6th ICML Workshop on Automated Machine Learning (2019)

AdaNet: A Scalable and Flexible Framework for Automatically Learning Ensembles

Charles Weill[1], **Javier Gonzalvo**[1], **Vitaly Kuznetsov**[1], **Scott Yang**[3], **Scott Yak**[1], **Hanna Mazzawi**[1], **Eugen Hotaj**[1], **Ghassen Jerfel**[1,2], **Vladimir Macko**[1], **Ben Adlam**[1], **Mehryar Mohri**[1,3], and **Corinna Cortes**[1]

[1] *Google AI*, New York, New York, United States;* [2] *Duke University, Durham, North Carolina, United States;* [3] *Courant Institute of Mathematical Sciences, New York, New York, United States*

**Corresponding authors:* {WEILL, XAVIGONZALVO}@GOOGLE.COM

Abstract

AdaNet is a lightweight TensorFlow-based (Abadi et al., 2015) framework for automatically learning high-quality ensembles with minimal expert intervention. Our framework is inspired by the *AdaNet* algorithm (Cortes et al., 2017) which learns the structure of a neural network as an ensemble of subnetworks. We designed it to: (1) integrate with the existing TensorFlow ecosystem, (2) offer sensible default search spaces to perform well on novel datasets, (3) present a flexible API to utilize expert information when available, and (4) efficiently accelerate training with distributed CPU, GPU, and TPU hardware. The code is open-source and available at: `https://github.com/tensorflow/adanet`.

1. Introduction

Recent years have seen the successful application of machine learning to a wide range of real-world problems. However each success requires making countless expert decisions in all aspects of data collection, feature engineering, and model search (Mendoza et al., 2018). The *AutoML* field has the objective to automate parts of this pipeline to produce high-quality models, free up expert time, and make machine learning more accessible. While the field has existed for many year, AutoML with large datasets and complex models is recently a viable option thanks to the expansion in computational power available through specialized hardware and cloud compute services.

Ensemble methods, one particular family of modeling techniques, consistently achieve state-of-the-art performance at challenges such as the Netflix Prize and Kaggle competitions (Zhou, 2012). They benefits from a rich history of theoretical guarantees (Freund and Schapire, 1997a), as well as new studies on their generalization capabilities (Cortes et al., 2014). However, building high-quality ensembles requires significant expertise, such as choosing the right base models, and knowing how to train them and combine their outputs. In this paper, we introduce AdaNet, our scalable and flexible TensorFlow framework for automatically learning ensembles.

2. Related Work

When designing AdaNet for our research and production needs within Google, we gave ourselves the following constraints. On one hand, we wanted a framework that could automatically produce a high-quality model given an arbitrary set of features and a model

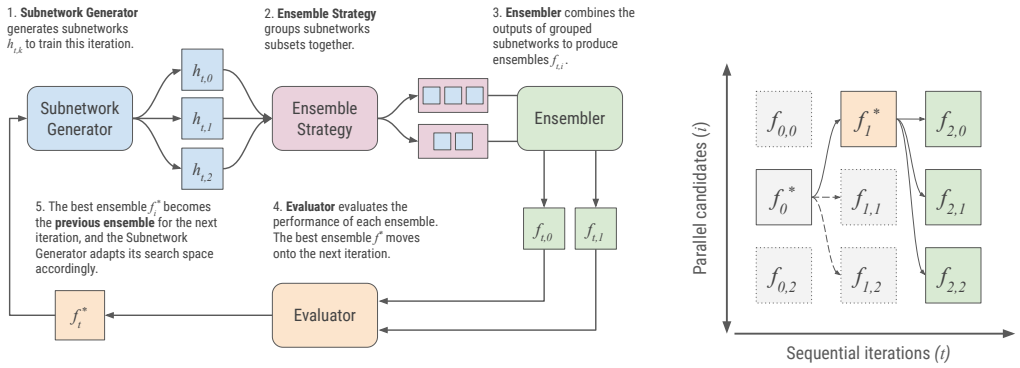


Figure 1: *Overview of the AdaNet adaptive search strategy.*

search space. On the other hand, we wanted it to build ensembles from productionized TensorFlow models to reduce churn, reuse domain knowledge, and conform with business and explainability requirements. Our framework should handle datasets containing thousands to billions of examples. Finally, we needed it to utilize available distributed compute and leverage accelerators when available, because training a single production model may take several days on hundreds of machines.

