The Power of Inception: Tackling the Tiny ImageNet Challenge

Pedro M. Milani
Stanford University
488 Escondido Mall, Stanford CA
pmmilani@stanford.edu

Abstract

The ImageNet Challenge is a fundamental tool to develop and benchmark visual recognition algorithms. For the CS-231N project, I worked on the Tiny ImageNet Challenge, which is a smaller version of the aforementioned competition. After a survey of previous work on the subject, I decided to use deep convolutional neural network algorithms based on the inception paradigm, first proposed by Szegedy et al. [18]. The goal of the present work is to maximize the top-1 accuracy on the test set. My final model obtained 30.9% top-1 test error, which put me tied for first place in the 2017 class leaderboard.

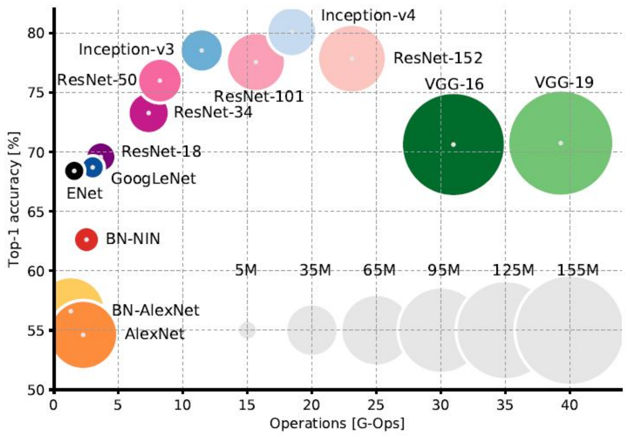


Figure 1. Comparison of different models taken from Canziani et al. [2]. Inception-v3 has the second highest accuracy at relatively modest computational cost and number of parameters.

1. Introduction

The ImageNet Large Scale Visual Recognition Challenge [14] is a competition where teams have access to a fixed dataset (1.2 million training images, 50k validation images, and 100k test images) and try to perform visual recognition tasks as well as possible on the test set. For the image classification task, each example belongs to one of 1000 possible categories. It is a critical benchmark of the current visual recognition algorithms, and at the same time an important tool for developing improved ones.

The Tiny ImageNet Challenge consists of a miniature version of the ImageNet Challenge, with fewer and smaller images sampled from the ImageNet dataset. It is therefore expected that appropriately adapted versions of the algorithms that perform well on ImageNet will also perform well on the classification task at hand. *The objective of the present work is to maximize the top-1 test accuracy on Tiny ImageNet, within CS-231N time and computational constraints*. So, it is important to analyze some of the previous successful approaches to the ImageNet Challenge.

Until 2011, hand-crafted features were extracted from images and used to predict their labels [3]. In 2012, Krizhevsky et al. made a groundbreaking improvement in classification accuracy by using deep convolutional neural networks (CNNs) [10]. Their AlexNet model obtained a single-crop, single-model top-1 validation error of 40.7% in the ImageNet Challenge. Ever since, CNNs have been used for most visual recognition tasks. Several improved architectures were developed following AlexNet. In 2015, He et al. [7] proposed ResNet, which consists of very deep networks using residual connections to assist gradient backpropagation. They obtained 21.4% 10-crop top-1 validation error on ImageNet. In the same year, Szegedy et al. debuted GoogleNet [18], which used inception layers as a way to perform convolutions efficiently. Follow-up papers improved on this idea: Inception-v3 [19] and Inception-v4 [17] obtained, respectively, 19.8% and 18.7% 12-crop top-1 validation errors on ImageNet.

Given the abundance of models that have been used recently, a survey of the most recent ones, along with their strengths and weaknesses, would be important to determine which one is the best choice for the current project. Canziani et al. [2] presents such survey. Figure 1 is taken from [2] and compares different architectures in terms of single-crop, single-model top-1 validation accuracy on ImageNet (y-axis), computational cost (x-axis), and number of parameters (size of the circle). The models with highest accuracy are Inception-v3 and Inception-v4. The latter requires significantly more operations and a somewhat higher number of parameters to achieve only a modest improvement over the former. Thus, to achieve the objective of maximizing top-1 accuracy on Tiny ImageNet under the tight time and computational constraints of this project, I will base my methods on the Inception-v3 architecture.

Besides the previous work done on the ImageNet challenge, CS231-N students have worked on the Tiny ImageNet Challenge before. Some of their results and approaches might serve as inspiration and as a baseline for the current project. Kim [9] reported the best openly accessible results of the 2016 competition. His approach involved using residual networks (inspired by ResNet). He both designed and trained a model from scratch, and applied transfer learning from pre-trained ResNet networks. The best results were obtained from the latter method: his lowest top-1 test error was 31.1%. Zhai [20] also compared training from scratch versus transfer learning from pre-trained models. His architectures were based upon ResNet [7, 6] and VGG [15, 4]. Even though his best reported test accuracy was inferior to Kim's (44.6% top-1 error), he reached the same conclusion: transfer learning produces the best results.

