Deep Learning Theory and Applications

Kernel Methods and Neural Tangent Kernels

between NN and kernel

Yale

CPSC/AMTH 663

Outline

- Kernel Methods and Neural Networks
- Linearization of gradient
- Lazy Regime
- Gradient Flow prove why SGD work
- Kernel Methods and Generalization

NTK and Generalization

- In the **infinite width** or **large scale scenario** neural networks become equivalent to a kernel method
- This creates a very interesting connection between neural networks and kernel methods
- This kernel method uses a specific kernel called the **Neural Tangent Kernel**
- Neural networks become equivalent to kernel ridge regression
  - Such a method is a linear method on top of a non-linear feature map
- Using SGD to solve this starting from low norm-initialization leads to a low functional norm, stable, generalizable solution

NTK and Gradient Descent

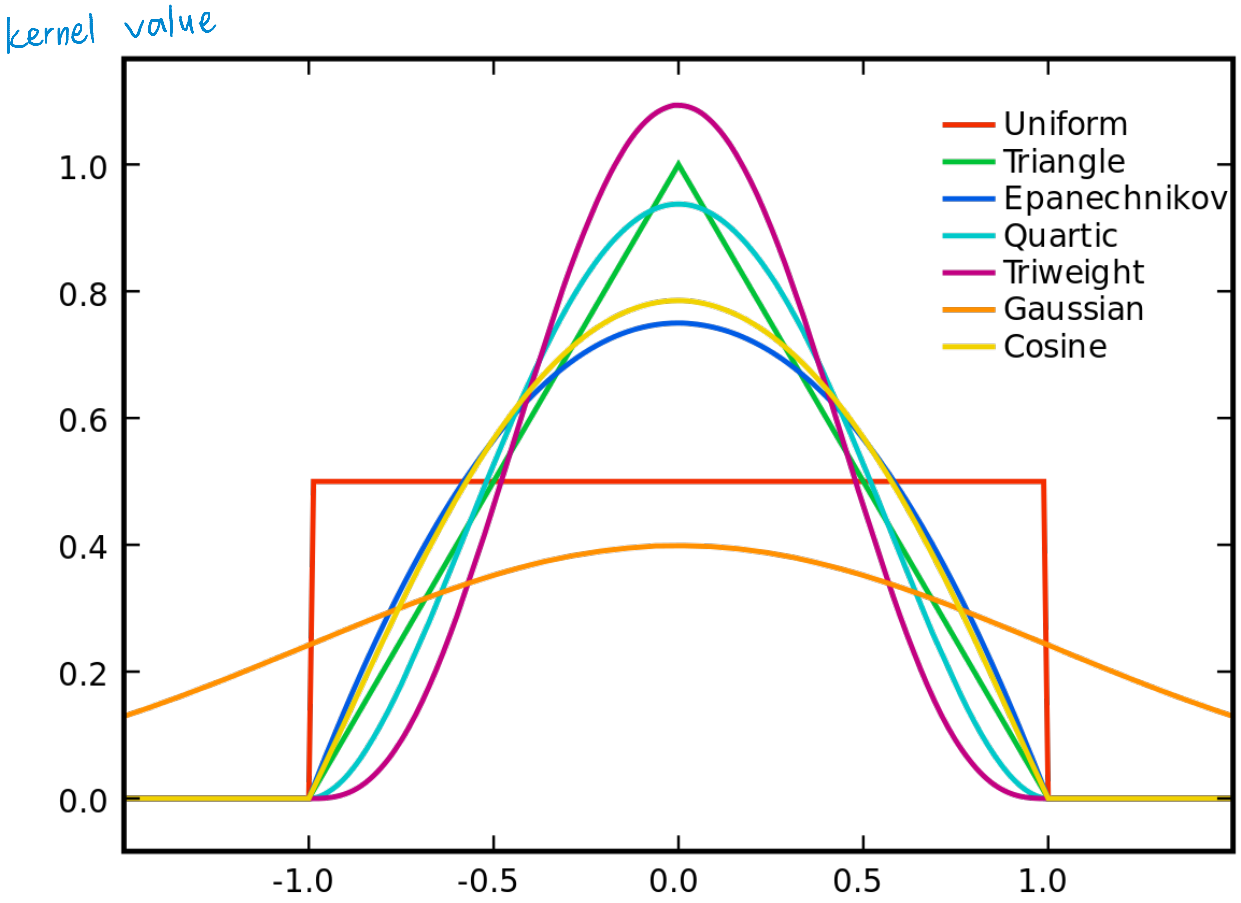
- In the **infinite width** or **large scale** scenarios the weights of a neural network don't move very much during training
- The derivative of weights WRT Loss function L is very close to linear also don't change a lot
- In this regime SGD can be written as a kernel gradient with the neural tangent kernel
- Using this kernel one can study gradient descent more mathematically
- Interesting result: One can prove that gradient descent converges to the global minima here

What is a kernel Method?

- Intuitively, A **kernel method** is a method that uses similarities between points rather than reasoning about features to perform a task (clustering, classifying etc)

- A kernel function is a similarity function between datapoints on the basis of the feature vector
  - Ex: Gaussian kernel
- Similarity functions are symmetric and non-negative

Distance to Affinity via Kernels



Real-valued
Non-negative
Symmetric

Affinities correlations in this hidden hypothetical space

Kernel Methods

- ***Mathematical definition**: K is a kernel function if it can be written as the inner product of two **feature maps** in an m dimensional space*
- ***K(x, y) = <f(x), f(y)>***
- *Here K is the kernel function, x, y are n dimensional inputs.*
- *f is a **feature-map** from n-dimension to m-dimension space*
- *< x,y> denotes the dot product*
- *Usually m is much larger than n*

$f: R^n \rightarrow R^m \qquad m >> n$

Kernel Matrix

- Kernel Methods often use a matrix of pairwise similarities
- This matrix is often called a **kernel matrix M(i,j) = $K(x_i, x_j)$**
- Each row of a kernel matrix gives the similarity of a datapoint x_i with all other datapoints $x_1, x_2, \ldots x_n$
- Kernel matrices have nice properties if you use a proper similarity function
- Kernel matrices are positive-semidefinite matrice
- **PSD matrices $z^T Mz$ is always positive for any non-zero vector z** quadatic form
  - They are symmetric matrices
  - Eigenvalues are all greater than or equal to 0

$\lambda_i \geq 0$

Kernel Trick

- **Mercer's Theorem Result:** *If M is a non-negative positive semi-definite matrix of dimension n x n, then it is the Gram matrix (matrix of inner products) of the data vectors in another space V*
  - *This space is usually higher dimensional and involves "lifting" of vectors via a feature map,* $x_i \rightarrow \phi(x)$

lift vector to another space

- *This space V could be infinite dimensional and does not have to be found!*
- *(But we do find them in NTK)*

