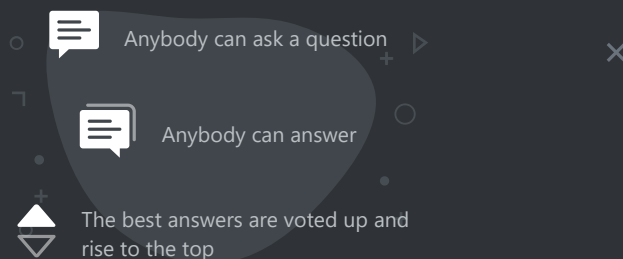


Cross Validated is a question and answer site for people interested in statistics, machine learning, data analysis, data mining, and data visualization. It only takes a minute to sign up.

Sign up to join this community



What should I do when my neural network doesn't learn?

Asked 4 years, 1 month ago Modified 2 months ago Viewed 270k times

- ▲ I'm training a neural network but the training loss doesn't decrease. How can I fix this?
- 315 ▼ I'm not asking about overfitting or regularization. I'm asking about how to solve the problem where my network's performance doesn't improve on the **training set**.
- ★ 401 This question is intentionally general so that other questions about how to train a neural network can be closed as a duplicate of this one, with the attitude that "if you give a man a fish you feed him for a day, but if you teach a man to fish, you can feed him for the rest of his life." See this Meta thread for a discussion: [What's the best way to answer "my neural network doesn't work, please fix" questions?](#)

If your neural network does not generalize well, see: [What should I do when my neural network doesn't generalize well?](#)

neural-networks faq

Share Cite Improve this question Follow

edited Apr 17, 2021 at 15:34

 kjetil b halvorsen ♦
66.3k 29 145 484

asked Jun 19, 2018 at 0:26

 Sycorax ♦
80.2k 21 193 321

6 Ivanov's blog "[Reasons why your Neural Network is not working](#)", especially sections II, III, and IV, could be helpful. – user5228 Jun 27, 2018 at 12:32

8 Answers

Sorted by: Highest score (default) ▾

▲ Verify that your code is bug free

392 ▼ There's a saying among writers that "All writing is re-writing" -- that is, the greater part of writing is revising. For programmers (or at least data scientists) the expression could be re-phrased as "All coding is debugging."

✓ Any time you're writing code, you need to verify that it works as intended. The best method I've ever found for verifying correctness is to break your code into small segments, and verify that each segment works. This can be done by comparing the segment output to what you know to be the correct answer. This is called [unit testing](#). Writing good unit tests is a key piece of becoming a good statistician/data scientist/machine learning expert/neural network practitioner. There is simply no substitute.

You have to check that your code is free of bugs before you can tune network performance! Otherwise, you might as well be re-arranging deck chairs on the *RMS Titanic*.

There are two features of neural networks that make verification even more important than for other types of machine learning or statistical models.

1. Neural networks are not "off-the-shelf" algorithms in the way that random forest or logistic regression are. Even for simple, feed-forward networks, the onus is largely on the user to make numerous decisions about how the network is configured, connected, initialized and optimized. This means writing code, and writing code means debugging.
2. *Even when a neural network code executes without raising an exception, the network can still have bugs!* These bugs might even be the insidious kind for which the network will train, but get stuck at a sub-optimal solution, or the resulting network does not have the desired architecture. ([This is an example of the difference between a syntactic and semantic error.](#))

This *Medium* post, "[How to unit test machine learning code](#)," by Chase Roberts discusses unit-testing for machine learning models in more detail. I borrowed this example of buggy code from the article:

```
def make_convnet(input_image):
    net = slim.conv2d(input_image, 32, [11, 11], scope="conv1_11x11")
    net = slim.conv2d(input_image, 64, [5, 5], scope="conv2_5x5")
    net = slim.max_pool2d(net, [4, 4], stride=4, scope='pool1')
    net = slim.conv2d(input_image, 64, [5, 5], scope="conv3_5x5")
    net = slim.conv2d(input_image, 128, [3, 3], scope="conv4_3x3")
    net = slim.max_pool2d(net, [2, 2], scope='pool2')
    net = slim.conv2d(input_image, 128, [3, 3], scope="conv5_3x3")
    net = slim.max_pool2d(net, [2, 2], scope='pool3')
    net = slim.conv2d(input_image, 32, [1, 1], scope="conv6_1x1")
    return net
```

Do you see the error? Many of the different operations are not *actually used* because previous results are over-written with new variables. Using this block of code in a network will still train and the weights will update and the loss might even decrease -- but the code definitely isn't doing what was intended. (The author is also inconsistent about using single- or double-quotes but that's purely stylistic.)

