10-707: Advanced Deep Learning **Spring 2020**

Lecture 6: February 3

Lecturer: Andrej Risteski Scribes: Chirag Pabbaraju, Srinivasa Pranav, Karmesh Yadav, Terrance Liu

6.1 Local Minima

No one really knows why local minima are good.

- For linear nets (activation is linear function), the network is equivalent to matrix multiplication. So, local minima are global minima. Similarly, all stationary points are global minima for linear ResNets.
- Small neural networks have many bad local minima.
- **Overparameterization** (networks with # parameters > # training samples) with random initialization somehow leads to finding local minima with low training error.

Saddle point *intuition*:

- Permuting order of neurons in a layer gives the same loss.
- Isolated minima corresponding to these permutations have saddle points between them since averaging the parameters within a network likely leads to larger loss.

6.2 SGD: Stochastic Gradient Descent

Instead of approximating $\nabla_\theta l(f_\theta(x), y)$using all of the training data, we randomly pick training sample (x_i, y_i) and use it to approximate the gradient:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta l(f_\theta(x_i), y_i)$$

Note: subscripts on θ are time steps.

6.2.1 Mini-Batch Gradient Descent

Usually, people approximate the gradient using a mini-batch B_j of training samples:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \frac{1}{|B_j|} \sum_{(x_i, y_i) \in B_j} l(f_\theta(x_i), y_i)$$

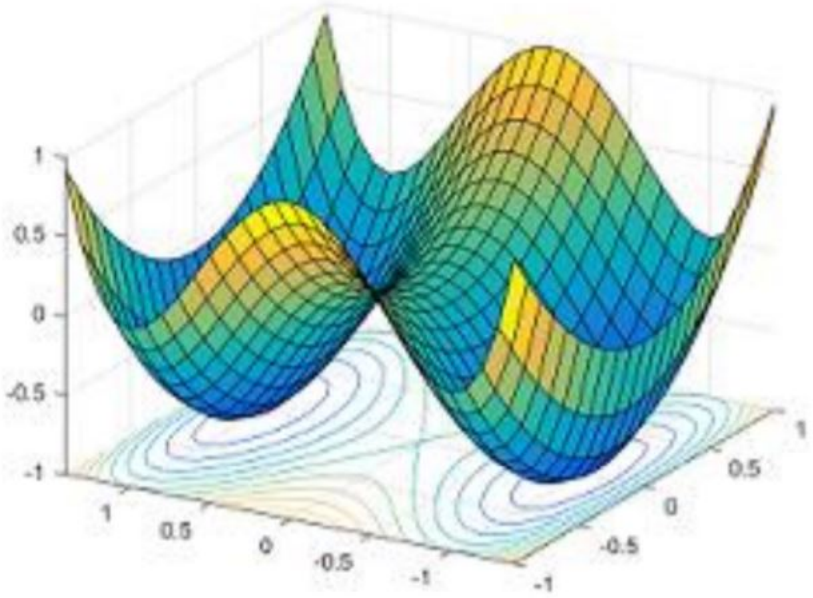


Figure 6.1: Saddle Point

In each iteration, a different training batch is used until all batches B_j have been used. If there are m batches, then an epoch involves m gradient updates, each using a different training batch.

6.2.2 Pros and Cons of SGD

- Con: slower convergence because gradient is noisier (may not move the weights in the exact direction of the minimum)
- Pro: SGD has a regularization effect and the local minima found via SGD tend to have better generalization than other local minima found using GD
- Pro: the noisier gradients help escape saddle points

6.2.3 Intuition for SGD Saddle Point Escape

Let $f(x) = \frac{1}{2}x^T Ax$ and A be symmetric and have a negative eigenvalue. It's clear that $\nabla_x f(x) = Ax$. We can write x_t in terms of the eigenvalues λ_i and eigenvectors v_i of A:

$$x_t = \sum_i c_{i,t}\lambda_i v_i$$

Then, the GD update rule becomes, by induction,

$$x_{t+1} = \sum_i (1-\eta\lambda_i)c_{i,t}\lambda_i v_i = \sum_i (1-\eta\lambda_i)^{t+1}c_{i,0}\lambda_i v_i$$

For $\lambda_i < 0$, the ith component grows every iteration. This means that movement in the ith component will cause SGD to escape saddle points.