Linear methods to find non-linear functions

kernel

kernel trick

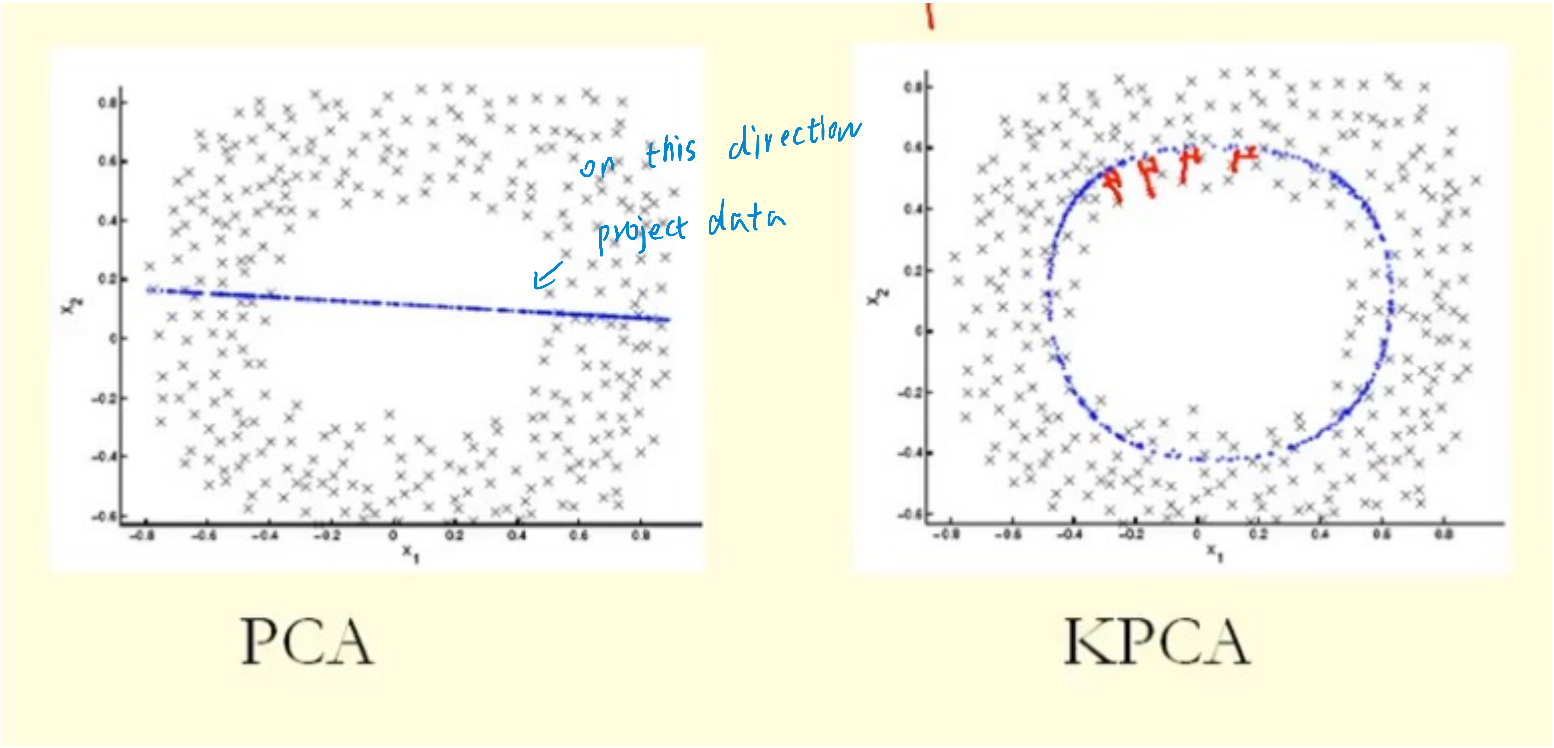
- Example: PCA → Kernel PCA

can't capture data's structu

dis of PCA : If data is non-Linear

- **PCA** is a method of dimensionality reduction where data is rotated into a new set of features that are linear combinations of original features
- New dimensions (coordinate) are eigenvectors of the feature covariance matrix $\sum$
- If instead of the feature covariance matrix, we use kernel matrix M of pairwise similarities between datapoints
  - Then we know this is a feature covariance matrix in some very high dimensional space
- Eigenvectors of the kernel matrix M give Kernel-PCA
  - Nonlinear directions of highest variation!
  - Modifications of this give Laplacian eigenmaps, diffusion maps

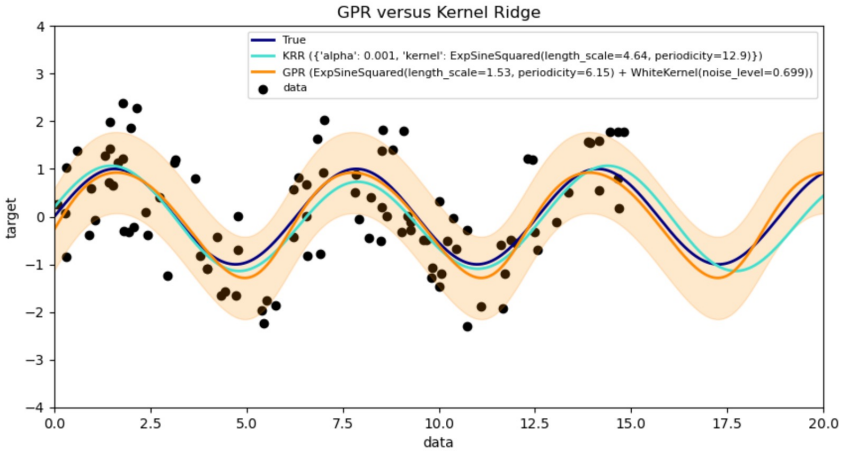
PCA vs Kernel PCA



Linear algorithm applied to find non-linear dimensions using kernel trick!

