Disguise Adversarial Networks for Imbalanced Ad Conversion Datasets

Progress Report II

James F. Xue, Derek Zhao

1 Review of first progress report

In our first progress report, we provided an overview of the Disguise Adversarial Network (DAN) architecture, the motivations for applying it to MediaMath's imbalanced advertising campaign data, and initial experimental results on synthetic data. While those results did not definitively show that using a disguise network for minority class augmentation improves classification performance over a common neural network, they allowed us to better visualize and understand the behavior of the disguise network. Through testing our implementation of the DAN on synthetic data, we were able to quickly debug and reasonably confirm the correctness of the model.

Although model design had progressed rapidly, the same was not true for the data processing pipeline required to serve data to the DAN. This caused a development bottleneck that prevented us from evaluating our implementation on MediaMath data. Since then, we have completed two data processing pipelines (Section 2), expanded the functionality of our DAN implementation (Section 3), and produced initial results for DAN performance on the CPC dataset (Section 4).

2 Data processing

Two data processing pipelines were developed to serve MediaMath data to the DAN, with each pipeline providing a different representation of the data as well as a different means of supplying the data. We refer to the pipelines as the 1) memory-based pipeline and 2) generator-based pipeline.

2.1 Memory-based pipeline

The memory-based pipeline was initially developed as a stopgap measure due to the lack of progress towards completing a more scalable one. Broadly speaking, it loads an entire raw dataset from disk into RAM, performs a set of transformations using rules learned from the training set, and writes the processed dataset back to disk. When training the DAN, the entire processed training and validation sets are loaded back into memory for the model to fit on in batches.

Due to its lack of scalability, this pipeline is only intended for use with the CPC dataset and only to produce a transformed representation of the data suitable for embedding high-cardinality features. Features to be represented as an embedding are typically index-encoded, with each unique value mapping to a unique integer that functions as the lookup index of an embedding matrix. Thus, the transformed data is compact compared to alternatives such as one-hot encoding or feature hashing, and a fully in-memory transformation is feasible.

As shown in Figure 1, raw data is downloaded as a series of GZIP files. The **fit and transform** stage of the pipeline decompresses the data and fits a set of custom scikit-learn transformers on the training data and applies them to all data, resulting in the following:

- A set of user-specified features (such as **column_weights**) are dropped.
- All continuous features are imputed with 0.
- All continuous features are standardized.
- All categorical features are imputed with -1.
- All categorical features with cardinalities above a user-specified threshold are index-encoded.

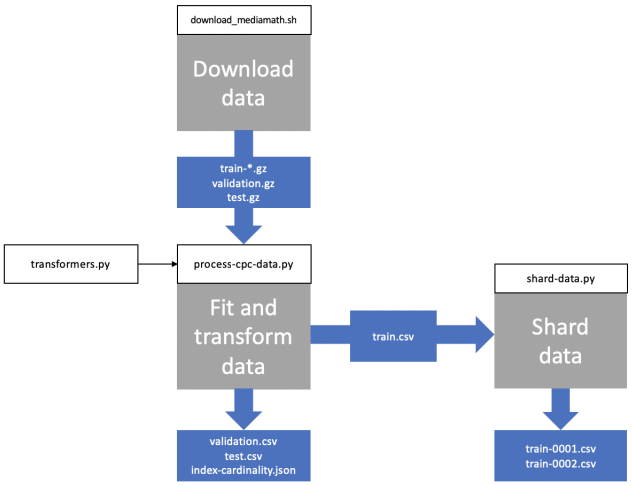

Figure 1: **A memory-based pipeline**

- All categorical features with cardinalities below or equal to a user-specified threshold are onehot-encoded.

The fit and transform stage outputs a single CSV file per dataset split (train, validation, and test) as well as a JSON mapping the column index of each categorical feature to its cardinality (**index-cardinality.json**). The latter is crucial for transforming categorical features into their embedded representations and described more fully in the next subsection. While sharding the training data is not necessary because the entire training split is loaded into memory during model fitting, it is applied nonetheless in case generators are developed for this pipeline later on.

