Lucas Hure-Maclaurin

Suraj Khetarpal

**Feed Forward Neural Networks Using Feature and Class Specialization**

CS81 Final Project

# Abstract

In this paper, we use specialized feed forward neural networks (FFNN) architectures to replicate the idea of “deliberate practice” in learning tasks. To do this, we create small ‘child’ FFNNs that specialize in learning only one feature or predicting only one class from a data set, and then feed the learned weights/biases to a larger ‘parent’ FFNN network. The parent network synthesizes what the child networks have learned and performs a classification task using all features and classes in the data set. We found that our specialized FFNN architectures performed similarly to standard FFNNs, albeit slightly worse on large data sets and slightly better on small data sets. However, they trained more quickly and required fewer node-to-node connections.

# Introduction

When humans attempt to learn a new high level skill, it can be helpful to employ deliberate practice. Deliberate practice is a process that starts by breaking the skill down into its subcomponents and learning each of them in a repetitive fashion and with immediate feedback until they become second nature, or intuitive. These newly developed sub-skill intuitions allow one to then develop the higher level skill with greater ease and with far more success. In this paper, we explore whether the idea of deliberate practice can be translated to the learning process of feed forward neural networks (FFNNs). We ask whether greater learning can be achieved by zooming in on particular aspects of a data set, developing low level intuitions based on those aspects, zooming back out, and then using those low level intuitions to better learn from the data set as a whole.

# Method

In order to replicate the idea of deliberate practice, we use a three step process:

1. Train many small FFNNs to each learn something different about the data set.
2. Create a larger FFNN that synthesizes the weights and biases learned by the smaller FFNNs
3. Train the larger FFNN to learn from the data set as a whole

In this paper, we will refer to the larger synthesis neural network as a “parent” neural network, and the smaller neural networks as the “child” neural networks. The parent network must be large enough to accommodate the multiple sets of weights and biases that emerge from the trained child networks. Fortunately, since the child networks have a narrow scope of learning focus, they can also have very narrow and very few hidden layers relative to the parent network. This makes it feasible to copy over child network weights and biases without having to utilize massive parent layers.

The parent neural network can have many different architectures depending on how its children are structured, how the children weights and biases are synthesized into the structure of the parent, and what additional structure the parent builds above the synthesized children. In this regard, our parent FFNNs can differ greatly from the structure of a standard FFNN.

Since there are many different ways to implement our algorithm, and since we require specialty architectures for both our parent and child FFNNs, we were forced to build all of our FFNN code from the ground up. We also coded a library of functions that give us the flexibility to build models based on any combination of our many design options. Design options include the following, each of which will be described in detail in subsequent sections of the paper:

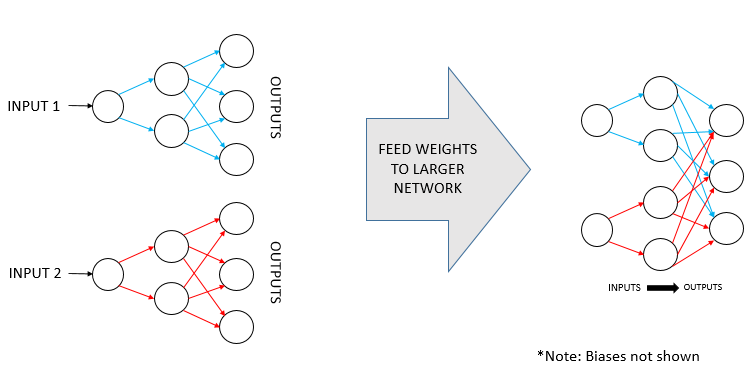
1. Children network specialization type
2. The addition of parent network virgin layers
3. Stripping of child network layers
4. Mutability of the pre-trained weights and biases
5. The addition of virgin cross weights

The first of our design options is the child network specialization type. In order to learn something specific about the data set, the child FFNNs will use one of two “specialization” schemes:

