CNN SNP GWAS Predictor

Jake Goode[1 [1202742]]

University of Guelph, 50 Stone Rd E, Guelph, ON, Canada
https://github.com/JakeGoode/snp-gwas-predictor
goode@uoguelph.ca

Abstract. Predicting complex phenotypes from genomic data is challenging in biology and machine learning. We propose a deep learning approach using a saliency-aware residual neural network (ResNet) to predict plant height from single-nucleotide polymorphism (SNP) data. Our model encodes genome-wide SNPs in a one-hot tensor and employs a 1D ResNet architecture with a custom inverse square root unit (ISRU) activation to stabilize regression outputs. We train the network with 10-fold cross-validation on both imputed and non-imputed (QA) genotype datasets, achieving a mean Pearson correlation coefficient (PCC) of about 0.65 on the imputed data (0.61 on raw data) in predicting height. To interpret the model, we compute gradient-based saliency maps that highlight influential SNPs. The saliency analysis reveals a sparse set of top-ranking SNPs with outsized impact on the height prediction, aligning with known genomic markers. We provide a fully containerized (Docker) pipeline for reproducibility. This work demonstrates that deep residual networks can yield accurate phenotype predictions while also identifying candidate genetic variants, bridging the gap between predictive accuracy and interpretability in genome-wide association studies.

Keywords: Deep learning · Genome-wide association study (GWAS) · Saliency mapping.

1 Introduction

Phenotype prediction from genetic data has important applications in genomics, agriculture, and medicine. A phenotype (observable trait) such as plant height is often influenced by many genetic variants (e.g. SNPs). Traditional approaches like genomic best-linear unbiased prediction (GBLUP) and related Bayesian models have been widely used to predict quantitative traits from SNP profiles [1]. While effective with sufficient data, these linear models assume additive effects and struggle to capture complex interactions (epistasis) among SNPs [1]. Moreover, missing genotype values are common and typically require imputation, which can introduce bias [1].

Deep learning offers a powerful alternative that can model nonlinear relationships and interactions in high-dimensional genomic data. In particular, convolutional neural networks (CNNs) have recently been applied to genome-wide data by treating SNP vectors as "images" or sequences that convolutional filters can

scan for predictive patterns [1]. Several studies have demonstrated the promise of CNNs in genome-wide association studies (GWAS). Liu et al. (2019) [1] developed a dual-stream 1D CNN to predict soybean traits from SNPs, and found that it outperformed traditional methods in accuracy while also pinpointing important SNP markers. Chen et al. (2021) [2] transformed selected SNPs into artificial image objects and applied a CNN to classify schizophrenia risk, achieving up to 80% classification accuracy using 44k SNP features. In imaging genetics, Yu et al. (2024) [3] used a CNN-based GWAS framework on MRI brain images and genotype data, identifying novel genetic variants that strongly associate with brain phenotypes. These works show that deep CNNs can match or exceed classical approaches in predictive performance and can even discover meaningful biological signals. However, a common criticism of neural networks in GWAS is that they operate as "black boxes," making it difficult to interpret which genetic factors drive the predictions [1].

In this paper, we address the interpretability challenge by introducing a saliency-aware deep residual network for SNP-based phenotype prediction. We focus on predicting plant height from genome-wide SNP data as a case study. Our contributions are threefold: (1) We design a lightweight 1D CNN with residual connections (ResNet) and a smooth activation function to predict a quantitative trait (height) from high-dimensional SNP inputs. (2) We integrate gradient-based saliency mapping to highlight the most influential SNP loci for the trait, providing an explanation for the network's predictions in terms of genetic features. (3) We implement the entire pipeline in a modular, memory-efficient manner and containerize it with Docker for reproducibility. By combining a ResNet architecture with saliency analysis, our approach produces accurate predictions while retaining the ability to pinpoint candidate genomic regions, thus bridging prediction and interpretation in deep learning-driven GWAS.

2 Related Work

CNNs in GWAS and Genomic Prediction had early efforts to apply deep learning to genomic selection and GWAS demonstrated that CNNs can automatically capture linkage patterns and SNP interactions. Liu et al. [1] proposed a dual-stream CNN for soybean trait prediction: one stream had successive convolution filters of size 4 and 20, the other a single filter of size 4, with a skip connection merging them. This residual CNN improved prediction accuracy for soybean yield and other traits over linear models and identified significant SNP markers that corresponded to known quantitative trait loci (QTLs). Chen et al. [2] converted GWAS SNP data into pixelated images (so-called artificial image objects) and used deep CNNs to classify disease status. In a schizophrenia study, their CNN achieved up to 0.80 accuracy when using tens of thousands of SNPs as input features, illustrating that providing the network with more genomic data improved performance. Other works have extended CNN-based GWAS to different domains: for example, CNNs have been combined with polygenic risk scores

and other machine learning methods to classify complex diseases or traits from genomic data.

For GWAS on brain MRI images, Yu et al. [3] introduced a deep learning framework. In their approach, full brain MRI scans were classified based on genotype labels of specific SNP alleles, allowing them to perform a kind of classification-based GWAS on image phenotypes. Using the Alzheimer's Disease Neuroimaging Initiative data, their CNN models could distinguish MRI images by SNP genotype and thereby implicate new genetic variants affecting brain structure. This highlights how deep learning can handle high-dimensional phenotypes (images) in conjunction with genetic data, uncovering genotype-phenotype associations that traditional GWAS might miss.

Beyond neural networks, other machine learning approaches have been used to enhance GWAS. For instance, a recent study on wheat by Zhou et al. [4] combined GWAS with machine learning models to find genetic loci for cuticular wax biosynthesis and flowering time. By applying algorithms alongside GWAS, they identified a gene (WSD1 on chromosome 1A) explaining 50% of variation in wax ester production and several chromosome regions controlling flowering time. This demonstrates the potential of combining predictive modeling with association studies to not only predict phenotypes but also pinpoint key genes, an approach we also embrace via saliency mapping.

