Neural Networks and Deep Learning

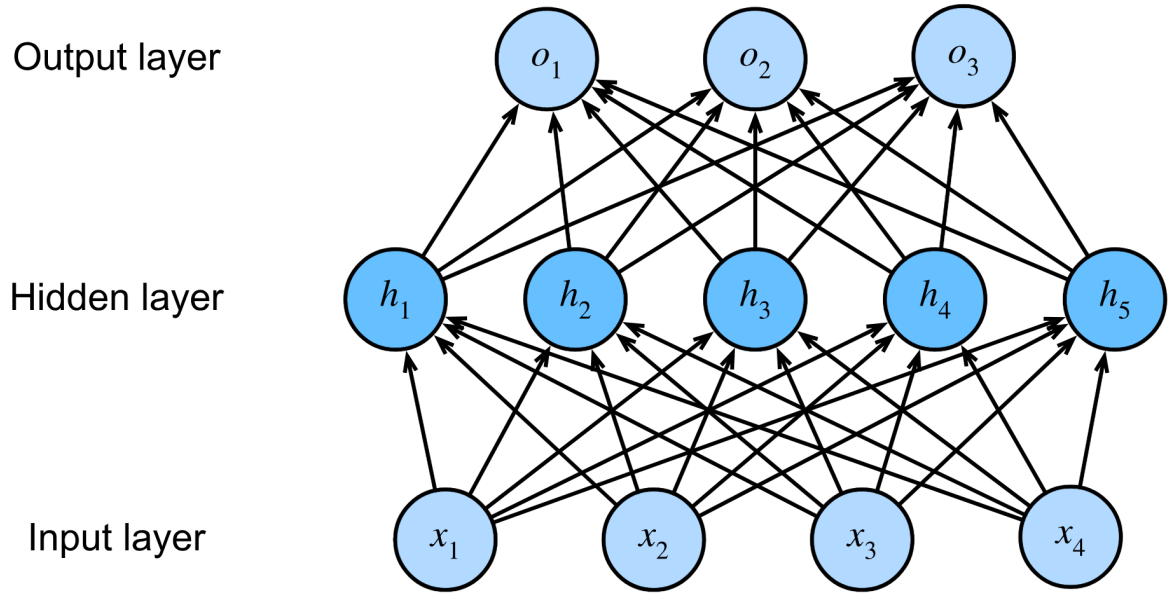
Advanced Statistical Inference

Simone Rossi

Introduction

- Neural networks are a class of parametric models that are widely used in machine learning and statistics.
- Build complex functions by composing simple functions.

$$f = f_L \circ f_{L-1} \circ \ldots \circ f_1$$



Composition of functions

Generally, each function f_i is a linear transformation followed by a non-linear activation function.

For example, in a feedforward MLP:

$$f_i(\boldsymbol{\theta}_i, \boldsymbol{x}) = a(\underbrace{\boldsymbol{W}_i \boldsymbol{x} + \boldsymbol{b}_i}_{\text{linear transformation}})$$

where $\boldsymbol{W}_i$ is a matrix of weights, $\boldsymbol{b}_i$ is a bias vector, and $a(\cdot)$ is the activation function.

From linear models to neural networks

From a probabilistic perspective, we still need to define a likelihood:

- Regression with noisy outputs: Gaussian likelihood

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = \prod_{i=1}^{N} \mathcal{N}(y_i \mid f(\boldsymbol{\theta}, \boldsymbol{x}_i), \sigma^2)$$

- Binary classification: Bernoulli likelihood

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = \prod_{i=1}^{N} \text{Bern}(y_i \mid \sigma(f(\boldsymbol{\theta}, \boldsymbol{x}_i)))$$

- Multi-class classification: Categorical likelihood

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = \prod_{i=1}^{N} \text{Cat}(y_i \mid \text{sfmx}(f(\boldsymbol{\theta}, \boldsymbol{x}_i)))$$

Loss vs Likelihood

Classic loss functions can be interpreted as negative log-likelihoods.

