A blog post on

Visualising the loss landscape

Sujal Vijayaraghavan

Sunday, 26 September 2021

> "You can't step past [someone] in this [two] dimension. Observe this two-dimensional egg. If we were in the third dimension, looking down, we'd be able to see an unhatched chick in it, just as a chick inside a three-dimensional egg could be seen by an observer in the fourth dimension"
>
> - Prof. Farnsworth
> E15S7 *Futurama*

1 Prologue

When plotting and monitoring an architecture's loss function, we are looking at the loss landscape through a toilet paper tube. On the y-axis is the loss function and on the x the epochs. We have only a one-dimensional view of the loss function's space, and that, too, for a small range of gradients of the parameters.

What if we could see, say, the 175bn-dimensional loss space for GPT[1] on a range of gradients of those billions of parameters? Well, let's not kid ourselves. What if we could at least see the loss space in a reduced dimensional space, say, two? This introductory article briefly demonstrates how it is achieved, and how simple yet fascinating an idea it is.

While training neural nets, the loss function we plot is a function of the model architecture, the optimisation method, initialisation, *etc*. The resulting loss function plotted varies differently for different configurations of them. Yet, the effect of these choices on the resulting objective is unclear. We visualise loss function convergence as much for fun as to gain insights into the training. Visu- alisation of landscapes offers richer insights and helps explain why neural nets can optimise even extremely complex non-convex functions and why the minimum optimised generalises well. (For example, one useful insight into skip connexions was observed with such visualisation: they prevent the model from turning the loss landscape chaotic and are hence useful in training[6]).

Note

- This piece was originally written on Medium
- For simplicity and brevity, this article works around the MNIST dataset and explores several aspects of visualisation. More scholastic research and text on other datasets and aspects of this idea can be found in papers published on this problem[4,6,2]

2 Introduction

Let $\boldsymbol{\theta}$ be a list of all the parameters in a neural network. Let $\mathcal{L}(y, t; \boldsymbol{\theta})$ be the loss function, where y is the prediction and t the target. We typically plot the convergence of $\mathcal{L}$ to visualise the difference between y and t. Our goal here slightly is different. The inputs to this loss function, y and t, are held constant. In other words, the plot we intend to draw is a function of $\boldsymbol{\theta}$, *i.e.*, $\mathcal{L}(\boldsymbol{\theta}; y, t)$, or simply $\mathcal{L}(\boldsymbol{\theta})$. What this is saying is, for a given domain, what do our choices of the network architecture, optimiser, loss function, *etc*., look like on a graph.

Needless to say, $\boldsymbol{\theta}$ has a high dimensionality (the simple network in the code snippet in Listing 2 has 1,199,882 dimensions!). And sadly, reality restricts us to only three—at least as far as visualisation is concerned. So, we need to reduce this dimensionality. One simple way is to move from the Euclidian space to a hyperspace of lower dimensions (one or two). In simpler terms, $\boldsymbol{\theta}$ with d dimensions in the Euclidian space can be thought of as having a one-dimensional representation in the hyperspace. The plot of $\mathcal{L}(\boldsymbol{\theta}; y, t)$ is then a two-dimensional graph. Likewise, if we assume $\boldsymbol{\theta}$ to be two-dimensional in the hyperspace, we have a desirable 3D plot.

3 Moving to lower-order dimensions

3.1 One dimension by linear interpolation

Plotting the loss as a 1D graph[3] is straightforward: it starts by measuring the loss from one set of parameters, $\boldsymbol{\theta}$, to another, $\boldsymbol{\theta}^*$, where $\boldsymbol{\theta}$ could be a randomly initialised set headed for the (already-found) local (or even global) optimum, $\boldsymbol{\theta}^*$.

