Dive into Conditional Generative Adversarial Networks and Deep Learning

Shaoxuan Chen, Advisor: Yves Atchadé

Abstract

The aim of this report is to summarize the research work that I did in 2020 summer under the guidance of Prof. Yves Atchadé, which basically including two topics: 1. The theory and math derivation of Conditional Generative Adversarial Networks(CGANs) and how it can be applied in regression, i.e., Adversarial Regression; 2. Basic Deep Learning topics that I tackled with, including Linear Neural Networks, Multilayer Perceptrons and basic Deep Learning computation in TensorFlow, Keras. The results of different experiments are also included, like comparing different methods in parameter approximation of linear/logistic regression, binary classification by Linear Neural Network and Multilayer Perceptrons, adjusting and checking the influence of different loss functions and hyperparameters, etc.

1 Conditional GANs

1.1 GANs & Regression

As for regression, if we have the data $\{x_i, y_i\}_{i=1}^n$ following the unknown true joint probability density function(pdf) $P_{data}(x, y)$, the aim of regression is to estimate the unknown true conditional distribution $P(y|x))$ and eventually do prediction if we have the new data came in, i.e., $P(y^*|x^*; x, y)$. However, there were some limitations of regression: 1.The parameters of the model were unknown so there might be too many solutions; 2.The true model is unknown, the underlying stochasticity of the model because of the random noise term ε; 3. The true model can be any functions even without closed forms. So the number of parameters can be very large, which will prevent the $X^T X$ from invertible.[1]

As for GANs, which contains the Generator and Discriminator. By feeding random noise to the Generator, it can generate the fake data follows an underlying pdf to fool the Discriminator. The Discriminator is fed by data both from Generator, i.e., the fake data, and data from true distribution. So it can be used to estimate the density by measuring the divergence of Generator distribution and true distribution for optimizing the objective function of Generator. Therefore, by combining GANs and Regression, we are also able to estimate the conditional distribution $P(y|x)$, which is also called Adversarial Regression.

1.2 Neural Network

Since GANs are neural network based, so neural network is also briefly mentioned this section. Below, as shown in Figure 1, are the two kinds of version of networks: shallow feedforward neural network and deep neural network.

We can think of neural network as non-linear multi-regression model. And it has three parts: the input layer, hidden layers and output layer. Each layer contains a number of units. For each units there basically happens two operations:

1. Each layer compute a linear combination of the outputs of the previous layer. Note: W represents the weight, b represents the bias, l represents the order of layers

$$z_j{}^{[l]} = [W_j{}^l]^T a^{[l-1]} + b_j{}^{[l-1]}$$

2. Linear combination $z_j{}^l$ transformed by non-linear activation function $g^{[l]}$, which is the output

$$a_j{}^{[l]} = g(z_j{}^{[l]}) = g[(W_j{}^{[l]})^T a^{[l-1]} + b_j{}^{[l-1]}]$$

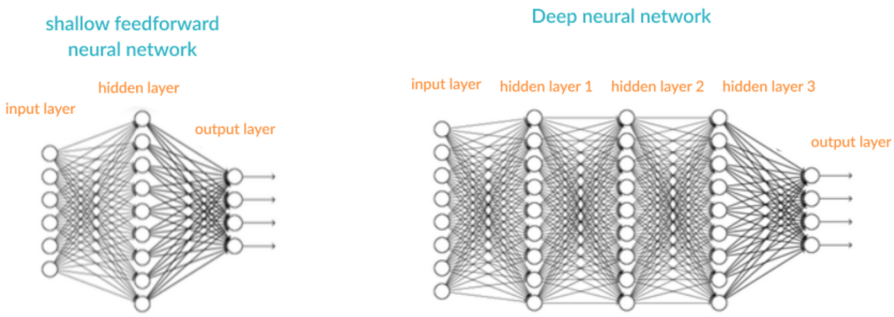

Figure 1: Simple Version of Neural Network. (reference link: missinglink.ai)

Therefore, combined what have been mentioned above, the whole neural network can be expressed as:

$$A^{[l]} = g(z^{[l]}) = g[(W^{[l]})^T A^{[l-1]} + b^{[l-1]}]$$

The non-linear activation functions allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality. [2] The common used non-linear activation functions are Sigmoid, ReLU, Leaky ReLU, Softmax activation functions.

1.3 Theory and math derivation of GANs

The GAN contains a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to $\frac{1}{2}$ everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation. There is no need for any Markov chains or unrolled approximate inference networks during either training or generation of samples.[3]

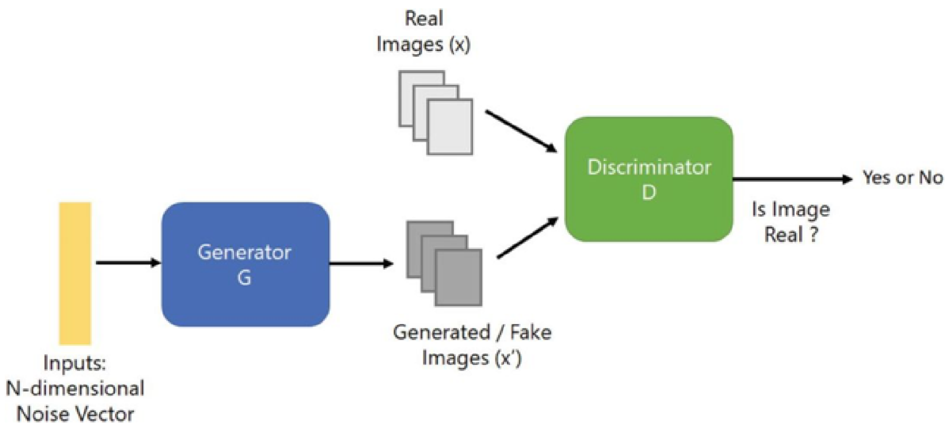

Figure 2: GANs basic framework (reference link: missinglink.ai)

Since the task of Discriminator D is about classification, so the form of the loss function should satisfy Cross Entropy loss function form. To be more specific:

$$V(D,G) = E_{x\sim P_{data(x)}}[logD(x)] + E_{Z\sim P_{Z(z)}}[log(1 - D(G(z)))]$$

The adversarial game comes from the Discriminator wants the D(G(z)) to decrease. But the Generator wants D(G(z)) to go up. Thus, the optimization equation satisfies the form:

$$\min_G \max_D V(D,G) = E_{x\sim P_{data(x)}}[logD(x)] + E_{Z\sim P_{Z(z)}}[log(1-D(G(z)))]$$

Below we want to show that, as for the optimization function, the minimum of cost function is achieved if and only if the probability distribution of Generator match the real data, i.e., the fake sample ≈ real sample.

