[Jonathan Hui blog](https://jhui.github.io/)

[About](https://jhui.github.io/about/)

“Deep learning without going down the rabbit holes. (Part 2)”

Mar 17, 2017

**This is a work in progress…**

[Part 1 of the deep learning can be found here](https://jhui.github.io/2017/03/18/Deep-learning-tutorial/)

Overfit

In part one, we prepare a simple model with 4 layers of computational nodes. The solutions for WW are not unique. Should we prefer one solution over the other or should we select smaller values for WW? Does part of the network cancel out each other?

This leads us to a very important topic. When we increase the complexity of our model, we risk the chance of including the noise in the model. If we do not have enough sample data to cancel out the noise, we make bad predictions. But, even without the noise, we can still have a bad model. Let’s walk through an example. We start with training samples with input values and output range from 0 to 20. How will you create an equation to link the data points below?

One possibility is y=xy=x which is simple and just misses 2 on the left and 2 on the right.

But, when we show it to our android, Pieter, who has a much higher computational capability than us, he models it as:

y=1.9⋅10−7x9−1.6⋅10−5x8+5.6⋅10−4x7−0.01x6+0.11x5−0.63x4+1.9x3−2.19x2+0.9x−0.0082y=1.9⋅10−7x9−1.6⋅10−5x8+5.6⋅10−4x7−0.01x6+0.11x5−0.63x4+1.9x3−2.19x2+0.9x−0.0082

Which model is correct? The answer is “don’t know”. Someone thinks the first one is simpler. Simple explanations deserve more credit. But, if you show it to a stock broker, they may say the second curve looks closer to the market closing price of a stock.

Instead, we should ask whether our model is too “custom tailored” for the training data, and fails to make generalized predictions. The second curve fits the sample data completely, but will make poor predictions if the true model is a straight line.

Validation

**Machine learning is about making predictions.** A model that has 100% accuracy in training can be a bad model. For that, we split our testing data into three parts: say 80% for training, 10% for validation and 10% for testing. During training, we use the training dataset to build models with different hyperparameters. We run those models with the validation dataset and pick the one with the highest accuracy. This strategy works if the validation dataset is similar to what we want to predict. But, as a last safeguard, we use the 10% testing data for a final insanity check. This testing data is for a final verification, but not for model selection. If your testing result is dramatically different from the validation result, the data should be randomized more or more data should be collected.

Visualization

We can train a model to create a boundary to separate the blue dots from the white dots below. A complex model can produce sophisticated boundaries compared to a low complexity model. In the yellow-circled area, if we miss the two left white dot samples in our training, a complex model may create an odd shaped boundary just to include this white dot. A low complexity model may produce a smoother surface, which by chance may include those two white dots. A complex model may mishandle outliers. For example, a complex model should ignore the white dot in the green circle just as a simple model does.

Recall from Pieter’s equation, our sample data can be modeled nicely with the following equations:

y=1.9⋅10−7x9−1.6⋅10−5x8+5.6⋅10−4x7−0.01x6+0.11x5−0.63x4+1.9x3−2.19x2+0.9x−0.0082y=1.9⋅10−7x9−1.6⋅10−5x8+5.6⋅10−4x7−0.01x6+0.11x5−0.63x4+1.9x3−2.19x2+0.9x−0.0082

In fact, there are infinite answers using as many polynomial orders xk…xk…

Compared with the linear model y=xy=x, we realize that the ||coefficient||||coefficient|| in Pieter’s equation is higher. In addition, the higher the order, the harder it will be to train the model because we have a bigger search space for the parameters. In additional, some areas of the search space have a very steep gradient.

Let us create a polynomial model with order 5 to fit our sample data.

y=c5x5+c4x4+c3x3+c2x2+c1x+c0y=c5x5+c4x4+c3x3+c2x2+c1x+c0

We need more repetitions to train this model and that will result is less accuracy than a model with order three.

So why don’t we focus on making a model with the right complexity. In real life problems, a complex model is the only way to push accuracy to an acceptable level. Overfitting is unavoidable. You may even find it harder to train a model if the model is over simplified. A better solution is to introduce methods to reduce overfitting rather than make the model simpler. One simple solution is to add more sample data such that it is much harder to overfit. Here, double the sample data produces a model closer to a straight line. Unfortunately, labeling large training dataset in real problems can be expensive.

Regularization

**Regularization punishes over-complexity.** As we have observed before, there are many solutions to a DL problem. In order to have a close fit, the coefficient of our training parameters will have larger magnitude.

||c||=(c25+c23+c23+c22+c21+c20)−−−−−−−−−−−−−−−−−−−−−−√||c||=(c52+c32+c32+c22+c12+c02)

For example, if we set c2,c3,c4 and c5c2,c3,c4 and c5 to 0, we get a simple straight line model. To have our training not to overfit the training data, we add a penalty in the cost function to penalize large magnitude.

J=mean square error+λ⋅||W||J=mean square error+λ⋅||W||

This is called **regularization**. Here, we introduce another hyperparameter called regularization factor λλ to penalize overfitting.