2.2 Generator-based pipeline

The generator-based pipeline was developed to address many of the limitations inherent to the memory-based approach. It achieves scalability through two means:

1. Serving data in batches to the DAN for lower memory consumption
2. Processing data only when served for lower disk usage

The first mechanism is achieved through Tensorflow's CsvDataset API while the second through Tensorflow's FeatureColumns API. Both mechanisms are encapsulated within a class called DataGenerator for ease of use when interfacing with the DAN. Because of the scalability afforded from using data generators, this pipeline is ideal for producing one-hot encoded representations of the data.

As shown in Figure 2, the pipeline begins with downloading the raw datasets as GZIP files. In the **fit and shard** stage, JSON metadata is created for each feature that provides instructions for how to transform the data when served by the data generator. In particular, **numerical-stats.json** contains the mean and standard deviation of each continuous feature so that they may be standardized in batches later, and **categorical-vocab.json** contains the vocabulary list of each categorical feature so that they may be one-hot encoded in batches later.

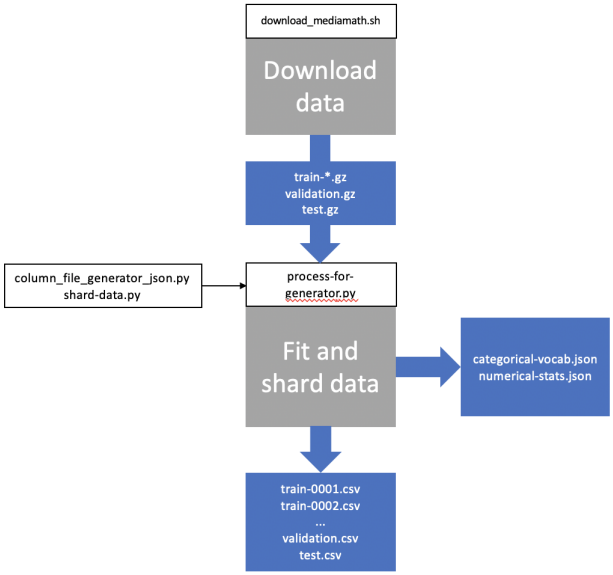

Figure 2: **A generator-based pipeline**

An important implementation detail is that because the training data was originally separated into CSVs containing exclusively negative class data and exclusively positive class data, the training data must be sharded into much smaller CSVs to facilitate random shuffling of negative and positive class data within the data generator.

3 Model development

Since the first progress report, the DAN module has been expanded with additional functionality that enables greater usability and transparency when running experiments.

3.1 Embeddings for high cardinality features

Because saving an entire dataset in its one-hot encoded or feature-hashed form would require prohibitive amounts of disk space, and because it is not apparent how to shard sparse matrices stored on disk, the memory-based pipeline was constructed to transform high-cardinality features into a much more compact index-encoded form. We built an **EmbeddingTransformer** class to facilitate the ingestion of index-encoded data by the model, a novel modification to the DAN framework that Deng et al. did not need to explore due to the much lower cardinality of their datasets. While embedding transformations are a common technique for neural classification tasks, in the context of a DAN, we must be careful about when the transformation is applied.

