```python
# some examples
a = array(np.array([1, 2]), 'a')
b = array(np.array([3, 4]), 'b')
c = b * a
d = c + 1
print(d.value)
print(d.grad(1))
```

```
[4 9]
{'b': array([1, 2]), 'a': array([3, 4])}
```

On the right you can see the result of adding two arrays and then computing the gradient at a particular location.

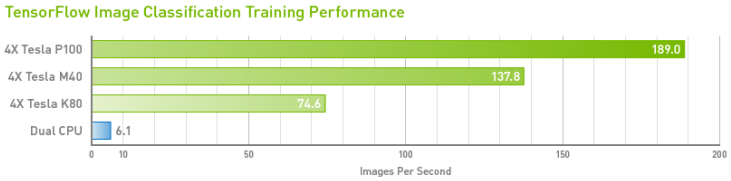
FIXME the gradient of multiplication?

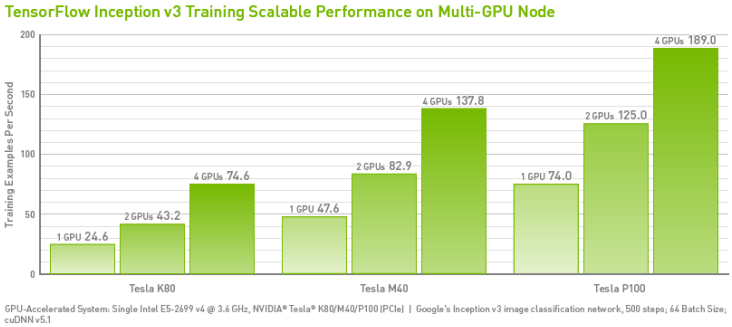
Any computation in the neural network is a simple operation like a matrix multiplication, addition or non-linearity. If we write down the derivative of each of these, we can keep track of our computation and automatically get the derivative.

It’s really easy to implement but really helpful.

Keep in mind that we actually hard-code the derivative for each operation, there is no symbolic differentiation involved.

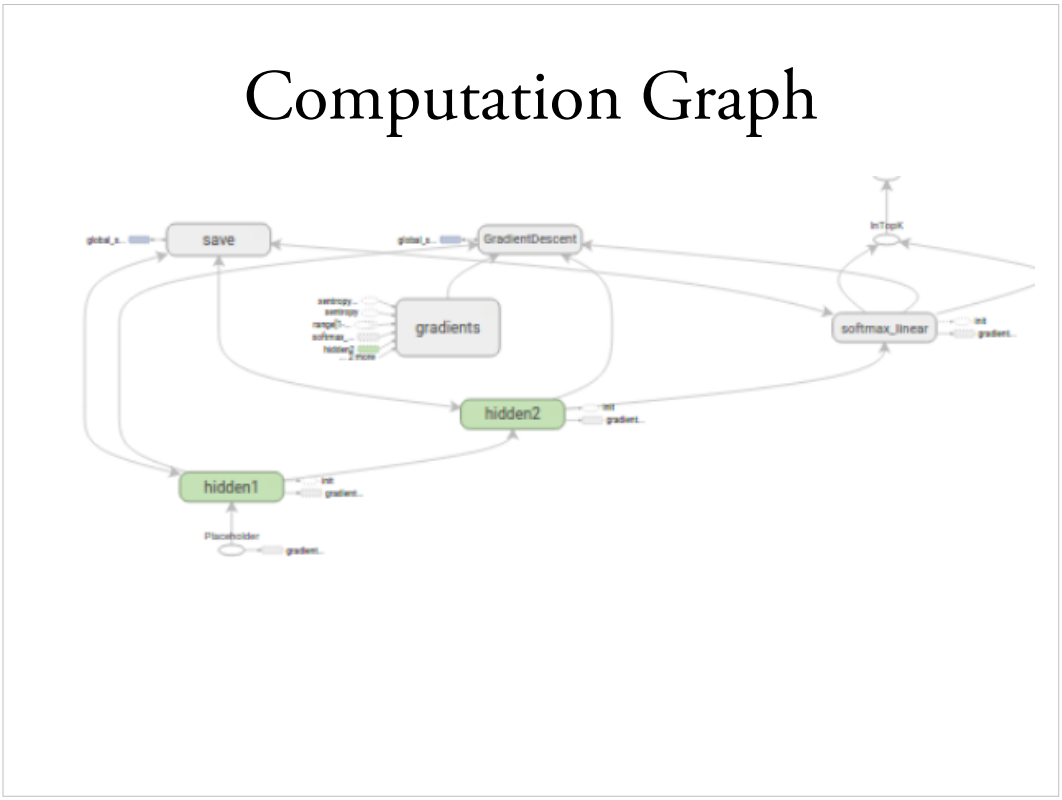
GPU Support





From
http://www.nvidia.com/object/gpu-accelerated-applications-tensorflow-benchmarks.html
Take with a grain of salt.