In this example, we use a L2 norm (**L2 regularization**) ||W||||W|| as the penalty.

After many repetitions of trial and error, we pick λλ to be 1. With regularization, our model makes better predictions.

Like other hyperparameters for training, the process is trial and error. In fact, we use a relativly high λλ in this problem because there are only a few trainable parameters in the model. In real life problems, λλ is lower because we are dealing with larger amounts of trainable parameters.

There is another interesting observation. The loss may jump up sharply and drop to the previous level after a few thousand repetitions..

If we look into the equation, we realize the gradient

y=xiy=xi

∂y∂ci=ixi−1∂y∂ci=ixi−1

can be very steep which causes the learning rate problem discussed before. The cost here escalates very high which takes many repetitions to undo. For example, from iteration 10,000 to 11,000, the coefficient for x5x5 changes from -0.000038 to -0.000021 but the cost jumps from 82 to 34,312.

When we build our model, we also try out a polynomial model with order of nine. Even after a long training, the model still makes poor predictions. We decide to start with a model with order of threeand increase it gradually. This is another example to demonstrate why we should start with a simple model first. At seven, we find the model is too hard to train. The following is what a seven-order model predicts:

Diminishing and exploding gradient

**Cannot train a model if gradients explode or diminish.** From our previous example, we demonstrate the importantance of tracing the gradient at different layers to troubleshoot problems. In our online dating model, we log||gradient||||gradient|| for each layer.

A couple of things we need to be monitored. Is the magnitude too high or too small? If the magnitude is too high at a later stage of training, the gradient descent is having problems finding the minimum.Some parameters may be oscillating. For example, when we have the scaling problem with the features (years of education and monthly income), the gradient is so huge that the model learns nothing.

If the gradient is small, the network learns slowly. In the following log, the gradient diminishing from the right layer (layer 6) to the left layer (layer 0). Layer 0 is learning much slower than layer 6.

A network with many deep layers may suffer from this gradient diminishing problem. Let’s come back to back propagation to understand the problems.

The gradient descent is computed as:

∂J∂l1=∂J∂l2∂l2∂l1=∂J∂l3∂l3∂l2∂l2∂l1∂J∂l1=∂J∂l2∂l2∂l1=∂J∂l3∂l3∂l2∂l2∂l1

∂J∂l1=∂J∂l10∂l10∂l9⋯∂l2∂l1∂J∂l1=∂J∂l10∂l10∂l9⋯∂l2∂l1

As indicated, the gradient descent depends on the loss ∂J∂l∂J∂l as well as the gradients ∂lk+1∂lk,∂lk∂lk−1…∂lk+1∂lk,∂lk∂lk−1…. Let’s look at a sigmoid activation function. If xx is higher than 5 or smaller than -5, the gradient is close to 0. Hence, in these regions the node learns close to nothing regardless of the loss.

We can visualize that the derivative of a sigmoid function behaves like a gate to the loss signal. If the input is > 5 or <-5, the derivative is so small, it blocks most of the loss signal to propagate backwards. So, nodes on its left sides learn little.

In addition, the chain rule in the gradient descent has a multiplication effect. If we multiple numbers smaller than one, it diminishes quickly. On the contrary, if we multiple numbers greater than one, it explodes.

0.1⋅0.1⋅0.1⋅0.1⋅0.1=0.000010.1⋅0.1⋅0.1⋅0.1⋅0.1=0.00001

5⋅5⋅5⋅5⋅5=31255⋅5⋅5⋅5⋅5=3125

So, if the network design and the initial parameters have some symmetry that output similar values, the gradient may diminish quickly or explode. However, we cannot say with certainty when and how it may happen because we lack a full understanding between the math of gradient descent and a complex model. Nevertheless, the empirical data for deep network indicates it is a problem.

Microsoft Resnet (2015) has 152 layers. Many natural language process (NLP) problems are vulnerable to diminishing and exploding gradients. How can they address the issue? This is the network design for Resnet. Instead of one long chain of nodes, a mechanism is built to bypass a layer to make learning faster. (Source Kaiming He, Xiangyu Zhang … etc)

As always in DL, an idea often looks complicated in a diagram or an equation. In LSTM, the state of a cell is updated by

Ct=gateforget⋅Ct−1+gateinput⋅C~Ct=gateforget⋅Ct−1+gateinput⋅C~

Bypassing a layer can be visualized as feeding the input to the output directly out=inout=in. For CtCt to be the same as Ct−1Ct−1, we need gateforget=1gateforget=1 and gateinput=0gateinput=0. So one way to address the diminishing gradient problem is to design a different function used in the node.

Gradient clipping

To avoid gradient explosion, we can apply gradient clipping to restrict values of the gradient.

Here, we use TensorFlow for coding. TensorFlow is an open source machine learning software from Google. In real life problems, Numpy is important in data preparation but people use a software package like TensorFlow to implement deep networks.

Here we set the maximum clip norm to be 5.0.

