Deep Learning in Insurance Risk Quantification: A Comparative Study

Eugene Kwak

September 8, 2018

Contents

I. Definition

Project Overview

Balancing losses and premiums is the crux of managing a successful insurance business. This balancing act consists of being able to quantify and understand these financial losses (risk) that will be incurred in order to set premiums at rates that are both profitable and attractive to consumers. Here I focus on methods for quantifying risk. Insurers invest heavily on teams of analysts, domain experts and actuaries to perform this task. Countless studies are also regularly published exploring machine learning approaches to solve this problem and evaluate commercial solutions (see comparative analysis of risk models in healthcare by the Society of Actuaries from 2016)[1]. In short, risk prediction is and always has been one of the biggest challenges in insurance.

With the advent of machine learning and big data in commercial insurance, companies are eagerly seeking new ways to apply novel techniques to improve their business, but are often challenged with the complexities that come with supporting these technologies. Traditional methods used in practice include actuarial sciences and mostly linear models. More robust techniques like ensembling are recently being utilized. However, the industry is still behind in terms of state of the art methods like deep learning and the technologies to support them.

The goal of this project is to [1] determine the feasibility of deep learning in this task and [2] to do a proof of concept to make a case for insurers to invest in deep learning technologies. Allstate ran a Kaggle competition in 2016 and provided an anonymized dataset well-suited for this task[2].

Problem Statement

Can deep neural networks provide a viable solution for risk modeling in insurance claims?

Predictive models for assessing future risk are commercially available across nearly all insurance domains from automobile to healthcare. For example, Equifax offers its Insight Score[3] product while companies like Verscend sells its DxCG[4] product for healthcare insurance. In general, these solutions do not utilize deep learning for predicting future risk. The upfront investments needed simply do not justify the marginal increase in performance deep learning often provides. Additionally, many domains have a strong preference for model interpretability, which is often why linear models are so well received.

I attempt to make a case for deep learning through this proof of concept. Three models were trained and evaluated.

1. Benchmark Model: A benchmark ensemble model was trained as a proxy for a competitive market solution that a deep learning model needed to beat.
2. Deep Learning Model (CPU): A deep neural network was trained to compare against the benchmark model. This model was also used to benchmark the cost-benefits of technologies optimal for deep learning (namely GPU's).
3. Deep Learning Model (GPU): The same deep neural network architecture was trained on a GPU cluster for comparison against the CPU trained model.

A successful deep learning model had to [1] beat the benchmark model in validation or cross-validation performance, [2] be comparable to the solutions provided via Kaggle submissions for Allstate's competition, and [3] be more computationally efficient on a GPU cluster. I prototyped a functional machine learning platform that can integrate with an organization's database technology stack to further simulate a business setting.

Inputs

The inputs to solve the problem were data provided by Allstate via Kaggle for the Claims Severity competition. Data included a `train.csv` file and a `test.csv` file without labels. The data are discussed in more detail within the *Analysis* section.

Learning to be Done

The learning consisted of the three models outlined above. For each model, the training data were split into training and validation partitions. Models were validated on the validation partition during training. The provided `test.csv` file served as the hold-out test set and used to score the models through the Kaggle submission API.

Target Outputs

The models predict the label provided by Allstate called `loss`. Each observation in the data is a unique insurance claim. The target variable is simply the cost of a claim. Basically, the goal is to predict the cost of a future claim based on all relevant features of the claim. Features are anonymized, but claim features typically include demographics like age/gender, date, and information encoding the details surrounding the event for which the claim is being made.

Metrics

The problem statement is broken into three comparisons as outlined above. Each comparison will be based on a different set of metrics specific to the goals at hand.

Deep Learning vs Benchmark

One of the most common metrics to compare solutions and models is the humble coefficient of determination (R^2). If a suitable benchmark cannot be beaten, then there is no point in continuing with deep learning. The R^2 was computed on validation partitions.

	R-Squared		MAE		95th Percentile of Error	
	1,000	10,000	1,000	10,000	1,000	10,000
Diagnosis-Only Models						
ACG System	12.1%	16.0%	9.4%	2.9%	22.5%	7.2%
CDPS	7.8%	9.6%	9.2%	3.1%	22.5%	7.3%
DxCG	14.5%	18.8%	9.2%	2.9%	21.9%	7.2%
Impact Pro	13.1%	18.1%	9.2%	2.9%	22.2%	6.9%
MARA	15.2%	19.8%	9.2%	2.9%	21.7%	7.1%
Truven	16.6%	20.7%	9.1%	2.8%	21.3%	7.3%
Wakely	13.8%	17.4%	9.2%	2.9%	21.8%	7.2%
Pharmacy-Only Models						
ACG System	10.6%	14.0%	9.4%	3.0%	22.2%	7.1%
DxCG	13.8%	14.4%	9.2%	3.0%	22.0%	7.2%
Impact Pro	13.8%	13.8%	9.2%	3.0%	21.9%	7.3%
MARA	14.0%	15.3%	9.2%	2.9%	22.1%	7.1%
MedicaidRx	8.8%	8.4%	9.5%	3.1%	22.6%	7.3%
Wakely	10.3%	9.7%	9.4%	3.0%	22.9%	7.5%
Diagnosis-and-Pharmacy Models						
ACG System	13.9%	18.5%	9.4%	2.9%	22.3%	7.1%
CDPS+MRX	9.7%	10.7%	9.5%	3.1%	22.6%	7.2%
CRG	11.8%	12.1%	9.3%	3.0%	22.2%	7.4%
Impact Pro	15.8%	20.6%	9.1%	2.9%	22.0%	6.8%
MARA	18.5%	21.6%	8.9%	2.8%	21.7%	7.0%
Wakely	16.0%	18.5%	9.1%	2.9%	21.4%	7.1%
Prior Cost Models						
ACG System	14.5%	20.0%	9.2%	2.9%	21.9%	6.9%
DxCG	22.1%	24.3%	8.8%	2.8%	23.0%	7.1%
MARA	22.4%	24.9%	8.7%	2.8%	21.9%	6.8%
SCIO	14.3%	15.8%	9.2%	2.9%	22.1%	7.2%

Figure 1: 2016 Risk model comparisons by Society of Actuaries

Figure 1 shows a great example of how different models are compared taken from a Society of Actuaries report[1]. I compared a deep learning model to the ensemble benchmark using the r2_score implementation from scikit-learn.

Eq 1. R^2

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{samples}-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{samples}-1} (y_i - \bar{y}_i)^2}$$

Deep Learning vs Kaggle

If a simple deep learning model beats the ensemble, it will be compared to other solutions provided through the 2016 Kaggle competition. The competition uses mean absolute error (MAE) on the provided test data to rate submissions. Again, I used scikit-learn's implementation during model training on validation partitions. The final MAE was computed via Kaggle submission and is a measure of how viable deep learning is as a risk assessment tool. I tried to beat the median submission MAE of 1,125, but fell short during the time I had for this project. I plan to continue modifying my network to beat this score.

