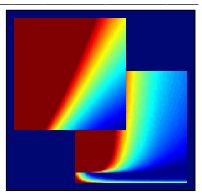
CS/CNS/EE 156a Learning Systems
Caltech - Fall 2021

https://learning.edge.edx.org/course/course-v1:Caltech+CSCNSEE156a+2021_T1/home

(Learning From Data campus version)



BONUS EXERCISE

– Experimenting with Deep Neural Networks –

Due: Friday, November 19, 2021, at 2:00 PM PST, by email to Yaser <yaser@caltech.edu>

This is an optional exercise. If you decide to do it, and you want to get the bonus for it, you need to complete it and turn it in by the above due date. The exercise builds on topics in Lectures 10-12.

(prepared by *Aadyot Bhatnagar*, extended by *George Stathopoulos*)

0 Introduction

In this exercise, you will be exploring different neural net architectures. If you wanted to do this in the past, you would have been forced to implement your own neural net library. However, with the recent explosion of interest in deep learning,[1] many high quality open source implementations have proliferated, including TensorFlow, Torch, Caffe, CNTK, Theano, and Keras, to name just a few. Moreover, many of these libraries can run your deep learning code on the GPU instead of the CPU with no extra work needed on the part of the developer.

While implementing the forwards and backwards propagation algorithms yourself is a good exercise, that is not the goal of this assignment. Rather, you will be exploring the impacts of different architectural choices (different numbers of layers, hidden units, types of regularization, etc.) on the performance of a neural net. Accompanying this mini-project is a support code base written in Python that uses Keras to implement a neural network that recognizes, and generates, handwritten digits (Figure 1). Most of your work will be focused on modifying the training script.

[1] Deep learning is synonymous with neural networks, and the number of layers to be considered deep can be as little as 2 hidden layers ($L = 3$).

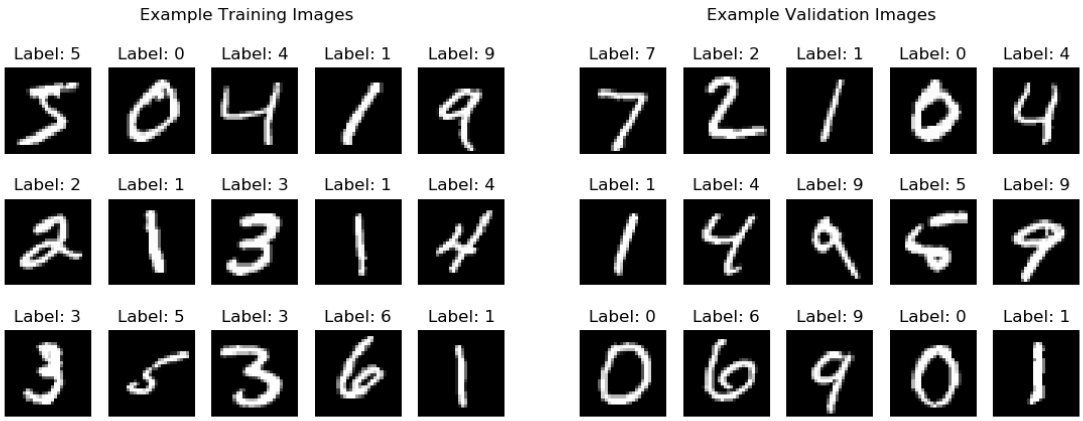


Figure 1: Example images from the MNIST dataset of handwritten digits.

We suggest that you use the `pip` package manager for Python 3 to run the code on your personal machine. This should come as a standard part of any Python 3 release. The support code has the following dependencies: `numpy`, `matplotlib`, `tensorflow`, `keras`, `simplejson`, `pillow`. To install a package `<pack>`, simply type `pip install <pack>` at the command line.

The support code for the first part is broken up into 3 scripts: `visualize.py`, `train.py`, and `evaluate.py` (see the Python archive). Use the `-h` command line option to see each of their usages. This is to reflect the typical workflow in machine learning and data science: you begin by exploring the type of data you have (images versus financial data, dimensionality, etc.), use that information to determine a model class to train on the data, and then evaluate the performance of that model. This is the way people typically modularize their code in industry. Why? If you decide that you need to make modifications to one part of the pipeline, you don't want to have to re-run everything else (which may be very time-consuming, as you will soon find out) to get new results.

