Supplementary Information for

Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning

Babak Alipanahi[1,2,*], Andrew Delong[1,*], Matthew T. Weirauch[3,4,5], Brendan J. Frey[1,2,3]

Corresponding author: frey@psi.toronto.edu

1. Department of Electrical and Computer Engineering, University of Toronto, Ontario Canada

2. Donnelly Centre for Cellular and Biomolecular Research, University of Toronto, Ontario, Canada

3. Canadian Institute for Advanced Research, Programs on Genetic Networks and Neural Computation, Toronto, Ontario, Canada

4. Center for Autoimmune Genomics and Etiology, Cincinnati Children's Hospital Medical Center, Cincinnati, USA

5. Divisions of Biomedical Informatics and Developmental Biology, Cincinnati Children's Hospital Medical Center, Cincinnati, USA

* Authors contributed equally.

Table of Contents

Nature Biotechnology: doi:10.1038/nbt.3300

1 DeepBind models

We adapted *deep convolutional neural networks* (ConvNets) to generate predictions from raw genomic sequences. Like any neural network, a ConvNet comprises one or more computational stages ('layers'), where the final stage's output is treated as a prediction. Each stage has tunable parameters (weights, biases) that, when trained to solve a prediction task, can discover features of the input data that are useful for that task. The first stage of a ConvNet is always *convolution*, where the raw input is convolved (or cross-correlated) with several tunable patterns called *filters*. ConvNets were originally developed for visual tasks, such as handwriting recognition, where they automatically learn patterns that respond to edges and curve fragments in images[1,2]. Likewise, our DeepBind ConvNets automatically learn motif detectors, along with rules for combining them to make good predictions on sequence analysis tasks. Our DeepBind model is developed for the simple and well-studied problem of protein-nucleotide binding. However, the ConvNet formalism easily extends in many useful ways: multiple input sequences, extra tracks (*e.g.* conservation, secondary structure), extra genomic features, or other tasks. See **Supplementary Sec. 1.3** for discussion.

A DeepBind model takes a single sequence $s = (s_1, \ldots, s_n)$ with alphabet $s_i \in \{\mathrm{A, C, G, T, N}\}$, and produces a real-valued score $f(s)$. There are four computational stages, in order: convolution, rectification, pooling, and neural network. Each is explained in **Supplementary Sec. 1.1** below. The convolution, rectification, and network stages have trainable motif detectors M, thresholds b, and weights W, respectively; the pooling stage is always fixed and has no trainable parameters. In a DeepBind model, the output score $f(s)$ for sequence s is computed by a feed-forward expression starting with convolution and ending in a neural network:

$$f(s) = \mathrm{net}_W\left(\mathrm{pool}\left(\mathrm{rect}_b\left(\mathrm{conv}_M(s)\right)\right)\right).$$

The specific details of each computation may vary (number of motif detectors; max- versus average-pooling; deep versus shallow neural net) but the general form is the same. During training, our calibration phase automatically chooses these details.

The meaning of score $f(s)$ depends on the particular task when the model was trained. For example, a model trained to predict measured binding affinity will score new sequences on the scale of those training measurements; see **Supplementary Sec. 1.3**. Once a specific error function has been chosen, e.g. mean squared error, the model is trained by stochastic gradient descent on the expected error across the training set; see **Supplementary Sec. 2**.

1.1 Feed-forward stages (predictions)

Convolution. The defining feature of any ConvNet is that it begins with at least one convolution stage. In the context of genomic sequences, a 1-dimensional convolution over a 4-channel input can play the same role as a "motif scan" operation in a PWM- or PSAM-based model. However, the coefficients of our tunable motif detector matrices are arbitrary and under no hard constraints such as being positive or summing to 1.

Specifically, given a genomic sequence $s = (s_1, \dots, s_n)$ and a maximum motif detector length m, our convolution stage begins by converting s to a padded "one-of-four" representation, stored as an $(n + 2m - 2) \times 4$ array S in the obvious way:

$$S_{i,j} = \begin{cases} .25 & \text{if } s_{i-m+1} = \mathrm{N} \text{ or } i < m \text{ or } i > n - m \\ 1 & \text{if } s_{i-m+1} = j^{\text{th}} \text{ base in } (\mathrm{A,C,G,T}) \\ 0 & \text{otherwise} \end{cases}$$

For example, if $s = \mathrm{ATGG}$ and motif detector length is $m = 3$ then the representation is

$$S = \begin{bmatrix} .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \end{bmatrix}$$

The output of our convolution stage is an $(n + m - 1) \times d$ array X where d is the number of tunable motif detectors within the DeepBind model. Element $X_{i,k}$ is essentially the score of motif detector k aligned to position i of padded sequence S. The tunable motif detectors (all length m) are stored in an $d \times m \times 4$ array M where element $M_{k,j,l}$ is the coefficient of motif detector k at motif position j and base l. Specifically, the expression $X = \mathrm{conv}_M(s)$ computes a discrete cross-correlation between padded sequence S and each detector M_k, where

$$X_{i,k} = \sum_{j=1}^{m} \sum_{l=1}^{4} S_{i+j,l} M_{k,j,l}$$

for all $1 \le i \le n + m - 1$ and $1 \le k \le d$. We call column $X_{\cdot,k}$ the *motif scan* of motif k applied to sequence s; in machine learning terminology, this is known as a 1-dimensional *feature map* using a 4-channel input.

Rectification. The rectification stage takes the $(n+m-1)\times d$ output array of convolution X and computes an array of the same size $Y = \mathrm{rectify}_b(X)$ where $b = (b_1,\dots,b_d)$ are tunable thresholds and

$$Y_{i,k} = \max(0, X_{i,k} - b_k).$$

We call this a *rectified motif scan*. In neural network terminology, this simple non-linear operation is known as a *rectified linear unit* (ReLU) layer, and has played an essential role in the recent success of deep learning for computer vision[2] and speech recognition[3]. For DeepBind models, each coefficient b_k can be understood as an activation threshold for motif detector k applied at every sequence position. If score $X_{i,k} \geq b_k$ then motif detector k's relative score at position i is passed to the next stage; otherwise motif detector k is deemed irrelevant at position i and so the relative score is zero.

Pooling. After convolution and rectification, matrix Y is still $(n+m-1)\times d$ and so its size depends on the length n of input sequence s. We implemented two choices for the pooling stage: "max pooling", or "max pooling and average pooling."

Our max pooling stage reduces Y to a d-dimensional vector z, where d is the number of motif detectors. In machine learning this is called 'pooling' because it pools (reduces) over the spatial dimension. The computation is $z = \mathrm{max\_pool}(Y)$ where

$$z_k = \max(Y_{1,k},\dots,Y_{n,k}).$$

for each $1 \leq k \leq d$. For DNA-binding proteins, max pooling performed well on its own.

RNA-binding protein models tended to benefit from knowing the average response of a motif detector within the sequence. So, instead of max pooling, all our RBP models calculate a $2d$-dimensional vector $z = \mathrm{max\_avg\_pool}(Y)$ where

$$\begin{aligned} z_{2k+0} &= \max\left(Y_{1,k},\dots,Y_{n,k}\right) \\ z_{2k+1} &= \mathrm{avg}\left(Y_{1,k},\dots,Y_{n,k}\right) \end{aligned}$$

for each $1 \leq k \leq d$. For example, the 'features' element in **Figures 2a** and **Supplementary Fig. 1** is depicted with d dimensions (like our TF models), but could have been depicted with $2d$ dimensions to indicate both types of pooling (like our RBP models).

Neural network. The neural network stage transforms feature vector z (the output of pooling) into a scalar output score f. Each time we train a model on a new data set, we allow the choice of either "no hidden layer" or "one hidden layer with 32 rectified-linear units," and use whichever kind of model gives the best validation performance (see **Supplementary Sec. 2.1**).

Without loss of generality, assume the incoming feature vector z is $1\times d$. In the case of "no hidden layer", a neural network is just a $(d+1)$-dimensional vector of tunable weights w, where the output score is computed as the linear combination

$$p = w_{d+1} + \sum_{k=1}^{d} w_k z_k$$

where weight w_k is the weight of z_k's contribution to the output, and w_{d+1} is an additive bias term.

In the case of "one hidden layer with 32 rectified-linear units", the neural network comprises a $32\times(d+1)$ matrix W of tunable weights, where $W_{j,k}$ is the weight of z_k's contribution to hidden unit h_j and $W_{j,d+1}$ is an additive bias. A second 33-dimensional parameter vector w then forms a weighted combination of the hidden unit values to give final score f. The computations are

$$h_j = \max\left(0, W_{j,d+1} + \sum_{k=1}^{d} W_{j,k} z_k\right) \text{ for } j = 1 \dots 32$$

$$f = w_{33} + \sum_{j=1}^{32} w_j h_j$$

For brevity, we often refer generally to "network weights W" throughout the text, meaning all neural network weights, including the specific matrices W and w described above (in the case of a hidden layer).

Neural network with dropout. We implement a recent enhancement of neural networks known as *dropout*[4]. The idea behind dropout is to occasionally "drop out" intermediate values (e.g. the z_k values) by randomly setting them to zero during training. Injecting this particular type of noise at training time has a strong regularization effect, preventing the neural network from learning complex rules ("conspiracies") that are often associated with over-fitting.

For the "no hidden layer" case, the contribution of each z_k is masked by a random binary variable m_k. Each mask variable is re-sampled every time a new training input is fed through the network. The expected value of m_k is determined by constant calibration parameter $\alpha \in (0,1]$. During training, the score is computed as

$$f = w_{d+1} + \sum_{k=1}^{d} m_k w_k z_k \quad \text{where} \quad m_k \sim \text{Bernoulli}(\alpha)$$

When evaluating the trained model on validation or test data, the stochasticity of dropout is removed so that the prediction for any given test input is consistent. Instead, values are scaled by the expected value of the masks:

$$f = w_{d+1} + \alpha \sum_{k=1}^{d} w_k z_k$$

Similarly, for the "one hidden layer with 32 rectified-linear units" we apply dropout to the hidden units h_j so that each has its own mask variable m_j.

$$f = w_{33} + \sum_{j=1}^{32} m_j w_j h_j$$

Again, all mask variables are sampled anew from distribution $\text{Bernoulli}(\alpha)$ each time a new input is fed through the network, and, at test time the contributions are scaled:

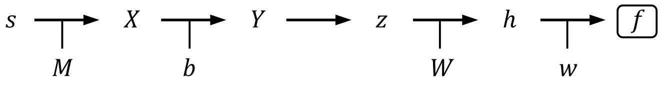
$$f = w_{33} + \alpha \sum_{j=1}^{32} w_j h_j$$

Setting $\alpha = 1$ reduces to the standard no-dropout case for both training and testing.