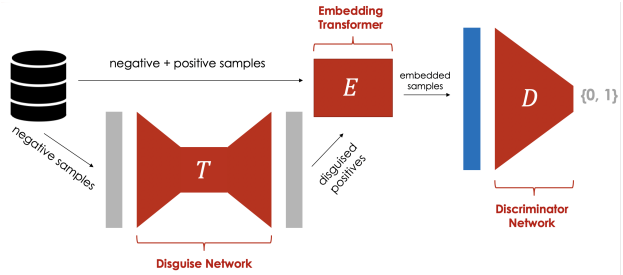

Figure 3: **Incorrect usage of the embedding transformer**

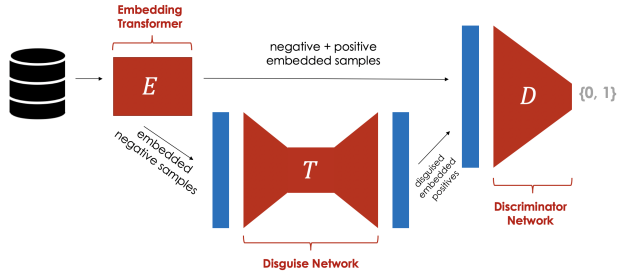

Figure 4: **Correct usage of the embedding transformer**

Embedding samples of data after they have been disguised (Figure 3) is not fruitful because this would require the disguise network to attempt to transform index-encoded categorical data. Such an encoding is devoid of any ordinal meaning and thus would be problematic for the disguise network to transform. Instead, we embed all data before they flow into the disguise or discriminator network (Figure 4), thus requiring the disguise network to create transformed samples in embedded space, and plan the architectures of both networks according to the data's dimensionality in embedded space.

One final implementation detail is that we only allow gradients from the discriminator network to affect how the embeddings are updated during training, as the disguise network should not have any bearing on how the embeddings are learned.

```
embedding_transformer = EmbeddingTransformer(index_map=index_map)

disguise = Disguise(
  num_inputs=embedding_transformer.calc_num_outputs(num_inputs),
  hidden_nodes=[16, 8, 8, 16])

discriminator = Discriminator(
  hidden_nodes=[16, 8])

dan = DAN(
  disguise, discriminator, CHECKPOINT_DIR, LOG_DIR,
  embedding_transformer=embedding_transformer,
  num_inputs=num_inputs)
```

Figure 5: **DAN usage with embedding layers**

Because not all features are categorical or index-encoded, the embedding transformer requires instructions (provided in the form of a map) as to which columns of input data are index-encoded and the cardinality of said index-encoded features. From the cardinality, the embedding transformer calculates the dimensionality of the embedding to be applied. In general, we set the embedded dimensionality of a feature to be the fourth root of the feature's cardinality.

3.2 Tensorboard and metric reporting

To aid in model debugging and monitoring during training runs, we modified the DAN to report metrics to Tensorboard. Table 1 provides details: most metrics are calculated each training step from the training set, but a handful are calculated every epoch, and two are evaluations on the validation set.

- **cross entropy**: Measures the discriminator's performance on labeled training data. Lower is better.
- **predictive entropy**: Measures the discriminator's confidence when classifying unlabeled data. Lower is generally better, but too low a value also suggests the disguise network has collapsed and is not performing useful data augmentation.
- **discriminator loss**: Weighted sum of cross entropy and predictive entropy. Lower means the discriminator network is learning.

metric	evaluation	frequency
cross entropy	train	step
predictive entropy	train	step
discriminator loss	train	step
disguise quality	train	step
regularizer	train	step
disguise success rate	train	step
disguise loss	train	step
train accuracy	train	epoch
train AUROC	train	epoch
validation accuracy	validation	epoch
validation AUROC	validation	epoch

Table 1: **Tensorboard metrics**

- **disguise quality**: Measures the effectiveness of the disguise network at transforming negative samples into positive-seeming samples (from the perspective of the discriminator network). Higher is better, but too high a value could be indicative of disguise network collapse.

- **regularizer**: Measures how much freedom the disguise network is exercising when transforming negative samples into disguised samples. Too low a value suggests the disguise network is behaving more like an autoencoder, yet too high a value suggests the disguise network is more likely to produce less diverse disguises.

- **disguise success rate**: A more interpretable proxy for **disguise quality**, the proportion of negative samples in a batch that, after being disguised, are classified as positive by the discriminator network.

- **disguise loss**: Weighted sum of negative disguise quality and the regularization term. Lower means the disguise network is learning.

- **train accuracy**: Accuracy of the entire training set.

- **train AUROC**: Area under ROC curve for the entire training set.

- **validation accuracy**: Accuracy of the entire validation set.

- **validation AUROC**: Area under ROC curve for the entire validation set.

3.3 Validation-based checkpointing

Previously, the DAN saved a checkpoint of itself following each training epoch and thus required intervention to prevent overfitting during a training run. With the availability of validation metrics for Tensorboard logging, the model now only checkpoints when its validation AUROC improves over its previous optimal validation AUROC and ceases training if no improvement in optimal validation AUROC is detected after three epochs of training.