If the gradient exceeds 5.0, the gradients are rescaled according to the ratio 5norm of the gradient5norm of the gradient. (L2 norm is the length of the vector.)

Classification

A very important part of deep learning is classification. We have mentioned face detection and object recognition before. These are classification problems asking the question: what is this? For example, for Android Pieter to safely walk in a street, he needs to learn what is a traffic light, and is the pedestrian facing him or not. Classification applies to non-visual problems also. We classify whether an email is a spam or does it approve/disapprove a loan etc…

Like solving regression problem using DL, we use a deep network to compute a value. In classification, we call this **a score**. We apply a classifier to convert the score to a probability value. To train the network, the training dataset provides the answers to the classification (like classifying an image as a school bus/truck/airplane) which we call **true label**.

Logistic function (sigmoid function)

A score computed by a network takes on any value. A classifier squashes it to a probability value between 0 and 1. For a “yes” or “no” type of prediction (the email is/is not a spam, the drug test is positive/negative), we apply a **logistic function** (also called a sigmoid function) to the score value. If the output probability is lower than 0.5, we predict “no”, otherwise we predict “yes”.

p=σ(score)=11+e−scorep=σ(score)=11+e−score

Softmax classifier

For many classification problems, we categorize an input to one of the many classes. For example, we can classify an image to one of the 100 possiblele object classes. We use softmax classifier to compute K probabilities, one per class for an input image (the combined probabilities remains 1).

The network computes K scores per image. The probability that an image belongs to the class ii will be.

pi=escorei∑escorecpi=escorei∑escorec

For example, the school bus above may have a score of (3.2, 0.8, 0) for the class school bus, truck and airplane respectively. The probability for the corresponding class is

pbus=e3.2e3.2+e0.8+e0=0.88pbus=e3.2e3.2+e0.8+e0=0.88

ptruck=e0.2e3.2+e0.8+e0=0.08ptruck=e0.2e3.2+e0.8+e0=0.08

pairplane=e0e3.2+e0.8+e0=0.04pairplane=e0e3.2+e0.8+e0=0.04

To avoid the numerical stability problem caused by adding large expotential values, we subtract the inputs by its maximum. Adding or subtract a number from the input produces the same probabilities in softmax.

softmax(z)=ezi−C∑ezc−C=e−Cezie−C∑ezc=ezi∑ezcsoftmax(z)=ezi−C∑ezc−C=e−Cezie−C∑ezc=ezi∑ezc

**logits** is defined as a mean to measure odd.

logits=log(p1−p)logits=log⁡(p1−p)

If we combine the softmax equation with the logits equation, it is easy to see that the score is the logit.

p=softmax(score)=ezi∑ezcp=softmax(score)=ezi∑ezc

That is why many literatures and APIs use the term logit for a score when softmax is used. However, more than one function that maps scores to probabilities and meet the definition of logits. Sigmoid function is one of them.

*Softmax is the most common classifier among others.*

SVM classifier

The Linear SVM classifer applies a linear classifier to map input to K scores, one per class.

y^=Wx+by^=Wx+b

The class having the highest score will be the class prediction. To train the network, SVM loss is used. We will discuss SVM in the cost function section later. Its main objective is to create a boundary to separate classes with the largest possible margin.

Entropy

*This section is about entropy in the information theory. Feel free to browse through it quickly.*

With a probabilistic model, we want a cost function that works with probability predictions. We need to take a break to address the information theory on entropy. It may be worth the time because entropy is heavily used in machine learning,

For example, suppose we have a string “abcc”, “a” and “b” occurs 25% (0.25) of the time and “c” with 50% (0.5). Entropy defines the minimum amount of bits to represent the string. For the most frequent character, we use fewer bits to represent it.

Entropy:

H(y)=∑iyilog1yi=−∑iyilogyiH(y)=∑iyilog⁡1yi=−∑iyilog⁡yi

The string “abcc” needs 1.5 bits per character. Here is our encoding scheme: 0 represents ‘c’, binary number 01 for ‘a’ and 10 for ‘b’. The average number of bit to represent the string is

H=0.25log(10.25)+0.25log(10.25)+0.5log(10.5)H=0.25log⁡(10.25)+0.25log⁡(10.25)+0.5log⁡(10.5)

H=0.25⋅2⋅2+1⋅0.5=1.5H=0.25⋅2⋅2+1⋅0.5=1.5

b **=** **-**( 0.25 **\*** math**.**log2(0.25) **+** 0.25 **\*** math**.**log2(0.25) **+** 0.5 **\*** math**.**log2(0.5) ) *# 1.5 bit*

Entropy is also a measure of randomness (disorder). A fair dice, compared with a biased dice, has more randomness with even distribution of outcomes. A biased dice is more predictable and therefore less entropic. In entropy, randomness means more information since it requires more bits to represent the information. For example, it takes more time to describe the details inside a messy room.