1 > For Discriminator D:

Claim: ① For a fixed Generator, the maximum of cost function w.r.t discriminator is a constant V(D*,G) = -log(4)

Proof:

$$\begin{aligned} V(D,G) &= E_{x\sim P_{data(x)}}[logD(x)] + E_{Z\sim P_{Z(z)}}[log(1-D(G(z)))] \\ &= \int_x P_{data(x)} log(D(x))dx + \int_z P_{Z(z)} log(1-D(G(z)))dz \\ &= \int_x P_{data(x)} log(D(x))dx + \int_x P_{g(x)} log(1-D(x))dx \\ &= \int_x P_{data(x)} log(D(x)) + P_{g(x)} log(1-D(x))dx \end{aligned} \tag{1}$$

The inner part satisfies the form of:

$$alog(y) + blog(1-y)$$

If we want to find y to maximize it:

$$\frac{\partial alog(y) + blog(1-y)}{\partial y} = 0$$

$$\frac{a}{y} + \frac{b}{1-y}(-1) = 0$$

Then we have:

$$y = \frac{a}{a+b}$$

Then we plug in the original variable for a and b, we then recover the final form for the optimal discriminator, which can maximize the loss function:

$$D(x)* = \frac{p_{data(x)}}{P_{data} + P_{g(x)}}$$

For $P_{data(x)} = P_{g(x)}$, the pdf of real data and generated data are identical. So the optimal discriminator returns value:

$$D(x)* = \frac{p_{data(x)}}{2p_{data(x)}} = \frac{1}{2}$$

Plug in D(x)*=$\frac{1}{2}$into our cost function, we then have,

$$\begin{aligned} V(D,G) &= E_{x\sim P_{data(x)}}[logD(x)] + E_{Z\sim P_{Z(z)}}[log(1-D(G(z)))] \\ &= E_{x\sim P_{data(x)}}[log(\frac{1}{2})] + E_{Z\sim P_{Z(z)}}[log(\frac{1}{2})] \\ &= -log(2) - log(2) \\ &= -log(4) \end{aligned} \tag{2}$$

Now we have two conclusions: The first one is that the optimal discriminator $D(x)* = \frac{P_{data(x)}}{P_{data(x)}+P_{g(x)}}$; The second one is that when $P_{data(x)} = P_{g(x)}$, then we have $D(x)* = \frac{1}{2}$, $V(D*,G) = -log(4)$. That is to say, when distribution is equal, the upper bound of cost function is -log(4).

2 > For Generator G:

For Generator, we want to minimize the cost function V((D, G). Here we need to introduce the basic concept of Kullback-Leibler(KL) divergence and Jensen–Shannon(JS) divergence, which both of them represent the distance measure between two probability distributions.

KL Divergence:

$$D_{KL}(P||Q) = E_{x\sim P}[log\frac{P(x)}{Q(x)}]$$

JS Divergence:

$$JSD(P||Q) = \frac{1}{2}D_{KL}(P||\frac{P+Q}{2}) + \frac{1}{2}D_{KL}(Q||\frac{P+Q}{2})$$

Note that P, Q represents two probability distributions. The JS Divergence is similar to KL Divergence, except that it is symmetric, which means that the JS Divergence from P to Q is the same as the JS Divergence from Q to P. And this property does not hold in KL Divergence.

Notice that the form of KL Divergence is exactly the same as if we plug in the expression of optimal Discriminator, i.e.,

$$\begin{aligned} D_{KL}(P||Q) &= E_{x\sim P}[log\frac{P(x)}{Q(x)}] \\ &= E_{x\sim P}[log\frac{\frac{1}{2}P(x)}{\frac{1}{2}Q(x)}] \\ &= E_{x\sim P}[log\frac{P(x)}{Q(x)/2}] - log(2) \end{aligned} \tag{3}$$

Then we rewrite the cost function, we have:

$$V(D*, G) = -log(4) + KL(P_{data}||\frac{P_{data}+P_g}{2}) + KL(P_g||\frac{P_{data}+P_g}{2})$$

Combined with the form of JS Divergence, we have:

$$V(D*, G) = -log(4) + 2JSD(P_{data}||P_g)$$

Since the minimum of any JS Divergence is 0, and it occurs if and only if $P_{data(x)} = P_{g(x)}$, then for the objective function with optimal Discriminator plugged in, we have the minimum of the function:

$$\min_G V(D*, G) = -log(4)$$

So we know that the objective function V(D*, G) has one minimum -log(4), and the minimum is achieved when generator perfectly match the real data distribution.

1.4 Conditional GANs Algorithm

The framework of Conditional GANs is almost the same as the basic GANs. The only difference is that the input is the data together with labels instead of just data. More details of comparison between the objective function are shown below, where y represent the label:

GAN:

$$\min_G \max_D V(D, G) = E_{x\sim P_{data(x)}}[logD(x)] + E_{Z\sim P_{Z(z)}}[log(1 - D(G(z)))]$$

CGAN:

$$\min_G \max_D V(D, G) = E_{x\sim P_{data(x)}}[logD(x|y)] + E_{Z\sim P_{Z(z)}}[log(1 - D(G(z|y)))]$$

Below in Figure 3, which is cited in paper Conditional Generative Adversarial Nets[4], the framework of Conditional GAN is shown clearly.

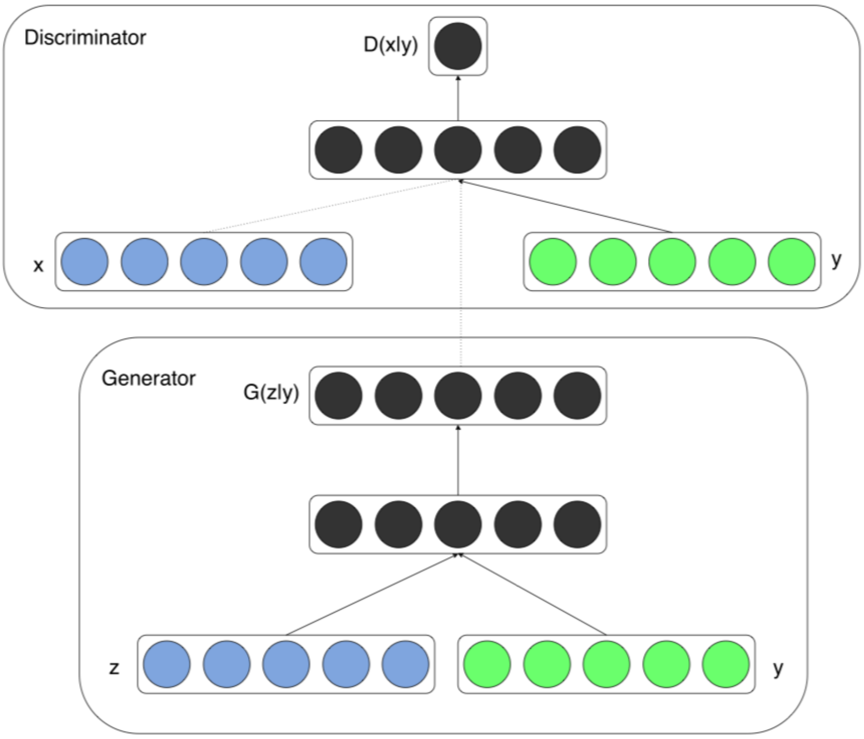

Figure 3: Conditional GANs basic framework[4]

As for Adversarial Regression, the algorithm of Conditional GANs for Regression is shown below:

Algorithm 1: Conditional GANs Algorithm for regression

Required: M the minibatch size. A prior function generating z. The hyperparameters d_{steps} and g_{steps}. An algorithm for gradient ascent. The number of iterations.

for *number of iterations* **do**
  for d_{steps} **do**
    Sample minibatch of labels x_1, ..., x_m from data $p_{data}(x) \sim$ Uniform(0,1);
    Produce sample data y_1, ..., y_m by corresponding x_1,...,x_m via the regression model, label with 1;
    Sample minibatch of examples z_1, ..., z_m from noise prior $p_g(z)$, transform with Generator to have $\{G(z_i)\}_{i=1}^{m}$, label with 0;
    Past the sample data and labels $\{x_i\}_{i=1}^{m}$ to Discriminator to get predictions $\{D(y_i \mid x_i)\}_{i=1}^{m}$ & $\{D(G(z_i \mid x_i))\}_{i=1}^{m}$;
    Update θ_D by ascending:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^{m} log(D(y_i \mid x_i)) + log(1 - D(G(z_i \mid x_i)))$$

  end
  for g_{steps} **do**
    Sample minibatch of examples z_1, ..., z_m from noise prior $p_g(z)$, transform with Generator to have $\{G(z_i \mid x_i)\}_{i=1}^{m}$;
    Update θ_G by ascending the non-saturating function:

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^{m} log(D(z_i \mid x_i))$$

  end
end

2 Computation Experiments

This section relates to the directed reading I did with Prof. Yves Atchadé of the e-book Dive into Deep Learning and the application of the knowledge I learned. It mainly covers the experiments I did related to parameter approximation by Mini-batch Stochastic Gradient Descent(SGD)/Stochastic Gradient Descent/analytic approach, and the experiments related to binary classification by Logistic Regression(Linear Neural Network) and Multilayer Perceptrons(MLP). Results of different hyperparameters were compared like learning rate, batch size, number of epochs. The performance of different methods were also compared in dealing with binary classification, like with/without L2 penalty terms in loss function, coding manually from scratch, concise implementation in Keras, etc.