1. Specialization by FEATURE
2. Specialization by CLASS

## Child Network Specialization: By Feature

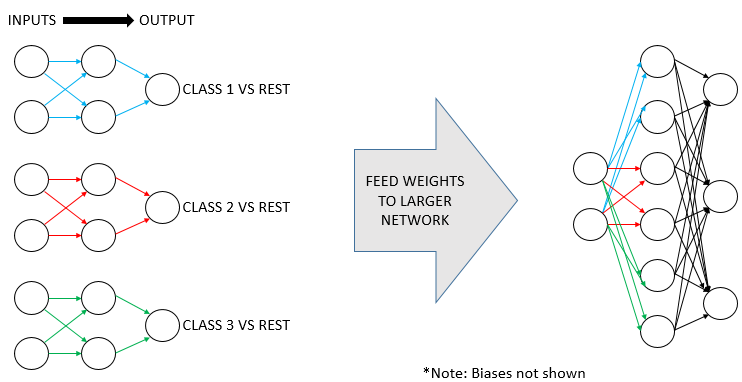
A child FFNN is specialized by feature when it has been trained using only one input feature from the data set, forcing the network to learn everything that it can from that feature. In order to build a parent network from feature specialized children, a separate child network must be trained for each feature in the data set. The below figure provides a visualization of the concept for the case of a data set with 2 input features and 3 output classes.



On the left are two child FFNNs that have been trained on one of the two features. The weights of the first child network are represented in blue and the second are in red. Note that the biases are not shown to make the image easier to follow. On the right, we see the most basic form of a parent network. It has been populated with the blue and red child weights, and there are still only two input features and three output classes. The input layer is not fully connected to the hidden layer, but the output layers of the children are merged into a single output layer. Later, we will discuss the design option called “mutability”, which determines whether the blue and red weights (and un-shown biases) in the parent are allowed to change via backpropagation. If they are immutable, or held fixed, then the parent network in the figure acts as a simple aggregator of the activation values that would have been generated by the child networks. The parent network would make its prediction based on the output node with the highest aggregate activation.

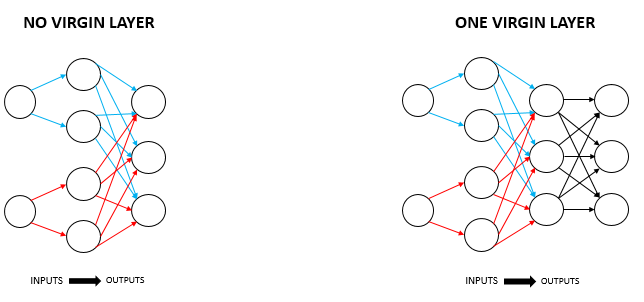
## Child Network Specialization: By Class

Specialization by class means that each child network is trained to differentiate a single class against all other classes, using all of the features in the data set. The figure below demonstrates the idea using the same example as before of a data set with 2 features and 3 classes.

On the left are three child networks, one per class. Each uses all features to train and outputs a single binary value, with a 1 meaning that the example belongs to the class that the child is being trained on, and a 0 for when the example belongs to any other class in the data set. On the right is a parent network that has been semi-populated with the blue, red, and green weights from the child networks. Only the first layer of child weights was used in this particular parent network. Note that any time child networks are specialized by class, the parent network’s input layer is automatically fully connected to its first hidden layer, since the same inputs are used by each child.

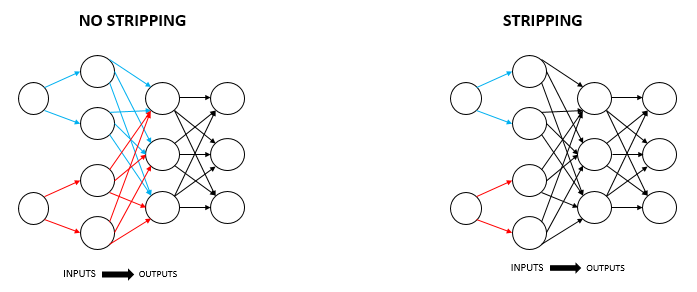
## Virgin Layers

We gave ourselves the design option to add virgin layers in our parent networks. Virgin layers are extra layers that are added above the children layers in the parent Neural Network. Essentially, they allow the parent neural network to learn how to better process the outputs of the children, and to learn feature relationships that were not captured by the children. Those extra layers are initialized with random weights.