The remaining of the paper will describe my attempts to attain the goal of maximizing the test accuracy on Tiny ImageNet. In Section 2, I briefly discuss the dataset used. Section 3 is dedicated to designing and training my own model from scratch, and presents the relevant results. Section 4 describes my transfer learning attempts and also presents results. Finally, Section 5 contains the conclusion and suggestions for future improvement.

2. Dataset

In the Tiny ImageNet Challenge, the dataset contains square images, of 64x64 pixels. Most of the images have 3 channels for color, RGB, meaning they are 64x64x3 arrays. However, around 1.8% of the examples are grayscale images, i.e. 64x64x1 arrays. For the sake of simplicity, all the grayscale images are immediately converted to RGB by replicating the pixel values across the three channels. Each image belongs to exactly one out of 200 categories. The training set contains 100k images (500 from each category), and the validation and test sets have 10k images each (50 from each category).

Figure 2 shows two examples from six distinct classes, picked from the training set of Tiny ImageNet. Some of the challenges are evident. First, the images are not very highly resolved (only 64x64), which makes details hard to spot. Moreover, while the algorithm is expected to perform some relatively easy, coarse grain classification (for example, distinguishing a cat from a car), it is also expected to perform

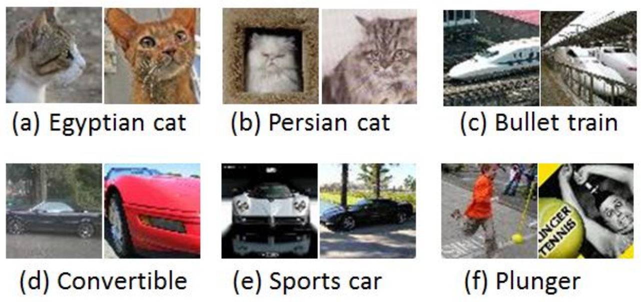


Figure 2. Example images from the training set of Tiny ImageNet

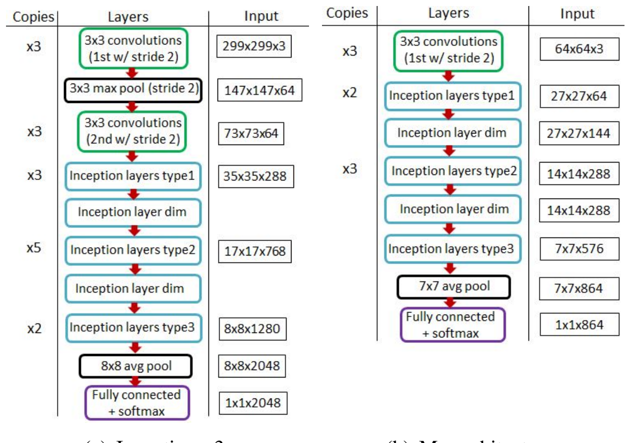
some very fine classification (for example, distinguishing a sports car from a convertible), which even humans (at least yours truly...) would have trouble performing. Since the top-1 accuracy is the objective on the leaderboard, there is no consolation in outputting Persian cat as the predicted class when Egyptian cat is the ground truth label, even if the model predicts Egyptian cat as the second most likely class by a small margin. Since there are only 500 images per class on the training set (compared to over 1000 for the actual ImageNet challenge), I expect models trained from scratch to be more sensitive to overfitting as well. Finally, since the images from Tiny ImageNet are smaller versions of ImageNet images, one can expect them to be noisy. In fact, Hansen [5] shows examples of images from the Tiny ImageNet database in which he identifies "scaling artifacts", "loss of texture", "loss of crucial information due to cropping", and "difficulty of locating small objects" [5]. Thus, even though the competition is on a smaller scale than ImageNet, the classification task is still significantly challenging.

3. Training a Model from Scratch

As explained in Section 1, I have decided to focus my efforts around the inception design, particularly the one used by Szegedy et al. in Inception-v3 [19]. This was chosen not just because of the very high accuracy, but also because of the modest model size and computational cost (crucial due to time and computational constraints).