Task	Loss Function	Likelihood Function
Regression	Mean Squared Error	Gaussian Likelihood with $\sigma^2 = 1$
Binary Classification	Binary Cross-Entropy	Bernoulli Likelihood
Multi-class Classification	NLL with Softmax	Categorical Likelihood

Training neural networks

- Training neural networks involves optimizing the parameters $\boldsymbol{\theta}$ to maximize the likelihood of the data.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{y} \mid \boldsymbol{\theta})$$

- This is typically done by using the **backpropagation** algorithm
- The backpropagation algorithm computes the gradient with respect to the parameters $\boldsymbol{\theta}$ using the chain rule

$$\frac{\partial \log p(\boldsymbol{y} \mid \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_i} = \frac{\partial \log p(\boldsymbol{y} \mid \boldsymbol{\theta})}{\partial f_L} \frac{\partial f_L}{\partial f_{L-1}} \cdots \frac{\partial f_{i+1}}{\partial \boldsymbol{\theta}_i}$$

Training neural networks

The gradient is used to update the parameters using an optimization algorithm (e.g., SGD, Adam)

For SGD, the update rule is:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\boldsymbol{y} \mid \boldsymbol{\theta})$$

where η is the learning rate.

More complex optimization algorithms involve momentum, adaptive learning rates, etc.

Landscape of neural networks

Training neural networks

- Optimization of neural networks is a non-convex optimization problem, with many local minima.
- Overfitting is a common problem in neural networks, especially when the number of parameters is large.
- Common techniques to prevent overfitting include:
  - Weight decay
  - Dropout

Weight decay is a Gaussian prior

Weight decay is a regularization technique that changes the update rule to:

$$\theta \leftarrow \theta + \eta \nabla_\theta \log p(\boldsymbol{y} \mid \theta) - \lambda \theta$$

where λ is the weight decay parameter.

$$\eta \nabla_\theta \log p(\boldsymbol{y} \mid \theta) + \lambda \theta = \eta \left(\nabla_\theta \log p(\boldsymbol{y} \mid \theta) - \frac{\lambda}{\eta} \theta \right)$$
$$\eta \left(\nabla_\theta \log p(\boldsymbol{y} \mid \theta) + \nabla_\theta \left(-\frac{\lambda}{\eta} \theta^T \theta \right) \right)$$

but $\log \mathcal{N}(\theta \mid \mathbf{0}, \frac{\eta}{2\lambda} \boldsymbol{I}) \propto -\frac{\lambda}{\eta} \theta^T \theta$

> Important note
>
> Weight decay is equivalent to placing a Gaussian prior on the weights, but the variance of the prior also depends on the learning rate.

Gaussian prior on the weights

- The Gaussian prior on the weights can be interpreted as a form of **Bayesian regularization**.
- When we have a weight decay term, the optimization problem is equivalent to maximizing the posterior distribution of the weights.

$$\theta^* = \arg\max_\theta p(\theta \mid \boldsymbol{y}) = \arg\max_\theta p(\boldsymbol{y} \mid \theta) p(\theta) = \arg\max_\theta \left[\log p(\boldsymbol{y} \mid \theta) + \log p(\theta) \right]$$

Dropout

- Dropout is a technique that randomly sets a fraction of the activations to zero during training.