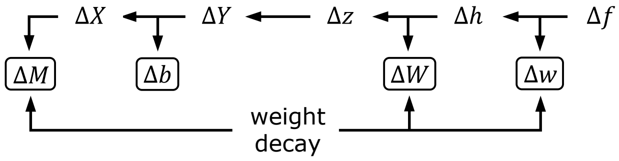
1.2 Back-propagation stages (gradients)

We train DeepBind models using a gradient descent algorithm, so as to approximately minimize the prediction error according to some loss function (**Supplementary Sec. 1.3**). The algorithm is known as *mini-batch stochastic gradient descent* (SGD), described in **Supplementary Sec. 2**. This requires calculating the gradient of motif detector parameters M, thresholds b, and network weights W with respect to the objective function used for training. In neural networks, including convolutional neural networks, gradients are efficiently computed by *back-propagation*; see Duda et al.[5] for an introduction to back-propagation in neural networks.

For a DeepBind model with one hidden layer, the computation of $f(s)$ has the following "forward-propagation" dependency structure showing, for example, that "motif scans X are a function of both sequence s and of motif detectors M".



Given a measure of prediction error Δf, given by a loss function, computation of gradients ΔM, Δb, and ΔW has the reverse dependency structure, hence the name *back-propagation*. Our training objective also has "weight decay" terms that contribute to the gradients in a way that is independent of Δf.



Once the above gradients are computed, they are used by gradient descent to update the original model parameters M, b, and W as depicted in **Fig. 2**.

All of the gradient calculations below can be derived via the chain rule.

Neural network with dropout. Here we describe the most general case, with 32-unit rectified linear layer and dropout. Given Δf, the 33-dimensional gradient vector Δw is computed as

$$\begin{aligned} \Delta w_j &= m_j h_j \Delta f \quad \text{for } j = 1 \dots 32 \\ \Delta w_{33} &= \Delta f \end{aligned}$$

where m_j is the same $\{0,1\}$ mask value that was sampled during forward-propagation. The 32-dimensional back-propagated error vector Δh is computed as

$$\Delta h_j = \begin{cases} m_j w_j \Delta f & \text{if } h_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

Then, the $32 \times (d+1)$ gradient array ΔW is computed as

$$\begin{aligned} \Delta W_{j,k} &= z_k \Delta h_j \quad \text{for } k = 1 \dots d \\ \Delta W_{j,d+1} &= \Delta h_j \end{aligned}$$

And finally, the back-propagated d-dimensional error vector Δz is computed as

$$\Delta z_k = \sum_{j=1}^{32} W_{j,k} \Delta h_j$$

Pooling. The pooling stage has no trainable parameters. To back-propagate through a max_pool stage, we must compute the $(n+m-1) \times d$ array ΔY from the d-dimensional vector Δz.

$$\Delta Y_{i,k} = \begin{cases} \Delta z_k & \text{if } i = \operatorname{argmax}(Y_{1,k}, \dots, Y_{n,k}) \\ 0 & \text{otherwise.} \end{cases}$$

To back-propagate through a $\mathrm{max\_avg\_pool}$ stage, incoming vector Δz is $2d$-dimensional and the computation is similar:

$$\Delta Y_{i,k} = \frac{\Delta z_{2k+1}}{n+m-1} + \begin{cases} \Delta z_{2k+0} & \text{if } i = \operatorname{argmax}(Y_{1,k}, \ldots, Y_{n,k}) \\ 0 & \text{otherwise.} \end{cases}$$

Rectification. To back-propagate through the rectification stage, we must compute a gradient vector Δb for the thresholds, and array ΔX from the $(n+m-1) \times d$ array ΔY.

$$\Delta b_k = \sum_{i=1}^{n+m-1} \Delta Y_{i,k}$$

$$\Delta X_{i,k} = \begin{cases} \Delta Y_{i,k} & \text{if } Y_{i,k} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Convolution. Since convolution is the first stage in forward-propagation, it is the last stage of back-propagation. The $d \times m \times 4$ gradient array ΔM is computed as

$$\Delta M_{k,j,l} = \sum_{i=1}^{n+m-1} S_{i+j,l}\, \Delta X_{i,k}$$

1.3 Training objective (loss functions, weight decay)

To train a neural network, including a DeepBind model, one must choose a *loss function* that is appropriate for the training data. Suppose we are given N training pairs $(s^{(1)}, t^{(1)}), \ldots, (s^{(N)}, t^{(N)})$ where $s^{(i)}$ is an input sequence (e.g. a PBM probe) and $t^{(i)}$ is the training target (e.g. a measured specificity score). Note that N may be a random subsample of a much larger training set. Let $f^{(1)}, \ldots, f^{(N)}$ be the corresponding DeepBind predictions $f^{(i)} = f(s^{(i)})$ using the current model parameters M, b, W, and w. At each training step, we aim to approximately minimize the following regularized objective:

$$\frac{1}{N} \sum_{i=1}^{N} \mathrm{LOSS}(f^{(i)}, t^{(i)}) \; + \; \beta_1 \|M\|_1 \; + \; \beta_2 \|W\|_1 + \; \beta_3 \|w\|_1$$

where $\|\cdot\|_1$ denotes the L1 norm (sum of absolute values) and each prediction $f^{(i)}$ is a function of parameters M, b, and W. Here the array W represents all multiplicative weight parameters in the neural network. The L1 norm is known as "L1 weight decay", and encourages weights to take value zero unless there is enough signal in the data to warrant paying the β penalties; the β values are automatically chosen by calibration.

When training a model to predict microarray (PBM) binding affinity measurements, we use the *mean squared error* (MSE) loss function:

$$\mathrm{MSE}(f, t) \; = \frac{1}{2}(f - t)^2$$

For ChIP and for SELEX data, each target is either $t = 1$ to indicate a preferred sequence or $t = 0$ to indicate a presumed background sequence. This is a two-class classification problem, so we use a standard *negative log-likelihood* (NLL) objective:

$$\text{NLL}(f,t) = -t\log\big(\sigma(f)\big) - (1-t)\log\big(1-\sigma(f)\big)$$

where σ is the logistic function

$$\sigma(f) = \frac{1}{1+e^{-f}}$$

The choice of loss function determines how each $\Delta f^{(i)}$ is computed from prediction $f^{(i)}$ and target $t^{(i)}$ during back-propagation; both MSE and NLL have similar form:

$$\Delta f_{\text{MSE}} = \frac{1}{N}(f-t)$$

$$\Delta f_{\text{NLL}} = \frac{1}{N}(\sigma(f)-t)$$

For each training example $i = 1 \dots N$ in a batch, the corresponding quantity Δf leads to independent gradient arrays $\Delta M^{(i)}$, $\Delta b^{(i)}$, $\Delta W^{(i)}$, and $\Delta w^{(i)}$. The final gradients for each group of parameters are:

$$\Delta M = \beta_1 \text{sign}(M) + \sum_{i=1}^{N} \Delta M^{(i)}$$

$$\Delta b = \sum_{i=1}^{N} \Delta b^{(i)}$$

$$\Delta W = \beta_2 \text{ sign}(W) + \sum_{i=1}^{N} \Delta W^{(i)}$$

$$\Delta w = \beta_3 \text{ sign}(w) + \sum_{i=1}^{N} \Delta w^{(i)}$$

where $\text{sign}(\cdot)$ is the element-wise sign function giving values from {-1,0,+1}.

1.4 Reverse-complement mode

When training a model on sequences derived from double-stranded DNA (PBM, SELEX ChIP-seq), it is unknown whether the protein bound to the input strand or whether it bound to the opposite strand. To handle this ambiguity, we evaluate a score for both the input sequence and its reverse complement using the same DeepBind model. The final output prediction for the sequence is the maximum of the two scores, to reflect the idea that a high binding score can be triggered by patterns in either strand. See **Supplementary Fig. 1** for a more accurate depiction of DeepBind feed-forward in this setting.

One important detail is that, during training, our implementation of dropout (see **Supplementary Sec. 1.1**) uses the same binary mask sample for both the original strand and its reverse complement. This ensures that, when an input is sent through the network, each z_k and h_j is either available for scoring both strands, or is unavailable (dropped out) for scoring both strands. Matching the binary masks significantly reduced the appearance of duplicate (but reversed) motif detectors in M, thereby making the models easier to interpret and making more efficient use of the allotted parameters.

2 Automatic training pipeline

Despite the recent success of deep learning, there remains a major, long-standing hurdle to widespread use: deep learning methods are still very sensitive to the many *calibration parameters*, also known as *hyper-parameters*, which must be expertly configured before training each model on each dataset. Some calibration parameters relate to the model architecture (number of layers, number of hidden units, type of non-linearity), some relate to the training objective (weight decay, sparsity), and others relate to the training algorithm itself (learning rate, momentum, batch size); our full list of parameters is explained in **Supplementary Sec. 1.1**. In machine learning this expert calibration is called "hyper-parameter search", or sometimes "grad student search." Without careful calibration, deep learning methods will either under-fit and fail to find any useful patterns, or will over-fit and fail to make accurate predictions on new data. Several machine learning methods like *support vector machines* (SVMs) and *random forests* are popular precisely because they often work reasonably well "out of the box," without heavy calibration or "black magic." In short, the need to hand-calibrate complex models is a major obstacle to accessible, reproducible research in deep learning.

To make deep learning practical for non-specialists, the calibration phase must be fully automatic. Recent work in machine learning is explicitly dedicated to automatic calibration[6–8]. Our calibration procedure is based on random sampling as suggested by Bergstra & Bengio[7]. **Supplementary Sec. 1.2** describes our calibration procedure in detail. All three of the above-cited calibration approaches require training tens or hundreds of similar models on the same data set, over and over, until a good calibration emerges. Each calibration trial uses a different set of calibration parameters, and is evaluated by the trained model's performance on held-out validation data. If the data and/or model is very large, it may take many hours, days, or even weeks of computation to find calibration parameters that work well on a given problem. These long time-scales are why GPU-accelerated implementations are becoming prominent in the deep learning community.