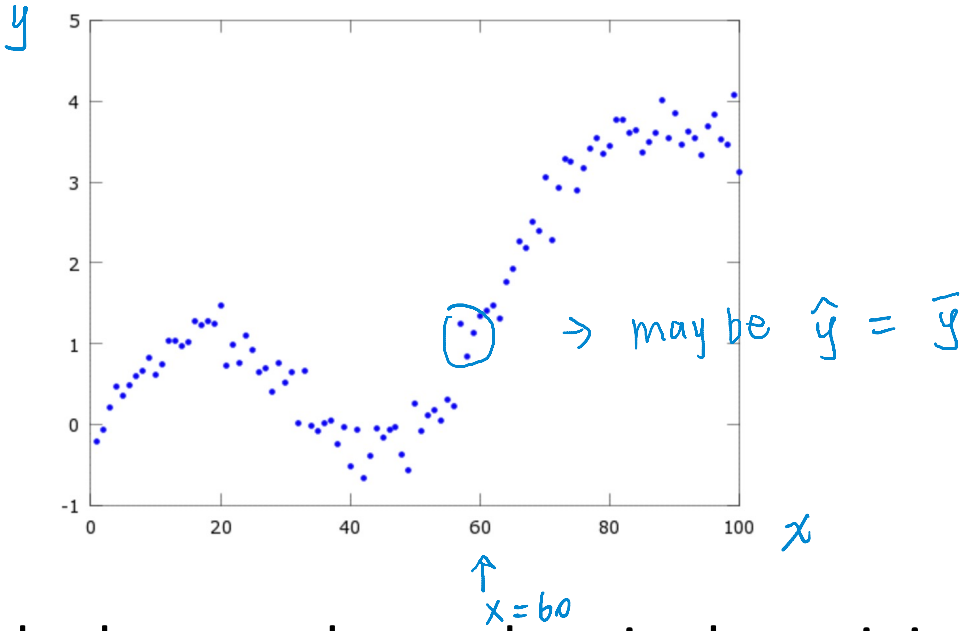
Kernel Ridge Regression

- Kernel ridge regression (KRR) combines ridge regression (linear least squares with l2-norm regularization) with the kernel trick
- Learns a linear function in the space induced by the respective kernel and the data
- For non-linear kernels, this corresponds to a non-linear function in the original space



Kernel Regression

- Suppose we want to know the value of y for a point x=60 for some non-linear function
- We are given the value of the function at several points $x_0\, x_1\, x_2 \ldots x_n$



- Idea: We should look at nearby x values in the training data, the closer the training datapoint is to x=60 the more relevant its y value for our estimation

Gaussian Kernel

- We can formalize this notion using a Gaussian Kernel

$$K(x^*, x_i) = e^{-\frac{(x_i - x^*)^2}{2\sigma^2}}$$

σ = bandwidth

- Then to estimate a y, we just need a weighting function

$$y^* = \frac{\sum_{i=1}^{m}(K(x^*, x_i)y_i)}{\sum_{i=1}^{m} K(x^*, x_i)}$$

- Our kernel has one parameter, sigma or the bandwidth, the NTK kernel has many, and the neural network tunes them during training in a certain regime

Ridge regression

- They perform ridge regression or minimize L2 loss because of the Neyshambur theory
- SGD prefers low norm solutions when initialized to low norm initializations

NTK as an explicit mapping

- NTK maps datapoints to a new feature space based on their derivative around the initialization of the neural network, i.e. how sensitive the output at that x is to changes in the parameters
- This is the new explicit feature map

$$\phi(x) = \nabla_w f(x, w_0)$$

gradient of NN f w.r.t weight given input x and initial weigh w_0

- Inner products in this feature map give you a notion of similarity and can be used to create a kernel matrix

$K = \phi^T \phi$

$$\begin{bmatrix} — \phi(\bar{x}_1)^T — \\ \vdots \\ — \phi(\bar{x}_n)^T — \end{bmatrix} \begin{bmatrix} | & & | \\ \phi(\bar{x}_1) & \cdots & \phi(\bar{x}_n) \\ | & & | \end{bmatrix}$$

Neural Tangent Kernel Matrix

p : # features

n : sample size

$$\underset{n}{\left[\quad \mathbf{H}(\mathbf{w}_0) \quad\right]}_{n} = \left[\quad \nabla_w \mathbf{y}(\mathbf{w}_0)^T \quad\right]_{p} \left[\quad \nabla_w \mathbf{y}(\mathbf{w}_0) \quad\right]^{p}_{n}$$

NTK

$$= \begin{bmatrix} \text{—} & \phi(\bar{x}_1)^T & \text{—} \\ & \vdots & \\ \text{—} & \phi(\bar{x}_n)^T & \text{—} \end{bmatrix} \begin{bmatrix} | & & | \\ \phi(\bar{x}_1) & \cdots & \phi(\bar{x}_n) \\ | & & | \end{bmatrix}$$

Full Batch Gradient Descent

- In this regime we will be studying full batch gradient descent, i.e., gradient descent on the whole dataset
- The expression for this can be simplified

$$L(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(f(\bar{x}_i, \boldsymbol{w}) - \bar{y}_i)^2$$

MSE

$$L(\boldsymbol{w}) = \frac{1}{N}\cdot\frac{1}{2}\|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2$$

$$L(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2$$

Taylor Expansion of weights around initialization

if you want to look at gradient of loss

first-order Taylor expansion

$$f(x, \boldsymbol{w}) \approx f(x, \boldsymbol{w_0}) + \nabla_w f(x, \boldsymbol{w_0})^T (\boldsymbol{w} - \boldsymbol{w_0})$$

linear approx

↑ initial weight

n = # datapoints

p = # parameters

$$\nabla_w \mathbf{y}(\mathbf{w_0}) \sim p \times n$$

$$\nabla_w \mathbf{y}(\mathbf{w_0})^T \sim n \times p$$

$$\underset{n \times 1}{\begin{bmatrix} \mathbf{y}(\mathbf{w}) \end{bmatrix}} \approx \begin{bmatrix} \mathbf{y}(\mathbf{w_0}) \end{bmatrix} + \underset{n \times p}{\begin{bmatrix} \nabla_w \mathbf{y}(\mathbf{w_0})^T \end{bmatrix}} \underset{p \times 1}{\left(\begin{bmatrix} \mathbf{w} \end{bmatrix} - \begin{bmatrix} \mathbf{w_0} \end{bmatrix} \right)}$$

$$y(\boldsymbol{w}) \approx y(\boldsymbol{w_0}) + \nabla_w y(\boldsymbol{w_0})^T (\boldsymbol{w} - \boldsymbol{w_0})$$

The lazy regime

- This model is only a good approximation in the **lazy regime** where the Jacobian does not change much during training

(gradient)

$$w_1 = w_0 - \eta \nabla_w L(w_0)$$

$$\text{net change in } y(w) \lesssim \|y(w_0) - \bar{y}\|$$

$$d = \text{distance } d \text{ moved in } w \text{ space} \approx \frac{\text{net change in } y(w)}{\text{rate of change of } y \text{ w.r.t } w} = \frac{\|y(w_0) - \bar{y}\|}{\|\nabla_w y(w_0)\|}$$

$$\text{change in model Jacobian} \approx \text{distance } d \times \text{rate of change of the Jacobian} = d \cdot \|\nabla_w^2 y(w_0)\| = d \cdot \|H\|$$

Hessian

Jacobian is a matrix, first-order derivative of a vector-value func with several variables

$$J = \nabla f = \nabla f(x_1, x_2, \cdots, x_n) \quad x_i \in R^p = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix}$$

