FROM HANDWRITTEN MATH TO LATEX: A DEEP LEARNING APPROACH WITH ERROR CORRECTION

Anthony Brogni
Department of Computer Science & Engineering
University of Minnesota - Twin Cities
Minneapolis, MN, USA
brogn002@umn.edu

Evan Pochtar
Department of Computer Science & Engineering
University of Minnesota - Twin Cities
Minneapolis, MN, USA
pocht004@umn.edu

Victor Hofstetter
Department of Computer Science & Engineering
University of Minnesota - Twin Cities
Minneapolis, MN, USA
hofst127@umn.edu

Ajitesh Parthasarathy
Department of Computer Science & Engineering
University of Minnesota - Twin Cities
Minneapolis, MN, USA
parth057@umn.edu

ABSTRACT

This paper presents a novel hybrid approach for converting handwritten mathematical expressions into syntactically correct LaTeX code. We combine an image-to-LaTeX model with a language-model-based postprocessor to address the challenges of accurately recognizing complex mathematical notation. Our system comprises a ResNet-34 based CNN encoder for feature extraction from handwritten images, a Transformer decoder for LaTeX sequence generation, and a fine-tuned Phi-4-mini language model for syntax correction. Trained on the MathWriting dataset with data augmentation techniques, our model achieves 85.59% accuracy with beam search decoding alone, increasing to 86.22% with LLM post-processing. This work provides a foundation for future improvements in mathematical content recognition systems for educational and research applications.

1 INTRODUCTION

This project aims to address the problem of converting handwritten mathematical expressions into accurate LaTeX code, a crucial task for digitizing mathematical content. The primary objective is to design a system that can efficiently recognize handwritten math expressions and generate LaTeX code while ensuring minimal syntax errors. Our project combines computer vision, sequence modeling, and error correction techniques to improve accuracy beyond existing methods.

2 MOTIVATION

Converting handwritten mathematical expressions into machine-readable formats, such as LaTeX, presents challenges due to the complexity of symbols, multi-line expressions, and varying handwriting styles. While several models exist for handwriting recognition, they often generate incorrect or incomplete LaTeX syntax. This problem is interesting because solving it would greatly benefit fields that require digitizing mathematical notes, such as education, scientific research, and online learning platforms.

3 RELATED WORK

There is significant research on handwritten math expression recognition, particularly using convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Papers such as IM2LaTeX Kanervisto (2016) and others have achieved reasonable accuracy in handwritten-to-LaTeX conversion. Recent advances like transformers and attention mechanisms have also been used to improve

sequence generation tasks. However, these models often suffer from LaTeX syntax errors in their output, which require manual post-processing.

One promising direction is MathWriting Gervais et al. (2024), a dataset for handwritten mathematical expression recognition, which provides a larger and more diverse set of handwritten math expressions than previously available datasets. We plan to leverage this comprehensive dataset for our project.

4 Proposed Approach

In our proposal, we proposed a hybrid approach to this problem involving two major components:

1. Handwritten Image to LaTeX Conversion: A convolutional neural network (CNN) for image feature extraction, followed by a transformer-based sequence generation model for LaTeX output. We will incorporate attention mechanisms to capture spatial relationships between symbols better.
2. LaTeX Error Correction using a Fine-Tuned Language Model: We will employ a fine-tuned large language model (LLM) to post-process the generated LaTeX and correct any syntax or semantic errors. The model will refine the output iteratively, ensuring valid LaTeX code that adheres to grammar and syntax rules.

5 Implementation

5.1 Data Acquisition and Preprocessing

Initial exploration involved accessing the synthetic portion of the MathWriting dataset ('synthetic_images' and 'synthetic_labels.txt'). We developed Python scripts to parse the InkML file format, extracting both the stroke coordinates and the ground truth LaTeX annotations.

To prepare the data for image-based deep learning models, we implemented a visualization function ('visualize_inkml') using 'matplotlib' and 'Pillow' (PIL) to convert the InkML stroke data into grayscale PNG images. This function handles the necessary coordinate transformations and saves the resulting images. Building upon this, we created a comprehensive preprocessing pipeline ('process_inkml_folder') that iterates through a specified directory of InkML files, generates corresponding PNG images (saved to an 'images' directory), and compiles a labels file ('labels.txt') mapping each image filename to its respective LaTeX ground truth string. This pipeline was successfully tested on a subset of the synthetic dataset (processing 3000 samples initially).

