A Machine Learning Approach to Convolutional Neural Network Hyperparameter Optimisation

Scott Brownlie
Centre for Simulation and Analytics
Cranfield University

15th October 2016

1 Introduction

In recent years convolutional neural networks (CNNs) have become the preferred class of models for image classification tasks. In such tasks, given a *training set* of images for which the true classes are known, the goal is to use the data to build a model which is able to predict the correct classes of new images. Model performance is evaluated on a separate *test set* of images and measured by some pre-defined error metric, such as percentage of examples classified incorrectly.

CNNs were first introduced by LeCun et al. in 1988 [11], but for many years the huge computation time required to train a useful CNN, which was traditionally done on a computer's CPU, severely limited their use. It was not until 2012 that Krizhevsky et al. demonstrated their true practical potential for image classification tasks when they entered their AlexNet model [12] in the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The competition "evaluates algorithms for object detection and image classification at large scale" [8] and the entries are a good gauge of the state-of-the-art in the field. Participants are provided with around 1.2 million images from the ImageNet database [7] with which to build their models and then asked to predict the classes of 150,000 test images from 1000 different categories.

Model performance is measured by the "top-5 test error rate", which is the percentage of test images for which the model fails to include the true class among its top five predictions. In 2012 AlexNet won with a score of 15.3%, compared to 26.2% for the second-best entry. Crucially, Krizhevsky et al. utilised a computer's GPUs to significantly reduce computation time, making it practical to train CNNs on large collections of images.

Every year since AlexNet's victory almost all ILSVRC entries, and certainly the top performers, have been CNNs. In 2014 researchers from Google came out on top with their 27-layer GoogeLeNet [13], which narrowly beat an even deeper network known as VGGNet [14] from the Visual Geometry Group at the University of Oxford. The 2015 winner ResNet [15] took network depth to a new level with 152 layers, marking the current state-of-the-art. Traditional approaches which involve engineers hand-coding feature extractors are unable to compete. By training on large collections of diverse images, well-designed CNNs are able to automatically learn rich feature representations for a wide range of image categories.

The core building block of all the networks mentioned above is a special type of layer known as the *convolutional layer* [3], which is responsible for most of the computation and feature extraction. However, the specific details of the network architectures are very different. As well as convolutional layers there are several other types, and before we can train a CNN we must decide how many of each type to use and in which order. For most layers we also have to specify properties, such as the number of filters in convolutional layers [3]. All these choices are known as model *hyperparameters*, and when designing a moderately deep network the number of hyperparameters can easily reach twenty or thirty, making the task of training a CNN from scratch seem overwhelming to anyone who is not an expert.

Hyperparameter optimisation [19] [20] [21] [22] is the process of selecting hyperparameter values with the goal of minimising the test error, which is usually estimated by separating a fraction of the training data into a *validation set* and evaluating each trained model on those examples. The most commonly used approaches to hyperparameter optimisation are grid search, random search and thoughtful manual search.

During grid search a list of values is defined for each hyperparameter and an exhaustive search is performed by training and evaluating a model for every possible combination of the hyperparameter values. For many machine learning algorithms it is common to optimise only two or three important hyperparameters, making it possible to search over a fairly large grid. When training a support vector machine [26], for example, we would typically only spend time fine-tuning the kernel scale and the box constraint. An exhaustive search of twenty values for both hyperparameters would require $20^2 = 400$ trials. This is not a huge amount, especially if powerful parallel computing resources are available. However, for a CNN with thirty hyperparameters, 20^{30} trials would be required to perform a similar exhaustive search, and a grid search of just two values each would require $2^{30} = 1,073,741,824$ trials. These numbers make grid search for CNN hyperparameter optimisation infeasible.

For random search we define a distribution of values for each hyperparameter. We specify how

many trials we want to run, and for each one we select the hyperparameters by sampling from the distributions. Bergstra and Bengio [22] showed that when configuring neural networks random search finds better models than grid search within a fraction of the computation time. They found that when searching over a large number of hyperparameters only a few have a significant impact on model performance, but which hyperparameters are important depends on the dataset. In such situations random search is far more efficient than grid search.

In practice, random search is usually combined with thoughtful manual search. A common strategy is to first perform a number of random search trials, sampling from wide hyperparameter domains and training for only a short time. Many of the models will fail to learn at all, and by analysing the validation results it is often possible to spot patterns and rule out large sections of the initial hyperparameter domains. More trials are then run, sampling from narrower domains and training for longer. This process is often repeated several times to home in on smaller and smaller domains which contain good hyperparameter values.

There are some drawbacks to this approach. Hyperparameter optimisation is generally seen as a nuisance and should ideally be as automated as possible, but this method requires that we periodically analyse validation results and update hyperparameter ranges as necessary. Also, when several hyperparameters are involved, as is the case with CNNs, it is not always obvious without detailed and time-consuming analysis which of them are contributing to the increase or decrease in the validation error. Why do we not get a machine to do the hard work and find the patterns for us?

Often several hundreds or even thousands of random search trials are required to find an acceptable model. For each trial we perform we know the hyperparameter values and the corresponding validation error. This data presents the opportunity to apply machine learning to the following problem: given the hyperparameter values, predict the error that the resultant CNN would achieve on the validation set if it were trained for real.

This is the approach we investigate by training a random forest [27] to approximate the relationship between the CNN hyperparameter values and the validation error. The trained random forest is used to indicate good hyperparameter values and the resultant CNNs are then trained and validated for real. We call this method *machine learning assisted search*, or *ML-assisted search* for short.

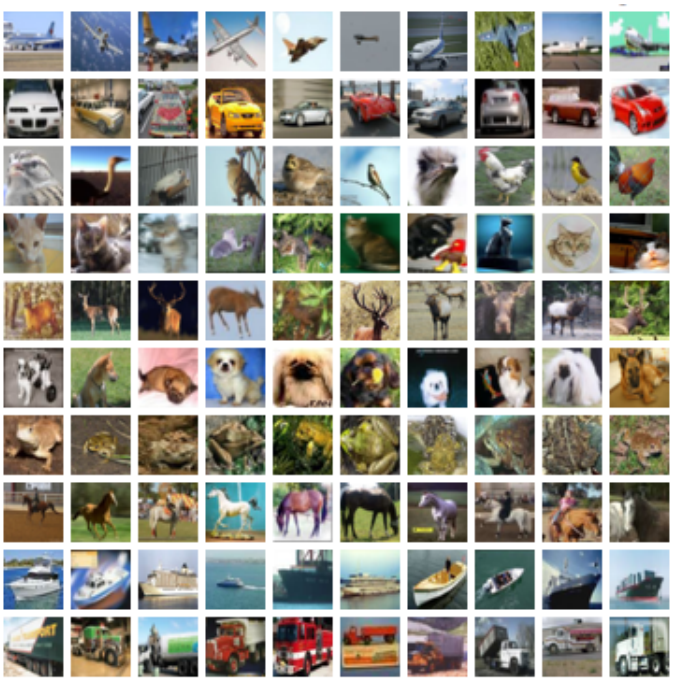
Using the famous CIFAR-10 image dataset [16] to run experiments, we show that the best model found during several hundred random search trials can be beaten by subsequently performing only a few iterations of ML-assisted search, and by allowing ML-assisted search to run for the same number of iterations as random search we can find significantly better models. We demonstrate that this method can work without the need to tune the random forest used to indicate good hyperparameter values, thus adding no extra manual effort to the process. We propose this completely automated approach to CNN hyperparameter optimisation as an alternative to a combination of random and thoughtful manual search.

We begin in Section 2 by describing the CIFAR-10 image dataset. We then provide a brief introduction to training regular classification neural networks in Section 3, before we illustrate in Section 4 why CNNs are preferred for image classification tasks and explain the function of each of the most common CNN layers. In Section 5 we describe the general architecture pattern on which the CNNs we train are based, and detail the hyperparameters associated with the architecture, as well as the training hyperparameters in Section 6. We then define the domain of each hyperparameter in Section 7 and outline the process of performing 512 random search trials. In Section 8 we explain in more detail how we use the results from random search to run 512 ML-assisted search trials. We present and compare the results of both methods in Section 9, and finish with a short discussion in Section 10.

2 The data

To run experiments we use the CIFAR-10 image dataset [16], which consists of 60,000 32×32 colour images from the following ten categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. A collection of ten sample images from each category is shown in Figure 1. Each image is represented by a $32 \times 32 \times 3$ numerical array, where the first two dimensions are the number of pixels

Figure 1: Sample images from CIFAR-10 dataset, with categories arranged in rows.



in width and height in each of the three grids which specify the red, green and blue (RGB) values for each pixel.

The data is split into 50,000 training and 10,000 test images. The training data consists of 5000 images from each category, partitioned randomly into five batches of 10,000 images. We use four of the batches for training and keep one for validation. That is, we train each model using the same 40,000 images and validate on the remaining 10,000. The test batch contains exactly 1000 images from each category and is used to compute the test error for each model once training and validation are complete.

The dataset was chosen due to its widespread use in academic research for evaluating image classification models. Also, the relatively small size of the images means less computation time, thus allowing us to run a large number of experiments.

3 Neural Networks

A regular classification neural network consists of an *input layer* which accepts a numerical feature vector, one or more *hidden layers* which perform mathematical operations on the input, and an *output layer* which returns class scores for the input [23]. Layers are represented as collections of *neurons* which hold numerical values called *activations*. The activations in the input layer are simply the values of the input feature vector $[x_1, x_2, \ldots, x_n]^\intercal$. The activation $a_i^{(1)}$ of the i-th neuron in the first hidden layer is computed by multiplying each x_j by a weight $w_{i,j}^{(1)}$, summing, adding a bias term $b_i^{(1)}$ and finally applying some non-linear function f. That is,

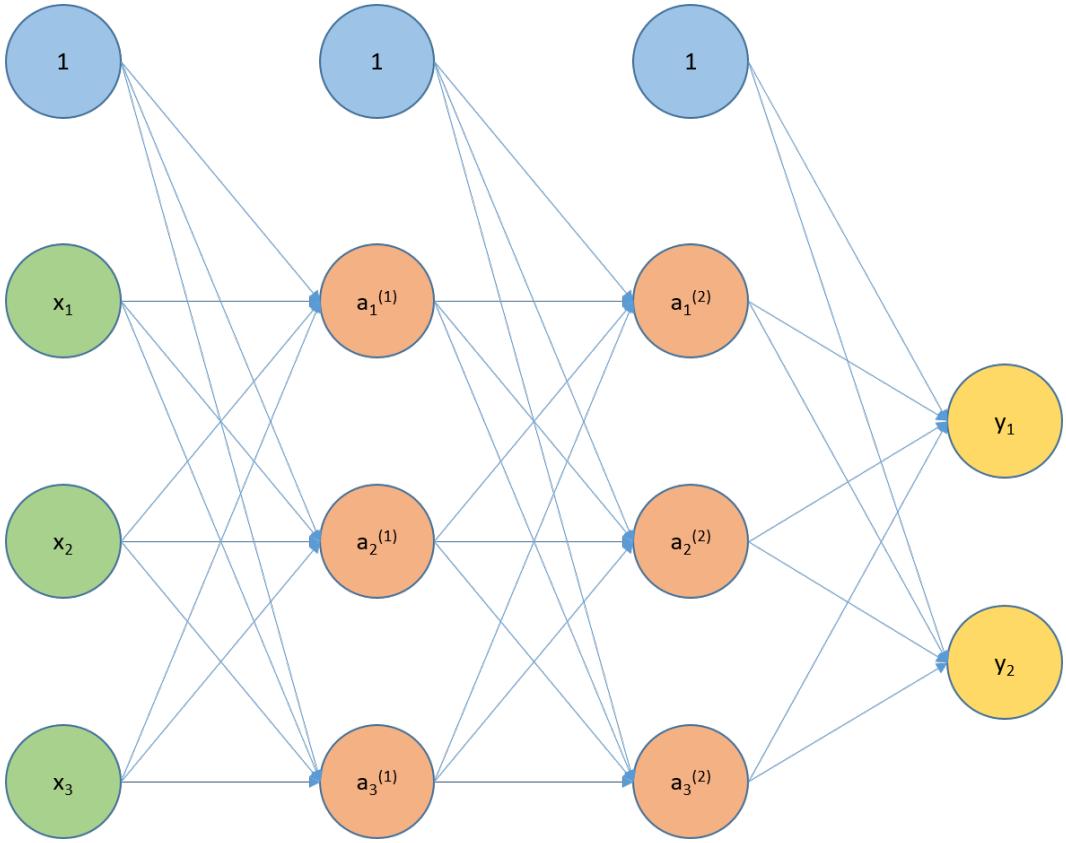
$$a_i^{(1)} = f\Big(\sum_{j=1}^{n} w_{i,j}^{(1)} x_j + b_i^{(1)}\Big). \tag{3.1}$$

The function f is known as the *activation function*, which we will discuss in more depth later.

Through this operation, all neurons in the input layer are said to be *connected* to the i-th neuron in the first hidden layer. Generally, in regular neural networks all layers are *fully-connected*, which means that every neuron in a layer is connected to every neuron in the next layer, but neurons within the same layer share no connections.

