Neural Networks and Transfer Learning for Image Classification

John Soper

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Data Science

Department of Mathematical Sciences

Central Connecticut State University

New Britain, Connecticut

Dec 2019

Thesis Committee:

Dr Zdravko Markov (Advisor)

Dr Daniel Larose

Dr Roger Bilisoly

Neural Networks and Transfer Learning for Image Classification

John Soper

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Data Science

Department of Mathematical Sciences

Central Connecticut State University

New Britain, Connecticut

Dec 2019

Thesis Committee:

Dr Zdravko Markov (Advisor)

Dr Daniel Larose

Dr Roger Bilisoly

Abstract

This paper examines the effectiveness of Transfer Learning for classification across three datasets. The Keras deep learning library supplies popular CNN architectures, many are image competition winners, which are pretrained with ImageNet weights. Transfer learning with these models had higher classification accuracy with dog breeds and flower species than with fully trained basic CNNs. Image augmentation was found to be helpful or neutral. Fine-tuning could shift performance is either direction. The Inception type models performed better than the others, and an ensemble of them performed best.

A quality difference between two dog breed image datasets reflected in different performances. However, it masked an issue about whether a dataset derived from ImageNet got an unfair advantage in transfer learning. A flower dataset with opposite properties from the dog breeds (i.e. not in ImageNet, few classes, many samples per class), showed the same training trends without surprises.

Table of Contents

Introduction

"The more you get done, the more that is expected from you." Many people would agree with that statement, and if Convolutional Neural Networks could answer, they probably would too. Images are a challenging data source because they are made of thousands or millions of bulk pixels with no inherent representation. Any abstraction of image content to a higher level must be performed by the model itself.

CNNs solve these issues with their sliding convolution windows and successive layers that perform a sequence of edge detection, line resolution, shape recognition, and finally feature detection. They are actually a good example of deep learning, where data is represented at multiple levels.

Image competitions provide an opportunity and incentive for researchers to learn and leverage cutting edge CNN models. Over the last decade, deep learning models have grown to millions of weights with excellent image recognition on standard datasets.

Transfer learning exploits communal features to run pretrained models on different image types. A CNN's fully connected layers at the end are replaced by ones sized for the new class set. The convolution weights are held frozen during the new training but may be fine-tuned towards the end. Some pretrained models will have higher accuracy than others, and an ensemble may outperform them all.

How well transfer learning works also depends on the nature of the data itself, especially how clearly the classes are differentiated and how similar they are to ImageNet. This paper explores these issues with three datasets.

Statement of Purpose

The objective of this thesis is in depth exploration of methods for classifying images with relatively small datasets (< 25k samples). Three datasets will be used, each with their own challenges:

1. Stanford Dog Breeds (fine-grained 120 classes, an ImageNet subset)
2. ImageNet_V2 Dog Breeds (same classes, but independent from ImageNet)
3. Flower Species (5 classes which are not represented in ImageNet)

The first section will discuss neural network theory starting from perceptron nodes. A simple, but comprehensive formulae set including forward- and backpropagation will be presented and analyzed. Deep learning definition, problems and abatement will be covered. A pure python coded neural network will be trained for XOR operation. CNN operation will be covered with analysis of how deep learning on images progresses in the order of edge detection, shape detection, lower-order features, higher-order features, and finishes with a one or more fully connected layer with softmax outputs.

The statistical makeup of the sample sets will be visualized, and any issues discussed. The V2 test data initially looks like it ill-conditioned with examples of both occlusion and conflicting classes (two or more in same image). It will be trimmed as necessary. The images will be used as is, but there will also be an augmented run which will include rotation, width and height shifts, shear and zoom range, and horizontal flipping.

The first procedure is to evaluate semi-advanced CNN trained from scratch. It contains five layers similar to VGG architecture but with just a single convolution per

layer. Due to the small sample sizes, it is expected to perform only moderately and will provide a baseline for more advanced models. Captured metrics will include top-1 and top-5 accuracy (top-2 for flower data).

The second procedure which forms the bulk of this thesis will be experimenting with Transfer Learning on pretrained Keras application models, such as VGG, ResNet, and Inception. These have built-in weights from ImageNet and are quite powerful as is. Five activities are planned:

1. Classifying the 120 dog breeds directly and evaluating the false predictions ratio between non-dog and other-dog classes. Note there are 880 non-dog labels.
2. Simple bottleneck operation when the pretrained model without a top layer is grafted to a single softmax output layer sized to the number of expected classes. This is the most basic form of transfer learning.
3. Cascading two fully connected layers on the output and evaluating possible performance improvement from the ability to train more higher order features.
4. Fine tuning with the last CNN layer also allowed to train. To do this correctly, it must be after Step-2 bottleneck training so the weights in the last CNN layer train slower and correctly.
5. If step 4 results are positive, fine tuning of two or more CNN layers will be performed.
6. A stack of the three best models for possible performance boost.

The results from above will be visualized, analyzed, and discussed, especially in the context of Transfer Learning:

1. Alleviating the problems of small number of samples per class
2. Overfitting to the original ImageNet data
3. Adapting to completely different classes

Once the best single classifier or ensemble is identified, its misclassifications will be analyzed for root cause and possible improvements. There are many issues with fine-grained image detection which is why top-5 matches is a standard metric

After transfer learning, image classification using Mixup will possibly be performed. This is a newer algorithm (Zhang, 2018) that embeds one image within another as a way to generalize and abate the effect of adversarial examples. It is founded on the idea that the features and class of a composite image can be considered a linear interpolation of its two source ones.

The final activity will hopefully be exploration of Contrastive Predictive Coding, an unsupervised algorithm for classifying images by training an image patch to predict another patch of itself. This involves training the neural network to learn a latent representation of the images. Once this is accomplished, a classifier can be formed by labeling a small subset of training images, which technically makes it semi-supervised learning, but different than the standard technique of imputing classes using similarity matrices.

DeepMind wrote the papers on CPC but did not release sample code. Therefore, the initial work will be done with the FashionMNIST dataset. This is 70k images of grayscale data with size 28x28 pixels. If results are successful, the code will be ported to the Flower dataset. Then there will make possible an interesting analysis and discussion of supervised versus unsupervised CNN learning on the same data.

Literature Review (3 parts)

1. General

The corpus of neural network (NN) technical papers is immense. One general overview (LeCun, 2015) discusses their advancement over time. An important benefit of NNs is that a hidden layer of perceptron nodes has the ability for representational learning of raw values. This is an improvement over other machine learning (ML) algorithms which may require complicated data transformations. The paper noted the invention of backpropagation by multiple groups in the 70s and 80s which allows multiple hidden layers and Deep Learning, the representation of data on multiple levels for in-depth processing.

NN popularity declined in the 1990s, then soared starting around 2006 due to larger datasets, more powerful computers, and the fact that high-dimensional cost function curves produce saddle-points instead of local minimums. The paper notes the success of supervised learning implemented with a variety of three major model families: Deep Feed Forward (DFF), Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). CNNs use sliding windows for input data which is large in one or more dimensions (typically 2D image pixels). RNNs generate output from the past output as well as current input and are usually preferred for sequential data such as time series or text. However, CNNs are sometimes used instead due to their versatility and ability for distributed training.

2. Transfer Learning

The large size of recent neural networks makes them quite powerful, but also difficult for the average user to train. A workaround is Transfer Learning (TL) which attaches a fully connected layer (or two) onto a pretrained NN (usually CNN-based) with some or all weights frozen. The new network can perform a more focused task with reduced data and training time.

(Pratt, Mostow, 1991) is one of the earliest papers on the subject. They applied a non-linearly separable dataset to a single hidden layer NN and observed the convergence results. The number of epochs needed with pretraining was about one third of that needed for random weight initialization. However, it would not converge at all about 20% of the time.

A survey paper (Pan, 2009) emphasized the point that TL is a minimal recalibration technique to make outdated training data (IE localized to a previous time or system) perform well on a newer target domain. The authors categorize the various implementations of TL as having three possible settings (inductive, transductive, and supervised), and four transfer contexts (instance, feature-representation, parameter, and relational-knowledge). The paper discusses current research issues concerning different source and target domains (or feature spaces). This reduces performance possibly to the point of negative transfer (worse than no transfer at all).

An additional TL paper (Bengio*, 2012), discussed how deep networks are like the human brain which forms higher-level abstractions as compositions of lower-level ones. And the lower-level ones can be useful for other domains which is what makes

transfer learning feasible. The ideal learned features are abstract and disentangle all the factors of input variations.

(Yosinski, 2014) analyzed NN layers for image data in depth. The first layer learned features similar to Gabor filters and color blobs, and the second layer was also generalized. The paper measured the results implementing TL with frozen weights for layers after this and found two separate effects. Layers 3-5 dropped in performance due to lowered co-adaption (neurons in adjacent layers working together to resolve a feature). Layers 6-7 degradation was dominated by specificity to the original data and lack of generalization. The authors found that fine-tuning abated these issues, in some cases even outperforming a NN trained from scratch.

ImageNet performance was studied in detail (Huh, 2016) and it was found that commonly held beliefs are uncertain. In particular, they still achieved good TL results even when the pre-training used a reduced number of classes (127 vs 1000) or a reduced number of samples per class (500 vs 1000). They admit this phenomenon could at specific to the AlexNet architecture they used or possibly the PASCAL and SUN target datasets having similarities to ImageNet. They conclude by calling for more research to determine just how “data-hungry” CNNs really are.

A Google Brain paper (Kornblith, 2019) focused in depth on TL tradeoffs. They found that model accuracy on ImageNet data correlated to TL performance using a simple logistic regression (bottleneck) output layer. However, improving the ImageNet performance with regularization tended to drop TL accuracy. Implementing fine-tuning improved results and became less sensitive to regularization (but more sensitive to differences between datasets). Finally, fine-grained (many classes) data was examined

for both TL and training from scratch. TL did not have higher accuracy but did have convergence times an order of magnitude less.

3. Unsupervised Learning

(Bengio, 2012) had this to say about unsupervised learning:

> Although we have not focused on it in this Review, we expect unsupervised learning to become far more important in the longer term. Human and animal learning is largely unsupervised: we discover the structure of the world by observing it, not by being told the name of every object.

(Hinton 1986) is the first or nearly first mention of autoencoders, the oldest unsupervised NN model. They use a small hidden layer which act as a bottleneck to force a lower dimensional representation of the data. Their model in the chapter, not yet called an autoencoder, has an N-sized input layer followed by a log2(N)-sized hidden layer which acts as a bit encoder after training.

(Mikolov 2013) discussed a groundbreaking algorithm called Word2Vec for creating word embeddings directly from unstructured text data. It's assignment of terms into 300-dimensional space was so accurate it even allowed vector math such as 'woman' + ('king' - 'man') = 'queen'.

In the next year, Generative Adversarial Nets (Goodfellow 2014) were introduced. This is a very interesting algorithm: two neural networks (a generator and a discriminator) have opposite performance goals. Training is alternated between them with the weights frozen on the other. Each keeps supplying the other with more complex

training data. The result is a GAN which models what the data looks like and can produce new samples on demand, i.e. density estimation.

(Radford, 2016) followed up the previous paper with improvements for unsupervised learning plus training stability. It discusses building GANs using simple convolution layers without max-pooling or fully connected layers. Once trained, parts of these DCGANs are used as feature extractors for supervised work, even to the point of image vector addition, similar to terms in Word2Vec.

Contrastive Predictive Coding (CPC) is a new unsupervised technique introduced in a DeepMind paper (Oord, 2019), and deals with extracting representations from unlabeled data. High dimensional data is encoded into latent space, then autoregression is used to summarize it into a context latent representation. For training, negative sampling is used to differentiate between downstream portions of the same distribution and different ones. The paper discussed the general theory and showed results in four different domains: speech, text, images, and reinforcement learning.

(Henaff, 2019) is a follow-up paper focusing on CPC for image data only. Its technique is to divide an image into overlapping patches each of which is encoded into a feature vector. Then feature vectors from a certain region (like the top half of an image) are aggregated with a context network into context vectors. A row of context vectors is then used to predict features vectors, in an unseen area (i.e. bottom half of the image).

Technical Overview

Neural networks are not new, but their popularity is. Perceptrons capable of the XOR function existed in the 70s, with backpropagation starting about 1975. However, the available datasets were small, and training either failed or was too slow due to limited computer power. SVMs, Linear Classifiers, and Random Forests were more popular, especially in the 1990s and early 2000s. In the words of Yann LeCun: "We were outcast a little bit in the broader machine learning community; we couldn't get our papers published" (Allen, 2015).

Logistic Regression Nodes

Linear algebra is based on mappings which preserve addition and scalar multiplication in a system:

$$F(x_1 + cx_2) = F(x_1) + cF(x_2) \quad (1)$$

A linear regression node has multiple uses but cannot be the basis of a data processing network because any ensemble can only make a larger "linear sandwich" where the output directly scales from the input. In comparison, a logistic regression node runs the linear sum through a nonlinear activation function, typically the logistic function:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2)$$

The term *logistic regression* may seem like an incorrect name for a classifier, but the logistic function is continuous and bounded between 0 and 1. This is an example of a

probabilistic classifier (Jaeger, 2003) where the output represents the probability of class membership. A simple decision rule, *max()*, makes it match other classifiers.

Perceptrons

A perceptron (aka neuron) is the building block of a neural network. It is basically a logistic regression node with a wider variety of activation functions but the same two input types:

- A constant bias value which effectively shifts the threshold point of the activation function left or right
- A weighted linear sum of inputs which is compared to the threshold to determine the pseudo-binary output value.

Single Perceptron Limitations

Even a non-linear activation function cannot meet all possible needs. For example, with the threshold set to 0.5, setting both inputs to a weight of 0.8 will produce the OR function, because either input is greater than the threshold by itself. However, weights of 0.4 will act like an AND function because both must be high to exceed the threshold. Multiple outputs can be created from a layer (column) of perceptrons, each performing AND and OR functions at different weighting. However, there is no way to perform the Exclusive-OR function with a single layer of perceptrons (Minsky, 1969).

Neural Networks

A neural network is a composition of perceptron nodes into multiple layers. The input layer will have one node for each feature, similarly the output layer will have one for each class. There will be one or more hidden layers in between to form intermediate products. With negative weighting acting as logical inversion, an exclusive-OR can be produced (Fig-1). Each node has a threshold of 0.5 and positive output of 1.0. The function of the single node in the hidden layer is to "kill" the output when both inputs are positive.

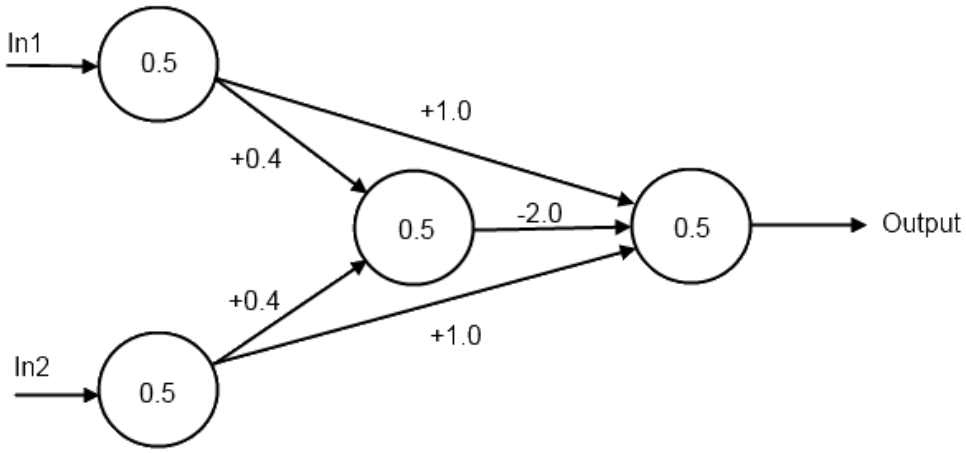


Figure 1: Hidden Layer Providing Exclusive-OR Operation

A classifier model to implement complex functions can be built from this foundation. In addition, neural networks can also perform regression by replacing the output layer activation with a simple identity function pass-through. Nonlinear regression is also possible from the contributions of multiple nodes in the hidden layer.

The creation of a functional neural network involves two interleaved steps, each using a dedicated labeled subset of the full data. Both are run through the model

(forward propagation) to produce outputs which can be compared to the ground-truth labels. The first is training data which determines the correct node weights and biases through convergence. The goal is that the model learns a general solution that also applies to unseen data. However, neural networks are very powerful and can memorize training set noise which is known as overfitting. Another point of view is that the hidden layers store a representation of the training data (Hinton, 2007).

The second step uses validation data to determine the correct hyperparameters (model conditions whose optimal settings are not obvious). Some examples are the number of hidden layers, their sizes, the learning rate, and the momentum value. The model is never trained on validation data, instead it acts as a substitute for unseen data. An *a-versus-b* type comparison is performed to determine which candidate performed better. This can be done in an ad-hoc manner under human control or a grid/random software search of multiple hyperparameter combinations (Claesen, 2015). Since validation performance improves with fitting but drops with overfitting, it is also used to trigger save points of model weights.

There are some variations of the above. Cross-validation is a time-sharing of training and validation data typically done with smaller datasets on other machine learning algorithms. There can also be a test data set which is only run at the end on the final model to determine if it is generalized. The reason for this is that a model can slowly become overfit to validation data when it is constantly adjusted based on it. A typical size balance between train/valid/test data is 60/20/20 percent for 10k points, but the last two can be shrunk as the data size grows because there is sufficient sampling of feature variance.

Neural networks are trained for a specified number of cycles through the entire training set known as epochs. Note: sometimes mini-epochs are used instead to decay the learning rate more frequently. An epoch is divided into segments based on batch size which is the number of samples run before updating weights. Modern computers support vectorization which performs vector operations with similar code statements and execution time as scalars. Since vectorization size is always a power of two, batch size should be also for an exact fit (32 is a good starting point).

Forward and Backpropagation

Training for both classifying and regression uses a cost function which is defined as a measure of difference between the predicted and ground truth distributions. It should be differentiable, always positive, and zero if the two distributions match. The first neural networks were more flexible and some even used a cost function based on genetic algorithms (Larose, 2005). However, backpropagation has superior performance and is now standard. This is also why neural networks use differentiable substitutes (i.e. logistic function for step function and softmax for max).

There are two reasons why differentiability is so critical for training neural nets. The first is gradient descent which minimizes a function by moving in the opposite direction of its gradient. Direct linear algebra solutions are ideal when possible (i.e. Ordinary Least Squares regression and Linear Discriminant Analysis), but gradient descent is a substitute when they are not. Note: when gradient descent is also not possible, hill-climbing (shifting one feature at a time and retaining if improvement) can be used. The second related reason is that backpropagation needs it to compute the cost

function with respect to all weights and biases using the chain rule. The chain rule formula states that the overall derivative for a composition formed with dependent variables is the product of their partial derivatives:

$$\frac{dz}{dx} = \frac{dz}{dy} * \frac{dy}{dx} \tag{3}$$

The complete back propagation algorithm of a NN can be formulated by applying four matrix equations derived from the chain rule. However, these equations are initially daunting and worse still, they are expressed in many different forms and notations across the literature. However, an intuitive version in vectorized form (Neilson, 2015) can be developed from the following schema (below and fig 2):

$\mathrm{Z} -$ weighted sums at a layer

$\sigma -$ nonlinear activation function

$a -$ output activations for a layer

$\delta -$ error vector at point Z

$\odot -$Hadamard (elementwise matrix) product

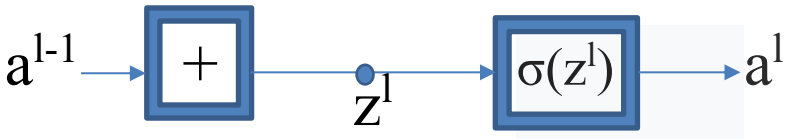


Figure 2: Basic Node Operation

Basic Feedforward NN Equations (for comparison)

Equations (4 & 5) specify forward operation. The output activations of the previous layer are linearly scaled with weights plus biases to form an input vector (z) which passes through a non-linear activation function (σ) to form activations which are the next layer's input.

$$Z^l = w^l a^{l-1} + b^l \tag{4}$$

$$a^l = \sigma(z^l) \tag{5}$$

First Equation: Error in Output Layer (L)

Backpropagation starts on the right at the output layer. The gradient (a vector of partial derivatives) is formed for the cost function with respect to the output activations. Computing the Hadamard product (elementwise matrix multiplication) with the activation function's derivative effectively "steps back" the error to point Z^L in the output layer.

$$\delta^L = \nabla_{\mathrm{a}} C \odot \sigma'(z^L) \tag{6}$$

Second Equation: Error in a Hidden Layer (*l*)

Similarly, repeated applications of the chain rule can step the error term backwards (leftwards) towards the preceding layers. Note: the layer weight column is transposed into a row for proper multiplication with the delta error column.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{7}$$

Third Equation: Derivative of Cost with respect to Bias (*b*)

This equation is quite simple: in a given layer, the previously calculated error term is directly equal to the rate of change of cost relative to bias.

$$\frac{\partial C}{\partial b} = \delta \tag{8}$$

Fourth Equation: Derivative of Cost with respect to Weight (*w*)

The last equation ties it all together. The rate of cost change relative to a weight is the product of the activation with the error right at the weights themselves. This term is scaled by the learning rate (such as $\alpha = 0.01$) to compute a small value to shift the current weight value by. As the cost function approaches a minimum (local or global) in its multi-dimensional manifold, its derivative is reduced which produces a smaller weight shift as desired. Since the activation value is retained from forward propagation, the training process is a combination of chain rule and memoization. The rule of thumb is that backpropagation takes 3x the time of forward prop.

$$\frac{\partial C}{\partial w} = a_{in}\delta_{out} \tag{9}$$

Output Layer Activations and Cost Functions

In gradient descent, it is desirable that the update term shrink as the predictions become more correct. Expressing the delta error at the output layer's Z^L point as the predicted values minus the ground truth values works well. Since this point is reached through a single backpropagation operation, the delta is determined by an interaction between the derivatives for both the cost function and the output activation.

The solution is trivial when regressing because the activation function is an identity pass-through with a derivative of 1. A simple quadratic cost function suffices:

$$C(reg) = \frac{1}{2}(\hat{y} - y)^2 \tag{10}$$

Classification is more involved because the output activation is either the logistic function or its multinomial counterpart known as softmax. Both take the exponent of the Z^L value which produces a sigmoidal curve with flat sides. These areas of almost zero slope create difficulty for backpropagation, but cross-entropy cost functions provide a remedy:

$$C(clf_{multinomial}) = -\sum Ylog(\hat{Y}) \tag{11a}$$

$$C(clf_{binary}) = -ylog(\hat{y}) - (1 - y)\log(1 - \hat{y}) \tag{11b}$$

The multinomial formula (11a) is a sum over all output nodes (all possible classes), while the binary version (11b) is equivalent to having two output nodes that are always in opposite states. Cross-entropy is sometimes called log loss, a slightly different concept, which resolves to the same thing when used in this manner. The negative sign is needed because the log of a number between 0 and 1 will always be negative and a cost function should be positive. The main benefit of these equations is that the log operation inverts the exponential behavior of the activation. In fact, the delta term at Z^L is a simple subtraction just like for regression.

The TensorFlow and Keras deep learning libraries take things a step further and can perform softmax together with cross entropy in a provided API cost function routine. They claim the calculations are better for difficult corner cases. When using these, the output node activation will be set to an identity pass-through just like in regression. The term *logit* is often mentioned in the API documentation, which is the weighted sum (score) at $Z^{L\cdot}$ that softmax turns into a probability.