For efficient model training, we implemented a custom PyTorch 'Dataset' class ('HandwrittenMathDataset') capable of loading the generated images and their corresponding LaTeX labels. This class incorporates standard image transformations (converting images to PyTorch tensors). We also implemented functionality ('create_train_test_split') to split the dataset into training and testing sets using PyTorch's 'random_split' and created 'DataLoader' instances ('train_loader', 'test_loader') to handle batching and shuffling during training.

Finally, a crucial step for the sequence generation task was building a character-level vocabulary ('build_vocab') from the LaTeX labels present in the dataset. This resulted in a vocabulary size of 95 unique tokens, including special tokens like `<start>`, `<end>`, `<pad>`, and `<unk>`. We developed helper functions ('string_to_tensor' and 'tensor_to_string') to convert between LaTeX strings and padded tensor representations suitable for input/output with sequence models. Below is a description of the purpose of each of these special characters:

<start> Marks the beginning of a sequence; signals the decoder to start generating.

<end> Marks the end of a sequence; signals the decoder to stop generation.

<pad> Used to pad shorter sequences to a fixed length for batch processing.

<unk> Represents any character not seen during vocabulary construction.

Figure 1: An example of a handwritten integral function that the model will transform into latex.

5.2 Initial Model Implementation and Training

Following the proposed approach, we implemented the first major component: the image-to-LaTeX conversion model.

Our initial implementation ('HandwrittenMathToLatexModel') utilizes a Convolutional Neural Network (CNN) based encoder ('CNNEncoder') for feature extraction from the input images and a standard Transformer decoder ('TransformerDecoder') for generating the LaTeX sequence. The CNN encoder consists of basic convolutional and pooling layers, while the decoder uses standard Transformer decoder layers with positional embeddings and causal masking. A linear layer projects the CNN features to match the decoder's expected embedding dimension. We established a training loop and a testing loop using PyTorch, employing the Cross-Entropy loss function (ignoring the padding index) and the Adam optimizer. Initial training runs over 10 epochs showed promising results, with both training and testing losses steadily decreasing (reaching approximately 1.4 and 1.5, respectively), indicating that the model architecture is capable of learning the task. This is demonstrated in Figure 2.

To enhance performance, we developed an improved version of this model ('ImprovedHandwrittenMathToLatexModel'). This iteration features a more sophisticated CNN encoder ('ImprovedCNNEncoder') incorporating batch normalization, more layers, and adaptive average pooling. The Transformer decoder ('ImprovedTransformerDecoder') was also enhanced with a larger model dimension ('d_model=512'), more layers, increased dropout for regularization, and proper sinusoidal positional encodings ('PositionalEncoding'). We refined the training process by switching to the AdamW Loshchilov & Hutter (2019) optimizer, implementing a OneCycleLR Smith (2018) learning rate scheduler, and incorporating label smoothing into the Cross-Entropy loss function. Training this improved model for 10 epochs also demonstrated successful learning, with losses decreasing consistently (reaching approximately 1.4 and 1.5 for training and testing, respectively). While the final loss was higher than the simpler model in this run, the enhanced architecture and training techniques provide a strong foundation for further tuning and scaling.

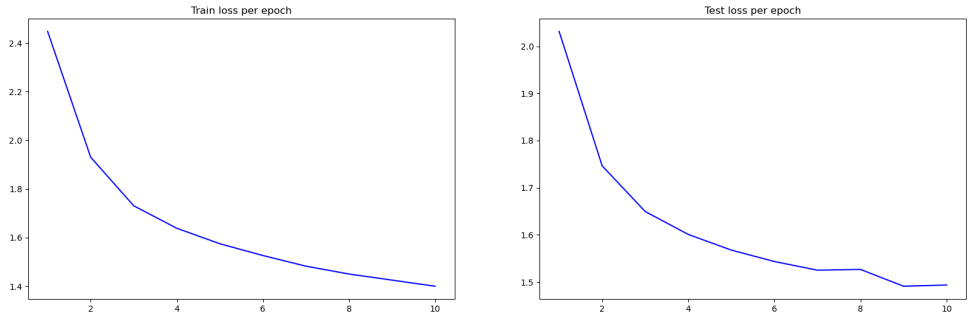

Figure 2: Train Loss and Test Loss per Epoch during Training.