Whereas the number of neurons in the input and output layers is fixed by the data, the number in each hidden layer is a hyperparameter of the network. Suppose there are k neurons in the first hidden

Figure 2: Fully-connected neural network with two hidden layers.



layer. Once all activations $a_1^{(1)}, a_2^{(1)}, \ldots, a_k^{(1)}$ of the first hidden layer have been computed, the activation $a_i^{(2)}$ of the i-th neuron in the next layer is given by

$$a_i^{(2)} = f\Big(\sum_{j=1}^{k} w_{i,j}^{(2)} a_j^{(1)} + b_i^{(2)}\Big),$$

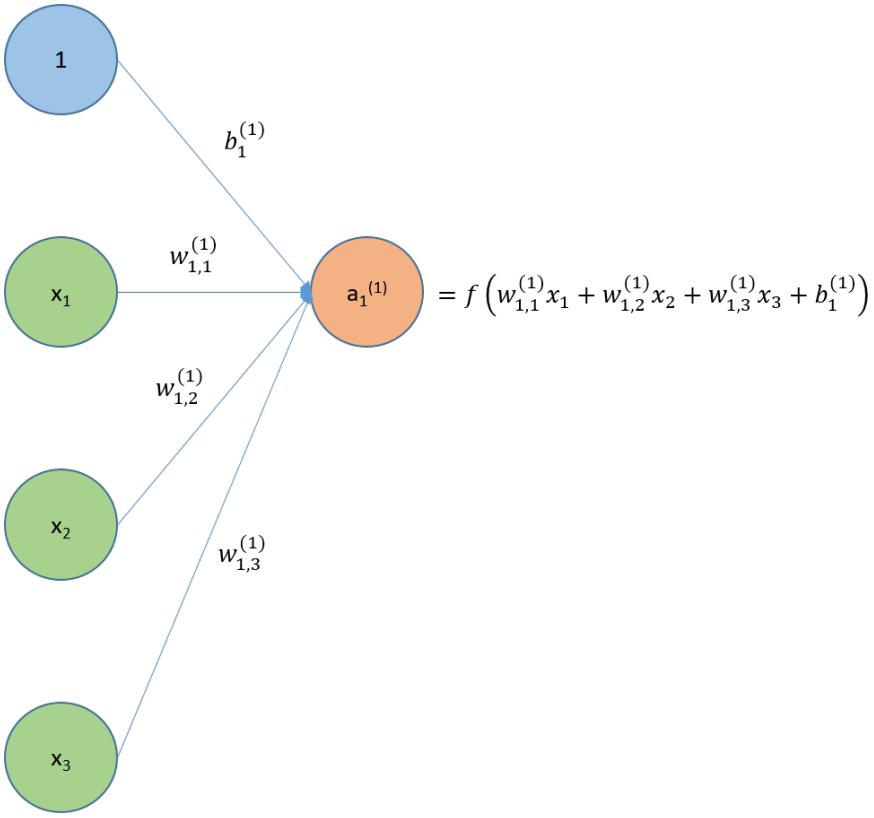
where each $w_{i,j}^{(2)}$ is a weight and $b_i^{(2)}$ a bias term. This process continues through the layers until the activations in the output layer are computed. The number of neurons in the output layer is equal to the number of unique classes in the data, and the activations correspond to the class scores. The greater the score the more the network believes the input belongs to that class. It is common to "squash" the activations in the output layer onto the interval $(0, 1)$ using the *sigmoid* activation function

$$s(t) = \frac{1}{1 + \exp^{-t}}.$$

The resultant values can be interpreted as the predicted probabilities that the input belongs to each class.

Figure 2 illustrates the architecture of a fully-connected classification neural network with two hidden layers. There are three neurons in the input layer and both hidden layers, plus bias terms shown in blue, and two neurons in the output layer. This is an example of a network which could be trained to classify observations from a dataset containing two unique categories, where each observation is a numerical feature vector $[x_1, x_2, x_3]^\intercal$ of length three. The network computes activations $[a_1^{(1)}, a_2^{(1)}, a_3^{(1)}]^\intercal$ in the first hidden layer and $[a_1^{(2)}, a_2^{(2)}, a_3^{(2)}]^\intercal$ in the second hidden layer, and outputs class scores $[y_1, y_2]^\intercal$. The arrows indicate connections between neurons, and the strength of each connection is determined by the corresponding network weight. Figure 3 illustrates how the weights and bias term are used with Equation 3.1 to compute the activation of the first neuron in the first hidden layer.

Figure 3: Computation of activation of first neuron in first hidden layer.



3.1 Updating weights via gradient descent

The process described above from input to output is called the *forward-pass*, and it is the first stage of training a neural network. The next step is to measure the error, or *loss*, between the true class of the input and the class probabilities that the network predicts. A common error metric for multi-class classification, and the one which we use, is multi-class log loss, also known as cross-entropy loss. Suppose there are M unique classes in our data, and for a particular example the network outputs class probabilities $p_1, p_2, \ldots, p_M$. The log loss L for this single example is given by

$$L = -\sum_{j=1}^{M} y_j \log(p_j), \tag{3.2}$$

where y_j is a binary variable indicating whether the example belongs to class j.

The only non-zero term in Equation 3.2 is $-y_c \log(p_c) = -\log(p_c)$, where c is the correct class. Intuitively, $-\log(p_c)$ is close to zero for p_c close to one, and tends to ∞ as p_c approaches zero. Thus, if the network predicts that the example belongs to the correct class with a probability close to one the loss will be low, but as that probability drops towards zero the loss increases logarithmically.

The goal is to minimise the loss, which is a function of the weights (including the bias terms). The weights are usually initialised randomly and as a result the initial loss is high. After performing a forward-pass through the network with a single example and calculating the loss, we use a technique called *back-propagation* [4] to compute the gradient of the loss function with respect to each of the weights. This tells us, for each of the weights, in which direction we need to move to decrease the loss. If we update each weight in the direction of the corresponding negative gradient and perform the forward-pass again the loss should be less than before. We repeat this process for every example in our training set, usually multiple times. This optimisation technique is known as *stochastic gradient descent* [5].

3.2 Learning rate

In the simplest version of stochastic gradient descent each weight is updated by subtracting the actual value of the gradient of the loss function with respect to that weight. However, it is more common to update using a positive step size other than one. That is,

$$w \leftarrow w - \sigma \frac{\partial L}{\partial w}, \tag{3.3}$$

where $\frac{\partial L}{\partial w}$ is the partial derivative of the loss function L with respect to the weight w and $\sigma > 0$ is called the *learning rate* [2]. The value of the learning rate is an important hyperparameter. If we set it too large we can "overshoot" good local minima of the loss function (there may be more than one) and end up actually increasing the loss, whereas if we set it too small the weights barely change each time we perform an update and convergence takes too long.

3.3 Mini-batch gradient descent

During stochastic gradient descent the network weights are updated based on the loss of a single training example at a time. In practice it is common to update the weights after computing the average loss over a larger subset of the training data. For example, the average loss over N examples is given by

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \log (p_{i,j}),$$

where M is the number of classes, $y_{i,j}$ is a binary variable indicating whether example i belongs to class j and $p_{i,j}$ is the probability assigned by the network that example i belongs to class j. This is called *mini-batch gradient descent* [5] and we do it because the forward-pass and back propagation steps can be vectorised for multiple examples, which is significantly more computationally efficient than doing one at a time. The batch size is usually set to some power of two to further increase the speed of vectorised operations. Common values are 32, 64, 128 or 256, but the choice is not believed to have a significant impact on performance and is more dependent on memory available during computation [1]. We train our networks with examples in batches of size 128.

We cycle through the full training set and repeat mini-batch gradient descent until we have used every single example exactly once to compute the loss and update the weights. This is a single *epoch*. In order for the loss function to converge we often have to run several hundred or more epochs, but if we run too many we can overfit the training data and end up with a model which does not generalise well to new data. The number of training epochs is an important hyperparameter.

3.4 Learning rate decay

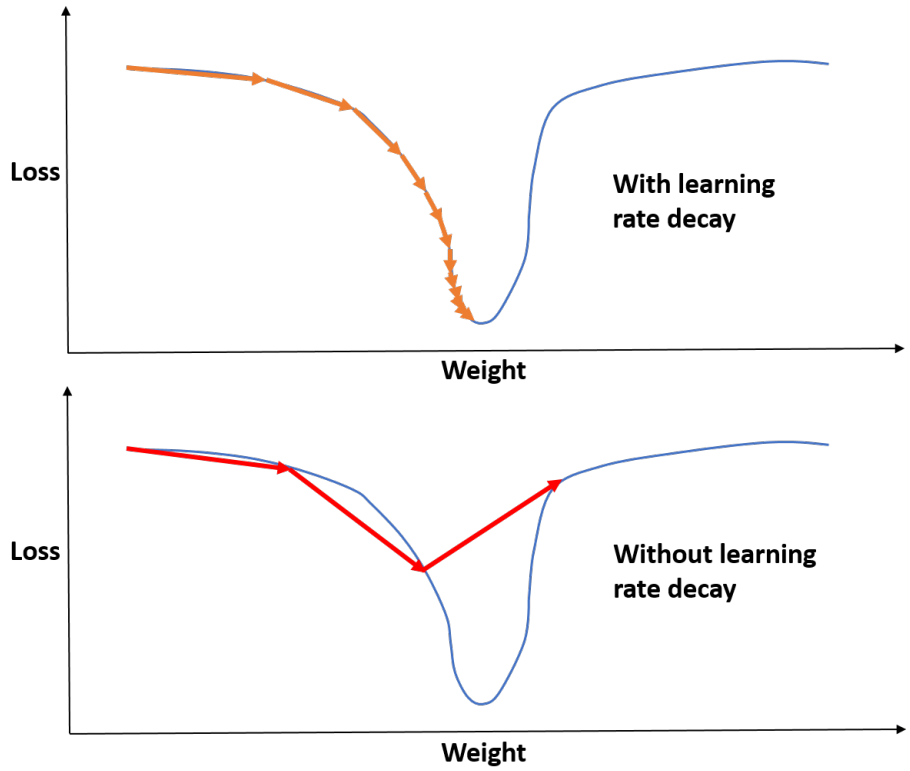
During training, as the number of epochs increases it is important to decrease the learning rate. To begin with we want a larger step size so that we move quickly towards a region containing a good minimum of the loss function. As we approach that minimum the chance of overshooting increases if we take too big a step in the direction of the negative gradient. We therefore take smaller and smaller steps by annealing the learning rate. We control this using a parameter $\epsilon > 0$ called the *learning rate decay*. After t epochs the learning rate σ_t is given by

$$\sigma_t = \frac{\sigma_0}{1 + t\epsilon},$$

where σ_0 is the initial learning rate [2].

The top diagram in Figure 4 illustrates the desired effect of learning rate decay, whereas the bottom diagram shows what can go wrong if a large constant learning rate is used.

Figure 4: Top diagram shows the learning rate gradually decreasing, leading to convergence at a minimum of the loss function, whereas in the bottom diagram a large constant learning rate results in overshooting the minimum.



3.5 Momentum

The parameter update in Equation 3.3 uses the gradient of the loss function computed on the last batch of training examples only. The batch gradient can be noisy and a running average is often used to smooth out the gradient samples [17]. Following this approach, for a fixed learning rate σ, the value of the k-th parameter update v_k is given by the recursive relationship

$$v_k = \mu v_{k-1} - \sigma\Big(\frac{\partial L}{\partial w}\Big)_k,$$

where v_0 is set to 0, $(\frac{\partial L}{\partial w})_k$ denotes the gradient of the loss function computed on the k-th batch and $\mu \in (0, 1)$ is called the *momentum* [6]. Then $w \leftarrow w + v_k$.

Observe the first few expansions of v_k for $k \geq 1$:

$$\begin{aligned}
v_1 &= -\sigma\Big(\frac{\partial L}{\partial w}\Big)_1, \\
v_2 &= \mu v_1 - \sigma\Big(\frac{\partial L}{\partial w}\Big)_2 = -\sigma\bigg(\Big(\frac{\partial L}{\partial w}\Big)_2 + \mu\Big(\frac{\partial L}{\partial w}\Big)_1\bigg), \\
v_3 &= \mu v_2 - \sigma\Big(\frac{\partial L}{\partial w}\Big)_3 = -\sigma\bigg(\Big(\frac{\partial L}{\partial w}\Big)_3 + \mu\Big(\frac{\partial L}{\partial w}\Big)_2 + \mu^2\Big(\frac{\partial L}{\partial w}\Big)_1\bigg), \\
v_4 &= \mu v_3 - \sigma\Big(\frac{\partial L}{\partial w}\Big)_4 = -\sigma\bigg(\Big(\frac{\partial L}{\partial w}\Big)_4 + \mu\Big(\frac{\partial L}{\partial w}\Big)_3 + \mu^2\Big(\frac{\partial L}{\partial w}\Big)_2 + \mu^3\Big(\frac{\partial L}{\partial w}\Big)_1\bigg).
\end{aligned}$$

We see that in each case the term in the large brackets being multiplied by the learning rate is a weighted average of all the batch gradients computed up to that point. As $0 < \mu < 1$, the term μ^k decreases as k increases, which has the nice effect of giving most weight to the gradient just computed on the last batch, and the contribution of other batch gradients decreases as we go further and further back. Although we used a constant learning rate σ to simplify the expansions above, the effect is similar with learning rate decay.