For the second part, there is simply a demo script that you should run: `GAN_demo.py`. This particular script uses features (f-strings) from Python 3.6. The script will not save models by default, though it will output files of the generator's improving output throughout the training process. This part is meant to be mainly recreational, in order to explore cool new ideas without too much overhead and worry about implementation details. If you read through the code, it should tie in some of the topics from the first part and apply them in a unique way.

Finally, please refer to the Keras documentation to figure out what specific calls in the support code are doing. This will be good practice for learning how to pick up new tools in the future.

1 CPU vs. GPU (Optional)

Complete this section only if you have access to a GPU machine, or if you want to set up a GPU instance on AWS. Having access to a GPU machine for this assignment is highly recommended. Note that with your `.edu` email address, you are eligible for free AWS credits! This would be a good time to practice using the cloud, as it's becoming more and more prevalent in industry. If you choose to use AWS, we suggest setting up an EC2 GPU instance with the Conda Deep Learning AMI, as described here. You can then SSH into this machine and use it for anything you need.

If you choose to go this route, the only differences to the package installation instructions will be for installing TensorFlow and Keras. To do so, you would type `conda install -c anaconda tensorflow-gpu` and `conda install keras` at the command line. However, with the Deep Learning AMI, it shouldn't be necessary to install TensorFlow, only Keras. You can also find a step-by-step guide here.

We will not be supporting you to get GPU code running on your personal machine. However, if you are curious, you need to first install CUDA 10 (NVIDIA's proprietary GPU programming language) and the associated library for deep learning, cuDNN 7.6. This isn't too hard on Windows, but if you're working on Linux, this **may brick your machine!** Once this is done, you can just `pip install tensorflow-gpu`. The `tensorflow-gpu` package works as a Python front-end wrapper that invokes CUDA calls on the GPU behind the scenes. Other high level packages, such as Keras, will usually further abstract out tensorflow (which is a general purpose computational language) so that it is more easily suited for neural network applications.

Once you've set up a GPU machine, run the training script for the dense[2] and convolutional[3] neural networks given, both on a CPU on your personal machine and on the GPU machine provided. Note that you don't have to make any modifications to the source code to do this! How long does training take on each device?

2 Number of Parameters

Look at the neural net architecture specified by the `build_dense_net()` function in `train.py`. This is a deep neural net with 2 hidden layers, the first with 200 units, and the second with 100. We use weight-decay regularization[4] on the learned weights matrix (more on this later), and follow each layer with a ReLU nonlinearity (also known as an activation; $ReLU(s) = \max\{0, s\}$ for any signal $s \in \mathbb{R}$) instead of tanh(). The final dense layer is just to take a softmax[5] and produce a probability distribution over $\{0, 1, \ldots, 9\}$ (the possible labels we can assign an image).

[2] *Dense* means fully connected from layer to layer, as the neural-network model given in class.

[3] *Convolutional* is a special neural-network model explained later.

[4] Also known as ℓ^2 regularization since it uses the 2-norm of $\mathbf{w}$.

[5] *Softmax* is a generalization of the logistic function to multiple variables; $\theta_i(s_1, \cdots, s_K) = e^{s_i} / \sum_{k=1}^{K} e^{s_k}$ which transforms signals into probabilities.

Leave the depth of this model fixed for now. 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? 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?

3 Regularization

In this document, we will use 'loss' as synonymous with error. Hopefully, you saw that the huge neural net implemented in the support code achieves much lower training loss than validation loss (validation loss is the error on a hold-out dataset that we use as an estimate for out-of-sample performance). While the MNIST dataset is a toy problem that many models have an easy time performing well on, this disparity is the kiss of death for machine learning systems used on harder real-world tasks (like predicting financial markets and planning autonomous navigation, to name two). Specifically, this difference is a symptom of a complex model overfitting to the training data. The whole reason we have a hold-out dataset used exclusively for validation is to get a feel for how well our model generalizes out of sample.

One way to address this issue is through *regularization* (Lecture 12). Think of it as follows: instead of having our model minimize the training loss $L(\mathbf{w} \mid x, y)$ (which is the in-sample error of the hypothesis defined by $\mathbf{w}$) in a vacuum, we also penalize it for being too complex. The most straightforward way to do so is to minimize an *augmented* loss function $L(\mathbf{w} \mid x, y) + \lambda\|\mathbf{w}\|^2$ instead of the loss $L(\mathbf{w} \mid x, y)$ on its own.