All the possible set of parameters along this line can simply be from 0 to 1, both weighed to add up to 1, and is given by the non-Euclidian transformation, τ:

$$\tau(\alpha; \boldsymbol{\theta}, \boldsymbol{\theta}^*) = \alpha\boldsymbol{\theta}^* + (1 - \alpha)\boldsymbol{\theta} \tag{1}$$

With α, a scalar, ranging from 0 to 1 on the x-axis and the loss $\mathcal{L}(\tau(\alpha; \boldsymbol{\theta}, \boldsymbol{\theta}^*))$ on the y, we have a one-dimensional loss landscape (Figure 1a). The range could be different and, since the resulting plot is configuration-dependent, must be discerned accordingly.

3.2 Two-dimensional landscape

Plotting in a two-dimensional space[3,4] is just as simple in principle. Given any random or optimised set of parameters, $\boldsymbol{\theta}^*$, we venture in two directions, δ and η. In both directions, we take small steps, α and β, respectively. The resulting graph is, therefore, a function of α and β.

Since δ and η are direction vectors, they represent directions in each dimension of $\boldsymbol{\theta}^*$, *i.e.*, δ and η have the same dimensionality as $\boldsymbol{\theta}^*$, which could both be sampled from a random Gaussian.

The non-Euclidian transformation is given by τ:

$$\tau(\alpha, \beta; \boldsymbol{\theta}^*) = \boldsymbol{\theta}^* + \alpha\delta + \beta\eta \tag{2}$$

A contour plot may then be drawn from $\alpha \in [0, 1]$ and $\beta \in [0, 1]$, or any range. In Figure 2a, both are in the range $[20, 20]$.

This graph gives us two pieces of information: the rate at which we can move in the two directions, and the range to use to get a larger picture. The snippet (Listing 7) that generated Figure 2a (from the same $\boldsymbol{\theta}^*$) is used to generate Figure 2b, except this time, α and β are in $[25000, 25000]$.

4 Utility

Such visualisations are extremely useful when comparing optimisation methods and network architectures. However, this is not always possible, because several kinds of layers do not contribute to any change in the effective result of the model. For example, ReLU does not change the network's behaviour if the effective result with and without the layer is the same, such as when the input to ReLU is scaled by a factor and the output divided by the same factor. Layers such as batch-wise normalisation also act likewise in the network's invariance to such layers. This prevents us from making meaningful comparisons.

On the other hand, there are cases when perturbing large weights of a network by some unit has very little effect. Other times, doing the same on sensitive weights by the same unit can wreak chaos. To tackle this problem, the randomly generated direction vectors, say, δ, can be normalised to have the same direction as $\boldsymbol{\theta}^*$. More specifically, each filer in δ is made to have the same direction as the corresponding layer in $\boldsymbol{\theta}^*$ [6]:

$$\delta_{i,j} = \frac{\delta_{i,j}}{\|\delta_{i,j}\|} \|\boldsymbol{\theta}_i^*\| \tag{3}$$

Doing this is shown to result in contour plots' being able to capture the distance scale of loss surfaces (comparing Figure 2a and Figure 2b, for example) when the directions (δ and η) are normalised in this fashion. [3]

4.1 Determining the solution space area

Consider two trained setw of parameters, $\boldsymbol{\theta}^{\text{l}}$ and $\boldsymbol{\theta}^{\text{s}}$. The former is trained on a dataset with larger batches and the latter smaller. Interpolating from one to the other shows the width of possible solution space depending on batch size.

For instance, consider the contours in Figure 2c generated from a model trained with a batch size of 256, as opposed to Figure 2b, which was trained with a batch size of 64.

With the two trained parameters, $\boldsymbol{\theta}^{\text{l}}$ and $\boldsymbol{\theta}^{\text{s}}$, the following set of parameters is interpolated:

$$\tau(\alpha; \boldsymbol{\theta}^{\text{l}}, \boldsymbol{\theta}^{\text{s}}) = \boldsymbol{\theta}^{\text{l}} + \alpha(\boldsymbol{\theta}^{\text{l}} - \boldsymbol{\theta}^{\text{s}}) \tag{4}$$

This interpolation is now a function of the parameters based on batch size between 64 and 256. This comparison helps discover any potential configuration bach-size-wise that yields a better optimum. And sure enough, we know when to stop (Figure 1).