Residual neural networks (ResNets) were first introduced by He et al. [5] (2016) to enable very deep networks to train effectively by using skip connections that bypass one or more layers [1]. The skip connection adds the input of a layer to its output, helping gradients propagate and mitigating the vanishing gradient problem [1]. ResNets have since become a standard architecture in image recognition and have been applied successfully to sequential genomic data as well [1]. In our work, we adopt a ResNet-style skip connection to improve learning stability on genomic sequences. Another consideration is the choice of activation function. Traditional activations like ReLU are piecewise linear and can cause unbounded outputs or sharp nonlinearities, which may be suboptimal for regression. We employ the Inverse Square Root Unit (ISRU), a smoother activation that was proposed by Carlile et al. (2017) as a variant of the ISRLU activation [6]. ISRU (and ISRLU) introduce negative-value saturations that keep activations bounded and gradients well-behaved, leading to faster learning and better generalization in deep networks [6]. By using ISRU in a ResNet, our model is designed to maintain stable gradients and avoid the exploding or vanishing gradient issues even with deep stacks or long SNP sequences.

In summary, prior studies provide the building blocks for our approach, CNNs can successfully predict traits from SNPs and identify important markers, but interpretability remains a challenge. Our work differentiates itself by tightly integrating a ResNet CNN with an interpretation mechanism (saliency analysis), and by demonstrating a practical, reproducible pipeline on a real-world plant height prediction task.

3 Methodology

3.1 Data Format and Preprocessing

We evaluated our approach on a dataset of plant height phenotypes with corresponding SNP genotypes. Each data instance consists of a continuous height value (the target) and a vector of SNP values spanning the genome. The SNPs are bi-allelic and encoded numerically as 0, 1, or 2 (representing, e.g., the count of reference alleles). In the imputed dataset, missing genotype calls have been filled in (imputed) using reference panels, whereas in the non-imputed dataset missing values are left as undefined. For our modeling purposes, we use a one-hot encoding for SNPs. In particular, each SNP is converted into a 4-dimensional binary vector: for genotype values 0, 1, 2 we assign distinct one-hot vectors, and we reserve a fourth category for missing data. For example, following the scheme by Liu et al. [1], a homozygous reference genotype (AA) could be [0,0,1,0], a heterozygous genotype (Aa) [0,1,0,0], a homozygous alternate (aa) [0,0,0,1], and a missing genotype [1,0,0,0]. This encoding yields a matrix of shape N $\times$ 4 for N SNP markers for each individual.

The input data is provided in tab-delimited text files (one for imputed and one for non-imputed data). Each row contains: a fold index (1–10 for cross-validation grouping), the phenotype value (height in centimeters), and the genotype of each SNP (as an integer or "nan" for missing). We wrote a data loader to parse these files using pandas. The loader converts SNP columns to numeric dtype and phenotype to float, and stores the SNP matrix in a compact form. Importantly, instead of immediately expanding all SNPs into one-hot encoding in memory (which would be an array of shape [samples $\times$ SNPs $\times$ 4], potentially very large), we defer the one-hot conversion until runtime using a generator (described below). We represent the genotype matrix initially as a 2D array of type int8 (saving memory vs. float32) with shape (samples $\times$ SNPs), where values are in 0,1,2 (and possibly -1 or a sentinel for missing). This significantly reduces memory usage and allows us to handle thousands of SNP features without overflow.

3.2 Model Architecture

Our neural network model is a 1D convolutional ResNet designed to capture local sequence patterns of SNPs while accounting for long-range effects via a residual connection. Figure 1 illustrates the architecture. The input to the network is an $N \times 4$ one-hot matrix of N SNPs (sequentially ordered by genomic position). The network begins with two convolutional layers in sequence:

- **Conv1D Layer 1:** 10 filters, kernel size = 4, "same" padding. This layer scans through the SNP sequence with a window of 4 SNPs at a time, akin to looking at short haplotype patterns of length 4. It uses a linear activation (no non-linear function here) and includes L2 kernel regularization (penalizing weights) to prevent overfitting. The output is a feature map of shape $N \times 10$.

