Supplementary Information

Supplement

Benchmark Dataset

Pfam contains one profile HMM for each family, constructed from the family seed alignment. These manually curated seed alignments contain between 1 and 4545 sequences, originally picked due to their trusted functional annotations [9], avoiding some of the circularity of using sequence labels assigned by a profile HMM model. However, many Pfam seeds have recently been rebuilt to incorporate sequences from Pfam full that were found by the original family HMM, reducing their model-free status [40]. Pfam `seed` sequences vary between 4 and 2037 amino acids in length, with 27045 seed sequences of length > 500. Supplementary Fig. 1 contains a histogram of Pfam `seed` family sizes, the Pfam `seed` sequence length distribution and also the frequency of amino acid usage in the Pfam `seed` dataset.

Baseline Classifiers

phmmer

Perhaps the simplest existing sequence classification method that we implement is based on the `phmmer` function from HMMER 3.1b [11]. For phmmer we used the dev sequence set to run a small hyperparameter optimization study for phmmer speed on a 12-core Intel Xeon workstation with a 3.6 GHz processor. Each trial of the optimization had three hyperparameters: the number of sequences to feed to each command line invocation of phmmer, the number to pass to the `--cpu` argument of phmmer, and the number of concurrent processes (coroutines) to run. The settings 131, 9, 8 (respectively) produced the lowest running time, though performance was robust to different choices. Overall, settings with much larger block size and few CPUs, or with many fewer threads produced significantly longer run times.

Gathering Thresholds

Family-specific gathering thresholds, shown in Supplementary Fig. 1D, are used by HMMer 3.1b to determine whether a sequence belongs to each family [12]. The role of these gathering threshold is to increase coverage and decrease false positives. However our setup simply takes the top match by score, regardless of the assigned gathering threshold. This raises the question of whether implementing the family-specific gathering thresholds would have improved the accuracy score achieved by the HMM top pick algorithm.

