Deep learning introduction

Brooks Paige

COMP0171 Week 6

Learning that isn't deep

$$f(\mathbf{x}) = \mathbf{w}^\top \Phi(\mathbf{x})$$

Learning that isn't deep

$$f(\mathbf{x}) = \mathbf{w}^\top \Phi(\mathbf{x})$$

- Supervised learning setting
- Data: $\mathcal{D} = \{\mathbf{x}_i, y_i\}$, $i = 1, \ldots, N$
- Φ: fixed feature map
- $\mathbf{w}$: model parameters

What is deep learning?

$$f(\mathbf{x}) = \mathbf{w}^\top \Phi(\mathbf{x})$$

Learning in a linear model: estimate $\mathbf{w}$.

What is deep learning?

$$f(\mathbf{x}) = \mathbf{w}^\top \Phi(\mathbf{x})$$

Learning in a linear model: estimate $\mathbf{w}$. (**Shallow learning**!)

What is deep learning?

$$f(\mathbf{x}) = \mathbf{w}^\top \Phi(\mathbf{x})$$

Learning in a linear model: estimate $\mathbf{w}$. (**Shallow learning**!)

Deep learning: **also estimate** Φ!

The simplest “deep” model

$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}\mathbf{x} + \mathbf{b})$$

The simplest “deep” model

$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}\mathbf{x} + \mathbf{b})$$

- The scalar function $g(\cdot)$ is a **nonlinearity**, e.g.

$$g(z) = \max(z, 0), \qquad \textit{(ReLU)}$$

$$g(z) = \log(1 + \exp(z)), \qquad \textit{(Softplus)}$$

$$g(z) = (1 + \exp(-z))^{-1} \qquad \textit{(sigmoid)}$$

The simplest "deep" model

$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}\mathbf{x} + \mathbf{b})$$

- The scalar function $g(\cdot)$ is a **nonlinearity**, e.g.

$$g(z) = \max(z, 0), \qquad \textit{(ReLU)}$$
$$g(z) = \log(1 + \exp(z)), \qquad \textit{(Softplus)}$$
$$g(z) = (1 + \exp(-z))^{-1} \qquad \textit{(sigmoid)}$$

- $\mathbf{h} = g(\mathbf{A}\mathbf{x} + \mathbf{b})$ is called a **hidden layer**, with *weights* $\mathbf{A}$ and *biases* $\mathbf{b}$

The simplest "deep" model

$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}\mathbf{x} + \mathbf{b})$$

- The scalar function $g(\cdot)$ is a **nonlinearity**, e.g.

$$\begin{aligned} g(z) &= \max(z, 0), && \textit{(ReLU)} \\ g(z) &= \log(1 + \exp(z)), && \textit{(Softplus)} \\ g(z) &= (1 + \exp(-z))^{-1} && \textit{(sigmoid)} \end{aligned}$$

- $\mathbf{h} = g(\mathbf{A}\mathbf{x} + \mathbf{b})$ is called a **hidden layer**, with *weights* $\mathbf{A}$ and *biases* $\mathbf{b}$
- With $\mathbf{x} \in \mathbb{R}^D$, the hidden layer can be any $\mathbf{h} \in \mathbb{R}^H$, even with $H \gg D$

The simplest "deep" model

$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}\mathbf{x} + \mathbf{b})$$

- The scalar function $g(\cdot)$ is a **nonlinearity**, e.g.

$$\begin{aligned} g(z) &= \max(z, 0), && \textit{(ReLU)} \\ g(z) &= \log(1 + \exp(z)), && \textit{(Softplus)} \\ g(z) &= (1 + \exp(-z))^{-1} && \textit{(sigmoid)} \end{aligned}$$

- $\mathbf{h} = g(\mathbf{A}\mathbf{x} + \mathbf{b})$ is called a **hidden layer**, with *weights* $\mathbf{A}$ and *biases* $\mathbf{b}$
- With $\mathbf{x} \in \mathbb{R}^D$, the hidden layer can be any $\mathbf{h} \in \mathbb{R}^H$, even with $H \gg D$
- Notation: generically, we will refer to networks as $f_\theta(\mathbf{x})$, where θ is all the parameters in the model (in this case, $\theta = \{\mathbf{A}, \mathbf{b}, \mathbf{w}\}$).

Compositionality

Repeat as desired:

$$
\begin{aligned}
y &= \mathbf{w}^\top \mathbf{h}_1 \\
\mathbf{h}_1 &= g(\mathbf{A}_1 \mathbf{x} + \mathbf{b}_1)
\end{aligned}
$$

.

Compositionality

Repeat as desired:

$$
\begin{aligned}
y &= \mathbf{w}^\top \mathbf{h}_1 \\
\mathbf{h}_1 &= g(\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1) \\
\mathbf{h}_2 &= g(\mathbf{A}_2 \mathbf{x} + \mathbf{b}_2)
\end{aligned}
$$

.

Compositionality

Repeat as desired:

$$\begin{aligned} y &= \mathbf{w}^\top \mathbf{h}_1 \\ \mathbf{h}_1 &= g(\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1) \\ \mathbf{h}_2 &= g(\mathbf{A}_2 \mathbf{h}_3 + \mathbf{b}_2) \\ &\vdots \\ \mathbf{h}_L &= g(\mathbf{A}_L \mathbf{x} + \mathbf{b}_L). \end{aligned}$$

Compositionality

Repeat as desired:

$$\begin{aligned}
y &= \mathbf{w}^\top \mathbf{h}_1 \\
\mathbf{h}_1 &= g(\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1) \\
\mathbf{h}_2 &= g(\mathbf{A}_2 \mathbf{h}_3 + \mathbf{b}_2) \\
&\vdots \\
\mathbf{h}_L &= g(\mathbf{A}_L \mathbf{x} + \mathbf{b}_L).
\end{aligned}$$

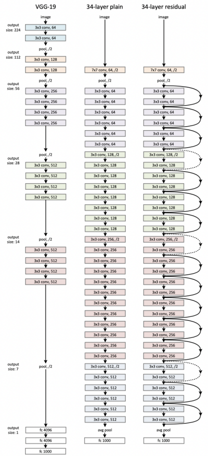
This model is called a **multi-layer perceptron** or **feed-forward network**.

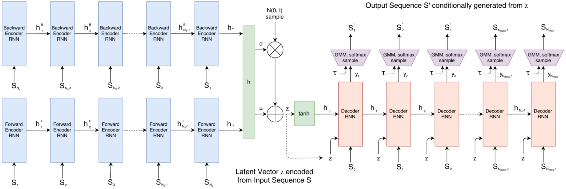
Hugely popular framework due to flexibility

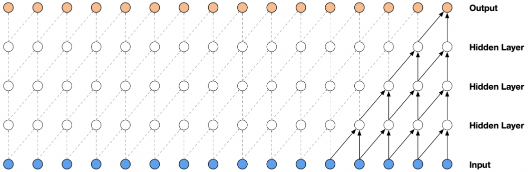
Training algorithm: **autodiff** plus **stochastic gradient descent**

- **Autodiff** is an insanely useful tool: it can algorithmically compute gradients $\nabla_\theta f_\theta(\mathbf{x})$ with the same runtime complexity as $f_\theta(\mathbf{x})$ itself!
- Deep learning libraries such as PyTorch and TensorFlow allow easy construction (and differentiation) of computation graphs
- Classic models include convolutions (for images), recurrent temporal models, etc.; new **architectures** developed constantly
- Define "feature transformation" $\Phi(\mathbf{x})$, or overall model $f_\theta(\mathbf{x})$, as arbitrary code, with parameters to be optimized

These can get big!



