Benjamin Ye
CS/CNS/EE 156a: Learning Systems (Fall 2023)
November 17, 2023

Bonus Exercise

1 Classifying digits with NNs

1.3 Number of Parameters

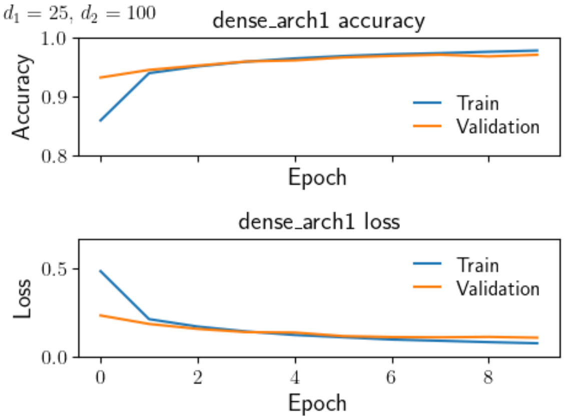
What happens to the learning curve when you vary the number of hidden units? Specify the number of hidden units you use in each layer, and report learning curves for each different architecture you try. What trends do you notice?

As the number of hidden units in the first layer decreases, the accuracy decreases and the loss increases on both the training and validation sets, but much more significantly on the training set. For example, the figure on the next page shows a downwards shift of both accuracy curves and an upwards shift of both loss curves as d_1 decreases from $d_1 = 200$ to $d_2 = 25$. This is likely caused by the order of magnitude difference in the number of parameters (see tabulated results in the following pages). In the other direction, we get diminishing returns; when the number of hidden units doubles in the first layer, the training accuracy and loss improves nearly imperceptibly, and the validation accuracy and loss worsen slightly. This is indicative of a sufficient number of parameters in the original model ($d_1 = 200$ and $d_2 = 100$) and suggests that the default dimensionalities are the optimal values since they maximize the accuracies while minimizing the losses and the number of parameters.

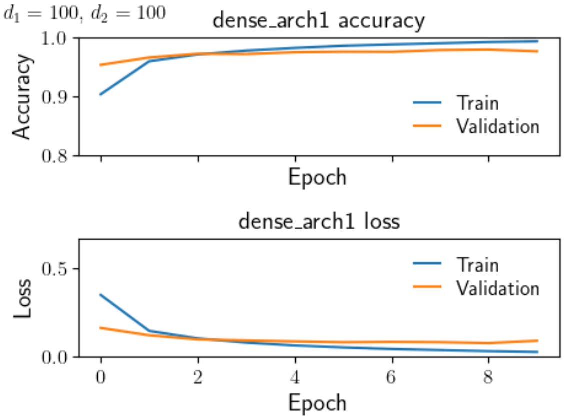
The performance of the model is extremely insensitive to the number of hidden units in the second layer for in the range of d_2 explored, with minimal fluctuations in the accuracies and losses as d_2 increases or decreases, because the changes in the number of parameters are inconsequential compared to when d_1 is varied. It is expected that when d_2 is extremely small, i.e., $d_2 \leq 10$, the model will perform poorly because the number of parameters will have decreased an order of magnitude.

In the following figures, the dimensionality of the first layer is adjusted from the default $d_1 = 200$.

