Bayesian Optimization for Automatically Generating Neural Networks to Model Customer Churn

By Patrick Di Rita

Washington University in St. Louis

Data

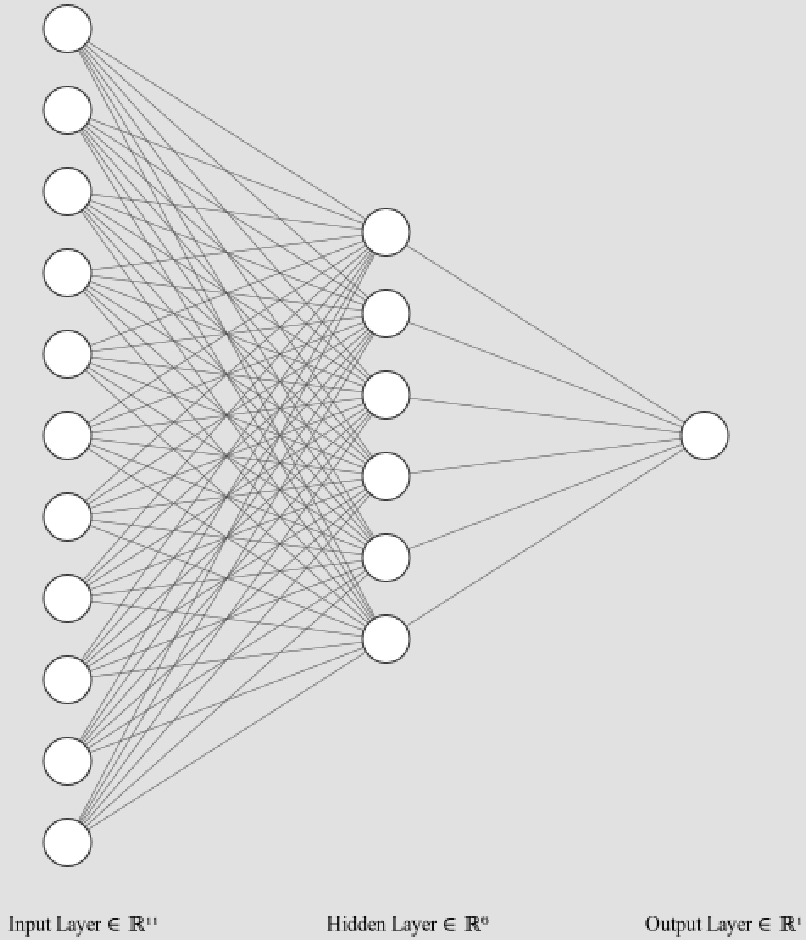
Goal: predict customer churn (exited field) with high accuracy

CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101348.88	1
15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
15619304	Onio	502	France	Female	42	8	159660.8	3	1	0	113931.57	1
15701354	Boni	699	France	Female	39	1	0	2	0	0	93826.63	0

CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Geography_Germany	Geography_Spain	Gender_Male	Exited
619	42	2	0	1	1	1	101348.88	0	0	0	1
608	41	1	83807.86	1	0	1	112542.58	0	1	0	0
502	42	8	159660.8	3	1	0	113931.57	0	0	0	1
699	39	1	0	2	0	0	93826.63	0	0	0	0

- 80/20 train/test split, scaled to standard normal
- Training further split into 80/10/10 train/val/hyp sets

Model: Artificial Neural Network



Control parameters:

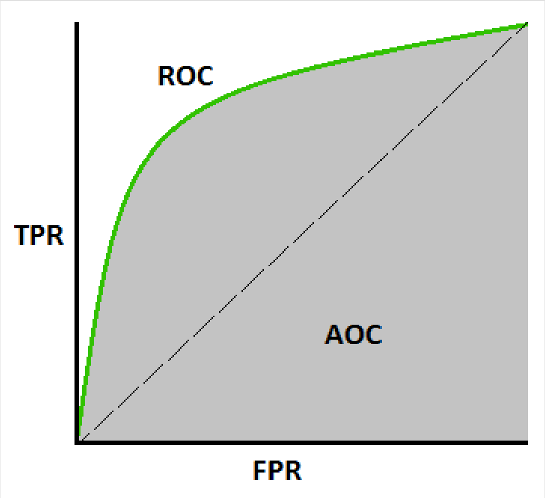
- ANN:
  - Input layer: 11 nodes
  - 1 hidden layer: 6 nodes
  - Dropout: 10%
  - Activation: Sigmoid
- ADAM Optimizer:
  - Learning rate: 0.001
  - Batch size: 10
  - Workers: 4
  - Weight Decay: 0

Optimization objectives and evaluation metrics

- Model fitting:
  - Neural network trained to minimize Binary Cross Entropy loss function

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

- Model evaluation:
  - Trained network evaluated by measuring area under the ROC curve (AUC)



https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

Bayesian optimization setup

Goal: use Bayesian optimization to optimize ADAM hyperparameters as well as ANN architecture hyperparameters

Implementation framework: PyTorch, GPyTorch, BoTorch, Ax stack

Surrogate model: Gaussian process with Matern 5/2 kernel

Optimization objective: Maximize AUC on hyp set

Acquisition function:

Control:	Expected Improvement
Experimental:	Bandit acquisition function

Bayesian optimization setup (cont.)

