Why Frameworks

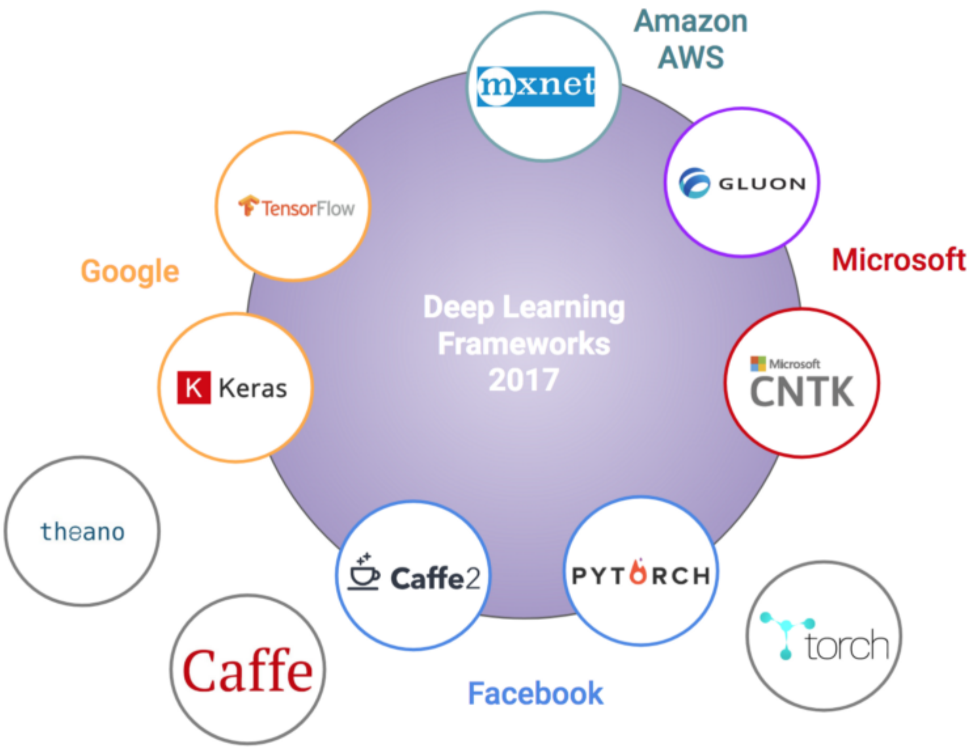
- building neural net systems with an array of complex features, optimizers, regularization and architecture requires flexible, scalable, and maintainable software
- Frameworks are scaleable and can work for large data on cpu or gpu(gpus allow for parallelization)
- Frameworks provide modularity to be able to build neural networks by plugging in different architectures (layers, regularization, optimizers) and processes easily

Approaches to differentiation

- Symbolic (requires symbolization of code and can be slow)
- numerical derivatives (slow and can have round off errors)
- Automated differentiation (forward pass, keeping history and applying chain rule of computations)

Frameworks Zoo

Source Battle of frameworks



f4

Differences in frameworks

- Caffe/torch use backprop through a graph for only the forward prop and traverse (less flexible)
- Tensorflow, MxNet and Theano create separate graphs for backprop (more flexibility as backprop can be applied to any graph)
- Caffe, Mxnet,torch treat parameters as part of operator nodes
- Tensorflow, Theano treat parameters as separate nodes in graph (more flexibility and re-use; variables and parameters are treated the same;)
- Performance similar across frameworks and use similar underlying kernels so choice based on development efficiency, portability and flexibility