5.3 Final Model Implementation and Training

Following the Project Progress Report and Lightning Talk, we implemented significant modifications to our model architecture, data handling, and training procedures, leading to substantial improvements over the initial prototypes.

The final model architecture, encapsulated in the 'ImprovedHandwrittenMathToLatexModel' class, leverages a more powerful encoder-decoder structure. For the encoder, we replaced the simple custom CNN with a pre-trained ResNet-34 backbone ('ImprovedCNNEncoder'), adapting it for our grayscale input images. Specifically, the initial convolutional layer of the ResNet was modified to accept single-channel input, and its weights were initialized using the mean of the original pre-trained weights across the color channels. This allows us to benefit from features learned on the large ImageNet dataset. The ResNet features (outputting 512 channels from 'layer4') are then projected down to the model's hidden dimension ($d_{\text{model}} = 512$) using a 1x1 convolutional layer. An adaptive average pooling layer ('nn.AdaptiveAvgPool2d') is employed before the final projection to ensure a consistent spatial output size (7×7) which is then flattened and reshaped to the sequence format expected by the decoder ($[\text{sequence length}, \text{batch size}, d_{\text{model}}]$).

The decoder ('ImprovedTransformerDecoder') remains based on the standard Transformer architecture, utilizing PyTorch's 'nn.TransformerDecoder' and 'nn.TransformerDecoderLayer'. It consists of 6 decoder layers, 8 attention heads, a feedforward dimension of 2048, and a dropout rate of 0.1. Crucially, we incorporated learned positional embeddings ('nn.Embedding') for the target sequence, replacing the previously used fixed sinusoidal embeddings, allowing the model to learn task-specific positional information up to a maximum sequence length of 200 tokens.

Data handling was enhanced through the 'EnhancedHandwrittenMathDataset' class. This introduced data augmentation techniques, specifically random rotation (up to $\pm 5^\circ$) and random scaling (between 90% and 110%), each applied with a probability of 25% during training to improve model invariance and generalization. Augmentation was disabled during testing. Both the dataset loading and vocabulary building ('improved_build_vocab') were made more robust against errors in the input files. Additionally, string-to-tensor conversion ('improved_string_to_tensor') was updated to correctly handle batch-first padding and generate padding masks.

The training process ('train_with_scheduling') was significantly refined. We utilized a larger dataset of 48,000 examples (split 90% train, 10% test) over the 3,000 we were initially training on and increased the training duration to 50 epochs. The AdamW optimizer was retained, configured with a learning rate of 1×10^{-4} and weight decay of 1×10^{-5}. While the previous iteration of our project experimented with learning rate scheduling (OneCycleLR) and label smoothing, this final training run employed a fixed learning rate and standard Cross-Entropy loss (ignoring the padding index) for stability and simplicity during extended training. Gradient clipping ('max_norm=1.0') was introduced to prevent exploding gradients, a common issue in training Transformers. For monitoring, 'tqdm' progress bars were added for visual feedback on batch processing, and model checkpoints were saved every 10 epochs. The explicit validation loop within the training function was removed to accelerate training iterations on the large dataset, focusing on observing the convergence of the training loss.

Finally, for improved inference quality, we implemented beam search decoding ('beam_search_decode') directly within the model class, using a beam size of 5. This allows the model to explore multiple potential output sequences at each step, generally producing more coherent and accurate LaTeX strings compared to greedy decoding. The training over 50 epochs with these enhancements showed convergence of the training loss down to 0.0175, indicating successful learning. This training loss is significantly lower than any of the previous model architectures and experiments that we tried.

5.4 Fine-tune the LLM for LaTeX Syntax Correction

The second major component of our proposed approach involves refining the potentially imperfect LaTeX output generated by the image-to-LaTeX model. To achieve this, we employed a reinforcement learning based fine-tuning technique called Group-Relative Policy Optimization (GRPO) on a pre-trained Large Language Model (LLM).