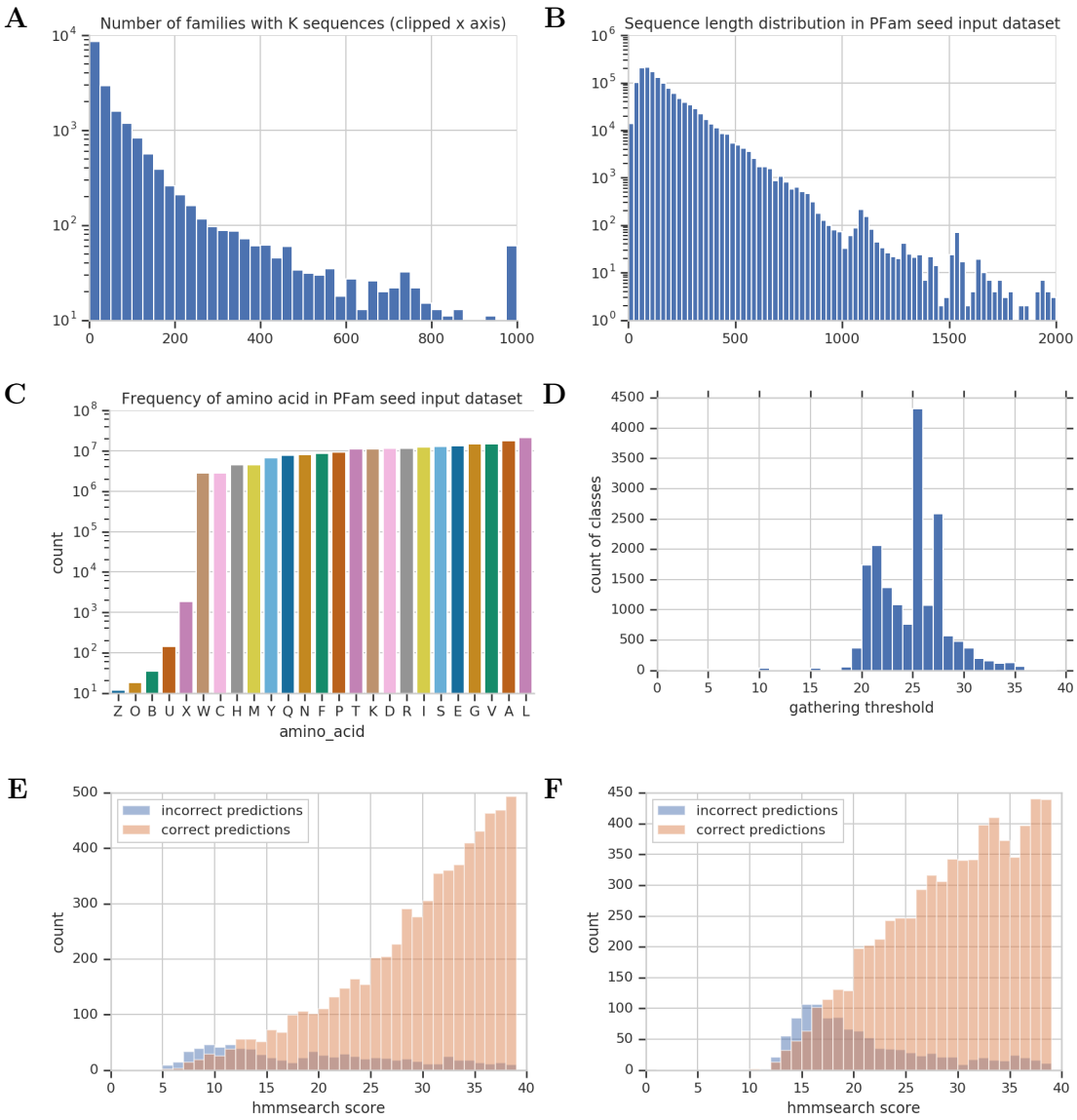

Figure 1: Benchmark Pfam `seed` dataset statistics calculated across the entire Pfam `seed` dataset. (A) Number of sequences per family. Values larger than 1000 are clipped to the last histogram bucket. (B) Sequence length distribution of unaligned sequences. (C) Frequency of amino acids in sequences in the dataset. (D) Histogram of gathering thresholds used in Pfam 32.0. Scores achieved by the top hits for (E) the HMM top pick model and (F) phmmer. The x axis has been truncated in both panels. Compare to (D) the actual histogram of gathering thresholds used by Pfam.

To address this, Supplementary Fig. 1E and F shows the distribution of top scores for both HMMer and phmmer, for both correct and incorrect predictions. We note that the majority of incorrect predictions have scores *below* the assigned gathering thresholds. However, there also appear to be at least as many correct predictions below these values. This qualitative analysis is backed up by experimentation wherein we determined 8.5% of top picks were below their family-specific gathering thresholds from Pfam. As such, using the assigned gathering thresholds would not have helped performance, and would required us to re-tune each of these values for the training dataset used in this benchmark study.

Hyperparameters for k-mer model

The hyperparameters that were tuned using the development set for the baseline kmer model are described in Supplementary Table 1. Though we expected the optimal kmer order to be larger, empirically, the optimal 'k' is 2.

	kmer
batch size	64
gradient clip	1
learning rate	.0005*
learning rate decay rate	0.997
learning rate decay step	1000
learning rate warmup steps	3000
kmer order	2*
number of hash buckets	10000*
train steps	300000

Table 1: Hyperparameters used in kmer benchmark models. An asterisk denotes a tuned value.

Multiple Family Membership

Pfam allows a domain to belong to multiple families if these families are in the same clan [12]. Our top pick formulation of HMMer for sequence classification does not allow for multiple membership. However, within the seed sequences, there are only two sequences that belong to more than one family. The first sequence has the two distinct names ABEC3_MOUSE/245-418 and E9QMH1_MOUSE/234-407, and the second has the two distinct names NLP_DROME/6-104 and B4HZJ8_DROSE/6-104. Both of these sequences are found in our training dataset, so the accuracy of their classification does not affect the model performance statistics that we report using the test dataset.

Details of Neural Network Architectures and Training

In a residual network (ResNet) [32], the layers are built up additively, with $f_i = f_{i-1} + g_i(f_{i-1})$. Here, each f_i is an $L \times F$ array and $g_i(\cdot)$ is an additional one-layer convolutional network (along with a kernel-size-one bottleneck convolution; see 5C) with trained weights specific to that layer. In our model, f_0 is obtained by a convolutional layer with F channels applied to the output of the input network, with no bottleneck convolution applied before the residual blocks. Each ResNet layer maintains a $L \times F$ representation; no downsampling is performed until the final pooling step. We also note that a convolutional layer is used before any residual block, so as to convert the per-residue representation into the correct shape before consumption by the residual blocks.

We primarily use convolutional neural networks (CNNs) to construct this $L \times F$ array, since they are fast to train and evaluate on modern hardware, an advantage that is even more pronounced when evaluating large sets of sequences in parallel. Convolutional architectures are also easily composed into higher-order interactions. The $L \times F$ array is then pooled along the length of the sequence, ensuring invariance to padding. Hyperparameters tuned for ProtCNN include the choice of F and max vs mean pooling in addition to network depth, which was varied between 1 and 6 layers.

Embedding Networks

Dilated convolutions are a popular method for enabling CNNs to capture long range interactions across the inputs [33]. One way to to model these long-range interactions would be to use convolutions with very wide kernels. However, doing so increases the computational complexity of prediction and introduces a considerable number of parameters to train. Instead, dilated convolutions use convolution kernels with holes in them, such that the complexity and number of parameters is the same as small, local convolutions, but the overall receptive field of the convolution is wide.

Consider a convolution with kernel width 5, and let $f_{i,j}$ be the representation in layer i of the CNN at position j in the sequence. In a traditional 1-dimensional convolution, f_i is a linear function of

$$\{f_{(j-2)}, f_{(i-1),(j-1)}, f_{(i-1),j}, f_{(i-1),(j+1)}, f_{(i-1),(j+2)}\}.$$

In a dilated convolution with dilation rate r, it is a function of

$$\{f_{(i-1),(j-2r)}, f_{(i-1),(j-r)}, f_{(i-1),j}, f_{(i-1),(j+r)}, f_{(i-1),(j+2r)}\}.$$

At each layer of our CNN, r is increased by a factor of k, so the overall receptive field size of the CNN is exponential in its depth. Specifically, if the model has n_1 non-dilated layers followed by n_2 dilated layers, k is the kernel width and r the dilation rate, then the receptive field size is $k + 2(k-1)(n_1 - 1) + 2(k-1)\sum_{i=1}^{n_2} r^i$. These terms correspond to the first layer, the remaining non-dilated layers, and the dilated layers respectively.

