**Problem Statement**

This project was motivated by the observation that evaluating a classification model on an unbalanced dataset using accuracy often gives misleading results. In many cases this problem is addressed by using non-accuracy metrics to evaluate the models classification of imbalanced datasets such as AUC-ROC, F1, and balanced accuracy. Rather than grade the model based on how many total samples were correctly classified, these metrics grade the model by measuring things such as its ability to separate binary classes, simultaneously minimize false positives and false negatives, and maximize true positive and true negative rates. All of these metrics come with their own blind spots, but in the case of using accuracy to grade a model's ability to classify an imbalanced dataset it's especially easy to see where this measurement might fall short. For example, if you have a dataset where 90% of the samples belong to one class and 10% belong to another, then achieving 90% accuracy is no better than always guessing that a sample belongs to the first class.

Yet, there are many classification tasks that deal with highly imbalanced data for which accuracy should be an important metric to evaluate a model's performance. These problems are typically categorized by the fact that while the classes being predicted are imbalanced, no asymmetric risk is introduced for misclassifying any particular class. For example, maybe you are training a neural network to classify whether clothing items match an online shopping search. There might be many more shirts available for purchase on your website than belts, but the ultimate goal of developing a robust classification tool is to show consumers appropriate clothing in order to increase the number of purchases through your website. In this sense, accuracy is an important metric to use when considering model performance because what truly matters is the number of relevant items that you can show the user (it doesn't matter whether you miss more shirts or belts except for the fact that it might introduce minor annoyance to the user).

The question that followed this observation was whether training machine learning models using surrogate loss functions for accuracy absolutely results in the best testing accuracy. One could imagine that training a binary classification model using a loss function such as binary crossentropy, which is essentially a differentiable and continuous version of accuracy, might result in that model achieving a high testing accuracy, but might also never be able to correctly classify ambiguous samples belonging to the minority class due to the fact that it will be skewed towards predicting more samples correctly which is more likely if it guesses the majority class. To address this problem the question becomes, can you improve testing accuracy by training a model using a loss function which is more suited for imbalanced data. Surely this model would be more likely to predict the ambiguous minority sample, but will this improvement be overshadowed by a massive loss in ability to predict the ambiguous majority sample?

The assumption going into this project is that a model trained using a loss function designed for imbalanced data classification will achieve worse immediate testing accuracy than one trained using binary classification. Therefore, the question that this project seeks to answer is: does training a CNN to classify imbalanced data using a loss function that is designed for imbalanced data class separation achieve better class separability than one trained using binary

crossentropy, and can that class separability translate into better classification accuracy through post-hoc decision threshold tuning? Additionally, does this result change depending on the loss function used, the class distribution, whether it's a binary or multiclass problem, and the method used to tune the decision threshold?

**Data Sources and Technology Used**

The Fashion-MNIST dataset was selected for this experiment for the following reasons:

- The dataset contains 60,000 training samples and 10,000 testing samples evenly distributed among 10 classes. This allowed me to take various distributions of data from various classes and have confidence in my models ability to learn patterns while not needing excessive training time.
- The multiclass nature of this dataset gave me control over whether I wanted my model to perform binary or multiclass classification. I was able to leverage this to run my experiment in a binary class setting, a three-class setting, and a 10-class setting.
- The widespread use of the Fashion-MNIST dataset as a benchmark for various ML models gives the dataset and my results credibility.

The following technologies were used:

- The experiment was conducted using Jupyter Notebook on a Virtual Machine issued by the Texas Advanced Computing Center (TACC)
- Code was written in Python
- Tensorflow/Keras was used for model design, data preparation, loss function design, training, and testing
- NumPy was used for selecting the desired classes out of the full fashion-MNIST dataset and for resampling from classes to obtain desired class balances
- Scikit-learn accuracy score, decision tree classifier, and train test split were used for model evaluation, decision threshold tuning, and creating validation sets
- Binary crossentropy, categorical crossentropy, pairwise/margin-based hinge loss, soft-F1 loss, and soft-balanced accuracy loss, were used as loss functions

**Methods Employed**

The following methods were used to conduct the experiment.

