Accelerating Deep Learning Inference via Freezing

Adarsh Kumar Arjun Balasubramanian Shivaram Venkataraman Aditya Akella

University of Wisconsin - Madison

Abstract

Over the last few years, Deep Neural Networks (DNNs) have become ubiquitous owing to their high accuracy on real-world tasks. However, this increase in accuracy comes at the cost of computationally expensive models leading to higher prediction latencies. Prior efforts to reduce this latency such as quantization, model distillation, and any-time prediction models typically trade-off accuracy for performance. In this work, we observe that caching intermediate layer outputs can help us avoid running all the layers of a DNN for a sizeable fraction of inference requests. We find that this can potentially reduce the number of effective layers by half for 91.58% of CIFAR-10 requests run on ResNet-18. We present Freeze Inference, a system that introduces *approximate caching* at each intermediate layer and we discuss techniques to reduce the cache size and improve the cache hit rate. Finally, we discuss some of the open research challenges in realizing such a design.

1 Introduction

The field of artificial intelligence (AI) has made rapid strides over the past few years largely due to the progress in Deep Neural Networks (DNNs). DNNs have surpassed human-level accuracy on tasks ranging from speech recognition [20], image classification [5, 10, 12, 14–16] to machine translation [4]. However, this gain in accuracy has come with models becoming deeper leading to increased computational requirements. For example, in object classification, the top-5 classification accuracy has increased from 71% in 2012 to 97% in 2015 on the ImageNet dataset, while the models have become 20× more computationally expensive. This increase in computation also leads to longer latencies during prediction or inference where low user response time is paramount. To efficiently serve these models, there is a need to reduce the overall computation needed for inference, without trading off accuracy.

There have been several efforts to reduce the computational complexity of DNNs to improve model serving. A number of previous efforts have proposed compressing the model using techniques such as quantization [1, 2] or model distillation [6], but such techniques typically hurt accuracy. On the other hand, ensemble methods [13] or any-time prediction models [7] aim to provide a better trade-off between accuracy and latency by building models of varying complexity. However, this either requires re-training using custom model architectures or training a number of models ahead of time. Systems such as Clipper [3] improve serving by batching queries and optimizing their execution within a batch. These techniques typically improve throughput and making inference latency-aware. Finally, PRETZEL [11] improves latencies using multi-model optimizations but does not focus on reducing compute for a single given model.

In this work, we introduce *caching* as a technique to reduce the prediction latency of DNNs. Caches in general are used to improve the latency of Web requests by storing the output of previous requests. Previous works [3] have used caching at the input layer to improve the prediction latency, but they consider the DNN as a black box. Thus, in the event of a cache miss at the input layer, these systems have to run all the layers of the DNN to obtain a prediction. Instead the question we ask is: *Can we design caching such that we can avoid the need to run all the layers of a DNN for every input request?*

To this end, we propose augmenting DNNs with a cache at each layer, where the cache holds a succinct representation of intermediate layer outputs and their relation to the final classification. The rationale behind this is that each layer of the DNN tries to normalize the variations in the input that do not correlate with the output, and by doing so, tries to learn an embedding space where similar data points are closer to each other. Thus, even when we do not get a "cache hit" for the input, we could get a "cache hit" at an intermediate layer. For example, with an object classification model, the background, brightness, contrast, etc of the input image do not correlate with the output class. The model will normalize these variations in the image, layer by layer. Thus, we can expect images of the same object with different backgrounds to be embedded closer in the projection space after some

DNN layers have been evaluated.

Maintaining a cache for intermediate layers comes with its own challenges. First, as the intermediate layer outputs are float tensors in a high dimensional space, the probability of exact match is low, leading to a low cache hit rate. Second, such a cache would require a large amount of memory owing to the high dimensionality of tensors and the size of training data. Finally, cache look-up time is poor for large caches that cannot fit in fast memories.

