CS231 Course Report

Liu Junnan

1 Introduction

In the past weeks I got dived into CS231 course and learned fundamentals on deep learning and specifically computer vision, including both supervised and unsupervised learning. Besides three assignments related to the course are finished with great effort but all of them turn out to be extremely useful to help understanding the ideas and principles of machine learning algorithms. In this report I will make a brief conclusion on what I have learned from CS231 course.

2 Supervised Learning Algorithms(Shallow, Classic)

In supervised learning, some "shallow" algorithms, like KNNs and SVMs, were widely used to solve classification problems before neural networks became overwhelming in this research field. KNNs do nothing but store all the data during training, and find the closest training data during predicting phase with Euclidean distance or other distance metrics. It is a simple algorithm and every beginner can understand pretty well. This course also talks about linear SVM, but some advanced topics on SVMs like soft margin and kernel tricks are not covered. Maybe it is because SVMs are less powerful than neural networks and thus not that popular now. Then softmax classifier was brought up, which is a pretty common loss function used in neural networks.

2.1 Algorithms

- KNN
- SVM
- Softmax

3 Feedforward Neural Networks

Neural networks became eye-catching and demonstrated its power in the last decade and with certainty will do so in the future. The course introduced neural networks by representing the architecture of a typical multi-layered feedforward neural network. A neural network model consists of two phases, training and inference/predicting. A training iteration consists of forward pass of computing matrix multiplications and activation functions, and backward pass of backpropagating gradients using the chain rule. Backpropagation algorithm plays a critical role in the training of neural network models. The course also introduced weight updating algorithms like SGD, SGD with momentum, Adam and so on.

3.1 Training Tricks

Hyperparameters of a neural network includes the architecture, learning rate, activation functions, regularization strength, update rules and so on. Tuning hyperparameters can be pretty tricky since there is no gold standard on how to set them properly, so researchers have to get their hands dirty to try all kinds of combinations of them to optimize the model. Besides, training a neural network model is time-consuming and resource-consuming. Here are some tricks concluded by CS231 course:

- Proper initialization. Some models can be initialization sensitive. The method proposed by He Kaiming are recommended. Batch normalization is also commonly used.
- Regularize the model to reduce overfitting. L2 regularization is a common form of regularization. Dropout is also common in CNN.
- Perform gradient checks.
- Overfit a tiny subset of data before training and make use zero cost is achieved.
- Draw plots on loss and iteration to make sure the process goes well.
- Consider update rules other than vanilla SGD.
- Search for hyperparameters on log scale. Prefer random search to grid search
- ...

4 Convolutional Neural Networks

Convolutional neural networks play a critical role in computer vision domain, because they improve the performance of almost all the CV tasks compared to traditional "shallow" machine learning methods. A typical convolutional neural network consists of a series of repeated Conv-ReLU layers and fully connected layers in the end. Because the input of CV problems, which are mostly images, have three dimensions of width, height and depth(RGB channels), a convolutional layer takes a 3d matrix as input and also yields a 3d matrix with the same depth of input. A convolutional layer has multiple filters, each of which convolves the input image, that is, slides over image spatially and computes dot products(imaging the 3d matrices are unraveled as 1d vectors. Actually the dot product of unraveled vectors is mathematically equal to the convolution operation of 3d matrices). A filter convolving the images produces a 2d matrix, and all the output of filters stacked along depth dimension produces the output of this convolutional layer.

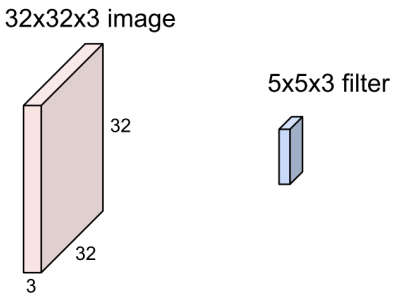


Figure 1: Example of convolutional Layer(with one filter)

Pooling layers are sometime used right after Conv-ReLU layers to squeeze the spatial size(height and width) of flowing tensors to lower the number of learnable weights of the remaining layers.