6.3 Second order methods on the cheap

Gradient descent on poorly conditioned objective functions e.g. quadratics with ellipsoidal contours can oscillate a lot and take a large number of iterations to converge. One possible way to mitigate this is to use Newton's method which has the following update rule

$$x_{t+1} \leftarrow x_t - \eta(\nabla^2 f(x_t))^{-1}\nabla f(x_t)$$

However, this requires inverting a Hessian matrix at each iteration, which is quite expensive.

6.3.1 AdaGrad[2]

AdaGrad tries to approximate the Newton update in a sense with a diagonal matrix in place of the true Hessian. AdaGrad has different step sizes for updates to different parameters, based on history. Formally, define the diagonal matrix G_t at time step t with diagonal entries $(G_t)_{ii}$ as follows:

$$(G_t)_{ii} = \sqrt{\sum_{j=1}^{t-1}(\nabla f(x_t)_{ii})^2}$$

Then,

$$(G_t)_{ii}^{-1} = \frac{1}{\sqrt{\sum_{j=1}^{t-1}(\nabla f(x_t)_{ii})^2}}$$

AdaGrad then has the following update rule:

$$x_{t+1} \leftarrow x_t - \eta G_t^{-1}\nabla f(x_t)$$

The factor $(G_t)_{ii}^{-1}$ scales down the learning rate for the components of x that have seen comparitively large gradient updates in the past time steps.

6.3.2 RMSProp

RMSProp is a slight modification to AdaGrad. The potential issue with AdaGrad that RMSProp is is trying to solve is that the scaling factor in each component $(G_t)_{ii}^{-1}$ increases with increasing t (since we are adding to it a positive quantity at each time step). This might cause the learning rates for the parameters to decay to extremely small values over time, so that the parameters essentially don't change much over further iterations. To mitigate this, instead of keeping track of the sum of squared norms of the gradients in each component, RMSProp maintains an exponentially weighted average over time. Formally, RMSProp performs the following updates

$$(g_{t+1})_i = \beta(g_t)_i + (1-\beta)(\nabla f(x_t)_{ii})^2$$
$$(x_{t+1})_i = (x_t)_i - \frac{\eta \nabla f(x_t)_i}{\sqrt{(g_{t+1})_i}}$$

The parameter β allows us to give more weightage to the behaviour of the most recent gradients in comparison to equally weighing in gradients from all past time steps.

6.3.3 AdaGrad + Momentum = Adam [7]

Recall Polyak's momentum method which collects momentum from the past iterates while doing gradient descent

$$v_{t+1} = -\nabla f(x_t) + \beta v_t$$
$$x_{t+1} = x_t + \eta v_{t+1}$$

Adam, which is the default optimization module in most packages today, combines momentum and AdaGrad to yield the following updates:

$$v_{t+1} = \beta_1 v_t + (1-\beta_1)\nabla f(x_t)$$
$$(g_{t+1})_i = \beta_2 (g_t)_i + (1-\beta_2)(\nabla f(x_t)_{ii})^2$$
$$(x_{t+1})_i = (x_t)_i - \frac{\eta v_{t+1}}{\sqrt{(g_{t+1})_i}}$$

The first step above collects momentum from the past time steps, while the second step maintains an exponentially weighted average of the norms of the gradients per component to scale the learning rates per component appropriately.

In spite of its widespread usage, Adam does not converge provably, even on certain convex functions [9].

6.4 Batch Normalization

6.4.1 Introduction

Yann Lecun et al. [8] discovered in 1998 that if you normalize or whiten your input, to have mean equal to 0 and variance equal to 1, then the training speeds up. However as the normalized input is passed through the different levels of hidden layers, the distribution gets transformed and it no longer remains a whitened distribution. The obvious question was if it would help to normalize the inputs at the different layers of the network.

- Batch normalization [6] is an attempt to do this
- Each unit's pre-activation is normalized (mean subtraction, stddev division)
- During training, mean and stddev is computed for each minibatch
- Backpropagation takes into account the normalization
- At test time, the global mean / stddev across batches is used.

The following algorithm gives a brief idea about how batch normalization is carried out over a network layer.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ \\mini-batch mean
$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ \\mini-batch variance
$\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ \\normalize
$y_i \leftarrow \gamma \widehat{x_i} + \beta \equiv \mathbf{B}\mathrm{N}_{\gamma,\beta}(x_i)$

Basically, the algorithm first calculates the mean ($\mu_{\mathcal{B}}$) and variance ($\sigma^2_{\mathcal{B}}$) of the mini-batch, $\mathcal{B}$. Then it normalizes the input data using the calculated mean and variance to get the normalized input ($\hat{x}_i$). Finally, the learnt linear transformation (γ and β are the learnt parameters) shifts the distribution back to the linear regime of the non-linearities used in the network.