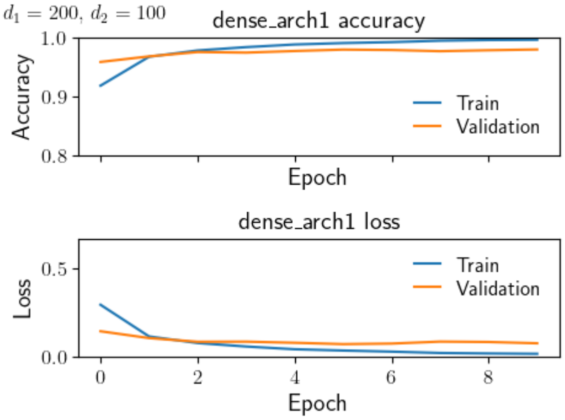
$d_1 = 25,\ d_2 = 100$



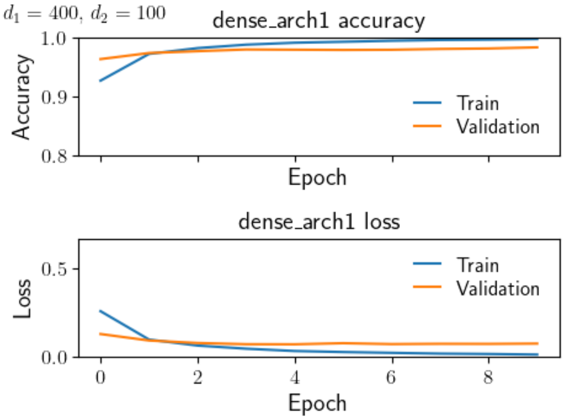
$d_1 = 100,\ d_2 = 100$



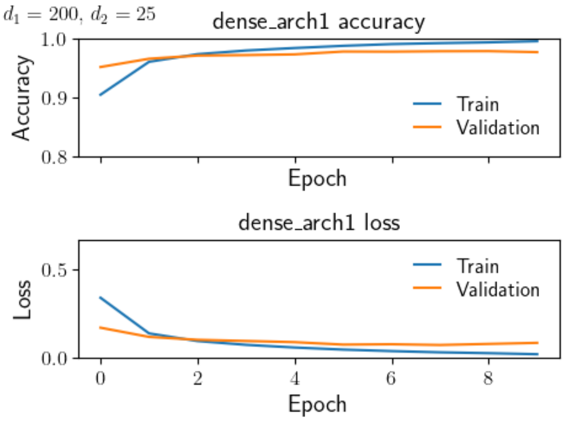
$d_1 = 200,\ d_2 = 100$

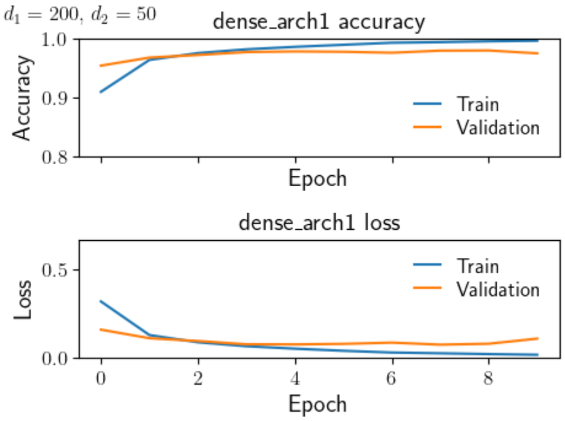


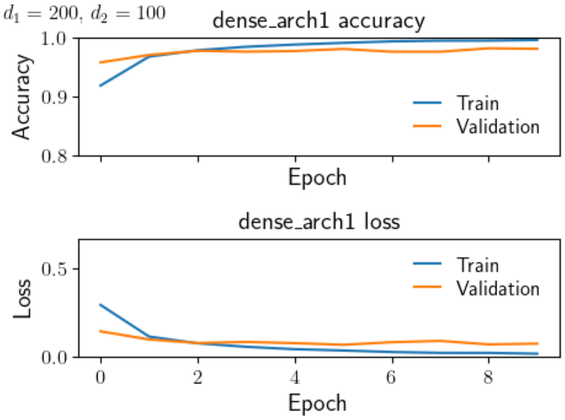
$d_1 = 400,\ d_2 = 100$

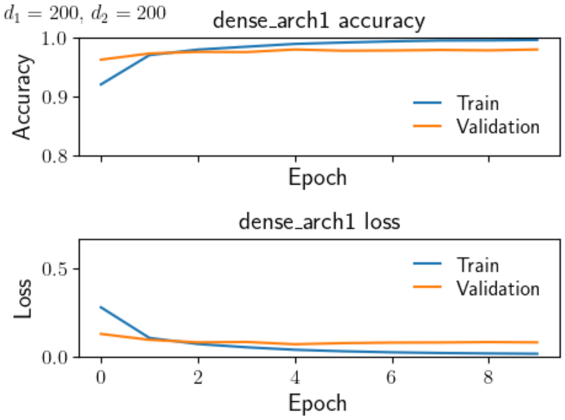


In the following figures, the dimensionality of the second layer is adjusted from the default $d_2 = 100$.