To gain some intuition for this augmented loss, consider the simpler example of polynomial regression, where our model class is $f(x \mid \mathbf{w}) = w_0 + w_1 x + \ldots + w_k x^k$. If we have large coefficients w_i, then we can have a highly oscillatory polynomial, which is unlikely to represent the underlying data. By forcing $\|\mathbf{w}\|^2$ to be small, we reduce the model complexity. From the perspective of the linear model, this regularization corresponds to guaranteeing that the smallest eigenvalues of $X^T X$ are capped at some minimum size, λ, so that the largest eigenvalues of $(X^T X)^{-1}$ are capped at a maximum size of $1/\lambda$. So, when we solve for the weights, this maximum size cap carries through. See Figure 2 for a concrete example of minimizing $L(\mathbf{w} \mid x, y) + \lambda\|\mathbf{w}\|^2$ (where $L(\mathbf{w} \mid x, y)$ is mean squared error) for different regularization strengths λ.

So in this problem, 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 by specifying the `-r` command line option in `train.py`. Report learning curves for different choices of λ (just a few). Can you use regularization to get a similar result to one of the smaller neural nets you implemented in the previous part?

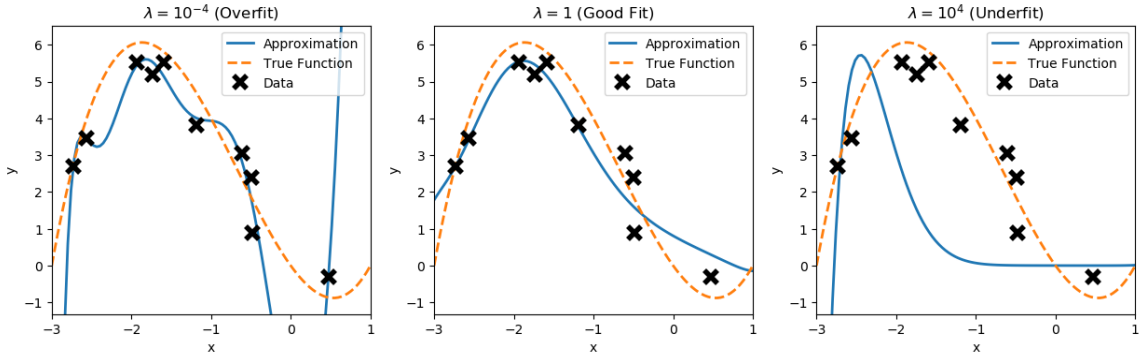


Figure 2: Fitting a 9^{th} order polynomial regression to data generated noisily from a cubic function. Small λ results in overfitting to the noise, intermediate λ results in good generalization, and large λ reduces model complexity too much to be useful.

4 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 $ReLU(s) = \max\{0, s\}$. 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 $ReLU(s)$.)

5 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.

6 Convolutional Neural Nets

Finally, we will discuss a different sort of regularization, known as an *inductive bias*. Inherently, a densely connected neural net is not a model class optimized for image data. It is unrealistic to model a connection between every pixel in an image. Dependencies in image data are much more local, i.e. adjacent pixels are much more likely to be correlated with each other than those found at opposite ends of an image. Moreover, images have spatial invariance: a face is a face, whether we find it at the top left, center, or bottom right of an image.

Without going too much into the specific details, a 2D convolution satisfies all these properties much better than a matrix multiplication. As a result, 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 `train.py` and `evaluate.py` 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.

7 Generative Adversial Network

7.1 Preliminaries

In 156a, most of the discussion is about supervised learning. All of the neural network techniques discussed above lie in that domain. However, the field of machine learning attempts to solve problems beyond that specific domain. One such problem is generative modelling. In generative modelling, we typically assume that the data that is given to us was generated by some sort of stochastic process, and consequetly we wish to model that process so that we can produce similar data points on our own, mimicking the real world.

