Neuro 140 Final Project Report

Spring 2020

Harvard University MAYA BURHANPURKAR

Contents

1 Project Overview and Methodology

In this project, we will explore a popular technique for classification and generation of image data, convolutional neural networks and variational autoencoders, that build upon the original artificial neural network paradigm. In this report, we will summarize the theory behind each technique and we will demonstrate them to the Quick, Draw! dataset. The goal of this project is to use examples to develop a first-principles understanding of these techniques by way of examples. To be clear, my goals were to

1. Modify a pre-trained CNN to perform classification on Quick, Draw! images
2. Compare performance to a from-scratch CNN made from first principles
3. Perform data generation using a CVAE and evaluate using the CNN from step 1

The code for this project can be found at this Github link. See Appendix A for a tour of the code.

1.1 Preliminaries

The Quick, Draw! dataset is one of the world's largest datasets, consisting of over 50 million sketches that are 28×28 pixels of over 300 different image categories. Users were prompted to make a drawing of a particular object class and were given only 20 seconds to do so. There are a variety of formats the data is available in, including time-series data for use in RNN training. I elected to use the `.npy` format that contains a static image of the final drawings submitted.

Due to the size of the dataset, I created a Mini Quick, Draw! dataset, consisting of only a subset of the total categories. I kept 10,000 images per category and included only Eyeglasses, Cup, Moon, Penguin, Canoe, Apple, and Lollipop. Representative sketches from each category are shown in Figure 1. It should be noted that there are often several sets of representations per category (e.g. some people draw the moon as a crescent, others draw it as a circle with dots for craters), which is part of what makes image recognition on this dataset difficult. Image recognition is further complicated by the inadequacy of many drawings. The dataset is by no means "clean" and consists many scribbles or inaccurate drawings that were created by malicious users.

Figure 1: Representative images taken from the Mini Quick, Draw! dataset.

In this project, I used transfer learning to train CNNs to recognize sketches from these seven categories, I built my own CNNs from scratch to go through the fine-tuning experience myself, and I trained several VAEs with varying degrees of success to generate images from these categories.

Initially, I wanted to work on Google Collab, but I found this was fraught with difficulties. Notably, I elected to use Tensorflow, for which the pre-built dataloaders require that each training example get its own file. This meant splitting the initial seven `.npy` files provided by Google that contain all samples in a given category into 70,000 separate `.npz` files. This process was straightforward and surprisingly quick to do on my own laptop (the script for doing this is contained in the notebook `Quick, Draw! Introduction`), but proved problematic for use on the Collab. Specifically, the

Collab has difficulties mounting Google Drive accounts containing a large number of folders, and it would only successfully load my files after re-loading the Collab at least five times, after which it had enough reloads to buffer my entire Drive folder.

For this reason, I chose to complete my project entirely locally, which proved sufficient for the purpose of training CNNs, but resulted in a bottleneck for training more intensive networks like CVAEs.

1.1.1 Data Preparation

My initial efforts with data preparation can be found in the notebook `Quick, Draw! Introduction`. First, all data were loaded, then seven categories were selected for Mini Quick, Draw! Then, each image stored in `x.npy` was converted into its own file `x_i.npz` in which `x` is the category label and `i` is the file index. The data were split into `train/` and `test/` directories. After then re-loading the data while dividing into the train and test arrays, the data were upsampled (Figure 2) as needed, reshaped to have three channel axes, and centered on the interval $[-1, 1]$. Finally, the numpy arrays were converted into data tensors with the data and the labels (which were binarized via the one-hot encoding process) stored as pairs.

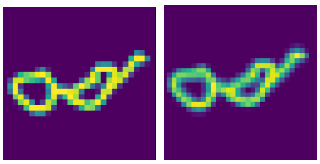

Figure 2: An original image (left) and an upsampled image (right).

While this process was done in a Jupyter notebook in `Quick, Draw! Introduction`, I found that I needed to repeat these steps in several parts of this project. As such, I wrote a script to do automatic data-loading for me, which is found in `load_data.py`. This script contains methods for automatic shuffling of data-label pairs and data binarization, rescaling, random sampling, and reshaping.

1.1.2 Generic Tricks

There are several "tricks of the trade" that are employed throughout this project. They are summarized below. Note that if any of these methods are not found explicitly in the notebooks presented, they were eliminated during the fine-tuning process as unhelpful.

Initialization

CNNs are known to suffer from the problem of vanishing gradients, and picking particlar distributions of weights to initialize networks has been shown to help. Initially, I initialized all hand-made networks for this project using Xavier initialization, but I found that this did not affect performance.

Early Stopping

Once the training loss plateaus, continued training can cause the validation loss to start increasing. The library `keras` has callback functions that monitor the validation loss and save the weights of the best performing model. If degradation occurs for a specified number of epochs, training is terminated and the best model is restored.

Model	Train Acc.	Val. Acc.	Test Acc.
VGG19	0.9836	0.9827	0.9830
MobileNetV2	0.9824	0.9070	0.9055
ResNet152V2	0.9115	0.8590	0.8592

Table 1: Final transfer learning training, validation, and testing accuracies for pre-trained CNNs.

Learning Rate Scheduling

After the loss function has plateaued, the model can often continue to be optimized with a smaller learning rate. Conceptually, a large learning rate will enable us to find the approximate minimum of the loss but we can get closer to the true minimum by taking smaller steps, preventing us from "bouncing around" the minimum as would happen with a larger learning rate. The library `keras` contains callback functions for scheduling the learning using the loss function.

Dropout

Dropout involves setting random nodes in the dense layers in a CNN to zero for a particular batch. The fraction of nodes affected is a hyperparameter that can be tuned. This might sound like an odd technique for regularization, but it has a cute theoretical justification that can be found in [2]. The argument is that dropout is an exponentially more efficient form of bagging. Instead of generating an ensemble of networks and averaging their results, dropout allows us to train multiple subnetworks at once that work in concert once the training phase is over. Of course, the subnetworks generated via dropout are not all independent as is the case in true bagging, but dropout provides an efficient approximation.

2 Results

2.1 CNN from Transfer Learning

Before attempting to train CNNs on my own, I gained intuition for expected performance by using transfer learning on pre-trained CNNs to perform classification on my dataset. Tensorflow provides a tutorial for doing so on Google's MobileNetV2, which I followed more or less directly (making the necessary modifications for my own dataset, of course) in the notebook `Transfer Learning on Quick, Draw!`. To see how other base models compare, I created a function `transfer_learn(...)` that performs transfer learning automatically given a particular base model and some parameters. The application of this function to the pre-trained VGG model and ResNet can be found in the notebook `VGG and Resnet Transfer Learning`. The learning rate was manually fine-tuned (not shown in the notebook) and learning rate scheduling was applied. The results of transfer learning on these three models are summarized in Table 1.