The most common *programming* errors pertaining to neural networks are

- Variables are created but never used (usually because of copy-paste errors);
- Expressions for gradient updates are incorrect;
- Weight updates are not applied;
- [Loss functions are not measured on the correct scale](#) (for example, cross-entropy loss can be expressed in terms of probability or logits)
- The loss is not appropriate for the task (for example, using categorical cross-entropy loss for a regression task).
- [Dropout is used during testing, instead of only being used for training.](#)
- [Make sure you're minimizing the loss function \$L\(x\)\$, instead of minimizing \$-L\(x\)\$.](#)
- [Make sure your loss is computed correctly.](#)

Unit testing is not just limited to the neural network itself. You need to test all of the steps that produce or transform data and feed into the network. Some common mistakes here are

- `NA` or `NaN` or `Inf` values in your data creating `NA` or `NaN` or `Inf` values in the output, and therefore in the loss function.
- Shuffling the labels independently from the samples (for instance, creating train/test splits for the labels and samples separately);
- Accidentally assigning the training data as the testing data;
- When using a train/test split, the model references the original, non-split data instead of the training partition or the testing partition.
- Forgetting to scale the testing data;
- Scaling the testing data using the statistics of the test partition instead of the train partition;
- Forgetting to un-scale the predictions (e.g. pixel values are in [0,1] instead of [0, 255]).
- Here's an example of a question where the problem appears to be one of model configuration or hyperparameter choice, but actually the problem was a subtle bug in how gradients were computed. [Is this drop in training accuracy](#)

For the love of all that is good, *scale your data*

The scale of the data can make an enormous difference on training. Sometimes, networks simply won't reduce the loss if the data isn't scaled. Other networks will decrease the loss, but only very slowly. Scaling the inputs (and certain times, the targets) can dramatically improve the network's training.

- Prior to presenting data to a neural network, **standardizing** the data to have 0 mean and unit variance, or to lie in a small interval like $[-0.5, 0.5]$ can improve training. This amounts to pre-conditioning, and removes the effect that a choice in units has on network weights. For example, length in millimeters and length in kilometers both represent the same concept, but are on different scales. The exact details of how to standardize the data depend on what your data look like.
- [Data normalization and standardization in neural networks](#)
 - [Why does \$\[0, 1\]\$ scaling dramatically increase training time for feed forward ANN \(1 hidden layer\)?](#)
- **Batch or Layer normalization** can improve network training. Both seek to improve the network by keeping a running mean and standard deviation for neurons' activations as the network trains. It is not well-understood why this helps training, and remains an active area of research.
 - ["Understanding Batch Normalization"](#) by Johan Bjorck, Carla Gomes, Bart Selman
 - ["Towards a Theoretical Understanding of Batch Normalization"](#) by Jonas Kohler, Hadi Daneshmand, Aurelien Lucchi, Ming Zhou, Klaus Neymeyr, Thomas Hofmann
 - ["How Does Batch Normalization Help Optimization? \(No, It Is Not About Internal Covariate Shift\)"](#) by Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry

Crawl Before You Walk; Walk Before You Run

Wide and deep neural networks, and neural networks with exotic wiring, are the Hot Thing right now in machine learning. But these networks didn't spring fully-formed into existence; their designers built up to them from smaller units. First, build a small network with a single hidden layer and verify that it works correctly. Then incrementally add additional model complexity, and verify that each of those works as well.

- Too *few* **neurons** in a layer can restrict the representation that the network learns, causing under-fitting. Too many neurons can cause over-fitting because the network will "memorize" the training data.

Even if you can prove that there is, mathematically, only a small number of neurons necessary to model a problem, it is often the case that having "a few more" neurons makes it *easier* for the optimizer to find a "good" configuration. (But I don't think anyone fully understands why this is the case.) I provide an example of this in the context of the XOR problem here: [Aren't my iterations needed to train NN for XOR with MSE < 0.001 too high?](#)
- Choosing the number of **hidden layers** lets the network learn an abstraction from the raw data. Deep learning is all the rage these days, and networks with a large number of layers have shown impressive results. But adding too many hidden layers can make risk overfitting or make it very hard to optimize the network.
- Choosing a clever **network wiring** can do a lot of the work for you. Is your data source amenable to specialized network architectures? Convolutional neural networks can achieve impressive results on "structured" data sources, image or audio data. Recurrent neural networks can do well on sequential data types, such as natural language or time series data. Residual connections can improve deep feed-forward networks.

