Training Neural Networks: Practical advice

Andrej Karpathy (https://karpathy.ai/) has an excellent blog post (https://karpathy.github.io/2019/04/25/recipe/) that conveys much practical wisdom for successfully training Neural Networks.

It was written back in 2019 and has the feel of advice from someone who has hand-coded Neural Networks from scratch, rather than someone using higher-level toolkits (e.g., Keras).

But: you learn a lot building Neural Networks from scratch (in fact, he has a current free course (https://karpathy.ai/zero-to-hero.html) (using Pytorch) that does just that.

In this module, we will attempt to distill some valuable points from the blog .

Neural training fails silently (https://karpathy.github.io/2019/04/25/recipe/#2-neural-net-training-fails-silently)

When writing a program using an imperative programming language (e.g., Python)

- failure mode is apparent: run-time error, exception, etc.

When creating a Neural Network

- failures are often silent
- they compute *something*, but not necessarily the something that you desire.
    - manifesting in a large Loss

This makes it hard to debug.

But there are some practical steps you can take to minimize the problem.

[Become one with the data (https://karpathy.github.io/2019/04/25/recipe/#1-become-one-with-the-data)](https://karpathy.github.io/2019/04/25/recipe/#1-become-one-with-the-data)

This is essentially the same as our Exploratory Data Analysis step of our Recipe.

- but with more intensity than most of us devote

Some key quotes:

[Set up the end-to-end training/evaluation skeleton + get dumb baselines (https://karpathy.github.io/2019/04/25/recipe/#2-set-up-the-end-to-end-trainingevaluation-skeleton--get-dumb-baselines)](https://karpathy.github.io/2019/04/25/recipe/#2-set-up-the-end-to-end-trainingevaluation-skeleton--get-dumb-baselines)

Start with a simple model (like the Baseline models we suggest in the Recipe).

The time to make naive mistakes is now, before you add complexity.

Set up a process to make thing repeatable (i.e., the scientific process for experimentation)

Acknowledge and control randomness

Be aware of the sources of randomness in training and try to eliminate them.

It's hard to debug/understand when each run is different

- Shuffling of dataset
- Random initialization of weights
    - the *distribution* may be fixed, but not the samples from the distribution
- Randomness you introduce explicitly: drawing random samples

Both Tensorflow and Python random number generation is controlled by a (separate) seed value.

Fixing this value for each run makes your program execution repeatable.

More recent versions of TensorFlow provide a `set_random_seed` (https://www.tensorflow.org/api_docs/python/tf/keras/utils/set_random_seed) method

- sets random seeds at multiple sources
- equivalent to

```
import random
import numpy as np
import tensorflow as tf
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)
```

If your version of TensorFlow does not implement `set_random_seed` i have created an equivalent

```
def set_seed(seed, Debug=False):
    try:
        from tensorflow.keras.utils import set_random_seed
        set_random_seed(seed)

        if Debug:
            print("Used set_random_seed")
    except:
        import random
        import numpy as np
        import tensorflow as tf
        random.seed(seed)
        np.random.seed(seed)
        tf.random.set_seed(seed)

        if Debug:
            print("Used individual setting of seeds")
```

Verify decreasing training loss

When training starts, the initial loss will be high.

If a model is "learning", the *training* (in-sample) loss will

- decrease rapidly in early epochs
- continue to decrease
    - not necessarily in a straight line
- reach a minimum (potentially bumpy)

If your training loss is not decreasing in early epochs: something is wrong !

- Visualize the inputs and labels to the Neural Network directly
    - is the input correct ?
        - learn how to obtain mini-batches
    - do the labels match the features ?
- Is your network "too small" to learn ?
    - try increasing the size of the network (e.g., number of units per layer)

Note: training loss will often decrease *without* a corresponding decrease in validation (out of sample) loss.

Set the bias on the head layer

Note: I've never seen anyone do this, but it's a great interview question at the least !

The head layer L is usually a Classifier or Regressor, implemented as a `Dense` layer.

