Variational Autoencoders

Faustine Li and Luyang Wang

Abstract

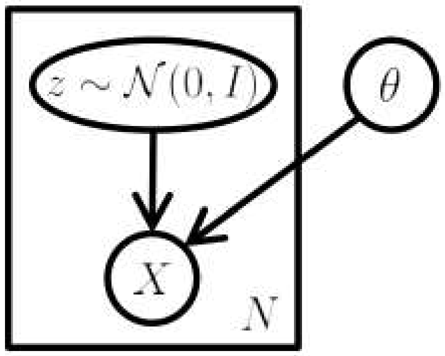
Variational autoencoders (VAE) are a popular unsupervised learning method. Built on top of neural networks, powerful function approximators, VAEs can do efficient posterior inference on complicated distributions. Because the data is encoded in a lower dimensional latent variable space, dimensionality reduction / compression is a common application. They are often trained with images, where they encode the several hundreds or thousands of pixels into a single digit number of latent variable parameters. VAEs can also be thought of as a generative probability model; realistic new images can be produced from a sample from the latent variable space. In this project, we build a variational autoencoder from scratch in Python and train it on MNIST handwritten digit images.

1. Introduction

1.1 Background

Variational autoencoders are primarily a form of a generative model; they produce data that is similar to, but not exactly like the data they are trained on. The data is often very high dimensional such as images, text, or speech. Trying to design a model from stratch that can produce, for example, realistic images of cats would be increadibly daunting. Not only is there considerable variation among different breeds of cats, but they are notorious deformable. VAEs have become a popular research topic because they can reduce the dimensionality of complex data into a latent variable representation. Generating new data points can be easily thought of as taking a sample from the latent probability distribution.

As a way of example, consider a data point $x \in X$. This could be a set of pixel intensities in an image of a handwritten digit. We assume that there is an unknown distribution with parameters θ that produced the image. We also assume that there is a latent variable z from some distribution with parameters ϕ that is unobserved, but influences that data generation step. Therefore x comes from the conditional distribution $p(x \mid z,\ \theta)$. Below is a figure (taken from the *Tutorial on Variational Autoencoders* paper) that illustrates the probability model as a diagram. Each data point $x_i \in \{x_1, \cdots, x_N\}$ is influenced by a random variable z_i and unknown parameters θ.



1.2 Variational Inference

If we could estimate the parameters of $p(x;\theta)$, the marginal likelihood, then we could sample a new data point from it. We have found parameters for the generative model. From the rules of probability, we can integrate out the latent variable.

$$p(x;\theta) = \int p(z)\, p(x \mid z)\, dz$$

The distribution $p(z)$ is assumed to be from some parametric model - in the case of VAEs often just $N(0, I)$. This is also called the prior in Bayesian inference. The term $p(x \mid z)$ is the conditional distribution. However, for very complex distributions the integral quickly becomes intractable. Other methods of inference such as MCMC are computationally expensive and have problems scaling with very large datasets. A method called variational inference seeks to very efficiently estimate parameters of even very complicated data.

Variational inference works by finding the posterior $p(z \mid X)$. We do this by finding an approximate function $q(z \mid \phi)$, where ϕ are variational parameters that need to be estimated. The function q does not need to be in the same family as p. We just tune ϕ so that q is close to p. The closeness is measured using Kullback–Leibler divergence.

$$KL(q \,||\, p) = E[log(\frac{q(z)}{p(z \mid x)})]$$

Because of the assumption that $z \mid x$ comes from a normal distribution $N(\mu, \sigma^2 I)$, the KL term becomes:

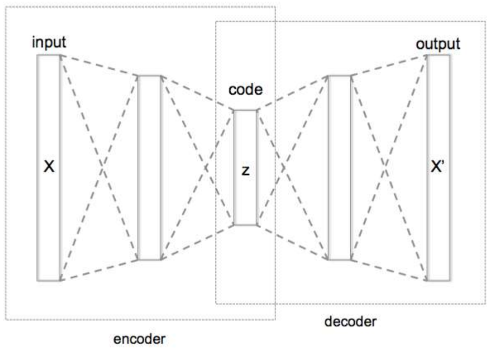
$$KL(N(\mu, \sigma^2) \,||\, N(0,1)) = -\frac{1}{2} \sum (1 + \sigma^2 - \mu^2 - exp(\sigma^2))$$

Now all we need some functional approximation technique to minimize the KLD.

1.3 Variational Autoencoders in Practice

In practice, we use neural networks for functional approximation. That means that variational autoencoders can be built using the large body of neural network packages and resources. In addition, they can borrow some of the techniques of efficient training including optimzation techniques such as stochastic gradient descent and mini-batching.