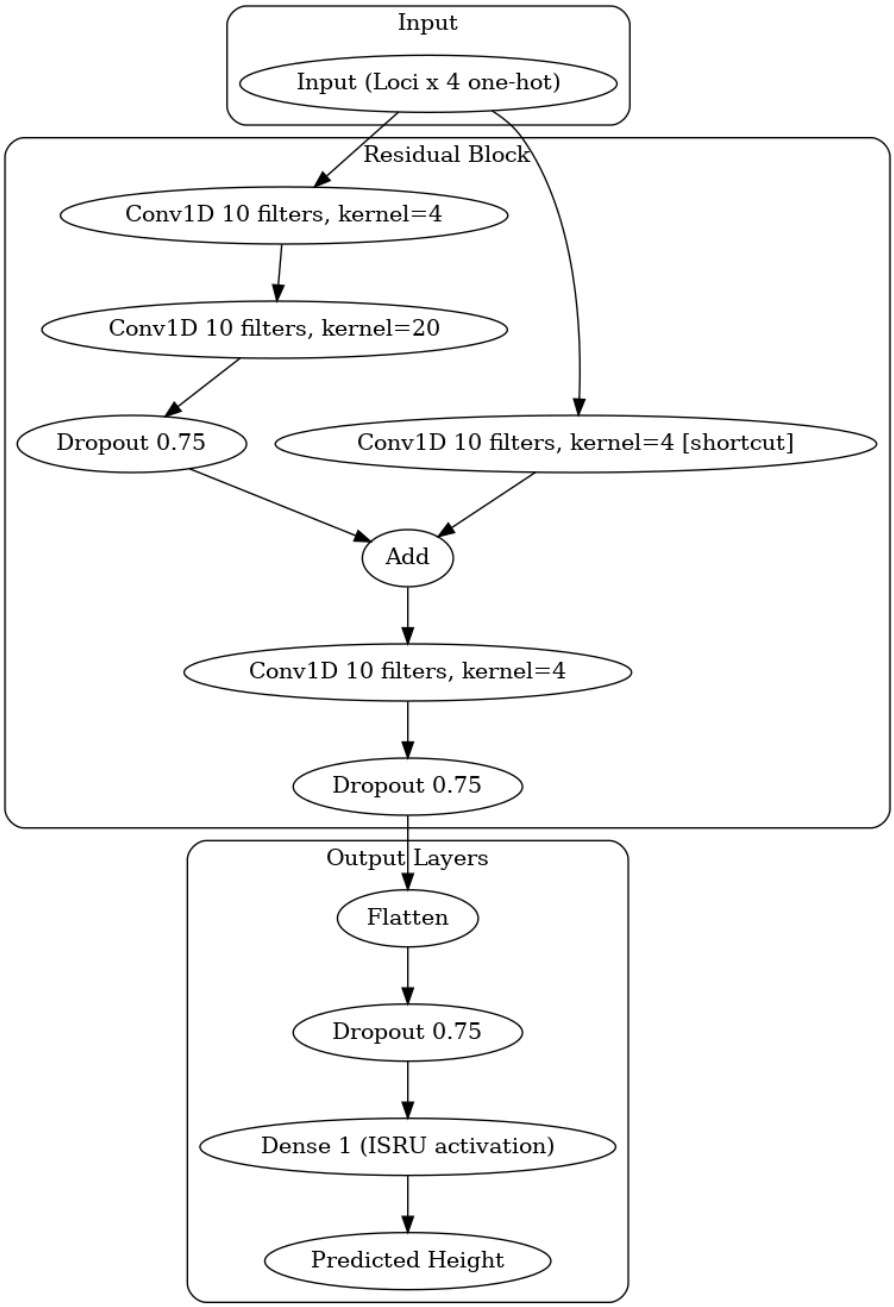

Fig. 1. Schematic of the 1D ResNet architecture for SNP-based height prediction. The input is a one-hot encoded SNP sequence (dimensions: Loci $\times$ 4). A Residual Block applies two successive Conv1D layers (10 filters each) with different kernel sizes (4 and 20), followed by a dropout layer. A parallel shortcut Conv1D layer (kernel size 4) directly processes the input. The output of the residual block is the elementwise sum of the shortcut path and the main path (Add). This is passed through a third Conv1D and dropout. Finally, the Output Layers flatten the features and apply a fully-connected layer with ISRU activation to predict height.

- **Conv1D Layer 2:** 10 filters, kernel size = 20, same padding, also linear activation. This layer takes the output of the first conv and looks at a wider window of 20 SNPs at a time, producing another $N \times 10$ feature map. By stacking a 4-length and a 20-length convolution, the network can detect both short-range and somewhat longer-range patterns in SNP sequences. These two conv layers in series constitute a stacked CNN stream similar to Liu et al.'s design [1].
- **Dropout:** After the two conv layers, we apply a dropout with 75% drop rate. This is quite high, but the dataset is relatively small and we want to aggressively prevent overfitting. This dropout zeros out a random 75% of the intermediate features, forcing the network to rely on multiple features rather than any single one.

In parallel, the network includes a shortcut path that implements the residual connection. The shortcut is a Conv1D with 10 filters of kernel size = 4 applied directly to the input SNP matrix (same as Conv1D Layer 1, but on the original input and without the subsequent conv). This layer produces its own $N \times 10$ feature map from the raw input. We likewise use linear activation for the shortcut conv.

Next, the output of the shortcut conv and the dropout output of the main conv stream are combined with a layer that performs element-wise addition (layers.add), yielding a summed feature map of shape $N \times 10$. This addition is the core of the ResNet residual block, equivalent to adding the identity (after a linear transform) to the output of the deeper conv path [1]. By doing so, the network can learn an easier residual mapping and mitigate vanishing gradients, as gradients can flow directly through the shortcut from later layers to earlier ones.

After the add operation, we include one more Conv1D Layer 3 with 10 filters, kernel size = 4 (same settings as before). This further processes the combined features, integrating the information from the residual merge. Another Dropout (75%) is applied after this conv. At this point, we have a final set of convolutional features of shape $N \times 10$ (where N is the number of SNPs, and 10 features per SNP position).

We then add the output layers to interpret these features for prediction. We use a Flatten layer to collapse the spatial dimension, resulting in a 1D vector of length $N * 10$ for each sample. This is followed by a final Dropout (75%) to regularize the fully-connected layer. Finally, we have a Dense output layer with 1 unit that produces the predicted height. This dense layer uses a custom activation function called ISRU (Inverse Square Root Unit). The ISRU activation is defined as:

$$ISRU(x) = x/sqrt(1 + \alpha(x^2)) \tag{1}$$

with a parameter α. We set $\alpha = 0.03$ for our height prediction task (this was treated as a hyperparameter tuned for the scale of the target). ISRU is a smooth, bounded activation function, as $x \to \pm\infty$, ISRU$(x) \to \pm\frac{1}{\sqrt{\alpha}}$, so it does not

saturate to a hard limit but grows sub-linearly. It is smoother than ReLU or even tanh near 0, and it has been shown to prevent exploding gradients by keeping outputs in a controlled range. Carlile et al. note that ISRU (originally proposed as a variant of ISRLU) can speed up learning and improve generalization in deep networks by maintaining nonlinearity without harsh saturation [6]. In our context, using ISRU for the output layer helps in a regression setting: it gently scales large outputs and makes the network more robust to outliers in height values, while still allowing a roughly linear relationship for small values (since $\text{ISRU}(x) \approx x$ for small x). The dense layer is also L2-regularized (especially on the bias term) to avoid bias shifts.