3.4 Generator functions

The DAN class originally supported only three functions, **fit(X, y)**, **predict(X)**, and **transform(X)**, all of which required an entire dataset to be loaded in memory as a large matrix. To allow the model to train and predict on datasets of arbitrarily large sizes, the following generator-based functions were added to the model after the **DataGenerator** class had been completed:

- **fit_generator(train_datagen, val_datagen)**: calls internal partial fit methods to iteratively train the DAN and evaluate the model's performance on validation data.

- **predict_generator(test_datagen)**: calls internal partial predict methods to infer predictions.

- **predict_proba_generator(test_datagen)**: calls internal partial predict_proba methods to infer predictive distributions.

- **evaluate_generator(test_datagen)**: returns the model's cross entropy loss, accuracy, AUROC.

A significant setback we have encountered in adapting the DAN to work with custom data generators is that while the model loads and checkpoints correctly when trained using the older memory-based functions, it does not do so with the generator-based functions. We suspect this is due to the generators holding their own internal Tensorflow sessions separate from that of the DAN's, resulting in a single process spawning multiple sessions. The DAN still trains and evaluates properly, so meaningful experimentation remains possible, but a trained model currently cannot be recovered once the process concludes.

hyperparameter	value
epochs	10
batch size	256
learning rate	1e-05
disguise hidden layers	128, 128, 128, 128
discriminator hidden layers	64, 32, 16
activations	RELU
batch normalization	yes
dropout	none

Table 2: **Constant hyperparameter settings - CPC dataset with embeddings**

4 Initial results

To train and evaluate the DAN on the CPC dataset, we provisioned a GCP instance with an Nvidia Tesla K80 GPU.

4.1 CPC dataset with embeddings

For the CPC dataset with categorical features represented through dense embedding vectors, we trained four models, one with no disguise network (resulting in a regular DNN classifier) and three with different values for hyperparameter λ, the weight in front of the regularization term in the disguise loss. Additional details are provided in Table 2.

The experiments were not thorough (multiple training runs per hyperparameter combination are necessary to account for the noise caused by random initializations of the DAN) nor were they comprehensive (far more different combinations of hyperparameters must be attempted to find an optimum setting), but they were instructive; through analyzing the DANs' training runs, we learned about a variety of model behaviors.

4.1.1 Setting λ too low

Recall that the loss used to train the disguise network (Figure 6) contains a hyperparmeter λ that balances the importance of minimizing the regularizer term against that of maximizing the disguise quality term.

Setting λ to a low value grants the disguise network more freedom to drastically alter incoming negative samples, but at too low a value, the output of the disguise network may collapse. That is, if the disguise

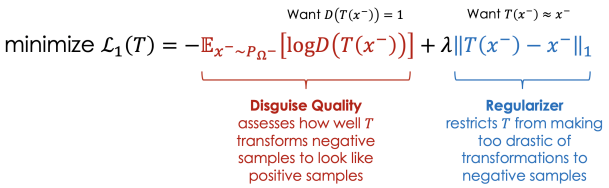

Figure 6: **The disguise loss**

network is unrestricted in how it may transform negative samples, its output will collapse to the same disguised sample, one that has a high probability of being classified as positive by the discriminator network.

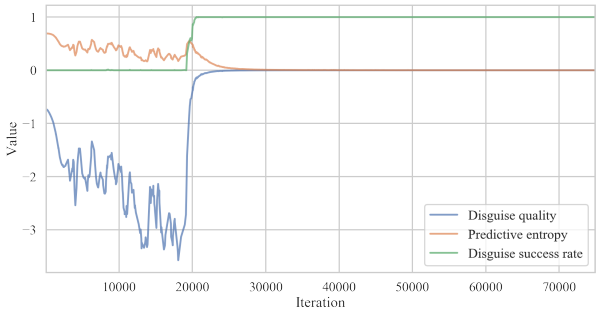

Figure 7: **Possible disguise network collapse at $\lambda = 0.5$**

Figure 7 shows how the disguise quality, disguise success rate, and predictive entropy unfold as a DAN trains with $\lambda = 0.5$. The disguise quality term collapses towards 0, meaning all disguised samples are confidently predicted by the discriminator to be positive. Because the predictive entropy is a measure of the discriminator network's prediction uncertainty for disguised samples, the collapse of the disguise quality towards 0 necessarily results in the collapse of the predictive entropy towards 0 as well.