source: Wattanavaekin, U. (2017) Large-Scale Learning Systems

- additional refs Differences -double backprop

History

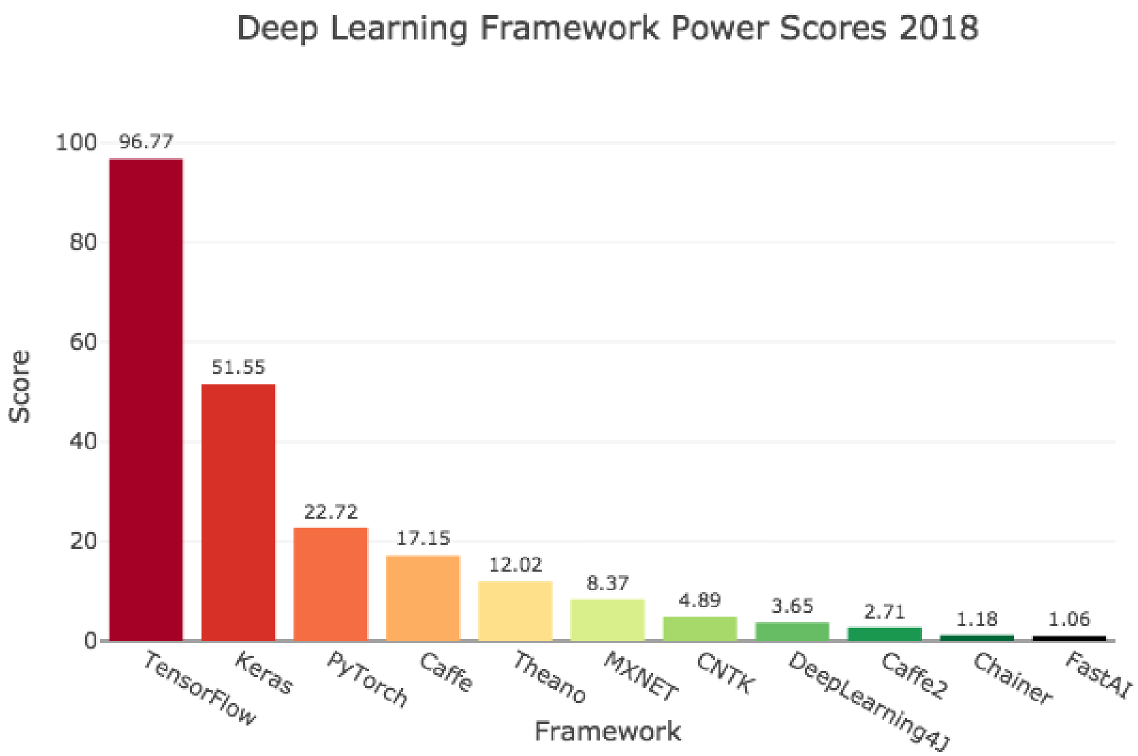
- An early framework was DistBelief but was in c++ and hard to add layers and scale up
- Tensor flow facilitated use of computational graphs to build large networks, automated differentiation (efficient backprop for computing gradients; c++ backend; declarative),
- Theano from U. of Montreal
- Berkeley created Caffe (old and large user base; imperative c level no auto differentiation)
- Facebook developed PyTorch (Fast AI keras like framework on top of PyTorch; flexibility imperative)
- CNTK by Microsoft
- Amazon's Mxnet (c++ backend; imperative & declarative)
- Keras (simple high level, more abstraction, less flexibility/examples)

Distbelief lead to tensorflow

> 'Where DistBelief fell short was in its ability to accommodate other machine-learning models and methods, and in smaller use cases, such as mobile. Maintaining different, separate systems for large and small-scale systems led to increased maintenance burdens and leaky abstractions. TensorFlow was born from this need for a more flexible programming model and the ability to use a wider variety of heterogeneous hardware platforms.'

by Jeff Dean

Popularity of Frameworks



Ranking based on usage, interest and popularity

Keras

- Aim to be for humans and not machines
    - high level abstraction with different backends (currently supports tensorflow, theano and CNTK; Amazon has a fork supporting mxnet)
- allows for easy prototyping (python based with interfaces to R, python)
- straight forward and easy to learn API and high level of abstraction