An important limitation of GPUs is that they usually have much less memory than the RAM, and memory copies between RAM and GPU are somewhat expensive.



Autodiff doesn’t solve all of the problems, though.

- Depending on which derivations you need to compute, you need to store different intermediate results (the net activation in backprob for example).

Given the limited amount of memory available, it’s important to know what to store and what can be discarded. That requires an end-to-end view of the computation, which can be summarized as a graph.

Then you can use knowledge about the computation graph to decide what results to cache, what results to throw away and when to communicate between CPU and GPU.

Having a representation of the graph also helps with visual debugging and understanding your network structure. The computation graph is more complex than the network graph because it includes intermediate results and gradient computations.

All I want from a deep learning framework

- Autodiff
- GPU support
- Optimization and inspection of computation graph
- on-the-fly generation of the graph?
- distribution over cluster?
- Choices:
  - TensorFlow
  - Theano
  - Torch (lua)

So create deep learning model efficiently, I need support for auto diff, computation on a GPGPU, and optimization and inspection of the computation graph.

There is some more advanced features that can come in handy for research, like on-the-fly generation of computation graphs, and distributing the network over multiple GPUs or over a cluster.

At least the first three are all provided by the TensorFlow, Theano and Torch frameworks, which are the most established deep learning frameworks.

These don't provide deep learning models, they are infrastructure, more like numpy on steroids than sklearn.

Theano was an early deep learning library for python, but I think it has become less relevant since tensorflow got released.

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

Then there are actual deep learning libraries that provide higher level interfaces built on top of this infrastructure.

There are a lot of these right now given the deep learning hype, I want to point out a couple.

There is tf.learn, a high-level interface to tensorflow, including a scikit-learn compatible API.

There's keras, which supports both Tensorflow and theano, there's lasagna, which I think has seen less activity recently, and torch.nn (lua) and pytorch building on top of the torch framework.

Then there are two projects that are deep learning libraries but that also come with their own framework underneath, chainer and mxnet.

I think right now Keras with tensorflow is the most commonly used one, and I want to go into those two a bit more. But these all have their pros and cons.

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/

A little more tf

- Everything passed to the graph is a “tensor”
- Either “variable”, “constant”, “placeholder”
- Learned parameters: variables
- Data: placeholders (data is assigned using “feed”)

Great Resources!

- https://www.tensorflow.org/versions/r0.10/tutorials/
- http://playground.tensorflow.org
- https://www.tensorflow.org/versions/r0.10/get_started/basic_usage
- https://www.tensorflow.org/versions/r0.10/tutorials/mnist/beginners/

- Note: tf.learn is now officially tf.contrib.learn
- Tensorboard web interface

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'),
])
```

There are two interfaces to keras, sequential and the functional, but we'll only discuss sequential.

Sequential is for feed-forward neural networks where one layer follows the other. You specify the layers as a list, similar to a sklearn pipeline.

Dense layers are just matrix multiplications. Here we have a neural net with 32 hidden units for the mnist dataset with 10 outputs. The hidden layer non-linearity is relu, the output if softmax for multi-class classification.

You can also instantiate an empty sequential model and then add steps to it.

For the first layer we need to specify the input shape so the model knows the sizes of all the matrices. The following layers can infer the sizes from the previous layers.