$\in R^{m \times n}$

Negligible Change in Jacobian

$$\text{relative change in model Jacobian} \approx \frac{d \cdot \text{rate of change of Jacobian}}{\text{norm of Jacobian}} = \frac{d \cdot \|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|} = \|(\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}})\| \frac{\|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|^2} \ll 1$$

Hessian $\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})$ referred to as $\boldsymbol{\kappa(w_0)}$

$\boldsymbol{\kappa(w_0)} \ll 1$ Hessian has low norm

Hessian matrix H of function $f: R^p \to R$ is second derivatives of f

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & & \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix} \in R^{n \times n} \qquad H_{[i,j]} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

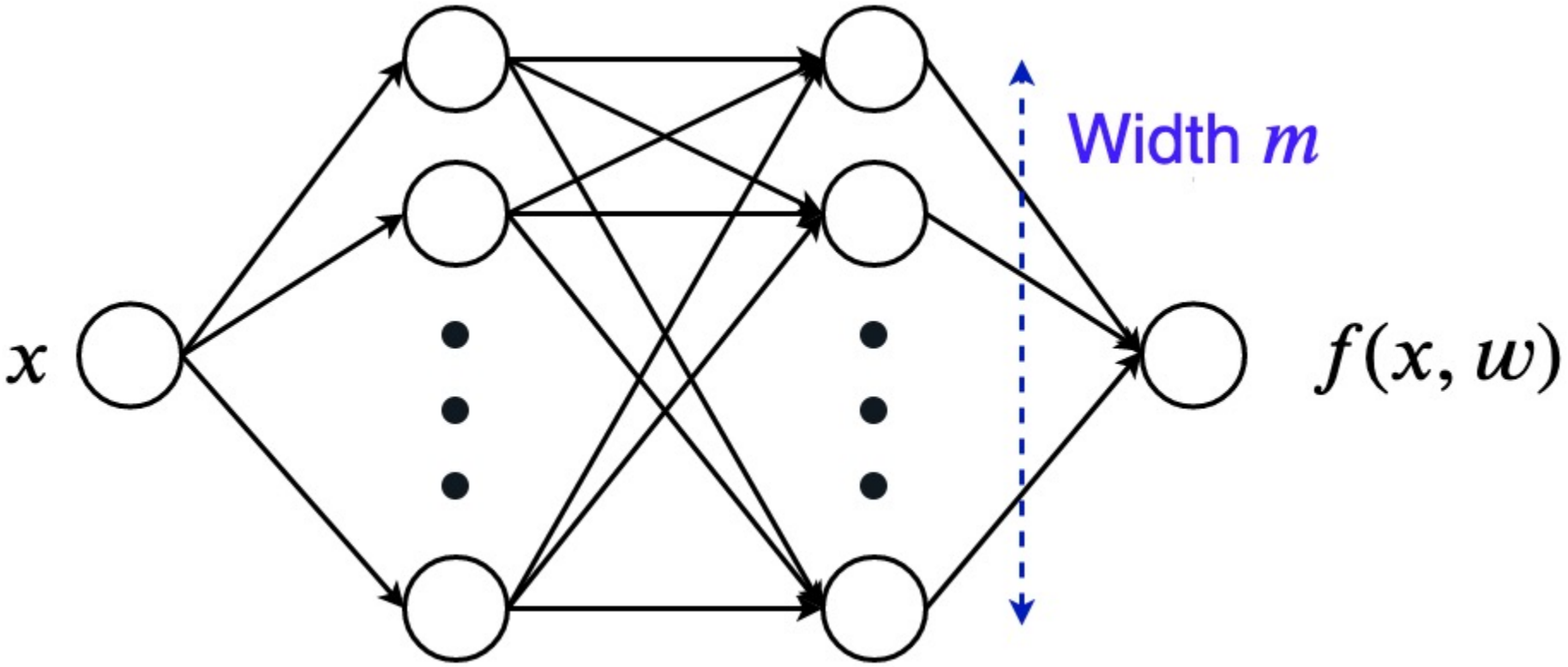
Means that the change in w required to produce a change in the function causes a Negligible change in the Jacobian

Hessian matrix of f is Jacobian matrix of gradient of f

$H = J(\nabla f)$

Thus the model is very close **to its linear approximation**

Width of Network



Infinite width limit

- The initial NTK of the network is a deterministic kernel as the width increases
- In the infinite limit the kernel stays constant as we optimize the weights
- Thus, the network is computing a kernel ridge regression
- SGD is actually doing kernel gradient descent which can be proven to have an optimal minima

Relationship to network width

$\kappa \to 0$ as the width $m \to \infty$.

If width of NN is large, weight changes less which is counter-intuition

- A large width for a neural network means that there are a lot of neurons affecting the output
- Why does this mean that the change in the Jacobian is small??
- A small change in all of these neuron weights can result in a very large change in he output, so the neurons need to move very little to fit the data.
- If the weights move less, the linear approximation is more accurate.