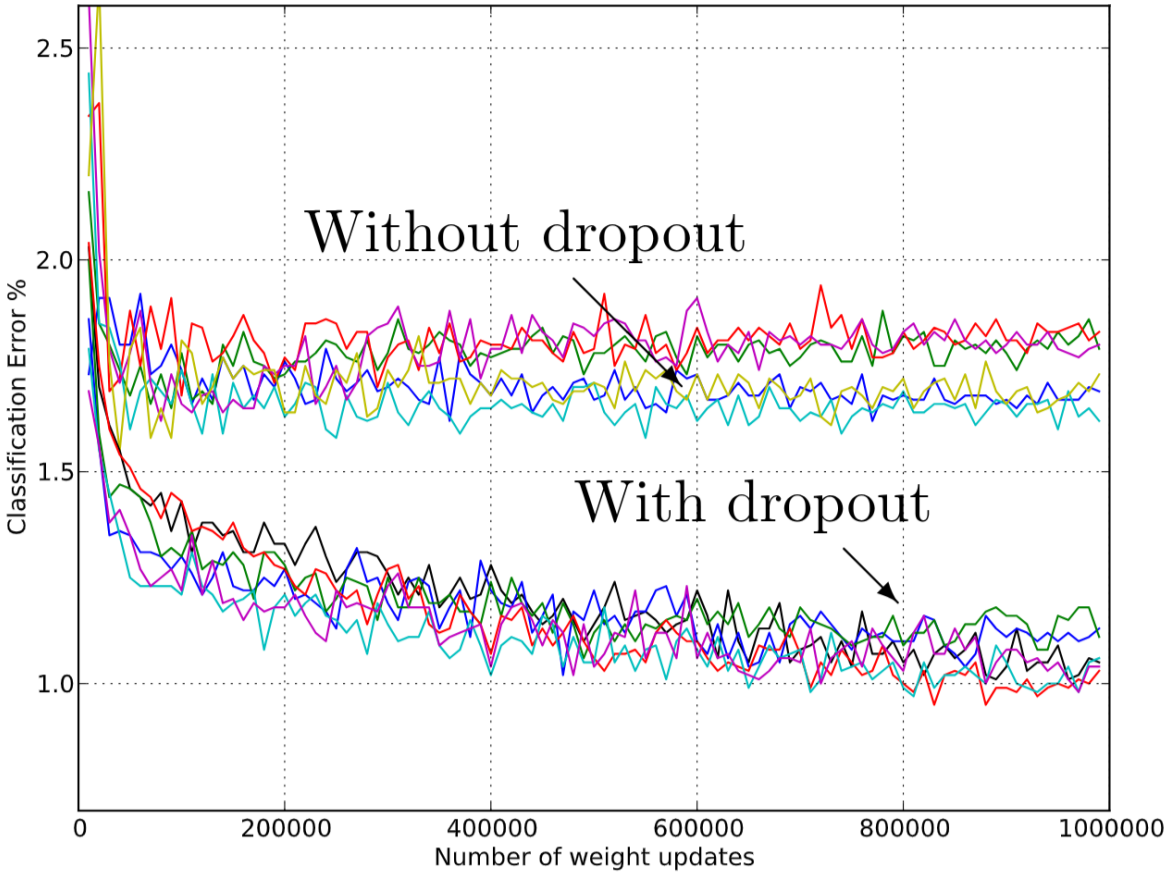
Dropout formulation

With dropout, during training the output of a layer is given by:

$$\begin{aligned} \boldsymbol{z} &\sim \text{Bern}(p) \\ \widetilde{\boldsymbol{x}} &= \boldsymbol{z} \odot \boldsymbol{x} \\ \boldsymbol{h} &= a(\boldsymbol{W}\widetilde{\boldsymbol{x}} + \boldsymbol{b}) \end{aligned}$$

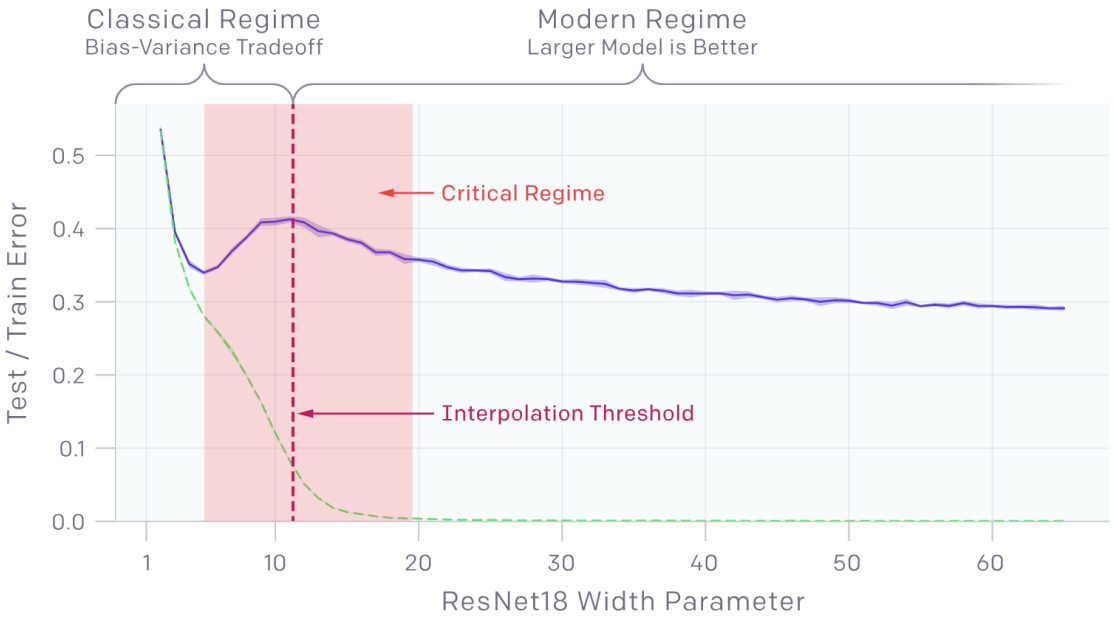
where $\boldsymbol{r}$ is a binary mask, p is the dropout probability, and $\odot$ is the element-wise product. **At test time**, the activations are scaled by p to account for the fact that more units are active during training.