Overfitting Abatement

As mentioned, neural networks are powerful enough to simply memorize their training data including its noise which hurts performance on unseen data. A large part of NN design focuses on avoiding this. There are three main categories of abatement methods: data, trimming, and tuning.

The simplest fix is to simply obtain more data. Noise is random and gets washed out with large populations. For datasets over a million samples, the validation and test sets can be a smaller percentage which increase the training data still more. For images, data augmentation is a way to effectively increase data. This involves operations such as shifts, flips, rotations, and skews. Some flips are harmful and should not be performed such as horizontal flips for numbers and vertical for horizons.

Trimming, the second category, reduces the possible solution space down to a subset which hopefully retains general solutions and discards overfit ones. Papers based on weight decay (Krogh, 1992) and weight sharing (Ullrich, 2017) have been published but the techniques have not become popular. In comparison, early stopping is supported in deep learning libraries and will save weights whenever a best validation score is reached. This hopefully takes a snapshot of the model at the sweet spot between fitting and overfitting.

Regularization is the most common trimming method and it penalizes the model for choosing a complex solution which usually indicates overfitting. L2 regularization adds the square of the weight value to the cost function of the corresponding node. Under these conditions, the algorithms will choose a more complex solution only when the cost reduction due to gradient descent is greater than the rise from the higher weight.

L1 regularization is less common and simply adds the weight value to the cost function. During gradient descent, a constant value with the sign of the weight is present in the calculation. L1 and L2 have many other names and are seen in other models such as plain (non-NN) regression. They are sometimes cost functions themselves.

The final category is tuning, kind of a catch-all term. Dropout (Srivastava, 2014) made a huge impact and is widely used. It is the random temporary disconnect of connections between layers which forces the NN to learn alternate (more general) pathways. Dropout is only active during training and typically only on hidden layer connections. The size of hidden layers should correspondingly increased (i.e. 20% more nodes for 20% dropout). Also, the activation strengths need to be scaled to equalize the dropout and no-dropout conditions, but deep learning libraries automate this. Dropout roughly corresponds to bagging in random forests and has the same goal: to hurt the overfit more than the fit.

Another tuning method is normalization which is the scaling of values passing through a NN to avoid vanishing gradients, exploding gradients and dead RELUs. Before it was available, a more primitive technique called gradient clipping was sometimes used. A word of caution: *normalization* is one of the most overloaded terms in machine learning and has other meanings depending on context:

1. Scaling a vector into a unit vector
2. Transforming a distribution to be more normal with the Box-Cox or Yeo-Johnson formulae
3. Standardizing data values to Z or min-max scales

Batch normalization (Ioffe, 2015) is the most commonly used flavor and calculates values for each location separately but using every sample in the minibatch. The formulae are similar to z-scores and perform scaling and shifting. Batch sizes should be

as large as possible to minimize differences between them. This is usually the case anyway because of current large GPU sizes, but smaller batch sizes can produce a regularization benefit with noise, so batch size is another machine learning tradeoff. Since RNNs take past outputs as inputs, batch normalization is very difficult and layer normalization is typically preferred. It performs calculations on a single sample at a time but using all values in a given layer. There are other normalizations available that are seldom used. Batch normalization has become very popular similar to dropout and is sometimes preferred over it. The dropout rate should be reduced if both are used together.

The last common tuning method is transfer learning which will later be covered in depth.

Basic Update Equations

Tying together the above information, the basic update formulae for weight and bias updates in gradient descent are:

$$W_{t+1} = W_t - \frac{\eta}{m}\sum (A^{l-1})\ \delta^l + uv_W + \ \eta\lambda W_t \tag{12}$$

$$B_{t+1} = B_t - \frac{k\eta}{m}\sum \delta^l + uv_B \tag{13}$$

Where:

η = learning rate
m = batch size (which sigma sums over)
u = momentum coefficient (0 if not used)
v = current momentum
λ = L2 coefficient (0 if not used)
k = fudge factor (some libraries update bias at a different rate than weights)

Remaining Topics

The current state of deep learning has moved beyond logistic activations and they are typically now seen only in the output node for binary classification. Rectified Linear Units (ReLUs) have become the current standard. They act as an identity function (y=x) for positive inputs and output 0 for negative ones. The right-sided derivative of 1 minimizes the problems of vanishing and exploding gradients. They can still occur, but only through an unlucky concatenation of weights. Since the slope at x=0 is discontinuous, the ReLU function is not differentiable, but this is not a problem. Each half is handled separately which is formally known as subgradient descent.

Other activations and their derivatives are shown in Appendix-B. Tanh is a shifted and sharper version of Logistic, while Leaky ReLU gives dead ReLUs (negative input) a backpropagation pathway to possibly turn back on.

Momentum is a term added to the cost function to fix a geometric issue. A gradient descent point may become stuck in a valley bouncing between walls with slow progress towards the exit. Momentum adds a decaying sum term to the weight update which increases the common component (moving out of the valley) while reducing the differential one (alternating directions towards valley walls).

A variation is Nesterov Accelerated Gradient which adds the momentum vector to the new weights instead of including it in their calculation.

Gradient descent optimizers (Duchi, 2010) were another leap forward in NN design. They allow parameters to have adaptive learning rates instead of a single global value. The most frequently occurring features have smaller updates and infrequent ones get large ones. In addition, the learning rate no longer needs tuning, the default value

works well in most cases. Adagrad was one of the first implementations, it scales the learning rate by a reciprocal root-sum-square term of decaying past gradients. Adadelta is similar but the number of past gradients gets capped at a set window size. Adaptive Moment Estimation (ADAM) (Kingma, 2015) is basically Adadelta with decaying past momentums included. It is good at all-around performer and used in many projects.

Finally, the initial values for weights in deep learning have evolved over time. The mean value is still 0.0 in all cases, but different variances can be used based on the activation function and both fan-in (N_i) and fan-out (N_o) with bordering layers (Eq 12a-d). Curiously, the Keras documentation recommends a truncated normal initialization with a stddev of 0.5.

$$Var(primitive) = 0.1 \tag{14a}$$

$$Var(LeCun) = \frac{1}{N_i} \tag{14b}$$

$$Var(He) = \frac{2}{N_i} \tag{14c}$$

$$Var(Xavier\ aka\ Glorot) = \frac{2}{N_i + N_o} \tag{14d}$$

Debugging Tip

When designing a neural network classifier, it helps to first verify basic operation with a tiny data subset such as 32 samples (16 samples each of two classes). It is used for both training and validation with a batch size also 32. The model should easily memorize the data and overfit to an accuracy of 100% while the cross-entropy cost function drops from nearly 1 to 0. This will not catch every issue but is a quick way to detect the "low-hanging fruit".

Simple Python Neural Network

As a demonstration, a primitive NN was coded up in Python using Numpy arrays and trained for the XOR function (code in Appendix-B). For simplicity, there were no biases, a simple passthrough was used for the output activation, and the cost function was quadratic. The training converged (Fig 3) and the output is high only for inputs (0, 1) and (1,0) which is proper XOR operation function.

```
 Epoch     Loss      Y(0,0)     Y(0,1)     Y(1,0)     Y(1,1)
 ------------------------------------------------------------
    0    4.47936     0.0000     0.2773     3.7598     0.9054
  100    0.54947     0.0000     0.3644     1.8336     0.0000
  200    0.21144     0.0000     0.4612     1.3640     0.0000
  300    0.11149     0.0000     0.5610     1.1738     0.0000
  400    0.06285     0.0000     0.6561     1.0860     0.0000
  500    0.03471     0.0000     0.7401     1.0432     0.0000
  600    0.01837     0.0000     0.8096     1.0219     0.0000
  700    0.00932     0.0000     0.8639     1.0111     0.0000
  800    0.00457     0.0000     0.9046     1.0057     0.0000
  900    0.00218     0.0000     0.9340     1.0029     0.0000
 1000    0.00102     0.0000     0.9548     1.0015     0.0000
 1100    0.00047     0.0000     0.9693     1.0008     0.0000
 1200    0.00022     0.0000     0.9792     1.0004     0.0000
 1300    0.00010     0.0000     0.9860     1.0002     0.0000
 1400    0.00004     0.0000     0.9906     1.0001     0.0000
 1500    0.00002     0.0000     0.9937     1.0001     0.0000
 1600    0.00001     0.0000     0.9957     1.0000     0.0000
 1700    0.00000     0.0000     0.9971     1.0000     0.0000
 1800    0.00000     0.0000     0.9981     1.0000     0.0000
 1900    0.00000     0.0000     0.9987     1.0000     0.0000
 2000    0.00000     0.0000     0.9991     1.0000     0.0000

final h_weights:
 [[-0.41675785 -0.05626683 -2.1361961   0.57535004]
 [-1.79343559 -0.84174737  0.98685426 -1.29664876]]

final y_weights:
 [[-1.05795222]
 [-0.90900761]
 [ 1.0124547 ]
 [ 1.7380754 ]]
```

Figure 3: Metrics of Python Neural Network Training

Convolutional Neural Networks

CNNs are designed for data which is large in one or more dimensions, typically images. When convolving a grayscale image, a small 2d matrix known as a filter (aka feature detector) traverses the image as a sliding window, computing dot products at each step, which builds up a feature map. Multiple filters in parallel can be trained to detect different features and produce a set of feature maps. Each feature map can also be pooled (aka subsampled) into a smaller one. Therefore, data traveling through a CNN tends to shrink in two dimensions, known as size and grow in the third, known as depth. The final layer is fully connected and produces softmax class predictions in the usual way (Fig-4).

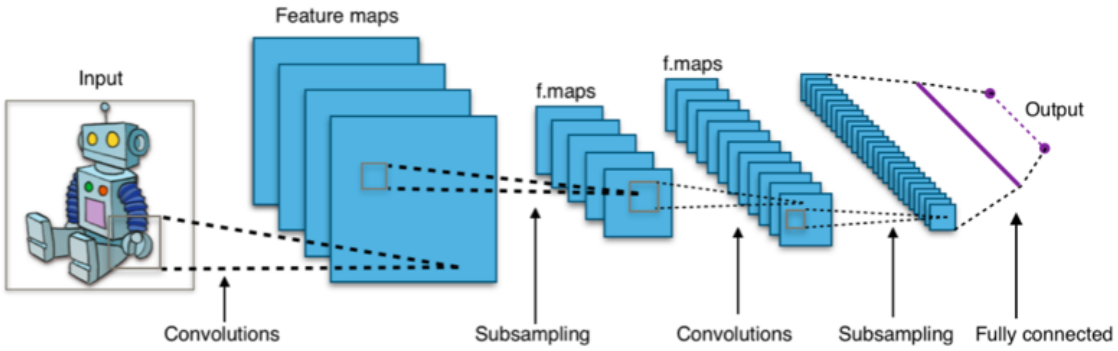


Figure 4: Basic CNN Structure (by Aphex34 - Own work, CC BY-SA 4.0), https://commons.wikimedia.org/w/index.php?curid=45679374

The layers in a CNN learn to represent more complex features as the input is propagated through. For example, the first layer performs edge detection, the second shapes, the third objects, and the fourth eyes.

CNNs have a couple of big advantages which makes them a popular workhorse in deep learning. The first is that the sliding windows can process larger dimensional data than a fully connected layer could handle. A basic example is a 1000x1000 image that would require a million FC input nodes. Secondly, the adjacent filters and their output

feature maps are independent so they can be processed in parallel on multiple cores, a feature which RNNs sorely lack. Simply put, there is nothing better than CNNs for image processing.

Once CNNs are understood at a higher level, several complex aspects remain. Firstly, the filter x- and y-axis sizes (aka receptive field) are hyper-parameters, but the depth must match the preceding volume (a set of either feature maps or pooling layers). A simple example is a color image which has a z-axis depth such as 3 (RGB) or 4 (CMYK) which is therefore also the depth of the first filter(s). The generated feature map is still 2D, but an output volume of multiple ones can be produced by multiple filters in parallel (another hyperparameter, sometimes denoted as *K*).

The filter interacts with several size settings due to its sliding window nature. The number of pixels it steps across is called stride. The image may be edge-padded with zeros to preserve the input size or to ensure the edge pixels interact with the center of the filter matrix. The formula for output size along the x or y axis is:

$$O = \frac{I - W + 2P}{S} + 1 \tag{15}$$

where:

O = output feature map size
I = input feature map size
W = filter size
P = padding per side
S = stride

If the stride is one and the padding set to P=(W-1)/2, the x and y dimensions stay constant. However, a stride of two or more will reduce dimensions. Often the filter size is odd to process a single pixel with respect to its neighbors. An example of this is edge detection which is the first layer when processing images. From fig-5 below, the

convolution dot product is set to produce large values when the center pixel is a different brightness from its surroundings.

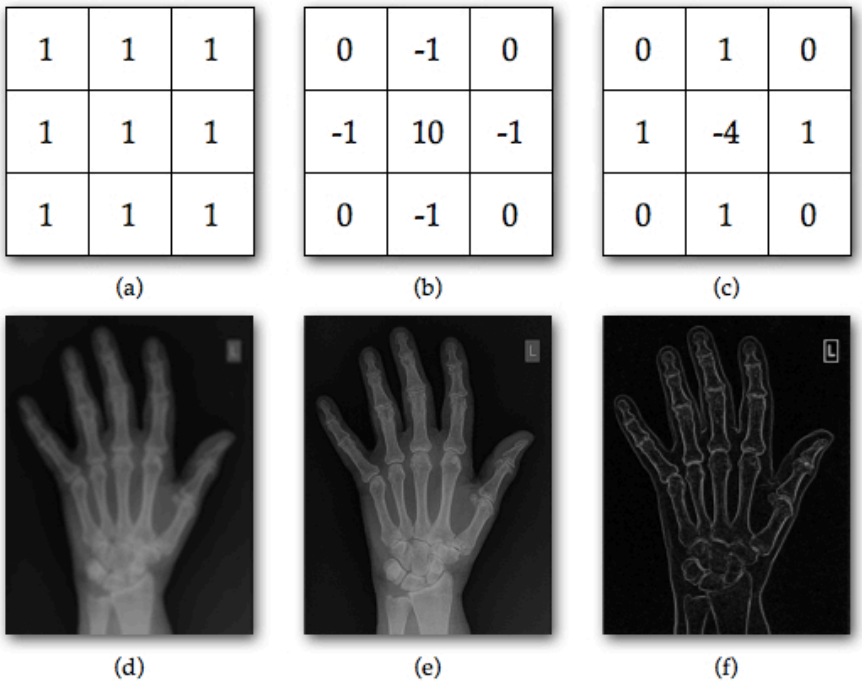


Figure 5: Edge Detection Filters (By Kieranmaher - Own work, Public Domain) https://commons.wikimedia.org/w/index.php?curid=13305900

The pooling layer is another confusing part of CNNs because it has its own filters and stride. They are usually set identically, so that each input value is processed only once. A *max* operation is standard (fig 6), but sometimes averaging is chosen instead. Not every convolution layer has a pooling layer after it and some people advocate against them (Springenberg, 2015). Therefore, a CNN has two ways to reduce dimensions: convolving with a 2+ stride and pooling. Both are optional, but a design without either one would be nonsensical.

The last aspect of CNNs to understand is that they are still neural networks with weights, training, and backpropagation, just without full connections. The CNN filter is

not a literal sliding window, that is just an abstraction to understand the process. Instead there is a stack of filters with identical weights, one for each possible convolution position, and each filter connects to one position in the previous feature map. Adjacent feature maps in the next layer will have their own filter stacks with the same dimensions but different weights feeding them.

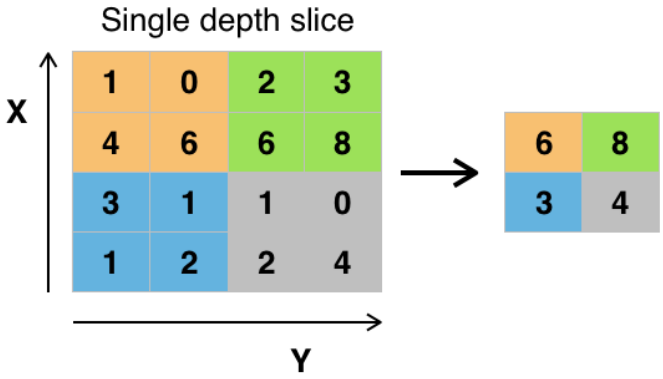


Figure 6: Simple Max-Pooling Example (By Aphex34 - Own work, CC BY-SA 4.0) https://commons.wikimedia.org/w/index.php?curid=45673581

Backpropagation is performed by rotating the filter 180 degrees and performing convolution from back to front. Filters and fully connected nodes are trained while pools and ReLUs (if used) are constant. As in other machine language areas, the CNN will focus on useful properties. If a CNN is trained to distinguish colors, it will carry the color information through all successive feature maps. However, for character recognition training, it will basically convert the image to a grayscale equivalent internally since a red *B* and a blue *B* are the same class.

To solidify some of these concepts, a numeric example for first layer of the 2012 ImageNet Challenge winner (Krizhevsky, 2012) is presented:

- Image size = 224 * 224 * 3 (color) layers
- Filter size 11 x 11 x 3 with stride 4 and double-sided padding of 3
- Output dimension = (224 -11 + 3) / 4 + 1 = 55
- K = 96 feature maps
- Output volume = 55 x 55 x 96
- Unique weights per filter (or per output node) = 11 x 11 x 3 = 363
- Total unique weights = 11 x 11 x 3 x 96 = 34848
- Unique biases = 96

Data Description and Preparation

Flower Dataset

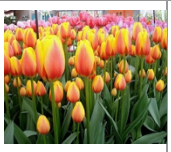
Three image sets were used for this thesis (links in Appendix C). The first is a Kaggle dataset of 4328 images over five flower species. The files are in jpeg format with various sizes from about 150 to 300 (sometimes up to 600) pixels per axis. About 80% are landscape format the rest square or portrait. They were split into 70% test, 30% validation, stratified for class (Table-1).

Flower Data Analysis

The images are well lit and almost always in focus. Some photos show additional content such as a vase or human, but not occluding the flower, except for an occasional insect in closeups. The images are a mix of single and multiple flowers. Most flowers are in bloom, some are buds or budding. Additionally, many of the dandelion images are seed pods (what children blow on). They appear different enough to be a sixth class which may explain why dandelion images have the largest representation (1053 images). Less than one percent of the samples look like an adversarial example, but some of them are very unfair.

A person must think in terms of shape and color to predict how a CNN might train on an image set. With flowers, the shape of the top parts (petals and pistils/stamens) will be the main differentiator between classes. For both top-down and level views, this will be a circular or oval shape, so the CNN layers after edge detection should train to recognize it.

Table 1: Flower Dataset

Species	Train samples	Valid. samples	Total Samples	Example 1	Example 2	Adversarial Example
Daisy	538	232	770			
Dandelion	736	317	1053			
Rose	549	236	785			
Sunflower	514	221	735			
Tulip	689	296	985			
Total	3026	1302	4328			

Color is a mixed bag and the occasional black and white images do not help. Daisies (white petals around yellow center) and sunflowers (yellow petals around orange-brown center) will probably benefit the most from color information. Dandelions have two modes (solid yellow petals and white seed pods) but at least those colorings are constant. Roses and Tulips are known for having many colors however, and the CNN

will have to focus more on shapes. Green from the surrounding vegetation and blue horizons from sky are worthless for class differentiation and will probably not be learned.

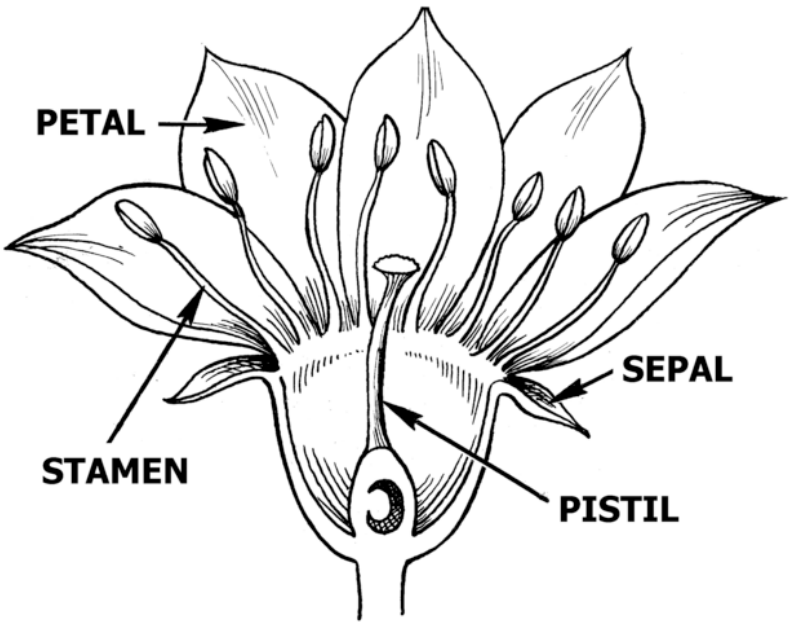


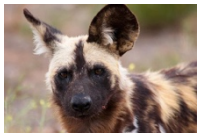
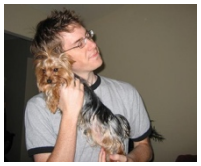
Figure 7: Flower Anatomy (Public domain)
https://commons.wikimedia.org/wiki/File:Flower_Anatomy_(PSF).png

Stanford Dog Breed Dataset

This set is composed of 20,580 images across 120 dog breed classes and is intended for fine-grained classification. This a branch of object recognition where the different classes share generic features and the model must be trained for additional aspects. The files are in jpeg format with slightly over two thirds in landscape mode. The average number of pixels per axis is higher than the Flower data usually in the 300-500 range.

The number of samples per class ranges from 148 for Redbone to 252 for Maltese_dog (full breakdown in Appendix D). For this study, the data was split for a constant 30 validation and 20 test samples per class. Therefore, the number of training samples for a class is 50 less than the total samples. Five breeds were sampled by taking the first and last ones alphabetically (Table-2). Differentiating between the two terrier breeds is an example of challenges the models will face.

Table 2: Stanford Breeds (PARTIAL)

Species	Total Samples	Example 1	Example 2	Adversarial Example
Affenpinscher	150			
Afghan Hound	239			
African Hunting Dog	169			
Wire-haired Fox Terrier	157			
Yorkshire Terrier	164			

Stanford Dog Breed Data Analysis

The images are similar to the Flower dataset in terms of being in focus, good lighting and hardly any occlusions. Most photos are of a single dog, a few have two or more of the same breed, and one adversarial example with two breeds was noticed. The settings were more varied than the flowers and included indoor scenes and even dog shows. There are more images with humans present, but still relatively few. Dog colors are subdued and will a less important feature than the bright hues of flowers.

The American Kennel Society (AKC) recognizes seven groups of dog breeds: Terrier, Toy, Working, Sporting, Hound, Non-Sporting, and Herder. Some of their characteristics will be learned by a classifier. Hounds tend to have pointed ears to hear prey and long legs to chase it down. Toy dogs have curled tails and relatively large eyes which reflect their status as cute indoor pets. Breeds known for fighting usually have large necks and full muscles for protection. Dog breeds are different sizes as well (from small toy breeds to large hounds) but photos have no sizing scale reference to supply the neural network.