3.6 Nesterov momentum

The parameter update with momentum

$$w \leftarrow w + \mu v_{k-1} - \sigma_t \Big(\frac{\partial L}{\partial w}\Big)_k$$

can be split into two parts:

$$w \leftarrow w + \mu v_{k-1}$$

followed by

$$w \leftarrow w - \sigma_t \Big(\frac{\partial L}{\partial w}\Big)_k.$$

As written here, we first update w by adding the term μv_{k-1}, and only then do we compute the batch gradient of the loss function with respect to w before performing the second part of the update. This technique, knows as *Nesterov momentum* [18], is a slightly different version of the standard momentum update where we compute the batch gradient before modifying w in any way. The use of Nesterov momentum has been gaining popularity and in practice usually leads to slightly better results than standard momentum.

4 Convolutional neural networks

In a regular neural network each neuron in a hidden layer is connected to every single neuron in the layer below. If we were to use a regular neural network to classify images from the CIFAR-10 dataset, every neuron in the first hidden layer would require $32 \times 32 \times 3 + 1 = 3073$ weights (including the bias term) to compute its activation from the raw pixel values [3]. Considering that it is common practice to have as many neuron in the first hidden layer as the input layer, we would require in the region of $3072 \times 3073 = 9,440,256$ weights for just one layer. Moreover, the CIFAR-10 images are relatively small in comparison to typical datasets of images with width and height greater than 200. In such cases the use of fully-connected layers to transform the input pixel values becomes impractical. Convolutional neural networks provide an alternative solution by taking advantage of the spacial structure of images.

4.1 Input layer

The input layer holds the raw pixel values of the image. Unlike in a regular neural network where the input values are stacked in a single dimension, the neurons in the input layer of a CNN are arranged in three dimensions with width, height and depth to match the RGB structure of the images [3]. Thus, the size of the input layer in a CNN trained on the CIFAR-10 images must be $32 \times 32 \times 3$.

4.2 Convolutional layer

We connect every neuron in the first hidden layer to only a small local region of the image input layer in width and height. This type of layer is called a *convolutional* layer and the local transformations are known as *filters* [3]. A filter is essentially a set of weights which computes activations by performing a dot product with neurons in local regions of equal size in the previous layer. Each of these local regions is connected to a unique neuron in the convolutional layer, and the dot product with the filter at that region plus a bias term gives the neuron's activation. Typical filter sizes in width and height are 3×3 and 5×5, but the filters always span the complete depth of the previous layer. As the filter width and height are always equal we use the term *filter size* to refer to both.

For example, a 3×3 filter operating on the image input layer would have $3 \times 3 \times 3 + 1 = 28$ weights, including the bias term. The filter starts at the top left corner of the image and computes the dot product of its weights with the RGB pixel values in the first three rows and three columns. This dot product plus the bias term gives the activation of the neuron in the first row and first column of the convolutional layer. The filter then slides to the right by some fixed number of pixels called the *stride*, which is usually set to either one or two, computes the dot product with the next local region

and adds the same bias term, giving the activation for the neuron in the first row and second column of the convolutional layer. This continues until the filter reaches the top right corner of the image, after which it moves back to the first column but slides downwards by the same stride. This process of sliding across rows one at a time is repeated until the filter reaches the bottom right corner of the image, having computed the dot product of its weights with the RGB pixel values at every 3×3 local region of the image [3].

For our experiments we always use a stride of one as this is the recommended setting so as not to throw away information, especially for relatively small images like those in the CIFAR-10 dataset [3]. For the same reason we always use filters of size three.

Suppose that W is the width and height of the image. It is easy to see that as a filter of size three slides from left to right with a stride of one it stops at $W - 3 + 1 = W - 2$ local regions. Due to the width of the filter its right-most weights start at column three of the image and move one pixel at a time until they reach column W, giving $W - 3 + 1$ unique positions. Similarly, as the filter slides from top to bottom it stops at $W - 2$ local regions. Hence, the filter computes activations for $(W - 2)^2$ neurons arranged in a $(W - 2) \times (W - 2)$ grid in the convolutional layer.

It is recommended that convolutional layers preserve the width and height of the layer below [3]. In order to achieve this we use *zero-padding*, which is the process of inserting zeros around the edges of the spacial dimensions of the previous layer. In the example above, if we pad the right, left, top and bottom of the RGB pixel grids each with a single row of zeros, then we have an extra two rows and columns at each depth and the filter fits in each row and column of the image a further two times. This results in a $W \times W$ grid of neurons, the same as in the previous layer.

Figure 5 illustrates this idea for a 5×5 RGB image. Each depth slice of the original pixel array is padded with zeros, shown as grey dots, to give three 7×7 grids. We denote the pixel value at row i, column j and depth k by $x_{i,j,k}$. The diagram shows how the neurons in the first three rows and three columns at each depth are connected to the first neuron in the convolutional layer. The three depth slices of the filter are shown in yellow, and we denote the weight at row i, column j and depth k of the filter by $w_{i,j,k}$. There is also a bias term b show in purple. The activation of the neuron in the first row and first column of the convolutional layer is given by

$$\sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} x_{i,j,k} \cdot w_{i,j,k} + b.$$

The activations corresponding to the remaining twenty-four local regions of the zero-padded input layer are computed similarly, using the same filter and bias term, resulting in a 5×5 grid of neurons in the convolutional layer.

The general formula for calculating the width and height of the neuron grids in a convolutional layers is

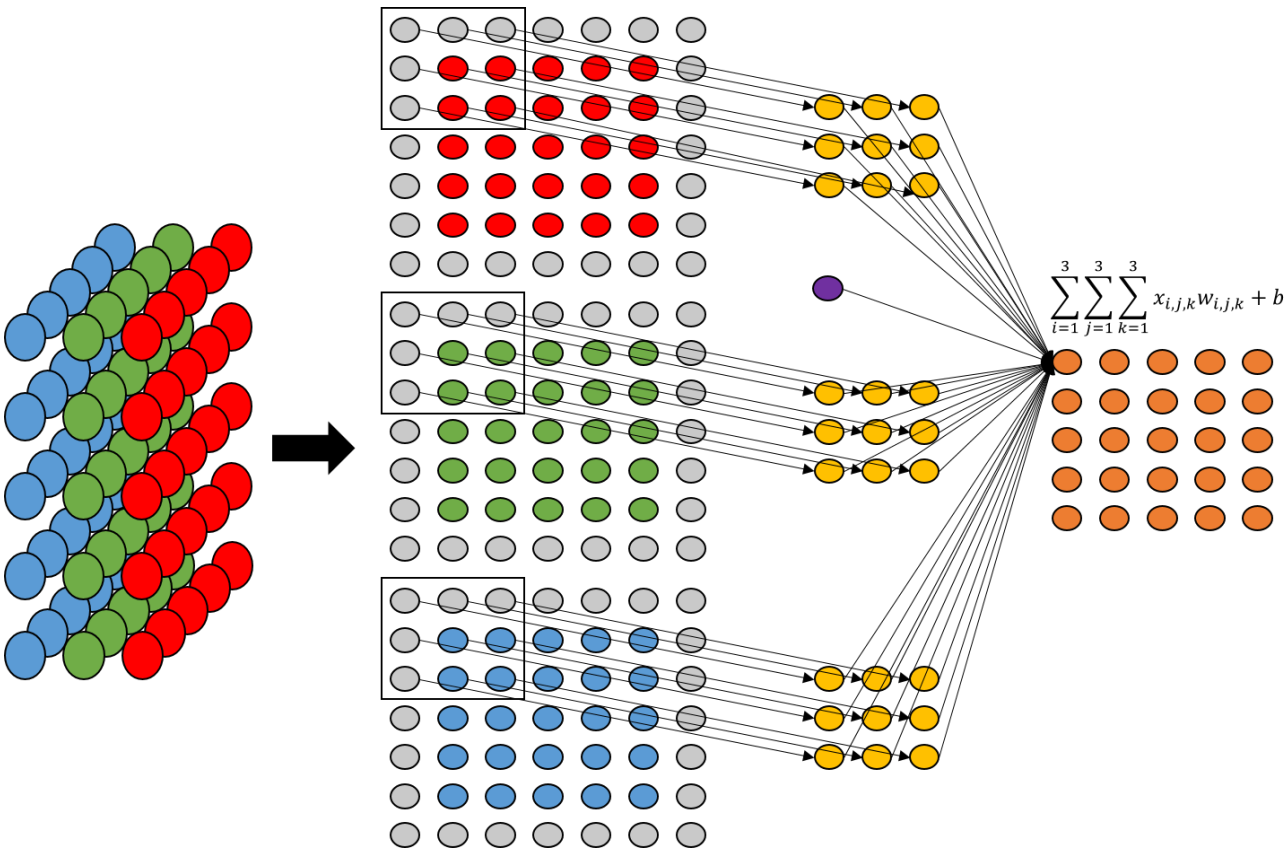
$$\frac{W - F + 2P}{S} + 1 \tag{4.1}$$

where W is the width and height of the previous layer, F is the filter size, S is the stride and P is the number of rows of zero-padding around the edges of each spacial dimension [3]. As we always set $F = 3$, $S = 1$ and $P = 1$, our convolutional filters always produce $W \times W$ grids of neurons.

In a convolutional layer we use multiple filters, and the exact number we choose is a hyperparameter. Each filter goes through the same process of sliding over every local region in the previous layer and computing dot products. However, as each set of weights is initialised randomly, the activations computed by each filter are different. We can think of the filters as looking at the same local regions in a different way, thus extracting different features. If K is the number of filters then the resulting convolution layer is of size $W \times W \times K$, one $W \times W$ grid of neurons for each filter.

In this section we have focused our discussion on a convolutional layer acting on the image input layer. However, deep CNNs contain multiple convolutional layers. In each one the process of computing activations is almost exactly the same, with the only difference being that the depth of each local region in the previous layer, and hence the depth of the filters, can be any positive integer as opposed to just three for RGB images.

Figure 5: Computation of activation in first row and first column of convolutional layer.



4.3 ReLU layer

The operations performed by convolutional layers are completely linear, as are those performed in fully-connected layers. In order for a CNN to be able to learn complex non-linear relationships between the inputs and outputs we add an *activation* layer after every convolutional and fully-connected layer.

The activation layer applies a non-linear function to the output of each individual neuron in the previous layer. Note that this leaves the shape of the previous layer unchanged. There are many activation functions to choose from, but currently the most popular for CNNs and the one we use is know as the *rectified linear unit* (ReLU) [9]. ReLU is a very simple non-linear function which sets negative values to zero and leaves all other values unchanged. Formally, the function $f : \mathbb{R} \to (0, \infty)$ is defined by

$$f(x) = \max(0, x).$$

4.4 Max pooling layer

As discussed in Section 4.2, the convolutional layers in our networks do not alter the width and height of the layers immediately before them. However, it is standard practice to progressively increase the number of filters in convolutional layers, and hence increase layer depth. As depth increases the number of weights in the network also increases, which requires greater computation and increases the risk of overfitting. To avoid this we offset the increase in depth by reducing layer width and height using *max pooling* layers [3].

In a layer of size $W \times W \times K$ the max pooling layer operates independently on each of the K grids. In our network we use max pooling layers with filters of size 2×2, applied with a stride of two and no zero-padding. Similar to what happens in convolutional layers, the max pooling filter slides over the neurons in the previous layer, however it only operates on a single depth slice at a time. The filter starts at the top left corner of the first $W \times W$ grid and each time it stops it computes the maximum of the four neuron activations in the 2×2 square that it covers. The maximum values computed on a single grid give the activations in the corresponding depth slice of the max pooling layer. Once it has finished operating on the top grid it moves down to the next and continues the process [3].

Figure 6: Max pooling filter of size 2×2 operating with a stride of two on a single 6×6 neuron grid.

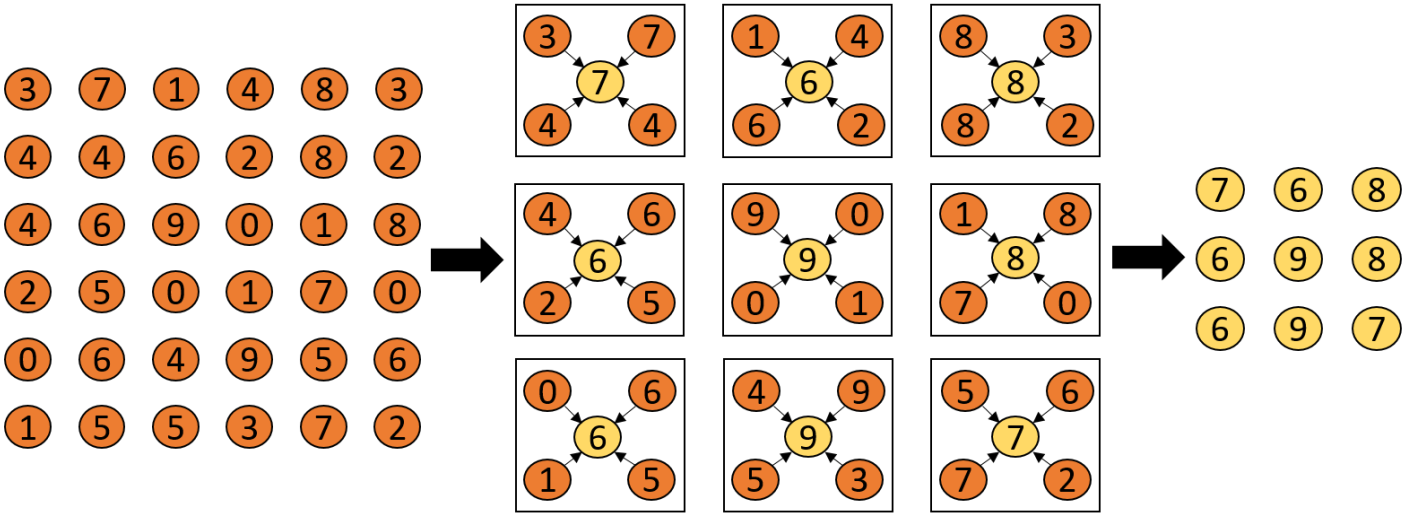


Figure 6 illustrates the idea of a 2×2 max pooling filter operating on a single depth slice of size 6×6 with a stride of two. The filter fits in every row and column three times, producing a 3×3 grid of neurons.