Note: Dropping out activations can be interpreted as dropping out rows of the weight matrix $\boldsymbol{W}$.



Double descent phenomenon

- Neural networks are highly overparametrized models, and they can fit the training data perfectly.
- However, they can also generalize well to unseen data.
- But the behavior of the generalization error is more complex: this is called the **double descent phenomenon**.



Bayesian Neural Networks

Bayesian Neural Networks

Bayesian neural networks are neural networks that are trained using a Bayesian approach.

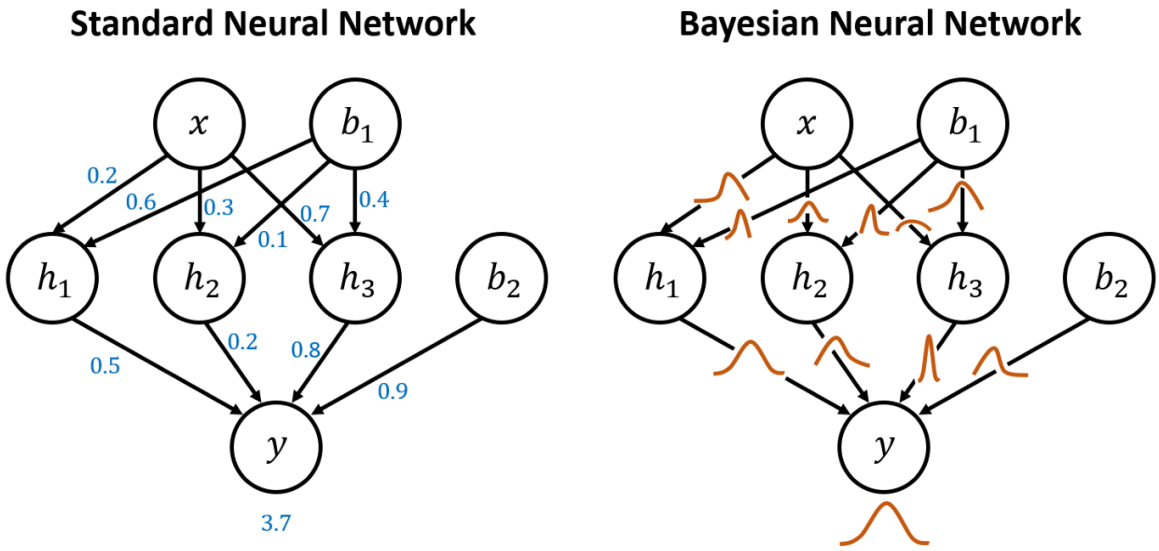
It follows the classic recipe of Bayesian inference:

1. Define a prior distribution over the parameters $\boldsymbol{\theta}$ (e.g., Gaussian on weights and biases).
2. Define a likelihood function $p(\boldsymbol{y} \mid \boldsymbol{\theta})$.
3. Compute the posterior distribution $p(\boldsymbol{\theta} \mid \boldsymbol{y})$.

. . .

Problem: The posterior distribution is intractable for neural networks, we need to approximate it.