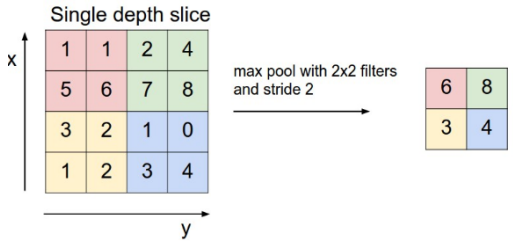


Figure 2: Example of pooling layer

The course listed some famous CNN architectures like AlexNet, VGGNet, GoogLeNet and ResNet, all of which are ImageNet competitions' winners. It then compared the space of memory that the

models used and number of weight parameters in each layers among those models, which made a good exercise to break down the models in more detail.

5 Recurrent Neural Networks and Long Short-Term Memories

RNN Unlike feed forward neural networks, recurrent neural networks take as their input not just the current input example they see, but also what they have perceived previously in time. The decision a recurrent net reached at time step $t-1$ affects the decision it will reach one moment later at time step t. So RNNs have two sources of input, the present and the recent past. The sequential information is preserved in the recurrent network's hidden state, which manages to span many time steps as it cascades forward to affect the processing of each new example. One way to think about RNNs is this: they are a way to share weights over time. Therefore RNNs are good at processing sequences like image caption and machine translation.

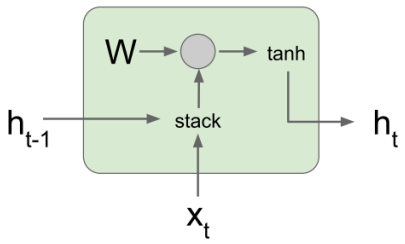


Figure 3: RNN unit

The forward pass of a RNN block can be mathematically described as:

$$h_t = f_W(h_{t-1}, x_t) = W_h^T \cdot h_{t-1} + W_x^T \cdot x_t \tag{1}$$

where h_t is the current hidden state at time step t; f_W the function – either sigmoid or tanh – that squashes the sum of the weight input and hidden state, making gradients workable for backpropagation; h_{t-1} the previous hidden state; x_t the input at the same time step.

Recurrent networks rely on an extension of backpropagation called backpropagation through time, or BPTT. Time, in this case, is simply expressed by a well-defined, ordered series of calculations linking one time step to the next, which is all backpropagation needs to work.

However, RNNs suffer a lot from vanishing/exploding gradient problems. Intuitively, backpropagation from h_t to h_{t-1} multiplies by W_h, so computing gradient of h_0 involves many factors of W_h. If the largest singular value of W_h is greater than 1, then exploding gradients would occur; on the contrary if the largest singular value of W_h if less than 1, then vanishing gradients would happen. Exploding gradients can be solved relatively easily, because they can be truncated or squashed. Vanishing gradients can become too small for computers to work with or for networks to learn – a harder problem to solve.

LSTM Long short-term memory was proposed to solve vanishing gradient problem of vanilla RNNs. LSTMs design complicated cells that help preserve the error that can be backpropagated through time and layers.

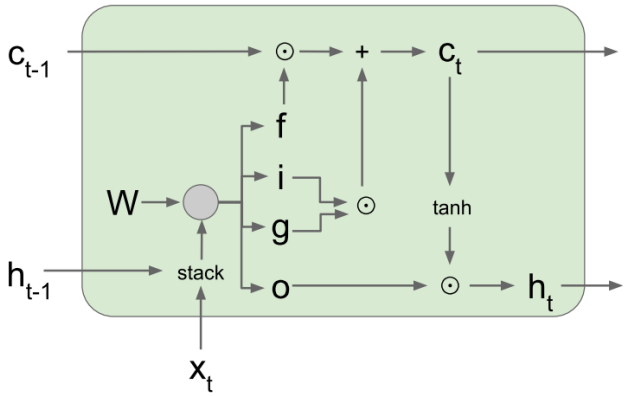


Figure 4: LSTM cell

The forward pass of a LSTM cell is defined as follows:

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t_1} \\ x_t \end{pmatrix} \tag{2}$$
$$c_t = f \odot c_{t_1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

where $\odot$ is element-wise multiplication.

As described in equation 2, all the computations involved, including sigmoid, element-wise multiplication and so on, are differentiable, which are suitable for backpropagation.