Eq 2. Mean Absolute Error

$$MAE(y, \hat{y}) = \frac{\sum_{i=0}^{n_{samples}-1} |y_i - \hat{y}_i|}{n_{samples}}$$

Return on Investment

Lastly, I gathered metrics for computation performance, financial costs and opportunity costs to make a case for the return on investment on a full deep learning solution (including the technology stack). More specifically:

- Cost per R^2 gain
- Training and scoring times
- Storage and computational overhead

Justification

I am using R^2 to evaluate my models as that is a common metric used in insurance to compare models. It is easily understood by even those without deep statistical knowledge as the value always ranges between 0 and 1. The idea here is to simulate an actual business scenario and report metrics meaningful to those in executive leadership. MAE is the metric that Allstate judges submissions on in the Kaggle competition. In order to see how my model compares to other solutions directly, MAE is required. Lastly, the supplemental metrics such as costs and run times are primarily to understand the investments needed for a deep learning solution.

II. Analysis

Data Exploration and Visualizations

Data Description

The data was provided in a fully anonymized state from Allstate via Kaggle. A training and test dataset were given.

Table 1: Counts from data provided by Allstate

partition	rows	columns	count_rows_na	count_rows_dupe_id
train	188318	133	0	0
test	125546	131	0	0

The training data had about 188k rows. There are 130 continuous and categorical features and 1 label "loss". I created an additional column called "log_loss" which is just the natural log of the label column provided. Each row represents a single claim. The data were scrubbed and cleaned by Allstate so minimal data processing was needed for this project. There were no duplications or nulls. This is actually how a machine learning application may work in practice. The application will only see data that is in some canonical format provided by the organization's data engineering processes.

There were 14 continuous features prefixed with "cont", 116 categorical features prefixed with "cat", and a claim identifier.

Target Variable

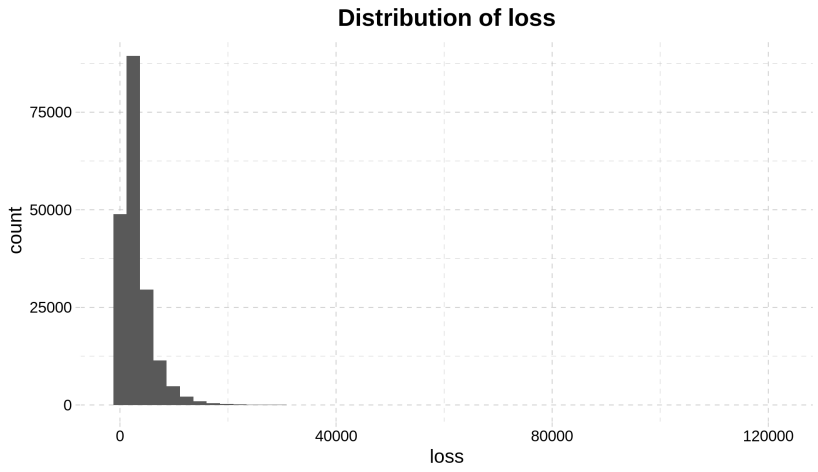

Figure 2: Histogram of the loss variable

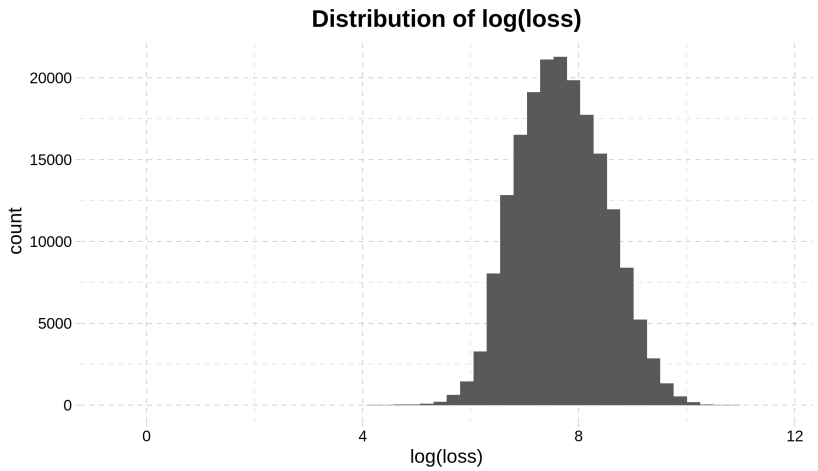

Figure 3: Histogram of the loss variable log transformed

Figures 2 and 3 show the distribution of the provided target variable before and after log transformation to improve the distribution for machine learning.

Continuous Features

The 14 continuous variables provided appear to have been normalized to a mean of 0.5 and standard deviation of 0.2 by Allstate.

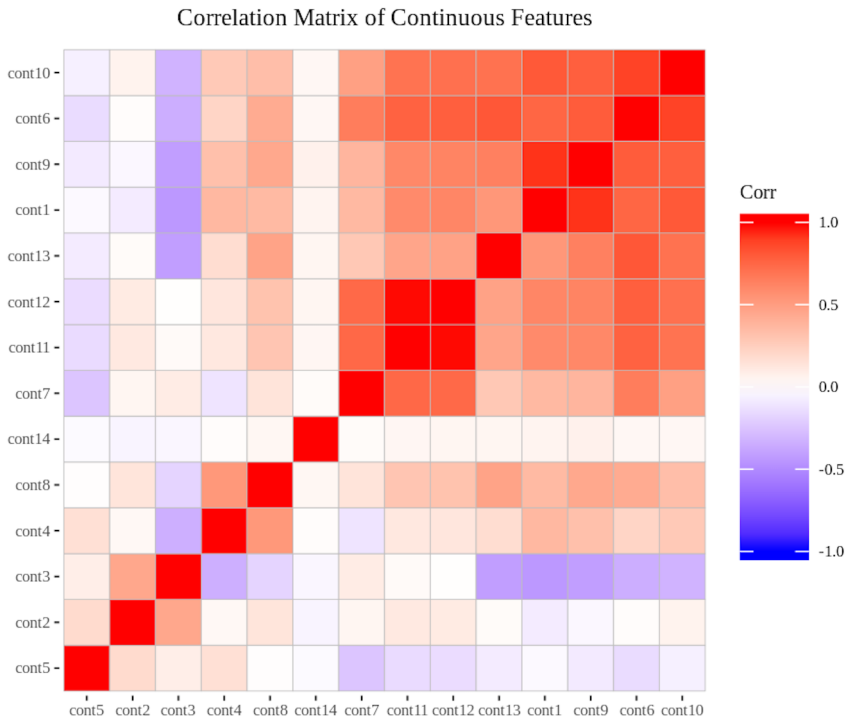

Figure 4: Correlation heatmap of the continuous features

Several features are highly correlated with each other such as cont6 and cont10. Those at the bottom left of the correlation heatmap in Figure 4 have much less collinearity and ended up being good predictors.

