**CNN-based Detection of Defects in Computer Chips**

**Author(s)**

*Mansi Vekariya, Neel Desai, Saipraneeth Konuri, Samarth Sharma*

### **Abstract**

This project aims to develop a convolutional neural network (CNN) model to detect defects in computer chips, achieving an accuracy of at least 90%. The approach involves using a dataset with significant class imbalance, which was addressed by converting the problem to binary classification and downsampling. The model training process utilized initial experimentation on Notebooks in Azure Data Studio and further optimization and deployment on Azure ML. Key improvements included implementing regularization techniques and optimizing the training process with adaptive optimizers, resulting in a final model accuracy of 93%. MLOps components such as experiment tracking, model registry, versioning, and automated deployment were integrated to ensure robust model management and deployment.

### **Introduction**

Detecting defects in computer chips is crucial for ensuring the reliability and performance of electronic devices. This project focuses on building a CNN model capable of accurately identifying defects in chip images. Given the class imbalance in the dataset, the project shifted from multiclass classification to binary classification to improve model performance. The final model achieved a validation accuracy of 93% by implementing various regularization techniques and optimizing the training process. The project also integrates MLOps components to streamline model tracking, versioning, and deployment.

### **Related Work**

### Defect detection in computer chips has been a critical area of research due to its significant impact on the reliability and performance of electronic devices. Traditional methods for defect detection often relied on manual feature extraction followed by classical machine learning algorithms such as Support Vector Machines (SVM), Decision Trees, and Random Forests. These methods required extensive domain knowledge to design and extract features that could effectively distinguish between defective and non-defective chips.

In recent years, deep learning, and specifically Convolutional Neural Networks (CNNs), have revolutionized the field by automating the feature extraction process. CNNs are capable of learning hierarchical features directly from raw image data, which has led to significant improvements in defect detection accuracy. Several studies have demonstrated the effectiveness of CNNs in detecting various types of defects in semiconductor wafers, including surface scratches, misalignments, and other anomalies. For instance, the work by Zhang et al. (2018) utilized a deep CNN to classify defect types in semiconductor manufacturing, achieving high accuracy and demonstrating the potential of deep learning in this domain.

However, one of the common challenges in using CNNs for defect detection is the issue of class imbalance, where the majority of the data belongs to the non-defective class. This imbalance can lead to biased models that perform poorly on the minority defect classes. Previous research has addressed this issue using techniques such as oversampling the minority class, undersampling the majority class, or employing cost-sensitive learning methods.

Our approach builds on these advancements by incorporating a robust Machine Learning Operations (MLOps) pipeline to ensure efficient Experiment tracking, model registry, model versioning model deployment, and POC for the retraining pipeline. Specifically, we addressed the class imbalance problem by converting the multi-class classification task into a binary classification problem (defective vs. non-defective). Additionally, we performed downsampling to create a balanced dataset, which helps in training a more robust model.

Furthermore, we enhanced our model's performance by implementing advanced regularization techniques such as batch normalization and dropout layers, which help prevent overfitting. We also utilized adaptive optimizers like AdamW to improve the convergence rate and model accuracy. Unlike many previous works, our project integrates comprehensive MLOps practices, including experiment tracking with MLFlow, model registry, and versioning, and automated deployment using Azure ML. This ensures that our models are not only accurate but also efficiently managed and deployed in a production environment.

In summary, while previous research has laid the foundation for using machine learning and deep learning techniques in defect detection, our approach advances the field by integrating modern MLOps practices and addressing the critical issue of class imbalance with innovative techniques. This results in a more reliable and scalable defect detection system for computer chips.

**Data**

The dataset used in this project originates from the Kaggle [Link to Kaggle Dataset](https://www.kaggle.com/datasets/qingyi/wm811k-wafer-map) and is a comprehensive collection of computer chip images, each labeled with various types of defects. The dataset comprises a total of 811,457 images with dimensions of 632x632 pixels. To ensure consistency and manageability for our project, we focused on a subset of these images, specifically those resized to 26x26 pixels, resulting in a dataset of 30,078 images.

The dataset is categorized into the following nine classes based on the type of defect:

* Center
* Donut
* Edge-Loc
* Edge-Ring
* Loc
* Random
* Scratch
* Near-full
* None (no defect)

Upon examining the distribution of these labels, we discovered a significant class imbalance. The "none" class, representing images without defects, constituted over 95% of the dataset. The remaining 5% was distributed among the other eight defect classes. This imbalance posed a challenge for training an effective multi-class classification model, as the model could become biased towards the predominant "none" class.

To address this imbalance, we decided to convert the multi-class classification problem into a binary classification problem. This simplified the task to identifying whether an image represents a defective chip or a non-defective chip. This approach helped to mitigate the issues arising from the severe class imbalance.