Neural Network Training Is Like Lock Picking

To achieve state of the art, or even merely good, results, you have to have to have set up all of the parts configured to work well *together*. Setting up a neural network configuration that actually learns is a lot like picking a lock: all of the pieces have to be lined up *just right*. Just as it is not sufficient to have a single tumbler in the right place, neither is it sufficient to have only the architecture. or only the optimizer. set up correctly.

Tuning configuration choices is not really as simple as saying that one kind of configuration choice (e.g. learning rate) is more or less important than another (e.g. number of units), since all of these choices interact with all of the other choices, so one choice can do well *in combination with another choice made elsewhere*.

This is a non-exhaustive list of the configuration options which are not also regularization options or numerical optimization options.

All of these topics are active areas of research.

- The network **initialization** is often overlooked as a source of neural network bugs. Initialization over too-large an interval can set initial weights too large, meaning that single neurons have an outsize influence over the network behavior.
- The key difference between a neural network and a regression model is that a neural network is a composition of many nonlinear functions, called **activation functions**. (See: [What is the essential difference between neural network and linear regression](#))

Classical neural network results focused on sigmoidal activation functions (logistic or \tanh functions). A recent result has found that ReLU (or similar) units tend to work better because they have steeper gradients, so updates can be applied quickly. (See: [Why do we use ReLU in neural networks and how do we use it?](#)) One caution about ReLUs is the "dead neuron" phenomenon, which can stymie learning; leaky relus and similar variants avoid this problem. See

- [Why can't a single ReLU learn a ReLU?](#)
- [My ReLU network fails to launch](#)

There are a number of other options. See: [Comprehensive list of activation functions in neural networks with pros/cons](#)

- Residual connections are a neat development that can make it easier to train neural networks. "[Deep Residual Learning for Image Recognition](#)" Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun In: CVPR. (2016). Additionally, changing the order of operations within the residual block can further improve the resulting network. "[Identity Mappings in Deep Residual Networks](#)" by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun.

Non-convex optimization is hard

The objective function of a neural network is only convex when there are no hidden units, all activations are linear, and the design matrix is full-rank -- because this configuration is identically an ordinary regression problem.

In all other cases, the optimization problem is non-convex, and non-convex optimization is hard. The challenges of training neural networks are well-known (see: [Why is it hard to train deep neural networks?](#)). Additionally, neural networks have a very large number of parameters, which restricts us to solely first-order methods (see: [Why is Newton's method not widely used in machine learning?](#)). **This is a very active area of research.**

- Setting the **learning rate** too large will cause the optimization to diverge, because you will leap from one side of the "canyon" to the other. Setting this too small will prevent you from making any real progress, and possibly allow the noise inherent in SGD to overwhelm your gradient estimates. See:
 - [How can change in cost function be positive?](#)
- **Gradient clipping** re-scales the norm of the gradient if it's above some threshold. I used to think that this was a set-and-forget parameter, typically at 1.0, but I found that I could make an LSTM language model dramatically better by setting it to 0.25. I don't know why that is.
- **Learning rate scheduling** can decrease the learning rate over the course of training. In my experience, trying to use scheduling is a lot like [regex](#): it replaces one problem ("How do I get learning to continue after a certain epoch?") with two problems ("How do I get learning to continue after a certain epoch?" and "How do I choose a good schedule?"). Other people insist that scheduling is essential. I'll let you decide.
- Choosing a good **minibatch size** can influence the learning process indirectly, since a larger mini-batch will tend to have a smaller variance ([law-of-large-numbers](#)) than a smaller mini-batch. You want the mini-batch to be large enough to be informative about the direction of the gradient, but small enough that SGD can regularize your network.
- There are a number of variants on **stochastic gradient descent** which use momentum, adaptive learning rates, Nesterov updates and so on to improve upon vanilla SGD. Designing a better optimizer is very much an active area

researcher updates and so on to improve open-source code. Designing a better optimizer is very much an active area of research. Some examples:

- [No change in accuracy using Adam Optimizer when SGD works fine](#)
- [How does the Adam method of stochastic gradient descent work?](#)
- [Why does momentum escape from a saddle point in this famous image?](#)
- When it first came out, the Adam optimizer generated a lot of interest. But some recent research has found that SGD with momentum can out-perform adaptive gradient methods for neural networks. "[The Marginal Value of Adaptive Gradient Methods in Machine Learning](#)" by Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht
- But on the other hand, this very recent paper proposes a new adaptive learning-rate optimizer which supposedly closes the gap between adaptive-rate methods and SGD with momentum. "[Closing the Generalization Gap of Adaptive Gradient Methods in Training Deep Neural Networks](#)" by Jinghui Chen, Quanquan Gu