Class

- We will be using Rstudio Keras with tensorflow backend
- you can install cpu or gpu version if you are in the cloud
- Tensor+Flow (is tensors (arrays) flowing through the computational graph; support parallelization)

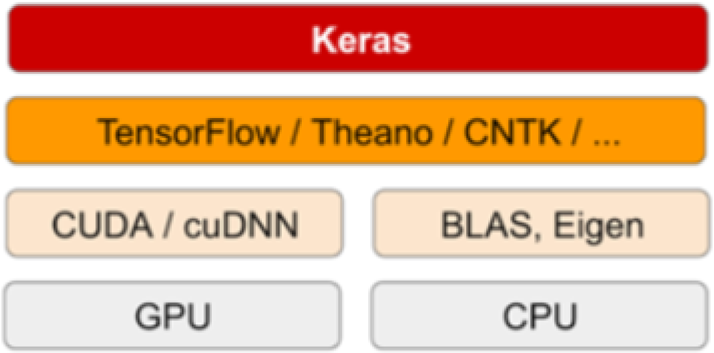
Data flow graph

Major gains in performance, scalability, and portability

- Parallelism—System runs operations in parallel
- Distributed execution—Graph is partitioned across multiple devices
- Compilation—Use the information in your dataflow graph to generate faster code (e.g. fusing operations)
- Portability—Dataflow graph is a language-independent representation of the code in your model (deploy with C++ runtime)

Keras Software Stack

- underlying Blas and other libraries are the same across frameworks



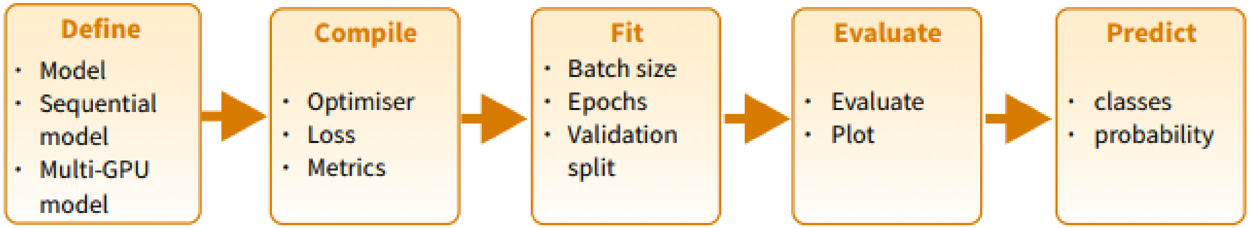
Installing Keras

- Installation (you should install Anaconda 3.x for Windows prior to installing Keras)

```
install.packages("reticulate")
install.packages("keras")
library(keras)
# only first time
#install_keras(method = "conda")
# if you gpu this would be:   install_keras(tensorflow = "gpu")
library(keras)
use_condaenv('r-tensorflow',required=TRUE)
# library to call python in R
library(reticulate)
conda_list()
reticulate::use_condaenv(condaname)
```

Keras Model

- Core steps in building a Keras model
- Define model (layers, regularization, architecture)
- Compile model (specify optimizer, learning rate, loss function, metrics to keep track of)
- Fit model to data (this updates your model object even if you don't set it equal to model)
- Plot and tune



Defining model

- use functional api call: keras_model
- this is more modular and flexible and requires you to create variables for your layers

```
input_layer <- layer_input(shape = 1,
                           name = 'input')
output_layer <- layer_dense(input_layer ,
                            units = 1, name='last_layer')
model_simple <- keras_model(inputs = input_layer,
                            outputs = output_layer )

summary(model_simple)
```

Layers

- Layers can have names, be re-used across models (as in pre-training)
    - 65 layers available: e.g.
- layer_dense() Add a densely-connected NN layer to an output
- layer_dropout() Applies Dropout to the input
- layer_batch_normalization() Batch normalization layer (Ioffe and Szegedy, 2014)
- each layer can also have weight initialization schemes specified
- Dense layer is one we will use most, has activation, initialization, name parameters and gives back: output = activation(dot(input, kernel) + bias) where kernel is weight matrix