2.1 Optimizer

Algorithm for SGD

Algorithm 2: SGD

Required: The loss function l. The number epochs. Size of data $\mathcal{N}$, learning rate η

Initialize parameters $(\mathbf{w}, b)$

for *number of epochs* **do**

  Compute gradient $\mathbf{g}$ on the whole data set$\leftarrow \partial_{(\mathbf{w},b)} \sum_{i\in\mathcal{N}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$

  Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta\mathbf{g}$

end

Algorithm for Mini-batch SGD

Algorithm 3: Mini-batch SGD

Required: $\mathcal{B}$ the minibatch size. The loss function l. The number epochs. learning rate η

Initialize parameters $(\mathbf{w}, b)$

Shuffle the data based on the batch size

for *number of epochs* **do**

  Compute gradient $\mathbf{g}$ based on the shuffled data $\leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i\in\mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$

  Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta\mathbf{g}$

end

2.2 Parameter Approximation of Linear Regression by Linear Neural Network

Model for generating the true data:

$$y = XW + b + \varepsilon,\ where\ X_i^\top \sim \mathcal{N}(\begin{pmatrix}0\\0\end{pmatrix}, diag(1,1)),\ W = [2, 3.5]^\top,\ b = 4,\ \varepsilon_i \sim N(0, 0.01)$$

Loss Function: Squared Loss Function

$$L = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

The results of different approaches/hyperparameters in parameter approximation of Linear Regression are shown in Table 1.

From below, as for Mini-batch SGD, we can see that for small fixed batch size, if the learning rate is too big, i.e., equals to 1. Then after the same number of epochs, the one with smaller learning rate have better performance than the one with larger learning rate. That may because that larger learning rate takes larger steps in finding the optimal point. If the learning rate is too big, instead of approaching the optimal point step by step, the gradient descent approach will bouncing around the optimal point and never reach the optimal point. Also, as for batch size in mini-batch SGD, we can see that it is better to use larger learning rate if the batch size is large, and smaller learning rate if the batch size is small. As for SGD, since it use the whole data set to do optimization, the result shows that larger learning have better performance. Also, comparing Mini-batch SGD with SGD, we can see

Table 1: Results of different approaches in Linear Regression parameter approximation(Note: $\sigma^2 = 1$)

Optimizer	lr	epochs	batch size	error in estimating w & b
Mini-batch SGD	0.03	5	10	[-0.00027895 -0.00031495], [0.00035882]
Mini-batch SGD	1	5	10	[0.00152123 0.00266123], [0.00423193]
Mini-batch SGD	0.03	5	50	[0.06462872 0.1595416], [0.16159701]
Mini-batch SGD	1	5	50	[-0.0001142 0.00045729], [-0.00081253]
SGD	0.03	5	\	[1.6979696 3.0017211], [3.4078]
SGD	0.5	5	\	[0.03197384 0.10298157], [0.09555578]
SGD	1	5	\	[-4.9352646e-05 -7.7486038e-05], [2.360344e-05]
Analytic approach	\	\	\	[-6.918907e-04 -4.553795e-05],[4.220009e-05]

that after adjustment, they can have almost the same performance in parameter approximation. The analytic approach also has very good performance.

2.3 Parameter Approximation of Logistic Regression by Linear Neural Network

Model for generating the true data:

$$log\frac{p}{1-p} = XW + b,\ where\ X_i^\top \sim \mathcal{N}(\begin{pmatrix}0\\0\end{pmatrix}, diag(1,1)),\ W = [2, 3.5]^\top,\ b = 4$$

Activation Function: Sigmoid Function

$$h(x) = \sigma(wx + b),\ where\ \sigma(z) = \frac{1}{1+e^{-z}}$$

Loss Function: Cross Entropy Loss Function

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i log a_i + (1-y_i) log(1-a_i)$$

$$a_i = sigmoid(z_i),\ where\ z_i = W^T x_i + b$$

Loss Function with L_2 penalty term:

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i log a_i + (1-y_i) log(1-a_i) + \frac{\lambda}{2}||\mathbf{w}||^2$$

$$a_i = sigmoid(z_i),\ where\ z_i = W^T x_i + b$$

Refined Loss Function built in TensorFlow: $tf.nn.softmax\_cross\_entropy\_with\_logits$

$$L = max(a_i, 0) - a_i * y + log(1 + exp(-abs(a_i)))$$

$$a_i = sigmoid(z_i),\ where\ z_i = W^T x_i + b$$

2.3.1 Results of using Cross Entropy Loss Function

The results of different approaches/hyperparameters in parameter approximation of Logistic Regression are shown in Table 2.

From the results shown below, we can see that Mini-batch SGD approach works well when the learning rate is relatively small. As the learning rate increase, even though the rate of converging to true values increase very fast, i.e., the true value of parameters are estimated relative well even at the first epoch, the value of the estimated parameters also fluctuate a lot. Since the objective function is a convex function, so the value of learning rate does not have too much effect. However, tuning the learning rate also make sure that we can have better results. Also, we notice that if we increase the

Table 2: Results of different approaches in Logistic Regression parameter approximation(Note: $\sigma^2 = 1$)

Optimizer	**lr**	**epochs**	**batch size**	**error in estimating w & b**
Mini-batch SGD	0.1	10	10	[0.04171479 -0.02211356], [0.08550978]
Mini-batch SGD	0.5	10	10	[0.08610749 -0.09116745], [-0.20888281]
Mini-batch SGD	1	10	10	[-0.03445697 0.09493375], [-0.39592457]
Mini-batch SGD	0.1	10	50	[0.37641394 -0.58421636], [0.65578675]
Mini-batch SGD	1	10	50	[0.01389563 -0.0771153], [-0.0222297]
SGD	0.01	100	\	[1.9053147 -3.225234], [3.9055605]
SGD	0.1	100	\	[1.4485135 -2.4557726], [2.7461467]
SGD	0.5	100	\	[0.82667005 -1.3771112], [1.5490735]
SGD	1	100	\	[0.5369544 -0.876415], [0.99448943]
SGD	1	500	\	[0.09156346 -0.10711193], [0.12568903]
SGD	1	1000	\	[0.05528092 -0.04446769], [0.05429029]

batch size, the one with larger learning rate have better performance. As for SGD approach, larger learning rate have better performance. And even using learning rate = 1, the estimated parameters converge very slow at the later steps.

However, as shown in Table 3, if we change the σ^2 to 100 instead of 1 when generating X. Neither the Mini-batch SGD nor the SGD approach work anymore.