For example, we could assume that a given set of data, X, was generated by independently sampling a gaussian distribution. We can build a generative model by creating a normal distribution with mean $\overline{X}$ and variance $\text{Var}(X)$, and then sampling from it independently. Models similar to this are useful in the realm of statistics, because they can help us understand information present in our data (e.g. hard drive failure rate by studing poisson processes). However, the cult of 'deep learning' wants to avoid making any hard assumptions; they wish to estimate this probabiltiy distribution directly! Furthermore, by easing away from a dependence on a fixed model we can use a much more descriptive model space.

Of course, increased complexity increases the difficulty of finding a good model. Consequetly, most such models either had significant issues fitting, or were too slow (in the case of Deep Bolzmann Machines). Today, we have acheived massive successes in applying neural networks to the generative problem, in particular image generation with convolutional networks. For example, such data distributions might be the space of all possible cat images described by 128x128 rgb photographs. Or, in MNIST's case the data distribution corresponds to 28x28 greyscale intensity values that represent handwritten digits. The three state of the art methods that solve this problem are Variational AutoEncoders (VAE), PixelCNN, and Generative Adversial Networks (GANs). GANs in particular typically produce the sharpest images, though PixelCNN is a strong contender. The demo generates

Due to the somewhat complex nature of setting up a GAN from scratch and the finicky nature of its hyperparameters during training, we will mostly focus on the idea that underpins them and seeing some modern(ish) results from the demo. These ideas are further explored in the other machine learning classes (such as 155 and 159).

7.2 How does a GAN work?

A GAN is actually two separate neural networks combined into one; a generator G and a discriminator D. The input of the generator comes from something called a latent space L, which is simply some abstract space where each point encodes the meaning of the generated value so $G : L \to X$. An eludicating example; Let $m, m_g, w \in L$. Suppose $G(m)$ correspond to an image of a man, $G(m_g)$ a man with glasses, and $G(w)$ a woman. Sometimes, $G(m_g - m + w)$ correspond to an image of a woman with glasses (this example was taken from Ian Goodfellow's 2016 NeurIPS GAN tutorial). The inputs to the discriminator come from the pixel space, X. The discriminator then maps that space to either true or false, which we represent as 0 and 1, $D : X \to \{0, 1\}$. The discriminator should determine whether an input from the pixel space is a member of the true distribution, e.g. it is a handwritten digit, or if it was made by the generator. Supposing that the generator's outputs follow some distribution, we can get that the loss function to the discriminator network is exactly the cross-entropy of incorrect classifications;

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

If we further define p_{model} as the distribution $G(z)$ follows, then

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

The generator should 'fool' the discriminator. Thus, the loss that it attempts to minimize is,

$$L^{(G)} = -L^{(D)}$$

Here, we see something interesting; since G and D are what we want to choose, a stable solution to those functions corresponds to something that appears like a saddle point for the above combination of equations! Basically, G is chosen such that the loss can't get any larger and vice versa D is chosen so that the loss cannot get any smaller. This corresponds to a game theoretic 'Nash-Equilibrium' (specifically a zero-sum, non cooperative game between the generator and the discriminator). As a theoretical exercise[6], lets see what this optimum looks like. (Beware, this is a bit more tricky than the rest of the extra credit. It is ok as long as you attempt it.)

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 $\frac{\delta L^{(D)}}{\delta D(X)}$ (similar to a vector derivative, don't worry about being rigorous). Use the following loss,

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

[6]courtesy of Ian Goodfellow, NIPS 2016

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!

2. Now that we have the derivative, we solve for $D(x)$ in terms of the probability distributions p_{model} and p_{data} when $\frac{\delta L^{(D)}}{\delta D(X)} = 0$. 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 $D(x)$

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?

It turns out that estimating this fraction is unique to how GANs operate, which might explain some of their unique features. But now, lets see it work in practice.

8 Running the Demo

In order to run the demo, simply run `python3 GAN_demo.py`. It should create a new folder in the working directory to store a sample of the generator's outputs after every epoch. Keep in mind that this will take a long time to run. Expect around 30 minutes to an hour if you are using a GPU and an overnight runtime on a CPU. How would you describe the training process in terms of the images that the model generated? What about the learning dynamics in the loss values for the generator and the discriminator?
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). Skimming over the code, did the current generator's implementation manage to avoid mode collapse?

9 Feedback

What did you think of this mini-project? What changes would you suggest we make to improve this for future iterations of the class? Were some parts of the assignment repetitive? If so, which ones?