3.1. Architecture

Figure 3 shows the relevant model architectures. Figure 3(a) represents the architecture used in Inception-v3. Note that they use 6 standard convolutional layers, and then use a total of 12 inception layers, of slightly different designs. The schematic shown is simplified: note, for example, that at training time the model uses a side classifier (not shown), that injects gradients between the last of the inception type-2 layers and the inception dim layer. Also, they use a dropout layer [16] between the output of the last average pooling layer and the fully connected layer. For more



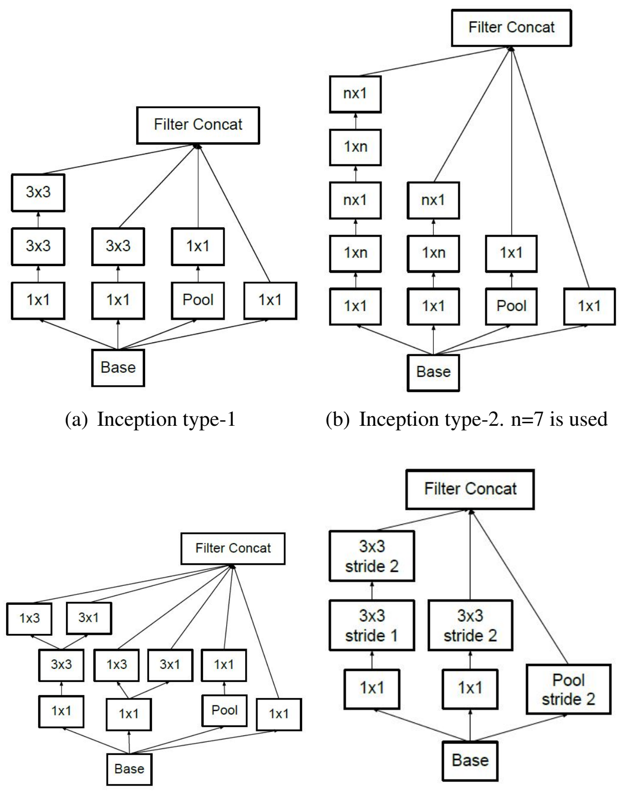
(a) Inception-v3 (b) My architecture

Figure 3. Architectures employed

details on the architecture, the reader is encouraged to consult [19].

Some more needs to discussed about the particular inception layers. Schematics of the distinct layers used are shown in Fig. 4, taken directly from [19]. The inception layers were developed specifically to perform powerful convolutional operations while minimizing the number of parameters and computation required. They use 1x1 convolutions to reduce the number of input channels and then perform different operations in parallel, which are concatenated in the output. Stacking 2 3x3 convolutions, for example, produces the same receptive field as a 5x5 convolution, but uses fewer parameters (a 28% reduction). The same is valid for the asymmetric convolutions: stacking a 1x7 and a 7x1 layer is much more efficient than a single 7x7 filter. Finally, the "inception dim" layer is a layer that uses the inception paradigm to reduce the spatial dimension of the input, while increasing its number of channels. Szegedy et al. argue that this particular design is more efficient than an equivalent (convolution $\rightarrow$ pooling) combination, and avoids the informational bottleneck of an equivalent (pooling $\rightarrow$ convolution) combination [19].

My own architecture was designed based on Inception-v3, and it is shown on Fig. 3(b). It contains the same inception layers as Inception-v3, but it is shallower and narrower: the number of layers and number of filters was reduced when compared to Fig. 3(a). This was done to reduce computational cost and at the same time produce a smaller model with less capacity which would be less prone to overfitting. The number of filters in each step was chosen such that, as a general principle, the number of channels of the activations increased as the depth of the network increased and their spatial dimensions decreased, as suggested by [19]. Also note that my architecture is designed for a differently sized input: the input image to Inception-v3 is 299x299, while the input image to my architecture is 64x64.



(a) Inception type-1 (b) Inception type-2. n=7 is used

(c) Inception type-3 (d) Inception dim

Figure 4. Inception layers used in Inception-v3. m x n refers to convolutional layers with kernels of height m and width n. Concatenations are performed depth-wise between activations of identical spatial dimensions

Unlike in Inception-v3, I did not use a side-classifier, since my network was not as deep. The ReLU activation was used after every convolution due to its simplicity and efficacy [10]. As suggested in [19], I also used Label Smoothing Regularization combined with the cross-entropy softmax loss for training. This means that, to evaluate the training loss which needs to be optimized, a cross-entropy loss was evaluated between the softmax function of the scores generated by my classifier and a *smoothed ground-truth*: for the i-th example, this consists of a probability distribution function given by Eq. 1. Note that the number of classes is $K = 200$, I picked $\epsilon = 0.1$, like suggested by Szegedy et al. [19], and $\delta y_i, j$ is the Kronecker delta, which is 1 if the class j is the correct class for the i-th example, and 0 otherwise.

$$q_{i,j} = \epsilon \times \frac{1}{K} + (1 - \epsilon) \times \delta y_i, j \qquad (1)$$