```
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'])
```

- Compile method picks optimization procedure and loss

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

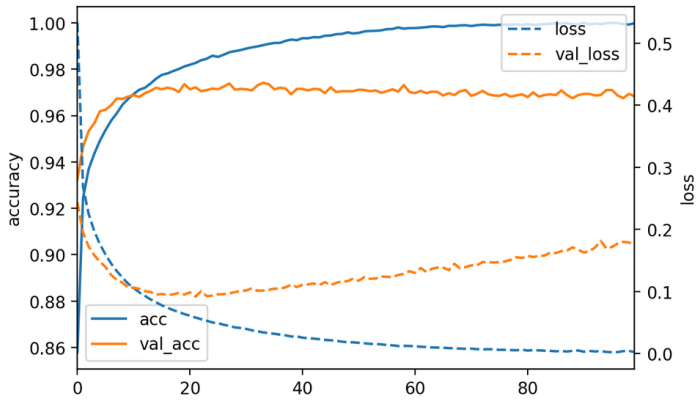
```
score = model.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
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()
```



Wrappers for sklearn

See https://keras.io/scikit-learn-api/

```
from keras.wrappers.scikit_learn import KerasClassifier, KerasRegressor
from sklearn.model_selection import GridSearchCV

def make_model(optimizer="adam", hidden_size=32):
    model = Sequential([
        Dense(hidden_size, input_shape=(784,)),
        Activation('relu'),
        Dense(10),
        Activation('softmax'),
    ])
    model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=['accuracy'])
    return model

clf = KerasClassifier(make_model)

param_grid = {'epochs': [1, 5, 10],  # epochs is fit parameter, not in make_model!
              'hidden_size': [32, 64, 256]}

grid = GridSearchCV(clf, param_grid=param_grid, cv=5)
```

```
grid.fit(X_train, y_train)
```

Useful for grid-search.

You need to define a callable that returns a compiled model.

You can search parameters that in Keras would be passed to “fit” like the number of epochs.

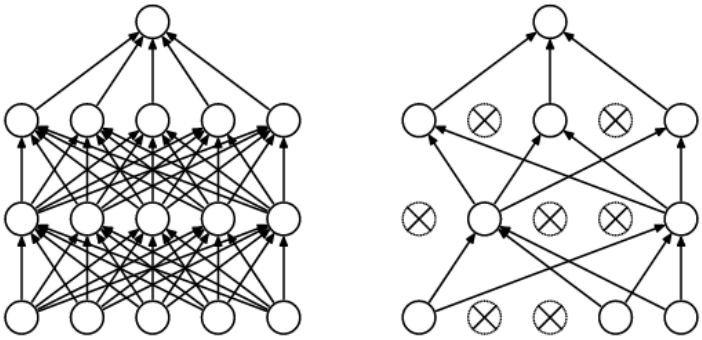
```
res = pd.DataFrame(grid.cv_results_)
res.pivot_table(index=["param_epochs", "param_hidden_size"],
                values=['mean_train_score', "mean_test_score"])
```

		mean_test_score	mean_train_score
param_epochs	**param_hidden_size**		
1	**32**	0.930017	0.935350
	64	0.941433	0.948358
	256	0.959117	0.966929
5	**32**	0.956417	0.969746
	64	0.967317	0.983113
	256	0.973900	0.992196
10	**32**	0.960100	0.979671
	64	0.968617	0.992025
	256	0.975050	0.996396

- Training longer overfits more and more units overfit more, but both also lead to better results.
- We should probably train much longer actually.
- Setting the number of epochs via cross-validation is a bit silly since it means starting from scratch again each time. Using early stopping would be better.

Drop-out Regularization

Randomly set activations to zero:



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

https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
Rate often as high as .5, i.e. 50% of units set to zero!
Predictions: use all weights, down-weight by rate

Drop out is a very successful regularization technique developed in 2014. It is an extreme case of adding noise to the input, a previously established method to avoid overfitting.

Instead of adding noise, we actually set given inputs to 0. And not only on the input layer, also the intermediate layer.

For each sample, and each iteration we pick different nodes. Randomization avoids overfitting to particular examples.

Ensemble Interpretation

- Every possible configuration represents different network.
- With p=.5 we jointly learn $\binom{n}{\frac{n}{2}}$ networks
- Networks share weights
- For last layer dropout: prediction is approximate geometric mean of predictions of sub-networks.

Implementing Drop-Out