Activations

- Activations are set on layer object available activations:
- ‘softmax’
- ‘elu’ – The exponential linear activation: x if x > 0 and alpha * (exp(x)-1) if x < 0.
- ‘selu’ -- The scaled exponential unit activation: scale * elu(x, alpha).
- ‘softplus’ -- The softplus activation: log(exp(x) + 1).
- ‘softsign’ -- The softplus activation: x / (abs(x) + 1).
- ‘relu’ -- The (leaky) rectified linear unit activation: x if x > 0, alpha * x if x < 0. If max_value is defined, the result is truncated to this value.
- ‘tanh’ -- Hyperbolic tangent activation function.
- ‘sigmoid’ – Sigmoid activation function.
- ‘hardsigmoid’

Compiling model

- Compiling the model converts it into a tensorflow graph and sets the optimizer and loss function/metrics
- Loss functions available: https://tensorflow.rstudio.com/keras/reference/#section-losses loss_binary_crossentropy() loss_categorical_crossentropy() loss_mean_squared_error() and more …
- Metric available for tracking: https://tensorflow.rstudio.com/keras/reference/#section-metrics metric_binary_accuracy() metric_binary_crossentropy() metric_categorical_accuracy() metric_categorical_crossentropy() metric_kullback_leibler_divergence() metric_mean_squared_error() …

```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

Optimizer

Available optimizers: (all have lr and decay params)

- optimizer_adadelta()
- optimizer_adagrad()
- optimizer_adam()
- optimizer_adamax()
- optimizer_nadam() (nestorov +adam)
- optimizer_rmsprop()
- optimizer_sgd() (has nestorov and momentum params)

API optimizers Keras Cheatsheet

Example of linear model with hidden units

```
input_layer =layer_input(shape = 1, name = 'input')
hidden_layer=layer_dense(input_layer,
                units = nh, activation = 'tanh',
                bias_initializer=
                initializer_random_uniform(
                minval = -0.1, maxval = 0.1,seed = 104),
                kernel_initializer=
                initializer_random_normal(
                mean=0,stddev=.1, seed = 104))
output_layer =layer_dense(hidden_layer ,units = 1,
                  bias_initializer=
                  initializer_random_uniform(
                  minval = -0.1, maxval = 0.1,
                  seed = 104),
                  kernel_initializer=
                  initializer_random_normal
                  (mean=0,stddev=.1, seed = 104))
model <- keras_model(inputs = input_layer,
                       outputs = output_layer )
```

example continued

```
#,clipnorm=1,,clipvalue=1
opt <- optimizer_sgd(lr = lr0,momentum=0)