The archtechture is similar to an autoencoder. The encoder part of the network encodes high dimensional data into a lower dimensional representation. The decoder turns that lower dimensional input into a reconstruction of the input data. Below is a schematic of the neural network architecture.



The loss function is a sum of the reconstruction loss (cross-entropy or square error) and the KL divergence loss. We can take the gradient of the loss with respect to the weights and train using backpropagation. The only snag is that we can't backpropigate through a stochastic node. We get around this by using a reparameterization trick. We change $z \sim N(\mu, \sigma^2)$ to $z = \mu + \sigma^2 * \epsilon, \epsilon \sim N(0, 1)$. Then the gradient is determanistic and we can train the network in the normal way.

2. Implementation

We first implement the main functions in Python and numpy. Numpy allows us to do matrix multiplication that is essential in the feedforward and backpropagation methods to train the network. An outline of the training process is given below:

1. Input data is fed forward through the encoder layers.
2. The output of the encoder is a set of learned parameters μ and σ^2 for $q(z \mid x)$
3. A sample is taken from $z_i \sim N(\mu, \sigma^2 I)$
4. The sample is fed forward through the decoder layers to produce an output image.
5. The gradient of the error terms between the image and the reconstructed image are calculated with backpropagation.
6. The weights are updated with gradient descent.

Generating new images is as simple as sampling values from the latent distribution.

1. Sample $z_i \sim N(\mu, \sigma^2 I)$
2. Feedforward through the decoder to obtain a new image x'

Code for the implementation can be found at this Github repository (https://github.com/FaustineLi/Sta663-Project). The package can be installed with `python setup.py install`. All of the functionality is in the `vae` class which has methods `train`, `predict`, and `generate`.

First we wrote a class that stores paramters of the model like the intended reconstruction loss, size of the encoder layers (including input) and size of the decoder layers (including output). For the most part we assume that the input and output dimensions are the same. We will intialize matricies that store the weights values.

In the initialization step, we take a dictionary with parameters such as a reconstruction loss function (as well as the gradient), activation function (and gradient). We also take the size of the latent dimensional variable. Often we choose to have two dimensions so that we can visualize the reduced dimensional manifold in 2D plots, but more a larger dimensional vector can encode more rich information.

We define feedforward and backpropagate methods that are used interally in the train function. Feedforwards takes as inputs the training data X and outputs an estimate $\hat{y}$. Backpropagate takes as inputs ground truth y and an estimate $\hat{y}$ and calculates the gradient of the error with respect to the weights. We update the weights with simple steepest descent given a learning rate α.

In [4]:
```
def train(self, X, y):
    '''trains the VAE model'''
    for i in range(self.max_iter):
        yhat = self.feedforward(X)
        grad_encoder, grad_decoder = self.backprop(X, y, yhat)

        for i in range(self.number_decoder_layers):
            self.decoder_weights[i] -= self.alpha * grad_decoder[i]

        for j in range(self.number_encoder_layers):
            self.encoder_weights[j] -= self.alpha * grad_encoder[j]
            change = grad_encoder[j]
    return None
```

Once the network is trained ie. once the network has minimized the distance between the true posterior and the variational approximation, we can use those learned weights to generate new images. This can happen in one of two ways.

1. Generation of images similar to a given input.
2. Generating images from a sample of the latent state.

In the first case we can provide an example image to the trained model, encode it, and add some noise to the encoding. The decoded version will have some of the characteristics of the original, but still produce a new digit. This is done with the `predict` method in the `vae` class which is nothing but a feed-forward pass through both the encoder and decoder.

In the second case, we sample a latent variable z or otherwise provide values to the method. The method performs a feed-forward pass on just the decoder layers to produce a new image.

In [6]:
```
def generate(self, z):
    '''generates new images from a trained VAE model'''
    # feedforward on decoder
    self.gen_input = {}
    self.gen_activation = {}
    self.gen_input[0]     = z.T @ self.decoder_weights[0]
    self.gen_activation[0] = self.activation(self.gen_input[0])

    for i in range(1, self.number_decoder_layers):
        self.gen_input[i] = self.gen_input[i-1] @ self.decoder_weights[i]
        self.gen_activation[i] = self.activation(self.gen_input[i])

    return self.gen_activation[self.number_decoder_layers - 1]
```