Adaptive gradient methods, which adopt historical gradient information to automatically adjust the learning rate, have been observed to generalize worse than stochastic gradient descent (SGD) with momentum in training deep neural networks. This leaves how to close the generalization gap of adaptive gradient methods an open problem. In this work, we show that adaptive gradient methods such as Adam, Amsgrad, are sometimes "over adapted". We design a new algorithm, called Partially adaptive momentum estimation method (Padam), which unifies the Adam/Amsgrad with SGD to achieve the best from both worlds. Experiments on standard benchmarks show that Padam can maintain fast convergence rate as Adam/Amsgrad while generalizing as well as SGD in training deep neural networks. These results would suggest practitioners pick up adaptive gradient methods once again for faster training of deep neural networks.

- Specifically for `triplet-loss` models, there are a number of tricks which can improve training time and generalization. See: [In training a triplet network, I first have a solid drop in loss, but eventually the loss slowly but consistently increases. What could cause this?](#)

Regularization

Choosing and tuning network regularization is a key part of building a model that generalizes well (that is, a model that is not overfit to the training data). However, at the time that your network is struggling to decrease the loss on the training data -- when the network is not learning -- regularization can obscure what the problem is.

When my network doesn't learn, I turn off all regularization and verify that the non-regularized network works correctly. Then I add each regularization piece back, and verify that each of those works along the way.

This tactic can pinpoint where some regularization might be poorly set. Some examples are

- L^2 regularization (aka weight decay) or L^1 regularization is set too large, so the weights can't move.
- Two parts of regularization are in conflict. For example, it's widely observed that layer normalization and dropout are difficult to use together. Since either on its own is very useful, understanding how to use both is an active area of research.
 - "[Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift](#)" by Xiang Li, Shuo Chen, Xiaolin Hu, Jian Yang
 - "[Adjusting for Dropout Variance in Batch Normalization and Weight Initialization](#)" by Dan Hendrycks, Kevin Gimpel.
 - "[Self-Normalizing Neural Networks](#)" by Günter Klambauer, Thomas Unterthiner, Andreas Mayr and Sepp Hochreiter

Keep a Logbook of Experiments

When I set up a neural network, I don't hard-code any parameter settings. Instead, I do that in a configuration file (e.g., JSON) that is read and used to populate network configuration details at runtime. I keep all of these configuration files. If

I make any parameter modification, I make a new configuration file. Finally, I append as comments all of the per-epoch losses for training and validation.

The reason that I'm so obsessive about retaining old results is that this makes it very easy to go back and review previous experiments. It also hedges against mistakenly repeating the same dead-end experiment. Psychologically, it also lets you look back and observe "Well, the project might not be where I want it to be today, but I am making progress compared to where I was k weeks ago."

As an example, I wanted to learn about LSTM language models, so I decided to make a Twitter bot that writes new tweets in response to other Twitter users. I worked on this in my free time, between grad school and my job. It took about a year, and I iterated over about 150 different models before getting to a model that did what I wanted: generate new English-language text that (sort of) makes sense. (One key sticking point, and part of the reason that it took so many attempts, is that it was not sufficient to simply get a low out-of-sample loss, since early low-loss models had managed to memorize the training data, so it was just reproducing germane blocks of text verbatim in reply to prompts -- it took some tweaking to make the model more spontaneous and still have low loss.)

Share Cite Improve this answer Follow

edited Feb 28 at 13:52

answered Jun 19, 2018 at 0:26



Sycorax ♦

80.2k 21 193 321

- 16 Lots of good advice there. It's interesting how many of your comments are similar to comments I have made (or have seen others make) in relation to debugging estimation of parameters or predictions for complex models with MCMC sampling schemes. (For example, the code may seem to work when it's not correctly implemented.) – Glen_b Jun 19, 2018 at 2:06 ✎
- 18 @Glen_b I don't think coding best practices receive enough emphasis in most stats/machine learning curricula which is why I emphasized that point so heavily. I've seen a number of NN posts where OP left a comment like "oh I found a bug now it works." – Sycorax ♦ Jun 19, 2018 at 2:27 ✎
- 8 I teach a programming for data science course in python, and we actually do functions and unit testing on the first day, as primary concepts. Fighting the good fight. – Matthew Drury Jun 19, 2018 at 4:58
- 12 +1 for "All coding is debugging". I am amazed how many posters on SO seem to think that coding is a simple exercise requiring little effort; who expect their code to work correctly the first time they run it; and who seem to be unable to proceed when it doesn't. The funny thing is that they're half right: coding *is* easy - but programming is hard. – Bob Jarvis - Слава Україні Jun 20, 2018 at 11:07 ✎
- 2 It is really nice answer. I knew a good part of this stuff, what stood out for me is **Keep a Logbook of Experiments**- it is a really good suggestion. Very intuitive but not very obvious ways to keep track of experiments. Mine was very messy. Thank you @sycorax-says-reinstate-monica. Btw, if I may ask, what do you suggest is the best to keep the log of experiments- JSON or YAML or any other way. I mean which one do you use or find it most convenient and readable. Please do tell. Thanks :) – zeal Mar 6, 2020 at 3:25 ✎