```
from keras.layers import Dropout

model_dropout = Sequential([
    Dense(1024, input_shape=(784,), activation='relu'),
    Dropout(.5),
    Dense(1024, activation='relu'),
    Dropout(.5),
    Dense(10, activation='softmax'),
])
model_dropout.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_dropout = model_dropout.fit(X_train, y_train, batch_size=128,
                                    epochs=20, verbose=1, validation_split=.1)
```

When to use drop-out

- Avoids overfitting
- Allows using much deeper and larger models
- Slows down training somewhat
- Wasn't able to produce better results on MNIST (I don't have a GPU) but should be possible

Convolutional neural networks

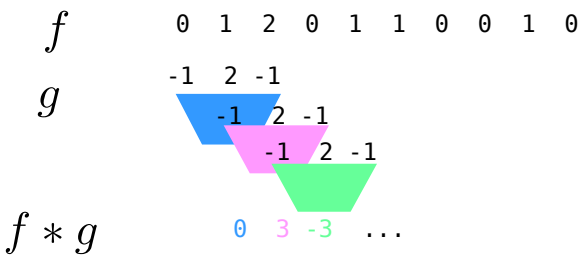
Idea

- Translation invariance
- Weight sharing

Definition of Convolution

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

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



The definition is symmetric in f, but usually one is the input signal, say f, and g is a fixed “filter” that is applied to it.

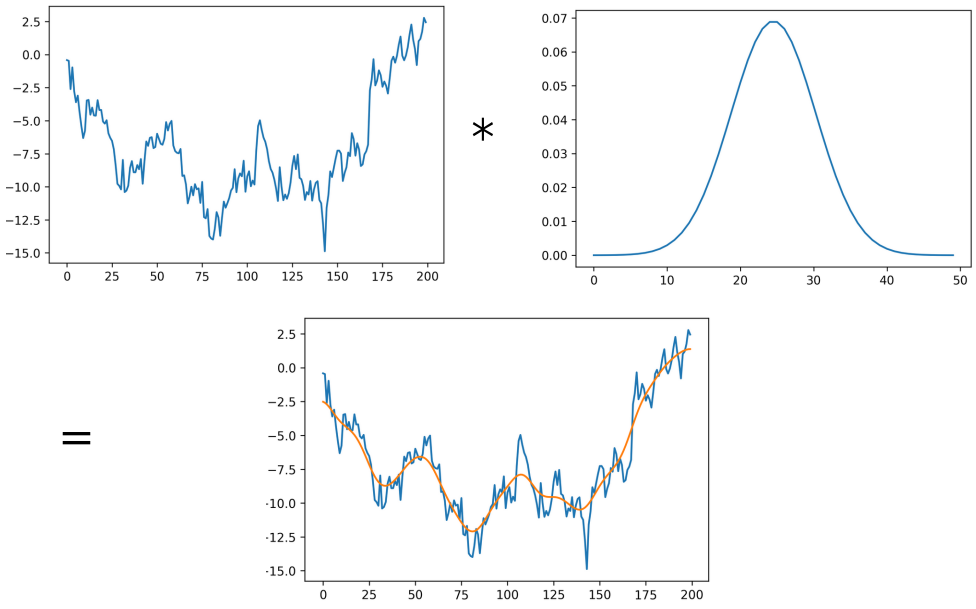
You can imagine the convolution as g sliding over f. If the support of g is smaller than the support of f (it’s a shorter non-zero sequence) then you can think of it as each entry in f * g depending on all entries of g multiplied with a local window in f.

Not that the output is shorter than the input by half the size of g. this is called a **valid** convolution.

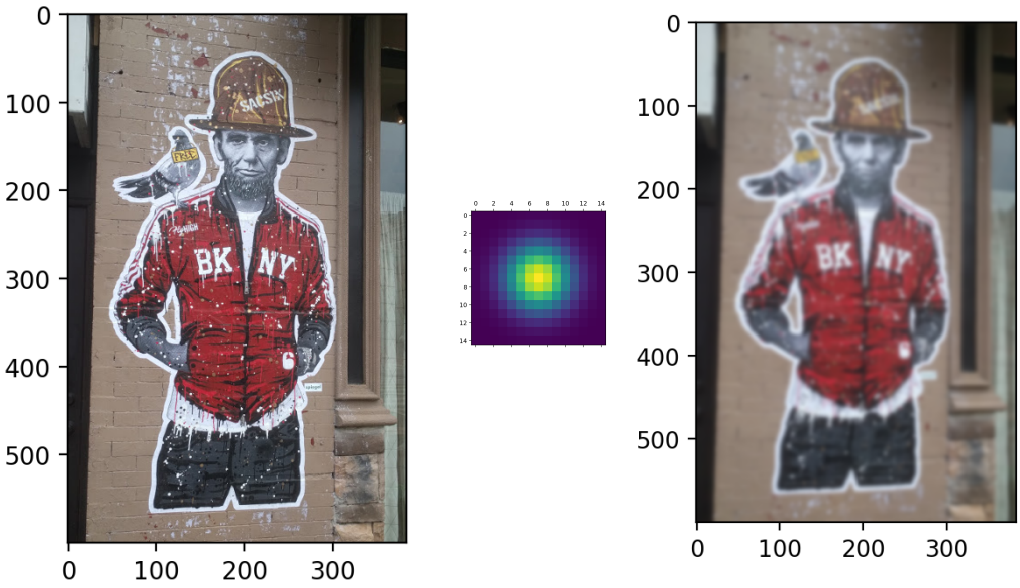
We could also extend f with zeros, and get a result that is larger than f by half the size of g, that’s called a **full** convolution. We can also just pad a little bit and get something that is of the same size as f.