Although all three models were pre-trained on the ImageNet datasetit is clear that VGG performs the best. It is also illuminating to review the training history of the three models, shown in Figure 3 (the full training history, including training accuracies and losses, can be found in the source notebook).

From Figure 3, it is clear that fine-tuning had little effect on the models. This is likely because the high-level features found in the ImageNet dataset are comparable to the high-level features were expect to be found in the Quick, Draw! dataset. If anything, the features in the Quick, Draw!

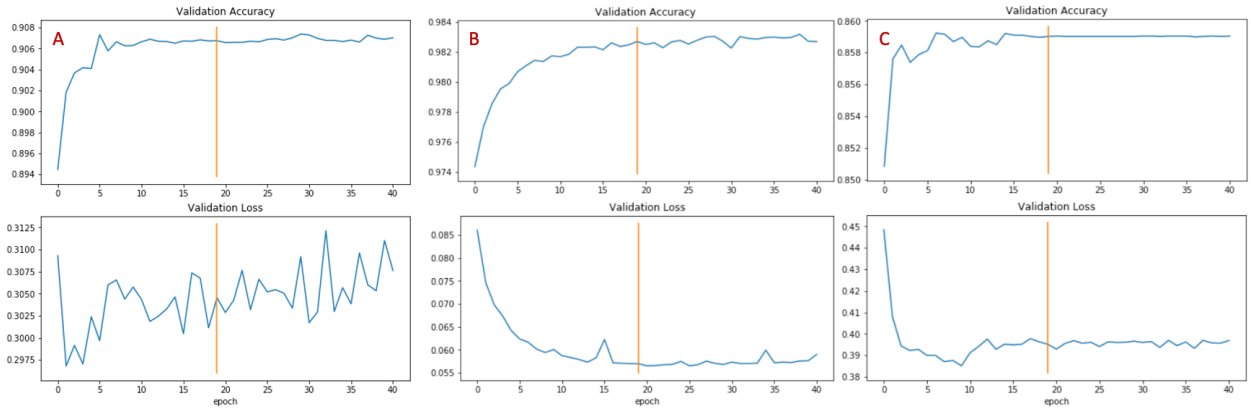

Figure 3: Validation accuracy and loss for A: MobileNetV2, B: VGG19, and C: ResNet152V2. All models had their top layers trained for 20 epochs and their top 100 layers were subsequently fine-tuned for 20 epochs. The orange vertical line marks the start of fine tuning.

dataset should be more straightforward than in ImageNet, so as expected, the models need only be briefly trained to re-purpose their outputs.

The varying performance between the three models can likely be attributed to their structures. MobileNet was created by Google to be lightweight and efficient. The key difference between standard CNNs and the MobileNet is that the MobileNet performs depthwise and pointwise convolutions [4]. That is, a standard convolution is factored into two smaller (and more efficient) operations. It has two hyper parameters which control the accuracy of the network. For my test, both hyper parameters were set at their maximum possible values. The ResNet AlexNet at LSVRC2015 made residual networks famous. In principle, this network too should be highly effective at classifying images. The key idea behind residual networks is the "identity shortcut connection" that allows samples to skip layers [3]. Finally, VGG is essentially a standard CNN, albeit with many more convolutional layers than a standard CNN [8].

While it is not clear to me why VGG so dramatically outshines the other two networks, it is clear that the three networks function in very different ways. Sketches are paired-down representations of images, so it is reasonable that these networks, which were trained to perform well on images, may not necessarily perform well on sketches.

A final point that should be made regards model efficiency. While VGG performed the best, it was the most time-consuming to train. While each step of training for MobileNet and ResNet took around 750ms to complete, each step of training for VGG took 1s. Furthermore, MobileNet, ResNet, and VGG have $(8967, 2257984)$, $(14343, 58331648)$, and $(3591, 20024384)$ (trainable, non-trainable) parameters for the pre-fine-tuning training respectively. Lastly, both VGG and ResNet accept a smallest image size of 32×32 pixels, so the required upsampling is fairly minimal. For the MobileNet, however, the images had to be upsampled to 96×96 pixels, which added significantly to the total computation time.

2.2 CNN from "First Principles"

Given that sketches should, in theory, be simple for CNNs to recognize, given that feature extraction should primarily amount to edge detection, they seemed like ripe candidates for building CNNs from scratch to perform recognition on. I began my tests with the basic architecture shown in Figure

4 consisting of three convolutional layers with an increasing number of filters per layer. After 20 epochs of training, it was clear that the validation accuracy had plateaued, and training ceased with a test accuracy of 0.9589, also shown in Figure 4. This is better than both the MobileNet and ResNet, but not better than the VGG pre-trained model. The subsequent optimizations can be found in the notebook `Width and Depth Experiments on Quick, Draw!`.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_1 (Conv2D)	(None, 14, 14, 32)	4640
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 32)	0
conv2d_2 (Conv2D)	(None, 7, 7, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 3, 3, 64)	0
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 576)	332352
dense_1 (Dense)	(None, 128)	73856
dense_2 (Dense)	(None, 7)	903

Total params: 430,407
Trainable params: 430,407
Non-trainable params: 0

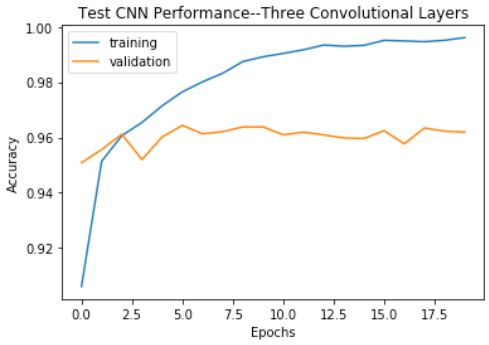

Figure 4: (Left) Initial architecture for width/depth experiments and fine-tuning and (right) corresponding validation and training accuracy on the base model.

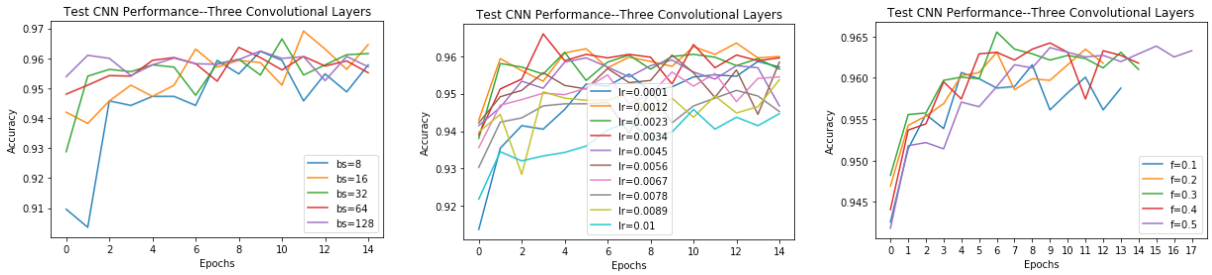

Figure 5: (Left) Batch size (center) learning rate (right) dropout rate optimization on the base model.