More generally, when operating on a layer with spacial dimensions $W \times W$ using a stride of two with no zero-padding, Equation 4.1 tells us that the resultant width and height of the max pooling layer is $(W - 2)/2 + 1$ (note this requires that W be even, which is always the case in the networks we train). For example, after passing a CIFAR-10 $32 \times 32 \times 3$ image through a convolution and ReLU layer, leaving its spacial dimensions unchanged, we could use a max pooling layer to reduce the width and height to $(32 - 2)/2 + 1 = 16$. This decreases the number of neurons in each depth slice from $32 \times 32 = 1024$ to $16 \times 16 = 256$, a reduction of 75%. This allows us to increase the number of filters in subsequent convolutional layers while progressively reducing the overall number of neurons.

4.5 Dropout layer

Using *dropout* layers is a very simple and effective way to combat overfitting in neural networks [10]. When inserted into a CNN, each time a forward pass is performed through the network a dropout layer leaves the activation of each neuron in the previous layer unchanged with probability p, or otherwise sets it to zero. The value of p is a hyperparameter which determines the degree of regularisation. The smaller p is the more neurons will be set to zero. Less non-zero activations means less chance of overfitting, but if p becomes too small we risk underfitting.

4.6 Fully-connected layer

As illustrated at the beginning of Section 4, it is not a good idea to employ fully-connected layers at the beginning of our network. However, as we progressively reduce the number of weights with max pooling layers, using one or more fully-connected layers as a final processing step becomes more practical.

The network always ends with a fully-connected layer where the number of neurons is equal to the number of unique classes in the dataset. For the CIFAR-10 dataset this number is ten. The activations in these final neurons are "squashed" onto the interval $(0, 1)$ using the sigmoid activation function. The resultant values represent the probabilities that the image belongs to each of the ten classes.

5 Network architecture

In CNNs the convolutional and fully-connected layers are always immediately followed by an activation layer. As discussed, ReLU is currently the most popular choice of activation function, except after the

final fully-connected layer when the sigmoid activation is used. The architecture usually begin with between one and three convolutional layers followed by a max pooling layer. This pattern is often repeated several times before transitioning to one or more fully-connected layers, the last of which computes the class score for each unique image category. Dropout layers are inserted at various points in the network to combat overfitting. The architectures of the networks we train are based on the following version of this general pattern:

`INPUT` $\rightarrow$ `[[CONV` $\rightarrow$ `ReLU]*M` $\rightarrow$ `POOL` $\rightarrow$ `DROPOUT]*2` $\rightarrow$ `[FC` $\rightarrow$ `ReLU]*N` $\rightarrow$ `DROPOUT` $\rightarrow$ `FC` $\rightarrow$ `OUTPUT`,

where `*` indicates repetition and M and N are positive integers [3]. The specific details of the architecture are hyperparameters.

5.1 Architecture hyperparameters

Number of convolutional layers. Determined by the value of M in the architecture, which defines the number of `[CONV` $\rightarrow$ `ReLU]` layers in the network. We stipulate that $M \in \{1, 2\}$, and it follows that the networks have either two or four convolutional layers. Due to the relatively small size of the CIFAR-10 images we felt that this level of processing would be sufficient to achieve a decent level of accuracy. Also, more layers means more computation, which would eventually have made running lots of experiments impractical in the limited time available.

Number of convolutional filters. Recall from Section 4.2 that in convolutional layers we always use 3×3 filters applied with a stride of one. The number of such filters in each layer is a hyperparameter. Our architecture has two `[[CONV` $\rightarrow$ `ReLU]*M` $\rightarrow$ `POOL` $\rightarrow$ `DROPOUT]` blocks, where each block contains either one or two convolutional layers, depending on the value of M. If $M = 2$, then copying the architecture of VGGNet [14], we stipulate that the number of filters in both convolutional layers in each such block must be the same. We choose the number of filters in the first block's layer(s) from the interval $[32, 128]$. As it is common to progressively increase the number of filters, we set the number of filters in the second block's layer(s) to be a multiple between one and three of the number in the first block. It follows that the convolutional layer(s) in the second block can have a minimum of 32 and a maximum of 384 filters.

Number of fully-connected layers. Determined by the value of N in the architecture, which defines the number of `[FC` $\rightarrow$ `ReLU]` layers in the network. We stipulate that $N \in \{1, 2\}$.

Number of neurons in fully-connected layers. Copying the approach of AlexNet [12], if $N = 2$ then we set the number of neurons to be the same in both `[FC` $\rightarrow$ `ReLU]` layers. The value is chosen from the interval $[256, 4096]$

Dropout. There are three dropout layers in our networks. To simplify things slightly we choose only two unique dropout values: one for the two dropout layers in the `[[CONV` $\rightarrow$ `ReLU]*M` $\rightarrow$ `POOL` $\rightarrow$ `DROPOUT]*2` block, and another for the dropout layer immediately before the final fully-connected layer. Both values are chosen from the interval $(0, 1)$

6 Training hyperparameters

There are various optimisers which can be used to train a neural network, and each one usually has two or three important hyperparameters that require tuning. The optimiser itself is a hyperparameter, but to simplify the task significantly we made the decision to use only mini-batch gradient descent with momentum, as outlined in Section 3.

Recall that in mini-batch gradient descent with momentum, for batch k and epoch t, a network weight w is updated as follows:

$$w \leftarrow w + v_k,$$

where

$$v_0 = 0, \quad v_k = \mu v_{k-1} - \sigma_t \left(\frac{\partial L}{\partial w} \right)_k$$

and

$$\sigma_t = \frac{\sigma_0}{1 + t\epsilon}.$$

The hyperparameters associated with this optimisation method are the initial learning rate σ_0, the learning rate decay ϵ, the momentum μ, and whether the update is performed using either standard or Nesterov momentum.

We also must choose how many epochs to run the training process for. That is, how many times we cycle through the full training set performing mini-batch gradient descent to update the network weights. Thankfully, we can optimise this hyperparameter "for free" by monitoring the validation loss of the network after every epoch. If after a set number of epochs, which we call the *patience*, the validation loss does not decrease, then we stop training. For our trials we set the patience to ten, and hence stop training if after epoch t the smallest validation loss was observed after epoch $t - 10$. The patience value is a matter of preference, but we felt that for this particular task ten was a good balance between avoiding stopping too early and wasting computational time on fruitless extra epochs.

7 Random hyperparameter search

For random hyperparameter search we define a distribution of values for each hyperparameter. These distributions can be continuous, discrete or, as in grid search, a list of values. We specify how many trials we want to run, and for each one we select the hyperparameter values by sampling from the distributions.

When the distribution is a list of values we sample from the list uniformly. We can also sample uniformly from continuous distributions, but for many hyperparameters it is preferable to draw samples *exponentially*. That is, if the original hyperparameter domain is the interval $[a, b]$, we sample uniformly between $\log(a)$ and $\log(b)$ and then exponentiate to get back a value between a and b. If the hyperparameter must take an integer value, such as the number of convolutional filters, we also have to round to the nearest integer.

We do this because many hyperparameters have multiplicative effects [2]. For example, a reasonable search domain for the initial learning rate could be between 1×10^{-5} and 10. Because the learning rate multiplies the gradient during the parameter update, adding 0.001 to go from a learning rate of 0.0001 to 0.0011 clearly has a far greater effect than going from 1 to 1.001. In such cases, sampling exponentially leads to us testing a more diverse set of models.

Table 1 shows the domain and sampling method for each of the eleven hyperparameters associated with our network architecture and training optimiser. We ran a total of 512 trials, each time sampling hyperparameter values from those domains and training for a maximum of 100 epochs, stopping after epoch t if the smallest validation loss was observed after epoch $t - 10$.

We could almost certainly achieve significantly better performance by training for many more epochs. For example, during their random search trials Bergstra and Bengio trained networks for a *minimum* of 100 and a maximum of 1000 epochs, while also employing a version of early stopping [22]. However, given the limited time and computing resources available and our desire to run a fairly large number of trials, permitting more than 100 epochs was impractical.

After training each model we computed the loss on the validation and test sets. Although strictly speaking the test set should be "locked away" until training and validation of all models is complete, it was more convenient to compute and save the test loss immediately after training each model and the results were only observed once both random and ML-assisted search were completely finished.

The hyperparameter domains were chosen to be fairly wide and are typical of coarse domains used as a starting point when beginning a combination of random and thoughtful manual search.

Hyperparameter	Domain	Sampling Method
Number of convolutional layers	$\{2, 4\}$	Uniformly
Number of convolutional filters in first block's layers	$[32, 128]$	Exponentially
Ratio between number of filters in second and first blocks	$[1, 3]$	Uniformly
Number of fully-connected layers	$\{1, 2\}$	Uniformly
Number of neurons in fully-connected layer(s)	$[256, 4096]$	Exponentially
Dropout after convolutional layers	$(0, 1)$	Uniformly
Dropout after fully-connected layer(s)	$(0, 1)$	Uniformly
Initial learning rate	$[1 \times 10^{-5}, 1]$	Exponentially
Learning rate decay	$[1 \times 10^{-5}, 1]$	Exponentially
Momentum	$\{0.5, 0.9, 0.95, 0.99\}$	Uniformly
Nesterov Momentum	$\{True, False\}$	Uniformly

Table 1: Hyperparameter domains and sampling methods.

8 ML-assisted hyperparameter search

After performing the random search trials we had 512 sets of hyperparameter values and their corresponding validation scores. Our ML-assisted search approach was to use this data to train a random forest to approximate the relationship between the hyperparameter values and the validation loss. Random forests are often considered the closest thing to a "black box" machine learning model [24]. That is, a model which works well on many diverse datasets with very little hyperparameter tuning. This is important since we are attempting to simplify the hyperparameter optimisation process, and clearly introducing another model which itself requires extensive tuning would only make things worse.

In fact, to demonstrate that this approach can work with no extra manual effort, we decided not to tune the random forest at all. We defined our model using the `RandomForestRegressor` class from the Python machine learning library scikit-learn [25], setting the number of trees to 500 and the minimum leaf size to 5, and leaving all other hyperparameters with their default values. For random forests more trees is generally considered better with diminishing returns and a minimum leaf size of 5 is standard for regression tasks [24]. No validation was performed on this or any other model.

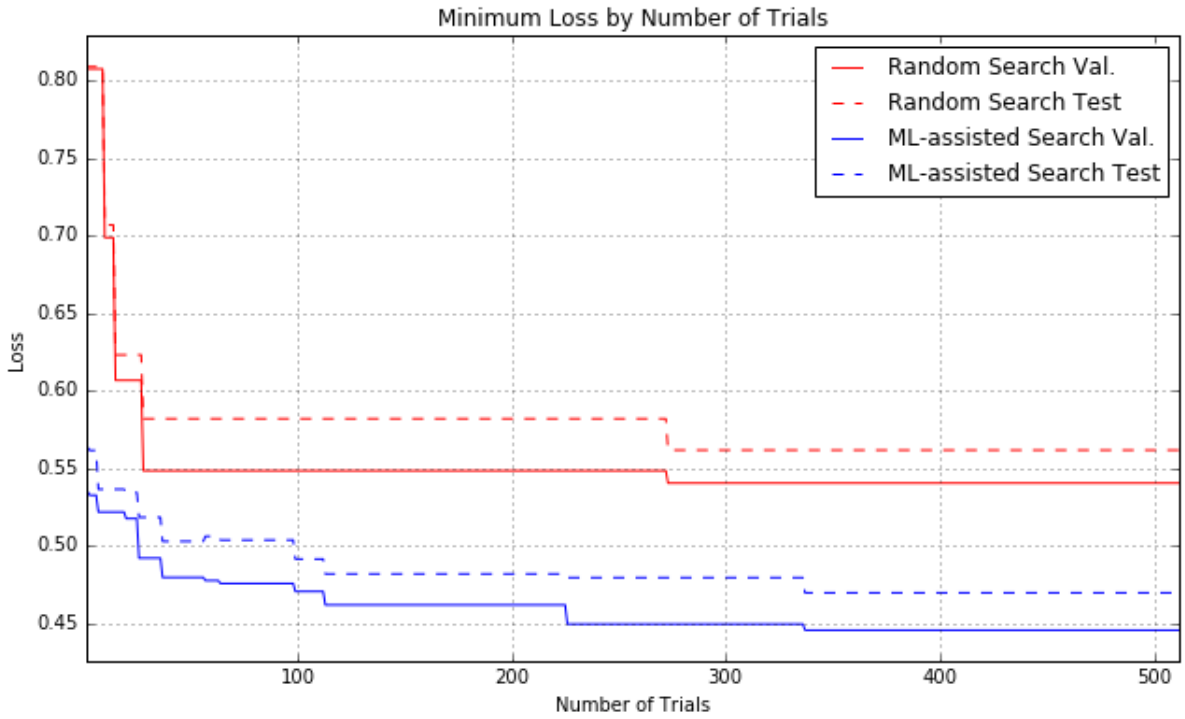
We trained the random forest with the 512 examples from random search, using the hyperparameter sets as features and the corresponding validation losses as targets. Then, again using the domains and sampling methods defined in Table 1, we drew 1,000,000 sets of hyperparameters. For each set we used the random forest to predict the validation loss that the resultant CNN would achieve on the CIFAR-10 images, provided it was trained in the same way as the models built during random search. This process of training the random forest and making 1,000,000 predictions takes less than a minute.