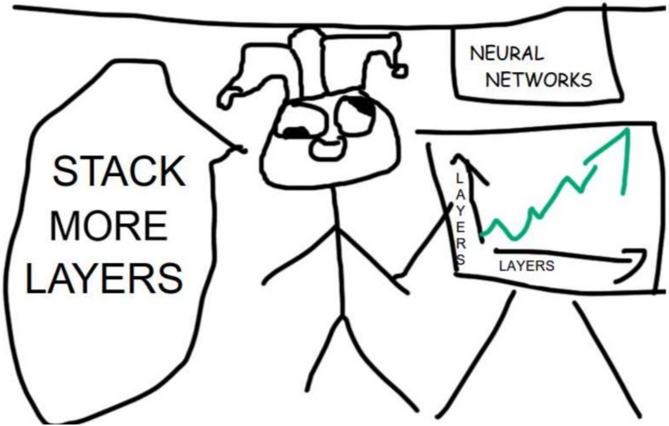




Millions of parameters is common. (One new language model has a **trillion**.)

[Figures: He et al.; Ha & Eck; van den Oord et al.]

Why make it deeper?



Universal approximation theorems

There are a number of different **universal approximation theorems**, with different guarantees:

- e.g. that a feedforward network with a single hidden layer and "squashing" nonlinearity can approximate any Borel-measurable function with arbitrarily smaller error, given sufficient hidden units.

Universal approximation theorems

There are a number of different **universal approximation theorems**, with different guarantees:

- e.g. that a feedforward network with a single hidden layer and "squashing" nonlinearity can approximate any Borel-measurable function with arbitrarily smaller error, given sufficient hidden units.

These theorems tend to be a bit underwhelming in my opinion:

- No indication of how big the network needs to be: number of hidden units required could be exponentially large
- No guarantee that any particular optimization procedure would actually find the appropriate network

Deep and narrow, or shallow and wide?

For a fixed number of parameters, should you have more "narrow" layers, or fewer "wide" layers?

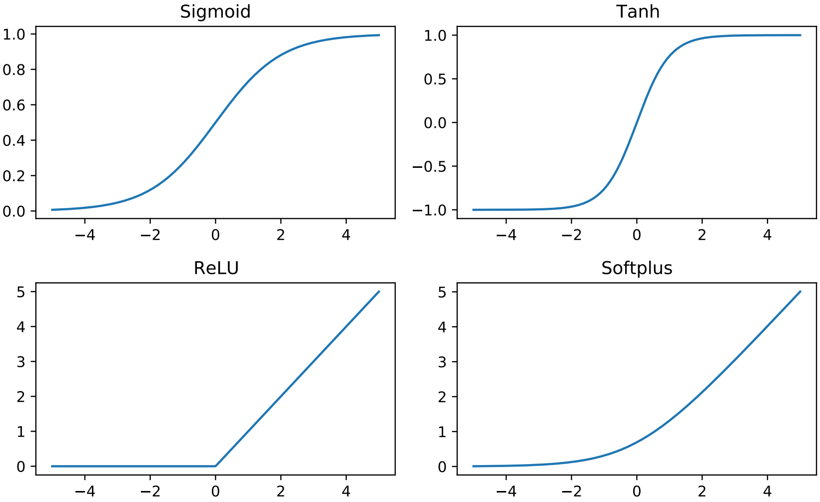
Deep and narrow, or shallow and wide?

For a fixed number of parameters, should you have more “narrow” layers, or fewer “wide” layers?

Generally the consensus is that it’s **useful for networks to be deeper**:

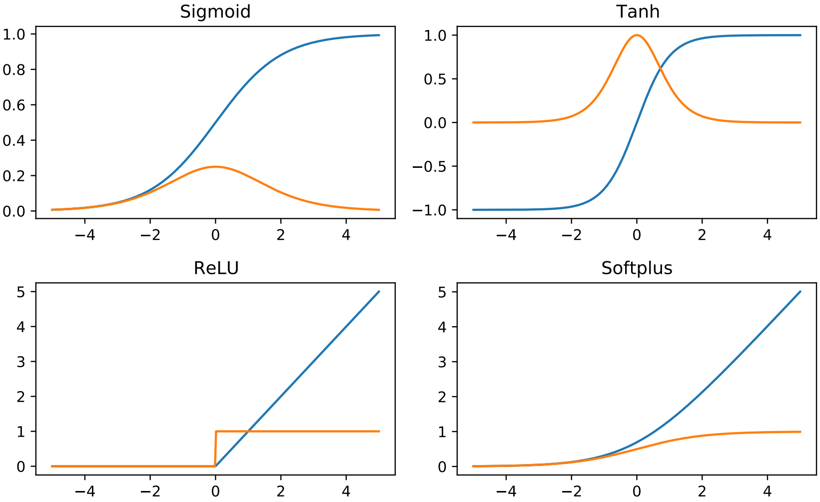
- Often much fewer-parameter “deep” networks are needed, relative to a wide single-hidden-layer network
- Composing “simple” features into more complex features makes sense conceptually
- Empirically, it seems like deeper networks generalize better to unseen data
- They also tend to be easier to optimize by gradient descent

Which activation function?



$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}\mathbf{x} + \mathbf{b})$$

Derivatives of activation functions



$$f(\mathbf{x}) = \mathbf{w}^\top g(\mathbf{A}_1 g(\dots g(\mathbf{A}_L \mathbf{x} + \mathbf{b}_L) \dots) + \mathbf{b}_1)$$

Vanishing gradients

It's a bit crazy, but the switch from using sigmoid or tanh nonlinearities to ReLU was one of the major developments than enabled easily training very deep models.

- The derivative of the sigmoid function is close to zero when the input "saturates". This shrinks the magnitude of the gradient.

- For deep networks, this causes the magnitude of the gradient to drop off roughly exponentially with depth.

Historically, this made it hard to estimate parameters in very deep models, because the gradients "vanished" towards zero.

What about a linear activation?

What if you choose $g(z) = z$?

- **Bad news** is it reduces to a linear model:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{h}_1 = \mathbf{w}^\top (\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1) = \underbrace{\mathbf{w}^\top \mathbf{A}_1}_{\mathbf{w}'} \mathbf{h}_2 + \underbrace{\mathbf{w}^\top \mathbf{b}_1}_{\mathbf{b}'}$$

What about a linear activation?

What if you choose $g(z) = z$?

- **Bad news** is it reduces to a linear model:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{h}_1 = \mathbf{w}^\top (\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1) = \underbrace{\mathbf{w}^\top \mathbf{A}_1}_{\mathbf{w}'} \mathbf{h}_2 + \underbrace{\mathbf{w}^\top \mathbf{b}_1}_{\mathbf{b}'}$$

- **Possibly interesting** though if you have a function from $\mathbb{R}^{D_1} \to \mathbb{R}^{D_2}$, where D_1, D_2 are both large:

$$f(\mathbf{x}) = \mathbf{W}^\top \mathbf{h} = \mathbf{W}^\top (\mathbf{A}\mathbf{x} + \mathbf{b}) = \mathbf{W}^\top \mathbf{A}\mathbf{x} + \mathbf{W}^\top \mathbf{b}$$

  Here $\mathbf{W} \in \mathbb{R}^{H \times D_2}$, $\mathbf{A} \in \mathbb{R}^{H \times D_1}$, so the projection matrix $\mathbf{W}^\top \mathbf{A} \in \mathbb{R}^{D_2 \times D_1}$ has $H(D_1 + D_2)$ parameters; if H is small this could be much less than $D_1 D_2$.

Loss functions for deep learning?

Most standard loss functions in deep learning have a **probabilistic interpretation**, and are exactly what we've seen before:

- Squared error $\leftrightarrow$ Gaussian log likelihood, fixed variance:

$$\begin{aligned} -\log \mathcal{N}(y|\hat{y}, \sigma^2) &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y - \hat{y})^2 \\ &= \alpha(y - \hat{y})^2 + \text{const.} \end{aligned}$$

- Binary cross entropy / log-loss $\leftrightarrow$ Bernoulli log likelihood:

$$-\log \text{Bernoulli}(y|p = \hat{y}) = y \log \hat{y} + (1 - y)\log(1 - \hat{y})$$

- Absolute error $\leftrightarrow$ Laplace log likelihood . . .

