Emel: Deep Learning in One Line using Type Inference, Architecture Selection, and Hyperparameter Tuning

BARAK OSHRI and NISHITH KHANDWALA

We present *Emel*, a new framework for training baseline supervised deep learning models. *Emel* is primarily a user interface for training excellent deep learning models in one line of code. Our framework presents three important contributions: 1. Type Inference, 2. Architecture Selection, 3. Intelligent Training. The *type inference* module infers the type of user data. The *model selection* step infers the task and architecture to solve it. The *heuristic-guided training* step incorporates expert insight in training the model. *Emel* demonstrates that deep learning languages can have successful type inferencing, and that for supervised learning applications, *program inference* can be achieved by inferring the model and automatically deducing the optimizations. We have implemented *Emel* for the three input types images, sequences, and features. We hope *Emel* can point towards a paradigm where data preprocessing, architecture choice, and model training is compiled into a single line in the programming pipeline.

Additional Key Words and Phrases: Deep Learning, Programming Abstractions, Type Inference

ACM Reference Format:
Barak Oshri and Nishith Khandwala. 2017. *Emel*: Deep Learning in One Line using Type Inference, Architecture Selection, and Hyperparameter Tuning. 1, 1 (December 2017), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Emel allows us to do machine learning in one line:

```
trained_model = Emel(X, Y)
```

Where X is the input data and Y is the label data, both in Numpy format.

Though the number of available deep learning frameworks for writing and training models continues to increase, their ease of use has not yet peaked. The choices for programming on these exist on a spectrum from writing *primitive* layers of the model architecture to loading *sophisticated* pretrained architectures.

Despite incredible advances on this spectrum, training machine learning models for simple, baseline usage remains out of reach even on the end of this spectrum that reduces machine learning models to high-level interfaces. The Keras framework, which is arguably the most versatile yet high-level deep learning framework, still includes many lines of redundant code across applications that makes the barrier-to-usage for machine learning applications high.

These redundancies include but are not limited to the following:

(1) Formatting: reshaping data to standard input sizes
(2) Preprocessing: running standard preprocessing functions on known input types
(3) Architecture Selection: loading an architecture from a small and finite set of provenly successful deep learning models

Authors' address: Barak Oshri, boshri@stanford.edu; Nishith Khandwala, nishith@stanford.edu.

Manuscript submitted to ACM

(4) Evaluation: setting up standard evaluation metrics
(5) Hyperparameter Selection: iterating through a set of standard hyperparameters

We argue that each of these steps is programmable when the data type is known. For example, when the input type is an *image*, we can say that

(1) Formatting: ensure that the number of channels appears as the last dimension of the image
(2) Preprocessing: normalize the input to zero mean and unit standard deviation
(3) Architecture Selection: load a ResNet model
(4) Evaluation: use accuracy if the dataset is imbalanced, F1 and ROC if the dataset is unbalanced
(5) Hyperparameter Selection: choose from a standard set of batch sizes and learning rates

The reason the input type alone dictates each of the choices in the model building is that there are a small number of provenly successful deep learning architectures that can perform well on most daily tasks. Since these architectures have standard input types, output types, hyperparameter choices, and loss functions, then the input type alone allows us to infer each of the parts.

We argue that the *Emel* framework makes an important statement on the potential future of programming language paradigms for machine learning usage. The current paradigm in these frameworks is that doing machine learning involves manually interacting with an *architecture specification* step and then a *training* step. For example, in Tensorflow, the first step of a program is to specify the model graph, and the second step is to build a session that executes the graph. In PyTorch, the two layers are present but mixed together in an interactive session where the models are defined and called at any point.

We wish to relinquish any programmatic control over these two steps to create a single *architecture compiler* that does machine learning behind-the-scene.

2 TYPE INFERENCE

The linchpin to the *Emel* framework is type inference of arbitrary data input, whose raw data is inputted in numpy format. Our framework assumes a set of finite types: *image*, *sequence*, and *features*. Though this set is limited, it is chosen to have one-to-one correspondence with principal machine learning architectures. We anticipate that as the scope and diversity of machine learning architectures increases in the long run that each architecture will become an enabler for a new type, thereby increasing the semantic capabilities of our Emel framework.

2.1 Inference Assumptions

Emel has to balance a fundamental tension between the amount it infers and the amount it assumes. Inference is not useful if it assumes that the input data comes in a specific format. Additionally, there are some attributes that are impossible to infer from the data itself, such as whether an image is in RGB or BGR format.

