Contents

Meeting Notes

Contents

Meeting Notes **Fall 2016**

First Meet:

Table of Contents Local *Scribe: Brandon McKinzie*

- **Fuzzing**: generation of inputs to break a progeram. Interested in making unusable .
- Goal:
  - pick up where other ppl left off on security. Take care of fuzzing itself.
  - Take care of the instrumentation. Modeling the way the program works. Give inputs to receive stats on what function calls were made.
- Three sources of info:
  - instrumentation info, structure
  - inputs modeling
  - The way the inputs affect the program.
- Problem is input is fluid. Need to generalize inputs.
- Example: one program received 2 arrays of char. ret true if match in the two.
  - feed to program with linear regression.
- The network will be able to learn what inputs render the program unusable.
- Long run: create something that is generic, can learn from .
  - Start with singular programs. get binary inputs, have program.
  - with basic architecture. 4 convulations. few fully convulioted. Represent as bytes and feed to network. Have network infer what's best.
- identify problems we will have later
- benchmark
- Want to come up with list of features.
  - number of functions
  - function sequence
  - memory usage

- automatic generation: grow it in scala.
- Action Items:
  - Read papers.
  - Black hat talk.
  - TF with goal of 3 separate inputs concat to single input. Connect it to a network. No RNN, just feedforward. Transform malware to input as images. Create a few separate networks and connect them later.
  - DPPN genetic algorithm and gradient descent

Meeting Notes **Fall 2016**

New Project Discussion: November 29

Table of Contents Local *Scribe: Brandon McKinzie*

- Distinction between candidate and meta-features.
- classifier produces prediction of how likely it is to improve.
- Want to generalize project to growing architectures.
- metafeatures on architecture. Statistics on performance of architecture.
- challenges:
  - embedding: need to represent the graph. Look at given architecture, and see what modifications will be beneficial.
  - Evaluation. Training ANN takes long time. Early stopping criteria helps.
  - Depth. Find methods to go deeper faster. One approach: replication (of layers). Another approach: adding blocks rather than layers.
- Working on generating metafeatures (already have architectures).
- Want to work on long-term goals: generation of deeper networks. Operators to generate blocks.
- Begin by
  - Familiarize with deep networks.
  - Obtain git repo. To see how architecture is generated. Learn scala.
- On Tuesday I will learn specifics.

Meeting Notes **Fall 2016**

New Paper and Assignment: November 8

Table of Contents Local *Scribe: Brandon McKinzie*

- We define a preset number of epochs and run program. Try to improve stopping condition. This is easy in tf. Remove redundant epochs.
- Fix ambiguous ADD test by defining two FC layers connected aside as a single FC layer.

Brainstorming

Contents

Brainstorming **Fall 2016**

Adding the + Operator:

Table of Contents Local *Scribe: Brandon McKinzie*

- Aside from all the theory, need to implement a very basic ”addition” operator. ResNet paper indicates this as adding a connection from some input to a hidden layer (assuming hidden bc ReLu activation) around and to the output, adding the learned mapping $\mathcal{F}(\boldsymbol{x})$ to itself.
- **Q**: Seem to be confused as to how all these classes really work together. The code all makes sense on an individual file basis, but I'm having trouble figuring out, for example, where certain operations really take place.
- The following would be helpful: A paragraph summary of DNNGraph, DNNComponent, Pipeline, DNNGraphProposal, etc. I want to make sure I really have a conceptual grasp of what these are. Couldn't hurt to add these descriptions as comments at the top of the files.
- Self-Assignment: Answer the following questions:
  - **Q:** In what file(s) *exactly* would I need to add code for the “+” operator? **A: Basically eveything where concat. Start with DNNComponentType and move from there. Should touch more than 5 files or so. Just start coding. Goal is to run some rudimentary code generation with the add operation.**
  - **Q:** Where is any tensorflow knowledge required? I can see some aspects within the CodeGeneration process, but for the majority of class, mention of tensorflow is absent. **A: It is entirely within the code generators. For this, basically need just one line.**
  - **Q:** In more detail what really *is* my assignment? I understand on a (very) high level what it *represents*, but I can't find an obvious location in the codebase where one could define something like “connection that identity maps an input x and adds this to output, doing tensor operations as necessary.” **A: the assignment is define a type ”add” that can be used by DNNGraph as some other generic DNNComponent. It should behave just like concat/any others. You are most certainly overthinking it. Just try stuff out.**
  - **Q:** Concat seems analogous to a stacking operation (eg. np.stack). Is this true? I'm trying to make sure I understand the conceptual difference between concat and add. **A**: No. Concat should be something like the tf.concat function here. It does indeed seem similar to add.