This was the starting point for subsequent optimizations, including

1. Batch size and learning rate fine-tuning: batch sizes in [8, 16, 32, 64, 128] and learning rates in `np.linspace(0.0001, 0.01, 10)` were tested with the Adam optimizer. The optimal learning rate was found to be 0.003 while the optimal batch size was found to be 16, as shown in Figure 5.

2. Width and depth fine-tuning: here, we compared the effectiveness of a variety of "flat" architectures. Nine models were tested with all combinations of `n_filters=[16, 32, 64]` and `n_conv_layers=[1, 2, 3]`. Interestingly, all models performed comparably—they all had enough parameters to describe the state space.

3. Dropout fine-tuning: dropout rates of [0.1, 0.2, 0.3, 0.4, 0.5] were tested, resulting in Figure 5. Ultimately, dropout was not used in the final model tested.

4. Learning rate scheduling: learning rate scheduling was applied to the optimal model found after the preceding optimizations using `tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, min_lr=1e-5, verbose=1)`. It was not found to significantly change outcomes on the test set.

After the aforementioned optimizations, and "optimal" from-scratch model was generated for comparison to the pre-trained VGG model.

2.3 Examples

The best from-scratch CNN had a test set accuracy of 96.5% while the best pre-trained CNN had a test set accuracy of 98.3%. These rates are remarkably comparable. It is illuminating to look at some examples of properly classified and misclassified data from both networks. In particular, we can plot the elements of the test set which (a) the CNN was most confident it was right about and got right and (b) the CNN was most confident it was right about and got wrong. Our criteria for (a) is sorting correctly classified samples by the predicted probability of their top prediction and our criteria for (b) is sorting incorrectly classified samples by the predicted probability of their top prediction. In both cases, the largest such elements are considered. The results are shown in Figures 6 and 7.

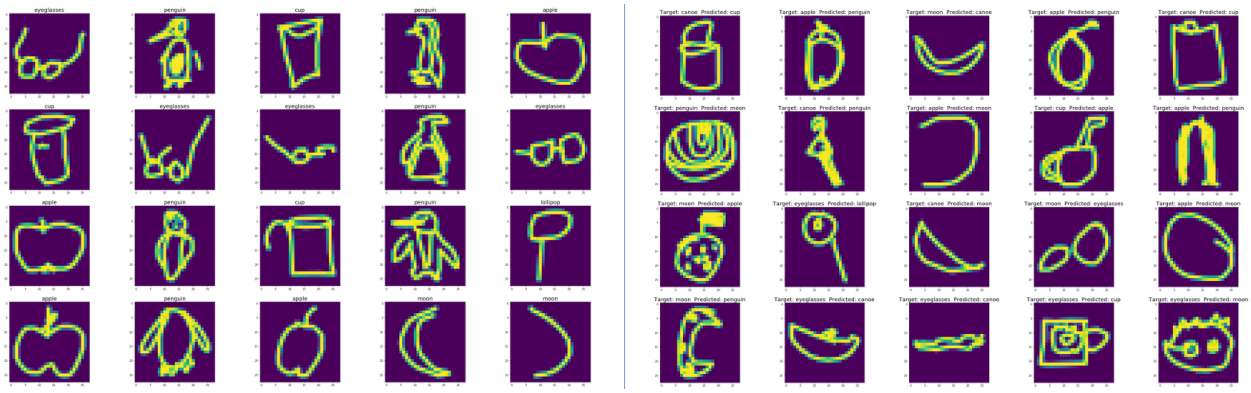

Figure 6: (Left) correctly classified images and (right) incorrectly classified images.

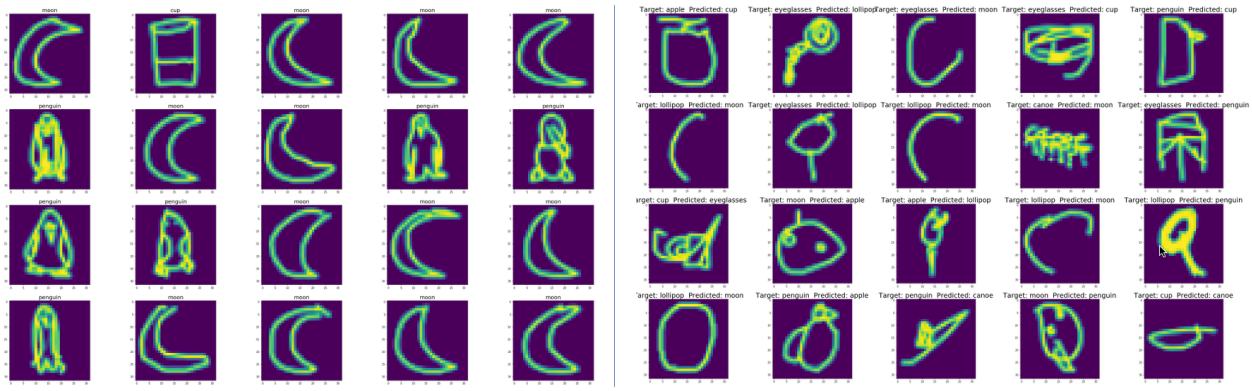

Figure 7: (Left) correctly classified images and (right) incorrectly classified images.

From these samples alone, no noteworthy patterns arise in correctly and incorrectly classified images. From the right columns in both figures, we can see that while some of the misclassifications are genuine errors, many misclassifications result from poor drawings. Many of the images particularly in Figure 7 are mere scribbles and could also not be correctly classified by a human, suggesting that there is some limit in test accuracy performance. In order to determine whether there are correlations in consistently misclassified images, the confusion matrices in Figure 8 are shown. Surprisingly, no obvious correlations (e.g. moons being misclassified as apples because

both are round) were found. Instead, there were a few less obvious correlations. For example, the hand-optimized model often mistook and cups for eyeglasses and penguins for moons. For both models, moons were often mistaken for eyeglasses.