A predictive entropy of 0 causes the discriminator loss to become equivalent to the discriminator network's cross entropy (Figure 8) since the discriminator loss is a weighted sum of cross entropy and predictive entropy. Assuming that the disguise network is fully responsible for the predictive entropy approaching 0, a collapsed disguise network would serve no useful purpose since the discriminator would train as if the disguise network did not exist. In actuality, the predictive entropy could also approach 0 because it was successfully minimized by the discriminator network.

We observed a surprising result in the relationship

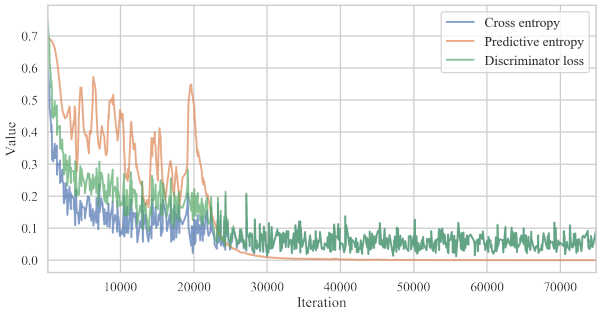

Figure 8: **A collapsed network serves no purpose**

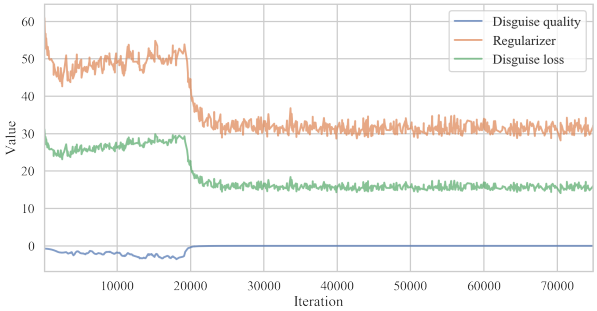

Figure 9: **Disguise loss at $\lambda = 0.5$**

between disguise quality and the regularization term when the DAN trains at $\lambda = 0.5$; we normally expect increases in the disguise quality to coincide with increases in the regularization term, but at around iteration 20000, the collapse of the disguise quality towards 0 coincides with a significant drop in the regularization term. A possible explanation is that if the disguise network is able to learn that only a few features are relevant to successfully disguising negative samples, then it will also learn to perform identity transformations on all non-relevant features to minimize the regularization term.

4.1.2 Setting λ too high

If setting λ to too low a value gives the disguise network too much flexibility, then setting λ to an excessively high value essentially forces the disguise network to behave like an autoencoder and attempt to learn an identity transformation for the negative samples.

Figure 10 shows metrics for a DAN trained at $\lambda = 5$. The disguise quality collapses towards the negative values, indicating that the disguise network is unable to generate successful transformations on the negative samples, resulting in disguise success rates of 0. Because the discriminator confidently predicts all disguised samples to be negative, the predictive entropy falls to 0, and once again the discriminator loss becomes equivalent to the network's cross entropy.

λ	η	accuracy	auroc	prec.	recall
NA	NA	**0.9172**	0.9843	**0.5794**	0.9779
0.5	0.2	0.9169	0.9842	0.5782	**0.9819**
1.0	0.2	0.9170	**0.9845**	0.5786	0.9807
5.0	0.2	0.9171	0.9844	0.5790	0.9789

Table 3: **Hyperparameters and test set metrics**

4.1.3 Setting λ somewhere in between

Figure 11 shows metrics for a DAN trained at $\lambda = 1$. Of the three training runs, this was the only one that generated a prolonged period where the disguise success rate was between 0 and 1, resulting in a non-zero predictive entropy. Unfortunately, it appears that the disguise network still collapses in the final moments of the run.

4.1.4 Evaluation on test set

We evaluated all three DAN models on the test set using their saved state corresponding to their highest validation AUROC and also trained and tested a stand-alone discriminator network with the same architecture as that of the DANs' discriminator networks (Table 3). All four models appear to perform equally and with similar prediction characteristics; they are rather aggressive in predicting positives, leading to high recall but low precision. One interesting pattern is that as λ decreases, recall increases while precision decreases, which matches our intuition about how the DAN is supposed to function.

