Neural Wave Hackaton Report: Duferco Project

Team RadYOmki

Mikhail Andronov*
mihandronov@gmail.com

Aleksandr Volkov
stareist@gmail.com

Roman Rakhlin
rrakhlin@gmail.com

Daniil Terekhin
terekhin.daniil.a@gmail.com

October 2024

1 Objective

At a steel mill, one of the tasks is to cut large steel bars into shorter pieces. Sometimes, multiple steel bars must be cut to the same length, and for that, the bars are rolled along a belt until they all hit a stopper. However, the bars sometimes get unaligned and do not hit the wall together. Such situations require constant attention from the worker, who must observe the belt on the camera in order not to press the "cut" button when the bars are unaligned. Therefore, we want to help the worker by building a neural model that can detect the alignment of steel bars with a stopper wall using image data.

2 Approach

We received image data from Duferco organized into a "training set" and an "example set". The data consists of individual frames from a camera that monitors the conveyor belt with steel bars. Sometimes, the bars in the image are aligned with the stopper, sometimes not. However, there was a problem with the data: only the example set contained ground truth labels, and the entire training set contained no labels (aligned/unaligned). The size of the example set contains only around 300 images, and the training set contains around 15000 images. We decided on the supervised binary classification approach, so we needed more labeled data for training image classification models.

*All authors contributed equally.

2.1 Our Data Labeling Application

First, we developed an interactive image classification application, that works like Tinder, designed to streamline the process of categorizing large datasets of images. The tool allows users to efficiently classify images using keyboard controls, provides visual feedback, saves classification progress, supports undo functionality, and can handle specific index ranges of images. Additionally, it replaces local file paths with server-specific prefixes in classification records, ensuring compatibility with remote storage systems. Using our application, we quite fast manually labeled more than 7000 images from the *training_set*.

2.1.1 Features

Image Display and Classification

The program displays images from a specified folder, resizing them to fit within the application window. Users can classify each image into one of two categories by pressing the left or right arrow keys. The classifications are recorded in separate text files for each category.

Keyboard Controls

- **Right Arrow Key**: Classify the current image into `class_aligned.txt`

Figure 1: Labeling application screenshot

- **Left Arrow Key**: Classify the current image into `class_not_aligned.txt`
- **'u' Key**: Undo the last classification, removing it from the classification file and returning to the previous image.
- **Escape Key**: Exit the application.

Progress Tracking

The tool reads existing classification files upon startup to determine which images have already been classified. It skips these images, ensuring that users do not have to reclassify images during subsequent sessions. This feature allows users to interrupt and resume the classification process at their convenience.

User-friendly Emoji Feedback

After each classification, the program displays a large emoji (✅, ❌, 🔙) over the image to provide immediate visual feedback. This helps users confirm that their input was registered correctly before the tool proceeds to the next image.

Index Range Filtering Users can specify a range of image indices to classify. This is particularly useful when dealing with large datasets, as it allows users to focus on a subset of images. The program filters the images based on their numerical indices extracted from filenames (e.g., `img_00001.jpg`).

2.2 Data Preprocessing

After gathering the first thousand of labeled images, we started the first experiments. We set up a training procedure in Pytorch Lightning to fine-tune a pre-trained ResNet50 [1] for binary classification on our images as they are. The model showed no signs of convergence. We concluded that the problem must have been to hard for the model because it gets lost in irrelevant image details. Therefore, we decided to preprocess the images to keep only the information relevant to the prediction.

2.2.1 Image Preprocessing

Our final image preprocessing pipeline consists of the following steps:

- **Cropping and Resizing** Each image is cropped to make it square, based on the shorter dimension (height), by trimming the excess width from the left side (opposite to the stopper wall). The resulting image is resized to a specified target size, denoted by `target_size`, as different neural models require specific input image dimensions.
- **Grayscale Conversion** We convert the images to grayscale without losing any information relevant to further prediction.
- **Edge Enhancement for Dark Images** We emphasize edges by adjusting brightness and contrast depending on the overall lightness or darkness of the image. If the grayscale brightness mean is below a threshold of 55, indicating a dark image, brightness and contrast are adjusted with values of 155 and 1.5, respectively; otherwise, they are set to 75 and 1.35. This gives us an almost entirely dark image with highlighted edges of the bars and the stopper wall.
- **Gaussian Blur and Edge Blending:** To enhance the edges further, a Gaussian blur with a kernel size of 31 is applied to a copy of the adjusted grayscale image. The blurred layer is subtracted from the grayscale image, resulting in an edge-enhanced version.

- **Masking The Left Half** Finally, we set all image values in the left half of the image to zero. In all images, the bars are rolling to the right where the wall is, so the parts of the image far from the stopper are irrelevant to the prediction.

The output of this pipeline is a `numpy` array representing the preprocessed image, which is subsequently converted into a tensor format.