We ordered the 1,000,000 examples in ascending order by predicted validation loss and kept only the best eight. For each of these eight hyperparameter sets we trained the resultant CNN for real, again allowing a maximum of 100 epochs with a patience of 10. After training, the actual validation loss was computed for each model. These eight examples were then added to the dataset containing the other 512 from random search to be used for training the random forest the next time around.

This process was repeated until a total of 512 CNNs, whose hyperparameter values were selected using a random forest, were trained and validated. Training eight models at a time allowed us to iteratively increase the size of the training set available to the random forest, with the expectation that it would gradually learn to make better predictions, resulting in a steady decrease in the validation loss.

Again for convenience we computed and saved the test loss for each model at the end of each trial, but the results were in no way used during training or validation.

Figure 7: Minimum validation loss and corresponding test loss by number of trials.



9 Results

9.1 Random search versus ML-assisted search

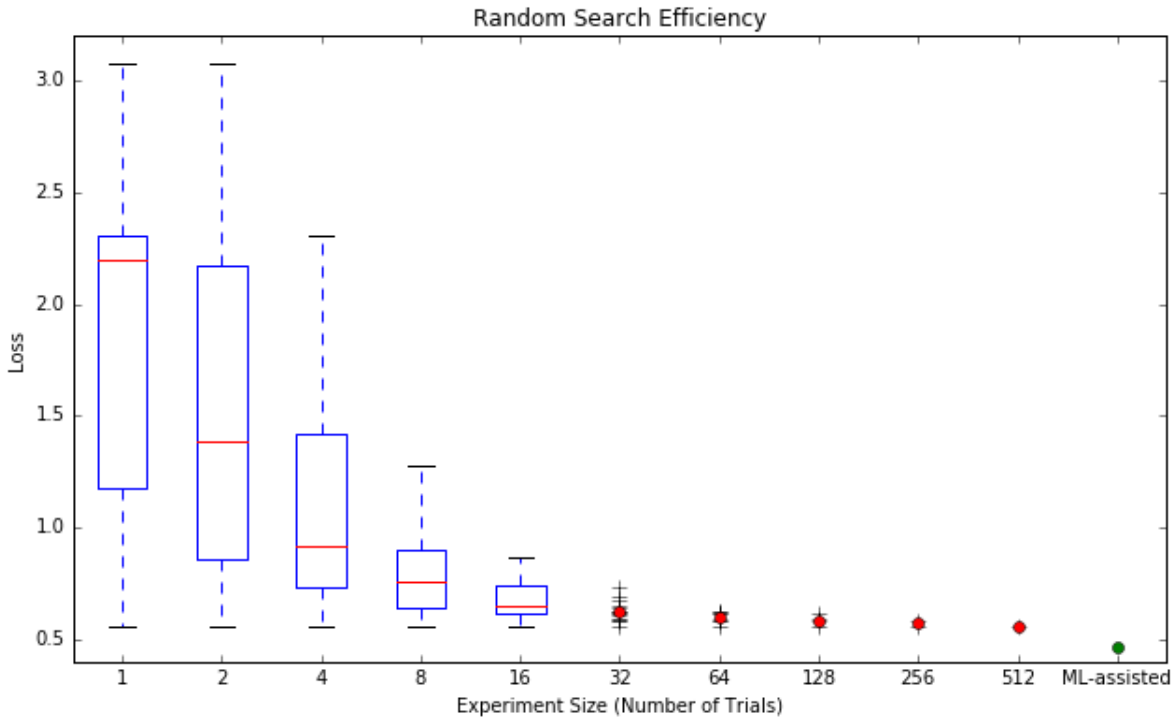
During the model building process of a machine learning task we seek to minimise the loss on a validation set. It is standard procedure to select the model which achieves the lowest validation loss, and then evaluate the generalisation performance of this model on a separate test set. We may train other models with worse validation scores which would actually perform better on the test set, but this is unknown to us during validation and these models would not be selected. When we refer to the "lowest test loss" we mean the test loss of the model with the lowest validation loss. That is, the test score of the model that would have been selected through the model validation process.

The graph in Figure 7 shows the minimum validation loss as a function of the number of trials for both random and ML-assisted search, as well as the corresponding test loss. That is, for every n from 1 to 512 we show the minimum validation loss achieved by any model up to and including the n-th trial, and the test loss that would have been reported if model building had been stopped at that point and the best validated model had been selected.

Overall, the best model from random search was found at iteration 273, corresponding to a validation loss of 0.5403 and a test loss of 0.5616. In comparison, the best model from ML-assisted search, which was found at iteration 337, achieved a validation loss of 0.4454 and a test loss of 0.4695. ML-assisted search needed only three iterations to beat the best model from all of random search, in terms of test score, and by the 37th iteration had already found a model reporting a test loss of 0.5027, an improvement of 0.0589 on 512 iterations of random search. Thus, after performing many random search trials, for little extra effort we were able to find substantially better models in just a few iterations of ML-assisted search.

The plot suggests that pure random search could have taken a long time to find a model comparable to the best found by ML-assisted search. Only once between the 28th and 512th iterations of random search did the minimum validation loss decrease, resulting in a reduction of 0.0203 in the test loss from 0.5819 to 0.5616. A further drop of 0.0921 to reach 0.4695 during a subsequent 512 trials would seem

Figure 8: Random search efficiency plot showing the distribution of the best test results among experiments of varying sizes.



unlikely, and we therefore believe that had all the training time been used to run 1024 random search trials the best model from ML-assisted search would not have been beaten.

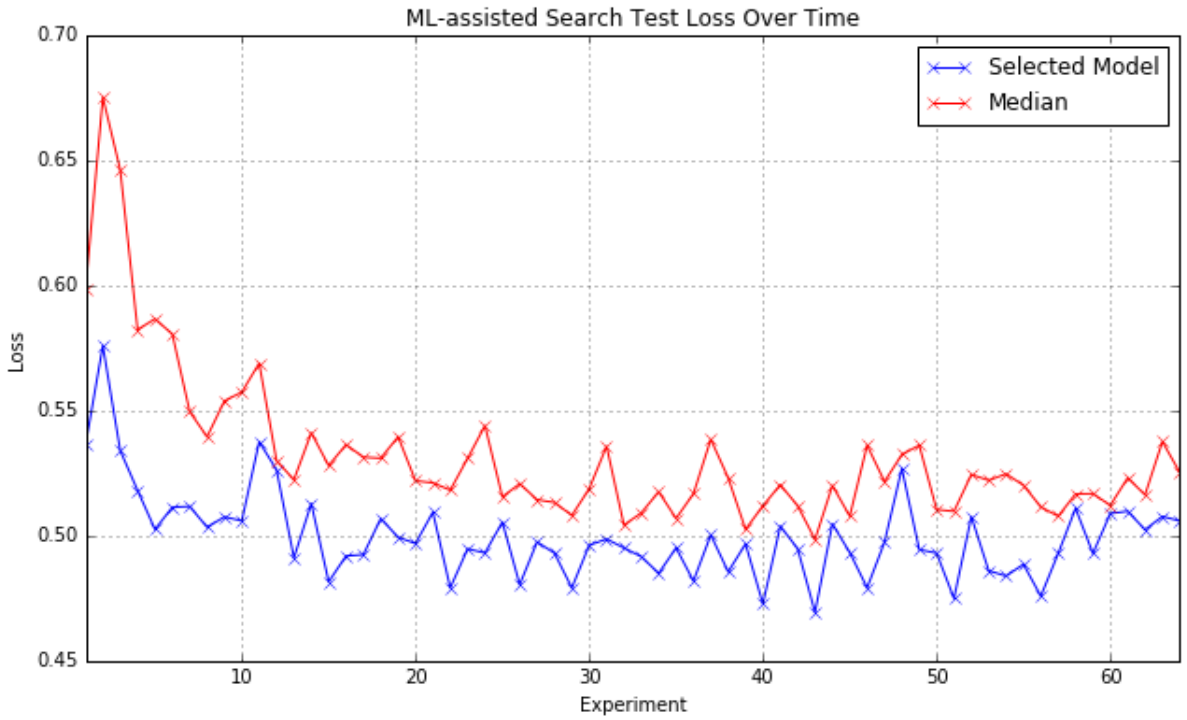
Further evidence to support this hypothesis is shown in the random search efficiency plot in Figure 8, inspired by [22]. Since the 512 random search trials are independent and identically distributed, the results can be interpreted as N independent experiments each consisting of s trials, as long as $sN = 512$. The trials are split into sets chronologically, i.e. the first s trials are grouped together, the next s are grouped together, and so on.

For various values of s, we plot the distribution of the test loss achieved by the model with the lowest validation loss in each experiment of s trials. Where s is small enough such that the number of experiments of s trials is greater than twenty ($s \leq 16$), we use a box plot to show the distribution of results. The box extends from the lower to upper quartile values, with a red line at the median. The whiskers extends to show the range of values, not including outliers. We omit outliers above the upper whisker because for small values of s there are test results greater than 14, which if shown would make the interesting part of the plot difficult to view. For values of s greater than 16 the best test loss in each experiment of s trials is shown by a scatter plot, with a red dot indicating the median value. Finally, we use a green dot to show the lowest test loss from all of ML-assisted search.

As s increases from 1 through to 16 the median sharply drops, since the minimum is taken over larger subsets of the 512 trials. We progressively lose the worst performing models, since those which are best in small experiments but poor overall are less likely to remain among the best in larger groups, and between $s = 2$ and $s = 64$ the distribution of results narrows significantly each time s increases. From $s = 64$ onwards the improvement in the median test result begins to level off. Observing the trend of this plot, it seems unreasonable to expect that if a further 512 random search trials were run the minimum test loss would drop below 0.4695, the best result from ML-assisted search.

Unlike random search, the ML-assisted search trials are not independent and identically distributed. Recall from Section 8 that during ML-assisted search we trained CNNs in experiments of eight trials, and after each experiment we added the results to the dataset that was used to train the random forest. Thus, in each experiment the selection of hyperparameter was influenced by all preceding experiments,

Figure 9: Best and median test loss in each ML-assisted search experiment of 8 trials.



and hence, we cannot plot efficiency in this case. However, we can analyse how the results varied as the search progressed.

Figure 9 shows, for each of the 64 experiments of 8 trials, the test loss achieved by the model with the lowest validation loss. We also plot the median of all test results in each experiment. Although the results are noisy, the general trend is that after the initial few experiments the test loss begins to steadily decrease. This is what we would expect, since the random forest is progressively given more data on which to train, and should therefore learn to make better predictions. Of course, the 1,000,000 candidate hyperparameter sets are chosen randomly, which could explain the noise.

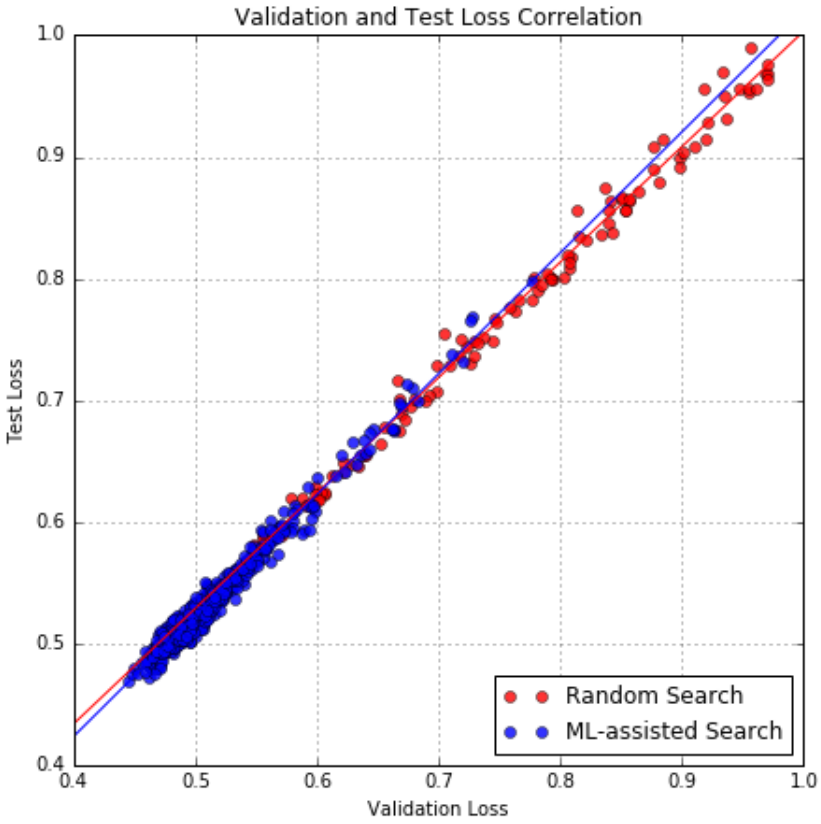
The best model was found during experiment 43, after which point the trend appears to be a slight increase in the test loss. This highlights one drawback to using this method. There is the danger that, because the ML-assisted search results are on average far better than the random search results, the random forest will eventually believe that hyperparameters similar to those previously chosen by itself are the only ones worth considering. This bias could potentially rule out good combinations of hyperparameters that have yet to be explored. The shape of the graph suggests that it is unlikely a new minimum would have been found had we continued ML-assisted search past 64 experiments.