4. Testing

We choose the unittest framework to implement the testing. The following are some of the testing we did.

```
In [ ]: def setUp(self):
            self.vaeT = vae([2, 2], [2, 2], params)

        def test_feedforward(self):
            train_data = np.array([[1,0],[0,1]])
            sol = np.array([[0.49,0.49],[0.49,0.49]])
            assert_almost_equal(self.vaeT.feedforward(train_data), sol,decimal = 2)

        def test_backprop(self):
            X = np.array([[0,0],[0,0]])
            y = np.array([[0,0],[0,0]])
            yhat = np.array([0,0])
            train_data = np.array([[1,0],[0,1]])
            tmp = self.vaeT.feedforward(train_data)
            sol = ({0: np.array([[ 0.,  0.],
                [ 0.,  0.]]), 1: np.array([[-0.04, -0.06],
                [-0.04, -0.06]])}, {0: np.array([[ 0.,  0.],
                [ 0.,  0.]]), 1: np.array([[ 0.,  0.],
                [ 0.,  0.]])})
            assert_almost_equal(self.vaeT.backprop(X,y,yhat)[0][0],sol[0][0],decimal =
         2)
```

The full suite of tests scripts can be found in the `tests` folder.

5. Optimization

5.1 Vectorization

In our first attempt, we wrote the code with plain Python, using only the built-in `math` and `random` libraries. Because our algorithm heavily depends on matrix multiplication, several matrix multiplication functions to handle various cases were written. The plain Python version can be found in under `vae/vae_unoptimized.py`. We ran the code on the MNIST data set and profiled the code with `prun`. The results are shown below.

In [2]:
```
import pstats
p = pstats.Stats('tests/train_unoptimized.prof')
p.print_stats()
pass
```

```
Sun Apr 30 15:20:47 2017    tests/train_unoptimized.prof

         2086296 function calls in 14.491 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   14.491   14.491 {built-in method builtins.exec}
  2012516    0.107    0.000    0.107    0.000 {method 'append' of 'list' obje
cts}
        2    0.034    0.017    0.035    0.017 <ipython-input-34-a7a6fecb4bff
>:31(grad_loss)
       18    0.000    0.000    0.013    0.001 ..\vae\vae_unoptimized.py:114
(t)
        2    0.001    0.000    6.706    3.353 ..\vae\vae_unoptimized.py:205(f
eedforward)
        8    0.146    0.018    0.174    0.022 ..\vae\vae_unoptimized.py:85(sc
alar_mult)
        8    0.212    0.027    0.243    0.030 ..\vae\vae_unoptimized.py:118(m
at_sub)
        1    0.015    0.015   14.487   14.487 ..\vae\vae_unoptimized.py:249(t
rain)
        8    0.013    0.002    0.017    0.002 <ipython-input-34-a7a6fecb4bff
>:1(act)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.P
rofiler' objects}
       18    0.013    0.001    0.013    0.001 ..\vae\vae_unoptimized.py:116(<
listcomp>)
       22   13.905    0.632   13.948    0.634 ..\vae\vae_unoptimized.py:72(ma
tmult)
        8    0.010    0.001    0.011    0.001 ..\vae\vae_unoptimized.py:95(mu
ltiply)
        8    0.020    0.002    0.027    0.003 <ipython-input-34-a7a6fecb4bff
>:11(grad_act)
     2512    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.003    0.001    7.349    3.674 ..\vae\vae_unoptimized.py:147(b
ackprop)
    71160    0.009    0.000    0.009    0.000 {built-in method math.exp}
        1    0.003    0.003   14.491   14.491 <string>:1(<module>)
```

Most of the time spent in the function is in the `feedforward` and `backprop` steps. The real culprit is the `matmul`, matrix multiplication function. Our original function has a very naive implementation (with nested `for` loops), so it is not surprising to see it struggle on very large matricies.

There are many libraries that implement effiecient matrix operations and vectorization. We rewrote the `vae` class using `numpy` (see `vae.py`) and profiled the code on the same test case. The results are dramatically better.

```
In [6]: import pstats
        p = pstats.Stats('tests/train_optimized.prof')
        p.print_stats()
        pass
```

```
Sun Apr 30 15:33:31 2017    tests/train_optimized.prof

         36 function calls in 0.016 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.000    0.000 ..\vae\vae.py:49(KLD_grad)
        1    0.000    0.000    0.016    0.016 {built-in method builtins.exec}
        1    0.004    0.004    0.016    0.016 ..\vae\vae.py:142(train)
        1    0.000    0.000    0.016    0.016 <string>:1(<module>)
       10    0.001    0.000    0.001    0.000 <ipython-input-47-b0bed5c11f7e
>:5(<lambda>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.P
rofiler' objects}
        2    0.003    0.002    0.004    0.002 ..\vae\vae.py:103(feedforward)
        2    0.000    0.000    0.000    0.000 <ipython-input-47-b0bed5c11f7e
>:7(<lambda>)
        2    0.007    0.003    0.008    0.004 ..\vae\vae.py:53(backprop)
        8    0.001    0.000    0.001    0.000 <ipython-input-47-b0bed5c11f7e
>:4(<lambda>)
        2    0.000    0.000    0.000    0.000 {built-in method numpy.core.mul
tiarray.array}
        4    0.000    0.000    0.000    0.000 {built-in method numpy.core.mul
tiarray.arange}
```

5.2 Other Considerations

In order to scale to larger datasets, we tried to implement stochastic gradient descent into our model. However it is somewhat tricky because we want to take a gradient with respect to each of the weights in the network. This is certainly an area of research to improve the performance.