Relationship to scale

- Lets try rescaling the output y by a factor a

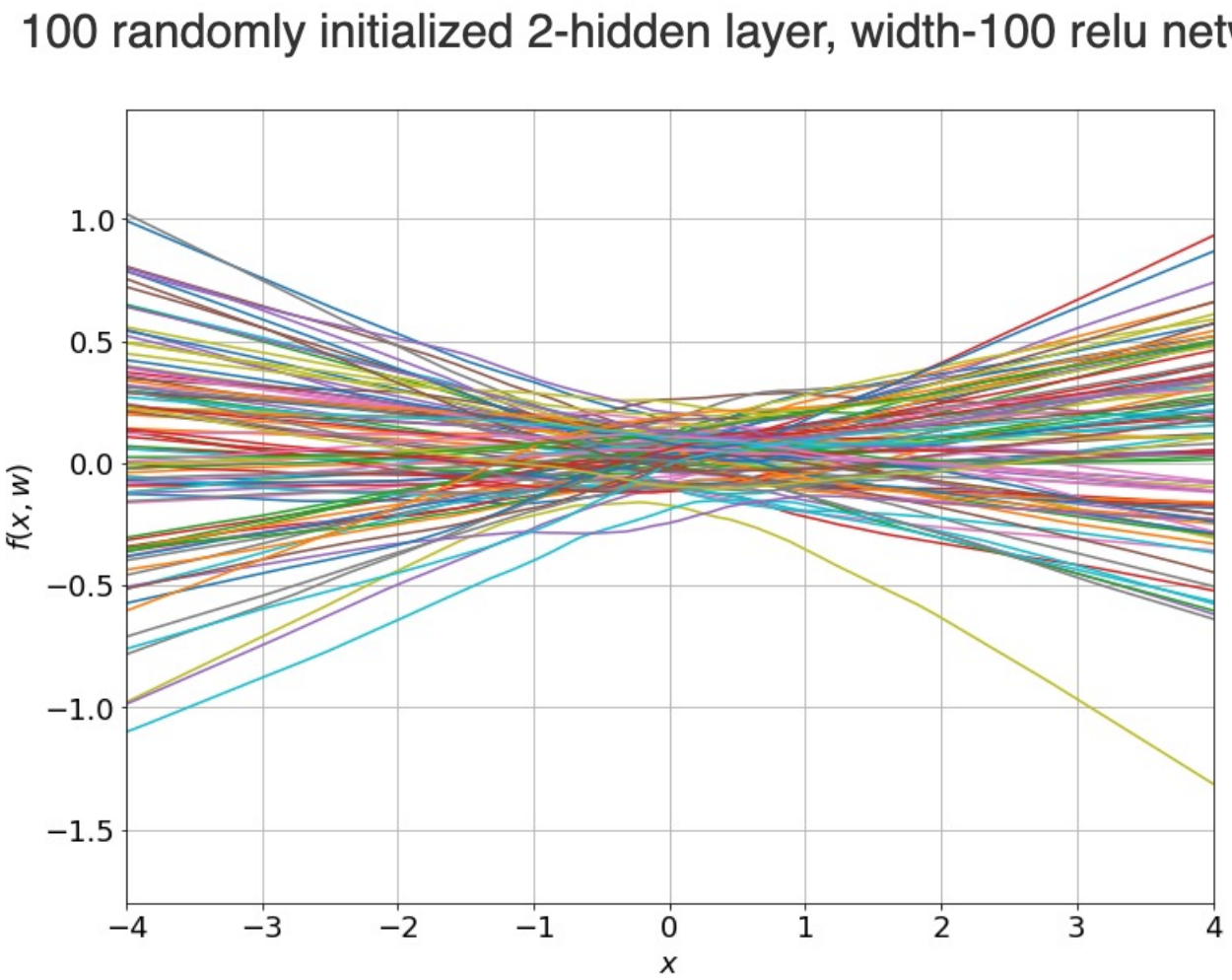
$$\kappa_\alpha(w_0) = \|(\alpha y(w_0) - \bar{y})\| \frac{\|\nabla_w^2 \alpha y(w_0)\|}{\|\nabla_w \alpha y(w_0)\|^2}$$

- Assume $y(w_0) = 0$

$$\implies \kappa_\alpha(w_0) \sim \frac{\|\nabla_w^2 \alpha y(w_0)\|}{\|\nabla_w \alpha y(w_0)\|^2} = \frac{\alpha \|\nabla_w^2 y(w_0)\|}{\alpha^2 \|\nabla_w y(w_0)\|^2} = \frac{1}{\alpha} \frac{\|\nabla_w^2 y(w_0)\|}{\|\nabla_w y(w_0)\|^2}$$

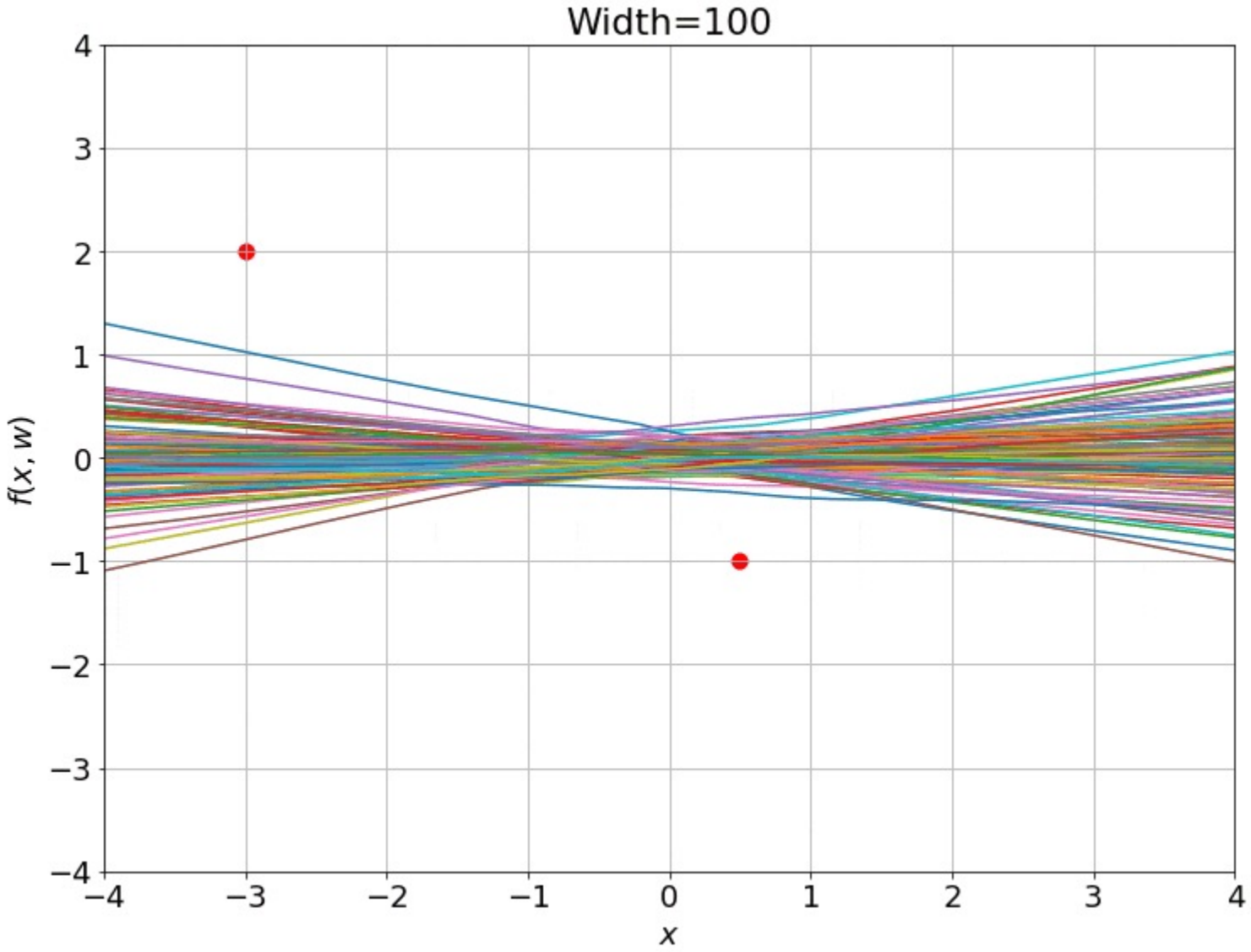
$\kappa(w_0) \to 0$) by simply jacking up $\alpha \to \infty$!