The entropy of a biased coin and a fair coin.

b **=** **-**(1.0 **\*** math**.**log2(1.0)) *# 0 : A 2-head coin*

b **=** **-**(0.5 **\*** math**.**log2(0.5) **+** 0.5 **\*** math**.**log2(0.5) ) *# 1 bit: 0 for head 1 for tail.*

Cross entropy

H(y,y^)=∑iyilog1y^i=−∑iyilogy^iH(y,y^)=∑iyilog⁡1y^i=−∑iyilog⁡y^i

Cross entropy is the amount of bits to encode y but use the y^y^ distribution to compute the encode scheme.The cross entropy is always higher than entropy until both distributions are the same. You need more bits to encode the information if you use a less optimized scheme. In classification, we make predictions using probabilities y^y^(0.88 chance for a school bus, 0.8 for a truck and 0.4 for an airplane.)

KL Divergence:

KL(y || y^)=∑iyilog1y^i−∑iyilog1yi=∑iyilogyiy^iKL(y || y^)=∑iyilog⁡1y^i−∑iyilog⁡1yi=∑iyilog⁡yiy^i

KL divergence is simply cross entropy - entropy: the extra bits needed to encode the information. In machine learning, KL Divergenence estimates the difference between 2 distributions. Since yy is the true labels, which do not change, we can treat it as a constant. Therefore finding a model to minimze KL divergence (the difference between probability distribution (1, 0, 0) & (0.88, 0.08, 0.04)) is the same as minimize the cross entropy. We, therefore, compute cross entropy most of the time.

*Use cross entropy to optimize a model.*

Maximum likelihood estimation (MLE)

*MLE helps you to understand where the cost functions come from. It is beneficial to know but not required.*

What is our objective in training a model? Our objective is to tune our trainable parameters so that the likelihood of our model is maximized (MLE). In plain terms, we want to train the parameters WW such that the prediction for the training data is close to the labels.

The likelihood is defined as the probabilty of making a prediction to be the same as the true labels give the input XXand WW. If we can find the WW to maximize the likelihood for all training data, we find our model. In probability, we write it as

p(y|x,W)=∏ip(yi|xi,W)p(y|x,W)=∏ip(yi|xi,W)

For each sample ii,

p(yi|xi,W)=yi^p(yi|xi,W)=yi^

which y1=(1,0,0)y1=(1,0,0) in our example (100% for school bus and 0% chance otherwise), and y1^y1^ = (0.88,0.08,0.04)(0.88,0.08,0.04).

Negative log-likelihood (NLL)

We want to take the log of the MLE because we can treat the product terms as additions, which are easier. Log is a monotonic increase function{herefore, finding xx to maximize f(x)f(x) is the same as find xx to maximize log(f(x))log⁡(f(x)).

Since probability is between 0 and 1 and its log is negative, we take a negative sign to make it positive.

nll=−logp(yi|xi,W)=−logyi^nll=−log⁡p(yi|xi,W)=−log⁡yi^

We call this the negative log-likelihood. To maximize the “maximum likelihood estimation” (MLE) is the same as minimizing the negative log-liklihood.

**To find**WW**, we minimize the negative log-likelihood**.

Logistic loss

As an exercise to demonstrate NLL, we derive the logistic loss (or even the mean square error) from NLL.

In logistic regression, we compute the probability by

p(yi|xi,W)=σ(zi)=11+e−zip(yi|xi,W)=σ(zi)=11+e−zi

Apply NLL,

nnl=−logp(yi|xi,W)=−log11+e−zj=−log1+log(1+e−zj)nnl=−log⁡p(yi|xi,W)=−log⁡11+e−zj=−log⁡1+log⁡(1+e−zj)

This becomes the logistic loss:

nnl=∑ilog(1+e−z)nnl=∑ilog⁡(1+e−z)

nnl=∑ilog(1+e−yiWTxi)nnl=∑ilog⁡(1+e−yiWTxi)

Cost function

**Deep learning is knowing your cost.** But, how do you define the cost? San Francisco is about 400 miles from Los Angeles. It costs about $60 for the gas. When you order food from a restaurant, they do not deliver to homes more than a few miles away. From their perspective, the cost grows exponentially with distance. So, there are many definitions of cost. In DL, our objective is not knowing the value of the cost, but finding a set of WW to make cost the lowest. Therefore, we have much flexibility for the cost function as long as we can find this set of WW. There are objectives to punish outliners heavily. But, there are other important considerations including how easy and how fast can we optimize the cost function. Does the cost function solve the gradient diminishing problem?