There are two big drawbacks to classifying this dataset balanced by one large advantage. The first issue will be all the possible dog orientations. Some photos are frontal, some sideview, but most are in between. There are also varying degrees of downward angle because of cameras being at the level of human eyes. The neural network will have to detect the dog's head from different perspectives with its body supplying additional features.

The second problem is that fine-grained classification of 120 different targets is a lot to expect from a model. Dog breeds make this harder still because some have very

similar appearances such as Malamute & Siberian Huskey or Italian Greyhound and Whippet (Fig-8). For this reason, papers about image classification also report if the correct class is in the top five predictions. One interesting piece of trivia is that dalmatians were left out of the dataset. Perhaps the spots made them too distinctive.

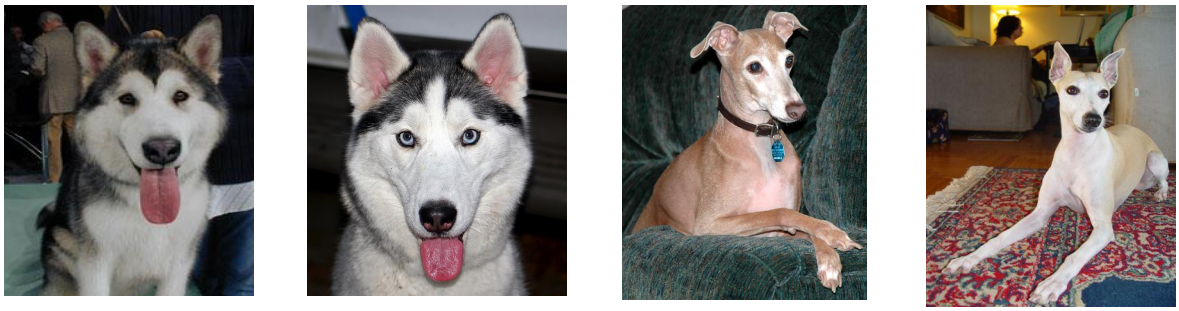


Figure 8(a-d): Similar dog breeds (Malamute & Huskey, Italian Greyhound & Whippet)

An important fact about the Stanford Dog Breed dataset is that it is a subset of ILSVRC (ImageNet Large Scale Visual Recognition Challenge), a thousand class dataset of 32,326 images, which itself is a subset of the main ImageNet (14 million images over 20 thousand classes). The ILSVRC dataset (aka ImageNet-1000) has become a standard for papers, competitions, and pretrained CNN models in deep learning libraries. Therefore, Keras models already support the 120 Stanford dog classes out of the box. However, they will have the drawback of sometimes predicting one of the 880 non-dog classes. This condition will be studied and remedied with transfer learning.

ImageNetV2 Dog Breed Dataset

In addition to the Stanford dog breed dataset, CIFAR-10 and CIFAR-100 are also subsets of ImageNet. Since it is so pervasive in industry, there is concern that some

results over the years were adaptively overfit to it without knowing. A group out of UC Berkeley developed ImageNetV2 (aka V2), an independent dataset with a matching class structure, and published results (Roelofs, 2019):

> We evaluate a broad range of models and find accuracy drops of 3% – 15% on CIFAR-10 and 11% – 14% on ImageNet. However, accuracy gains on the original test sets translate to larger gains on the new test sets. Our results suggest that the accuracy drops are not caused by adaptivity, but by the models' inability to generalize to slightly "harder" images than those found in the original test sets.

This thesis will use the dog breed images in V2, but just as an additional test set because there are only ten images per class. The data seems similar to the Stanford dog data but does seem to have more adversarial examples (Fig-9), and an annoying number of images with two dogs either different species or at least different colors.

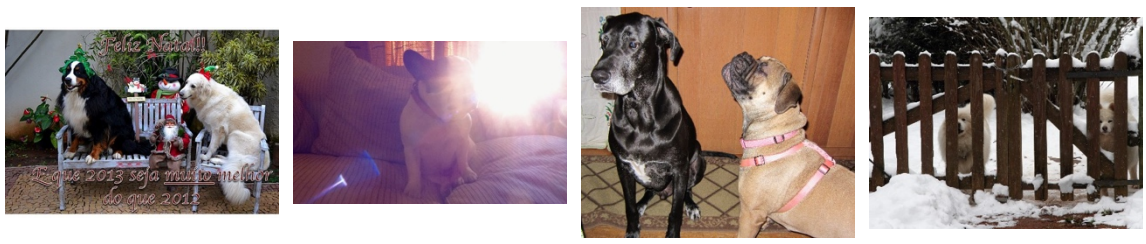


Figure 9(a-d): V2 Adversarial and Multiple Dog Examples

Image Augmentation

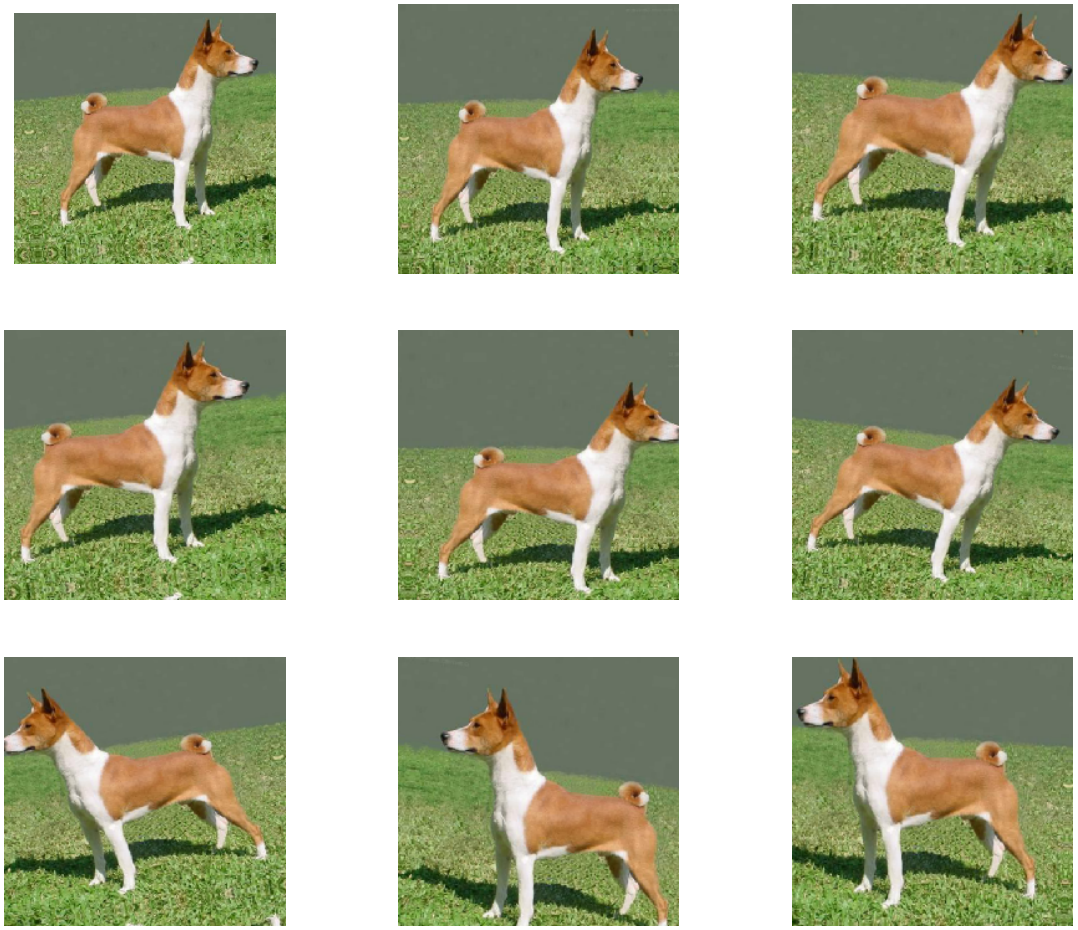
The ImageDataGenerator class in the Keras library provides the ability to augment images which hopefully prevents overfit and improves the validation results.

Performance will be evaluated on original images and with the following augmented settings:

- Horizontal flip = True -- Dogs will sometimes flip between left and right
- Vertical flip = False -- Upside down dogs do not provide value
- Width shift range = 0.1 -- Shift left or right up to 10%
- Height shift range = 0.1 -- Shift up or down up to 10%
- Rotation range = 10.0 -- Rotate up to 10 degrees in either direction
- Shear range = 0.05 – 5 degrees shear angle in counter-clockwise direction
- Zoom range = 0.1 – Random zoom between 90 and 110%
- Fill mode = reflect – Fill points outside input boundaries in an *abcddcba* pattern

The flips, shifts, rotations and zooms are evident in the nine examples below (fig-10).

Figure 10: Augmentation Demo



Basic CNN Image Classification Experiments

To establish a baseline for the pretrained models, a moderate size CNN model was trained from scratch. It performs five convolutions all with a 3x3 filter and stride of 1. After each is a 2x2 max pooling layer with a stride of 2 to cut the x and y dimensions in half each time. The last volume is flattened and passed to a batch-normalized dense layer with either 5 or 120 nodes for each possible class. (Fig-11). This is a stripped-down version of the VGG19 architecture which also has five pooling layers but performs multiple convolutions between them.

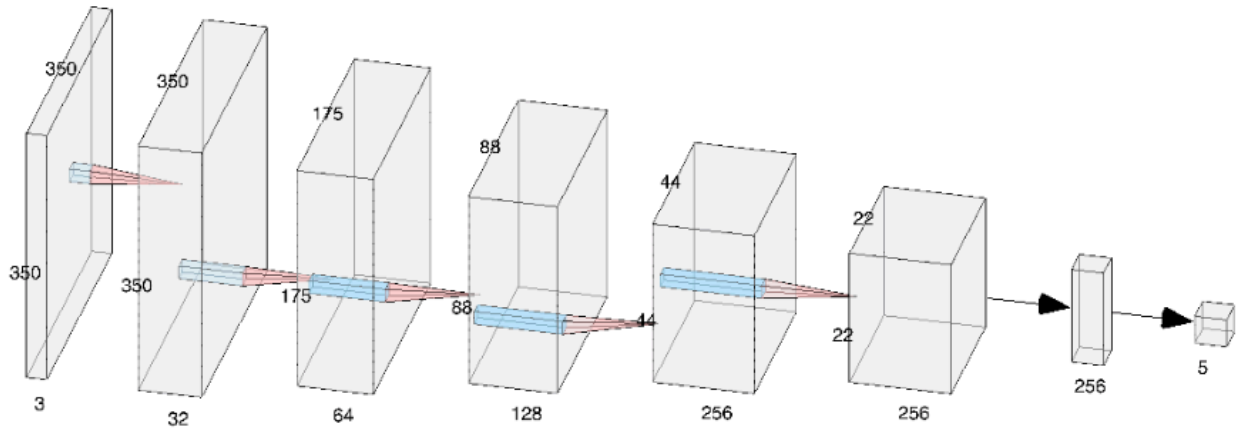


Figure 11: Data flow through basic CNN

Flower Data

The Flower data was trained for 100 epochs with both direct and augmented data. Epoch runs times were about 30 and 90 seconds respectively, so augmentation roughly triples the training time. The learning curves for both are plotted in Fig-12. For training data, the direct case shows an elbow at nearly 0 indicating overfitting while the

augmented case gradually decreases due to shifting training data. The validation cost function is very volatile for both, but the direct data shows a net rising trend, another indicator of overfitting which is much smaller in the augmented data.

Fig 12a: Basic CNN Learning Curve, no augmentation

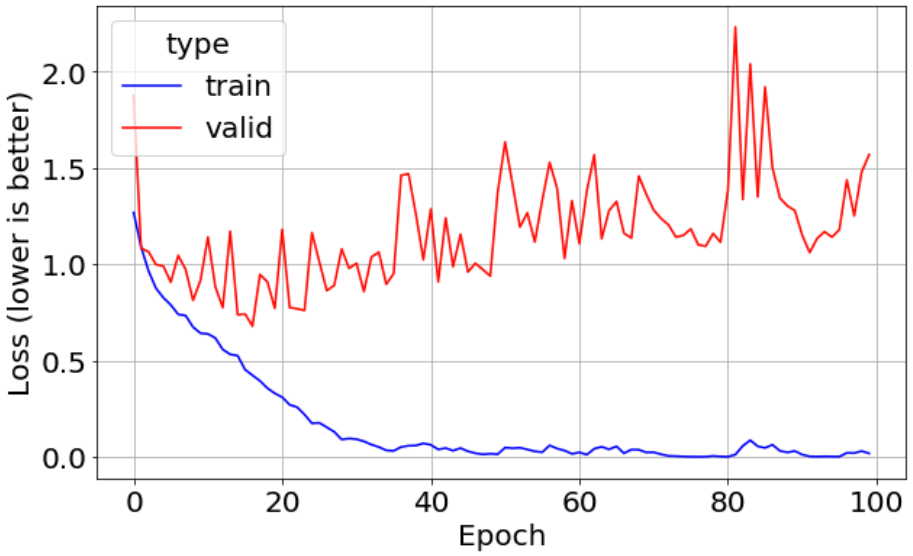
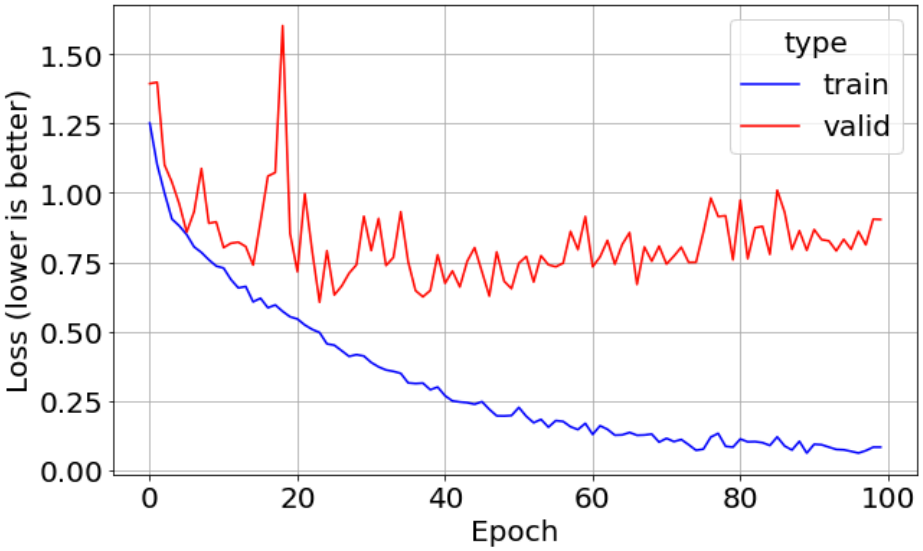


Fig 12b: Basic CNN Learning Curve, augmented



The accuracies of the top and top-2 predictions were also plotted. Augmented validation (fig-14) was about 79% which was higher and more steady than direct at approximately 74% (fig-13). The training result for direct shows a knee corresponding to the elbow in the learning curve. The top-2 results show the same trends as the top ones.

Fig 13: Basic CNN Model Accuracy, no augmentation

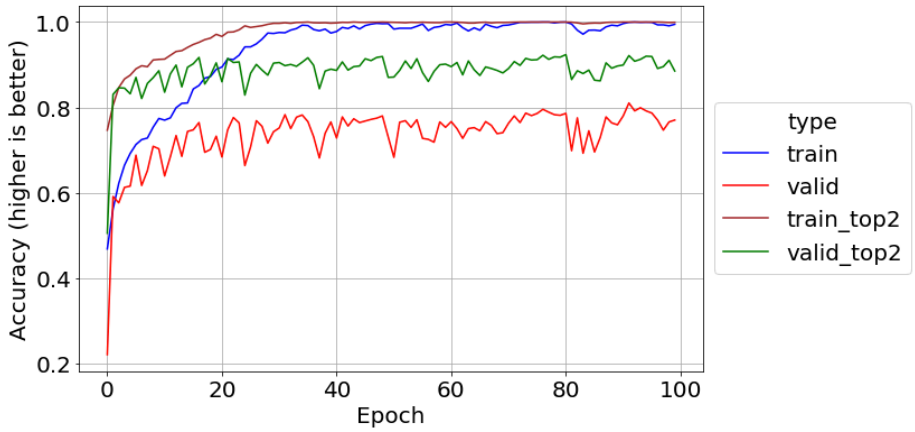
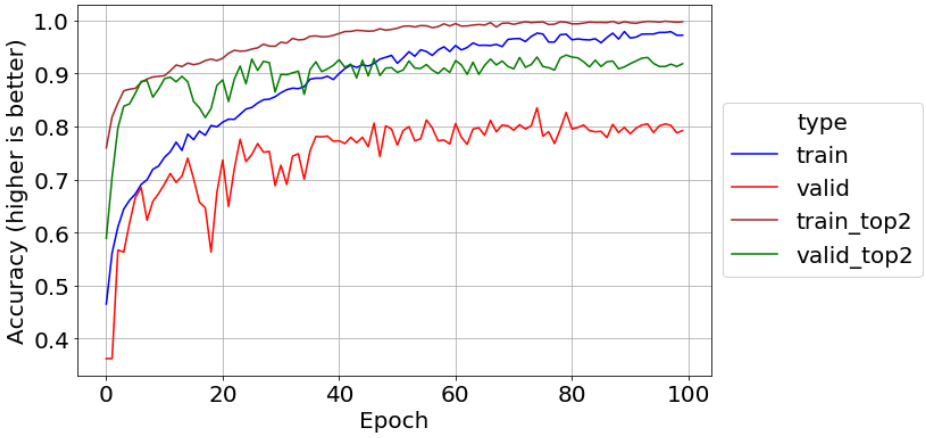


Fig 14: Basic CNN Model Accuracy, augmented



The fact that the training data converges to 1 or nearly 1 is evidence that this five-layer convolution design is powerful enough to memorize the 3026 training samples. If more powerful architectures were not available, the next task would be seeing if stronger regulation moved the train and validation curves closer together.

A second dense layer of 256 nodes was added, but augmented performance stayed the same, just under 80% validation accuracy.

Stanford Dog Breed Data

The previous activities were repeated for the Stanford data which is much larger, but has 115 more classes and less samples per class, about 180 vs about 600. Training time was almost 3 minutes per epoch without augmentation, 8 minutes with (the same 3x ratio as the flower data). The learning curves (fig-15) showed the same trend, a rising validation value for direct, with a much slower rise for augmented images.

Fig 15a: CNN Learning Curve, (dogs, direct)

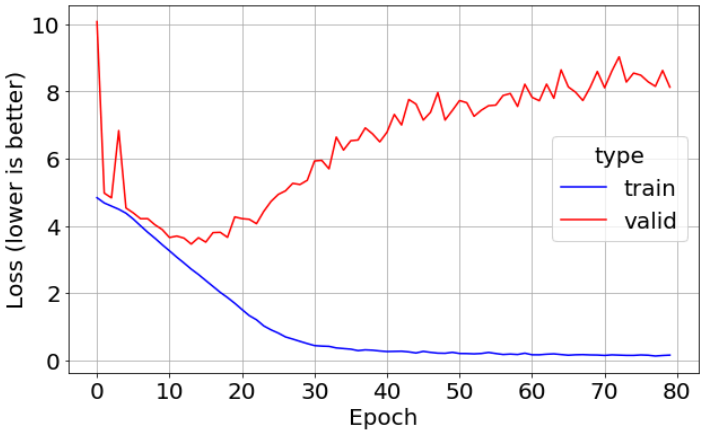
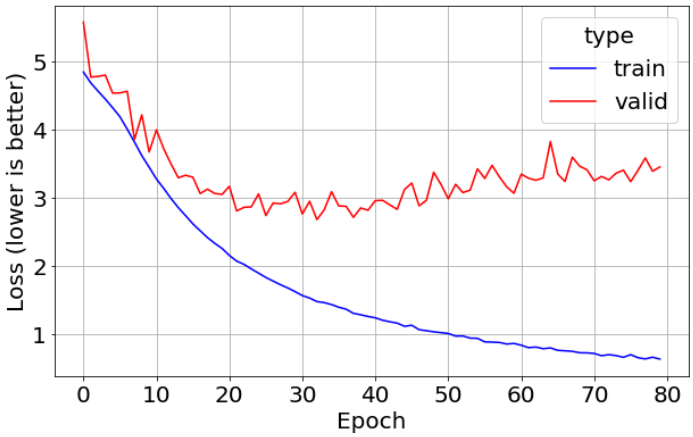


Fig 15b: CNN Learning Curve, (dogs, augmented)



The accuracy of the un-augmented Stanford data (Fig-16) shows a training overfit with validation performance peaking at around 22%. However as previously discussed, this is a simple architecture trying to classify 120 dog breeds with less than 200 training samples per class. This sample size is enough for strongly distinctive classes, but similar ones like dog breeds will need something in the four to five figure range.

Augmentation (Fig-17) improves this to around 38% validation accuracy with the

Fig 16: CNN Model Accuracy, (dogs, direct)

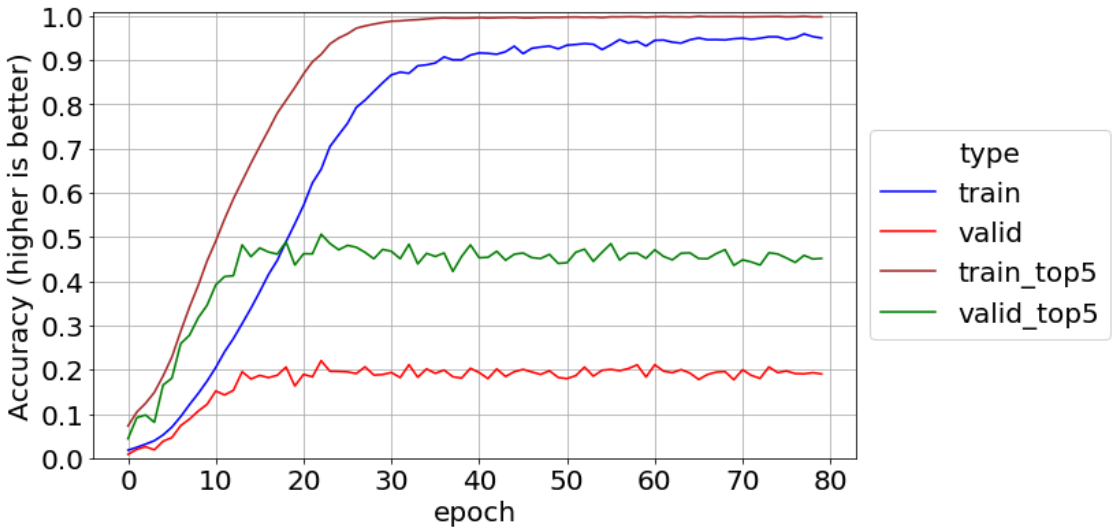
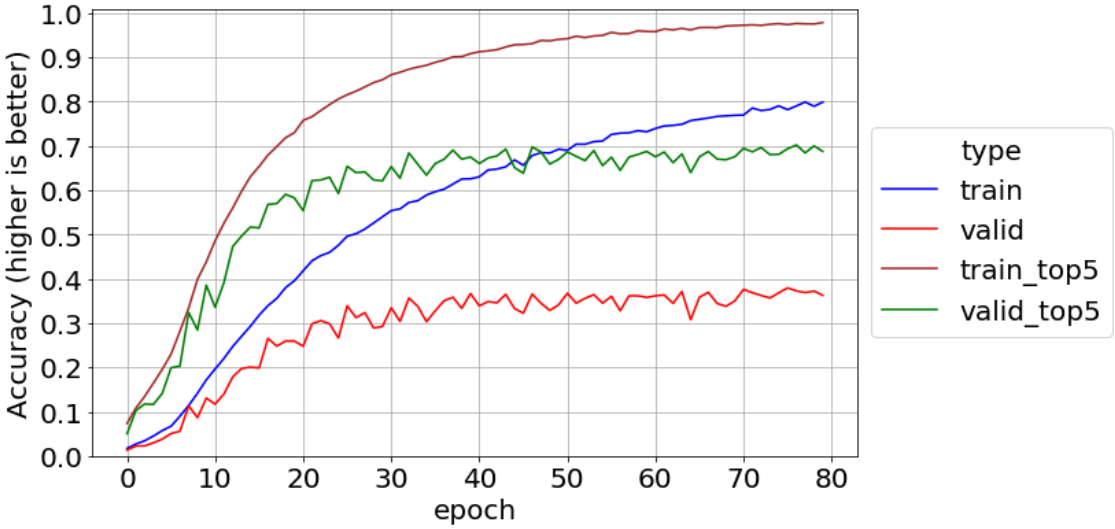


Fig 17: CNN Model Accuracy, (dogs, augmented)



training accuracy not clamped. Ideally the training should have been continued longer due to the slightly rising curves on the two validations, but the current training took 8 hours with a NVIDIA Tesla T100 GPU.