We introduce Freeze Inference, a system which augments DNNs with intermediate layer caches to reduce the prediction latency and addresses the above issues. For our initial prototype, Freeze Inference creates an offline cache, which stores the intermediate layer outputs computed over training data. We then train a dimensionality reduction model for each layer, which projects high dimensional intermediate layer tensors to a low dimensional space. Finally, we perform k-means clustering on the reduced dimensional space and only store the centroids of the clusters, which further reduces the memory footprint and look-up time.

The rest of the paper is organized as follows. In Sec. 2, we introduce the rationale behind caching in the context of DNNs. Next, we describe the design of our system Freeze Inference in Sec. 3. Finally, we present the initial results of our prototype (Sec. 4) and conclude with discussions on future research directions (Sec. 5).

2 Intuition

We begin by describing some key properties of DNNs and how layer-wise caching as we envision it can be applied in this setting. In a DNN, the input is represented as a set of features, where each feature is a value provided to individual nodes at the DNN's input layer. The DNN has a number of hidden layers each consisting of multiple nodes, where each node applies a non-linear function to a weighted sum of its inputs. We refer to the individual hidden layer outputs as *intermediate layer* outputs in our work. Finally, there is an output layer which consists of one or more neurons that can cumulatively be viewed as making a prediction.

Given the architecture of DNNs, we make two important observations which form the basis for our work:

(O1) Given two inputs X_i and X_j which are exactly same, the DNN will predict the same label Y for both the inputs. This is because the same set of learned weights are used during inference which effectively means that each layer executes a deterministic function on its input. On similar lines, if two inputs X_i and X_j result in the same intermediate layer output at a given hidden layer, we expect the DNN to predict the same label for both the inputs.

(O2) Consider two inputs X_i and X_j whose output feature vectors reside close to each other in the output feature space. To predict labels, DNNs typically use a function such as softmax at the output layer which draws decision boundaries in the

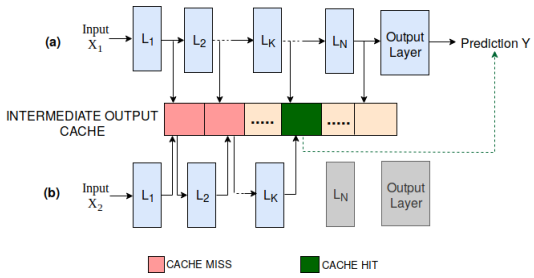


Figure 1: The basic idea behind Freeze Inference. (a) The intermediate outputs are cached. (b) During inference, a cache look-up is done after every layer and a cache hit yields a faster prediction

output feature space. Proximity in the output feature space means there is a high probability that the points lie within the same decision boundary and are assigned the same prediction Y by the DNN.

Figure 1 builds up towards the intuition behind Freeze Inference. Consider a DNN with N layers. Let us take two input feature vectors X_1 and X_2. Let us say that each input produces intermediate layer outputs $L_{i,j}$, where i is the layer number and j is an identifier for the input under consideration. Now, let us consider a situation where X_1 has already run through the DNN to obtain a prediction Y. We store each of it's intermediate layer outputs $L_{i,1}$ for $i = 1,2,3..N$ along with the final predicted label Y in a per-layer cache. We now try to predict the label for input X_2 as follows: after the computation at each layer, we additionally compare the obtained intermediate output to the contents of the corresponding layer's cache. During this process, let us say that at some layer K we observe that $L_{K,1}$ and $L_{K,2}$ are the same. From observation O1, we can conclude that the intermediate outputs of successive layers would also be the same ultimately leading to the same prediction. More formally, if K is the smallest layer at which we have $L_{K,1} = L_{K,2}$, then we can say that $L_{i,1} = L_{i,2}$ for $i = K+1, K+2, ..N$ and both X_1 and X_2 would have the same predicted label Y. Hence, it is possible to skip expensive computations for layer $K+1, K+2, ..N$ when the intermediate layer output for a layer K matches an intermediate output that has been cached. In such a scenario, we can *freeze* the computation at layer k and return the cached output.