Cross entropy cost function

Apply NLL to find the cost function for a classification problem:

p(y|x,W)=∏np(yi|xi,W)p(y|x,W)=∏np(yi|xi,W)

nll=−log(p(y|W,x)=−∑nlogp(yi|W,xi)nll=−log⁡(p(y|W,x)=−∑nlog⁡p(yi|W,xi)

nll=−∑nlogyi^nll=−∑nlog⁡yi^

Since yi=(0,⋯,1,…,0,0)yi=(0,⋯,1,…,0,0) We can put back yiyi, and the equation becomes the cross entropy.

nll=−∑n∑iyilogyi^nll=−∑n∑iyilog⁡yi^

As a summary, the cross entropy cost function can be written as

H(y,y^)=∑iyilog1y^i=−∑iyilogy^iH(y,y^)=∑iyilog⁡1y^i=−∑iyilog⁡y^i

or as negative log likelihood

nll=−∑nlogyi^nll=−∑nlog⁡yi^

*Cross entropy cost function with softmax classifier is one of the most popular combinations to solve classification problems.*

SVM loss (also called Hinge loss or Max margin loss)

J=∑j≠yimax(0,scorej−scoreyi+1)J=∑j≠yimax(0,scorej−scoreyi+1)

If the margin between the score of a class and that of the true label is greater than -1, we add it to the cost. For example, a score of (8, 14, 9.5, 10) with the last entry being the true label.

J=max(0,8−10+1)+max(0,14−10+1)+max(0,9.5−10+1)J=max(0,8−10+1)+max(0,14−10+1)+max(0,9.5−10+1)

J=0+5+0.5=5.5J=0+5+0.5=5.5

For SVM, the cost function creates a boundary with the maximum margin to separate classes.

Mean square error (MSE)

mean square error=1N∑i(hi−yi)2mean square error=1N∑i(hi−yi)2

We have used MSE for regression problems before. We can use MSE in classification. But, in practice, we use cross entropy loss. Classification uses a classifier to squash values to a probability between 0 and 1. The mapping is not linear. For a sigmod classfier, a large range of values (less than -5 or greater than 5) is squeezed to 0 or 1. As shown before, those areas have to be close to 0 partial derivative. Based on the chain rule in the back propagation

∂J∂score=∂J∂out∂out∂score∂J∂score=∂J∂out∂out∂score

The loss signal is hard to propagate backward in those regions regardless of loss because ∂out∂score≈0∂out∂score≈0 . However, there is a way to solve this issue. The partial derivative of the sigmoid function can be small but we can make∂J∂out∂J∂out very large if the prediction is bad. The sigmoid function squashes values exponentially. We need a cost function that punishes bad predictions in the same scale to counter that. Squaring the error does not make it. Cross entropy punishes bad predictions exponentially. That is why the cross entropy cost function trains better than MSE in the classification problems.

Deep learning network (Fully-connected layers) CIFAR-10

Let’s put together everything to solve the CIFRA-10. CIFAR-10 is a computer vision dataset for object classification. It has 60,000 32x32 color images belonging to one of 10 object classes, with 6000 images per class.

(Source Alex Krizhevsky)

We implement a fully connected network similar to the following to classify the CIFRA images to the corresponding classes. In our implementation, we allow the user to control how many hidden layers to create and the number of nodes per layer.

Let’s have some boiler plate code that we did before. This is the forward feed and the back propagation code fory=Wx+by=Wx+b and the ReLU.

We combine them to form a forward feed and a backpropagation “affine relu” layer.

Our softmax function

We are creating a FullyConnectedNet network with 3 hidden layers with (100, 50, 25), nodes respectively.

Here is the key part in computing the loss. We do a feed forward, use softmax to compute the loss, and compute all the gradients for the back propagation. The code should be familiar, and the comment should be self-explanatory.

Here is the code of training the model:

And, the code of making predictions:

We will not show the code of preparing the data and the code performing the gradient descent. For the gradient descent, people use out of the box libraries to solve it. We will demonstrate it using TensorFlow in the next section.

**Deep learning is about creating a model by learning from data**. We have solved a visual recognition problem that is otherwise difficult to solve. Instead of coding all the rules, which is impossible for the CIFRA problem, we create a FC network to learn the model from data. The accuracy of our model is reasonable using the FC layer, but easily beat by adding convolution layers. Hence, we will not spend more effort for now.

MNist

One of the first deep learning datasets that most people learn is the MNist. It is a dataset for handwritten numbers from 0 to 9.

We will show the TensorFlow code to solve the problem with 98%+ accuracy.

Unlike the code with numpy, TensorFlow constructs a graph describing the network first. Here, we declare a placeholder for our input features (the pixel values of the image) and the labels, which will be provided later in the training.

We declare WW and bb as variables and provide methods to initiate it later.

We define 2 hidden layers. Each has a matrix multiplication operation followed by ReLU. Then, another operation multiplies it with a matrix. Note, so far we are just making declarations, no variables are initialized and no matrix multiplication is done.

Now, we define our loss function including a cross entropy and the regularization penalty for our WW. We use Adam as an optimizer for the gradient descent. We also have a placeholder for λλ so a user can supply it later to control the regularization.

We created a session to execute the graph and train the network for 10,000 iterations. For each repetition, we retrieve the next batch of sample data and run the operation “train step” (the Adam optimizer). TensorFlow runs the operation as well as all operations that it depends on.

Once the training is complete, we create 2 more operations, the correct\_prediction compare the predictions with the true labels. “Accuracy” computes how many predictions are correct.

We run the accuracy operation with our testing dataset and print out the results.