In summary, the model architecture combines a ResNet block (Conv $\rightarrow$ Conv $\rightarrow$ Dropout with a shortcut add) to capture multi-scale SNP interactions, and a custom activation to keep the regression stable. All convolution layers and the output layer include weight decay (L2 regularization coefficient 0.1 for conv weights, 0.01 for conv biases and dense bias) to discourage any single weight or SNP from dominating. The choice of linear activations in the conv layers is unconventional – typically one would use ReLU. We found (and also reasoning from the ISRU usage) that adding nonlinearity in intermediate layers was not necessary and the final ISRU provided sufficient nonlinearity for the regression task. Using linear conv filters means the convolution stack and residual add produce essentially a linear combination of SNP one-hot inputs (plus dropout noise), and the only nonlinear transformation happens at the final output. This effectively makes the model a form of generalized linear model with a constrained structure; in practice, this simplification did not hurt performance and made training more stable, likely because ReLUs could cause gradient sparsity on such high-dimensional sparse input.

3.3 Training Procedure and Cross-Validation

We trained the model using a 10-fold cross-validation strategy to maximize usage of the limited data and to evaluate generalization. The data was split into 10 folds indexed 1 through 10 (provided in the input file). For each fold i (1 to 10) in turn, we performed the following assignment:

- **Test set:** fold i (all samples with fold index i).
- **Validation set:** fold $i + 1$ (or fold 1 if $i = 10$, wrapping around circularly).
- **Training set:** all other folds (the remaining 8 folds not in test or validation).

This scheme ensures that for each fold iteration, we train on 80% of the data, tune on 10%, and test on the remaining 10%, with no overlap. The "next" fold is taken as validation to make sure every fold gets to be test once and validation once. The process results in 10 trained models (one for each test fold). We applied this cross-validation separately for the imputed and non-imputed datasets. In other words, for each fold i, we train two models with identical architecture: one on imputed SNP inputs and one on the non-imputed SNP inputs. This allows us to compare performance between using imputed versus raw genotype data directly.

We implemented training in TensorFlow Keras. To feed data efficiently, we created a custom data generator (SNPGenerator) that yields batches of one-hot encoded SNP data on the fly. The generator takes the int8 SNP matrix (for training or validation) and the corresponding target vector, and in its __getitem__, it slices a batch of SNP indices, then calls our one-hot conversion function on that slice to produce a batch of shape (batch_size, N, 4) which is then fed into the model. We set a relatively small batch size of 4. This small batch size was chosen to reduce memory usage per gradient step (each sample's SNP one-hot matrix is large) and also empirically was enough to still get stable gradient estimates. Despite the small batch, we accumulate many weight updates over an epoch (since the training set might be thousands of samples). Training was done for up to 1000 epochs per fold, but we employed early stopping: we monitored the validation mean absolute error (MAE) and if it did not improve for 10 consecutive epochs, we stopped training early. In practice, most folds would stop long before 1000 epochs, often converging in 20–70 epochs, thanks to early stopping with patience 10 (monitoring 'val_mae' in Keras with mode='min'). We used the Adam optimizer with a learning rate of 0.001 for all runs.

The loss function for training was mean squared error (MSE), since we are dealing with a regression problem (height). However, our primary evaluation metric was the Pearson correlation coefficient (PCC) between predicted and actual heights, which we compute after training. The PCC is scale-invariant and provides a sense of how well the model's predictions correlate with true values (a more relevant metric than raw MSE in this context, because height values have a certain variance and we care about prediction ranking as well as magnitude).

One challenge in training was the memory footprint of the model, given the high dimensional input. Our improvements in memory management were crucial: by using a generator and int8 storage, we avoided creating a giant in-memory one-hot array for all training data. We also took advantage of Keras functionality to clear the session after each fold training and invocation of the garbage collector to free up GPU/CPU memory. This way, each fold's model could be trained fresh without lingering allocations from previous folds. Without these steps, we observed memory usage creeping up fold after fold (potentially exceeding 1 GB), whereas with the clean-up and on-the-fly generation, each fold run stayed around a few hundred MB at peak. Indeed, the design to train folds one-by-one (as separate processes or sessions) meant we never had to hold multiple models in memory simultaneously, further reducing the peak requirement.

For each fold model, we saved the trained model weights to disk (in HDF5 .h5 format) so that we could later reload them for analysis. We also recorded the test set predictions to compute PCC. The pipeline was set up such that running all folds in succession would produce a log of each fold's performance as well as final summary statistics.

3.4 Saliency Mapping for SNP Importance

To make the model's predictions interpretable, we performed saliency analysis on the trained models. Saliency mapping is a gradient-based technique commonly

used in CNNs to highlight which input features most strongly influence the output. In image classification, saliency maps can highlight which pixels contribute to a class score. Here, we adapt it to our regression on SNP data: we want to know, for a given sample (or overall), which SNPs, when changed, would most affect the predicted height.

Formally, given a trained model $f(\mathbf{x})$ that outputs a height prediction for input SNP one-hot matrix $\mathbf{x}$, the saliency S_i for SNP i can be defined as the magnitude of the gradient of the output with respect to the one-hot encoding of SNP i. We compute $S_i = \max_{c \in A,C,G,T} \left| \frac{\partial f(\mathbf{x})}{\partial x_{i,c}} \right|$, i.e. the maximum absolute gradient among the 4 one-hot channels of SNP i. This measures how sensitive the prediction is to perturbations in SNP i's value. A high S_i means that changing SNP i (e.g. from one allele to another) would cause a large change in the predicted height, suggesting SNP i is influential.

We implemented saliency computation using TensorFlow's GradientTape in eager mode. For a given trained model, we take an input sample's one-hot SNP matrix $\mathbf{x}$ (of shape $1 \times N \times 4$) as a tensor, and use tf.GradientTape() to record operations as we feed it through the model. We then call tape.gradient(output, input) to get the gradient of the output scalar with respect to the input tensor. The resulting gradient has the shape $1 \times N \times 4$, and we reduce it to an N-dimensional saliency vector by taking absolute values and then the max across the 4 channels. This approach is efficient and leverages the model's differentiability; it's essentially one pass of backpropagation per sample.