Also not that the filter g is flipped as it’s indexed with -m

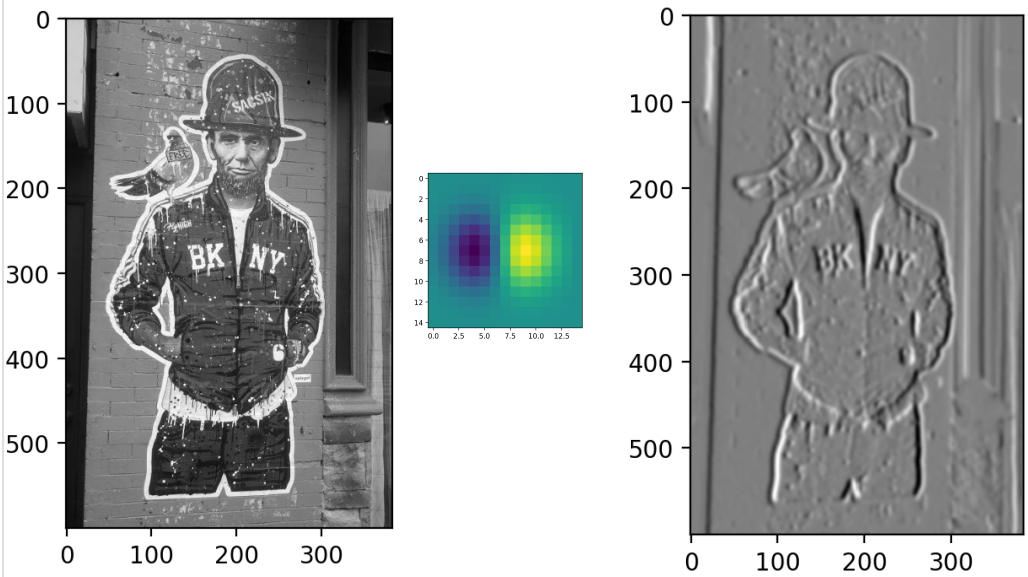
1d example: Gaussian smoothing



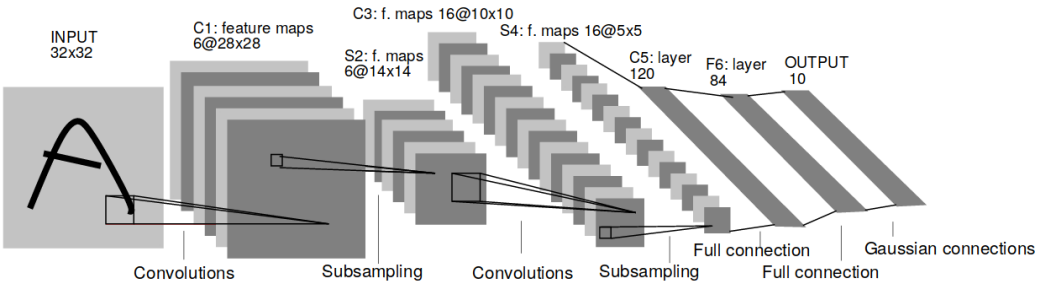
2d smoothing



2d Gradients



Convolutional Neural Networks



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

Gradient-based learning applied to document recognition

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

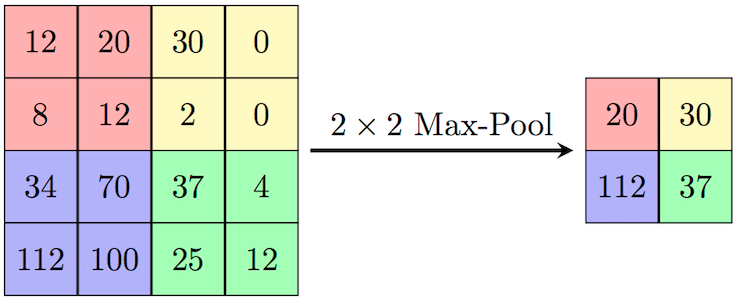
You can multiple layers of convolutions and resampling operations. You start convolving the image, which extracts local features. Each convolutions creates new “feature maps” that serve as input to later convolutions.

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

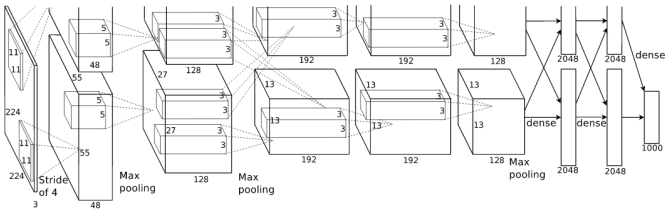
So you end up with more and more feature maps with lower and lower resolution.

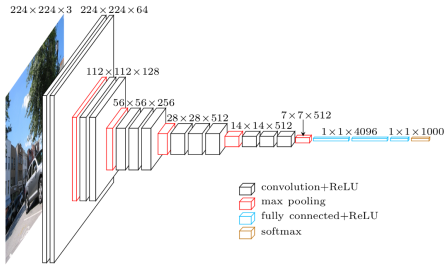
At the end, you have some fully connected layers to do the classificattion.

Max pooling



Need to remember position of maximum for back-propagation.
Again not differentiable → subgradient descent





VGG 16

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

conv <receptive field size>-<number of channels>

Here are two more recent architectures, AlexNet from 2012 and VGG net from 2015.

These nets are typically very deep, but often have very small convolutions. In VGG there are 3x3 convolutions and even 1x1 convolutions which serve to summarize multiple feature maps into one.

There is often multiple convolutions without pooling in between but pooling is definitely essential.

Conv-nets with keras

Preparing data

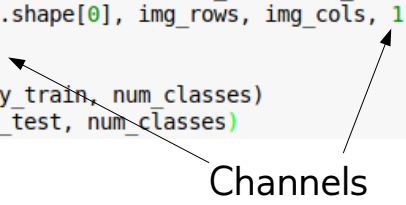
```
batch_size = 128
num_classes = 10
epochs = 12