Supplementary Fig. 2 illustrates the relationship between receptive field size and classification accuracy of the resulting ProtCNN model for the Pfam `seed` dataset.

Model Invariance to Padding

At both train and test time, our model processes sequences in batches. The batches are of variable length, so input one-hot sequences are padded with zeros before being stacked together in a rectangle that can be processed in parallel on a GPU (see Fig. 5C. It is imperative that our model's predictions are insensitive to the padding, as the amount of padding depends on the other sequences in the batch (we pad to the longest sequence in the batch). For RNNs, this can be achieved by ending the recurrence of the RNN at the end of the un-padded sequence. For CNNs, our model maintains an $L \times F$ array of features at every layer, where each column corresponds to a specific location in the input sequence. Before each convolution or batch normalization operation, we zero-out the features in any location that corresponds to padding in the input sequence. This ensures that the model's predictions are insensitive to padding at test time. However, the dynamics of training our CNNs are still effected by padding, since batch normalization computes feature averages across the length of the sequence, and these lengths vary due to padding.

Neural Network Hyperparameters

Our embedding network architectures involve a variety of hyperparameters as outlined in Supplementary Table 3. For all networks, the "dev" fold is used to identify the optimal hyperparameter settings, while model performance statistics are reported using the completely distinct "test" fold. The CNN hyperparameters are tuned using values sampled at random from each hyperparameter search range, reported in Supplementary Table 2. The number of searched values is reported in Supplementary Table 5. We carried out an initial study that identified the most promising architecture. Supplementary Fig. 2B illustrates hyperparameter tuning.

We studied the impact of different hyperparameter settings on ProtCNN. As shown in Supplementary Table 2 we also allowed the batch size to vary, and introduced additional learning rate decay parameters in this study. Moreover, the number of filters was greatly increased, and the number of layers was allowed to vary as a hyperparameter. These modifications helped to maximize the performance of ProtCNN in terms of accuracy. However, they made the resulting model more difficult to interpret, in the sense that it became difficult to attribute increases in performance to specific parameters such as the size of the receptive field.

Supplementary Table 4 shows the ProtCNN hyperparameters used for the Pfam `full` dataset. For the RNN models, we tuned the hyperparameters jointly using Bayesian optimization with a Gaussian process regressor [41]; see Supplementary Table 5 for the number of searched values, Supplementary Table 3 for the values, and Supplementary

Model Type	Hyperparameter	Search Range
ProtCNN	batch size	32, 64, 128, 256
	dilation rate	1, 2, 3, 5
	filters	300 thru 3000, increments of 100
	ResNet block of first dilated layer	2, 3
	kernel size	3, 7, 9, 11, 21, 31
	ResNet layers	1 thru 6
	learning rate	1e-05, 5e-05, 1e-4, 5e-4, 1e-3
	learning rate decay steps	1e3,1e4,1e6, decay off
	pooling	max, mean
1, 2 ResNet block CNN	dilation rate	1, 2, 3, 5
	filters	150 thru 500, increments of 50
	ResNet block of first dilated layer	2, 3
	kernel size	3, 7, 9, 11, 21, 31
	learning rate	1e-4, 5e-4, 1e-3
	pooling	max, mean
RNN	learning rate	1e-4, 5e-4, 1e-3
	number of hidden units	25 thru 2048, increments of 1
	pooling	max, mean
kmer	embedding rank	100, 1000, 10000
	learning rate	1e-4, 5e-4, 1e-3
	ngram order	1 thru 5

Table 2: Search ranges for hyperparameter values, by model.

	ProtCNN	2 Block CNN	1 Block CNN	RNN
batch size	32	64	64	64
dilation rate	3*	2*		
filters	1100*	500*	500*	
first dilated layer	2*	2*		
gradient clip	1	1	1	1
kernel size	21*	31*	31*	
Number hidden units				1244*
learning rate	.0001*	.001*	.0001*	.0005*
learning rate decay rate	0.997	0.997	0.997	0.997
learning rate decay steps	1000*	1000	1000	1000
learning rate warmup steps	3000	3000	3000	3000
number of ResNet layers	5*	2	1	1
pooling	max*	max*	max*	mean*
ResNet bottleneck factor	0.5	0.5	0.5	
train steps	500000**	400000	400000	300000

Table 3: Hyperparameters used in neural networks for Pfam `seed` dataset. Column headers are model-type. An asterisk denotes a tuned value. Two asterisks denote that the model was overfit, and the number of tuning steps was chosen post-hoc so as to maximize dev-set performance.

	ProtCNN
batch size	64
filters	2000*
first dilated layer	NA
gradient clip	1
kernel size	21*
learning rate	.001*
learning rate decay rate	0.997
learning rate decay steps	1000*
learning rate warmup steps	3000
pooling	max*
ResNet bottleneck factor	0.5
train steps	1100000**

Table 4: Hyperparameters used in neural networks for Pfam `full` dataset. An asterisk denotes a tuned value. Two asterisks denote that the model didn't necessarily converge, but was ended after a reasonable time training (17 days).

Table 2 for the search ranges. RNNs using the final sequence-dimension LSTM cell output could only predict the mode of the distribution, however using the mean or max across the sequence dimension fixed this inability to propagate features through time. Increasing the number of layers did not improve performance, additional RNN hyperparameter settings are provided in Supplementary Table 3. It was hard to find RNN configurations with stable training dynamics, and RNNs took substantially longer to train. Furthermore, while CNN computations can be parallelized along the length of the sequence, RNN computations must be done sequentially, resulting in considerable slowdown for long sequences.