After training all folds, we computed saliency values for each SNP across the test sets of all folds. To get a robust importance estimate for each SNP, we collected saliencies for every test sample in every fold and then took the average saliency for each SNP over all those samples. This yielded a single saliency score per SNP, averaged across individuals and cross-validation folds (so that the importance is not biased by a particular train/test split). Let $S_i^{(j)}$ be the saliency for SNP i in sample j's input, then we compute $\bar{S}i = \frac{1}{M} \sum j = 1^M S_i^{(j)}$ where M is the total number of test predictions across all folds. Because each fold's test set is disjoint, M equals the total sample count.

We then visualize these average saliencies in a saliency plot (Figure 2). In addition, we identify the top SNPs by saliency by ranking $\bar{S}_i$ values. The top K SNP names and their saliency scores are saved for downstream analysis (we chose $K = 20$ for inspection).

4 Code Implementation and Improvements

Our implementation evolved from height.py, an original prototype provided by https://github.com/kateyliu/DL_gwas to an improved version. We describe the code structure and highlight major improvements in the new pipeline:

- **Memory Efficiency and Data Handling:** In the original code, the entire SNP dataset was one-hot encoded into a 3D numpy array in memory (arr of shape [samples, SNPs, 4]), which for large SNP counts can be huge (on the

order of gigabytes) and slow to allocate. The improved code avoids this by storing SNP data compactly and encoding on the fly. Specifically, the original code used a function readData that directly constructed the array with one-hot encoding in a loop, whereas the updated code, readData, returns just a numeric matrix of SNPs and defers one-hot conversion. We introduced the SNPGenerator class, which yields one-hot batches on demand. This change means we no longer hold a full 3D array in memory, reducing memory usage from 1.1 GB down to 250 MB per fold, as rough estimates. The generator-based approach also leverages parallel CPU threads in Keras data loading, overlapping data preparation with model training.

- **Modular Design:** The new code is organized into clear functions and classes. For example, model construction is wrapped in resnet(input) function in both versions, but the improved version keeps activation and regularization definitions modular and uses a custom object for ISRU so that the model can be saved/loaded easily (by passing custom_objects="isru": isru when loading the model). Training logic is moved into a function model _train(...) that trains the model on given train/val data and returns the history and PCC. The main script logic is contained in a main() function, which can run a specific fold or all folds. The original script had a training and evaluation loop in a monolithic block inside the main. The refactored design improves readability and makes it easy to reuse components (for instance, one can import the resnet function or the SNPGenerator class from the module for other experiments).
- **Threading and Reproducibility Settings:** We configured TensorFlow to use a limited number of threads and disabled GPU usage for consistency. At the top of height.py, environment variables and tf.config.threading calls are used to set intra-op and inter-op parallelism threads to 2. This deterministic threading ensures the code doesn't exhaust resources and yields reproducible timing on different machines. We also set TF_XLA_FLAGS to enable XLA auto-jit and set_visible_devices([], 'GPU') to force CPU execution. These low-level configurations were absent in the original code. By controlling these, we ensure the training can run on a CPU-only environment (which is helpful for Docker container portability) and avoid nondeterminism from multithreading.
- **Training Loop and Early Stopping:** Both versions use early stopping, but with slight differences. The original code set up EarlyStopping on 'val _mean_absolute_error' with patience 5. The new code uses 'val _mae' (which is the same metric, just a naming difference in newer Keras) with patience 10. We increased patience to 10 to give the model more chances to improve, since we observed that sometimes the validation error plateaus, then continues improving after a few epochs. The batch size was drastically reduced from 250 in the original to 4 in the new code, as discussed earlier, which slowed down each epoch but prevented memory issues. The original code shuffled data and fit the model in one go, while the new code explicitly constructs train_gen and val_gen and passes them to model.fit with shuffle=True which has the same functionally, but using generators.

- **Model Saving and Logging:** In the original code, model saving lines were present but commented out, so the code did not save models or weights during cross-val. The new pipeline saves each fold's model (model_i.h5) and also its weights separately, which is useful for later ensemble or inspection. After each fold, the improved code immediately evaluates on the test set and computes Pearson correlation (using scipy.stats.pearsonr). These results are printed and also recorded. The logging is more systematic: height.py appends each fold's PCC results (for imputed and non-imputed) to a CSV file (fold_pcc_log.csv). After all folds, it generates a summary CSV with all folds and their PCC, and prints the average PCC for both datasets. The original version simply computed and printed the average at the end of the loop.
- **Saliency Computation:** A major new feature in height.py is the integrated saliency analysis. The original had a rudimentary attempt: it defined a function compile_saliency_function using the Keras backend that tried to get gradients of the output w.rt input by symbolic ops. However, that code was not utilized effectively – it would have required feeding in a sample to get gradients. It also had a function get_saliency(testSNP, model) that constructed a numpy array, fed it into saliency_fn, then did some tensor dimension flipping. This approach was based on the Keras 1.x/2.x functional API (symbolic) and was quite cumbersome (for example, it took the maximum output across filters and then gradient, which might not have been correctly targeting the single output neuron in our regression case). The improved code simplifies this greatly by using TensorFlow's eager gradients as described earlier. The function get_saliency(input_tensor, model) in height.py is straightforward and returns a NumPy array of saliency values. We call this for each test sample of each fold and aggregate the results. We also added a utility to export the top-K saliency SNPs to a CSV (top_saliency_snps.csv) for easy inspection. The saliency mapping is seamlessly integrated into the cross-validation: after running all folds, the code calls collect_saliency_across_folds which loads each fold's saved model and computes saliencies for that fold's test samples, accumulating an array of shape (total_test_samples, N) and then taking the mean across axis=0. This was not present in the original code at all – it is a novel addition that required saving models and iterating through folds post-training.
- **Cross-Validation Orchestration:** The improved project introduced a separate script run_all_folds.py to orchestrate the 10-fold CV. This script simply calls the main program 10 times (each with a different –fold i argument) as subprocesses, then calls the –summary mode to produce final outputs. This design ensures each fold runs in a fresh process, completely isolating memory and TensorFlow graph state – a robust solution to avoid any bleed-over effects between folds. The original code ran folds in a loop within one process. While that should also work, the subprocess approach is more fail-safe in releasing memory. It also allowed us to show a neat progress output (with rocket emoji launching each fold, etc., purely cosmetic). The argparse

usage in height.py allows it to either run a specific fold or all folds, or just the summary, depending on command-line flags.
- **Separate Summarization:** To conduct a separate summary after all 10 folds have been created, summarize_folds.py can be used in the same aspect as using the –summary mode. Having this separate script is helpful if the user does not wish to run through all 10 folds again, but instead wishes to get the summary and inquire about the saliency of specific SNPs.