Those gates act on the signals they receive, and similar to the neural network's nodes, they block or pass on information based on its strength and input, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.

6 Generative Models

Neural network models we have learned so far, such as feed forward networks, CNNs, RNNs, LSTMs, are all basically discriminative models, which directly estimate posterior probabilities. Generative models focus on modeling class-conditional probabilistic distribution functions and prior probabilities with a good outcome that generative models can generate synthetic data points. Therefore generative adversarial networks become an active research field in the last few years due to this feature.

CS231 course discussed generative models like PixelRNN, PixelCNN, variational autoencoder and generative adversarial networks, and used them to generate instances or do other interesting tasks.

GAN consists of generator network and discriminator network. The generator network tries to fool the discriminator by generating real-looking images from random noise, while discriminator network tries to distinguish between real and fake images. These two networks are trained jointly in minmax game. The minmax objective function is defined as follows:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Training this minmax objective function can be separated apart as two functions:

1. Gradient ascent on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. Gradient descent on generator

$$\min_{\theta_g} \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \tag{3}$$

However in practice optimizing generator objective does not work well because when sample is likely fake, we want to learn from it to improve generator, but gradient of function $\log(1-x)$ near $x = 0$ is relatively flat. Therefore instead of minimizing likelihood of discriminator being correct(see equation3), now maximize likelihood of discriminator being wrong(see equation5). It's the same objective of fooling discriminator, but now there is higher gradient signal for bad samples so that it works much better.

1. Gradient ascent on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x\sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right] \tag{4}$$

2. Gradient ascent on generator

$$\max_{\theta_g} \mathbb{E}_{z\sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z))) \tag{5}$$

In the experiment these two objectives(equation 4 and 5) are negated as we will be minimizing these losses. More implementation details are covered in section 9.3.

7 Assignment 1

Assignment 1 requires us to implement KNN algorithm, SVM and softmax classifier.

7.1 KNN

In this part KNN is implemented with two loops, one loop and fully vectorized. KNN with two loops is naive and a nested for-loop will do. The one loop version just iterates test data, and computes the distances between this data and all the training data, with the help of broadcast feature of numpy.

```
1 for i in range(num_test):
2     dists[i, :] = np.sqrt(np.sum((X[i, :] - self.X_train)**2, axis=1))
3 return dists
```

These two versions are not so difficult, but the no loop version demands full vectorized computation, which makes it much trickier to implement. To be honest I didn't figure it out, so I had to search the Internet and then found the solution, which makes good use of broadcast feature of numpy. The code is as follows:

```
1 P = np.sum(X**2, axis=1)
2 Q = np.sum(self.X_train**2, axis=1)
3 PQ = X.dot(self.X_train.T)
4 dists = np.sqrt(np.transpose([P]) + Q - 2*PQ)
```

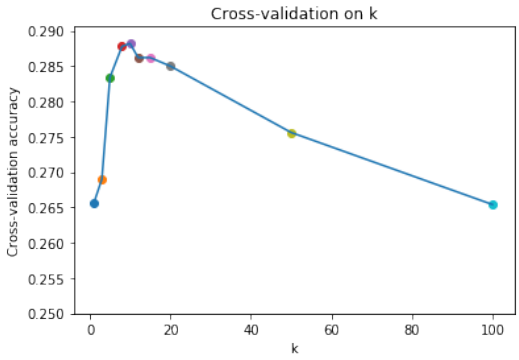
Let p denotes test data matrix, q training data matrix, and $dist$ distance matrix. The equation of distance is $(p-q)^2$. There is no way to just simple subtract test and training data and square the subtraction because they don't have the same shape. However, the solution applies the equation $(p \pm q)^2 = p^2 \pm 2pq + q^2$, and the term p^2 and q^2 are easy to vectorize, the middle term $2pq$ is dot multiplication of two matrix, so it is also vectorized. But actually these three terms have totally different shapes. In code line 1 above, P has shape (num_test,) — which in numpy indicates that this array has 1 dimension with size num_test. Similarly, Q has shape (num_train,), and PQ has shape (num_test, num_train). Here is the trick. if we compute $[P].T - Q$ instead, these two array will be broadcast, and yields result with shape (num_test, num_train).