The table below summarizes the number of parameters and the performance of the models as d_1 and d_2 are varied. The values original/default model are bolded for comparison.

d_1	d_2	n_{param}	$\text{acc}_{\text{train}}$	$\text{loss}_{\text{train}}$	acc_{val}	loss_{val}
25	25	20535	0.9692	0.1026	0.9612	0.1264
25	50	21435	0.9739	0.0865	0.9661	0.1139
25	100	23235	0.9758	0.0788	0.9672	0.1101
25	200	26835	0.9786	0.0692	0.9694	0.1025
50	25	40785	0.9835	0.0541	0.9718	0.0941
50	50	42310	0.9849	0.0500	0.9745	0.0880
50	100	45360	0.9845	0.0500	0.9701	0.1001
50	200	51460	0.9888	0.0360	0.9738	0.0902
100	25	81285	0.9911	0.0299	0.9643	0.1240
100	50	84060	0.9911	0.0288	0.9758	0.0843
100	100	89610	0.9928	0.0244	0.9747	0.0858
100	200	100710	0.9930	0.0224	0.9760	0.0860
200	25	162285	0.9947	0.0181	0.9761	0.0823
200	50	167560	0.9954	0.0150	0.9741	0.1060
200	**100**	**178110**	**0.9950**	**0.0154**	**0.9801**	**0.0722**
200	20	199210	0.9951	0.0150	0.9789	0.0791
400	25	324285	0.9960	0.0123	0.9802	0.0773
400	50	334560	0.9958	0.0131	0.9786	0.0839
400	100	355110	0.9955	0.0141	0.9802	0.0756
400	200	396210	0.9960	0.0121	0.9750	0.1024
47	25	38355	0.9808	0.0628	0.9700	0.1037

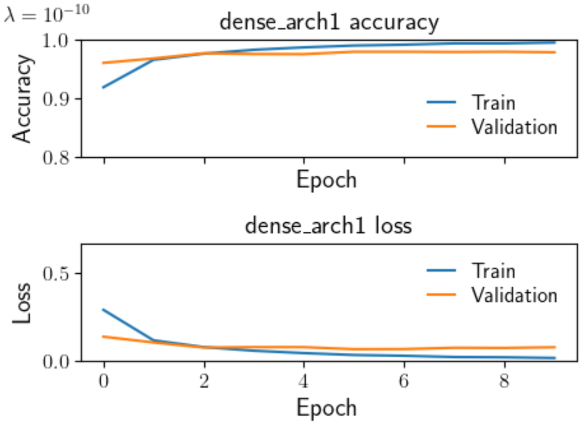
What is the smallest number of model parameters (not hidden units; view this via the summary function) for which you can achieve over 97% validation accuracy? An estimate is fine.

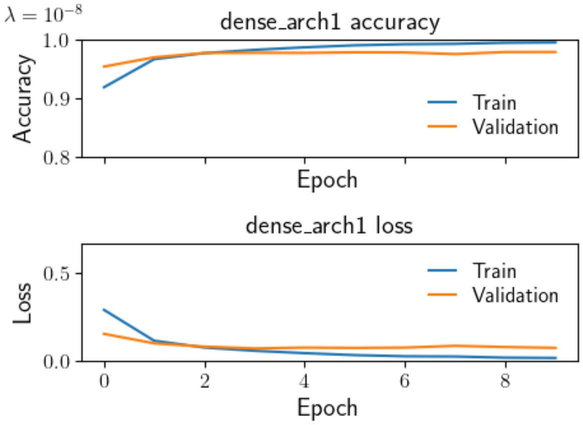
Based on the tabulated results above and some cursory testing, it appears that $n_{\text{param}} \cong 40{,}000$ (rounded up to the nearest ten thousand) will give $\text{acc}_{\text{val}} \geq 0.97$ in ten epochs.