As a side note, the many hills and valleys explain that when a large batch is used, the resulting weights tend to be smaller than with smaller batches. [3]

While this is just an illustration to finding good solution spaces based on batch size, other parameters and hyper-parameters could also be optimised. A few key takeaways include the following: [3]

- Wider networks prevent chaotic landscapes
- Skip connexions widen solution space (or minimisers)
- Chaotic landscapes have shallow valleys and result in worse train and test losses
- Visually flatter landscapes correspond to consistently lower test errors

5 Epilogue

- Normalisation of the entire parameter set at once (as opposed to a filter-by-filter basis) has also been tried out [4]
- More discussions on the flatness versus sharpness of contours and graphs and their uses are discussed in detail [5,3]
- *Loss Landscape* [7] on GitHub has numerous utility functions and useful classes for further studies

5 References

1 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 1

2 Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. In *International Conference on Machine Learning*, pages 1019–1028. PMLR, 2017. 1

3 Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. *arXiv preprint arXiv:1412.6544*, 2014. 2, 3

4 Daniel Jiwoong Im, Michael Tao, and Kristin Branson. An empirical analysis of the optimization of deep network loss surfaces. *arXiv preprint arXiv:1612.04010*, 2016. 1, 2, 3

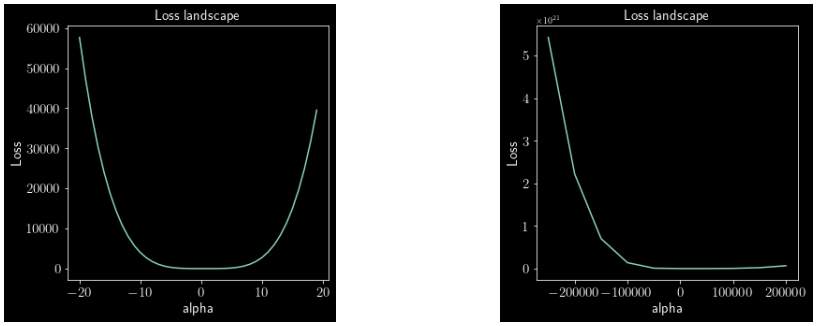
5 Kenji Kawaguchi, Leslie Pack Kaelbling, and Yoshua Bengio. Generalization in deep learning. *arXiv preprint arXiv:1710.05468*, 2017. 3

6 Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018. 1, 3

7 Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Neural Information Processing Systems*, 2018. 3

6 Appendix

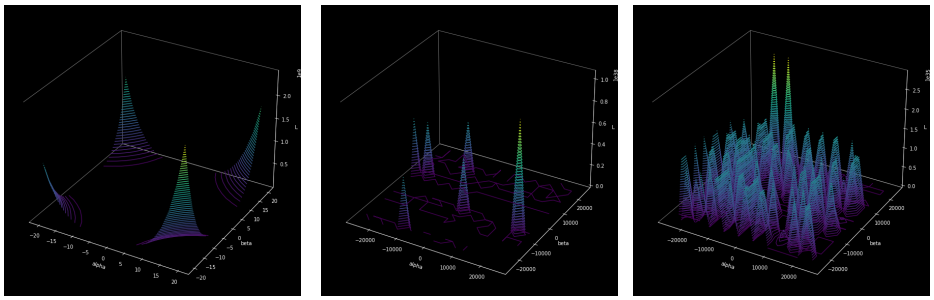
6.1 Figures



(a) A 1D loss landscape plotted along the linearly interpolated set of parameters with Listing 5