On the left, we see a parent network made up of two children that were specialized by feature. On the right, virgin weights (and biases) have been added (shown in black) to create a fully connected virgin layer above the original output layer.

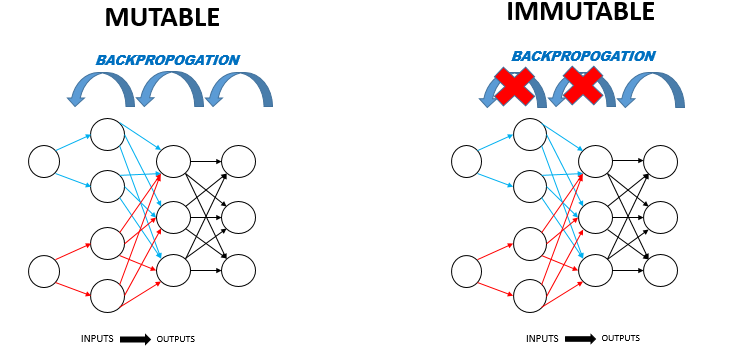
## Stripping Layers

We have the option of whether or not to strip away the output layers of the children weights and biases when populating the parent network. This option allows us to control whether we retain all of the children’s trained learning (by not stripping), or just the lower level feature learning (by stripping the highest layer). If we do strip a layer, we replace it with a fully connected virgin layer that has randomly initialized weights.

On the left, the middle layer has been built using the blue and red weights returned by the children after training. On the right, those same weights are drawn in black to show that the output layers of the children have been stripped and replaced by a virgin layer of weights.

## Mutability of Child Weights and Biases

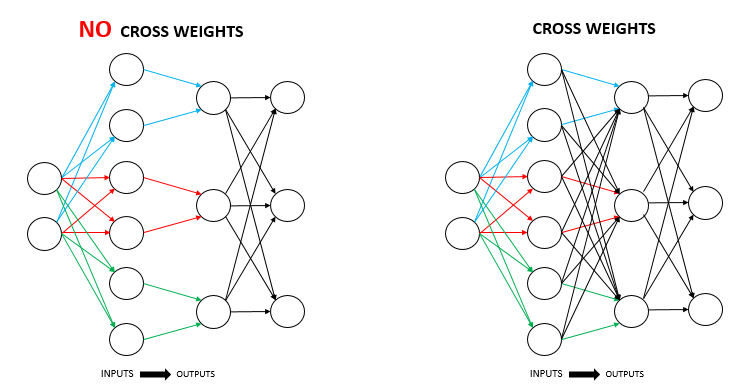
We gave ourselves the option of making the pre-trained children’s weights and biases either immutable or mutable after they are placed into the parent network.  If they are immutable, they will not be changed by the backpropagation process when the parent network trains.

On the left is a parent network with mutable child weights.  Backpropagation is done so that the blue and red child weights (and un-shown biases) are updated.  On the right side is the immutable case, where backpropagation is not done on the child weights and biases.

Mutability gives us the ability to control the rigidity of the parent network. When the children’s weights and biases are mutable, they are essentially an initialization scheme for the parent network’s weights/biases. The parent uses the children’s learned knowledge only as a starting point. On the other hand, when the children’s weights and biases are immutable, they serve as a rigid base structure for the virgin layers above them and for the ‘virgin cross weights’ (virgin cross weights are described in the next section). Immutability prevents the parent from being able to “forget” anything that was learned by the children.

## Virgin Cross Weights in Child Layers

The next option was whether to include virgin cross weights in the child layers.  Often times, when layers in the parent network are populated with children weights, these layers are not fully connected since the children do not connect all nodes from one layer to all nodes in the next layer (as shown on the left side of the figure below). Adding virgin cross weights serves to fill in the missing connections, making each layer fully connected. This allows the parent network to better learn inter-feature relationships at a low level, rather than relying entirely on virgin layers for this.