In addition to redefining the classification problem, we performed downsampling to ensure that both classes (defective and non-defective) were evenly distributed in our training dataset. This resulted in a balanced dataset comprising 900 images for each class, totaling 1,800 images. This balance is crucial for training a robust model that performs well in both classes.

During the data preprocessing phase, we explored various techniques to enhance the dataset. We considered image augmentation strategies such as rotation, flipping, and scaling to artificially increase the diversity of our training data. However, the results from these augmentations were not satisfactory, and given our objective to run the auto-training loop on low-cost cloud resources, we opted to keep the training computation as efficient and light as possible.

Overall, the data preprocessing steps included resizing images, addressing class imbalance through binary classification and downsampling, and evaluating the impact of image augmentation. These steps were essential in preparing a clean and balanced dataset, conducive to training an effective CNN model for defect detection in computer chips.

**Methods**

1. Project Setup and Configuration:

* The project utilizes MLflow for experiment tracking and version control, with a setup for storing logs in Google Drive and configuration specified for Azure ML.
* A range of libraries and tools are integrated for different tasks, including data handling (pandas, numpy), visualization (matplotlib, seaborn), machine learning (TensorFlow, sklearn), and web applications (Flask).

2. Data Preparation and Processing:

* Data is loaded from cloud storage, with preprocessing steps handled in Python using libraries like ‘sklearn’ for standardizing and binarizing labels.
* A systematic approach to train-test splitting ensures data integrity and proper model evaluation.

3. Model Training and Validation:

* Models are trained using TensorFlow with structures like ‘CNNTrainer2’ class encapsulating the entire process.
* Use of callbacks like EarlyStopping to optimize training and prevent overfitting.

4. Automation and MLOps:

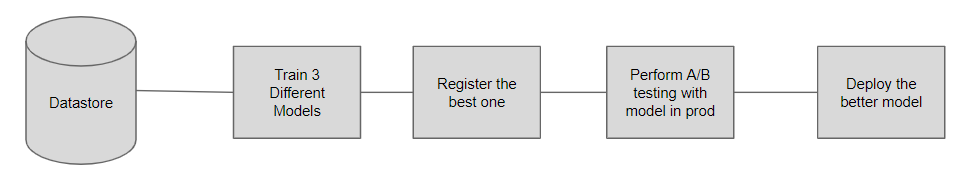
* An automated training and deployment pipeline (AutoTrainingDeploymentPipeline.py) facilitates the end-to-end process from data preprocessing to model deployment.
* Integration with MLflow and Azure ML enhances tracking, management, and deployment capabilities, ensuring systematic updates and traffic management for models in production.

5. Web Application Interface:

* A Flask application (app.py) is developed for user interaction, allowing image data submission and retrieval of model predictions.
* Configuration adjustments for development, including SSL certification handling, are implemented to ensure secure and flexible development environments.

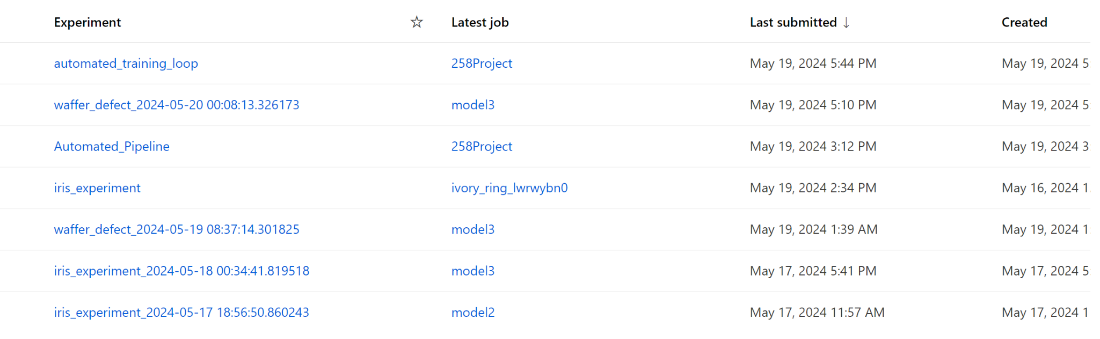
**MLOps**

For the MLops, we're assuming that the data cleaning pipeline is different, which will channel data to the data store. Thus, always only the clean and updated data will be there in the data store.