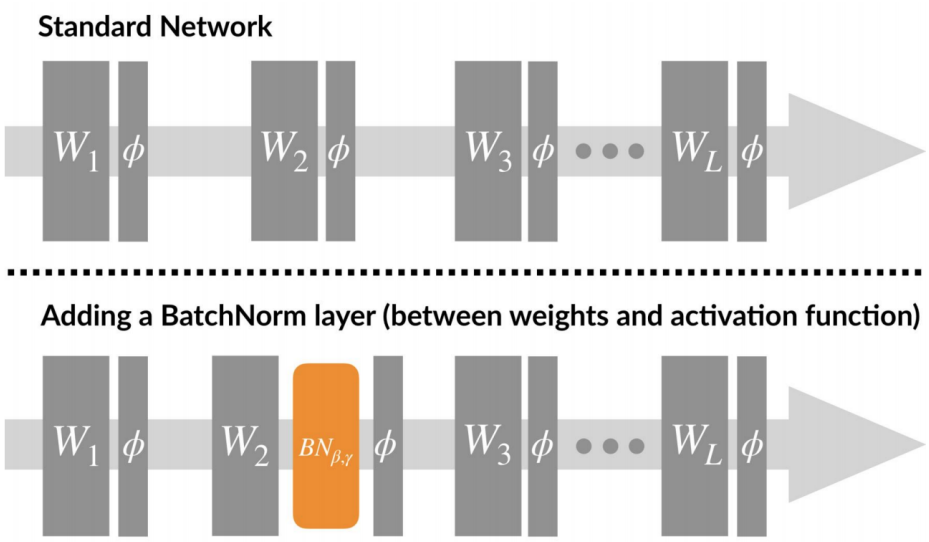


Figure 6.2: Adding a batch norm layer to a neural network between the weight matrix and non linearity [1]

6.4.2 How Does Batch Normalization Help Optimization?

Comparing the version of training with and without BatchNorm (BN), the one with BN is able to converge even with higher learning rates like 0.5 while the network without BN diverges. This allows networks with BN to train and converge faster.

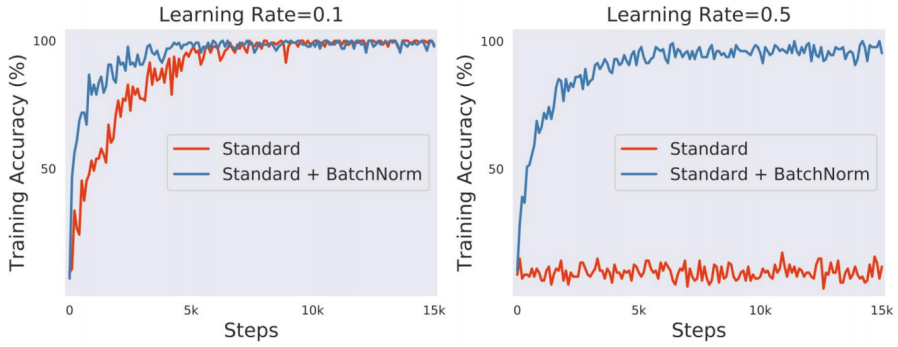


Figure 6.3: Training curves of a Standard Network and a Standard Network with BN at different learning rates

[1] Link: http://gradientscience.org/batchnorm/

The reason why BatchNorm helps improve training is still debated. Here are a few hypotheses regarding why it might help:

- BatchNorm helps keep the non-linearities in the active regime, where the weight parameters can actually contribute towards improvement of the performance of the network.
- Stabilizes the training by bringing the inputs for every layer throughout the training to one distribution with mean 0 and variance 1. This solves the problem of internal covariate shift.
- Hessian of the loss is better, which means there is less jitteriness in gradient descent and this leads to faster convergence. (Proved for 1-layer linear nets [8])

There is also some reasoning available to explain why mini-batches are required for BatchNorm:

- Since hidden units depend on parameters, can't compute mean/stddev once and for all.
- Adds stochasticity to training, which might provide regularization

However a recent paper [10] challenged the old notions of why BatchNorm works. By looking at the histogram of activations for architectures with and without BN, they show that the distribution at nodes of the network without BN is also stable and internal covariate shift might not be a factor at play.