Model Type	Search Algorithm	Approx. number of samples
CNN (all depths)	random sampling	17000 †
RNN	Gaussian process	250
kmer	random sampling	50

Table 5: Search algorithms and number of samples for hyperparameter tuning, by model. † Many of these configurations were not feasible, as they did not fit in GPU memory.

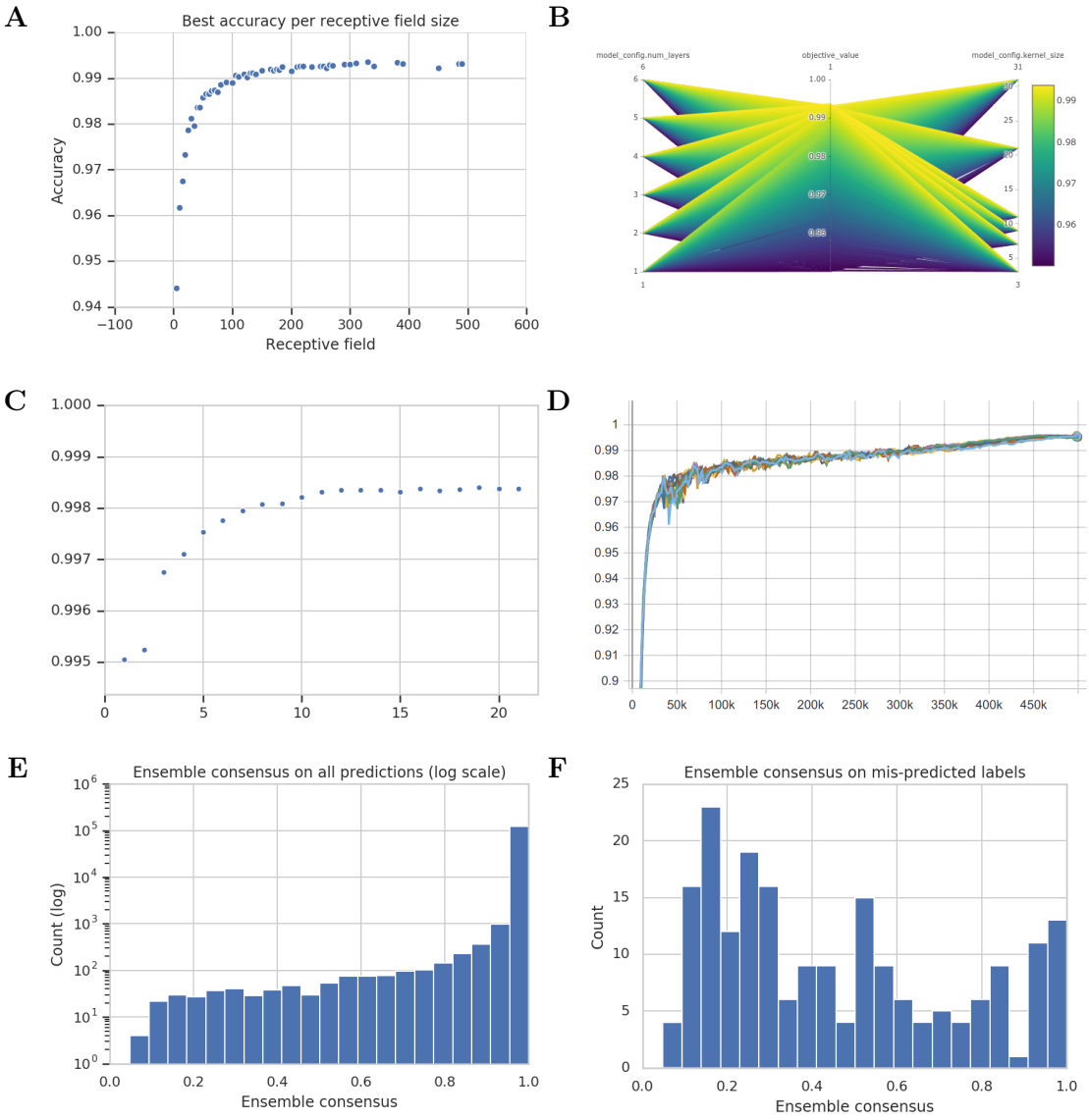

Figure 2: (A) Effect of receptive field on accuracy in a dilated residual network (ProtCNN). This experiment used a fixed batch size so as to decrease the number of "noise" parameters to maximize over. (B) A graph illustration of hyperparameter configurations of a ProtCNN model that achieve $> 95\%$ accuracy. (C) Predictive accuracy on the held out Pfam `seed` test data as a function of the number of ensemble elements. (D) Accuracy of training many replicates as ensemble elements on the Pfam `seed` training dataset, e.g. a value at 100K on the x-axis indicates the model's accuracy on the test set after seeing 100,000 training minibatches. E, F) Histogram of percentage agreement on predicted element among all ensemble elements for (E) incorrect predictions, and (F) all predictions.