Loss functions for deep learning?

Most standard loss functions in deep learning have a **probabilistic interpretation**, and are exactly what we've seen before:

- Squared error $\leftrightarrow$ Gaussian log likelihood, fixed variance:

$$\begin{aligned} -\log \mathcal{N}(y|\hat{y}, \sigma^2) &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y-\hat{y})^2 \\ &= \alpha(y-\hat{y})^2 + \text{const.} \end{aligned}$$

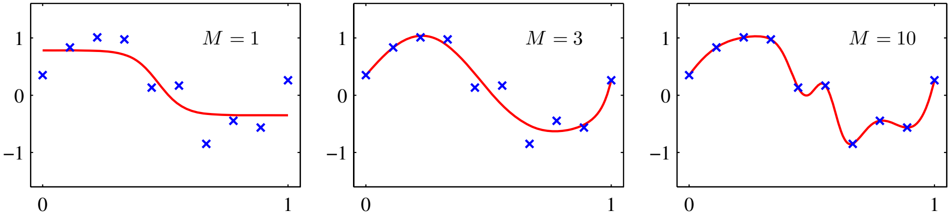
- Binary cross entropy / log-loss $\leftrightarrow$ Bernoulli log likelihood:

$$-\log \mathrm{Bernoulli}(y|p=\hat{y}) = y\log\hat{y} + (1-y)\log(1-\hat{y})$$

- Absolute error $\leftrightarrow$ Laplace log likelihood . . .

In general: define an appropriate likelihood to define a loss.

Controlling model complexity



Adjusting the number of hidden units in each layer of a two-layer network (here denoted "M") changes the complexity of the solution.

Figure: Bishop PRML

Regularization for deep learning?

The most common approach is one we've already seen: adding an L_2 or L_1 norm as a penalty on the weights of the parameters, i.e. $\|\boldsymbol{\theta}\|_2$.

Regularization for deep learning?

The most common approach is one we've already seen: adding an L_2 or L_1 norm as a penalty on the weights of the parameters, i.e. $\|\boldsymbol{\theta}\|_2$.

The second most common approach is **early stopping**:

Instead of minimizing the training error entirely, stop optimization "early", typically by monitoring a validation set.

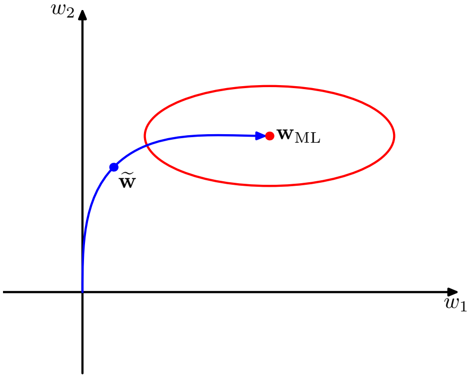


Figure: Bishop PRML

Other regularization approaches

There are some other deep-learning-specific approaches to regularization — we'll come back to this topic later.

Bayesian and non-Bayesian deep learning

$$p(y|\mathbf{x}) = p(y|\mathbf{w}^\top \phi(\mathbf{x}))$$

- Not deep, not Bayesian: fix ϕ, find a point estimate $\hat{\mathbf{w}}$
- Not deep, Bayesian: fix ϕ, estimate a posterior distribution $p(\mathbf{w}|\mathcal{D})$

Bayesian and non-Bayesian deep learning

$$p(y|\mathbf{x}) = p(y|\mathbf{w}^\top g(\mathbf{A}_1 g(\ldots \mathbf{x}) + \mathbf{b}_1))$$

- Not deep, not Bayesian: fix ϕ, find a point estimate $\hat{\mathbf{w}}$
- Not deep, Bayesian: fix ϕ, estimate a posterior distribution $p(\mathbf{w}|\mathcal{D})$
- Deep, but not Bayesian: let $\phi_\ell(\mathbf{h}) = g(\mathbf{A}_\ell \mathbf{h} + \mathbf{b}_\ell)$, find point estimates
- Deep and Bayesian: estimate posterior $p(\mathbf{w}, \{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^L|\mathcal{D})$

Bayesian and non-Bayesian deep learning

$$p(y|\mathbf{x}) = p(y|\mathbf{w}^\top g(\mathbf{A}_1 g(\ldots \mathbf{x}) + \mathbf{b}_1))$$

- Not deep, not Bayesian: fix ϕ, find a point estimate $\hat{\mathbf{w}}$
- Not deep, Bayesian: fix ϕ, estimate a posterior distribution $p(\mathbf{w}|\mathcal{D})$
- Deep, but not Bayesian: let $\phi_\ell(\mathbf{h}) = g(\mathbf{A}_\ell \mathbf{h} + \mathbf{b}_\ell)$, find point estimates
- Deep and Bayesian: estimate posterior $p(\mathbf{w}, \{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^L|\mathcal{D})$
- Deep and sorta Bayesian: find point estimates for $\{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^L$, and estimate a posterior distribution $p(\mathbf{w}|\mathcal{D})$ ("**Bayesian last layer**")

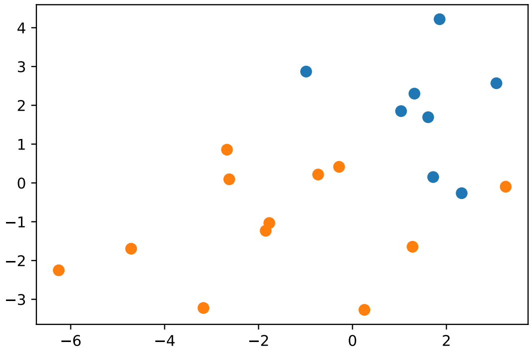
Bayesian and non-Bayesian deep learning

$$p(y|\mathbf{x}) = p(y|\mathbf{w}^\top g(\mathbf{A}_1 g(\ldots \mathbf{x}) + \mathbf{b}_1))$$

- Not deep, not Bayesian: fix ϕ, find a point estimate $\hat{\mathbf{w}}$
- Not deep, Bayesian: fix ϕ, estimate a posterior distribution $p(\mathbf{w}|\mathcal{D})$
- Deep, but not Bayesian: let $\phi_\ell(\mathbf{h}) = g(\mathbf{A}_\ell \mathbf{h} + \mathbf{b}_\ell)$, find point estimates
- Deep and Bayesian: estimate posterior $p(\mathbf{w}, \{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^L|\mathcal{D})$
- Deep and sorta Bayesian: find point estimates for $\{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^L$, and estimate a posterior distribution $p(\mathbf{w}|\mathcal{D})$ ("**Bayesian last layer**")
- Deep and sorta Bayesian: find S different point estimates for $\hat{\mathbf{w}}^s, \{\hat{\mathbf{A}}_\ell^s, \hat{\mathbf{b}}_\ell^s\}_{\ell=1}^L$ from different initializations, and treat them as a set of "samples" ("**Deep ensembles**")

Bayesian estimation in linear models

Fitting a linear classifier?

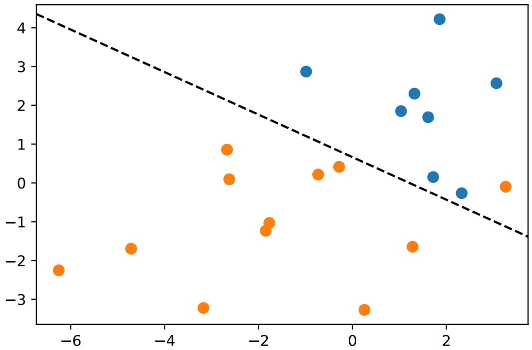


Bayesian estimation in linear models

Fitting a linear classifier?