compile(model,
        optimizer = opt,
        loss = "mse",
        metrics = c("mae")
)
summary(model)
print(system.time({results[[counter]]=
  fit(model,t(X),t(Y),validation_data =validation_data,
                         epochs=150000,
                          verbose=0,batch_size=20);}))
```

Example with binary classification spirals

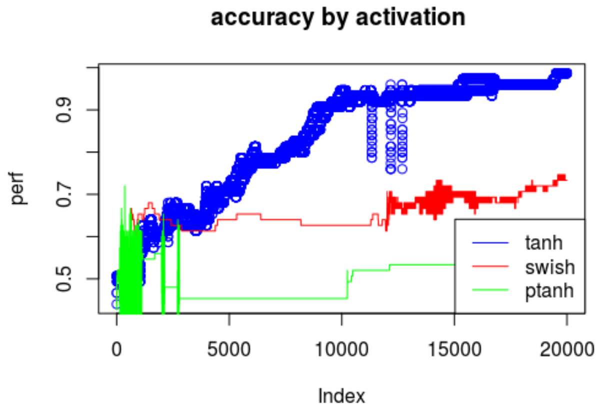
```
nh=30; lr0=1;
input_layer <- layer_input(shape = 2, name = 'input')
hidden_layer<-   layer_dense(input_layer,units = nh,
                      activation = 'tanh',
                      bias_initializer=
                      initializer_random_uniform(
                      minval = -0.1, maxval = 0.1,
                      seed = 104),
                      kernel_initializer=
                      initializer_random_normal(
                      mean=0,stddev=.1, seed = 104))
output_layer <- layer_dense(hidden_layer ,units = 1,
                      activation='sigmoid',
                      bias_initializer=
                      initializer_random_uniform(
                      minval = -0.1, maxval = 0.1,
                      seed = 104),
                      kernel_initializer=
                      initializer_random_normal
                      (mean=0,stddev=.1, seed = 104))
```

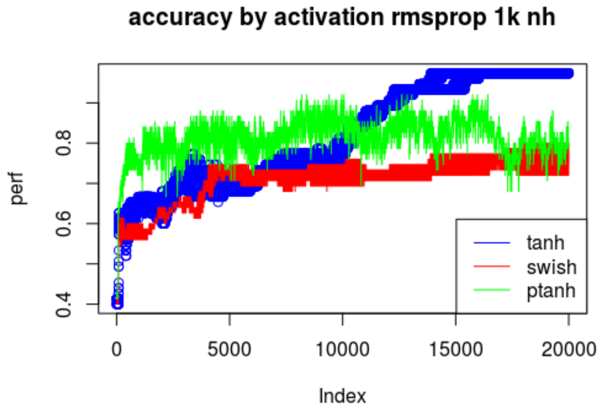
Binary continued

```
model_logit <- keras_model(inputs = input_layer,
                           outputs = output_layer )

#,clipnorm=1,,clipvalue=1 for exploding gradient options
opt <- optimizer_sgd(lr = lr0,momentum=0)
compile(model_logit,
        optimizer = opt,
        loss = "binary_crossentropy",
        metrics = c("acc")
)
summary(model_logit)
results_binary=fit(model_logit,X,Y,
                   epochs=20000,verbose=0,batch_size = 75)
```

Comparison of tanh vs swish vs penalized tanh (Spiral binary)





MNIST example

```
Y=to_categorical(y,10)
testX=t(mnist[["test"]]$x)/255.
testY=mnist[["test"]]$y
nh=30;lr=.15;
input_layer <- layer_input(shape = 784, name = 'input')
hidden_layer<-   layer_dense(input_layer,units = nh,
                 activation = 'tanh',
                 bias_initializer=
                 initializer_random_uniform(
                 minval = -0.1, maxval = 0.1, seed = 104),
                 kernel_initializer=
                 initializer_random_normal(
                 mean=0,stddev=.1,
                 seed = 104))
```

MNIST continued

```
output_layer <- layer_dense(hidden_layer ,units = 10,
                            activation='sigmoid',
                            bias_initializer=
                            initializer_random_uniform(
                            minval = -0.1, maxval = 0.1,
                            seed = 104),
                            kernel_initializer=
                              initializer_random_normal(
                              mean=0,stddev=.1,
                              seed = 104))
model=keras_model(
  inputs = input_layer,  outputs = output_layer )
opt <- optimizer_sgd(lr = lr0,momentum=0)
compile(model_logit,
        optimizer = opt,
        loss = "categorical_crossentropy",
        metrics = c("acc")
)
```

MNIST deeper

```
# 2 hidden layer  mnist and tsne of each layer
lr=lr0=.1
nh=30
input_layer <- layer_input(shape = 784, name = 'input')
hidden_layer<-   layer_dense(input_layer,name='h1',
                             units = nh, activation = 'relu'
)                             kernel_initializer=
                  initializer_random_normal(
                    mean=0,stddev=.1, seed = 104))
hidden_layer2<-   layer_dropout( layer_dense(
                                  hidden_layer,name='h2',
                                  units = 1, activation = 'softmax'
                                  ), rate=.4,name='h3')
```

Deep net example continued