On the left is a parent network where the first and second layers come from three ‘class specialized’ children.  Notice that the middle layer of weights is not fully connected.  On the right side, we have introduced virgin cross weights (shown in black) to make the middle layer fully connected.

## Experimentation

We did experiments using many different specialty architectures. We narrowed the focus of our analysis by only choosing from a combination of the following option choices:

* Child specialization type: Class or Feature
* Number of virgin layers: 0, 1, or 2
* Number of stripped child layers: 0 or 1
* Number of child network hidden layers: 1 or 2
* Number of parent network hidden layers: 1 or 2
* Mutable or Immutable child weights/biases
* Virgin cross weights or no virgin cross weights

Although we wrote our code for maximum flexibility, we further narrowed our analysis by only using the following FFNN parameters:

* Learning type: Stochastic gradient descent
* Activation function: Logistic
* Batch size: 1 (online learning)
* Learning rate: Constant
* Momentum: None

In order to assess the results of our specialty architectures, we compared their accuracy scores to those of a standard FFNN of equivalent hidden layer size. Since our code is able to run both standard and specialty FFNNs, we did our accuracy comparisons using our own standard FFNNs rather than using prepackaged FFNN software.

## Data Sets

To test our architectures, we used multiple datasets, most notably the MNIST and Abalone data sets.  MNIST is a hand-written digit data set with 70K examples, 784 features, and 10 classes. Since the data set has 784 features and we cannot feasibly create 784 child networks, we had to perform dimensionality reduction prior to running any feature specialization analyses. We used a combination of PCA and LDA reduction techniques to create a new data set with 19 features (10 from PCA and 9 from LDA). Abalone is a data set of measurements of a species of sea snails, where the classes are the discrete age categories. It has 5K examples, 8 features, and 19 categories. While we used MNIST primarily for class specialization analyses, we used Abalone primarily for feature specialization analyses.

# STORY

We initially planned to do a project around feature specialization only, but then quickly realized that it may not work well on data sets with very large numbers of features, for example image recognition tasks, or data sets with very strong inter-feature relationships. With this in mind, we decided to add the class specialization option.

We started our work by building the standard FFNN code, and testing it using toy examples such as the famous XOR problem, and simple data sets such as IRIS. Once we had it working, we added many additional features to the FFNN so that it could do the following:

* Take in pre-trained weights/biases or initialize weights/biases itself
* Perform regularization
* Contain layers of any size
* Do limited backpropagation:
  + Backpropagation through child weights/biases but not through cross weights
  + Backpropagation through cross weight but not through child weights/biases
  + Backpropagation through only certain layers
* Validation accuracy scoring after each epoch

We then set to work creating the “Child Spawner” function which generates the child networks and feeds each of them a different manipulated version of the original data set. Finally, we coded all of the options described above in the ‘METHOD’ section of this paper.

When we tested using MNIST, we realized that the class specialized child networks were receiving imbalanced data sets. For each child, the data contains only 10% “class 1” examples (i.e. the example is the correct digit) while the other 90% is “class 0” (i.e. the example is not the correct digit). We wrote code to balance the data for each child by cutting down the number of “class 0” examples to match the number of “class 1”. To our surprise, this resulted in lower accuracy scores rather than higher. This indicates to us that the children train better using an imbalanced but larger data set.

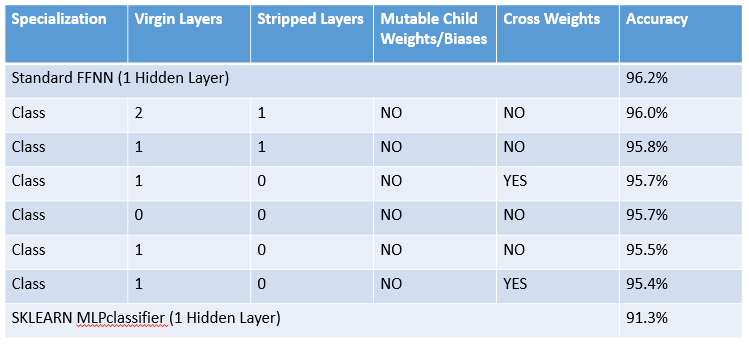
Since MNIST has 784 features, we had to perform dimensionality reduction prior to doing ‘feature specialization’ testing (or else we would have 784 child networks). We used PCA and LDA to reduce the data set to only 19 features and trained the child networks using this data set. Unfortunately, this resulted in poor accuracy scores, telling us that ‘feature specialization’ is not ideal for image recognition tasks. Therefore, going forward, we primarily used ‘class specialization’ in our analysis of MNIST.