Table-3 lists top and top-5 accuracy results for the three dog breed training sets. Keras was set to store the weights from the epoch with the highest validation accuracy, in addition to the final weights when training ended. With the augmentation option, this forms a four-corner matrix listed under *Model Source*. The best performing settings was augmented data with *best weights* which reached 38% (top-5 70%) accuracy. There was some suspicion that the *best weight* option would be overfit to the validation data as previously discussed, but the *Stanford Test* results were within a percent of it. V2 performance was poor, roughly have that of Stanford.

As is, the model will correctly identify the dog breed from a new image almost 4 times out of 10 (7 out of 10 times if given five choices). This somewhat poor performance establishes a baseline for pre-trained models with transfer learning to beat.

Table-3: Full Dog Breed Results with Basic CNN Model

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Best Weights	20.75	19.50	9.75	48.69	47.46	29.08
Final Weights	16.94	15.00	8.50	43.06	38.75	25.50
Best Weights (augmented)	38.06	37.58	18.92	70.22	69.83	44.33
Final Weights (augmented)	37.14	35.50	16.42	68.06	67.29	41.08

Transfer Learning Image Classification Experiments

VGG19 Pretrained Model

The Visual Geometry Group out of the Oxford University did well in the 2014 ImageNet Challenge: first place in the localization track, second in classification. The two main members of their model family, VGG-16 and VGG-19, are named for the number of weight layers (Simonyan, 2014). VGG-19 has five max-pool layers which are proceeded by features maps of various amount and size: (2x64, 2x128, 4x256, 4x512, and 4x512). The last one feeds into three fully connected layers of size 4096, 4096, and 1000 for a softmax output of the predicted classes.

Past winners used large filters such as 11x11 with stride 4, but VGG went the other direction: 3x3 with stride 1. This works because when convolutions are stacked without pooling in between, the effective receptive field is larger. For example, a convolution stack of three 3x3 filters has the following benefits:

1. Equivalent to a 7x7 filter
2. Three ReLU layers provides a stronger discriminative function than one
3. Number of weights reduced to 55 percent of original (3*3*3 / (7*7))

This requires a depth increase, in fact 19 layers was extreme at the time, but the performance spoke for itself.

The 3x3 filter size is the smallest size that can capture left/right and up/down information with regards to the center. Curiously, one of their VGG-16 models had a filter with size of 1x1 which cannot actually convolve but does still apply scaling and a nonlinear function.

Keras provides the VGG-16 and -19 models with fully trained ImageNet weights. Transfer Learning techniques must be used to train on the *Flowers* data because its five

classes are not in ImageNet. However, the models also have an *include-top* option which produces the trained model without the fully connected layers. Users can then add on their own with the final layer sized to the number of classes. Training is performed as usual, but the weights of the original layers are frozen to keep the learned features from being disrupted.

Starting with the *flowers* dataset with augmented images, the first experiment was to determine the optimal amount of fully connected layers (all with 256 nodes). Fig-18 is the learning curve for one layer and it seems a little better (lower) and less ragged than fig-19 for two layers. The extra layers gave the model the capacity to overfit.

Fig 18: VGG19 Learning Curve, (flowers, single FC)

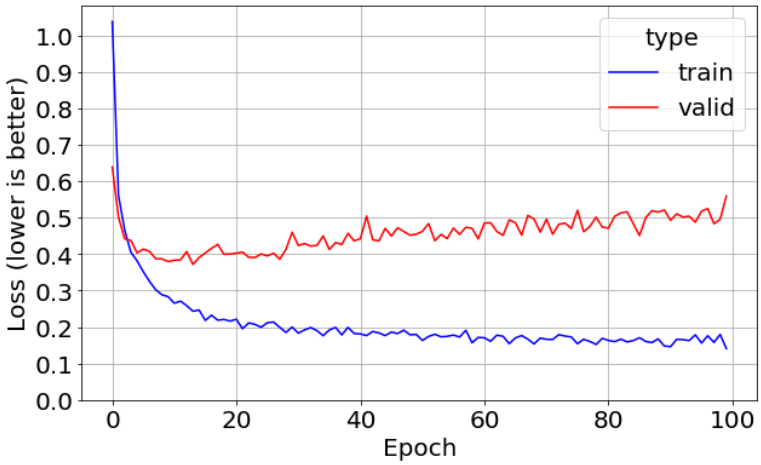
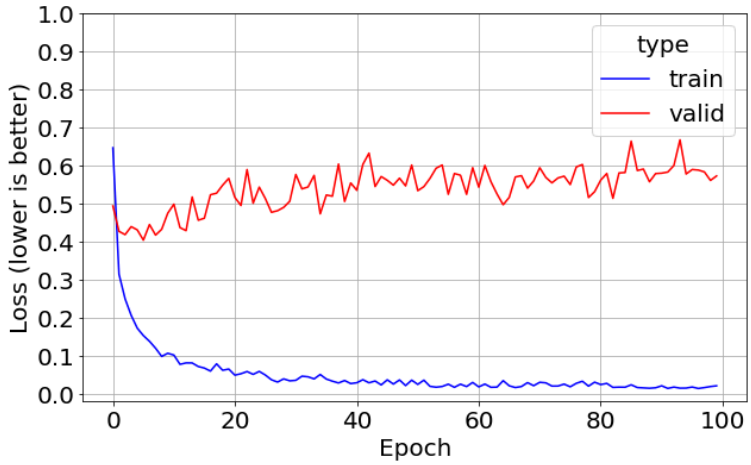


Fig 19: VGG19 Learning Curve, (flowers, double FC)



For accuracies, the model with two fully connected layers (fig-21) also shows some overfit to the training data, but this does not hurt validation performance which peaks at about 89%. In comparison, the single fully connected layer model is a percent or two less (fig-20). This suggests that the layer helps the model fit more than overfit. However, ll the noise in the red validation curves makes it difficult to form a definite conclusion.

Fig 20: VGG19 Accuracy, (flowers, single FC)

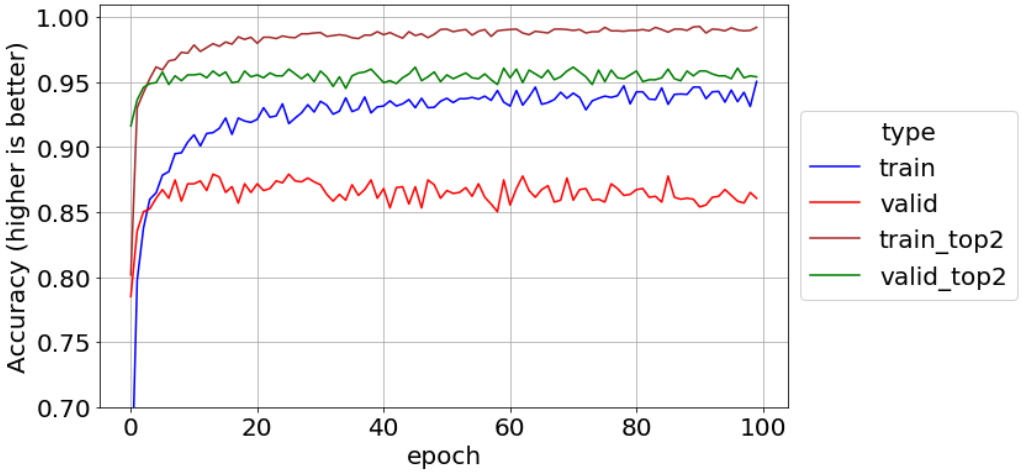
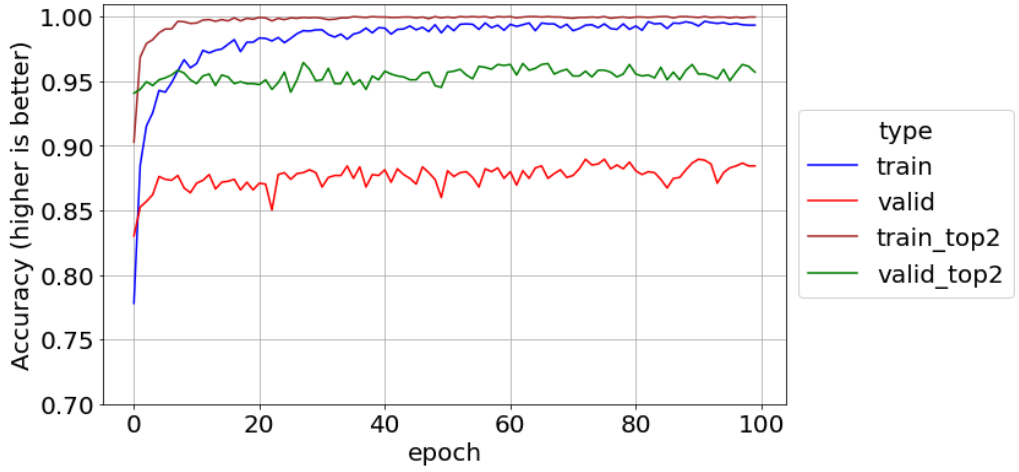


Fig 21: VGG19 Accuracy, (flowers, double FC)



Fine-tuning in Transfer Learning

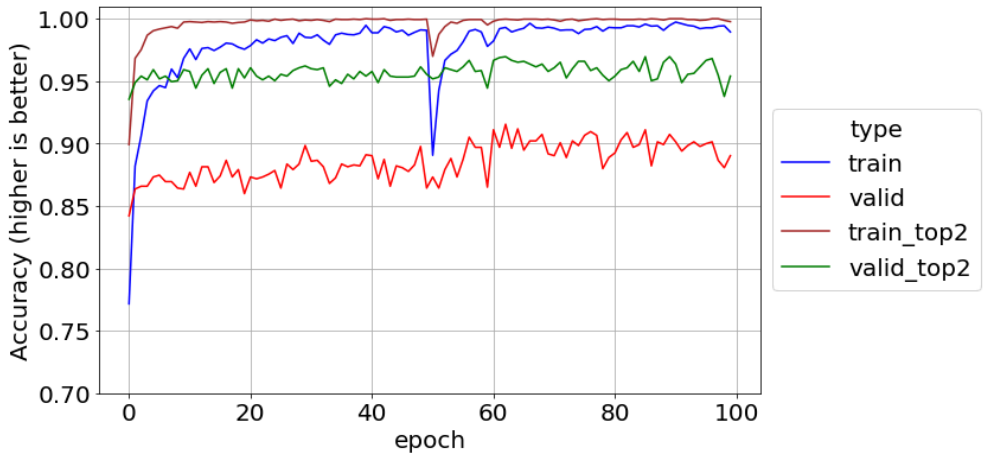
The previous activities all dealt with using transfer learning as a feature extractor, but models can be fine-tuned by letting layers in the convolution path also train. However, guidelines must be followed to perform this correctly. Firstly, the selected layers must be those closest to the fully connected ones which perform higher order activations as compared to the edge and line detections in the first layers. Secondly, the fully connected layers should already be partly trained, or the backpropagation might be too disruptive. Fig-22 shows the trainable layers used to tune VGG19, they start at the last convolution layer. The previous layers with their edge, line, and feature detection still have frozen weights.

```
Layer (type)                 Output Shape              Param #
=================================================================

                        <SKIPPED LAYERS>

_________________________________________________________________
block5_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
global_average_pooling2d_4 ( (None, 512)               0
_________________________________________________________________
batch_normalization_8 (Batch (None, 512)               2048
_________________________________________________________________
dense_8 (Dense)              (None, 256)               131328
_________________________________________________________________
batch_normalization_9 (Batch (None, 256)               1024
_________________________________________________________________
dense_9 (Dense)              (None, 5)                 1285
=================================================================
Total params: 20,160,069
Trainable params: 20,158,533
Non-trainable params: 1,536
_________________________________________________________________
```

Figure 22: Printout of all trainable layers during fine-tuning

One good approach to fine-tuning is to start training without it, archive the model, make the convolution layer trainable, then train some more. This was attempted on the *flowers* dataset, but a nasty spike appeared at the crossover point (fig-23). Investigation showed it was caused by the ADAM optimizer which calculates a dynamic learning rate per feature. The previous work did the multi-hour training in segments because of dropped internet connections and frozen Google Cloud Platform instances, but this was not a problem because Keras saves gradient values along with the rest of the model. However, for fine-tuning, the model object must be recompiled into a new model after layers are made trainable, and ADAM starts from scratch.

Fig 23: VGG19 (flowers, single FC, fine-tuned)



In online tutorials and examples, fine-tuning was always performed with standard SGD and momentum instead of an optimizer like ADAM. Since the first 50 epochs were saved, the fine-tuning was repeated with lr = 0.001 and momentum = 0.9 (fig-24). The spikes disappeared and the graphs are less noisy. It is not easy to determine which approach is better because the peak values with ADAM (91-92%) were higher than the SGD peaks.

In fig-25, the learning rate during tuning was increased to 0.004 for epochs 51-75 and kept at 0.001 after that. The validation results climb up similar to ADAM but without the spike. A higher learning rates value of 0.01 was tried but did not show any improvement.

Fig 24: VGG19 (flowers, SGD=0.001 fine-tuning)

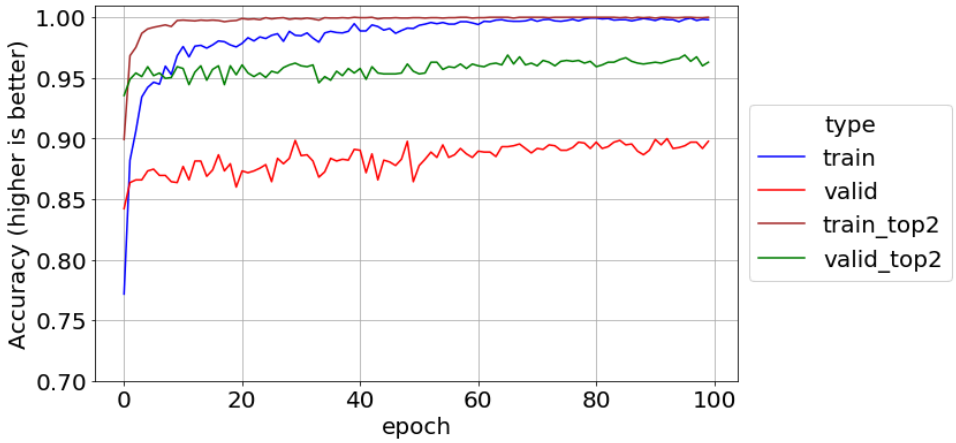
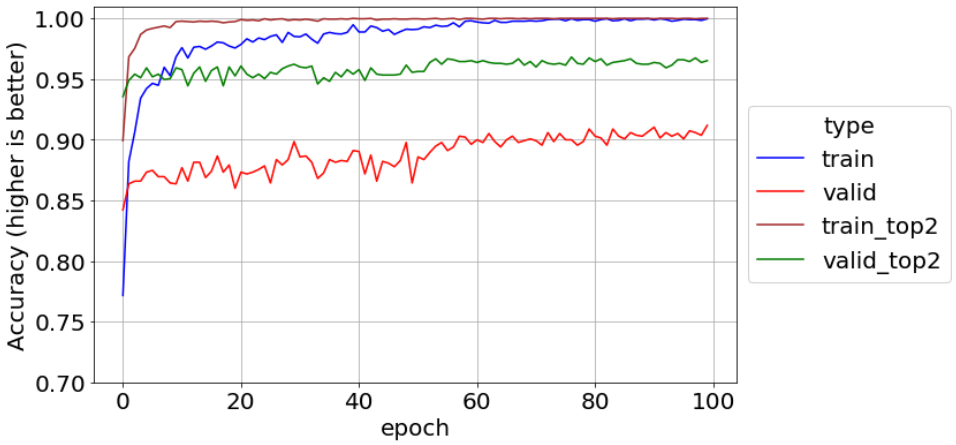


Fig 25: VGG19 (flowers, SGD=0.004, then 0.001 fine-tuning)



VGG19 Dog Breeds

The previous activities on *flowers* were repeated on the *Stanford* dataset. The VGG19 model was the same except for the final layer being 120 nodes instead of 5. Since the same trends appeared and for brevity, only two plots are presented. Fig-26 uses augmented data and a single fully connected layer of 256 nodes. Its validation accuracy peaks early at almost 74% then slowly declines due to a slight overfit. In comparison, fig-27 has two fully connected layers and fine-tuning after epoch 60. This causes the validation accuracy to start rising again up to about 76%. The training accuracy also improves by a couple percent in a noticeable step.

Fig 26: VGG19 (dogs, augmented, single FC)

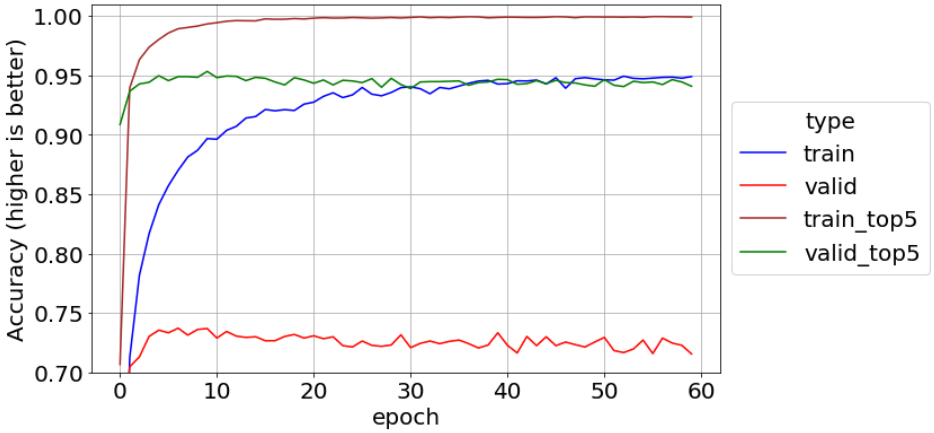
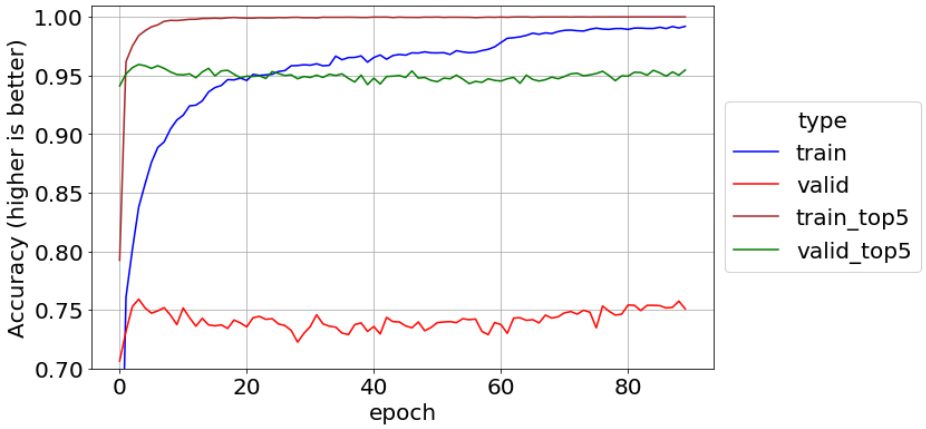


Fig 27: VGG19 (dogs, double FC, lr=0.0001 tuning)



One benefit the *Stanford* data has over *flowers* is that models with ImageNet weights can be used in their full original form to predict dog breeds. However, they might also predict any of the other 880 classes they support. The *Full Original* row in Table-4 reports these numbers with the value in parenthesis being the percentage of non-dog predictions. In fact, this value roughly corresponds to the accuracy difference between the two models which makes intuitive sense. Transfer Learning effectively states: "you can only predict certain values now, not your full original set."

Table 4: Dog Breed Results with Original and Transfer Learning VGG19 Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	60.06 (12.52)	60.54 (14.26)	34.92 (19.29)	82.72	81.46	60.75
Best Weights	75.50	74.75	46.08	95.08	94.92	73.16
Final Weights	73.56	72.04	44.50	94.81	93.91	71.33

Table 3 (Repeated): Full Dog Breed Results with Basic CNN Model

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Best Weights	20.75	19.50	9.75	48.69	47.46	29.08
Final Weights	16.94	15.00	8.50	43.06	38.75	25.50
Best Weights (augmented)	38.06	37.58	18.92	70.22	69.83	44.33
Final Weights (augmented)	37.14	35.50	16.42	68.06	67.29	41.08

As before, the *Best Weights* values were slightly higher than the *Final Weights* from the model saved after the final epoch. The highest accuracy was the *Stanford* validation set at 75.5%, but the test set was close enough behind (74.75%) to allay concerns about overfitting. The VGG-19 greatly outperformed the original basic CNN model (table-3, repeated for convenience), around 35% higher accuracy and 25% higher for top-5.

Resnet-50 Pretrained Model

2015 was a repeat of the year before with a disruptive new model sweeping an image competition. The Deep Residual Network (ResNet) architecture from Microsoft Research won first place in all five ILSVRC tracks. ResNet works by adding skip connections where past values bypass intermediate layers and get summed in after them (fig-28).

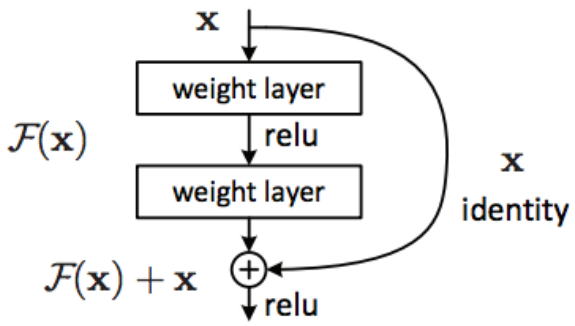


Figure 28: ResNet Structure (Public Domain)
https://commons.wikimedia.org/wiki/File:Resnet.png

A plain network learns a feature at a given layer, but in residual learning, the network learns the difference between the feature and the layer input. This term, known as the residual, is generally easier to optimize for. A trivial example is the identity

function $F(x) = x$. Without a skip connection, x must be brought through multiple non-linearities without alteration. In a ResNet, x is already available through the skip connection, so the weights determining $F(x)$ can simply be zeroed out. During backpropagation, the skip connection acts as a "gradient highway", and many online tutorials spread a misconception about it. They claim that deep CNNs with many layers always suffer from the vanishing gradient problem and ResNets are the cure. However, the original paper (He, e.a 2015) claims their use of batch normalization already eliminates vanishing and exploding gradients.