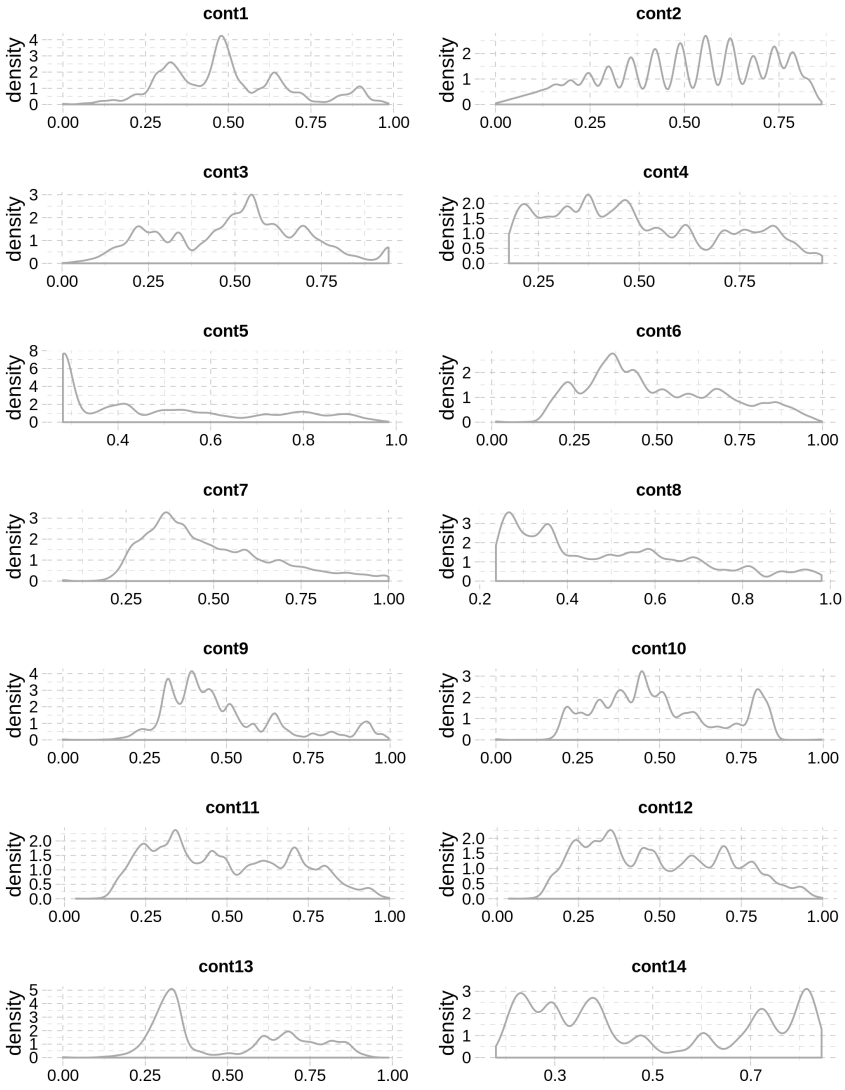

Figure 5: Density plots of the continuous features

The density plots show that not all continuous variables appear to be truly continuous. In particular cont2, looks like it could have been a discretized numeric variable given its density patterns.

Categorical Features

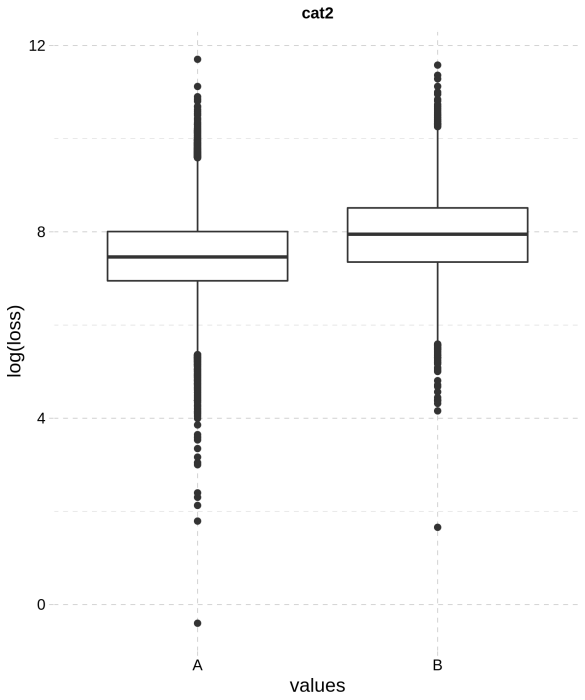

Figure 6: Box plot of cat2

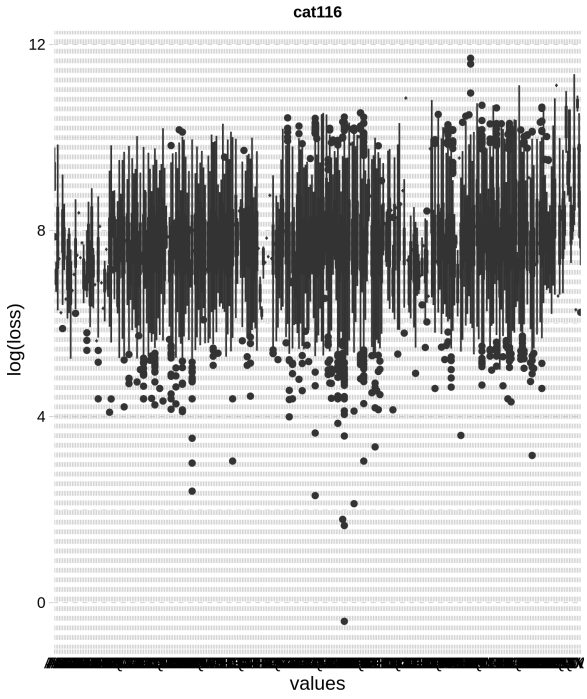

Figure 7: Box plot of cat116

Figure 6 shows an example of how most categorical variables were distributed. They had a cardinality of two with some outliers. There were 17 categorical variables with a cardinality greater than ten with cat116 (Figure 7) having the highest cardinality of 326.

Note: I did not apply any logic for dealing with high cardinality as my goal was not winning Kaggle, but rather to prove a case for deep learning. Also, this kept things simple enough for this project to remain tractable given the limited time provided.

Algorithms and Techniques

Feature Selection

The only pre-processing step on the feature space was to one-hot encode all of the categorical variables. This gave me 1,153 features in total. A decision tree regressor was fit to this processed dataset and feature importances extracted. Over many iterations, I found that a threshold of $1.05 \times mean(allfeatureimportances)$ gave me reasonably performant models. In other words, I selected variables with a feature importance score greater than this threshold value, netting me 152 input features.

Benchmark Ensemble Model

I tried multiple regressor classes available in scikit-learn's library and settled on simpler algorithms for the three sub-models that I ensembled. Two were simple linear models and one was a decision tree. I started with more complex algorithms, but found them to be too slow to iterate on and decided to use very simple models that are easy to tune and scale well.

Below are descriptions of the individual sub-models used based on the official scikit-learn documentation.