  - **Q:** The low-level details of how the CodeGeneration process actually encounters TensorFlow commands still remains a mystery to me. The codebase is abstracted very well, but almost so well that it is difficult to find exactly where the real work takes place, which is presumably where I would need to be in order to program up addition operations, especially considering the suggestion on Slack that I should be going through Tensorflow-related material. **A: It is in the codegenerator classes.**
  - **Q**: It would be really helpful to be walked through the task of "implementing the concat operation" so that I could see an analogy of what I need to do by example of a different operation. Perhaps this would nullify all of my questions above in one sweep. **A: You've already seen it. Stop thinking too much.**

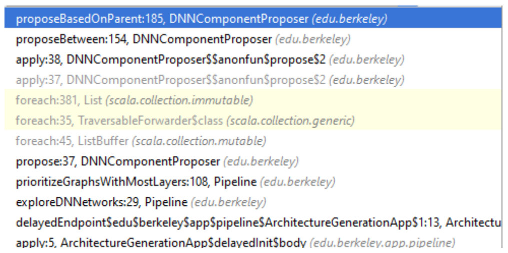
- It seems like I wouldn't have to add much more code if I implement add operation as a combo of insertbetween(..., add) and insertaside, where the insertion aside is either (1) identity(x) or (2) maybe just x ? Whichever is easier. I guess they are the same. Anyway.

Brainstorming **Fall 2016**

ADD Implementation Thoughts:

Table of Contents Local *Scribe: Brandon McKinzie*

- Debugging for proposer: In exploreDNNetworks, proposer.propose(...) is what triggers the entrance to the DNNComponentProposer class. The proposer object is a class attribute of Pipeline.
- Useful stack trace, shown in figure below, of how ArchitectureGenerationApp eventually reaches DNNComponentProposer::proposeBasedOnParent. This is useful to know because, if we keep ADD in DNNComponentType.computation(), then *ADD will get inserted into the proposed graphs.*



```
proposeBasedOnParent:185, DNNComponentProposer (edu.berkeley)
proposeBetween:154, DNNComponentProposer (edu.berkeley)
apply:38, DNNComponentProposer$$anonfun$propose$2 (edu.berkeley)
apply:37, DNNComponentProposer$$anonfun$propose$2 (edu.berkeley)
foreach:381, List (scala.collection.immutable)
foreach:35, TraversableForwarder$class (scala.collection.generic)
foreach:45, ListBuffer (scala.collection.mutable)
propose:37, DNNComponentProposer (edu.berkeley)
prioritizeGraphsWithMostLayers:108, Pipeline (edu.berkeley)
exploreDNNetworks:29, Pipeline (edu.berkeley)
delayedEndpoint$edu$berkeley$app$pipeline$ArchitectureGenerationApp$1:13, Architectu
apply:5, ArchitectureGenerationApp$delayedInit$body (edu.berkeley.app.pipeline)
```

- CodeGeneration crashes in DNNComponent.outTensorEstimate, due to entering 'case 4' as reshape expand with not match.
- Now I'm going to start looking into the methods filterCompatibleDimensions and isCompatibleWithParentRank, see section III.2 for details.

Published Papers

Contents

Published Papers **Fall 2016**

Microsoft: Residual Networks:

Table of Contents Local *Scribe: Brandon McKinzie*

- Denote the underlying mapping (hypothesis) as $\mathcal{H}(x)$. Should be able to learn the **residual function** $\mathcal{F}(x) := \mathcal{H}(x) - x$. Then we can find the desired $\mathcal{H}(x)$ by evaluating $\mathcal{F}(x) + x$.
- Motivation: degradation problem of deeper networks.
- Adopt residual learning to every few stacked layers, a.k.a. **building blocks**, defined as
$$y = \mathcal{F}(x, \{W_i\}) + x \tag{1}$$
which is performed by *a shortcut connection and element-wise addition.*
- If $\dim(x) \neq \dim(\mathcal{F})$, then can just do a linear projection W_s such that the dimensions of $W_s x$ do equal that of $\mathcal{F}$.
- Note: Their figure of the ResNet implementation appears to apply shortcut connections across every other conv block.
- Their model has few **filters** (convnet filters review below):
  - The parameters for a CONV layer are learnable filters, that will learn to activate when they see some type of visual feature (e.g. an edge of some orientation).
  - Each set of filters in each CONV layer will produce a separate 2D activation map.
  - From stanford: "*During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position.*[1]"

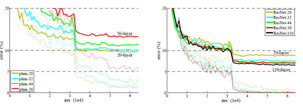


Figure 1: Training on **CIFAR-10**: Dashed lines denote training error, and bold lines denote testing error. **Left:** plain networks. **Right:** ResNets.

