Lecture 8:
Raising the level of abstraction for ML

Parallel Computing
Stanford CS348K, Spring 2020
Note

- Most of this class involved in-class discussion of the Ludwig and Overton papers

- I am posting these slides as some were used during parts of the discussion
Services provided by ML “frameworks”

- **Functionality:**
  - Implementations of wide range of useful operators
    - Conv, dilated conv, relu, softmax, pooling, separable conv, etc.
  - Implementations of various optimizers:
    - Basic SGD, with momentum, Adagrad, etc.
  - Ability to compose operators into large graphs to create models
  - Carry out back-propagation

- **Performance:**
  - High performance implementation of operators (layer types)
  - Scheduling onto multiple GPUs, parallel CPUs (and sometimes multiple machines)
  - Automatic sparsification and pruning

- **Meta-optimization:**
  - Hyper-parameter search
  - More recently: neural architecture search
TensorFlow/MX.Net 
data-flow graphs

- Key abstraction: a program is a DAG of (large granularity) operations that consume and product N-D tensors
Leveraging domain-knowledge: more efficient topologies (aka better algorithm design)

- **Original DNNs** for image recognition where over-provisioned
  - Large filters, many filters
- **Modern DNNs** designs are hand-designed to be sparser

**SqueezeNet:** [Iandola 2017] Reduced number of parameters in AlexNet by 50x, with similar performance on image classification.

---

Inception v1 (GoogleLeNet) — 27 total layers, 7M parameters

ResNet (34 layer version)
Modular network designs

Inception v4

Input (299x299x3)
Stem
4 x Inception-A
Output: 35x35x384
Reduction-A
Output: 17x17x1024
7 x Inception-B
Output: 17x17x1024
Reduction-B
Output: 8x8x1536
3 x Inception-C
Output: 8x8x1536
Avarage Pooling
Dropout (keep 0.8)
Output: 1536
Softmax
Output: 1000

1x1 Conv (96)
1x1 Conv (64)
3x3 Conv (96)
1x1 Conv (96)
3x3 Conv (64)
3x3 Conv (96)

1x1 Conv (128)
1x7 Conv (256)
7x1 Conv (256)
1x7 Conv (224)
7x1 Conv (224)
1x1 Conv (384)
1x1 Conv (224)
7x1 Conv (224)
1x1 Conv (192)
7x1 Conv (192)
1x1 Conv (192)

Filter concat
A block
B block

Figure 4. The schema for \(35 \times 35\) grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

Figure 5. The schema for \(17 \times 17\) grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

Figure 6. The schema for \(8 \times 8\) grid modules of the pure Inception-v4 network. This is the Inception-C block of Figure 9.

Figure 7. The schema for \(35 \times 35\) to \(17 \times 17\) reduction module. Different variants of this blocks (with various number of filters) are used in Figure 9, and 15 in each of the new Inception(-v4, -ResNet-v1, -ResNet-v2) variants presented in this paper. The \(k, l, m, n\) numbers represent filter bank sizes which can be looked up in Table 1.

Figure 8. The schema for \(17 \times 17\) to \(8 \times 8\) grid-reduction module. This is the reduction module used by the pure Inception-v4 network in Figure 9.
Inception stem

Historically, we have been relatively conservative about changing the architectural choices and restricted our experiments to varying isolated network components while keeping the rest of the network stable. Not simplifying earlier choices resulted in networks that looked more complicated than they needed to be. In our newer experiments, for Inception-v4 we decided to shed this unnecessary baggage and made uniform choices for the Inception blocks for each grid size. Please refer to Figure 9 for the large scale structure of the Inception-v4 network and Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of its components. All the convolutions not marked with “V” in the figures are same-padded meaning that their output grid matches the size of their input. Convolutions marked with “V” are valid padded, meaning that input patch of each unit is fully contained in the previous layer and the grid size of the output activation map is reduced accordingly.

3.2. Residual Inception Blocks

For the residual versions of the Inception networks, we use cheaper Inception blocks than the original Inception. Each Inception block is followed by filter-expansion layer (1×1 convolution without activation) which is used for scaling up the dimensionality of the filter bank before the addition to match the depth of the input. This is needed to compensate for the dimensionality reduction induced by the Inception block.