*Data Preparation*

The first task that I faced was to prepare the dataset in the desired way. This consisted of selecting the desired classes, creating the desired class balances, normalizing the data, and preparing the data for the neural network.

To get a robust understanding of the influence that the number of classes has on the classification problem, I chose to run the experiment in a binary classification setting, a three-class classification setting, and a 10-class classification setting. For the binary problem, I

simply kept tops and trousers categories of the fashion MNIST dataset and dropped the rest of the categories. For the three-class classification setting, I included the pullovers class in the dataset and again dropped the rest of the categories. For the 10-class setting, I simply kept all classes originally in the dataset.

To experiment with the way that class distributions impacted results, I chose to run the binary and three-class problems using multiple different class distributions. For the binary classification problem I used 90/10, 80/20, 70/30, and 60/40 splits. For the three-class classification problem I used 90/7/3, 80/15/5, 70/20/10, and 60/30/10 splits. I felt that these selections were justified because they included a wide range of class imbalances which gave me a better understanding of the differences in results when the data was highly imbalanced versus moderately imbalanced. For the 10-class classification problem, I simply kept 90% of class 0, 80% of class 1, 70% of class 2, 60% of class 3, 50% of class 4, 40% of class 5, 30% of class 6, 20% of class 7, 10% of class 8, and 5% of class 9. No other distribution splits were used for the 10-class problem due to the computational expense of testing many decision thresholds across 10 classes.

For each split, the training dataset was divided into training and validation sets using an 80/20 split. The validation set was essential for the success of this project as it was used for decision threshold tuning to ensure no overfitting or data leakage. Once each of these datasets were constructed, the data was normalized and the data labels were converted into categorical structures to prepare the data to be fed into the CNN.

*CNN Architecture*

The next task was to design a CNN to train to classify the data. Due to the focus of this project on comparing the results of models trained using various loss functions, and not trying to build the best classifier of the fashion-MNIST dataset, I chose to use a relatively small model which would train and run inference faster. I was rewarded for this as I ended up training 76 different versions of my CNN.

My architecture consisted of 3 convolutional layers followed by relu cavitation functions and 2x2 maxpooling layers. The first convolutional layer consisted of 64 3x3 filters and the second and third layers consisted of 32 3x3 filters. The convolutional layers were followed by a flattening layer, a dense layer consisting of 100 neurons, and an output layer. The output layer used a sigmoid activation function in the binary setting and a softmax activation function in the multiclass setting.

*Loss Functions*

After my architecture had been designed, I developed custom loss functions to train my CNN on. These loss functions were designed to be surrogates of the AUC-ROC, F1, and balanced accuracy metrics. They consisted of a pairwise-hinge loss function (AUC-ROC surrogate), a soft-F1 loss function (F1 surrogate), and a soft-balanced accuracy loss function

(balanced accuracy surrogate). Each of these loss functions were later adapted to suit the multiclass problem.

A pairwise hinge loss was well suited for this experiment because it emphasizes class separation by penalizing the model for predicting the outputting similar probabilities for combinations of samples which belong to different classes. A soft-F1 loss was suited for this experiment because it penalizes majority class overconfidence. Finally a soft balanced accuracy loss was suited for this problem because it penalizes the model for falsely classifying samples relative to the rate that their class appears in the dataset.

*Pipeline, Training, and Inference*

The next step was to build a pipeline and a search space to automate training using combinations of loss functions, data distributions, and threshold tuning methods in the case of multiclass classification. Functions such as gridsearchCV, or scikit-learn Pipeline could have been imported to help with this, but I chose to implement these methods myself for fun and for the sake of being able to customize the data which I collected and stored. These implementations were relatively straightforward and just consisted of iterating over python dictionaries containing the parameters that I wanted to use for training, then training the model, tuning the decision threshold, and appending the results to a results list. For each model that was trained, I stored the class distribution that it was trained on, the loss function that was used in training, the decision threshold that was selected, the test accuracy at the selected threshold, and the test accuracy at a threshold of 0.5 (the default threshold used by model.evaluate).