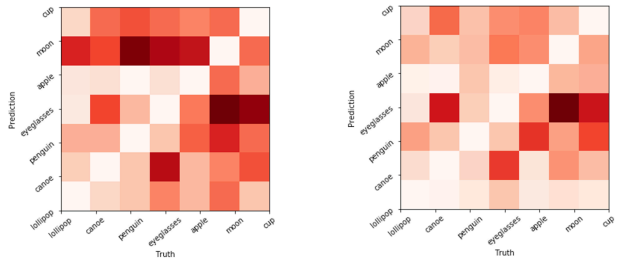

Figure 8: (Left) Best hand-optimized model (right) pre-trained VGG model.

Finally, the activations for my own CNN can be found in the notebook `Width and Depth Experiments on Quick, Draw!`. They are not particularly illuminating.

2.4 Variational Autoencoder Results

Because VAEs are more complicated to use than traditional CNNs, I based my code off the code found at this link, although I adapted it somewhat (particularly the visualization code) to fit my own needs. While the code I referred to trained their CVAE on the MNIST dataset, and it appeared to work reasonably well, the Quick, Draw! dataset is somewhat more complicated, and my results were met with mixed success. In particular, I found that I faced computational limitations due to training my networks locally—it was clear that performance improved with larger network size, but training quickly became infeasible with larger networks.

As a result, I tried only three network architectures:

1. Two convolutional layers with 5×5 kernel and 8 and 16 filters with a latent space of 14 dimensions.
2. Two convolutional layers with 5×5 kernel and 64 and 128 filters with a latent space of 14 dimensions.
3. Two convolutional layers with 5×5 kernel and 16 and 16 filters with a latent space of 2 dimensions.

The losses and validation losses for each are shown in Figure 9. From these plots (note the differing scales on the y-axes) we see that each network is close to being saturated in terms of what it can learn at the time the training was finished, in the sense that the loss functions appear to plateau. Perhaps with learning rate scheduling this could be improved, but by comparing the first and second networks at least, it is clear that increasing the depth of the network without increasing the width is not helping. Given more computational resources, I believe that a deeper or larger CVAE would help decrease the loss further.

Furthermore, a PCA was performed on the 14-dimensional spaces to project them into three dimensions. This resulted in Figure 10. From these, we can tell that in all three cases, the CVAE still has far to go in order to learn the distinctions between the seven image classes. The latent spaces are messy and not separated from each other, meaning that the CVAE has not learned an efficient representation. In theory, given 14 latent dimensions, the CVAE should have been able

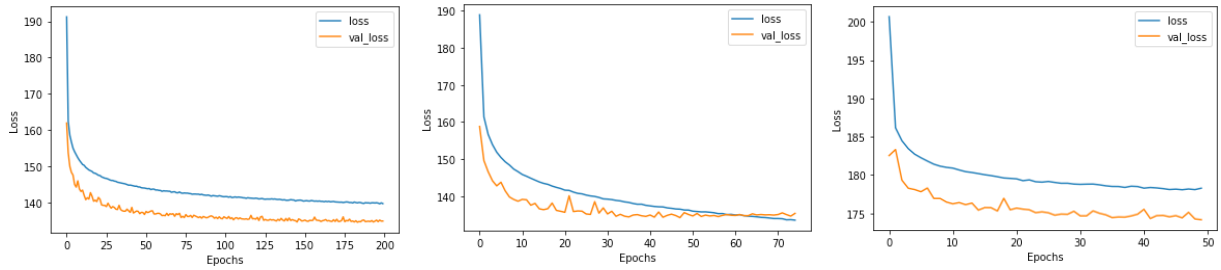

Figure 9: Loss functions for model 1, 2, and 3 respectively.

to associate one dimension of the multidimensional Gaussian with a particular category, but it did not learn this. Forcing the CVAE to be extra efficient by restricting it to two latent dimensions, however, did not appear to help much either.

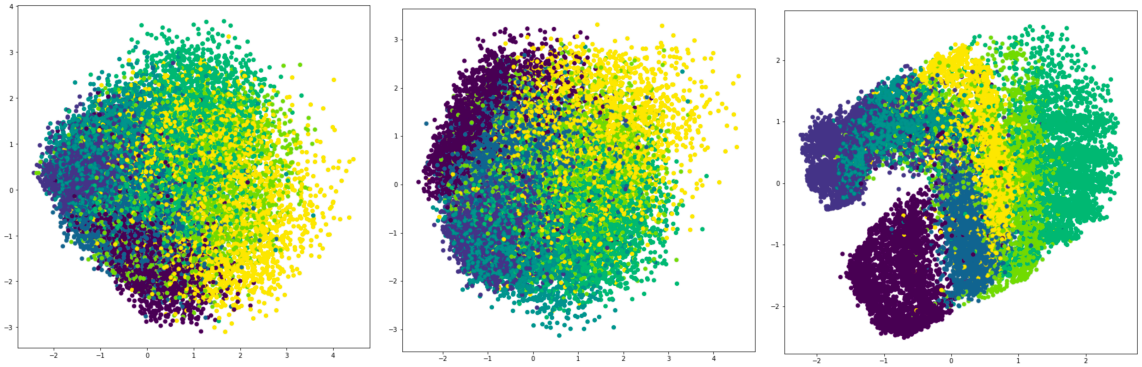

Figure 10: Latent spaces for model 1, 2, and 3 (spaces for model 1 and 2 are PCA projections).

Finally, for the two-dimensional latent space, the latent space was sampled in the usual manner, resulting in 11. Here we can see what the network had the most difficulty classifying. Representations of lollipops, apples, cups, and canoes can be clearly seen, while the network appears to have become confused between penguins and moons and the corners of the image (corresponding to samples far from the mean) are garbled. Training the model with two latent dimensions was surprisingly difficult–gradient clipping needed to be implemented in order to avoid exploding gradients.

With this said, a quick Google search reveals plenty of examples of people who have achieved comparable results to what I have found here (for example, this applet–see, in particular, the "lion" object category). Given the general tendency of CVAEs to "blur" out the original data, it may be necessary to switch to a GAN-like architecture to achieve better results.

3 Conclusions

The doodle recognition element of this project was highly successful, resulting in 98.4% and 96.5% test set success rates for the pre-trained and hand-made CNNs respectively. By comparison, a human volunteer was given a 200 image subset of the test set and was correctly able to identify all but six of the doodles (see `Load Images for Human.ipynb`).