Several open-source AutoML frameworks, such as auto-sklearn (Feurer et al., 2015) and auto-pytorch (Mendoza et al., 2018), encode the expertise necessary to automate ensemble construction, and achieve impressive results. In comparison, our framework is built using TensorFlow to facilitate integration with TensorFlow-based production infrastructure and tooling. Furthermore, it is uniquely designed to efficiently execute on hundreds of heterogeneous workers in a distributed cluster and run on TPU. We designed AdaNet to meets our aforementioned needs, and have open sourced it to share with the entire AutoML community across companies and universities to accelerate open research.

We structured this paper as follows. Section 3 presents an overview of the system design. In Section 4 we outline framework implementation details and introduce our novel *adaptive computation graph* for handling TensorFlow limitations. Finally, in Section 5 we present some applications of AdaNet in production at Google.

3. Overview

An AdaNet run produces an *ensemble* model f composed of k *subnetworks* h_i (a.k.a *base learners* or *weak learners* in the literature (Zhou, 2012)) where $i \in [0, k-1], k \geq 1$. These ensembles are model-agnostic: subnetworks can be as simple as an if-statement, or as complex as convolutional or recurrent neural networks.

AdaNet combines two orthogonal ensembling paradigms (Zhou, 2012): parallel (similar to bagging) and sequential (similar to boosting). Together, these form the axes of the *adaptive search space* that the framework iteratively explores for an optimal ensemble (Figure 1). The search space is defined by the combination of *Subnetwork Generators* which generate candidate subnetworks h_t for iteration t, *Ensemble Strategies* which form discrete groups of subnetworks, and *Ensemblers* which combine the predictions of grouped subnetwork into

ensembles f_t. The framework is responsible for managing and training these ensembles and subnetworks. The *Evaluator* evaluates the candidate ensembles once they are finished training to select and fix f_t^*, the ensemble with the best performance according to the objective, along with its component subnetworks. The search then proceeds to iteration $t+1$, where the *Subnetwork Generator* adapts its search space according to the previous best ensemble. For example, if the subnetwork search space explores increasingly deeper neural networks, and the deepest subnetwork in the ensemble is l layers deep, the Subnetwork Generator could generate one candidate subnetwork with l hidden layers, and another with $l+1$.

4. Implementation

In this section we cover the implementation details of the AdaNet framework including APIs, its novel *adaptive computation graph*, and distributed training strategies.

4.1 Application Programming Interface

AdaNet extends `tf.estimator.Estimator` by Cheng et al. (2017) to encapsulate training, evaluation, prediction and export for serving. This abstraction enables the same user code to run on different hardware including CPUs, GPUs and TPUs. To specify the target task, such as regression, classification, or multi-task, the user passes a `tf.estimator.Head` instance when constructing the `adanet.Estimator`. Being an `Estimator` allows AdaNet models to be drop-in replacements for existing Estimator models, and integrate with tools in the TensorFlow open-source ecosystem (`https://github.com/tensorflow`) like TensorFlow Hub, Model Analysis, and Serving. A Keras (Chollet et al., 2015) API equivalent is under development.

```
import adanet
import tensorflow as tf
estimator = adanet.AutoEnsembleEstimator(
    head=tf.contrib.estimator.multi_class_head(n_classes=10),
    ensemblers=[adanet.ensemble.ComplexityRegularizedEnsemble()],
    ensemble_strategies=[adanet.ensemble.GrowStrategy()],
    candidate_pool={
        "linear": tf.estimator.LinearCLassifier(...),
        "dnn":    tf.estimator.DNNClassifier(...),
        "gbdt":   tf.estimator.BoostedTreesClassifier(...)})
estimator.train(...).evaluate(...).export_saved_model(...)
```

Users specify their search spaces using the `adanet.subnetwork` package to define how subnetworks adapt at each iteration, and the `adanet.ensemble` package to define how ensembles are composed, pruned, and combined. AdaNet provides `AutoEnsembleEstimator` to users who want a higher-level API for defining a search space in only a few lines of code using canned `Estimators` like `DNNEstimator` and `BoostedTreesClassifier`.