2.1 Towards Approximate Caching

Since feature vectors have high dimensionality and are represented by a set of floats, it is highly unlikely that that two intermediate layer outputs would be exactly the same. Therefore, a caching mechanism based on exact matches would not generate enough cache hits to provide meaningful computational benefits. To address this, we leverage the insight from observation O2 in that DNNs try to identify decision boundaries in order to classify items. To empirically validate this, we took a set of 50,000 images belonging to the CIFAR-10 [8] dataset and ran a complete forward pass for each image on

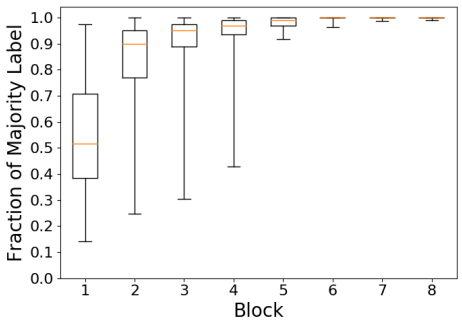


Figure 2: Distribution of fractional share of majority label per cluster for ResNet-18 on CIFAR-10

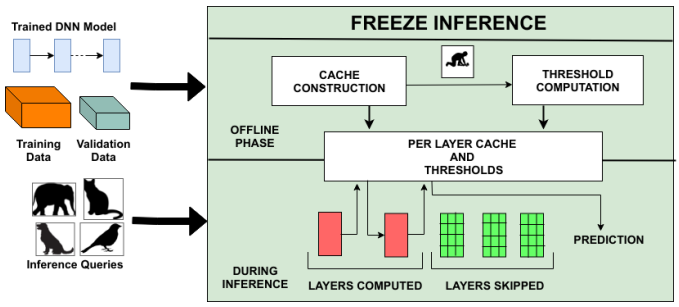


Figure 3: Freeze Inference High Level Design

the ResNet-18 [5] model. From this, we constructed a set of intermediate layer outputs for each ResNet block[1] and tagged each output with the label predicted by the model. For each ResNet block, we then arranged the outputs into 200 clusters using k-means and computed the majority label occupying each cluster along with the fractional share of the label within that cluster. From Figure 2, we notice that there is a dominant majority label in each cluster. For instance, in Block 4, we see that the mean fraction of the majority label is 0.95. This provides empirical backing that there exists a semantic relationship between points that lie nearby to each other in the intermediate feature space. Another interesting observation is that the mean fraction of the majority label increases as we move across the layers, indicating that points get better correlated in the feature space as we go deeper in the DNN.

We leverage the above ideas in Freeze Inference by constructing an offline, per-layer cache consisting of intermediate layer outputs and their corresponding labels. We augment the inference control flow to perform an approximate cache look-up for the intermediate output at each layer by using an algorithm like k-nearest neighbors. We use information such as the labels of the k neighbors and their distances from the input point to make a prediction and offer a notion of *confidence* about the prediction. We characterize the cache look-up at that layer as a *hit* if the offered *confidence* exceeds a defined *threshold* for that layer.

3 Freeze Inference Design

The first major challenge in Freeze Inference is that the intermediate layer outputs reside in high-dimensional space. Apart from resulting in low cache hit rates, high memory usage, and increasing the computational complexity of cache look-up, prior work [18] has shown that performance of similarity search degrades in high dimension. This is a problem since Freeze Inference relies on the notion of closeness to infer semantic similarity. We overcome this by using dimensionality reduction. Inspired by Metric Learning [19], we do this using a one layer neural network whose hidden layer consists of 1024 nodes. We train a per-layer dimensionality reduction model using the intermediate layer outputs and labels predicted by the model for the training data set.