4.2 CPC dataset with one-hot encodings

Having developed a generator-based pipeline for both CPC and CTR datasets, built a DataGenerator class for serving processed data in batches, and modified the DAN to fit, predict, and evaluate using generators, we were hopeful to run mass experiments using the Test Tube framework on a GCP instance with 8 Nvidia Tesla K80 GPUs. While progress has been made in deploying Test Tube to the cloud, we have

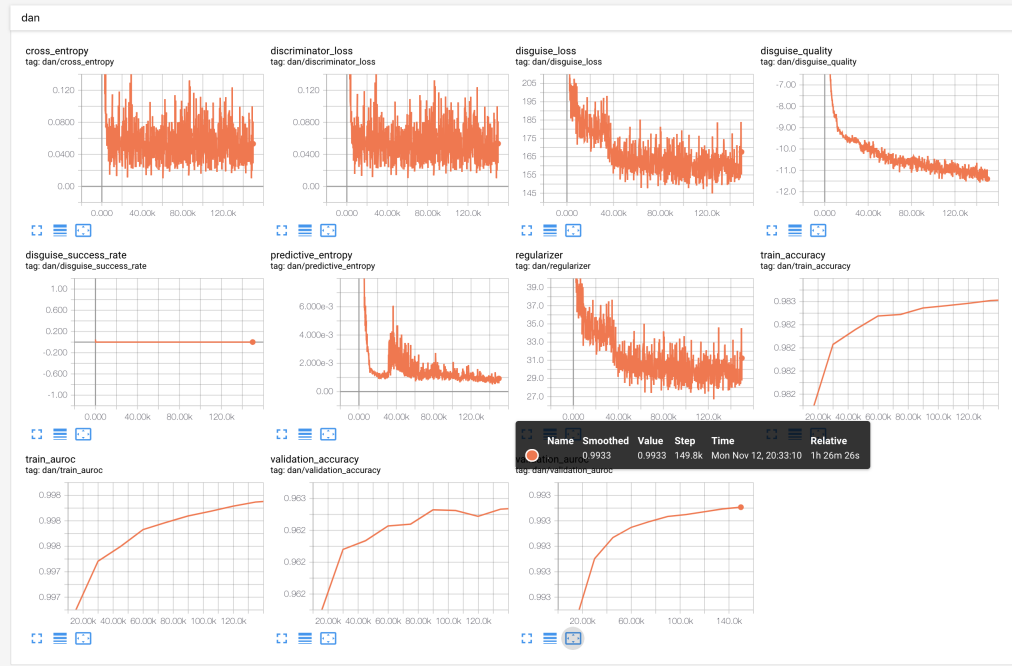

Figure 10: **Tensorboard metrics for DAN training at $\lambda = 5$**

Figure 11: **Tensorboard metrics for DAN training at $\lambda = 1$**

also been met with many challenges, typically related to the interaction of Tensorflow graphs and Test Tube's experimentation framework, that have delayed our ability to show results.

5 Next steps

Due to resource constraints on time remaining and dedicated persons, we do not believe it is feasible to test the performance of alternate methods like

SMOTE or ADASYN with these datasets. Both methods would require the data to be one-hot encoded first. Because many features are of such high cardinality and because we are not aware of a scalable implementation of either that does not require ingesting the entire dataset at once, we will forego this line of investigation. For the remainder of this project, we will focus on the following goals unless instructed otherwise:

- Train and test logistic regression baselines on CPC and CTR datasets with one-hot encodings
- Train and test logistic regression on CPC and CTR datasets with one-hot encodings and positive class upsampling
- Deploy test tube and run mass DAN experiments on CPC dataset with one-hot encodings
- Deploy test tube and run mass DAN experiments on CTR dataset with one-hot encodings
- Fix the DAN model so that saving and loading behaves correctly when using the DataGenerator class. This will involve removing the internal Tensorflow session of the DataGenerator so that only a single session is ever called within a process.