Mini-Project 3- Tiny ImageNet Classification, Team: Dirichlet's Brain

Zahra Khambaty
260577706
zahra.khambaty@mail.mcgill.ca

Razvan Ciuca
260675672
razvan.ciuca@mail.mcgill.ca

Robert Fratila
260615193
robert.fratila@mail.mcgill.ca

Abstract

We train and evaluate three different types of classifiers on a subset of images from the imagenet competition. We use logistic regression implemented using the out-the-box method in sklearn, we implement a fully connected feedforward net trained using our own implementation of the back-propagation algorithm, we also use a convolutional neural network comparable to current state-of-the-art models, we find that this performs by far the best on both our validation sets and the test set.

1. Introduction

The project revolves around image classification using the data obtained from ImageNet.The objective is to down-scale the images to manageable proportions in order to accurately classify images into one of the 40 designated classes. Three different approaches are taken: A Logistic Regression Model, fully connected feed-forward neural network and a deep convolutional neural network (CNN) model are implemented to compare and contrast the results with the aim of finding the most robust model with the highest accuracy rate.

2. Related Work

Recent work in image classification involves working with convolutional networks. In Krizhevsky et al's (Krizhevsky, 2012) seminal paper on applying deep convolutional neural networks to the entire ImageNet data set, they found significant improvement in top 5 classification accuracy compared to older models not based on convolutional networks, this paper caused the computer vision community to almost completely shift towards convolutional networks for the tasks of image classification, segmentation, etc. In the past three years there has been an explosion in the number of architectures used for image recognition tasks, the trend is towards networks with many layers stacked on top of each other, (Simonyan, 2015) takes a step in this direction with their VGG network. Next, using the Inception module, (Szegedy, 2014) take a step further in depth using their GoogleNet, finally, using residual connection between layers, (He, 2015) are able to train models with hundreds or even thousands of layers. The very deep models using residual connections are the current state-of-the-art in object classification on the ImageNet Dataset.

3. Problem Representation

3.1. Data Pre-Processing

The test set consists of 6600 images that need to be categorized into 40 classes whereby each of the image is down sampled to be represented by 3 channels of colour (RGB), each with a 64*64 grid of pixel intensities. We convert this multi dimensional array to a single vector of 12288 elements. The labels assigned are binary i.e can be snapped into a 0 or 1 classification. 1 represents being a part of a class while 0 represents otherwise. This is taking a one-vs-all approach for training out predictors and the highest class confidence becomes the prediction of the classifiers. In order to artificially generate more training examples, we will "augment" them via a number of transformations, so that we can limit the amount of times our algorithms sees the same picture twice. This helps prevent over-fitting and assists the model to generalize better and not be thrown off by slight modifications to the same object. Also a rule of thumb for creating robust predictive models, the more the data the better your chances for reaching a better understanding for your problem space.

3.2. Feature Selection

The process of feature selection was implemented using the Python library Scikit-Learn.

1. The images are broken down into 26344 overlapping image tiles.
2. All images are fed into the neural network in batches due to memory limits.
3. Results are saved from each tile into a new array
4. Downsampling to pick out the most representative features from the images (i.e. picking a combination of pixels that are pertinent to recognizing that class)

4. Algorithms and Implementation

4.1. Logistic Regression

Logistic regression is a technique that will serve as a good baseline method for multi-class classification reformatted from the traditional binary classification model.

The inputs of these functions can be represented by pixel intensities ($3 \times 64 \times 64 = 12288$ input neurons) and need to be segmented before they are fed to the functions. This is achieved by segmenting the images to a small region of the input neurons in our case a 55 region, corresponding to 25 input neurons, which is labeled as the local receptive field, basically a small window of pixels whereby each connection carries a weight. Each neuron is backed by an activation function called as sigmoid which is defined by:

$$\sigma(x) \equiv \frac{1}{1 + \exp(-x)}$$

The weights and biases are used to approximate our output and the the goal is to find those that minimize the costs hence we use gradient descent with the theano cross-entropy loss function which outputs a scalar value.