For visualizing model performance during training, the framework integrates with TensorBoard (Wongsuphasawat et al., 2017). When training is finished, the framework exports a TensorFlow SavedModel that can be deployed with TensorFlow Serving or similar services.

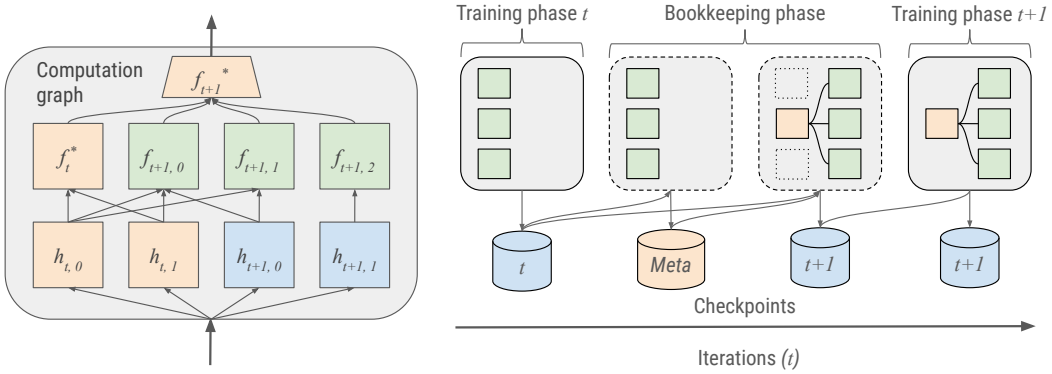


Figure 2: *The adaptive computation graph. Within an iteration, a static computation graph contains multiple subnetworks and ensembles, including the best ensemble and subnetworks from the previous iteration. Each new ensemble candidate is composed of a subset of the present subnetworks. The iteration tracks and exposes the predictions of the best ensemble at each training step. Across iterations, AdaNet implements an adapts its computation graph for the next iteration.*

4.2 Adaptive Computation Graph

A common mechanism for training an ensemble is to first train each base learner in a separate process, and then ensemble them in a second phase (Caruana et al., 2004). However, accelerators such as TPUs do not allow multiple processes to share the same hardware, which limits the number of candidates that can be trained in parallel. Instead, AdaNet creates all candidate within the same computation graph and session, including new subnetworks and ensembles for iteration t, and the best ensemble and corresponding subnetworks from $t-1$. (see Figure 2). This design allows candidates to share tensors. For example, subnetworks can share the same input pipeline and ensembles can share subnetwork outputs. It also makes many complex ensembling routines possible such as knowledge distillation (Hinton et al., 2015) and Born Again Networks (Furlanello et al., 2018), and can straightforwardly be extended to more complicated ones like population based training (Jaderberg et al., 2017). Furthermore, having all the candidates within the same graph allows compilers such as the TensorFlow compiler and XLA to optimally place ops on logical devices, which is particularly important for maximizing multi-core accelerator utilization such as when training many small subnetworks on a TPU.

Across iterations, the computation graph must evolve. However, TensorFlow and XLA were designed for a static computation graph, which limits our ability to create new operations during training. There are a number of workarounds: one is to dynamically store a model in a resource variable. This doesn't work for us, since it would require users to rewrite their models in low-level operations in C++ instead of Python, which would drastically increase the cost of adopting AdaNet and would limit flexibility. Another is to implement the framework using Eager execution, which unfortunately supports neither distributed nor TPU training.