- A "best fit" decision boundary:

$$\mathbf{w}^{\star} = \arg\max_{\mathbf{w}} p(\mathbf{w}|\mathcal{D})$$



Bayesian estimation in linear models

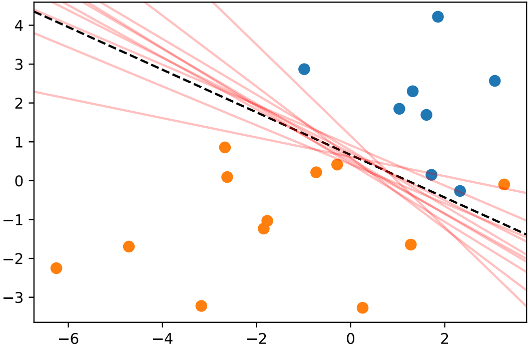
Fitting a linear classifier?

- A "best fit" decision boundary:

$$\mathbf{w}^\star = \arg\max_{\mathbf{w}} p(\mathbf{w}|\mathcal{D})$$

- Posterior samples:

$$\mathbf{w}^{(k)} \sim p(\mathbf{w}|\mathcal{D})$$



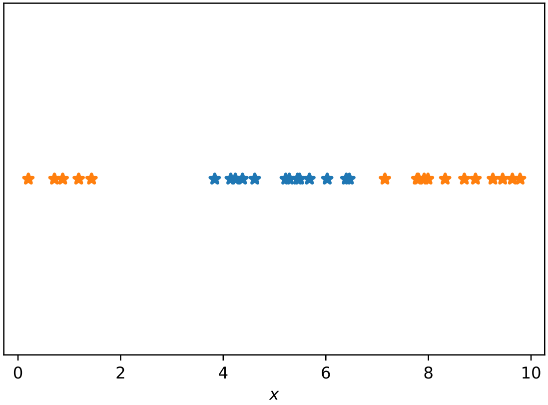
Is Bayesian “deep” learning any different?

Intuition: One-dimensional classification

Toy problem:

- one-dimensional data,
- two class labels.

Not linear separable!



Intuition: One-dimensional classification

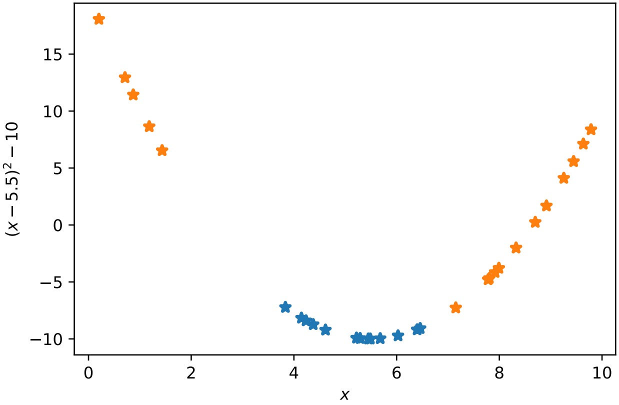
Toy problem:

- one-dimensional data,
- two class labels.

Not linear separable!

...except it is, with a (hand-selected) feature map!

$$\phi(x) = [x, (x - 5.5)^2 - 10]^\top.$$



Intuition: One-dimensional classification

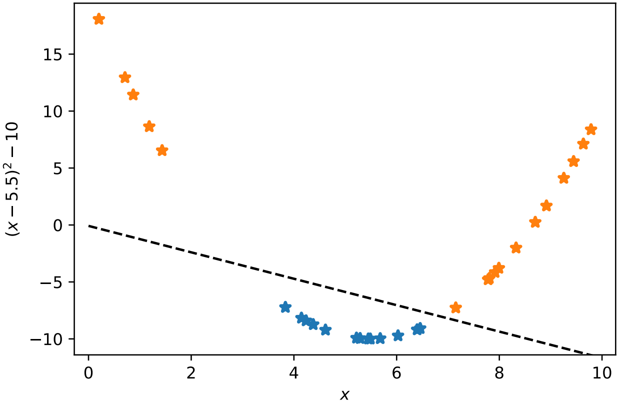
Toy problem:

- one-dimensional data,
- two class labels.

Not linear separable!

...except it is, with a (hand-selected) feature map!

$$\phi(x) = [x, (x - 5.5)^2 - 10]^\top.$$



Random one-layer networks

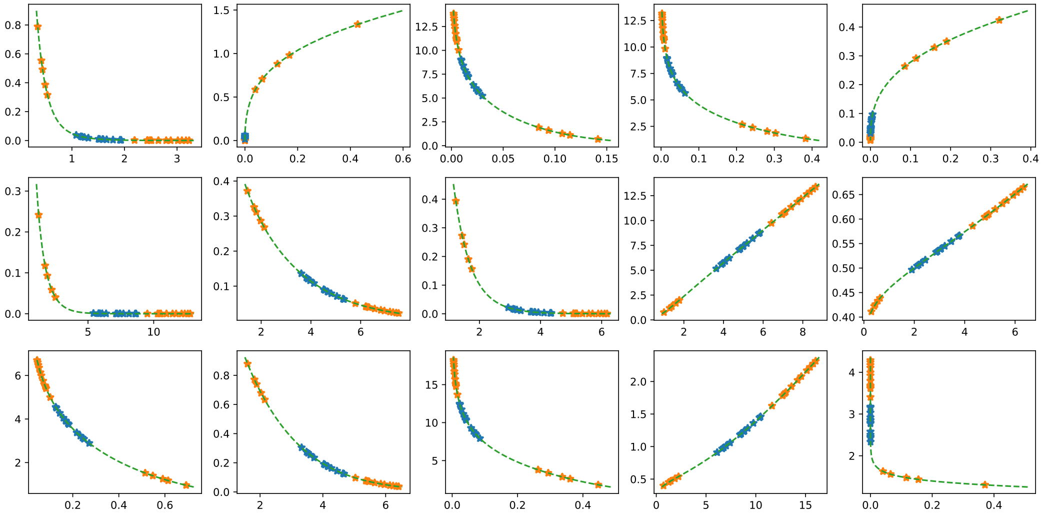
Model:

$$\mathbf{h} = g(\mathbf{A}x + \mathbf{b}) \qquad\qquad p(y = 1|x) = \sigma(\mathbf{w}^\top \mathbf{h})$$

- Parameters: $\mathbf{A} \in \mathbb{R}^{2\times 1}, \mathbf{b} \in \mathbb{R}^2$
- Softplus function $g(z) = \log(1 + \exp(z))$
- Each A_{ij} and b_i are i.i.d. $\mathcal{N}(0, 1)$

With random parameters from the prior, this defines a 2d feature space.

Random one-layer networks



Random two-layer networks

How can we get a wider variety of functions?

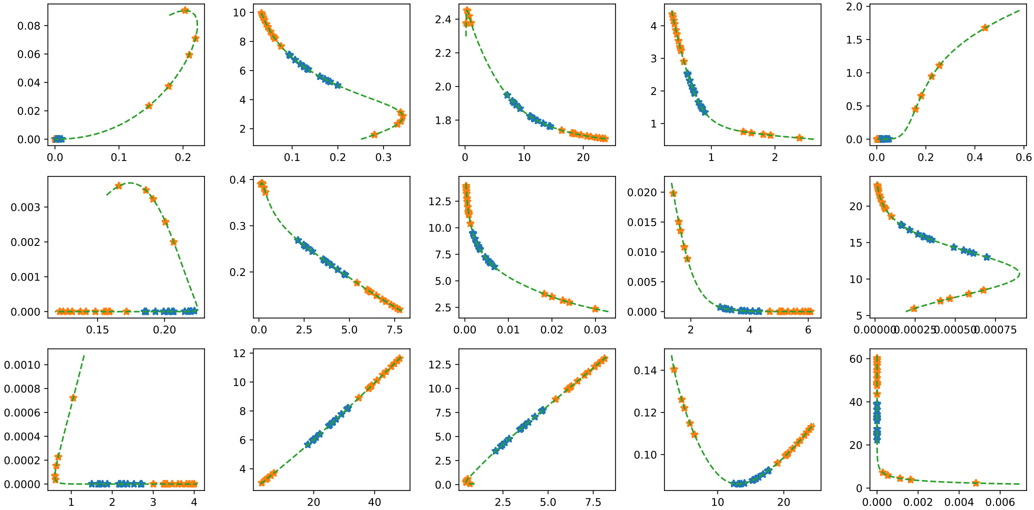
- Easy answer is to **increase depth**.