9.2 Overfitting

There is also the danger that ML-assisted search could lead to us severely overfitting the validation set. In Figure 10 we compare the correlation between validation and test results for ML-assisted and random search. Due to the difference in the range of the results, we have chosen to only show random search trials where the test loss is less than one. We can see from the overlaid linear regression lines that ML-assisted search leads to worse overfitting. The overall mean absolute error between validation and test loss for ML-assisted search is 0.0230, whereas for random search the value is around half that at 0.0119.

This is not surprising, since during random search the only time we use the validation set is to decide at which epoch to stop training our network, whereas in the case of ML-assisted search we use a random forest to choose hyperparameter values which it believes work well on the validation set.

Figure 10: Correlation between validation and test loss for random and ML-assisted search.



However, in this case overfitting is not particularly severe, and as we have already seen, does not stop ML-assisted search from finding far superior models to those found by random search.

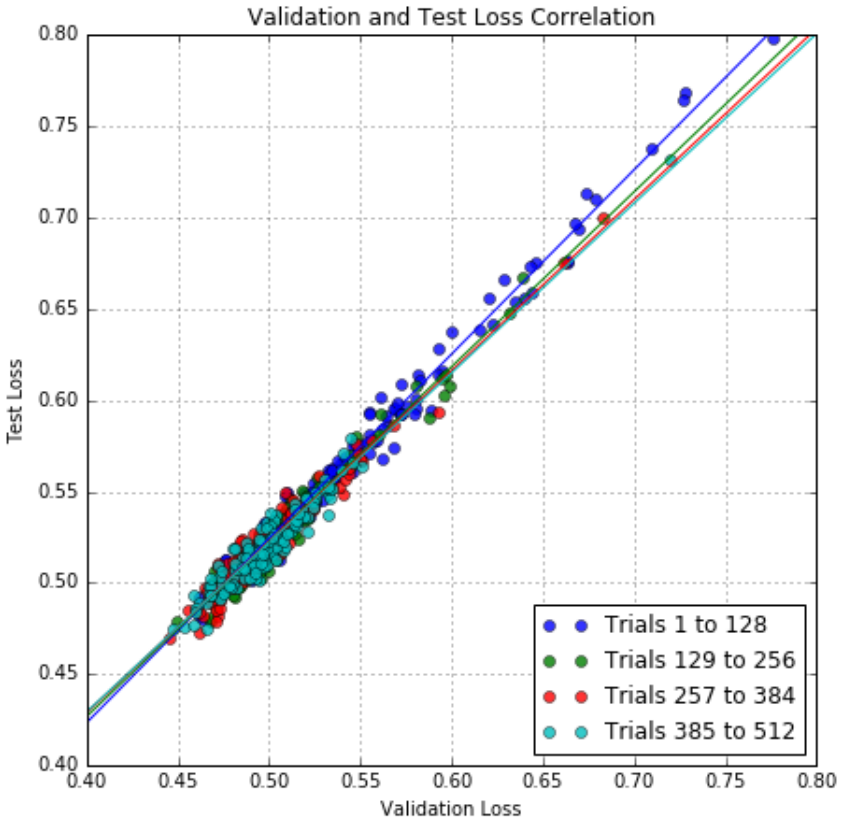
Each time we run a new experiment during ML-assisted search we provide the random forest with extra training data, and hence, more information about the validation set, which increases the risk of overfitting. In Figure 11 we compare the validation and test loss correlation at different stages of ML-assisted search by splitting the trials into four groups of equal size: trials 1 to 128, 129 to 256, 257 to 384 and 385 to 512. We see that overfitting is in fact worst during trials 1 to 128, and for trials 129 and above the overlaid linear regression lines are fairly consistent. Therefore, overfitting does not appear to become more of an issue as the search progresses.

9.3 Evaluation of random forest

Recall that we did not perform any validation on the random forest which was used to choose hyper-parameter values during ML-assisted search. Now that we know the actual validation losses achieved by the trained CNNs we can evaluate how good the predictions were. Here we do not consider the test loss since the random forest was trained to predict validation loss and had no knowledge of test set performance.

For each of the 64 ML-assisted search experiments of 8 trials we computed the mean absolute error (MAE) between the predicted validation loss and the actual validation loss. The results are show in Figure 12. In Figure 13 we plot the lowest actual validation loss in each experiment, as well as the corresponding predicted validation loss. In each of the first six experiments the MAE in Figure 12 is relatively high. During this initial period Figure 13 shows that the random forest predicts that the best model in each experiment will achieve a validation loss of around 0.675, when in fact the true values are much lower. Strangely, at the 7th experiment the prediction for the best model suddenly drop to around 0.56, moving it significantly closer to the actual value. It is understandable that the random forest would lower its predictions as it is exposed to more examples where the actual loss is low, but

Figure 11: Correlation between validation and test loss during four stages of ML-assisted search.



there is no obvious reason as to why the change is so sharp.

Although the plots are noisy, the general trend is that the predicted and actual validation losses get closer as the search progresses, with a maximum MAE of 0.113 at experiment 4 and a minimum of 0.007 at experiment 59. It is probable that the MAE would have continued to decrease had we ran the search for longer, but as discussed in Section 9.1, it is unlikely that this would have led to an improvement on the best model.

One interesting observation from Figure 13 is that the predicted loss of the best model in each experiment is always greater than the actual loss. It appears that the random forest only makes predictions which are within the domain of the training data. That is, if the lowest actual validation loss across all examples in the training set is 0.5, say, then the random forest is unlikely to predict that any CNN will achieve anything better than this.

While we would like the predictions to be as close to the actual results as possible, what determines the usefulness of ML-assisted search is how well the random forest is able to order the hyperparameter candidate sets. That is, if the random forest predicts that one CNN will be better than other, we want this to be true when the models are trained and validated for real.

Every time we ran an experiment we used the random forest to predict the validation loss for 1,000,000 randomly selected hyperparameter sets, then trained the top eight. Ideally these eight would have been better than the other 999,992, if they too had been trained and validated for real. Unfortunately we do not know the true validation loss for any of the CNNs which were not selected, however, we can analyse the ordering of the eight which were trained in each experiment.

In Figure 14 we plot the actual validation loss achieved by the eight CNNs trained in every fourth experiment (showing more experiments makes the data difficult to view). The colour of each dot indicates how it was ranked by the random forest. For example, blue indicates that the corresponding CNN was predicted to achieve the lowest validation loss of all examples, green indicates the second lowest prediction, and so on. We see that in most experiments the ordering is fairly mixed, and there

Figure 12: Mean absolute error between predicted and actual validation loss in each ML-assisted search experiment of eight trials.

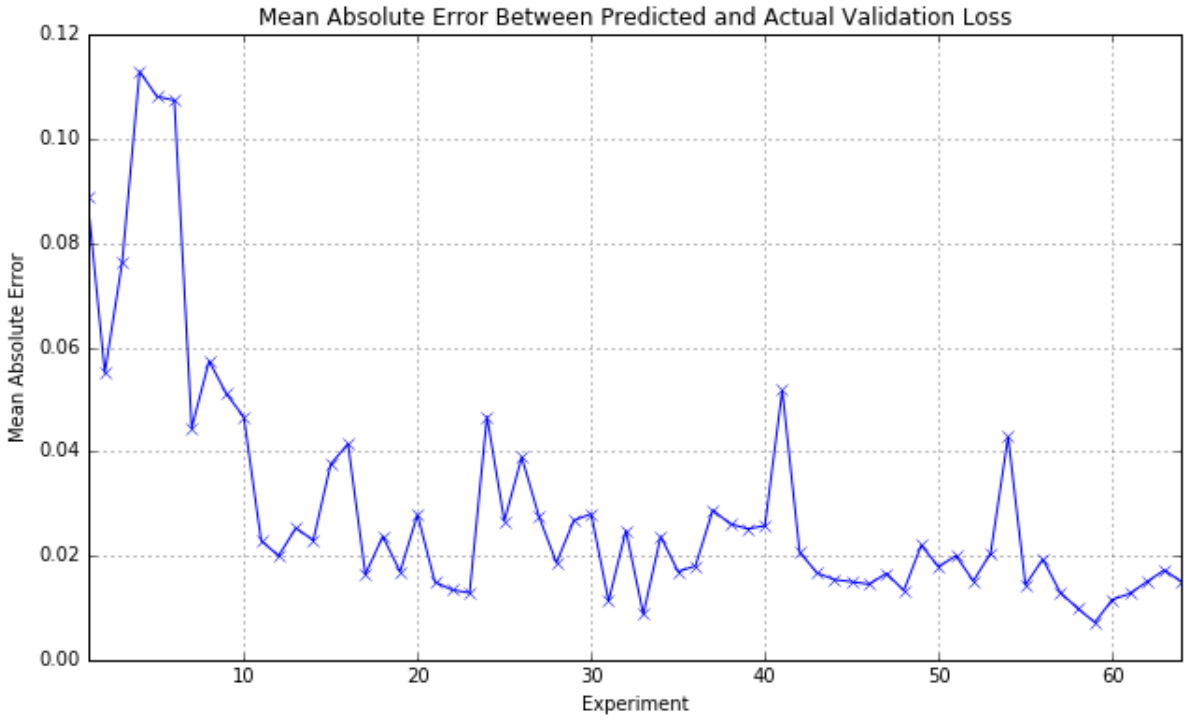


Figure 13: Lowest actual validation loss and corresponding predicted validation loss in each ML-assisted search experiment of eight trials.