In our work, we trained a separate model for over 2603 different protein binding data sets, the vast majority of which are large-scale (≥10,000 sequences). Each data set required its own calibration phase involving 30 trials, so tens of thousands of models were trained in total. To train these models in reasonable time, we built a specialized GPU-accelerated implementation of DeepBind. Our implementation is built from the ground up in C++ and Python, with only low-level dependencies (CUDA and Numpy).

For DeepBind models, GPU acceleration poses a unique challenge because the models are much smaller (16 motif detectors, 32 hidden units) than comparable models used in computer vision[2]. Modern GPUs are *single instruction multiple data* (SIMD) architectures, and their speed only comes by massive data parallelism, not by faster calculations. For example, a GPU is comparable to a CPU at multiplying two 32x32 matrices, but is upwards of 20x faster at multiplying two 1024x1024 matrices, even compared to a 4-core CPU running the aggressively optimized Intel MKL libraries. When training a large neural network, that is the difference between hours and days of training time. However, a straightforward GPU implementation of neural networks will only bring speedups for very large models and, as we stated earlier, our DeepBind models are not individually large. In **Supplementary Sec. 2.4** we show how to leverage the power of GPUs in small-model scenarios, such as for DeepBind.

2.1 Calibration parameters

First we describe each calibration parameter and its purpose. Second, we explain the different types of random sampling used to generate these parameters at calibration time. **Supplementary Table 10** summarizes the complete list of calibration parameters and how their values are sampled.

For reference, we have reproduced a Python configuration file that determines the search space of a DeepBind calibration phase. The example below constrains the search space to models with one max-pooling followed by a 32-unit ReLU hidden layer.

```
deepbind_model = [
    motif_scan(num_motifs   = 16,
               motif_len    = 24,
               weight_decay = loguniform(1e-10,1e-3),
               init_scale   = loguniform(1e-7,1e-3),
               ),
    bias(),
    rectify(),
    maxpool(),
    full(num_units = 32,
         weight_decay = loguniform(1e-10,1e-3),
         init_scale   = loguniform(1e-5,1e-2),
    rectify(),
    dropout(expected_value = choice([0.5, 0.75, 1.0])),
    full(num_units = 1,
         weight_decay = loguniform(1e-10,1e-3),
         init_scale   = loguniform(1e-5,1e-2),
    bias(init_bias = -4.0),
]
```

Motif detector length. The motif detector length corresponds to the quantity m in **Supplementary Sec. 1.1**, which determines the number of unique positions in the motif detectors learned by the DeepBind model. For example, a model trained with motif_len=20 will have 80 trainable parameters for each motif detector. Note that this is a maximum detector length, in that shorter motifs can still be learned with irrelevant positions having small (near-zero) coefficients.

When the approximate motif detector length is known beforehand, we advise to choose a motif_len of ~1.5 the estimated length; these flanking positions are important because, in the early steps of learning, a motif pattern may begin to form close to the edge of the position range, and then get truncated before a full-length motif can be learned. As a concrete example, suppose a TF binds to GAAA and we train with motif_len=4. After several steps of gradient descent, the motif detector may settle on AAAN as predictive, due to the influence of random initialization; however, this length-4 motif detector is now stuck: there is no position remaining to learn that GAAA is even more predictive of binding.

For all RBP models we used motif_len=16 because the RNAcompete sequences are rather short (32-43nt) and most RBPs have short motifs. For all ENCODE and DREAM5 transcription factor models we used motif_len=24 because several TF models learn long patterns (for a few TFs we manually increased motif_len to 32). Each SELEX dataset comprised probe sequences of a fixed length: 14, 20, 30, or 40. Depending on the particular SELEX dataset, we used a fixed motif_len of 14, 20, 24, and 32 respectively.

Number of motif detectors. We use num_motifs=16 for all experiments. Deep neural networks, including DeepBind models, perform best when trained with more parameters (in our case, motifs) than is strictly required to model the data; this is due to the way models are randomly initialized at the start of training, and to the subsequent dynamics of gradient descent learning. For example, if a protein has only one true binding motif, training with num_motifs=1 will often fail to learn the correct pattern even with good calibration, whereas training with num_motifs=4 will succeed much more frequently because there are more avenues for the gradient-based learning to improve the model. Another consideration is that the 'dropout' technique, though highly effective, requires more motif detectors to perform well.

Neural network hidden layer. For each dataset we only consider the performance of two neural network configurations: either no hidden layer, or one hidden layer with 32 rectified-linear (ReLU) units. A hidden layer allows the network to learn more complex rules for combining motif detections, and also allows non-linear response to those detections at the final output. However, adding a hidden layer may lead to over-fitting, and such DeepBind models are more difficult to calibrate because they are sensitive to the extra "weight decay" and "initialization scale" parameters for the new layer. For many data sets the simpler network works best, so we always evaluate both configurations.

Learning rate. The learning rate of a neural network is a very important parameter that, in conjunction with momentum, determines the step size of the learning. A large learning rate may lead to oscillating or even divergent model parameters. A small learning rate may fail to make any useful progress, or may over-fit the data in some circumstances. The good range of learning rates is different for each combination of data set and model configuration. We evaluate learning rates in the range [.0005, .05].

Learning momentum. Momentum methods, also known as "accelerated gradient" methods, are a simple way to speed up learning by scaling the learning rate for each individual parameter being trained. Momentum is based on the observation that, if a particular parameter has been steadily increasing (decreasing) for several consecutive training steps, then it is likely to continue increasing (decreasing) in future steps, and so the step size for that one parameter should be temporarily scaled up. Much like the 'global' learning rate parameter, a high rate of momentum can cause learning to diverge, and the optimal rate is typically just below the unstable regime. However, in practice, even a conservative momentum rate (0.9) will drastically speed up training, and can even act as a regularizer that improves prediction performance of the final network. Several "accelerated gradient" schemes have been proposed in mathematical optimization; see[9] for a contemporary review in the context of deep neural networks. Our experiments use Nesterov momentum[10] with rate coefficient sampled from range [.95, .99].

Batch size. The stochastic gradient descent (SGD) method estimates the training objective gradient using only a subset of training examples. The batch size determines how many training pairs to sample for each parameter update step. If batch_size=1 then only a single example is used for each update, and if batch_size=n then all samples are used. For most machine learning problems, a "mini-batch" size of between 30 and 200 performs well. There are speed and performance advantages of training with "mini-batch" SGD instead of standard "full batch" GD; see[1] for a review of these advantages in the context of machine learning. We use batch_size=64 for all experiments.

Number of learning steps. For each calibration trial, we train a DeepBind model for 20,000 parameter update steps. At 4,000-step intervals, we evaluate the current performance of the trained model on held-out validation data, resulting in five performance ratings for each calibration trial. If a model's validation performance was better at step 8,000 than it was at step 20,000, this is taken as evidence that the model began to over-fit the data midway through training. The final model will be trained with the combination of calibration parameters and number of learning steps that performed best on validation, a variant of the "early stopping" regularization technique. The number of learning steps can therefore be thought of as an extra calibration parameter that is used to avoid over-fitting to the training data.

Initial weight scale. At the outset of training, the parameters of a neural network are typically initialized with small random weights. For DeepBind models, this also applies to the motif detector weights. The scale of these initial random values is extremely important for the success of deep models[9]. In our experience, DeepBind models may need different scales at each layer of the network (motifs, neural network) to perform well. If the initial scale is too small, then a DeepBind model fails to learn any useful patterns, or learns several copies of some primary motif. If the initial scale is too large, motif detectors may be heavily biased to certain patterns and it is difficult to find a learning rate that works. For each calibration trial, we initialize each weight by sampling from $\mathrm{Normal}(0, \sigma^2)$; for motif detectors, the calibration parameter σ_{motifs} is chosen from the interval [1e-7, 1e-3]; for neural network weights, the calibration parameter σ_{net} is chosen from the interval [1e-5, 1e-2].

Weight decay. We use separate weight decay coefficients (β values in **Supplementary Sec. 1.3**) for the motif detectors and for each neural network layer. In each case, the coefficient is sampled from the interval [1e-10, 1e-3]. If the overall weight decay is too small, the model may over-fit to the training data. If the weight decay is too large, the model may under-fit and fail to learn any useful patterns. Note that we do not apply weight decay to the thresholds b, nor to the neural network additive weight; when describing the training objective we use notation $\|W\|_1$ and $\|w\|_1$ for simplicity, but in practice we exclude the last element of w and the last element of each $W_{j,:}$ from those penalties.

Dropout expected value. For each calibration trial we choose a shared expected value for all dropout masks m_j. The expected value takes on one of three values: 0.5 (strong dropout), 0.75 (weak dropout), or 1.0 (no dropout).

Log uniform sampler. The log uniform sampler generates a real number in some interval $[a, b)$ as follows: sample $x \in [0,1)$ from a uniform distribution, and then return

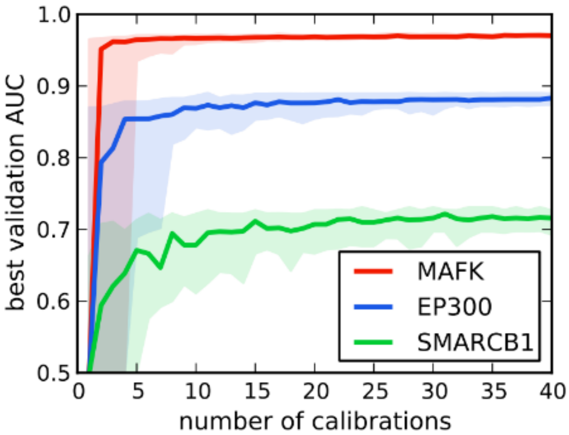
$$10^{(\log_{10} b - \log_{10} a)x + \log_{10} a}$$

Sqrt uniform sampler. The square root uniform sampler generates a real number in some interval $[a, b)$. Compared to log uniform, it is less aggressively skewed towards b. Again, we sample $x \in [0,1)$ uniformly, and then return

$$(b - a)\sqrt{x} + a$$

2.2 Calibration procedure and final training

The calibration phase begins by sampling 30 sets of random calibration parameters, which we will denote $\theta^{(1)}, \dots, \theta^{(30)}$. **Supplementary Table 10** lists the sampling used for each particular parameter in a set $\theta^{(i)}$. Though we sample each set $\theta^{(i)}$ independently[7], there exist sophisticated schemes to deliberated improve the expected quality of calibration samples as the calibration phase progresses[6,8]. In our early testing, we found that random sampling worked as well as these alternatives for DeepBind models, so we relied on the simplest approach. Below is a plot of the best calibration score (best validation AUC) seen so far as a function of the number of calibrations attempted; the median and the 10th/90th percentiles over 100 trials are shown below for three example TFs.