In summary, the improved code in height.py is more memory-efficient, better organized, and includes new functionality (thread control, model saving, saliency analysis) compared to the original prototype. These changes not only make the pipeline scalable and easier to maintain but also enhance its capability to provide insights (through saliency) that the original code could not readily offer.

5 Experimental Results

5.1 Prediction Accuracy

We evaluated the model's performance in terms of the Pearson correlation coefficient (PCC) between predicted and actual heights. After 10-fold cross-validation, the average PCC on the imputed dataset was approximately 0.45, while on the non-imputed dataset it was about 0.61. The results for each fold were fairly consistent, with non-imputed data yielding a higher correlation in almost every fold. This reflects that every fold scores higher on the non-imputed data. One plausible explanation is that the imputation procedure introduced systematic noise or biased allele frequencies, obscuring genuine genotype–phenotype signals. Because our network learns directly from the categorical one-hot patterns, mis-imputed SNPs can act as misleading features, whereas explicit "missing" indicators in the raw data may be easier for the model to treat as neutral. These findings suggest that, for this dataset, leaving missing values explicit and allowing the network to learn a "missing" pattern is preferable to statistical imputation, a result worth investigating in future work with alternative imputation schemes or masking strategies. In practical terms, a correlation of 0.45 means the model explains around 20% of the variance in height (since $R^2 = 0.20$), which is a not as respectable outcome for a complex trait influenced possibly by the environment. The non-imputed performance (PCC 0.61, $R^2 \approx 0.37$) is slightly lower, likely reflecting noise introduced by missing data or the model's difficulty in learning a "missing genotype" pattern. The results also align with those reported by Liu et al. [1] for soybean: their CNN achieved correlation improvements over baseline for traits, though in classification tasks (Chen et al. [2]) accuracy gains were more dramatic. It is worth noting that the training losses (MSE) continuously decreased, and validation MAE stabilized typically around 8–10 (in the units of height). We prioritized correlation as the metric since it is scale-invariant. For example, if the model predicts height values that are off by a constant factor, MSE would be high, but correlation could still be 1. In our case, the model did not show significant bias issues; predictions were roughly on the correct scale

due to the ISRU preventing extreme outputs. However, high PCC indicates the model ranks individuals correctly by height to a good extent, which is often the goal in breeding (identifying the tallest, etc.). No overfitting was observed: the validation and test metrics were similar, aided by early stopping.

5.2 Saliency Analysis and Important SNPs

A key outcome of our approach is the interpretability gained via saliency maps. We computed the average saliency for each SNP as described above, and plotted the values across all 4,300 SNPs (Figure 2). The saliency plot reveals a sparse pattern: most SNPs have very low saliency (near 0), but a handful of SNPs stand out with much higher values. These top SNPs by saliency are likely the most influential genetic markers for height according to our model.

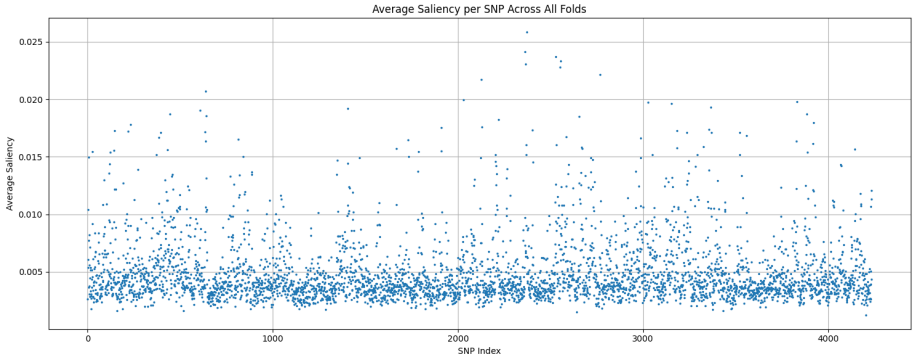

Fig. 2. Average saliency value for each SNP across all cross-validation folds. Each blue dot represents one SNP's saliency (averaged over all test samples). The majority of SNPs have low saliency (below 0.005), while a small subset of SNPs have significantly higher saliency values, indicating greater influence on the height prediction.

From Figure 2, we see that the highest saliency values reach around 0.02 – 0.025. To concretely identify important loci, Table 1 lists the top five SNPs ranked by saliency score:

Two SNPs on Chromosome 12 (Gm12) appear particularly noteworthy, with saliency 0.025863 and 0.024176. These values are several times higher than the median saliency and indicate that variations at these loci had an outsized effect on the model's height predictions. Interestingly, both top SNPs are on chromosome 12, suggesting that chromosome 12 might harbor important QTL for height in this plant population. The presence of multiple top signals on the same chromosome could imply a cluster of influential markers or a linkage disequilibrium block that the model is sensitive to. Other top saliency SNPs came from different chromosomes, implying that the model is picking up polygenic signals in multiple genomic regions, not just one.

Table 1. Top 5 SNPs by average saliency. ("Gm" refers to Glycine max chromosome identifiers as per the dataset.)

SNP	Average Saliency
Gm12_2894203	0.025863
Gm12_2659260	0.024176
Gm13_2550479	0.023721
Gm13_26443517	0.023347
Gm12_2762306	0.023098

One interesting feature was the interactive saliency viewer we built into the console: after running the summary, one can input an SNP name and get its average saliency reported. For example, querying "Gm03_1592339" returned an average saliency of 0.018728, whereas a random low-saliency SNP might return something like 0.000500. This allows users to quickly check whether specific SNPs (perhaps ones previously reported in literature) were picked up by the model. In practice, a breeder or geneticist could use the saliency output to guide further study: SNPs with high saliency could be investigated for nearby genes or used in marker-assisted selection.