At this point, we tested MNIST and Abalone using dozens of different combinations of the design options described above. We also modified our code to generate validation test set accuracy scores at each epoch so that we could see how the network learn over time. Results are shown below. Since we got mixed results using large training data sets, we also tested using small training data sets.

# RESULTS

## MNIST- Trained on 70% of Data Set

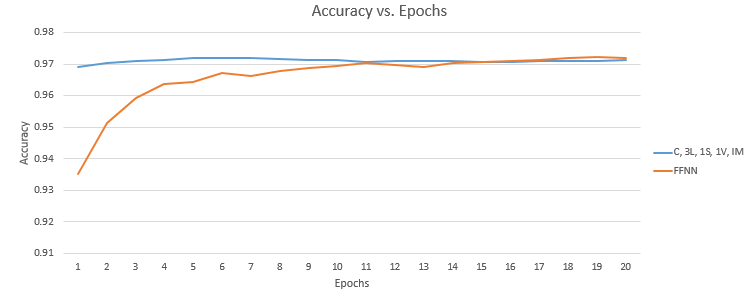
Classification of the MNIST data set went well for both the standard FFNN and for the specialty architectures, reaching around 95% accuracy. The below table shows accuracy scores for a selection of our specialty architectures that were trained using 70% of the data. We also include our standard FFNN results, as well as the results for the multilayer perceptron classifier that is built into the Scikit-Learn library.



Our specialty architectures’ accuracy scores are just slightly below that of our standard FFNN, and there isn’t a great deal of score variation. During our testing, we noted that we got consistently got better results by holding the child weights immutable.  The best scores came from stripping away a layer of the child network derived weights/biases, replacing it with one or two virgin layers, and then holding the child derived weights/biases immutable. Surprisingly, the built in Scikit-Learn MLP classifier performed the worst, at 5% below our standard FFNN.

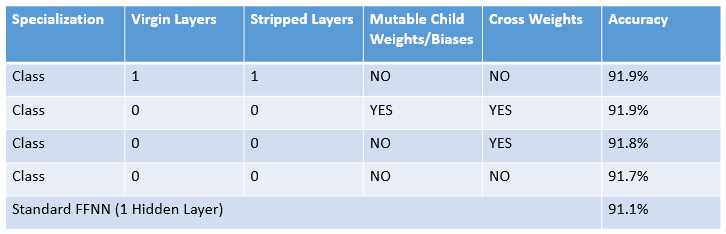
## Learning Speed Using MNIST

Our specialty architectures trained much more quickly than standard FFNNs.  By using the weights and biases from our small pre-trained child networks (which themselves train very quickly), the synthesized parent neural network achieves near maximum accuracy in just 1 epoch, while the standard FFNN takes around 10. The below figure shows the accuracy at each epochs of one of our specialty architectures (shown in blue) vs a standard FFNN (shown in orange). Both were trained using 70% of the MNIST data set, and tested using 7%.



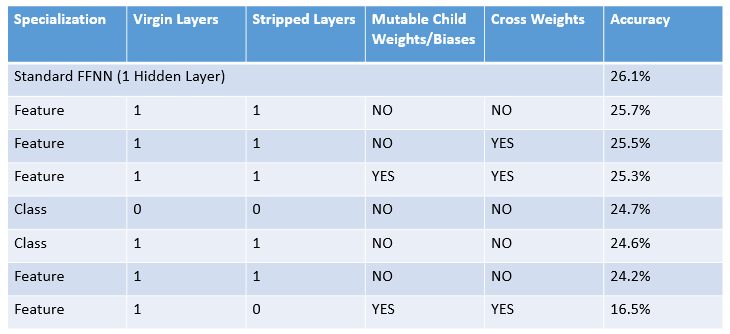
## MNIST- Trained on 5% of Data Set

We next tested the effect of training on a small dataset, using only 5% of the MNIST data set.  Here, we found that our specialty architectures surpassed the standard FFNN, meaning that our specialty architectures might be a good way to handle small data sets by maximizing what is derived from few examples in a data set.