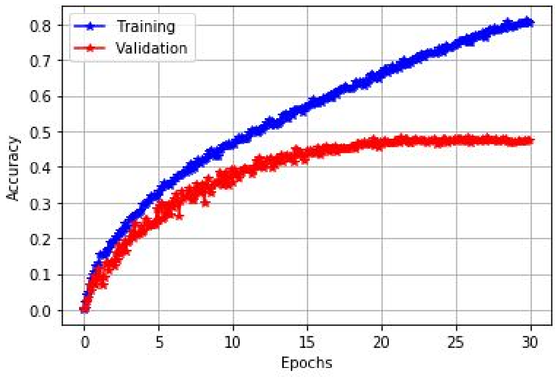
3.2. Training details

The architecture was coded and trained using TensorFlow, Google's deep learning system [1]. Everything was performed on Google Cloud, using 1 NVidia Tesla K80 GPU. Even with my reduced model and GPU acceleration, computational time quickly became a bottleneck for model optimization. For early testing, I sub-sampled the training and validation sets (by a factor of 4) but once I used the full dataset, the training characteristics of the architecture changed (accuracies, losses, speed of training) so I could not use a reduced dataset to reliably tune the architecture and hyperparameters of the full scale problem. Training on a few epochs of the full dataset is also not entirely appropriate, since it was reported (e.g. in [8] and [19]), and I also observed that, some choices of hyperparameters and architectures can yield better results in the first few epochs, but worse results at convergence. Since training of my model took 20-30 min per epoch (and it took anywhere between 20 and 50 epochs for the validation loss to reach a plateau), it was clear that good hyperparameter tuning would be all but impossible within the time constraints of CS-231N.

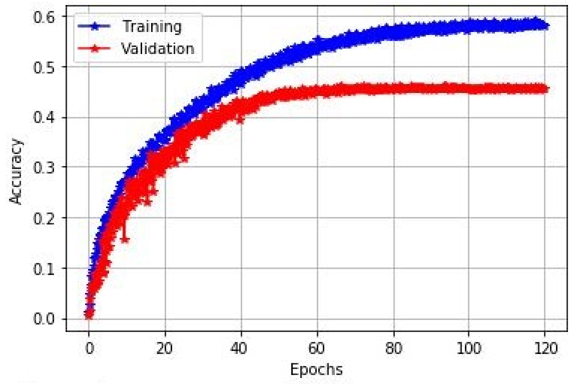
With this in mind, I adopted a few ideas. First, I decided to make aggressive use of batch-normalization [8]. A (spatial) batch normalization layer was added before every single ReLU unit. This is known to accelerate training, and it also makes training less sensitive to initial conditions and hyperparameters picked (for instance, in homework 2 this quarter it was shown that the sensitivity of the final validation accuracy with respect to the learning rate was greatly reduced when batch normalization was employed). As part of the recommendations made by Ioffe et al. when batch normalization is employed, I also decided not to use the dropout layer [8]. For optimization, I used the Adam scheme with default TensorFlow parameters. I tested a few learning rates within a reasonable range for the first few epochs and settled on 1×10^{-3}, decayed by 0.9 at every epoch. With a fixed learning rate, I tried a few different weight decays and settled on 0.08. The hyperparameters are most likely not optimal, but with the limited time I had, this was the best I could do.

3.3. Results

The architecture shown in Fig. 3(b) was trained from scratch as described in the previous section. Figure. 5(a) shows the evolution of a running average of the minibatch training accuracy in blue and the validation accuracy on the full validation set versus epoch. This network will be referred to as "Baseline". The best recorded model achieved 48.5% top-1 validation accuracy. Even though hyperparameters were not properly tuned, a respectable accuracy was achieved given the difficulties of the problem as discussed in Section 2. However, we note that the model is strongly overfitting the training data: at 30 epochs, the training accuracy was already above 80% and steadily increasing, while the validation error had plateaued at just under 50%. This hints that some more regularization techniques might improve the ability of our model to generalize to the validation set.



(a) Initial model (Baseline)



(b) Higher regularization (Regularized)

Figure 5. Training and validation accuracy as model trains.

This lead me to a second attempt, of training the same architecture with extra regularization. Two changes were made: first, I added a dropout layer before the fully connected layer [16], with keep probability of 80%. This should increase generalization, but increase the number of epochs necessary for convergence (and thus increase training time). Second, I added data augmentation at training time. This means that a pre-processing step was added to each batch before performing forward and back-propagation on it: each image was randomly flipped left-right (with probability 50%) and its brightness and contrast were randomly adjusted. The training parameters described before were kept identical, except that the initial learning rate was doubled and it decayed by a factor of 0.9 once every *two* epochs (since we expect it to take more epochs and we don't the learning rate to vanish). This network will be referred to as "Regularized". The results are shown in Fig. 5(b). As expected, the model is not overfitting as severely. But, the highest top-1 validation accuracy achieve was only 46.1%, over 2% lower than before! This was unexpected and disappointing; however, it can be explained by the lack of hyperparameter tuning. If I had months and more computational power, I could have tuned the hyperparameters (like learning rate, learning rate decay, weight decay, and dropout keep probability) to much closer to their optimal value. Then, I expect that the second setup (with more regularization) would have been able to achieve higher top-1 validation accuracy than the first. Note that the regularized model took over 30 hours to train on a single GPU.