Random Initializations

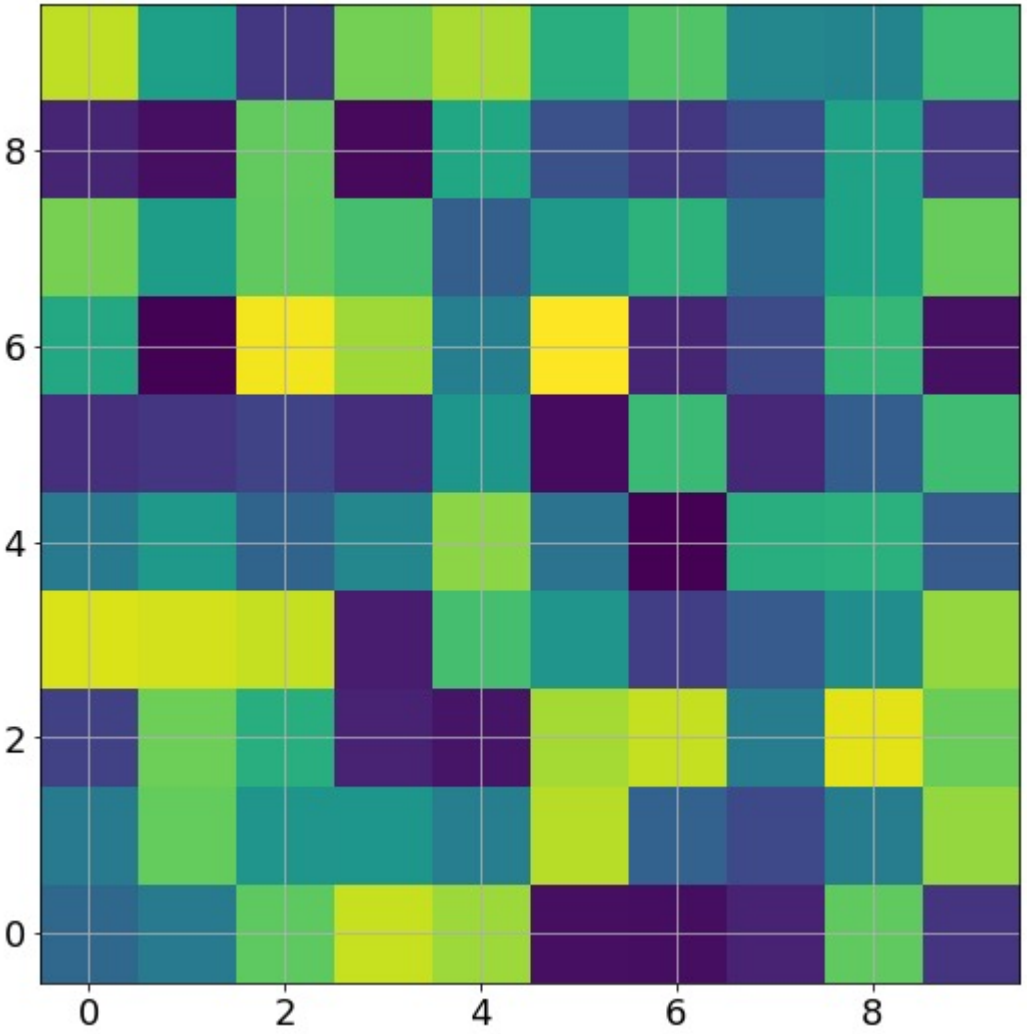


Equivalent to random functions drawn from a Gaussian Process

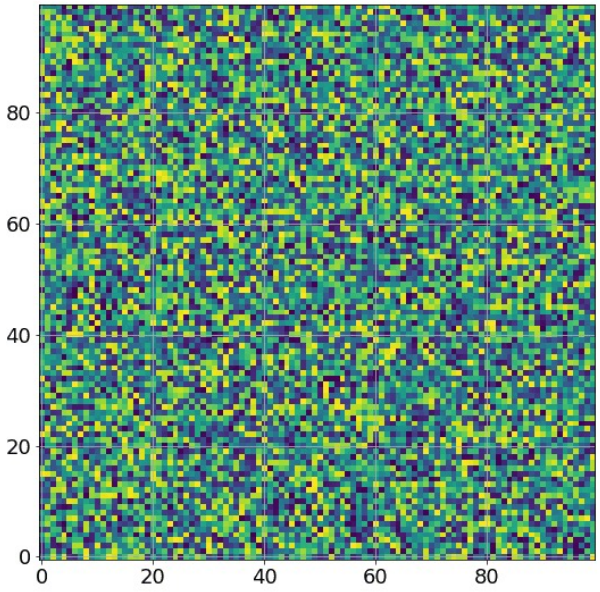
After training with GD

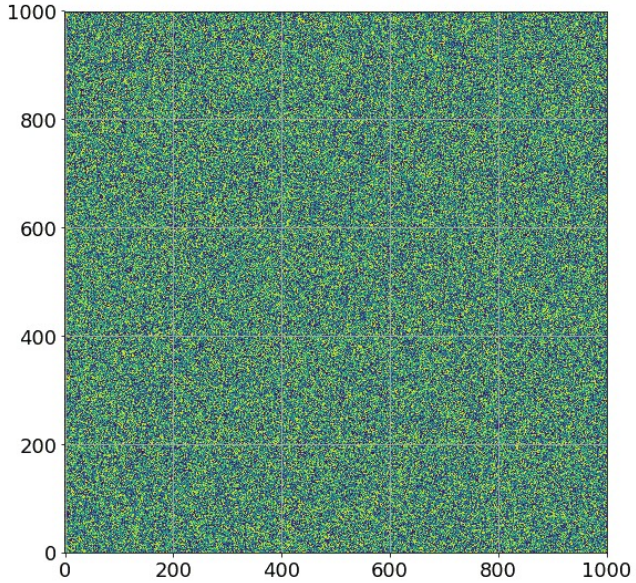


Training of the weights

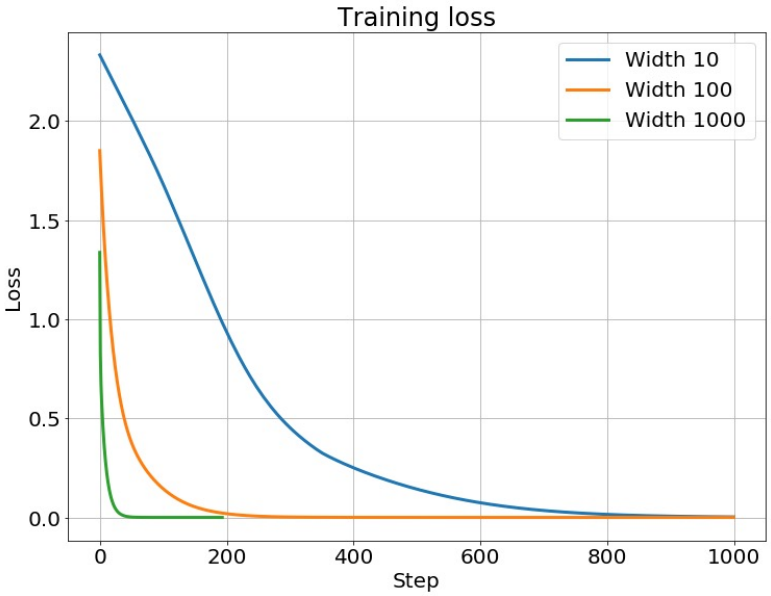


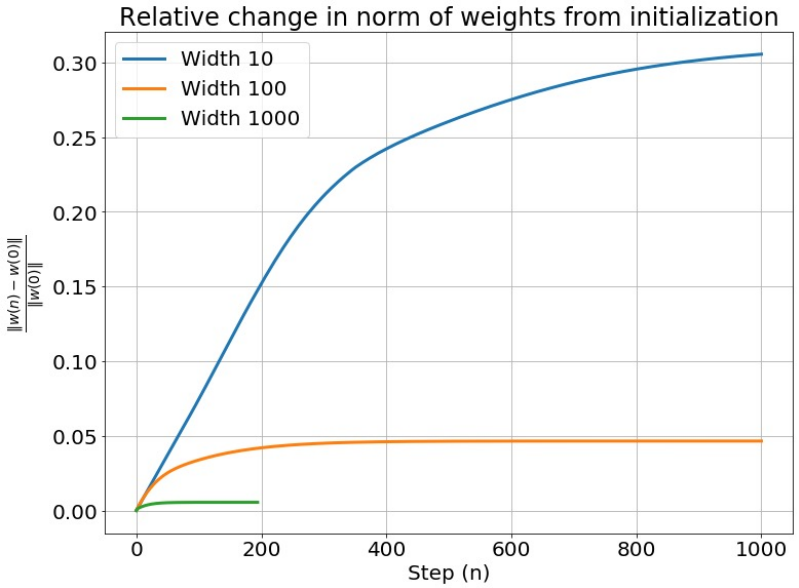
Increasing width, Weights don't Budge!





Relative Change in weights w/training



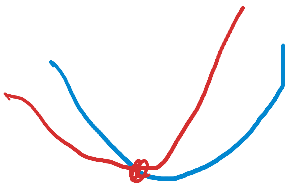


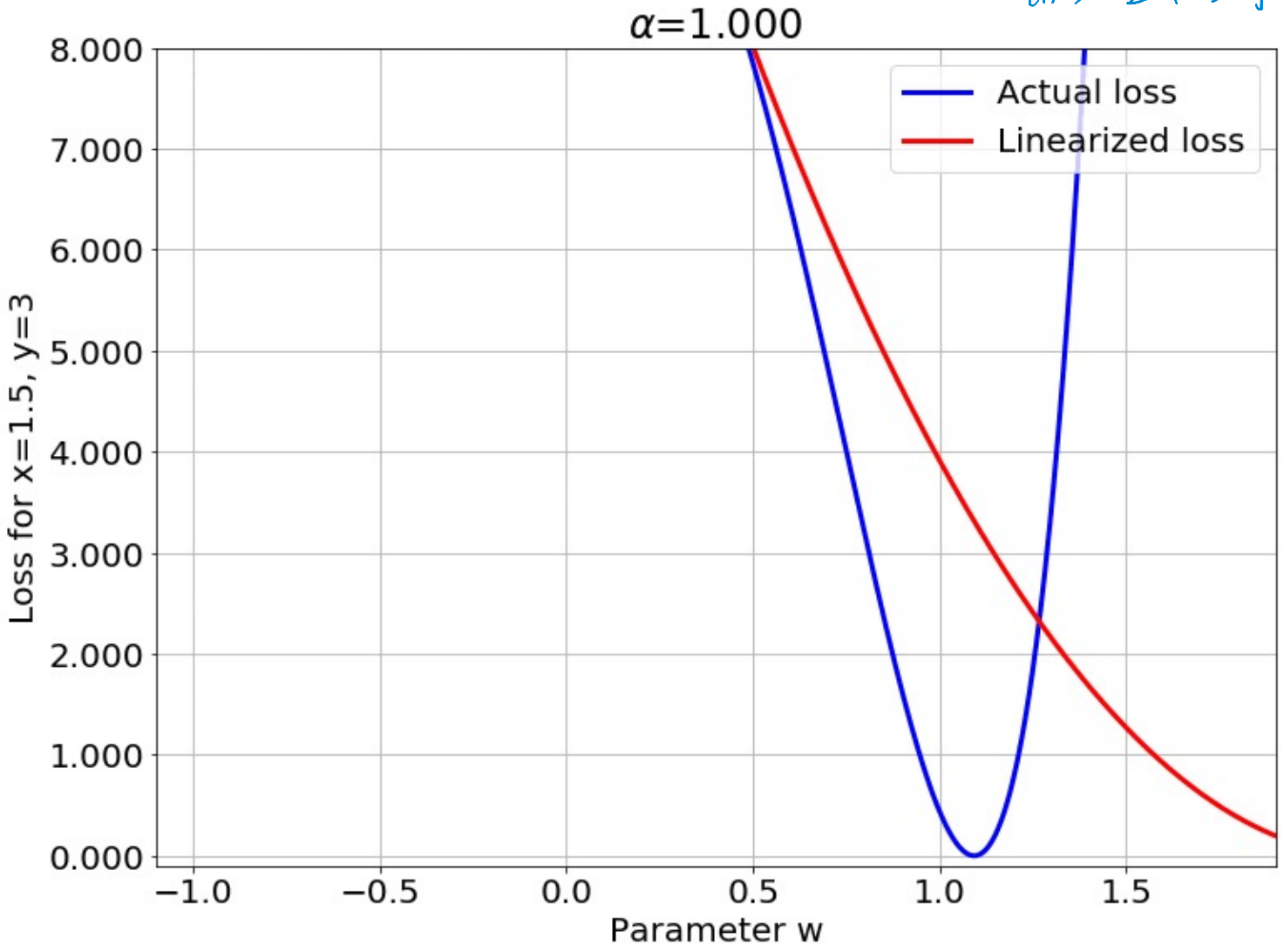
Relative change in norm of weight $= \dfrac{\|\boldsymbol{w}(n) - \boldsymbol{w}_0\|_2}{\|\boldsymbol{w}_0\|_2}$

Actual vs Linearized Loss

approximated by Taylor theorem

these 2 loss funcs coincidide





Continuous evolution of weights

- We can the view evolution of weights as a differential equation and study it continuously in time

$$w_{k+1} = w_k - \eta \nabla_w L(w_k)$$

$$\frac{w_{k+1} - w_k}{\eta} = -\nabla_w L(w_k)$$

- This is a finite difference equation but lets take learning rate to be very small to turn into a diff eq

