2010-2014: a new deep learning toolkit is released every 47 days. 2015: every 22 days. tensorflow & caffe top github

![Graph showing the popularity of different deep learning toolkits on GitHub from 2010 to 2015. The toolkits are ranked by the number of stars they have received.]
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Write custom layer/loss function
- Design your own architecture
Deep Learning Software

Use web API
Use pre-trained model
Fine-tune pre-trained model
Train your own model
Write custom layer/loss function
Design your own architecture
Use web API

Microsoft Azure cognitive services

https://azure.microsoft.com/en-us/services/cognitive-services/

Google Cloud AI

https://cloud.google.com/products/machine-learning/
Microsoft/Google API Demo
Use web API: pros and cons

● GOOD:
  ○ State-of-the-art models trained on huge datasets you would never have access to.
  ○ Very simple to integrate, does not require any deep learning knowledge.
  ○ Free trial versions.
  ○ Ideal for prototyping - if it proves useful, you can always run your own.

● BAD:
  ○ Bandwidth requirements if your dataset is big (think videos).
  ○ Latency issues, when you need near real-time performance.
  ○ If your dataset is sensitive and you cannot upload it.
  ○ Maybe you just don’t have the money?
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Write custom layer/loss function
- Design your own architecture
Use pre-trained model

- **Keras** (only image classification models)
  [https://keras.io/applications/](https://keras.io/applications/)
- **TensorFlow** (only image classification models)
  [https://github.com/tensorflow/models/tree/master/research/slim#Pretrained](https://github.com/tensorflow/models/tree/master/research/slim#Pretrained)
- **PyTorch** (only image classification models)
  [http://pytorch.org/docs/master/torchvision/models.html](http://pytorch.org/docs/master/torchvision/models.html)
- **Caffe** (most comprehensive selection)
  [https://github.com/BVLC/caffe/wiki/Model-Zoo](https://github.com/BVLC/caffe/wiki/Model-Zoo)
- **Darknet** (image classification and real-time object detection)
- **Converters**
  [https://github.com/ysh329/deep-learning-model-convertor](https://github.com/ysh329/deep-learning-model-convertor)
from keras.applications.resnet50 import ResNet50, preprocess_input, decode_predictions
from keras.preprocessing import image

# load model and pre-trained weights
model = ResNet50(weights='imagenet')

# load image
img = image.load_img('elephant.jpg', target_size=(224, 224))

# preprocess image
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

# feed-forward pass through model
preds = model.predict(x)

print('Predicted:', decode_predictions(preds, top=3)[0])

Predicted: [('n02504458', 'African_elephant', 0.57238054), ('n01871265', 'tusker', 0.34006864), ('n02504013', 'Indian_elephant', 0.087232716)]
Keras pre-trained / segmentation demo
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Write custom layer/loss function
- Design your own architecture
Fine-tune pre-trained model

1. Train on Imagenet

2. Small dataset: 
   feature extractor
   Freeze these

3. Medium dataset: 
   finetuning
   more data = retrain more of 
   the network (or all of it)
   Freeze these
   Train this

NB! Convolutional layers do not depend on input image size!!!
When to use which option?

- **Small dataset, similar data**
  - Just train a new classification layer (or standalone classifier on extracted features).

- **Small dataset, dissimilar data**
  - Train a new classifier on top of lower convolutional layers.

- **Big dataset, similar data**
  - Replace classification layer and fine-tune all layers.

- **Big dataset, dissimilar data**
  - Train a network from scratch. Might be good idea to initialize couple of lower convolutional layers from pre-trained network.

from keras.applications import VGG16
from keras.models import Sequential
from keras.layers import Flatten, Dense

# Load pre-trained model without classification layers (include_top=False)
cnv_base = VGG16(weights='imagenet', include_top=False,
                 input_shape=(150, 150, 3))

# Freeze pre-trained layers
cnv_base.trainable = False

# Add our own classification layers
model = Sequential()
model.add(cnv_base)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
from keras.applications import VGG16
from keras.models import Sequential
from keras.layers import Flatten, Dense

# Load pre-trained model without classification layers (include_top=False)
conv_base = VGG16(weights='imagenet', include_top=False,
                  input_shape=(150, 150, 3))

# Freezing all layers up to a specific one
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    layer.trainable = set_trainable

# Add our own classification layers
model = Sequential()
model.add(conv_base)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
Keras fine-tuning demo
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Write custom layer/loss function
- Design your own architecture
The retirees...