Model:

$$\begin{aligned} p(y=1|x) &= \sigma(\mathbf{w}^\top \mathbf{h}_1) \\ \mathbf{h}_1 &= g(\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1) & \mathbf{h}_1 \in \mathbb{R}^2 \\ \mathbf{h}_2 &= g(\mathbf{A}_2 x + \mathbf{b}_2) & \mathbf{h}_2 \in \mathbb{R}^8 \end{aligned}$$

- Still with each A_{ij} and b_i are i.i.d. $\mathcal{N}(0,1)$

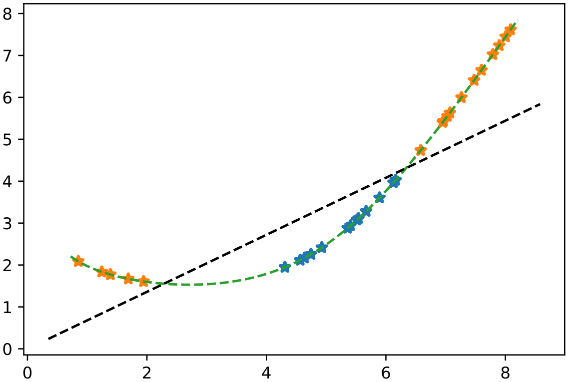
Random two-layer networks



Learning a two-layer network

Model: $p(y = 1|x) = \sigma(\mathbf{w}^\top g(\mathbf{A}_1 g(\mathbf{A}_2 x + \mathbf{b}_2) + \mathbf{b}_1))$

- It worked!
- Finding maximum likelihood (or *maximum a posteriori*, MAP) parameters learns a usable representation space

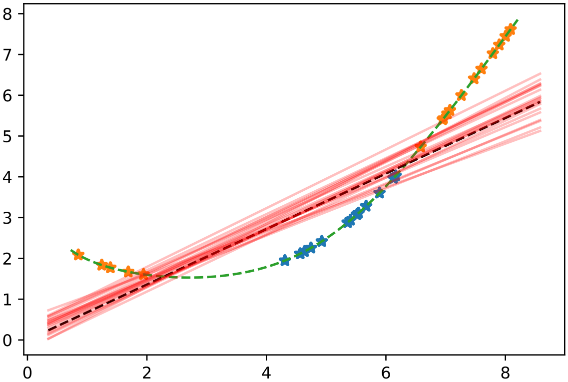


Learning a two-layer network

Model: $p(y=1|x) = \sigma(\mathbf{w}^\top g(\mathbf{A}_1 g(\mathbf{A}_2 x + \mathbf{b}_2) + \mathbf{b}_1))$

- Running Bayesian inference of the **final layer only** captures uncertainty, conditioned on the features

- Here we ran an MCMC algorithm over $\mathbf{w}$

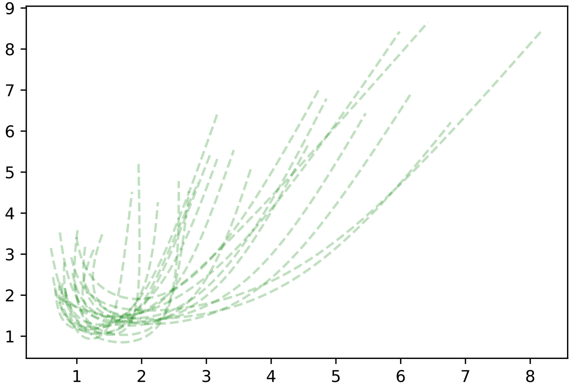


Learning a two-layer network

Model: $p(y = 1|x) = \sigma(\mathbf{w}^\top g(\mathbf{A}_1 g(\mathbf{A}_2 x + \mathbf{b}_2) + \mathbf{b}_1))$

- Running Bayesian inference on the hidden layer parameters captures uncertainty in the features themselves

- These are from running MCMC over all parameters



That wasn’t so hard.

That wasn't so hard.

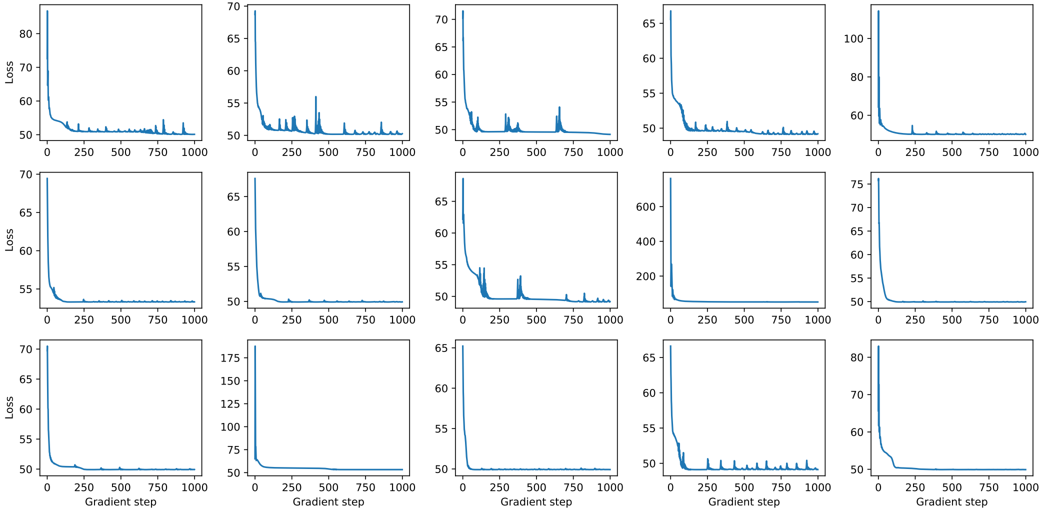
Are we done?

A multi-modal posterior distribution?

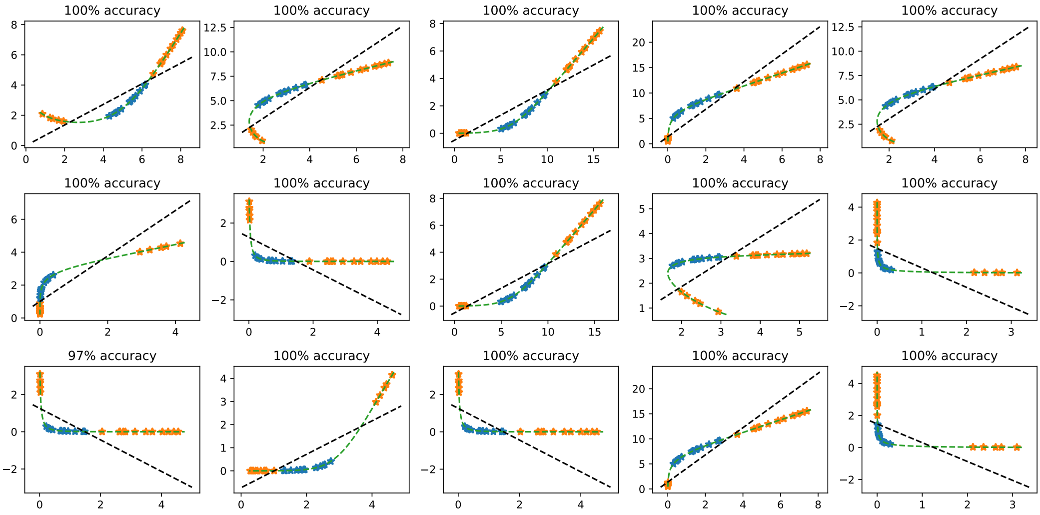
What happens if we re-train the model from scratch?

