ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections

Sujith Ravi
Google Research, Mountain View, CA, USA
`sravi@google.com`

Abstract

Deep neural networks have become ubiquitous for applications related to visual recognition and language understanding tasks. However, it is often prohibitive to use typical neural network models on devices like mobile phones or smart watches since the model sizes are huge and cannot fit in the limited memory available on such devices. While these devices could make use of machine learning models running on high-performance data centers with CPUs or GPUs, this is not feasible for many applications because data can be privacy sensitive and inference needs to be performed directly "on" device.

We introduce a new architecture for training compact neural networks using a joint optimization framework. At its core lies a novel objective that jointly trains using two different types of networks–a full trainer neural network (using existing architectures like Feed-forward NNs or LSTM RNNs) combined with a simpler "*projection*" network that leverages random projections to transform inputs or intermediate representations into bits. The simpler network encodes lightweight and efficient-to-compute operations in bit space with a low memory footprint. The two networks are trained jointly using backpropagation, where the projection network learns from the full network similar to apprenticeship learning. Once trained, the smaller network can be used directly for inference at low memory and computation cost. We demonstrate the effectiveness of the new approach at significantly shrinking the memory requirements of different types of neural networks while preserving good accuracy for several visual recognition and text classification tasks. We also study the question "how many neural bits are required to solve a given task?" using the new framework and show empirical results contrasting model predictive capacity (in bits) versus accuracy on several datasets. Finally, we show how the approach can be extended to other learning settings and derive projection models optimized using graph structured loss functions.

1   Introduction

Recent advances in deep neural networks have resulted in powerful models that demonstrate high predictive capabilities on a wide variety of tasks from image classification [1] to speech recognition [2] to sequence-to-sequence learning [3] for natural language applications like language translation [4], semantic conversational understanding [5] and other tasks. These networks are typically large, comprising multiple layers involving many parameters, and trained on large amounts of data to learn useful representations that can be used to predict outputs at inference time. For efficiency reasons, training these networks is often performed with high-performance distributed computing involving several CPU cores or graphics processing units (GPUs).

In a similar vein, applications running on devices such as mobile phones, smart watches and other IoT devices are on the rise. Increasingly, machine learning models are used to perform real-time inference directly on these devices—e.g., speech recognition on mobile phones [6], medical devices

to provide real-time diagnoses [7] and Smart Reply on watches [8], among others. However, unlike high-performance clusters running on the cloud, these devices operate at low-power consumption modes and have significant memory limitations. As a result, running state-of-the-art deep learning models for inference on these devices can be very challenging and often prohibitive due to the high computation cost and large model size requirements that exceed device memory capacity. Delegating the computation-intensive operations from device to the cloud is not a feasible strategy in many real-world scenarios due to connectivity issues (data cannot be sent to the server) or privacy reasons (certain types of data and processing needs to be restricted to a user's personal device). In such scenarios, one solution is to take an existing trained neural network model and then apply compression techniques like quantization (e.g., reducing floating point precision [9]) to reduce model size. However, while these techniques are useful in some cases, applying them post-learning to a complex neural network tends to dilute the network's predictive quality and does not yield sufficiently high performance. An alternate strategy is to train small models for on-device prediction tasks, but these can lead to significant drop in accuracies [10] which limits the usability of such models for practical applications. In particular, feature or vocabulary pruning techniques commonly applied to limit parameters in models like recurrent neural networks, while yielding lower memory footprint, can affect the predicitive capacity of the network for language applications.

This motivates the learning of efficient on-device machine learning models with low memory footprint that can be run directly on device for inference at low computation cost.

Contributions. In this paper, we propose a joint learning framework based on neural projections to learn lightweight neural network models for performing efficient inference on device.

- The neural projection framework can leverage any existing deep network like feed-forward or recursive neural network to teach a lightweight projected model in a joint optimization setup which is trained end-to-end using backpropagation. We use projections based on locality sensitive hashing to represent the hidden units for the lightweight network which encodes operations that are extremely efficient to compute during inference (Section 3).

- The framework permits efficient distributed training but is optimized to produce a neural network model with low memory footprint that can run on devices at low computation cost. The model size is parameterized and configurable based on the task or device capacity.

- We demonstrate the effectiveness of the new approach in achieving significant reduction in model sizes while providing competitive performance on multiple visual and language classification tasks (Section 4).

- The proposed framework is further used to study and characterize the predictive capacity of existing deep networks in terms of the number of *neural projection bits* required to represent them compactly (Section 5).

2 Related Work

There have been a number of related works in the literature that attempt to learn efficient models under limited size or memory constraints. Some of these works employ techniques ranging from simple dictionary lookups to feature pruning [11] or hashing [12, 13, 14] to neural network compression. In the past, researchers have proposed different methods to achieve compact representations for neural networks using reduced numerical precision [9], vector quantization [15], binarization strategies for networks [16] or weight sharing [17, 18]. Most of these methods aim to exploit redundancy in the network weights by grouping connections using low-rank decomposition or hashing tricks. In contrast, our work proposes to learn a simple projection-based network that efficiently encodes intermediate network representations (i.e., hidden units) and operations involved, rather than the weights. We also introduce a new training paradigm for on-device models where the simple network is coupled and jointly trained to mimic an existing deep network that is flexible and can be customized by architecture or task. As we show in Section 3.1, the specifics of the training process can also include a choice to optimize towards soft targets as in model distillation approaches [19].