For completeness, here is the code listing. This file depends on “tensorflow.examples.tutorials.mnist” which is used to read the MNist data.

This code demonstrates the power to solve a complex visual problem with few lines of DL code. With 10,000 iterations, we achieved an accuracy above 98%.

Regularization

We apply regularization to overcome overfit. The idea is to train a model that makes generalized predictions. It forces the model not to memorize the small bits of an individual sample that is not part of the generalized features. Part of the overfit problem is that the model is too powerful. We can reduce the complexity of the model by reducing the number of features: remove features that can be directly or indirectly derived by others. We can also eliminate some layers or switch to a design that explores better on the locality of the information. For example, we use CNN for images to explore the spatial locality and LSTM for NPL. Overfit can be overcome by adding more training data also.

L0, L1, L2 regularization

Large W tends to overfit. L2 regularization adds the L2 norm to a cost function as the regularization cost.

J=data cost+λ⋅||W||J=data cost+λ⋅||W||

||W||2=∑i∑j(W2ij)−−−−−−−−−−√||W||2=∑i∑j(Wij2)

There are different functions to compute the regularization cost of tunable parameters:

L0 regularization

||W||0=∑{1,0,if w≠0otherwise||W||0=∑{1,if w≠00,otherwise

L1 regularization

||W||1=∑|w|||W||1=∑|w|

L0, L1 and L2 regularization penalize on WW but on different extents. L2 puts the highest attention (or penalty) on large parameters and L0 pays attention on non-zero parameters. L0 penalizes non-zero parameters, which forces more parameters to be 0, i.e. increase the sparsity of the parameters. L2 focuses on large parameters; therefore, we have more non-zero parameters. L1 regularization also rewards sparsity. Sparsity of parameters effectively reduces the features in making predictions. Sometimes it is used as feature selections. L2 is more popular but some problem domains may prefer higher sparsity.

Dropout

A non-intuitive regularization method called dropout discourages weights with large values. To avoid overfit, we may not want some weights to be too dominating. By randomly dropping connections from one layer to the other layer, we force the network not to depend too much on a single node and try to learn from many different ways. This has an effect similar to forcing the weights smaller.

In the following diagram, for each iteration, we randomly drop off the connection during training.

Here is the code to implement the dropout for the forward feed and back propagation. In the forward feed, it takes a parameter on the percentage of nodes to be dropout. Notice that, dropout applies to training only.

Weight initialization

Weight initialization is one important area in implementing a network. If you start the parameters incorrectly, you may not even beat the random odd of guessing. The initial input to the activation function (or non-linear functions) should not fall into its low partial derivative areas. Otherwise, the network learns slowly. Sometimes those parameters are accidentally initialized with 0s. This is close to turning every neuron dead and not able to back propagate the loss correctly. In fact, you do not want the output values for the next layer to look the same. Some non-symmetry is preferable, otherwise, the loss will evenly distribute back to previous layers. To introduce such non-symmetry, we initialize those parameters with Gaussian distribution with mean = 0. But should we use σ=1σ=1

The value to the activation is produced by z=Wx+bz=Wx+b. We will demonstrate that if WW has σ=1σ=1, zz will be much flatter with high variance value.

We generate 20,000 values of WW with mean = 0 and σ=1σ=1. Then we plot the distribution of W and re-calculate the variance again.

σ=0.998σ=0.998

We generate 20,000 values of yy using our familiar formula with 1000 of input xx in which half of them are 1 and another half are 0:

y=Wx+by=Wx+b

We plot the distribution of yy.

σ=497.6σ=497.6

The plot has different scales for the x and y-dimension. If we decrease the scale in y-direction a little bit, you will realize the plot is actually close to flat.

The plot is flat. i.e. it is more uniform. Therefore to generate an input to the activation function with a gaussian distribution of mean = 0 and σ=1σ=1, we need to take into the account of the number of inputs to the node. Hence, we use the following formula for the variance:

2number of input−−−−−−−−−−−−−√2number of input

Note: Some research paper indicates using 2 as numerator has better performance than 1.

Training parameters

In previous sections, we discuss many problems in training a network and how bad learning rate produces bad predictions. We now come back to the gradient descent and discuss different methods for updating the trainable parameters. This is not an easy topic because the shape of the cost function can be very different in different problem domains. Fortunately, most DL software libraries provide many different optimization methods. We will cover a couple of core concepts here.

Rate of decay

To maintain a constant learning rate is not be a good idea. It is like using a saw to finish the last part of making a table. One common way to reduce the rate is after some initial phase, we start to decay the learning rate for every N iterations. For example, after 10,000 iterations, the learning rate will be decay by the formula below for every 20,000 iterations:

learning rate=learning rate⋅decay factorlearning rate=learning rate⋅decay factor

which decay factor is another hyperparameter say 0.95.