▲ The posted answers are great, and I wanted to add a few "Sanity Checks" which have greatly helped me in the past.

82

1) **Train your model on a single data point. If this works, train it on two inputs with different outputs.**



This verifies a few things. First, it quickly shows you that your model is able to learn by checking if your model can overfit your data. In my case, I constantly make silly mistakes of doing `Dense(1,activation='softmax')` vs `Dense(1,activation='sigmoid')` for binary predictions, and the first one gives garbage results.



If your model is unable to overfit a few data points, then either it's too small (which is unlikely in today's age), or something is wrong in its structure or the learning algorithm.

2) **Pay attention to your initial loss.**

Continuing the binary example, if your data is 30% 0's and 70% 1's, then your initial expected loss around $L = -0.3 \ln(0.5) - 0.7 \ln(0.5) \approx 0.7$. This is because your model should start out close to randomly guessing.

A lot of times you'll see an initial loss of something ridiculous, like 6.5. Conceptually this means that your output is heavily saturated, for example toward 0. For example $-0.3 \ln(0.99) - 0.7 \ln(0.01) = 3.2$, so if you're seeing a loss that's bigger than 1, it's likely your model is very skewed. This usually happens when your neural network weights aren't properly balanced, especially closer to the softmax/sigmoid. So this would tell you if your initialization is bad.

You can study this further by making your model predict on a few thousand examples, and then histogramming the outputs. This is especially useful for checking that your data is correctly normalized. As an example, if you expect your

output to be heavily skewed toward 0, it might be a good idea to transform your expected outputs (your training data) by taking the square roots of the expected output. This will avoid gradient issues for saturated sigmoids, at the output.

3) Generalize your model outputs to debug

As an example, imagine you're using an LSTM to make predictions from time-series data. Maybe in your example, you only care about the latest prediction, so your LSTM outputs a single value and not a sequence. Switch the LSTM to return predictions at each step (in keras, this is `return_sequences=True`). Then you can take a look at your hidden-state outputs after every step and make sure they are actually different. An application of this is to make sure that when you're masking your sequences (i.e. padding them with data to make them equal length), the LSTM is correctly ignoring your masked data. Without generalizing your model *you will never find this issue*.

4) Look at individual layers

Tensorboard provides a useful way of [visualizing your layer outputs](#). This can help make sure that inputs/outputs are properly normalized in each layer. It can also catch buggy activations. You can also query layer outputs in keras on a batch of predictions, and then look for layers which have suspiciously skewed activations (either all 0, or all nonzero).

5) Build a simpler model first

You've decided that the best approach to solve your problem is to use a CNN combined with a bounding box detector, that further processes image crops and then uses an LSTM to combine everything. It takes 10 minutes just for your GPU to initialize your model.

Instead, make a batch of fake data (same shape), and break your model down into components. Then make dummy models in place of each component (your "CNN" could just be a single 2x2 20-stride convolution, the LSTM with just 2 hidden units). This will help you make sure that your model structure is correct and that there are no extraneous issues. I struggled for a while with such a model, and when I tried a simpler version, I found out that one of the layers wasn't being masked properly due to a keras bug. You can easily (and *quickly*) query internal model layers and see if you've setup your graph correctly.

6) Standardize your Preprocessing and Package Versions

Neural networks in particular are extremely sensitive to small changes in your data. As an example, two popular image loading packages are `cv2` and `PIL`. Just by virtue of *opening* a JPEG, both these packages will produce **slightly different** images. The differences are usually really small, but you'll occasionally see drops in model performance due to this kind of stuff. Also it makes debugging a nightmare: you got a validation score during training, and then later on you use a different loader and get different accuracy on the same darn dataset.

So if you're downloading someone's model from github, pay close attention to their preprocessing. What image loaders do they use? What image preprocessing routines do they use? When resizing an image, what interpolation do they use? Do they first resize and then normalize the image? Or the other way around? What's the channel order for RGB images?