The solution is to modify the train loop to create an *adaptive computation graph* as described in Figure 2: after completing training of all candidates in iteration t, AdaNet begins a *bookkeeping* phase. During this phase it reconstructs the graph with the evaluation

dataset and evaluates all the candidates to determine the best ensemble f_t^* for iteration t. Next it serializes metadata about the architecture of f_t^*, and uses this metadata to construct a new graph for $t+1$. The new graph includes f_t^* and all the new subnetworks h_{t+1} and ensembles f_{t+1}, and warm-starts the variables of f_t^* from the most recent checkpoint. Finally, it creates a new checkpoint with the modified graph for $t+1$ and increments the *iteration number* variable in the new checkpoint. When Estimator resumes training, it will construct the new graph based on the architecture metadata from iteration t, and will restore variables from the new checkpoint, and treat it as a static graph. Evaluate and predict have no effect on the iteration number so their methods require no modification.

4.3 Distributed Training

When training on small datasets or when debugging, AdaNet can be executed in single process. To speed up training and evaluate in parallel, we use `Estimator` to distribute work across worker machines and parameter servers. There are currently two distribute strategies: *replication* and the AdaNet-specific *round-robin*. `Estimator` provides the default replication strategy, where workers replicate a copy of the full computation graph containing all candidates, and share variable updates through the parameter servers. In the second strategy, candidate subnetworks are placed on dedicated workers in a round-robin fashion. This is only possible because subnetworks can be trained independently from one another. Certain designated workers load every read-only subnetworks, and train only the ensemble parameters. The round-robin strategy reduces the load on the workers and parameter servers, speeds up training when subnetworks are large, and allows the system to scale linearly with the number of subnetworks and workers.

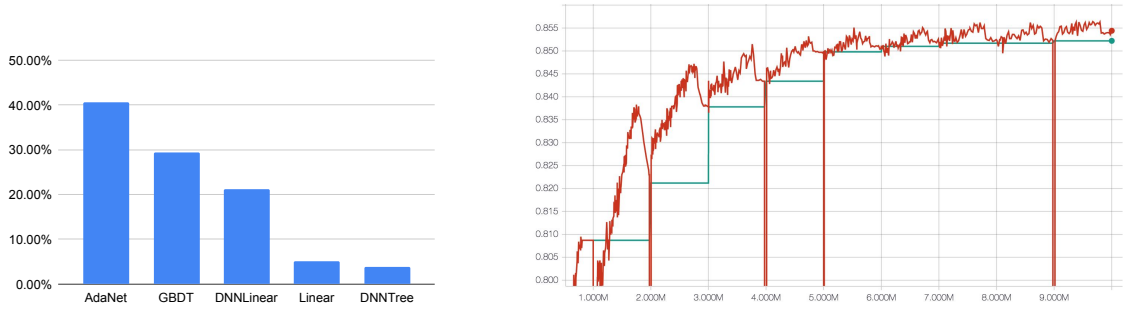
The system's distributed training and search execution is fault tolerant. If a worker or a cluster terminates due to preemption or an exception, it restores itself from a checkpoint to continue training with minimal loss in training time.

In order to support the adaptive computation graph during distributed training, one worker is designated as chief, and is responsible for bookkeeping. Other workers idly loop until the chief writes the expanded checkpoint with an incremented iteration number.

5. Applications and Results

We battle-tested AdaNet through several engagements and launches with Google products. Each of the following applications applied our predefined `ComplexityRegularizedEnsembler` to learn scalar mixture weights that balances the trade-off between ensemble train loss and *complexity* of the underlying subnetworks in order to obtain the learning guarantees from the algorithm from Cortes et al. (2017). See Appendix A for an overview of the theory behind the algorithm.