Dropouts [20] and other similar variants commonly used in practice for deep learning attempt to reduce parameters during neural network training by dropping unimportant neurons. However, they serve a different purpose, namely for better regularization.

[21] offers a survey of binary hashing literature that is relevant to the projection functions used in our work. The coupled network training architecture proposed in this paper (described in Section 3.1) also resembles, conceptually at a high level, generative adversarial networks (GANs) [22] which are used in unsupervised learning settings to reconstruct or synthesize data such photorealistic images.

3 Neural Projection Networks

In this section, we present Neural Projection Networks, a joint optimization framework for training neural networks with reduced model sizes. We first introduce the objective function using a coupled full+projection network architecture and then describe the projection mechanism used in our work, namely locality sensitive hashing (LSH) and how it is applied here.

3.1 ProjectionNets

Neural networks are a class of non-linear models that learn a mapping from inputs $\vec{x}_i$ to outputs y_i, where $\vec{x}_i$ represents an input feature vector or sequence (in the case of recursive neural networks) and y_i is an output category for classification tasks or a predicted sequence. Typically, these networks consist of multiple layers of hidden units or neurons with connections between a pair of layers. For example, in a fully-connected feed-forward neural network, the number of weighted connections or network parameters that are trained is $O(n^2)$, where n is the number of hidden units per layer.
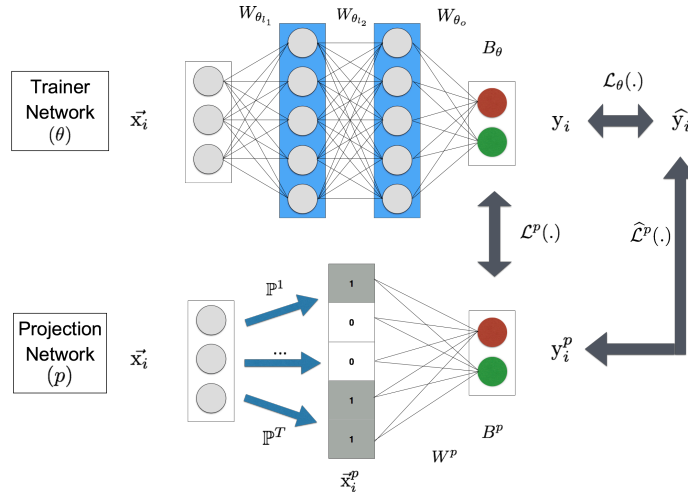

Figure 1: Illustration of a Neural Projection Network trained using feed-forward NN. *Notation:* $\vec{x}_i$ represents the input feature vector, $\widehat{y}_i$ the ground-truth, y_i the prediction from the full network and y_i^p the prediction from projection network. $\mathbb{P}^1...\mathbb{P}^T$ denote the T projection functions that transform the input $\vec{x}_i$ into d-bit vectors, one per function. W_θ, B_θ and W^p, B^p represent the weights/bias parameters for the trainer network and projection network, respectively. The training objective optimizes a combination of NN loss $\mathcal{L}_\theta(.)$ and projection loss $\mathcal{L}^p(.)$ that biases the projection network to mimic and learn from the full trainer network. The objective also incorporates a labeled loss $\widehat{\mathcal{L}}^p$ for the projection network.

We propose a new objective and joint optimization framework for training compact on-device models for inference. The architecture uses a *trainer* network coupled with a *projection* network and trains them jointly. Figure 1 illustrates the Neural Projection Network architecture using a feed-forward NN for the trainer network. The coupled networks are jointly trained to optimize a combined loss function:

$$\mathcal{L}(\theta, p) = \lambda_1 \cdot \mathcal{L}_\theta(.) + \lambda_2 \cdot \mathcal{L}^p(.) + \lambda_3 \cdot \widehat{\mathcal{L}}^p(.) \tag{1}$$

3

where $\mathcal{L}_\theta(.)$, $\mathcal{L}^p(.)$ and $\widehat{\mathcal{L}}^p(.)$ are the loss functions corresponding to the two networks as defined below.

$$\mathcal{L}_\theta(.) = \sum_{i \in N} \mathcal{D}(h_\theta(\vec{x}_i), \widehat{y}_i)$$

$$\mathcal{L}^p(.) = \sum_{i \in N} \mathcal{D}(h^p(\vec{x}_i), h_\theta(\vec{x}_i))$$

$$\widehat{\mathcal{L}}^p(.) = \sum_{i \in N} \mathcal{D}(h^p(\vec{x}_i), \widehat{y}_i) \tag{2}$$

N indicates the number of training instances in the dataset, $\vec{x}_i$ represents the input feature vector in a feed-forward network or sequence input in an RNN, and $\widehat{y}_i$ refers to the ground-truth output classes used for network training. $h_\theta(\vec{x}_i)$ represents a parameterized representation of the hidden units in the *trainer* network that transforms $\vec{x}_i$ to an output prediction y_i. Similarly, $h^p(\vec{x}_i)$ represents the *projection* network parameters that transforms the input to corresponding predictions y_i^p. We apply softmax activation at the last layer of both networks to compute the predictions y_i and y_i^p.

