# Appendix A

# Simulation results supplementary

| sample size | additive | 2nd order | 3rd order | 4th order |
|---|---|---|---|---|
| **10%** | 0.48 | 0.44 | 0.58 | 0.54 |
| **25%** | 0.24 | 0.23 | 0.30 | 0.46 |
| **50%** | 0.23 | 0.24 | 0.34 | 0.52 |
| **75%** | 0.23 | 0.21 | 0.32 | 0.61 |
| **100%** | 0.19 | 0.22 | 0.35 | 0.71 |

Table A.1 **Fractions of experiments where a non-linear solution was found by the NN out 100 simulations with a causal fraction of 0.5 of SNPs involved in statistical epistasis.** The values in the 'additive' column represent experiments where the ground truth genetic architecture was purely additive.

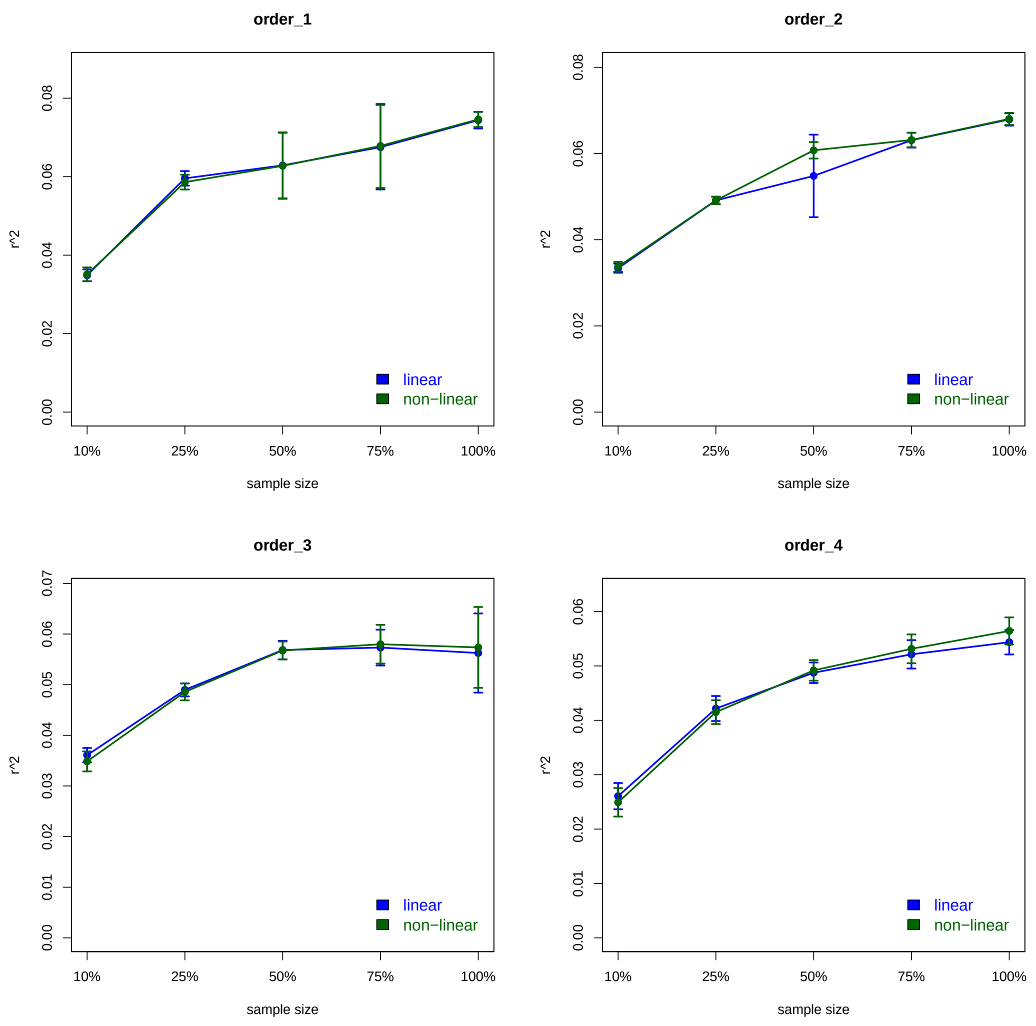

Fig. A.1 **NN performance on obtaining non-linear solutions under varying conditions for the experiments with a causal fraction of 0.5 of SNPs involved in statistical epistasis.** x-axis represents the % of sample size used and y-axis represents the $r^2$ of predicted vs observed phenotypes on the test set. Facets display experiments of genetic architectures that involve either additive, second, third and fourth-order interactions.

# Appendix B

# Neural-network supplementary

#### B.0.0.1 Convolutional neural-networks

The NN I described in Chapter one is what is known as a fully-connected NN, or FNN, which fit a hypothesis-free model that assumes that all input features are equally likely to interact with each other. However, for many data types, features that are spatially closer together in the input space are more likely to interact. Consider two intuitive examples: in most natural images the values of nearby pixels are more likely to form salient features like edges; similarly, nearby nucleotides are more likely to be part of the same regulatory element in a DNA sequence. Considering such local structures forms the basis of the convolutional neural-network (CNN) models.