The image generation element of this project was less successful, but this was likely due to limited

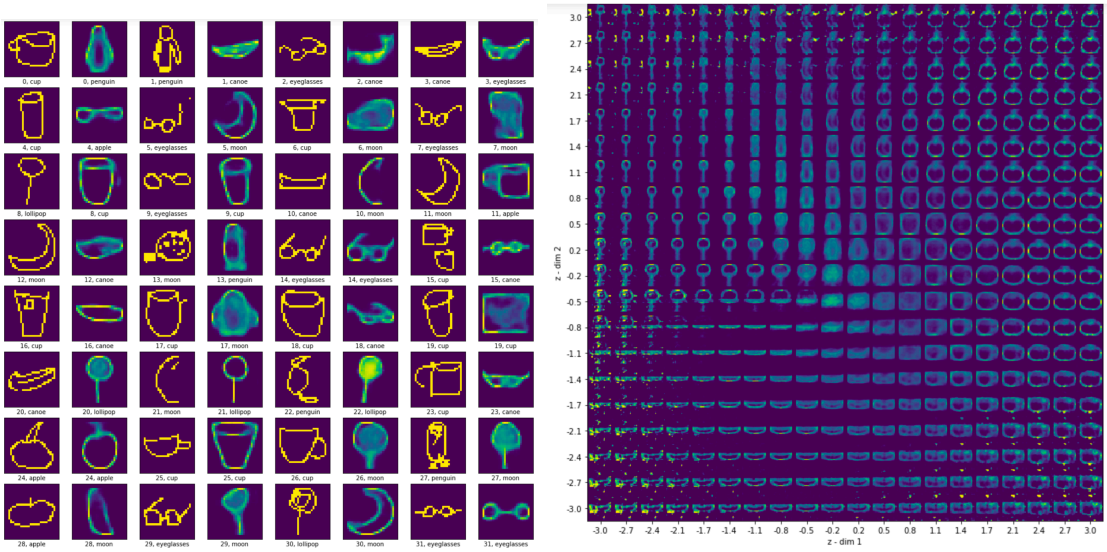

Figure 11: (Left) A sample of input images from the test set and corresponding output images from the decoder from model 1. The left image in a pair of images is a test doodle with a label while the right image in a pair is the generated image labeled with the class the VGG network thought it should belong to. Additional plots of this nature can be found in the source notebook. (Right) A comprehensive sample of the latent space of model 3.

computational power. The most successful CVAE generated images with input from the test set that could be correctly identified by the pre-trained CNN from the previous part of the project with only 69% accuracy. However, even though the CVAEs did not learn "good" encoding, in that the latent spaces for each object category overlapped, it still generated *recognizable* doodles. A volunteer was asked to look at 100 generated images and determine which of the seven object categories (if any) it belonged to (see `Load Images for Human.ipynb`). The volunteer was able to do this for all images, showing at least that the images generated by the CVAE were plausible.

Future extensions of this work could include experimenting with a GAN-style architecture to achieve more realistic doodles or running the training code on a cluster in order to evaluate CVAEs with grader width and depth.

A Guide to the Code

All files listed below can be found at this Github repository.

- `load_data.py`: this contains methods for loading the Quick, Draw! data and example usage can be found throughout the notebooks below. The function `load_for_cnn` allows for the files stored in `object_files/` stored as `object_category_i.npz` to be loaded into train, test, and validation sets, with optional shuffling, binarization, standardization, reshaping for multiple channel axes, and upsampling. There is also a hand-made class provided for data augmentation `ImageDataGenerator`. This was ultimately not used because data augmentation was proven to harm more than it helps (e.g. a flipped canoe can become a moon), but it took an awfully long time to write.
- `Quick, Draw! Introduction.ipynb`: initial data processing, later "factorized" into `load_`

`data.py`. Contains samples of images and labels that were instructive to examine

- `Transfer Learning on Quick, Draw!.ipynb`: contains the transfer learning tests using MobileNetV2 following the Tensorflow tutorial script. This was later "factorized" into a function in `VGG and Resnet Transfer Learning.ipynb`: self-explanatory. Contains a function that can be used with any model available in `tf.keras.applications`
- `Width and Depth Experiments on Quick, Draw!.ipynb`: this contains my optimization tests and visualization code for my hand-made CNNs. All of this code is my own with the exception of the activation visualization code (which was ultimately not included in this report), which was adapted from the code found at this link.
- `Quick, Draw! CVAE with Visualization.ipynb`: this contains all of the code I used to train and test my CVAEs. The code for the CVAEs was adapted from the code found at this link, but the visualization and testing code is entirely my own. NB: for some bizzare reason, the code for training the CVAE only works with Tensorflow 1.12 while the code for visualizing the CVAE only works with Tensorflow 2.1. I don't know why this is the case, but swapping out kernels mid-notebook was an easy enough hack.
- `Load Images for Humans`: this was used to compare model performance to human performance. The labels were covered and the subjects (my parents) were made to guess the labels.
- `graveyard/`: this contains tutorials and preliminary pieces of code not used for the final report.

I have also included my model files, with a straightforward naming convention. Thanks for reading this report!

B A Lightning Introduction to CNNs and VAEs

I wrote this when I was studying CNNs and CVAEs in detail because I find it useful to write down what I study to understand them better. This section is can be skipped.

B.1 Deep Feed-Forward Neural Networks

Conventional machine learning techniques, like PCA and logistic regression, are limited in the forms of functions they can estimate. Artificial neural networks (ANNs) present a solution. ANNs boil down to function compositions: each input is transformed by the same sequence of vector-valued functions to produce the desired output. Given a large number of training examples, the ANN can be trained to approximate the desired output. Each layer of the ANN consists of a linear transformation and an activation function. Specifically, a layer takes in N inputs, where N the number of outputs of the previous layer and returns N' outputs, where N is not necessarily equal to N'. Many such layers can be chained together such that the input, often having a very large number of dimensions, is gradually transformed into the output, which typically has a far smaller number of dimensions.

Rather than thinking of each layer as a vector valued function, we can instead think of each layer consisting of an array of scalar functions. Each node applies a linear transformation to the nodes of the previous layer and then applies a nonlinear function to the output. This results in a scalar that is passed to the next layer, with a new set of weights and biases comprising the next linear

transformation. This combination of linear transformations and nonlinear activations allows the ANN to approximate complicated, nonlinear functions.

We can then think of ANNs as consisting of an array of neurons (nodes), each of which takes in the outputs of the previous layer and generates a new scalar that gets passed on to the next layer of neurons. The number of input neurons must match the number of input dimensions and the number of output neurons must match the number of output dimensions, but the layers in between, called "hidden layers" can have many number of nodes. The "deep" in "deep feed-forward neural networks" refers to the usage of a large number of such hidden layers. Before introducing the main topic of this report, convolutional neural networks, we will explain the intuition behind choices of nonlinearities in ANNs as well as the basic training procedure that allows ANNs to learn.