- Ideally, we might expect no non-trivial changes:
  - the model seemed to fit well
  - taking posterior samples yielded a variety of decision boundaries and representations

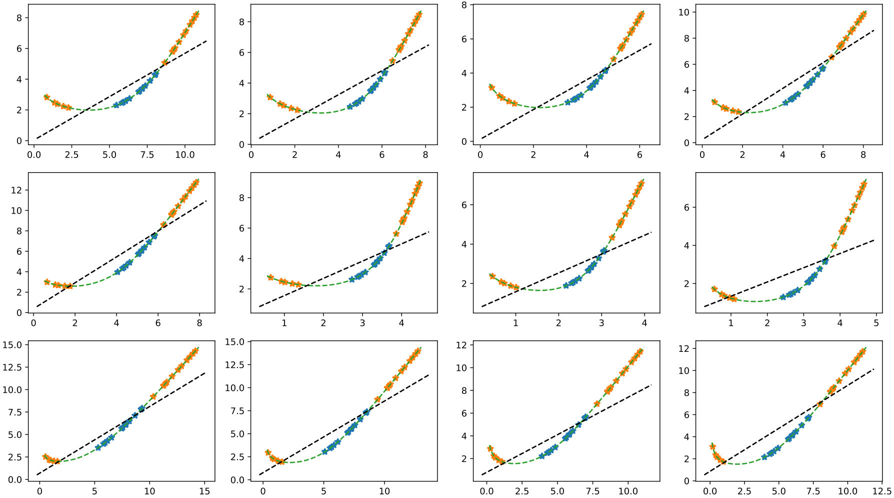
Run optimization 15 times



Learn 15 different embeddings?



Compare with variation among MCMC samples!



What’s the deal?

Local and global uncertainty

Many Bayesian inference algorithms don't deal well with disconnected modes.

- Laplace approximation
- Variational Bayes
- MCMC

Ensembles capture "global" variation...

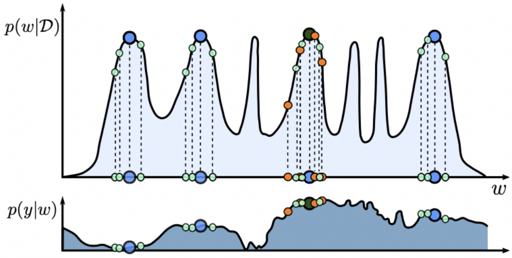
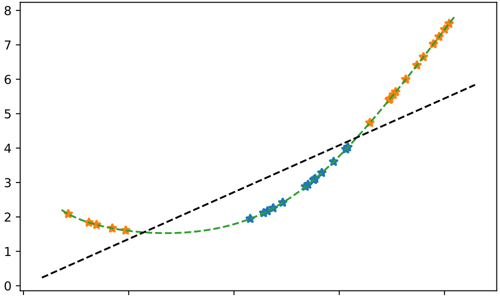


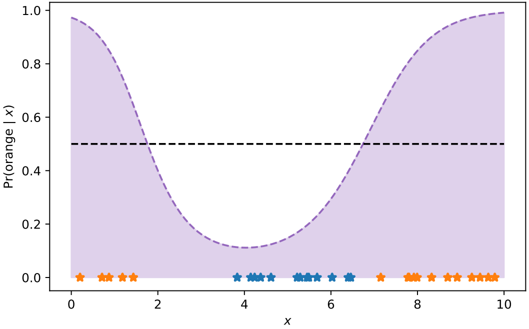
Figure: Wilson & Izmailov, 2020

Does it matter?

Comparing predictive distributions

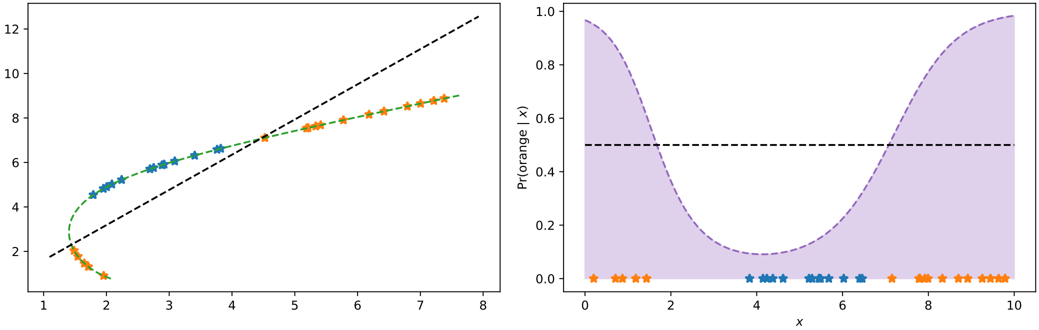
Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.





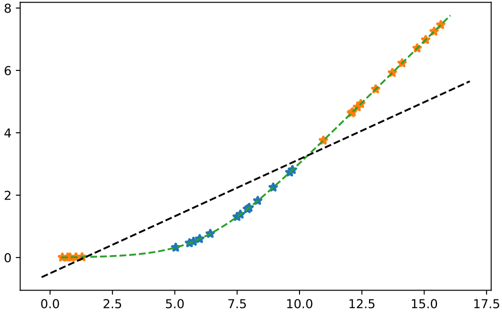
Comparing predictive distributions

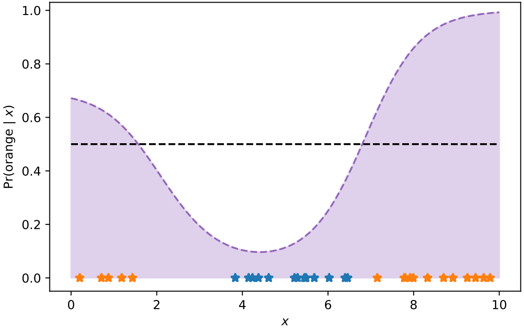
Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.



Comparing predictive distributions

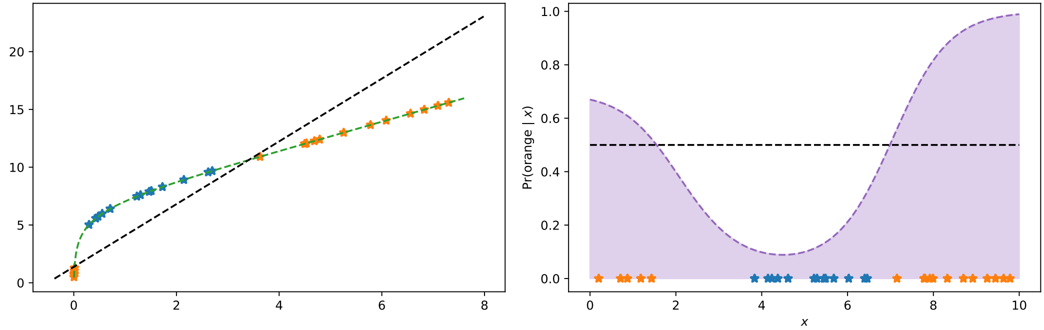
Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.





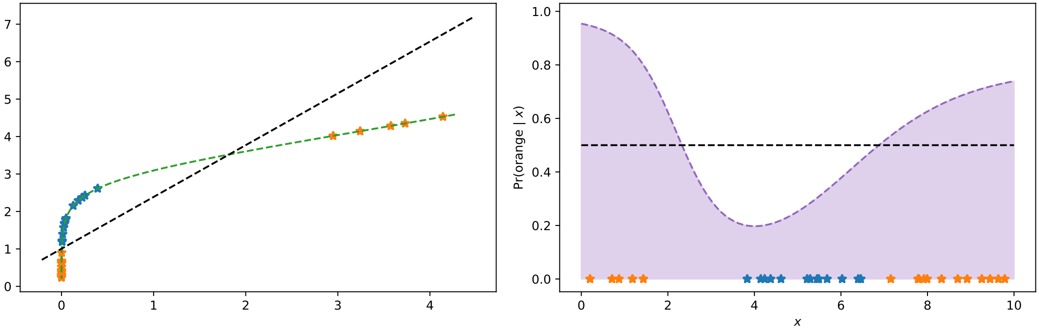
Comparing predictive distributions

Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.