ProtENN Analysis

Supplementary Fig. 2A shows the rapid increase in accuracy at sequence classification for the Pfam `seed` dataset as a function of the number of ProtENN elements, with saturation at about 13 models. The training of our neural networks was subject to a couple of sources of stochasticity: variable initializations, example ordering, and floating point computations on GPUs. As such, a natural question to ask is "How repeatable is the training of these neural networks?" The accuracy is very stable: even with different hyperparameters (Supplementary Fig. 2B), the networks' final accuracies were very close. Moreover, you can see multiple ensemble elements (replicates) of identical hyperparameter configurations in Supplementary Fig. 2C. However, as reported in the main text, the actual sequences that were misclassified were less stable, leading us to employ the use of an ensemble.

To describe the extent to which the predictions of each model in ProtENN agree, we calculate the ensemble consensus as the ratio of votes for a particular label divided by the number of ensembled elements. We report this statistic for both misclassified sequences, and over the entire test dataset in Supplementary Fig. 2E and F. These figures show that on the whole, the ensemble elements agree with each other. However, there is a nontrivial tail of disagreement, which sometimes resulted in an incorrect prediction.

In particular, we found that there were 11 sequences that were classified incorrectly in exactly the same way by every element of the ensemble used for ProtENN, listed in Supplementary Table 6). Our analysis of these sequences suggested that there may be some ambiguity over their correct family label in each case. For example, the sequence R7EGP4_9BACE/13-173 has a 100% identity match (found in UniProtKB) to a sequence that is classified as belonging to the YIP family. Moreover, this sequence has been independently annotated with a Gene Ontology term GO:0016021 (integral component of membrane), which matches that of the YIP family.

In a second example, the true family for Q7UQN3_RHOBA/655-688 and Q9VRV2_DROME/636-669, PF07719, has the following description: *This Pfam entry includes outlying Tetratricopeptide-like repeats (TPR) that are not matched by PF00515.* This indicates that this family was created because there were TPR domains that didn't match the HMM for PF00515. This finding is significant because ProtENN was able to correctly "smooth" over the noise in the training labels, and classify this protein as a member of PF00515. In a third example, the true family description for A0A1U8BSR7_MESAU/33-212 describes that this family is only for amphibian domains, but MESAU is a not an amphibian. Moreover, the predicted family is also a Novel AID APOBEC clade 1, but allows other species than mammals. This leads us to believe that the correct label is indeed the label chosen by ProtENN. annotating newly sequenced organisms to understand the functional capabilities.

sequence name	predicted label family name	true label family name
A0C5A2_PARTE/441-468	PF00036.32 EF_Hand_1	PF13202.6 EF_Hand_5
B2V696_SULSY/121-156	PF00132.24 HEXAPEP	PF14602.6 HEXAPEP2
R7EGP4_9BACE/13-173*	PF04893.17 YIP1	PF06930.12 DUF1282
H0W9E2_CAVPO/1073-1095 †	PF00560.33 LRR_1	PF05923.12 APC repeat
Q7UQN3_RHOBA/655-688 †	PF00515.28 TPR	PF07719.17 TPR_2
Q9VRV2_DROME/636-669	PF00515.28 TPR	PF07719.17 TPR_2
C5FK45_ARTOC/187-218	PF00023.30 ANK	PF13606.6 ANK_3
Q9T217_BPPHC/205-280	PF01751.22 Toprim	PF13662.6 Toprim_4
A0A1U8BSR7_MESAU/33-212 ‡	PF18778.1 NAD1	PF18782.1 NAD2
Q01NX2_SOLUE/7-54	PF13400.6 TAD	PF07811.12 TADe
F6ZV43_XENTR/909-973	PF00536.30 SAM_1	PF07647.17 SAM_2

Table 6: The 11 test sequences that all ensemble elements classify incorrectly, in exactly the same way. $*$ This sequence (annotated as belonging to a DUF) is likely a YIP. † These sequences were split into a different family due to the inability of a single HMM to match all elements with this function. ‡ The true label in Pfam may be incorrect, and instead be the predicted label.

Comparison of Model Accuracy

In the main text, we benchmark Pfam `seed` dataset sequence classification task. Here we provide the same analysis using classification accuracy instead of error rate. In Supplementary Fig. 3A we plot the accuracy of sequence classification for the Pfam `seed` heldout test dataset as a function of the pairwise sequence identity with the closest sequences in the training set as measured using BLASTp. The pairwise sequence identities for each of the 126171 heldout test sequences are binned into 10 equal sized deciles, and the average accuracy for each model in each bin is plotted.

In Supplementary Fig. 3B we report the accuracy as a function of the family size in the unsplit Pfam `seed` dataset. Again, the sequences are split into equal sized deciles based on family size, and the average accuracy in each decile is reported.