The calibration phase evaluates the quality of each parameter set $\theta^{(i)}$ by 3-fold cross-validation on the training set. Each model is trained on a different 2/3 of the data, and its performance is evaluated on the other held-out 1/3. The calibration parameters are scored by averaging these three values, as depicted in **Fig. 2b**. If the training task is regression (e.g. microarray-based models) then the evaluation metric is mean squared error; if the training task is classification, the metric is AUC with respect to a dinucleotide-shuffled background (**Supplementary Sec. 3**).

Once the best calibration parameters have been identified, we train six new models in parallel, using *all* the training data (a single fold). The model with the best *training* performance is then chosen as the final model returned by the entire pipeline. We train several models to ensure that the final training phase is not 'unlucky' due to random initialization effects and the stochasticity of the SGD training algorithm. Training performance is not, in general, a good indicator of a model's performance on test data because it may over-fit; however, the calibration phase automatically chooses enough regularization (weight decay, learning rate, momentum, dropout, early stopping) to mitigate the over-fitting problem for the final model.

2.3 Multi-model training on the GPU

We accelerated DeepBind training on the GPU by training several independent models at the same time on the same GPU. This is necessary because, for example, the GeForce GTX TITAN has 2,880 computation units but each unit is actually slower than a typical Intel processor. The speed of a GPU comes from all these units executing in parallel, but it can only do so if each job sent to the GPU has enough work to keep the majority of these computational units busy. If the GPU is loaded with many small jobs, then most of the GPU units sit idle and the speedups will be modest, or the GPU may even be slower than a CPU.

For example, for a single DeepBind model with num_motifs=20 and motif_len=24, our convolutional stage is ~10x faster on a GeForce GTX TITAN than on an Intel Core i7 3.5GHz running a single threaded C implementation. However, for an aggregate model with num_models=25, the TITAN is ~70x faster than the CPU implementation. These speedups are consistent with those reported in the literature[11].

Our implementation of DeepBind is built from scratch, with custom CUDA kernels to compute predictions (forward-propagation) and to do learning (back-propagation) on an aggregate model that contains an array of independent sub-models. **Supplementary Fig. 1** depicts an aggregate with num_models=3, but in practice we use num_models=30 for calibration and num_models=6 for final training (**Fig. 2a** depicts num_models=3 during final training, but this was to make the figure compact). The aggregate models' parameters are stored in an interleaved memory layout designed to work efficiently with modern GPUs' memory access constraints; specifically, reads and writes to GPU memory must be 'coalesced' in large contiguous chunks. For each input sequence, an aggregate model generates num_models output predictions, each of which are trained to predict the same target value. However, during training, each sub-model within the aggregate is trained with different calibration parameters, and so the accuracy of sub-models varies. Aggregate training is used to accelerate both the calibration phase and the final training phase, as illustrated in **Supplementary Fig. 1**. Once an aggregate model is trained, we evaluate the performance of each sub-model separately, and the final model's parameters are extracted so that they can be used to make predictions on new data. We use 32-bit floating-point for all calculations, as 64-bit floating point was slower but did not improve prediction accuracy.

3 Background sequences for ChIP and SELEX experiments

A ChIP or SELEX data set is essentially a list of foreground sequences for which the TF model is trained to predict a positive score. We must generate a corresponding set of background sequences for which the model will be trained to predict a negative score. We generate these background sequences by dinucleotide-preserving shuffle, or *dinuc shuffle*, applied to the original foreground sequences. A dinuc shuffled sequence preserves the count of all 16 dinucleotides (AA,AC,…,GT,TT) from the original sequence.

The advantage of a dinuc shuffled background is that it is more ‘difficult’ than standard nucleotide shuffling. By preserving dinucleotide counts, a model is prevented from simply relying on the low-level statistics of genomic regions, such as promoters or coding regions, to discriminate foreground from background. For example, the CG dinucleotide is extremely rare among ChIP peaks for many TFs, so a model can often achieve excellent training and testing performance (an AUC of 0.99) by learning to reject sequences containing CG, rather than learning TF-specific motifs. Moreover, GC-content in ChIP-seq peaks is considerably higher than average[12]: 61±5% for peaks close to transcription start sites versus 40% genome average. Wang et al. report that nearly 15% of 300bp regions flanking the 300bp ChIP-seq peak window had the SPI1 motif, suggesting that a background comprising sequences upstream or downstream of the peak might introduce SPI1 motifs into the background. Last, the effects of dinucleotide composition or dinucleotide steps on DNA structure[13], DNA bending rigidity or persistence length[14], stability and melting behavior[15], geometry of base pairs[16], and base stacking energies[17,18] are well-known.

Background set size. As is common practice in deep learning, we train all our binary classification models with balanced classes, i.e. as many background sequences as there are foreground sequences. However, a minority of ChIP and SELEX data sets contain only a small number (~80-300) of foreground sequences. In these extreme cases we found that a larger background size helped to avoid over-fitting to a small background. Our DeepBind ENCODE ChIP and SELEX models were trained with at least 10,000 background sequences generated by dinucleotide shuffle. If the data set has less than 10,000 foreground sequences, then we generate a new foreground set by randomly sampling 10,000 times with replacement from the pool of foreground sequences.

4 RNA-binding protein (RBP) models

Our main source of training and testing data is RNAcompete[19]. Here we describe the data collection, preprocessing, and evaluation methods, along with our rationale for relying on only a subset of *in vivo* experiments.

4.1 RNAcompete *in vitro* data and training

We use the normalized probe intensity data from RNAcompete[19]. The only non-linear preprocessing we did was to clamp probe intensities to be no larger than that of the 99.95th percentile intensity. The motivation for clamping was the observation that the top probe intensities for some well-known RBPs contained outliers. For example, MBNL binds to GCU-rich sequences, but the top two training sequences for MBNL report very large probe intensities despite containing no GCU motif at all. We do not claim that clamping is beneficial overall—for example, our QKI and Vts1p models would be better without clamping—but in a few isolated cases it helped our RBP models find better agreement with published motifs on well-known RBPs. We used the same clamping percentile for all RBPs.

When training a DeepBind model, the probe intensities undergo a linear transformation before being used as training targets. The linear transformation ensures each set of targets has mean 0.0 and variance 1.0. This is standard practice in neural network training because it allows all layers of the network to use consistent initialization scales and learning rates without regard to the scales of the training targets.

4.2 RNAcompete *in vitro* evaluation

We used several performance measures to assess the quality of DeepBind models. For each RBP, adapting ideas from[20] we first computed the *E*-scores, *Z*-scores, and AUCs for all possible 7-mers as described below.

***Z*-scores.** Let s_i denote the i^{th} probe in the array, and let $\mathcal{I}_k$ denote the indices of all probes that contain 7-mer k as a substring. The RNAcompete array design guarantees that $|\mathcal{I}_k| \geq 155$ for each possible k. Letting y_i denote the normalized intensity of s_i, we define the intensity of 7-mer k as the median intensity of all probes that contain it, that is $m_k = \text{median}([a_i]_{i\in\mathcal{I}_k})$. Taking the median of a large number of probe intensities ensures that a few noisy measurements cannot introduce any distorting effects. After computing intensities for all 7-mers, we transform them into *Z*-scores, Z_k. If k has a *Z*-score much larger than the bulk of the 7-mers, say $Z_k \geq 4$, it indicates that most probes containing k have higher intensities than the rest of probes. Ray et al. compute the PFM for RBPs by aligning 7-mers corresponding to the top ten *Z*-scores[19,21]. It should be noted that RNAcompete normalized intensities are both positive and negative and consequently, we cannot take the $\log(\cdot)$ of intensities as described in Berger et al.[20].

AUCs. Before describing *E*-scores, we explain how AUCs for each 7-mer are computed. Assuming there are n probes under consideration, we label each probe 1 if it contains k as a substring, and 0 otherwise. That is

$$l_i^{(k)} = \begin{cases} 1, & i \in \mathcal{I}_k \\ 0, & \text{otherwise} \end{cases}$$

Then, using vectors $L^{(k)} = \left[l_1^{(k)}, \ldots, l_n^{(k)}\right]$ as labels and intensities $Y = [y_1, \ldots, y_n]$ as predictions, we compute the AUC for k and call it A_k. If A_k is close to 1, it implies that the majority of the probes containing k have larger intensities than those that do not, hence k potentially has useful information about the sequence specificity of the target RBP.

***E*-scores.** Berger et al.[20] argued that AUCs computed as described above might be distorted due to a few noisy outliers. Indeed, in most array designs only a tiny fraction of probes contain a *k*-mer as a substring (usually less than 0.1%). To circumvent this issue, they discard half the probes as described below.

Let $\mathcal{I}'_k$ be the indices of all probes that do *not* contain k. Then the following two vectors are defined:

$$Y_k = [y_i]_{i\in\mathcal{I}_k}$$
$$Y'_k = [y_i]_{i\in\mathcal{I}'_k}$$

Then Y_k and Y'_k are each sorted and the bottom half of each is discarded. Let $\mathcal{T}_k$ be the original indices of the remaining probes (top halves of Y_k and Y'_k). The new label and intensity vectors are formed as:

$$\tilde{Y}^{(k)} = [y_i],\ \tilde{L}^{(k)} = \left[l_i^{(k)}\right], \qquad i \in \mathcal{T}_k$$

Now we compute the AUC using the new labels and intensities (as predictions), *E*-score, E_k, is simply AUC minus 0.5 (so a random prediction gets $E_k = 0$). Note that, unlike computing AUC in which all k had the same intensity vector, each k has a different intensity vector when computing *E*-scores.