In one case, we integrated AdaNet into an internal platform designed for structured data using a search space composed of linear models, fully-connected neural networks that add a hidden layer at each iteration, and gradient-boosted decision trees. We benchmarked our search space versus other popular algorithms including wide-and-deep models, gradient-boosted trees, on over a hundred tabular datasets with thousands of datapoints. We also used bayesian optimization to tune each algorithm's hyperparameter set and trained each



(a) *Percentage of times an algorithm achieved the best performance on a set of production tabular datasets which include binary classification and regression tasks.*

(b) *Test accuracy per train step on CIFAR-100 using NASNet-A subnetworks. A new subnetwork begins training every million steps, and eventually improves the performance of the ensemble.*

Figure 3: *Examples of successes from applying AdaNet to different tasks.*

model for two hours using 10 CPU workers. Adanet produced the best model 40.56% of the time, followed by gradient boosted trees 29.44% (Figure 3).

In another case, the production model was already an ensemble of 20 large and complex neural networks, built with custom ensembling infrastructure and trained on 200 CPU workers and 40 parameter servers. Using `AutoEnsembleEstimator` with the round-robin distributed strategy, we replaced the custom infrastructure with a simpler and more robust system, and enabled them to iterate more quickly by trying different search spaces.

Finally, in our own research, we applied AdaNet with convolutional subnetworks using the NASNet-A structure from Zoph et al. (2017) to CIFAR-10 and CIFAR-100 to achieve 2.26% and 14.58% error rates respectively. See Figure 3 for a TensorBoard from a run training with 10 distributed V100 GPUs over 10 iterations of 1M steps each.

6. Conclusion and Future Work

AdaNet is a flexible and scalable framework for training, evaluating, and deploying ensembles of TensorFlow models (e.g. deep neural networks, trees, and linear models). It provides AutoML capabilities including automatic search over a space of candidate ensembles, supports CPU, GPU, and TPU hardware, and can scale from a single process to a cluster seamlessly using `tf.estimator.Estimator` infrastructure. The framework is flexible and can be extended to include a prior (i.e. fine-tuned production models) in its search space. It offers several out-of-the-box means of training and ensembling them (e.g. uniform average weighting, learning mixture weights). We open-sourced AdaNet to enable the AutoML community to leverage the large-scale computational offerings of industrial cloud providers just as we do within Google to accelerate research.

Future infrastructure work involves improving the ability to scale the number of candidates per iteration, and better handling the case when there are more candidates than machines. Future research involves providing better search spaces out-of-the-box for a wide range of common machine learning tasks, and developing new search algorithms for producing better ensembles more quickly, consistently, and in a principled manner.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 18–, New York, NY, USA, 2004. ACM. ISBN 1-58113-838-5. doi: 10.1145/1015330.1015432. URL `http://doi.acm.org/10.1145/1015330.1015432`.

Heng-Tze Cheng, Zakaria Haque, Lichan Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D. Sculley, Jamie Smith, David Soergel, Yuan Tang, Philipp Tucker, Martin Wicke, Cassandra Xia, and Jianwei Xie. Tensorflow estimators: Managing simplicity vs. flexibility in high-level machine learning frameworks. *CoRR*, abs/1708.02637, 2017. URL `http://arxiv.org/abs/1708.02637`.

François Chollet et al. Keras. `https://keras.io`, 2015.

Corinna Cortes, Mehryar Mohri, and Umar Syed. Deep boosting. In *Proceedings of the Thirty-First International Conference on Machine Learning (ICML 2014)*, 2014.

Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. AdaNet: Adaptive structural learning of artificial neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 874–883, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL `http://proceedings.mlr.press/v70/cortes17a.html`.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf`.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997a. doi: 10.1006/jcss.1997.1504. URL `https://doi.org/10.1006/jcss.1997.1504`.

Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997b. ISSN 0022-0000. doi: 10.1006/jcss.1997.1504. URL http://dx.doi.org/10.1006/jcss.1997.1504.

Tommaso Furlanello, Zachary C Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born again neural networks. *arXiv preprint arXiv:1805.04770*, 2018.

Stuart Geman, Elie Bienenstock, and Ren Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992. doi: 10.1162/neco.1992.4.1.1. URL https://doi.org/10.1162/neco.1992.4.1.1.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017. URL http://arxiv.org/abs/1711.09846.

V. Koltchinskii and D. Panchenko. Empirical margin distributions and bounding the generalization error of combined classifiers. *Ann. Statist.*, 30(1):1–50, 02 2002. doi: 10.1214/aos/1015362183. URL https://doi.org/10.1214/aos/1015362183.

Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, Matthias Urban, Michael Burkart, Max Dippel, Marius Lindauer, and Frank Hutter. Towards automatically-tuned deep neural networks. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *AutoML: Methods, Sytems, Challenges*, chapter 7, pages 141–156. Springer, December 2018. To appear.

Roman Novak, Yasaman Bahri, Daniel A Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study. *arXiv preprint arXiv:1802.08760*, 2018.

Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda Viegas, and Martin Wattenverg. Visualizing dataflow graphs of deep learning models in tensorflow. 2017. URL http://idl.cs.washington.edu/files/2018-TensorFlowGraph-VAST.pdf.

Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC, 2012.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017.

Appendix A. Complexity Regularized Ensembler Theory

Let x be some input feature vector in the input space $x \in \mathcal{X}$, the ensemble consists of l base-learner functions $\{h_k : h_k \in \mathcal{H}_k, k \in [l]\}$, where $\mathcal{H}_k$ is a class of functions so that $h_k : x \to \mathbb{R}^{n_k}$ where n_k is the dimension of the output. The function f, which represents the ensemble that we are trying to learn, is a convex sum of the base-learner functions:

$$f = \sum_{k=1}^{l} w_k h_k. \tag{1}$$

Building an automatic ensemble of neural networks has two main challenges: (1) choosing the best subnetwork architectures; and (2) using the right number of subnetworks. The AdaNet framework supports different ways to handle the learning of the weights w_k and tackle these two challenges. A very robust baseline is obtained by having a uniform ensemble where the weights are $w_k = 1/l$. More advances techniques are related to complexity and generalization theory. While having numerous parameters is necessary for a neural network to obtain high expressivity power, they may not generalize to unseen data due to their greater complexity which could make the network memorize noise. As proposed in (Cortes et al., 2017), the solution for the first problem is to define an upper bound on the Rademacher complexity of a DNN (see Section A.2). The second challenge is related to the contribution of a DNN to the general ensemble (see Section A.3).

A.1 Definitions and notation

To simplify the presentation, we restrict our attention to the case of binary classification. All our results can be straightforwardly extended to multi-class classification, by augmenting the number of output units and by using existing multi-class counterparts of the margin bounds.

Let $\mathcal{X}$ be the input space, and $\{-1, +1\}$ be the output space for the function we are trying to learn. The training set S consists of m labeled examples, which are drawn from the true distribution $\mathcal{D}$ over $\mathcal{X} \times \{-1, +1\}$. The ensemble consists of l base-learner functions $\{h_k : h_k \in \mathcal{H}_k, k \in [l]\}$, where $\mathcal{H}_k$ is some class of functions that maps $\mathcal{X}$ to $\{-1, +1\}$. The function f, which represents the ensemble that we are trying to learn, is a convex sum of the base-learner functions:

$$f = \sum_{k=1}^{l} w_k h_k, \qquad \text{s.t.} \quad \sum_{k=1}^{l} |w_k| \leq 1. \tag{2}$$

A.2 Bounding the generalization error of ensembles

Our learning bounds are expressed in terms of the Rademacher complexities of the hypothesis sets $\mathcal{H}_k$. The generalization error bound of an ensemble f is given by the Deep Boost theorem(Cortes et al., 2014) as:

$$R(f) \leq \hat{R}_{S,\rho}(f) + \frac{4}{\rho} \sum_{k=1}^{l} |w_k| \mathfrak{R}_m(\mathcal{H}_k) + \widetilde{O}\left(\frac{1}{\rho}\sqrt{\frac{\log l}{m}}\right) \tag{3}$$

where $\rho \in (0, 1]$, $R(f)$ is the generalization error $\mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathbb{1}_{yf(x)\leq 0}]$, $\hat{R}_{S,\rho}(f)$ is the empirical margin error $\mathbb{E}_{(x,y)\in S}[\mathbb{1}_{yf(x)\leq\rho}]$, and $\mathfrak{R}_m(\mathcal{H}_k)$ is the Rademacher complexity of the function class $\mathcal{H}_k$ with m examples.