$\mathcal{D}$ denotes a distance function that measures the prediction error used in the loss functions. This is decomposed into three parts—trainer prediction error, projection simulation error and projection prediction error. Reducing the first leads to a better trainer network and decreasing the latter in turn learns a better projection network that is simpler but with approximately equivalent predictive capacity. In practice, we use cross-entropy for $\mathcal{D}(.)$ in all our experiments. For the *projection* $\mathcal{L}^p$ in Equation 2, we follow a distillation approach [19] to optimize $\mathcal{D}(.)$ since it has been shown to yield better generalization ability than a model trained on just the labels $\widehat{y}_i$. λ_1, λ_2 and λ_3 are hyperparameters that affect the trade-off between these different types of errors. These are tuned on a small heldout development set and in our experiments, we set them to $\lambda_1 = 1.0, \lambda_2 = 0.1, \lambda_3 = 1.0$.

Trainer Network (θ). The *trainer* model is a full neural network (feed-forward, RNN or CNN) whose choice is flexible and depends on the task. Figure 1 shows a trainer using feed-forward network but this can be swapped with LSTM RNNs (as we show later in Section 4.3) or other deep neural networks. For the network shown in the figure, the activations for $h_\theta(.)$ in layer l_{k+1} is computed as follows:

$$A_{\theta_{l_{k+1}}} = \sigma(W_{\theta_{l_{k+1}}} \cdot A_{\theta_{l_k}} + B_{\theta_{l_{k+1}}}) \tag{3}$$

where σ is the ReLU activation function [23] applied at each layer except the last and A indicates the computed activation values for hidden units.

The number of weights/bias parameters W_θ, B_θ in this network can be arbitrarily large since this will only be used during the training stage which can be effectively done using high-performance distribtuted computing with CPUs or GPUs.

Projection Network (p). The *projection* model is a simple network that encodes a set of efficient-to-compute operations which will be performed directly on device for inference. The model itself defines a set of efficient "projection" functions $\mathbb{P}(\vec{x}_i)$ that project each input instance $\vec{x}_i$ to a different space $\Omega_\mathbb{P}$ and then performs learning in this space to map it to corresponding outputs y_i^p. We use a simplified *projection* network with few operations as illustrated in Figure 1. The inputs $\vec{x}_i$ are transformed using a series of T projection functions $\mathbb{P}^1, ..., \mathbb{P}^T$, which is then followed by a single layer of activations.

$$\vec{x}_i^p = \mathbb{P}^1(\vec{x}_i), ..., \mathbb{P}^T(\vec{x}_i) \tag{4}$$

$$y_i^p = \text{softmax}(W^p \cdot \vec{x}_i^p + B^p) \tag{5}$$

The projection transformations use pre-computed parameterized functions, i.e., they are not trained during the learning process, and their outputs are concatenated to form the hidden units for subsequent operations. During training, the simpler *projection* network learns to choose and apply specific projection operations $\mathbb{P}^j$ (via activations) that are more predictive for a given task. It is possible to stack additional layers connected to the bit-layer in this network to achieve non-linear combinations of projections.

The *projection* model is jointly trained with the *trainer* and learns to mimic predictions made by the full trainer network which has far more parameters and hence more predictive capacity. Once learning is completed, the transform functions $\mathbb{P}(.)$ and corresponding trained weights W^p, B^p from the projection network are extracted to create a lightweight model that is pushed to device. At

inference time, the lightweight model and corresponding operations is then applied to a given input $\vec{x}_i$ to generate predictions $y_i^{\mathbb{P}}$.

The choice of the type of projection matrix $\mathbb{P}$ as well as representation of the projected space $\Omega_{\mathbb{P}}$ in our setup has a direct effect on the computation cost and model size. We propose to leverage an efficient randomized projection method using a modified version of locality sensitive hashing (LSH) to define $\mathbb{P}(.)$. In conjunction, we use a bit representation $\mathbf{1}^d$ for $\Omega_{\mathbb{P}}$, i.e., the network's hidden units themselves are represented using projected bit vectors. This yields a drastically lower memory footprint compared to the full network both in terms of number and size of parameters. We highlight a few key properties of this approach below:

- There is no requirement for committing to a preset vocabulary or feature space unlike typical machine learning methods which resort to smaller vocabulary sizes as a scaling mechanism. For example, LSTM RNN models typically apply pruning and use smaller, fixed-size vocabularies in the input encoding step to reduce model complexity.