Table 3: Results of different approaches in Logistic Regression parameter approximation(Note: $\sigma^2 = 100$)

Optimizer	**lr**	**epochs**	**batch size**	**error in estimating w & b**
Mini-batch SGD	0.1	10	10	[nan nan], [nan]
Mini-batch SGD	1	10	10	[nan nan], [nan]
SGD	0.1	100	\	[nan nan], [nan]
SGD	0.5	100	\	[nan nan], [nan]
SGD	1	100	\	[nan nan], [nan]

The reason of that may because the large variance when generating the data may cause the $1 - a_i$ term, i.e., parts including sigmoid function, in the loss function equals to zero, so the $log(1 - a_i)$ term will blow up to infinite, and the gradient of it will become extremely small, which is the problem of vanishing gradient. Below in Figure 4 also clearly show the rationale of that. So we can conclude that the non-refined loss function, i.e., Cross Entropy Loss Function, is not robust enough to deal with all kinds of data.

2.3.2 Results of using L_2 penalty terms in loss function.

The result of including L_2 penalty term in loss function is a standard techniques for regularizing models. However, the result of adding penalty terms in these experiments did not have better performance compared to no penalty term in loss function. So the penalty term might be better used in higher dimension data.

2.3.3 Results of using TensorFlow Built in Loss Function

The results of different approaches/hyperparameters in parameter approximation of Logistic Regression using the TensorFlow built in loss function, i.e., $tf.nn.softmax\_cross\_entropy\_with\_logits$, are shown in Table 4.

The results shown that if we use the refined loss function, i.e., the built in cross entropy loss function in TensorFlow, after tuning the hyperparameters, the true value of parameters can be approached at certain epochs, e.g., the Mini-batch SGD approach at epoch 2, SGD approach at epoch 1000. However,

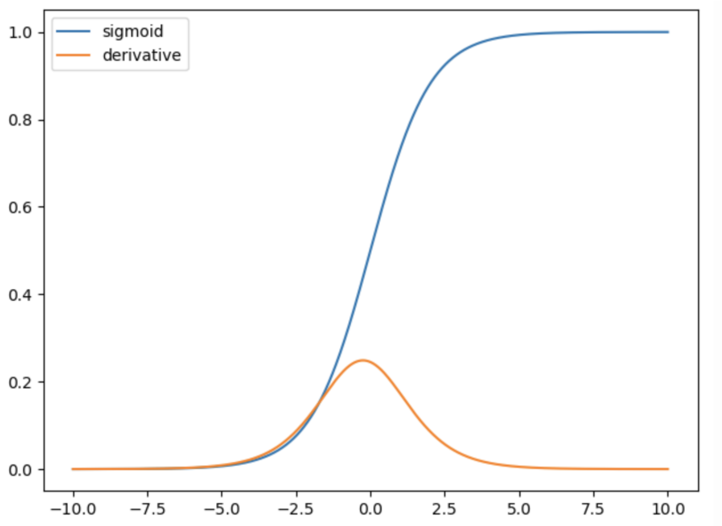

Figure 4: Sigmoid Function and the Gradient (reference link: Vanishing Gradient with Sigmoid activation in dnns)

Table 4: Results of different approaches in Logistic Regression parameter approximation(Note: $\sigma^2 = 1$)

Optimizer	lr	epochs	batch size	error in estimating w & b
Mini-batch SGD	0.5	2	10	[-0.1762023 0.4935906], [-0.26229382]
Mini-batch SGD	0.5	5	10	[-1.0851371 1.7294331], [-1.6065059]
Mini-batch SGD	0.5	10	10	[-1.7718003 2.7897825], [-2.9039044]
SGD	0.5	100	\	[1.2511575 -2.1367278], [2.7508674]
SGD	0.5	1000	\	[0.02554774 -0.05595708], [0.46379805]
SGD	0.5	1000	\	[-0.48387456 0.7847338], [-0.47854042]

in stead of converging to the true parameter, the estimated value of parameters continue increase or decrease as the number of epochs increase and they never converge. Also, the same experiments were also did if $\sigma^2 = 100$. The results shown that the built in loss function in TensorFlow can deal with data with large variance, i.e., no more vanishing gradient problems shown in the results. But the estimated results are also bad. One of the reason might be the true loss function is fundamentally changed to deal with the vanishing gradient problem. Thus, the optimal point of the objective function is also changed, which means the parameters we used in generating the true data are not the optimal parameters for the result anymore. So it is a more robust loss function to deal with data with more randomness at the expense of the accuracy.

2.4 Binary Classification by Linear Neural Network & MLP

Last section focus primarily on parameter approximation. This section focus on training and testing the model. The data was spilt into training & testing data set(note: training v.s testing = 4:1), and the model was trained on the training data set by Linear Neural Network/MLP approach and tested on the testing data. The results after certain number of epochs are shown in Table 5. And the plot of the result is also shown in Figure 5.

Linear Neural Network: define above in 2.3, using the Sigmoid Function as the activation function

Size of the data: 1000

Non-Linear Activation Function in MLP: Rectified Linear Unit(ReLU)

$$ReLU(x) = max(0, x)$$

MLP Model:

$$H = \sigma(XW^{(1)} + b^{(1)})$$
$$O = HW^{(2)} + b^{(2)}$$

Note: σ: the non-linear activation function; X$\in R^{n\times d}$, H$\in R^{n\times h}$, hidden-layer weights $W^{(1)}\in R^{n\times d}$, out-put layer weights $W^{(2)}\in R^{h\times q}$, biases $b^{(1)}\in R^{1\times h}$, biases $b^{(2)}\in R^{1\times q}$; d:number of inputs; h:number of hidden units; q:number of outputs; H:Hidden Layer; O: Output Layer

Loss Function: Refined Loss Function built in TensorFlow(defined above in 2.3)

Number of Hidden Layer, Output Layer: 1, 1

Number of Hidden Units: 256

2.4.1 Experiments did on data generated by Logistic Regression

Model for generating the true data:

$$log\frac{p}{1-p} = XW + b,\ where\ X_i^\top \sim \mathcal{N}(\begin{pmatrix}0\\0\end{pmatrix}, diag(1,1)),\ W = [2, 3.5]^\top,\ b = 4$$

The experiment was done by mainly two computing approaches: coding manually from scratch v.s. concise implementation in Keras. Both of the results are shown in Table 5. And the training, testing processes are also shown in Figure 5 & 6. From the results shown below, we can see that after certain number of epochs, the model was trained very well and the test accuracy reach up to 0.87. Among different approaches, we can see that the performance of them are almost the same. All of them have good test accuracy. Moreover, from the training and testing plot shown in Figure 5 and 6. Note that the train loss was scaled by 35 to fit in the plots. It is more clear in Figure 6 that there is a big jump at epoch 6, which is the reflection of the gradient descent process.