The cross-entropy is positive, and tends toward zero as the neuron gets better at computing the desired output, y for all training inputs, x. These are both properties we'd intuitively expect for a cost function and this is mainly used to avoid learning slow down caused by otherwise quadratic loss function.Note that the larger the error corresponds to faster learning of the neuron.The weights are initialized randomly using shuffle from sklearn.utils. We trained the neural network using mini-batch stochastic gradient descent. While training we noticed that the weights were increasing overtime with all others being constant hence we introduced L2 as the regularization technique This kind of averaging scheme is often found to be a powerful way of reducing overfitting. The output layer contains 40-dimensional vector as the desired output where values of less than 0.5 indicating "input image is not of a particular class", and values greater than 0.5 indicating "input image is of a particular class ". The results obtained from logistic regression were used as a baseline to develop the fully connected neural networks in high hopes of achieving better results.

4.2. Fully Connected Feed-Forward Neural Networks

4.2.1 Introduction

Fully connected feed-forward networks are a class of differentiable function approximators mapping from $\mathbf{R}^m$ to $\mathbf{R}^n$, they alternate between applying fully linear layers and applying element-wise nonlinearity, which is usually either the sigmoid, the hyperbolic tangent, or a rectifier nonlinearity. By the universal approximation theorem of (Hornik, 1989), a 1-hidden layer neural network can approximate arbitrarily well any bounded function on the unit hypercube. However, depth in neural networks can give us exponential gains in data efficiency (Goodfellow, 2017, chapter 5), this makes feedforward nets reasonable candidates for our task.

4.2.2 Problem Setting

In the context of feedforward nets, we completely discard the 2-dimensional structure of the images and simply feed the pixel color intensities to the network as a single list. We interpret the outputs of the network to be the probability of each class given the input image, then we optimize the Cross Entropy function, see section 4.3.2 below for more details.

4.2.3 Optimization

To optimize the cross-entropy loss function, we implement the backpropagation algorithm (Rumelhart, 1986) and use stochastic gradient descent with a minibatch size of 40, the minibatch size is primarily dictated by the memory size and computational power available to us. Whereas we can implement the convolutional network to be discussed in the next section on a graphics processing unit (GPU), we cannot do so with the feedforward net because of the need to hand-code the backpropagation algorithm, GPUs having on the order of 50 times the computational power of CPUs, this severely limits us. We implement early stopping in our optimization routines, i.e. we stop the gradient updates as soon as the evaluation set error begins to worsen, as discussed in (Goodfellow, 2017), this is an effective way to stop overfitting.

4.2.4 Hyper-Parameter Selection

The hyperparameters of the feedforward net are:

1. Learning rate of SGD = 0.02
2. Number of hidden layers = 2
3. Nonlinearity between layers = Tanh
4. Number of nodes in each hidden layer = [1024, 512]

We select the hyperparameters above using cross validation with an evaluation set fraction of 0.3. However, because of computational and time constraints we are unable to exhaustively explore the hyperparameter space, the best we can do is guess a few instances of the hyperparameters and select the ones which yield the best validation set error.

4.3. Convolutional Neural Network

4.3.1 Introduction

Convolutional Neural Networks (ConvNets) are a type of neural networks invented by Lecun (Lecun, 1998) in order to increase the performance of traditional feedforward nets on inputs with a natural two dimensional structure. ConvNets achieve this by employing so-called convolutional layers in the structure of the network, these layers allow us to exploit the 2-dimensional structure of natural images in order to reduce the number of parameters required to correctly approximate a function. ConvNets are *compositionally local* in that they build representations of the data which are made up of local compositions of the lower level representations (Goodfellow 2017, chapter 9). This structure makes them the current state-of-the-art models for natural image processing.

4.3.2 Problem Setting

In the context of object recognition from natural images, ConvNets are a class of function approximators from the space of all RGB images $\mathbf{R}^{3\times m\times m}$ of side length m to $\mathbf{R}^d$, where d is the number of classes to classify. Let F_θ be some ConvNet parametrized by the vector of parameters θ and let $\mathbf{x} \in \mathbf{R}^{3\times m\times m}$ be a candidate image to classify, then we interpret $y = F_\theta(\mathbf{x})$ to represent the vector of probabilities that the image $\mathbf{x}$ belongs to class j, i.e. $y_j = P(class(\mathbf{x}) = j|\mathbf{x})$. This interpretation allows us to use the cross-entropy loss function (Goodfellow, 2017, chapter 2) to cast the function approximation problem as a minimization problem. Formally, let $(\mathbf{x}, j) \in \mathbf{D}$ be an element of the dataset, where $\mathbf{x}$ is the image and j is the class it belongs to, them the function approximation problem is to minimize the following function:

$$L(\theta) = \sum_{(\mathbf{x},j)\in\mathbf{D}} -\log[F_\theta(\mathbf{x})]_j$$

Once the optimal parameters $\tilde{\theta}$ are found, then we define the prediction label to then simply be $\text{argmax}_j\ [F_{\tilde{\theta}}(\mathbf{x})]_j$