The assignment then compare the time cost among these three versions of implementation. The result shows marvelous improvement that vectorization does.

The assignment also compares the influence of different Ks.

two loops	one loop	no loops
36.73	121.6	0.33

Table 1: Comparing KNN Performance



7.2 SVM, Softmax and Two layer Neural Network

The assignment compares the accuracy tested on CIFAR-10 dataset among linear SVM, softmax, and two layer neural network with L2 regularization.

	SVM	softmax	NN
accuracy	0.377	0.357	0.494

We can see that a simple two layer neural network can easily exceed SVM and softmax by 10 more percentage.

8 Assignment 2

Assignment 2 requires us to implement fully connected neural network and convolutional neural network in modular style and batch normalization layer.

8.1 Fully Connected Neural Network

In assignment 1 a fully-connected two-layer neural network is implemented on CIFAR-10. The implementation is simple but not very modular since the loss and gradient are computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models.

Then it compares different update rules like vanilla SGD, SGD with momentum, Adam and RMSProp. The results shows that vanilla SGD is worse than all the other more sophisticated ones on both speed and accuracy.

8.2 Batch Normalization

Figure 6 shows that batch normalization helps models to converge much faster.

Besides, as represented in figure 7, using batch normalization can make network less sensitive the the scale of weight initialization and therefore make it more stable.

The forward pass of batch normalization layer is easy to write codes, but the lecture doesn't derive the backward pass so I have to do it on my own. The naive backward pass can be further simplified because some internal variables are unnecessary to compute the final results. And the experiment shows that the simplified version is twice faster than naive one.

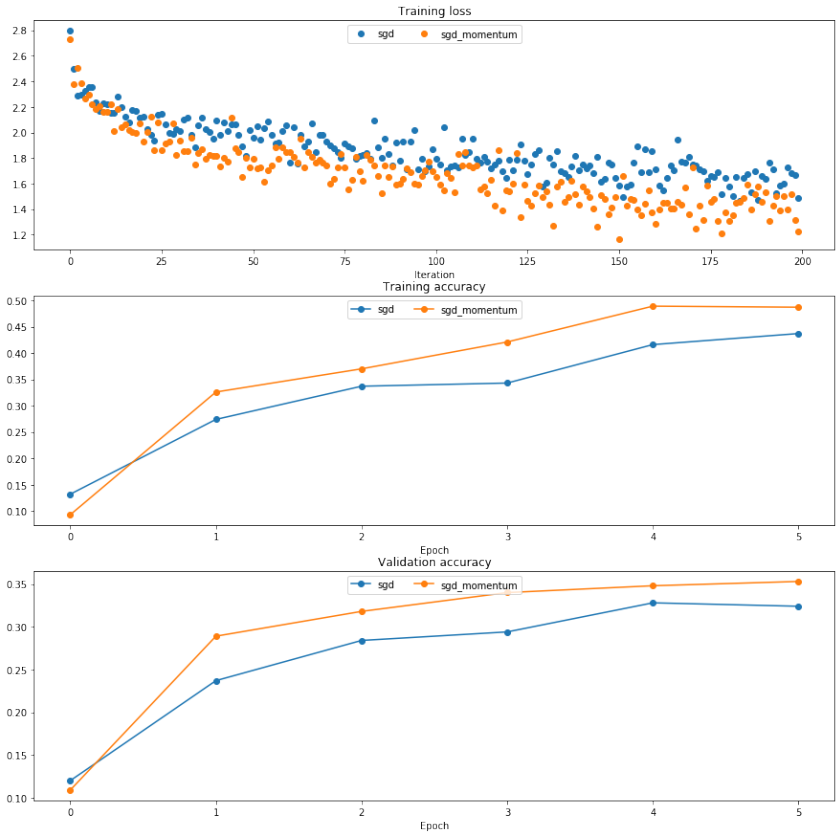


Figure 5: Comparing update rules

8.3 Convolutional Neural Network

The lecture doesn't talk much about the backpropagation of convolutional layer either. I didn't figure out how to derive it in the beginning so I searched some blogs and tutorials. But luckily I understood how to derive the gradients of convolutional layer in the end.