Bayesian Neural Networks



Approximate inference in Bayesian neural networks

We have seens several methods to approximate intractable posterior distributions:

- Variational inference
- MCMC
- Laplace approximation

Everything we have seen so far can be applied to Bayesian neural networks.

. . .

But neural networks have particular properties that make them challenging to approximate:

- High-dimensional parameter space
- Non-convex optimization landscape => non-Gaussian posteriors
- Many symmetries => many local minima => multimodal posteriors
- Computationally expensive likelihood evaluations
- Priors are difficult to specify

Approximate inference in Bayesian neural networks

Let's focus on three different problems:

1. Specifying good priors for Bayesian neural networks

2. Reusing some techniques from neural networks to approximate the posterior (e.g., dropout, ensembles)
3. Using Gaussian processes to approximate the posterior

Specifying priors for Bayesian neural networks

- The choice of prior is crucial in Bayesian inference, as it encodes our prior beliefs about the parameters.
- For neural networks, the choice of prior is not straightforward, as the parameters are high-dimensional and they are difficult to interpret.

Specifying priors for Bayesian neural networks

- **Gaussian prior**: the most common choice, can be different for weights and biases. It has a close connection with weight decay.

$$p(\boldsymbol{\theta}) = \prod_{i=1}^{L} \mathcal{N}(\boldsymbol{\theta}_i \mid 0, \sigma^2)$$

- **Laplace prior**: can be used to promote sparsity in the weights.
- **Scale mixture prior**: can be used to promote heavy-tailed distributions.

$$p(\boldsymbol{\theta}) = \prod_{i=1}^{L} \mathcal{N}(\boldsymbol{\theta}_i \mid 0, \sigma^2) \quad \text{with} \quad \sigma^2 \sim p(\sigma^2)$$

where $p(\sigma^2)$ can be a Gamma, Inverse-Gamma, Exponential, or other distributions.

Dropout as a Bayesian approximation

Idea: Instead of applying dropout only during training, we can sample from the dropout mask at test time many times and use the ensemble of predictions.

This is called **Monte Carlo dropout**.

Dropout as a Bayesian approximation

Dropout can be interpreted as a form of variational inference.

$$\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{q(\theta)}[\log p(\boldsymbol{y} \mid \boldsymbol{\theta})] - \text{KL}\left(q(\boldsymbol{\theta}) \| p(\boldsymbol{\theta})\right)$$

where $q(\boldsymbol{\theta})$ is the variational distribution, and $p(\boldsymbol{\theta})$ is the prior.