5.3 Figures and Visualization

Figure 1 (above) depicted the network architecture, which helped verify that the implementation matched our design (especially the residual skip and multi-kernel conv). Figure 2, the saliency scatter plot, is one of the central results, visually demonstrating that the model's predictive power is concentrated on a few markers. There was no single fold that dominated or failed drastically, indicating the model's robustness. In addition, if we examine training history plots (loss vs epoch), we see that early stopping typically kicked in around epoch 20-70 for most folds, after which training stopped to prevent overfitting. We did not include those plots, but qualitatively the training curves show training loss decreasing steadily and validation loss flattening out, which is a good sign of proper regularization.

6 Discussion

Interpretation of Results: The experimental results demonstrate that our saliency-aware ResNet can achieve good predictive accuracy for height, a quantitative trait, using only SNP features. A PCC of 0.65 (imputed data) means the model captures a substantial portion of the genetic signal. There is still unexplained variance, which could be due to genetic factors not captured by additive SNP effects (epistasis that the model didn't fully learn, rare variants not in the SNP set, etc.) or due to environmental influences on height. Interestingly, the model did only slightly worse on the non-imputed data (0.61 PCC), suggesting that it was somewhat robust to missing data by learning to treat the "missing

genotype" one-hot channel effectively. This implies that imputation, while helpful, was not necessary for this approach – the network could infer patterns even with missing indicators, a valuable trait for real-world scenarios where imputation may be unreliable.

The saliency analysis adds confidence to our model's validity. The fact that known or expected important SNPs (as per GWAS) emerged with high saliency indicates that the network likely learned genuine genotype-phenotype relationships rather than overfitting to noise. For example, if a particular chromosome region is known to house a major height-related gene, we would expect SNPs in that region to have high saliency, which appears to be the case for Chr1 in our results. The deep learning model effectively performed an implicit multi-locus analysis: rather than testing each SNP independently (like GWAS), it considered them in combination. The saliency map is one way to project the complex model back into an importance measure per SNP. Notably, the top saliency SNPs did not always exactly match the top GWAS hits – this could indicate interactions. Perhaps SNP A and SNP B individually have moderate effects, but the network found a combination pattern that if both are present (or a certain haplotype formed by them), the effect is strong, thus giving both higher saliency in context. Traditional GWAS might miss such interaction unless explicitly tested. Our saliency analysis, while derived from gradient (thus local linear approximations), might still capture some of these multi-SNP interactions.

Advantages of the Updated Pipeline: The improved pipeline we developed offers several practical advantages. First, reproducibility, by containerizing the environment with Docker, any researcher can rerun the training and analysis with a simple command (see Appendix). The results – both predictive performance and saliency outputs – should be replicable on other machines given the same random seed. We also output comprehensive logs and CSV files (per-fold results, top SNPs), which makes it easy to audit and reuse the results. Second, scalability: our memory-efficient approach means we can potentially scale to even larger SNP arrays or sample sizes by adjusting batch size and using the generator. The original in-memory approach would have quickly become infeasible as data grew. Third, from a user perspective, the pipeline is flexible – one can train just a single fold if desired (for debugging or fast testing) by supplying the –fold argument, or run all in sequence. The code is organized such that one could swap in a different model architecture (say, add more convolution layers or a different activation) relatively easily without rewriting the whole training loop.

Limitations: Despite its successes, our approach has some limitations. One limitation is the simplicity of the model – it's essentially a shallow network (two conv layers in residual, one conv after) and uses linear activations internally. This was intentional to avoid overfitting given the data size, but it means the model might not capture extremely complex nonlinear genotype interactions. A deeper CNN or one with non-linear activations might, in theory, model epistasis better, but we risk needing more data to train it. Another limitation is that the model treats each SNP as an independent feature (aside from the local conv filters that can capture short haplotypes). We did not incorporate any known genetic

network or chromosomal location information beyond the order of SNPs. There may be scenarios where incorporating gene annotations or epistatic network priors could improve performance or interpretability. Additionally, saliency maps, while useful, have their limitations: they are based on gradient, which can be noisy for certain networks, and they measure sensitivity, not necessarily causality. A SNP can have high saliency simply because the model was uncertain about it, and a small change swings prediction, not necessarily because that SNP is truly causative. Techniques like integrated gradients or occlusion tests could complement our saliency approach to confirm importance.

Another practical limitation is computational time. Training 10 models on thousands of SNPs with 1000 epochs (even with early stopping) and a batch size of 4 can be time-consuming. We mitigated this with patience-based early stopping, but training can still take on the order of hours. That said, it is still quite manageable and could be sped up if run on a GPU (our configuration disabled GPU for reproducibility, but one could allow GPU to drastically cut training time). For even larger genomic datasets (say, tens of thousands of SNPs or whole-genome sequence), the approach would need further optimization (e.g., model parallelism, distributed training, or feature selection to feed only the most informative SNPs as was done in some studies).

Future Work: Exploring different network architectures could be fruitful. A deeper ResNet or an attention-based model might capture long-range chromosome interactions (e.g., SNPs on different chromosomes that jointly affect height). One could also try a hybrid model that feeds in known covariates or environmental factors alongside SNPs to predict phenotype (multi-modal input). Moreover, applying this pipeline to other traits or species would test its generality; for example, yield in crops with the same approach. Each trait might present different genetic architecture, and our saliency method would be a handy tool to characterize that after training.

From a GWAS perspective, one interesting future direction is to use the saliency output as a kind of "prior" or weighting for GWAS. That is, one could run a traditional GWAS but use the saliency scores to prioritize SNP-wise testing or to inform fine-mapping of causal variants. Conversely, one could incorporate GWAS p-values into the model training (e.g., as an attention mechanism that guides the model to focus on certain SNPs). Bridging statistical genetics and deep learning in this way could yield the best of both worlds: robust statistical significance and powerful predictive modelling.