2.2.2 Data augmentation and splitting

In addition to the preprocessing, we also applied data augmentations, and we were careful about their choice. We noticed that different images may have different camera angles or lighting conditions, and augmented the images with random small-angle rotations, random brightness adjustments, and random Gaussian blur just in case. The LightningDataModule class in PyTorch Lightning handles the loading of images from separate directories of positive (aligned) and negative (not aligned) samples, preprocessing, and data augmentations. Figure 2a shows an image from the dataset before preprocessing, and 2b shows the same image after preprocessing and augmentations.

We decided to use the images we labeled ourselves as the training set and the example set labeled by the data provider as the validation set.

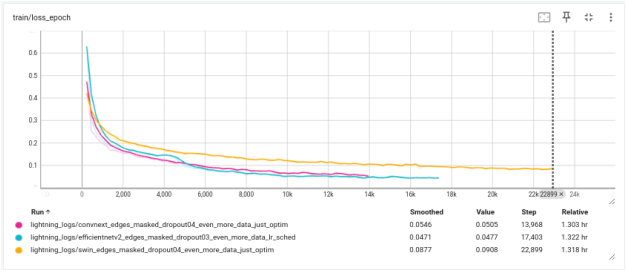
2.3 Modeling

We tried several deep architectures for the binary classification of out images: ResNet50 [1], EfficientNetV2s [4], ConvNeXt [3], and Swin Transformer [2], all pretrained on ImageNet. In all models, we removed the last layer and replaced it with a linear head that predicts a scalar value for binary classification. This is a necessary step, because originally all those models were pre-trained for multiclass classification. At first, we tried fixing the weights ("freezing") in all layers in a model but the last one before training it. However, the models did not converge. Then we allowed full pretraining (the learning of all layers), and the models finally started converging in a meaningful way!

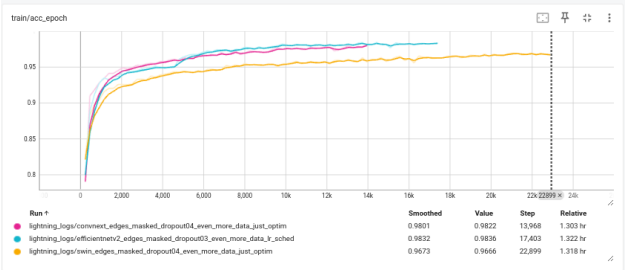
3 Results

3.0.1 Model training

The three models we tried showed reasonable training dynamics. Figure 3a shows the training curves with the loss steadily decreasing, and Figure 3b shows the steadily increasing accuracy on the training set. Table 1 shows the performance of the best checkpoints of the models on the validation set, which in our case consisted of images labeled by Duferco. The F score our models demonstrate exceeds 82%, which we find encouraging. We introduced dropout (30-40%) in the final model layers and used the AdamW optimizer with weight decay to reduce overfitting.



(a) Training Loss curves (BCEWithLogitsLoss)



(b) Training Metric curves

Figure 3: Training curves of different models: EfficientNetV2, Swin Transformer, and ConvNeXt.

3.1 Desktop application

We built a desktop application to test how our models work with a video stream "in production". Our application is written in Swift and powered by our best EfficientNetV2 checkpoint, which we converted to the

Table 1: The performance of several models on the validation dataset. EfficientNetV2, ConvNeXt, and Swin Transformer show similar performance, reaching over 82% F score in their best checkpoints on the example set provided by Duferco with labels. The corresponding accuracies are around 96%. The models are relatively small, demanding a couple of hundreds of megabytes.

Model	**Best F Score, %**	**Accuracy, %**	**Model size (MB)**	**Image side (px)**
EfficientNetV2	82.3	96.3	232	384
ConvNeXt	83.4	95.9	319	384
Swin Transformer	84.5	96.3	315	224

CoreML format native to MacOS. The model in this format weighs only 80 Mb of memory. The application displays a video stream with a frame around it, which glows red if the current frame is "not aligned", and green if it is "aligned".

4 Challenges

The biggest challenge in the discussed problem is, of course, the lack of labeled data for training the image classification models. We addressed this problem by making our own labeling application discussed in section 2.1. We still could not label all the data provided to us and trained our final models on around 7000 images out of around 15000. We noticed that the same models become better on validation when the training set grows, so the performance we report could be even higher going forward. Regarding preprocessing, we did not arrive at the final procedure until we tried training models on plain images and grayscale images, which did not work. When we decided on the final models, we struggled to find a good learning rate schedule and dropout rate, eventually settling for the constant standard learning rate for Adam (1e-4) and a dropout rate of 30%. We really enjoyed solving the problem and we conclude that binary classification of images combined with a clever preprocessing and full fine-tuning of state-of-the-art pre-trained image classifiers is a good approach for this problem.

References

[1] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)

[2] Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B.: Swin transformer: Hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF international conference on computer vision. pp. 10012–10022 (2021)

[3] Liu, Z., Mao, H., Wu, C.Y., Feichtenhofer, C., Darrell, T., Xie, S.: A convnet for the 2020s. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 11976–11986 (2022)

[4] Tan, M., Le, Q.: Efficientnetv2: Smaller models and faster training. In: International conference on machine learning. pp. 10096–10106. PMLR (2021)

(a) Image before preprocessing

(b) Image after preprocessing

Figure 2: Example of image preprocessing