We tried several versions of the residual version of Inception. Only two of them are detailed here. The first one “Inception-ResNet-v1” roughly the computational cost of Inception-v3, while “Inception-ResNet-v2” matches the raw cost of the newly introduced Inception-v4 network. See Figure 15 for the large scale structure of both variants. (However, the step time of Inception-v4 proved to be significantly slower in practice, probably due to the larger number of layers.)

Another small technical difference between our residual and non-residual Inception variants is that in the case of Inception-ResNet, we used batch-normalization only on top of the traditional layers, but not on top of the summations. It is reasonable to expect that a thorough use of batch-normalization should be advantageous, but we wanted to keep each model replica trainable on a single GPU. It turned out that the memory footprint of layers with large activation size was consuming disproportionate amount of GPU-memory. By omitting the batch-normalization on top of those layers, we were able to increase the overall number of Inception blocks substantially. We hope that with better utilization of computing resources, making this trade-off will become unnecessary.
ResNet

![ResNet Diagram]

Figure 10. The schema for $35 \times 35$ grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.
**Depthwise separable convolution**

Main idea: factor NUM_FILTERS 3x3xNUM_CHANNELS convolutions into:
- NUM_CHANNELS 3x3x1 convolutions for each input channel
- And NUM_FILTERS 1x1xNUM_CHANNELS convolutions to combine the results

### Convolution Layer

- **Inputs**: NUM_CHANNELS inputs
- **Weights**: $K_w \times K_h \times \text{NUMCHANNELS}$ weights (for each filter)
- **Work per output pixel**: $K_w \times K_h \times \text{NUMCHANNELS}$ work per output pixel (per filter)

### Depthwise Separable Conv Layer

- **Inputs**: NUM_CHANNELS inputs
- **Weights**: $K_w \times K_h$ weights (for each channel)
- **Results of filtering each of NUM_CHANNELS independently**:
- **Weights**: NUM_CHANNELS weights (for each filter)
- **Work per output pixel**: NUM_CHANNELS work per output pixel (per filter)

**Image credit**: Eli Bendersky
How to improve system support for ML?

Hardware/software for...
faster inference?
faster training?

Compilers for fusing layers, performing code optimizations?

List of papers at MLSys 2020 Conference
But as a user wanting to create a model, where does most of my time really go?
ML model development is an iterative process

Example workflow:

1. **Define Task**
   - Task Spec
   - (Sec 6.1)

2. **Define Inputs**
   - (Sec 6.1)

3. **Data Selection**
   - Data
   - (Sec 5)

4. **Generate Supervision**
   - Training Points
   - (Sec 4)

5. **Train Model**
   - Training Labels
   - (Sec 6.1, 6.2)

6. **Validate Model**
   - Model Outputs
   - (Sec 5)

7. **Identify Important Data**

8. **Refine Task**

9. **Increase Supervision**

10. **Change Training Process**

   - New Supervision Sources

   - New Spec, Different Pre-Trained Inputs
     - Different Training Points, Rare Examples, New Failure Modes

   - New Architectures, Augmentations, Training Procedure

Stanford CS348K, Spring 2020
Example: how does TensorFlow help with data curation?

“We cannot stress strongly enough the importance of good training data for this segmentation task: choosing a wide enough variety of poses, discarding poor training images, cleaning up inaccurate [ground truth] polygon masks, etc. With each improvement we made over a 9-month period in our training data, we observed the quality of our defocused portraits to improve commensurately.”
Thought experiment: I ask you to train a car or person detector for a specific intersection.
A good system provides valuable services to the user. So in these papers, who is the “user” (what is their goal, what is their skillset?) and what are the painful, hard, or tedious things that the systems are designed to do for the user?
Let’s specifically contrast the abstractions of Ludwig with that of a lower-level ML system like TensorFlow. TensorFlow/MX.Net/PyTorch largely abstract ML model definition as a DAG of N-Tensor operations. How is Ludwig different?

Then let’s compare those abstractions to Overton.
Comparison to Google’s AutoML?