These issues were mentioned briefly earlier in this thesis but are worth revisiting. The nearly flat regions on each size of a sigmoidal curve sharply reduce any error value that is backpropagated though it. If this happens in multiple layers, there is not enough error left to properly update weights in the starting layers. Even worse, the later layers cannot function properly if they do not get healthy inputs from the starting layers, so the whole network may fail to train. There is also an exploding gradient issue when the steep parts in the center of the curve get cascaded, but this is less common. Using ReLUs instead of sigmoidal activations helps because their gradient is 1, but there can still be a gradient problem caused by a cascade of extreme weights.

Batch normalization, as specified in Eq 16-19, works well for preserving healthy gradients. For a given weight, all the activation values in a minibatch are analyzed as an aggregate and scaled appropriately with the goal of reducing internal covariate shift. In other words, the mean and variance of the activations in each layer are independent of the values themselves. This should ideally be done with a large batch size to reduce differences between batches, but that is usually the case with GPUs in recent times

anyway. As shown in Eq 16-18, it is nearly identical to z-scaling, just with a small constant named epsilon which improves stability. Eq-19 performs an additional scaling and shifting with alpha and beta. These are learned parameters which compensate for higher order interactions between layers. The net effect of all this is to eliminate the vanishing gradient problem as well as allowing higher learning rates.

$$\mu_{\mathrm{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{16}$$

$$\sigma_{\beta}{}^{2} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathrm{B}}) \tag{17}$$

$$\widehat{x_i} = \frac{x_i - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\beta}{}^{2} + \epsilon}} \tag{18}$$

$$y_i = \alpha\widehat{x_i} + \beta \tag{19}$$

The paper's authors discuss a training error in large plain CNNs. This is not an overfitting or vanishing gradient problem but instead a degradation, somewhat like signal to noise ratio in electronics:

> We have observed the degradation problem – the 34-layer plain net has higher training error throughout the whole training procedure, even though the solution space of the 18-layer plain network is a subspace of that of the 34-layer one. We argue that this optimization difficulty is unlikely to be caused by vanishing gradients. These plain networks are trained with BN which ensures forward propagated signals to have non-zero variances. We also verify that the backward propagated gradients exhibit healthy norms with BN. So, neither forward nor backward signals vanish.

This thesis covers the issue in depth mainly because so many tutorials are misleading. The paper states that degradation is usually masked by the larger problem of vanishing gradients and theorize that degradation is caused by the difficulty of doing identity

mapping through non-linearities as discussed previously. Skip connections facilitate a deeper layer stack and the paper discussed training ImageNet with a 152-layer model as well as CIFAR-10 with a 1000 layer one.

About the same time, a similar architecture appeared called Highway Networks where the connections can be open and closed, similar to Gated Recurrent Units (GRUs) in the RNN family.

One possible issue is that a skip connection may arrive in a layer that has a smaller size and larger depth than that is departed from. The size issue is solved by using 1x1 convolution layers with the proper stride. Multiple 1x1s can be used to increase (or decrease) the depth (aka number of channels). The *x* and *y* dimensions can be reduced with stride. Once their dimensions match, the skip connection and residual arrays are simply added together.

The Keras library supplies Resnet-50 pretrained with ImageNet weights. It consists of five stages, each having a convolution block and two to five Identity blocks. Each type of block contains three convolution layers for a total of 50 layers and over 23 million trainable parameters.

The *flowers* dataset was trained for 50 epochs with the ADAM optimizer, then fine-tuned with 50 epochs of SGD at lr=0.001. The learning curve (fig-29) indicates the training data was memorized and the validation data settled on a midrange value of about 1.15. The accuracy plot (fig-30) shows the same trends. It appears very ragged, but part of this is because the y-scale is expanded. Validation accuracy is about 79%, and no real improvement is seen from tuning. This performance is much lower than VGG19, and a touch below the basic CNN.

Fig 29: Resnet50 Learning Curve, (flowers)

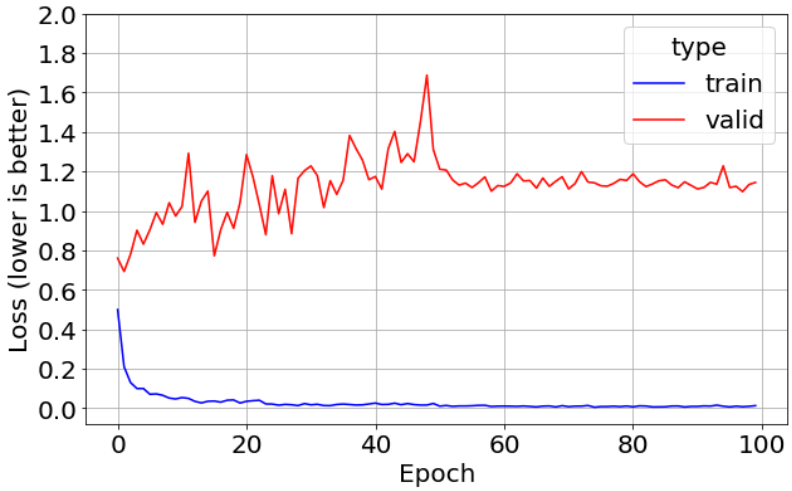
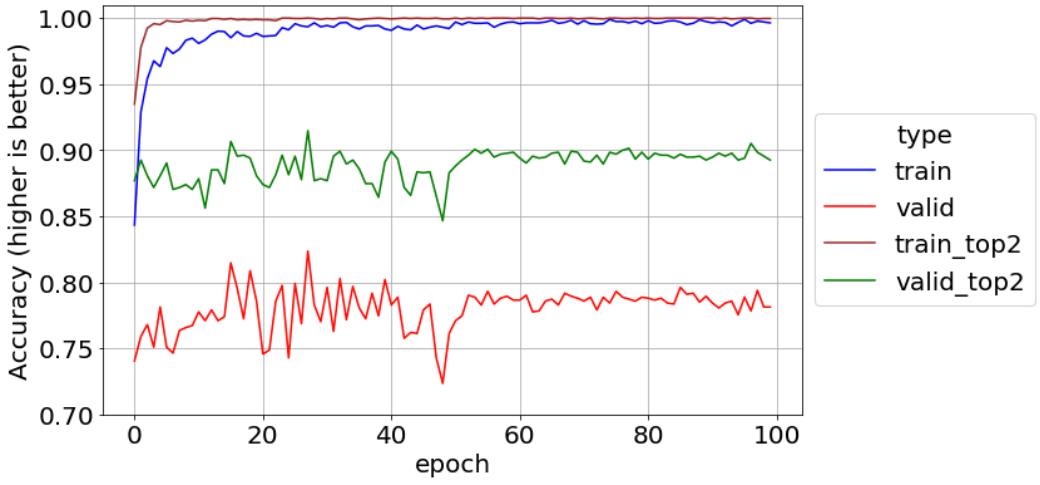


Fig 30: Resnet50 (flowers)



For the *Stanford* data, the learning curve (fig-31) had a much smoother validation line than for *flowers*. It gradually rises up due to overfit until epoch 50 when the fine-tuning starts, then drops somewhat. The training accuracy (fig-32) rises from about 98 to 99%, while the validation also rises from roughly 79 to 81% all due to fine-tuning. The shift is about the same size as that seen for the VGG-19 model.

Fig 31: Resnet50 Learning Curve (Stanford)

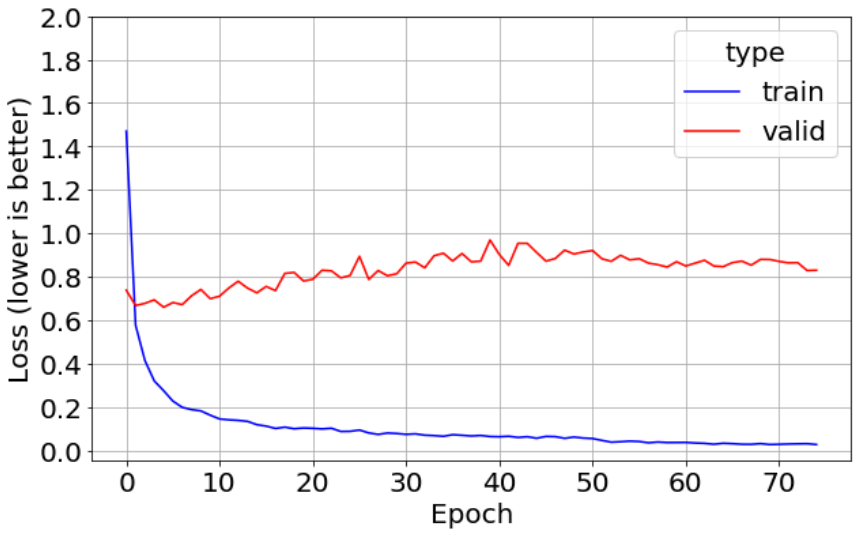
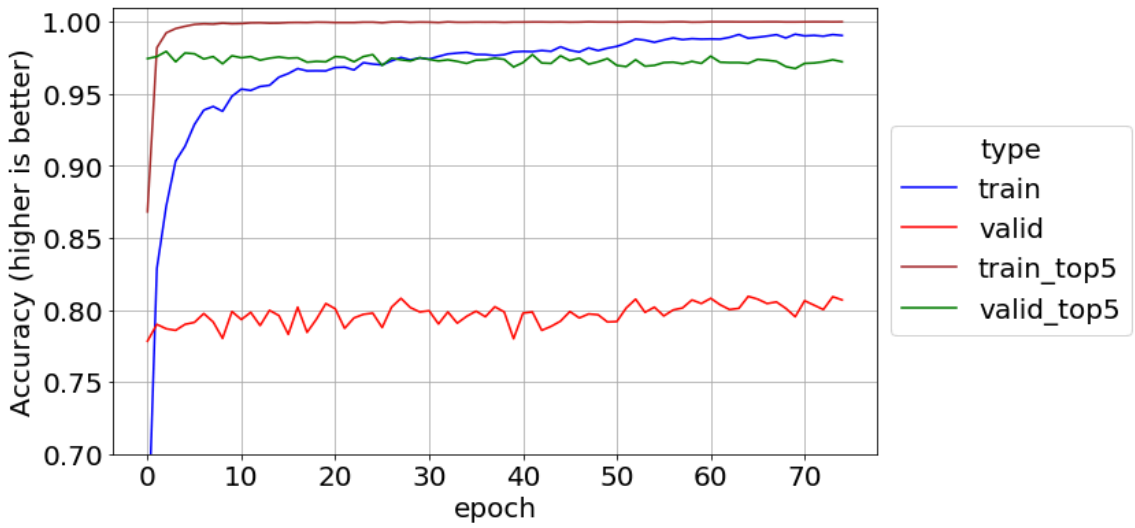


Fig 32: Resnet50 Accuracy (Stanford)



From Table-5, the ResNet-50 model out of the box will classify *Stanford* validation breeds correctly 64 % of the time, as an incorrect breed 25%, and as a non-dog 11.1% of the time. In comparison, a ResNet-50 with transfer learning and fine-tuning will hit 81% accuracy on the same data after 80 epochs of train (final model).

The ResNet-50 performs top 5% accuracy better than VGG-19 (at 75.5%)

Table 5: Dog Breed Results with Original and Transfer Learning ResNet50 Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	64.08 (11.1)	65.50 (11.1)	36.00 (25.42)	84.00	83.75	64.31
Best Weights	80.25	80.21	49.42	97.33	96.75	75.58
Final Weights	81.06	79.67	50.67	97.42	97.12	74.83

DenseNet Pretrained Model

The authors of the original DenseNet paper describe how it exists as one in an ecosystem of models for similar solutions:

> ResNets and Highway Networks bypass signal from one layer to the next via identity connections. Stochastic depth shortens ResNets by randomly dropping layers during training to allow better information and gradient flow. FractalNets repeatedly combine several parallel layer sequences with different number of convolutional blocks to obtain a large nominal depth, while maintaining many short paths in the network. Although these different approaches vary in network topology and training procedure, they all share a key characteristic: they create short paths from early layers to later layers. (Huang, e.a. 2018)

ResNet was followed up in 2018 by a more extreme version known as DenseNet. It actually connects each layer with every layer before it in a feed-forward fashion. For a given layer, the feature maps of all preceding layers are used as inputs and it also becomes an input to successive layers. The number of direct connections *N* in a standard CNN with *L* layers is *L*, but for DenseNet it is:

$$N = L * \frac{(L+1)}{2} \tag{20}$$

As a reminder, ResNets add the input from the previous layer which also requires that weights preserve and pass on needed values which are older (closer to input). In one form of the model, the inputs are concatenated instead of added. This creates are tradeoff where less weight parameters are needed compared to ResNets, but more memory is used for backpropation because the effective number of layers has increased.

The second and more common form of DenseNets is called the residual interpretation with shortcuts to all previous layers. The memory requirements are even higher, and the authors recommend using a wider and shorter (less depth) design than common with ResNets. As a rule of thumb, DenseNets require quadratic memory with respect to depth, but implementations that extensively reuse feature maps can be done with linear memory.

Keras provides the pretrained DenseNet201 model with 201 layers and about 20 million parameters. The learning curve on the *flowers* data (fig-33) was not promising: the validation cost function increased with additional training and also grew in variance after finetuning was started (lr=0.0004 for epochs 51-75, lr=0.0001 for epochs 76-100). The validation accuracy (fig-34) actually dropped from about 76% to around 73% with a small recovery to near 75% towards the end. The extensive interconnections in DenseNet seem to easily get disrupted when convolutional layers are opened for fine-tune training.

Fig 33: DenseNet Learning Curve (flowers)

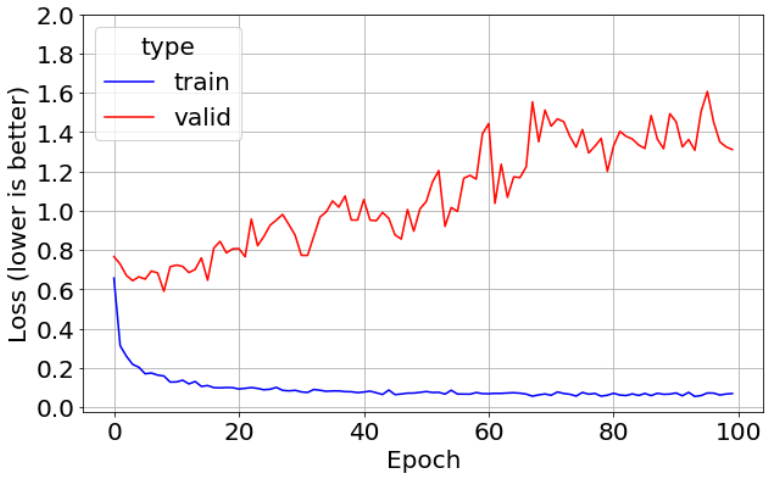
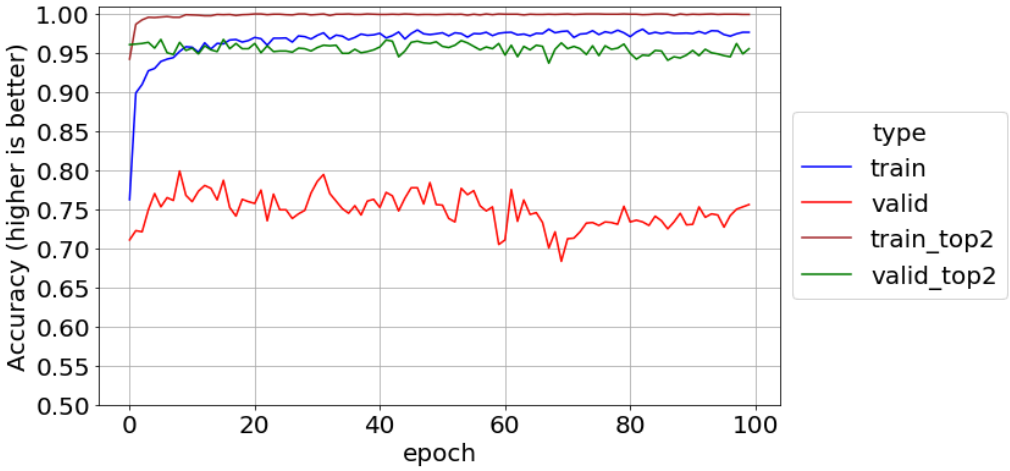


Fig 34: DenseNet Accuracy (flowers)



Curiously, the *Stanford* data behaved in an opposite fashion. The validation training curve (fig-35) stayed mostly level and its variance decreased during fine-tuning. For validation accuracy (fig-36) neither the ADAM optimizer nor fine-tuning ever shifted it much from an 81% baseline.

Fig 35: DenseNet Learning Curve (Stanford)

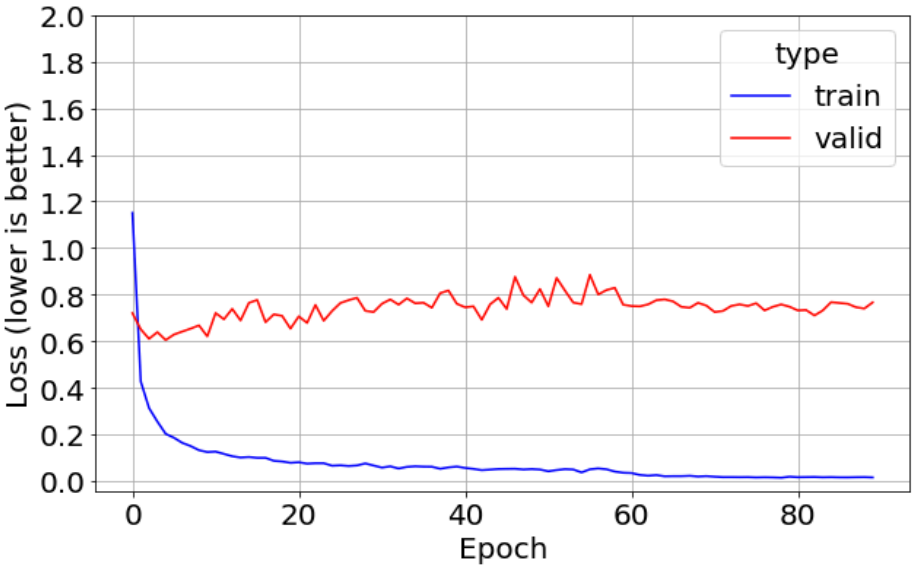


Fig 36: DenseNet Accuracy (Stanford)

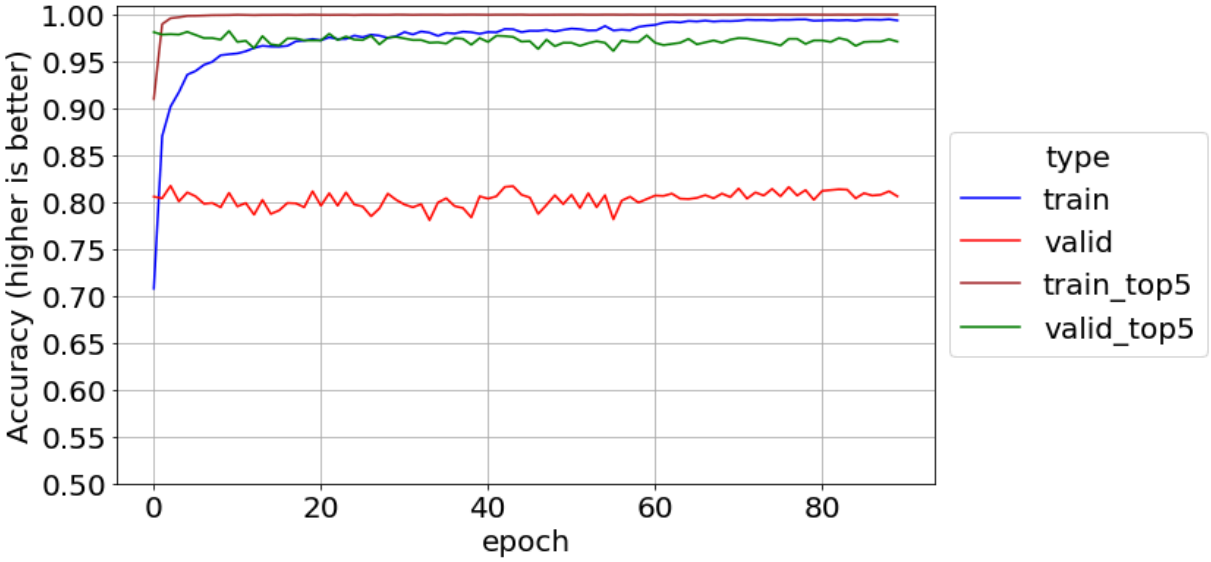


Table-6 is a full breakdown the DenseNet performance on the *Stanford* data. When compared to ResNet50 (repeated below), the fine-tuning results are almost the same (81.31% vs 81.06%), so the extra connectivity did not improve performance in this case. However, the original full model beat ResNet50 by about 5% (70.52 vs 64.08) while the non-dog predictions were about 11% for both. This means DenseNet102 has learned extra features that improve its ability to tell one dog breed from another.

Table 6: Dog Breed Results with Original and Transfer Learning DenseNet102 Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	70.52 (11.88)	71.67 (11.32)	45.50 (31.47)	86.03	86.42	69.75
Best Weights	81.06	80.79	51.75	97.22	97.12	77.42
Final Weights	81.31	80.96	52.92	97.44	97.04	77.33

Table 5 (REPEATED): Dog Breed Results with ResNet50 Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	64.08 (11.1)	65.50 (11.1)	36.00 (25.42)	84.00	83.75	64.31
Best Weights	80.25	80.21	49.42	97.33	96.75	75.58
Final Weights	81.06	79.67	50.67	97.42	97.12	74.83

Inception V3 Pretrained Model

The defining characteristic of the Inception family is that they avoid the overfitting problems of deep CNN networks by going wider instead. GoogLeNet is the common name of Inception v1, which won ILSVRC 2014 for image classification. A convolution layer uses multiple filters of different size in parallel, such as 3x3, 5x5, and 7x7. The rationale behind this is that the larger filter will detect features that are distributed more globally while smaller filters is for more local features. A simple example is a dog's head which will be large in a closeup but small in a distance shot.

This inception layer also has additional features. The larger filters are typically fed from 1x1 convolutions. Like with ResNets, these reduce the depth dimension with its costly computations, plus a ReLU transform can easily be included. A max pooling operation is also performed in parallel with the filters. Lastly, to keep the middle parts of the network from degrading or vanishing, two auxiliary classifiers are used during training, each applying softmax to their inception module outputs. These loss values are scaled and added to the standard loss.

For Inception v2, the costly large filters were changed to compositions of little ones. As discussed previously, two cascaded 3x3 filters is equivalent to a 5x5. In addition, a 3x3 can be factorized as a 1x3 feeding into a 3x1. Inception layers in v2 are completely made up of 1xN and Nx1 filters for performance improvement.