Specifically, we chose the `microsoft/Phi-4-mini-instruct` model as our base LLM, known for its strong instruction-following capabilities for its relatively small size of about 3 billion parameters. The fine-tuning process aimed to teach the model to correct syntactic errors commonly found in machine-generated LaTeX, and was conducted on four Nvidia A40 GPUs at the Minnesota Supercomputing Institute (MSI).

The training data for this phase was constructed from the outputs of our image-to-LaTeX model on a test set. We created a dataset where each entry contained the predicted (potentially incorrect) LaTeX string and the corresponding ground truth LaTeX string. Using this data, we formatted prompts for the LLM, instructing it to fix the provided text. The prompt structure was: `"Please ensure that the following text is valid LaTeX by fixing syntax issues as needed. Here is the potentially invalid LaTeX: [predicted LaTeX]. What is the fixed valid LaTeX: "`.

The core of the GRPO process lies in its reward mechanism. We defined a custom reward function using the `rapidfuzz` library Bachmann et al. (2025). This function calculates the normalized similarity ratio (`fuzz.ratio`) between the LLM's generated correction and the ground truth label. The reward, a similarity score scaled between 0.0 and 1.0, incentivizes the model to produce outputs that are highly similar to the correct LaTeX string, thus guiding it towards syntactic correctness. A reward of 1.0 is given for a perfect match, while dissimilar or empty outputs receive lower rewards.

We utilized the `GRPOTrainer` class from the `trl` (Transformer Reinforcement Learning) library from Hugging Face to manage the fine-tuning process. Key training arguments included a `per_device_train_batch_size` of 4 and `num_generations` set to 4 to manage memory usage on the available hardware. We enabled `bf16` mixed-precision training for efficiency. Due to storage constraints, intermediate checkpoints were not saved (`save_strategy="no"`), but logging occurred every 50 steps to track the reward progress over time.

To handle the computational demands of fine-tuning the LLM, we leveraged distributed training using `accelerate` and DeepSpeed Rasley et al. (2020). The DeepSpeed configuration employed ZeRO Stage 3 optimization (`zero_stage: 3`), which partitions the model's parameters, gradients, and optimizer states across multiple GPUs (4 GPUs in the case of this project) to significantly reduce memory requirements per device. This setup, combined with `bf16` mixed precision, allowed us to efficiently fine-tune the Phi-4-mini model on our available GPU resources (4 NVIDIA A40s).

The final output of this stage is a fine-tuned version of the `Phi-4-mini-instruct` model, specialized in correcting errors within LaTeX code generated by our primary recognition model. This model serves as a post-processing step, taking the initial LaTeX prediction as input and outputting a refined, syntactically valid version. Figure 3 shows the Weights & Biases graphs from our attempts at fine-tuning the LLM.

Figure 3: Weights & Biases training graphs. The first run (green) is our initial attempt to fine-tune the LLM with only 300 samples, and the second run (orange) is our final attempt using 4,800.

5.5 Evaluation Method for the Fully Functional System

Evaluating the performance of the complete system, which comprises the image-to-LaTeX conversion model followed by the LLM-based error correction, requires a comprehensive approach to assess both the initial translation accuracy and the effectiveness of the post-processing step. Our evaluation focuses on comparing the system's final output (the LLM-corrected LaTeX string) against the ground truth LaTeX labels from the test set. We also analyze the performance before and after the LLM correction to quantify its impact.

The evaluation is performed on the held-out test split of the enhanced MathWriting dataset, which consists of 4800 samples (10% of the total 48,000 examples). The process for each sample in the test set is as follows:

1. The input handwritten mathematical expression image is fed into the trained 'Improved-HandwrittenMathToLatexModel'.
2. The model generates an initial LaTeX prediction sequence using beam search decoding with a beam size of 5. This sequence is then converted into a string.
3. This initial predicted LaTeX string is then passed to the fine-tuned 'microsoft/Phi-4-mini-instruct' LLM. A specific prompt format, "Please ensure that the following text is valid LaTeX by fixing syntax issues as needed. Here is the potentially invalid LaTeX: [predicted LaTeX]. What is the fixed valid LaTeX: ", is used to instruct the LLM to perform the correction.
4. The LLM processes the input and generates a corrected LaTeX string. Test-time scaling is applied by generating multiple possible corrections (`num_generations=4` with 'num_beams=4') and selecting the one with the highest reward (similarity to the ground truth) using the RapidFuzz ratio.
5. The final corrected LaTeX string from the LLM is compared against the ground truth LaTeX label for evaluation.