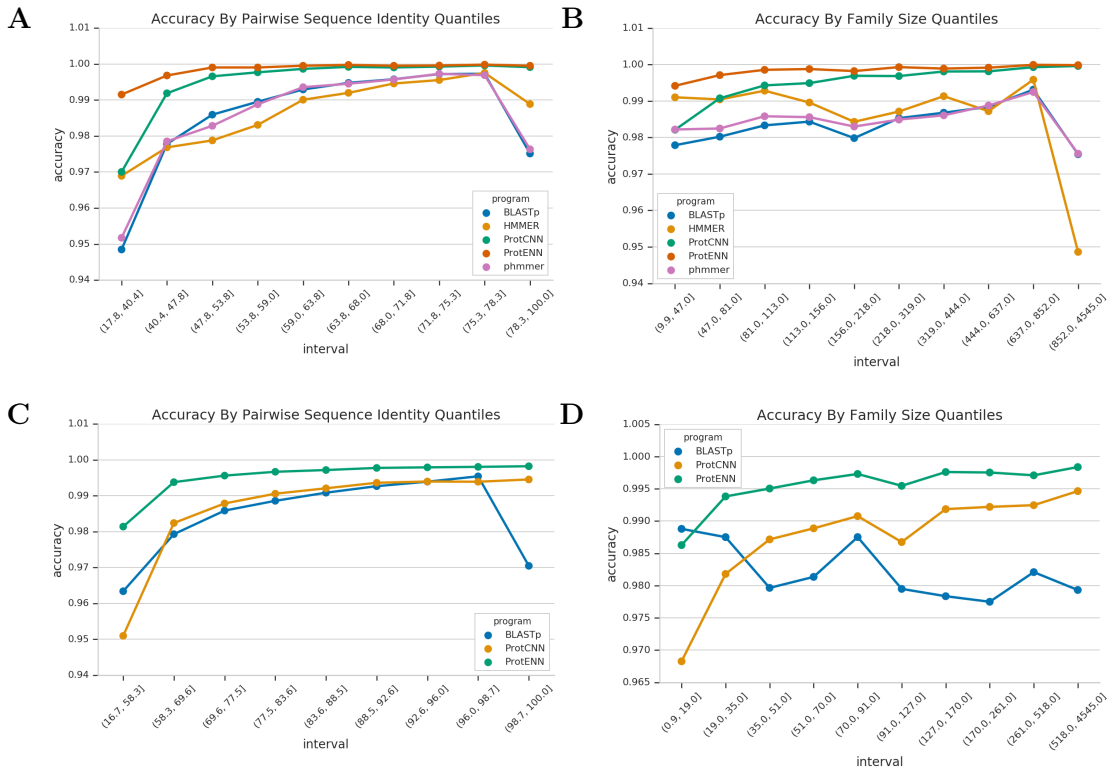

Figure 3: Performance accuracy for the Pfam `seed` dataset as a function of (A) the distance to the nearest training sequence, as measured using BLASTp, and (B) as a function of family size. Performance accuracy for the Pfam `full` dataset as a function of (C) the distance to the nearest training sequence, as measured using BLASTp, and (D) as a function of family size.

We note that the error rate of all models increases in the last decile of pairwise sequence

	phmmer	hmmsearch with filters	hmmsearch --max	BLASTp	ProtCNN
Seed train time		9 minutes	9 minutes	24 seconds	3 days
Always produces prediction?	no	no	yes	no	yes
Hardware	12-core Intel Xeon CPU 3.6 GHz	12-core Intel Xeon CPU 3.6 GHz	12-core Intel Xeon CPU 3.6 GHz	12-core Intel Xeon CPU 3.6 GHz	P100 GPU
Hyperparameters tuned for improved inference speed	Batch size: 131 Coroutines: 8 Cpus/Thread: 9	Coroutines: 12	Coroutines: 12	Threads: 12	

Table 7: Additional information about inference experimental setup.

identity for the held out test sequences from Pfam `seed`. The Pfam `seed` sequence sets are constructed such that different sequences within the same family are typically at least 80% different from one another [40]. The final decile spans 78.3%-100% pairwise identity, meaning that sequences in this decile will, in many cases, be closer in terms of sequence identity to a member of a different family than to their own. This finding further reflects the fact that pairwise sequence identity as computed by BLASTp is not a perfect classifier.

For the split of Pfam `full`, we also observe this reduction in accuracy for BLASTp in the last decile (see Supplementary Fig. 3C). Here, the situation is different and there is no threshold on sequence identity within each family. This is reflected by the fact that sequences within the last decile have between 98.7% and 100% reported identity (for the highest scoring pair found in the database as reported by BLASTp) with elements in the training set. There is another factor involved in the reduction of performance of BLASTp within this decile, which is that some sequences in the dataset are sub-sequences of others. Where the sub-sequence is in the test set, BLASTp measures "100%" sequence identity with the super-sequence contained in the training set. Discerning the correct classification in these cases can be quite difficult. For example, in Pfam `full`, one of the test sequences is A0A010NMM2_9MICC/241-409, and one of the training sequences is A0A010NMM2_9MICC/4-495. In this case, the former sequence has is identical to part of the latter, but it is classified differently by Pfam: the test sequence is the NAD binding domain of AdoHcyase, while the latter is the full AdoHcyase domain. This may explain some of the difficulty that BLASTp has with sequences that are very similar to those in the training set.

Computational Performance

Model Error Analysis

Supplementary Fig. 4 and Supplementary Table 8 provide further analysis of the type of errors made by the HMM top pick, ProtCNN and ProtENN on the Pfam `seed` sequence classification task. We note that the HMM top pick model tends to make errors for both small and large families. In particular, it makes errors for many more families with more than 100 members than either ProtCNN or ProtENN, as shown by comparing the figures in the left panels of Supplementary Fig. 4.

	Number wrong predictions	Imperfect families	Avg imperfect family size
HMMer	1784	392	1091
ProtCNN	625	550	127
ProtENN	201	164	141

Table 8: Errors made by HMMer and CNN models.