Next, we define the semantics of approximate cache look-ups by describing a cache look-up API. The API takes in an intermediate layer representation as an argument and returns a prediction along with a confidence value. Following from the discussion in Section 2.1, the API computes k-nearest neighbors on the input and obtains a set of k tuples, where each tuple consists of the label of the neighbor and its distance from the input. In our initial design, we use a heuristic for computing the predicted label and the associated confidence value. Our heuristic is based on the intuition that predictions can be more confident if (a) more neighbors agree on the same label and (b) the neighbors are close to the input under consideration. Let us say that the dataset has N labels $n_1, n_2 ..., n_N$. For a given input point, suppose label n_i has m_i occurrences amongst the k neighbors for the input at a specific layer of the DNN. Let the distances associated with the m_i occurrences be $d_1, ., d_j, ., d_{m_i}$. We first compute n_i label's fractional share amongst the k neighbors as $S_i = \frac{m_i}{k}$. We then compute the confidence $C_i = S_i \times \sum_{j=1}^{m_i} \frac{1}{d_j}$ for each label. The API returns the label having maximum confidence as the predicted label for the layer along with the associated confidence.

Figure 3 presents the Freeze Inference pipeline. It consists of an offline phase which aggregates information to be used during inference. The offline phase consists of two parts - (i) Cache Construction (ii) Threshold Computation. The above two steps lead to the construction of per-layer caches and per-layer thresholds which are then passed onto the online phase. These structures are used during the online inference phase to perform cache look-up and characterize cache hits. We describe the individual steps in detail below.

3.1 Offline Phase

(i) Cache Construction: Freeze Inference takes in a trained model and constructs per-layer caches by running a forward pass of the DNN for each example and caching the dimensionally reduced intermediate layer outputs for each layer (Line 22–30 in Algorithm 1).

[1] ResNet-18 consists of 8 blocks, where each block consists of 2 convolutional layers and 1 residual connection

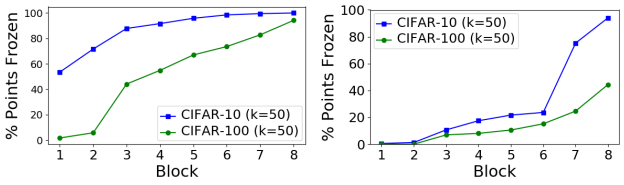
Pseudocode 1 Freeze Inference Workflow

```
1: CACHE = {}                                              ▷ Per-layer cache
2: THRESHOLDS = {}                                         ▷ Per-layer thresholds
3:
4: ▷ Given a model M, perform offline pre-processing for Freeze Inference
5: procedure OFFLINEPHASE(Model M, TrainData TD, ValidationData VD)
6:     CACHE = CONSTRUCTCACHE(M, T)
7:     THRESHOLDS = COMPUTETHRESHOLDS(M, V, CACHE)
8: end procedure
9:
10: ▷ Given a model M, perform Freeze Inference on input I
11: procedure FREEZEINFERENCE(Model M, Input I)
12:     for all layer ∈ M.layers() do
13:         layerOutput = forward pass on M for next layer
14:         pred_label, confidence = prediction for layerOutput from CACHE[layer]
15:         if confidence > THRESHOLDS[layer] then
16:             return predicted_label
17:         end if
18:     end for
19:     return label predicted by output layer of M
20: end procedure
21:
22: procedure CONSTRUCTCACHE(Model M, TrainData TD)
23:     IO[i][j]                                  ▷ Intermediate output for TD[i] at layer j
24:     Y[i]                                      ▷ Label Predicted by M for TD[i]
25:     for all item ∈ TD do
26:         for all layer ∈ M.layers() do
27:             CACHE[layer].append(<IO[item][layer], Y[item]>)
28:         end for
29:     end for
30: end procedure
31:
32: procedure COMPUTETHRESHOLDS(Model M, ValidationData VD, Cache C)
33:     IO[i][j]                                  ▷ Intermediate output for VD[i] at layer j
34:     Y[i]                                      ▷ Label Predicted by M for VD[i]
35:     Prediction[i][j]      ▷ Label Predicted by look-up from C for VD[i] at layer j
36:     Confidence[i][j]      ▷ Confidence value of look-up from C for VD[i] at layer j
37:     for all item ∈ VD do
38:         for all l ∈ M.layers() do
39:             if Prediction[item][l] not equals Y[item] then
40:                 THRESHOLD[l] = max(THRESHOLD[l], Confidence[item][l])
41:             end if
42:         end for
43:     end for
44: end procedure
```