B.1.1 The Cost Function

We can view the task of learning as an optimization problem. We start with a function that generates predictions $\hat{y}(x)$ from samples x which can be compared to some target output y. Learning can then be viewed as a process in which the approximating function is modified so as to minimize the distance (in the most general sense) between $\hat{y}$ and y. In the case of ANNs, an algorithm called stochastic gradient descent is typically used to optimize the cost function, which gives a measure of how far the output of the ANN $\hat{y}(x)$ approximates the desired output y. An obvious choice of cost function is the mean squared error, given by

$$J(\hat{y}, y) = \sum_j ||\hat{y} - y||^2$$

where j indexes output dimensions and we use the L^2 norm. The mean squared error is not an ideal choice of cost function for categorical data, however. If we assume our data is distributed according to $P(y|x) \sim \mathcal{N}(\mu, \sigma^2)$, then given iid samples, the negative log-likelihood of the batch will be mean squared error. For N iid samples,

$$\mathcal{L}(\widehat{\mathbf{y}}, \mathbf{y}) = \prod_{i=1}^{N} \sqrt{\frac{1}{2\pi\sigma^2}} e^{-\frac{(\hat{y}_i - y_i)^2}{2\sigma^2}} \implies -\log \mathcal{L} \propto \frac{1}{2} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

where $\widehat{\mathbf{y}}$ is a tensor of predictions while $\mathbf{y}$ is a tensor containing the corresponding true labels [2]. Since the prefactor to the negative log likelihood is irrelevant to a minimization problem, we can optimize the function producing $\hat{y}$ by minimizing the mean squared error. Therefore, the mean squared error is a reasonable cost function for normally distributed data. Binary variables are Bernoulli, so this is not a reasonable assumption. For categorical labels in particular, though, there is a better choice–the crossentropy.

For a random variable X the information contained in that random variable's distribution from information theory is given by $I(x) = -\log P(x)$. Intuitively, this makes sense–if x is highly likely then x occurring gives us little information (a trivial example is X representing the probability the sun rises tomorrow–telling me this does not give me any information, and $\log(1) = 0$). The Shannon entropy of a random variable is given by

$$H(X) = \mathbb{E}_{X \sim P}[I(x)] = -\mathbb{E}_{X \sim P}[\log P(x)]$$

which is also the familiar formula from statistical mechanics (up to a choice of logarithm base). Applying this to a binary random variable W, for example, we find

$$H(W) = -\sum_{i=1}^{2} P(x_i) \log P(x_i) = P(x_1) \log P(x_1) + (1 - P(x_1)) \log(1 - P(x_1))$$

Therefore, minimizing the negative log-likelihood is equivalent to minimizing the entropy [2]. Equivalently to our treatment of normally distributed sample, consider N iid samples from a categorical distribution. If the observed frequency of x_i is p_i while the true frequency is q_i, then

$$\mathcal{L}(\mathbf{p}, \mathbf{q}) = \prod_i q_i^{Np_i} \implies -\mathbb{E} \log \mathcal{L} = \sum_i p_i \log q_i \equiv H(\mathbf{p}, \mathbf{q})$$

(where we have taken an expectation to convert $N \to p_i$). This H defines the categorical crossentropy. In this project, we will therefore minimize the categorical crossentropy, which we call our cost function.

B.1.2 Stochastic Gradient Descent and Backpropagation

When I initially started reading about Stochastic gradient descent (SGD) and the backpropagation algorithm in [2], I was frustrated by the unnecessary abuse of notation that complicated algorithms I thought should be straightforward applications of calculus. The paired-down notation in [7] I found particularly helpful. I will summarize the arguments for each algorithm here.

Stochastic Gradient Descent

This algorithm is super straightforward in the language of calculus. As we know, our goal is to minimize $\mathcal{L}$ for all of the training examples (in the hopes that this also minimizes $\mathcal{L}$ for all possible inputs). In SGD, you pick a learning rate, ϵ, compute $\mathcal{L}$ over a training batch (hence the "stochastic" in SGD) and then add $-\epsilon \nabla \mathcal{L}$ to whatever your current state vector is. That's it!

This may have been somewhat abstract, so let's consider a concrete example. Let's say we have a network with three layers other than the input layer, and each layer has 3 neurons. For a fully connected network, this means that we have 3^9 weights and 9 biases to tune. For each training example in our minibatch, we can compute $\frac{\partial \mathcal{L}^i}{\partial w_{jk}^{(l)}}$ and $\frac{\partial \mathcal{L}^i}{\partial b_j^{(l)}}$ where i indexes the sample, j and k index neurons, and l indexes a particular layer using the backpropagation algorithm. Once we have these, we can regard our neural network as forming a $3^9 + 9$-dimensional vector space, with our loss function being a scalar field over this space. Because $\nabla \mathcal{L}$ gives us the direction of steepest ascent in this high-dimensional space, then $-\nabla \mathcal{L}$ gives us the direction of steepest descent. Taking our current state vector (that is, the vector containing all of our weights and biases for the entire neural network) and adding to it $-\epsilon \nabla \mathcal{L}$ will therefore reduce our loss function as efficiently as possible (assuming we have chosen ϵ well).

Backpropagation

Consider first a network with N non-input layers, but only one neuron per layer. There are then $N-1$ weights and $N-1$ biases to be tuned via SGD, so we want to compute $\frac{\partial \mathcal{L}}{\partial w_i}$ and $\frac{\partial \mathcal{L}}{\partial b_i}$. First, let's introduce some notation. Let the activation of the ith neuron be $a^{(i)}$, and let the desired value of $a^{(N)}$ be y. Let our loss function be $\mathcal{L}(y, a^{(N)})$. Some more definitions

$$\begin{aligned} z^{(i)} &= w^{(i)} a^{(i-1)} + b^{(i)} \\ a^{(i)} &= \sigma(z^{(i)}) \end{aligned}$$

where σ is our choice of activation function. Note that if there were multiple neurons per layer, then our expression for $z^{(i)}$ would simply contain more terms (which could be indexed by subscripts), but the overall form of the equation remains the same. Then,

$$\frac{\partial \mathcal{L}^0}{\partial w^{(N)}} = \frac{\partial z^{(N)}}{\partial w^{(N)}} \frac{\partial a^{(N)}}{\partial z^{(N)}} \frac{\partial \mathcal{L}^0}{\partial a^{(N)}} = a^{(N-1)} \sigma'(z^{(N)}) \frac{\partial \mathcal{L}^0}{\partial a^{(N)}}$$

Note that this is only for one particular training example (and the last node in our network at that). In general,

$$\frac{\partial \mathcal{L}}{\partial w^{(N)}} = \frac{1}{n}\sum_j \frac{\partial \mathcal{L}^j}{\partial w^{(N)}}$$