Performance measures. RNAcompete probe data is split into nearly equally sized sets called SetA, $\mathcal{A}$, and SetB, $\mathcal{B}$. We train all models on probes in $\mathcal{A}$ and after training was fully done, we tested the models on probes in $\mathcal{B}$. Let $Y_\mathcal{A}$ and $Y_\mathcal{B}$ denote the observed intensities for probes in $\mathcal{A}$ and $\mathcal{B}$, respectively. Let $Y_\mathcal{B}^{(p)}$ denote the predicted intensities for probes in $\mathcal{B}$. For $Y_\mathcal{A}$, $Y_\mathcal{B}$ and $Y_\mathcal{B}^{(p)}$ we compute all 7-mer scores: *Z*-scores, *E*-scores and AUCs. For example, for $Y_\mathcal{A}$ these scores are called $Z_\mathcal{A}$, $E_\mathcal{A}$ and $U_\mathcal{A}$, respectively. Each vector holds the corresponding scores of all 7-mers, i.e., $Z_\mathcal{A} = [Z_{AAAAAAA}, \ldots, Z_{UUUUUUU}]$.

We compute the Pearson and Spearman correlations between the following pairs of vectors:

- $A_{\mathcal{B}}$ and $P_{\mathcal{B}}$: predicted and observed intensities for probes in $\mathcal{B}$.
- $E_{\mathcal{B}}$ and $E_{\mathcal{B}}^{(p)}$: *E*-scores computed from predicted and observed intensities for probes in $\mathcal{B}$.
- $Z_{\mathcal{B}}$ and $Z_{\mathcal{B}}^{(p)}$: Z-scores computed from predicted and observed intensities for probes in $\mathcal{B}$.
- $A_{\mathcal{B}}$ and $A_{\mathcal{B}}^{(p)}$: AUCs computed from predicted and observed intensities for probes in $\mathcal{B}$.

Note that while $Y_{\mathcal{A}}$ and $Y_{\mathcal{B}}$ hold intensities of different probes, $Z_{\mathcal{A}}$ and $Z_{\mathcal{B}}$ hold the *Z*-scores of the same 7-mers, thus, they can be compared with each other. We compute the Pearson and Spearman correlations between the following pairs of vectors, both of which are computed from observed intensities:

- $Z_{\mathcal{A}}$ and $Z_{\mathcal{B}}$
- $E_{\mathcal{A}}$ and $E_{\mathcal{B}}$
- $A_{\mathcal{A}}$ and $A_{\mathcal{B}}$

Supplementary Fig. 3 shows correlations (Pearson and Spearman) for all four measures, also provided in **Supplementary Table 2**.

4.3 RNAcompete *in vivo* evaluation (CLIP/RIP)

We downloaded the *in vivo* test datasets provided by Ray et al.[19]. The data sets comprise foreground and background sequences from several CLIP-seq, RIP-seq, RIP-chip, and PAR-CLIP experiments, compiled from several different sources. Note that several RBPs in that study contain multiple *in vitro* replicates and multiple *in vivo* test sets, so there are many combinations of (replicate, test set) on which to report the performance of a single RBP model, e.g. HuR has 5 replicates × 4 test sets = 20 possible AUCs. We reproduced the results in Fig 1C of Ray et al.[19] using their published motifs and test protocol. Specifically, for each RBP, they select the combination of (replicate, test set) that gives highest AUC and then report that result. Each test sequence is scored by scanning a PWM and summing the scores at each position. Note that the sum of scores correlates highly with sequence length, especially for short motifs; we also tried taking the average score across positions, but average performance of Ray et al. did not improve. The AUCs that we report for RNAcompete are slightly different from Ray et al. because we performed 100 bootstrap samples and show the mean and standard deviation. To evaluate DeepBind, we used the same specific data sets that performed best for Ray et al. PWMs. We score a sequence with DeepBind by averaging the scores of each subsequence of length 16.

See **Supplementary Fig. 4** and **Supplementary Table 3** for complete *in vivo* results. In **Fig. 3d** we only show the subset of RBPs for which the corresponding *in vivo* test sequences had average length $\leq$ 1000 (see **Supplementary Fig. 5** for corresponding ROC curves); this is because most RBPs have short motifs, and we argue that extremely long test sequences are not informative in that case. For reference, **Supplementary Fig. 4** also shows the best AUCs achievable by simple "base counts", where we rank test sequences by the enrichment of a single nucleotide (As, Cs, Gs or Us) or by sequence length. Base counts outperform both DeepBind and RNAcompete on several test sets. For example, the TAF15 and FUS data sets were excluded from analysis in[19], and ranking those sequences by 1/(fraction of Gs) gives AUC $\geq$ 0.95 for those data sets, and also for QKI and HuR. However, of the data sets for which either DeepBind or Ray et al. achieve higher AUC than any "base count" feature, DeepBind achieves much higher average AUC (0.85) than Ray et al. (0.76).

4.4 MatrixREDUCE software configuration

Set up. Following the DREAM5 TF challenge results[22], we contacted FeatureREDUCE authors for the code and proper parameter setting to run it on RBPs. However, we were advised to use MatrixREDUCE[23] instead [Harmen Bussemaker, personal communication]. We used the MatrixREDUCE 2.0 from the REDUCE Suite and ran it using the following parameters:

```
-topo_list up_to_octamers -p_value=0.001 –max_motif=20 –strand=1 –nstop=10
```

All of which except `–strand=1` (only scan the forward strand) are the default parameters.

Run times. On 244 RNAcompete RBPs, the median running time of MatrixREDUCE was nearly 15 minutes and the mean was nearly 110 minutes on Intel(R) Xeon(R) CPU E5-4650 @ 2.70 GHz. Moreover, the 10th and 90th percentile were nearly 10 minutes and two hours, respectively. The maximum time running for a single run was nearly three days.

Making predictions. MatrixREDUCE by default iteratively finds up to 20 PSAMs. It then fits a linear regression model using all of the computed PSAMs for accurate prediction of observed probe intensities. Since the MatrixREDUCE package does not perform the prediction for new sequences using all PSAMs [Harmen Bussemaker, personal communication], we wrote a script that loads the regression coefficients from "model_coeff.tsv" generated by MatrixREDUCE and linearly combines the scores from each of the PSAMs.

5 DREAM5 transcription factor (TF) binding models

We evaluated DeepBind models on the DREAM5 TF-DNA Motif Recognition Challenge using the revised 2013 evaluation protocol by Weirauch et al.[22]. The aim of the TF-DNA challenge is to learn the sequence specificity of 66 TFs. The data set comprises a complete set of PBM probe intensities for two distinct microarray designs named HK and ME. The evaluation protocol is to train TFs 1..33 on HK probes, and test these TFs on the corresponding ME probes, then train TFs 34..66 on ME probes, and test them on the corresponding HK probes. Finally, a subset of TF models are evaluated by their overall ability to rank 500 ChIP-seq peaks against three different backgrounds: dinucleotide shuffled sequences, random genomic sequences, and random promoter sequences upstream of the peak.

5.1 PBM data preprocessing

A major component of the 2013 revised challenge is comprehensive preprocessing of the probe intensities. Each competing algorithm is evaluated using the best combination of up to 8 preprocessing steps (c.f. Weirauch et al.[22], Supplementary Note 3). We used the spatially de-trended data provided by the organizers, which was reported to improve the performance of all algorithms. For DeepBind we only evaluated two additional pre-processing possibilities from the DREAM5 paper: subtracting each probe's median intensity, or dividing by each probe's median intensity. The per-probe normalization is intended to factor out biases in the microarray design or experiments; we computed each probe's median intensity across all 66 experiments for that microarray design. Dividing by median intensity improved the overall PBM performance of DeepBind from 3rd place (no pre-processing) to 1st place overall.-

5.2 DREAM5 PBM training and evaluation

We followed the revised 2013 training and evaluation protocol by Weirauch et al.[22]. Each algorithm's final score is the average of Pearson correlation-based score and AUC-based score. Both of these component scores are computed relative to the best-performing algorithm, so an algorithm that performed best for both would receive a final DREAM5 PBM score of 1.0. **Supplementary Table 1** contains the DREAM5 probe intensity correlations and AUCs achieved by each DeepBind TF model ("DREAM5 PBM scores"), and shows the individual scores of top-performing algorithms from Weirauch et al. (c.f. Table 2) along with their final score after being re-scaled to include DeepBind's overall score.

5.3 DREAM5 ChIP-seq evaluation

To evaluate an algorithm's robustness across technologies, Weirauch et al.[22] also evaluated several PBM-trained models on 500 ChIP-seq peaks of length 100 and length 50. We evaluated DeepBind using the 5 ChIP-seq data sets for which there was a corresponding PBM experiment. **Supplementary Fig. 2a** shows the average AUC of each PBM-trained TF model across all three backgrounds (see Weirauch et al.[22], Online methods); **Supplementary Table 1** contains the "DREAM5 ChIP AUCs" for individual backgrounds in this experiment. **Supplementary Fig. 2b** shows the average 3-fold cross validation performance of DeepBind when trained directly on the length-100 ChIP-seq data with dinuc shuffled background, along with ChIPMunk[24] and MEME-ChIP[25] performance using 10-fold cross validation as reported by Weirauch et al.

6 ENCODE ChIP-seq TF binding models

6.1 Data, training, and evaluation

We downloaded all peaks for 506 different ENCODE ChIP-seq experiments from Uniform Peaks of Transcription Factor ChIP-seq from ENCODE/Analysis available at

http://hgdownload.cse.ucsc.edu/goldenPath/hg19/encodeDCC/wgEncodeAwgTfbsUniform/

Details of the analyzed ChIP-seq experiments and their "Table Names" are provided in **Supplementary Table 4**. These experiments represent 137 unique TFs. We only considered experiments with no experimental treatment and a "Good" quality annotation. Each ChIP-seq data set was given a name using the scheme *TF_CellLine_Antibody_Lab*, e.g., "GATA2_K562_eGFP-GATA2_UChicago." Each positive (foreground) sequence was a 101nt extraction from hg19 genome centered at the reported peak. For each experiment, we ranked all ChIP-seq peaks by their coverage, with the most confident peaks ranked at the top.

We trained both DeepBind and MEME-ChIP models on the top 500 odd-numbered peaks, and then evaluated the models on the top 500 even-numbered peaks and reported the results in **Fig. 3e**. See **Supplementary Fig. 6** for a careful illustration of what we mean by "top 500 odd" and "top 500 even". We also trained DeepBind using the remaining peaks (with rank >1000), and demonstrated that training with these extra peaks (denoted DeepBind* in **Fig. 3e**) improves test performance on the top 500 even peaks. Note that the 506 ChIP-seq data sets represent only 137 unique TFs; **Supplementary Table 5** includes the median AUCs when these data sets are grouped by TF name or by cell line, and the overall relative performance of DeepBind and MEME-ChIP is similar regardless of grouping, specifically: (*i*) over-represented TFs (e.g. CTCF) are not dominating the average performance, (*ii*) over-represented cell-lines are not skewing the performance measures, and (*iii*) DeepBind performs best overall regardless of the method or grouping used for summarizing the performance measures. Below we detail how DeepBind and MEME-ChIP were trained and applied to the test set.