- The proposed learning method scales efficiently to large data sizes and high dimensional spaces. This is especially useful for natural language applications involving sparse high dimensional feature spaces. For dense feature spaces (e.g., image pixels), existing operations like fully-connected layers (or even convolutions) can be efficiently approximated for prediction without relying on a large number of parameters. Such operations can also be applied in conjunction with the projection functions to yield more complex *projection* networks while constraining the memory requirements.

- Computation of $\mathbb{P}(x_i)$ is independent of the training data size.

- We ensure that $\mathbb{P}(.)$ is efficient to compute on-the-fly for inference on device.

Next, we describe the projection method and associated operations in more detail.

3.2 Locality Sensitive Projection Network

The *projection* network described earlier relies on a set of transformation functions $\mathbb{P}$ that project the input $\vec{x}_i$ into hidden unit representations $\Omega_{\mathbb{P}}$. The projection operations outlined in Equation 4 can be performed using different types of functions. One possibility is to use feature embedding matrices pre-trained using word2vec [24] or similar techniques and model $\mathbb{P}$ as a embedding lookup for features in $\vec{x}_i$ followed by an aggregation operation such as vector averaging. However, this requires storing the embedding matrices which incurs additional memory complexity.

Instead, we employ an efficient randomized projection method for this step. We use locality sensitive hashing (LSH) [25] to model the underlying projection operations. LSH is typically used as a dimensionality reduction technique for applications like clustering [26]. Our motivation for using LSH within Projection Nets is that it allows us to project similar inputs $\vec{x}_i$ or intermediate network layers into hidden unit vectors that are nearby in metric space. This allows us to transform the inputs and learn an efficient and compact network representation that is only dependent on the inherent dimensionality (i.e., observed features) of the data rather than the number of instances or the dimensionality of the actual data vector (i.e., overall feature or vocabulary size). We achieve this with binary hash functions [25] for $\mathbb{P}$.

Theorem 1 *For $\vec{x}_i, \vec{x}_j \in \mathbb{R}^n$ and vectors $\mathbb{P}_k$ drawn from a spherically symmetric distribution on $\mathbb{R}^n$ the relation between signs of inner products and the angle $\angle(\vec{x}_i, \vec{x}_j)$ between vectors can be expressed as follows:*

$$\angle(\vec{x}_i, \vec{x}_j) = \pi \, \Pr\{sgn[\langle \vec{x}_i, \mathbb{P}_k \rangle] \neq sgn[\langle \vec{x}_j, \mathbb{P}_k \rangle]\} \tag{6}$$

This property holds from simple geometry [25], i.e., whenever a row vector from the projection matrix $\mathbb{P}$ falls inside the angle between the unit vectors in the directions of $\vec{x}_i$ and $\vec{x}_j$, they will result in opposite signs. Any projection vector that is orthogonal to the plane containing $\vec{x}_i \vec{x}_j$ will not have an effect. Since inner products can be used to determine parameter representations that are nearby, $\langle \vec{x}_i, \vec{x}_j \rangle = ||\vec{x}_i|| \cdot ||\vec{x}_j|| \cdot cos\angle(\vec{x}_i, \vec{x}_j)$, therefore we can model and store the network hidden activation unit vectors in an efficient manner by using the signature of a vector in terms of its signs.

Computing Projections. Following the above property, we use binary hashing repeatedly and apply the projection vectors in $\mathbb{P}$ to transform the input $\vec{x}_i$ to a binary hash representation denoted by $\mathbb{P}_k(\vec{x}_i) \in \{0, 1\}^d$, where $[\mathbb{P}_k(\vec{x}_i)] := sgn[\langle \vec{x}_i, \mathbb{P}_k \rangle]$. This results in a d-bit vector representation, one bit corresponding to each projection row $\mathbb{P}_{k=1...d}$.

The projection matrix $\mathbb{P}$ is fixed prior to training and inference. Note that we never need to explicitly store the random projection vector $\mathbb{P}_k$ since we can compute them on the fly using hash functions rather than invoking a random number generator. In addition, this also permits us to perform projection operations that are linear in the *observed* feature size rather than the *overall* feature size which can be prohibitively large for high-dimensional data, thereby saving both memory and computation cost. The binary representation is signficant since this results in a significantly compact representation for the *projection* network parameters that in turn reduces the model size considerably compared to the *trainer* network.

Note that other techniques like quantization or weight sharing [9] can be stacked on top of this method to provide small further gains in terms of memory reduction.

Projection Parameters. In practice, we employ T different projection functions $\mathbb{P}^{j=1...T}$ as shown in Figure 1, each resulting in d-bit vector that is concatenated to form the projected activation units $\vec{x}_i^p$ in Equation 4. T and d vary depending on the *projection* network parameter configuration specified for $\mathbb{P}$ and can be tuned to trade-off between prediction quality and model size.