Similarly, for the bias term,

$$\frac{\partial \mathcal{L}^0}{\partial b^{(N)}} = \frac{\partial z^{(N)}}{\partial b^{(N)}}\frac{\partial a^{(N)}}{\partial z^{(N)}}\frac{\partial \mathcal{L}^0}{\partial a^{(N)}} = \sigma'(z^{(N)})\frac{\partial \mathcal{L}^0}{\partial a^{(N)}}$$

Now, to get the remainder of the partials for the gradient, we simply recurse (hence the name "back"proagation):

$$\frac{\partial \mathcal{L}^0}{\partial a^{(N-1)}} = \frac{\partial z^{(N)}}{\partial a^{(N-1)}}\frac{\partial a^{(N)}}{\partial z^{(N)}}\frac{\partial \mathcal{L}^0}{\partial a^{(N)}} = w^{(L)}\sigma'(z^{(N)})\frac{\partial \mathcal{L}^0}{\partial a^{(N)}}$$

Once we have $\frac{\partial \mathcal{L}}{\partial a^{(N-1)}}$, then we can compute $\frac{\partial \mathcal{L}}{\partial w^{(N-1)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(N-1)}}$ using the chain rule as we did before, and we simply repeat this process until we have computed all of the gradients for the entire network. Easy!

Now that we have all of the $\frac{\partial \mathcal{L}}{\partial w^{(i)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(i)}}$, how do we apply SGD? It's very simple–we have created a $2(N-1)$-dimensional vector space, so all of the $\frac{\partial \mathcal{L}}{\partial w^{(i)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(i)}}$ form the $2(N-1)$ components of the gradient vector. These quantities directly tell us which direction to step in to minimize the loss function.

Now, let's make contact with the annoying notation I had mentioned earlier. What if we have a network with more than one neuron per layer? We simply give all of our activations subscripts referring to which neuron they belong to, we give all baises similar subscripts, and we give all weights two subscripts, describing which neurons they connect. Now, for our $\mathcal{L}^i$ terms, we simply replace our single-term with a sum over the number of dimensions in y. All of the equations above transform straightforwardly

$$\begin{aligned}
\frac{\partial \mathcal{L}^0}{\partial w_{jk}^{(N)}} &= \frac{\partial z_j^{(N)}}{\partial w_{jk}^{(N)}}\frac{\partial a_j^{(N)}}{\partial z_j^{(N)}}\frac{\partial \mathcal{L}^0}{\partial a_j^{(N)}} \\
\frac{\partial \mathcal{L}^0}{\partial a_k^{(N-1)}} &= \sum_{j=0}^{n_L-1}\frac{\partial z_j^{(N)}}{\partial a_k^{(N-1)}}\frac{\partial a_j^{(N)}}{\partial z_j^{(N)}}\frac{\partial \mathcal{L}^0}{\partial a_j^{(N)}}
\end{aligned}$$

And that's it! Note that this algorithm is particularly efficient computationally, because starting from the "back" instead of trying to compute every partial derivative from the ground up means that we can leverage dynamic programming to avoid having to compute and exponential number of derivatives. This approach, of course, isn't particularly concerned with memory saving.

B.1.3 Other Methods of Optimization

If that's all there is to it, then what are the other choices of optimizers doing? Several alternatives to SGD implement improvements to the optimization process. There are "momentum"-based algorithms that incorporate information about the direction of previous steps to infer where the next step should be. There are also algorithms with adaptive learning rates, like AdaGrad, RMSProp, and Adam [2]. AdaGrad and RMSProp both use the size of previous steps to determine step sizes in the future. Adam, which is the optimizer I used throughout my project, builds on RMSProp by adding in momentum considerations [2].

B.1.4 Hidden and Output Units

As with any constituent function for an ANN, the hidden unit activations must be differentiable. However, common activation choices are in fact not differentiable for all inputs. For example, the rectified linear unit (`ReLu`) given by

$$g(z) = \max(0, z)$$

is clearly not differentiable for $z = 0$. In practice, this does not pose a problem as most neural network training packages use only either the right- or left-handed derivatives and do not check for continuity, since numerical errors will dominate the calculation anyway [2].

`ReLu` units are often preferred because, in principle, their linear behaviour makes their derivatives straightforward to compute so optimization should be straightforward [2]. (Note that truly linear activation functions do not make sense, for their gradients to not depend on their input.) However, if a neuron produces zero input to the activation, the neuron will get stuck with an activation of zero, and SGD will be unable to allow the neuron to learn. In practice, this is fixed by activation functions like the "leaky ReLu" that have vanishing but nonzero gradient for small activations [5].

The sigmoid function, and the closely-related hyperbolic tangent function, are also common choices for activation function. The sigmoid function is given by

$$g(z) = \frac{1}{1 + e^{-z}}$$

and should be used for random variable prediction, for their range is $(0, 1)$. There is a cute theoretical justification found in [2] for the use of the sigmoid functions for the output of a Bernoulli variable: given an un-normalized Bernoulli distribution $\tilde{P}(y)$,

$$\log \tilde{P}(y) = yz \implies P(y) = \frac{e^{yz}}{\sum_y e^{yz}} = \sigma(z(2y - 1))$$

While this theoretical motivation is tempting, in practice the choice of activation function should be optimized by hand. The sigmoid function is conveniently generalized to multidimensional outputs (needed when performing one-hot encoding for a generic categorical random variable) by the `softmax` function.

Note that the sigmoid and tanh functions saturate when the input strays too far from zero–this can result in decaying gradients, dramatically slowing learning. Care must be taken, then, to normalize their inputs.

B.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a restricted subclass of ANNs. They are based on the convolution operation, which, in two dimensions, looks like

$$(I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

where I represents the input to the convolution and K represents the fixed kernel being applied. This somewhat counterintuitive form of the convolution operation is meant to show explicitly that the convolution operation is commutative (the arguments of the two functions can be switched

while resulting in the same output), but most libraries implement a simpler version of the convolution

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

which, in general, makes no practical difference to the training process [2]. This amounts to dotting the (m, n)-dimensional with the input. In this way, the convolution operation naturally reduces the dimensionality of the input

$$w' = \frac{w - k + 2p}{s} - 1$$

where w is the old width, w' is the new width, k is the kernel width, p is the amount of padding, s is the stride.

CNNs can be thought of as a restricted subclass of ANNs because the application of each kernel can be thought of as a layer in a fully connected ANN, except with the prior that all weights outside the kernel window are set to zero [2]. In other words, it defines a length scale on which it can recognize correlations, for the weights must all be zero other than in a small region. By applying successive filters in this way, we can build larger features out of smaller ones.

Of course, CNNs are not implemented in this way in practice. Instead they perform batches of filters in parallel so multiple features can be learned at the same length scales, but this way of thinking demonstrates why CNNs are so effective at object recognition.