The safest way of standardizing packages is to use a `requirements.txt` file that outlines all your packages just like on your training system setup, down to the `keras==2.1.5` version numbers. In theory then, using Docker along with the same GPU as on your training system should then produce the same results.

Share Cite Improve this answer Follow

edited Oct 14, 2018 at 16:39



Matti Wens

412 2 11

answered Jun 19, 2018 at 18:45



Alex R.

13.3k 2 26 50

10 (+1) Checking the initial loss is a great suggestion. I regret that I left it out of my answer. – [Sycorax](#) ♦ Jun 19, 2018 at 18:47

11 Making sure that your model can overfit is an excellent idea. I am so used to thinking about overfitting as a weakness that I never explicitly thought (until you mentioned it) that the *ability* to overfit is actually a strength. – [John Coleman](#) Jun 20, 2018 at 13:41

1 Testing on a single data point is a really great idea. If it can't learn a single point, then your network structure probably can't represent the input -> output function and needs to be redesigned. – [Azmisov](#) Aug 5, 2020 at 18:10

@Alex R. I'm still unsure what to do if you do pass the overfitting test. In my case it's not a problem with the architecture (I'm implementing a Resnet from another paper). Although it can easily overfit to a single image, it can't fit to a large dataset, despite good normalization and shuffling. Likely a problem with the data? – [A Tyshka](#) May 13, 2021 at 21:50

Point 1 is also mentioned in Andrew Ng's Coursera Course: [coursera.org/learn/...](https://www.coursera.org/learn/) I previously had the issue you mentioned in point 6. It is called "Training-Serving Skew": [developers.google.com/machine-learning/guides/...](https://developers.google.com/machine-learning/guides/) – [Imran Kocabiyik](#) Jun 1, 2021 at 8:47

Do not train a neural network to start with!

31 All the answers are great, but there is one point which ought to be mentioned : is there anything to learn from your data ? (which could be considered as some kind of testing).


If the label you are trying to predict is independent from your features, then it is likely that the training loss will have a hard time reducing.

Instead, start calibrating a linear regression, a random forest (or any method you like whose number of hyperparameters is low, and whose behavior you can understand).

Then, if you achieve a decent performance on these models (better than random guessing), you can start tuning a neural network (and @Sycorax 's answer will solve most issues).

Share Cite Improve this answer Follow

answered Jun 20, 2018 at 11:25

 **RUser4512**
9,346 5 30 59

8 I agree with this answer. Neural networks and other forms of ML are "so hot right now". Often the simpler forms of regression get overlooked. Also, when it comes to explaining your model, someone will come along and ask "what's the effect of x_k on the result?" and all you will be able to do is shrug your shoulders. **Only look to Machine Learning solutions when the simpler techniques have failed you.** – Ingolifs Jun 22, 2018 at 1:17

I like to start with exploratory data analysis to get a sense of "what the data wants to tell me" before getting into the models. The asker was looking for "neural network doesn't learn" so I majored there. – EngrStudent Aug 4 at 3:00

At its core, the basic workflow for training a NN/DNN model is more or less always the same:

- 26
1. define the NN architecture (how many layers, which kind of layers, the connections among layers, the activation functions, etc.)
 2. read data from some source (the Internet, a database, a set of local files, etc.), have a look at a few samples (to make sure the import has gone well) and perform data cleaning if/when needed. This step is not as trivial as people usually assume it to be. The reason is that for DNNs, we usually deal with gigantic data sets, several orders of magnitude larger than what we're used to, when we fit more standard *nonlinear parametric statistical models* (NNs belong to this family, in theory).
 3. normalize or standardize the data in some way. Since NNs are nonlinear models, normalizing the data can affect not only the numerical stability, but also the training time, and the NN outputs (a linear function such as normalization doesn't commute with a nonlinear hierarchical function).
 4. split data in training/validation/test set, or in multiple folds if using cross-validation.
 5. train the neural network, while at the same time controlling the loss on the validation set. Here you can enjoy the soul-wrenching pleasures of non-convex optimization, where you don't know if any solution exists, if multiple solutions exist, which is the best solution(s) in terms of generalization error and how close you got to it. The comparison between the training loss and validation loss curve guides you, of course, but don't underestimate the *die hard* attitude of NNs (and especially DNNs): they often show a (maybe slowly) decreasing training/validation loss even when you have **crippling bugs** in your code.
 6. Check the accuracy on the test set, and make some diagnostic plots/tables.
 7. Go back to point 1 because the results aren't good. Reiterate *ad nauseam*.

Of course details will change based on the specific use case, but with this rough canvas in mind, we can think of what is more likely to go wrong.