3.4. Ensemble

As a final attempt to improve the top-1 validation accuracy, I used ensemble averaging of the predictions of different networks. A simple trick, that requires virtually no extra computation, is to save all the weights of the network during the same training run, at different epochs. During a training run, the program would keep track of the best validation accuracy seen so far. If the new validation accuracy, calculated after some number of training steps, surpassed the best seen so far, then the new weights would be saved on the hard disk. A maximum number of 20 models can be saved at once (thus, when the model improves on the best accuracy seen so far for the 21st time, the first saved file would be overwritten). This also guarantees that the best accuracy ever seen is saved.

At test time, each of the 20 networks is loaded and produces its own score for each validation/test example. The overall ensemble prediction is done by taking the softmax function of the scores, summing them for all the N networks of the ensemble, and predicting the class with the maximum probability. This is shown in Eq. 2. Indices i,j,k represent, respectively, the particular training example, the class, and the network belonging to the ensemble. The variable $scores_{i,j,k}$ is the result from the fully connected layer at the end of the network, and $p_{i,j,k}$ is the probability generated by applying the softmax function to $scores_{i,j,k}$. Finally, $\hat{y}_i$ is the predicted label from the ensemble for a particular training example.

$$\begin{gathered} p_{i,j,k} = \frac{exp(scores_{i,j,k})}{\sum\limits_{j} exp(scores_{i,j,k})}, \\ \hat{y}_i = \underset{j}{\operatorname{argmax}}(\sum_{k} p_{i,j,k}). \end{gathered} \tag{2}$$

I investigated how many networks should be added to the ensemble to maximize the top-1 validation accuracy. Each new network should decrease the variance; but, individually, it will have a lower accuracy than the previous one added, so at some point those two effects balance out. Naturally, the first network that is added is the one with highest validation accuracy; then, the one with the second highest

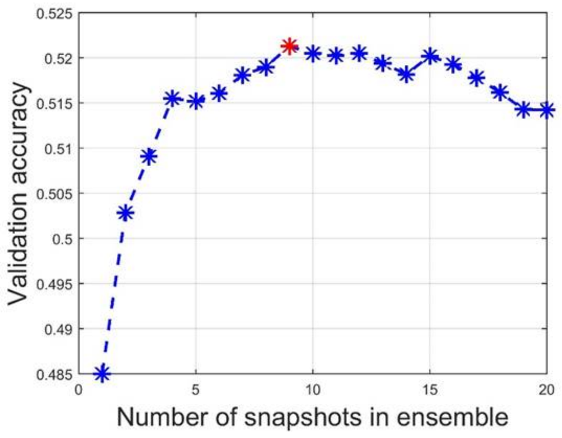


Figure 6. Top-1 Validation accuracy versus number of networks in the ensemble. The point marked in red represents the highest accuracy obtained.

and so on. If the 20th network to be added to the ensemble has much lower accuracy than the first, it might be better to leave it out. Figure 6 shows the results of this analysis for the Baseline network. Note that ensembling allows us to increase the accuracy significantly at no extra cost! Just by saving different snapshots of the model during a single training run, the top-1 validation accuracy can increase from 48.5% to 52.2%. Also, my initial conjecture was right: adding more networks is not always advantageous. In this case, the highest accuracy was obtained from an ensemble of the best 9 networks that were saved. Interestingly, such strong benefits were not observed when I constructed ensembles of the more regularized network: the top-1 validation accuracy only increased from 46.1% (single best network) to 46.8% (best ensemble). I hypothesize that ensembling has a regularizing effect: by averaging different networks, a smoothed out prediction is made that better generalizes to unseen data. Thus, a network that overfits strongly would benefit from ensembling much more than one that doesn't.

Table 1 summarizes the results obtained in this section. Note that only the ensembles were submitted to the evaluation server to determine the test error.

Table 1. Summary of the errors of my own networks

	Top-1 Val. Error	**Top-1 Test Error**
Baseline	51.5%	n/a
Baseline ens.	47.8%	52.3%
Regularized	53.9%	n/a
Regularized ens.	53.2%	57.2%

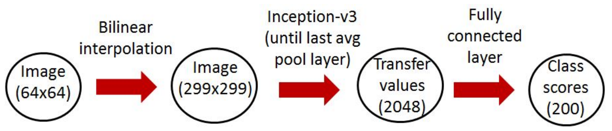


Figure 7. Schematics showing how transfer learning is performed in the context of the present work.