3.3 Training and Inference

We use the compact bit units to represent the *projection* network as described earlier. During training, this network learns to move the gradients for points that are nearby to each other in the projected bit space $\Omega_{\mathbb{P}}$ in the same direction. The direction and magnitude of the gradient is determined by the *trainer* network which has access to a larger set of parameters and more complex architecture. The two networks are trained jointly using backpropagation. Despite the joint optimization objective, training can progress efficiently with stochastic gradient descent with distributed computing on high-performance CPUs or GPUs.

Once trained, the two networks are de-coupled and serve different purposes. The *trainer* model can be deployed anywhere a standard neural network is used. The simpler *projection* network model weights along with transform functions $\mathbb{P}(.)$ are extracted to create a lightweight model that is pushed to device. This model is used directly "on" device at inference time by applying the same operations in Equations 4, 5 (details described in Sections 3.1, 3.2) to a new input $\vec{x}_i$ and generate predictions y_i^p.

Complexity. The overall complexity for inference is $O(n \cdot T \cdot d)$, where n is the observed feature size (*not* overall vocabulary size) which is linear in input size, d is the number of LSH bits specified for each projection vector $\mathbb{P}_k$, and T is the number of projection functions used in $\mathbb{P}$. The model size (in terms of number of parameters) and memory storage required for the *projector* inference model in this setup is $O(T \cdot d)$.

Other Extensions. It is possible to extend the proposed framework to handle more than one type of *trainer* or *projection* network and even simultaneously train several models at multiple resolutions using this architecture We leave this for future work.

As an alternative to the bit vector representation $\Omega_{\mathbb{P}}$, the projection matrix $\mathbb{P}$ can instead be used to generate a sparse representation of hidden units in the *projection* network. Each d-bit block can be encoded as an integer instead of a bit vector. This results in a larger parameter space overall $O(T \cdot 2^d)$ but can still be beneficial to applications where the actual number of learned parameters is tiny and inference can be performed via efficient sparse lookup operations.

4 Experiments

In this section we demonstrate the effectiveness of the proposed approach with several experiments on different benchmark datasets and classification tasks involving visual recognition and language understanding. The experiments are done using a TensorFlow implementation [27].

Baselines and Method. We compare ProjectionNets at different model sizes with full-sized deep neural network couterparts for each task. The deep network architecture varies depending on the task type–feed-forward network are employed for visual tasks whereas recursive neural networks (LSTMs) are used for language understanding task. In each case, the full neural network model used as baseline is also employed as *trainer* network to learn the small-sized ProjectionNets.

Evaluation. For each task, we compute the performance of each model in terms of precision at different ranks K, i.e., accuracy % within the top K predicted output classes. Models were trained

6

using multiple runs, each experiment was run for a fixed number of (400k) time steps with a batch size of 200 for the visual tasks and 100 for the text classification task. The observed variance across runs wrt accuracy was small, around $\pm 0.1\%$.

We also compute the *Compression Ratio* $= \frac{\text{\# parameters in Baseline deep network}}{\text{\# parameters in proposed model}}$ achieved by varying-sized models compared to the baseline deep network model.

The model size ratios reported here are based on number of free parameters and not wrt actual model size stored on disk. Note that this is slightly unfair to the ProjectionNet models and overlooks some of its advantage since the internal bit representations are far more efficient to store compactly on disk than standard neural network representations.

4.1 MNIST Handwritten Digit Classification

We compare the performance of different approaches using the original MNIST handwritten digit dataset [28]. The dataset contains 60k instances for training and 10k instances for testing. We hold out 5k instances from the training split as dev set for tuning system parameters.

For the baseline and *trainer* network, we use a feed-forward NN architecture (3 layers, 1000 hidden units per layer) with L2-regularization. Table 1 shows results of the baseline and comparison to ProjectionNet models with varying sizes (T, d). The results demonstrate that a tiny ProjectionNet with a remarkably high compression ratio of 388x is able to achieve a very high accuracy of 99.1%, in fact slightly better than the baseline that is significantly larger in memory footprint. Moreover, ProjectionNet models are able to achieve even further reduction in model size (upto 2000x-3500x) while yielding around 70-80% precision for top-1 and 90-94% precision among top-3 predictions.

Going deeper with projections: Furthermore, going deeper with the projection network architecture (i.e., adding more layers) improves prediction performance even though it adds more parameters, overall size still remains significantly small compared to the baseline model. For example, a single-layer ProjectionNet with $T = 60, d = 10$ produces an accuracy of 91.1% whereas a 2-layer deep ProjectionNet with the same projection layer followed by a fully connected layer (128 hidden units) improves the accuracy considerably to 96.3%, yielding a gain of +5.2%.