(ii) Threshold Computation: A critical piece of the Freeze Inference design is to develop the notion of a cache hit. For this purpose, we use a validation dataset to compute per-layer thresholds. For each item in the validation data, we perform a forward pass, reduce the dimension of the layer output and then do a cache look-up at each layer. For each layer, we set the threshold as the maximum confidence value that resulted in a wrong prediction on the validation set (Line 40 in Algorithm 1). Thus, Freeze Inference adopts a pessimistic approach by establishing strict thresholds and ensuring zero error on the validation data, which in turn maximizes the accuracy of cache hits during inference.

3.2 Online Phase - Inference

When an inference request comes in, we do forward propagation one layer at a time and a cache look-up on the dimensionally reduced output at each layer. If the confidence returned by cache look-up is greater than the established threshold for that layer, we skip the computation of the remaining layers and return the label predicted by cache look-up as the final predicted label. (Lines 11 to 20 in Algorithm 1).



(a) Upper Bound (b) Actual

Figure 4: CDF of points frozen w.r.t. blocks for ResNet-18 using k-NN

Cache Construction Scheme	Total Memory
k-NN without Dimensionality Reduction	37500.0 MB
k-NN with Dimensionality Reduction	2500.0 MB
k-Means with Dimensionality Reduction	12.5 MB

Table 1: Memory requirements for caching on ResNet-18

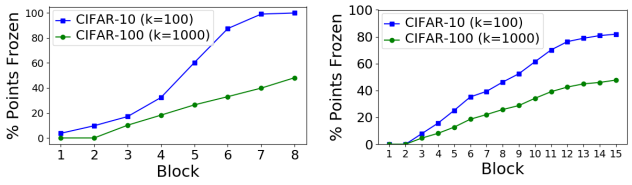
4 Results and Challenges

We evaluate our Freeze Inference system for the CIFAR-10 [8] and CIFAR-100 [9] datasets on the ResNet-18 model [5]. Figure 4(a) presents the earliest block at which an inference request can be *frozen* assuming that we have a perfect threshold calculation scheme. In this scenario, we observe that can potentially save half the computation time (run half of the total layers) for around 91% data-points in CIFAR-10 and 55% of the data-points for CIFAR-100. This represents an upper bound on the potential of Freeze Inference. Figure 4(b) shows the distribution of layers at which inference requests are *frozen* using our naive threshold calculation scheme. From the graph, we observe that our naive Freeze Inference saves half the computation for about 20% of the points on CIFAR-10 and around 15% for CIFAR-100 respectively. Overall, we are able to freeze about 95% and 44% of the data-points before the output layer in CIFAR-10 and CIFAR-100 respectively. The *k*-NN approach with our naive threshold calculation scheme achieves an accuracy of 97.48% and 99.03% with respect to the model's prediction for CIFAR-10 and CIFAR-100 datasets respectively. This shows that we are able to save computation without trading off too much on the accuracy.

Even though we are able to Freeze a significant percentage of data-points, *k*-NN has following overheads:

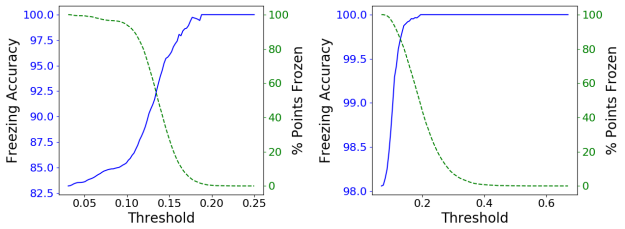
Computational Complexity: Computing *k*-nearest neighbors incurs significant overheads due to a large number of cached intermediate points to compare against. In our experiment, we had 40,000 training data points in the cache at each layer. To extract the most out of Freeze Inference, we would require the cache-lookup to be computationally cheap.