To understand this in more detail, in the right panels of Supplementary Fig. 4 we further break out the number of incorrect predictions made by each model type for families of different sizes. These figures show that the HMM top pick model tends to make multiple errors per family. In contrast, both ProtCNN and ProtENN tend to make just one error per family in the majority of cases. This is further illustrated by the statistics given in the main text, which reveal that the HMM top pick model makes an average of 4.55 errors per imperfect family, while in contrast ProtCNN makes an average of just 1.14 errors per imperfect family, while ProtENN makes 1.23 errors per imperfect family.

Concordance with known science

As reported in the main text, we challenged ProtENN trained on the Pfam `full` dataset to distinguish between single amino acid variants of protein domain sequences. Using the above method, we examined the purported helical propensity of each amino acid in the transmembrane regions (the amino acids whose substitutions least changed the predicted function of the domain), and found that the list, from most to least favored, is `VLIAM FTWCY SNGQH PRDKE`. The fact that the charged amino acids, along with proline, are the least favored is in accord with our understanding of 1) the unfavorability of polarity in the transmembrane region; 2) the sharp binding angle of proline. We also note the visual effect of substituting glycine is much like that of substituting proline: we know that glycine has low helical propensity [42].

For reference, the wild-type sequences that were used for Figures 4B and Supplementary Fig. 5 are available in the above table.

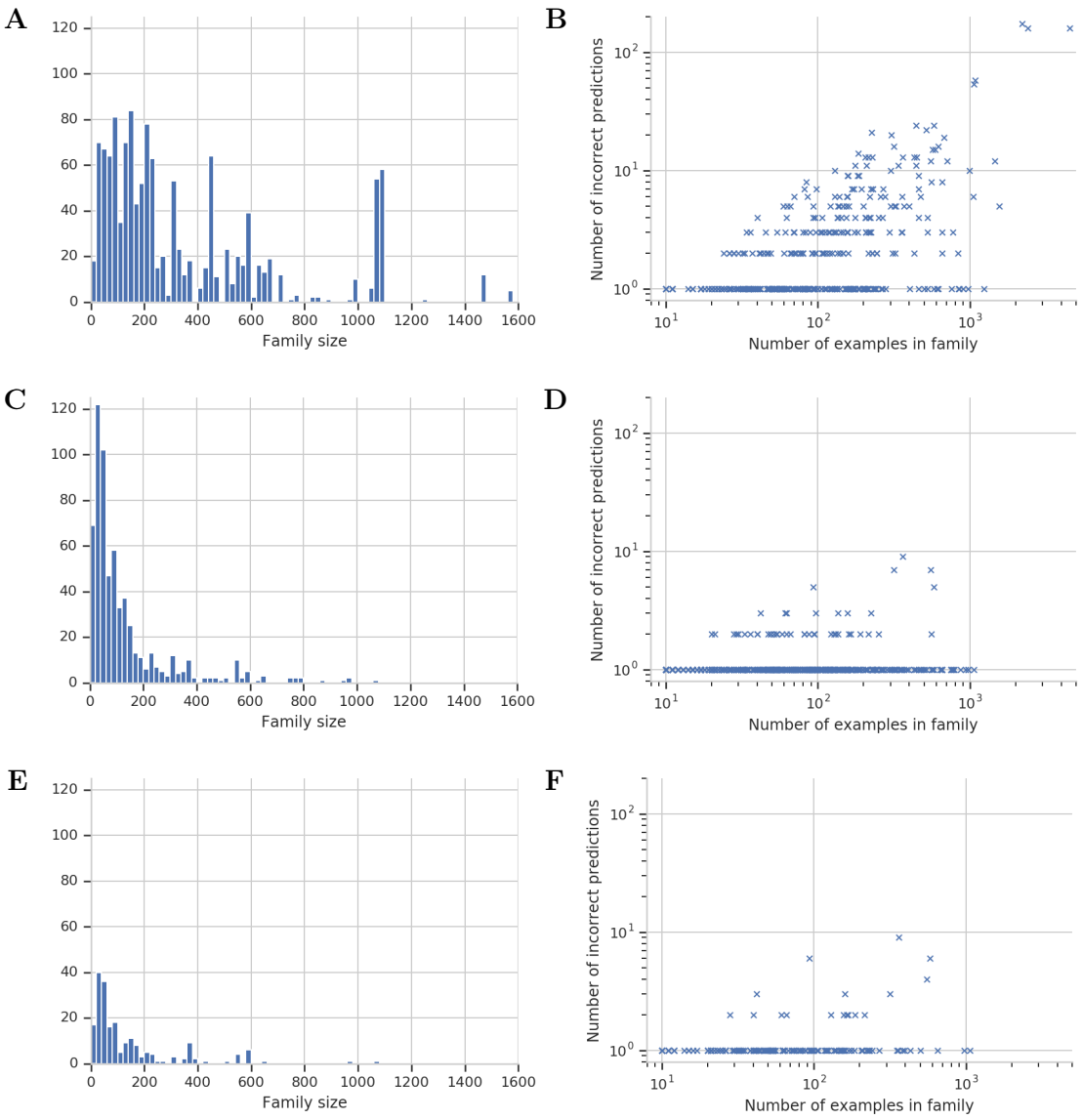

Figure 4: Comparison of errors made by the different models: A, B: HMM top pick; C, D: ProtCNN; E, F: ProtENN. We note that the HMM tends to make multiple errors for specific, and often quite large Pfam classes. In contrast, the ProtCNN model tends to make spurious errors for many small Pfam classes, and the specific errors made vary between different replicate models. This leads to the performance increase observed for ProtENN.