To quantitatively measure the performance, we utilize several metrics:

- **Rapidfuzz Ratio:** This metric, calculated using 'rapidfuzz.fuzz.ratio', provides a similarity score between two strings ranging from 0.0 to 1.0. We use this to calculate an average reward score for both the initial encoder-decoder predictions and the final LLM corrections, allowing us to directly assess the improvement brought by the LLM. A custom reward function based on this ratio incentivizes the LLM to produce outputs highly similar to the ground truth.
- **BLEU (Bilingual Evaluation Understudy):** We use the Hugging Face 'evaluate' library's BLEU metric to assess the n-gram overlap between the generated corrected LaTeX and the reference ground truth. This provides an indication of the syntactic similarity and the presence of correct sequences of tokens. We report the mean, median, and standard deviation of the precision scores across the test set.
- **BERTScore:** Also using the 'evaluate' library, BERTScore measures the semantic similarity between the generated corrected LaTeX and the reference using contextual embeddings from a BERT model. This metric is valuable as it can capture semantic correctness even if the exact wording or token sequence differs slightly from the ground truth. We report the mean, median, and standard deviation of the precision scores.

By reporting both the RapidFuzz ratio improvement and widely accepted natural language generation metrics like BLEU and BERTScore, we aim to provide a comprehensive evaluation of the effectiveness of our hybrid image-to-LaTeX conversion and LLM-based error correction system. The comparison of average RapidFuzz scores before and after LLM correction specifically highlights the contribution of the fine-tuned language model in improving the syntactic and semantic correctness of the generated LaTeX.

6 Results and Analysis

Overall, our encoder-decoder model architecture performs very well on the task of handwritten math image to LaTeX conversion, demonstrating an average test accuracy of 85.59% on the test set of 4,800 examples. Adding our fine-tuned LLM based on Phi-4-mini-instruct on top of this raises the average test accuracy to 86.22%, which is a noticeable increase of about 0.63%, but is much less than we were hypothesizing the increase in accuracy would be. We think the reason why the fine-tuned LLM as a post-processing step was not able to improve performance substantially is because it was trained and prompted to correct syntax issues in LaTeX code as we originally anticipated our encoder-decoder model having issues with producing valid LaTeX, but this did not turn out to be the

case. Most of the model's outputs were already valid LaTeX without syntax issues, which pleasantly surprised us.

Instead, the main mistakes that our trained encoder-decoder model makes are simply confusing similar symbols with each other, as they can be very hard to differentiate sometimes, depending on the handwriting. For instance, our model would often confuse the symbols 'w' and 'ω' with each other, which results in a large error (low fuzzy accuracy) as the strings 'w' and '`\omega`' are very different from each other. Similarly, as seen in Figure 5, it would often predict '`\prime`' as '′', which similarly results in a large error (low fuzzy accuracy) as those two strings are very different. This is a very interesting problem, as we expected our encoder-decoder model to struggle with even producing valid LaTeX, but it turns out that it can do that very well. Instead, the main issue it has is differentiating between similar handwritten symbols and deciding whether to generate a signal character like '′' or the LaTeX command for that character like '`\prime`'. The full final evaluation results of our model can be found below in Figure 4.

```
LLM Evaluation Results:
LLM Mean Bleu Score: 0.8064
LLM Median Bleu Score: 0.8000
LLM Bleu Score Std Dev: 0.0792
LLM Mean Bert Score: 0.9734
LLM Median Bert Score: 0.9891
LLM Bert Score Std Dev: 0.0353
Average Encoder-Decoder Prediction Reward: 0.8558662811989127
Average LLM Correction Reward: 0.8621804360954791
len(prediction_list): 4865, len(reference_list): 4865 len(llm_correction_list): 4865
Saved predictions to eval_with_llm_results.csv
```

	prediction	reference	llm_correction
0	w_{1}=\frac{L}{mr^{2}}	\omega_{1}=\frac{L_{1}}{mr^{2}}	w_{1}=\frac{L}{mr^{2}}
1	A = 3\sqrt{25+15\sqrt{2a^2}}	A=3\sqrt{25+10\sqrt5a^2}	A = 3\sqrt{25+15\sqrt{2a^2}}
2	W = \int F dX	W = \int F dx	W = \int F dX
3	\left(\frac{5}{\sqrt{7}}\right)^{4}\cdot\frac{...	\left(\frac{5}{\sqrt{7}}\right)^{4}\cdot\frac{...	\left(\frac{5}{\sqrt{7}}\right)^{4}\cdot\frac{...
4	d = \sqrt{h(2R + h)}	d = \sqrt{h(2R + h)}	d = \sqrt{h(2R + h)}