Table 5: Results of different approaches in Binary Classfication(Note: $\sigma^2 = 1$, LNN:Linear Neural Network, acc: accuracy, ns: not shown; manual: coding manually from scratch; Keras: concisely implemented by Keras)

Method	**Optimizer**	**lr**	**epochs**	**batch size**	**train loss**	**train acc**	**test acc**
LNN-manual	Mini-batch SGD	0.01	15	32	16.355714	0.8475	0.8705
MLP-manual	Mini-batch SGD	0.01	30	32	16.649528	0.8475	0.8705
MLP-Keras	Mini-batch SGD	0.01	10	32	16.655714	ns	0.8612

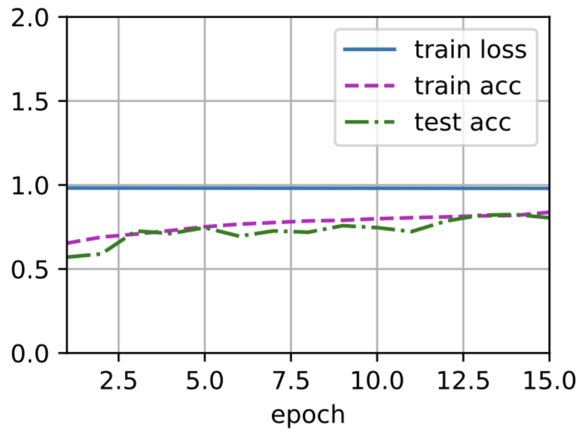

Figure 5: LNN-manual training/testing process

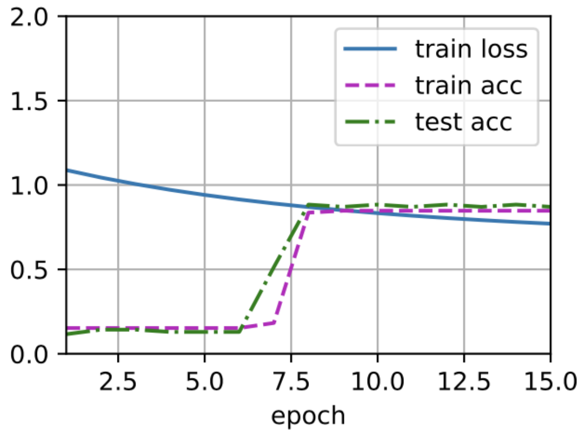

Figure 6: MLP-manual training/testing process

2.4.2 Experiments did on NBA data

The methods shown above were also applied in real life data—the 5-Year Career Longevity for NBA Rookies data, which includes 1340 entries and 19 features like Games Played, Points Per Game, 3 Point Attempts. The response variable is whether the career of the rookies is greater or equal to five years. The data was also centered and normalized beforehand. The results of different approaches are shown in Table 6. And the training, testing processes are also shown in Figure 7 & 8. Note that the train loss was also scaled by 35 to fit in the plots here.

From the results, we can see that after tuning the hyperparameters, all of the three approaches will eventually reach to almost the same test accuracy. However, overall the test accuracy of three approaches are relative low compare to the experiments did above. So real life data is relatively hard to handle with. From Figure 7 we notice that the test accuracy reach to relative high at the beginning and increase very slow later. The MLP method in Figure 8 shows clearly the increase of test accuracy.

Table 6: Results of different approaches in Binary Classfication(Note: $\sigma^2 = 1$, LNN:Linear Neural Network, acc: accuracy, ns: not shown; manual: coding manually from scratch; Keras: concisely implemented by Keras)

Method	Optimizer	lr	epochs	batch size	train loss	train acc	test acc
LNN-manual	Mini-batch SGD	0.001	10	64	17.035067	0.6523	0.6271
MLP-manual	Mini-batch SGD	0.001	10	32	18.476337	0.6544	0.6285
MLP-Keras	Mini-batch SGD	0.001	10	32	16.655714	ns	0.6230

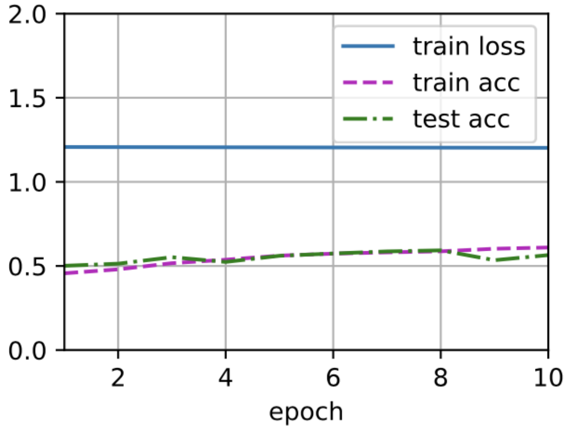

Figure 7: LNN-manual training/testing process

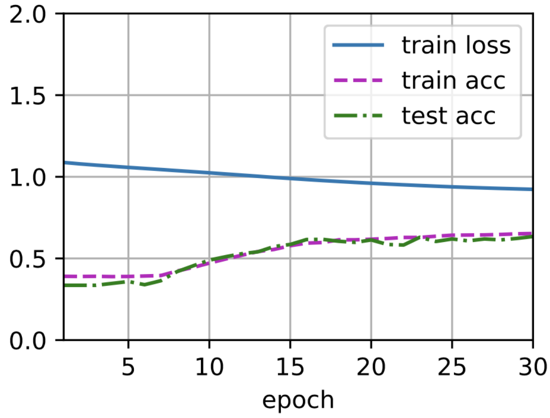

Figure 8: MLP-manual training/testing process

3 Discussion

As for the first part of this report, it is not hard to understand the math derivation of GANs. However, to really do some computing experiments related to GANs, or Adversarial Regression mentioned above, a bunch of computing background knowledge and Deep Learning topics should be mastered with before truly tackle with GANs.

As for the computing experiments did in the later part of the report. It is clear that the hyperparameters have big effects on the training/testing results. So tuning the hyperparameter is an very important task in Deep Learning model building. And if we tune the hyperparameter correctly, our models can reach the same accuracy via any methods. Also, there is still one thing which may need to put more effort which is the higher dimension data. In the experiments shown above, the dimension of data was small and all the approaches work well. But whether those approaches still work well need more systematically experiments to be done. Moreover, for binary classification, I still think the Cross Entropy Loss Function define in TensorFlow is not good enough. Even though it is robust enough to deal with any kind of data, the testing accuracy of that approach is relatively low. The data sets of this project are also relatively easy to deal with. More complicated experiments can be done in the future like dealing with image data.

Overall, this summer project gave me the sense of basic Deep Learning computation in Python and provided me with the chance to get to know the GANs. It also helped me realized the limitation of my current knowledge and push me to learn more to solid my background to tackle with more interesting problems!

References

[1] Yoann, B. Adversarial Regression. Generative Adversarial Networks for Non-Linear Regression: Theory and Assessment. ArXiv e-prints, Oct 2019

[2] Website: missinglink.ai. 7 Types of Neural Network Activation Functions: How to Choose?

[3] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative Adversarial Networks. ArXiv e-prints, June 2014

[4] Mirza, Mehdi Osindero, Simon. (2014). Conditional Generative Adversarial Nets.