Learning guarantees in Equation 3 are finer than those that can be derived via a standard Rademacher complexity analysis (Koltchinskii and Panchenko, 2002). This is because it admits an explicit dependency on the mixture weights w_k that allows a balanced contribution of subnetworks with very distinct complexities.

A.3 Balancing empirical risk and generalization

AdaNet approach (Cortes et al., 2017) seeks to find a function f by optimizing the set of weights $\mathbf{w}$ in Equation 2 using the constraints of Equation 3. The idea is to balance the trade-off between the ensembles performance on the training set and its ability to generalize to unseen data:

$$F(\mathbf{w}) = \underbrace{\frac{1}{m}\sum_{i=1}^{m}\Phi\left(1 - y_i\sum_{j=1}^{N} w_j h_j\right)}_{\text{Empirical error}} + \underbrace{\sum_{j=1}^{N}\left(\lambda \cdot r(h_j) + \beta\right)|w_j|}_{\text{Complexity penalty}} \tag{4}$$

where h_j is the j-th subnetwork, w_j is the weight of the j-th subnetwork, Φ is the surrogate loss function, $r(h_j)$ is model j's complexity, and λ and β are tunable hyperparameters. Φ is a non-increasing convex function (e.g., exponential function as in AdaBoost (Freund and Schapire, 1997b)). Its purpose is to facilitate a convex surrogate of the empirical error.

By optimizing the weights of the ensemble $\mathbf{w}$ in Equation 4 the model will include a candidate subnetwork only if it improves the ensembles training loss more than it affects its ability to generalize. This guarantees that the generalization error of the ensemble is bounded by its training error and complexity while simultaneously minimizing this bound.

The main benefit of optimizing the objective in Equation 4 is to eliminate the need for a hold-out set for choosing which candidate subnetworks to add to the ensemble. This has the added benefit of enabling the use of more training data for training the subnetworks.

A.4 Complexity and sensitivity measures

The idea of complexity is to regularize the learning of mixture weights of each subnetwork based on their sensitivity to training data as a proxy for the potential generalization error on test data.

In addition to the Rademacher complexity used in Section A.2 and the upper bounds presented in (Cortes et al., 2017) we have introduced two new sensitivity measures that work as proxy functions of the Rademacher complexity: (1) the norm of the variance at the final layer across data points for each batch; and (2) the matrix norm of the Jacobian of the logits layer with respect to the input layer.

The motivation for introducing additional complexity measures comes from our experimental observations on the behaviour of DNNs over many datasets. While there is a general relationship between Rademacher complexity and the size of a subnetwork, it can be loose because the Rademacher complexity also depends on the shape of the architecture. This was supported by our preliminary experimental results in (Cortes et al., 2017) where it was

shown that the variance can perform almost as well in regularizing the mixture weights. The idea has to do with that of variance-bias trade-off (Geman et al., 1992). With similar training losses one learner with a higher variance could be a sign of fitting to random noise in each training batch. Novak et al. (2018) explored the norm of the input-output Jacobian of the network and found it to correlate well with generalization on simplified fully-connected networks. The hypothesis is that both measures can fare as least as good as our Rademacher-based bound on convolutional networks as well as deeper fully connected networks.

Variance of subnetwork h_k is calculated as,

$$\sigma^2(h_k) = \frac{1}{m}\sum_{j=1}^{m}(h_k(x_j) - \overline{h}_k)$$

$$\overline{h}_k = \frac{1}{m}\sum_{j=1}^{m} h_k(x_j),$$

where $h_k(x_j)$ is the output of subnetwork k for the j-th input example.

Similar patterns were seen for the Jacobian but at a lower correlation. As inspired by (Novak et al., 2018) we introduce the Frobenius norm of the Jacobian matrix at the softmax layer. Accordingly, this is obtained by computing the matrix of point-wise derivatives of the number of outputs activations and also computes the Euclidean matrix norm before averaging over the training points.