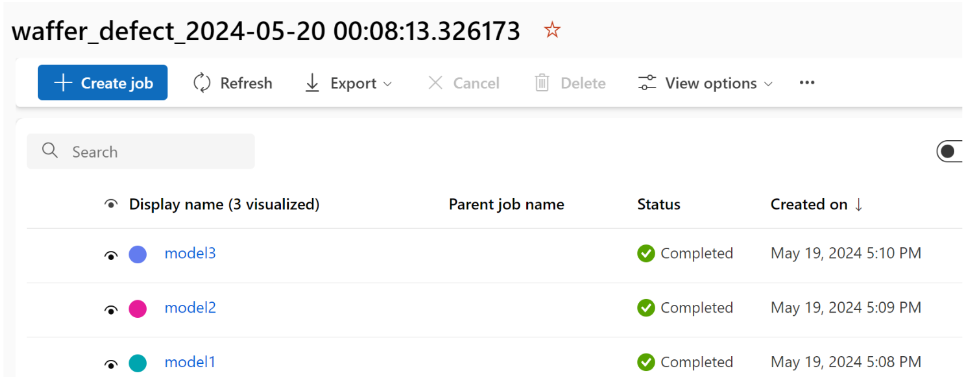


This pipeline can be scheduled to run at regular intervals or run when new data arrives. It can be configured directly as a pipeline on Azure ML.

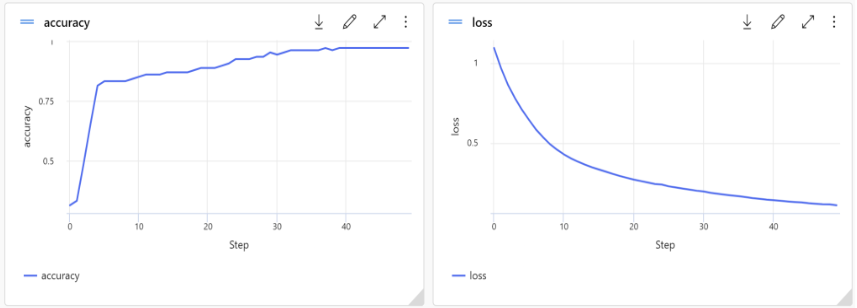
An experiment is nothing but a collection of different models you built when trying to find the best one:



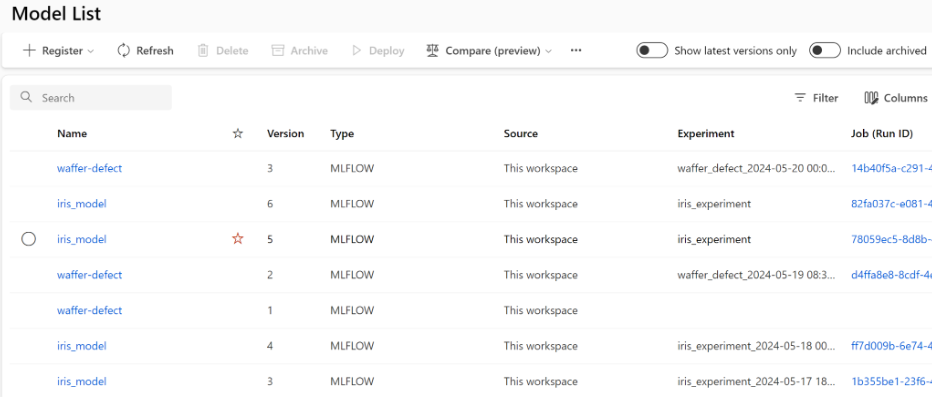
With each experiment, we have the list of all the models trained for it, we use MLFlow for logging models' performance and its artifacts



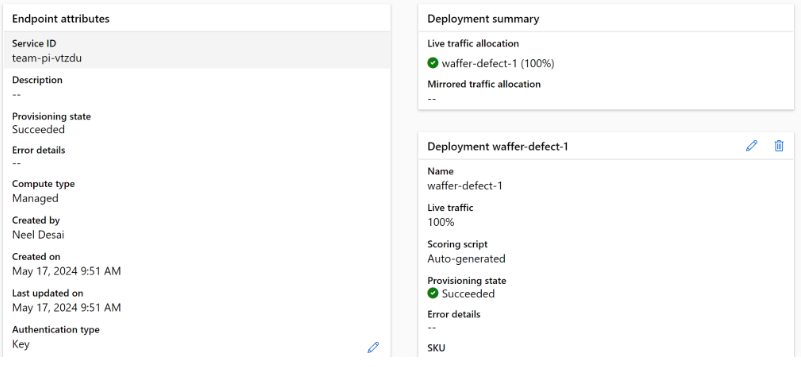
For each logged model we have all the logged artifacts and metrics, along with the hyperparameter used for training it



Model Registry is a means through which we can apply version control on our models, it could be as simple as the same model trained on data at 2 different times

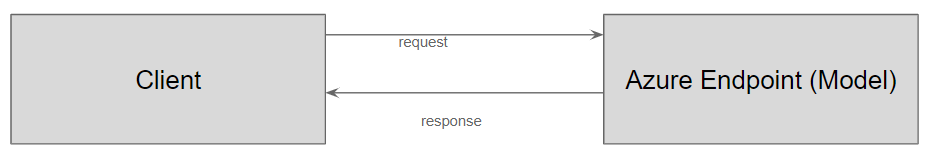


Endpoint Deployment allows us to expose our model as an API to the rest of the world.