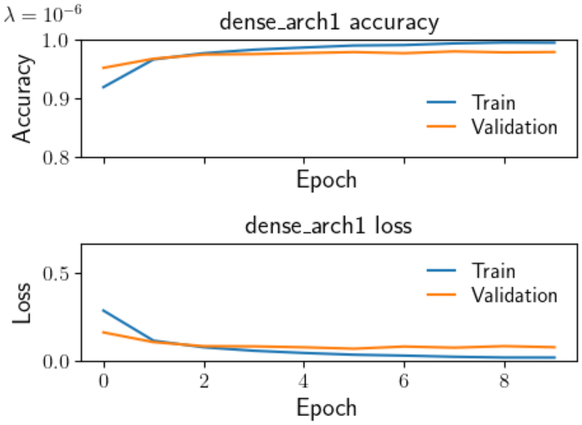
1.4 Regularization

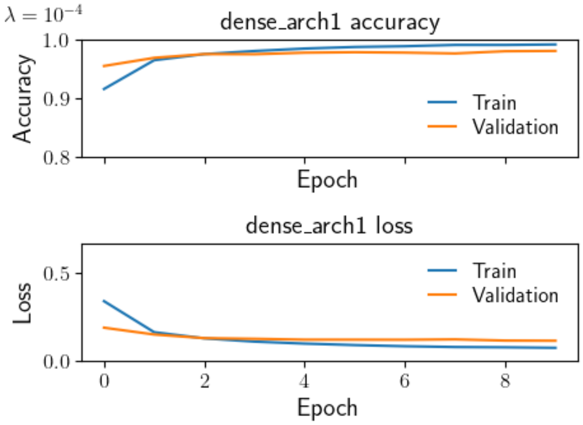
Explore the usefulness of weight decay regularization (minimizing $L(\mathbf{w} \mid x, y) + \lambda \|\mathbf{w}\|^2$ instead of just $L(\mathbf{w} \mid x, y)$) on the large neural net implementation given in the support code. You can do so changing the `REGULARIZATION` parameter in the IPython notebook, or by specifying the `-r` command line option in `train.py` in the command shell. Report learning curves for different choices of λ (just a few).

The λ value was varied from 10^{-10} to 1. Below, the accuracy and loss curves are shown for a subset of the runs, and the final accuracies and losses are tabulated.









(Discussion continued on the following page.)

λ	n_{param}	$\text{acc}_{\text{train}}$	$\text{loss}_{\text{train}}$	acc_{val}	loss_{val}
0	**178110**	**0.9950**	**0.0154**	**0.9801**	**0.0722**
10^{-10}	178110	0.9952	0.0143	0.9786	0.0754
10^{-8}	178110	0.9954	0.0145	0.9791	0.0720
10^{-6}	178110	0.9950	0.0170	0.9790	0.0755
10^{-4}	178110	0.9920	0.0723	0.9809	0.1126
10^{-2}	178110	0.9154	0.7611	0.9163	0.7514
1	178110	0.1124	2.3013	0.1135	2.3011

Compared to the model with no regularization, the models with $\lambda = 10^{-10}$ and $\lambda = 10^{-8}$ performed slightly better in the training accuracy and loss but regressed slightly in the validation accuracy. All other models with $\lambda \geq 10^{-6}$ performed worse in all metrics, suggesting significant underfitting as $\lambda \to \infty$.

Can you use regularization to get a similar result to one of the smaller neural nets you implemented in the previous part?

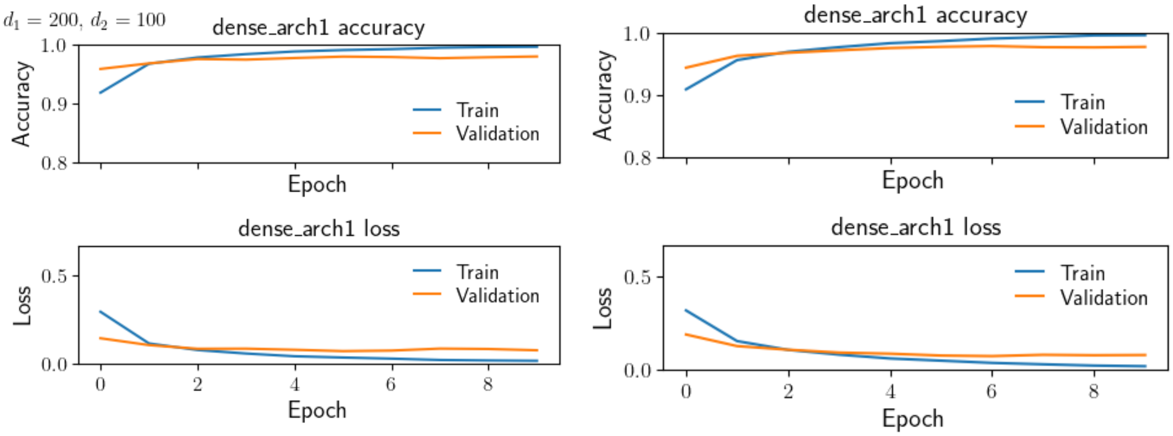
No. Regardless of the λ value used, the validation accuracy using the smaller model with $d_1 = 47$ and $d_2 = 25$ hovers around 0.97 and is always lower than the 0.995 attainable using the larger model.