Different training objectives: The training objective can be varied by removing specific loss components from Equation 1. We observe that using the joint architecture helps significantly and results in the best performance. The ProjectionNet $[T = 60, d = 12]$ from Table 1 achieves better accuracy (99.1%) than fully trained baseline network. The same ProjectionNet trained *without* the full joint objective does significantly worse—91-92% when trained using $\widehat{\mathcal{L}}^p$ alone or using only $L_\theta + L^p$. We also trained smaller baseline neural networks with fewer layers and parameters for comparison and observed that the best ProjectionNet significantly outperforms a smaller NN that has 14.2x more parameters but only achieves 98.0%. The same trend follows when comparing against other simpler regularized linear baseline methods which perform far worse and produce accuracies comparable or worse than projection networks trained in isolation without the joint architecture. We notice that on more complex problems involving large output space, pre-training the trainer network and then performing joint training helps the projection net converge faster to a good performance.

Table 1: Results for MNIST Classification using Neural Projection Nets and baselines.

Model	Compression Ratio	Precision@1	Precision@3	Precision@5
NN	1	98.9	99.9	100.0
ProjectionNet				
$[T = 8, d = 10]$	3453	70.6	89.6	96.5
$[T = 10, d = 12]$	2312	76.9	93.9	98.1
$[T = 60, d = 10]$	466	91.1	98.7	99.6
$[T = 60, d = 12]$	388	99.1	99.9	100.0

4.2 CIFAR-100 Image Classification

Next we evaluate the proposed approach on the CIFAR-100 [29] image classification task. The CIFAR-100 training dataset consists of 50k 32x32 color images in 100 classes, each with 500 images. Each image is tagged with a fine-grained class that it belongs to (e.g., *dolphin, seal, tulips, orchids*). The test dataset comprises of a separate 10k images, with 100 images per class.

This task is much harder than MNIST digit classification. Our goal for this setup is not an attempt to build a new image recognition system and outperform existing state-of-the-art which would require a lot of complex task-specific architecture customizations and convolution operations. Instead, we aim to train a standard deep neural network for this task and contrast it with various ProjectionNets to determine how much compression can be achieved and at what quality trade-off compared to the baseline.

We use a feed-forward NN architecture (3 layers, 1000 hidden units per layer) for the baseline and *trainer* network. Table 2 shows results comaprisons between ProjectionNets and the baseline. As mentioned, this task is more complex than MNIST hence precision numbers are lower. However, we observe that a ProjectionNet model [T=60, d=12] is able to achieve more than 70x model compression with 33% relative reduction in precision@5 compared to the baseline. The model size can be further reduced, upto 430x [T=10,d=12], while incurring an overall 42% relative reduction in precision@5. On the other hand, a deeper projection network significantly reduces this performance gap to just 24% while achieving a 49x model compression rate.

Table 2: Results for CIFAR-100 Image Classification using Neural Projection Nets and baselines.

Model	Precision@1	Precision@3	Precision@5
NN	32.9	50.2	58.6
ProjectionNet			
$[T = 10, d = 12]$	12.8	25.2	33.8
$[T = 60, d = 12]$	17.8	30.1	39.2
Projection layer $[T = 60, d = 12]$ + FullyConnected [128 units]	20.8	35.6	44.8

4.3 Semantic Intent Classification

We compare the performance of the neural projection approach for training RNN sequence models (LSTM) for a semantic intent classification task as described in the recent work on SmartReply [5] for automatically generating short email responses. One of the underlying tasks in SmartReply is to discover and map short response messages to semantic intent clusters.[1] We choose 20 intent classes and created a dataset comprised of 5,483 samples (3,832 for training, 560 for validation and 1,091 for testing). Each sample instance corresponds to a short response message text paired with a semantic intent category that was manually verified by human annotators. For example, *"That sounds awesome!"* and *"Sounds fabulous"* belong to the *sounds good* intent cluster.

We use an RNN sequence model with multilayer LSTM architecture (2 layers, 100 dimensions) as the baseline and *trainer* network. The LSTM model and its ProjectionNet variant are also compared against other baseline systems—*Random* baseline ranks the intent categories randomly and *Frequency* baseline ranks them in order of their frequency in the training corpus. We show in table 3 that a ProjectionNet trained using LSTM RNNs achieves 82.3% precision@1, only 15% relative drop compared to the baseline LSTM but with significant reduction in memory footprint and computation (compared to LSTM unrolling steps). At higher ranks, the ProjectionNets achieve very high accuracies (90+%) close to the baseline LSTM performance.

Table 3: Results for Semantic Intent Classification using Neural Projection Nets and baselines.

Model	Precision@1	Precision@3	Precision@5
Random	5.2	15.0	27.0
Frequency	9.2	27.5	43.4
LSTM	96.8	99.2	99.8
LSTM-ProjectionNet			
$[T = 60, d = 12]$	82.3	93.5	95.5

5 Computing Model Predictability in Neural Bits

We further study the notion of predictability of deep neural networks in the context of *neural projections*. More specifically, using the ProjectionNet framework described in Section 3.1, we formulate the questions: "How many neural bits are required to solve a given task?" "How many bits are required to capture the predictive power of a given deep neural network?"

---

[1]For details regarding SmartReply and how the semantic intent clusters are generated, refer [5].