Code in Python:

```python
""#linear Regression
%matplotlib inline
from d2l import TensorFlow as d2l
import TensorFlow as tf
import random
import numpy as np
import pandas as pd
import TensorFlow_probability as tfp
tfd = tfp.distributions
from sklearn.model_selection import train_test_split

num_classes = 1
num_features = 2
#data generating & processing
def logit_data(w, b, num_examples):
    """Generate y = Xw + b."""
    X = tf.zeros((num_examples, w.shape[0]))
    X += tf.random.normal(shape=X.shape, stddev=1)
    y = tf.matmul(X, tf.reshape(w, (-1, 1))) + b
    #y += tf.random.normal(shape=y.shape, stddev=0.01)
    y = tf.reshape(y, (-1, 1))
    return X, y

true_w = tf.constant([2, -3.4])
true_b = 4.2
x, y_in = logit_data(true_w, true_b, 10000)
p = 1/(1+tf.exp(-y_in))
bernoulli_distribution = tfd.Bernoulli(probs=p)
X=x
sample = bernoulli_distribution.sample(1)
y = sample[0] #choose 2 dimensions out of 3 in tensor
y = tf.dtypes.cast(y, tf.float32)

#data processing
##spilt data into training & testing  training: testing = 4:1
X_train, X_test, y_train, y_test = X[0:8000,:], X[8000:10000,:], y[0:8000,:], y[8000:10000,:]
# Feature Matrix  # Data labels
#X_test.shape, y_train.shape


#shuffle the data
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = tf.constant(indices[i: min(i + batch_size, num_examples)])
        yield tf.gather(features, j), tf.gather(labels, j)


#initialization
w = tf.Variable(tf.random.normal(shape=(num_features, num_classes), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros([num_classes]), trainable=True)
w,b
#activation function
def sigmoid(X):
    return 1 / (1 + np.exp(-X))  #note: use tf.sigmoid later
#model
def logistic(X, w, b):
    return tf.sigmoid(tf.matmul(X, w) + b)
                               # -1: keep block, use W.shape[0] to calculate left dimension
#net(X_train, w, b)

#loss function
def loss(y_hat, y):
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=y_hat, labels=y))

def loss1(y_hat, y):
    return tf.nn.sigmoid_cross_entropy_with_logits(logits=y_hat, labels=y)

def loss2(y_hat, y):
    return tf.losses.binary_crossentropy(
        y, y_hat, from_logits=True)
```

```

#loss function---manual
#def loss_manual(y_hat, y):
#    return (-(tf.reduce_sum(y_train*tf.math.log(logistic(X_train, w, b)) +
#                           (1-y_train)*(tf.math.log(1 - logistic(X_train, w, b)))))) /X_train.shape[0]
#def loss_manual(y_hat, y, batch_size):
#    return (-(tf.reduce_sum(y*tf.math.log(y_hat) +
#                           (1-y)*(tf.math.log(1 - y_hat))))) /batch_size
#def loss_manual_indi(y_hat, y):
#    return -(y*tf.math.log(y_hat) + (1-y)*(tf.math.log(1 - y_hat)))

def loss_check(X, w, b, y):
    return (tf.matmul(X, w) + b)-(tf.matmul(X, w) + b)*y + tf.math.log(1+tf.math.exp(-(tf.matmul(X, w) + b)))

def loss_check2(X, w, b, y):
    return -(tf.matmul(X, w) + b)*y + tf.math.log(1+tf.math.exp(tf.matmul(X, w) + b))

epsilon = 1e-14
def loss_check3(X, w, b, y):
    return -(tf.matmul(X, w) + b)*y + tf.math.log(1+tf.math.exp(tf.matmul(X, w) + b)+ epsilon)

def loss_check4(X, w, b, y):
    return -(tf.matmul(X, w) + b)*y + tf.math.log(tf.clip_by_value(1+tf.math.exp(tf.matmul(X, w) + b), clip_value_min=1e-15, clip_value_max=1.0))

#optimizer---sgd
def sgd(params, grads, lr, batch_size):  #@save
    """Minibatch stochastic gradient descent."""
    for param, grad in zip(params, grads):
        param.assign_sub(lr*grad/batch_size)
### minibatch sgd---loss

w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

lr = 0.1
num_epochs = 10
net = logistic
loss = loss_check2
batch_size = 10
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, X_train, y_train):
        with tf.GradientTape() as g:
            l = loss(X, w, b, y)  # Minibatch loss in `X` and `y`
        # Compute gradient on l with respect to [`w`, `b`]
        dw, db = g.gradient(l, [w, b])
        # Update parameters using their gradient
        sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(X_train, w, b, y_train)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}, w {tf.reshape(w, true_w.shape)}, b {b-0}')
# true W[2, -3.4], ture b [4.2]
### sgd---loss_check2 build in loss function

w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

lr = 1
num_epochs = 1000
net = logistic
loss = loss_check2
batch_size = 8000
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, X_train, y_train):
        with tf.GradientTape() as g:
            l = loss(X, w, b, y)  # Minibatch loss in `X` and `y`
        # Compute gradient on l with respect to [`w`, `b`]
        dw, db = g.gradient(l, [w, b])
        # Update parameters using their gradient
        sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(X_train, w, b, y_train)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}, w {tf.reshape(w,
```

```
    true_w.shape)}, b {b-0}')
# true W[2, -3.4], ture b [4.2]
### minibatch sgd--- l o s s 2  build in loss function

w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

lr = 0.1
num_epochs = 5
net = logistic
loss = loss2
batch_size = 10
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, X_train, y_train):
        with tf.GradientTape() as g:
            l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Compute gradient on l with respect to [`w`, `b`]
        dw, db = g.gradient(l, [w, b])
        # Update parameters using their gradient
        sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(net(X_train, w, b), y_train)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}, w {tf.reshape(w,
    true_w.shape)}, b {b-0}')
# true W[2, -3.4], ture b [4.2]
#### with panelty
#L2
def l2_penalty(w):
    return tf.reduce_sum(tf.pow(w, 2)) / 2

### minibatch sgd---loss

w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

lr = 0.1
num_epochs = 10
net = logistic
loss = loss_check2
batch_size = 10
lambd = 1
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, X_train, y_train):
        with tf.GradientTape() as g:
            l = loss(X, w, b, y) + lambd * l2_penalty(w)  # Minibatch loss in `X` and `y`
        # Compute gradient on l with respect to [`w`, `b`]
        dw, db = g.gradient(l, [w, b])
        # Update parameters using their gradient
        sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(X_train, w, b, y_train)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}, w {tf.reshape(w,
    true_w.shape)}, b {b-0}')
# true W[2, -3.4], ture b [4.2]

#1.2 logistic regression manual---nba data
#shuffle the data
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = tf.constant(indices[i: min(i + batch_size, num_examples)])
        yield tf.gather(features, j), tf.gather(labels, j)

#model
def net(X):
    return tf.sigmoid(tf.matmul(X, w) + b)


#optimizer---sgd
def sgd(params, grads, lr, batch_size):
    """Minibatch stochastic gradient descent."""
    for param, grad in zip(params, grads):
        param.assign_sub(lr*grad/batch_size)
```

```python

#loss function
def loss(y_hat, y):
    return tf.losses.binary_crossentropy(
        y, y_hat, from_logits=True)

#accuracy
def accuracy(y_hat, y):
    m = y.shape[0]
    Y_prediction = np.zeros((1, m))
    for i in range(y.shape[0]):
        if y_hat[i] > 0.5:
            Y_prediction[[0],[i]] = 1
        else:
            Y_prediction[[0],[i]] = 0
    #ac = tf.dtypes.cast(tf.reshape(Y_prediction, [m, -1]), tf.float32)-y
    return 1-tf.reduce_mean(tf.math.abs(tf.dtypes.cast(tf.reshape(Y_prediction, [m, -1]), tf.float32)-y))

#evaluate the model on the whole test dataset
def evaluate_accuracy(net, data_iter):
    metric = Accumulator(2)  # No. of correct predictions, no. of predictions
    for _, (X, y) in enumerate(data_iter):
        metric.add(accuracy(net(X), y), 1)  # use 1 to count the number of accumulation
    return metric[0] / metric[1]

def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Construct a TensorFlow data iterator."""
    dataset = tf.data.Dataset.from_tensor_slices(data_arrays)
    if is_train:
        dataset = dataset.shuffle(buffer_size=1000)
    dataset = dataset.batch(batch_size)
    return dataset
#batch_size = 10
#data_iter = load_array((features, labels), batch_size)
#next(iter(data_iter))

class Accumulator:  #@save
    """For accumulating sums over `n` variables."""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

#updater
class Updater():
    """For updating parameters using minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.params = params
        self.lr = lr

    def __call__(self, batch_size, grads):
        d2l.sgd(self.params, grads, self.lr, batch_size)
def train_mlp(net, train_iter, loss, updater):
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        with tf.GradientTape() as tape:
            y_hat = net(X)
            # Keras implementations for loss takes (labels, predictions)
            # instead of (predictions, labels) that users might implement
            # in this book, e.g. `cross_entropy` that we implemented above
            l = loss(y_hat, y)
        if isinstance(updater, tf.keras.optimizers.Optimizer):
            params = net.trainable_variables
            grads = tape.gradient(l, params)
            updater.apply_gradients(zip(grads, params))
        else:
```

```
            updater(X.shape[0], tape.gradient(l, updater.params))
        # Keras loss by default returns the average loss in a batch
        l_sum = l * float(tf.size(y)) if isinstance(
            loss, tf.keras.losses.Loss) else tf.reduce_sum(l)
        metric.add(l_sum, accuracy(y_hat, y), 1)
    # Return training loss and training accuracy
    train_loss = metric[0] / metric[2]
    train_accuracy = metric[1] / metric[2]
    return train_loss, train_accuracy

#return the test accuracy & print the result step by step
def train_test_mlp(net, train_iter, test_iter, loss, num_epochs, updater):
    for epoch in range(num_epochs):
        train_metrics = train_mlp(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        #animator.add(epoch + 1, train_metrics + (test_acc,))
        train_loss, train_acc = train_metrics
        print(f'epoch {epoch + 1}, train_loss {float(train_loss):f}, train_acc {train_acc-0},
    test_acc {test_acc-0}')
#plot & test accuracy
def train_mlp_plot(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """Train a model (defined in Chapter 3)."""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 2],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_with_plot(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics

# train loss & train accuracy
def train_with_plot(net, train_iter, loss, updater):  #@save
    """The training loop defined in Chapter 3."""
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        with tf.GradientTape() as tape:
            y_hat = net(X)
            # Keras implementations for loss takes (labels, predictions)
            # instead of (predictions, labels) that users might implement
            # in this book, e.g. `cross_entropy` that we implemented above
            l = loss(y_hat, y)
        if isinstance(updater, tf.keras.optimizers.Optimizer):
            params = net.trainable_variables
            grads = tape.gradient(l, params)
            updater.apply_gradients(zip(grads, params))
        else:
            updater(X.shape[0], tape.gradient(l, updater.params))
        # Keras loss by default returns the average loss in a batch
        l_sum = l * float(tf.size(y)) if isinstance(
            loss, tf.keras.losses.Loss) else tf.reduce_sum(l)
        metric.add(l_sum, accuracy(y_hat, y), 1)
    # Return training loss and training accuracy
    train_loss = metric[0] / metric[2]
    train_accuracy = metric[1] / metric[2]
    return train_loss/35, train_accuracy

class Animator:  #@save
    """For plotting data in animation."""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # Incrementally plot multiple lines
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda function to capture arguments
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