[1]A **stride** characterizes the step we slide the filter. The higher the stride, the smaller the output volumes will be spatially.

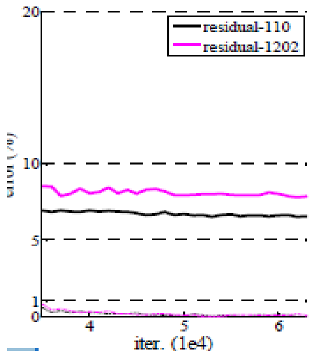


Figure 2: Comparison between ResNets with 110 and 1202 layers. 1202 layers appears to absolutely demolish the competition.

- The following paragraph has been downsized since authors later claim that *projection shortctus are not essential for addressing the degradation problem.* When dimensions change across shortcut connections: (1) if they increase, can just zero-pad and thus still identity-mapping of x; (2) if they decrease (projection) match dimensions, in practice done by 1x1 convolutions. For both (1) and (2), "when the the shortcuts go across feature maps of two sizes, they are performed with a stride of 2." However, they later note that projection shortcuts used for increasing dimensions actually do better than zero-padding, along with all other shortcuts being identity. (If *all* shortcuts are projections, this only marginally better.).
- So-called economical choice of **bottleneck architectures** appear to decrease training time.
- **Q**: I'd be interested in hearing more about their decision to note use dropout at certain times, yet they go on to say (pg 8) that they obtained best results with "strong regularization such as maxout or dropout." **A**: They do it simply so as to not overcomplicate their analysis. They do admit that such techniques may improve results.

Published Papers **Fall 2016**

Exploring Deep Learning for Supervised Learning:

Table of Contents Local *Scribe: Brandon McKinzie*

Introduction. Argue that DNNs aren't widespread due to (1) difficulty of designing architectures for new domains/non-experts; and (2) the large amount of computing power/time required. Here we explore applying effective architectures from one domain to others. Also compare wider networks vs. deeper networks.

> Our results show that while architectures do not perform well across multiple datasets, parallel architectures are surprisingly effective.

When looking for DNNs to a new domain, successively sample architectures, analyze performance, then continue sampling more from a subset of the best architectures. Can make this more efficient by first ranking architectures on their *expected* performance.

Related Work.

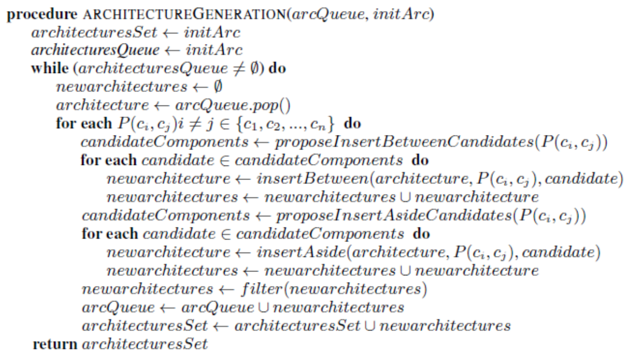
- First are studies that aim to derive more insights regarding inner workings of DNN. Notable results include a new version of RELU and a simplified version of the LSTM[2]
- Automatic DNN *Hyperparameter* Tuning. One interesting approach was having one LSTM optimize another.

Problem Definition. Current limitation of DNNs is needing a lot of testing/tuning. We explore the following:

- Architectures for multiple domains.
- Architectures that are effective in general for supervised learning. Other options than deep?
- DNN architectures compared with conventional machine learning classifiers in general.
- Identification of top-performing networks early on in training.

[2]This following site is also great at describing LSTMs: link.

