Software and Practical Methodology
2010-2014: a new deep learning toolkit is released every 47 days. 2015: every 22 days. tensorflow & caffe top github
Number of papers on arxiv.org that mention a given framework

Source: Data from RISELab; graphic from gradientflow.com
Deep Learning Software

Use web API
Use pre-trained model
Fine-tune pre-trained model
Train your own model
Custom layers/losses/architectures
Research on neural networks
Deep Learning Software

Use web API

Use pre-trained model

Fine-tune pre-trained model

Train your own model

Custom layers/losses/architectures

Research on neural networks
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/

AWS AI & ML Services
https://aws.amazon.com/machine-learning/
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.

- **BAD:**
  - Bandwidth requirements if your data 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?

- **Use-cases:**
  - Quick deployment.
  - Short term cost of production.
  - Ideal for prototyping - if it proves useful, you can always run your own.
Deep Learning Software

Use web API

Use pre-trained model

Fine-tune pre-trained model

Train your own model

Custom layers/losses/architectures

Research on neural networks
Use pre-trained model

- **Keras** (only image classification models)
  [https://keras.io/applications/](https://keras.io/applications/)
- **TensorFlow**
  [https://www.tensorflow.org/hub](https://www.tensorflow.org/hub)
- **PyTorch**
  [https://pytorch.org/hub/](https://pytorch.org/hub/)
- **Caffe** (comprehensive selection, but outdated)
  [https://github.com/BVLC/caffe/wiki/Model-Zoo](https://github.com/BVLC/caffe/wiki/Model-Zoo)
- **Model Zoo**
  [https://modelzoo.co/](https://modelzoo.co/)
- **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)
preds = decode_predictions(preds, top=3)[0]

print('Predicted:', preds)

Predicted: [('n02504458', 'African_elephant', 0.57238054), ('n01871265', 'tusker', 0.34006864), ('n02504013', 'Indian_elephant', 0.087232716)]
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Custom layers/losses/architectures
- Research on neural networks
Fine-tune pre-trained model

1. Train on ImageNet
2. Small dataset: feature extractor
   - Freeze these
   - Train this
3. Medium dataset: finetuning
   - more data = retrain more of the network (or all of it)
   - Freeze these
   - Train this

NB! You can also use network as a feature extractor and train classifier on features!

Slide from Andrej Karpathy, Bay Area Deep Learning School 2016
## When to use which option?

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Similar data</th>
<th>Dissimilar data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small dataset</td>
<td>Just train a new classification layer (or standalone classifier on extracted features).</td>
<td>Train a new classifier on top of lower convolutional layers.</td>
</tr>
<tr>
<td>Big dataset</td>
<td>Replace classification layer and fine-tune all layers.</td>
<td>Train a new network from scratch. Might be good idea to initialize couple of lower convolutional layers from pre-trained network.</td>
</tr>
</tbody>
</table>

```python
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))

# Freeze pretrained layers
conv_base.trainable = False

# Add our own classification layers
model = Sequential()
model.add(conv_base)
model.add(Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(Dense(1000, activation='softmax'))
```

2. Small dataset: **feature extractor**

Freeze these

Train this
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(layers.Dense(256, activation='relu'))
model.add(Dense(1000, activation='softmax'))
Fine-tuning process

1. Add your custom network on top of an already trained base network.
2. Freeze the base network.
3. Train the part you added.
4. Unfreeze some layers in the base network.
5. Jointly train both these layers and the part you added.
Deep Learning Software

Use web API
Use pre-trained model
Fine-tune pre-trained model
Train your own model
Custom layers/losses/architectures
Research on neural networks
Software