```

```
    def add(self, x, y):
        # Add multiple data points into the figure
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
num_classes = 1
num_features = 19
data_size = 1340
train_size = tf.cast(0.8*data_size, tf.int32)
nba = pd.read_csv("nba_logreg.csv") #shape: 1340x21
#fill in NA value with meanx
nba = nba.fillna(nba.mean())
# split into input and output columns
X, y = nba.values[:, 1:-1], nba.values[:, -1]
all_features = nba.iloc[:, 1:-1]
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
nba = all_features
X = nba.values
X = tf.convert_to_tensor(X, dtype = tf.float32)
y = tf.convert_to_tensor(y, dtype = tf.float32)
y = tf.reshape(y, [data_size,1])

# get the train/test data
X_train, X_test, y_train, y_test = X[0:train_size,:], X[train_size:data_size,:], y[0:train_size,:], y[train_size:data_size,:]
batch_size = 64
num_epochs = 10
lr = 0.001

#initialization
w = tf.Variable(tf.random.normal(shape=(19, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)


train_iter = d2l.load_array((X_train, y_train), batch_size)
test_iter = d2l.load_array((X_test, y_test), batch_size)

updater = d2l.Updater([w, b], lr)
train_test_mlp(net, train_iter, test_iter, loss, num_epochs, updater)
batch_size = 64
num_epochs = 10
lr = 0.001      #note: better result with 0.1
#initialization
w = tf.Variable(tf.random.normal(shape=(19, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

train_iter = load_array((X_train, y_train), batch_size)
test_iter = load_array((X_test, y_test), batch_size)

updater = d2l.Updater([w, b], lr)
train_mlp_plot(net, train_iter, test_iter, loss, num_epochs, updater)
  #note: the loss was scaled by 35 to fit in the plot
```

```



# regularization: L1/L2, regu parameter
###adding logistic Regression---manual data
num_classes = 1
num_features = 2
data_size = 1000
train_size = tf.cast(0.8*data_size, tf.int32)
#data generating & processing
def logit_data(w, b, num_examples):
    """Generate y = Xw + b."""
    X = tf.zeros((num_examples, w.shape[0]))
    X += tf.random.normal(shape=X.shape, stddev=1)
    y = tf.matmul(X, tf.reshape(w, (-1, 1))) + b
    #y += tf.random.normal(shape=y.shape, stddev=0.01)
    y = tf.reshape(y, (-1, 1))
    return X, y

true_w = tf.constant([2, -3.4])
true_b = 4.2
x, y_in = logit_data(true_w, true_b, data_size)
p = 1/(1+tf.exp(-y_in))
bernoulli_distribution = tfd.Bernoulli(probs=p)
X=x
sample = bernoulli_distribution.sample(1)
y = sample[0] #choose 2 dimensions out of 3 in tensor
y = tf.dtypes.cast(y, tf.float32)

#data processing
##spilt data into training & testing  training: testing = 4:1

X_train, X_test, y_train, y_test = X[0:train_size,:], X[train_size:data_size,:], y[0:
    train_size,:], y[train_size:data_size,:]
# Feature Matrix  # Data labels
#X_test.shape, y_train.shape
batch_size = 64
num_epochs = 15
lr = 0.001

#initialization
w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)


train_iter = d2l.load_array((X_train, y_train), batch_size)
test_iter = d2l.load_array((X_test, y_test), batch_size)

updater = d2l.Updater([w, b], lr)
train_test_mlp(net, train_iter, test_iter, loss, num_epochs, updater)
###
batch_size = 64
num_epochs = 15
lr = 0.001

#initialization
w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

train_iter = load_array((X_train, y_train), batch_size)
test_iter = load_array((X_test, y_test), batch_size)

updater = d2l.Updater([w, b], lr)
train_mlp_plot(net, train_iter, test_iter, loss, num_epochs, updater)
#1.3 logistic regression Keras---nba data
# mlp for binary classification
import TensorFlow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from TensorFlow.keras import Sequential
from TensorFlow.keras.layers import Dense
nba = pd.read_csv("nba_logreg.csv") #shape: 1340x21
#fill in NA value with meanx
```

```
nba = nba.fillna(nba.mean())
# split into input and output columns
X, y = nba.values[:, 1:-1], nba.values[:, -1]
# ensure all data are floating point values
X = X.astype('float32')
# encode strings to integer
y = LabelEncoder().fit_transform(y)
# split into train and test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# determine the number of input features
n_features = X_train.shape[1]
n_features
# define model
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, activation='sigmoid',kernel_initializer='he_normal',
    input_shape=(n_features,)))
# compile the model
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=2)
# evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=2)
                              #verbose:By setting verbose 0, 1 or 2 you just say how do you
    want to 'see'
                               #the training progress for each epoch.
print('Test Accuracy: %.3f' % acc)

#2.1 MLP manual
num_inputs, num_outputs, num_hiddens = 2, 1, 256
batch_size = 32
num_epochs = 15
lr = 0.01       #note: better result with 0.1
#initialization
W1 = tf.Variable(tf.random.normal(
    shape=(num_inputs, num_hiddens), mean=0, stddev=0.01))
b1 = tf.Variable(tf.zeros(num_hiddens))
W2 = tf.Variable(tf.random.normal(
    shape=(num_hiddens, num_outputs), mean=0, stddev=0.01))
b2 = tf.Variable(tf.random.normal([num_outputs], stddev=.01))

train_iter = load_array((X_train, y_train), batch_size)
test_iter = load_array((X_test, y_test), batch_size)

updater = d2l.Updater([W1, W2, b1, b2], lr)
train_mlp_plot(net, train_iter, test_iter, loss, num_epochs, updater)
#keras
# mlp for binary classification
import TensorFlow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from TensorFlow.keras import Sequential
from TensorFlow.keras.layers import Dense
# determine the number of input features
n_features = X_train.shape[1]
n_features
# define model
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(256, activation='relu', kernel_initializer='he_normal',
    input_shape=(n_features,)))
#model.add(tf.keras.layers.Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=2)
# evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=2)
                              #verbose:By setting verbose 0, 1 or 2 you just say how do you
    want to 'see'
                               #the training progress for each epoch.
print('Test Accuracy: %.3f' % acc)
num_classes = 1
num_features = 19
data_size = 1340
```

```
train_size = tf.cast(0.8*data_size, tf.int32)
nba = pd.read_csv("nba_logreg.csv") #shape: 1340x21
#fill in NA value with meanx
nba = nba.fillna(nba.mean())
# split into input and output columns
X, y = nba.values[:, 1:-1], nba.values[:, -1]
all_features = nba.iloc[:, 1:-1]
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
nba = all_features
X = nba.values
X = tf.convert_to_tensor(X, dtype = tf.float32)
y = tf.convert_to_tensor(y, dtype = tf.float32)
y = tf.reshape(y, [data_size,1])

# get the train/test data
X_train, X_test, y_train, y_test = X[0:train_size,:], X[train_size:data_size,:], y[0:
    train_size,:], y[train_size:data_size,:]
#without plot
num_inputs, num_outputs, num_hiddens = 19, 1, 256
batch_size = 32
num_epochs = 30
lr = 0.01

W1 = tf.Variable(tf.random.normal(
    shape=(num_inputs, num_hiddens), mean=0, stddev=0.01))
b1 = tf.Variable(tf.zeros(num_hiddens))
W2 = tf.Variable(tf.random.normal(
    shape=(num_hiddens, num_outputs), mean=0, stddev=0.01))
b2 = tf.Variable(tf.random.normal([num_outputs], stddev=.01))

train_iter = d2l.load_array((X_train, y_train), batch_size)
test_iter = d2l.load_array((X_test, y_test), batch_size)

updater = d2l.Updater([W1, W2, b1, b2], lr)
train_test_mlp(net, train_iter, test_iter, loss, num_epochs, updater)
#with plot
num_inputs, num_outputs, num_hiddens = 19, 1, 256
batch_size = 32
num_epochs = 30
lr = 0.01        #note: better result with 0.1
#initialization
W1 = tf.Variable(tf.random.normal(
    shape=(num_inputs, num_hiddens), mean=0, stddev=0.01))
b1 = tf.Variable(tf.zeros(num_hiddens))
W2 = tf.Variable(tf.random.normal(
    shape=(num_hiddens, num_outputs), mean=0, stddev=0.01))
b2 = tf.Variable(tf.random.normal([num_outputs], stddev=.01))

train_iter = load_array((X_train, y_train), batch_size)
test_iter = load_array((X_test, y_test), batch_size)

updater = d2l.Updater([W1, W2, b1, b2], lr)
train_mlp_plot(net, train_iter, test_iter, loss, num_epochs, updater)
#2.2 MLP Keras---nba
nba = pd.read_csv("nba_logreg.csv") #shape: 1340x21
#fill in NA value with meanx
nba = nba.fillna(nba.mean())
nba
# split into input and output columns
X, y = nba.values[:, 1:-1], nba.values[:, -1]
# ensure all data are floating point values
X = X.astype('float32')
# encode strings to integer
y = LabelEncoder().fit_transform(y)
# split into train and test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# determine the number of input features
n_features = X_train.shape[1]
n_features
# define model
```