Memory Overheads: With caching, Freeze Inference incurs an additional overhead with respect to memory. Table 1 captures memory usage that the *k*-NN implementation of Freeze Inference would require. Though dimensionality reduction significantly reduces the memory overhead, we would still like the requirement to be as low as possible to allow Freeze Inference to scale well for larger datasets and models.



(a) ResNet-18 (b) ResNet-50

Figure 5: CDF of points frozen w.r.t. blocks using k-means



(a) Block 7 (b) Block 15

Figure 6: Trade-off between accuracy of frozen points and percentage of total points frozen as the threshold varies for block 7 and block 15 of ResNet-50

We can overcome the computational and memory overheads by leveraging the fact that intermediate layer points are semantically related to each other (Figure 2). In this light, we use k-means to cluster neighboring intermediate layer points and represent them by a single cluster center. This reduces the number of points that need to be stored in the cache and consequently reduces both the computational complexity and memory overheads. We construct a per-layer cache such that each item consists of a cluster center, the majority label and the fraction of majority label in that cluster. During inference, the API returns the majority label of the closest cluster as the predicted label and the ratio of the fraction of majority label to distance from the cluster center as the confidence value. The thresholding scheme used by k-means is the similar to the one described earlier for k-NN.

Figure 5 shows the distribution of layers at which inference requests are *frozen* using the k-means clustering approach for ResNet-18 and ResNet-50[2]. For ResNet 18, we observe that Freeze Inference saves half the computation for about 33% and 19% of the points on CIFAR-10 and CIFAR-100 respectively. Similarly, for ResNet-50, we are able to save half the computation for 46% and 26% of the points on CIFAR-10 and CIFAR-100 respectively. Overall, this approach achieves an accuracy of 92.85% and 88.86% with respect to the model's prediction for CIFAR-10 and CIFAR-100 datasets respectively. Figure 6 captures the trade-off between the percentage of points that Freeze Inference can *freeze* and the accuracy of those points for CIFAR-10 on ResNet-50. We observe that as we increase the threshold, the percentage of points frozen decreases which results in points getting frozen more accurately. This tells us that we can model thresholding as an optimization problem where we need to simultaneously maximize the percentage of points frozen and the accuracy with which they are frozen.

With respect to computation, our experiments show that cache look-up is about 20X faster than the compute for a single layer. Additionally, from Table 1 we see that the cache requires a mere 12.5MB of memory for ResNet-18. Thus, k-means solves the problems both with respect to computational complexity and memory requirements.

[2]ResNet-50 consists of 15 blocks, where each block consists of 3 convolutional layers and 1 residual connection.

5 Research Directions

Cache Size and Accuracy Trade-off: From the results, we notice that while k-means is able to lower the memory requirements this comes at the cost of reduced accuracy. We plan to study techniques that can further improve the heuristic used to compute confidence and thresholds per layer and also study the effect of disabling freeze inference at earlier layers (e.g., Block 4) as most of errors happen in the initial few layers.

Freeze Inference on GPUs: Batching of inference requests is a popular technique used to increase prediction throughput. With Freeze Inference, we would need to reconstruct the batch after each layer to remove the items that have been *frozen*. Handling dynamic batch sizes in GPUs across layers is an interesting research problem that needs to be investigated.

Cache lookup performance: Optimizing the cache lookup performance is important for realizing the benefits from freezing. While our current prototype uses a single thread on CPU to compute distance from centroids, we plan to investigate techniques to pipeline cache lookups with the forward pass being executed on GPUs.

Incremental Cache Update: In the current design, we use a cache that is constructed offline from the training data. To handle updates, we plan to identify common inference requests that were not frozen over a period of time, collect their intermediate layer representations and labels and once enough examples have accumulated, we can use these examples to recompute the thresholds. Performing online cache updates is a challenging problem especially with k-means as clusters and thresholds need to be re-computed for every update.