In the final part of assignments we are asked to train a model with whatever ConvNet architecture on CIFAR-10 and achieve at least 70% accuracy on the CIFAR-10 validation set within 10 epochs.

The default setting(3-layered ConvNet) achieves 50% accuracy, so there is a long way to go. First I tune the learning rate to find the optimal result. It turns out that setting learning rate to be 1.8e-3 will increase the accuracy to 59.8%. Then I insert batch normalization layers after each convolutional layer and stack more convolutional layers(6 conv layers in total), and all kernels have size 3×3. At the same time I set every two conv layers striding 2 instead of 1 to perform as dropout. The final 7-layered ConvNet trained in 10 epochs easily achieves 74.86% accuracy on validation set, outperforming the baseline by 4.86%. The model also has 74.8% accuracy on test set, showing that there is little overfitting problem.

9 Assignment 3

Assignment 3 requires us to implement image captioning with RNN and LSTM, style transfer, network visualization and GAN.

9.1 Image Captioning with RNN and LSTM

Implementing RNN and LSTM cells are not difficult because all the functions inside are basic and differentiable. In this part I implemented image captioning with a single RNN/LSTM cell, which

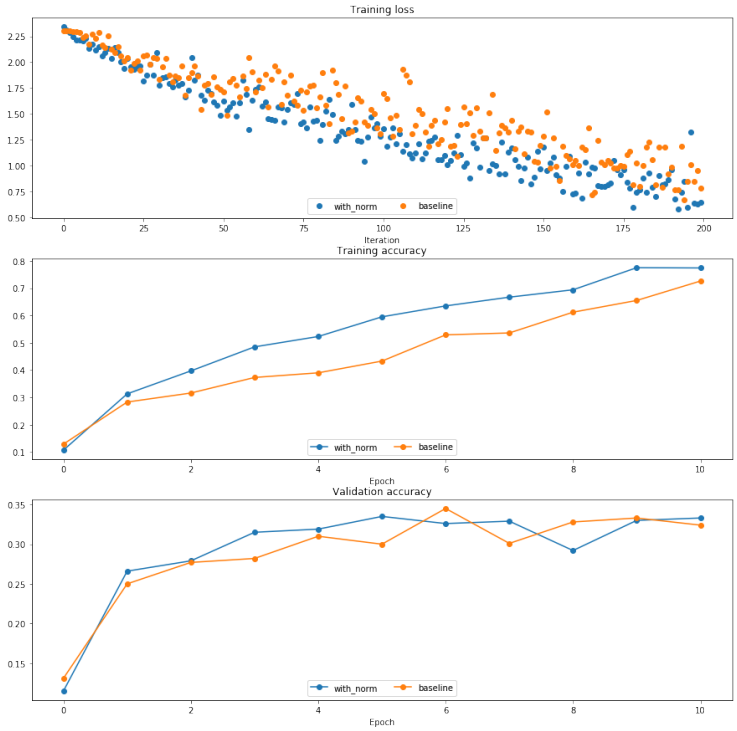


Figure 6: Batch Normalization helps model to converge faster.

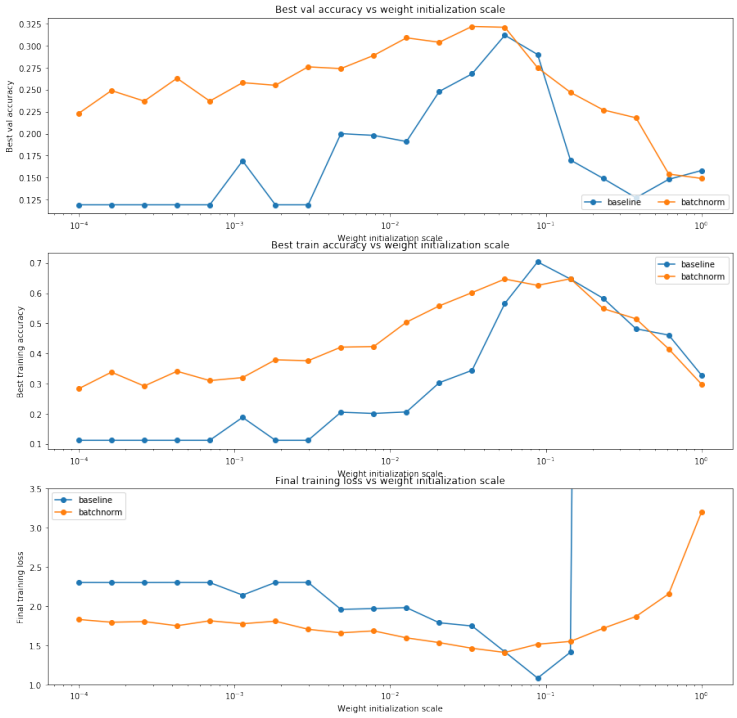


Figure 7: Batch normalization makes model more stable.