The only controlling assumption we make is that the first dimension of any numpy input is the number of examples in the data.

The approach we take to specifying what our input types are is with a list of rules. For *Image* types, we have a set of necessary conditions that determine its type. If an input satisfies all of these conditions, it is likely an image.

For sequences, we have a set of sufficient conditions, any of which determine that the input is a *sequence* type.

These rules inform the complete set of conditions we use to specify the inference of a particular type. Whenever the data doesn't satisfy the necessary conditions in the *Image* or *Sequence* types, we automatically make it a *Feature* type.

Any data can be trained in the *Feature* type. This allows us to circumvent the problem of not being able to write down sufficient conditions for meeting these types.

We therefore program the type inference using the following backbone code:

```
if all([rule(X) for rule in image_rules]):
    X.type = "Image"
else if any([rule(X) for rule in sequence_rules]):
    X.type = "Sequence"
else:
    X.type = "Features"
```

2.2 Image

We infer an input to be an *image* type if it satisfies all the following rules:

(1) The number of channels is a number that is smaller than the width or height.
(2) The number of channels appears as the last or the first feature dimension
(3) An image can be greyscale with no dimension on the number of channels
(4) The length and width channels have the same size
(5) The length and width channels are larger than 47 pixels (otherwise the input is too small to be an image)

If an input has the *image* type, we transpose the channel layers to make it the last dimension of the array. We normalize our image to zero mean and unit standard deviation.

2.3 Sequence

We infer any input to be a *sequence* if it satisfies any of the following rules:

(1) The input is ASCII characters
(2) The input has variable size length

The main challenge with sequence inputs is to correctly tokenize the inputs. For example, the following are three inputs that are tokenized differently:

(1) "This is a sentence" - String of words
(2) "Paris,London,Vienna" - Comma separated string of words
(3) "ACTAGTGC" - String of DNA characters
(4) 1, 5, 3, 9 - List of arbitrary classes

If the input is a list of numbers, then we treat those as the class numbers in an arbitrary class domain.

We then have a set of delimiters that we test for tokenization, including *spacebar*, *comma*, *comma/spacebar*, etc. The delimiter that leads to the most number of split items is that one that is used. If none of these delimiters split the items well, then we separate by character.

The items are then represented as indicies into the dictionary of items in the domain space. That is, we make a dictionary mapping of all tokens seen in the dataset and map those tokens to integers starting from 0. We will use these as indices to one-hot vectors to train the architecture for *sequence* types.

2.4 Features

Any input that doesn't fall under the above types is automatically mapped to a *feature* type. For this type, we reshape any input to be single dimensional and then we normalize each column to zero mean and unit standard deviation.

2.5 Labels

We also infer the training label types. We limit our current *Emel* framework to train to a single scalar *categorical* or *continuous* label types. In theory, we could train to a vector of categorical or continuous values.

3 ARCHITECTURE SELECTION

The architecture selection stage of the pipeline deterministically chooses the architecture based on the input and label types. The input type determines which *prediction* architecture we use, and the label type determines which *loss* function we use. We choose common deep learning architectures that are known to do well on these input types:

(1) Images -> ResNet
(2) Sequences -> LSTM
(3) Features -> Fully Connected Neural Network

The loss function choices based on the label types are:

(1) Continuous -> L2 loss
(2) Categorical -> Cross-entropy loss

Our architecture selection module combines the prediction architecture with the loss function layer to produce an end-to-end trainable function. We also infer in here what size of these architectures to use. For example, with ResNet, we have 18 layer, 35 layer, 50 layer, and 101 layer resnets available. Similar options are available for choosing larger and smaller versions of the LSTM and fully connections networks. We choose this based on the size of the dataset and the level of overfitting of the data. We start with smaller architectures and if the smaller architecture doesn't overfit the data, we use larger architectures.

4 HEURISTIC-GUIDED TRAINING

Training deep learning models is widely considered to be a craft - many key decisions for the model architecture and optimization are backed by intuition and experience, and not by rigorous theorems. In *Emel*, we try to embed much of this intuition into the training procedure. More concretely, machine learning practitioners usually train several models, say n - each of which takes t hours to train - as a part of hyperparameter tuning. The model developers might spend an additional x hours debugging their code to avoid unsuccessful training runs, overfitting etc. We hope to reduce this $nt + x$ hours of development time to $t + \epsilon$ hours (where $\epsilon < t$) by using the heuristic-guided training module.