What we do is perform an experiment whenever new data comes in or as per the schedule, and it trains 3 models. The best-performing model aka the champion model is registered to the model registry. Then getting the model in the registry we perform A/B testing. Then finally the model is ready for deployment, but if there is already one model deployed, then the deployed model and the challenger model (the champion model from the experiment) are evaluated on the new data and if the challenger wins, it is deployed or the model already in production continues to serve.

**Inference Pipeline**



As the Endpoint and model are different, even if we deploy a different model the endpoint URL remains the same so no changes there.

Based on the comprehensive details provided, here's how to address the question regarding the approach for solving the chip defect detection problem using machine learning techniques:

**Approach to Solving the Problem**

Our approach to solving the chip defect detection problem involved a systematic exploration and optimization of deep learning models to ensure robustness and accuracy in identifying defects. We initiated our project with a foundational model and progressively refined the architecture and training strategies based on empirical observations and performance metrics.

1. Initial Model Architecture:

We started with a basic neural network consisting of an input layer, three convolutional layers, a flatten layer, two fully connected layers, and an output layer, using Adam as the optimizer. This setup served as our baseline to understand the initial performance and shortcomings, such as overfitting, as evidenced by diverging training and validation loss curves.

2. Incorporation of Regularization and Architectural Adjustments:

To combat overfitting, we introduced Dropout layers and implemented early stopping. Additionally, we refined the architecture by adding pooling layers after convolutional layers to reduce dimensionality and focus on important features, enhancing model generalization.

3. Experimentation with Optimizers:

Our exploration included testing different optimizers. Transitioning from Adam to Adadelta with a lower learning rate showed that while it may stabilize training, it led to early plateauing of loss and accuracy, indicating underfitting.

4. Final Model Adjustments:

Returning to an Adam variant, specifically AdamW, and combining it with pooling and dropout layers resulted in significant improvements in both training and validation metrics. This model exhibited robust generalization capabilities and maintained consistent improvement, validating our architectural choices.

5. Further experimentation:

Attempted advancements like separable convolutions showed initial promise but led to overfitting, reinforcing our decision to finalize the model with the AdamW optimizer and additional regularization techniques.

**Justification of the Approach**

The iterative approach starting with a simple model and gradually introducing complexity allowed us to pinpoint specific factors impacting performance.

**Model Complexity and Overfitting**: Initially observed overfitting prompted the inclusion of Dropout layers and pooling, which are well-documented for their effectiveness in improving model generalization.

**Optimizer Choice**: Experimentation with different optimizers helped us find the optimal balance between speed and stability of convergence.

**Continuous Evaluation and Iteration**: Using MLflow for tracking experiments facilitated systematic evaluation and comparison, enabling informed decisions about architecture and training adjustments.

**Alternative Approaches Considered**

We considered various model architectures. However, given the complexity of image data in chip defect detection, convolutional neural networks (CNNs) were deemed most appropriate due to their prowess in feature extraction from images. Alternative regularizations and optimizers were tested to ensure robustness against overfitting and underperformance.

**Experiments**

1. Exploratory Data Analysis (EDA):

* Initial data exploration using statistical and visualization techniques to understand data characteristics and prepare for model training.
* Insights from EDA inform preprocessing and model configuration decisions.

2. Model Evaluation and Selection:

* Systematic training procedures with extensive logging in MLflow, allowing for performance tracking and comparison across different model iterations.
* Experiments are conducted under controlled conditions, utilizing techniques like traffic splitting in Azure ML to compare new models against current production models safely.

3. Deployment and Real-World Testing:

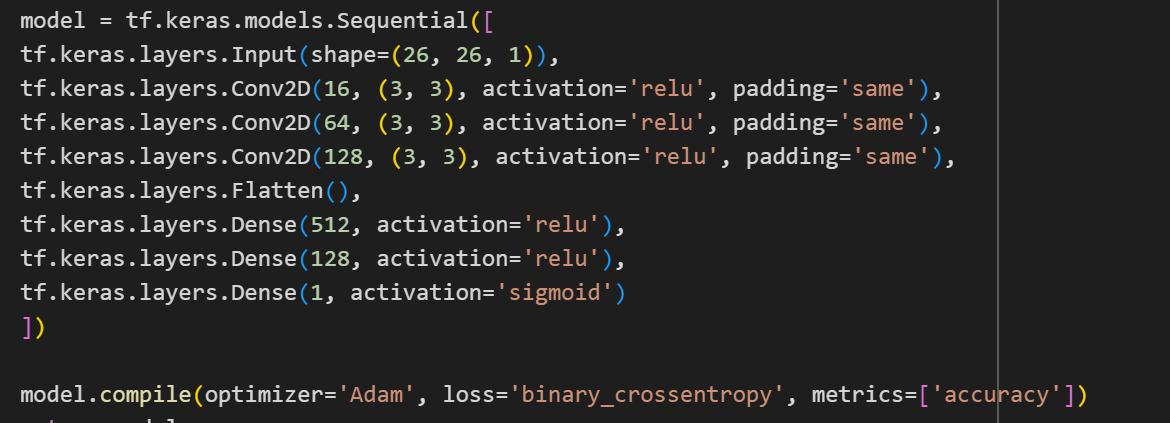
* Deployment strategies include the use of Azure ML for setting up production endpoints, with dynamic traffic management to facilitate the gradual rollout of new models.
* The Flask web application serves as a practical interface for real-world testing, enabling users to interact with the model.