(My understanding of this question is whether the smaller neural net from 1.3 that can achieve a validation accuracy of 0.97 is able to compete with the larger neural net if regularization with the optimal λ value is used. If the question is be interpreted literally, then the answer is yes, the large neural net can perform worse and achieve a validation accuracy of 0.97 like the smaller neural net if a λ value between 10^{-2} and 10^{-4} is used.)

1.5 Activations

In class, we have primarily discussed using the tanh and sigmoid activations. In practice nowadays, people often use the ReLU activation, which is defined as $\mathrm{ReLU}(s) = \max\{0, s\}$. We use this activation function in the support code, and in practice, people have observed that it results in much faster convergence. Verify this for yourself by replacing the ReLU activation with a tanh activation. Why do you think this happens? (Hint: How does $\tanh(s)$ behave as $|s| \to \infty$? Compare this to $\mathrm{ReLU}(s)$.)

The ReLU and tanh activations performed similarly in terms of convergence. The accuracy and loss curves are shown below on the right, with those for the default neural net on the left for comparison.



Generally, the ReLU activation converges faster than the tanh activation because at large values of s, tanh plateaus to a value of 1 while ReLU continues to grow linearly. As such, the gradient of tanh vanishes, which impedes the learning process, but the gradient of ReLU does not.

1.6 Different Architectures

Now, let's say that I give you a fixed budget of 200 hidden units. What is the best validation accuracy you can achieve? Feel free to vary the number of layers, the kind of regularization (e.g., ℓ^1 which is based on the 1-norm of $\mathbf{w}$ as opposed to ℓ^2 which is the usual weight decay based on the 2-norm) and its strength (λ), the activation, and the optimizer you use.

The best validation accuracy achieved was 0.9784 with $d_1 = 150$ and $d_2 = 50$ without regularization.

Based on the findings in 1.3, 1.4, and 1.5, it appears that I should maximize the number of trainable parameters to get the best performance.

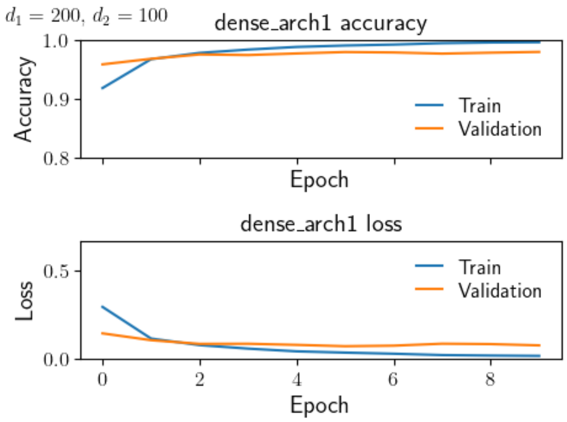
For two layers, the maximum number of parameters is achieved when $d_1 = 199$ and $d_2 = 1$, but models with more balanced distributions (e.g., $d_1 = 100$ and $d_2 = 100$, $d_1 = 150$ and $d_2 = 50$, or $d_1 = 175$ and $d_2 = 25$) performed better with training and validation accuracies of 0.99 and 0.98, respectively, compared to 0.11 and 0.11 for $d_1 = 199$ and $d_2 = 1$.