Basic Architecture checks

This can be a source of issues. Usually I make these preliminary checks:

• Look for a simple architecture which works well on your problem (for example, Multi-Net/2 in the case of image

- look for a simple architecture which works well on your problem (for example, mobilenetv2 in the case of image classification) and apply a suitable initialization (at this level, random will usually do). If this trains correctly on your data, at least you know that there are no glaring issues in the data set. If you can't find a simple, tested architecture which works in your case, **think of a simple baseline**. For example a Naïve Bayes classifier for classification (or even just classifying always the most common class), or an ARIMA model for time series forecasting
- Build unit tests. Neglecting to do this (and the use of the bloody Jupyter Notebook) are usually the root causes of issues in NN code I'm asked to review, especially when the model is supposed to be deployed in production. As the most upvoted answer has already covered unit tests, I'll just add that [there exists a library which supports unit tests development for NN](#) (only in Tensorflow, unfortunately).

Training Set

Double check your input data. See if you inverted the training set and test set labels, for example (happened to me once - ___-), or if you imported the wrong file. Have a look at a few input samples, and the associated labels, and make sure they make sense. Check that the normalized data are really normalized (have a look at their range). Also, real-world datasets are dirty: for classification, there could be a high level of label noise (samples having the wrong class label) or for multivariate time series forecast, some of the time series components may have a lot of missing data (I've seen numbers as high as 94% for some of the inputs).

The order in which the training set is fed to the net during training may have an effect. Try a random shuffle of the training set (**without breaking the association between inputs and outputs**) and see if the training loss goes down.

Finally, the **best** way to check if you have training set issues is to use another training set. If you're doing image classification, instead than the images you collected, use a standard dataset such CIFAR10 or CIFAR100 (or ImageNet, **if** you can afford to train on that). These data sets are well-tested: if your training loss goes down here but not on your original data set, you may have issues in the data set.

Do the Golden Tests

There are two tests which I call Golden Tests, which are very useful to find issues in a NN which doesn't train:

- reduce the training set to 1 or 2 samples, and train on this. The NN should immediately overfit the training set, reaching an accuracy of 100% on the training set very quickly, while the accuracy on the validation/test set will go to 0%. If this doesn't happen, there's a bug in your code.
- the opposite test: you keep the full training set, but you shuffle the labels. The only way the NN can learn now is by memorising the training set, which means that the training loss will decrease very slowly, while the test loss will increase very quickly. **In particular, you should reach the random chance loss on the test set.** This means that if you have 1000 classes, you should reach an accuracy of 0.1%. If you don't see any difference between the training loss before and after shuffling labels, this means that your code is buggy (remember that we have already checked the labels of the training set in the step before).

Check that your training metric makes sense

Accuracy (0-1 loss) is a crappy metric if you have strong class imbalance. Try something more meaningful such as cross-entropy loss: you don't just want to classify correctly, but you'd like to classify with high accuracy.

Bring out the big guns

If nothing helped, it's now the time to start fiddling with hyperparameters. This is easily the worse part of NN training, but these are gigantic, non-identifiable models whose parameters are fit by solving a non-convex optimization, so these iterations often can't be avoided.

- try different optimizers: SGD trains slower, but it leads to a lower generalization error, while Adam trains faster, but the test loss stalls to a higher value
- try decreasing the batch size
- increase the learning rate initially, and then decay it, or use [a cyclic learning rate](#)

- add layers
- add hidden units
- remove regularization gradually (maybe switch batch norm for a few layers). The training loss should now decrease, but the test loss may increase.
- visualize the distribution of weights and biases for each layer. I never had to get here, but if you're using BatchNorm, you would expect approximately standard normal distributions. See if the norm of the weights is increasing abnormally with epochs.
- if you're getting some error at training time, **google that error**. I wasted one morning while trying to fix a perfectly working architecture, only to find out that the version of Keras I had installed had buggy multi-GPU support and I had to update it. Sometimes I had to do the opposite (downgrade a package version).
- update your CV and start looking for a different job :-)