Inception v3 contains all the enhancements of v2 with some additions. There is also a 7x7 filter and RMSProp optimizer. Regularization is performed by using Label Smoothing in the cost function and adding batch normalization. (and sometimes dropout) to the auxiliary classifiers. There is also a v4 version that is not yet supported by Keras.

The Inception V3 performance on *flowers* data was worse than DenseNet but did improve with fine-tuning instead of dropping. The validation learning curve (fig-37), shows a major cost improvement on the switch from ADAM to fine-tuning with its stochastic gradient descent and momentum. The validation accuracy (fig-38) also jumps from about 67% to 74%. For comparison, DenseNet hit peaks of 80%.

Fig 37: InceptionV3 Learning Curve (flowers)

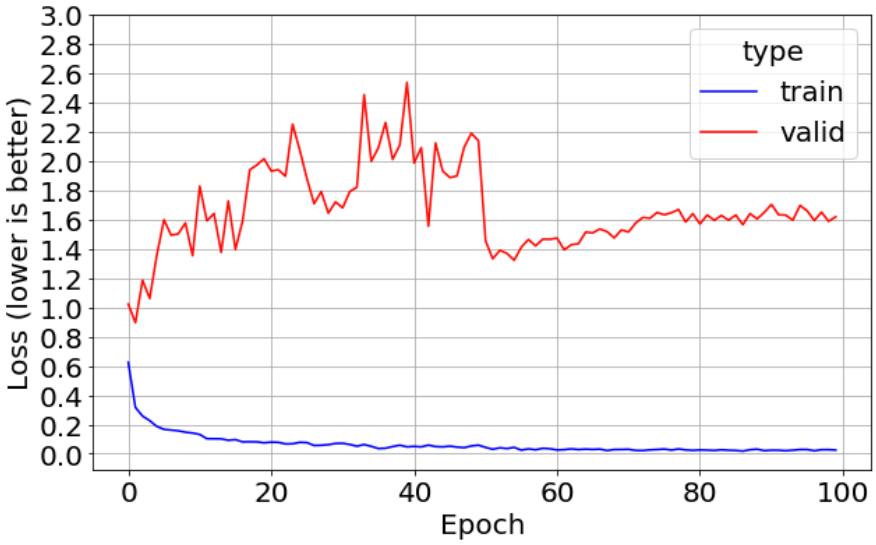
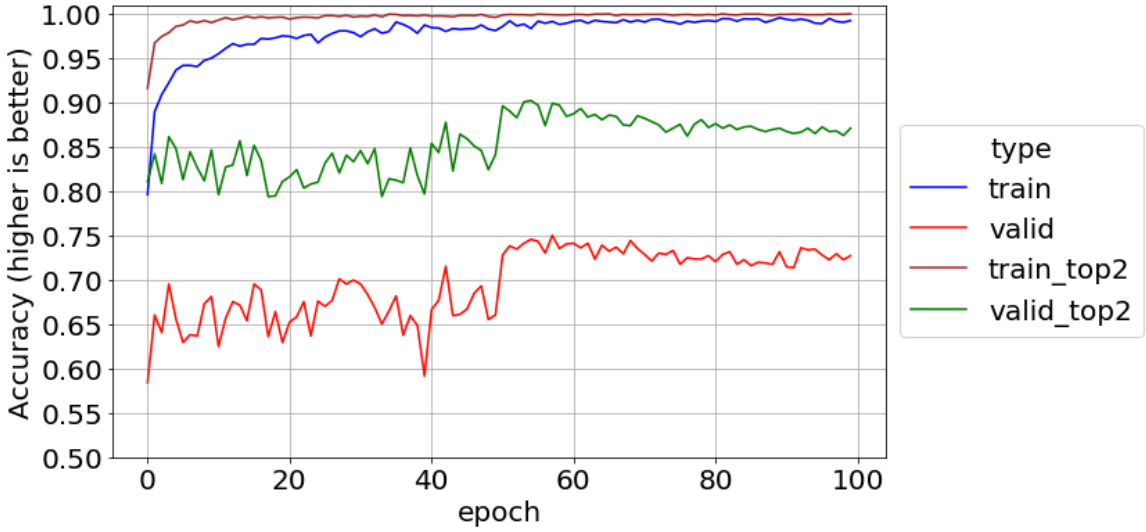


Fig 38: InceptionV3 Accuracy (flowers)



By comparison, the *Stanford* turned out to be almost textbook examples of different effects. The validation learning curve (fig-39) shows overfitting when it climbs as the training curve drops. However, they jump sharply towards each other when fine-tuning starts. Likewise, the validation accuracy increases by about 3% (fig-40). Something in the fine-tuning SGD with the 0.0004 learning rate or the 0.9 momentum caused an immediate regularization jump on the cost function manifold.

Fig 39: InceptionV3 Learning Curve (Stanford)

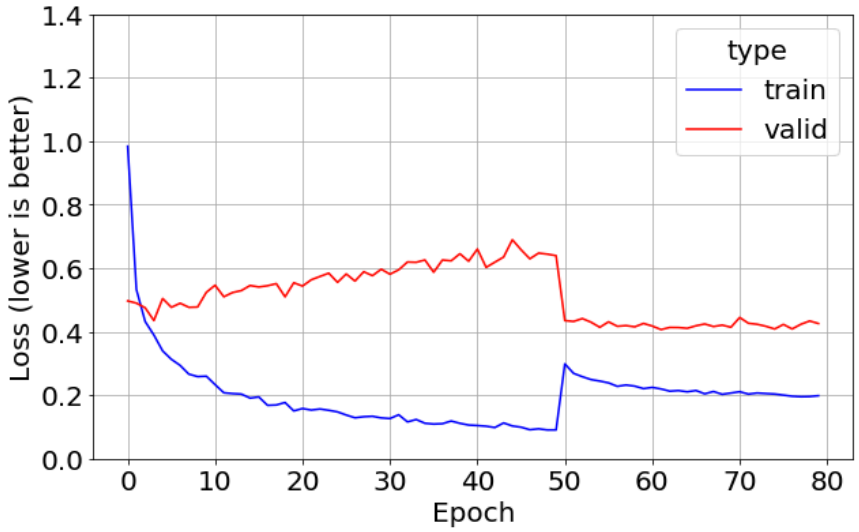
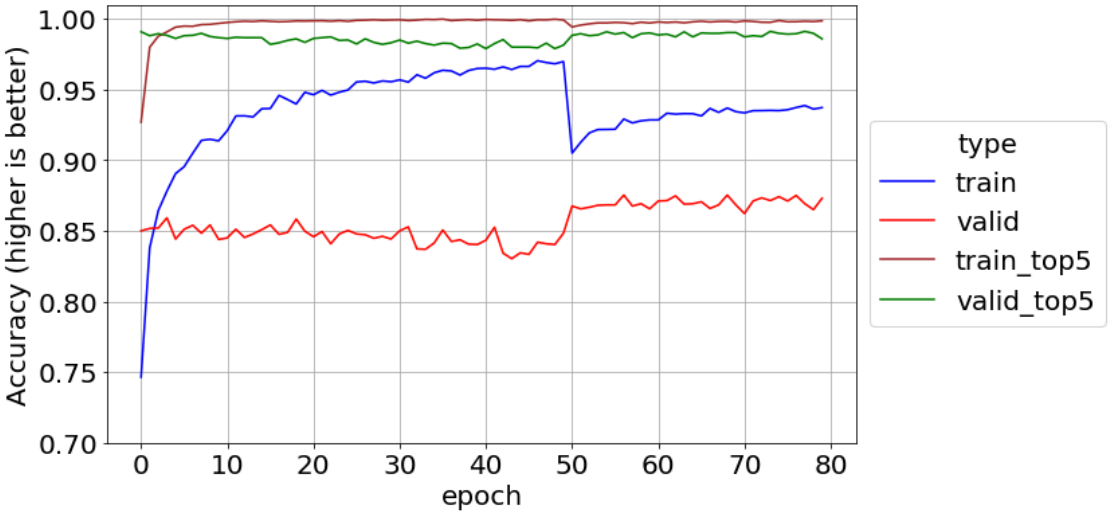


Fig 40: InceptionV3 Accuracy (Stanford)



The full result breakdown of Inception V3 (table-7) shows a large improvement over the previous contender DenseNet (repeated below). The validation accuracy peaks at 87.14%. Since this is great than the sum of the original model (73.3+12.4 = 85.7), the transfer learning not only corrected all the non-dog corrections to the correct breed, but also get better than before at distinguishing between breeds after the fine-tuning.

Table-7: Dog Breed Results with Original and Transfer Learning Inception Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	73.33 (12.40)	74.21 (11.14)	46.92 (28.34)	86.42	86.42	71.42
Best Weights	87.06	86.96	82.41	99.00	98.75	58.08
Final Weights	87.14	87.63	56.17	99.14	98.71	82.17

Table-6 (REPEATED): Dog Breed Results for DenseNet102 Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	70.52 (11.88)	71.67 (11.32)	45.50 (31.47)	86.03	86.42	69.75
Best Weights	81.06	80.79	51.75	97.22	97.12	77.42
Final Weights	81.31	80.96	52.92	97.44	97.04	77.33

Xception Pretrained Model

The Xception architecture easily found a home on Keras because it was first proposed by Francois Chollet, the creator and foremost maintainer of Keras. Google researchers developed it as an extension to the Inception architecture with one major change known as Depth-wise Separable Convolutions (DSCs).

A standard convolution with a single filter produces a volume with one dimension of length C, the number of channels or depth (ex: 3 for the first layer of an RGB image) which is a costly operation as the number of filters grow. In comparison, a DSC is only over a single channel and has size d x d x 1 which produces a depth dimension of C instead of N, the number of filters.

DSCs have a counterpart known as a Pointwise Convolution (PC) which have size 1 x 1 x N and basically translate the depth dimension back to N. The benefit of using this pair is that the number of operations is reduced by a factor of 1/N.

Curiously, the Xception architecture starts with PCs which feed into DSPs and then max-pooling which slightly outperforms InceptionV3 with the same number of model parameters. It is specified in the original paper as having three states denoted as: Entry, Middle and Exit Flows with extensive use of batch normalization. The goal is the complete decoupling of all correlations:

> We propose a convolutional neural network architecture based entirely on depth wise separable convolution layers. In effect, we make the following hypothesis: that the mapping of cross-channels correlations and spatial correlations in the feature maps of convolutional neural networks can be entirely decoupled. Because this hypothesis is a stronger version of the hypothesis underlying the Inception architecture, we name our proposed architecture Xception, which stands for “Extreme Inception” (Chollet 2017)

Xception with Flower Data Set

Training went well with the *flowers* data set. Similar to Inception, the validation learning curve (fig-41) took a big drop when training was switched to finetuning. The ADAM optimizer is widely used, but some researchers claim that plain old SGD with momentum can outperform it. The Inception family model results seems to indicate this, at least to the extent that a data scientist should try both.

Validation accuracy (fig-42) bounced around in the high 70s than took a major leap up to 86%, much better than the 75% peak with Inception.

Fig 41: Xception Learning Curve (Flowers)

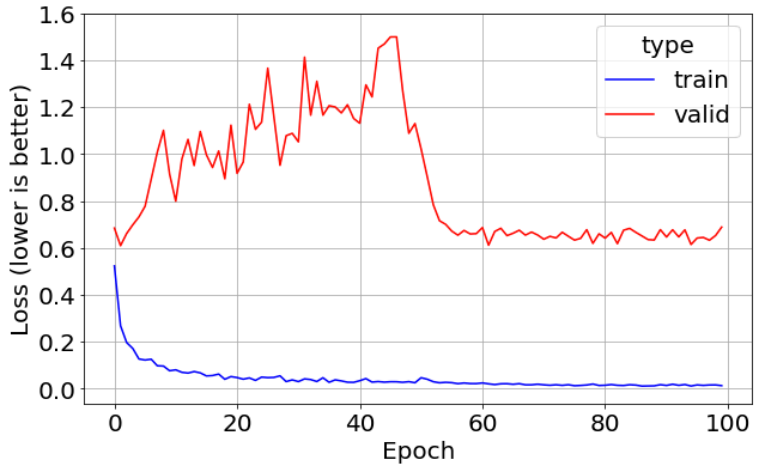
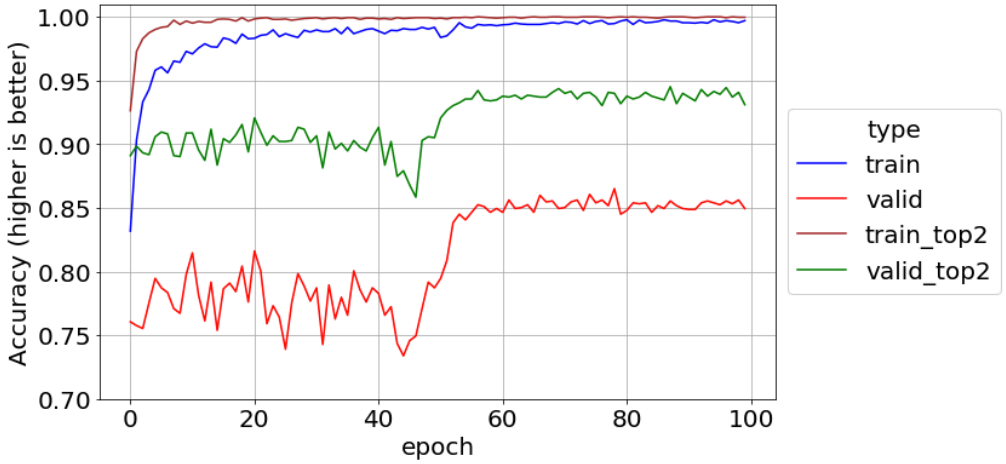


Fig 42: Xception Accuracy (Flowers)



Xception with Stanford Dataset

As usual, the respective loss functions for training and validation show a slow decline and rise (fig-43). The validation once again dropped due to fine-tuning than stayed constant. However, the validation accuracy (fig-44) started with peaks of 88% and never improved beyond that. Around epoch 38, it dropped a couple percent, then dropped a couple more when fine-tuning was started. It basically started at the performance level the Inception model trained 100 epochs to reach.

Fig 43: Xception Learning Curve (Stanford)

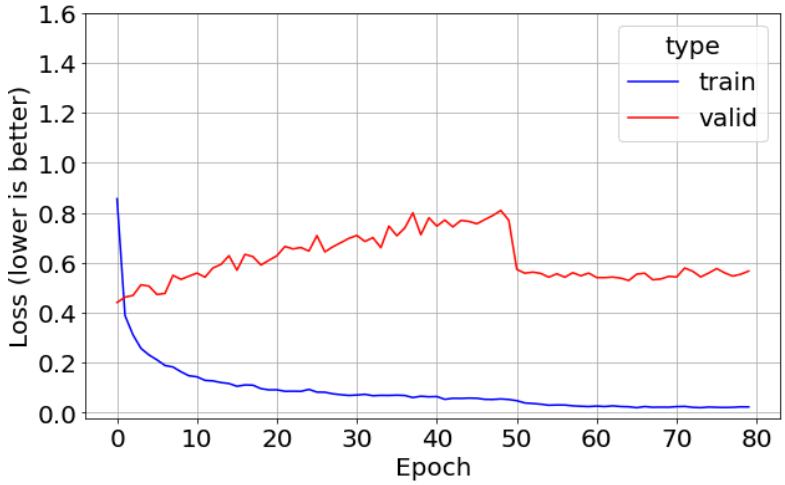
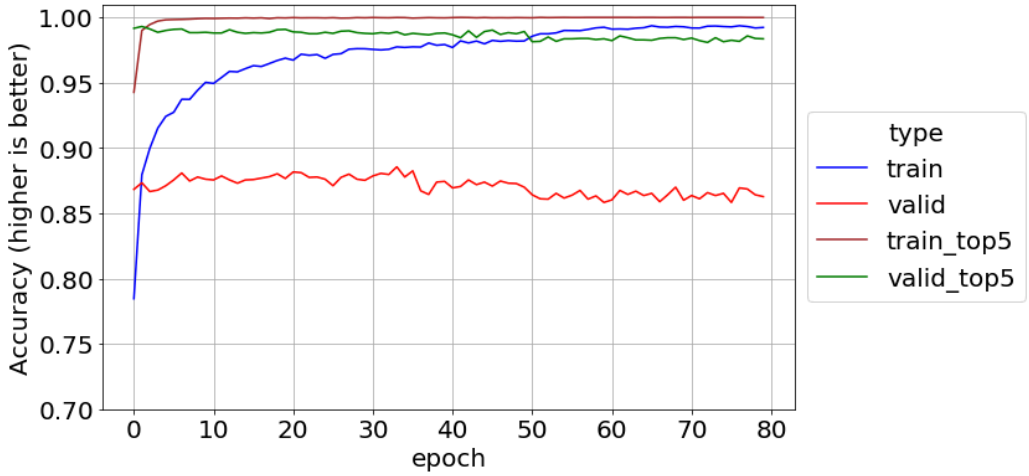


Fig 44: Xception Accuracy (Stanford)



The calculated accuracies for Xception (Table-8) did not quite reach those of Inception (repeated below) but were very similar at 86.64% vs 87.14% for validation accuracy. The closeness to the test set results are a guarantee the model did not overfit during validation.

The *V2* test accuracy is still below the two Stanford variants and no model has broken this trend.

Table 8: Dog Breed Results with Original and Transfer Learning Xception Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	74.08 (11.79)	74.58 (11.15)	47.08 (26.38)	86.69	86.92	71.92
Best Weights	86.64	86.87	56.33	98.19	98.46	79.17
Final Weights	86.17	86.71	57.33	98.28	98.58	79.67

Table 7 (REPEATED): Dog Breed Results with Inception Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	73.33 (12.40)	74.21 (11.14)	46.92 (28.34)	86.42	86.42	71.42
Best Weights	87.06	86.96	62.41	99.00	98.75	58.08
Final Weights	87.14	87.63	56.17	99.14	98.71	82.17

Inception-ResNet Pretrained Model

The last model considered is a hybrid developed at Google in 2016. Inception-ResNet combines the Inception architecture with residual connections:

> In this work we study the combination of the two most recent ideas: Residual connections introduced by He et al. in and the latest revised version of the Inception architecture. In [5], it is argued that residual connections are of inherent importance for training very deep architectures. Since Inception networks tend to be very deep, it is natural to replace the filter concatenation stage of the Inception architecture with residual connections. This would allow Inception to reap all the benefits of the residual approach while retaining its computational efficiency. (Szegedy, 2016)

There are two sub-versions of the architecture, but they have the same structure for their three modules which were renamed from Inception V2 and are now called A, B, and C. The two sub-versions also use the same reduction blocks, whose purpose is to change the width and height of the grid. However, they have different stems (operations performed before the inception blocks) and hyperparameters. The first sub-version, Inception-ResNet v1, has a computational cost similar to Inception v3, while v2 matches Inception v4. As seen in ResNets and elsewhere, 1x1 convolutions are used as needed to match up tensor dimensions. The original paper left out batch normalization in order to train the model with a single GPU, but it crept into later implementations.

One interesting effect in the paper is that with over 1000 filters, the residual variants became unstable and the network would die instead of train. Lowering the learning rate did not help and neither did adding batch normalization to the trouble layers. Reducing the residual by a scale factor between 0.1 and 0.3 was most helpful and never seemed to hurt the final accuracy, even when not needed.

Inception-ResNet with Flower Data Set

A pretrained Interception-ResnetV2 is available in Keras and was used on our datasets. Fig-45 shows the same learning curve trends that were seen with Inception and Xception where the validation loss function exhibits high variance for ADAM training, then converges for SGD/Momentum fine-tuning. The validation accuracy (fig-46) reaches a peak of 78% during fine tuning, which is not very impressive because Xception was 85% under similar conditions.

Fig 45: InceptionResNetV2 Learning Curve (flowers)

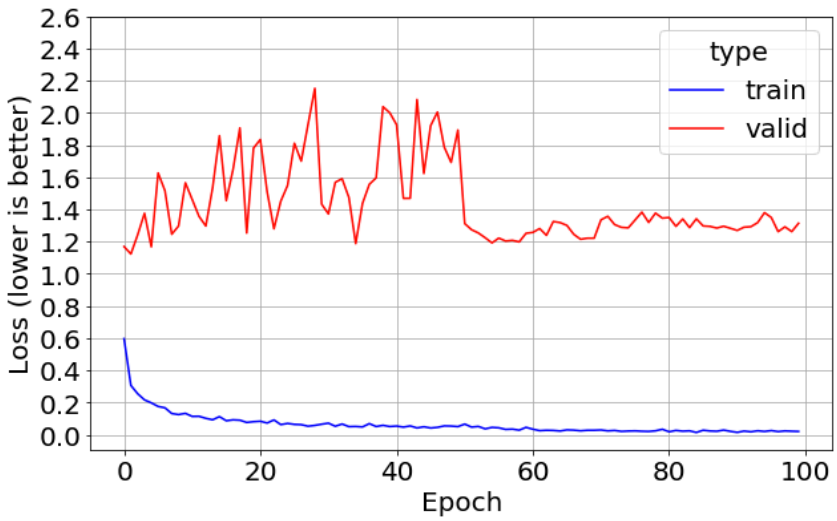
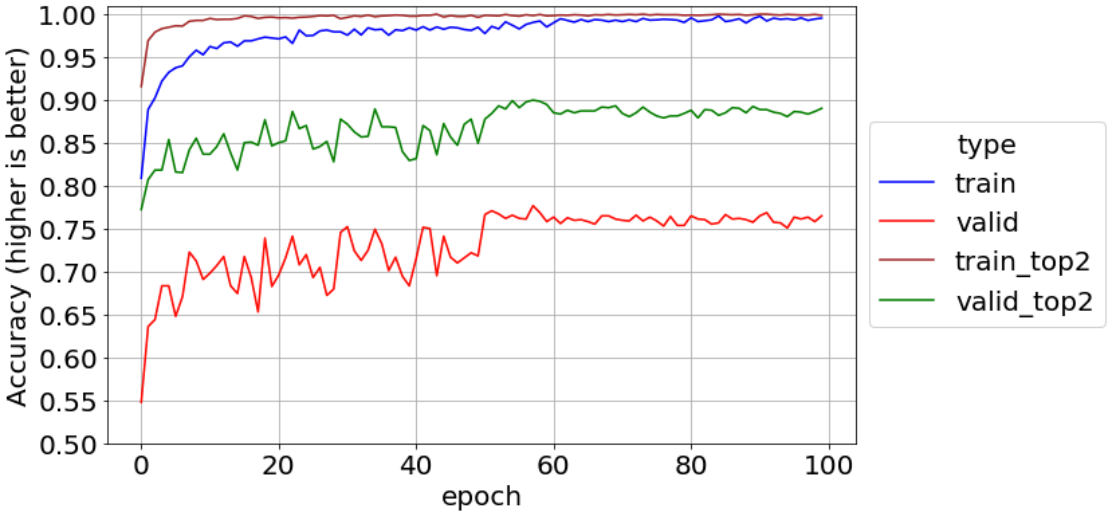


Fig 46: InceptionResNetV2 Accuracy (flowers)



Inception-ResNet with Stanford Data Set

Hybrid models (of anything) can be complex and there is a lot going on with the *Stanford* data. The learning curve (fig-47) shows the training loss function jumping down once finetuning starts, but the validation loss surprising jumps way up. This trend was seen with the Inception model, but not to this extent.

One the other hand, the validation accuracy (fig-48) acts like Xception's which starts high and declines. It does peak at 90% however which no other model managed to reach.