DeepBind training. To train DeepBind models on ChIP we performed the calibration procedure described in **Supplementary Sec. 2**. We used motif_len=24 to train all TF models, except RAD21, ZNF274, ZNH143 and SIX5 for which we used motif_len=32 because their motif detectors learned particularly long patterns. Since DeepBind is a discriminative classification model we generated background sequences using dinucleotide shuffling as described in **Supplementary Sec. 3**. We use the NLL training objective (**Supplementary Sec. 1.3**) with peaks having target of 1 and background sequences having a target of 0.

MEME-ChIP training. We used the top 500 odd peaks as input to MEME-ChIP, which (adapting ENCODE paper's parameter setting[12]) produces up to 5 PWMs (with `--meme-nmotifs 5`) named M1, M2, M3, M4, and M5. These PWMs are ranked by statistical significance according to a generative background model assumed by MEME, where M1 is the most statistically significant PWM. MEME does not specify how to use these 5 PWMs to score a new test sequence with regard to binding affinity. We considered six ways to score a test sequence: M1-SUM (scan M1 and take sum of per-position scores), M1-MAX (scan M1 and take max of position scores), SUM-SUM (scan each M1..M5, sum over positions, then sum of five scores), MAX-SUM (scan M1..M5, max over positions, then sum of five scores), SUM-MAX (scan M1..M5, sum over positions, then max of five scores), and MAX-MAX (scan M1..M5, max over positions, then max of five scores). In **Fig. 3e**, "MEME-M1" refers to the M1 SUM strategy because it performed best overall between the two alternatives using M1 only. "MEME-SUM" refers to the SUM-SUM strategy because it performed best overall among the four alternatives that use all of M1..M5.

Run times. The median and mean running time of MEME-ChIP (only the MEME module) were nearly 16 minutes on Intel(R) Xeon(R) CPU E5-4650 @ 2.70 GHz. The standard deviation of running times was nearly 100 seconds. Note that MEME-ChIP has a default maximum number of input sequences of 600, and maximum total number of input bases of 100,000. The median number of input sequences for each ChIP-seq data set is 17,000. Considering MEME's $O((nl)^2w)$ running time[26] (n, l, w denote number and length of the input sequences, and motif width, respectively), we do not report results for MEME-ChIP using "all remaining" peaks because even with 10,000 peaks the program did not terminate after 24 hours of CPU time.

6.2 Criteria for choosing a representative TF model

There are 506 ENCODE ChIP-seq data sets, but these represent 137 distinct TFs, each performed with a different cell line, antibody, and/or lab. The quality of the data sets varies. To use DeepBind ChIP-seq models in our disease-causing variant analysis experiments (**Supplementary Sec. 9**), we first chose one 'best' representative model for each of the 137 TFs based on subjective criteria. Our main criterion for choosing a ChIP-based model is that its motif detectors should respond to binding sites for the TF of interest, and not respond to binding sites for unrelated or co-factor TFs. ChIP-seq data sets are known to contain binding sites for unrelated TFs[27]. Both DeepBind and MEME have the capacity to learn these 'extra' motifs, and for a large proportion of ChIP experiments, we observed this phenomenon.

For example, our DeepBind GATA2_HUVEC_GATA-2_USC model learned to recognize the core GATAA motif plus the AGGAA motif recognized by ETS1 and the TGANTCA AP1 motif; the corresponding ChIP analysis in FactorBook (factorbook.org) found nearly identical motifs using MEME-ChIP. Instead of the HUVEC cell line data set, we chose the K562 experiment for GATA2 (GATA2_K562_GATA-2_USC) as the representative DeepBind model in our disease-causing variant analysis. It was not always possible to find a dataset that contained only the core motif. Examples of unrelated motifs appearing in our final ChIP-seq models can be seen in EBF1 and PRDM1 shown in **Fig. 6**. These motifs could be easily removed from a DeepBind model by removing specific motif detectors after training (by inspection), but we did not do any post-processing/cleaning of DeepBind models to account for the 'unrelated motif' effect.

6.3 Evaluation of FoxA2 model on EMSA data

We acquired the 64 FoxA2 binding probes that were used for EMSA verification in Levitsky et al.[28] [V. Levitsky, personal correspondence]. We applied our sole FOXA2 model (FOXA2_HepG2_FOXA2-(SC-6554)_HudsonAlpha) to these sequences, outputting a binding score for each. We were also provided with the 64 probe scores for several other FoxA2 ChIP-trained models that were originally evaluated in[28]. The other methods are ChIPMunk[24], diChIPMunk[29], JASPAR[30] (PFMs MA0047.2 and

MA0148.1), TRANSFAC[31] (PFMs M01261 and M01012), and SiteGA[32]. **Supplementary Fig. 7** shows the Spearman correlation of each method's 64 scores and the measured EMSA scores); we used Spearman instead of Pearson because a ChIP-trained model is essentially a classifier and its score should be viewed as a measure of confidence, not as an estimate of actual binding affinity.

7 HT-SELEX TF binding models

To explore the effect of using different in vitro data for training, we used data from *high-throughput systematic evolution of ligands by exponential enrichment* (HT-SELEX), a prominent in vitro technique for profiling TF-DNA interactions. In HT-SELEX, all *k*-mers of a specific length are iteratively selected[33], so that the fraction of TF-bound sequences grows until only very high-affinity sequences are enriched. Compared to PBMs, the *k*-mers are significantly longer (14 to 40 vs. 10 for PBMs), so they can be used to learn longer motifs. However, learning TF binding models from HT-SELEX data is challenging because there are usually hundreds of thousands of short sequence reads and ascertaining their quality is difficult.

7.1 Data and training

We downloaded 1786 HT-SELEX data sets provided by Jolma et al.[34], representing 461 distinct TFs from the Sequence Read Archive (SRA ID: ERP001824). The complete list of all SRA experiment IDs, factor names, together with ligand, cycle, and batch information are provided in **Supplementary Table 6**. Each SELEX experiment was named using the scheme *TF_CloneType_PrimerName_Cycle_Batch*, where the CloneType is either DBD (DNA binding domain) or FL (full length). For example, the DeepBind model named CTCF_FL_TAGCGA20NGCT_3_AJ was trained on the SELEX sequences for factor CTCF, full length, with a 20 nucleotide variable sequence (20N), cycle 3, and batch AJ. Unlike ChIP-seq peaks, SELEX probes are not ranked by confidence, and so there is no notion of training on the "top 500."

Jolma et al. provide a scheme to choose the cycle that will result in the best PFM using their semi-automatic algorithm[34]. For our "SELEX models" comparisons in **Fig. 3e**, we use DeepBind models trained on the same cycle as reported by Jolma et al. We noticed that several of the resulting models contained unwanted biases, e.g., a preference for CCCNCCCNCCC motifs across many unrelated TFs. So, we trained models on all HT-SELEX data sets and found that, in some cases, training on a lower cycle eliminated these artifacts. So, for all other experiments with SELEX models, including **Fig. 3f** and the variant analyses in **Fig. 4** and **Supplemental Sec. 9**, we used the CloneType/PrimerName/Cycle for each TF that seemed to give the best representative DeepBind model according to the subjective criteria outlined in **Supplementary Sec. 7.2**.

DeepBind training. To train DeepBind models on SELEX we performed the calibration procedure described in **Supplementary Sec. 2**. Each particular SELEX data set has foreground sequences of a fixed size, indicated by the PrimerName field (14N, 20N, 30N or 40N). We used a different motif_len of 14, 20, 24 and 32 for each of these cases respectively, as noted in **Supplementary Table 10**. Since DeepBind is a discriminative classification model, we generated background sequences by dinucleotide shuffling as described in **Supplementary Sec. 3**. We use the NLL training objective (**Supplementary Sec. 1.3**) with a target of 1 for all SELEX probes, and a target of 0 for all synthetic background sequences.

MEME-ChIP training. We trained MEME-ChIP on SELEX in much the same way as for the ENCODE ChIP-seq data sets, and evaluated performance using both MEME-M1 and MEME-SUM (M1..M5). The main difference for SELEX is that the probe sequences are shorter (14-40nt) than ChIP peaks (101nt), and so MEME was able to process a larger number of the available foreground sequences. For each TF we tried both 1,000 random foreground sequences (MEME-SUM-1k and MEME-M1-1k) and 5,000 random foreground sequences (MEME-SUM-5k and MEME-M1-5k).

Run times. For MEME-1K the median and mean run time were nearly three minutes, and the standard deviation was 165 seconds. However, for MEME-5K, the mean and median were nearly 12 hours and the standard deviation was 115 minutes. The 90th percentile of running times was more than 13 hours (for 30nt and 40nt sequences).

7.2 *In vitro* evaluation

We applied DeepBind to HT-SELEX data for 303 human DNA binding domains (DBDs), 84 mouse DBDs and 151 full-length (FL) human TFs, representing 411 different TFs[34]. For training, the raw HT-SELEX sequences for all TFs and all cycles (1786 in total) were used as positives and dinucleotide-shuffled sequences were used as negatives. To compare DeepBind to the PWMs obtained by Jolma et al., we used the same 566 experiments that they hand-selected for their PWM discovery algorithm. DeepBind achieved higher AUC (0.71 average) than those obtained by PWMs (0.63 average; *P*=2.8e-94, Wilcoxon one-sided rank test, *n*=566) (**Fig. 3e**; **Supplementary Table 7**). The Jolma PWMs were used to score each position on the forward and reverse strands, and sequences were padded with "N" [.25,.25,.25,.25] at both ends because this improved performance dramatically. For Jolma et al. each sequence's final score was the average of all per-position PWM scores; in cases where multiple PWMs were provided (i.e. multiple 'seeds'), we report only the PWM that gave best test performance; summing the scores of alternative PWMs for Jolma et al. nearly always performed worse, unlike the case for MEME.