These motivate a series of studies on the different datasets. We show empirical results to answer these by computing the number of bits encoded in the *projection* network (Section 3.1) used in each task. Since the projection neural network architecture is represented in bits (i.e., output of projection operations result in bit vectors) and we only use a single layer of projections in our experiments, we can compute the total number of bits required to represent a specific ProjectionNet model. If we compare this to the accuracy achieved on a given task by applying the *projection* network alone for inference, this helps answers the first question.

On the visual recognition task for MNIST (Section 4.1), 80-100 neural projection bits are enough to solve the task with 70-80% accuracy, and increasing this to 720 bits already achieves 99.1% precision. For the language task of semantic classification (Section 4.3) involving sequence input, 720 neural projection bits are required to achieve a top-1 accuracy of 82.3%. To answer the second question, we use a given deep neural network (e.g, feed-forward NN) with specified configuration to model the full *trainer* network in the framework described in Section 3. We then use this to train a corresponding neural *projection* network with hidden bit representations. Finally, we compute the number of neural projection bits used in the second network that simulates the *trainer* and plot this value against the *predictive quality ratio*, which is defined as the ratio of accuracies achieved by separately performing inference with simple versus full network on a given task. Figure 2 shows this plot for the MNIST and CIFAR-100 tasks. The plot shows that the predictive power of a 3-layer feed-forward network with 3-5M parameters can be succinctly captured to a high degree (ratio=$\sim$0.8) with a simple 100-bit ProjectionNet for MNIST classification and just 720 bits are required to fully recover (and even outperform) the base deep network. On more complex image recognition tasks that involve larger output classes, a higher number of bits are required to represent a *trainer* deep network with the same architecture. For example, on the CIFAR-100 task, we observe that a 3-layer feed-forward network can be projected onto 720 neural bits at a predictive ratio of 0.5. However, we also notice a steep increase in predictive ratio moving from 120 to 720 neural bits. We expect a similar trend at even higher bit sizes and on more complex tasks.
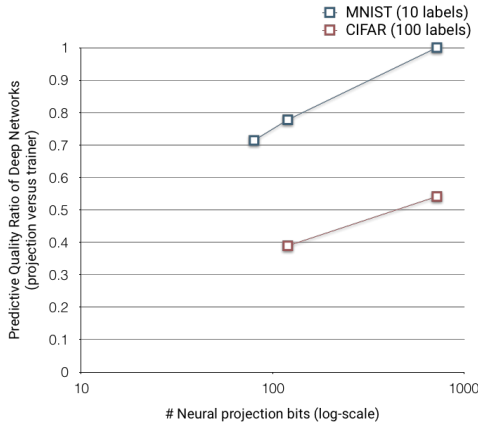

Figure 2: Comparing predictive power of deep neural networks using neural projection bits on different visual classification tasks.

6 Discussion and Future Work

We introduced a new Neural Projection approach to train lightweight neural network models for performing efficient inference on device at low computation and memory cost. We demonstrated the flexibility of this approach to variations in model sizes and deep network architectures. Experimental results on different visual and language classification tasks show the effectiveness of this method in achieving significant model size reductions and efficient inference while providing competitive performance. We also revisited the question of predictability of deep networks and study this in the context of neural bit projections.

A few possible future extensions to the framework are discussed at the end of Section 3. Going beyond deep learning, it is also possible to apply this framework to train lightweight models in other types of learning scenarios. For example, the training paradigm can be changed to a semi-supervised
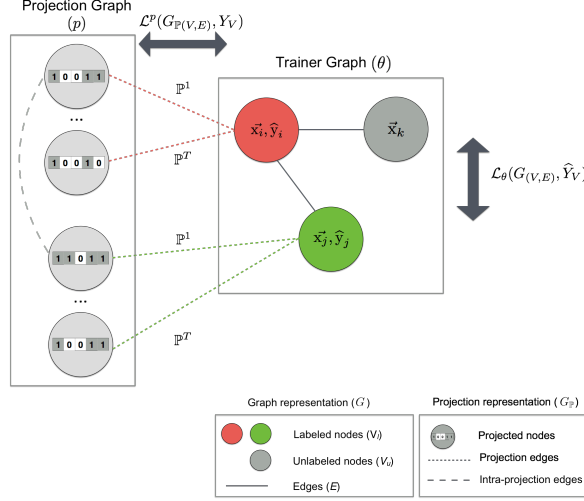
Figure 3: Illustration of a Projection Graph trained using graph learning algorithms. *Notation for trainer graph:* $\vec{x}_i$ represents an input node and $\vec{x}_j$, $\vec{x}_k$ represent its neighborhood $\mathcal{N}(\vec{x}_i)$ in the original graph $G = (V, E)$. E refers to edges and $V = V_l \cup V_u$ the nodes in this graph, where V_l indicates labeled nodes *(red, green)* and V_u the unlabeled nodes *(grey)*. $\hat{y}_i$ indicates the ground-truth corresponding to a labeled node $\vec{x}_i \in V_l$, with *red, green* colors indicating different output values for $\hat{y}_i$. If $V = V_l$, the graph is trained in a supervised manner whereas $V_l \subset V; V_u \neq \varnothing$ yields a semi-supervised graph learning formulation. *Notation for projection graph:* Each node $\vec{x}_i$ in the trainer graph G is connected via an edge to a corresponding projection node in $G_\mathbb{P}$. The projection nodes are discrete representations of trainer node $\vec{x}_i$ obtained by applying the projection functions $\mathbb{P}^1...\mathbb{P}^T$ to the feature vector associated with the trainer node. In addition, $G_\mathbb{P}$ may also contain intra-projection edges computed using a similarity metric applied to the projection vector representations. For example, we can use Hamming distance $\mathcal{H}(.)$ to define a similarity metric $1 - \frac{\mathcal{H}(.)}{d}$ between projection nodes represented as $d-$bit vectors. The training objective optimizes a combination of graph loss $\mathcal{L}_\theta(.)$ and projection loss $\mathcal{L}^p(.)$ that biases the projection graph to mimic and learn from the full trainer graph. $\mathcal{L}_\theta(.)$ optimizes the trainer's predicted output y_i against the ground-truth $\hat{y}_i$ whereas $\mathcal{L}^p(.)$ optimizes predictions from the projection graph y_i^p against the neigbhoring trainer predictions y_i.