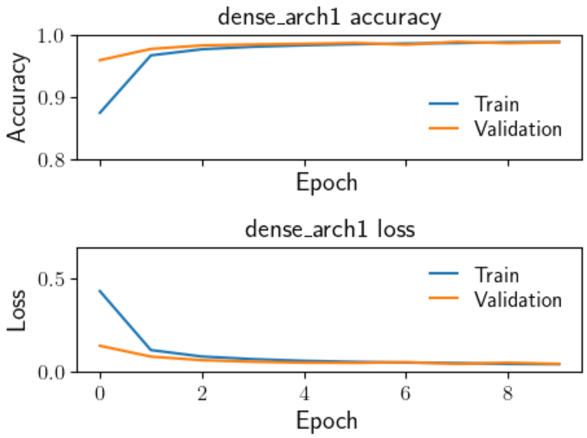
Increasing the number of layers and optimizing the number of nodes in each layer did not improve the performance. Neither did using regularization with a ℓ^1 or ℓ^2 regularizer.

1.7 Convolutional Neural Nets

By learning a few smaller convolutional filters instead of a series of huge matrices, a convolutional neural net performs much better on this image classification task with a fraction of the parameters. Verify this for yourself. Run the train script and the evaluate script using the convolutional neural net and no regularization. Compare the number of parameters between any of the convolutional and dense neural nets, and report learning curves.

Indeed, the convolutional neural net can achieve comparable accuracies and losses as the default neural net, but with two orders of magnitude fewer parameters (7,240 vs. 178,110). Notably, the validation accuracy is higher and now comparable to the training accuracy. The learning curves are shown below on the right, with those for the default neural net on the left for comparison.





2 Generative Adversarial Network

2.2 How does a GAN work?

1. Since this solution occurs at a saddle point, let's take the derivative of the loss with respect to $D(x)$. Since D is a function, take a functional derivative $\delta L^{(D)}/\delta D(X)$ (like a vector derivative, don't worry about being rigorous). Use the following loss,

$$L^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] - \frac{1}{2}\mathbb{E}_{x\sim p_{\text{model}}}\left[\log\left(1 - D(x)\right)\right]$$

Hint: We can interchange an integral and a functional derivative:

$$\frac{\delta}{\delta D(X)}\mathbb{E}_{x\sim p}[\log D(x)] = \frac{\delta}{\delta D(X)}\int \log D(x)\, p(x) = \int \frac{\delta}{\delta D(X)}\log D(x)\, p(x) = \int \frac{1}{D(x)} p(x)$$

Don't forget the chain rule!

$$\begin{aligned}\frac{\delta L^{(D)}}{\delta D(X)} &= -\frac{1}{2}\left[\frac{\delta}{\delta D(X)}\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \frac{\delta}{\delta D(X)}\mathbb{E}_{x\sim p_{\text{model}}}\left[\log\left(1 - D(x)\right)\right]\right] \\ &= -\frac{1}{2}\left[\int \frac{\delta}{\delta D(X)}\log D(x)\, p_{\text{data}}(x) + \int \frac{\delta}{\delta D(X)}\log\left(1 - D(x)\right) p_{\text{model}}(x)\right] \\ &= -\frac{1}{2}\left[\int \frac{1}{D(x)} p_{\text{data}}(x) - \int \frac{1}{1 - D(x)} p_{\text{model}}(x)\right]\end{aligned}$$

2. Now that we have the derivative, we solve for $D(x)$ in terms of the probability distributions p_{model} and p_{data} when $\delta L^{(D)}/\delta D(X) = 0$. Hint: Move one integral to the other side of the equality. Notice that if the values under both integrals are the same pointwise, we also have equality with integrals. Solve with respect to $D(x)$.

$$\frac{\delta L^{(D)}}{\delta D(X)} = 0 = -\frac{1}{2}\left[\int \frac{1}{D(x)} p_{\text{data}}(x) - \int \frac{1}{1 - D(x)} p_{\text{model}}(x)\right]$$

$$\int \frac{1}{D(x)} p_{\text{data}}(x) = \int \frac{1}{1 - D(x)} p_{\text{model}}(x) \longrightarrow \frac{1}{D(x)} p_{\text{data}}(x) = \frac{1}{1 - D(x)} p_{\text{model}}(x)$$

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{model}}(x) + p_{\text{data}}(x)}$$

3. You should get

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{model}}(x) + p_{\text{data}}(x)}$$

Do you think this is reasonable?