Convolutional neural networks are often used for image recognition because they are more efficient than ANNs due to their inflexibility. Because the kernel is made to be smaller than the input, only local features are identified. In the next layer, those local features can be put together into larger features. We do not need to re-learn these features every time we slide the kernel–by assumption we keep the kernel constant while we slide it across the image such that we can recognize features regardless of their location in the image. In this way, the learning process can be made more efficient for there are fewer parameters to learn than if we tried to train an ANN to recognize a feature in any part of an image directly.

A final important feature the CNN is the pooling layer. By pooling over a window size after applying the convolution operation (and a nonlinearity, customary of any ANN), combining the elements in that window either by averaging, taking the maximum, or any other similar operation, makes CNNs invariant to small translations in the input, where the smallness is controlled by the size of the window. In situations in which the precise location of a feature is important, such operations should not be used.

B.3 Variational Autoencoders

So far we have described techniques for approximating scalar functions. Variational autoencoders allow us to tackle the problem of generating examples based on previous examples. First, we will start with the autoencoder. An autoencoder is designed to generate new samples $\hat{\mathbf{x}}$ from old ones $\mathbf{x}$. Autoencoders consist of two neural networks, an encoder and a decoder, that are joined together by a hidden layer, $\mathbf{h}$, that defines a latent space, in principle characterizing the features of any possible input to the encoder and output to the decoder. If the dimension of $\mathbf{h}$ is less than the dimension of the original input, then $\mathbf{h}$ will represent a compressed version of the input

With variational autoencoders, the goal is to use the decoders as generative models. In an undercomplete autoencoder, we may try minimizing the loss function of two variables $\mathcal{L}(\mathbf{x}, g(f(\mathbf{x})))$

where f is the encoding function and g is the decoding function. This results in deterministic data generation. The "variational" element of a variational autoencoder allows for some uniqueness in the the data generation process.

The following section is a summary of the lecture notes found in [1]. Consider the KL Divergence. This is a way of measuring the similarity between two probability distributions P and Q:

$$D_{KL}(P||Q) = -\sum_x p(x)\log q(x) + \sum_x p(x)\log p(x) = \sum_x p(x)\log\frac{p(x)}{q(x)}$$

Intuitively, this is *almost* the information in P subtracted from the information in Q however we the first term is taken with respect to P so it is the crossentropy instead of the entropy. Note that this breaks the symmetry in its arguments, so it is not a metric, although it is always positive. In the autoencoder prescription, we assume there is a hidden variable $\mathbf{z}$, such that one can compute $p(\mathbf{z}|\mathbf{x})$ (i.e. compute the encoding of a sample). Using standard notions of probability

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{x})p(\mathbf{z})}{p(\mathbf{x})} = \frac{p(\mathbf{x},\mathbf{z})}{p(\mathbf{x})}$$

The term in the denominator is often intractable–for example, in our case, our inputs are $28\times 28 = 784$-dimensional, so evaluating $p(x)$ is a 784-dimensional integral. To simplify the problem, we approximate $p(\mathbf{z}|\mathbf{x})$ with the distribution $q(\mathbf{x})$. That is $\hat{p} = q$. Then, if we choose q to be a straightforward parametric distribution (e.g. a multivariate Gaussian) then we can vary the free parameters of the distribution to maximize the log likelihood of the estimator's accuracy. The KL divergence between the two distributions is

$$\begin{aligned}
D_{KL}[q(z|x)||p(z|x)] &= -\sum_z q(z|x)\log\frac{p(z|x)}{q(z|x)} \\
&= -\sum_z q(z|x)\log\frac{p(x,z)}{p(x)q(z|x)} \\
&= -\sum_z q(z|x)\log\frac{p(x,z)}{q(z|x)} + \sum_z q(z)\log p(x) \\
&= -\sum_z q(z|x)\log\frac{p(x,z)}{q(z|x)} + \log p(x)
\end{aligned}$$

Re-arranging, we have found that

$$\log p(x) = D_{KL}[q(z|x)||p(z|x)] + \mathbb{E}_q\log\frac{p(x,z)}{q(z|x)}$$

Note that since we are given x, $\log p(x)$ is simply a constant. Our goal is to minimize D_{KL}, which we see in this form is equivalent to maximizing $\mathbb{E}_q\log\frac{p(x,z)}{q(z|x)} \equiv \mathcal{L}$, where $\mathcal{L}$ is called the variational lower bound. We can now further simplify,

$$\mathcal{L} = \sum_z q(z|x)\log p(x|z) - D_{KL}(q(z|x)|p(z)) = \mathbb{E}_{q(z)}\log[p(x|z)] - D_{KL}[q(z|x)||p(z)]$$

We have $p(x|z)$ and we can construct a $q(x|z)$ by maximizing $\mathcal{L}$ which is an approximation of $p(z|x)$. In the language of neural networks, we can make our encoder network $q(z|x)$ and our decoder network $p(\hat{x}|z)$. So if we want to make x as close to $\hat{x}$ as possible, we make the combined

CVAE, including both p and q minimize $-\mathcal{L}$ as the loss function. The first term in the variational lower bound is conceputally the construction error while the second term is the similarity of the probability distribution with which you choose to model the latent space with the true distribution. This is the basic principle behind a CVAE.

A very useful simplification of this result is found in [6]. If we take $P(z) \sim \mathcal{N}(0, 1)$ and $Q(z|x)$ to be a multivariate Gaussian with $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ then the KL divergence term simplifies to

$$D_{KL}[q(z|x)||p(z)] = \frac{1}{2} \sum \left(\boldsymbol{\Sigma}(x) + \boldsymbol{\mu}^2(x) - 1 - \log \boldsymbol{\Sigma}(x) \right)$$

(assuming the covariance matrix is diagonal). Doing a change of variables from $\quad \rightarrow \log \quad$ then

$$D_{KL}[q(z|x)||p(z)] = \frac{1}{2} \sum \left(\exp \boldsymbol{\Sigma}(x) + \boldsymbol{\mu}^2(x) - 1 - \boldsymbol{\Sigma}(x) \right)$$

This simplifies what we want to calculate significantly–if we allow the latent space to represent $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ then the KL divergence term can be easily calculated using the formula above and the other term in the loss is the usual binary crossentropy between samples.

References

[1] Ali Ghodsi. Variational autoencoders lecture, 2017.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[4] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[5] Andrej Karpathy. Convolutional neural networks for visual recognition lecture notes (cs231n), 2015. `https://cs231n.github.io/neural-networks-1/`.

[6] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[7] Grant Sanderson. Backpropagation calculus, 2017.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.