# 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.

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

num_classes = 10
cnn = Sequential()
cnn.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(32, (3, 3), activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Flatten())
cnn.add(Dense(64, activation='relu'))
cnn.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.summary()
```

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_5 (MaxPooling2	(None, 13, 13, 32)	0
conv2d_6 (Conv2D)	(None, 11, 11, 32)	9248
max_pooling2d_6 (MaxPooling2	(None, 5, 5, 32)	0
flatten_3 (Flatten)	(None, 800)	0
dense_163 (Dense)	(None, 64)	51264
dense_164 (Dense)	(None, 10)	650

Total params: 61,482.0
Trainable params: 61,482.0
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

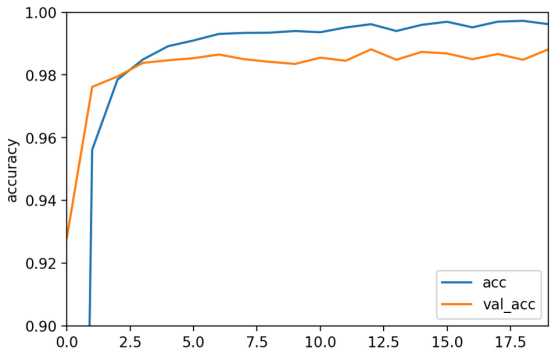
Total params: 669,706.0
Trainable params: 669,706.0
Non-trainable params: 0.0

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]
```