```
output_layer <-layer_dense(hidden_layer ,
                           units = 10,
                           activation='softmax',
                            bias_initializer=
                             initializer_random_uniform(
                               minval = -0.1, maxval = 0.1,
                               seed = 104),
                            kernel_initializer=
                             initializer_random_normal(
                               mean=0,stddev=.1, seed = 104))
model_2h <- keras_model(inputs = input_layer,
                        outputs = output_layer )

opt <- optimizer_sgd(lr = lr0,momentum=0)
compile(model_2h,
        optimizer = opt,
        loss = "categorical_crossentropy",
        metrics = c("acc")
)
summary(model_2h)
results_2h=fit(model_2h,t(X),Y,epochs=10,verbose=0,batch_size = 128)
```

data processing

- Keras has nice data generator functions for working with images and text
- Also functions to resize and shape images

```
# Rescale the pixel value for the 2 splits
train_datagen <- image_data_generator(rescale = 1 / 255)
validation_datagen <- image_data_generator(rescale = 1 / 255)

train_generator <- flow_images_from_directory(
  train_dir,
  train_datagen,
  target_size = c(150, 150), # resizes all images to 150x150
  batch_size = 20,
  class_mode = "binary"
)
```

data processing-augmentation

Example with ImageDataGenerator

```
datagen = ImageDataGenerator(
        rotation_range=40,width_shift_range=0.2,
        height_shift_range=0.2, rescale=1./255,
        shear_range=0.2, zoom_range=0.2,
        horizontal_flip=True,fill_mode='nearest')
```

- rotation_range is a value in degrees (0-180), a to randomly rotate pictures
- width_shift and height_shift are ranges to randomly translate pictures vertically or horizontally
- rescale is a value by which we will multiply the data before any other processing. scaling 255 to be between 0/1
- shear_range is for randomly applying shearing transformations
- zoom_range is for randomly zooming inside pictures
- horizontal_flip is for randomly flipping half of the images horizontally

Regularization

There are 3 types of regularizers in Keras:

```
kernel_regularizer: applied to the kernel weights matrix.
bias_regularizer: applied to the bias vector.
activity_regularizer: applied to the output of the layer (its "activation").
```

```
layer_dense(input_layer,units = num_hidden_units,
            activation = act1,
            kernel_initializer
             =kernel_initializer,
            bias_initializer
             =bias_initializer,
            kernel_regularizer
            =kernel_regularizer )

# e.g. kernel_regularizer = regularizer_l2(.000000001)
```

- Dropout is implemented as additional layer e.g. layer_dropout(, rate=.4)

custom learning schedule and call backs

- Besides using optimizers you can also provide a custom learning schedule using a call back function in fit

```
#custom learning rate schedule
lr_schedule <- function(epoch,lr) {  lr0/(1+(epoch/epochs)) }
lr_reducer <- callback_learning_rate_scheduler(lr_schedule)
callbacks= list(
 callback_lambda( on_epoch_end = function(epoch, logs=list()) {
   if (epoch %% 1000==0){
     cat("Epoch End\n");
     print(paste('epoch:',epoch))
     wgts=get_weights(model_simple)
     print(paste('wgt norm:',sum(unlist(wgts)^2),
                 'gradient norm:',
                 get_grad(model_simple,train_data),
                 'loss:',logs[["loss"]])) }
 }),
 callback_terminate_on_naan(),
 lr_reducer,
 callback_reduce_lr_on_plateau(monitor = "loss", factor = .5)  )
```

Other useful callbacks

-Logging

- Callbacks can do logging and also be used for early stopping: callback_early_stopping(monitor = "val_loss", min_delta = 0)
- callback_tensorboard(log_dir = "logs/run_b") so you can try different runs and name them and compare in tensorboard if you use tensorboard callback to store results

-Early callback

Example of early stopping. There are some parameters:

```
monitor - quantity to be monitored
min_delta -- minimum change in the monitored quantity
patience -- number of epochs with no improvement after which to stop
```

Defaults in Keras

- defaults: adam is default optimizer, batch size of 32 is default

-in fit you can provide validation_split proportion or instead give a validation_data (validation_data overrides split)

- logging is important as verbose can become slow to plot lots of iterations
- plot the history from fit
- default weight initialization $U[-\frac{\sqrt{6}}{(nu_{in}+nu_{out})}, \frac{\sqrt{6}}{(nu_{in}+nu_{out})}]$
- Glorot uniform or Xavier
- default bias all 0s

Based on Understanding the difficulty of training deep feedforward neural networks)

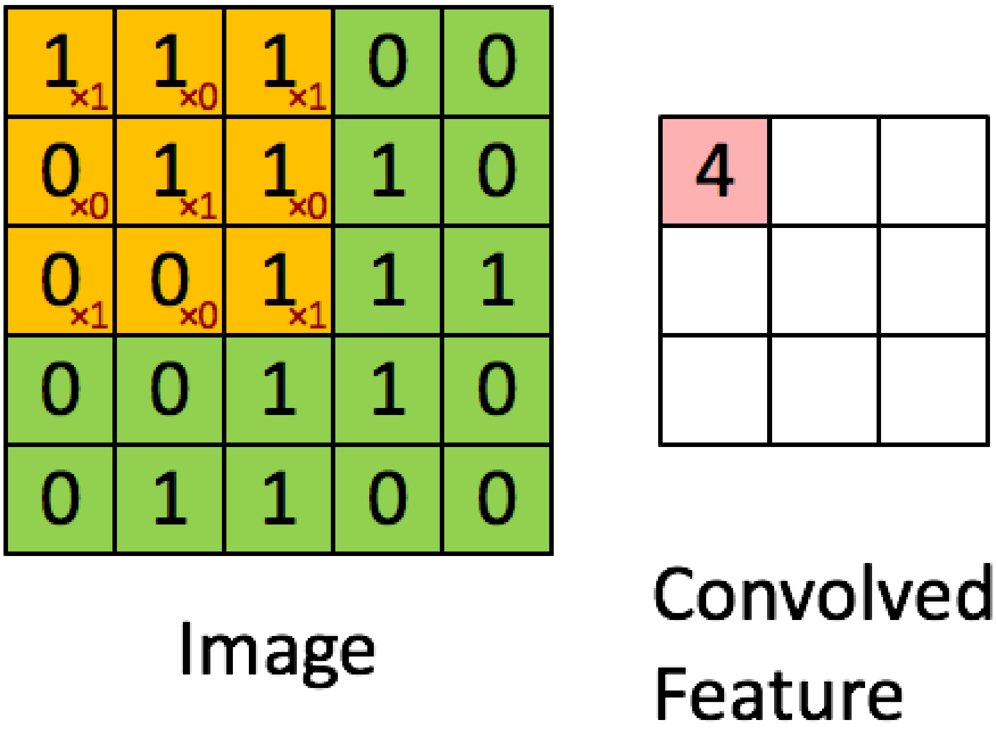
pre-trained models

- Keras allows you to be able to easily plug and re-used pre-trained models
- for images like vgg or inception etc.
- for text embeddings can be used to pre-train
- 2 options:
    1. freeze original layers and and train classifier at end
    2. unfreeze and re-train but that for small data and without computational resources is prohibitive

Keras allows you to try and mix different architectures

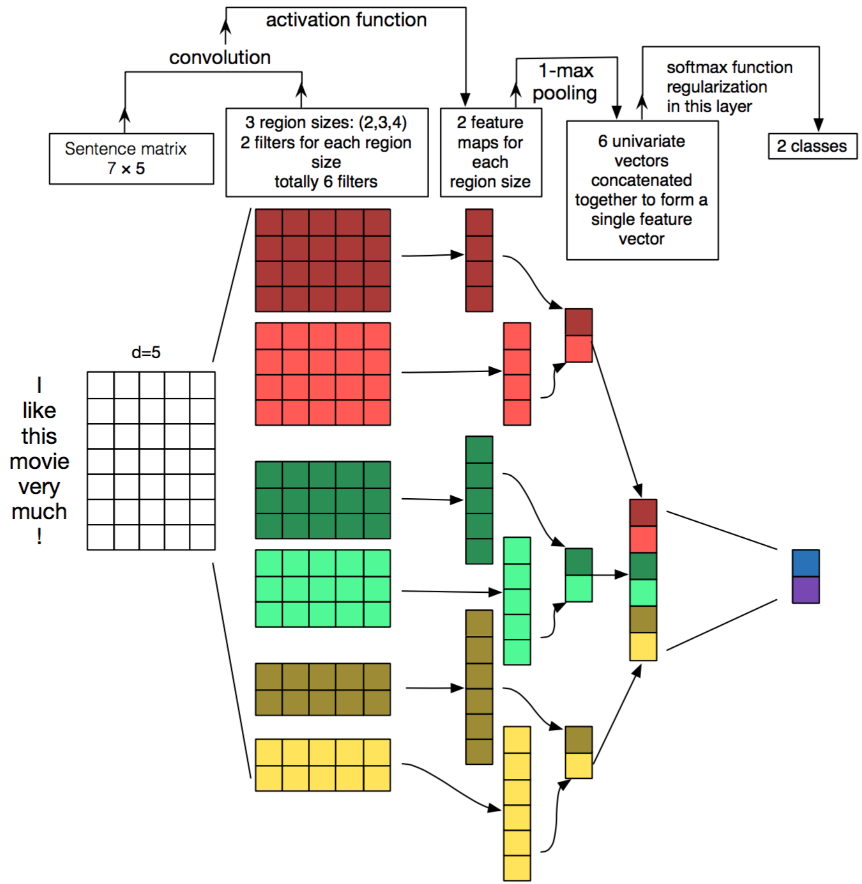
- Convolution nets have been successful for images (99.2% for MNIST; tutorial MNIST CNN Keras)
- Convolution and Recurrent neural networks like LTSM, GRU

Convolution and Pooling example



]

Convolution net for text



Text processing functions

- besides image classification deep learning has been a success for unstructured text