First, *Emel* eliminates a large numbers of hyperparameters (model variables that can be tweaked for better performance) by fixing model architectures to that of the current state-of-the-art models. Some examples of hyperparameters we do not need to consider include dropout, batch normalization parameters and number of layers. The state-of-the-art models we borrowed the architectures from are the result of months of tuning already and can, therefore, be assumed to close to optimal.

The hyperparameters yet to be selected include batch size, number of training epochs, learning rate, learning rate decay (if applicable) and the optimization algorithm. In the following paragraph, we describe the heuristics we coded to

find a well-performing set of hyperparameters. Note that these heuristics represent the intuition and experience of the authors and cannot be defended by anything but empirical evidence.

For all hyperparameters, we quit training early if the training and validation error do not decrease or if the training loss has plateaued.

(1) Learning rate: For image models, we start off with a low learning rate of $1e-4$ and a decay rate of $5e-3$. The image models are initialized with weights trained on the ImageNet dataset and thus can be simply fine-tuned on the new dataset. For text models, we begin with a learning rate of $1e-3$ and a decay rate of 0.0.
(2) Batch size: We pick a batch size depending on the size of the dataset. If the dataset has more than 1,000 examples, we try a batch size of 64. If there are more than 10,000 data points, we try 128 and if more than 100,000, we try 256. In case we are working with a rather small dataset of less than 1,000 examples, we pick a batch size of 32. Currently we assume that the machines we are using can accommodate these batch sizes. In future work, we will make provisions for GPU and memory specs.
(3) Number of epochs: The maximum number of epochs is 200. We quit training early and save the model if the training loss has plateaued, the training accuracy has reached 100%.

We also experiment with several training tricks such as data augmentation, preprocessing steps (normalizing, whitening and unit variance and mean) and dataset rebalancing.

Finally, we return the model with the best performing score on the validation dataset.

5 RESULTS

Emel is able to infer and train on multiple types of data, from images, to text, to arbitrary features.

The test cases we use for each type are:

(1) Image: We get to 91% accuracy on CIFAR-10.
(2) Sequence: We get to 81% accuracy on Twitter sentiment analysis
(3) Features: We get to 98% accuracy on MNIST.

Additionally, *Emel* achieves these accuracies in less than ten cycles of its hyperparameter updates, which is a significant speedup from writing models to train on these datasets from scratche.

Emel runs into a few difficulties in training. Most of the updates to its hyperparameters are incremental from an initial set of hyperparameters given to it in its first run. For some training runs, the trajectory from that initial set of hyperparameters is not successful. We tell *Emel* to stop training if there is no improvement over many runs, but we are lenient rather than strict in its stopping conditions. When the initial set of hyperaparameters is not good for the task, *Emel* takes a long time to stop.

6 CONCLUSION AND FUTURE WORK

Emel is a case study in demonstrating that compilers can be written for compiling machine learning architectures and training models automatically on any input data in untyped languages. In typed languages, the advantage is even greater, as we can enforce in the programming language compiletime the data requirements for runing our architecture selection and training modules. However, we show as our main contribution that in the data programming paradigm we can implement type inferences to shift the focus of machine learning programming languages from model creation to application usage.

Our framework doesn't come with a few key shortcomings that would need to be addressed in further work on it:

(1) *Emel* expects the entire dataset to be loaded as a single numpy array, assuming infinite memory to do so. Most machines can't load entire large datasets into a single array. We could solve this by loading the numpy arrays as h5py files, or by allowing users to pass filename paths. However, working with filename paths essentially bypasses the type inference stage, since from path extensions we can easily infer the data type.
(2) The tradeoff for extreme simplicity in *Emel* is a loss of some configurations that come with deep learning frameworks, such as specifying when a user would like to train on multiple GPUs or otherwise. We can ask *Emel* to infer what resources are available from the central worker and to parallelize if necessary across them to maximize utility, but these are inferences that might go against the wishes of the user who doesn't want to utilize all the resources. Like many other programming languages, *Emel* eschews control for user-friendliness.

Nishith and Barak participated in equal share of the project.

7 REFERENCES

Keras Framework - https://keras.io/