The alternatives...

The cool kids...
<table>
<thead>
<tr>
<th>Rank</th>
<th>Popularity</th>
<th>Repository</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>377.51</td>
<td>tensorflow/tensorflow</td>
<td>Java</td>
</tr>
<tr>
<td>2</td>
<td>174.15</td>
<td>fchollet/keras</td>
<td>Python</td>
</tr>
<tr>
<td>3</td>
<td>143.84</td>
<td>BVLC/caffe</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>128.26</td>
<td>dmlc/mxnet</td>
<td>Go</td>
</tr>
<tr>
<td>5</td>
<td>72.85</td>
<td>Theano/Theano</td>
<td>Pascal</td>
</tr>
<tr>
<td>6</td>
<td>69.32</td>
<td>Microsoft/CNTK</td>
<td>C#</td>
</tr>
<tr>
<td>7</td>
<td>67.30</td>
<td>deeplearning4j/deep learning4j</td>
<td>Mono</td>
</tr>
<tr>
<td>8</td>
<td>61.54</td>
<td>baidu/paddle</td>
<td>Erlang</td>
</tr>
<tr>
<td>9</td>
<td>54.07</td>
<td>pytorch/pytorch</td>
<td>Scala</td>
</tr>
<tr>
<td>10</td>
<td>29.65</td>
<td>pfnet/chainer</td>
<td>Haskell</td>
</tr>
<tr>
<td>11</td>
<td>29.35</td>
<td>torch/torch7</td>
<td>Lisp</td>
</tr>
<tr>
<td>12</td>
<td>29.33</td>
<td>NVIDIA/DIGITS</td>
<td>BASIC</td>
</tr>
<tr>
<td>13</td>
<td>28.42</td>
<td>tflearn/tflearn</td>
<td>Spring/Spark/Struts</td>
</tr>
<tr>
<td>14</td>
<td>28.09</td>
<td>caffe2/caffe2</td>
<td>C++</td>
</tr>
<tr>
<td>15</td>
<td>21.41</td>
<td>davisking/dlib</td>
<td>R</td>
</tr>
</tbody>
</table>
Q: How do I know what architecture to use?

A: don’t be a hero.

1. Take whatever works best on ILSVRC (latest ResNet)
2. Download a pretrained model
3. Potentially add/delete some parts of it
4. Finetune it on your application.
Q: How do I know what hyperparameters to use?

A: don’t be a hero.

- Use whatever is reported to work best on ILSVRC.
- Play with the regularization strength (dropout rates)
## Hyperparameters

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Starting value</th>
<th>Range / other options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rate</td>
<td>0.001</td>
<td>$10^{-5}$ … $10^{-1}$, powers of 10</td>
</tr>
<tr>
<td>Number of hidden layers</td>
<td>1</td>
<td>1, 3, 5, …, 101</td>
</tr>
<tr>
<td>Number of hidden nodes</td>
<td>128</td>
<td>32 … 4096, powers of 2</td>
</tr>
<tr>
<td>Conv. filter size</td>
<td>3</td>
<td>3, 5, 7, 11, at least two successive conv. layers with 3</td>
</tr>
<tr>
<td>Conv. number of filters</td>
<td>32</td>
<td>16 … 128, powers of 2</td>
</tr>
<tr>
<td>L2 regularization</td>
<td>0</td>
<td>$10^{-5}$ … $10^{-1}$, powers of 10</td>
</tr>
<tr>
<td>Dropout</td>
<td>0</td>
<td>0.1, 0.25, 0.5</td>
</tr>
<tr>
<td>Activation function</td>
<td>relu</td>
<td>tanh, LeakyReLU, PReLU, ELU</td>
</tr>
<tr>
<td>Weight initialization</td>
<td>glorot_uniform (xavier)</td>
<td>he_normal</td>
</tr>
<tr>
<td>Mini-batch size</td>
<td>32</td>
<td>1-256, powers of 2, GPU mem!</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
<td>RMSProp, SGD+Momentum</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.9</td>
<td>0.5, 0.9, 0.95, 0.99</td>
</tr>
</tbody>
</table>
Hyperparameter search