Simulate behavior of NN without training

$$\frac{dw(t)}{dt} = -\nabla_w L(w(t))$$

Gradient flow

is continuous time differential equation

time is iteration step

- This continuous version of the weight derivative is called Gradient flow

$$\frac{dw(t)}{dt} = -\nabla_w L(w(t))$$

- Think of a continuous version of stochastic gradient descent
- Trajectory of this gradient over time closely follows GD in the **lazy regime**

Simplified Gradient Flow

- Dot notation for derivatives gives

$$\dot{w}(t) = -\nabla L(w(t))$$

- Dropping time variable from equation, and writing out the loss function gives
-

$$\dot{w} = -\nabla y(w)(y(w) - \bar{y})$$

Output Dynamics

- To get model output dynamics apply chain rule on

NTK

$$\dot{y}(w) = \nabla y(w)^T \dot{w} = -\nabla y(w)^T \nabla y(w)(y(w) - \bar{y})$$

- Recall the Neural Tangent Kernel, this is the associated matrix (matrix of inner products)

$$\nabla y(w)^T \nabla y(w)$$

- Give it a new symbol $H(w)$

- Recall this is the matrix induced by the kernel feature map
-

$$\phi(x) = \nabla_w f(x, w_0)$$

Training in the Kernel Regime

- Training changes to simple ODE solving

$$\dot{y}(w) = -H(w_0)(y(w) - \bar{y})$$

- Set $y(w) = \bar{y}$
- This is the solution or *equilibrium* of the ODE

$$\dot{u} = -H(w_0)u$$

$$u(t) = u(0)e^{-H(w_0)t}$$

- If the matrix can be diagonalized then exponent of matrix is just the exponent applied to the eigenvalues

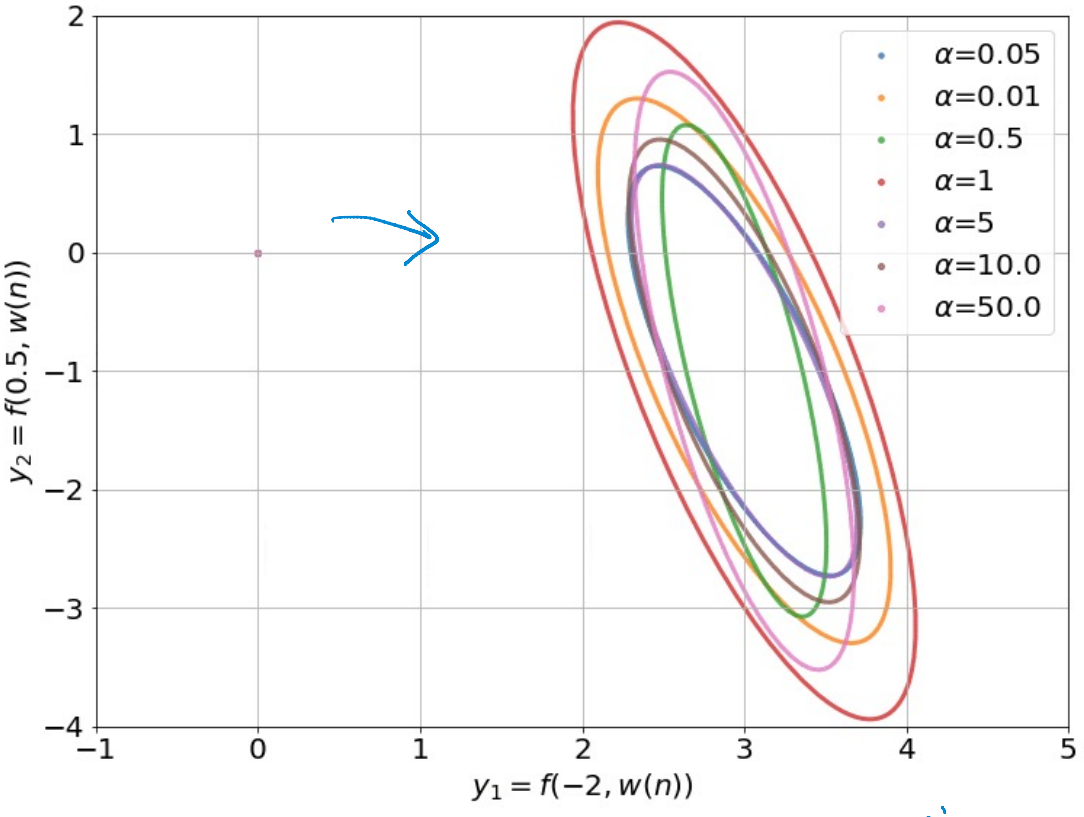
eigendecompose the NTK, eigenvalue will tell you how fast gradient change in that direction e^{λ}

Descent along Eigenvectors of NTK

- $H(w_0)$ positive semi-*definite*
- *Lets assume we have no 0 eigenvalues, then it is* ***positive definite***
- This means that we can eigendecompose $H(w_0)$
- Eigendecomposition decomposes the gradient into n independent directions we have to go
- Steps in these directions causes those eigenvalues to decay
- Eventually gradient will be 0
- This is also our global minimum solution where y(w)-y!

Eigenvector Decay Visualized

This is the case of a 2-point dataset $K \in \mathbb{R}^{2\times 2}$



eigenspace of kernel

Notice this is also happening with small alpha! Why?

Generalization

- Training the model in the kernel regime is equivalent to solving a linear system. (equation)
- Happy that gradient descent does something reasonable in this domain!
- What does this mean for generalization? with a large NN you can always reach global minima
- This system of equations is under-determined as you increase width to infinity
- However it can be shown that in the infinite width limit this converges to the minimium norm solution
- Gradient descent chooses that solution which has minimum $\|\boldsymbol{w}\|2$ as long as you start with a low norm initialization
- Thus this has small complexity in terms of Neyshambur et al.

Are NTKs equivalent to NNs? NO

NN > NTK

- Turns out they are not equivalent
- Neural networks generally perform better than NTKs even at “infinite width limit”
- This means that NTKs are actually missing something
- Recently proposed: enhanced NTKs
- But still provides some means of studying neural networks in an idealized setting good stopping point in training
- Still an active area of research

Further reading

- Jacot et al. **Neural Tangent Kernel: Convergence and Generalization in Neural Networks**, NeurIPS 2018
- Chizat et al. **On Lazy Training in Differentiable Programming** NeurIPS 2019 Lazy regime
- Arora, Sanjeev, et al. **On exact computation with an infinitely wide neural net** NeurIPS 2019
- Li, Zhiyuan, et al. **Enhanced Convolutional Neural Tangent Kernels** NeurIPS 2019
- https://towardsdatascience.com/kernel-function-6f1d2be6091
- https://rajatvd.github.io/NTK/