**Overview of Experimental Strategy:**

Our experimental strategy was meticulously structured to rigorously test and optimize convolutional neural network (CNN) architectures for detecting defects in computer chips. The experiments were conducted in two primary environments: initial testing on Google Colab and refined auto-training loops on Azure ML, with all iterations tracked using MLFlow.

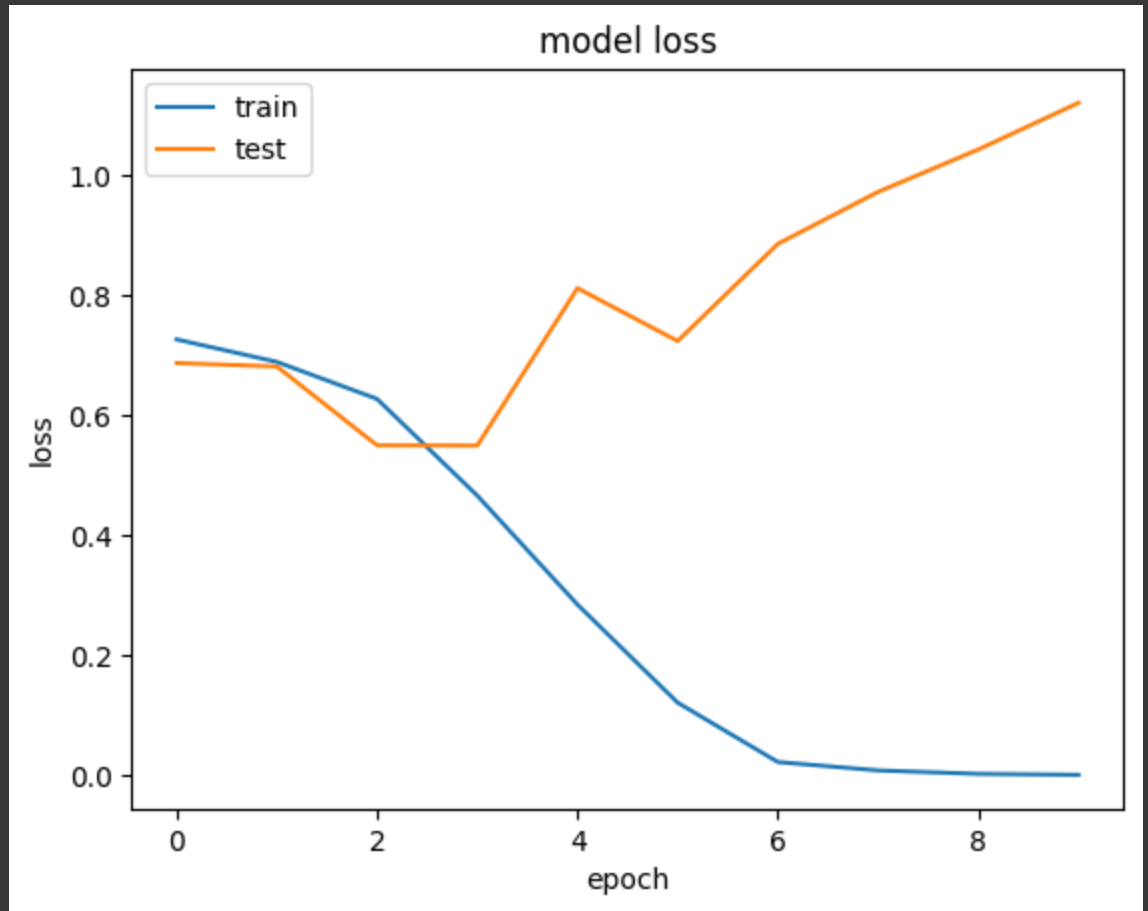
**Initial Training and Model Iterations:**

**Baseline Model:**



Architecture: Started with a simple CNN comprising an input layer, three convolutional layers, a flatten layer, two fully connected layers, and an output layer. Adam optimizer was used.