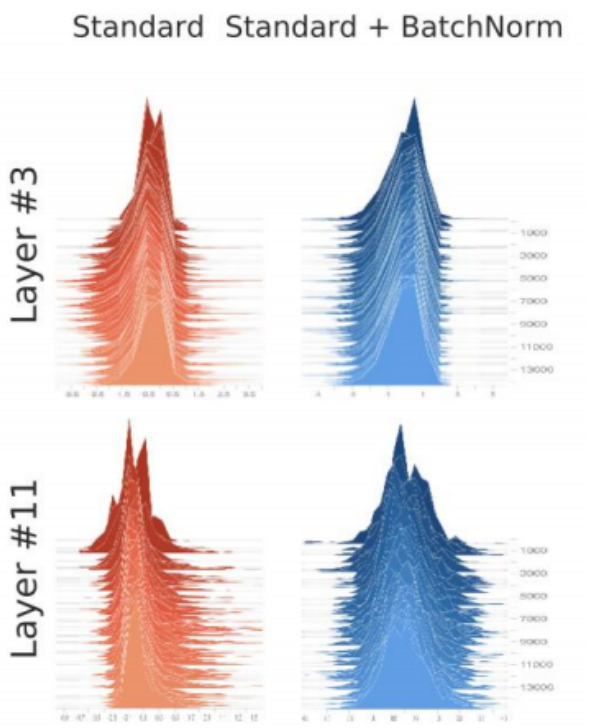


Figure 6.4: Activation histogram for a Standard Network and a Standard Network with BN at different layers

Experiment: The authors tried adding Gaussian noise after batch norm which changes with time and is different at each layer. This undoes whatever the BN layer did because it adds noise to the layer output, making the inputs substantially more unstable for further layers.

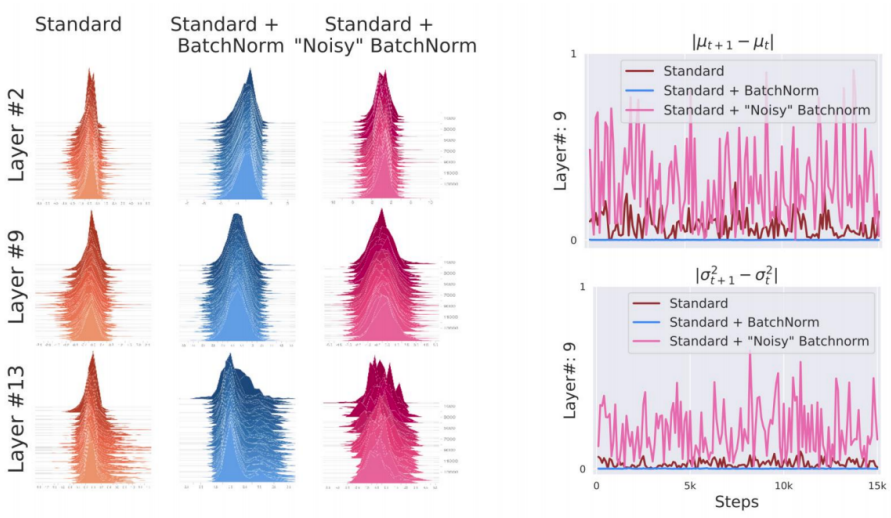


Figure 6.5: Activation histogram after adding noise to the output of batch norm

However, training performance still remains nearly the same as seen in the next plot.

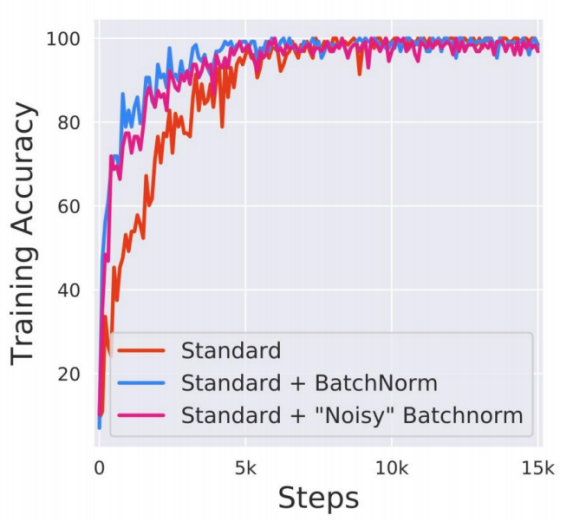


Figure 6.5: Accuracy curve for the Standard, Standard + BatchNorm and Standard + 'Noisy' BatchNorm training

This made the authors question the old hypothesis of why BatchNorm helps. The authors then came up with the following two new hypotheses:

- BN makes the objective easier for optimization by improving the Lipschitz constant of the loss function.
- The Hessian of the "new" loss evaluated in the direction of the gradient is smaller in the network with BN. This basically means that the first order term is more accurate, thus improving training.