- **Manual coordinate descent a.k.a. informal hyperparameter search**
  - Find the best value for one hyperparameter, then pick the next.
  - Beware of correlations, i.e. learning rate and regularization!

- **Grid search**
  - Try all possible combinations.
  - Combinatorial explosion!

- **Random search**
  - If you try 4 values for 2 hyperparameters, you make $4 \times 4 = 16$ evaluations. What if one HP was completely unimportant? Then you learn only about 4 different values for the other parameter.
  - If you instead sample randomly from given range, you get evaluations at 16 different values of both hyperparameters.

- **Hyperopt / Spearmint / BayesOpt / SigOpt / HyperBand / AutoML**
  - Might be useful to establish initial range or fine-tune manually chosen values.
  - No silver bullet! At some point the curse of dimensionality kicks in.
  - About 5 HPs should be fine, some claim success with 20.
Monitoring - use TensorBoard

TensorBoard

Scalars | Images | Audio | Graphs | Distributions | Histograms | Embeddings | Text

Write a regex to create a tag group
Cross:

Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default

Smoothing
0.6

Horizontal Axis
STEP | RELATIVE | WALL

Runs
Write a regex to filter runs

n_samples_1/20170530_141631
n_samples_5/20170530_141605

TOGGLE ALL RUNS

Parameter

loss

loss

loss/kl_penalty

loss/log_lik
Monitoring

Handling of different runs - save each run to separate folder, so that they can be compared in TensorBoard.

Save weights regularly and make sure your code can resume training from saved weights.

In Keras:

model.fit(X_train, y_train, batch_size=32, epochs=10, callbacks=[
    callbacks.TensorBoard(log_dir=log_dir),
    callbacks.ModelCheckpoint(save_path, monitor='val_acc',
    verbose=1, save_best_only=True)]

Loading model:

from keras.models import load_model
model = load_model(model_path)
Hardware: university resources

- **Rocket** ([https://www.hpc.ut.ee/](https://www.hpc.ut.ee/))
  - **falcon1**: 2 GPUs with 16GB of VRAM each
  - **falcon2**: 8 GPUs with 12GB of VRAM each

Send e-mail to support@hpc.ut.ee with your username and supervisor to get access!

- **EE Net grid** ([http://www.eenet.ee/EENet/grid.html](http://www.eenet.ee/EENet/grid.html))
  - 7x Nvidia Tesla K20m GPGPU (2496 cores, 5 GB GDDR5 VRAM)
  - 1x Nvidia Tesla K40m GPGPU (2880 cores, 12GB GDDR5 VRAM)

Sign up at [https://taat.grid.ee/](https://taat.grid.ee/), you just need UT account and SSH public key.
See nodes and partitions in the cluster:
sinfo

See currently queued jobs:
squeue
squeue -u myusername

Run a new job in Rocket cluster:
srun -p gpu --gres=gpu:tesla:1 -t 3600 python train_cats_and_dogs.py