1. **Decision tree** - The scikit-learn implementation of the DecisionTreeRegressor class is a non-parametric model that learns decision rules based on the input space. It starts by finding the most relevant feature based on entropy reduction as the root node of the tree, creating two splits. At the next level, it finds the next most important features at each of the splits from the parent root, continuing along until all features have been accounted for. Some advantages of decision trees include easy interpretability and implementation, ability to handle multiple data types, and they are computationally efficient. However, they are prone to overfitting, especially if they are allowed to grow freely. Scikit-learn provides several important hyperparameters to control the growth of the tree and prevent overfitting. The hyperparameters I focused on are:
   - `max_depth` - How many levels a tree is allowed to grow. A larger value allows a tighter fit, but is more prone to be overfitting.
   - `min_samples_split` - Minimum number of observations needed to make a split at a node. Controls for overfitting.
   - `min_samples_leaf` - Similar to `min_samples_split` but limits the number of observations needed at a leaf node.
   - `max_features` - Maximum number of features to consider when splitting at a node.
2. **Ordinary least squares (OLS)** - The scikit-learn implementation of the LinearRegression class does not offer hyperparameters for controlling the model. OLS looks at the spatial relationships within the feature space to find a linear equation that fits most closely to the observations provided. It assumes independence among the features. It is a parametric model that finds a vector of weights (coefficients) that minimizes the residual sum of squares between the predictions and the actual target variable.
3. **Stochastic Gradient Descent** - The scikit-learn implementation of the SGDRegressor class is an extremely flexible and efficient linear estimator that uses the stochastic gradient descent

algorithm to minimize a loss function. To minimize loss with gradient descent, the gradient is computed at each observation and weights updated. There are a number of hyperparameters that control the fit of the model and those that define the type of model. The hyperparamters I used were:

- `loss` - SGDRegressor allows the user to select from a list of loss functions to define the type of linear model. I used the default loss of `squared_loss`, which is OLS.
- `penalty` - Regularization methods available are `l1`, `l2`, and `elasticnet`. I found `l2` was well suited for this project as it is fast.
- `alpha` - Learning rate parameter and controls the regularization. I used the default of 0.0001.
- `max_iter` - Maximum number of iterations that the model can make to converge. I set this at a constant value of 10,000. Lower numbers prevented convergence occassionally.
- `tol` - A value that allows for early stopping. It creates a threshold to stop training based on how the loss changes. I tested various values using grid search.
- `learning_rate` - The learning rate schedule used to change the learning rate at each step of training. I used the default of `optimal`.

The final ensembler was an ordinary least square linear regression model to stack the predictions of the three sub-models into a single prediction. Stacking is an ensembling technique where the predictions made by subordinate models are used as features for a final model. In contrast, a simpler method for regression ensembling could be to simply average the outputs of the subordinate models. Stacking is particularly powerful as the final model is able to base its predictions on the strengths (and weaknesses) of the subordinate models. Also, the smoothing effect of linear regression allows for better bias-variance trade-off.

Grid search and 10-fold cross-validation were used to explore a wide hyperparameter space. Using the GridSearchCV class, I used R^2 as the metric for evaluating each validation fold, refit the model based on the best hyperparameters, and used parallel processing with two cores to accelarate the processing time.

Deep Neural Network Model

For the purposes of this exercise, I designed a simple Multilayer Perceptron that can train relatively quickly, allowing me to iterate and test my development in the shortest amount of time possible. Multilayer Perceptrons are fully connected deep neural networks (with more than one hidden layer) where each node in the network is connected. Each node is essentially a linear model (perceptron) and are weighted based on how important that node is. Each node has its own weights that the model learns. These weights make up the majority of the parameter space of the model. After weights are computed at each layer as data travels through the network, an activation function is applied to smooth the weighted values using methods such as a sigmoid function or rectified linear units (ReLU). This process is repeated at each layer until the final output layer is reached, where all of the weights that were learned and the final representation of the data is used to make the final prediction. After the final layer is reached, the process of back propagation begins. In back propagation, the model predictions are evaluated for each observation. It then travels backwards in the network starting from the final layer and readjusts the weights and nodes in order to minimize the error using gradient descent.

Keras is a powerful deep learning library that is a wrapper for many deep learning implementations like Theano and TensorFlow. It provides a simple interface for building complex networks with all the necessary parameters and settings.

For my network, I used the following Keras options:

- `dropout` - A dropout layer is a step where the features are randomly down-sampled in order to reduce complexity for the next hidden layer. It is used to prevent overfitting.
- `activation` - Activation functions normalize the outputs from each of the nodes (e.g. to a value between 0 and 1) and help solve the vanishing gradient problem. Here, I use the ReLU function, which sets the derivatives to 1 or 0 using conditional logic. Negative values are all 0 making this a simple non-linear function.
- `epoch` - The number of iterations of back propagation to do. I set the model to allow for up to 500 epochs.
- `batch_size` - The subset of data to push through the network at a time for back propagation. I use 32.
- `callbacks` - Keras provides a `callbacks` feature which allows the user to control the training with a variety of functions. I use three for this project.
  - `ModelCheckpoint` - Saves the best weights and/or model from a given epoch during training.
  - `LearningRateScheduler` - Control scheduling of the learning rate decay function defined. I use a step-based decay where the learning rate is decremented every five epochs. Learning rate helps reduce variance in training to better find local minima.
  - `EarlyStopping` - Early stopping is the method of stopping training if no improvements in a chosen metric are seen after a certain number of epochs. I set the number at 12 epochs of no improvement and based improvements on the validation error.
- `validation_split` - Keras allows the user to split the input data into train and validation partitions directly in the network. I use 15% of the data for computing validation loss in gradient descent.
- `kernel_initializer` - The method for setting the initial random weights of the model. This is required prior to training. For example, setting them all to zero. I use random variables from a normal distribution at each layer.
- `loss` - A number of loss functions are available in Keras. I use `mean_absolute_error` as that is the same metric used for the Kaggle competition.
- `optimizer` - Gradient Descent optimization algorithms such as `adam` and `SGD`. I use `SGD`, or the stochastic gradient descent optimizer, which works by subsetting the data for training and

gradient descent. Doing this greatly speeds up the gradient descent calculations by reducing the data overhead that would be needed when using the full training set.