6.5 Initialization

We present two common ways of initializing weights and a general intuition for how they work. The goal of initialization is to make sure the non-linearities are in the "active" (linear) regime. Note: the point is to provide some intuition, so the material was presented with some handwaviness.

6.5.1 Lecun [8]

We have

$$\mathbb{E}[w_{ij}] = 0 \text{ and } \text{Var}[w_{ij}] = \frac{1}{\text{fan-in}}$$

where fan-in is the number of inputs to the unit y_i

Intuition: Suppose your x_i's have mean 0 and variance 1, then your output $y_i = \sum_j w_{ij}x_j$ has

$$\mathbb{E}[y_i] = 0 \text{ (linearity of expectation)}$$

and

$$\begin{aligned}
\text{Var}[y_i] &= \text{Var}\left[\sum_j w_{ij}x_j\right] \\
&= \sum_j \text{Var}[w_{ij}x_j] \quad \text{(independent random variables)} \\
&= \sum_j \left(\text{Var}[w_{ij}]\text{Var}[x_j] + \text{Var}[w_{ij}](\mathbb{E}[x_j])^2 + \text{Var}[x_j](\mathbb{E}[w_{ij}])^2\right) \\
&= \sum_j \left(\frac{1}{\text{fan-in}} + 0 + 0\right) \\
&= (\text{fan-in})\frac{1}{\text{fan-in}} \\
&= 1
\end{aligned}$$

In addition, note that generally, our activations maintain the variance of y_i. For example, the gradient of tanh is around 1 in the linear regime (Figure 6.1), giving us that $\text{Var}[\tanh(y_i)] \approx 1$

Thus in this way, the neural network maintains outputs of mean 0 and variance 1 through every layer, staying in the linear regime of the activations.

6.5.2 Xavier [5]

We have

$$\mathbb{E}[w_{ij}] = 0 \text{ and } \text{Var}[w_{ij}] = \frac{2}{\text{fan-in} + \text{fan-out}}$$

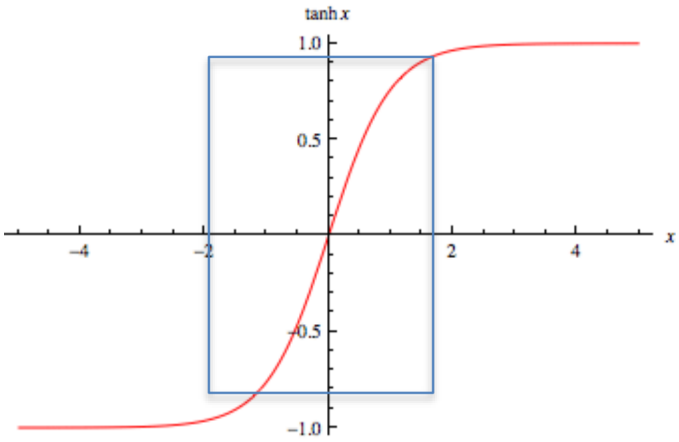


Figure 6.1: tanh in the linear regime

where fan-in is the number of inputs to the unit y_i and fan-out is the number of outputs from this unit.

The intuition is similar to the Lecun initialization, except that you also want to maintain the same mean and variance in the backprop step as well. In the same way as in the previous subsection, where you want the the forward step to have $\text{Var}[w_{ij}] = \frac{1}{\text{fan-in}}$, from the backward step, you want $\text{Var}[w_{ij}] = \frac{1}{\text{fan-out}}$. Thus, this initialization tries to combine these two intuitively.