(b) Loss landscape as a function of network parameters varied by batch size

Figure 1. One-dimensional loss landscapes



(a) $\alpha, \beta \in [-20, 20]$

(b) $\alpha, \beta \in [-2.5M, 2.5M]$; model training batch size = 64

© $\alpha, \beta \in [-2.5M, 2.5M]$; model training batch size = 256

Figure 2. Contour plots

6.2 Listings

```
from torchvision import transforms as T

transform = T.Compose([
    T.ToTensor(),
    T.Resize((28,28)),
    T.Normalize((0.1307,), (0.3081,))
  ])
dataset = Dataset(root='data', download=True, train=False, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset,
                                batch_size=len(dataset),
                                pin_memory=True, num_workers=4)
```

Listing 1. A sample PyTorch data loader for the MNIST dataset

```
import torch
from torch.nn import functional as F


class Net(torch.nn.Module):
  def __init__(self):
    super(Net, self).__init__()

    self.conv1 = torch.nn.Conv2d(1, 32, (3,3))
    self.conv2 = torch.nn.Conv2d(32, 64, (3,3))

    self.drop1 = torch.nn.Dropout2d(0.25)
    self.drop2 = torch.nn.Dropout2d(0.5)

    self.fc1 = torch.nn.Linear(9216, 128)
    self.fc2 = torch.nn.Linear(128, 10)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.relu(self.conv2(x))
    x = self.drop1(F.max_pool2d(x, 2))

    x = x.flatten(1)
    x = F.relu(self.fc1(x))
    x = self.drop2(x)
    x = self.fc2(x)

    return F.log_softmax(x, 1)
```

Listing 2. A simple NN with two convolutional layers enough to classify the MNIST dataset sufficient for demonstration

```
from torch.nn.utils import (
  parameters_to_vector as Params2Vec,
  vector_to_parameters as Vec2Params
)

learnt_model = 'models/learnt.pt'

```

```
learnt_net = Net()
learnt_net.load_state_dict(torch.load(learnt_model))
theta_ast = Params2Vec(learnt_net.parameters())

infer_net = Net()
theta = Params2Vec(infer_net.parameters())

loss_fn = torch.nn.NLLLoss()
```

Listing 3. A learnt model (with over 98% accuracy) is loaded. Interpolation starts from a random initialisation

```
def tau(alpha, theta, theta_ast):
  return alpha * theta_ast + (1 - alpha) * theta
```

Listing 4. A function implementing linear interpolation of the parameters from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}^*$

```
losses = []

for alpha in torch.arange(-20, 20, 1):
  for _, (data, label) in enumerate(dataloader):
    with torch.no_grad():
      Vec2Params(tau(alpha, theta, theta_ast),  infer_net.parameters())
      infer_net.eval()
      prediction = infer_net(data)
      loss = loss_fn(prediction, label).item()
      losses.append(loss)
```

Listing 5. Computing the loss of all parameters from the random set to the globally optimised one

```
def tau_2d(alpha, beta, theta_ast):
  a = alpha * theta_ast[:,None,None]
  b = beta * alpha * theta_ast[:,None,None]
  return a + b
```

Listing 6. Computing the loss of all parameters from the random set to the globally optimised one (Code snippet is for illustration. Note the time complexity; model parallelism should come in handy)

```
x = torch.linspace(-20, 20, 20)
y = torch.linspace(-20, 20, 20)
alpha, beta = torch.meshgrid(x, y)
space = tau_2d(alpha, beta, theta_ast)

losses = torch.empty_like(space[0, :, :])

for a, _ in enumerate(x):
  print(f'a = {a}')
  for b, _ in enumerate(y):
    Vec2Params(space[:, a, b], infer_net.parameters())
    for _, (data, label) in enumerate(dataloader):
      with torch.no_grad():
        infer_net.eval()
```

```
        losses[a][b] = loss_fn(infer_net(data), label).item()
```

Listing 7. A demo snippet for illustration; used to generate the contour plot shown below

```
def tau_compare(alpha, theta_l, theta_s):
  return theta_s + alpha * (theta_l - theta_s)
```

Listing 8. A demo snippet for illustration; used to generate the contour plot shown below