4. Transfer Learning

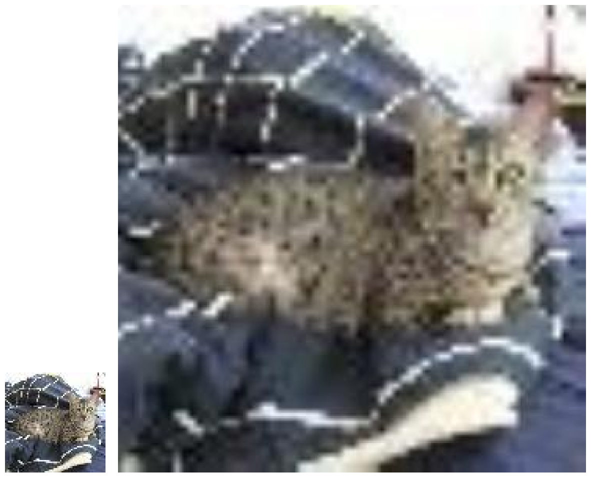
Transfer learning consists of using models that were trained for a certain task and leveraging the knowledge that they acquired on a different, but related task [12]. This can be highly advantageous when there is not much data available for training directly on the related task, or when it is desired to leverage powerful models that would otherwise take several weeks to properly train and cross-validate. Given the computational constraints that were discussed in the previous section and the similarity between Tiny ImageNet (the task at hand) and ImageNet (a dataset in which state-of-the-art models were trained), it is unsurprising that previous CS-231N students found that transfer learning was the technique that yielded the best results for Tiny ImageNet.

In this section, I based a significant part of the code on online tutorials provided by Hvass Laboratories [13]. They also provided utility functions on an open-source GitHub account, which was used in the context of this work [13]. Unless otherwise specified, I used the same dataset, training procedures, hardware, and software packages specified in Sections 2 and 3.

4.1. Methods

For the task at hand, I downloaded the pre-trained Inception-v3 model from TensorFlow.org [1], which was described before and is shown in Fig. 3(a). The last fully connected layer (going from 2048 neurons to 1000 class scores) is removed and the rest of the network serves as a feature extractor: an image is fed in, and a set of 2048 activations is produced (the transfer values). These activations will in turn be used to classify Tiny ImageNet images through a fully connected layer, which will be trained on the present dataset. The flowchart in Fig. 7 shows this process graphically. Note that one difficulty is that the Inception-v3 architecture takes in 299x299 images, while ours are 64x64. There are different approaches for dealing with this. I chose the default behavior of the pre-trained Inception model: the images are re-scaled using bilinear interpolation before being fed into the network. Visually, the images seem smoother and, in fact, less noisy than the originals. This can be seen in Fig. 8.

Note that I kept the pre-trained weights unchanged and did not attempt to finetune them. This allowed training to be much faster. Because of this, I was able to much more effectively tune hyperparameters and experiment with different options, which will be discussed in the following sections. For future work, finetuning the pre-trained weights (after the last fully connected layer is trained) can be a viable strategy to achieve slightly better performance than reported here.



(a) Original (b) Re-scaled

Figure 8. Example of the changes produced by re-scaling with bilinear interpolation. (a) is 64x64 and (b) is 299x299

Again, I used the Adam optimization scheme with default parameters. I tried a few different batch sizes (ranging from 25 to 100) and the final results seemed virtually independent of it (thus, training was always performed with a mini-batch of 100). I also experimented with adding extra layers between the transfer values and the scores (for instance, transfer values $\rightarrow$ fully connected $\rightarrow$ ReLU $\rightarrow$ fully connected $\rightarrow$ scores), but that did not increase validation performance. I hypothesize that this is because the pre-trained parameters were optimized to produce transfer values that can be directed translated into scores by a single fully connected layer; thus, adding complexity should not improve results.

4.2. Results

The first attempt (vanilla transfer learning) just involved using the transfer values taken from 100k training images to train the linear layer with a cross-entropy softmax loss (using the same label smoothing regularization described before). A thorough cross-validation was performed on the learning rate, learning rate decay, and weight decay (i.e., L2 regularization strength) using the validation set, and the final predictions were submitted on the test set. The accuracy values were much higher than obtained in Section 3: in under 5 minutes of training, top-1 validation accuracies

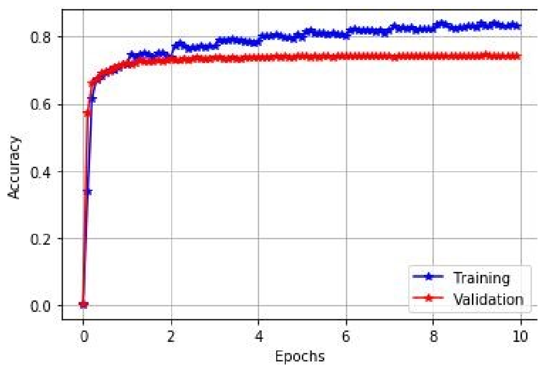


Figure 9. Training curve for vanilla transfer learning.