Check GPU usage (on GPU node only!)
nvidia-smi
Hardware: service providers

**Amazon EC2** *(https://aws.amazon.com/ec2/instance-types/p2/)*

<table>
<thead>
<tr>
<th>Name</th>
<th>GPUs</th>
<th>vCPUs</th>
<th>RAM (GiB)</th>
<th>Network Bandwidth</th>
<th>Price/Hour*</th>
<th>RI Price / Hour**</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2.xlarge</td>
<td>1</td>
<td>4</td>
<td>61</td>
<td>High</td>
<td>$0.900</td>
<td>$0.425</td>
</tr>
<tr>
<td>p2.8xlarge</td>
<td>8</td>
<td>32</td>
<td>488</td>
<td>10 Gbps</td>
<td>$7.200</td>
<td>$3.400</td>
</tr>
<tr>
<td>p2.16xlarge</td>
<td>16</td>
<td>64</td>
<td>732</td>
<td>20 Gbps</td>
<td>$14.400</td>
<td>$6.800</td>
</tr>
</tbody>
</table>

Apply for credits at [www.awseducate.com](http://www.awseducate.com)!


<table>
<thead>
<tr>
<th>INSTANCE</th>
<th>CORES</th>
<th>RAM</th>
<th>TEMPORARY STORAGE</th>
<th>GPU</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC6 v2</td>
<td>6</td>
<td>112.00 GiB</td>
<td>336 GiB</td>
<td>1X P100</td>
<td>$0.90/hour</td>
</tr>
<tr>
<td>NC12 v2</td>
<td>12</td>
<td>224.00 GiB</td>
<td>672 GiB</td>
<td>2X P100</td>
<td>$1.80/hour</td>
</tr>
<tr>
<td>NC24 v2</td>
<td>24</td>
<td>448.00 GiB</td>
<td>1,344 GiB</td>
<td>4X P100</td>
<td>$3.60/hour</td>
</tr>
<tr>
<td>NC24r v2</td>
<td>24</td>
<td>448.00 GiB</td>
<td>1,344 GiB</td>
<td>4X P100</td>
<td>$3.96/hour</td>
</tr>
</tbody>
</table>
Hardware: build yourself

- NVidia DevBox
  https://developer.nvidia.com/devbox

- Building a Deep Learning (Dream) Machine
  http://graphific.github.io/posts/building-a-deep-learning-dream-machine/

- Hardware Guide: Neural Networks on GPUs

Demo: running code on HPC, monitoring with TensorBoard
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- **Write custom layer/loss function**
- Design your own architecture
Google FaceNet
1. Pass three face images through convolutional network: anchor image, image of the same person, and image of another person.
2. Make the distance between anchor and same person vectors smaller than distance between anchor and another person vectors (by some margin).

**Triplet loss**

\[
\sum_{i}^{N} \left[ \left\| f(x^a_i) - f(x^p_i) \right\|_2^2 - \left\| f(x^a_i) - f(x^n_i) \right\|_2^2 + \alpha \right]_+ 
\]
FaceNet architecture

- Anchor image
  - CNN
  - Fully-connected
  - L2 normalization
  - Anchor embedding

- Positive image
  - CNN
  - Fully-connected
  - L2 normalization
  - Posit. embedding

- Negative image
  - CNN
  - Fully-connected
  - L2 normalization
  - Negat. embedding

- Triplet loss
  \[ \sum_{i}^{N} \left[ \| f(x_i^a) - f(x_i^p) \|_2^2 - \| f(x_i^a) - f(x_i^n) \|_2^2 + \alpha \right]_+ \]
input_tensor = Input(shape=(250, 250, 3))
resnet50 = ResNet50(input_tensor=input_tensor, weights='imagenet', include_top=False)
resnet50.trainable = False

model_embed = Sequential()
model_embed.add(resnet50)
model_embed.add(Flatten())
model_embed.add(Dense(128))
model.add(Lambda(lambda x: x / K.sqrt(K.sum(x**2, axis=-1, keepdims=True))))

xa = Input(shape=(250, 250, 3))
ea = model_embed(xa)

xp = Input(shape=(250, 250, 3))
ep = model_embed(xp)

xn = Input(shape=(250, 250, 3))
en = model_embed(xn)

alpha = 1.
def triplet_loss(x):
    anchor_embed, pos_embed, neg_embed = x
    dists_pos = K.sum((anchor_embed - pos_embed)**2, axis=-1)
    dists_neg = K.sum((anchor_embed - neg_embed)**2, axis=-1)
    return K.maximum(dists_pos - dists_neg + alpha, 0.)