To achieve this mean and variance, people typically sample from some distribution such as

- $\text{N}\left(0, \frac{2}{\text{fan-in+fan-out}}\right)$
- $\text{Unif}\left(\left[\frac{\sqrt{-6}}{\sqrt{\text{fan-in+fan-out}}}, \frac{\sqrt{6}}{\sqrt{\text{fan-in+fan-out}}}\right]\right)$

6.6 Surprises

6.6.1 Mode Connectivity

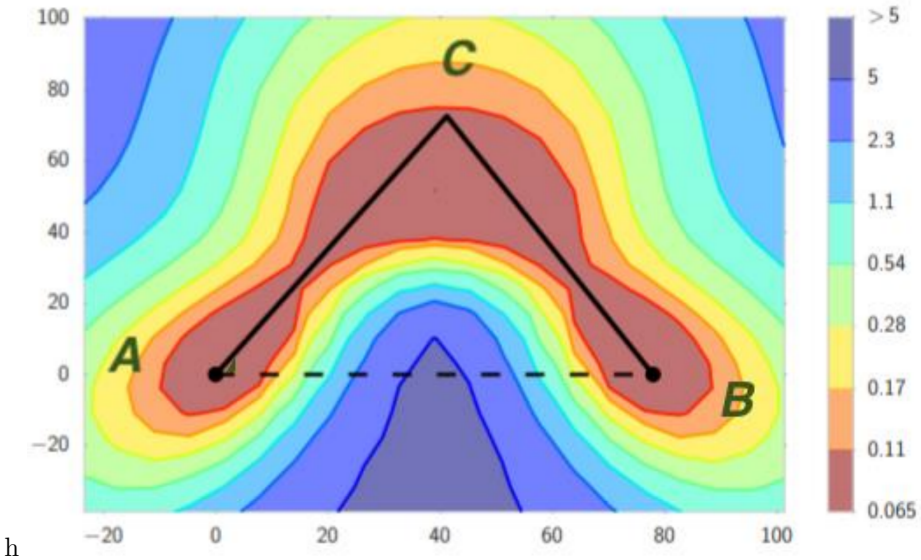
If we train a neural network multiple times, we can connect the local minima with very simple paths (eg. piecewise linear function) of near-same cost (Figure 6.2) [7, 4, 1]. This is surprising, since local minima were thought to be isolated points that would make it difficult to move from one to the other (Figure 6.3).

6.6.2 Lottery Hypothesis [3]

Main idea: There exist smaller subnetwork structures that could have been trained to achieve results comparable to that of typical model architectures we train today. In other words, models are much more complex than is in principle necessary.

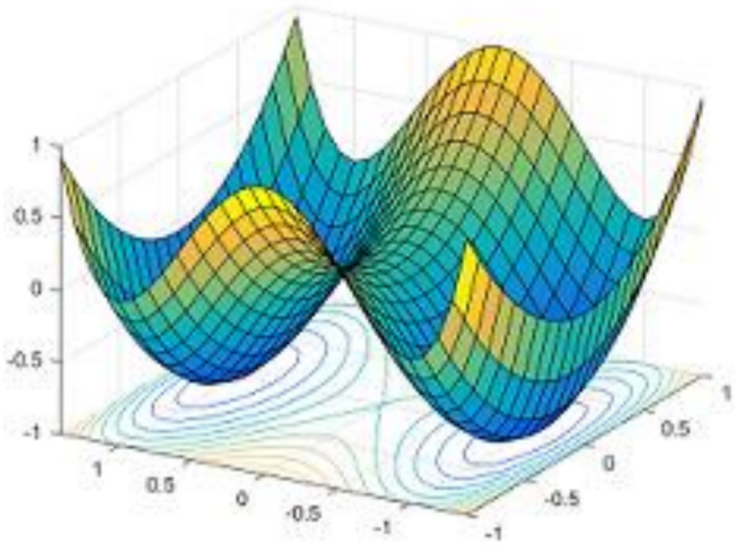
Repeating the following procedure,

1. Initialize a network and train it



h

Figure 6.2: Mode connectivity



h

Figure 6.3: Initial intuition that training would achieve isolated solutions

2. "Prune" superfluous structure. (e.g. smallest weights)
3. Reset unpruned weights to values in 1
4. Repeat 2 and 3

one can train subnetworks that achieve similar results that are anywhere from 15% to 1% of the size of the original networks.

References

[1] Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred A. Hamprecht. Essentially no barriers in neural network energy landscape, 2018.

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[3] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2018.

[4] C. Daniel Freeman and Joan Bruna. Topology and geometry of half-rectified network optimization, 2016.

[5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[9] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[10] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.