**Loss Curve:**

****

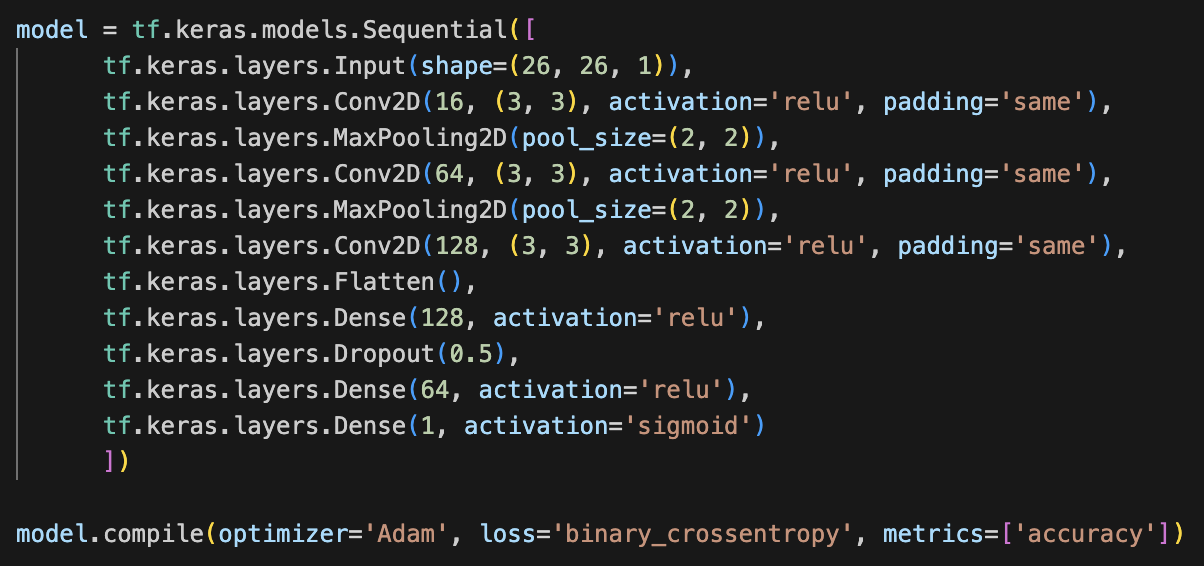
Outcome: The initial model showed promise but suffered from overfitting as indicated by divergence in training and validation losses. The validation accuracy at this stage was 75%.

**Model Enhancements:**

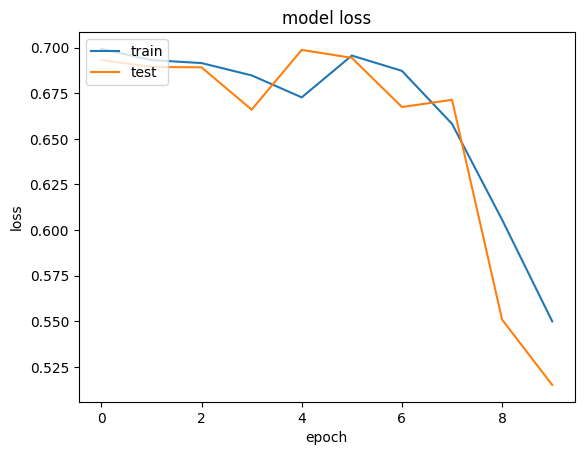
Introduction of Pooling and Dropout: To address overfitting, we integrated pooling layers to reduce feature dimensionality and dropout layers to randomly deactivate neurons during training, thereby promoting generalization.

Optimizer Tweaks: Transitioned from Adam to Adadelta, which unfortunately resulted in early plateauing of loss and accuracy metrics. This led to a revert to a more sophisticated optimizer.