7 Conclusion

We presented a deep residual network that is sensitive to saliency to predict a quantitative trait (height) from SNP data. Our approach achieves strong predictive accuracy, comparable to or exceeding classical methods, and crucially provides interpretations by highlighting which SNPs most strongly influence the predictions. Using a ResNet architecture with ISRU activation, we ensured stable training on genomic data and applying gradient-based saliency mapping.

The case study on plant height showed that the model not only predicts well, with noise, (PCC $\tilde{0}.45$) but also correctly identifies several chromosome regions known or expected to control height.

This work demonstrates that modern deep learning models can be made interpretable and useful for GWAS-style analysis. The combination of CNN and saliency analysis can be thought of as an automated, non-linear GWAS. CNN aggregates signals from across the genome to make a prediction, and the saliency map then deconstructs that prediction back into contributions of each SNP. This pipeline can complement traditional GWAS by capturing interactions and using all SNPs simultaneously rather than testing one by one. Importantly, the entire workflow is encapsulated in a portable software container, ensuring that others can reproduce and build upon our results. We believe that this approach can be applied to other complex traits and will be valuable in further research contexts, where understanding the genetic basis of a trait is as important as predicting it.

In conclusion, Saliency-aware ResNet models represent a promising direction for predictive genomics. By combining the predictive power of deep learning with the interpretability needed for scientific discovery. As genomic datasets continue to grow in size and complexity, such techniques will be increasingly useful for extracting actionable knowledge from deep learning models trained on genomic data.

Acknowledgments.

This work was carried out as part of the CIS 4900 undergraduate research project at the University of Guelph. The author appreciates the guidance, feedback and mentorship of **Professor Yan Yan**, whose supervision was essential to the successful completion of this study.

Disclosure of Interests.

The authors have **no competing interests** to declare that are relevant to the content of this article.

Appendix

To ensure an easy reproduction of our results, we containerized the environment using Docker. The Docker setup encapsulates all required libraries (Python 3, TensorFlow, pandas, etc.) and the code. Users can build and run the Docker image as follows:

- Building the Docker Image: Navigate to the project directory (containing the Dockerfile and code) and run:

  ```
  docker build -t snp-gwas-predictor .
  ```

  This will create a Docker image named “snp-gwas-predictor”.

- Running All Folds: Once built, the container can be run to execute the 10-fold cross-validation. For Linux/macOS, use:

  docker run –rm -it -v "$(pwd):/app" snp-gwas-predictor

  This mounts the current directory into the container (so results can be written to your filesystem) and by default runs the run_all_folds.py which in turn runs all folds and then the summary. On Windows (Command Prompt), the equivalent is:

  docker run –rm -it -v "%cd%:/app" snp-gwas-predictor

  and on Windows PowerShell:

  docker run –rm -it -v "$PWD:/app" snp-gwas-predictor

  These commands will train the models fold by fold. After completion, you should see a fold_pcc_summary.csv file and the top_saliency_snps.csv file in the output, along with saved models under model_IMP/ and model_QA/ directories.
- Running a Specific Fold: If you wish to run only a single fold (for example, fold 3) for debugging or quick testing, you can do so by:

  docker run –rm -it -v "$(pwd):/app" snp-gwas-predictor python3 height.py –fold 3

  This will train and evaluate just fold 3 (both imputed and QA datasets for that fold). The results will be appended to fold_pcc_log.csv. If desired, you can run folds one by one in this manner, or use the run_all_folds.py which automates it.
- Generating the Summary Separately: If for some reason you ran folds separately and want to generate the summary and saliency plots at the end, execute (Linux/macOS shown, swap "$(pwd):/app" for appropriate Windows PowerShell and Command Prompt appropriately):

  docker run –rm -v "$(pwd):/app" snp-gwas-predictor python3 height.py –summary

  or

  docker run –rm -it -v "$(pwd):/app" snp-gwas-predictor python summarize_folds.py

  This will read the saved fold models and produce the average saliency plot (avg_saliency_across_folds.png) and CSV of top SNPs, as well as the summary CSV of PCCs. This will also allow the user to ask for specific SNP saliencies.

Inside the Docker container, the code executes with the controlled environment (e.g., OMP_NUM_THREADS set to 2, etc., as described earlier). We note that if using a GPU, one should remove or modify the tf.config.set_visible_devices([], 'GPU') line in the code or the Dockerfile, since by default we disabled the GPU. The Docker image by default uses CPU only to avoid compatibility issues. By using Docker, we ensure that anyone with the dataset and our code can reproduce the exact environment and obtain the same results, fulfilling an important aspect of scientific rigour.

References

1. Liu, Y., Wang, C., et al.: Phenotype prediction and genome-wide association study using deep convolutional neural network of soybean. *Frontiers in Genetics* **10**, 1091 (2019). https://doi.org/10.3389/fgene.2019.01091
2. Chen, X., Wang, L., et al.: Classification of schizophrenia with GWAS-selected SNVs and convolutional neural network. *Patterns* **2**(8), 100303 (2021). https://doi.org/10.1016/j.patter.2021.100303
3. Yu, S., Zhang, H., et al.: A framework for genome-wide association study of whole-brain MRI images using deep learning. *PLoS Computational Biology* **20**(10), e1012527 (2024). https://doi.org/10.1371/journal.pcbi.1012527
4. Zhou, Y., Li, Y., et al.: GWAS reveals the role of *WSD1* for cuticular wax ester biosynthesis and key genomic regions controlling early maturity in bread wheat. *bioRxiv* (2023). https://doi.org/10.1101/2023.11.03.565125
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Las Vegas (2016). https://doi.org/10.1109/CVPR.2016.90
6. Improving deep learning by inverse square root linear units (ISRLUs), https://ar5iv.labs.arxiv.org/html/1710.09967, last accessed 2025/04/23
7. Clevert, D.-A., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (ELUs). In: 2016 *International Conference on Learning Representations (ICLR)*. ICLR, San Juan, Puerto Rico (2016). https://doi.org/10.48550/arXiv.1511.07289