Network architecture (layer shape is number of nodes):

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1024)              156672
_________________________________________________________________
dense_2 (Dense)              (None, 512)               524800
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 256)               131328
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                16448
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_6 (Dense)              (None, 16)                528
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 17
=================================================================
Total params: 831,873
Trainable params: 831,873
Non-trainable params: 0
_________________________________________________________________
```

Benchmark Results

The benchmark model was trained and validated on the training data with a 15% validation split. The best parameters were chosen using grid search with 10 fold cross validation and determined by generalizability on the validation set.

Best parameters:

- Sub-model 1: Decision Tree Regressor
  - max_depth: 20
  - max_features: 50
  - min_samples_leaf: 0.01
  - min_samples_split: 0.05
- Sub-model 2: SGD Regressor
  - learning_rate: optimal
  - max_iter: 10000
  - penalty: l2
  - tol: 0.01
- Sub-model 3: Linear Regression

– No parameters searched

Figures 8 and 9 show the R^2 and MAE results, respectively. We can see that the model is not overfit with relatively decent performance scores indicating that it is a suitable benchmark for the deep learning comparison.

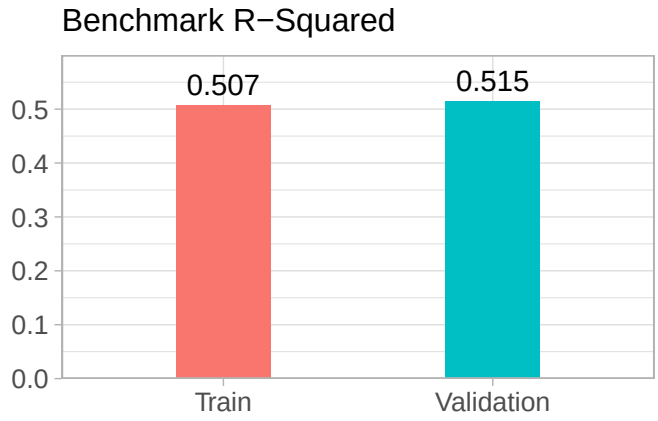

Figure 8: R-Squared results of the benchmark ensemble model

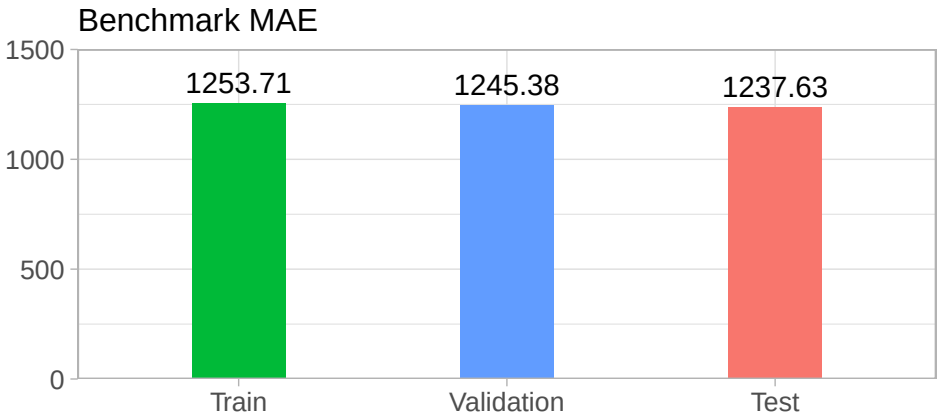

Figure 9: MAE results of the benchmark ensemble model

Below, I summarize the results of the iterations I made while training the benchmark model.

Table 2: Benchmark ensemble model training iterations

Model.Algorithms	Feature.Selection.Threshold	Hyperparameter.Space	Result
LinearRegression	Gridsearch	{'varImp___threshold': ['0.5*mean', '0.75*mean', 'mean', '1.05*mean', '1.25* mean', '1.5*mean']}	0.5*mean had best cross validation results, but model was overfit on validation partition; Higher values underfit
LinearRegression	Comparison between mean and 1.05*mean	None	Similar performance; Decided to use 1.05*mean to be on the conservative side
Ensemble pipeline; Sub-models include ExtraTreesRegressor, Lasso, LinearRegression	1.05*mean	None	Highly overtrained and took a long time to run a single iteration
Ensemble pipeline; Sub-models include ExtraTreesRegressor, Lasso, LinearRegression	1.05*mean	{'submodels___etr___model___n_estimators': [5, 10, 20], 'submodels___etr___model___max_depth': [5, 10, 20], 'submodels___etr___model___min_samples_leaf': [0.01, 0.05, 0.10, 0.25, 0.50], 'submodels___las___model___alpha': [0.1, 0.01, 0.001],}	ExtraTreesRegressor was too slow (grid search on this hyperparameter space would have taken about a week); Killed the job
Ensemble pipeline; Sub-models include DecisionTreeRegressor, Lasso, LinearRegression	1.05*mean	{'submodels___dtr___model___max_features': [5, 10, 20], 'submodels___dtr___model___max_depth': [5, 10, 20], 'submodels___dtr___model___min_samples_leaf': [0.01, 0.05, 0.10, 0.25, 0.50], 'submodels___dtr___model___min_samples_split': [0.15, 0.25], 'submodels___las___model___alpha': [0.1, 0.01, 0.001],}	Still too slow to iterate effectively using GridSearchCV; Issue is with Lasso; R-squared reached 0.50 range in the CV folds; Killed the job
Ensemble pipeline; Sub-models include DecisionTreeRegressor, SGDRegressor, LinearRegression	1.05*mean	{'submodels___dtr___model___max_features': [5, 10, 20], 'submodels___dtr___model___max_depth': [5, 10, 20], 'submodels___dtr___model___min_samples_leaf': [0.01, 0.05, 0.10, 0.25, 0.50], 'submodels___dtr___model___min_samples_split': [0.15, 0.25], 'submodels___sgd___model___penalty': ['l1', 'l2'], 'submodels___sgd___model___learning_rate': ['optimal'], 'submodels___sgd___model___max_iter': [2000, 7000],}	CV iterations were much quicker with SGDRegressor; Appears to be slower with L1 regularization than L2 confirming Lasso was issue with slowness in previous iterations; SGD was not able to converge consistently with max_iter values of 2,000 and 7,000; Killed the job
Ensemble pipeline; Sub-models include DecisionTreeRegressor, SGDRegressor, LinearRegression	1.05*mean	SGDRegressor penalty comparison between L1 and L2 regularization and max_iter increased to 10,000	L2 (Ridge) regularization was much faster; A performant combination found for effective grid search; SGD convergence consistently achieved with max_iter at 10,000
Ensemble pipeline; Sub-models include DecisionTreeRegressor, SGDRegressor, LinearRegression	1.05*mean	{'submodels___dtr___model___min_samples_split': [0.01, 0.05, 0.10, 0.25, 0.50], 'submodels___dtr___model___max_depth': [2, 5, 10, 20, 30, 50], 'submodels___dtr___model___min_samples_leaf': [0.01, 0.05, 0.10, 0.25, 0.50], 'submodels___dtr___model___max_features': [2, 5, 10, 20, 30, 50],'submodels___sgd___model___learning_rate': ['optimal'], 'submodels___sgd___model___max_iter': [10000], 'submodels___sgd___model___penalty': ['l2'], 'submodels___sgd___model___tol': [0.01, 0.001, 0.0001],}	27,000 total fits made on 2,700 hyperparameter combinations; Model trained in about 16 hours; Best parameters found and validated

III. Methodology

Data Preprocessing