The assumption of structure in the data is leveraged by CNNs via the introduction of a new layer type: the *convolution layer*. Here, neurons only consider a smaller local subset of the full input space which is termed the 'receptive field' of the neuron (a term which originated in the neuroscience literature (Hubel and Wiesel, 1962)). Instead of the hypothesis-free learning of FNN, in a CNN the neurons learn a vocabulary of features of a pre-defined size, known as 'kernels' or filters. As the term 'kernel' has many completely unrelated mathematical meanings, from here on, I will be using the term filter, to avoid confusion. Since these neurons no longer consider the entire input space; instead, they attempt to extract smaller reoccurring features, they need far fewer parameters to learn which results in greater power.

I will now move on to describe the details of the convolution operation itself. The convolution operation is a linear transformation where the filter is slid, or 'convolved', across the entire feature space which obtains a final output via element-wise multiplications. As my work involves one dimensional data (SNPs), I will illustrate this concept with 1D convolutions. Consider the following $1 \times 2$ size filter weight $w$ and a $1 \times 4$ size $Input$:

$$w = \begin{bmatrix} w_1 & w_2 \end{bmatrix}, Input = \begin{bmatrix} I_1 & I_2 & I_3 & I_4 \end{bmatrix}. \tag{B.1}$$

Assuming a stride of one and no padding, the filter $w$ may be applied to the *Input* at three locations, or patches, to obtain the following *Output*:

$$Output = \begin{bmatrix} O_1 & O_2 & O_3 \end{bmatrix}, \tag{B.2}$$

where the entries in the *Output* are defined by

$$O_1 = w_1 I_1 + w_2 I_2 \tag{B.3}$$
$$O_2 = w_1 I_2 + w_2 I_3 \tag{B.4}$$
$$O_3 = w_1 I_3 + w_2 I_4. \tag{B.5}$$

However, looping through the input space this way is inefficient, as high performance applications rely on massive parallelisation of computations via generalized matrix multiplications (Vasudevan et al., 2017). To facilitate this, the *Input* is first transformed via an 'im2col' function that stretches the input out so that all possible patches are represented in a single matrix **L** as

$$\mathbf{L} = im2col(Input) = \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_3 & I_4 \end{bmatrix}. \tag{B.6}$$