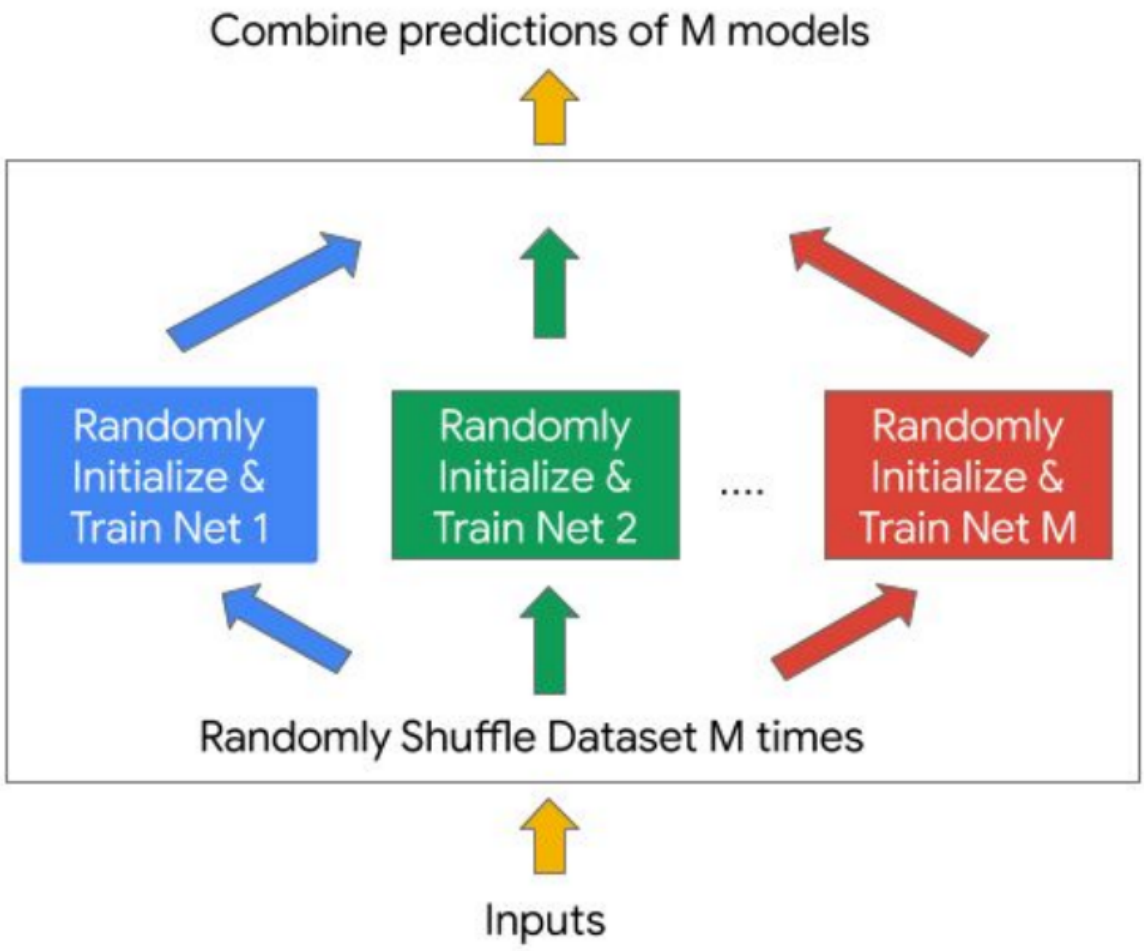
For MC dropout with dropout probability p, the variational distribution is:

$$q(\boldsymbol{\theta}) = (1-p)\mathcal{N}(\boldsymbol{m}, \boldsymbol{\sigma}^2) + p\mathcal{N}(\mathbf{0}, \boldsymbol{\sigma}^2)$$

Note: The KL divergence is not tractable analytically, but it can be approximated using the Monte Carlo method.

Deep ensembles

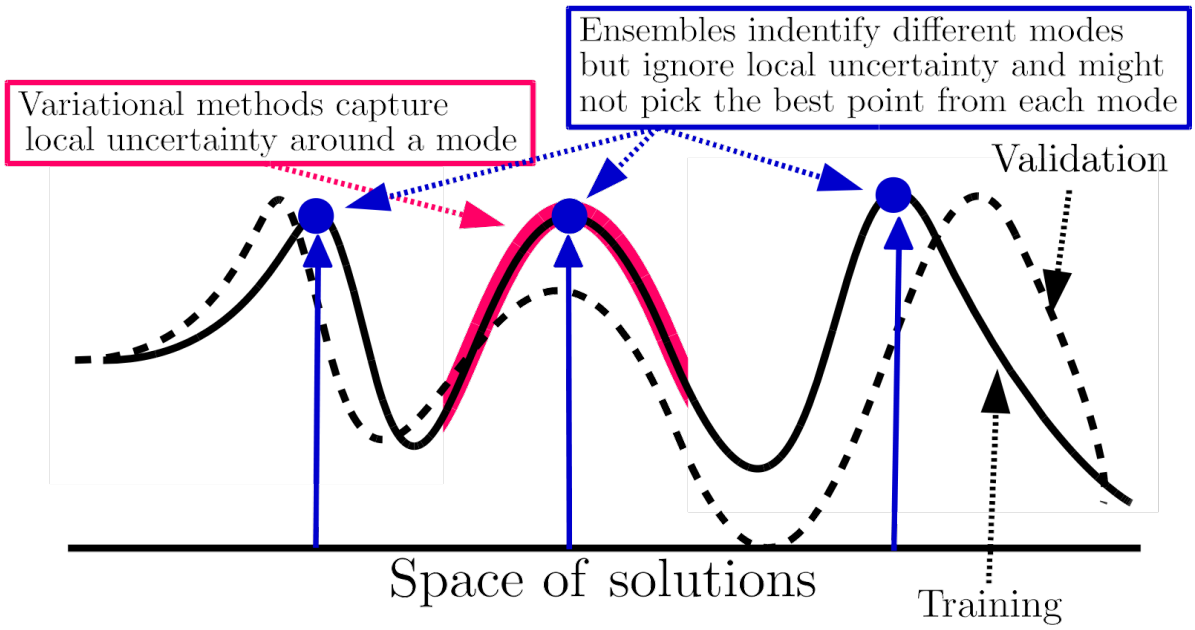
Idea: Train multiple neural networks *independently* with different initializations and average their predictions.