As mentioned earlier, categorical variables were one-hot encoded, the target variable was log-transformed, and a decision tree was used to select relevant predictors. To allow more time on the deep learning portion and prototyping a machine learning platform, I decided to forego any further

improvements to the feature space as I found reasonable comparisons can be made with just these few. Also, the data were anonymized so my ability to engineer better features was greatly limited. The only other pre-processing step was to shuffle the training data to remove bias from ordering.

Implementation

Technology Stack

This project was developed on a personal computer with the following specifications:

- Software (Open Source)
  - Ubuntu 18.04 Linux Distribution
  - Anaconda 4.5.10 with Python 3.6.5
  - numpy==1.14.3
  - pandas==0.23.0
  - scikit-learn==0.19.1
  - scipy==1.1.0
  - tensorflow==1.10.0
  - Keras==2.2.2
  - seaborn==0.8.1
  - matplotlib==2.2.2
- Hardware ($1,100)
  - Processor: AMD Ryzen 3 4-Core 3.4GHz
  - Storage: 2TB Hybrid SSD+HD
  - Memory: 16gb DDR4 RAM
  - GPU: 2x GeForce GTX 1070

Machine Learning

Classes

In order to simulate/prototype a functional application, I created classes that wrap the benchmark ensemble learning and deep learning pipelines into self contained modules. All data, predictions, model objects and diagnostics are written to standard output directories. The user only needs to pass configurations to corresponding run scripts to invoke learning.

Methods

Each class contains 3 methods: [1] buildModel to run feature selection and train the model, [2] makePredictions to make predictions on canonical data, and [3] getDrivers to get feature rankings based on the feature selection algorithm. These classes also store all metrics of interest such as training scores, validation scores, and training run times as attributes.

The ensemble learner's buildModel method contains a pipeline for the algorithms. The user passes a parameter grid as a dictionary, fraction of data to partition for validation, and an argument for a feature selection cutoff threshold to control learning and model complexity.

With the deep learner, the user can define the network architecture as a function, a learning rate decay function, epochs, feature selection cutoff, batch size, number of epochs and the fraction of data to partition for validation. Early stopping of 12 epochs without improvement was applied.

Coding Complications

The most challenging part of the implementation was combining scikit-learn's Pipeline API with the GridSearchCV class for sub-models within a FeatureUnion for stacking. Firstly, a helper class was needed to provide an estimator with a fit_transform() method for each sub-model so that these

models can output predictions within a FeatureUnion. The sub-model predictions are then passed to a final ensemble algorithm to stack the sub-models in order to create a single output. Being able to pass parameters to these nested sub-methods is where the bulk of trial and error (and to be honest, frustration) occured.

Docstrings were included where needed with examples of how to configure each of these components and pass hyperparameters to sub-models.

Helper Functions

Three helper functions were created for preparing data for the learner classes. The first simply reads in data from a standard location and outputs Pandas dataframes. The next method shuffles, one-hot encodes and separates the input features from the label column. It also stores down an ordered list of the features. The final helper function reconciles new data to the same layout as the training data using that ordered list. If columns are missing from the new data, they get added in the correct position and filled in with zeroes.

Outputs

Important outputs include:

1. Pipeline model object as *.pkl (benchmark) or Keras best model checkpoint as *.hdf5 (deep learner).
2. Feature selection pipeline object as a *.pkl file.
3. Reports:
    - Feature importances and rankings spreadsheet
    - Training reports: training and validation scores, run times, best model parameters, helpful charts for diagnostics

Refinement

I started out building a working benchmark model pipeline first with fairly complex algorithms as my sub-models such as the Extra Trees Regressor. I eventually found that the computational costs of running this would be a serious impediment to completing this project. So I decided to take the approach of treating this as a proof of concept and greatly simplify the algorithm space with simpler more efficient algorithms optimal for this data.

Before finding a set of sub-models that trained reasonably fast, I worked on building out the feature selection pipeline. I started going down the path of an ensembling technique for feature selection, which I found to be far too computationally expensive and ended up with a single model (decision tree). I did leave the feature selector as a pipeline to work on this at a later point since I believe an ensemble approach could be powerful in providing better data for learning. I attempted various thresholds to find a standard cut-off to use to avoid having to re-run the feature selector at each training iteration using a single linear regression (OLS) model to evaluate the performance of the threshold.

Once these two components were brought to a state to allow quicker development, the remaining effort was focused on building out the classes and methods to build out the prototype. I relied mostly on built-in methods provided by the Python libraries to simplify the task, continuously testing on a small sample of the Allstate data to ensure things worked correctly before iterating the actual model training on the full dataset.