7.3 *In vivo* evaluation

To test the performance of DeepBind models in vivo, we next examined the HT-SELEX TFs for which we also had ENCODE ChIP-seq data (n=35). We tested the HT-SELEX-derived DeepBind models by generating scores for the top 500 even-numbered ChIP-seq peaks, and then compared the resulting AUCs to those achieved by PWMs from Jolma et al. and by MEME-ChIP. The HT-SELEX sequences were only 14–40nt long, so we were able to input up to 5,000 sequences to MEME-ChIP (MEME-M1-5k and MEME-SUM-1k). We also evaluated the effect of using only 1,000 sequences (MEME-M1-1k and MEME-SUM-1k). PWMs were applied the same way as the *in vitro* evaluation above, with the best-performing Jolma PWM used. DeepBind achieves higher AUC (0.85 average) than Jolma et al. in 27 out of 35 experiments, and better than the best MEME-ChIP result (MEME-SUM-5k) in 26 out of 35 experiments, suggesting that DeepBind was able to generalize from HT-SELEX to other data acquisition technologies (**Fig. 3f**; **Supplementary Table 8**). Note that each method performed better than other methods on at least one TF. For example, for MEF2A DeepBind performed worst; for GAPBA the semi-automatic method of Jolma et al. had by far the best AUC (0.97) on the ChIP data set despite a poor AUC (0.51) on the original HT-SELEX training data for their GAPBA PWM. The DeepBind SELEX models for CTCF, PAX5, PRDM1, GATA3, MAX, and ZNF143 performed significantly better than their MEME-SUM-5k counterparts. No method completely dominates, but DeepBind was best overall, which we found surprising for a first attempt at applying this general-purpose machine learning framework.

7.4 Criteria for choosing a representative TF model

For any one TF, there may be several HT-SELEX experiments with different CloneType, PrimerName, and/or Cycle number. To use DeepBind HT-SELEX models in our disease-causing variant analysis experiments (**Supplementary Sec. 9**), we first chose one 'best' representative model for each of the 461 TFs based on subjective criteria. Unlike ChIP-seq data, SELEX experiments do not contain motifs from unrelated factors. However, there were two competing artifacts governing the quality of data and thereby the quality of the model. The first artifact is a well-known bias in SELEX, where a large proportion of foreground probes in the early cycles (typically cycles 2 and 3) are essentially random, with no discernible motif(s). For example, the core motif for ATF7 appeared in only 0.2% of sequences for cycle 2, 2.8% for cycle 3, 13% for cycle 4, 23% for cycle 5, and 21% for cycle 6. The second artifact we observed is a preference for CCCNCCCNCCC motifs in the later cycles, regardless of which TF is associated with the experiment. The authors [Jolma et al., 2013] also observed this CCCNCCCNCCC trend, but it is not well-understood [Arttu Jolma, personal communication]. For each TF we therefore chose a representative DeepBind model by inspecting the clarity of motif detectors, but avoiding any model that contained a CCCNCCCNCCC detector when possible. MEME-ChIP also learned these CCCNCCCNCCC motifs, often reporting it as the most significant motif (M1), so we used our alternate CloneType/PrimerName/Cycle selection for MEME-ChIP as well.

8 Splicing regulation experiments

For TIA, hnRNP C, PTBP1, and Mbnl we used DeepBind models trained on Ray et al.'s data[19], and for Nova we used a model trained using CLIP clusters from[35]. We used the regulated exons curated in Cereda et al.[36] for all RBPs except for Mbnl, for which we used the regulated exons from Wang et al.[37]. We considered four regions, all of the same size: two from the 3' flanking intron, and two from the 5' flanking intron. We scored a region by applying DeepBind to each subsequence of length 16 (the default length of our RBP motif detectors) and taking the average of those scores (**Fig. 5**). The information regarding the analysis is listed in **Supplementary Table 9**. We excluded TDP43 data from Cereda et al., since the total number of Up-/Down-regulated exons was 80 but it had 8,000 controls (ratio 1:100); the next dataset in terms of disparity between regulated/control exons was Nova (ratio 4.1:100). It also should be noted that our approach is strikingly different from that taken by Cereda et al.: they use the regulated exons to learn the binding motifs, while we use the in vitro- and in vivo-learned models to explain RBP-mediated splicing regulation modes.

9 Quantitative analysis of disease-causing variants

We trained a model using the simulated and observed SNVs from the CADD framework[38]. We call this model DeepFind, because it uses it uses deep learning to find putatively deleterious variants. In the following we describe the steps involved.

9.1 CADD SNVs

Kircher et al. developed a system that simulates *de novo* germline mutations[38]. This model, inspired by a General Time Reversible (GTR) model, incorporates a separate rate for CpG dinucleotides to correct for the heavily skewed CpG distribution in the genome. The simulator is available from http://cadd.gs.washington.edu/simulator. The 'simulated' SNVs are used as positives or potentially deleterious SNVs. For negatives, they used derived alleles (with regard to the human-chimp ancestral genome from the Ensembl EPO 6 primate alignments) for which the observed frequency was above 95% in the 1000 Genomes data. This high allele frequency threshold ensures that only mutations that passed many generations of natural selection and became fixated were analyzed. Kircher et al. kindly provided us with 44,182,238 simulated SNVs and 14,893,290 observed SNVs [Martin Kircher, personal correspondence].

9.2 Mapping SNVs to promoters

We used ANNOVAR[39] to map observed and simulated SNVs to RefSeq genes' core promoters. We defined core promoters as the sequence stretch up to 2,000 bases upstream of the transcription start site (TSS). Moreover, we filtered all SNVs that overlapped any transcribed part of the genes, since such a SNV might induce coding disruption or splicing aberration. A total of 769,840 SNVs mapped to promoters (simulated: 598,602, observed: 171,238).

9.3 DeepFind model structure

The structure of the DeepFind model is depicted in **Supplementary Fig. 9**. For each SNV, we extract the 51bp of genomic sequence centered at the SNV and call it the wild type sequence. Next, we apply the mutation to the sequence and call it the mutant sequence (the wild type and mutant match at 50 positions and differ at only one position). Then, we pass the wild type and the mutant sequence through 596 high quality HT-SELEX and ChIP-seq DeepBind models, resulting in 596x2=1192 input features. We used 5-fold cross-validation (training: 3 parts, validation: 1 part, testing: 1 part) and trained a neural network with 500 hidden units on the training folds, using early stopping by the performance on the validation fold. Each model takes less than an hour to train (number of training SNVs is nearly 462,000). Then we made the predictions for the test fold. We repeated this process for all folds.

Using only the TF binding scores DeepFind achieves AUC of 0.71. When we added two more features (distance to the closest TSS and transversion/transition) the AUC increased to 0.73. Last, we added nine features derived from conservation tracks: mammalian, vertebrates, and primates PhyloP and PhastCons data[40,41]. For PhyloP tracks, we used both the value at the SNV and the averaged value across the 51bp window centered at the SNV. Including these features, AUC increased to 0.76. It should be noted that we did not exclude human genome from PhyloP and PhastCons tracks, whereas CADD removes human from all conservation tracks. **Supplementary Fig. 10** depicts the distribution of observed and simulated SNV DeepFind scores.

10 Visualizations for DeepBind models

10.1 Mutation maps (as seen in Fig. 4)

Given an input sequence $s = (s_1, \ldots, s_n)$ of length n and a particular DeepBind model, we can visualize the model's sensitivity to mutations in s by generating a $n \times 4$ matrix ΔS, which we call a *mutation map*. **Fig 4.** uses mutation maps (transposed to be $4 \times n$) to show how several SNPs, including the known disease-associated SNVs highlighted from HGMD and COSMIC 69, can destroy or create binding sites for a specific TF. Above each mutation map we show the original sequence s being evaluated for sensitivity; as a visual aid, we scale the height of each base s_i by that position's ability to damage binding (decrease the binding score), thereby providing a kind of *interaction logo* for that input sequence. In **Fig. 4**, the scales (red/blue) of each mutation map are scaled independently for each TF.

There are several possible ways to define a mutation map ΔS, but we found that the following provided the clearest visualization in most cases. First we compute the score $p(s)$ of the original sequence s. For ΔS_{ij} we mutate s by replacing s_i with base b_j where $b = (\mathrm{A,C,G,T})$, giving a sequence $\hat{s}$. We then compute the score $p(\hat{s})$ and the final sensitivity of the $s_i \rightarrow b_j$ variant is assign to be $\Delta S_{ij} = (p(\hat{s}) - p(s)) \cdot \max(0, p(s), p(\hat{s}))$. The second term in the product is designed to ignore variations in the "no binding" regime, and to emphasize score changes that either destroy a strong binding site (large $p(s)$) or create a strong binding site (large $p(\hat{s})$).

A final but important detail is how to generate mutation maps for long sequences, specifically when s is long enough to contain several binding sites that each need to be recognized independently. In that scenario, scoring the entire sequence with a single DeepBind evaluation can provide a misleading mutation map: if two binding sites exist, and the mutation destroys one, the DeepBind model will find the other binding site and thereby report a large score for $p(\hat{s})$ even if one binding site is destroyed. Instead, we first extract every length-m subsequence $s_i, \ldots, s_{i+m-1}$ of s where m is the size of the motif detectors (motif_len) for the DeepBind model. We then generate a separate $m \times 4$ sensitivity map $\Delta S^{(i)}$ for each of these $n + m - 1$ subsequences, using the procedure already described. To compute the final $n \times 4$ sensitivity map ΔS we align each $\Delta S^{(i)}$ to start at position i and then average the 1×4 submatrices that overlap each position.

Note that these sensitivity maps can easily be extended beyond the four SNV tracks that we show in our figures. For example, one could add a 'del' track and four additional 'ins' tracks to predict the effect of single base pair del/ins events.

10.2 Sequence logos (as seen in Fig. 6)

Here we describe how the sequence logos in Figure 6 were generated. To visualize the pattern learned by a particular DeepBind motif detector M_k, we generate a PWM derived from the detector's response to actual sequences. Specifically, we feed all sequences from the test set (foreground and background) through the convolutional and rectification stages of the DeepBind model, and we align all the sequences that passed the activation threshold, i.e. some $Y_{i,k} > 0$ for at least one position i. Once the sequences are aligned, we generate a position frequency matrix (PFM) and transform it into a sequence logo in the standard way.