Acknowledgements. We thank Yingyu Liang, Arjun Singhvi and reviewers for their valuable feedback and suggestions. This work is supported by the National Science Foundation (CNS-1838733). Shivaram Venkataraman is also supported by a Facebook faculty research award and support for this research was also provided by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin, Madison with funding from the Wisconsin Alumni Research Foundation. Aditya Akella is also supported by a Google Faculty award, a gift from Huawei, and H. I. Romnes Faculty Fellowship.

6 Discussion Topics

In this paper we introduced Freeze Inference, a general technique to improve the latency of serving deep learning models by using cache and have shown that this direction has potential. This paper is likely to generate a discussion regarding the opportunities for not running all the layers of a DNN. Some points that we think will lead to discussion include:

Design approaches for an approximate cache: In this paper, we presented an initial approach at designing an approximate cache using k-NN and k-means clustering. If other techniques can improve the trade-off between cache size, cache lookup time, and accuracy, it will make for an interesting discussion.

Dynamic batching on GPUs: As discussed in Section 5, the problem of dynamically adjusting the batch size on GPUs is very interesting from a systems perspective. Solutions to this problem could also improve other techniques like SkipNets [17].

Effect of non-uniform request popularity: Finally our evaluation results consider a uniform distribution of requests from the test dataset. However in real world scenarios we often see a zipfian pattern with a few very popular requests and it will be interesting to discuss on how we can achieve greater benefits for such requests.

References

[1] CAI, Z., HE, X., SUN, J., AND VASCONCELOS, N. Deep learning with low precision by half-wave gaussian quantization. *CoRR abs/1702.00953* (2017).

[2] COURBARIAUX, M., AND BENGIO, Y. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR abs/1602.02830* (2016).

[3] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 613–627.

[4] HASSAN, H., AUE, A., CHEN, C., CHOWDHARY, V., CLARK, J., FEDERMANN, C., HUANG, X., JUNCZYS-DOWMUNT, M., LEWIS, W., LI, M., ET AL. Achieving human parity on automatic chinese to english news translation. *arXiv preprint arXiv:1803.05567* (2018).

[5] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[6] HINTON, G., VINYALS, O., AND DEAN, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[7] HUANG, G., CHEN, D., LI, T., WU, F., VAN DER MAATEN, L., AND WEINBERGER, K. Q. Multi-scale dense convolutional networks for efficient prediction. *CoRR abs/1703.09844* (2017).

[8] KRIZHEVSKY, A., NAIR, V., AND HINTON, G. Cifar-10 (canadian institute for advanced research).

[9] KRIZHEVSKY, A., NAIR, V., AND HINTON, G. Cifar-100 (canadian institute for advanced research).

[10] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105.

[11] LEE, Y., SCOLARI, A., CHUN, B.-G., SANTAMBROGIO, M. D., WEIMER, M., AND INTERLANDI, M. PRETZEL: Opening the black box of machine learning prediction serving systems. 611–626.

[12] LIN, M., CHEN, Q., AND YAN, S. Network in network. *CoRR abs/1312.4400* (2013).

[13] SHEN, H., PHILIPOSE, M., AGARWAL, S., AND WOLMAN, A. Mcdnn: An execution framework for deep neural networks on resource-constrained devices. Tech. rep., December 2015.

[14] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[15] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)* (2015).

[16] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. *CoRR abs/1512.00567* (2015).

[17] WANG, X., YU, F., DOU, Z., AND GONZALEZ, J. E. Skipnet: Learning dynamic routing in convolutional networks. *CoRR abs/1711.09485* (2017).

[18] WEBER, R., SCHEK, H.-J., AND BLOTT, S. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB* (1998), pp. 194–205.

[19] XING, E. P., JORDAN, M. I., RUSSELL, S. J., AND NG, A. Y. Distance metric learning with application to clustering with side-information. In *Advances in neural information processing systems* (2003), pp. 521–528.

[20] XIONG, W., , HUANG, X., SEIDE, F., , AND STOLCKE, A. Toward human parity in conversational speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing 25* (Sept 2017), 2410–2423.