Fig 47: InceptionResNetV2 Learning Curve (Stanford)

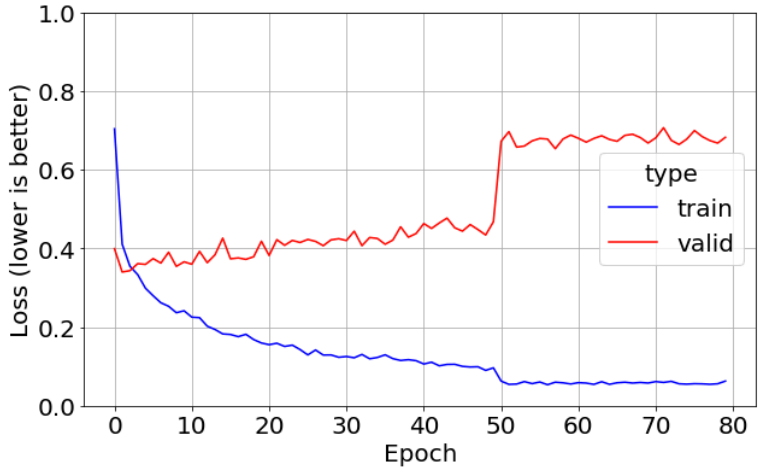
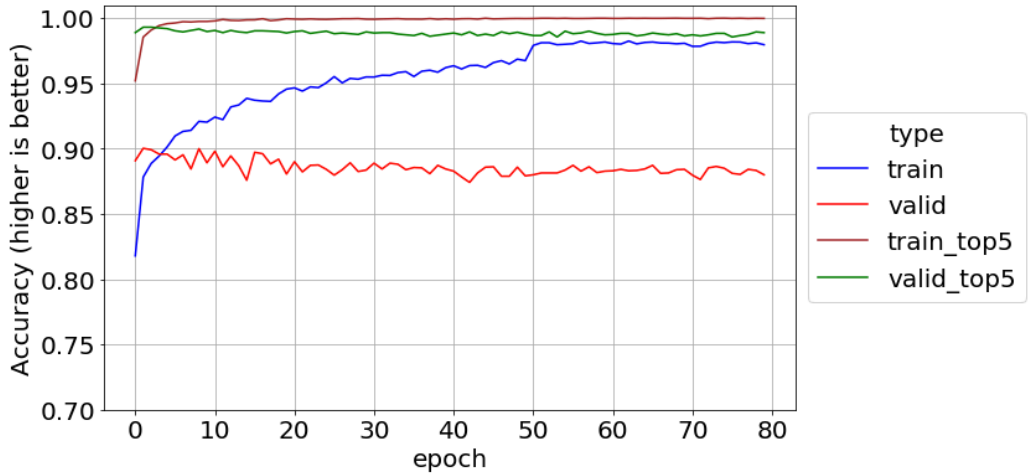


Fig 48: InceptionResNetV2 Accuracy (Stanford)



The full accuracy calculations (Table 9) reveal that Inception-ResNet beats Inception's performance (below) and is the new top performer. *Stanford* accuracy on both validation and test data is just over 90% with Top-5 accuracy above 99%. Using the same best model weights, the *V2* accuracies are still low (about 62% and 85%).

Table-9: Results with Original and Transfer Learning InceptionResnetV2 Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	74.28 (11.77)	74.79 (10.08)	50.50 (24.71)	86.83	87.08	74.00
Best Weights	90.08	90.13	62.08	99.16	99.17	84.92
Final Weights	89.81	90.25	61.41	99.08	99.29	85.67

Table-7 (REPEATED): Dog Breed Results with Inception Models

	Top-1 Accuracy			Top-5 Accuracy		
Model Source	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Full Original	73.33 (12.40)	74.21 (11.14)	46.92 (28.34)	86.42	86.42	71.42
Best Weights	87.06	86.96	62.41	99.00	98.75	58.08
Final Weights	87.14	87.63	56.17	99.14	98.71	82.17

Ensemble Models with Stanford Data Set

After reaching 90% validation accuracy with the InceptionResnetV2 model, there is one technique left to try. Often, CNN accuracy can be raised a percent or two by combining the output of different models which is known as an ensemble:

> Ensemble learning is an approach that combines the predictions of multiple learned models to enhance accuracy. Ensemble prediction using different learning algorithms or different CNN architectures may capture complementary information, which will in turn enable more robust predictions. An ensemble model is typically more stable model and less noisy than a single model. Ensembles usually make the predictive model more robust and give better accuracy on test cases. Ensembles reduce variance and sometimes bias. There are a number of approaches to create the final output value from an ensemble e.g. averaging probabilities, voting, median, max, etc. (Paul e.a. 2018)

This thesis will combine three models from the inception family since they performed so much better than the others. Another trick in CNN development is to combine models from different points in the same training run. These are not as helpful as models from different architectures, but as a training by-product, their cost is free. In fact, they are usually generated anyway to recover from network and dropped internet connections during long training runs. In this study, there is actually a four-corner matrix of models available between training and fine-tuning where each saved the final model and best weights to disk

Creating an ensemble is quite easy within Keras because the *model predict* function outputs a matrix of softmax scores with one row per sample and 120 columns for the possible classes. Result matrices from multiple models can simply be added together and argmax used to find the ensemble's predicted class. Each of the three models was used with the following combinations:

- Model skipped
- 4 single combinations of weight/model and train/fine-tuned
- 4 double combinations of weight/model and train/fine-tuned
- 1 quadruple combination of all settings for the architecture

This set of 1000 different combinations was looped over in software with the accuracy captured and sorted. Table-10 shows top five models whose components are specified by the following keys:

- i, x, r: Inception, Xception, and Inception-ResNet respectively
- w, m: Best weights, final model respectively
- 50, 80: Training, fine-tuning respectively

The top performer reaches 92.5% by using the training weights and models from Inception and Xception plus the fine-tuned weights and model from Inception-ResNet. The *Stanford* validation and test results are much closer now, usually 0.2% or less apart. The rank 4 entry is an auto-generated ensemble of ten different deep learning CNNs, each with millions of weights, something not even possible in the recent past.

Table 10: Top Five Ensemble Models for Validation Accuracy

Rank	Model and/or Weight Combination	Valid Acc	Test Acc	V2 Acc
1	im50, iw50, xm50, xw50, rm80, rw80	92.47	92.25	64.41
2	im50, iw50, xm50, xw80, rm50, rw50, rw80, rm80	92.39	92.38	65.58
3	im50, iw50, xm50, xw50, rm50, rw50, rw80, rm80	92.36	92.25	64.83
4	im50, iw50, iw80, im80, xm50, xw80, rm50, rw50, rw80, rm80	92.36	92.33	65.67
5	im50, iw50, xw80, rm50, rw50, rw80, rm80	92.36	92.34	64.42

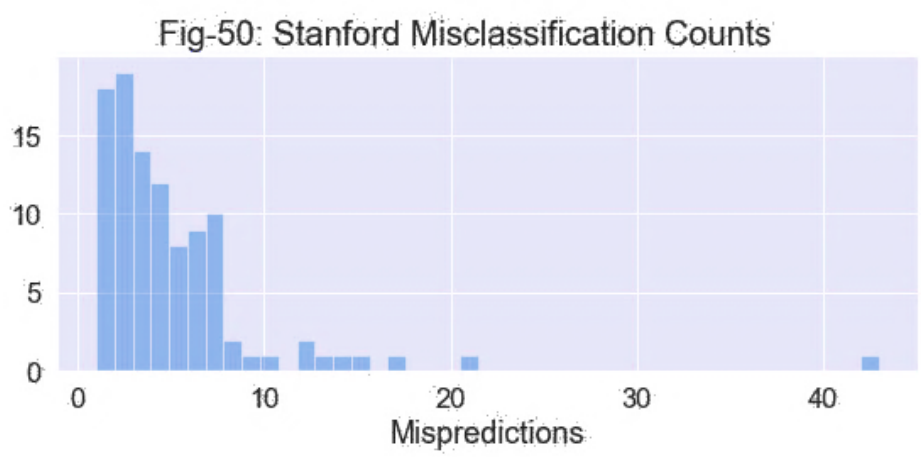
For the sake of curiosity, the ten worst performing ensembles are presented in raw form without analysis (fig-49). With no models at all, the accuracy across 120 breeds is 0.8% as expected.

Figure 49: Bottom 10 Ensembles

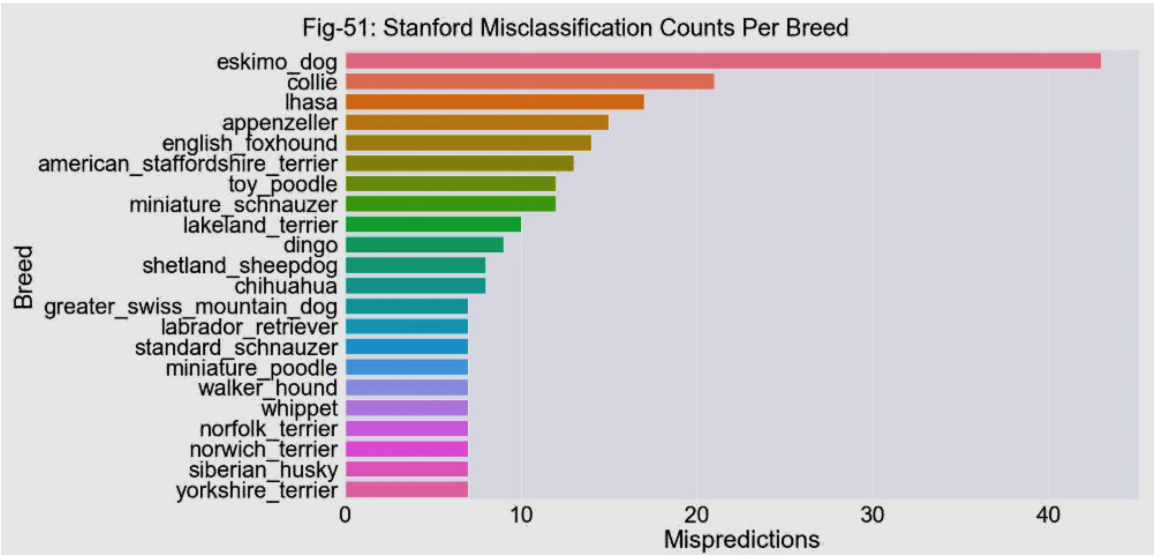
```
['im80', 'skip', 'skip'] 0.8725
['skip', 'xm50', 'skip'] 0.8716666666666667
['skip', 'xm80', 'xw80', 'skip'] 0.8708333333333333
['skip', 'xm80', 'xw50', 'skip'] 0.8675
['skip', 'xw80', 'skip'] 0.8666666666666667
['skip', 'xm80', 'skip'] 0.8613888888888889
['skip', 'xw50', 'skip'] 0.8608333333333333
['iw50', 'skip', 'skip'] 0.8597222222222223
['im50', 'skip', 'skip'] 0.8486111111111111
['skip', 'skip', 'skip'] 0.008333333333333333
```

Misclassification Analysis of Ensemble Models (starting with Stanford Data Set)

Given a 92% accuracy from the best model, it is a good practice to study what is happening with the remaining 8%. The Stanford validation and test sets have exactly 50 samples per breed. The combined mispredictions were combined into a file with 492 samples. From the fig-50 histogram, error counts from 1 to 7 per breed dominate with a slightly curious rise for 6 and 7. There are 7 breeds with error counts in the teens, an outlier with 21, and an even nastier one with 42 mispredictions.



When breeds with more than 6 errors were split off onto a Seaborn bar plot (fig-51), to find the problem breeds.



The top offenders were examined. From the fig-52 Seaborn countplot, the ensemble model often predicts an Eskimo Dog to be a Siberian Husky or Malamute, which was predicted back in fig-8. From fig-53, Eskimo Dogs come with both black or brown fur and the model is probably mispredicting the blackish ones. There are no bulk mispredictions in the other directions (towards Eskimo Dog).

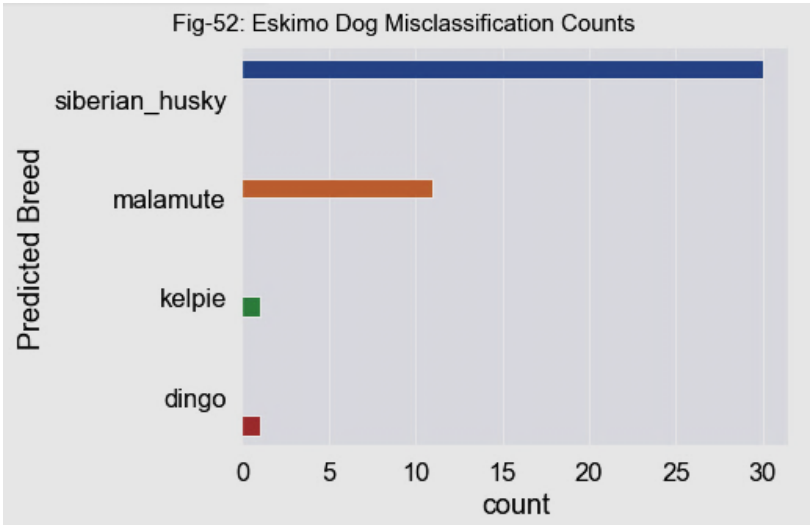
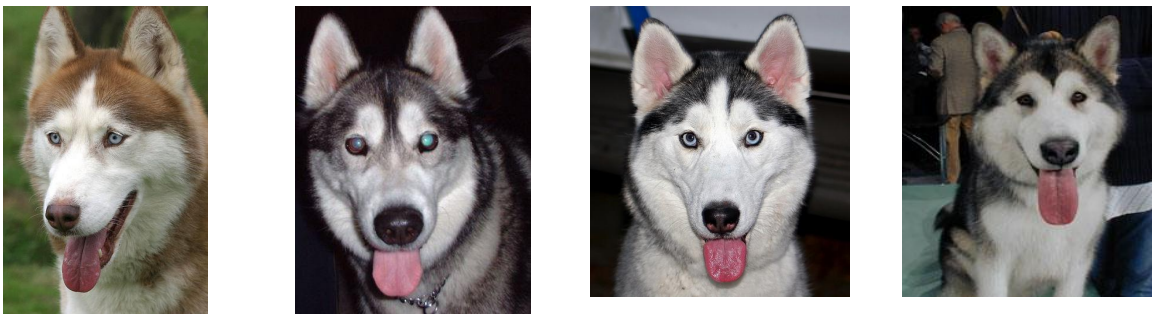
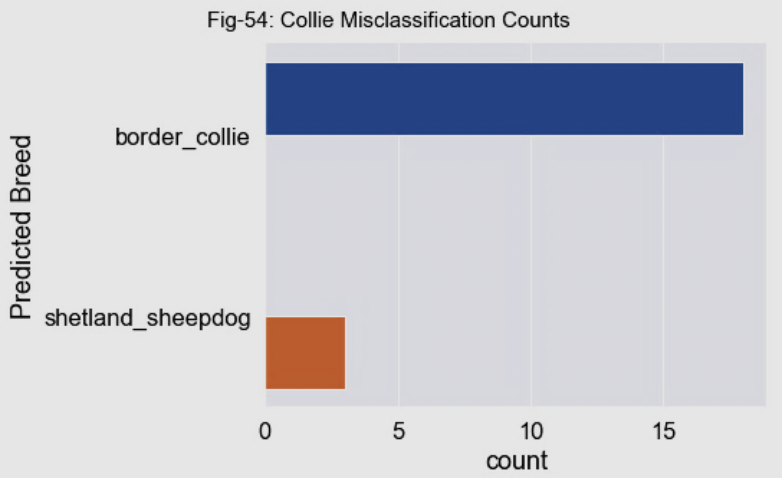


Fig 53(a-d): (brownish) Eskimo Dog, (blackish) Eskimo D., Malamute, Siberian Husky



The next worst performer is the Collie breed with 22 mis-predictions. When graphed in fig-54, the root cause is a similar appearance to Border Collies with a slight secondary effect due to Shetland Sheepdogs.



Similar to the last example, Collies come with both black and brown coloring, and even the nose structure looks slightly different (fig-55a & b). All of this causes the model to confuse them with other breeds (fig-55c & d). Border Collies could possibly be considered a sub-breed. As previously mentioned, size differences do not help, because the CNNs have no scale to go by.

Fig 55(a-d): (blackish) Collie, (brownish) Collie, Border Collie, Shetland Sheepdog

The Lhasa breed had 17 misclassifications out of 50 samples (fig-56). Shih-tzu was the most common mistake where the ensemble confused their mop-like faces 8 times (fig-57a & b). Tibetan Terriers and Dandie Dinmonts had the same issue to a lesser extent (fig-57c & d). Maltese Dogs (not displayed) had the same trend.

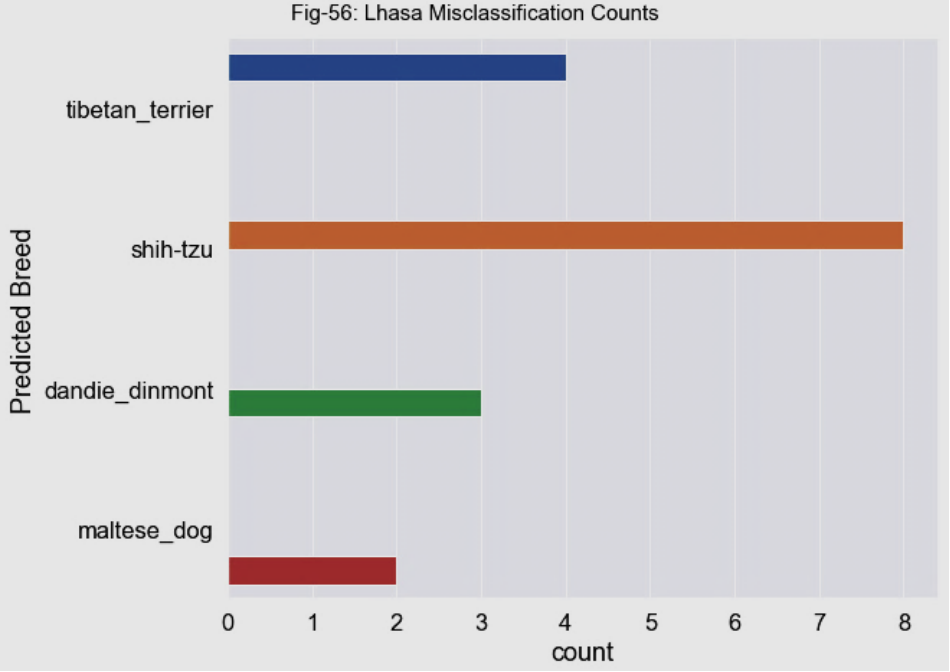


Figure 57(a-d): Lhasa, Shih-tzu, Tibetan Terrier, Dandie Dinmont

Appenzeller dogs are the fourth most difficult to classify. Fig-58 indicates that Entlebucher is the most common failure mode. From fig-59a & b, the two breeds look identical, possibly beyond the ability of humans to visually differentiate. Bernese Mountain Dogs (fig-59c) also appears similar which caused two misclassifications. Notice that all three have a center white facial strip and brown half-eyebrows. Rottweilers appeared only once (fig-59d) due to a lack of white fur even though the black and brown patterns somewhat match.

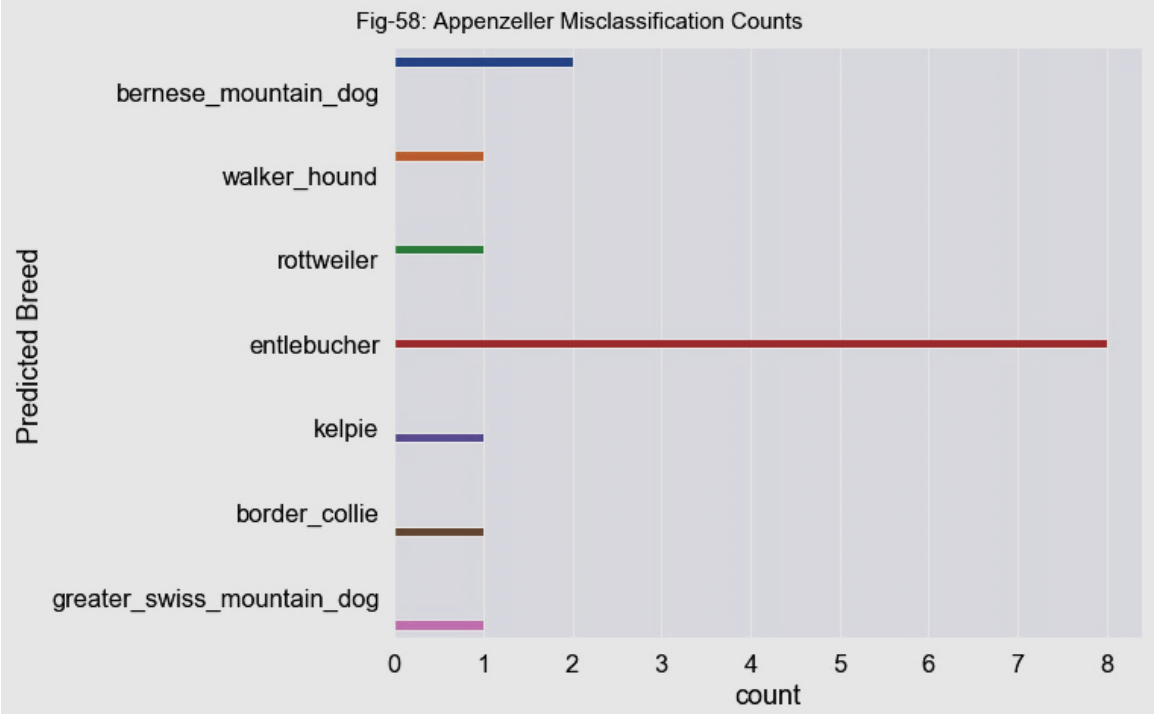
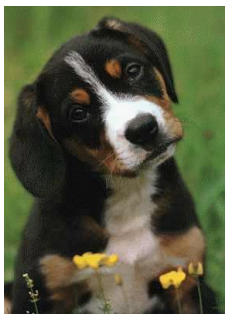
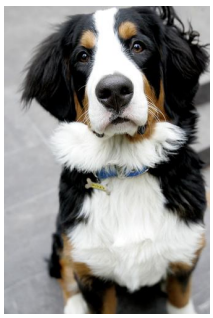
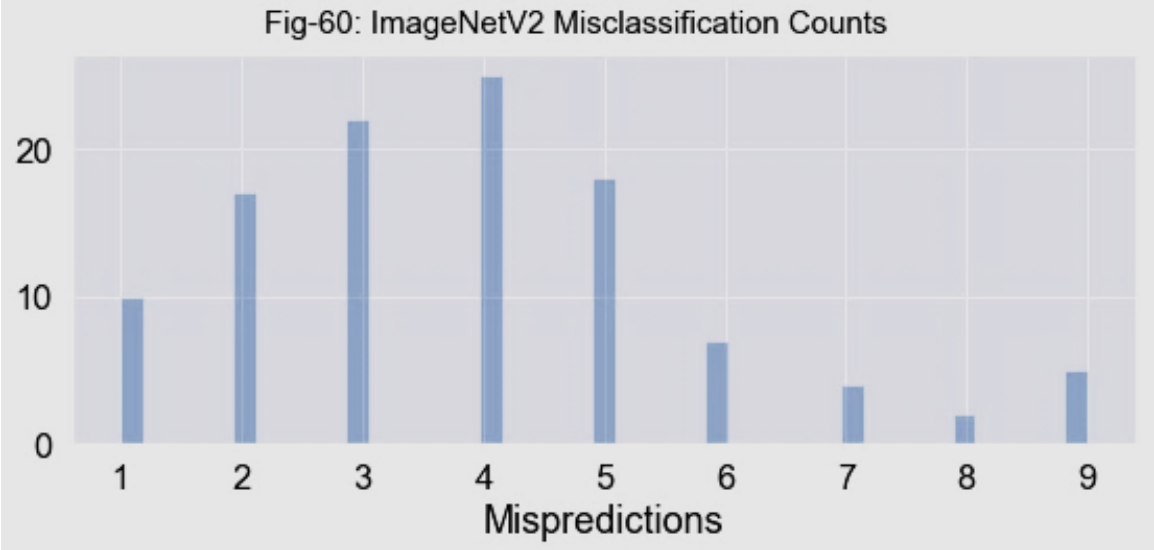