Batch Normalization

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

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

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

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

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

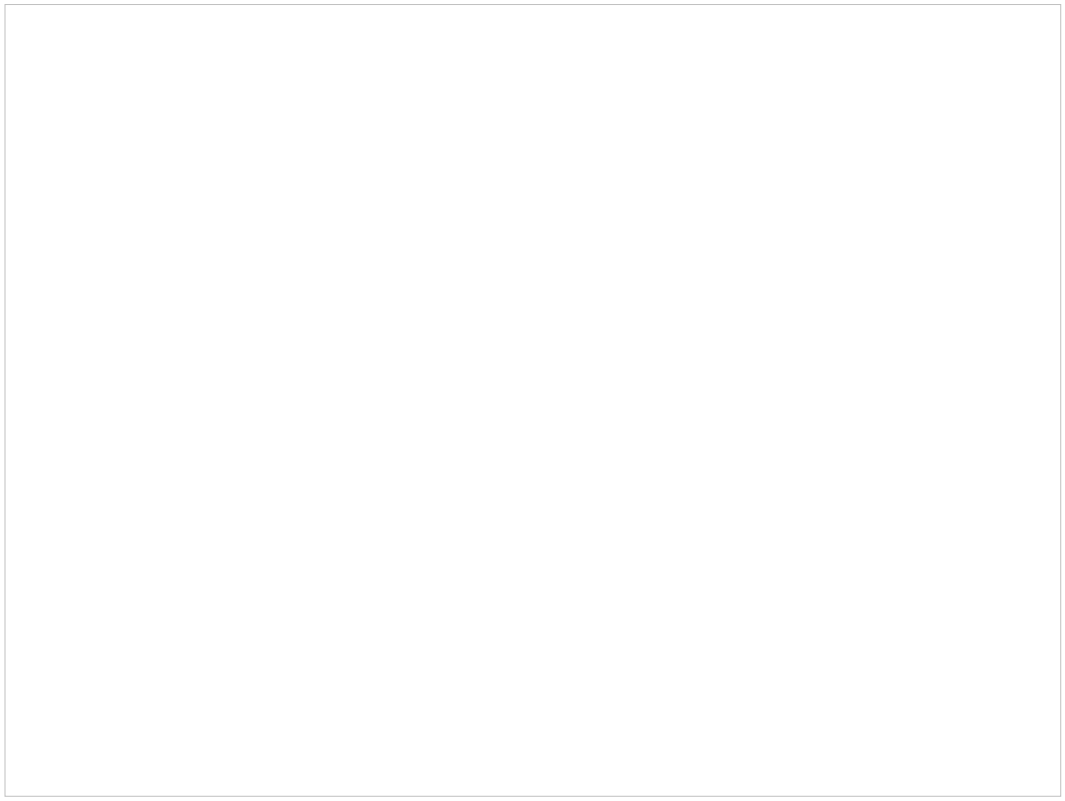
https://arxiv.org/abs/1502.03167

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.



Inspecting Conv-Nets

Tensorboard?

Deconvolution?

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.

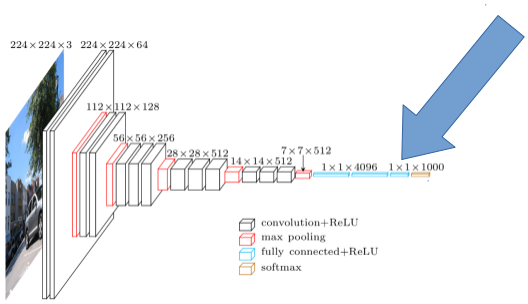
The picture on the left is change just slightly, and went from correctly classified to classified as Ostrich.

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.

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.



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.

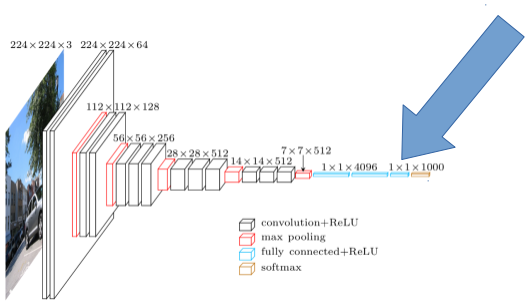
The easiest way to adapt a trained network to a new task is to just apply it to our dataset and take the activations of the second to last or last layer.

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.

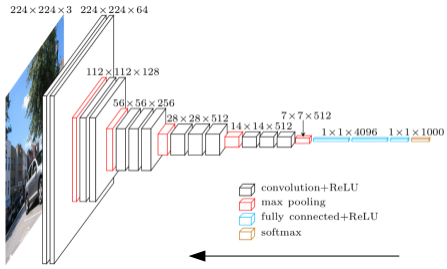
You can download pre-trained neural networks for many architectures online.

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.

Finetuning

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



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.