Momentum update

We mentioned that gradient descent is like dropping a ball in a bowl. But, our gradient descent adjusts the parameters by the gradient of the current location of WW only. In the physical world, the movement of the ball not only depends on the location, but also the velocity of the ball. We could adjust WW by the gradient and its history rather than throw all the information away. If we recall the stochastic gradient descent, it follows a zip zap pattern rather than a smooth curve. With this historical information, we can make stochastic gradient or mini-batch gradient to behave more smoothly.

Here we introduce a variable vv, which behaves like the velocity (momentum) in the physical word. In each iteration, we update vv by keeping a portion of v minus the change caused by the gradient at that location. mumucontrols how much history information to keep, and this will be another hyperparameter. Researchers may describe mu\_mu\_ as a fraction. If you recall the parameter oscillation problem before, this actually becomes a damper to stop the oscillations. Momentum based gradient descent often has a smoother path and settles to a minimal closer and faster.

Adagrad

If the input features are not scaled correctly, it is impossible to find the right learning rate that works for all the features. This indicates the learning rate needed to self-adapt for each tunable parameter. One way to do it is to remember how much change has made to a specific WiWi. We will reduce the parameter of change if that parameter has been changed frequently. This will absolutely help the oscillation problem because it acts like a damper again. In Adagrad, it was done slightly different by allowing the rate of change to drop inversely by the L2 norm of all the previous gradients dwidwi.

RMSprop

RMSprop uses a similar concept with the following formula and the hyperparameter decay\_rate to control how much previous histories you keep.

Adam

Adam combines the concepts of momentum with RMSprop:

*Adam is the most often used method now.*

Here is an example of using Adam Optimizer in TensorFlow

Visualization

Here are some animations produced by Alec Radford in demonstrating how the gradient descent behaves for different algorithms. Regular gradient descent (red) learns the slowest and different algorithms have different descending patterns and speed. One point that is more interesting is how some algorithms overshoot the minimal or oscillate around it.

Feature Scaling (normalization)

As we find out, we want the feature input to the network to be scaled correctly (normalized). If the features do not have the proper scale, it will be much more difficult for the gradient descent to work. The training parameters may oscillate.

For example, with 2 input features, we want the shape to be as close to a circle as possible.

We normalize the features in the dataset to have zero mean and unit variance.

z=x−μσz=x−μσ

For images, we normalize every pixel independently. We compute a mean and a variance at each pixel location for the entire training dataset. Therefore, for an image with NxN pixels, we use NxN means and variances to normalize the image.

zij=xij−μijσijzij=xij−μijσij

In practice, we do not read all the training data at once to compute the mean or variance. We compute a running mean during the training. Here is the formula for the running mean:

μn=μn−1+k⋅(xi−μn−1)μn=μn−1+k⋅(xi−μn−1)

which kk is a small constant.

Whitening

In machine learning, we prefer features to be unrelated. For example, in a dating application, a person may prefer a tall person but not too thin. However, weight and height are co-related. A taller person is heavier than a shorter person on average. Re-scaling these features independently can only tell whether a person is lighter than average in the population, but not whether the person is thin. Weigth increases with height:

A network learns faster if features are un-related. We express the co-relations between a feature xixi and xjxj in terms of a covariance matrix below:

∑=⎡⎣⎢⎢⎢⎢⎢E[(x1−μ1)(x1−μ1)]E[(x2−μ2)(x1−μ1)]⋮E[(xn−μn)(x1−μ1)]E[(x1−μ1)(x2−μ2)]E[(x2−μ2)(x2−μ2)]⋮E[(xn−μn)(x2−μ2)]……⋱…E[(x1−μ1)(xn−μn)]E[(x2−μ2)(xn−μn)]⋮E[(xn−μn)(xn−μn)]⎤⎦⎥⎥⎥⎥⎥∑=[E[(x1−μ1)(x1−μ1)]E[(x1−μ1)(x2−μ2)]…E[(x1−μ1)(xn−μn)]E[(x2−μ2)(x1−μ1)]E[(x2−μ2)(x2−μ2)]…E[(x2−μ2)(xn−μn)]⋮⋮⋱⋮E[(xn−μn)(x1−μ1)]E[(xn−μn)(x2−μ2)]…E[(xn−μn)(xn−μn)]]

Which EE is the expected value.

Consider 2 data samples: (10, 20) and (32, 52). The mean of x1x1 will be μ1=10+322=21μ1=10+322=21 and μ2=36μ2=36

The expected value of the first element in the second row will be:

E[(x2−μ2)(x1−μ1)]=(20−36)(10−21)+(52−36)(32−21)2E[(x2−μ2)(x1−μ1)]=(20−36)(10−21)+(52−36)(32−21)2

From the covariance matrix ∑∑, we find a matrix WW by ∑∑ to convert the input XX to Y=WXY=WX. The purpose of whitening is to change the feature distribution from the left to the right one.

This sounds complicated, but can be done by Numpy linear algebra library

*Image data usually require 0 centered but does not require whitening.*

Batch normalization

We have emphasized so many times the benefits of having features with mean = 0 and σ=1σ=1

But, why do we stop at the input layer only. Batch normalization renormalizes a layer output. For example, we renormalized the output of the linear layer before feeding it into the ReLU.