Figure 4: Summary of our final results where "Reward" means fuzzy accuracy.

```
Reference: f=f^{\prime}

Encoder-Decoder Prediction: f=f'
```

Figure 5: One example of a tricky mistake made by our encoder-decoder model.

7 Conclusion and Future Work

In this report, we have presented a hybrid deep learning pipeline for converting handwritten mathematical expressions into syntactically valid LaTeX code, combining a ResNet-based encoder–Transformer decoder architecture with a fine-tuned LLM for error correction. Our final system achieves an average RapidFuzz accuracy of 85.59% after beam search decoding and yields a modest improvement to 86.22% when applying the LLM post-processor. These results indicate that the core image-to-LaTeX model is already proficient at producing valid syntax, and that the primary remaining errors stem from symbol-level ambiguities rather than grammar mistakes.

Reflecting on our experiments, we observed that the pre-trained ResNet backbone provided strong feature representations even on grayscale inputs, and that beam search decoding effectively balances exploration of candidate sequences. However, the limited impact of the LLM post-processor suggests that further gains will require targeted approaches to the hardest cases—namely, visually similar symbols and unusual handwriting styles. We also noted that, while our data augmentation strategies improved generalization, there remains a long tail of rare symbols and complex two-dimensional layouts (such as matrices and nested fractions) that challenge the current architecture.

As future work, we would like to explore several promising avenues:

- **Symbol Classification Module:** Leverage the MathWriting dataset's symbol-level annotations to train a standalone classifier for high-confusion pairs (such as w vs. ω and ' vs. `\prime`). We could integrate this module by loading its weights into the encoder and adding a parallel symbol-prediction head to guide the decoder.

- **Structural Layout Modeling:** Extend the model to incorporate two-dimensional layout transformers or graph-based representations, improving handling of multi-line expressions, matrices, and spatially arranged constructs.
- **Interactive Correction Interface:** Develop a user-in-the-loop system where ambiguous symbols are flagged during inference, allowing users to confirm or correct them and thereby generate better training examples for continual learning.
- **Domain Adaptation and Style Transfer:** Apply domain adaptation techniques or style-transfer networks to normalize diverse handwriting styles before recognition, reducing variability at the input stage.
- **End-to-End Joint Training:** Investigate a unified training framework in which the symbol classifier, encoder–decoder, and LLM post-processor are fine-tuned jointly, potentially via multi-task or curriculum learning to maximize overall transcription fidelity.

By pursuing these directions, we believe we could further close the gap between human-level readability and machine transcription accuracy, moving toward a practical tool for seamless digitization of handwritten mathematical content.

REFERENCES

Max Bachmann, layday, thomasryde, Georgia Kokkinou, Henry Schreiner, Jeppe Fihl-Pearson, dheeraj, Vioshim, pekkarr, Michał Górny, Moshe Sherman, Cristian Le, odidev, glynn, Trenton H, tr halfspace, dotlambda, Nicolas Renkamp, Kwuang Tang, Hugo Le Moine, Guy Rosin, Dan Hess, Christian Clauss, Blake V., and Delfini. rapidfuzz/rapidfuzz: Release 3.13.0, April 2025. URL https://doi.org/10.5281/zenodo.15133267.

Philippe Gervais, Asya Fadeeva, and Andrii Maksai. Mathwriting: A dataset for handwritten mathematical expression recognition, 2024. URL https://arxiv.org/abs/2404.10690.

Anssi Kanervisto. im2latex-100k , arxiv:1609.04938, June 2016. URL https://doi.org/10.5281/zenodo.56198.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pp. 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703.

Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay, 2018. URL https://arxiv.org/abs/1803.09820.