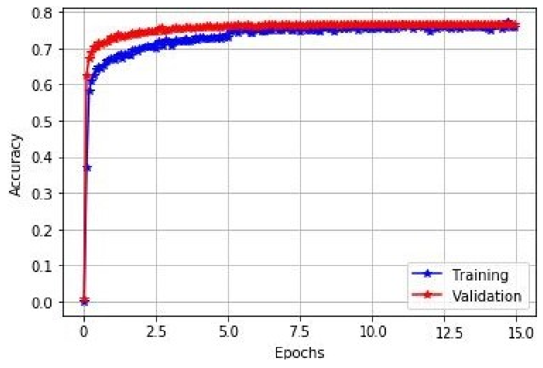


Figure 10. Training curve for transfer learning v3.

over 70% were obtained! Clearly, the features produced by the pre-trained Inception-v3 network were very effective in distinguishing the images. In vanilla transfer learning, the highest accuracy obtained was 74.7% on the validation set and 66.8% on the test set (with initial learning rate of 6.5×10^{-4}, decayed by 0.713 per epoch, and L2 regularization strength of 0.0116). Note that here, taking ensemble of different snapshots during training was ineffective: the validation accuracies were virtually unchanged (oscillating at most one tenth of a percent up or down). Since the model doesn't overfit much, this observation strengthens the hypothesis of Section 3.4. The reported values refer to the model that obtained the highest top-1 validation accuracy (whether or not it is an ensemble). The training curve for vanilla transfer learning is shown in Fig. 9. Note that just training the last layer is much faster (10 epochs is sufficient for convergence). There is also some overfitting, since the training accuracy is consistently a little higher than the validation one.

The Inception-v3 model (just like many of the state-of-the-art models) is trained using data-augmentation. At test time, they are usually fed different crops of the test image, then they average the predictions for each crop, and obtain a single prediction for that image. This is done to average out data augmentation effects introduced at train time and also to allow the algorithm to have a closer look at different regions of the image. According to Kim, this simple test time modification can yield 1-2% better performance [9]. To try to address the modest overfitting observed before, I also added a dropout layer between the transfer values and the fully connected layer with keep probability of 0.8 (same used in Inception-v3). A model using 10-crops at test time plus the dropout layer will be referred to as the Transfer Learning v2. The 10-crops include the original image (64x64), 4 square crops at each of the four corners (56x56), and the left-right flipped versions of these; the predictions from different crops are combined as explained in section 3.4. A new cross-validation was performed and the best hyperparameters for this configuration were used. The best top-1 accuracies obtained were 76.1% and 68.6% on the validation and test set respectively. For brevity, the training curve will not be shown, but it is similar to Fig. 9, albeit with less overfitting and slightly more epochs to convergence.

Finally, inspired by what is done in Inception-v3 and Kim [19, 9], I decided to also use data augmentation during training. This means that, at train time, the images have their brightness and contrast randomly adjusted, and are randomly cropped (either the original 64x64 image is kept, or a corner crop of 56x56 is generated). The training showed that the validation accuracy was higher than the training accuracy! This hints that too much regularization was used, and leads to the conclusion that, with data augmentation, dropout becomes redundant. So, I removed the dropout layer that was added in v2. This new model (10-crop testing, data augmentation for training, no dropout) will be referred to as Transfer Learning v3. The best top-1 accuracies obtained were 76.7% and 69.1% on the validation and test set respectively. The training curve, shown in Fig. 10, shows that I finally obtained an optimal training procedure, with no overfitting. This is my final model, with which I will enter the competition.

Table 2 summarizes the best results obtained by transfer learning techniques. For comparison, the best results from last year and the best result from Section 3 are also displayed.

Table 2. Summary of different model performances.

	Top-1 Val. Error	**Top-1 Test Error**
Best from Kim [9]	24.99%	31.1%
Baseline ens.	47.8%	52.3%
TL Vanilla	25.25%	33.2%
TL v2	23.89%	31.4%
TL v3	23.30%	30.9%

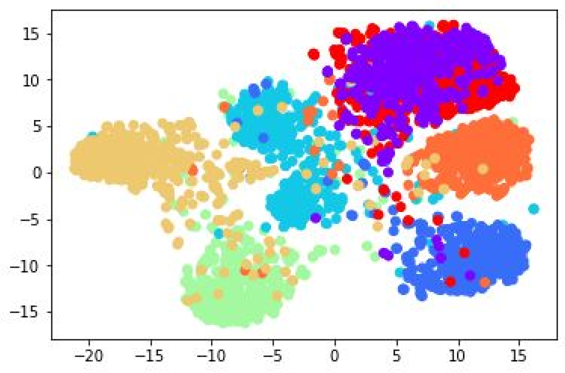


Figure 11. Transfer values after tSNE dimensionality reduction.

4.3. Analysis

In this section, I will interpret the predictions of the transfer learning model to gain insight into its performance. First, one can visualize the transfer values produced by the Inception-v3 pre-trained model. This is done by taking the 2048-dimensional vector and reducing its dimensionality to 2 so it can be plotted in 2D. t-SNE, introduced by Maaten et al., is considered an effective technique to do so [11]. I applied it to all training examples of 7 distinct classes: convertible (purple), sports car (red), plunger (cyan), Egyptian cat (light green), Persian cat (brown), golden retriever (orange), and bullet train (dark blue). The 2D arrays are then plotted in Fig. 11: each example is a dot, and they are colored according to their classes. There are a total of 500 dots of each color.