**L** may then be conveniently used in a single matrix multiplication to obtain a vector identical to B.2 by

$$Output = w\mathbf{L}. \tag{B.7}$$

To generate the entire output (**C**) of a layer with $d$ filters, $w$ is replaced by a matrix representing all neuron weights ($\mathbf{W} \in \mathbb{R}^{d \times q}$) which modifies the above equation to

$$\mathbf{C} = \mathbf{WL}. \tag{B.8}$$

It is notable, that in contrast to the fully-connected NN (eq 1.39), this weight matrix is now on the left hand side. This is because the layer has $d$ neurons that are restricted to be able to only learn pre-defined filters of size $q$. The left multiplication by **W** also illustrates the parameter-saving attribute of the convolution layer, as the number of parameters to be learned ($d \times q$) no longer depends on the number of features in the *Input*. This allows CNNs to surmount high-dimensional data, such as high-resolution images or long DNA sequence reads, which would be beyond the reach of FNNs. Equation B.8 obtains the output $\mathbf{C} \in \mathbb{R}^{d \times (3*n)}$, where all individual observations are flattened to be stored along one dimension. To clarify, this would mean that the transformed genotype observations for $n$ individuals are concatenated into one dimension. Therefore, to connect the output of this layer to the flow of the rest of the NN function, the matrix $\mathbf{C}$ needs to be reshaped and transposed so that each of the $n$ individuals stay on the rows as

$$\mathbf{C}' = Vec^{-1}(\mathbf{C})^T, \tag{B.9}$$

where $Vec^{-1}$ denotes the reshaping operation.

In summary, the convolutional layer's function may be described as the extraction of smaller subsets from the input space. These reusable features are then passed forward as inputs from which subsequent layers learn higher-order representations, which result provides an explanation why it is a common CNN architectural trait that shallower convolution layers have fewer filters and deeper ones have more. Shallower layers' filters learn lower-order features (such as edges in an image or short motifs in a DNA sequence), and deeper layers' filters learn higher-order features made up from the shallower layers' representations. This is in contrast with fully-connected layers which tend start wide and each subsequent layer narrows towards the output.

As a side note, my description so far was a simplified explanation of how convolution layers generate an output, as in most practical applications there is an extra dimension to be considered. These would represent either the three colour channels for images, or one of the four nucleotides in the case of DNA sequence data. The equations would then change to involve tensors instead of 2D arrays, but otherwise would remain identical.

After the aforementioned convolution operation, a subsampling step is commonly used as the dimensionality of the output would increase by a factor of $d * Q/p$, where $Q$ is the number of patches (three in the example) and $p$ is the size of the input. To manage the dimensionality, and also to make the layer less sensitive to a small local changes, either another convolution layer is used with a larger stride (Springenberg et al., 2014), or a so called '*pooling layer*' is applied that summarises the output of the a convolution (Weng et al., 1992).

A popular method that accomplishes the subsampling operation is the '*Max Pooling*' function which is applied by taking the maximum of each image patch. In the example that I described so far, this would be equivalent to $MP = max(\sigma(\mathbf{C}))$, which would return the largest scalar value from the output after the activation by $\sigma$. In practice, pooling layers may have different sizes than the filters. The pooling size used most frequently is two which downsamples the output of each convolution layer by half. While 'Max Pooling' is primarily used to down sample the activations, it is important to note that this also adds non-linearity as the $max()$ function depends on more than one value.

In conclusion, a single convolution layer may be added into the network I described in eq 1.39 by

$$Y = \sigma_k(\ldots\sigma_2(\sigma_1(\mathbf{C}'\mathbf{W_1})\mathbf{W_2})\ldots W_k), \tag{B.10}$$

where $\mathbf{C}'$ is the output of the last convolution layer I derived in B.9.

To maintain clarity of the overall model, a short-hand notation may be used that emphasises the layer-by-layer sequential transformations from the input towards the output. In this notation the model I described so far can be expressed as

$$NN : [In, C_1, FC_1, FC_2, \ldots, FC_k, Out], \tag{B.11}$$

where $C_1$ is the convolution layer, $FC$ are $k$ fully connected-layers and $in$ and $Out$ are the input and output layers, respectively. The element-wise activation functions are also not shown, but are assumed to take place after each layer with trainable weights. The advantage of this format is that adding $j$ convolution layers may then simply be expressed by

$$NN : \left[In, C_1, C_2, \ldots, C_j, FC_1, FC_2, \ldots, FC_k, Out\right]. \tag{B.12}$$