Generating multiple DNN architectures. Generate a large number of both deep and wide architectures and train each on all datasets. Below is the corresponding algorithm.



```
procedure ARCHITECTUREGENERATION(arcQueue, initArc)
    architecturesSet ← initArc
    architecturesQueue ← initArc
    while (architecturesQueue ≠ ∅) do
        newarchitectures ← ∅
        architecture ← arcQueue.pop()
        for each $P(c_i, c_j) i \neq j \in \{c_1, c_2, ..., c_n\}$ do
            candidateComponents ← proposeInsertBetweenCandidates($P(c_i, c_j)$)
            for each candidate ∈ candidateComponents do
                newarchitecture ← insertBetween(architecture, $P(c_i, c_j)$, candidate)
                newarchitectures ← newarchitectures ∪ newarchitecture
            candidateComponents ← proposeInsertAsideCandidates($P(c_i, c_j)$)
            for each candidate ∈ candidateComponents do
                newarchitecture ← insertAside(architecture, $P(c_i, c_j)$, candidate)
                newarchitectures ← newarchitectures ∪ newarchitecture
            newarchitectures ← filter(newarchitectures)
            arcQueue ← arcQueue ∪ newarchitectures
            architecturesSet ← architecturesSet ∪ newarchitectures
    return architecturesSet
```

Define a **component** as a layer, normalization, or activation function. Specifically, this paper studied the following components: FC layers, softmax, batch normalization, dropout and the [ReLU, sigmoid, tanh] activation functions.

Beginning with only an input and output, iteratively do: foreach pair of components, indentify all components that could be inserted *between or aside*. For all such allowed components, generate a new copy of the architecture with the performed insertion. **Limitation**: hyper-parameters are fixed for each component.

Meta-learning for architecture ranking. Three types of meta-features generated:

1. **Dataset-based**. Includes *general information* (number samples, classes, features, correlations), *entropy-based measures* (information gain IG of features), and *feature diversity* (chi-squared and paired-t tests for similarities).

2. **Topology-based**. Includes *composition* (num/types of layers, their functions, etc) and *connectivity-based measures* (graph-analysis on e.g. ratio of incoming/outgoing edges).

3. **Training based**. Includes *static evaluation* (values at fixed times) and *time series-based evaluation* (ratios and value distributions over time).

Experiments and Analysis.

- **Setup**. For each dataset, train same set of 11,170 architectures. Data split: 80% training, 10% validation, 10% test. Train until convergence, as determined by performance on validation set. To train the **ranking classifier**, use LOO cross-validation where each "sample point" is now one of the *datasets* (and one is left out). Use 70% of the architectures to train the ranking classifier, and 30% for evaluation.

- **Analysis**.

  → Accuracy distribution of architectures varies a lot for each dataset.
  → In most cases, DNN architectures do not perform well across multiple datasets. ☹
  → Neither Random Forests nor the DNN architectures consistently outperform the other.
  → *Architectures with parallel layers significantly outperform all of their serial counterparts.* Did not find evidence that batch normalization[3] improves performance of such parallel architectures.

- **Meta-learning evaluation**. Settings: Define top 10% architectures of each dataset as "good." Calculate actual percentage of good architectures output by the meta-classifier as its X top ranked. Use the Random Forest algorithm for the training of the meta-model.

  > *We show that we are able to identify multiple architectures in the top-ranking spots in a much higher ratio than their share of the population. It is also clear that the joint set of all meta-features outperforms both of the examined subsets.*

[3]Batch normalization review: BN is a technique to provide any layer in a Neural Network with inputs that are zero mean/unit variance. See this link for more.

Class Summaries

Contents

Class Summaries **Fall 2016**

DNN:

Table of Contents Local *Scribe: Brandon McKinzie*

DNNGraph. The highest-level representation of a deep network. One can traverse the graph by moving up/down through children/parents of the input/outputs. Supports insertion operation of DNNComponentProposals in relation with DNNComponents already within the graph. Aside from insertion operations (which appear to comprise most of the operations), one can also perform various checks such as whether or not DNNComponent dimensions match such that they could be concatenated.

DNNComponent. Can be any of the following *DNNComponentTypes:*

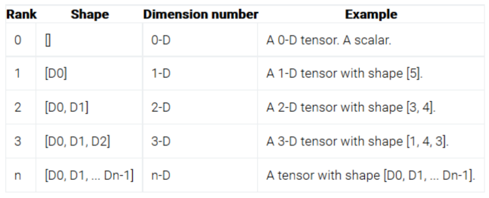
- readin
- fully connected
- dropout
- convolution 2d
- maxpool 2d
- reshape expand or reshape shrink
- concat
- batchnorm
- lrn
- relu
- sigmoid
- tanh
- softmax

DNNComponentProposer. Finally, this appears to be the first class that performs some of the lower-level heavy lifting. For example, when performing the ”propose” operation, it will try things like attaching activation layers to DNNComponents and inspecting the resulting architectures. Here we truly have the first instance of tinkering with new types of graphs.