We apply the normalization using the formula before:

z=x−μσz=x−μσ

We use the mean and variance computed from the current mini-batch samples. We then feed the output to a linear equation with the trainable scalar values γγ and ββ (1 pair for each normalized layer).

out=γz+βout=γz+β

If gamma=σgamma=σ and β=μβ=μ, we can see the normalization can be undone.

In the training, we use the mean and variance of the current training sample. But, for testing, we do not use the mean/variance of the testing data. Instead, we record a running mean & variance during training and apply it.

We normalize the input during testing with the running mean/variance in the training.

Hyperparameter tuning

Because the model is such a black box to us in real life problems, the hyperparameter tuning is usually a try and error. Some parameters are dependent on each other and cannot tune separately. Sometimes the relationship is subtle. For example, the regularization rate changes the shape of the cost function. Therefore it impacts how we tune the learning rate. We can create a mesh of values to be used for tuning. For example, with learning rates of (1e-1, 1e-2, … 1e-8) and regularization of (1e-3, 1e-4, .. 1e-6), we have a potential of 8x4 combinations to test ( (1e-1, 1e-3), (1e-1, 1e-3), …, (1e-8, 1e-5), (1e-8, 1e-6) ). We may not want to use an exactly rectangular shape of a mesh. For example, we may want to slight deviations at each mesh point with the hope that some irregularity may help us to explore more information.

*Start tuning parameters from coarse grain with fewer iterations before fine tuning.*

Troubleshooting

Many places can go wrong when training a deep network. Here are some simple tips:

* Unit test the forward pass and back propagation code.
  + At the beginning, test with non-random data.
* Compare the backpropagation result with the naive gradient check.
* Always start with a simple network that works.
  + Increasing accuracy should not be the first priority.
  + Handling multiple challenges in a complex network is not the way to go. Issues grow exponentially in DL.
* Create simple scenarios to verify the network:
  + Train with a small dataset with few iterations.
  + Compare the loss/accuracy value with the corresponding value of a random guess.
  + Verify if loss drops and/or accuracies increase during training.
  + Drop regularization - training accuracies should go up.
  + Overfit with a small dataset to see if the loss is small.
* Monitor or plot out the loss closely to see its trend.
* Keep track of the norm of W and gradient (and ratios) preferable in key layers. Looks for gradient vanishing/exploding problems.
* Do not waste time on a large dataset with long iterations during early development.
* Verify how trainable parameters are initialized.
* Always keep track of the shape of the data and document it in the code.
* Display and verify some training samples and the predictions.
* Plot out accuracy between validation and training to identify overfit issues.
* Plot activation/gradient histograms for all layers. If initialization is not done correctly, many dead or saturated nodes will be seen.
* For visualization problem, try to display the filter in an early layer and the activations.

Convolution Net (CNN) & Long short term memory (LSTM)

FC network is rarely used alone. Exploring all possible connections among nodes in the previous layer provides a complex model that is wasteful with small returns. A lot of information is localized. For an image, we want to extract features from neighboring pixels. CNN applies filters to explore localized features, and then apply FC to make predictions. LSTM applies time feedback loop to extract time sequence information. CNN & LSTM make changes to the design of a computation node and how it is connected. The core part of DL remains the same and learning CNN after FC is easier since the foundation is the same. Nevertheless, you will go nowhere in learning DL without CNN and/or LSTM. Hence, we have provided separate tutorials on both CNN and LSTM. Fortunately, with most DL techniques already discussed, the tutorials on both CNN and LSTM are much shorter.

Data augmentation

We have focused on the mechanics of the DL. One significant improvement for network training is to have more data. This avoids overfitting and has better coverage of your feature spaces. However, getting labeled samples can be expensive. One alternative is data auufmetation. For example, for visual recognition, we can flip the image, slightly rotate or skew the images with software libraries. This helps us to avoid overfitting and produces generalized predictions invariant of the spatial location of the objects. Some research may even expand further by allowing some data without labels to be used as training data if they produce a very high score with the model.

*Very simple effort to augment your data can have a significant impact on the training.*

Model ensembles

So far, we try to find the best models. In machine learning, we can take a vote from different decision trees to make the final prediction. This is based on the assumption that mistakes can be localized. There is a smaller chance for 2 different models to make the same mistake. In DL, each training starts with random guesses and therefore the models usually are not unique. We can pick the best models after training the networks multiple times. We can vote from different models to make the final predictions. This requires us to run the program multiple times, and can be prohibitively expensive. The alternative, we can run the training once and pick the best models during the latter phase of the training. We can have one vote per model, taking an average or use weights based on the confidence level of each prediction.

Credits

For the CIFRA 10 example, we start with assignment 2 in the Stanford class “CS231n Convolutional Neural Networks for Visual Recognition”. We start with some skeleton codes provided by the assignment and put it into our code to complete the assignment.

* Jonathan Hui blog
* [jhui](https://github.com/jhui)

“Software architect”