loss = Lambda(triplet_loss, output_shape=(1, ))([ea, ep, en])

model_train = Model(inputs=[xa, xp, xn], outputs=[loss])
model_train.compile(loss=lambda y_true, y_pred: y_pred, optimizer='adam')
input_tensor = Input(shape=(250, 250, 3))
resnet50 = ResNet50(input_tensor=input_tensor, weights='imagenet',
                   include_top=False)
resnet50.trainable = False

alpha = 1.
def triplet_loss(x):
    n = K.shape(x)[0] // 3
    anchor_embed = x[:n]
pos_embed = x[n:2*n]
neg_embed = x[2*n:]
dists_pos = K.sum((anchor_embed - pos_embed)**2, axis=-1)
dists_neg = K.sum((anchor_embed - neg_embed)**2, axis=-1)
return K.mean(K.maximum(dists_pos - dists_neg + alpha, 0.))

model = Sequential()
model.add(resnet50)
model.add(Flatten())
model.add(Dense(128))
model.add(Lambda(lambda x: x / K.sqrt(K.sum(x**2, axis=-1, keepdims=True)))))
model.add(Lambda(triplet_loss))
model.compile(loss=lambda y_true, y_pred: y_pred, optimizer='adam')
model.summary()
In Keras

- Custom loss functions can be implemented using Keras backend.
  - Make yourself comfortable with Keras backend functions. They are mostly similar to Numpy, but they construct graph instead of performing computation!

- Simple computational layers can be implemented using Lambda wrapper in Keras.

- For custom weight layers you have to write your own Keras layer class.

- If your model does not have target values, then you need to hack around.
  - Keras assumes the network always has inputs and outputs, and outputs have target values. It is not the case with many advanced models.
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Write custom layer/loss function
- Design your own architecture
Sinusoid approximation

\[ y = \sin(x) \]

Sigmoid non-linearity after hidden nodes.
Can it learn from noisy inputs?
How well it works outside training data?
What hidden nodes are actually doing?
Main points

● Don’t think in layers and neurons, think about computational graph.
  ○ Additive neurons can approximate large class of functions, but not all.

● Have a really good understanding of your problem. What kind of computation would solve it, if you somehow had the correct weights?
  ○ Naively throwing deep networks at the problem just gets you so far.

● Pick/design a differentiable (in a weak sense) loss function.
  ○ Negative log-likelihood is always a good candidate, whatever the underlying probability distribution is.

● Optimize the hell out of it using gradient descent.
  ○ Use all tips and tricks learned from this course.
SOFTWARE

GOOD:

- Google support.
- A lot of tutorials and examples.
- Backpropagation for most basic ops already implemented.

BAD:

- Verbose
- Big and clunky?
- Slow compile step?

GOOD:

- Research community support.
- Many latest models implemented.
- Control over all aspects, easy debugging.

BAD:

- Slightly archaic, not as polished as TensorFlow/Keras.
- Have to manually implement backprop for custom layers.

For thorough comparison see https://docs.chainer.org/en/stable/comparison.html
Symbolic differentiation

1. Define the graph with backpropagatable ops.
2. Compile the graph into executable (GPU) code.
3. Run the graph, calculate gradients, update weights.
4. Gradient calculation is also a graph!!!

Define-by-run

1. Run the layers one-by-one, dependency graph is recorded on the go.
2. When calculating gradients, the dependency graph is traversed backwards.
3. Update weights.
4. Immediate feedback when there is error in computation.

For thorough comparison see [https://mxnet.incubator.apache.org/architecture/program_model.html](https://mxnet.incubator.apache.org/architecture/program_model.html)
Questions?

tambet.matiisen@ut.ee