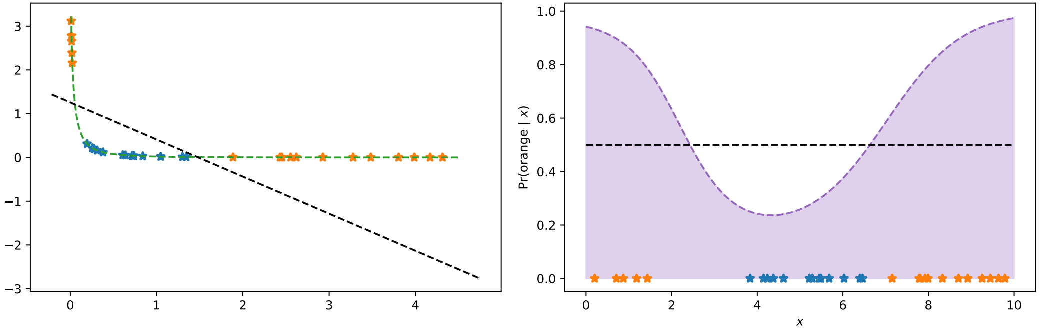
Comparing predictive distributions

Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.



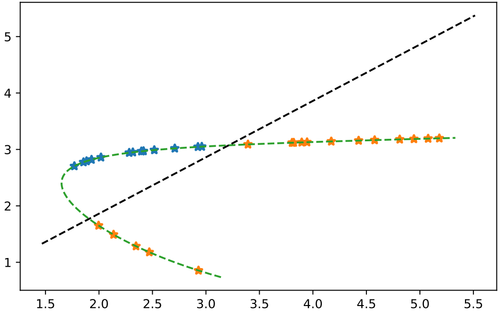
Comparing predictive distributions

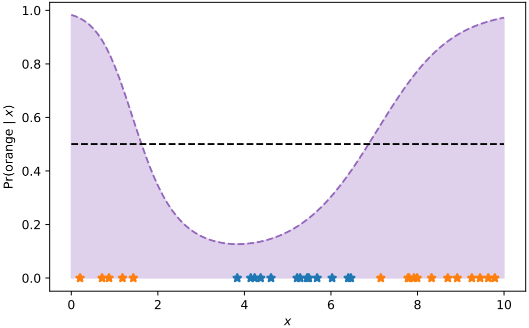
Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.



Comparing predictive distributions

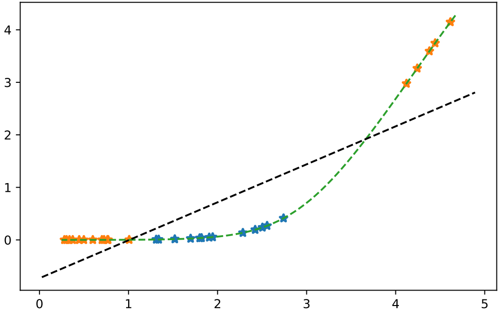
Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.

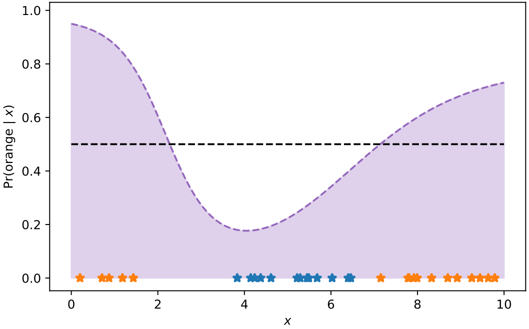




Comparing predictive distributions

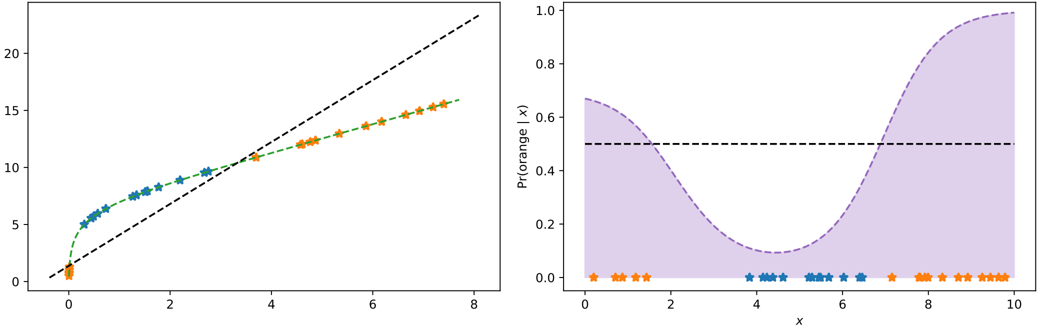
Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.





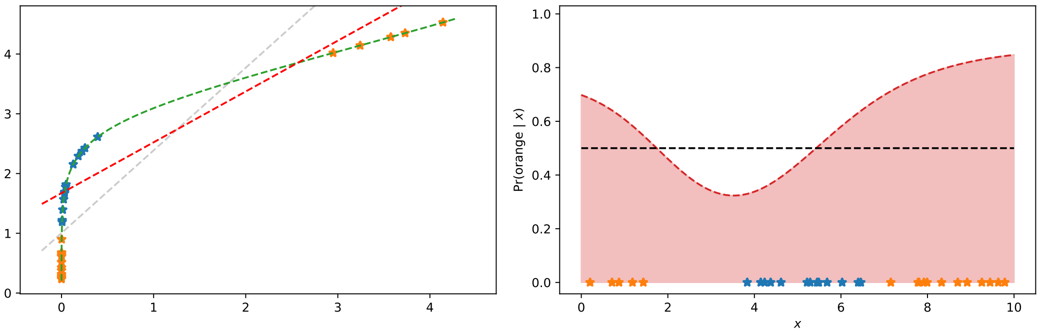
Comparing predictive distributions

Predictive distribution $p(y|x, \theta)$, across different optimized parameters θ.



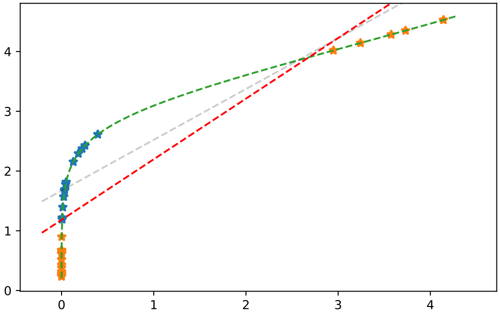
Comparing MCMC samples of final layer

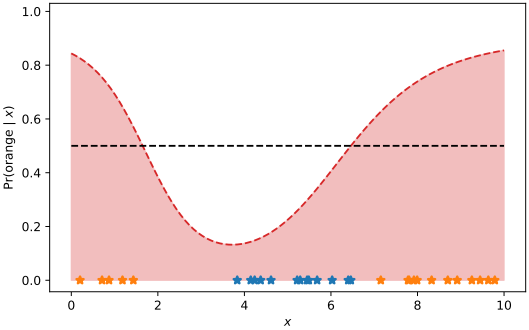
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.



Comparing MCMC samples of final layer

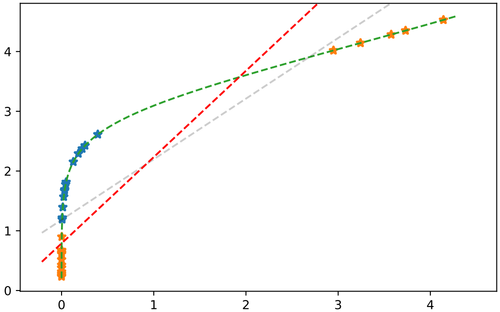
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.