Finally, we attempted to use other methods such as numba jit and Cython, but because the numpy matrix math is already highly optimized, we did not find a substational speedup.

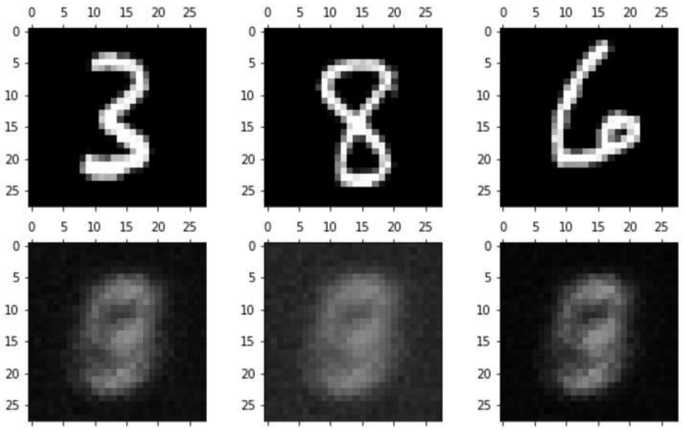
6. Applications and Results

Once we had the algorithm designed, written, tested, and optimized we ran various experiments with two different datasets. We chose to use grayscale images, but the VAE class can handle any numerical data and arbitary sizes for the fully connected layers. Running the code for the examples can be found in the Jupyter notebooks in the Github repository.

6.1 MNIST Digits

We first ran our optimized variational autoencoder on the MNIST handwritten images data set. This is classic dataset in the machine learning community for tasks such as image classification, autoencoding, and deep learning. The dataset consists of 50,000 training and 10,000 test images, along with labels. Each image consists of a 28 by 28 pixel, grayscale image of a handwritten digit. The digits are centered, free of noise, and of fairly uniform size. This makes the set easy to use without preprocessing.

We trained our autoencoder on the training digits. Because the variational autoencoder technique is an unsupervised, we don't use the class labels. We feed the raw data (as a 50000 x 784 array) to the variational autoencoder. We chose to have one fully connected layer of 200 neurons on both the encoder and decoder sides to more closely match the architecture of the paper. We used a sigmoid activation function and a squared error reconstruction loss. We ran the network for five iterations and tested on a couple of images in the test set. A full running example can be found in the `examples` directory of the Github repository. Below is an figure of the typical output, where the top row is the test image and bottom row is the VAE output.

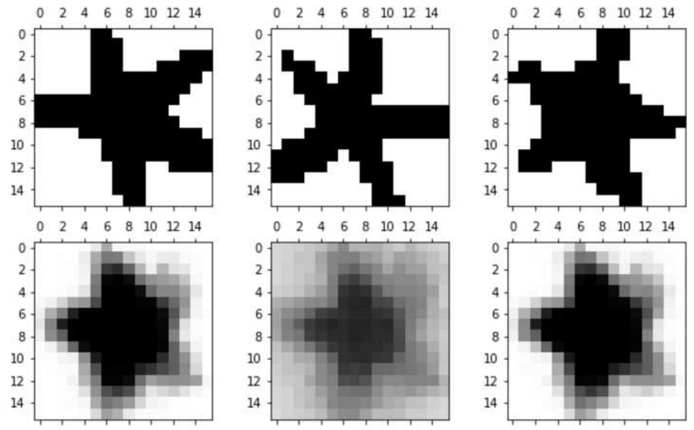


At five iterations, the results are fairly blurry and indistinct. However this appears to be inline with examples found online (http://kvfrans.com/variational-autoencoders-explained/) at a lower number of training epochs. We tried to increase the number of training iterations, but found the results to be similar to the above at a small number of iterations and too long to train, even with our optimizations, for a larger number of iterations. Given enough iterations we expect the images to be more defined and to capture more varience.