Figure 8: Image captioning with RNN

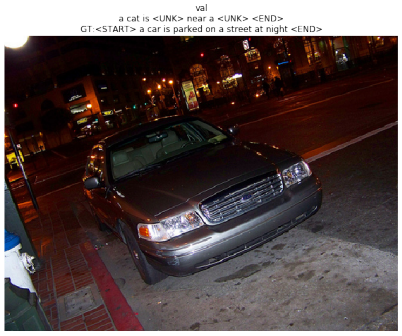


Figure 9: Image captioning with LSTM

turns out that they both can fit the training set but behave somewhat bad on validation set(shown in Figure 8 and 9).

9.2 Network Visualization

A saliency map tells us the degree to which each pixel in the image affects the classification score for that image. In this experiment we use pre-trained SqueezeNet to compute the scores given input image, and then compute gradients with respect to each pixels rather than parameters as we do in discriminative models. Figure 10 shows almost accurate saliency map of the given images and classes.

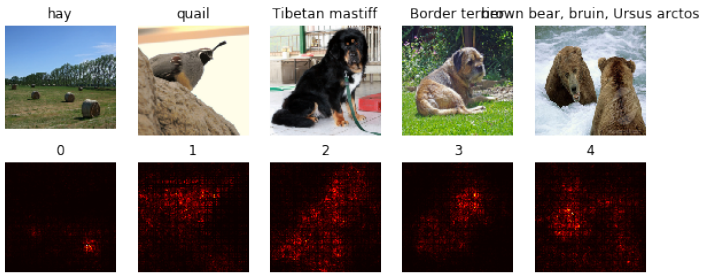


Figure 10: Saliency Map

9.3 Generative Adversarial Networks

In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the discriminator. We will train the discriminator to take images, and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the generator, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

Original GAN In this experiment we alter the loss function as well as architecture of discriminator and generator of GAN. In the original GAN, we use fully-connected discriminator and generator, and the generator loss and discriminator loss are as follows:

$$\begin{aligned} \ell_G &= -\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right] \\ \ell_D &= -\mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] - \mathbb{E}_{z\sim p(z)}\left[\log\left(1 - D(G(z))\right)\right] \end{aligned}$$

And we use binary cross entropy loss to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy

loss is

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

Figure 11 shows what it can generate after 3000+ iterations of training. Most of them have random noises and are hard to recognize for human.

LSGAN Least squares GAN alter the loss functions of discriminator and generator to make it more stable.

$$\begin{aligned} \ell_G &= \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z)) - 1)^2\right] \\ \ell_D &= \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[(D(x) - 1)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z)))^2\right] \end{aligned}$$

Figure 12 shows that the results are slightly better than the original GAN because most of them are recognizable, but still have some random noises.

DCGAN Deeply convolutional GANs use CNN in their discriminator and generator. The results shown in Figure 13 are pretty clear and without any random noise.

Figure 11: Original GAN

Figure 12: LSGAN

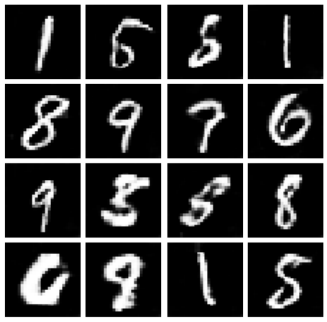


Figure 13: DCGAN

10 Conclusion

This course is a deep dive into details of the deep learning architectures with a focus on learning end-to-end models for visual recognition tasks, particularly image classification. From the course I learned to implement, train and debug my own neural network models and gained a detailed understanding of cutting-edge research in computer vision.