or unsupervised setting. The *trainer* model itself can be modified to incorporate structured loss functions defined over a graph or probabilistic graphical model instead of a deep neural network. Figure 3 illustrates an end-to-end *projection graph* approach to learning lightweight models using a graph optimized loss function which can be trained efficiently using large-scale distributed graph algorithms [30] or even neural graph approaches [31, 32]. The projection model training can also be further extended to scenarios involving distributed devices using complementary techniques [33]. We leave these explorations as future work.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

[2] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[5] A. Kannan, K. Kurach, S. Ravi, T. Kaufmann, A. Tomkins, B. Miklos, G. Corrado, L. Lukacs, M. Ganea, P. Young, and V. Ramavajjala, "Smart reply: Automated response suggestion for email," in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[6] M. Schuster, "Speech recognition for mobile devices at google," in *Proceedings of the 11th Pacific Rim International Conference on Trends in Artificial Intelligence*, PRICAI'10, (Berlin, Heidelberg), pp. 8–10, Springer-Verlag, 2010.

[7] K. H. Lee and N. Verma, "A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals," *IEEE Journal of Solid-State Circuits*, vol. 48, pp. 1625–1637, July 2013.

[8] "Android Wear 2.0: Make the most of every minute." `https://blog.google/products/android-wear/android-wear-20-make-most-every-minute/`.

[9] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[10] B.-G. Chun and P. Maniatis, "Augmented smartphone applications through clone cloud execution," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2009.

[11] A. Stolcke, "Entropy-based pruning of backoff language models," *CoRR*, vol. cs.CL/0006025, 2000.

[12] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, (New York, NY, USA), pp. 1113–1120, ACM, 2009.

[13] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan, "Hash kernels for structured data," *J. Mach. Learn. Res.*, vol. 10, pp. 2615–2637, Dec. 2009.

[14] K. Ganchev and M. Dredze, "Small Statistical Models by Random Feature Mixing," in *Proceedings of the ACL-08: HLT Workshop on Mobile Language Processing*, 2008.

[15] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," *CoRR*, vol. abs/1412.6115, 2014.

[16] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016.

[17] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, (USA), pp. 2148–2156, Curran Associates Inc., 2013.

[18] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, pp. 2285–2294, JMLR.org, 2015.

[19] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *ArXiv e-prints*, Mar. 2015.

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, Jan. 2014.

[21] J. Wang, W. Liu, S. Kumar, and S. Chang, "Learning to hash for indexing big data - A survey," *CoRR*, vol. abs/1509.05472, 2015.

[22] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *ArXiv e-prints*, June 2014.

[23] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (J. Frnkranz and T. Joachims, eds.), pp. 807–814, Omnipress, 2010.

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[25] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, (New York, NY, USA), pp. 380–388, ACM, 2002.

[26] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[28] "The MNIST database of handwritten digits." `http://yann.lecun.com/exdb/mnist/`.

[29] "The CIFAR dataset." `http://www.cs.toronto.edu/~kriz/cifar.html`.

[30] S. Ravi and Q. Diao, "Large scale distributed semi-supervised learning using streaming approximation," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pp. 519–528, 2016.

[31] T. D. Bui, S. Ravi, and V. Ramavajjala, "Neural graph machines: Learning neural networks using graphs," *CoRR*, vol. abs/1703.04818, 2017.

[32] Z. Yang, W. W. Cohen, and R. Salakhutdinov, "Revisiting semi-supervised learning with graph embeddings," *CoRR*, vol. abs/1603.08861, 2016.

[33] J. Konecný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *CoRR*, vol. abs/1610.02527, 2016.