## Abalone- Trained on 70% of Data Set

We next tried the Abalone dataset and focused more on ‘feature specialization’.  Again, the standard FFNN performed best, but only by a tiny percentage.  Of our specialty architectures, the best results came from using feature specialization, stripping the child output layer, replacing it with a virgin synthesis layer in the parent network, and holding the child network derived weights immutable.



# Discussion

When we started the project, we hypothesized that there would be some advantage to building a neural network atop a layer or two of pre-trained child network weights/biases. Our logic was that the parent network would easily build upon the lower level intuitions derived by the child networks. We knew that feature specialization has the drawback that the child networks are prevented from learning inter-feature relationships, and might prevent network from ignoring less useful features. However, we assumed that the addition of virgin layers in the parent network might resolve these deficiencies. To our surprise, we were unable to surpass the accuracy scores of our standard FFNN using either feature or class specialization when analyzing large data sets. We conclude that standard FFNNs are very effective at finding the optimal learning schemes even when initialized using strictly random weights. It is worth noting that our specialty architectures achieve almost the same accuracies as the standard FFNN, but often times with fewer node-to-node connections in the lower ‘child’ layers, and require far less backpropagation in the parent networks. Furthermore, the specialty architectures perform better than standard FFNNs on smaller data sets, telling us that the specialization by the children networks allows the specialty networks to “make the most of” the limited training data.

We found it interesting that the best specialty architectures were those with immutable child network derived weights/biases. This tells us that the children were learning useful information during their training, and that the parent networks worked best when they were unable to ‘forget’ what the children had learned. A major advantage of this revelation is that it allows for faster training, since the parent network does not need to perform backpropagation through any of the child derived layers. Overall, this results in very fast parent network training. Since the children networks are relatively small and only learn either only one feature or only one class, they also train very quickly. Overall, this means that our specialty architecture trains very quickly from beginning to end, and results in nearly the same accuracy as a slower training standard FFNN.

We were also surprised by the results from when the child weights/biases were used as an initialization scheme for the parent (which is done by making all child weights/biases fully mutable). Prior to the analysis, our hypothesis was that the child networks would provide a weight/bias initialization scheme that would allow the parent network to start closer to the ideal weight space solution than if the weights/biases were randomly initialized. For this reason, we expected the specialty architecture to find a more ideal solution than a standard FFNN, and to do it faster. But the results of our analysis are puzzling. The specialty architectures do seem to train much faster than standard FFNNs, suggesting that they do indeed start closer to an optimal set of weights/biases. However, after enough epochs, the standard FFNN eventually surpasses the specialty architecture in accuracy score, suggesting that the standard FFNN has found a more ideal set of weights/biases. We conclude that when the children networks specialize, they are learning some non-generalizable and very prominent data processing patterns that the parent network is unable to “unlearn” and that hold the parent back slightly.

# Summary and Future Work

The specialty FFNN architectures proposed in this paper had mixed results relative to the standard FFNN. They had roughly the same accuracy scores, albeit slightly lower for large data sets and slightly higher for small data sets. However, they train very quickly and use fewer inter-node connections than standard FFNNs.

In the future, we will redesign our algorithm so that the parent network is more selective in which children it takes weights/biases from, so that it ignores children who did not learn well. This may allow the parent to avoid having to ‘unlearn’ the children weights/biases that were less helpful in classification.

# Instructions to Run Code

In order to run the code, run the ‘story.ipynb’ file which walks through all of the analysis. Note that all of the functions that we wrote are stored in a separate library file called ‘NN.py’, which is imported in the story.ipynb.