**Model 2:**

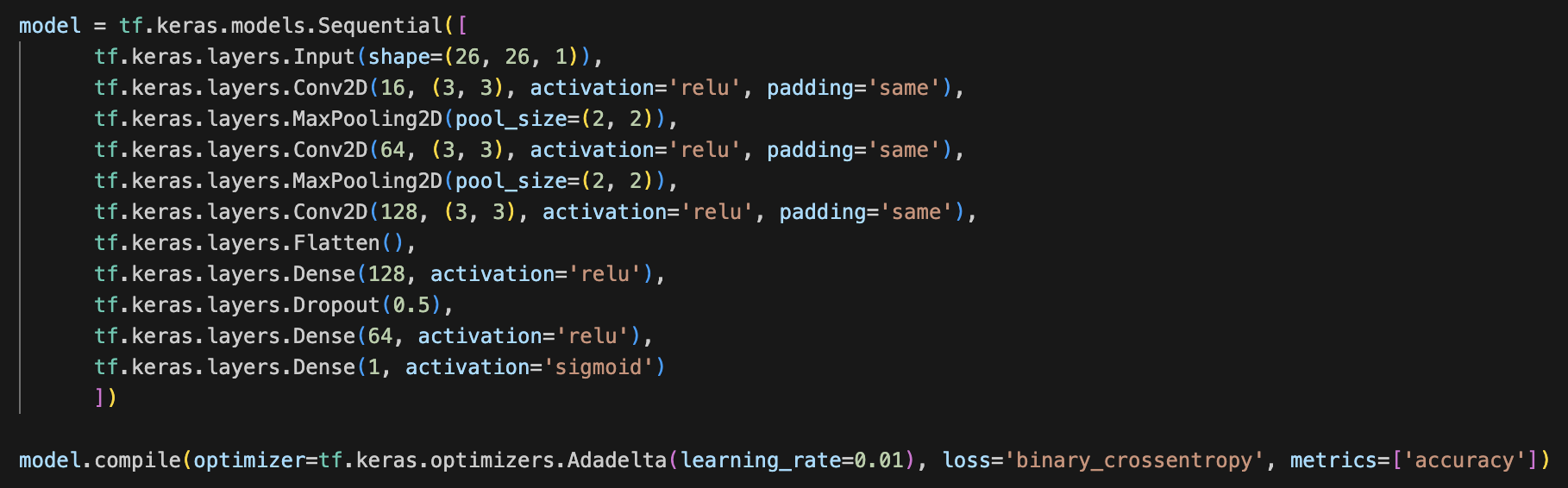
****

**Loss Curve:**

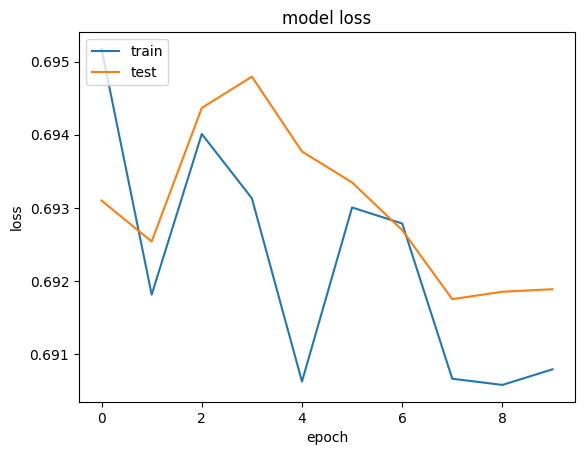
****

After adding the dropout layer and pooling layers, we observed an improvement where the initial training was slower, but eventually showed better generalization with reduced overfitting compared to the initial model. and also the loss and accuracy showed more consistent improvement.