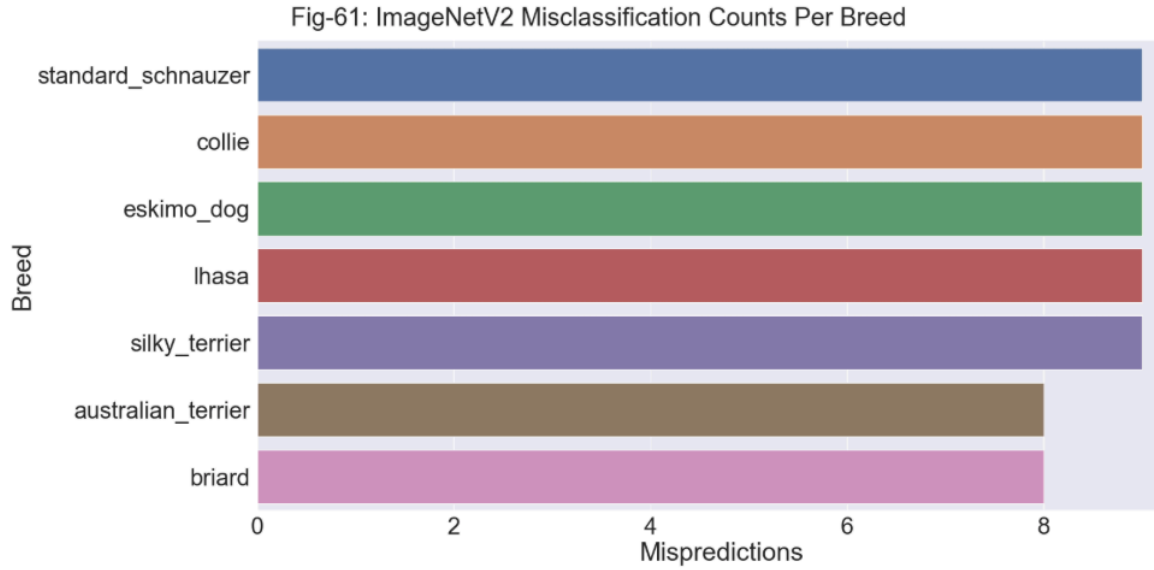
Figure 59(a-d): Appenzeller, Entlebucher, Bernese Mountain Dog, Rottweiler

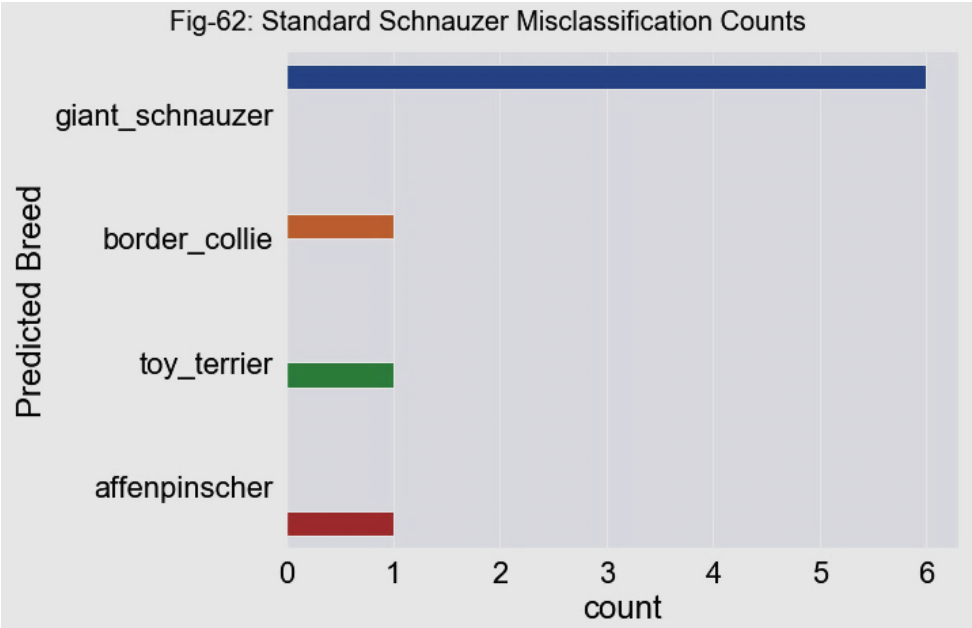

ImageNetV2 Dataset

The same steps were repeat on the *V2* dataset, which had much lower accuracy than *Stanford*. Fig-60 is a histogram of the failure count per breed. These results are fairly grim because there were only 10 samples per breed. The five breeds with a count of 9 represent an almost complete failure of the model. Fig-61 displays the name of the 7 most problematic classes.





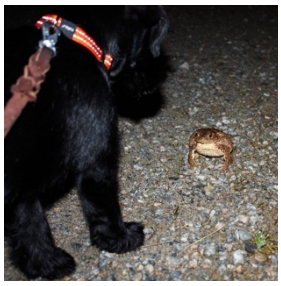
Standard Schnauzer errors were investigated. Fig-62 indicates that 60% of the time, the model mistakes the image for that of a Giant Schnauzer which is an issue not seen to any large extent with *Stanford* data (but fig-51 does show some slight evidence).



Since there are only 10 Standard Schnauzer samples in *V2*, they were visually compared to their counterparts in *Stanford*. Fig-56a & b are *Stanford* images that typically have good display qualities. In comparison, fig-56c & d show *V2* images that are basically adversarial examples. The first is two dogs playing with one occluding the other, while the second show a rearview of a dog confronting a frog.

Fig 56(a, b): Stanford Standard Schnauzer Images, (c, d): V2 Standard Schnauzer Images

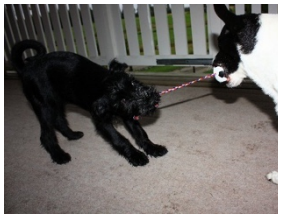


An occasional adversarial example is good for image detection, but *V2* is way overloaded with them. To make things clearer, fig-57 shows four more images from the same set of ten. There are two more examples of occlusion and two images of puppies where do not the characteristic "rounded beard" of the breed. This proves that at least 60% of the Standard Schnauzer images are adversarial which is entirely too many.

This along with similar *V2* issues noticed over time, caused a loss of confidence in the data quality, and further investigation into differences with *Stanford* data was halted.

Figure 57(a-d): *V2* Standard Schnauzer Images

Conclusion

This study examined various options for image recognition using modern CNN architectures. The best accuracy for each (determined by Top-1 Stanford Validation) is given in Table 11. Firstly, it was determined that a basic CNN trained from scratch would only perform in the low 30s.

Pretrained models from the Keras library (usually previous image competition winners) were tried next. It was found that image augmentation was less helpful on some models than others but was still used everywhere because it never hurt (except for tripling the training time). Fine-tuning raised accuracy sometimes and not others, but the top performer was kept available by saving models and best weights from both conditions. This is a good practice anyway for model ensembles and recovering from cloud-based training crashes.

An older architecture, VGG19 reached the mid 70s, almost twice the basic CNN it resembled. ResNet50 and DenseNet both have signal paths for skipping model layers for easier/better training. They peaked in the low 80s accuracy range.

The Inception family of models is based on going wide instead of deep to avoid vanishing gradients and signal degradations. They make extensive use of larger filter sizes, 1x1 convolution blocks, and auxiliary classifiers. InceptionV3 and Xception both performed in the mid-upper 80s, while the hybrid Inception-ResNet reached 90%. Every possible ensemble of the three was tried with software looping and reached the highest Top-1 accuracy of the study, 92.5 %.

Table 11: Top Accuracies of Various CNN Architectures

	Top-1 Accuracy			Top-5 Accuracy		
Model	Stanford Valid	Stanford Test	V2 Test	Stanford Valid	Stanford Test	V2 Test
Basic CNN	38.1	37.6	18.9	70.2	69.8	44.3
VGG19	75.5	74.8	46.1	95.1	94.9	73.2
ResNet50	81.1	79.7	50.7	97.4	97.1	74.8
DenseNet	81.3	81.0	52.9	97.4	97.0	77.3
InceptionV3	87.1	87.6	56.2	99.1	98.7	82.2
Xception	86.6	86.9	56.3	98.2	98.5	79.2
Inception-ResNetV2	90.1	99.2	62.1	99.2	99.2	84.9
Ensemble	92.5	92.3	64.4	99.7	99.7	85.3

This thesis admittedly focuses heavily on the Stanford dog breed dataset versus the other two. The Flowers dataset was a comparison point because there are no flower classes embedded in the pretrained weights. It has only 5 classes with around 600 samples each, so it complements the Stanford data which has 120 classes, but around 150-200 samples each. *Flowers* would train to accuracies in the 70s or 80s. It was intended as a sanity check in case the Stanford data did something strange.

ImageNetV2 turned out to be a disappointment. The accompanying paper claims it determines overfit issues due to ImageNet weights because those samples are also in CIFAR and many other datasets. However, the *V2* images were too polluted with adversarial examples which completely disrupted the comparison.

Several conclusions can be drawn from this study. Mainly that a good data scientist should not be too attached to any one dataset, model architecture, or training

point. When a new model will not train, the user will usually debug by simplifying it. However, some datasets behave better than others, and the simpler data of a "known-good" one can also be just as helpful as a simpler model. For example, the 8% ensemble misclassification rate was found to be dominated by image classes so similar, even a human would have trouble differentiating them.

Model architectures are always be outperformed by newer ones and relative performance can vary with data, so the more models tried the better. As well, saving the model and best weights during training is incredible useful for three reasons:

1. Quicker recovery from training crashes, especially cloud connections
2. The best performer can be kept. More epochs are not always better.
3. More available components for model ensembles. A model can even form an ensemble with itself.

Future "Blue Sky" Research

Currently there is a growing interest in unsupervised image detection. None of the unofficial Contrastive Predictive Coding code examples online worked, but there is a similar technique from Microsoft called Deep InfoMax with code released by the author.

In addition, some of the fine-tuning attempts that did not improve performance should be retried with a variety of different learning rates and momentum values. A properly decaying learning schedule takes effort to determine. However, each full training session for the Stanford data took at least six hours (longer if it crashed during sleeping hours). So, time and Google Cloud Platform GPU fees become restrictive.

References

note – identical references differentiated with *

LeCun, Yann; e.a. (2015), Deep Learning. Retrieved from https://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf

Bengio, Yoshua. (2012), Deep Learning of Representations for Unsupervised and Transfer Learning. Retrieved from http://proceedings.mlr.press/v27/bengio12a/bengio12a.pdf

Pratt, Lorien, e.a. (1991), Direct Transfer of Learned Information Among Neural Networks. Retrieved from https://www.aaai.org/Papers/AAAI/1991/AAAI91-091.pdf

Bengio*, Yoshua (2012), Deep Learning of Representations for Unsupervised and Transfer Learning. Retrieved from http://proceedings.mlr.press/v27/bengio12a/bengio12a.pdf

Yosinki, Jason, e.a. (2014), How transferable are features in deep neural networks? Retrieved from https://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf

Huh, Minyoung, e.a. (2016) What makes ImageNet Good for Transfer Learning. Retrieved from https://arxiv.org/pdf/1608.08614.pdf

Kornblith, Simon, e.a. (2019) Do Better ImageNet Models Transfer Better? Retrieved from https://arxiv.org/pdf/1805.08974.pdf

Hinton, Geoffrey, e.a. (1986) “Learning Internal Representation by Error Propagation” a chapter in Parallel Distributed Processing Vol 1: Foundations. Retrieved from https://web.stanford.edu/class/psych209a/ReadingsByDate/02_06/PDPVolIChapter8.pdf

Mikolov, Tomas, e.a. (2013) Distributed Representations of Words and Phrases and their Compositionality. Retrieved from https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

Goodfellow, Ian, e.a. (2014) Generative Adversarial Nets. Retrieved from https://arxiv.org/pdf/1406.2661.pdf

Radford, Alec, e.a. (2016) Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Retrieved from https://arxiv.org/pdf/1511.06434.pdf

Oord, Aaron, e.a. (2019) Representation Learning with Contrastive Predictive Coding. Retrieved from https://arxiv.org/pdf/1807.03748.pdf

Henaff, Olivier, e.a. (2019) Data-Efficient Image Recognition with Contrastive Predictive Coding. Retrieved from https://arxiv.org/pdf/1905.09272v1.pdf

Jaeger, Manfred, (2013) Probabilistic Classifiers and the Concepts they Recognize. Retrieved from https://www.aaai.org/Papers/ICML/2003/ICML03-037.pdf

Hinton, Geoffrey, (2007) Learning Multiple Layers of Representation. Retrieved from http://www.csri.utoronto.ca/~hinton/absps/ticsdraft.pdf

Claeson, Marc, e.a. (2015) Hyperparameter Search in Machine Learning. Retrieved from https://arxiv.org/pdf/1502.02127.pdf

Larose, Daniel (2005), Data Mining Methods and Models. Wiley InterScience Books

Nielsen, Michael, (2015), Neural Networks and Deep Learning. Determination Press, Retrieved from http://neuralnetworksanddeeplearning.com

Minsky, Marvin, e.a. (1969), Perceptrons An Introduction to Computational Geometry. Massachusetts Institute of Technology

Allen, Kate (2015) How a Toronto professor's research revolutionized artificial intelligence. The Star (a Toronto newspaper). Retrieved from https://www.thestar.com/news/world/2015/04/17/how-a-toronto-professors-research-revolutionized-artificial-intelligence.html

Krogh, Anders, e.a. (1992), A simple Weight Decay Can Improve Generalization. Retrieved from http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf

Ullrich, Karen, e.a. (2017), Soft Weight-Sharing for Neural Network Compression. Retrieved form https://arxiv.org/pdf/1702.04008.pdf

Srivastava, Nitish, e.a. (2014), Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15. Retrieved from http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf

Ioffe, Sergey, e.a. (2015), Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Retrieved from https://arxiv.org/pdf/1502.03167.pdf

Duchi, John, e.a. (2010), Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research 12. Retrieved from http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

Kingma, Diederik, e.a. (2015), Adam: A Method for Stochastic Optimization. Retrieved from https://arxiv.org/pdf/1412.6980.pdf

Springenberg, Jost Tobias, e.a. (2015), Striving for Simplicity: The All Convolutional Net. Retrieved from https://arxiv.org/pdf/1412.6806.pdf

Krizhevsky, Alex, e.a. (2012), ImageNet Classification with Deep Convolutional Neural Networks. Retrieved from https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Roelofs, Rebecca, e.a. (2019), Do ImageNet Classifiers Generalize to ImageNet? Retrieved from http://people.csail.mit.edu/ludwigs/papers/imagenet.pdf

Simonyan, Karen, e.a. (2014), Very Deep Convolutional Networks for Large-Scale Image Recognition. Retrieved from https://arxiv.org/pdf/1409.1556.pdf

He, Kaiming, e.a. (2015), Deep Residual Learning for Image Recognition. Retrieved from https://arxiv.org/pdf/1512.03385.pdf

Huang, Gao, e.a. (2018), Densely Connected Convolutional Networks. Retrieved from https://arxiv.org/pdf/1608.06993.pdf

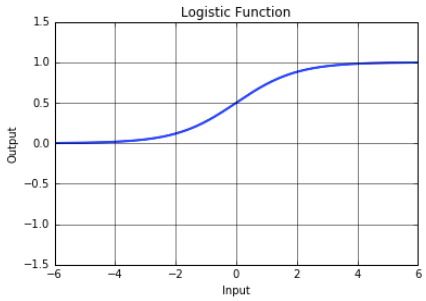
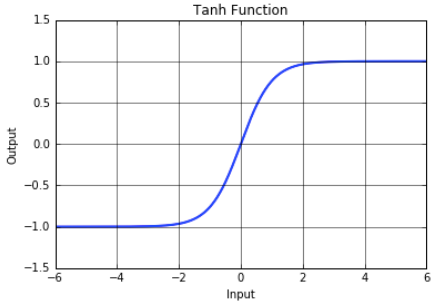
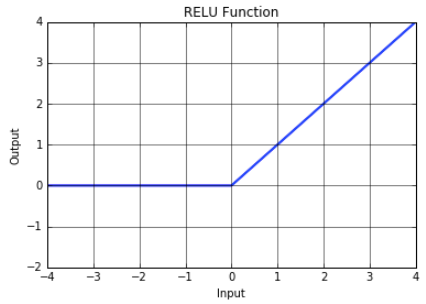
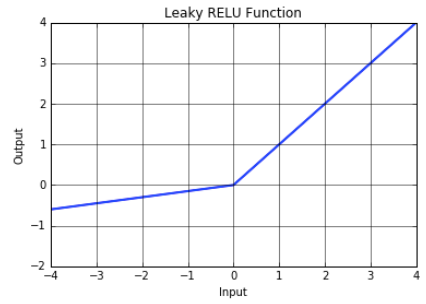
Chollet Francois, (2017), Xception: Deep Learning with Depthwise Separable Convolutions. Retrieved from https://arxiv.org/pdf/1610.02357.pdf

Szegedy, Christian, e.a. (2016), Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. Retrieved from https://arxiv.org/pdf/1602.07261.pdf

Paul Rahul, e.a. (2018), Predicting Nodule Malignancy using a CNN Ensemble Approach. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6233309/

Appendix

Appendix A1: Activation Function Table

	$Y = \frac{1}{1 + e^{-x}}$
	$Y' = Y * (1 - Y)$
	$Y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
	$Y' = 1 - Y^2$
	$Y = x * (x > 0)$
	$Y' = 1 * (x > 0)$
	$Y = x * \big(0.15 * (x < 0) + (x > 0)\big)$
	$Y' = 0.15 * (x < 0) + 1 * (x > 0)$

Appendix A2: Simple Python Neural Network Code

```
# Define input and output data for XOR function
x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

input_layer_size = 2
hidden_layer_size = 4
output_layer_size = 1

# Randomly initialize 2-D weight matrices
np.random.seed(2)
h_weights = np.random.randn(input_layer_size, hidden_layer_size)
y_weights = np.random.randn(hidden_layer_size, output_layer_size)

for t in range(2001):
    # Predict Y with forward propagation
    h_linear = x.dot(h_weights)         # weighted sum of two inputs
    h_relu = np.maximum(h_linear, 0)    # ReLU activation function
    y_linear = h_relu.dot(y_weights)    # weighted sum of 4 hidden layer neurons
    y_pred = y_linear                   # no activation function, passing thru

    # Compute and print loss
    loss = 0.5 * np.square(y - y_pred).sum()
    relu_pretty_print(t, loss, y_pred)

    # Back-propagate to find gradients of h and y weights wrt loss
    y_delta = (y_pred - y) * 1.0  # loss derivative * pass-thru derivative
    y_weight_gradient = h_relu.T.dot(y_delta)

    h_delta = y_delta.dot(y_weights.T) * 1.0  # ReLU derivative is either 1 or 0
    h_delta[h_linear < 0] = 0
    h_weight_gradient = x.T.dot(h_delta)

    # Update weights with product of learning rate and gradient
    h_weights -= 0.002 * h_weight_gradient
    y_weights -= 0.002 * y_weight_gradient
```

Appendix A3: Dataset Links

Flowers

https://www.kaggle.com/alxmamaev/flowers-recognition

Stanford Dog Breeds

http://vision.stanford.edu/aditya86/ImageNetDogs

ImageNet V2

https://github.com/modestyachts/ImageNetV2

Full list of ImageNet 1000 classes (for informational purpose)

https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a

Appendix A4: Content and Size of Stanford Dog Breed Dataset

breed_1	n_samples_1	breed_2	n_samples_2	breed_3	n_samples_3
affenpinscher	150	bouvier_des_flandres	150	english_setter	161
afghan_hound	239	boxer	151	english_springer	159
african_hunting_dog	169	brabancon_griffon	153	entlebucher	202
airedale	202	briard	152	eskimo_dog	150
american_staffordshire_terrier	164	brittany_spaniel	152	flat-coated_retriever	152
appenzeller	151	bull_mastiff	156	french_bulldog	159
australian_terrier	196	cairn	197	german_shepherd	152
basenji	209	cardigan	155	german_short-haired_pointer	152
basset	175	chesapeake_bay_retriever	167	giant_schnauzer	157
beagle	195	chihuahua	152	golden_retriever	150
bedlington_terrier	182	chow	196	gordon_setter	153
bernese_mountain_dog	218	clumber	150	great_dane	156
black-and-tan_coonhound	159	cocker_spaniel	159	great_pyrenees	213
blenheim_spaniel	188	collie	153	greater_swiss_mountain_dog	168
bloodhound	187	curly-coated_retriever	151	groenendael	150
bluetick	171	dandie_dinmont	180	ibizan_hound	188
border_collie	150	dhole	150	irish_setter	155
border_terrier	172	dingo	156	irish_terrier	169
borzoi	151	doberman	150	irish_water_spaniel	150
boston_bull	182	english_foxhound	157	irish_wolfhound	218

breed_1	n_samples_1	breed_2	n_samples_2	breed_3	n_samples_3
italian_greyhound	182	norwegian_elkhound	196	shih-tzu	214
japanese_spaniel	185	norwich_terrier	185	siberian_husky	192
keeshond	158	old_english_sheepdog	169	silky_terrier	183
kelpie	153	otterhound	151	soft-coated_wheaten_terrier	156
kerry_blue_terrier	179	papillon	196	staffordshire_bullterrier	155
komondor	154	pekinese	149	standard_poodle	159
kuvasz	150	pembroke	181	standard_schnauzer	155
labrador_retriever	171	pomeranian	219	sussex_spaniel	151
lakeland_terrier	197	pug	200	tibetan_mastiff	152
leonberg	210	redbone	148	tibetan_terrier	206
lhasa	186	rhodesian_ridgeback	172	toy_poodle	151
malamute	178	rottweiler	152	toy_terrier	172
malinois	150	saint_bernard	170	vizsla	154
maltese_dog	252	saluki	200	walker_hound	153
mexican_hairless	155	samoyed	218	weimaraner	160
miniature_pinscher	184	schipperke	154	welsh_springer_spaniel	150
miniature_poodle	155	scotch_terrier	158	west_highland_white_terrier	169
miniature_schnauzer	154	scottish_deerhound	232	whippet	187
newfoundland	195	sealyham_terrier	202	wire-haired_fox_terrier	157
norfolk_terrier	172	shetland_sheepdog	157	yorkshire_terrier	164

Appendix A5: Diagram of VGG19 Architecture