The standard structure of modern ConvNets alternates between using convolutional layers with unit stride and $5\times5, 3\times3$ or 1×1 kernel sizes (while zero padding the input image so that the final image stays at the same size), and using pooling operations which progressively decrease the size of the image propagated forward in the network. There exist multiple types of pooling layer types, of which mean pooling and max pooling over 2×2 receptive fields with a stride of 2 are the most popular. The nonlinearity applied between convolutional layers is most often the rectified linear function, i.e. $F(x) = \max(x, 0)$, this has been shown (see Goodfellow, 2016) to yield much better results than either $sigmoid$ or $tanh$ nonlinearities.

After passing the image through a multitude of convolutional layers and pooling operations, it is common to use a fully connected network to map to the final class probability predictions.

4.3.3 Optimization

We optimize the loss function defined above using a variation of stochastic gradient descent named Adam (Kingma, 2015) implemented using the $pytorch.optim.adam$ method, we also experimented with SGD with momentum and AdaGrad, but we found that Adam was the most stable, rigorous testing the methods is quite hard because of our computational constraints, even with many gpus available. Rigorous testing of the different optimization algorithms is highly impractical, instead, we heuristically try different methods and keep the one which seems to yield faster convergence times. Adam essentially computes a noisy estimate of the gradient (minibatch gradient) at each time step and computes some statistic of the past gradients to automatically adjust the learning rate, in studies of optimizer performance on convolutional neural networks, it often performs among the best (Ruder, 2016). We let the number of gradient iterations between halvings of the learning rate as a free parameter. Finally, every 500 iterations of the gradient update, we compute the classification error on the validation set, we then also perform early stopping (Goodfellow, 2016, chapter 6), stopping gradient updates when the validation error begins to worsen, this is a form of regularization, stopping training before the network has time to overfit the training set too much.

4.3.4 Data Augmentation

Our dataset contains approximately 27000 images, relatively "small" modern convolutional networks can contain on the order of 10^5 or 10^6 parameters, this means that overfitting to the training set is a real possibility, the regularization techniques to be discussed in the next section are one way to deal with overfitting, however, one effective way is to augment our dataset with more data (Goodfellow 2017 chapter 8). Besides simply gathering more data, which would possibly be expensive and time-consuming, we simply augment our dataset by adding to it transformations of the data which we believe should not change the class of the item, before transforming the data, we normalize it by making sure that each color channel is in the range $[0, 1]$, here are the transformations we use:

1. X-axis flips
2. Y-axis flips
3. $\pi/2$ rotations
4. adding white noise with $\sigma = 0.01$

Another possible transformation to apply to our dataset is to add random crops of the images to the dataset, we do not do this because our dataset already consists of cropped images from the original ImageNet dataset, further cropping of the images risks removing crucial information.

4.3.5 Dealing with Class imbalance

One large problem with our dataset which we need to deal with is the large class imbalance among the images, even though there are 40 possible classes, 3 of them make up for more than 50% of all the data. This large imbalance risks causing the network to simply learn how to classify a few populous classes, but not the others. We test two main ways to deal with this imbalance, first, we can modify the cross entropy function defined above by weighting rare classes more, i.e. we penalize the network more for making mistakes on rare classes than for making them on common classes, i.e. we can optimize

$$L'(\theta) = \sum_{(\mathbf{x},j)\in\mathbf{D}} -\frac{1}{f_j} \log[F_\theta(\mathbf{x})]_j$$

Where f_j is the fraction of data belonging to class j. By weighting by the inverse frequency, we effectively simulate a dataset with equiprobable classes.

The second way to deal with class imbalance is to change the way we sample data withing each minibatch. Instead of sampling uniformly from the training set, for each element in the minibatch we first sample a class j uniformly from the set of classes $[0, 39]$, then we sample the minibatch element from the images in the selected class. This selection process essentially simulates a sampling from an unbalanced dataset.