**Model 3:**

****

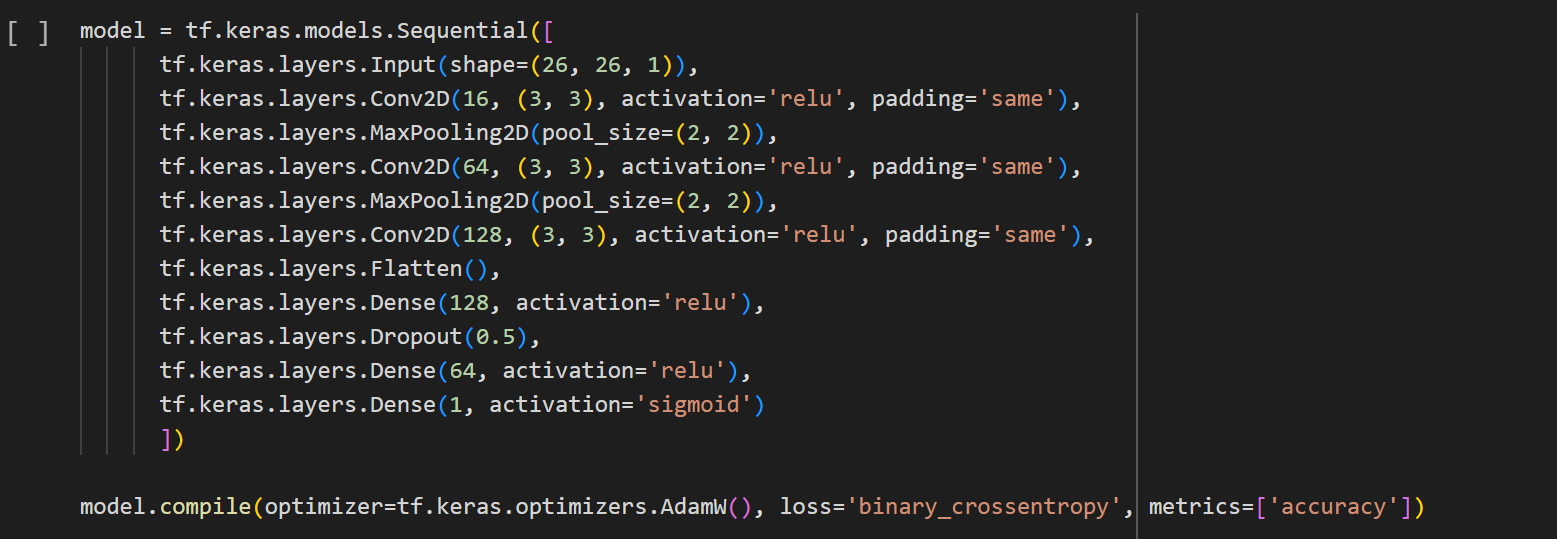
**Loss Curve:**

****

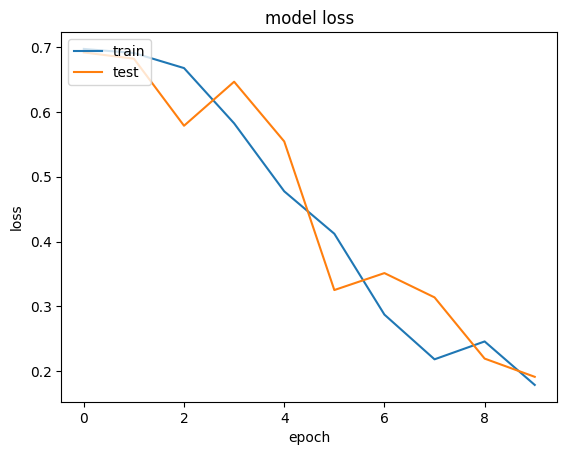
We tried changing the optimizer from Adam to Adadelta with a learning rate of 0.01. It struggled with accuracy, and both training and validation loss plateaued early.

**Final Model Configuration:**

Advanced Optimizer and Regularization: The final model utilized AdamW, known for its weight decay capability that helps in avoiding overfitting. We also introduced batch normalization to ensure inputs to layers are normalized, further aiding in stable and faster training.

****

**Loss Curve:**



Changed the optimizer to AdamW and it has pooling layers, dropout layers and it Demonstrated significant improvement in both training and validation metrics and also Better generalization with consistent improvement in val\_loss and val\_accuracy.

Outcome: This configuration significantly improved model performance, pushing the validation accuracy to 93% and demonstrating excellent generalization as seen in the consistent decrease in validation loss.

**Model Selection and Performance Validation**

Selection Criteria: Models were selected based on their performance metrics, particularly focusing on validation loss and accuracy to ensure they generalize well to unseen data.

Performance Tracking: MLFlow played a crucial role in logging all experiments, enabling easy comparison of model versions and configurations. This facilitated the selection of the best-performing model for deployment.

**Deployment and Real-World Testing**

Automated Deployment on Azure ML: The chosen model was deployed using Azure ML, allowing automated retraining and deployment cycles, crucial for maintaining model efficacy over time.

Inference and Application: Deployed models were made accessible via a web interface, enabling real-world testing and user interaction. This setup not only validated the model's practical utility but also allowed for continuous feedback which could be leveraged for further improvements.

**Model Summary:**

Here’s a summarized model comparison in tabular form:

| **Model** | **Final Training Accuracy** | **Final Validation Accuracy** | **Lowest Training Loss** | **Lowest Validation Loss** |
| --- | --- | --- | --- | --- |
| 1 | 100.0% | 74.72% | 0.0006 | 0.5500 |
| 2 | 71.92% | 72.19% | 0.5499 | 0.5151 |
| 3 | 52.22% | 57.30% | 0.6906 | 0.6917 |
| 4 | 93.81% | 92.70% | 0.1790 | 0.1915 |

This table highlights the final training and validation accuracies, and the lowest training and validation losses recorded for each model at the end of the 10 epochs. **Model 4** shows the best balance with high accuracies and the lowest losses, indicating superior generalization and learning capability among the tested models.

**Conclusion**

In this project, we tackle a significant challenge in the electronics industry: detecting defects in computer chips using a convolutional neural network (CNN).

Our aim was to develop a CNN model that not only performs well but also addresses the common issues found in similar tasks, such as class imbalance in the data. We started with a basic CNN model and systematically enhanced it by experimenting with different architectural changes and optimization techniques.

The initial phase of our project involved significant experimentation with model architectures and training strategies on Google Colab and this phase was crucial for understanding the behavior of different models under various configurations.

Recognizing the challenges posed by class imbalance, we simplified our task from a multi-class to a binary classification problem. This strategic shift, along with our decision to downsample the data, helped us manage the imbalance effectively, allowing our models to learn to distinguish between defective and non-defective chips more accurately.

As we refined our models, we incorporated advanced techniques like dropout layers and pooling to prevent overfitting and improve the model's ability to generalize. This means the model can perform well not just on the data it was trained on but also on new, unseen data. We also explored different optimizers, eventually settling on AdamW for its robust performance in handling complex learning tasks.

We implemented MLOps practices, including MLFlow for tracking experiments and Azure ML for deployment. These tools not only helped us manage the lifecycle of our models efficiently but also ensured that we could deploy the best-performing models quickly and reliably.

Our final model achieved 93% accuracy on validation data and this high level of accuracy is crucial for practical applications, where the cost of missing a defect can be very high.