*Threshold Tuning*

In the binary classification problem, tuning the threshold simply involved searching over 500 numbers between 0 and 1 and choosing the decision threshold which optimized classification accuracy on the validation set. When I got to the multiclass classification problem, decision threshold tuning became non-trivial. The complexity introduced here is characterized by the fact that in the multiclass setting, you aren't guaranteed to have just one output probability that exceeds its class-specific decision threshold. To address this problem, I tried three different threshold tuning methods.

The first method was simply a decision tree trained on the CNNs output probabilities. By training a decision tree on the output probabilities, I hoped to develop a heuristic which could be used to successively evaluate class probabilities in some meaningful order which improved classification accuracy. For example, a decision could have looked like:

1. Check if the probability that the sample belongs to class A > threshold A. If yes then its class A, if no then continue.
2. Check if the probability that the sample belongs to class B > threshold B. If yes then its class B, if no then continue.
3. And so on

The second and third methods searched over all possible sets of valid thresholds for the classification problem. A valid set in the three-class problem might have looked like {0.25, 0.25, 0.5}, for instance. Note that in this case, you could get output probabilities {0.4, 0, 0.7}, in which case the tuning method would not be able to clearly choose whether to classify the sample as belonging to class A or to class C. The second method resolved this issue by subtracting class thresholds from output probabilities and classified them based on the largest distance to the threshold. This method will be referred to as absolute threshold tuning. The third method resolved this issue by dividing the output probability by the threshold and classifying samples based on the largest relative distance to the threshold. This method will be referred to as relative threshold tuning. In the case described above, the second method would classify the sample as belonging to class C, while the third method would classify the sample as belonging to class A.

In the 10-class classification problem searching over all possible combinations of decision thresholds for each of the 10 classes was too computationally expensive and np.random was used to randomly select 2000 threshold combinations to search over.

**Results**

Results for binary classification:

| Split | Loss Function | Best Threshold | Tuned Accuracy | Default (0.5) |
|---|---|---|---|---|
| 90 / 10 | Binary CE | 0.036 | **0.999** | 0.998 |
| | Pairwise Hinge (AUC) | 0.044 | 0.995 | **0.998** |
| | Balanced Accuracy | 0.200 | 0.996 | **0.997** |
| | F1 | 0.269 | **0.995** | 0.993 |
| 80 / 20 | Binary CE | 0.509 | **0.996** | 0.996 |
| | Pairwise Hinge (AUC) | 0.968 | **0.993** | 0.991 |
| | Balanced Accuracy | 0.531 | **0.993** | 0.993 |
| | F1 | 0.691 | **0.994** | 0.994 |
| 70 / 30 | Binary CE | 0.192 | **0.995** | 0.994 |
| | Pairwise Hinge (AUC) | 0.721 | **0.991** | 0.991 |
| | Balanced Accuracy | 0.729 | **0.995** | 0.994 |
| | F1 | 0.008 | **0.990** | 0.989 |

| | | | | |
|---|---|---|---|---|
| 60 / 40 | Binary CE | 0.828 | **0.996** | 0.994 |
| | Pairwise Hinge (AUC) | 0.930 | **0.995** | 0.994 |
| | Balanced Accuracy | 0.924 | 0.992 | **0.995** |
| | F1 | 0.928 | **0.995** | 0.994 |

Results for three-class classification:

| **Split** | **Loss Function** | **Best Tuning Method** | **Tuned Acc.** | **Default Softmax** |
|---|---|---|---|---|
| 90 / 7 / 3 | Categorical CE | Relative Threshold | **0.994** | 0.994 |
| | Pairwise Hinge | Relative Threshold | **0.992** | 0.992 |
| | Balanced Accuracy | Relative Threshold | **0.967** | 0.955 |
| | F1 | Decision / Threshold | **0.967** | 0.967 |
| 80 / 15 / 5 | Categorical CE | Absolute Threshold | 0.984 | **0.988** |
| | Pairwise Hinge | Relative Threshold | 0.988 | **0.989** |
| | Balanced Accuracy | Relative Threshold | **0.984** | 0.981 |
| | F1 | Relative Threshold | **0.945** | 0.944 |
| 70 / 20 / 10 | Categorical CE | Absolute Threshold | 0.980 | **0.982** |
| | Pairwise Hinge | Relative Threshold | **0.986** | 0.984 |
| | Balanced Accuracy | Relative Threshold | **0.972** | 0.968 |
| | F1 | Relative Threshold | **0.981** | 0.980 |
| 60 / 30 / 10 | Categorical CE | Absolute Threshold | 0.977 | **0.981** |
| | Pairwise Hinge | Relative Threshold | 0.980 | **0.982** |
| | Balanced Accuracy | Relative Threshold | **0.974** | 0.973 |
| | F1 | Relative Threshold | **0.976** | 0.977 |