We implement both the methods above and test them, we find that the second works better for our purposes.

4.3.6 Regularization

Modern convolutional networks have multiple million free parameters, whereas we only have on the order of 10^5 datapoints, to make sure that we do not over-fit the training set, we test two strong regularizers for neural networks: Dropout (Srivastava, 2014) and Batch Normalization (Ioffe, 2015), we apply these only on the multi-layered perceptrons part of the function.

Dropout randomly drops nodes of the network with probability p during training, effectively preventing the weights of separate nodes from co-adapting, i.e. it forces nodes to be separately useful towards the prediction. Another way of viewing dropout is that it effectively trains an exponential number of separate networks, then averages them, therefore using a mechanism like the ensemble method to strongly reduce the variance of the model. In practice Dropout is very effective at preventing over-fitting, this is especially true in convolutional networks.

Batch Normalization works by computing statistics over the current minibatch and using those to scale the outputs of the intermediate layers of the network, (Ioffe, 2015) argues that it regularises by a similar mechanism to Dropout. BatchNorm has become very popular in the last year, large models such as Residual Networks make heavy use of Normalization layers and works very well with large amounts of data.

After heuristic tests with both methods, we found that Dropout produced better results, we found the networks regularized with BatchNorm to overfit significantly more than those regularized with Dropout.

4.3.7 Hyper Parameter Selection

To select the model structure, the type of class imbalance solution and the regularisation method, we use cross-validation, i.e. we split our data into a training part making up 70% of the dataset and we use the rest for evaluation. We use the Pytorch (Pytorch.org) framework to implement our model. We train the models on 4 Nvidia Tesla K80 GPUs for 8 hours each, however, even this relatively large amount of computational power is not enough for us to satisfactorily search the hyperparameter space, the best we can do is intuitively search the space for good models. We do so by first testing small models to make sure we are not overfitting, from there we slowly test bigger and bigger models until we see the validation set error drop. The final setting for the hyper-parameters is:

1. Adam Learning Rate = 0.001
2. MiniBatch Size = 512
3. Regulariser = Dropout, $P(\text{dropping node}) = 0.5$
4. Way of Dealing with class imbalance = Minibatch Sampling
5. Network structure: See Table 4 in Appendix

5. Results

5.1. Logistic Regression

Given all the specification, multi-class logistic regression produced a cross-validation accuracy of 22.9%.

5.2. Fully Connected Feed-Forward Neural Network

Table 1. Fully-connected network classification accuracy vs. dataset type

dataset type	accuracy
Training	30.6%
Validation	30.4%
Test	30.4%

5.3. Convolutional Neural Network

Table 2. Convolutional Network classification accuracy vs. dataset type

dataset type	accuracy
Training	80.2%
Validation	70.3%
Test	43%

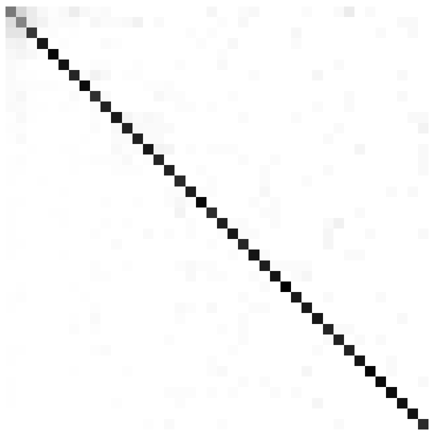

Figure 1. The confusion matrix for the ConvNet, complete black is 1.0 and complete white is 0.0

6. Discussion

The baseline logistic regression model achieved extremely poor results, in fact, simply predicting the mode of the distribution yields better accuracy than our implementation of logistic regression, this is likely because the data is extremely far from being linearly separable, natural images of different object classes do not lie on linearly separable clusters.