The cool kids...
- TensorFlow
- Keras
- PyTorch

The alternatives...
- mxnet
- CNTK
- Caffe2
- Chainer

The retirees...
- Caffe
- theano
- Torch
- Neon
## Aggregate Popularity

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>377.51</td>
<td>tensorflow/tensorflow</td>
</tr>
<tr>
<td>#2</td>
<td>174.15</td>
<td>fchollet/keras</td>
</tr>
<tr>
<td>#3</td>
<td>143.84</td>
<td>BVLC/caffe</td>
</tr>
<tr>
<td>#4</td>
<td>128.26</td>
<td>dmlc/mxnet</td>
</tr>
<tr>
<td>#5</td>
<td>72.85</td>
<td>Theano/Theano</td>
</tr>
<tr>
<td>#6</td>
<td>69.32</td>
<td>Microsoft/CNTK</td>
</tr>
<tr>
<td>#7</td>
<td>67.30</td>
<td>deeplearning4j/deeplearning4j</td>
</tr>
<tr>
<td>#8</td>
<td>61.54</td>
<td>baidu/paddle</td>
</tr>
<tr>
<td>#9</td>
<td>54.07</td>
<td>pytorch/pytorch</td>
</tr>
<tr>
<td>#10</td>
<td>29.65</td>
<td>pfnet/chainer</td>
</tr>
<tr>
<td>#11</td>
<td>29.35</td>
<td>torch/torch7</td>
</tr>
<tr>
<td>#12</td>
<td>29.33</td>
<td>NVIDIA/DIGITS</td>
</tr>
<tr>
<td>#13</td>
<td>28.42</td>
<td>tflearn/tflearn</td>
</tr>
<tr>
<td>#14</td>
<td>28.09</td>
<td>caffe2/caffe2</td>
</tr>
<tr>
<td>#15</td>
<td>21.41</td>
<td>davisking/dlib</td>
</tr>
</tbody>
</table>

Image from François Chollet, [https://twitter.com/fchollet/status/915366704401719296](https://twitter.com/fchollet/status/915366704401719296)
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>$10^{-5} \ldots 10^{-1}$, powers of 10</td>
</tr>
<tr>
<td>Number of hidden layers</td>
<td>1</td>
<td>0, \ldots, 101</td>
</tr>
<tr>
<td>Number of hidden nodes</td>
<td>128</td>
<td>32 \ldots 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 \ldots 128, powers of 2</td>
</tr>
<tr>
<td>L2 regularization</td>
<td>0</td>
<td>$10^{-5} \ldots 10^{-1}$, powers of 10</td>
</tr>
<tr>
<td>Dropout</td>
<td>0</td>
<td>0.1, 0.2, 0.3, 0.4, 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

- **Coordinate descent**
  - 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 4x4=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.
The process

**Start simple**
- Choose the simplest model & data possible (e.g., LeNet on a subset of your data)

**Implement & debug**
- Once model runs, overfit a single batch & reproduce a known result

**Evaluate**
- Apply the bias-variance decomposition to decide what to do next

**Tune hyp-eparams**
- Use coarse-to-fine random searches

**Improve model/data**
- Make your model bigger if you underfit; add data or regularize if you overfit

Slide from Josh Tobin, [Troubleshooting Deep Neural Networks](#)
Interlude: practicalities
Monitoring - use TensorBoard
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:

```python
model.fit(X_train, y_train, batch_size=32, epochs=10, callbacks=[
    callbacks.TensorBoard(log_dir=log_dir),
    callbacks.ModelCheckpoint(os.path.join(log_dir, 'model.h5'),
        monitor='val_acc', verbose=1, save_best_only=True)])
```