Note that the transfer values are mostly clustered in different locations according to their classes. For example, the bullet train class, in dark blue, is clustered on the bottom right, well separated from other classes (except for a few outliers). The separation between Egyptian cat and Persian cat (light green from brown) is much better than I expected, which indicates that the algorithm is well-suited to distinguish the two types of cat (probably much better than I am...). The golden retriever class (orange, in the center right) is also pretty far away from the two cat classes. I conjecture that, since the full ImageNet database has so many different types of animals in its training set (and in particular, several breeds of dogs and cats), Inception-v3 became very good at distinguishing them since it was trained with all that data. The only classes whose transfer values have significant overlap are convertible and sports car (purple and red, both clustered in the top right). Some of this clustering might be due to the dimensionality reduction algorithm destroying some of the available information. However, I would expect these two classes to be mistaken somewhat often by my transfer learning methodology.

As another way of interpreting the transfer learning algorithm's limitations, I analyzed the predictions of the transfer learning v3 method on the validation set. Its top-1 error rate was 23.3% on average, but different classes contributed to this figure differently. The class with most mistakes was the plunger: a whooping top-1 56% error rate on the validation set. Looking at examples from Fig. 2, it is clear that plungers are usually small in the picture and show up in many different contexts. This explains why the errors are not focused on a few classes: throughout the 50 validation images where plunger was the ground truth, the model predicted a total of 21 distinct classes (plunger 22 times, and 20 incorrect categories 28 times). This is in contrast to the convertible class: it was misclassified 42% of the time, but only 8 incorrect classes were predicted. The errors focused on the model mistaking one for either a sports car or a beach wagon (14% of the time each). The model was most accurate with the bullet train class: only 1 out of 50 validation images was misclassified. The reason, again, is evident from Fig. 2: bullet trains usually occupy most of the image and have very peculiar visual features. Besides, no class in the dataset is semantically similar to it.

5. Conclusions and Future Work

In the present paper, I described my approach to maximizing the accuracy on the CS-231N Tiny ImageNet Challenge. Section 1 presented the problem statement and described previous work that was performed in this area. In Section 2, the dataset was described, and Section 3 and 4 described two parallel approaches. The best top-1 test error obtained was 30.9%, which put me tied for first place in CS231-N (as of 6/11/2017). Through the present work, I had the opportunity to study state-of-art visual recognition algorithms and try to develop my own, in an attempt to perform a very challenging classification task. I also gained insight into how the ImageNet challenge works and what are the possible techniques one might try to obtain incremental improvements in CNN accuracy.

Future improvement, given time and computational constraints, should focus on perfecting transfer learning techniques. Something that should be attempted is to finetune the whole network instead of keeping it fixed while training just the weights of the last fully connected layer. Another observation from the current work is that the accuracy on the test set seems to be considerably lower than the accuracy on the validation set. This was observed in previous Tiny ImageNet entries (e.g. in Kim's results [9]), but not in ImageNet entries (in most papers, test and validation accuracies are so close that they are discussed interchangeably, e.g. in [10]). If one manages to close this gap, the accuracy on the leaderboard (based on test error) would increase significantly. Finally, using different pre-trained networks (such as Inception-v4 and Inception-ResNet from Szegedy et al. [17]) might also produce marginally better results.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] A. Canziani, A. Paszke, and E. Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *CoRR*, abs/1605.07678, 2016.

[3] K. Chatfield, V. Lempitsky, A. Vedaldi, and A. Zisserman. The devil is in the details: an evaluation of recent feature encoding methods. In *British Machine Vision Conference – BMVC 2011*, 2011.

[4] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *CoRR*, abs/1405.3531, 2014.

[5] L. Hansen. Tiny ImageNet Challenge Submission. *CS-231N Final Project Report*, 2015.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, abs/1502.01852, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[8] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[9] H. Kim. Residual Networks for Tiny ImageNet. *CS-231N Final Project Report*, 2016.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[12] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[13] M. E. H. Pedersen. Hvass Laboratories GitHub and Tutorials. `https://github.com/Hvass-Labs/TensorFlow-Tutorials`. Accessed: 2017-06-10.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[15] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[16] 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*, 15:1929–1958, 2014.

[17] C. Szegedy, S. Ioffe, and V. Vanhoucke. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *CoRR*, abs/1602.07261, 2016.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *CoRR*, abs/1409.4842, 2014.

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *CoRR*, abs/1512.00567, 2015.

[20] A. Zhai. Going Deeper on the Tiny ImageNet Challenge. *CS-231N Final Project Report*, 2016.