6.2 Caltech 101 Images

Because the MNIST data set is slighly too large to efficiently train on hardware available to us (laptops without GPUs), we looked at a smaller dataset. This is the Caltech 101 image dataset. The set has 101 image classes spread over several thousand thumbnail images. The version we used is just a 16 by 16 silluette, a solid black image on a white background. Although there are many image classes, we focused on a subset of the data - 86 images of starfish. Although these thumbnails are very simple, they have more variety than the MNIST set - images are often rotated, truncated, or have multiply objects.

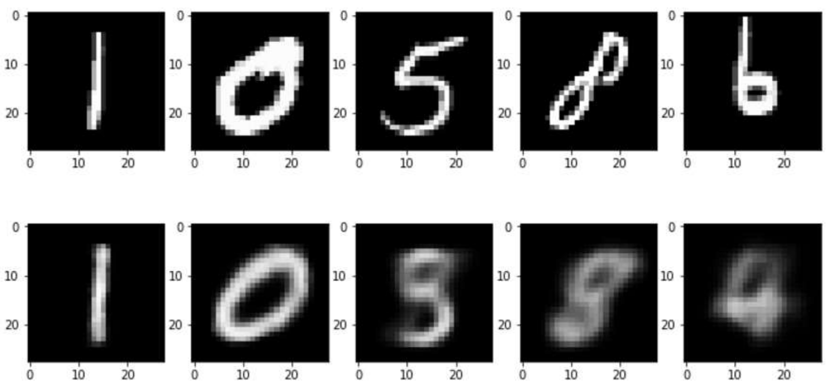
Similar to the MNIST version, we initialized the VAE network to to have an input layer, two hidden layers, and an output layer. Because the dataset is smaller, we significantly increased the number of training iterations. Below are examples of the results on the training data, in a side-by-side comparison.



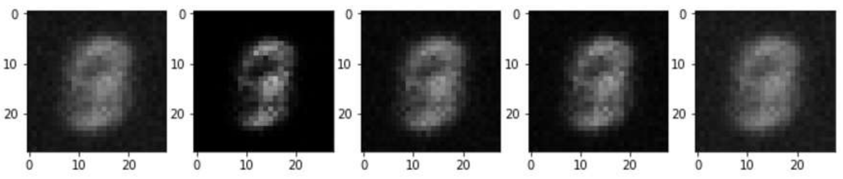
This time the results are marginally more believable. The starfish in the first generated image looks similar to a slighly blurred starfish. We can clearly see features of the five arms. However, it is also important to show when the autoencoder is not working as well as it should. The second generated image is significantly smoother than the other two images. The third image shows a starfish, but hasn't represented the rotation adequately.

7. Comparison

We wanted to compare our implementation with another version. Because VAEs borrow from the extensive body of work in deep learning and neural networks, many tutorials written in TensorFlow are availible online. We compared our VAE python code with a TensorFlow version code. Below is the test output after five training epochs, with the input digit on top and the generated image on the bottom row. Training took about one minute.



Now here is the output of our Python implementation on the same test digits. Training took about 20 seconds.



We can see TensorFlow gives much better results, and our version is very blurry. Our version is not captureing the variability of the digits adequately. However this isn't exactly a fair comparison because the TensorFlow implementation uses a more complicated training algorithm with minibatching. In addition, it utlizies a slighly deeper network architecture.

8. Conclusion

In this project we have implemented the Variational Autoencoder algorithm from the paper *Tutorial on Variational Autoencoders* (2013) by Kingma and Welling. We implemented the main class and methods in Python and optimized it my using the `numpy` library for efficient matrix math. After optimizing and testing our code, we tested our code on two datasets, the MNIST handwritten digits and the Caltech 101 silhouettes. Finally, we compared our results to a implementation in TensorFlow.

Although our VAE did not perform at the same level as our TensorFlow version or the examples in the literture, we are confident that we could further refine and develop the algorithmn to improve the results. For further research, we can implement more efficient and robust backprogagation and optimization techniques to improve training time. Finally, we would like to expand our test set to include more complex / variable images that more approximate the real world.

References

1. Tutorial on Variational Autoencoders - Kingma, Welling (https://arxiv.org/abs/1312.6114)
2. Autoencoding Variational Bayes - Doersch (https://arxiv.org/abs/1606.05908)
3. Variational Inference - Blei (https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf)
4. Neural Networks Demystified - Welch (https://github.com/stephencwelch/Neural-Networks-Demystified)
5. MNIST Dataset (http://yann.lecun.com/exdb/mnist/)
6. Caltech 101 Datasets (http://www.vision.caltech.edu/Image_Datasets/Caltech101/)
7. Building Autoencoders in Keras (https://blog.keras.io/building-autoencoders-in-keras.html)