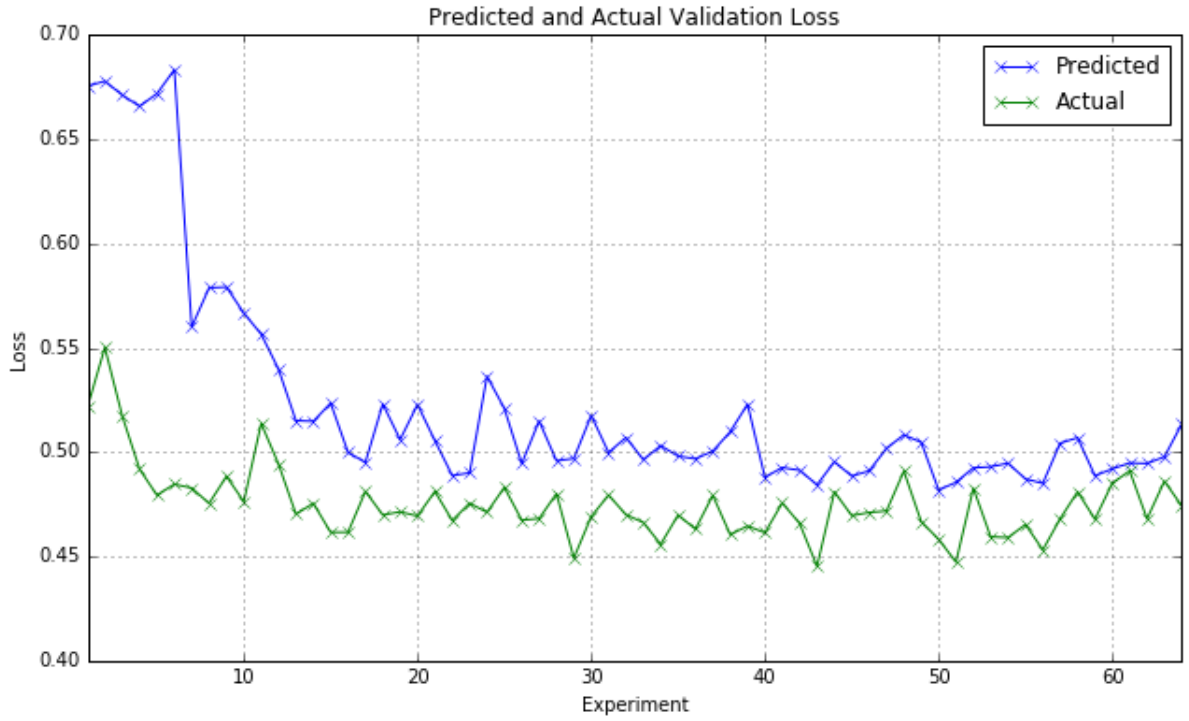
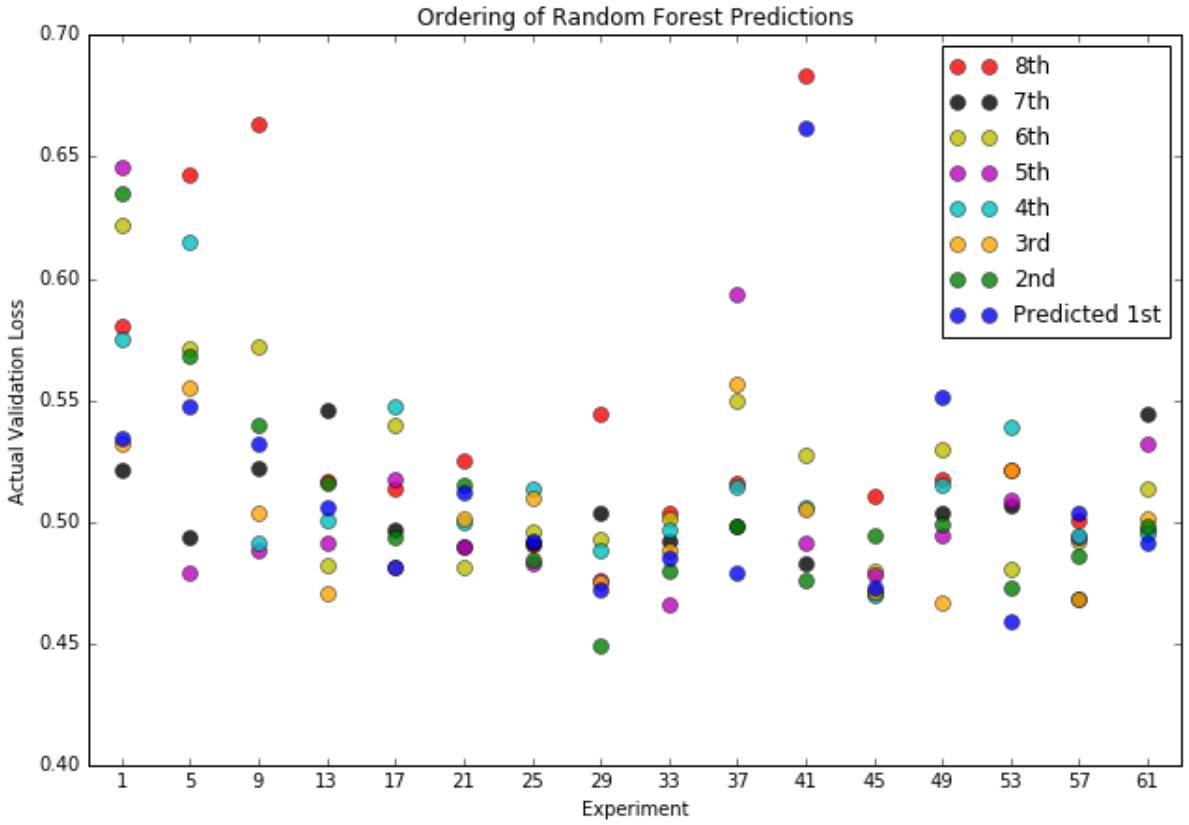


Figure 14: Actual validation loss of all eight CNNs trained in every fourth ML-assisted search experiment, as well as colours showing order of random forest predictions.



are no experiments in which the random forest predicted the order perfectly. However, considering that each time we selected only the best eight from 1,000,000 candidates, there was not much to separate the top few predictions.

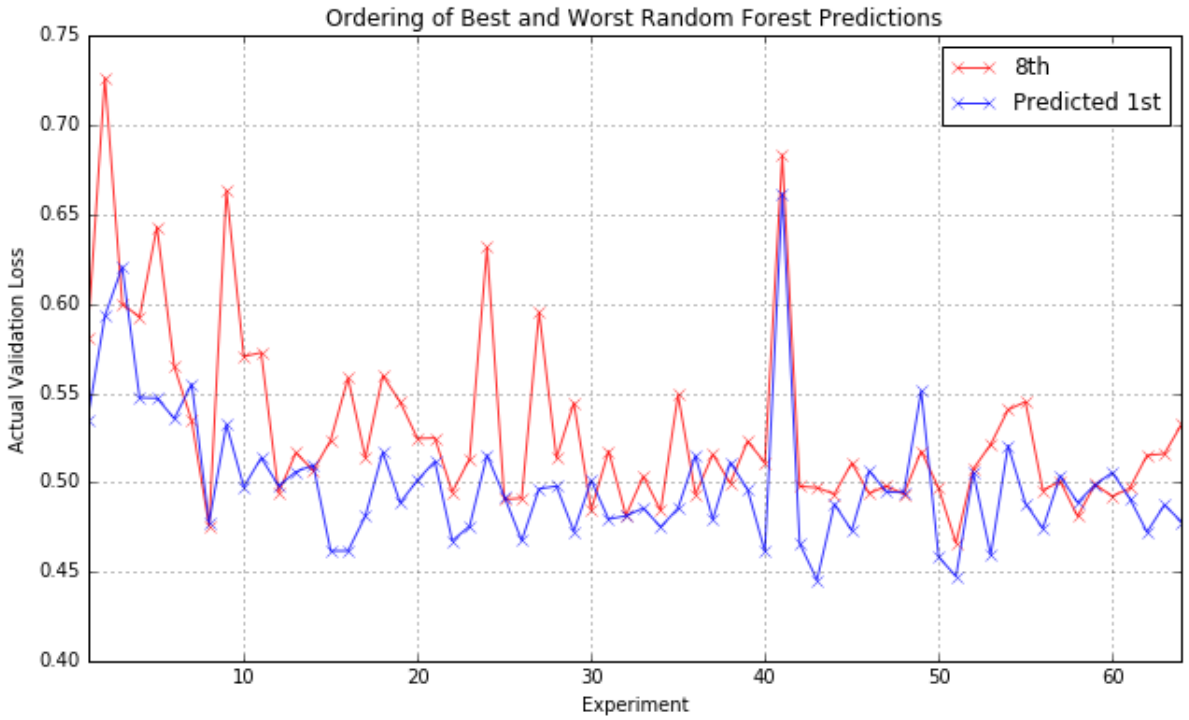
If we concentrate on just the blue and red dots we see that more often than not blue appears below red. That is, the actual validation loss of the CNN ranked 1st by the random forest is, in the main, less than that of the CNN ranked 8th. Figure 15 shows the actual validation loss for only the CNNs ranked 1st and 8th by the random forest in each experiment. On 49 out of 64 occasions the random forest ordered these two examples correctly, and in almost all cases where red does appear below blue the difference is small. On average the 1st prediction is 0.0263 less than the 8th, in comparison to 0.0299 for the actual difference in validation loss. Thus, considering the limited training data available, the random forest appears to order the examples fairly well, at least for this small section of the 1,000,000 candidate hyperparameter sets.

9.4 Hyperparameter importance

The `RandomForestRegressor` class we used to make predictions can also estimate how important each feature in the training data is for predicting the target. In other words, how big an impact each of the eleven CNN hyperparameters has on the validation loss.

This is done using the *out-of-bag* prediction error [24]. Each tree in a random forest is grown using only a subset of the training data, and we can compute the prediction error for those examples left "out the bag". Then, one at a time, we randomly permute the values of each feature used to grow the tree while holding all others fixed, and re-calculate the out-of-bag prediction error. "The increase in the error is a measure of the predictive performance of the feature" [28]. In this case, where features are hyperparameters of the CNN, the greater the increase in the error of the predicted validation loss

Figure 15: Actual validation loss of CNNs ranked 1st and 8th by random forest in each experiment.



when the hyperparameter's values are randomly permuted, the more significant the hyperparameter.

To estimate hyperparameter importance in this way we trained the random forest using all 1024 examples from random and ML-assisted search. Figure 16 shows the average increase in the out-of-bag prediction error across all trees caused by randomly permuting the values of each hyperparameter. By far the most important hyperparameter is the initial learning rate in mini-batch gradient descent. In their random search experiments using eight different datasets, Bergstra and Bengio found that which hyperparameters are important varies for different data, but on seven out of eight occasions the initial learning rate was the most significant [22]. Also, in his practical guide to training deep neural networks, Bengio recommends "if there is only time to optimize one hyper-parameter and one uses stochastic gradient descent, then this is the hyper-parameter that is worth tuning" [17]. Evidently, the CIFAR-10 dataset is no different.

In Figure 17 we plot the distribution of the initial learning rate values chosen for the ML-assisted search trials using the random forest, that is, the values among the top eight predictions in each experiment. For comparison we also plot the distribution of the learning rate decay, which was also sampled exponentially from the same interval between 1×10^{-5} and 1, but considered far less important by the random forest. We see that in the case of the initial learning rate the random forest considers good values to come from a much smaller region on the log scale.

A similar effect is observed for dropout after the convolutional layers, considered the second most important hyperparameter by the random forest, and dropout after the fully-connected layers, which is much less significant. Both hyperparameters were sampled uniformly from the interval $(0, 1)$, but we see in Figure 18 that values considered good for dropout after the convolutional layers come from a far narrower interval.

Thus, the random forest appears to adopt the same strategy as a human being might when performing thoughtful manual search. That is, it recommends fine searches for hyperparameters which it considers important, but is not as concerned about tuning the rest.

Figure 16: Hyperparameter importance, as estimated by the random forest used to make predictions.

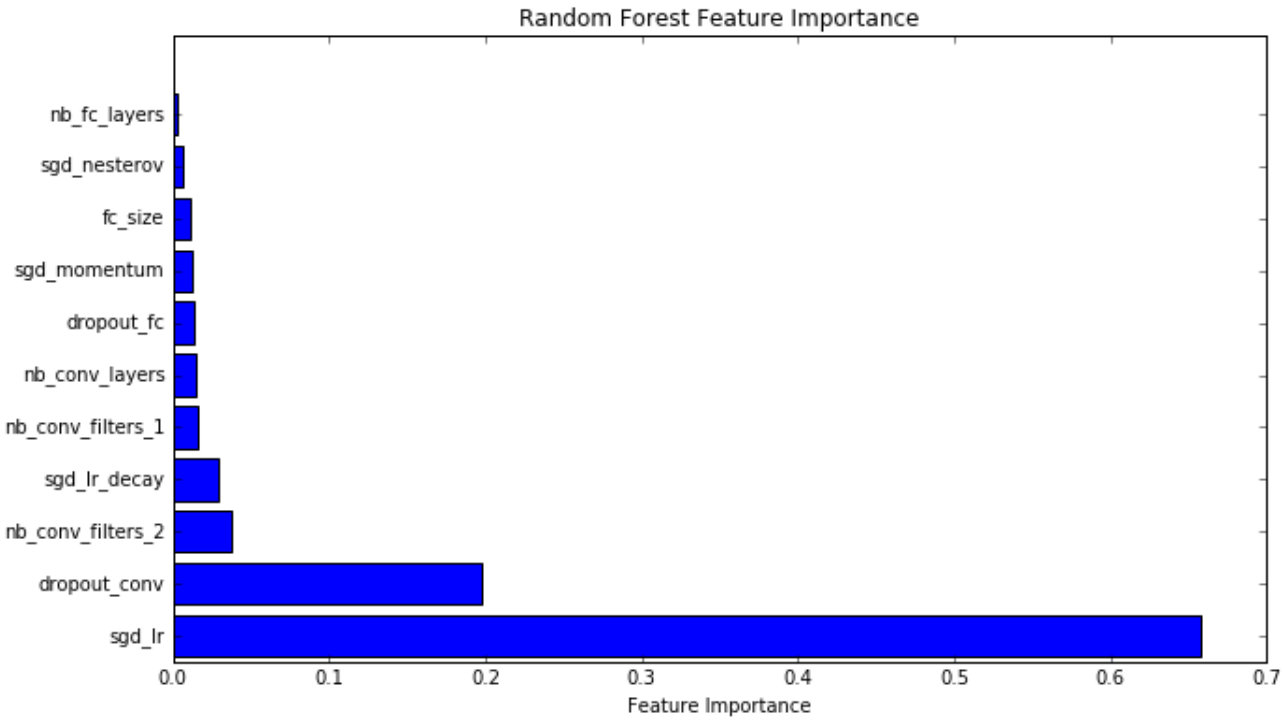


Figure 17: Distribution of initial learning rate and learning rate decay values of CNNs trained during ML-assisted search. These values were from the top eight random forest predictions in each experiment.