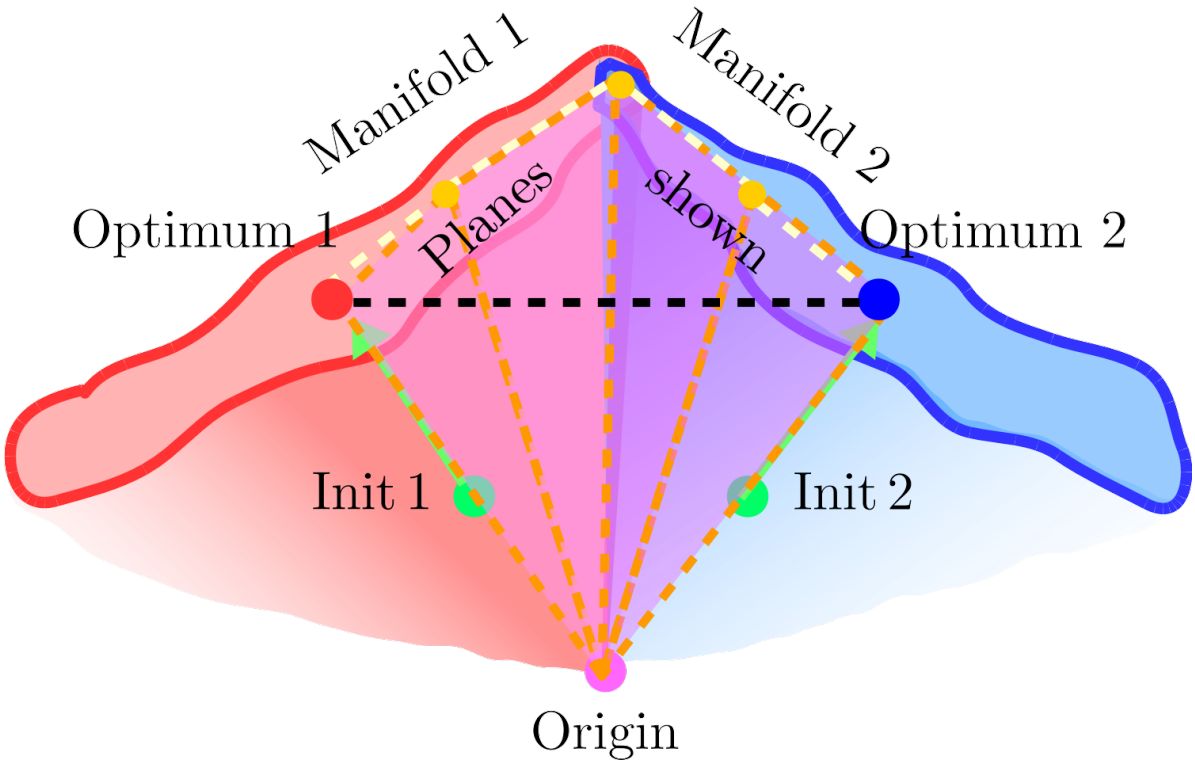


- **Advantages**:
  - Deep ensemble methods are easy to implement and training is highly parallelizable.
  - They can be used with any neural network architecture, loss function, and optimization algorithm
  - They can be used to estimate the uncertainty of the predictions.

Why do deep ensembles work? An intuitive explanation

- **Diversity**: each neural network is trained with a different initialization, so they will converge to different local minima, separating the modes of the posterior distribution.