With the deep learning model, I started with a single layer neural network as my test case. From here I started adding layers until I found a model that was able to beat the benchmark model and with limited overfitting.

Below is a summary outlining the iterations I made with the deep learning model.

Table 3: Deep learning model training iterations

Changes.to.MLP	Feature.Selection.Threshold	Result
Single Hidden Layer Neural Network	1.05*Mean	Model learned nothing; R2 scores were near 0 and MAE was about 2,500
Add two hidden layers; Learning Rate Decay; Early Stopping	1.05*Mean	Model predictions was just predicting the mean and not learning; Found issue to be ADAM optimizer but unclear as to why
Swap out ADAM for SGD optimizer	1.05*Mean	Model learning and predictions improved dramatically reaching R2 scores of around 0.40-0.45
Add 3 more layers; Dropouts; Increase Nodes	1.05*Mean	Significant improvement in R2 and MAE; reaching 0.5 (R2) and about 1,400 MAE
Lowered batch size to 16	0.5*Mean	Model was overtrained due to wider feature space
Increased batch size 100	1.5*Mean	Model performance dropped back down to the 0.4 (R2) range
Lowered batch size to 64	0.75*Mean	Model was overtrained but not as much as 0.5*Mean in feature selection
Lowered batch size to 32	1.25*Mean	Model fit was back to around the 0.45 (R2) range
Kept the network	1.05*Mean	Model performance increased to 0.55 (R2) range and was not overfit; Benchmark beaten

IV. Results

Model Evaluation and Validation

The final network architecture for the deep learning model is a Multi-Layer Perceptron with 5 hidden layers. It's architecture is outlined earlier in the *Algorithms and Techniques* section. I built up from a simple single layer model until I found an architecture and parameters that were able to beat the benchmark.

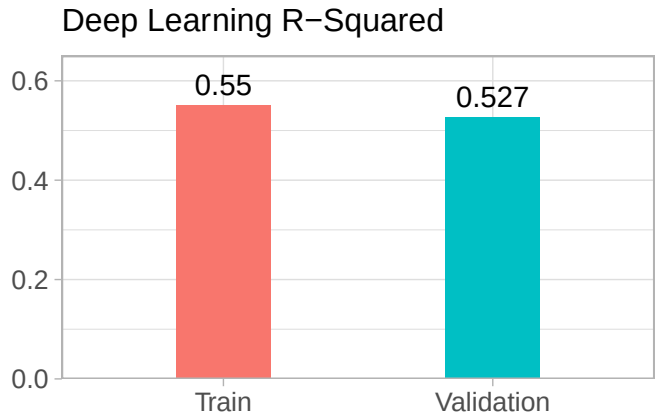

Figure 10: R-Squared results of the deep learning model

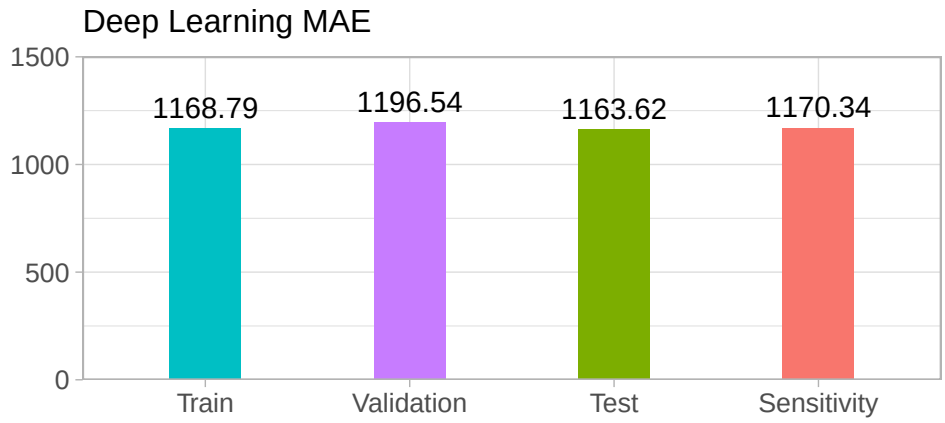

Figure 11: MAE results of the deep learning model

Figure 10 shows that I was able to achieve better fitted models than our benchmark in both training and validation. It is important to note that the computation and complexity in implementing the benchmark model was far greater than the deep learning solution, which I found interesting.

Figure 11 shows how the deep learning model performed in terms of MAE. Again, it did much better than the benchmark across the board. The Test MAE was extracted from Kaggle (as labels are hidden in the test data provided). Although, this simple model was not able to beat the median score of 1,125, it still performs reasonably well given the simplicity of the network. I also did a sensitivity test on the model's robustness against noise by converting one of the strongest predictors (cont14) to random noise. Although the error increased in the test data, it is still reasonably close to the training MAE.

GPU vs CPU

In the final comparison, I looked at metrics for understanding the costs both financially and computationally for a deep learning solution.

- Cost per R^2 gained over benchmark
  - 2 Nvidia GeForce GTX 1070's cost about $800 market value
  - Assuming that the only investment is GPU's, and with about a 1.2 point increase in validation R^2, the cost equates to approximately $667 per point increase in R^2
- Training and scoring times
  - Training
    - CPU: 1,434.7 seconds (total) | ~42 seconds (per epoch)
    - GPU: 548.9 seconds (total) | ~20 seconds (per epoch)
  - Scoring New Data (time includes feature selection transformation and writing results on 125k records)
    - CPU: 8.5 seconds
    - GPU: 4.7 seconds
- Storage costs of each model
  - Benchmark: 926KB
  - Deep Learning: 6,700KB

Model Robustness Testing

Beyond the simple sensitivity test using the test data, I tested the model consistency in both predictions and learning. For the predictions, I bootstrap resampled with replacement 25% of the

training data 1,000 times and measured the R^2 and MAE of the predictions using the final network. For the learning consistency, I bootstrap resampled with replacement 50% of the training data 50 times, split the data into 80%/20% train-validation splits at each sampling, trained the network on 80% and validated on the 20% partition. I measured R^2 and MAE in both the train and validation splits to determine consistency in both performance and generalizability. Using many samples, I captured the mean, standard deviation and 90% confidence intervals presented in the following tables. The performance held steady across the bootstraps.

Table 4: R-Squared Robustness Testing

Test	Bootstraps	Mean	SD	CI_Lower	CI_Upper
Prediction Stability	1000	0.5478	0.0037	0.5476	0.5479
Train Stability In-Sample	50	0.5446	0.0066	0.5430	0.5461
Train Stability Out-of-Sample	50	0.5281	0.0065	0.5266	0.5296

Table 5: MAE Robustness Testing

Test	Bootstraps	Mean	SD	CI_Lower	CI_Upper
Prediction Stability	1000	1158.890	7.2358	1158.514	1159.266
Train Stability In-Sample	50	1163.139	14.2433	1159.825	1166.453
Train Stability Out-of-Sample	50	1186.702	14.5125	1183.326	1190.078

Justification

As showcased above, the deep learning solution performed significantly better than the benchmark in both R^2 and MAE. The benchmark itself is a fairly decent model that generalizes well making it a good comparison. The deep learning model also outperformed the benchmark in train, validation, and test iterations by a signficant amount. I believe that this clearly demonstrates that deep learning can be a viable solution for building competitive models for predicting financial risk.