Share Cite Improve this answer Follow

edited Jun 29, 2018 at 7:05

answered Jun 28, 2018 at 14:25



DeltaIV

16.1k 5 63 106

- +1, but "bloody Jupyter Notebook"? Care to comment on that? :) – [amoeba](#) Jun 28, 2018 at 14:38
- Here's [why I hate Jupyter Notebooks](#). TL;DR: hidden state, diffing is a pain, security issues and it encourages bad programming practices, such as not using unit/regression/integration tests. Training NNs is already hard enough, without people forgetting about the fundamentals of programming. – [DeltaIV](#) Jun 28, 2018 at 16:20
- I'm possibly being too negative, but frankly I've had enough with people cloning Jupyter Notebooks from GitHub, thinking it would be a matter of minutes to adapt the code to their use case and then coming to me complaining that nothing works. For crapes' sake, get a real IDE such as PyCharm or VisualStudio Code and create a well-structured code, rather than cooking up a Notebook! Especially if you plan on shipping the model to production, it'll make things a lot easier. – [DeltaIV](#) Jun 28, 2018 at 16:24
- Lol. 'Jupyter notebook' and 'unit testing' are anti-correlated. – [Sycorax](#) Jun 28, 2018 at 16:38
- (+1) This is a good write-up. The suggestions for randomization tests are really great ways to get at bugged networks. – [Sycorax](#) Aug 1, 2018 at 4:05



10



If the model isn't learning, there is a decent chance that your backpropagation is not working. But there are so many things can go wrong with a black box model like Neural Network, there are many things you need to check. I think Sycorax and Alex both provide very good comprehensive answers. Just want to add on one technique haven't been discussed yet.

In the [Machine Learning Course](#) by Andrew Ng, he suggests running [Gradient Checking](#) in the first few iterations to make sure the backpropagation is doing the right thing. Basically, the idea is to calculate the derivative by defining two points with a ϵ interval. Making sure the derivative is approximately matching your result from backpropagation should help in locating where is the problem.

Share Cite Improve this answer Follow

answered Jun 19, 2018 at 19:22



Anthony Lei

401 2 11



4



Check the data pre-processing and augmentation.

I just learned this lesson recently and I think it is interesting to share. Nowadays, many frameworks have built in data pre-processing pipeline and augmentation. And these elements may **completely destroy the data**.

For example, suppose we are building a classifier to classify 6 and 9, and we use random rotation augmentation ...

A toy example can be found here

[Why can't scikit-learn SVM solve two concentric circles?](#)

My recent lesson is trying to detect if an image contains some hidden information, by stenography tools. And struggled for a long time that the model does not learn.

The reason is many packages are rescaling images to certain size and this operation completely destroys the hidden information inside.

Share Cite Improve this answer Follow

edited May 12 at 4:31

answered Jul 21, 2020 at 7:55



Haitao Du

33.7k



18



123



223



3



In my case the initial training set was probably too difficult for the network, so it was not making any progress. I have prepared the easier set, selecting cases where differences between categories were seen by my own perception as more obvious.

The network picked this simplified case well. After it reached really good results, it was then able to progress further by training from the original, more complex data set without blundering around with training score close to zero. To make sure the existing knowledge is not lost, reduce the set learning rate.

Share Cite Improve this answer Follow

edited Feb 16, 2020 at 21:28

answered Feb 16, 2020 at 10:27



h22

141



4

1 This is a good addition. A similar phenomenon also arises in another context, with a different solution. When training triplet networks, training with online hard negative mining immediately risks model collapse, so people train with semi-hard negative mining first as a kind of "pre training." This is an easier task, so the model learns a good initialization before training on the real task. Then training proceed with online hard negative mining, and the model is better for it as a result. – Sycorax ♦ Feb 16, 2020 at 16:44

AFAIK, this triplet network strategy is first suggested in the FaceNet paper. "FaceNet: A Unified Embedding for Face Recognition and Clustering" Florian Schroff, Dmitry Kalenichenko, James Philbin – Sycorax ♦ Feb 16, 2020 at 17:01

+1 Learning like children, starting with simple examples, not being given everything at once! – kjetil b halvorsen ♦ Jun 2 at 1:56



1



I had a model that did not train at all. It just stuck at random chance of particular result with no loss improvement during training. Loss was constant 4.000 and accuracy 0.142 on 7 target values dataset.

It become true that I was doing regression with ReLU last activation layer, which is obviously wrong.

Before I was knowing that this is wrong, I did add Batch Normalisation layer after every learnable layer, and that helps. However, training become somehow erratic so accuracy during training could easily drop from 40% down to 9% on validation set. Accuracy on training dataset was always okay.

Then I realized that it is enough to put Batch Normalisation before that last ReLU activation layer only, to keep improving loss/accuracy during training. That probably did fix wrong activation method.

However, when I did replace ReLU with Linear activation (for regression), no Batch Normalisation was needed any more and model started to train significantly better.

Share Cite Improve this answer Follow

answered Mar 12, 2020 at 6:30



vedrano

111



3

Highly active question. Earn 10 reputation (not counting the **association bonus**) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.