Results for 10-class classification:

| Loss Function | Best Tuning Method | Tuned Accuracy | Default Softmax |
|---|---|---|---|
| Categorical Cross-Entropy | Relative Threshold | **0.890** | 0.887 |
| Pairwise Hinge (AUC) | Absolute Threshold | **0.895** | 0.880 |
| Balanced Accuracy Loss | Absolute Threshold | **0.821** | 0.812 |
| F1 Loss | Relative Threshold | **0.852** | 0.849 |

For full per-tuning method results in the multiclass setting, see FinalProject (3).ipynb: (https://github.com/kwatts12/COE-379L/blob/main/Project3/FinalProject%20(3).ipynb)

In the binary classification problem, binary crossentropy achieved better results than any other loss function across all class distributions. Interestingly, selected decision thresholds tended to gravitate towards near 0 and near 1 values. Additionally, even though results achieved through threshold tuning ubiquitously dominated results achieved without threshold tuning, the classification accuracy achieved with and without the use of threshold tuning were marginal. These observations demonstrate that models trained on the binary classification task were able to predict sample classifications with a very high degree of confidence (few output probabilities fell between 0.5 and the threshold).

In the three-class classification problem, categorical crossentropy outperformed other loss functions only in the case of the 90/7/3 class split. On all other splits, the pairwise hinge loss function performed best. Also of note is the fact that models trained using non-categorical crossentropy loss function almost unanimously selected the relative threshold tuning method as the best threshold tuning method. Meanwhile, models trained using categorical crossentropy typically selected the absolute threshold tuning method as the best threshold tuning method. This behavior is consistent with the objectives of the respective loss functions. Training with categorical cross-entropy encourages models to classify samples as absolutely belonging to a class or not. In contrast, loss functions designed for imbalanced classification emphasize relative class separability, which defines class membership in terms of how strongly a sample aligns with one class compared to others.

Finally in the 10-class classification problem, Pairwise Hinge loss outperformed the other loss functions, and selected the absolute threshold tuning method as its best tuning method. The difference between accuracy using default softmax outputs and accuracy using the tuned decision threshold is the most apparent in this setting, as the tuned thresholds featured significant improvements over the default predictions in every case. This is best interpreted when one considers the fact that as the number of classes increases, neural networks have more possible

points of failure when calibrating output probabilities for each class. Each calibration mistake that a neural network makes compounds on any previous mistakes which can ultimately have a huge effect on the testing accuracy of the network. Decision threshold tuning has a greater effect in this setting because decision regions exist in higher dimensions and therefore feature more degrees of freedom to be tuned.

In future works, I would further investigate the multiclass setting. I would particularly be interested in using more loss functions to attempt to achieve higher classification accuracy. It would also be interesting to investigate the effect of different class distributions in the 10-class problem and to investigate the change in results and the number of classes increases from 3 to 10.

**References**


"Convolutional Neural Networks (Cnns)." *Convolutional Neural Networks (CNNs) - COE 379L: Software Design For Responsible Intelligent Systems Documentation*, coe379l-fa25.readthedocs.io/en/latest/unit03/cnn.html. Accessed 12 Dec. 2025.

*Loss Functions of Rankers: Pairwise vs Listwise | by Jimmy Wang | Medium*, jimmy-wang-gen-ai.medium.com/loss-functions-of-rankers-pairwise-vs-listwise-c5c43d82bc20. Accessed 12 Dec. 2025.

See Use of AI doc for more