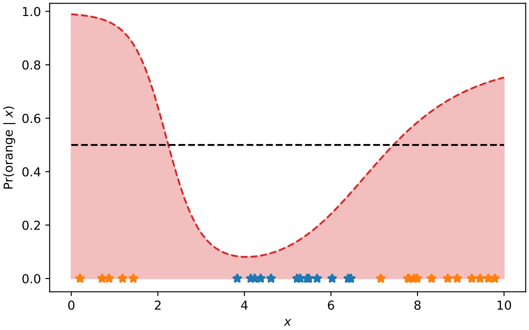




Comparing MCMC samples of final layer

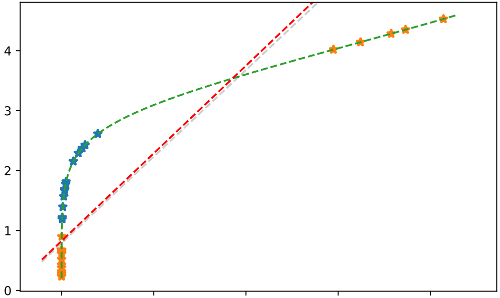
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.

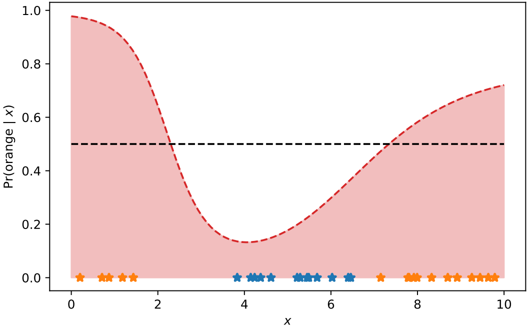




Comparing MCMC samples of final layer

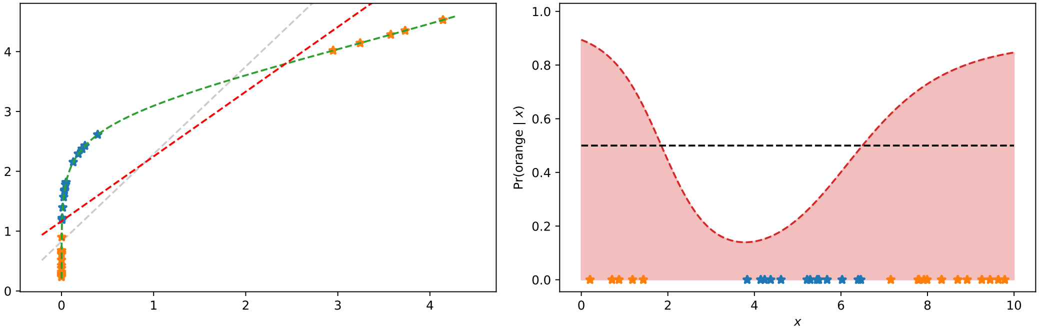
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.





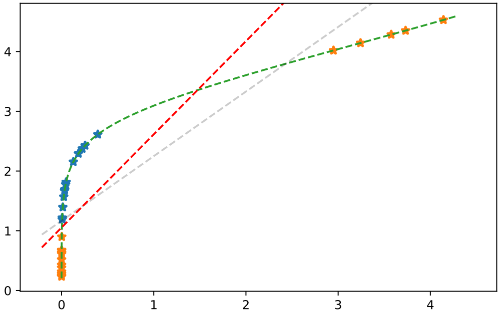
Comparing MCMC samples of final layer

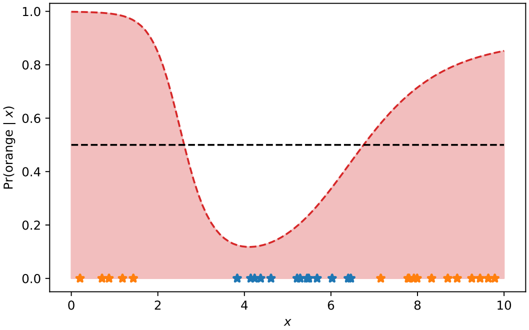
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.



Comparing MCMC samples of final layer

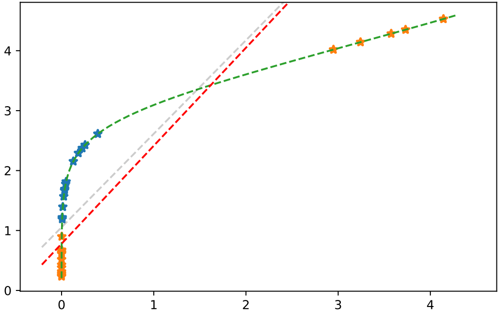
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.

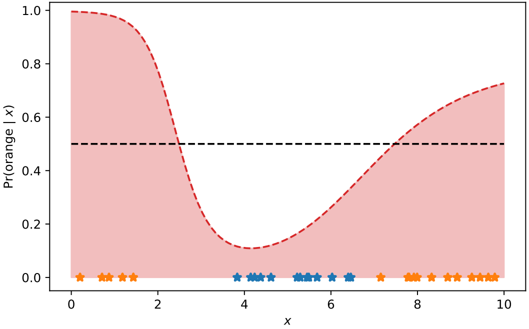




Comparing MCMC samples of final layer

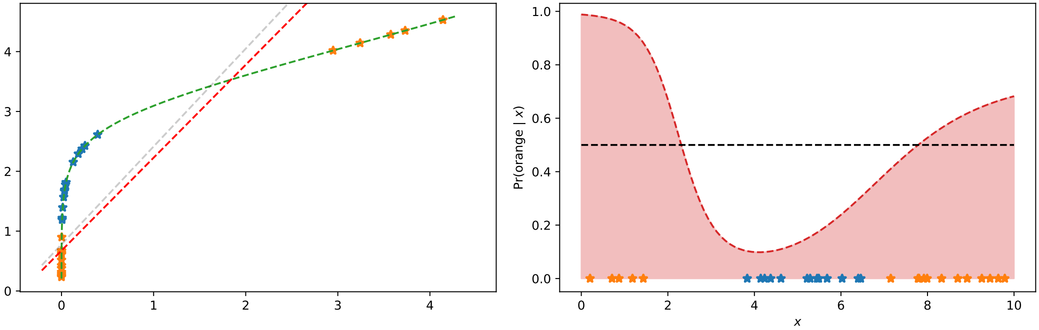
Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.





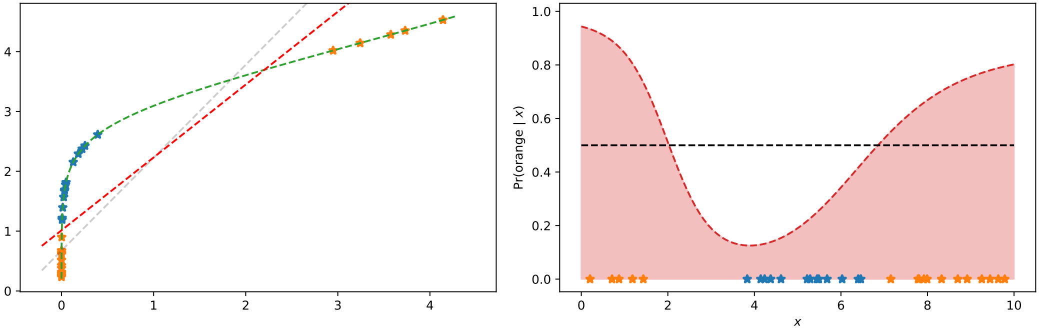
Comparing MCMC samples of final layer

Predictive distribution $p(y|x, \theta)$, MCMC inference over $\mathbf{w}$ **only**.



Comparing MCMC samples of final layer

Predictive distribution $p(y|x,\theta)$, MCMC inference over $\mathbf{w}$ **only**.



Those looked sort of similar. . .

Summary

- Deep learning learns representations
- Bayesian deep learning learns distributions over representations
- Large data and non-convex posteriors make inference hard (more on algorithms later)
- Non-obvious what sort of posterior uncertainty matters:
    - local or global?
    - entire representation? or just the predictor?