Recall the SOA comparison of healthcare models in Figure 1. In terms of performance across solution offerings, it is common to see R^2 values be really close. Vendors are fighting over small improvements over each other. At face value, a 1.2 point improvement in validation R^2 for the deep learning solution may not seem like much. However, given this context, we can imagine that even a marginal improvement in accuracy can be a highly valuable result. According to a 2015 study by the US Department of Transportation[6], automobile accidents cost about $241 billion in the US in 2010. When putting dollars like that into perspective, the opportunities provided by even a seemingly marginal increase can lead to significant outcomes for both insurers and consumers alike.

Lastly, a $667 per R^2 investment for a fully production grade system in-house consisting of hundreds of GPU's is high. Factoring in the other costs like time and systemic infrastructure improvements required, then it stands to reason large organizations may be hesitant to invest in a solution with no guarantee. However, these techonologies are becoming more affordable and better. The market is seeing more competition with Google's TPU and AMD making strides in deep learning. It is only a matter of time before the question is no longer whether it is worth investing in. Plenty of cloud solutions are available, and are being used in restricted use cases. However, certain insurance domains are naturally wary of passing data to cloud solutions, even if only for computations, given the nature of the data they collect (e.g. healthcare data).

To summarize, deep learning is certainly a viable solution for financial risk prediction in insurance. As of now, such incremental gains in performance is still difficult to justify investing in the labor and infrastructure required for an optimal system. However, as with all other technologies, the marketplace will soon offer affordable and better solutions to help make deep learning a norm in this industry.

V. Conclusion

Free-Form Visualization

Here I want to highlight that the deep learning networks converge very quickly. I believe this could be due to either the network architecture itself or perhaps the learning rate was too aggressive. Regardless, this is an area I am continuing to research as I further develop my expertise in this problem.

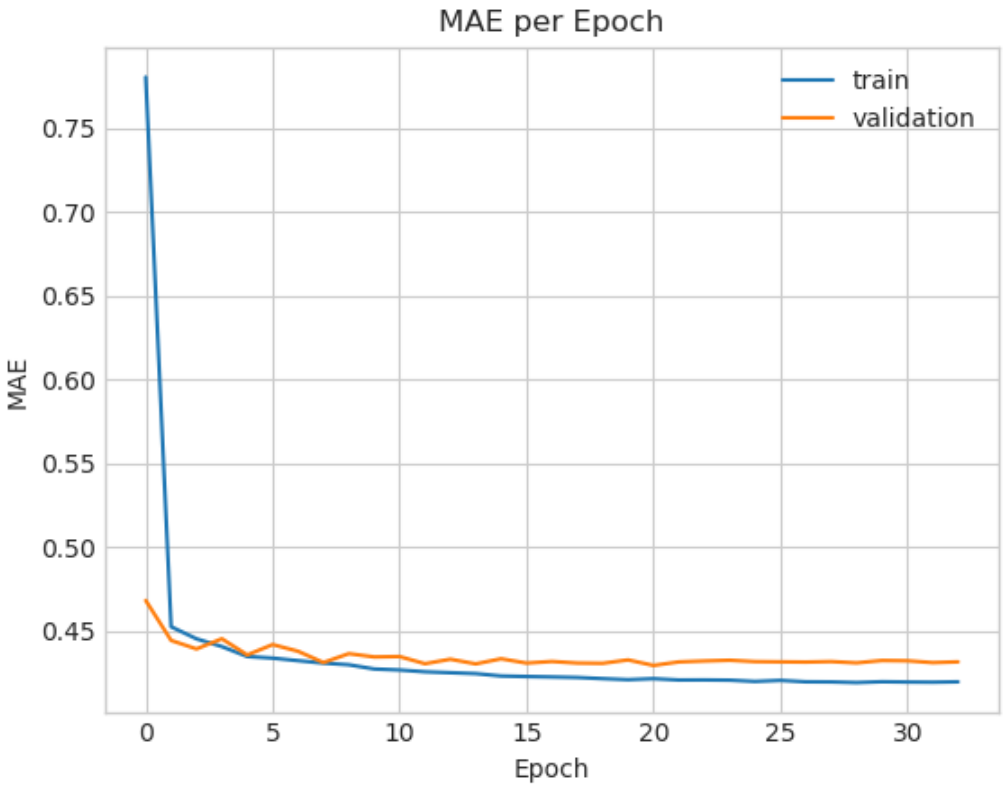

Figure 12: MAE per epoch for CPU run

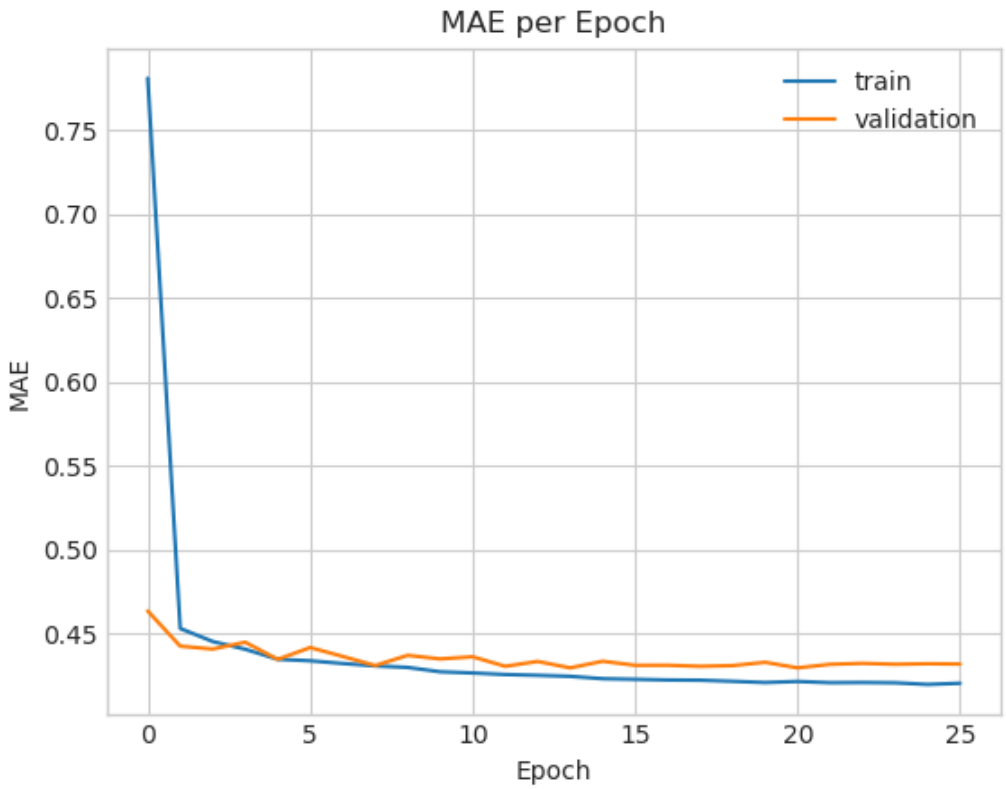

Figure 13: MAE per epoch for GPU run

Reflection

This project can be summarized in the following steps:

1. The problem domain was researched for a better understanding of what I can solve with this data.
2. The data were downloaded and explored in R (my preferred ad-hoc analysis tool) with relevant visualizations created.
3. Determined the approach to take. I decided to build learner classes and a prototype over an interactive approach via notebooks to have a reusable tool.
4. Build benchmark learner and train many iterations until a reasonable set of hyperparameters was found.
5. Build a deep neural network learner and try different networks until I beat the benchmark model without overfitting.
6. Run the same model through GPUs for performance comparisons.
7. Send all predictions to Kaggle for one true test evaluation.

The most difficult part was building working learner classes for this exercise for a problem that is relatively complex. I had to make several concessions along the way to build a simpler solution. One of my main reasons for going with this Kaggle competition was that the data were already cleaned and anonymized so I could focus less on the time consuming part of data preparation and focus more

on the machine learning aspects. There is still quite a bit of work needed to make my classes more generalizable to new problem sets, but I believe the core functionality is laid out.

Improvement

In order to have a more generalizable solution, I plan to look at the following in the near-term:

1. Decouple the prediction functionality from the model learner classes to have a more modular solution.
2. Implement methods optimized for GPU processing of neural networks (e.g. multi_gpu_model() in Keras)
3. Currently, the classes are only tested on regression problems. I plan to create classifiers as well.
4. To build a production grade system, there is a balancing act between how much flexibility to give the user and how automated the tool needs to be to iterate quickly. I think one test case is certainly not enough to find this balance. More problems to solve will help find it.
5. Test various network architectures like RNN's to see if they work.
6. Minor improvements to pathing and file naming.

Sources

[1] https://www.soa.org/research-reports/2016/2016-accuracy-claims-based-risk-scoring-models/

[2] https://www.kaggle.com/c/allstate-claims-severity

[3] https://www.equifax.com/business/insight-score-insurance/

[4] https://www.verscend.com/solutions/performance-analytics/dxcg-intelligence

[5] Exploratory Data Analysis guided by Kaggle Kernel submissions (https://www.kaggle.com/c/allstate-claims-severity/kernels)

[6] https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812013

Supplemental Information

All code and additional documentation and visualizations can be retrieved from my github repository.

```
git clone https://github.com/eugenekwak/Allstate_DL.git
```