ADAM hyperparameter search space:

lr:	[1e-6, .4] – log scale
batch size:	[1, 100]
workers:	[0, 16]
weight decay:	[0, .5]
epochs:	100 – fixed

ANN hyperparameter search space:

hidden layers:	[1, 10]
first hidden dim:	[3, 100]
hidden growth:	[.1, 2]
dropout:	[0, .9]
activation:	[ReLU, LReLU, tanh]

SOBOL random initialization

↓

T rounds of Bayesian optimization:

1. Train network on nominated parameterization
2. Evaluate network on hyp set
3. Update model with observation

↓

Train final model using best parameterization on full training data

↓

Evaluate on test set to compare models

Bandit acquisition function

Intuition:

At each nomination step within the Bayesian optimization loop, optimize a set of acquisition functions rather than just one, and choose a nomination from the set with probability proportional to the cumulative rewards of its corresponding acquisition function. In other words, turn the acquisition function choice into its own multi-armed bandit problem

Reasoning:

There is no guarantee that an acquisition function chosen at the beginning of an experiment will remain optimal over the duration of the experiment

Observing points from a set of acquisition functions will lead to greater observation diversity and exploration of the search space

Bandit acquisition function implementation

```python
def run_bandit_acquisition(experiment, acq_funcs, num_trials, search_space, w, lmb):

    acq_rewards = np.zeros(len(acq_funcs))
    reward_history = []
    acq_choices = []

    for i in range(num_trials):
        print(f'-------BANDIT TRIAL {i}-------')
        trial_nominations = []

        for acq_name, acq in acq_funcs.items():
            print(f'---Generating nomination for {acq_name}---')
            if acq_name == 'ucb':
                nomination = acq(experiment=experiment, data=experiment.eval(), acqf_constructor=get_UCB)
            else:
                nomination = acq(experiment=experiment, data=experiment.eval())
            trial_nominations.append((nomination, nomination.gen(1)))

        if i:
            for j in range(len(acq_funcs)):
                currNom, currPoint = trial_nominations[j]
                model_posterior_mean = currNom.predict(currPoint.arms)[0]['auc'][0]
                acq_rewards[j] = w * acq_rewards[j] + model_posterior_mean
            reward_history.append(acq_rewards)

        chosen_idx = select_acq(acq_rewards, lmb)
        chosen_nom, chosen_point = trial_nominations[chosen_idx]
        acq_choices.append(list(acq_funcs.items())[chosen_idx][0])
        print(f'Chosen acquisition function: {acq_choices[-1]}')

        batch = experiment.new_trial(generator_run=chosen_point)


    for j in range(len(acq_funcs)):
        currNom, currPoint = trial_nominations[j]
        model_posterior_mean = currNom.predict(currPoint.arms)[0]['auc'][0]
        acq_rewards[j] = w * acq_rewards[j] + model_posterior_mean
    reward_history.append(acq_rewards)

    return reward_history, acq_choices
```

1. Optimize all acquisition functions and get their nominations
2. Update reward history based on cumulative rewards of each acquisition function
3. Choose a nomination with probability proportional to cumulative rewards
4. Add chosen nomination to experiment and evaluate at that point

Results and analysis

Control Architecture + Control ADAM:	0.8569
Control Architecture + Optimized ADAM (EI):	0.8650
Optimized Architecture + Optimized ADAM (EI):	0.8709
Optimized Architecture + Optimized ADAM (Bandit):	0.8734

Optimal parameters:

ANN:

Hidden layers:	5
First hidden dim:	15
Dropout:	.082
Hidden growth:	1.779
Activation:	TanH

ADAM Optimizer:

Learning rate:	0.00169
Batch size:	65
Workers:	5
Weight Decay:	0

Conclusion

- Bayesian optimization is a viable and efficient option for neural architecture search and hyperparameter optimization

- The proposed bandit acquisition function was able to outperform vanilla Expected Improvement, confirming the original hypothesis

- The improvement was not by a huge margin, but the bandit-optimized model performed better than any other models found on Kaggle

- Further work is to apply this same framework to a harder dataset, as ANNs reach the limit of this dataset very easily