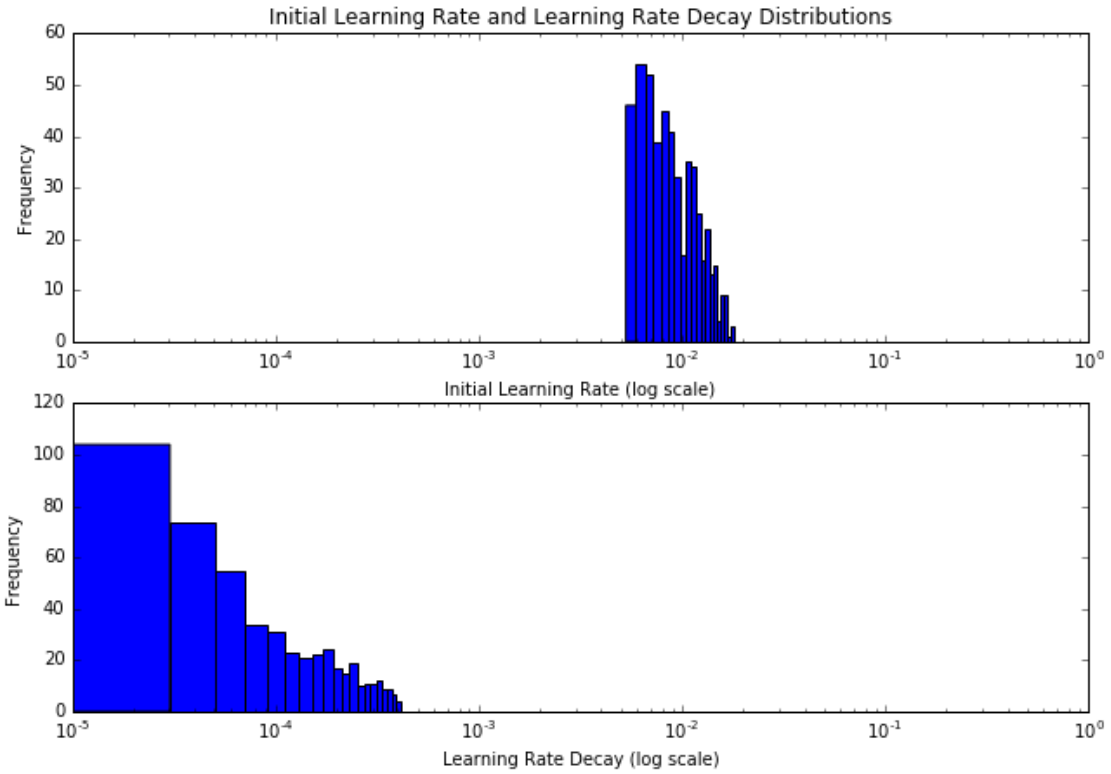
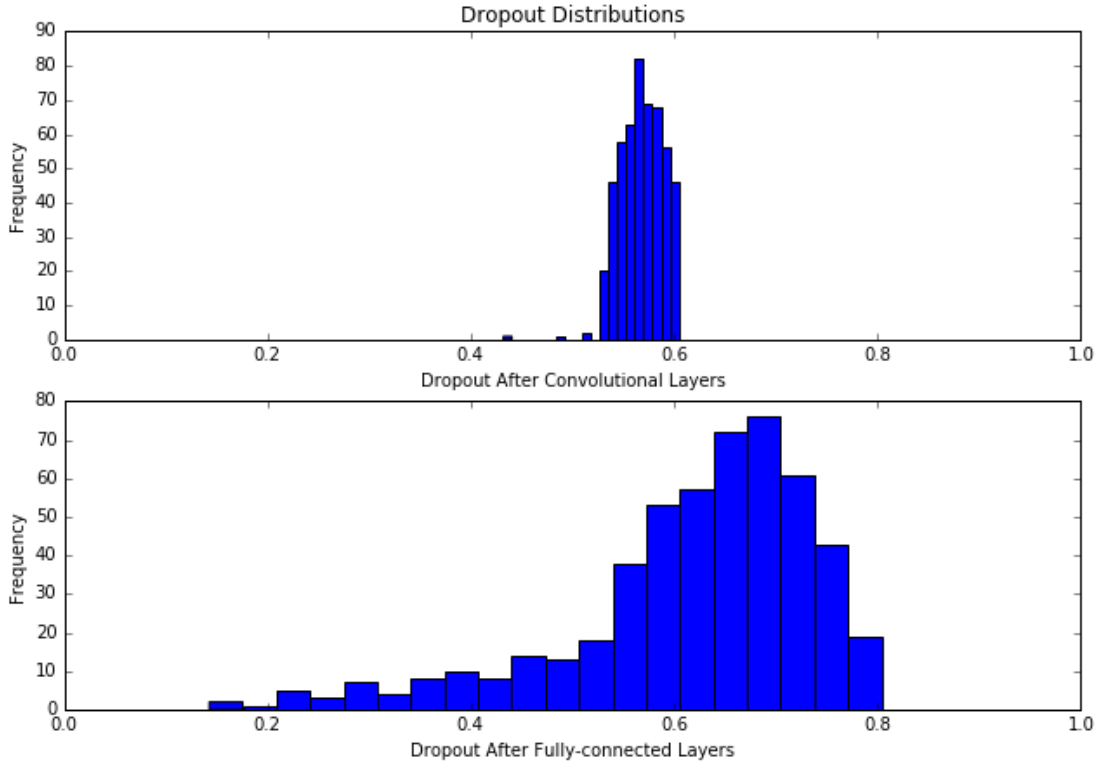


Figure 18: Distribution of values of dropout after convolutional layers and dropout after fully-connected layers of CNNs trained during ML-assisted search. These values were from the top eight random forest predictions in each experiment.



10 Conclusion

Random search is a method commonly used to optimise CNN hyperparameters. Due to the large number of hyperparameters involved, often several hundred or more trials are needed to thoroughly search over the high-dimensional hyperparameter space. We have shown that in such situations we can subsequently find significantly better models by using random search results to perform ML-assisted search. Importantly, ML-assisted can be led by an untuned, unvalidated random forest, thus requiring no extra manual effort.

We demonstrated that the best results from several hundred random search trials can be beaten in only a few iterations of ML-assisted search. Thus, even if random search is preferred and allocated the vast majority of available training time, we suggest ML-assisted search as a quick and easy way to make further performance gains. However, we believe that a significant amount of the total training time should be dedicated to ML-assisted search, and showed that it can lead to significantly better performance if allowed to run for the same number of iterations as random search.

We propose ML-assisted search as an alternative to the widely adopted combination of random and thoughtful manual search. As the name suggests, manual search requires significant human input and can often seem like more of an art than a science to beginners with little experience of training CNNs. We demonstrated that a random forest can learn to focus ML-assisted search on narrow intervals for hyperparameters which it considers important, effectively performing the same role as a human during manual search.

Further work should include a direct comparison of our results with a combination of random and manual search. Overall we spent roughly 15 hours per day for 24 consecutive days training CNNs on a single NVIDIA GTX 980 Ti GPU. During this period the only human involvement required was to set the code running each morning. Using the CIFAR-10 dataset and starting from the same hyperparameter domains, it would be interesting to see the level of performance that could be achieve

by random and manual search over a similar time period using similar computing resources.

ML-assisted search should also be tested on other datasets. Here an untuned random forest was used to indicate good hyperparameter values due to its reputation as the closest thing to a "black box" machine learning algorithm. This approach worked well for the CIFAR-10 images and we should try using the exact same random forest to optimise CNNs for other datasets. The usefulness of ML-assisted search is largely dependent on being able to use the same model to predict validation scores, irrespective of the data. We want to avoid complicating the process by introducing a second model which itself requires extensive tuning and validation. The model does not have to be particularly accurate as long as it is able to tell a good hyperparameter set from a bad one. In the first instance we would recommend using the same model with which we had success: a regression random forest with 500 trees and a minimum leaf size of 5.

One drawback to ML-assisted search which we observed was that after around 350 trials the test loss of the trained models began to increase slightly. We believe this was due to the random forest becoming "biased" and ruling out good hyperparameter combinations which had yet to be explored. To avoid this we propose a slightly different approach to combining ML-assisted search with random search. As before we would first run a significant number of random search trials to obtain an initial training set for the random forest. But then instead of performing only ML-assisted search from that point on, we could alternate between the two methods. That is, we would run an ML-assisted search experiment of eight trials, say, followed by a random search experiment of eight trials, and repeat. As before, after each experiment the validation results would be added to the dataset used to train the random forest. The difference is that the hyperparameters values in every second set of eight trials would be chosen randomly, maintaining the diversity of both the random forest's training set and the CNNs being explored. Future work could trial this approach on the CIFAR-10 images, again running an initial 512 random search trials before alternating until we train a total of 1024 CNNs.

To summarise, we see CNN hyperparameter optimisation as a nuisance and propose ML-assisted search as a completely automated approach which improves on pure random search. For anyone who does not have sufficient knowledge, time or enthusiasm to perform thoughtful manual search, we suggest ML-assisted search as a worthy alternative.

The code used to run random and ML-assisted search experiments on the CIFAR-10 dataset can be found at `https://github.com/ayeright/ml-assisted-hyperparameter-search`.

References

[1] L. Fei-Fei, A. Karpathy and J. Johnson. *Convolutional Neural Networks for Visual Recognition. Optimization: Stochastic Gradient Descent*. Retrieved from http://cs231n.github.io/optimization-1/, 15/10/16.

[2] L. Fei-Fei, A. Karpathy and J. Johnson. *Convolutional Neural Networks for Visual Recognition. Neural Networks Part 3: Learning and Evaluation*. Retrieved from http://cs231n.github.io/neural-networks-3/, 15/10/16.

[3] L. Fei-Fei, A. Karpathy and J. Johnson. *Convolutional Neural Networks for Visual Recognition. Convolutional Neural Networks: Architectures, Convolutional/Pooling Layers*. Retrieved from http://cs231n.github.io/convolutional-networks/, 15/10/16.

[4] Y. LeCun, L. Bottou, G. B. Orr and K.-R. Muller. *Efficient BackProp*. In G. B. Orr and K.-R. Muller, editors, *Neural Networks: Tricks of the Trade*, Springer, 1998.

[5] S. Ruder. *An Overview of Gradient Descent Optimization Algorithms*. Technical paper, Insight Centre for Data Analytics, NUI Galway, 2016.

[6] N. Qian. *On the Momentum Term in Gradient Descent Learning Algorithms*. Neural Networks: The Official Journal of the International Neural Network Society, 12(1), pp. 145-151, 1999.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei. *ImageNet: A Large Scale Hierarchical Image Database*. IEEE Computer Vision and Patter Recognition (CVPR), 2009.

[8] O. Russakovsky and J. Deng. *ImageNet Large Scale Visual Recognition Challenge*. IJCV, 2015.

[9] V. Nair and G. Hinton. *Rectified Linear Units Improve Restricted Boltzmann Machines*. Proc. of 27th International Conference on Machine Learning, 2010.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research 15, pp. 1929-1958, 2014.

[11] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner. *Gradient-Based Learning Applied to Document Recognition*. Proc. of the IEEE, November 1998.

[12] A. Krizhevsky, I. Sutskever and G. Hinton. *Imagenet Classification with Deep Convolutional Neural Networks*. Advances in Neural Information Processing Systems, 2012.

[13] C. Szegedy and W. Liu. *Going Deeper with Convolutions*. Proc. of 28th IEEE Conference on Computer Vision and Patter Recognition, 2015.

[14] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Classification*. Proc. of 5th International Conference on Learning Representations, 2015.

[15] K. He, X. Zhang, S. Ren and J.Sun. *Deep Residual Learning for Image Recognition*. Proc. of 29th IEEE Conference on Computer Vision and Patter Recognition, 2016.

[16] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. MSc thesis, University of Toronto, 2009. Retrieved form http://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf, 12/12/16.

[17] Y. Bengio. *Practical Recommendations for Gradient-Based Training of Deep Architectures*. Technical report, Universite de Montreal, 2012. Retrieved from https://arxiv.org/pdf/1206.5533v2.pdf, 12/12/2016.

[18] I. Sutskever, J. Martens, G. Dahl and G. Hinton. *On the Importance of Initialization and Momentum in Deep Learning*. Proc. of 30th International Conference on Machine Learning, pp. 1139-1147, 2013.

[19] J. Bergstra, R. Bardenet, Y. Bengio and K. Balazs. *Algorithms for Hyperparameter Optimization*. Advances in Neural Information Processing Systems, 2011.

[20] D. Maclaurin, D. Duvenaud and R. P. Adams. *Gradient-based Hyperparameter Optimization through Reversible Learning*. Proc. of 32nd International Conference on Machine Learning, Lille, France, 2015.

[21] J. Snoek, H. Larochelle and R. P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms*. Advances in Neural Information Processing Systems, 2012.

[22] J. Bergstra and Y. Bengio. *Random Search for Hyper-Parameter Optimization*. Journal of Machine Learning 13, pp. 281-305, 2012.

[23] T. Hastie, R. Tibshirani, J. Friedman. *Neural Networks*. In *The Elements of Statistical Learning*. New York, NY, Springer, 2009, pp. 389-415.

[24] T. Hastie, R. Tibshirani, J. Friedman. *Random Forests*. In *The Elements of Statistical Learning*. New York, NY, Springer, 2009, pp. 587-603.

[25] Random forest regressor class. *scikit-learn: Machine Learning in Python*. Retrieved from http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html, 12/12/2016.

[26] C. Cortes, V. Vapnik. *Support-Vector Networks*. AT&T Labs-Research, USA. Retrieved from http://homepages.rpi.edu/ bennek/class/mmld/papers/svn.pdf, 12/12/16.

[27] L. Breiman. *Random Forests*. University of California, USA, 2001. Retrieved from https://www.stat.berkeley.edu/ breiman/randomforest2001.pdf, 12/12/16.

[28] S. Brownlie. *Validation Strategies for Automated Decision Making*. Technical report, Advanced Module 1, Cranfield University, 2016.