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(10, activation='relu', kernel_initializer='he_normal', input_shape=(n_features,)))
model.add(tf.keras.layers.Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(X_train, y_train, epochs=30, batch_size=32, verbose=2)
# evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=2)
                              #verbose:By setting verbose 0, 1 or 2 you just say how do you want to 'see'
                               #the training progress for each epoch.
print('Test Accuracy: %.3f' % acc)
# make a prediction
row = [1,0,0.99,-0.889,0.853,0.036,0.898,-0.37,1,10,1,2,3,4,9,0.7,0.5,0.1,0.4]
yhat = model.predict([row])
print('Predicted: %.3f' % yhat)
#linear regression
%matplotlib inline
from d2l import TensorFlow as d2l
import TensorFlow as tf
import random
def synthetic_data(w, b, num_examples):
    """Generate y = Xw + b + noise."""
    X = tf.zeros((num_examples, w.shape[0]))
    X += tf.random.normal(shape=X.shape)
    y = tf.matmul(X, tf.reshape(w, (-1, 1))) + b
    y += tf.random.normal(shape=y.shape, stddev=0.01)
    y = tf.reshape(y, (-1, 1))
    return X, y

true_w = tf.constant([2, 3.5])
true_b = 4
features, labels = synthetic_data(true_w, true_b, 1000)
features
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = tf.constant(indices[i: min(i + batch_size, num_examples)])
        yield tf.gather(features, j), tf.gather(labels, j)
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = tf.constant(indices[i: min(i + batch_size, num_examples)])
        yield tf.gather(features, j), tf.gather(labels, j)
w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)
w, b
def linreg(X, w, b):  #@save
    """The linear regression model."""
    return tf.matmul(X, w) + b
def linreg(X, w, b):  #@save
    """The linear regression model."""
    return tf.matmul(X, w) + b
def sgd(params, grads, lr, batch_size):
    """Minibatch stochastic gradient descent."""
    for param, grad in zip(params, grads):
        param.assign_sub(lr*grad/batch_size)
w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)
lr = 1
num_epochs = 5
net = linreg
loss = squared_loss
batch_size =50
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        with tf.GradientTape() as g:
            l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Compute gradient on l with respect to [`w`, `b`]
        dw, db = g.gradient(l, [w, b])
        # Update parameters using their gradient
        sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(net(features, w, b), labels)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}')
print(f'error in estimating w: {true_w - tf.reshape(w, true_w.shape)}')
print(f'error in estimating b: {true_b - b}')
w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)

lr = 1
num_epochs = 5
net = linreg
loss = squared_loss
batch_size = 1000

for epoch in range(num_epochs):
    with tf.GradientTape() as g:
            l = loss(net(features, w, b), labels)
        # Compute gradient on l with respect to [`w`, `b`]
    dw, db = g.gradient(l, [w, b])
        # Update parameters using their gradient
    sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(net(features, w, b), labels)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}')
print(f'error in estimating w: {true_w - tf.reshape(w, true_w.shape)}')
print(f'error in estimating b: {true_b - b}')
#generate one column of ones to the left
x_left = tf.ones([1000, 1], tf.float32)
#combine the dataset with one column
X = tf.concat([x_left, features], 1)
X
```