Dense layer l computes a dot product of weights $W_{(l)}$ and layer inputs $\mathbf{y}_{(l-1)}$

- weights for each unit j of layer l consists of a single "bias" $b_{(l),j}$ and vector of $\mathbf{w}_{(l),j}$ of non-bias weights
    - our convention is $b_{(l),j} = \mathbf{W}_{(l),j,0}$ and $\mathbf{w}_{(l),j} = \mathbf{W}_{(l),j,[1:]}$
- so unit j of layer L computes

$$\mathbf{y}_{(L),j} = \mathbf{y}_{(L-1)} \cdot \mathbf{w}_{(L),j} + b_{(L),j}$$

Suppose we initialize the non-bias weights $\mathbf{w}_{(L),j}$

- from a random distribution (e.g., Normal, Uniform)
- with mean 0

Then the Expected value of unit j is equal to the bias $b_{(L),j}$

$$
\begin{aligned}
\mathbb{E}\mathbf{y}_{(L),j} &= \mathbb{E}\left(\mathbf{y}_{(L-1)} \cdot \mathbf{w}_{(L),j} + b_{(L),j}\right) \\
&= b_{(L),j} & \text{since } \mathbb{E}\mathbf{w}_{(L),j,k} = 0
\end{aligned}
$$

This suggests that a good value for initializing the bias is

- $b_{(L)} = \bar{\mathbf{y}}$ for a Regression task
    - $\bar{\mathbf{y}}$ is average $\mathbf{y^{(i)}}$ over all training examples
    - we omit subscript j from $b_{(L)}$ since we assume a single regression output
    - error for example i would be
    $$\mathbf{y^{(i)}} - \bar{\mathbf{y}}$$
    which has expected value (over all i) of 0

- $b_{(L),j} = \log p_j$ for logit j of a Classification task
    - where p_j is the probability (over the training set) of examples with the j^{th} label
    - $\log p_j$ is the value of the logit corresponding to probability p_j
    - the initial predicted probability distribution *for each example* matches the training distribution (across all examples)

Setting the bias manually may speed up training

- initial epochs of training may be primarily to *learn* this bias

Verify the loss

We can manually calculate the training Loss after one epoch and compare it to the actual training loss.

If the computed and actual losses are not close: perhaps our Neural Network is not computing what we thought

- incorrect loss function
- mismatched features and labels

Assuming that the model's predictions are uninformed, due to random initialization of all weights (including bias) with mean 0

- Regressor expected to predict near 0 values
    - so per-example error is $\mathbf{y}^{(\mathbf{i})}$
    - translate error into Loss $\mathcal{L}^{(\mathbf{i})}$ depending on Loss function (e.g., MSE, MAE)
- Classifier expected to predict equal probability for each class
    - logit value of $\log \frac{1}{\text{number of classes}}$ for each class j
    - corresponding to equal probability across classes that label is class j
    - Loss is negative of this value: $\mathcal{L}^{(\mathbf{i})} = -\log \frac{1}{\text{number of classes}}$
        - we are *minimizing* loss

Overfit

One danger in training a large Neural Network with a small number of examples is overfitting

- Low training loss
    - model has used the overly large number of weights to memorize the training set
- High validation loss

We can take advantage of this property to gain confidence that our Neural Network is performing the desired task

- fit the model on a small subset of the Training examples
- expect near 0 loss. If not
    - mismatched features and labels ?

[Regularize (https://karpathy.github.io/2019/04/25/recipe/#4-regularize)](https://karpathy.github.io/2019/04/25/recipe/#4-regularize)

Regularization is often used to minimize the chances of overfitting

- improve out of sample prediction

Regularization is best performed *after* you have already successfully fit a model.

Tune (https://karpathy.github.io/2019/04/25/recipe/#5-tune)

Use random search, rather than grid search, for tuning hyper-paramters.

Ensembles (https://karpathy.github.io/2019/04/25/recipe/#6-squeeze-out-the-juice)

Ensembing (running a cohort of models) works for Deep Learning in the same way as in Classical Machine Learning.

In [2]:

```python
print("Done")
```

```
Done
```