- **filterCompatibleDimensions:** First, I need to figure out exactly what **rank** refers to here. **A: tensor rank**, also called tensor order, is the number of indices required to write a tensor. Therefore, all matrices have a tensor rank of 2.
- For an overview of Tensor Ranks, Shapes, and Types, see this tensorflow tutorial. I will summarize its contents here below. The following tensor has a rank of 2:

  ```
  t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
  ```

  Make sure you understand the distinction between rank, shape, and dimension number in tensorflow, as indicated in table below.

Rank	Shape	Dimension number	Example
0	[]	0-D	A 0-D tensor. A scalar.
1	[D0]	1-D	A 1-D tensor with shape [5].
2	[D0, D1]	2-D	A 2-D tensor with shape [3, 4].
3	[D0, D1, D2]	3-D	A 3-D tensor with shape [1, 4, 3].
n	[D0, D1, ... Dn-1]	n-D	A tensor with shape [D0, D1, ... Dn-1].

- Now, with all that said, **Q**: Why are the values assigned to the various component types (e.g. rank(softmax) = 2) the values that they are?
- **Traversing the class**. Pipeline will call proposer.propose (on various DNNGraph objects) when finding the best architectures.

Pipeline. The “brains” of the architecture generation process. It employs the following class attributes to intelligently find new architectures:

- proposer: DNNComponentProposer
- graphDAO: GraphDAO
- evaluator: Evaluator
- statsCollector: StatsCollector

It uses these objects in its primary method “ExploreDNNetworks,” wherein the proposer and evaluator work in conjunction, through a seemingly trial-and-error process, to find networks that are deemed ’best’ by a predefined criterion (e.g. prioritizeGraphsWithMostLayers).

CodeGenerationApp and TFLearnCodeGenerator. This is probably the most mysterious part for me. Below, I log my observations as I step through the program.

- Break on

  ```
      val codegenerator = new TFLearnCodeGenerator
  ```

  which is syntactically confusing because no parens, but I can look into that later (**TODO**).

- Taken through TFLearnCodeGenerator → (abstract) CodeGenerator → **GraphDAO**[4] → (abstract) AbstractDAO → **GraphDAO::retrieveALL()**[5]. The application then runs with what appears to be "all maps from DB (presumably database).

- The first u we encounter if we break on persist.appy("mnist"...) is a simple input-output DNNGraph with some reshape expand/shrink components in between.

- Stepping into persist.apply(): Just calls a stringbuilder in a while loop (See DNNGraph.hash()). Ah, but this stringbuilder is most likely what is actually coding the Tensorflow in python. Let's investigate. It's actually pretty difficult to tell. Might just be some generic Scala object.

- [SOLVED]. Ah, the tensorflow code is (surprise surprise) inside the methods of TFLearnCodeGenerator.scala. For example, we have the method[6]

  ```
    override protected def visit(idmapping: Map[String, Int],
        idToNameMapping: Map[String, String], c: DNNComponent, experiment:
        Experiment): String =
  ```

  which, among other things, will match the componentType and then return a **line** (most likely just a string) of the form, e.g. for **CONCAT**:

  ```
  case CONCAT => line(s"${outname(c)} =

  tflearn.layers.merge_ops.merge(
  [${c.parents.map(p => outname(p)) mkString ","}],
  'concat', axis=1)")
  ```

- Note that HIGHWAY_2D and RESIDUAL_BLOCK cases are already implemented.

[4]Start of mystery. Why does it go here? Answer: Perhaps due to Scala not requiring thing like parens(?) we can pass the new GraphDAO to TFLearnCodeGenerator, which is why we enter the class GraphDAO.

[5]Mystery 2: Why does it also go here? Seems to be jumping to methods that I can't find where they were called. **Answer:** Immediately after new GraphDAO() is called, it (in the spirit of functional programming) calls retrievAll which is subsequently sent into a foreach loop on the returned collection from retrieveAll().

[6]Not really sure why it is called "visit" based on the code inside the method, but we'll come back to this.

- differentiate between insert aside with concat and insert aside with sum.

V. Assignments Log

Each bullet-point that follows is, in chronological order, the communication I've received as it relates to research assignments.

- **October 6th meeting**. Read the ResNet paper and Wide networks paper. Add the "+" operator of Resnet. Expand/shrink operator for convolutions & fully-connected layers.
- **October 10th Slack response (jmrozanec)**. Instructed to focus on add operation and to go over tensorflow tutorials (had already completed).
- **October 13th meeting**. Still need to implement add. Add tests and try to generate different graphs locally.