- Working with IMDB sentiment data
- original Stanford paper got 88.89% accuracy

Bag of words meets popcorn imdb data(2011)

Approaches to text

- Bag of words (create a column for each word with counts of word occurrence)
- simple loses order of words
- more involved to create n-grams (bi-gram; all possible 2 words next to each other and count)
- vectorize words into unique numbers and pad text

Text preparation

- remove stop words (the, and, a)
- standardize case
- stem (e.g. driven, drove, and drive to the stem drive)
- term document frequency matrix (counts of words)
- remove sparse terms (e.g. drop words that occur <2% of time)

Embedding

- a layer in keras that can reduce vocabulary to dense representation while preserving geometry of words in vector space
- embedding layer as has 3 inputs (input_dim: This is the size of the vocabulary in the text data, input_length: This is the length of input sequences)

Some results on Imdb

- simple bag of words gets .834 with logistic regression
- adding bi-gram features and bag of words with embedding layer and 5 epochs get 90.5% Tutorial on n-gram+embedding
    - virtual adversarial learning (perturbing embeddings inside net) achieved 94%; Adversarial Training for semi-supervised text classification (2017))
- 95% has been achieved with using large scale pre-trained embeddings using wiki large corpus with different learning rates for layers (triangle up and down schedules) Universal Language Model Fine-tuning for Text Classification (2018)