Sequence Name	Residues	Sequence
AT1A1_PIG	161-352	NMVPQQALVIRNGEKMSINAEEVVVG DLVEVKGGDRIPADLRIISANGCKVD NSSLTGESEPQTRSPDFTNENPLETR NIAFFSTNCVEGTARGIVVYTGDRTV MGRIATLASGLEGGQTPIAAEIEHFI HIITGVAVFLGVSFFILSLILEYTWL EAVIFLIGIIVANVPEGLLATVTVCL TLTAKRMARK
V2R_HUMAN	54-325	SNGLVLAALARRGRRGHWAPIHVFIG HLCLADLAVALFQVLPQLAWKATDRF RGPDALCRAVKYLQMVGMYASSYMIL AMTLDRHRAICRPMLAYRHGSGAHWN RPVLVAWAFSLLLSLPQLFIFAQRNV EGGSGVTDCWACFAEPWGRRTYVTWI ALMVFVAPTLGIAACQVLIFREIHAS LVPGPSERPGGRRRGRRTGSPGEGAH VSAAVAKTVRMTLVIVVVYVLCWAPF FLVQLWAAWDPEAPLEGAPFVLLMLL ASLNSCTNPWIY

Table 9: Wildtype sequences, keyed by Uniprot ID, that were used for saturation mutagenesis predictions.

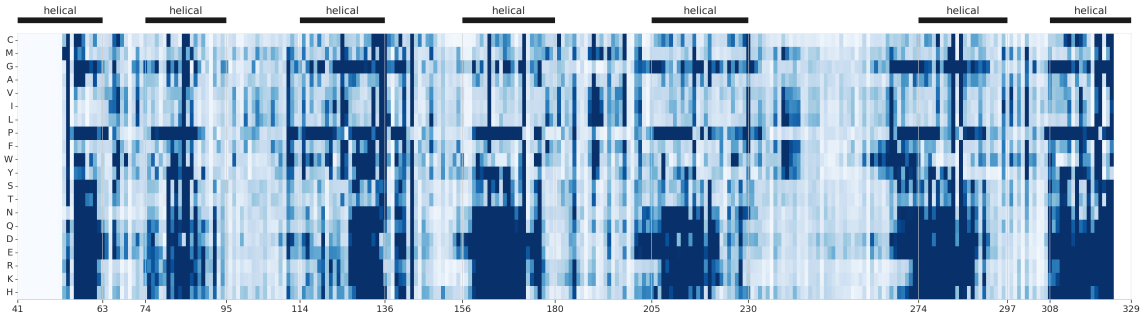

Figure 5: Predicted change in function for each missense mutation in vasopressin domain V2R_HUMAN/54-325 from family PF00001.21. The x-axis is residue indices in the protein P30518 (the domain starts at index 54), the y axis is the substitution of a particular amino acid, and a dark color saturation describes a large predicted change in function.
The model (trained on Pfam-full) appropriately predicts that substituting proline, glycine, or charged amino acids in the transmembrane helix regions is very likely to change the function of the protein substantially.

Few Shot Sequence Classification

As described in the main text, we construct F-dimensional embeddings using an *embedding network* consisting of all but the final layer of a pretrained CNN. Protein embeddings may, for example, provide annotation for various domains of unknown function or help avoid overfitting when performing supervised learning using small labeled datasets.

We evaluate our embeddings in terms of their ability to provide accurate nearest-neighbor classification. When confronted with a new sequence, we compute its embedding and find proteins with known labels that have nearby embeddings. This approach is fundamentally different than using a CNN for classification, where we are constrained to only predict families that were seen during model training. Along the lines of popular tools such as BLAST or phmmer, we can perform nearest neighbors classification with any set of proteins and any annotation scheme, independent of how the embedding network was trained. A key difference is that we avoid direct comparisons between sequences and instead compute similarity in terms of embeddings, which uses linear algebra routines that can be accelerated substantially using modern hardware. The shared model has been trained across Pfam families, so that general protein sequence statistics need not be rediscovered from a few examples of the novel family, but instead can be used as a prior from which the statistics of the novel family can be derived.

Table 4 evaluates the performance of nearest-neighbor classification in embedding space using the ProtCNN embedding network on the above randomly-split set of Pfam `seed` sequences. We first train ProtCNN using the procedure described in the main text, leaving out small families from the training set, and discard the final layer. Then, we compute the output of this embedding network for every sequence in the training set and compute a linear whitening transformation such that their covariance becomes the identity.

Then, we compute whitened embeddings for test-set sequences. We consider three classification methods. *Per-Instance 1-NN* performs 1-nearest-neighbor classification using cosine similarity between whitened embeddings. In *Per-Family 1-NN*, we first pre-compute an average whitened embedding for every family in the train set, and then form a prediction for a test sequence by finding the nearest neighbor between the embedding of a sequence and the set of average embeddings. This approach is desirable because its computational complexity scales with the number of families (the same as prediction in the original CNN) instead of with the number of training examples. *Per-Instance 1-NN* is analogous to classification with phmmer or BLASTp, whereas in terms of computational complexity *Per-Family 1-NN* is analogous to using HMMER.