Details of the alignment and PFM calculation are as follows. Let s be a test sequence, and let Y be the corresponding rectified motif scans as described in **Supplementary Sec. 1.1**. For motif k we only consider sequence s if $Y_{i,k} > 0$ for some position i. We find the position $j = \mathrm{argmax}_i\, Y_{i,k}$ where motif M_k gives of maximum response, and extract subsequence $s_{j-m+1 \dots j}$ of length m where m is the length of the tunable motifs in the DeepBind model; if these subsequence indices go beyond the extents of s, we replace these positions with a special 'empty' character which does not contribute to the final PFM counts. Once all the subsequences are extracted, they are stacked and the nucleotide frequencies are counted to compute a PFM of length m. From this PFM we compute a sequence logo, and we trim the low-information positions from the start and end of the logo before visualization. We also tried weighting the contribution of each sequence by the magnitude of $Y_{i,k}$ but this did not have a strong effect on most models so we used the simpler procedure.

The references for the logos used in Figure 6 are as follows:

- HNRNP A1[42]
- PTBP1[19]
- ZC3H10[19]
- CTCF[43]
- Sox10[44]
- Tp53[45]
- Pou2f2[46]
- PRDM1[47]
- EBF1[44]
- NR4A2[44]

10.3 Motif detector matrices (as seen in Fig. 2a)

We visualize an $m \times 4$ motif detector M by showing it as a $4 \times m$ matrix with a color map. For example, a small "GCAC" detector can be seen in **Fig. 2a** using a red/blue color map. Before display, we manipulate the coefficients by subtracting the mean from each column (each motif position) of the matrix. The resulting visualization is more easily interpreted, and is equivalent to the original motif detector in that it would respond to input sequences in exactly the same way.

Subtracting manipulation is equivalent because, for any quantity δ we subtract from a column of M, the subsequent bias in the rectification step can be manipulated as $b' = b + \delta$ to compensate. A model with manipulated motif detector would make exactly the same predictions as the original, so long as for each row of the input matrix S we have $\sum_j S_{i,j} = 1.$, which means an additive constant

11 References

1. LeCun, Y. *et al.* Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.* **1,** 541–551 (1989).

2. Krizhevsky, A., Sutskever, I. & Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. in *Adv. Neural Inf. Process. Syst.* 1097–1105 (2012). at <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>

3. Zeiler, M. D. *et al.* On rectified linear units for speech processing. in *2013 IEEE Int. Conf. Acoust. Speech Signal Process.* 3517–3521 (IEEE, 2013). doi:10.1109/ICASSP.2013.6638312

4. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **15,** 1929–1958 (2014).

5. Duda, R. O., Hart, P. E. & Stork, D. G. Pattern Classification (2nd Edition). (2000). at <http://dl.acm.org/citation.cfm?id=954544>

6. Bergstra, J., Bardenet, R., Bengio, Y. & Kégl, B. Algorithms for Hyper-Parameter Optimization. in *Adv. Neural Inf. Process. Syst.* 2546–2554 (2011). at <https://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf>

7. Bergstra, J. & Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* **13,** 281–281–305–305 (2012).

8. Snoek, J., Larochelle, H. & Adams, R. Practical Bayesian Optimization of Machine Learning Algorithms. *NIPS* 1–9 (2012). at <https://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf>

9. Sutskever, I., Martens, J., Dahl, G. & Hinton, G. On the importance of initialization and momentum in deep learning. *JMLR W&CP* **28,** 1139–1147 (2013).

10. Nesterov, Y. A method of solving a convex programming problem with convergence rate O (1/k2). *Sov. Math. Dokl.* **27,** 372–376 (1983).

11. Raina, R., Madhavan, A. & Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. in *Proc. 26th Annu. Int. Conf. Mach. Learn. - ICML '09* 1–8 (ACM Press, 2009). doi:10.1145/1553374.1553486

12. Wang, J. *et al.* Sequence features and chromatin structure around the genomic regions bound by 119 human transcription factors. *Genome Res.* **22,** 1798–812 (2012).

13. Geggier, S. & Vologodskii, A. Sequence dependence of DNA bending rigidity. *Proc. Natl. Acad. Sci. U. S. A.* **107,** 15421–6 (2010).

14. Zheng, G., Colasanti, A. V, Lu, X.-J. & Olson, W. K. 3DNALandscapes: a database for exploring the conformational features of DNA. *Nucleic Acids Res.* **38,** D267–74 (2010).

15. Breslauer, K. J., Frank, R., Blöcker, H. & Marky, L. A. Predicting DNA duplex stability from the base sequence. *Proc. Natl. Acad. Sci. U. S. A.* **83,** 3746–50 (1986).

16. El Hassan, M. A. & Calladine, C. R. The assessment of the geometry of dinucleotide steps in double-helical DNA; a new local calculation scheme. *J. Mol. Biol.* **251,** 648–64 (1995).

17. Delcourt, S. G. & Blake, R. D. Stacking energies in DNA. *J. Biol. Chem.* **266,** 15160–9 (1991).

18. Ornstein, R. L. & Rein, R. An optimized potential function for the calculation of nucleic acid interaction energies I. base stacking. *Biopolymers* **17,** 2341–60 (1978).

19. Ray, D. *et al.* A compendium of RNA-binding motifs for decoding gene regulation. *Nature* **499,** 172–7 (2013).

20. Berger, M. F. *et al.* Compact, universal DNA microarrays to comprehensively determine transcription-factor binding site specificities. *Nat. Biotechnol.* **24,** 1429–35 (2006).

21. Ray, D. *et al.* Rapid and systematic analysis of the RNA recognition specificities of RNA-binding proteins. *Nat. Biotechnol.* **27,** 667–70 (2009).

22. Weirauch, M. T. *et al.* Evaluation of methods for modeling transcription factor sequence specificity. *Nat. Biotechnol.* **31,** 126–34 (2013).

23. Foat, B. C., Morozov, A. V & Bussemaker, H. J. Statistical mechanical modeling of genome-wide transcription factor occupancy data by MatrixREDUCE. *Bioinformatics* **22,** e141–9 (2006).

24. Kulakovskiy, I. V, Boeva, V. A., Favorov, A. V & Makeev, V. J. Deep and wide digging for binding motifs in ChIP-Seq data. *Bioinformatics* **26,** 2622–3 (2010).

25. Machanick, P. & Bailey, T. MEME-ChIP: motif analysis of large DNA datasets. *Bioinformatics* **27,** 1696–1697 (2011).

26. Bailey, T. L. & Elkan, C. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. *Proc. Int. Conf. Intell. Syst. Mol. Biol.* **2,** 28–36 (1994).

27. Teytelman, L., Thurtle, D. M., Rine, J. & van Oudenaarden, A. Highly expressed loci are vulnerable to misleading ChIP localization of multiple unrelated proteins. *Proc. Natl. Acad. Sci. U. S. A.* **110,** 18602–7 (2013).

28. Levitsky, V. G. *et al.* Application of experimentally verified transcription factor binding sites models for computational analysis of ChIP-Seq data. *BMC Genomics* **15,** 80 (2014).

Nature Biotechnology: doi:10.1038/nbt.3300

29. Kulakovskiy, I. *et al.* From binding motifs in ChIP-Seq data to improved models of transcription factor binding sites. *J. Bioinform. Comput. Biol.* **11,** 1340004 (2013).

30. Mathelier, A. *et al.* JASPAR 2014: an extensively expanded and updated open-access database of transcription factor binding profiles. *Nucleic Acids Res* **42,** D142–D147 (2014).

31. Matys, V. *et al.* TRANSFAC and its module TRANSCompel: transcriptional gene regulation in eukaryotes. *Nucleic Acids Res.* **34,** D108–10 (2006).

32. Levitsky, V. G. *et al.* Effective transcription factor binding site prediction using a combination of optimization, a genetic algorithm and discriminant analysis to capture distant interactions. *BMC Bioinformatics* **8,** 481 (2007).

33. Jolma, A. *et al.* Multiplexed massively parallel SELEX for characterization of human transcription factor binding specificities. *Genome Res* **20,** 861–873 (2010).

34. Jolma, A. *et al.* DNA-Binding Specificities of Human Transcription Factors. *Cell* **152,** 327–339 (2013).

35. Zhang, C. *et al.* Integrative modeling defines the Nova splicing-regulatory network and its combinatorial controls. *Science* **329,** 439–43 (2010).

36. Cereda, M. *et al.* RNAmotifs: prediction of multivalent RNA motifs that control alternative splicing. *Genome Biol.* **15,** R20 (2014).

37. Wang, E. T. *et al.* Transcriptome-wide regulation of pre-mRNA splicing and mRNA localization by muscleblind proteins. *Cell* **150,** 710–24 (2012).

38. Kircher, M. *et al.* A general framework for estimating the relative pathogenicity of human genetic variants. *Nat Genet* **46,** 310–315 (2014).

39. Wang, K., Li, M. & Hakonarson, H. ANNOVAR: functional annotation of genetic variants from high-throughput sequencing data. *Nucleic Acids Res* **38,** e164–e164 (2010).

40. Siepel, A. *et al.* Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes. *Genome Res.* **15,** 1034–50 (2005).

41. Pollard, K. S., Hubisz, M. J., Rosenbloom, K. R. & Siepel, A. Detection of nonneutral substitution rates on mammalian phylogenies. *Genome Res.* **20,** 110–21 (2010).

42. Cook, K. B., Kazan, H., Zuberi, K., Morris, Q. & Hughes, T. R. RBPDB: a database of RNA-binding specificities. *Nucleic Acids Res.* **39,** D301–8 (2011).

43. Barski, A. *et al.* High-resolution profiling of histone methylations in the human genome. *Cell* **129,** 823–37 (2007).

44. Portales-Casamar, E. *et al.* PAZAR: a framework for collection and dissemination of cis-regulatory sequence annotation. *Genome Biol.* **8,** R207 (2007).

45. Funk, W. D., Pak, D. T., Karas, R. H., Wright, W. E. & Shay, J. W. A transcriptionally active DNA-binding site for human p53 protein complexes. *Mol. Cell. Biol.* **12,** 2866–71 (1992).

46. Berger, M. F. *et al.* Variation in homeodomain DNA binding revealed by high-resolution analysis of sequence preferences. *Cell* **133,** 1266–76 (2008).

47. Doody, G. M. *et al.* An extended set of PRDM1/BLIMP1 target genes links binding motif type to dynamic repression. *Nucleic Acids Res.* **38,** 5336–50 (2010).