Seems reasonable. Consider the case when $p_{\text{data}}(x) = p_{\text{model}}(x)$, or when the distribution of images as classified by the discriminator D matches that of real and generated images created by generator G. As expected at this quasi-Nash-equilibrium state, D assigns the (expected) probability of $D(x) = 1/2$ to both real and generated images because it is unable to distinguish between them, indicating that G has created images realistic enough to not be detectable as fakes.

2.4 Reflecting on GAN Behavior

How would you describe the training process in terms of the images that the model generated?

At the beginning (epoch 0), the images are grayscale artifacts that do not resemble anything.

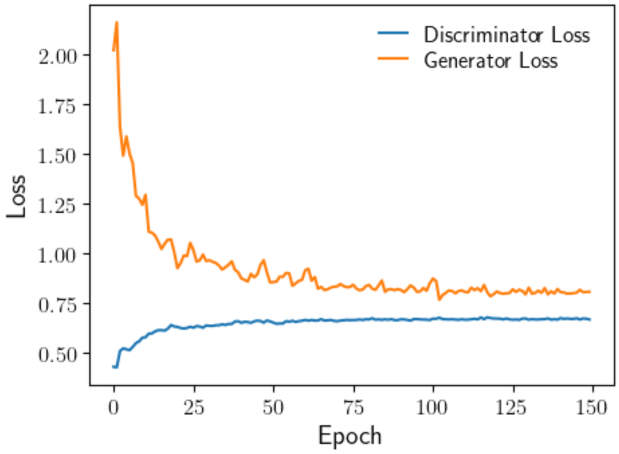
As training commences, the images start to resemble numbers (by around epoch 10) but are far from being able to be recognized as any specific number. Some simpler numbers, like 1, becomes legible due to their single stroke.

More complex numbers become distinguishable at around epoch 20. However, most look unnatural because of the gray uncertainty between the white digits and the black background and can easily be identified as generated images.

Further refinement occurs all the way up to the final epoch (150). In the final generated image, many of the digits look like they could have been written by a human (with bad handwriting), largely because the edges between the numbers and the backgrounds are much cleaner and pronounced.

What about the learning dynamics in the loss values for the generator and the discriminator?

As training proceeds, discriminator loss increases and the generator loss decreases. The generator starts out creating obviously fake samples and learns to make them realistic, while the discriminator initially can discern generated samples from real ones but begins to incorrectly classify samples as the generator feeds it more realistic samples.



A large concern in training GANs is a phenomenon known as "mode collapse". Look up this term and explain it in your own words (it doesn't have to be rigorous).

Mode collapse occurs when the generator continuously creates data samples are that are too similar or even identical because it has learned that certain features are more likely to fool the discriminator. As a result, the distribution of samples does not align with that of the real-world data. Worse yet, the discriminator learns to classify all samples coming from the generator as real images, so the discriminator does not provide the right feedback to the generator so that it can generate more realistic samples.

Skimming over the code, did the current generator's implementation manage to avoid mode collapse?

The `BatchNormalization` layers in the generator can help stabilize the training process by preventing a vanishing gradient (which causes the generator to continue producing similar samples).

3 Feedback

Overall, this was a fantastic way to get hands-on experience with perhaps the most widely used machine learning package (Keras and TensorFlow). Furthermore, the exercise reinforced most of the concepts covered in class and provided a practical application to showcase why certain algorithms or techniques lead to better trained models.

However, there are a few technical issues with the provided scripts, such as outdated Keras and TensorFlow packages, that

- caused some modules (like `keras.utils.np_utils` and `keras.layers.core`) to not load correctly without some debugging because their namespaces have been updated in newer versions of Keras,
- resulted in segmentation faults when saving the training models using `train.py`, and
- raised a plethora of vulnerability alerts when I pushed code to GitHub.

Moreover, the default neural net for Part 1 performed perhaps a bit too brilliantly, so much so that I was unable to get better results with weight decay regularization, show that the inferior tanh activation should have worse convergence, or highlight the improvements of a convolutional neural net. (It is also entirely possible that I implemented the approaches incorrectly.)