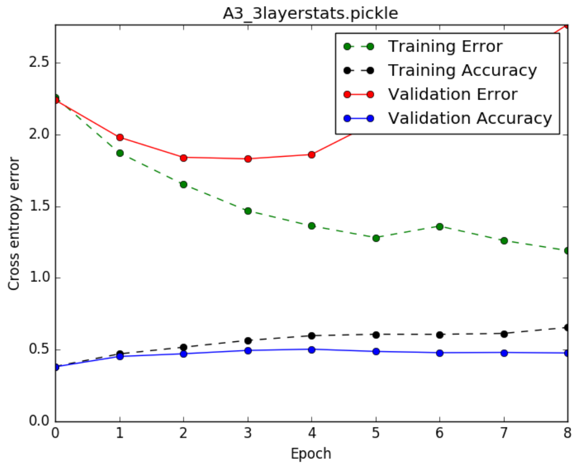

Figure 2. The training and evaluation curves for the Convnet

Our best Feed-forward network simply managed to learn to predict the mode of the distribution, as can be seen in the second table in the appendix, the frequency of class 0 is 30.4%, which is exactly the accuracy rate of our network. This is expected, while fully connected networks can learn nonlinear decision boundaries, they are not well-adapted for inputs with a 2-dimensional structure.

Lastly, our trained convolutional neural network seems to be doing decently well on the training dataset as well as the validation set, however, there is a sharp drop in accuracy when evaluating the accuracy on the hidden test set on the Kaggle website, this was unexpected, one way to make sense of this is to postulate that the distribution of examples in our validation set and the test set are not the same. As can be seen in Figure 1 in this section and table 4 in the appendix, our network manages to get good results at predicting most of the rare classes, however, it is much worse at predicting the most common classes, this is likely what contributes the most to the test set error.

7. Statement of contribution

Razvan Ciuca: I implemented the convolutional network which performed the best on the test set and wrote the sections on the convolutional network and the fully connected network

Robert Fratila: I implemented a version of the convolutional neural network, implemented the logistic regression and wrote parts of the logistic regression section and the fully connected network sections.

Zahra Khambaty: I implemented a version of the convolutional network, wrote the introduction and abstract and the algorithm and implementation part of the logistic regression section.

8. Appendix

Table 3. **Structure of the ConvNet**

type	**# of filters**	**kernel size**	**stride size**
convolution	64	5×5	1
ReLU			
convolution	64	5×5	1
ReLU			
max pooling		2×2	2
convolution	128	3×3	1
ReLU			
convolution	128	3×3	1
ReLU			
max pooling		2×2	2
convolution	128	3×3	1
ReLU			
convolution	128	3×3	1
ReLU			
max pooling		2×2	2
convolution	256	3×3	1
ReLU			
convolution	256	3×3	1
ReLU			
max pooling		2×2	2
type	**# of Nodes**	**Nonlinearity**	**Dropout**
Linear	512	Tanh	0.5
Linear	256	Tanh	0.0
Linear	40	SoftMax	0.0

Table 4. **Recall of the ConvNet vs. Class and Class frequency**

class	**recall**	**class frequency**
0	0.530	0.304
1	0.470	0.152
2	0.774	0.079
3	0.875	0.039
4	0.942	0.030
5	0.921	0.027
6	0.875	0.024
7	0.943	0.022
8	0.756	0.020
9	0.919	0.019
10	0.869	0.017
11	0.836	0.016
12	0.842	0.015
13	0.947	0.014
14	0.841	0.013
15	0.871	0.013
16	0.906	0.012
17	0.848	0.012
18	0.839	0.011
19	0.845	0.011
20	0.850	0.010
21	0.896	0.010
22	0.838	0.009
23	0.887	0.009
24	0.812	0.009
25	0.894	0.008
26	0.969	0.008
27	0.887	0.008
28	0.883	0.008
29	0.948	0.007
30	0.947	0.007
31	0.909	0.007
32	0.907	0.007
33	0.865	0.007
34	0.941	0.006
35	0.920	0.006
36	0.917	0.006
37	0.894	0.006
38	0.891	0.006
39	0.844	0.006

9. References

1. Goodfellow et al., Deep Learning, MIT Press, 2017
2. Srivastava et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014
3. Ioffe et al. , Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015
4. Kingma et al., Adam: A Method For Stochastic Optimization, 2015
5. Ruder, An overview of gradient descent optimization algorithms, 2016
6. LeCun. Gradient-based learning applied to document recognition, 1998
7. Pytorch.org
8. Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks. NIPS.2012
9. Hornik et al. Multilayer feedforward networks are universal approximators, 1989
10. Rumelhart, Learning representations by back-propagating errors, 1986
11. Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks, 2012
12. Simonyan et al. Very deep convolutional neural networks for image recognition, 2015
13. Szegedy et al. Going Deeper with Convolutions, 2014
14. He et al. Deep Residual Learning for Image Recognition, 2015