Loading model:

```python
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/))

6 GPU nodes, falcon1-6, purchase funded by Institute of Computer Sciences:

- 2 x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (48 cores total)
- 512 GB RAM
- 5TB of local SSD storage
- Infiniband:
  - Falcon 1-3 – 2x 40 Gbps each
  - Falcon 4-6 – 5x 100 Gbps each
- 24x NVIDIA Tesla V100 GPUs:
  - Falcon3 versions have 16 GB of VRAM.
  - Falcon 4-6 versions have 32 GB of VRAM.
Hardware: university resources

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

Logging in:
ssh username@rocket.hpc.ut.ee

Once logged in, peeking to GPU nodes:
ssh falcon1

See what GPUs are doing (only on GPU nodes):
nvidia-smi
Scheduling your jobs with SLURM

See nodes and partitions in the cluster:
```
sinfo
```

See currently queued jobs:
```
squeue
squeue -u myusername
squeue -p gpu
```

Run a new job in Rocket cluster:
```
srun -p gpu --gres=gpu:tesla:1 -t 3600 python train_cats_and_dogs.py
```
- partition gpu
- request one Tesla gpu
- your job takes max an hour
- NB! All environment variables are retained!
Hardware: free resources on the internet

**Google Colab** ([https://colab.research.google.com/](https://colab.research.google.com))
- Jupyter notebook, Tesla K80 GPU 12GB RAM or Google TPU, max 12 hours

**Kaggle Kernels** ([https://www.kaggle.com/kernels/](https://www.kaggle.com/kernels))
- Jupyter notebook, Tesla K80 GPU 11GB RAM, max 6 hours

**Google Cloud GPUs** ([https://cloud.google.com/gpu/](https://cloud.google.com/gpu))
- Tesla K80, P100, P4, T4, and V100 GPUs, $300 credit for free

**Amazon SageMaker** ([https://aws.amazon.com/sagemaker/](https://aws.amazon.com/sagemaker))
- Apply for free credits at [www.awseducate.com](http://www.awseducate.com)

**Azure Notebooks** ([https://notebooks.azure.com/](https://notebooks.azure.com))
- 4GB memory limit, no GPUs?
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/

- Which GPU(s) to Get for Deep Learning
  - https://timdettmers.com/2020/09/07/which-gpu-for-deep-learning/
Normalized 1 and 2-GPU Performance per Dollar

- GTX 1060 (6 GB)
- GTX 1070 (8 GB)
- GTX T1070 Ti (8 GB)
- GTX 1080 (8 GB)
- GTX 1080 Ti (11 GB)
- Titan X Pascal (12 GB)
- Titan Xp (12 GB)
- Titan V (12 GB)
- V100 (32 GB)
- V100 (16 GB)
- RTX 2060 Super (8 GB)
- RTX 2070 (8 GB)
- RTX 2070 Super (8 GB)
- RTX 2080 (8 GB)
- RTX 2080 Super (8 GB)
- RTX 2080 Ti (11 GB)
- Titan RTX (24 GB)
- RTX 6000 (24 GB)
- RTX 8000 (48 GB)
- RTX 3070 (8 GB)
- RTX 3080 (10 GB)
- RTX 3090 (24 GB)
- A100 (40 GB)

Normalized Performance per Dollar (Higher is Better)
Deep Learning Software

- Use web API
- Use pre-trained model
- Fine-tune pre-trained model
- Train your own model
- Custom layers/losses/architectures
- Research on neural networks
Google FaceNet

Schroff F., Kalenichenko D., Philbin J., FaceNet: A Unified Embedding for Face Recognition and Clustering
Triplet loss

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 the same person smaller than distance between anchor and another person (by some margin).

\[
\sum_{i}^{N} \left[ \| f(x_i^a) - f(x_i^p) \|_2^2 - \| f(x_i^a) - f(x_i^n) \|_2^2 + \alpha \right]_+
\]

Schroff F., Kalenichenko D., Philbin J., FaceNet: A Unified Embedding for Face Recognition and Clustering
FaceNet architecture

Anchor image

Positive image

Negative image

CNN

Fully-connected

L2 normalization

Anchor embedding

Posit. embedding

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(bodyParser)

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, 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()
Summary

- Use **Lambda layer** to implement simple computational steps
  - Familiarize yourself with keras.backend functions for this!

- Implement **custom loss functions** using keras.backend.
  - Need to be inventive to work around keras limitations.

- Write your own Layer class for **trainable weight layers**.
  - See this: [https://keras.io/layers/writing-your-own-keras-layers/](https://keras.io/layers/writing-your-own-keras-layers/)

- If your model does not have target values, 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
- Custom layers/losses/architectures
- Research on neural networks
Sinusoid approximation

\[ y = \sin(x) \]

\[ h_i = f(w_i x + b_i) \]

\[ y = \sum_i v_i h_i + c \]

\[ f(x) = \frac{1}{1 + e^{-x}} \]
Summary

- While neural networks can be universal approximators, they only **interpolate between training points**, they do not extrapolate.
  - Outside of training data they are generally useless.

- To generalize better, we need to **induce priors** in our networks.
  - Examples of successful priors are convolution (translation invariance), attention, recurrence, residual connections, etc.

- Do not think of network as sequence of layers, think of it as **differentiable computation**. What computation would solve given problem?
  - You can always figure out the unknown coefficients in this computation using gradient descent and backpropagation.
Automatic differentiation

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.

https://medium.com/data-science-at-microsoft/a-tale-of-two-frameworks-pytorch-vs-tensorflow-f73a975e733d
Automatic differentiation

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
Summary

- Feel free to use the available work (model, architecture, hyperparameters) and build on top of it.
- Start with the best reported defaults and then make changes to suit your problem.
- Simple solution is the best solution as long as it fulfills all the requirements.
- Well maintained code, experimental setup will go a long way in helping you to be more efficient and hence more successful.
Summary

• An important tip for any ML project, in whatever way you can, check what finally goes into your model (what are you training on?).

Questions?
tarunkhajuria42@gmail.com