Ensembles indentify different modes
but ignore local uncertainty and might
not pick the best point from each mode
Variational methods capture
local uncertainty around a mode
Validation
Space of solutions
Training


Manifold 1
Manifold 2
Optimum 1
Planes
shown
Optimum 2
Init 1
Init 2
Origin

Connecting the modes of the posterior distribution

Why do deep ensembles work?

Under some assumptions, deep ensembles can be interpreted as a form of approximate Bayesian inference.

Theoretical connections between neural networks and kernel methods

Behaviour of wide neural networks

Some interesting insights from Radford M. Neal's 1993 thesis:

- Gaussian processes are a powerful tool for machine learning
- Neural networks are universal function approximators and they can generalize well despite overparametrization
- The infinite-width limit of neural networks is a Gaussian process

Infinite-width neural networks

- Consider a single hidden layer neural network:

$$\boldsymbol{f}(\boldsymbol{\theta}, \boldsymbol{x}) = \boldsymbol{w}^\top a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

  where $\boldsymbol{W} \in \mathbb{R}^{M \times D}$, $\boldsymbol{w} \in \mathbb{R}^M$, $\boldsymbol{b} \in \mathbb{R}^M$ and $a(\cdot)$ is the activation function.

- In the limit of infinite width ($M \to \infty$), randomly initialized neural networks becomes a Gaussian process, regardless of the choice of activation function.

Infinite-width neural networks

Remember the definition of a Gaussian process:

- A Gaussian process is a collection of random variables, with some index set, such that any finite subset of the random variables has a joint Gaussian distribution.

$$f(\boldsymbol{x}_1), f(\boldsymbol{x}_2), \ldots, f(\boldsymbol{x}_N) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{K})$$

where $\boldsymbol{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is the covariance matrix computed using the kernel function $k(\cdot, \cdot)$.

Infinite-width neural networks

Depending on the choice of activation function, the infinite-width neural network can be interpreted as a Gaussian process with a specific kernel.

Activation function	Finite-limit Kernel
ReLU	Arc-cosine kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \frac{\|\boldsymbol{x}\|\|\boldsymbol{x}'\|}{2\pi}(\sin\theta + (\pi - \theta)\cos\theta)$ with $\theta = \cos^{-1}\left(\frac{\boldsymbol{x}^\top \boldsymbol{x}'}{\|\boldsymbol{x}\|\|\boldsymbol{x}'\|}\right)$
Trigonometric (sin/cos)	Squared exponential or RBF kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2\ell^2}\right)$

Infinite-width neural networks

- So far, we have seen that one-hidden layer neural networks can be interpreted as Gaussian processes in the infinite-width limit.

. . .

- But during the years, many other results have been obtained for more complex architectures:
  - G. de G. Matthews, et al. "Gaussian Process Behaviour in Wide Deep Neural Networks". ICLR 2018.
  - J. Lee, et al. "Deep Neural Networks as Gaussian Processes". ICLR 2018.
  - A. Novak, et al. "Bayesian Deep Convolutional Networks with Many Channels are Gaussian Processes". NeurIPS 2018.
  - A. Garriga-Alonso, et al. "Deep convolutional networks as shallow Gaussian processes". ICLR 2019.
  - G. Yang, et al. "Wide Feedforward or Recurrent Neural Networks of Any Architecture are Gaussian Processes". NeurIPS 2019.

Infinite-width neural networks

- The main idea is that as the width of the network increases, the output of randomly initialized neural networks $f_0(\boldsymbol{x})$ converges to a Gaussian process.

$$f_0(\boldsymbol{x}) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{K}) \quad \text{with} \quad \boldsymbol{K} = \lim_{M \to \infty} \mathbb{E}[\boldsymbol{f}_0(\boldsymbol{x}) \boldsymbol{f}_0(\boldsymbol{x})^\top]$$

Infinite-width neural networks

Controversial opinions on infinite-width neural networks:

- **Pros**:
  - Infinite-width limits be used to derive theoretical results for deep learning.
  - They can be used to design better optimization algorithms.
  - They can be used to design better architectures.

. . .

- **Cons**:
  - Feature-learning is one of the main advantages of neural networks, but in the infinite-width limit there are no features to learn.